CD: M B

JC : 3

1h

1_ Réutilisation

(1) V

1. Il s'agit du paramétrage par spécialisation.

La méthode drawDescription() de Figure est paramétrée par la méthode toString(). Dans ce type de paramétrage, le paramètre est l'identificateur this (ou self selon les cas). Cette méthode de toString() est variable par spécialisation

(0,25) 1/4

2. Une fonction d'ordre supérieur

drawDescription() est d'ordre supérieur car ~~sto~~ c'est une méthode ~~spécial~~ paramétrée par spécialisation qu'on retrouve dans les sous-classes de Figure sous une autre variante

(1) V

4. L'affectation polymorphique est l'affectation d'une ~~variable~~ valeur de type ST à une variable de type T où ST est un sous-type de T. Dans le listing 2 nous avons : Se contenu = new HashSet<Figure>, où contenu est un attribut de type Set<Figures>. La classe HashSet héritant de Set, il y a bien une affectation polymorphique.

PARTIE A

Rsb dans la réutilisation

Il serait possible de définir une spécialisation de la classe Dessin qui utiliserait une spécialisation de la collection Set, autre que HashSet. Ceci permettrait d'adapter l'attribut contenu au nouveau contexte d'utilisation

3) class Cercle extends Figure {
private Point centre; private int rayon;
public Cercle (Point c, int r){
this.centre = c; this.rayon = r;
}
public String toString() {
String description = "-un cercle de centre " +
centre.toString() +
" et rayon " + r.toString() + "\n";
return description;
}
}

5) - Création d'une classe de test pour tester la méthode drawDescription() des types de Figure

- Création d'une classe de test pour tester la classe Dessin
  - test de la méthode add(Figure) :
    vérifier qu'une Figure est bel et bien ajoutée à la collection contenu (test de la taille de la collection, du 1er élément de la collection (vérifier qu'il s'agit bien de la Figure que nous avons ajouté), etc.

- test de drawDescription()

drawDescription() ayant déjà été testée pour tous les types de Figure et add(Figure) testée aussi, on peut supposer que drawDescription() de Dessin aura le bon comportement. Toutefois, on peut rajouter un test de cette méthode pour s'en assurer.

- test de drawGraphic()

Cette méthode sera testée en ~~effectuant~~ essayant de dessiner plusieurs figures (à l'affichage).

2 - Réutilisation et typage statique

1 - public boolean Equals (Object o){
    Cercle c = (Cercle) o;
    return this.centre.equals(c.getCentre()) &&
        this.rayon == c.getRayon();
}

2 - Les règles de substitutabilité de Liskov ~~stipulent~~ exigent une contra-variance ~~sur~~ des types des paramètres de méthodes et une covariance des type de retour. Cependant ces règles de substitutabilité s'opposent à la sémantique de la spécialisation qui impose une spécialisation des types des paramètres.
Par exemple dans notre contexte, nous ne voulons comparer un cercle qu'avec un autre cercle, et aucun autre objet; d'où la nécessité d'utiliser un transtypage descendant afin de s'assurer que nous ne recevons que des objets de type Cercle comme paramètre de equals. Dans le cas contraire, une TypecastException sera levée.

l'adapteur lui-même.

5) 1. Utilisation ^plus variante^ du design pattern Composite

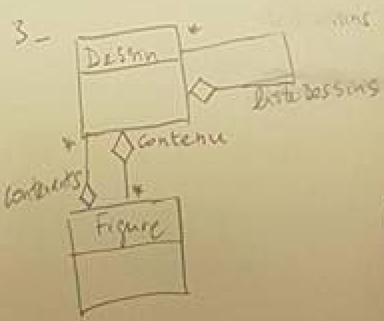
0,5 V 1/3

2 - class Dessin {
- - -
protected List<Dessins> listeDessins = new ArrayList<Dessins>();
- - -

public boolean equals (Object o) {
Dessin d = (Dessin) o ;
return this.contenu.equals (d.contenu) &&
this.listeDessins.equals (d.listeDessins) &&
this.title.equals(d.title);
}
}

