IUT Sénart-Fontainebleau, UPEC — Florent Madelaine
CPOO
IUT Info 2A — 2025-26

Séquence 4 (FA)

1 Introduction

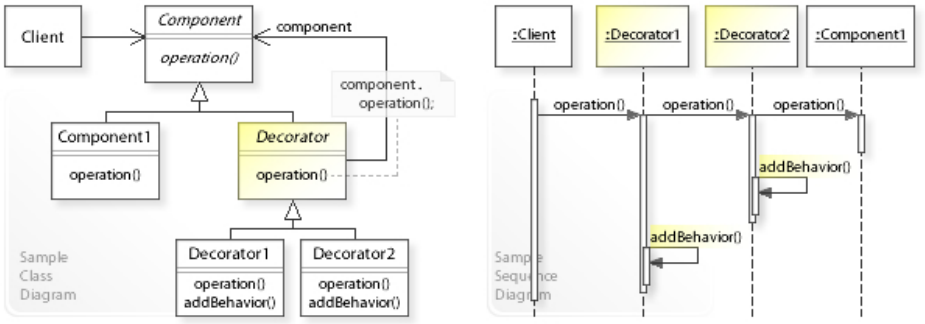
Programme

CM: (mercredi 11/02) patron de conception. Nouveaux exemples : Composite, Décorateur.
TP: (mercredi 11/02) Mise en pratique Décorateur.
CM,TD,TP: (vendredi 13/02) Party Problem avec composite.

Concepts à connaître • Patrons de Conception • Composite • Décorateur

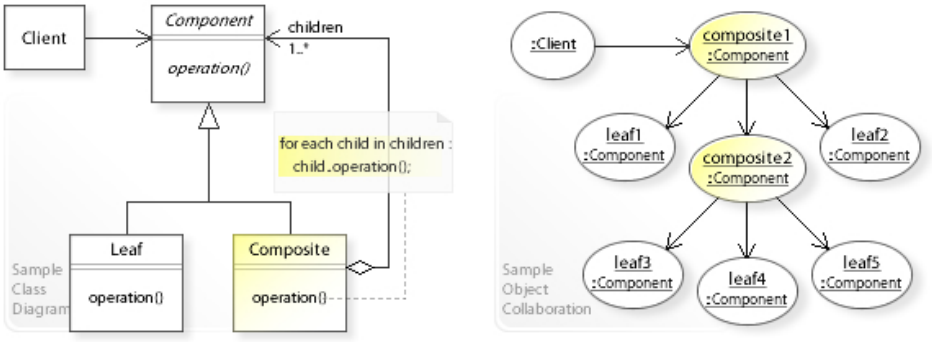
Cette séquence nous étudions en détail deux patrons assez similaire sur la forme mais différent sur leur usage.

Tout d'abord, décorateur, qui n'est pas très compliqué et permet de bien comprendre comment la délégation fonctionne.



By Vanderjoe [CC BY-SA 4.0], from Wikimedia Commons

Ensuite Composite, qui permet de travailler sur des arbres.



By Vanderjoe [CC BY-SA 4.0], from Wikimedia Commons

Il y a plusieurs difficultés.

D'une part, il convient de comprendre comment on peut coder un arbre enraciné. C'est une structure de donnée récursive.

D'autre part, il faut comprendre le principe de calcul de bas en haut dans l'arbre.

Finalement, il y a la subtilité de l'intérêt du patron composite, pas simple à digérer tant que vous n'avez pas essayé de coder avec.

L'exemple utilisé (Party problem) permet d'illustrer comment la programmation dynamique permet de résoudre un problème difficile en général (Maximum Weighted independent set dans un graphe) dans le cas des arbres.

2 Les exercices

Mise en oeuvre : décorateur

On souhaite modéliser des boissons à emporter du genre café au lait, double expresso avec de la crème, ou café calva (espresso corretto comme disent les italiens).

Ce cadre de travail correspond bien au patron de conception décorateur.

