

# Patterns Modèles de Conception Orientée Objets

Bibliographie:

Design Patterns

E.Gamma et ...

International Thomson Publishing

Patterns in Java (tomes 1 et 2)

M. Grang

Wiley

-1-

## Les bonnes pratiques de la Conception Objet

- Qu'est-ce qu'une bonne conception Objet ?
  - Une conception qui permet l'évolution du logiciel sans re-programmations excessives !
- Solution
  - Séparer du reste ce qui est susceptible de changer
  - Utiliser des interfaces et non des implémentations pour typer les données.

-2-

## Exemples de patterns: leur raison d'être

- Strategy
  - Le client ne dépend plus de l'algorithme qui peut changer
- Observer
  - Le client ne dépend plus du nombre des observateurs
- Decorator
  - Le client n'est pas concerné par d'éventuelles extensions
- Factory
  - Le client ne dépend plus du choix des objets de service
- Visitor
  - Le code de la structure de donnée n'est plus perturbé par les nouvelles opérations

-3-

## Commencer par le simple

Les patterns (bonnes pratiques) peuvent relever du bon sens

Comme par exemple la Façade

ou le 'pattern Adapter'

Ils peuvent également relever des propriétés des langages de POO

Comme le 'pattern Template'

ou le 'pattern Strategy'

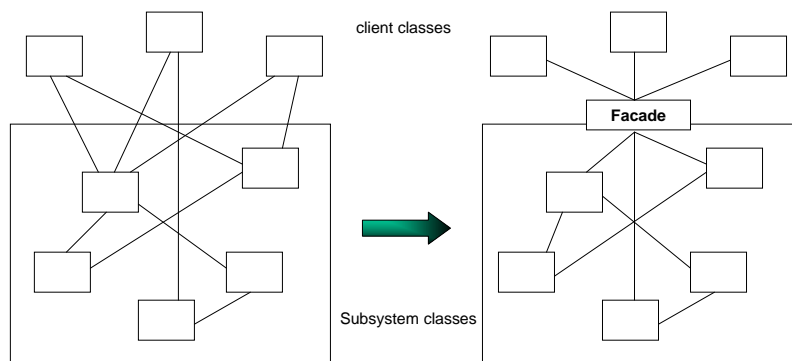
-4-

# Pattern Facade

- **Problème:**
  - Un sous-système est fait de n interfaces différentes compliquant son utilisation
- **Solution:**
  - Créer une seule façade pour ce sous-système.

-5-

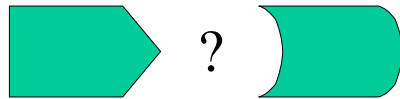
# Pattern FACADE



-6-

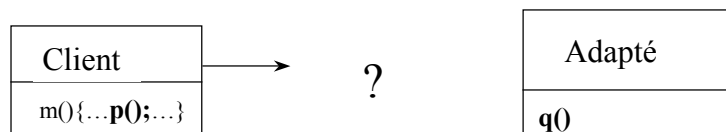
## Autre Pattern : Adapter les différences

- Problème
  - On dispose de deux logiciels non prévus l'un pour l'autre !
- Solution
  - Mettre entre les deux un Adapteur !



-7-

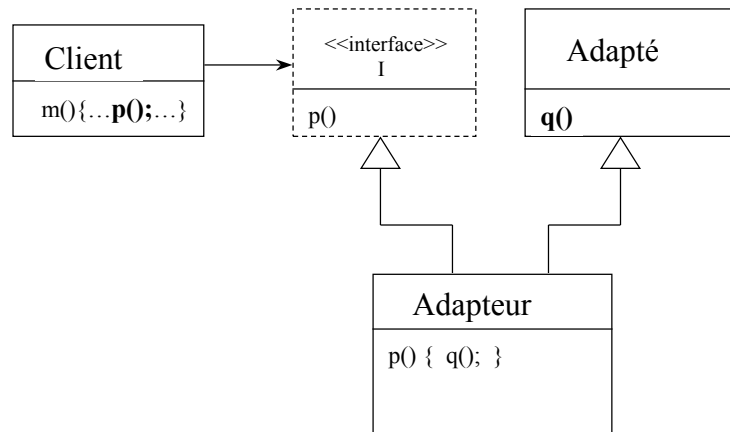
## Adapter ?



On veut utiliser une classe existante dont l'interface ne convient pas

-8-

## Pattern Adapter



On peut faire cela en utilisant une interface par exemple

-9-

## Encore un pattern simple Permettre l'évolution du code d'une méthode ?



Notre application appelle la méthode p()

Comment faire évoluer à l'avenir le code de p() si celui-ci doit changer?

-10-

# Utiliser les classes abstraites

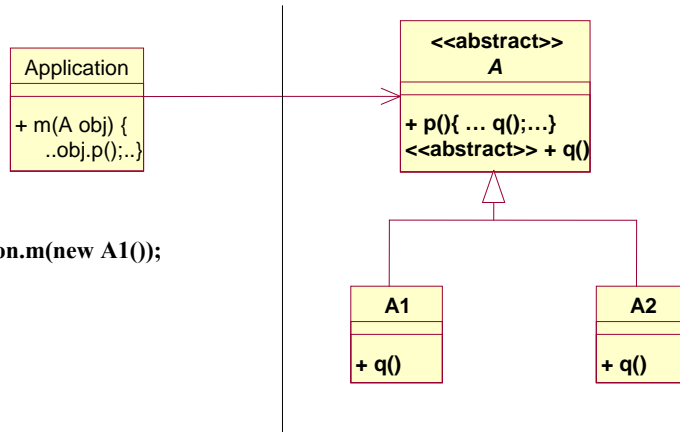
A peut être rendue abstraite relativement à q()



