Réutilisation

1. Le type de paramétrage utilisé ici est un paramétrage par spécialisation.
La méthode drawDescription() est paramétrée par la méthode toString() variable via spécialisation. Le paramètre est l'identificateur this.

2. Une fonction d'ordre supérieur (fonctionnelle) est une fonction qui accepte une fonction en argument et/ou rend une fonction en valeur. Dans notre cas, drawDescription() de Figure peut être considérée comme telle car elle rend comme valeur le contenu de la fonction toString().

3.
```
Class Cercle extends Figure {
        protected point centre;
        protected int rayon;
        public Cercle(point c, int r){
          centre = c;
          rayon = r ;
        }
        public void toString{
          system.out.println("- un cercle  de centre :"+ centre.toString()+"et rayon"+ rayon)
        }
}
```

4. Une affectation polymorphique est l'affectation d'une valeur de type ST, sous type de T, à une variable de type T.
contenu = new HashSet <Figure>
contenu étant un attribut de type Set<Figure> et comme HashSet hérite de Set donc il y'a une affectation polymorphique.
- Rôle dans la réutilisabilité d'examdraw :
L'attribut contenu pourrait alors être au nouveau contexte d'utilisation en définissant une spécilisation de la classe Dessin qui utiliserait une spécialisation de la collection Set, différente de HashSet.

5. Création d'une classe de test pour tester la méthode drawDescription() des sous types de Figure :
- Création d'une instance de Dessin
- Création de 4 instances de chaque sous classe de figure
- Test de la méthode drawDescription() de Figure sur les instances.
- Leur ajout successif dans l'ensemble HashSet de l'ensemble des Dessin : S'assurer que l'insertion s'est bien faite(test de la taille de la collection, du 1er élément de la collection).
- L'appel de la méthode drawDescription() par l'instance de Dessin pour tester si les descriptions correspondent bien aux Figures de la collection.

- L'appel de la méthode drawGraphic() pour s'assurer qu'on obtient bien les figures voulues 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 substituabilité de Liskov exigent une contra variance des types des paramètres de méthodes et une covariance des types de retour. ***si S est un sous-type de T, alors tout objet de type T peut être remplacé par un objet de type S sans altérer les propriétés désirables du programme concerné.***

Ici, nous utilisons le transtypage déscendant pour avoir que des objets de type cercle comme paramètre du equals redéfinit précedemment dans le but d'imposer de ne pouvoir comparer un cercle qu'avec un autre cercle. Dans le cas contraire, une exception de type TypeCastException est levée.

3. Figure f1 = new Cercle(newPoint(3, 3), 1);

On a dans en premier lieu l'execution de la méthode drawDescription() de Figure suivi de la méthode toSting() de la classe Cercle, elle-même suivi du toSting() de Point pour terminer par le toString() de la classe des entiers(Integer).

3. Lignes de produit

1. Avantages des lignes de produit :

- Permet le développement d'une multitude de produits ou systèmes logiciels avec un gain considérable en termes de coût, de temps et de qualité.
- Ils permettent de connaître les différentes configuration valide d'un système afin de pouvoir respecter les contraintes de la Software line.
- Ils fournissent une réutilisation afin de faire un processus de création des systèmes logiciels à partir des logiciels existants au lieu de les créer à partir de zéro ainsi que des systèmes de configuration en masse des logiciels(production à grande échelle de biens adaptés aux besoins individuels des clients)

2. Configuration valide : (examdraw, dessin, figure, carré)

Configuration non valide : (examdraw, couleur, primaire, RGB)

Exemple de modification sans changer son ensemble de configuration valide :

Tout le schéma + la contraine : ligne « requires » dessin

3. Lorsqu'on ajoute une contrainte cross-tree "figure requires couleur", la caractéristique couleur sera toujours présente dans les configurations valides comme Figure est obligatoire.
Dans cas, on pourra remplacer le signe optionnel sur couleur et le mettre en obligatoire(Mandatory).

4. Schéma Adapter

1. Parceque Point n'hérite pas de la classe Figure alors que le paramètre de add() est un objet de type Figure.

2.

```
class PAdapter extends Figure {

private Point p;

public PAdapter(point p) {
        this.p=p;
}

public string toString() {
   System.out.println("un point de coordonnées" + p.toString());
}
}
```

PointAdapter hérite de Figure et contient un attribut de type point. La méthode toString() de cette classe rédifinit celle de point.

Ainsi, l'ajout d'un point se fera alors par : d1.add(new PAdapter(new Point(x, y)));

A noter que Dessin se compose d'un ou de plusieurs figures dont Point grâce à l'adapteur fait parti à présent.

3. Exemple du problème de perte du receveur initial

Point pourrait être une sous classe d'une classe A et spécialiser la méthode toString() de cette classe A en effectuant un traitement complexe qui appelle une autre méthode m() (spécialisation de toString() par cette méthode m()).
Le receveur initial qui devrait être l'adapteur lui-même est donc perdu par la mise en oeuvre d'un

héritage multiple par composition.

5. Des dessins arborescents

1. Relation reflexive d'agrégation sur dessin d'arité *(0 ou plusieurs) afin d'avoir une liste de dessins(design pattern composite).

2. class Dessin {
   protected List<Dessin> ListeDessins = new ArrayList<Dessin>() ;

   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);
   }
}

3.

En plus de l'architecture défini dans la 1ère question, on rajoute deux relations d'agrégations entre Dessin et Figure. L'une où l'agrégation se situant sur la classe Dessin et dont l'arité est *(0 ou plusieurs, on pourrait nommer cette relation contenu et l'autre inversemment(conteneurs). A celà s'ajoute les classes Carre, Cercle et ligne représentant des sous classes de Figure.

6. Schéma Command

1. Pour traiter ce cahier des charges, il nous faudra d'abord une classe "invoker" ayant les méthodes add() et undo().

Il faut en plus une interface "Commande" spécifiant les méthodes communes aux commandes concrètes. Deux autres classes "CommandeAdd" et "CommandeUndo" implementeront cette interface. Chaque classe classe concrète implémente la méthode executer() en appelant des méthodes de l'objet Recepteur.
CommandeAdd stockera dans une variable de type string l'ordre d'ajout des figures.
Enfin, un Recepteur(une classe) recevra les commandes et réalisera les opérations associées.
Chaque objet Commande concret possède un lien avec l'objet Recepteur.

La partie cliente de ce schéma configurera le lien entre les objets Commande et le Récepteur.