

# **PLD Agile**

## **Partie 2 : Principes et patrons de conception orientée objet**

Christine Solnon

INSA de Lyon - 4IF

2016 / 2017

# Plan de la partie 2

## Principes et patrons de conception orientée objet

- 1 Introduction
- 2 Illustration de design patterns avec PlaCo
- 3 Encore quelques patrons du GoF

# Quelques principes de conception orientée objet

**Protection des variations :** Identifier les points de variation et d'évolution, et séparer ces aspects de ceux qui demeurent constants

**Faible couplage :** Réduire l'impact des modifications en minimisant les dépendances entre classes

**Forte cohésion :** Faciliter la compréhension, gestion et réutilisation des objets en concevant des classes à but unique

**Indirection :** Limiter le couplage et protéger des variations en ajoutant des objets intermédiaires

**Programmer pour des interfaces :** Limiter le couplage et protéger des variations en faisant abstraction de l'implémentation

**Composer au lieu d'hériter :** Utiliser la composition au lieu de l'héritage pour déléguer une tâche à un objet, changer dynamiquement le comportement d'une instance, ...

**Ces principes se retrouvent dans beaucoup de Design Patterns...**

# Patrons de conception (Design patterns)

## Qu'est-ce qu'un patron de conception ?

- Solution générale et réutilisable d'un problème récurrent  
  ~> Formalisation de bonnes pratiques

## Comment décrire un patron de conception

- Nom ~> Vocabulaire de conception
- Problème : Description du sujet à traiter et de son contexte
- Solution : Description des éléments, de leurs relations/coopérations et de leurs rôles dans la résolution du problème
  - Description générique
  - Illustration sur un exemple
- Conséquences : Effets résultant de la mise en œuvre du patron  
  ~> Complexité en temps/mémoire, impact sur la flexibilité, portabilité, ...

# 23 patrons du Gang of Four (GoF)

[E. Gamma, R. Helm, R. Johnson, J. Vlissides]

## Patrons illustrés avec PlaCo :

- Création : *Factory, Singleton*
- Comportement : *Iterateur, Etat, Observateur, Commande, Visiteur*
- Structure : *PoidsPlume*

## Patrons introduits en fin de cours :

- Création : *Abstract factory*
- Comportement : *Stratégie*
- Structure : *Décorateur, Adaptateur, Facade, Composite*

## Patron introduit pour le projet :

- Comportement : *Template*

## Patrons qui ne seront pas vus dans ce cours :

- Création : *Prototype, Builder*
- Comportement : *Chaine de resp., Interpreteur, Mediateur, Memento*
- Structure : *Pont, Proxy*

# Plan de la partie 2

## Principes et patrons de conception orientée objet

- 1 Introduction
- 2 Illustration de design patterns avec PlaCo
- 3 Encore quelques patrons du GoF

# L'application PlaCo (rappel)

Une scierie, équipée d'une machine de découpe de planches au laser, veut un système pour dessiner les plans à transmettre à la machine.

- Chaque plan a une hauteur et une largeur.
- Les formes à positionner sur le plan sont des rectangles et des cercles :
  - un rectangle est défini par les coordonnées de son coin supérieur gauche, sa largeur et sa hauteur ;
  - un cercle est défini par son rayon et les coordonnées de son centre.

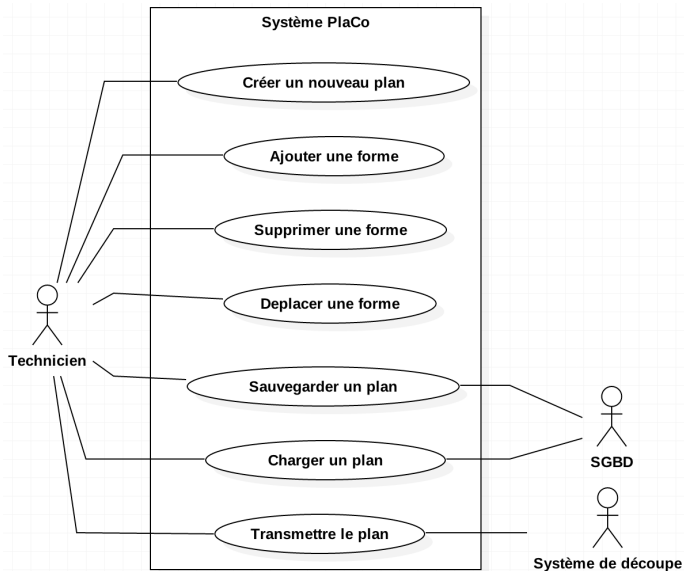
Les coordonnées et les longueurs sont des valeurs entières exprimées dans une unité donnée. Les formes ne doivent pas se superposer.

- L'application doit permettre d'ajouter, supprimer et déplacer des formes sur un plan, de sauvegarder et charger des plans, et de transmettre un plan au système de découpe.

**Pour télécharger le code Java de PlaCo :**

`liris.cnrs.fr/csolnon/PlaCo.jar`

# Diagramme de cas d'utilisation de PlaCo (rappel)





# Polymorphisme

## Problème :

Si les affaires marchent bien, la scierie envisage d'étendre le système pour découper des triangles, des ellipses, ...

## Solution : Utiliser le polymorphisme pour se protéger des variations

- Créer une interface Forme déclarant les méthodes communes à toutes les formes
- Créer deux classes Cercle et Rectangle réalisant Forme
- Utiliser le polymorphisme pour traiter de façon uniforme les instances de Cercle et Rectangle

## Principes mis en œuvre :

- Programmer pour des interfaces
- Protection des variations

```

public class Plan {
    private Collection<Forme> formes;
    public Plan(){
        formes = new ArrayList<Forme>();
    }
    public Forme cherche(Point p){
        Iterator<Forme> it = formes.iterator();
        while (it.hasNext()){
            Forme f = it.next();
            if (f.contient(p)) return f;
        }
        return null;
    }
}

```

```

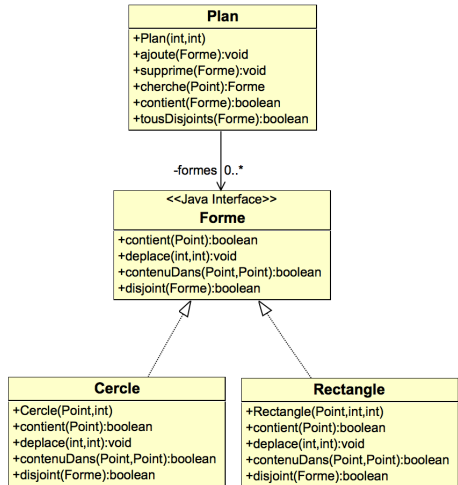
.....
public class Cercle implements Forme{
    private Point centre;
    private int rayon;
    @Override
    public boolean contient(Point p) {
        return centre.distance(p) <= rayon;
    }
}

```

```

.....
public class Rectangle implements Forme {
    private Point coin;
    private int largeur;
    private int hauteur;
    @Override
    public boolean contient(Point p) {
        return (p.getX() >= coin.getX()) && (p.getX() <= coin.getX() + largeur) && (p.getY() >= coin.get
    }
}

```



# Pattern GoF : Itérateur (1/3)

## Problème :

L'équipe de développement hésite sur le choix de la structure de données à utiliser pour mémoriser les formes du plan

## Solution : Utiliser le pattern Itérateur

Permet de parcourir la collection de formes sans connaître la structure de données utilisée

## Principes mis en œuvre :

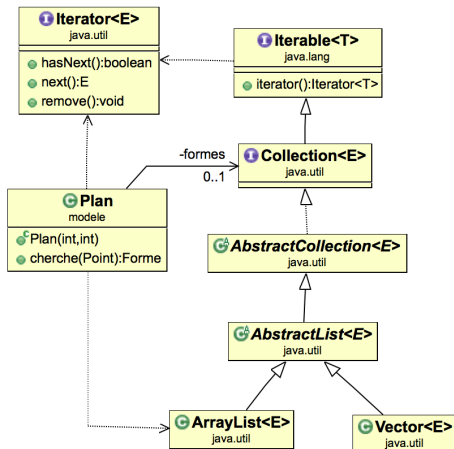
- Programmer pour des interfaces
- Protection des variations
- Forte cohésion