On a une interface pour les boissons.

```
public interface Boisson {
  public double getCost();
  public String getIngredients();
}
```

On a des exemples concrets de boisson.

```
public class Expresso implements Boisson {
  @Override
  public double getCost() { return 1; }

  @Override
  public String getIngredients() { return "expresso"; }
}
```

On a une classe abstraite pour les boissons décorées. Par défaut la boisson décorée (abstraite) ne fait rien de plus et délègue le traitement à la boison qu'on souhaite décorer.

```
public abstract class BoissonEtPlus implements Boisson {
  private final Boisson aDecorer;
```

```
  public BoissonEtPlus(Boisson b) {
    this.aDecorer = b;
  }

  @Override
  public double getCost() {
    return aDecorer.getCost();
  }

  @Override
  public String getIngredients() {
    return aDecorer.getIngredients();
  }
}
```

On peut ensuite définir des décorations concrètes.

```
class AvecLait extends BoissonEtPlus {

  public AvecLait(Boisson b) {
    super(b);
  }

  @Override
  public double getCost() {
    return super.getCost() + 0.5;
  }

  @Override
  public String getIngredients() {
    return super.getIngredients() + ", nuage de lait";
  }
}
```

ou encore

```
class Double extends BoissonEtPlus {

  public Double(Boisson b) {
    super(b);
  }

  @Override
  public double getCost() {
    return super.getCost() * 2;
  }

  @Override
  public String getIngredients() {
    return "double dose de (" + super.getIngredients() + ")";
  }
}
```

Exercice 1. Dessiner un diagramme de classe.

Ajouter à ce diagramme des classes pour permettre : de servir du thé, de corriger une boisson en y ajoutant de la goutte.

Dessiner un diagramme objet pour les boissons suivantes : thé au lait, double expresso corrigé, expresso avec deux doses de gouttes, une commande de deux espressi.

Exercice 2 (Comportement)**.** Dessiner un diagramme de séquence pour l'appel printInfo(b1); ci dessous.

```
public class Exemple {

    public static void printInfo(Boisson b) {
        System.out.println("Coût : " + b.getCost() + "; Ingrédients: " + b.getIngr
    }

    public static void main(String[] args) {
        Boisson b = new Expresso();
        Boisson b1 = new AvecLait(b);
        printInfo(b1);
    }
}
```

Mise en oeuvre : Party Problem

Exercice 3 (patron)**.** Vous devez mettre en oeuvre des classes java utilisant le patron de conception *composition* pour résoudre le problème de la fête. Plus de détails concernant ce patron sont donnés dans le cours, voir sur internet. Plus de détails concernant java sont disponibles à la fin du sujet.

Le problème de la fête est le suivant. Une organisation comporte différentes personnes qui ne sont pas toutes très marrantes. Chaque personne a un *fun factor* qui est un entier positif ou nul.

La hiérarchie de l'entreprise est un arbre. On ne peut pas s'amuser à une fête si on est en présence de son supérieur direct (n+1) ou bien si on est en présence de son subalterne direct (on est le n+1 de cette personne).

Une fête consiste donc à choisir des personnes de sorte qu'aucun participant ne soit en présence de son supérieur direct. La qualité d'une fête est mesuré par la somme du *fun factor* des participants.

Étant donné une hiérarchie de personnes, on veut calculer la qualité de la meilleure fête possible.

Indication. Il faut procéder de bas en haut et calculer pour une personne x deux meilleures fêtes possibles pour lui et ses subalternes (pas forcément directs): celle où x n'est pas invité, celle où x peut être invité.

Détails concernant Java. Vous devez avoir une classe Travailleur et une classe Chef avec des constructeurs prenant en argument un entier positif qui est le *fun factor*. On ajoute un subalterne à un chef avec la méthode `addSubalterne`. On calcule la meilleure fête pour une organisation donnée par son chef en appelant la méthode `bestParty`.

Votre réalisation doit fonctionner avec tous les fichiers `Exemple?.java` fournis.

Exercice 4 (patron)**.** Si un sommet ne mémorise pas les résultats dans un attribut, vous parcourez inutilement l'arbre plusieurs fois. Reprenez l'exercice précédent en sauvant les résultats. Ainsi le calcul est fait lors du premier appel mais pas pour les suivants qui se contentent de retourner la valeur de ce premier calcul.

Exercice 5 (patron)**.** Vérifiez les calculs des exercices précédents en utilisant une méthode plus brutale.

Indication. Soit vous procédez de manière exhaustive en regardant tous les sous-ensembles de personnes (méthode *perebor*), soit pour éviter de faire trop de calcul vous faites un *bactrack*.