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

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.

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

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

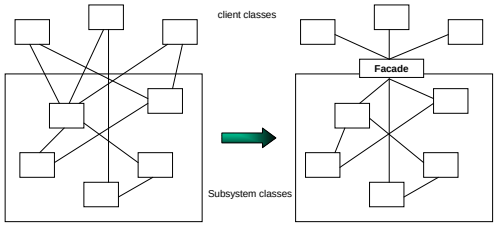
Comme le 'pattern Template'

ou le 'pattern Strategy'

Pattern Facade

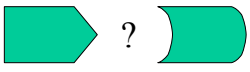
- Problème:
  - Un sous-système est fait de n interfaces différentes compliquant son utilisation
- Solution:
  - Créer une seule façade pour ce sous-système.

Pattern FACADE

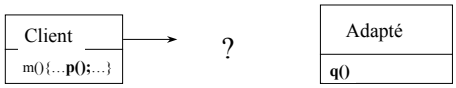


Autre Pattern : Adapter les différences

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

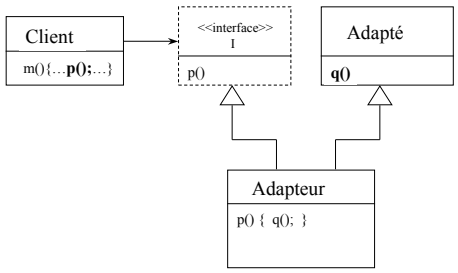


Adapter ?



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

Pattern Adapter



On peut faire cela en utilisant une interface par exemple

Encore un pattern simple

Permettre l'évolution du code d'une méthode ?

Notre application appelle la méthode p()

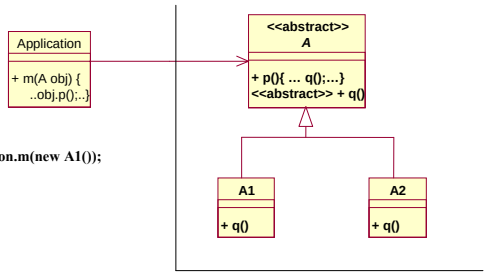
Comment faire évoluer à l'avenir le code de p() si celui-ci doit changer?

Utiliser les classes abstraites

A peut être rendue abstraite relativement à q()

Pattern Template

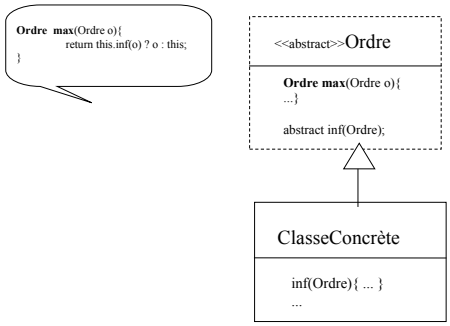
Et l'héritage avec liaison dynamique



Application.m(new A1());

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

Template : application



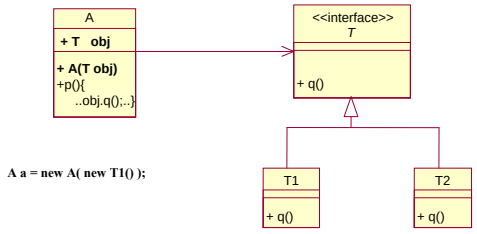
Application à Java

```
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 Chaine extends Ordre {
        String s;
        ...
        public boolean  inf(Ordre autre){ return (s.compareTo(((Chaine)autre).s)<=0)?true:false;
        }
}
```


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

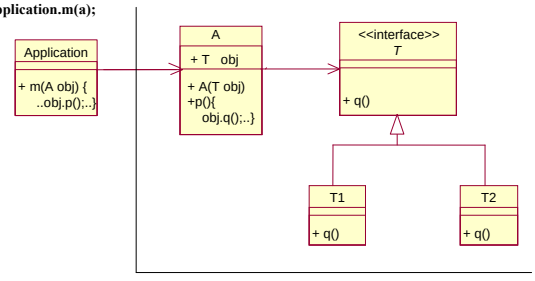


A a = new A(new T1());