# Pattern GoF : Itérateur (2/3)

```
public class Plan {  
    private Collection<Forme> formes;  
    public Plan(){  
        formes = new ArrayList<Forme>();  
    }  
    public Forme cherche(Point p){  
        Iterator<Forme> it = formes.iterator();  
        while (it.hasNext()){  
            Forme f = it.next();  
            if (f.contient(p)) return f;  
        }  
        return null;  
    }  
}
```

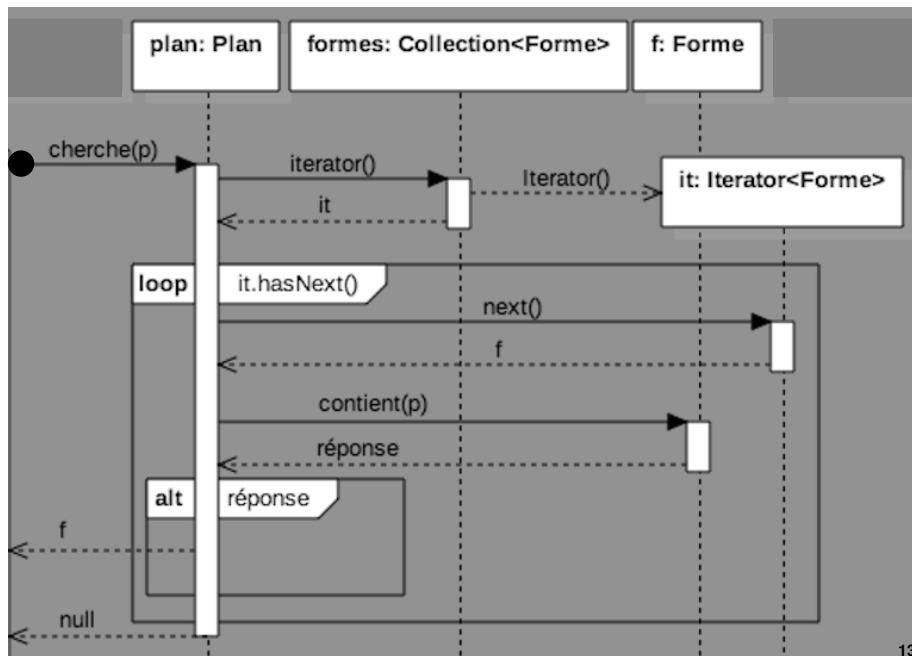
Syntaxe alternative (iter. implicite) :

```
public Forme cherche(Point p){  
    for (Forme f : formes){  
        if (f.contient(p)) return f;  
    }  
    return null;  
}
```



- Exercice : Dessiner le diagramme de séquence de `cherche(p)`
- Que faut-il changer si on veut utiliser `Vector` au lieu de `ArrayList` ?
- Pourquoi séparer `Iterator` de `Collection` ?

# Diagramme de séquence de `cherche(p)`



## Pattern GoF : Itérateur (3/3)

Séparer `Iterator` de `Collection` permet d'avoir plusieurs itérateurs sur une même collection en même temps :

```
public boolean disjoint(){
    Iterator<Forme> it1 = formes.iterator();
    while (it1.hasNext()){
        Forme f1 = it1.next();
        Iterator<Forme> it2 = formes.iterator();
        while (it2.hasNext()){
            Forme f2 = it2.next();
            if (f1 != f2 && !f1.disjoint(f2)) return false;
        }
    }
    return true;
}
```

# Architecture Modèle-Vue-Contrôleur (MVC)

## Problèmes :

- L'utilisateur peut demander de changer la façon d'interagir avec PlaCo :
  - Ajouter un menu avec liste déroulante pour sélectionner une forme à ajouter
  - Ajouter une description textuelle du plan, en plus de la visualisation graphique
  - Changer la façon de saisir les informations pour ajouter une nouvelle forme dans le plan
  - etc
- La technologie utilisée pour la visualisation peut changer
- La classe Plan perd en cohésion si elle doit s'occuper d'afficher le plan

## Solution :

Architecture MVC !

# Architecture Modèle-Vue-Contrôleur (MVC)

## Modèle : Met à jour et traite les données "métier"

- Ajouter/supprimer/déplacer des formes sur un plan

## Vue : Visualise le modèle et interagit avec l'utilisateur

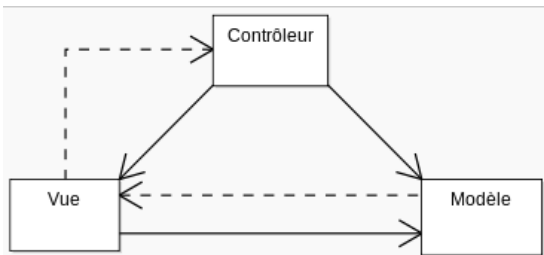
- Dessiner le plan à l'écran, afficher la liste des formes, etc
- Détecter les clics de souris, frappe de caractères, etc

## Contrôleur : Traduit les interactions entre l'utilisateur et la vue en actions pour le modèle ou la vue

- Signaler au modèle qu'il doit déplacer une forme quand l'utilisateur a frappé sur une flèche après avoir sélectionné une forme
- Signaler au modèle qu'il doit ajouter un rectangle au plan quand l'utilisateur a cliqué sur un point après avoir cliqué sur le bouton "Ajouter un rectangle"
- etc



# Architecture Modèle-Vue-Contrôleur (MVC)



- Flèche pleine = dépendance
- Pointillés = événements

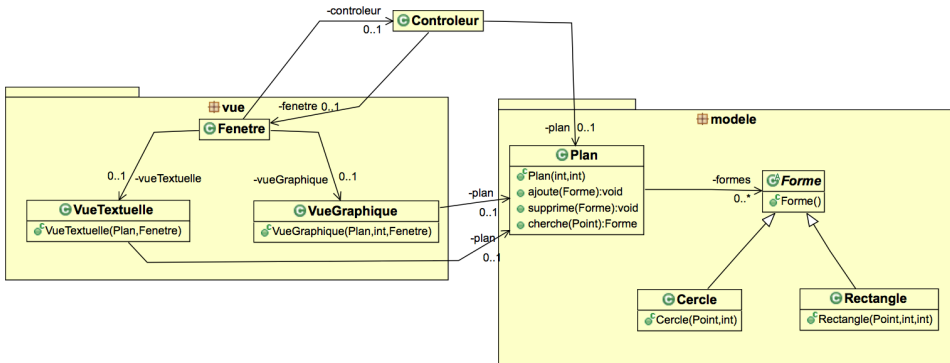
## Principes mis en œuvre :

- Protection des variations
- Forte cohésion

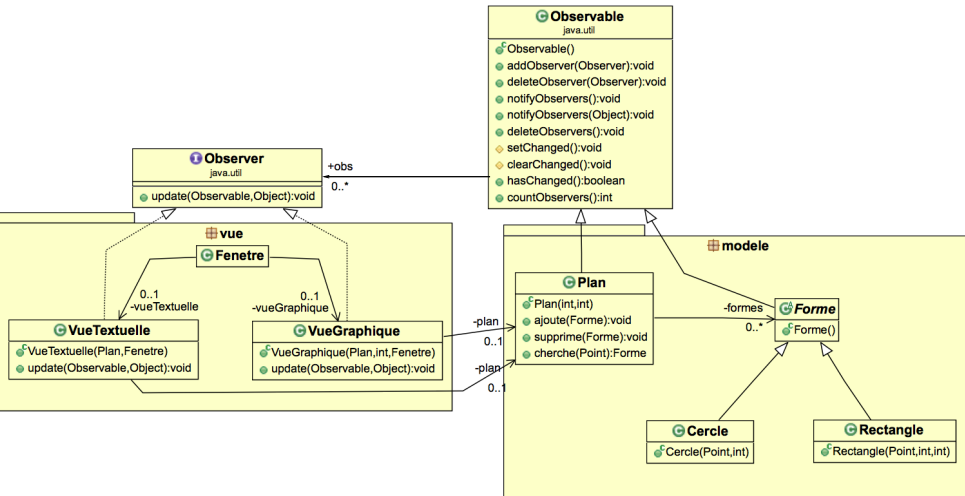
## Problème : Comment indiquer à Vue les modifications de Modèle ?

