2i002 - Design Pattern

Vincent Guigue

De la Bidouille OO à la Programmation OO

Principes Objets :

- Encapsulation
- Héritage & abstraction
- Polymorphisme

Question (difficile)

Comment combiner ces différents outils pour écrire des programmes ou boîtes-à-outils flexibles, réutilisables, efficaces, etc, etc... ?

Design Patterns

- Définition :
  Elément de conception réutilisable permettant de résoudre un problème récurrent en programmation orientée objet.
- Historique :
  - Inspirées d'une méthode de conception d'immeuble en architecture [Alexander, 77]
  - Introduites par le «GOF» dans le livre Design Patterns en 1999 (dans le «GOF» 23 patterns «standards»)

Pourquoi les patterns ?

Des « recettes d'expert » ayant fait leurs preuves.
Un vocabulaire commun pour les architectes logiciels.
Incontournable dans le monde de la P.O.O

Idiome de la programmation

Un «bout de programme Java» que l'on tape systématiquement lorsque l'on veut résoudre un problème récurrent

- Parcours d'un tableau

```
1 for(int i=0;i<tableau.length;i++) {
2    tableau[i] = ... patati patata ...
3 }
```

- Rattrapage d'exception

```
1 try {
2   // fonctionnalité XYZ du jdk
3   XYZ(...);
4 } catch(XYZException e) {
5   e.printStackTrace(System.err);
6 }
```

Conception vs. Programmation

Les designs patterns sont des éléments de conception (objet)
Pas au niveau du langage de programmation (donc pas spécifique à Java, les patterns «marchent» aussi en C++, en Python...)

Citations

« Chaque patron décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le coeur de la solution à ce problème, d'une façon telle que l'on puisse réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière » Christopher Alexander - 1977.

« Les patrons offrent la possibilité de capitaliser un savoir précieux né du savoir-faire d'experts » [Buschmann] - 1996.

Coté implémentation :

- Principe 1 :
  Favoriser la composition (lien dynamique, flexible) sur l'héritage (lien statique, peu flexible) La délégation est un exemple d'outil pour la composition
  Attention : favoriser ne veut pas dire remplacer systématiquement, l'héritage est largement utilisé aussi !
- Principe 2 :
  Les clients programment en priorité pour des interfaces (ou abstractions, classes abstraites, etc) plutôt qu'en lien direct avec les implémentations (classes concrètes)

Description standard (GOF, wikipedia...)

- Nom et classification (ex. bridge/structurel)
- Intention : description générale et succinte
- Alias : autres noms connus pour le pattern
- Motivation : au moins 2 exemples/scénarios
- Indications d'utilisation : une liste des situations qui justifient de l'utilisation du pattern
- Structure : un diagramme de classe UML indépendant du langage de programmation
- Constituants : Explication des différentes classes qui interviennent dans la structure du pattern
- Implémentation : les principes, pièges, astuces, techniques pour implanter le pattern dans un langage objet donné (pour nous, Java).
- Utilisations remarquables : des programmes réels dans lesquels on trouve le pattern
- Limites : les limites concernant l'utilisation du pattern

Iterator

Objectif

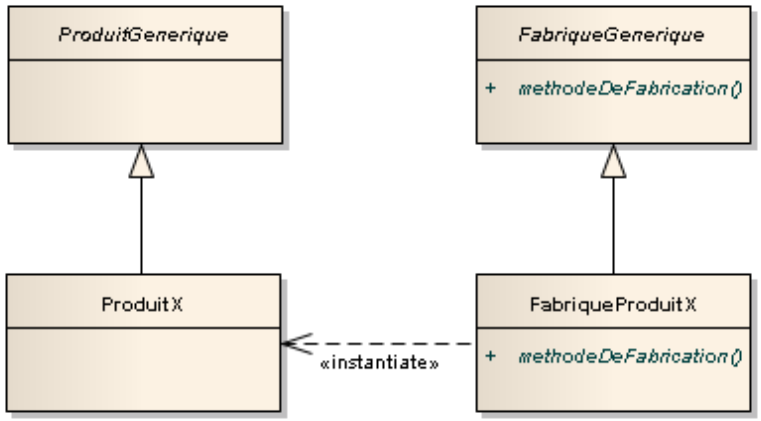
Découpler le choix des structures de données des implémentations d'algorithmes

```
1 // code spécifique
2 Iterator<MyType> iter = list.iterator();
3
4 // code commun à toute structure de données
5 while (iter.hasNext()) {
6     System.out.print(iter.next());
7     if (iter.hasNext())
8         System.out.print(", ");
9 }
```

Factory (Construction)

Problème du `new`

Appelé directement sur une classe concrète : moins évolutif...
Plein de variantes normalisées (méthode, classe, classe abstraite...)