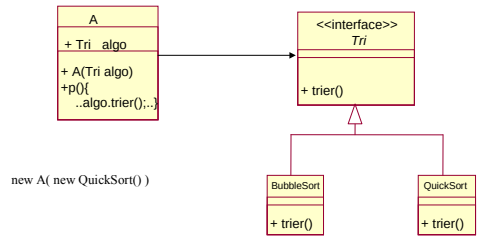
Pattern Strategy

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

Application.m(a);

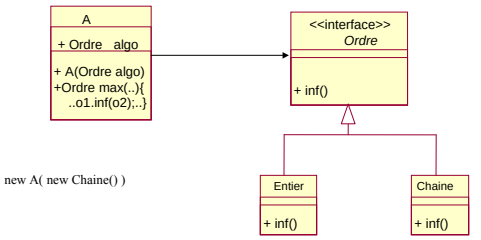


Stratégie



new A(new QuickSort())

Stratégie

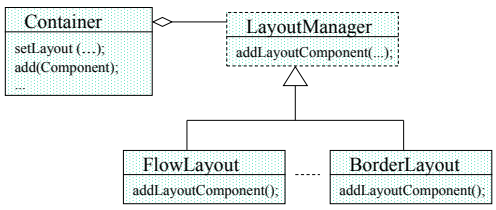


new A(new Chaine())

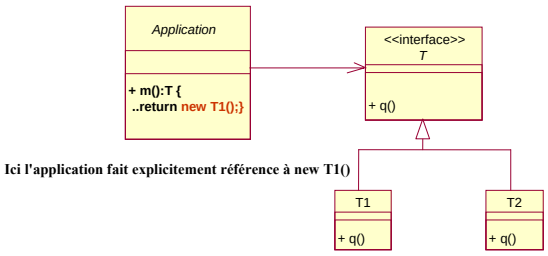
Stratégie

```
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;}
}
```


Statégie et Awt



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

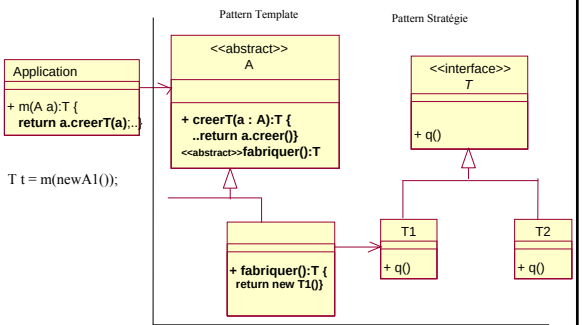


Ici l'application fait explicitement référence à new T1()

-21-

Pattern Factory

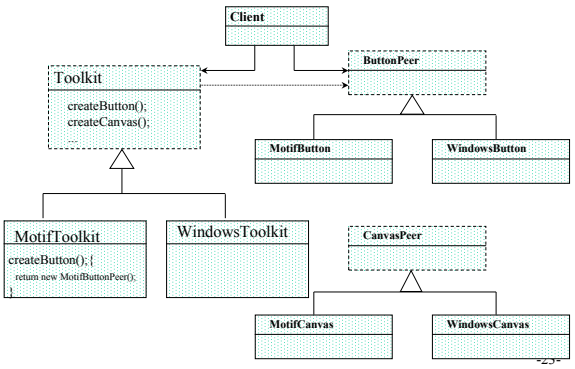
On combine le pattern template et le pattern strategy



