

## A.

### A.1.

#### A.1.a.

1. // On initialise f sur la classe abstraite Forme
2. // On assigne un Cercle a f (polymorphisme).
3. // On assigne une Ligne a f (substituabilité)

#### A.1.b.

“Fonction d’ordre supérieur (fonctionnelle) : fonction qui accepte une fonction en argument et/ou rend une fonction en valeur (fonction qui est fonction d’une fonction). Analogie conceptuelle avec les équations différentielles.”

1. drawDescription peut être considéré comme une fonction d’ordre supérieur car celle-ci rend la fonction getDescription().

#### A.1.c.

1. Paramétrage par sous-typage OU paramétrage par spécialisation

source :

#### 2. ← Configurer et Composer

- (a) Configurer<sup>[4]</sup> les éléments paramétrés en (les instantiant) et valuant<sup>[5]</sup> les paramètres.

```
1 (carré 5)
2 (carré 7)
```

*Listing (3): valuation d’un paramètre lors d’une application (Scheme)*

```
1 new A(new B1()) //avec B1 et B2 sous-types de B
3 new A(new B2())
```

*Listing (4): valuation d’un paramètre utilisant du sous-typage, lors d’une instantiation, syntaxe (Java).*

2. Il n’y a que le paramètre implicite ‘this’ (car il n’y a rien dans les parenthèses)
3. L’instance de Cercle puis l’instance de Ligne sont les arguments. Car en programmation objet le receveur est un argument distingué (il prend la place de this lors de l’exécution).

source :

**Paramètre Implicite** : **this** (ou **self**) est un paramètre implicite (n’ayant pas besoin d’être déclaré explicitement) de toute méthode. Ceci est vrai dans tous les langages à objets à classes. Ce paramètre est lié au receveur courant à chaque invocation de la méthode.

## A.2.

```
public class Ligne extends Forme {

    private Point p1;
```

```

private Point p2;

public Ligne(Point p1, Point p2) {
    this.p1 = p1;
    this.p2 = p2;
}

@Override
public String getDescription() {
    return "— une ligne de " + this.p1.toString() + " à " + this.p2.toString();
}
}

```

### A.3.

1. Tests Boite Noire : On commence par identifier les entrées (préconditions) et les sorties attendues (postconditions) sans regarder le code (test par une autre équipe). Le but est de savoir si le logiciel/composant répond aux spécifications.
2. Définition des Critères : On liste les critères de variation (ex: tableau trié ou non, taille paire/impair, limites).
3. Construction des Classes d'Équivalence : On regroupe les cas de tests qui "testent la même chose" pour éviter de tester 1000 fois la même logique inutilement. Dans ce cas là sur la classe Dessin.
4. Ajout des Tests Structurels (Boîte Blanche) : Tester les éléments à même le code, comme un `if(x>0)` tester avec `x>0` et `x<0`.
5. Automatisation : On implémente ces tests avec JUnit pour pouvoir les rejouer automatiquement à chaque modification (tests de régression).

Source : <https://www.lirmm.fr/~dony/notesCours/cTest.pdf>

## B.

### B.1. &

### B.2.

```

// Pas de paramètres additionnels
// Redéfinition Contra-Variante : Accepte Objet qui est supérieur ou égale à l'objet de
// la méthode d'origine
// Redéfinition co-variante : Renvoie un Boolean qui est supérieur ou égale au
// Boolean que renvoie la méthode d'origine.
@Override
public boolean equals(Object o){

    if (!(o instanceof Ligne)) {
        return false;
    }

    Ligne li = (Ligne)o;

    if(li.p1.equals(this.p1) && li.p2.equals(this.p2)){
        return true;
    }
    return false;
}
}

```

L'analyse statique impose la règle (dite règle de substitution de [Liskov](#)<sup>[10]</sup>) qui stipule pour une méthode prétendant au status de spécialisation :

1. pas de paramètres (donc de requis) additionnels,
2. redéfinition contra-variante (inverse à l'ordre de sous-typage) des types des paramètres,
3. redéfinition co-variante (respectant l'ordre de sous-typage) des types de retour, y compris les cas d'exceptions.

### B.3.

1. f1.drawDescription() -> classe Forme
2. getDescription() -> classe Ligne
3. toString() -> classe Point
4. toString() -> classe Point

### B.4.

Non, cela ne compilera pas car `Vector<Cercle>` n'est pas un sous-type de `Vector<Forme>`, si c'était possible on pourrait ajoute une Ligne a v (car v est une Vecteur<Forme>) dans un Vecteur<Cercle> ce qui est impossible.

La solution est soit d'utiliser un wildcard (`Vecteur<? extends Forme>`) mais cela empecherais l'ajout de nouveau éléments, soit de crée un `Vecteur<Forme> = new Vector<Forme>()` mais qui permetrais de mettre tout types de formes.

### B.5.

Factoriser `equals()` semble impossible car la façon d'identifier une forme est spécifique a la forme, une Ligne compare ces deux points, un Cercle sont point centrale et rayon, si on vient a factoriser `equals()` dans Forme il faudrait vérifier de quelle instance appartient la Forme ce qui rendrais la factorisation inutile.

Une solution qui semble possible serait d'ajouter une iste de Points et une liste de mesures a Forme, afin de trouver si une forme est la meme a condition que les 2 objets soit de la meme instance et qu'ils ai les meme points et mesures (rayons, longueur, ect...).

```
@Override
public boolean equals(Object o){

    Class selfClass = this.getClass();

    if (!(this.getClass() == o.getClass())) {
        return false;
    }

    Forme fo = (Forme)o;

    return this.points.equals(fo.points) && this.mesures.equals(fo.mesures);
}
```

Une autre solution qui ne demande pas d'ajouter des attribut est de comparer la description, si deux formes de la meme classe on la meme description alors ce sont les meme formes.

```
@Override
public boolean equals(Object o) {

    if (this.getClass() != o.getClass()) return false;
```

```
Forme fo = (Forme) o;  
  
return this.getDescription().equals(fo.getDescription());  
}
```

## **C.**

### **C.1. &**

### **C.2.**

Il faut utiliser le designe patterne Composite et faire que Dessin extends Forme, de plus les methodes sont deja presentes il suffit d'ajouter un override sur getDescription(), de ce fait lors de l'appelle de getDescription() soit c'est une forme et donc on affiche le String habituel, soit c'est un dessin et il affichera tout les formes qu'il as.

## **D.**

### **D.1.**

Il est impossible que point soit considéré comme une forme dans le framework FramPict car un point n'etends pas Forme ce qui est la base d'un Dessin.

### **D.2.**

UML fleme

## **E.**

### **E.1.**

UML fleme