- Exemple : Lecteur d'image
- Le client voit une fabrique générique...
- ... Qui invoque la bonne factory concrète en fonction de l'extension de l'image

Singleton (Construction)

Garantir l'unicité de l'instance

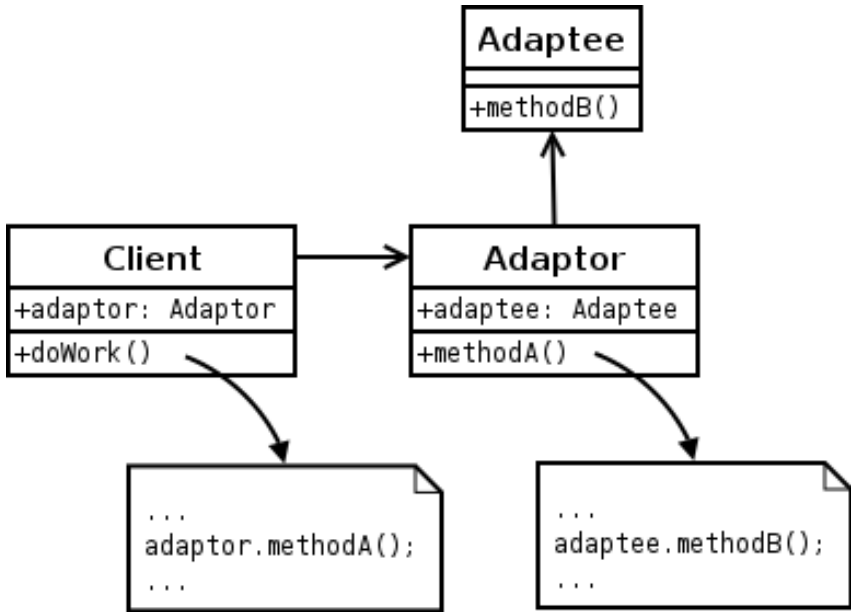
- Pouvoir utiliser ==
- A utiliser quand il ne peut y avoir qu'une instance (eg Console)

```
1  public class Singleton {
2      private static Singleton INSTANCE = null;
3
4      private Singleton() {}
5
6      private synchronized static void createInstance() {
7          if (INSTANCE == null) {
8              INSTANCE = new Singleton();
9          }
10     }
11     public static Singleton getInstance() {
12         if (INSTANCE == null) createInstance();
13         return INSTANCE;
14     }
15 }
```

Adapter (Structure)

Idée : réutiliser une fonction déjà implémentée...

... Mais dans une architecture contrainte
= Adapter la classe
Repose sur le principe de la délégation

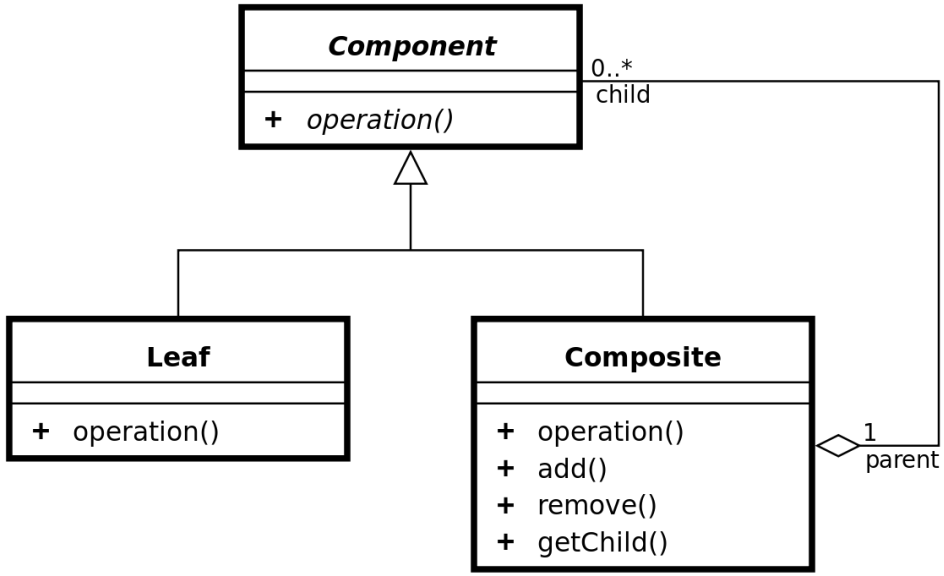


- Très très fréquent : réutiliser une classe sans la modifier...
  Alors que le client est déjà spécifié
  - on a un objet d'une classe `Client` qui veut utiliser un objet de classe B mais le client ne sait manipuler des objets de classe A. Il faut donc adapter la classe B à l'interface de A pour que le Client puisse finalement utiliser B.
- Technique d'optimisation, calcul de graphes...
  - Adapter les objets à passer en paramètres... Ou adapter les algorithmes.