1 V

3 -

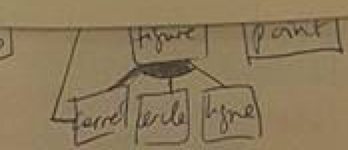


parent!

manquait le lien typé

class Figure {
protected List<Dessins> contenants = new ArrayList<Dessins>();
// liste des dessins contenants la figure

0,25



3)
Figure f1 = new Cercle (new Point(3,3), 1);
Ordre des méthodes et leur classe de définition

drawDescription() : Figure
↓
toString() : Cercle
↙ ↘
~~toString() : Point~~ toString() : Integer
toString() : Point

3. Lignes de produit

1. Avantages

- Les lignes de produits permettent de définir les spécifications d'un système : contraintes, dépendances, caractéristiques.
- Ils permettent de connaître les différentes configurations valides d'un système
- Ils favorisent une meilleure réutilisation et une extension d'un système.

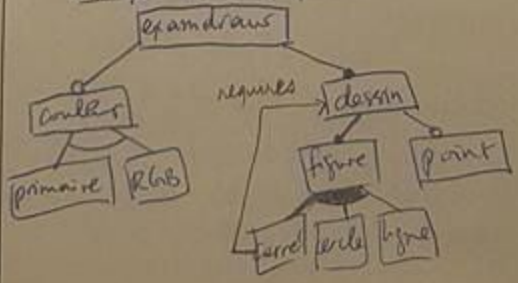
2. Configuration valide

{exemDraw, dessin, figure, carre, cercle, ligne}

Configuration non valide

{exemDraw, couleur, primaire, RGB}

Exemple de modification



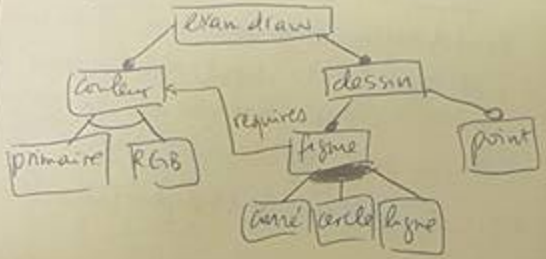
Ne pas écrire dans cette marge

NOTE

Cadre réservé au correcteur

3-

3- couleur devient obligatoire, passer couleur en "mandatory". Cela est dû au fait que figure soit obligatoire.

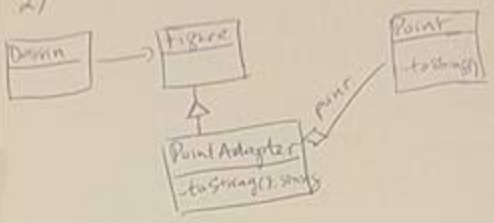


4- Schéma Adapter

1) La méthode add() attend en paramètre une ~~instance~~ objet de type Figure. Point n'hérite pas de cette classe Figure

PARTIE A
RABATTRE

2)




```
class PointAdapter extends Figure {
  private Point point;
  public PointAdapter(Point p){
    this.point = p;
  }
  public String toString(){
  String descr="Un point de coordonnées" +
                point.toString() + "\n";
    return descr;
  }
```

Explication

PointAdapter hérite de Figure et se compose d'un attribut de type Point. Nous redéfinissons la méthode toString() sur cette classe PointAdapter qui est spécialisée par celle de Point. Ainsi pour ajouter un point on utilisera l'adapteur.

ex: dr.add(new PointAdapter(new Point(x,y)));

3 - Exemple du problème du receveur initial

Point pourrait être une sous-classe d'une classe C et spécialiser la méthode toString() de cette classe C en effectuant un traitement complexe qui appelle une autre méthode m() (spécialisation de toString() par cette méthode m()).
La mise en œuvre d'un héritage multiple par composition nous fait donc perdre le receveur initial qui devrait être