-11-

# Pattern Template

Et l'héritage avec liaison dynamique



`Application.m(new A1());`

Le code de q et donc de p dépend de la sous-classe de A qui est passée en paramètre

-12-

## Template : application

```
Ordre max(Ordre o){  
    return this.inf(o) ? o : this;  
}
```

```
<<abstract>>Ordre
```

```
Ordre max(Ordre o){  
    ...}
```

```
abstract inf(Ordre);
```

```
ClasseConcrète
```

```
inf(Ordre){ ... }
```

```
...
```

-13-

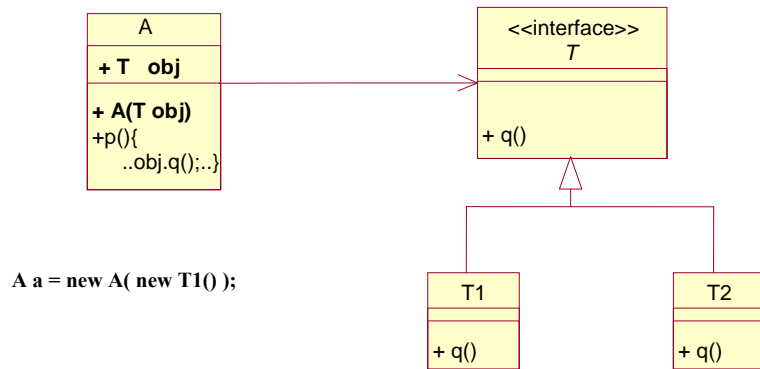
## Application à Java

```
abstract class Ordre {  
    Ordre max(Ordre o) {  
        return this.inf(o) ? o : this;  
    }  
    abstract boolean inf(Ordre autre);  
}  
  
class Entier extends Ordre {  
    int i;  
    ...  
    public boolean inf(Ordre autre){ return (i<=((Entier)autre).i)?true:false;  
    }  
}  
  
class Chaîne extends Ordre {  
    String s;  
    ...  
    public boolean inf(Ordre autre){ return (s.compareTo(((Chaîne)autre).s)<=0)?true:false;  
    }  
}
```

-14-

# Une autre façon de faire !

Au lieu d'une classe abstraite on peut aussi utiliser une interface  
 Au lieu de la liaison dynamique utiliser la composition