- Solution 1 : Modèle envoie un message à Vue quand il est modifié  
**Inconvénient : Modèle devient dépendant de Vue**
- Solution 2 : Contrôleur envoie un message à Vue qd Modèle est modifié
- Solution 3 : Utiliser le pattern Observateur

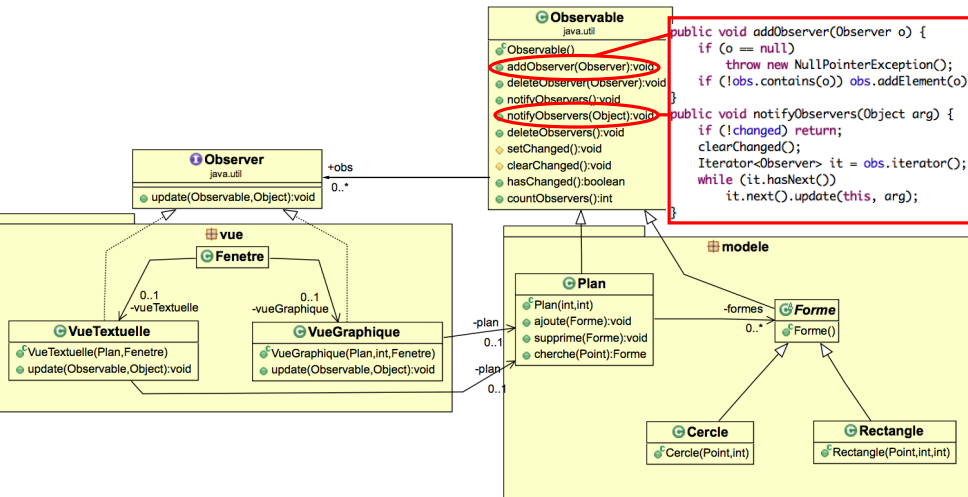
# Pattern GoF : Observateur (aka *Publish/Subscribe*) (1/2)



# Pattern GoF : Observateur (aka *Publish/Subscribe*) (1/2)

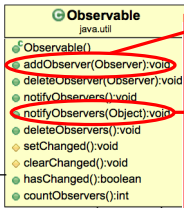


# Pattern GoF : Observateur (aka Publish/Subscribe) (1/2)

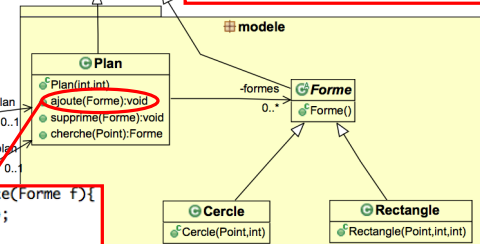
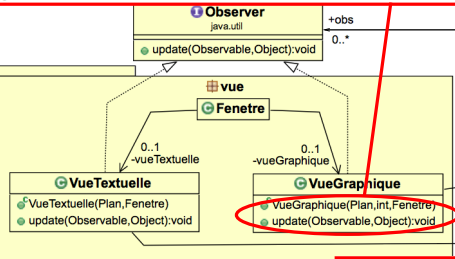


# Pattern GoF : Observateur (aka Publish/Subscribe) (1/2)

```
public VueGraphique(Plan plan, int echelle, Fenetre fenetre) {
    plan.addObserver(this); // this observe plan
    // ...
}
@Override
public void update(Observable o, Object arg) {
    // code pour afficher plan dans vueGraphique
}
```



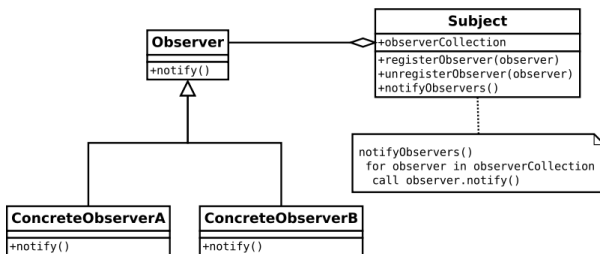
```
public void addObserver(Observer o) {
    if (o == null)
        throw new NullPointerException();
    if (!obs.contains(o)) obs.addElement(o);
}
public void notifyObservers(Object arg) {
    if (!changed) return;
    clearChanged();
    Iterator<Observer> it = obs.iterator();
    while (it.hasNext())
        it.next().update(this, arg);
}
```



```
public void ajoute(Forme f){
    formes.add(f);
    setChanged();
    notifyObservers(f);
}
```

## Pattern GoF : Observateur (*aka Publish/Subscribe*) (2/2)

### Solution générique [Wikipedia] :



### Principes mis en œuvre :

- Faible couplage entre `ConcreteObserver` et `Subject`
- Protection des variations : Ajout d'observateurs sans modifier `Subject`

### Remarque :

- Les données de `Subject` peuvent être "poussées" (dans `notify`) ou "tirées" (avec des `getters`)

# Pattern GoF : Visiteur (1/3)

## Problème :

Perte de la classe effective (Cercle ou Rectangle) qd on itère sur des Formes

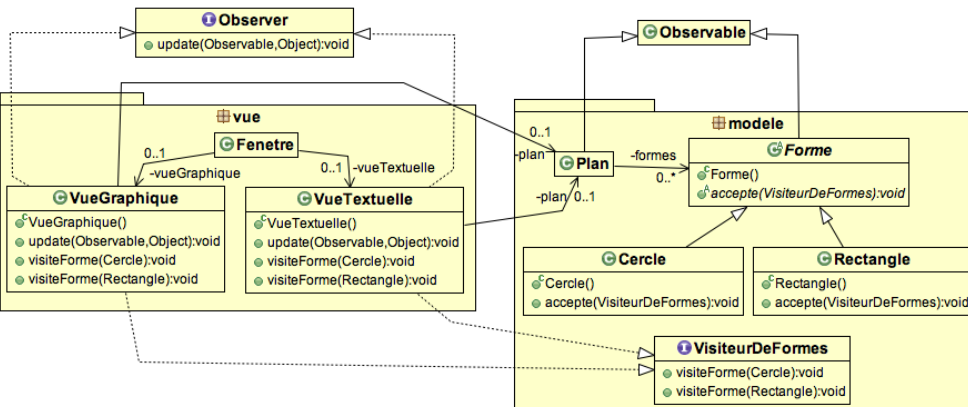
## Solution 1 : Tester la classe des instances avant de les traiter

```
public class VueGraphique implements Observer{
    private Plan plan;
    public void update(Observable o, Object arg) {
        Iterator<Forme> it = plan.getIterateurFormes();
        while (it.hasNext())
            dessine(it.next());
    }
    private void dessine(Forme f) {
        if (f instanceof Cercle) dessine((Cercle)f);
        else dessine((Rectangle)f);
    }
    public void dessine(Cercle c) { // code pour dessiner le cercle c
    }
    public void dessine(Rectangle r) { // code pour dessiner le rectangle r
    }
}
```

→ Peut devenir lourd si Forme a beaucoup de sous-classes

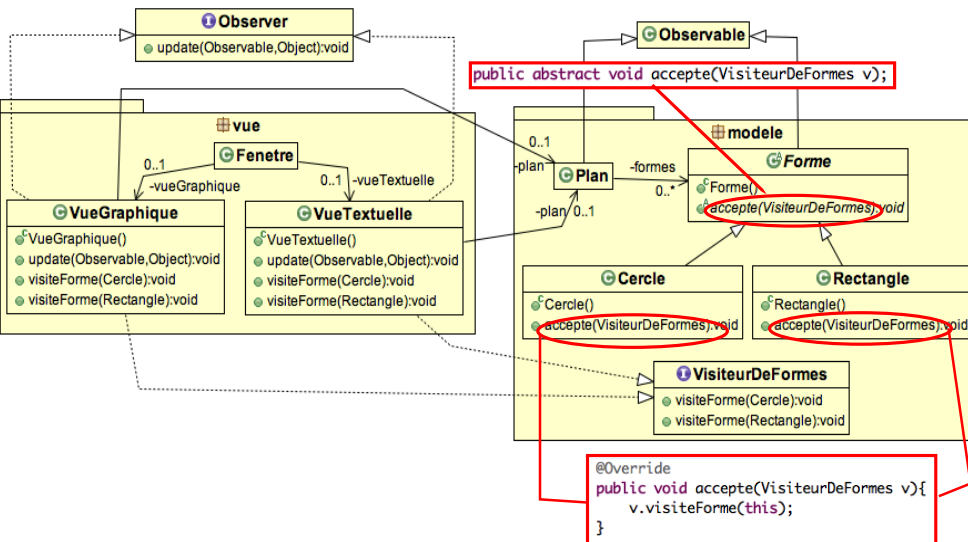
## Solution 2 : Utiliser le pattern Visiteur