Composite (Structure)

Un objet E, composé d'objets E

- une sous-figure, composée de figure
- une stratégie, composée de sous-stratégie



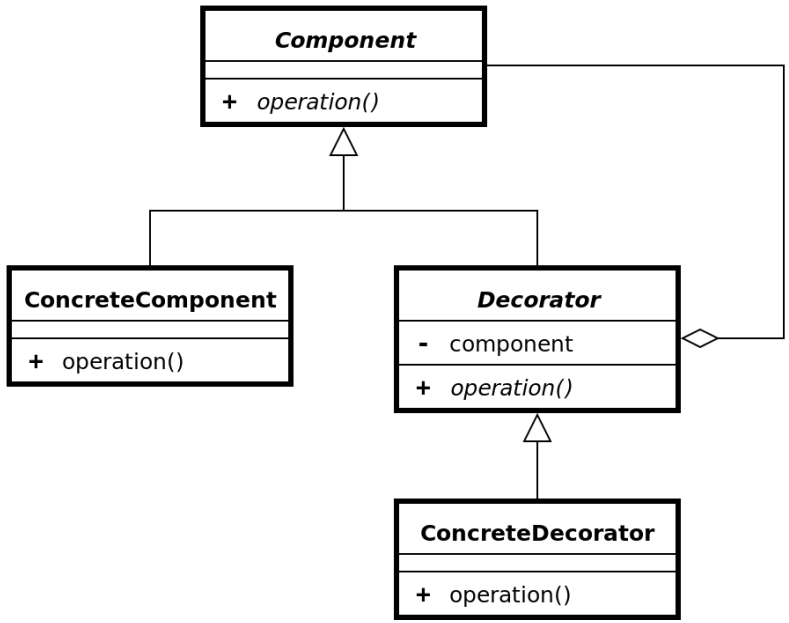

```
1 public class GroupeDeFigure extends Figure{
2   private ArrayList<Figure> composition;
3   // constructeur
4   public GroupeDeFigure(){
5     super(...);
6     composition = new ArrayList<Figure>();
7   }
8   // Instanciation des méthodes de Figure
9   // ... En délégant svt vers les éléments de la composition
```

Decorator (Structure)

Ajouter une fonctionnalité...

... sur n'importe quel objet d'une arborescence de classe

- Permettre de lire des informations de haut niveau (String, double...) dans n'importe quel flux
- Rendre n'importe quelle stratégie prudente

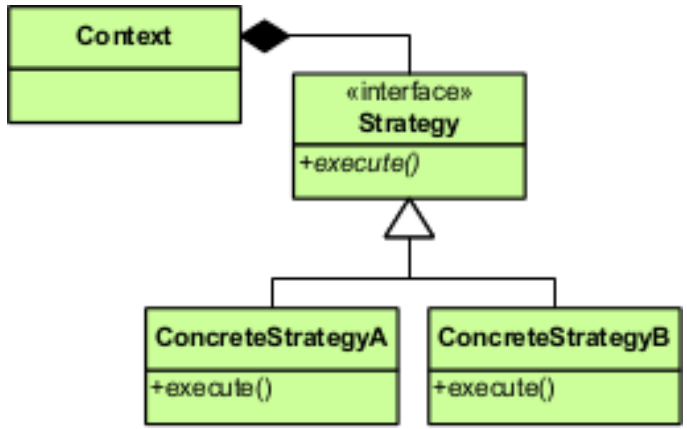



```
1 public MonObjetDecore extends MonObjet{
2   private MonObjet obj;
3   public MonObjetDecore(MonObjet obj){ this.obj = obj;}
4
5   public void mafonction(){
6     if(cas 1) return obj.mafonction()
7     else // code spécifique
8   }
```

Strategy (Comportement)

Gérer le comportement distinctement de l'objet

- Robot... Qui marche, rampe, vole... Ou une combinaison des 3
- Le client gère la stratégie
- On peut créer de nouvelles stratégies avec des robots existants




```
1 public class Robot{
2   private Strategy str;
3   public Robot(Strategy str){this.str = str;}
4
5   public void action(){
6     str.action(); // ou str.action(this);
7   }
```

⇒ Une alternative (beaucoup plus flexible) à l'héritage sur les robots

Note : DP compatible avec `Composite`

Et plein d'autres encore

- Visitor
  - vient éxecuter un algorithme dans un objet
- MVC : model view controler
  - pour les interfaces graphiques, séparation des éléments clés
- ...

Comme un second niveau de programmation (de la conception en fait !)
A continuer avec de l'UML et d'autres UE de génie logiciel