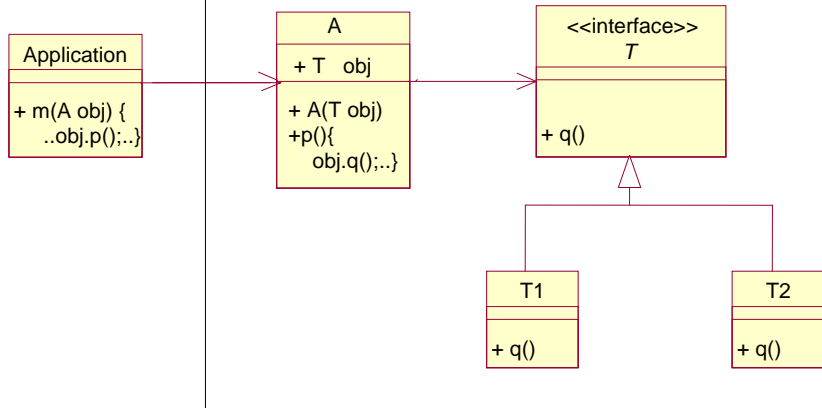


-15-

# Pattern Strategy

`A a= new A( new T1() );`

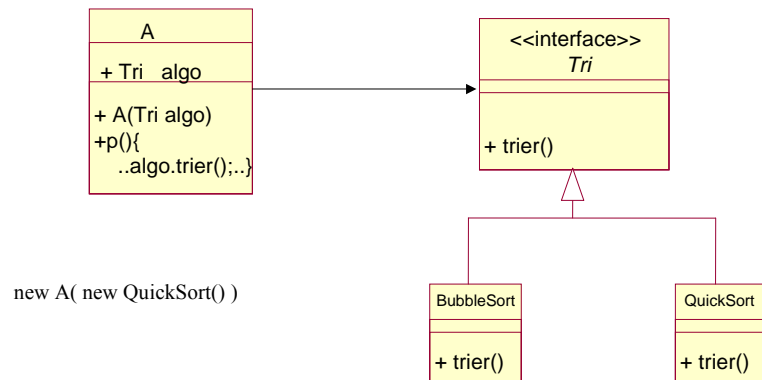
`Application.m(a);`



-16-

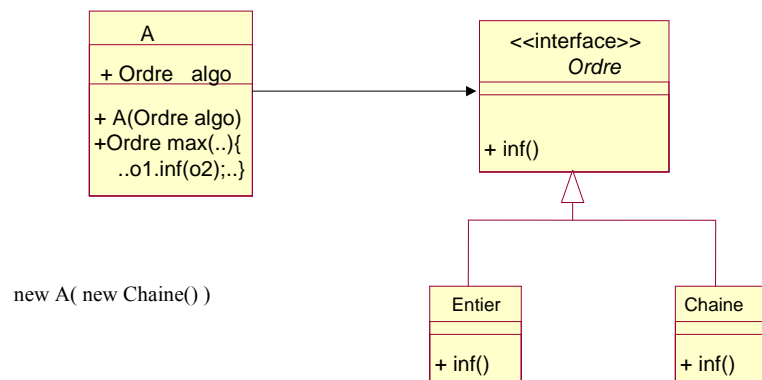


# Stratégie



-17-

# Stratégie



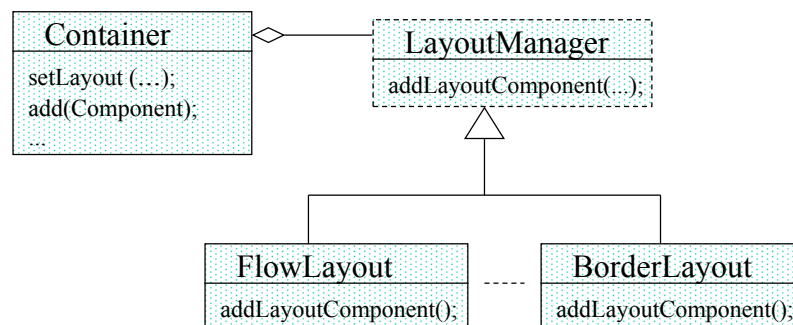
-18-

# Stratégie

```
public class Contexte {  
    ...  
    Ordre max(Ordre o1, Ordre o2) {  
        return o1.inf(o2) ? o2 : o1;  
    }  
}  
  
interface Ordre {  
    boolean inf(Ordre autre);  
}  
class Entier implements Ordre {  
    int i; ...  
    public boolean inf(Ordre autre){ return (i<=((Entier)autre).i)?true:false;}  
}  
class Chaine implements Ordre {  
    String s;...  
    public boolean inf(Ordre autre){ return (s.compareTo(((Chaine)autre).s)<=0)?true:false;}  
}
```

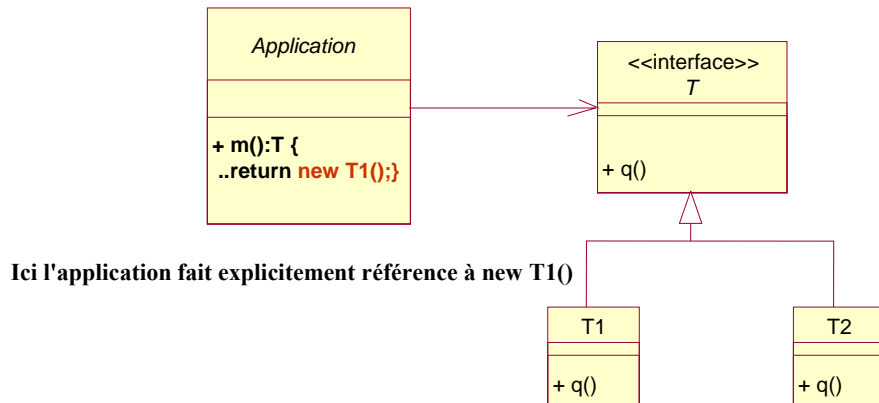
-19-

# Stratégie et Awt



-20-

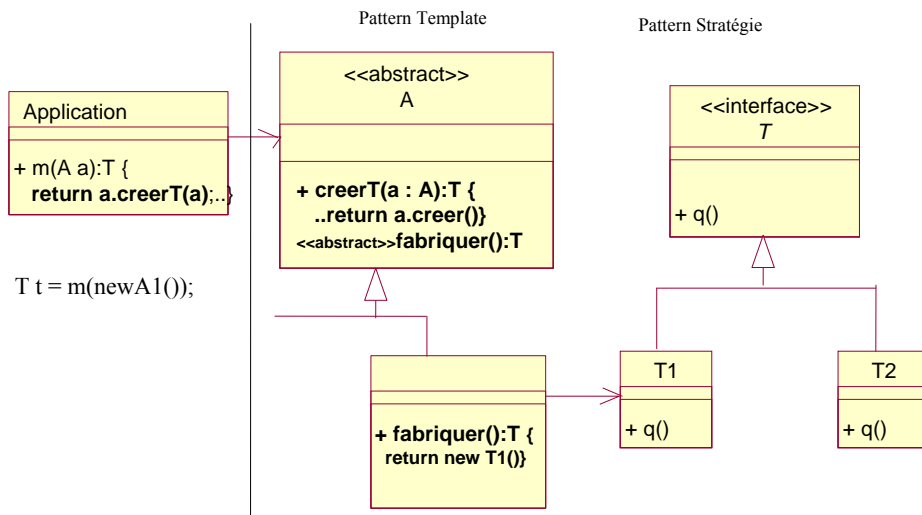
## Un autre pattern ? Rendre notre code indépendant de l'objet à créer ?