# Pattern GoF : Visiteur (2/3)

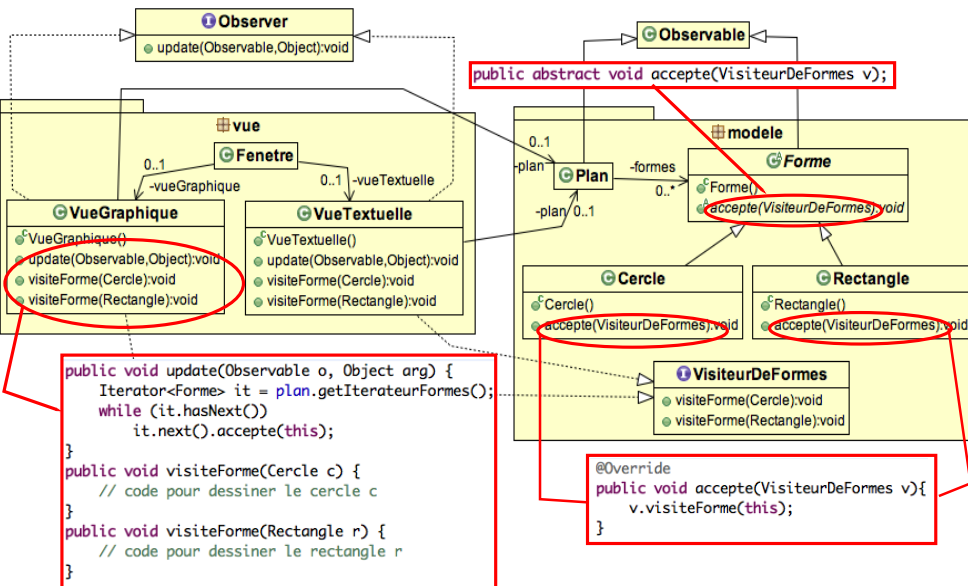




# Pattern GoF : Visiteur (2/3)

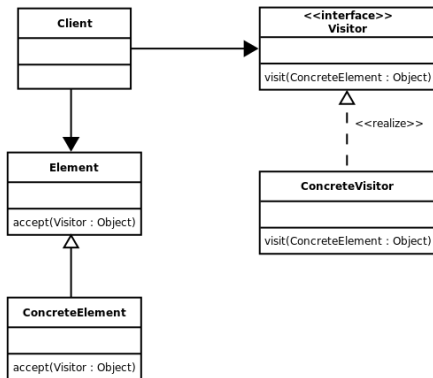


# Pattern GoF : Visiteur (2/3)



# Pattern GoF : Visiteur (3/3)

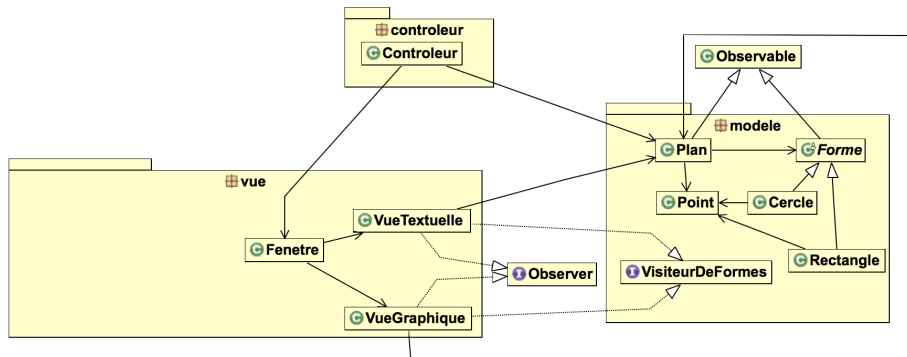
## Solution générique [Wikipedia] :



## Principes mis en œuvre :

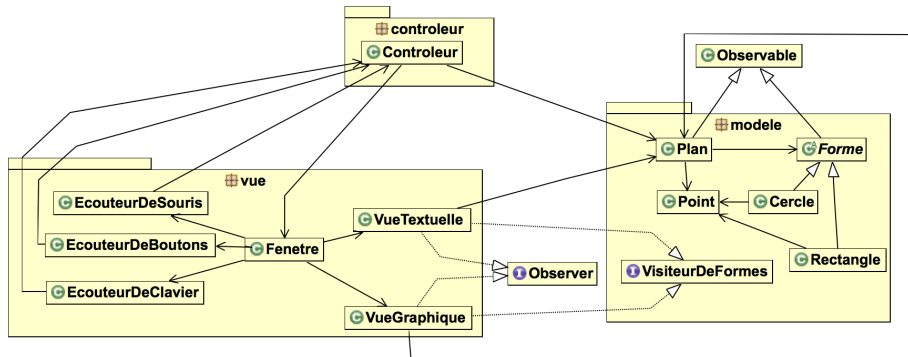
- Forte cohésion : Permet de regrouper dans chaque réalisation de Visitor toutes les méthodes liées à un aspect (visualisation, persistance, ...) de toutes les sous-classes de Element
- Protection des variations : Ajout de nouvelles réalisations de Visitor sans modifier ConcreteElement

# Architecture actuelle de PlaCo



Comment l'utilisateur interagit-il avec PlaCo ?

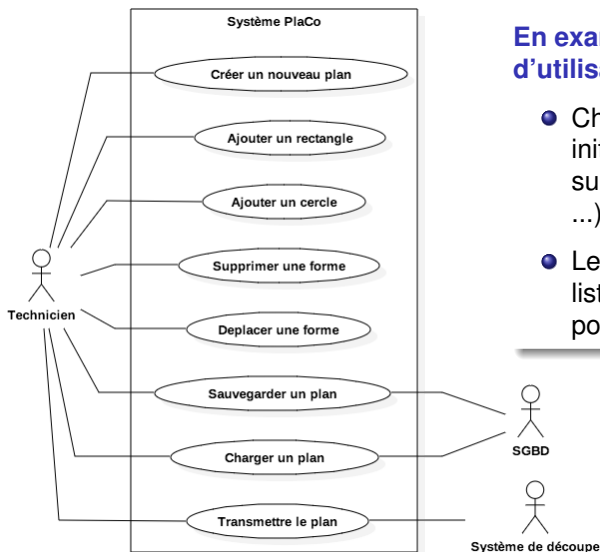
# Architecture actuelle de PlaCo



## Comment l'utilisateur interagit-il avec PlaCo ?

- Fenêtre utilise des écouteurs d'événements (Listeners)
- Les écouteurs transmettent les événements à Contrôleur

# Comment déterminer les événements à écouter ?



## En examinant les cas d'utilisation :

- Chaque cas d'utilisation est initié par un événement (clic sur bouton, sélection menu, ...)
- Les scénarios permettent de lister les autres événements possibles

# Extraction des événements à partir des scénarios

## Scénario principal de "Ajouter un rectangle" :

- ❶ L'utilisateur indique au système qu'il veut ajouter un rectangle
- ❷ Le système demande de saisir les coordonnées d'un angle du rectangle
- ❸ L'utilisateur saisit les coordonnées d'un point  $p_1$
- ❹ Le système demande de saisir les coordonnées de l'angle opposé
- ❺ L'utilisateur saisit les coordonnées d'un point  $p_2$
- ❻ Le système ajoute le rectangle correspondant dans le plan
- ❼ Le système affiche le plan avec le rectangle ajouté

Alternative [1-5a] : L'utilisateur indique qu'il souhaite annuler la saisie

## Événements utilisateurs :

- Clic sur le bouton "Ajouter un rectangle"
- Clic gauche de la souris sur la vue graphique du plan
- Clic droit de la souris ou [Esc]

# Extraction des événements à partir des scénarios

## Scénario principal de "Ajouter un rectangle" :

- 1 L'utilisateur indique au système qu'il veut ajouter un rectangle
- 2 Le système demande de saisir les coordonnées d'un angle du rectangle
- 3 L'utilisateur saisit les coordonnées d'un point  $p_1$
- 4 Le système demande de saisir les coordonnées de l'angle opposé
- 5 L'utilisateur saisit les coordonnées d'un point  $p_2$
- 6 Le système ajoute le rectangle correspondant dans le plan
- 7 Le système affiche le plan avec le rectangle ajouté

