Bases de la conception orientée objet
Classes, objets, messages
TP 2

Steven Costiou
Stéphane Ducasse
Inria

May 20, 2021

1 Exercice 2 : formes géométriques

Nous allons maintenant abstraire pour modéliser d'autres formes géométriques, comme des cercles, triangles, carrés.

1.1 La classe abstraite GeometricShape

Cette classe définit de manière abstraite comment les formes géométriques se structurent et se comportent. Les sous classes de GeometricShape, elles, mettent en oeuvre cette structure et ce comportement.

1. Écrire la classe abstraite GeometricShape, quels sont ses éléments structurels et comportementaux devant être partagés par toutes les formes géométriques ?
2. Modifier la classe Rectangle pour hériter de GeometricShape. Les tests doivent continuer à passer.
3. Écrire les classes Circle, Triangle et Square :
   (a) Pour chaque classe, hériter de la classe qui vous semble la plus judicieuse et proposer un constructeur,
   (b) écrire les méthodes devant être redéfinies (en TDD).

1.2 Dessin des formes géométriques

Nous allons utiliser une classe de test PainterApp, qui utilise un objet Painter dédié à l'affichage de formes graphiques. La méthode paint(Graphics g) de Painter est appelée automatiquement lors de l'affichage. Cette méthode doit utiliser l'objet de dessin Graphics g pour afficher des objets graphiques.

Récupérer les classes PainterApp[1] et Painter[2]. Dans la classe PainterApp, nous stockons des formes géométriques dans un objet ArrayList<GeometricShape> shapes. Les objets de type ArrayList<T> permettent de stocker tout objet de type T dans une liste ordonnée, et proposent une interface simple pour ajouter, accéder, énumérer et supprimer des objets de type T.

Dans cette liste nommée shapes, nous souhaitons stocker des formes géométriques sans se préoccuper de leur nature concrète (rectangle, cercle, triangle...). Comme les classes Rectangle, Circle, Triangle et Square héritent toutes de GeometricShape, elles répondent toutes à l'interface de GeometricShape et peuvent donc être utilisées comme des formes géométriques abstraites (polymorphisme).

1. Instanciez diverses formes géométriques (rectangles, cercles, etc.) et ajoutez les à la liste de formes géométriques en utilisant la méthode add(GeometricShape s.

Puis, compléter la classe Painter avec les éléments suivants :

2. Écrire une méthode public void paintRectangle(Rectangle r, Graphics g)) qui dessine le rectangle r sur le médium graphique g. Vous utiliserez la méthode de dessin offerte par g pour dessiner des rectangles :
   drawRect (int x1, int y1, int x2, int y2).

3. Écrire une méthode public void paintCircle(Circle c, Graphics g) qui dessine le cercle c sur le médium graphique g. Vous utiliserez la méthode de dessin offerte par g pour dessiner des cercles :
   drawOval(int centerX, int centerY, int radius, int radius).

4. Écrire une méthode public void paintTriange(Triangle t, Graphics g) qui dessine le triangle t sur le médium graphique g. Vous utiliserez la méthode de dessin offerte par g pour dessiner des lignes :
   drawLine(int x1, int y1, int x2, int y2).

5. Dans la méthode paint(), parcourir la liste des formes géométriques contenues dans le tableau shapes, et appeler la méthode de dessin appropriée en fonction de la classe de chaque objet. Pour cela vous pouvez utiliser le code suivant qui renvoie true si la forme géométrique s est un rectangle et false sinon : s instanceof Rectangle.

L'utilisation de instanceof n'est pas *objet*. En programmation objet, on va laisser les objets décider comment ils répondent à un message plutôt que de contrôler et sélectionner impérativement les messages qu'on leur envoie.

Nous allons introduire une méthode de dessin dans chaque objet géométrique afin de les laisser décider comment ils doivent être affichés.

[1] https://kloum.io/costiou/teaching/PainterApp.java
[2] https://kloum.io/costiou/teaching/Painter.java

6. Écrire dans la classe abstraite GeometricShape une méthode abstraite public abstract void drawOn(Graphics g).

7. Dans chaque sous classe de GeometricShape, implémentez la méthode drawOn(Graphics g) en conséquence.

8. Dans la classe Painter, supprimer toutes les instructions conditionnelles et les tests instanceOf et remplacer cela par un envoi du message drawOn(g) aux objets géométriques. Supprimer le code devenu inutile.

1.3 Une meilleure conception objet

Dans l'exercice précédent, nous avons introduit une dépendance entre notre modèle de formes géométriques et la librairie d'affichage Java. Cela veut dire qu'on ne peut pas déployer notre code sans, soit inclure la librairie d'affichage Java, soit poser la contrainte qu'il n'est pas utilisable hors du système de base Java. Cela nous posera des contraintes pour déployer notre code, et permettre à d'autres utilisateurs de l'utiliser de manière indépendante. En effet, on peut vouloir réutiliser notre modèle de formes géométrique avec un autre médium d'affichage (par exemple *OpenGL*). Nous avons donc besoin que notre code soit *modulaire*. Pour cela, nous allons séparer notre modèle (les formes géométriques) et leur affichage en introduisant des adapteurs dédiés à la librairie d'affichage Java que nous utilisons :

1. Écrire une classe abstraite GeometricShapeDrawer avec une méthode abstraite public abstract void drawOn(Graphics g). La classe GeometricShapeDrawer possède une variable d'instance GeometricShape avec son accesseur.

2. Pour chaque sous classe de GeometricShape, écrire une classe dédiée à son affichage, sous classe de GeometricShapeDrawer, et implémentez la méthode drawOn(Graphics g) en conséquence.

3. Dans la classe PainterApp, encapsuler les instances de formes géométriques dans des objets GeometricShapeDrawer. Attention :

   (a) Choisissez la sous-classe de *drawer* adéquate en fonction de la forme géométrique que vous encapsulez,

   (b) vous pouvez avoir besoin de modifier le type de la collection des formes géométriques.