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

-22-

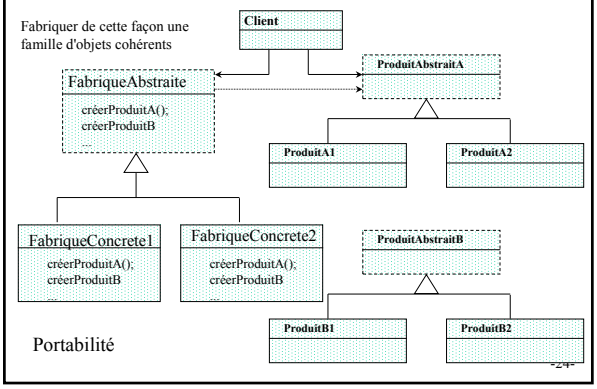
Abstract Factory et Toolkit AWT



-23-

Abstract Factory (Fabrique Abstraite p.101)

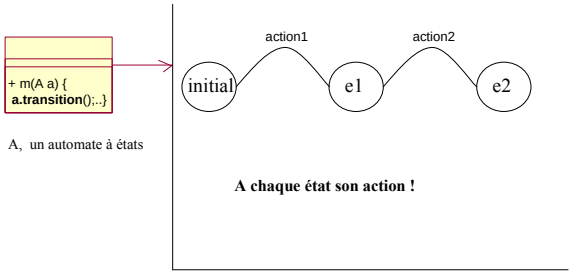
Fabriquer de cette façon une famille d'objets cohérents



Portabilité

-24-

Un pattern de plus
Modéliser un graphe état transition



A, un automate à états

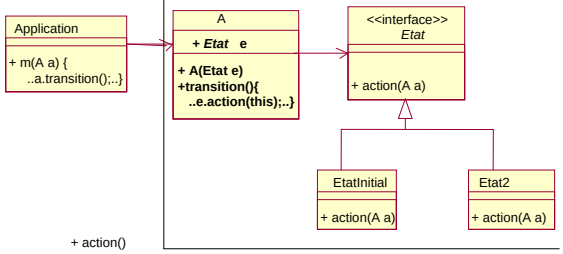
A chaque état son action !

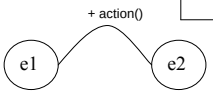
Pattern State (Etat)

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

Application.m(a);

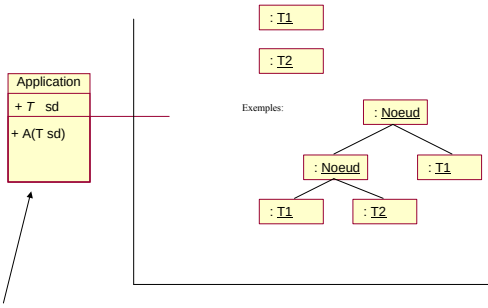
Proche du pattern stratégie, mais but et comportement différent !





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

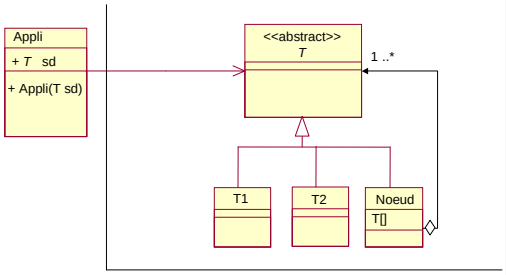
Construire des structures de données complexes