Alternative [1-5a] : L'utilisateur indique qu'il souhaite annuler la saisie

## Evénements utilisateurs :

- Clic sur le bouton "Ajouter un rectangle"
- Clic gauche de la souris sur la vue graphique du plan
- Clic droit de la souris ou [Esc]



# Extraction des événements à partir des scénarios

## Scénario principal de "Ajouter un rectangle" :

- 1 L'utilisateur indique au système qu'il veut ajouter un rectangle
- 2 Le système demande de saisir les coordonnées d'un angle du rectangle
- 3 L'utilisateur saisit les coordonnées d'un point  $p1$
- 4 Le système demande de saisir les coordonnées de l'angle opposé
- 5 L'utilisateur saisit les coordonnées d'un point  $p2$
- 6 Le système ajoute le rectangle correspondant dans le plan
- 7 Le système affiche le plan avec le rectangle ajouté

Alternative [1-5a] : L'utilisateur indique qu'il souhaite annuler la saisie

## Événements utilisateurs :

- Clic sur le bouton "Ajouter un rectangle"
- Clic gauche de la souris sur la vue graphique du plan
- Clic droit de la souris ou [Esc]

# Extraction des événements à partir des scénarios

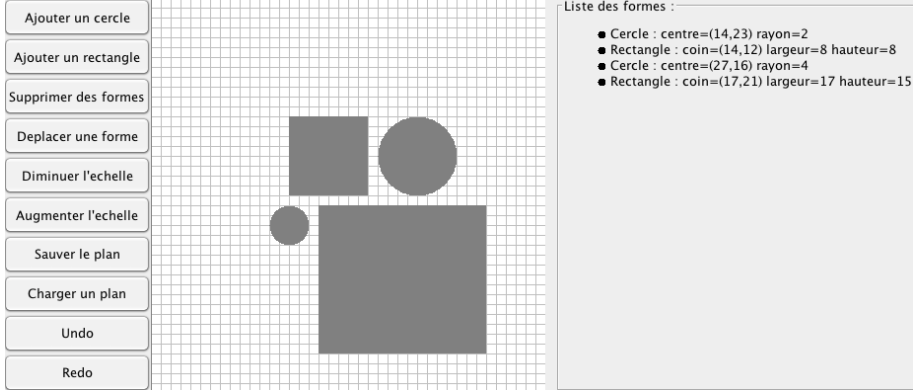
## Scénario principal de "Ajouter un rectangle" :

- ➊ L'utilisateur indique au système qu'il veut ajouter un rectangle
- ➋ Le système demande de saisir les coordonnées d'un angle du rectangle
- ➌ L'utilisateur saisit les coordonnées d'un point  $p_1$
- ➍ Le système demande de saisir les coordonnées de l'angle opposé
- ➎ L'utilisateur saisit les coordonnées d'un point  $p_2$
- ➏ Le système ajoute le rectangle correspondant dans le plan
- ➐ Le système affiche le plan avec le rectangle ajouté

Alternative [1-5a] : **L'utilisateur indique qu'il souhaite annuler la saisie**

## Événements utilisateurs :

- Clic sur le bouton "Ajouter un rectangle"
- Clic gauche de la souris sur la vue graphique du plan
- Clic droit de la souris ou [Esc]

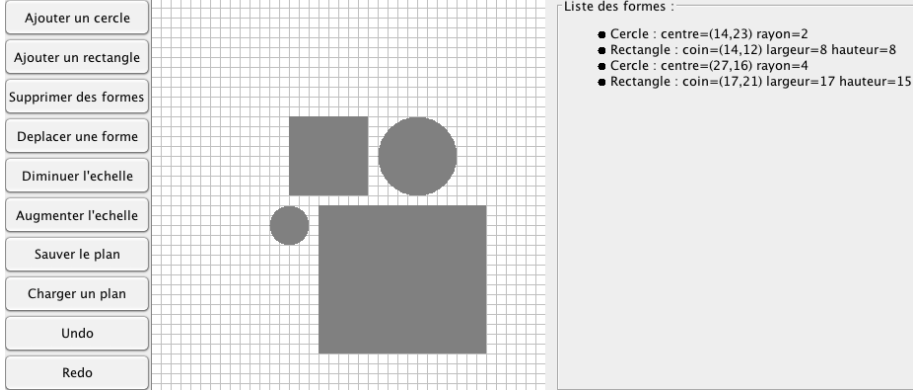


## Liste des événements utilisateur de PlaCo :

- Clic sur un bouton : AjouterCercle, AjouterRectangle, . . . , Undo, Redo
- Frappe d'une touche au clavier : flèches, [Ctr Z], [Shift Ctr Z], [Esc]
- Clic gauche de la souris sur la vue graphique
- Clic droit de la souris sur la vue graphique
- Déplacement de la souris sur la vue graphique

**Vous pourrez critiquer cette IHM dans le cours d'IHM...**

Nous voyons ici comment concevoir PlaCo pour pouvoir facilement changer l'IHM



## Liste des événements utilisateur de PlaCo :

- Clic sur un bouton : AjouterCercle, AjouterRectangle, ... , Undo, Redo
- Frappe d'une touche au clavier : flèches, [Ctr Z], [Shift Ctr Z], [Esc]
- Clic gauche de la souris sur la vue graphique
- Clic droit de la souris sur la vue graphique
- Déplacement de la souris sur la vue graphique

## Vous pourrez critiquer cette IHM dans le cours d'IHM...

Nous voyons ici comment concevoir PlaCo pour pouvoir facilement changer l'IHM

# Que font les écouteurs ?

```
public class EcouteurDeBoutons
implements ActionListener {
    private Controleur controleur;
    public EcouteurDeBoutons(Controleur c){
        this.controleur = c;
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        switch (e.getActionCommand()){
            case "Ajouter un cercle":
                controleur.ajouterCercle();
                break;
            case "Ajouter un rectangle":
                controleur.ajouterRectangle();
                break;
            case "Supprimer des formes":
                controleur.supprimer();
                break;
            // etc
        }
    }
}
```

Ils délèguent à Controleur !

```
public class EcouteurDeSouris extends MouseAdapter {
    private Controleur controleur;
    private VueGraphique v;
    private Fenetre f;
    public EcouteurDeSouris(Controleur c, VueGraphique v, Fenetre f) {
        this.controleur = c;
        this.v = v;
        this.f = f;
    }
    @Override
    public void mouseClicked(MouseEvent evt) {
        switch (evt.getButton()){
            case MouseEvent.BUTTON1:
                controleur.clicGauche(coord(evt)); break;
            case MouseEvent.BUTTON3:
                controleur.clicDroit(); break;
            default: break;
        }
    }
    @Override
    public void mouseMoved(MouseEvent evt) {
        controleur.clicGauche(coord(evt));
    }
    private Point coord(MouseEvent evt){
        MouseEvent e = SwingUtilities.convertMouseEvent(f, evt, v);
        int x = Math.round((float)e.getX()/(float)v.getEchelle());
        int y = Math.round((float)e.getY()/(float)v.getEchelle());
        return new Point(x, y);
    }
}
```

# Que fait Contrôleur ?

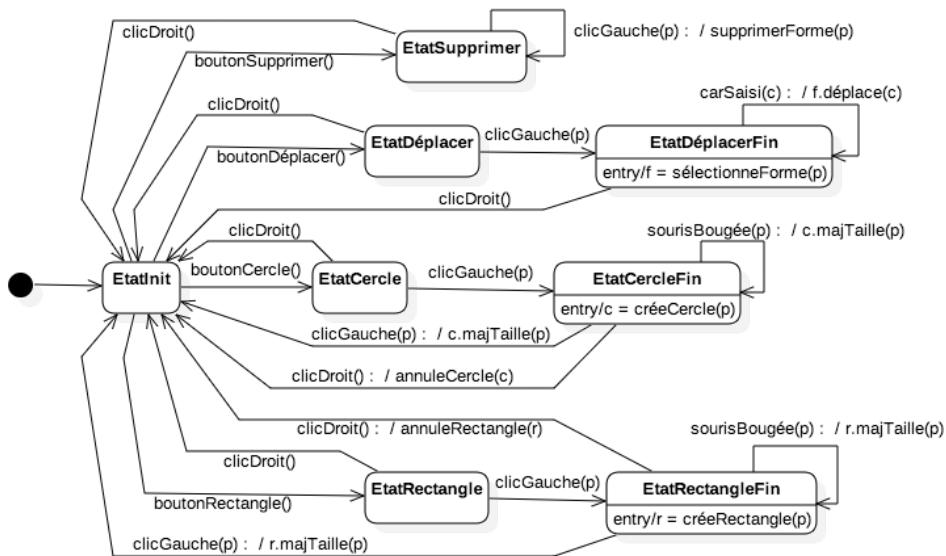
Contrôleur a une méthode pour chaque événement utilisateur :