-21-

## Pattern Factory

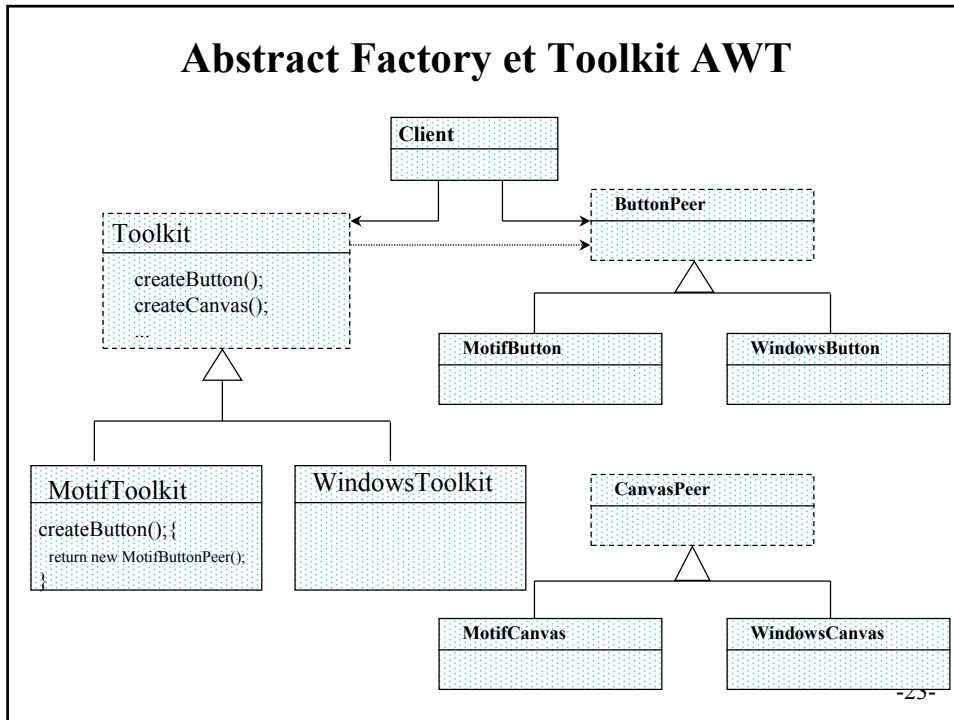
On combine le pattern template et le pattern strategy



Utilisé quand on souhaite déléguer à un producteur la création d'objets de types différents

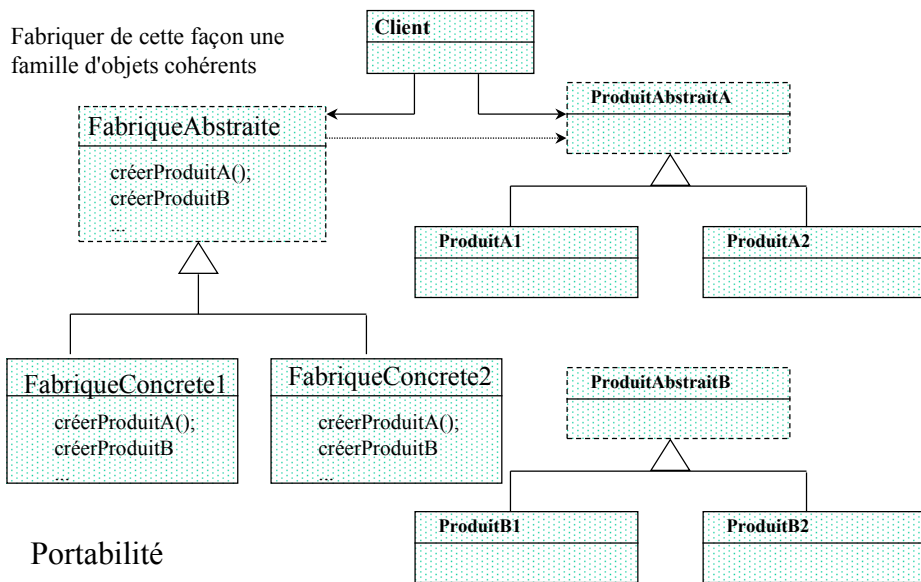
-22-

## Abstract Factory et Toolkit AWT



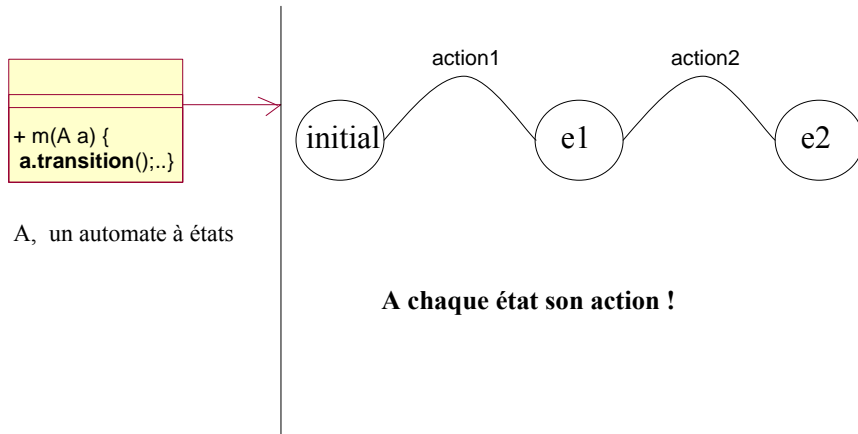
## Abstract Factory (Fabrique Abstraite p.101)

Fabriquer de cette façon une famille d'objets cohérents



Portabilité

## Un pattern de plus Modéliser un graphe état transition



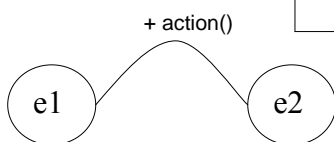
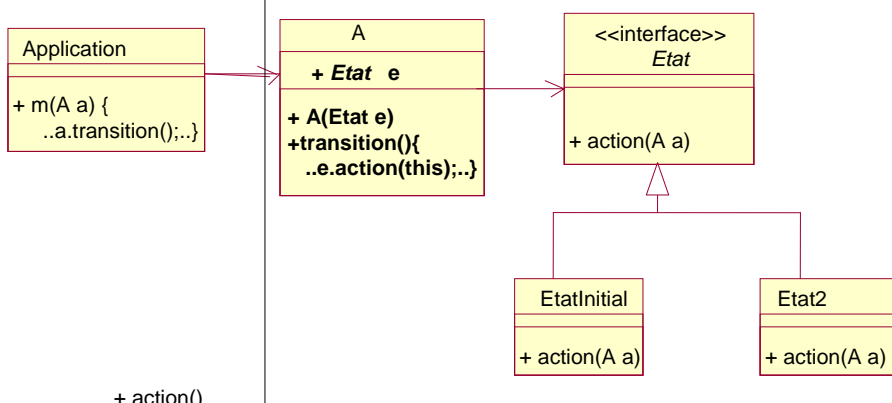
-25-