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

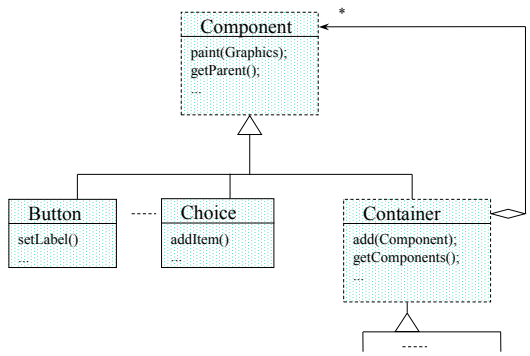
Pattern Composite

structures de données complexes



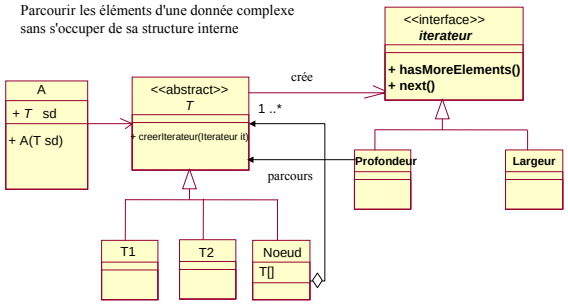
T sd = new Nœud(new T[]{new T1(), new Noeud(…)});

le modèle Composite dans AWT



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

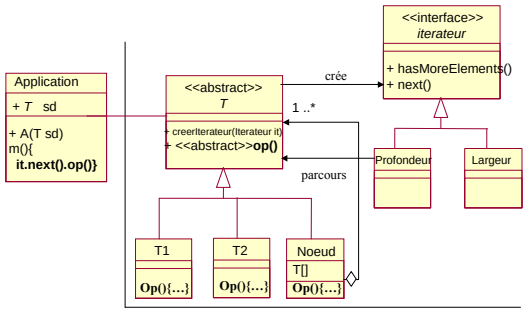
itérateur java: Enumeration

```
public  interface  java.util.Enumeration
{

    public abstract boolean hasMoreElements();
    public abstract Object nextElement();
}
```


Java: itérateurs sur une Hashtable

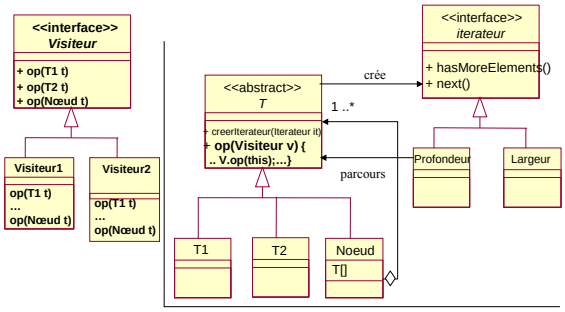
```
public  class  java.util.Hashtable
                                extends  java.util.Dictionary
                                implements java.lang.Cloneable  {
    public Hashtable();

    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();
}
```

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 ?

Pattern Visitor

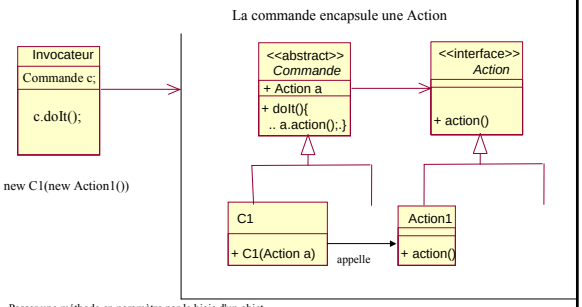


Si l'on doit parcourir une structure pour différents calcul sur les nœuds , plutôt que de polluer la structure de donnée, mieux vaut créer les opérations dans un visiteur.

Call back

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

Pattern Command



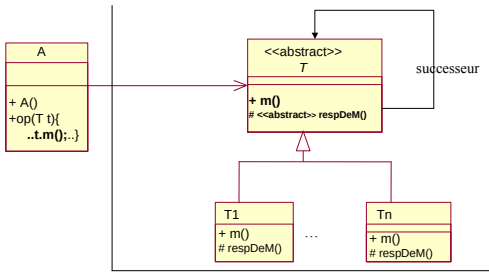
Passer une méthode en paramètre par le biais d'un objet