Contrôleur
+Contrôleur(Plan,int) +ajouterCercle():void +ajouterRectangle():void +supprimer():void +deplacer():void +undo():void +redo():void +clicGauche(Point):void +clicDroit():void +sourisBougee(Point):void +caractereSaisi(int):void

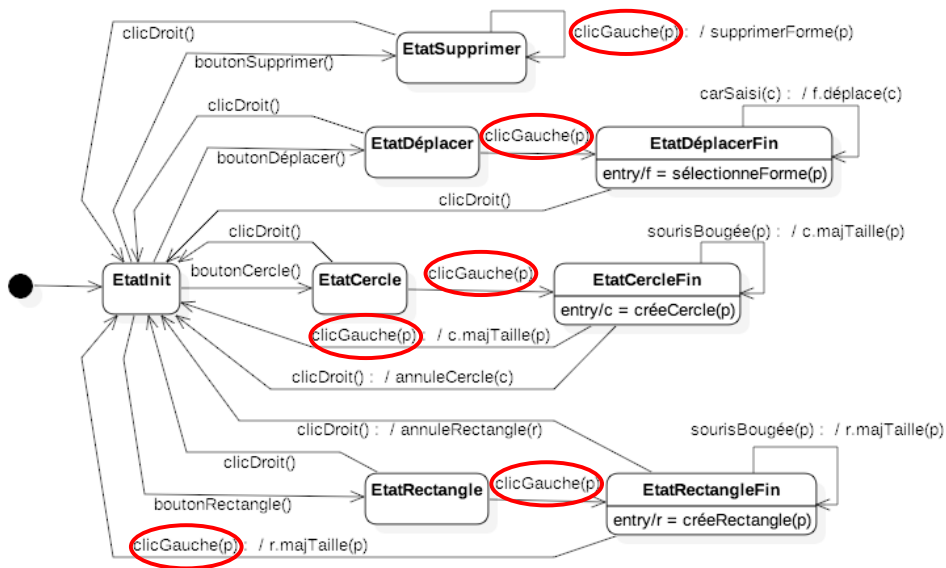
Que font les méthodes de Contrôleur ?

- Ca dépend du cas d'utilisation  
    ~> Dessiner un diagramme Etats-Transitions

# Diagramme Etats-Transitions de PlaCo



# Diagramme Etats-Transitions de PlaCo



Que fait la méthode `clicGauche(p)` de Contrôleur ?



# Pattern GoF : Etat (*State*) (1/5)

## Problème :

Ce que doit faire `Contrôleur` à la réception du message `clicGauche(p)` dépend de son état courant

## Solution 1 :

- `Contrôleur` a un attribut `etatCourant` mémorisant son état
- `clicGauche(p)` contient un cas par état possible

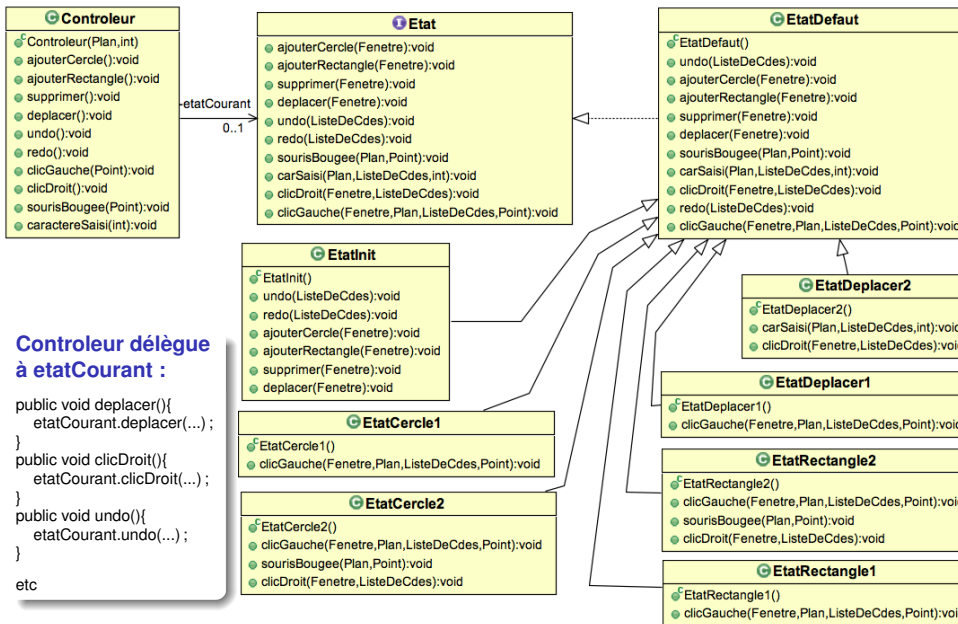
## Avantages et inconvénients ?

```
public void clicGauche(Point p) {  
    switch (etatCourant){  
        case ETAT_CERCLE:  
            formeCourante = new Cercle(p,1);  
            plan.ajoute(formeCourante);  
            etatCourant = ETAT_CERCLE_FIN;  
            break;  
        case ETAT_RECTANGLE:  
            formeCourante = new Rectangle(p,1,1);  
            plan.ajoute(formeCourante);  
            etatCourant = ETAT_RECTANGLE_FIN;  
            break;  
        case ETAT_DEPLACER:  
            formeCourante = plan.cherche(p);  
            if (formeCourante != null)  
                etatCourant = ETAT_DEPLACER_FIN;  
            break;  
        // etc  
    }  
}
```

## Solution 2 : Utiliser le pattern Etat

Encapsuler les états dans des classes implémentant une même interface

# Pattern GoF : Etat (State) (2/5)



# Pattern GoF : Etat (*State*) (3/5)

## Comment Controleur change-t-il d'état ?

Méthode `setEtatCourant(Etat e)` de Controleur

## Comment récupérer les instances d'Etat ?

- Solution 1 : Créer une nouvelle instance à chaque changement d'état
- Solution 2 : Les classes Etat sont des singletons (cf fin du cours)
- Solution 3 : Contrôleur possède une instance de chaque classe Etat

```
public class Controleur{  
    protected final EtatInit etatInit = new EtatInit();  
    protected final EtatDeplacer1 etatDeplacer1 = new EtatDeplacer1();  
    protected final EtatDeplacer2 etatDeplacer2 = new EtatDeplacer2();  
    protected final EtatRectangle1 etatRectangle1 = new EtatRectangle1();  
    protected final EtatRectangle2 etatRectangle2 = new EtatRectangle2();  
    protected final EtatCercle1 etatCercle1 = new EtatCercle1();  
    protected final EtatCercle2 etatCercle2 = new EtatCercle2();  
    protected final EtatSupprimer etatSupprimer = new EtatSupprimer();  
  
    protected void setEtatCourant(Etat etat){etatCourant = etat;}
```

# Pattern GoF : Etat (*State*) (4/5)

## Code de la méthode `clicGauche` :

- Dans la classe `Contrôleur`

```
public void clicGauche(Point p) {  
    etatCourant.clicGauche(this, fenetre, plan, listeDeCdes, p);  
}
```

- Dans la classe `EtatDefaut`

```
public void clicGauche(Contrôleur controleur, Fenetre fenetre, Plan plan, ListeDeCdes listeDeCdes, Point p) {}
```

- Dans la classe `EtatCercle1`

```
public void clicGauche(Contrôleur controleur, Fenetre fenetre, Plan plan, ListeDeCdes listeDeCdes, Point p) {  
    controleur.etatCercle2.actionEntree(p, plan, listeDeCdes);  
    controleur.setEtatCourant(controleur.etatCercle2);  
}
```

- Dans la classe `EtatCercle2`

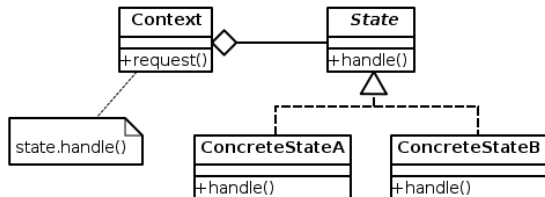
```
public void clicGauche(Contrôleur controleur, Fenetre fenetre, Plan plan, ListeDeCdes listeDeCdes, Point p) {  
    cercle.miseAJourTaille(p, plan);  
    controleur.setEtatCourant(controleur.etatInit);  
}  
protected void actionEntree(Point p, Plan plan, ListeDeCdes listeDeCdes) {  
    cercle = new Cercle(p, 1);  
}
```

- etc.