## Pattern State (Etat)

`A a= new A( new EtatInitial() );`

`Application.m(a);`

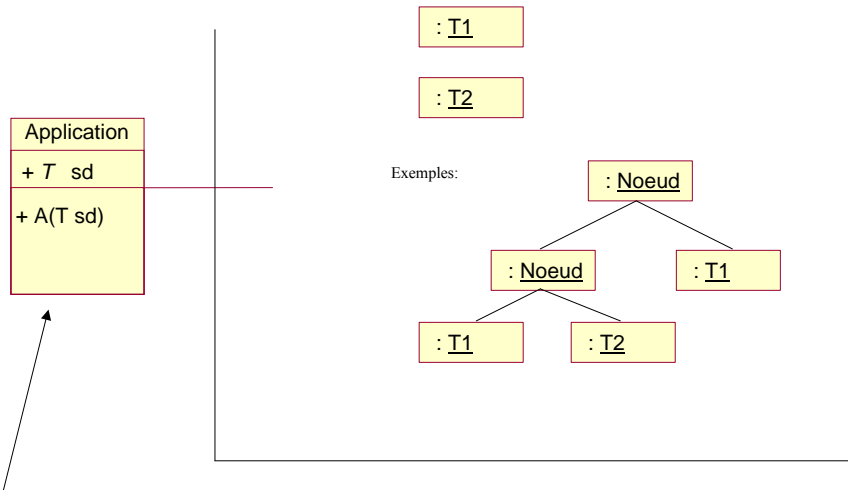
Proche du pattern stratégie, mais but et comportement différent !



a.e est modifié par l'action ce qui modifie la prochaine action ...

-26-

# Construire des structures de données complexes

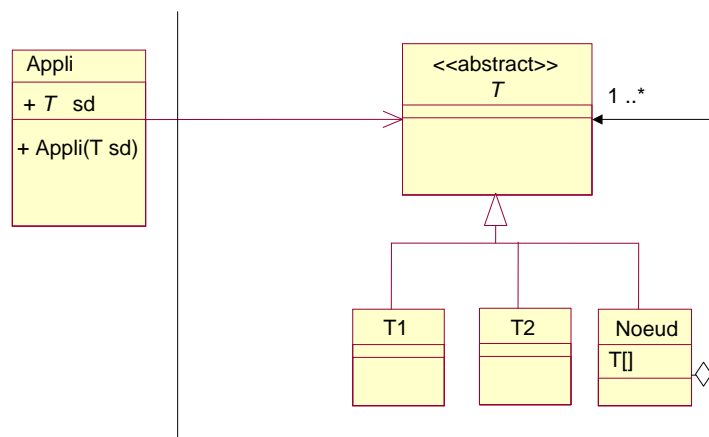


Je souhaite que ma structure de donnée ( ici sd) soit aussi bien une donnée élémentaire qu'une donnée complexe

-27-

# Pattern Composite

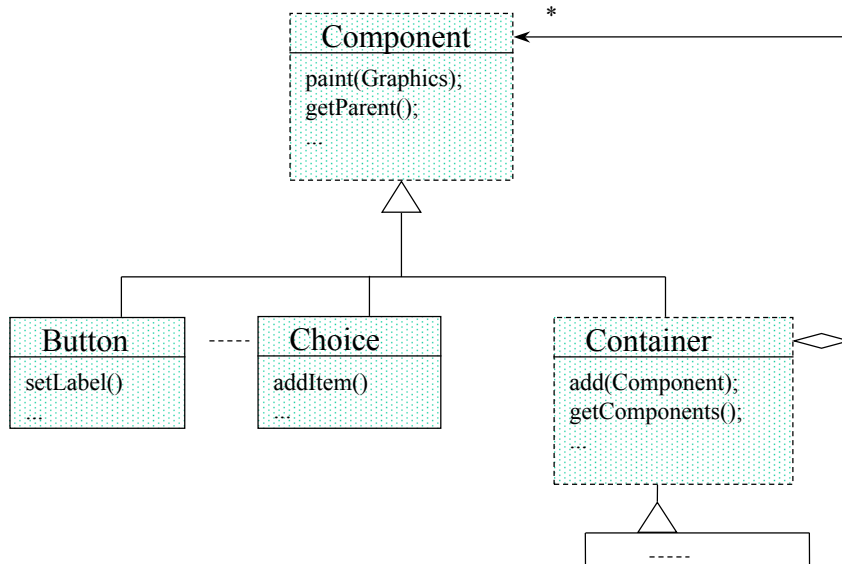
structures de données complexes



T sd = new Nœud(new T[] {new T1(), new Noeud(...)});