Responsabilité d'un traitement

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

Pattern Chaîne de responsabilité

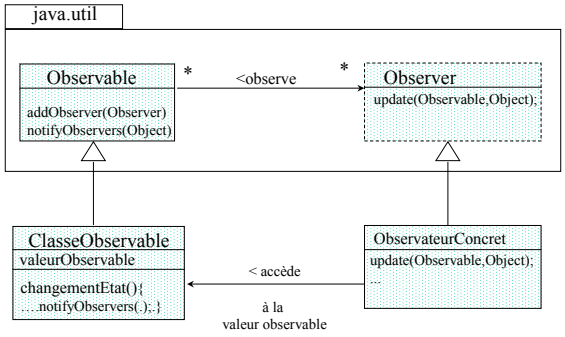
L'appel de m est confié à une liste d'objets. L'un de ceux-ci se reconnaît compétent et traite m



Exemple

```
abstract class T  {

        private protected t successor;

        public void m() {
                if(responsabilitéDeM()) return;
                        else if( successor != null)
                        successor.m();
        }

        abstract protected boolean responsabilitéDeM();
}
```

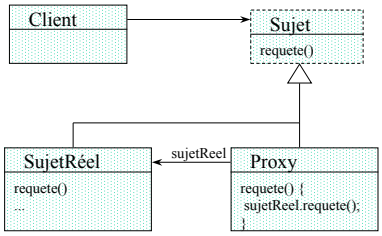

Pattern Observer



Subrogation

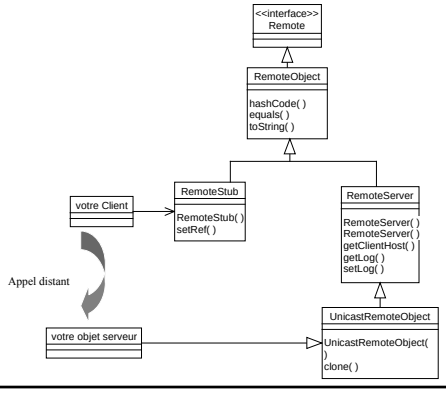
- Problème
  - Ne pas fournir l'objet demandé mais un représentant
    - Soit par soucis de sécurité
    - Soit pour des raisons de durée d'attente
    - Soit pour implémenter un protocole d'accès distant masqué
- Solution
  - Le proxy !

Proxy (ou Procuration)



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

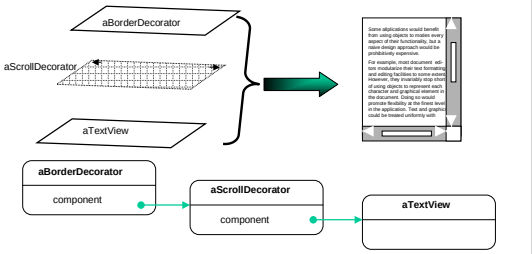
Proxy Rmi



Décorer un objet !

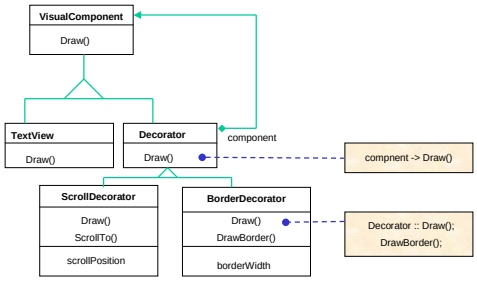
- Problème:
  - Etendre les fonctionnalités d'un objet sans sous-classer
- Solution
  - Emballer l'objet dans un autre qui ajoutera cette fonctionnalité!

Exemple



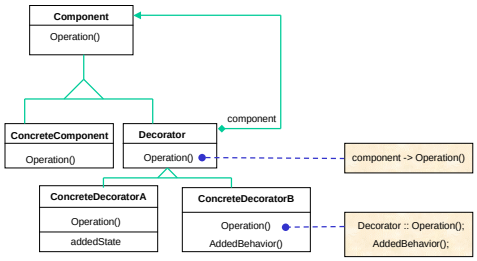
-45-

DECORATOR - Motivation



-46-

DECORATOR – Structure



-47-