# Pattern GoF : State (5/5)

## Solution générique :

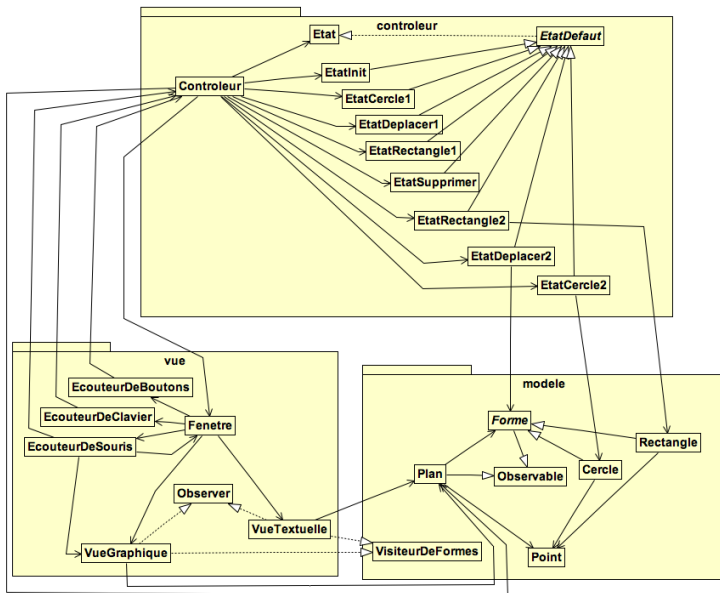
[Wikipedia]



## Principes mis en œuvre :

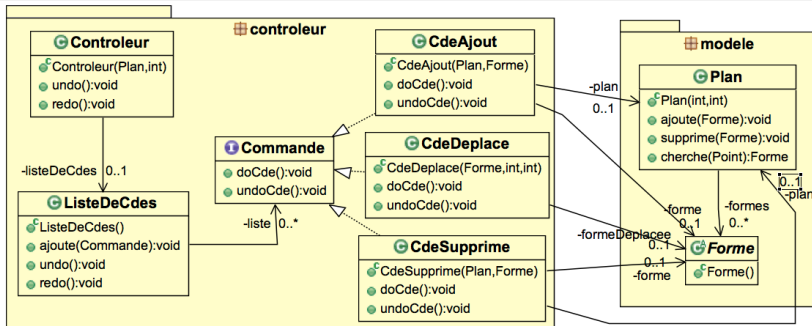
- Forte cohésion : Chaque **ConcreteState** contient les méthodes des événements qui ont un effet sur lui
- Protection des variations : Ajout d'un nouvel état facile (mais ajout d'un nouvel événement plus fastidieux)
- Programmer pour des interfaces

# Architecture actuelle de PlaCo)



**Problème : Comment mettre en œuvre des undo/redo ?**

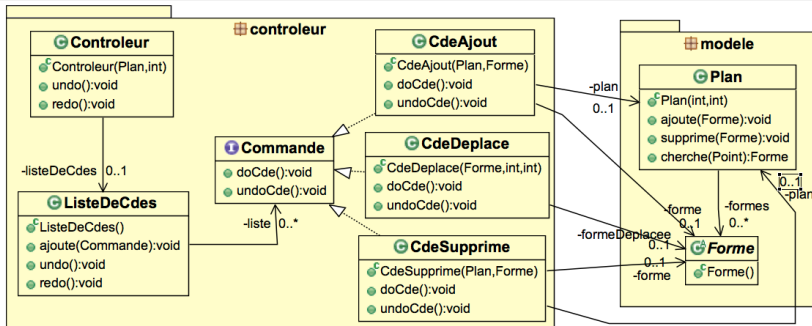
# Pattern GoF : Commande (1/2)



## Cde Ajout :

```
public class CdeAjout implements Commande {  
    private Plan plan;  
    private Forme forme;  
    public CdeAjout(Plan p, Forme f){this.plan = p; this.forme = f;}  
    public void doCde() {plan.ajoute(forme);}   
    public void undoCde() {plan.supprime(forme);}   
}
```

# Pattern GoF : Commande (1/2)

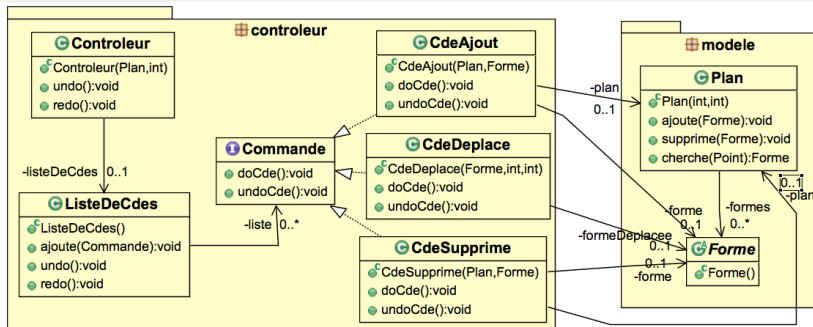


## Ajout d'un nouveau cercle en entrant dans EtatCercle2 :

```
public class EtatCercle2 extends EtatDefault {  
    private Cercle cercle;  
    protected void actionEntree(Point p, Plan plan, ListeDeCdes listeDeCdes) {  
        cercle = new Cercle(p, 1);  
        listeDeCdes.ajoute(new CdeAjout(plan, cercle));  
    }  
}
```



# Pattern GoF : Commande (1/2)



## Undo/Redo :

```

public class EtatInit extends EtatDefaut {
    public void undoCde(ListeDeCdes l){
        l.undo();
    }
    public void redoCde(ListeDeCdes l){
        l.redo();
    }
}

```

```

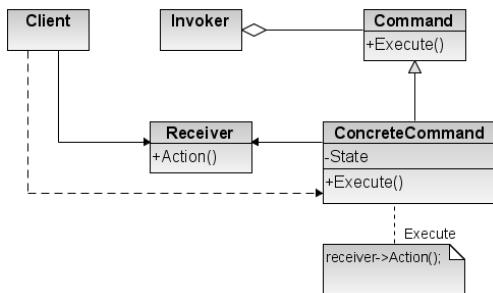
public class ListeDeCdes {
    private LinkedList<Commande> liste;
    private int i;
    public void undo(){
        if (i >= 0)
            liste.get(i--).undoCde();
    }
    public void redo(){
        if (i < liste.size()-1)
            liste.get(++i).doCde();
    }
}

```

# Pattern GoF : Commande (2/2)

## Solution générique :

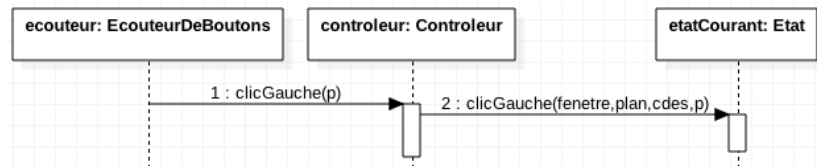
- Client crée les instances de ConcreteCommand
- Invoker demande l'exécution des commandes
- ConcreteCommand délègue l'exécution à Receiver



## Remarques :

- Découple la réception d'une requête de son exécution
- Les rôles de **Client** et **Invoker** peuvent être joués par une même classe (par exemple le contrôleur)
- Permet la journalisation des requêtes pour reprise sur incident
- Permet d'annuler ou ré-exécuter des requêtes (undo/redo)