-28-

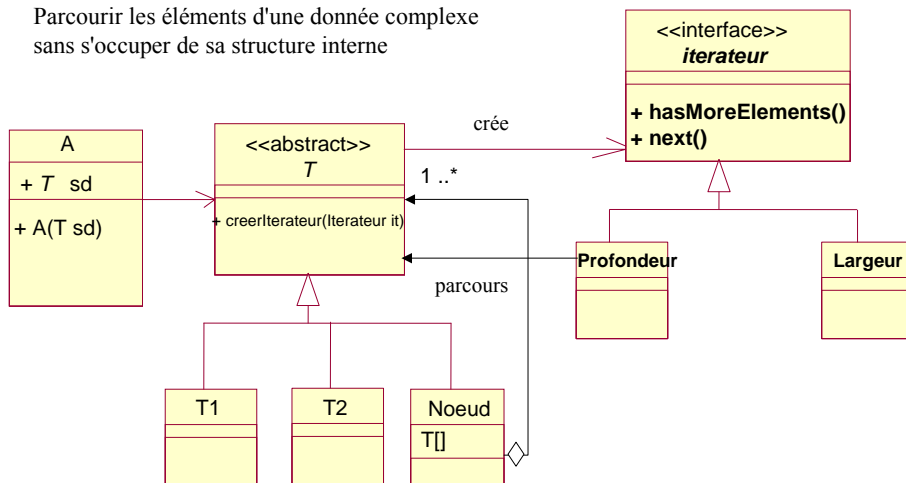
## le modèle Composite dans AWT



-29-

## Pattern Iterator

Parcourir les éléments d'une donnée complexe sans s'occuper de sa structure interne



A une structure de donnée, on associe différents parcours des éléments

-30-

## itérateur java: Enumeration

```
public interface java.util.Enumeration
{
    public abstract boolean hasMoreElements();
    public abstract Object nextElement();
}
```

-31-

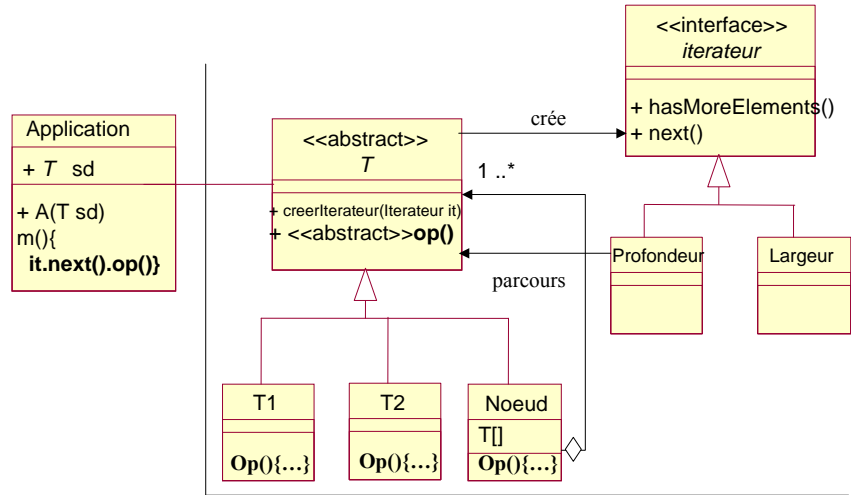
## Java: itérateurs sur une Hashtable

```
public class java.util.Hashtable
    extends java.util.Dictionary
    implements java.lang.Cloneable {
    public Hashtable();
    public void clear();
    public Object clone();
    public boolean contains(Object value);
    public boolean containsKey(Object key);
    public Enumeration elements();
    public Object get(Object key);
    public boolean isEmpty();
    public Enumeration keys();
    public Object put(Object key, Object value);
    public Object remove(Object key);
    public int size();
    public String toString();
}
```

-32-



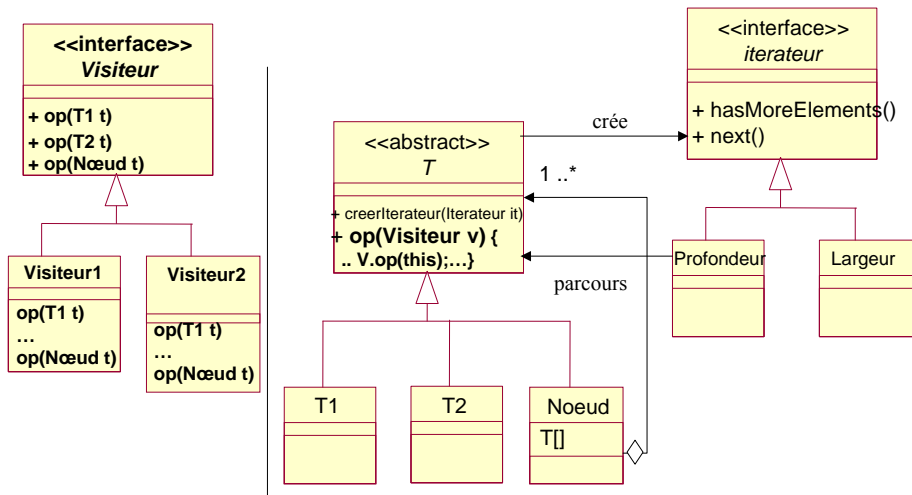
## Exécuter des opérations lors d'un parcours de structure de données



Mais comment faire si l'on veut ajouter de nouvelles opérations sur un parcours de la structure ?

-33-

## Pattern Visitor



Si l'on doit parcourir une structure pour différents calcul sur les nœuds, plutôt que de polluer la structure de donnée, mieux vaut créer les opérations dans un visiteur.

-34-

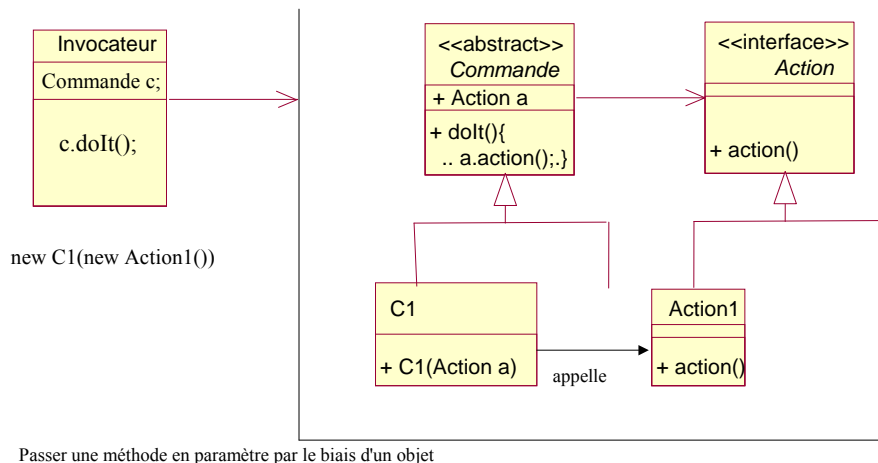
## Call back

- Passer une méthode en paramètre à différents clients qui devront ensuite l'appeler
- Problème !
  - On ne peut passer que des objets
- Solution
  - Encapsuler la méthode dans un Objet Commande

-35-

## Pattern Command

La commande encapsule une Action



-36-

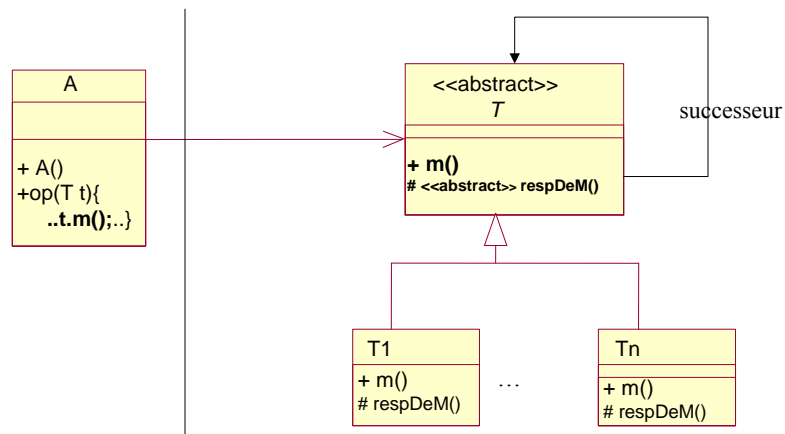
## Responsabilité d'un traitement

- Problème:
  - Ne pas lier strictement l'émetteur d'une requête de traitement et son récepteur
- Solution
  - Chaîner entre eux les objets récepteurs et leur laisser choisir celui qui traitera la requête.

-37-

## Pattern Chaîne de responsabilité

L'appel de m est confié à une liste d'objets. L'un de ceux-ci se reconnaît compétent et traite m



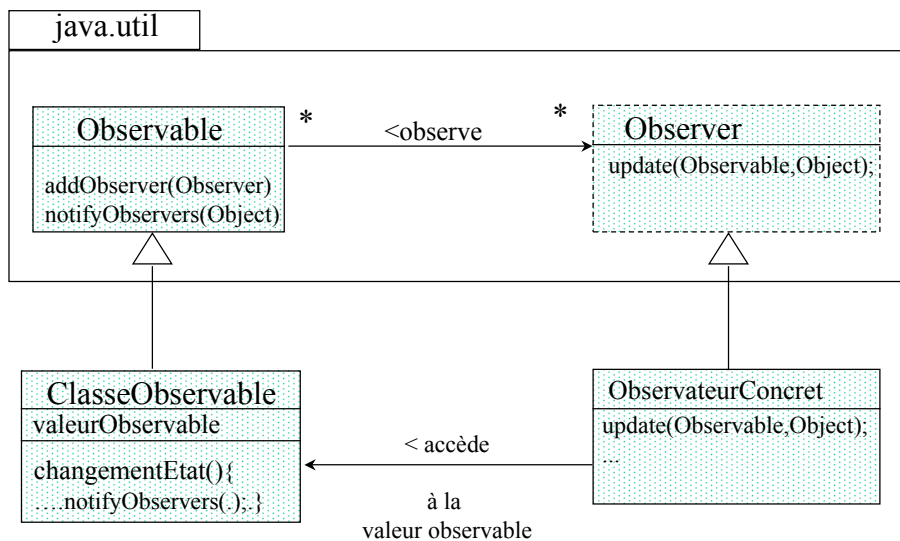
-38-

# Exemple

```
abstract class T {  
  
    private protected t successor;  
  
    public void m() {  
        if(responsabilitéDeM()) return;  
        else if( successor != null)  
            successor.m();  
    }  
  
    abstract protected boolean responsabilitéDeM();  
}
```

-39-

# Pattern Observer



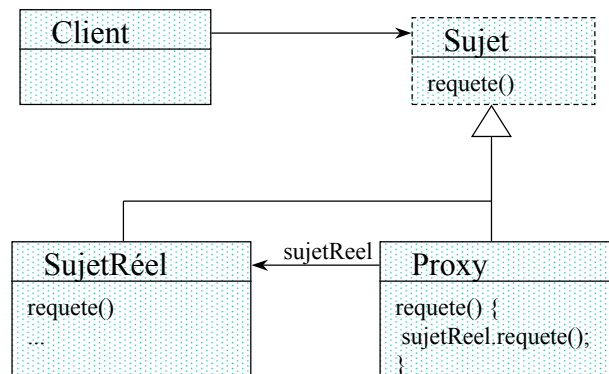
-40-

## Subrogation

- Problème
  - Ne pas fournir l'objet demandé mais un représentant
    - Soit par soucis de sécurité
    - Soit pour des raisons de durée d'attente
    - Soit pour implémenter un protocole d'accès distant masqué
- Solution
  - Le proxy !

-41-

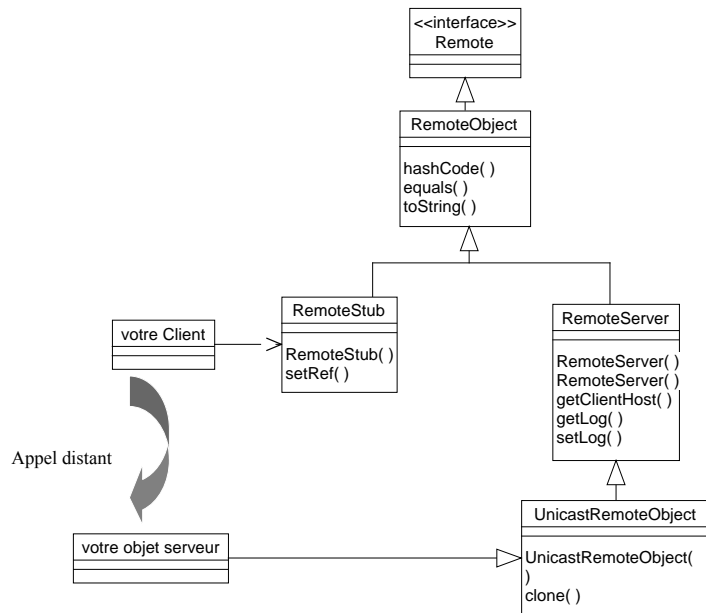
## Proxy (ou Procuration)



Création d'objets lourds à la demande (Image)  
Fourniture d'un représentant local d'un objet distant

-42-

## Proxy Rmi



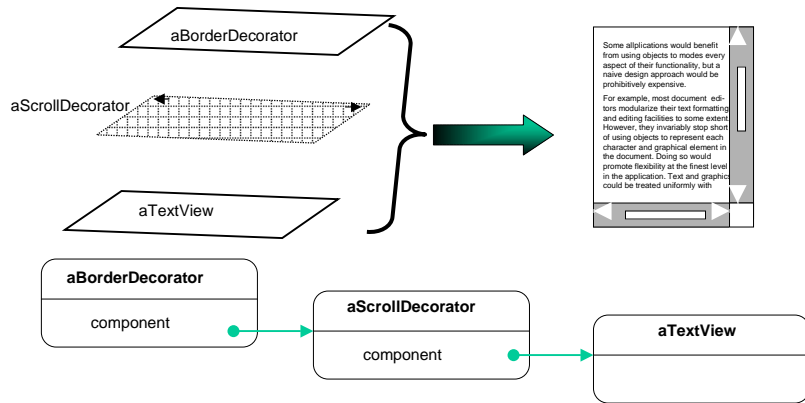
-43-

## Décorer un objet !

- Problème:
  - Etendre les fonctionnalités d'un objet sans sous-classer
- Solution
  - Emballer l'objet dans un autre qui ajoutera cette fonctionnalité!

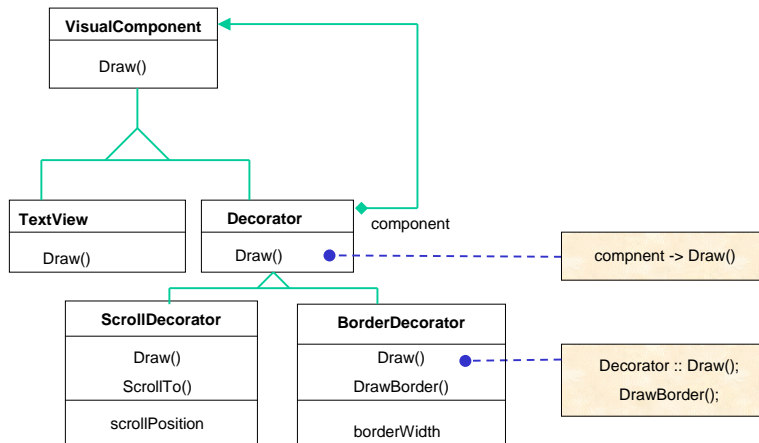
-44-

# Exemple



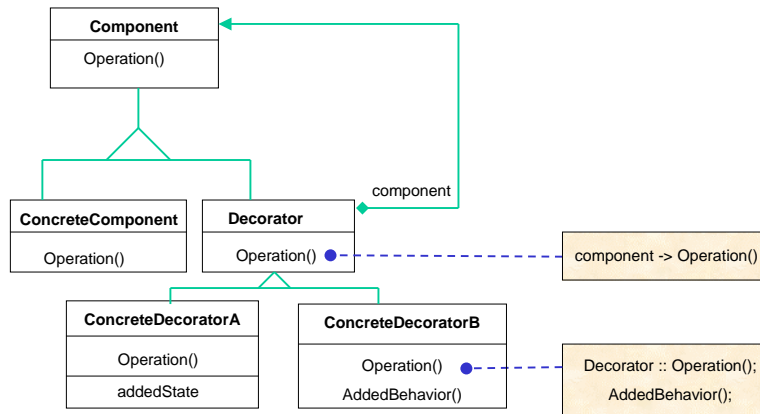
-45-

# DECORATOR - Motivation



-46-

# DECORATOR – Structure



-47-