# Diagrammes de séquence (1/2)



## Vérification de la cohérence avec le diagramme de classes :

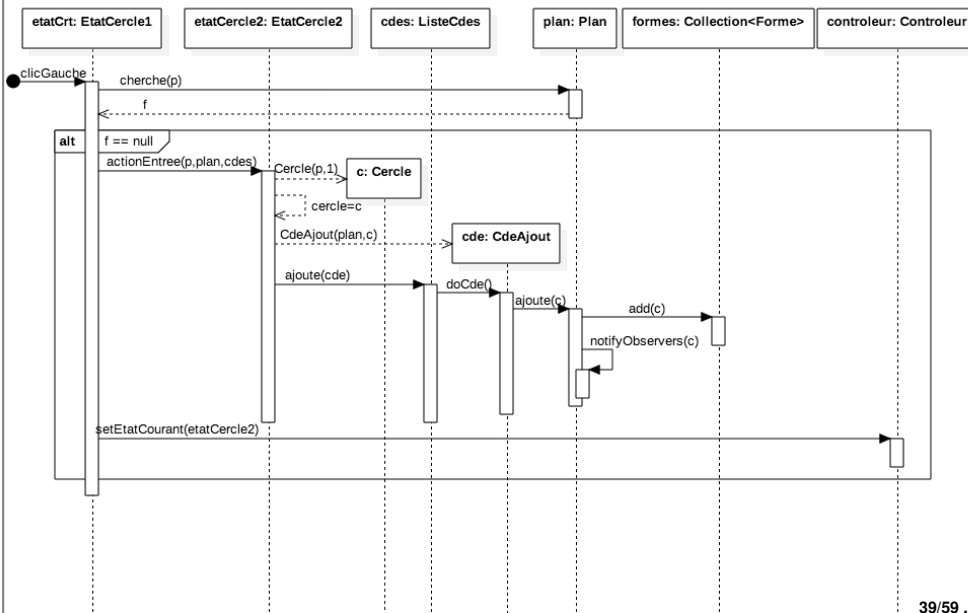
- **EcouteurDeBoutons** doit avoir une visibilité sur **controleur**  
~> **EcouteurDeBoutons** a un attribut **controleur**
- **Controleur** doit avoir une méthode **clicGauche (p)**
- **Controleur** doit avoir une visibilité sur **etatCourant**  
~> **Controleur** a un attribut **etatCourant**
- **Etat** doit avoir une méthode **clicGauche (fenetre,plan,cdes,p)**

## Remarque :

**clicGauche (fenetre,plan,cdes,p)** est un message polymorphe  
~> Faire un diagramme de séquence pour chaque classe réalisant **Etat**

# Diagrammes de séquence (2/2)

interaction Diag. de seq.: clicGauche(fenetre,plan,cdes,p)



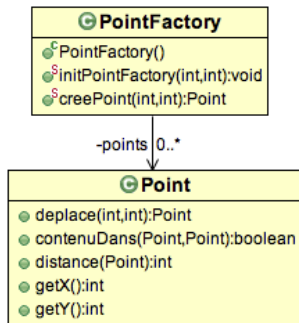
# Patterns GoF : Poids Plume (*FlyWeight*) et Factory

## Problème :

Nombreuses créations/destructions d'instances de Point

## Solution :

- Partager la même instance pour les points de mêmes coordonnées  
~> Attention : changer d'instance pour déplacer un point !
- Utiliser une Factory pour créer/mémoriser les instances



```
public class PointFactory {
    private static Point points[];
    private static int largeur;
    private static int hauteur;
    public static void initPointFactory(int l, int h){
        largeur = l; hauteur = h; points = new Point[l][h];
    }
    public static Point creePoint(int x, int y){
        if (x>=largeur || x<0 || y>=hauteur || y<0)
            return null;
        if (points[x][y] == null)
            points[x][y] = new Point(x,y);
        return points[x][y];
    }
}
```

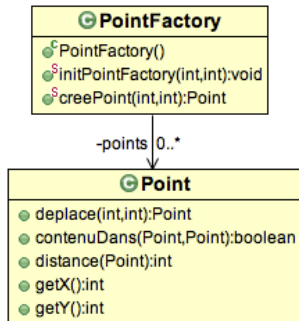
# Patterns GoF : Poids Plume (*FlyWeight*) et Factory

## Problème :

Nombreuses créations/destructions d'instances de Point

## Solution :

- Partager la même instance pour les points de mêmes coordonnées  
~> Attention : changer d'instance pour déplacer un point !
- Utiliser une Factory pour créer/mémoriser les instances



```
public class Point {
    private int x;
    private int y;
    protected Point(int x, int y){
        this.x = x; this.y = y;
    }
    public Point deplace(int deltaX, int deltaY) {
        return PointFactory.creePoint(x+deltaX, y+deltaY);
    }
    public boolean contenuDans(Point p1, Point p2) {...}
    public int distance(Point p) {...}
    public int getX() {return x;}
    public int getY() {return y;}
}
```

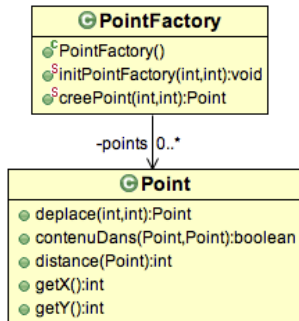
# Patterns GoF : Poids Plume (*FlyWeight*) et Factory

## Problème :

Nombreuses créations/destructions d'instances de Point

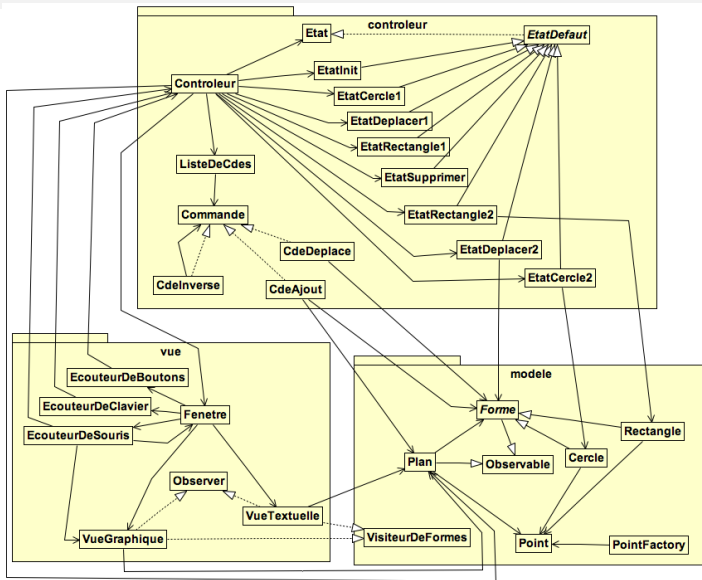
## Solution :

- Partager la même instance pour les points de mêmes coordonnées  
~> Attention : changer d'instance pour déplacer un point !
- Utiliser une Factory pour créer/mémoriser les instances



```
public class EcouteurDeSouris extends MouseAdapter {
    public void mouseMoved(MouseEvent evt) {
        Point p = coordonnees(evt);
        if (p != null)
            controleur.sourisBougee(p);
    }
    private Point coordonnees(MouseEvent evt){
        int x = // code pour calculer x à partir de evt
        int y = // code pour calculer y à partir de evt
        return PointFactory.creePoint(x, y);
    }
}
```

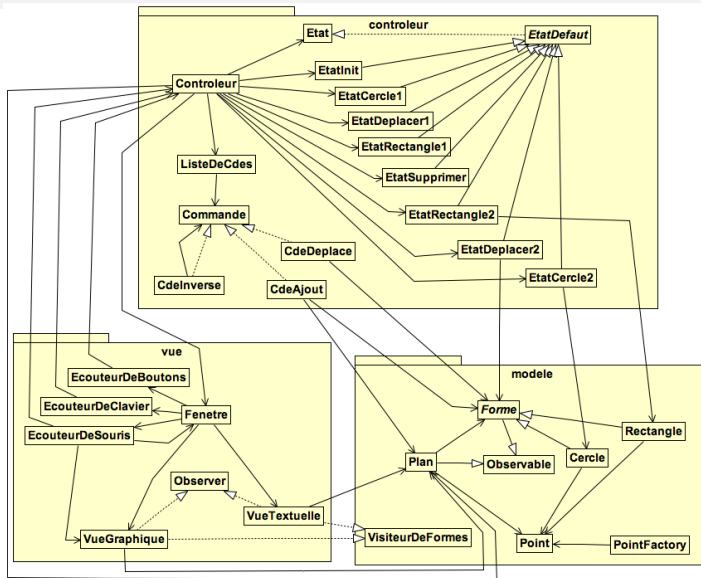
# Architecture actuelle de PlaCo



Il manque encore une chose ?



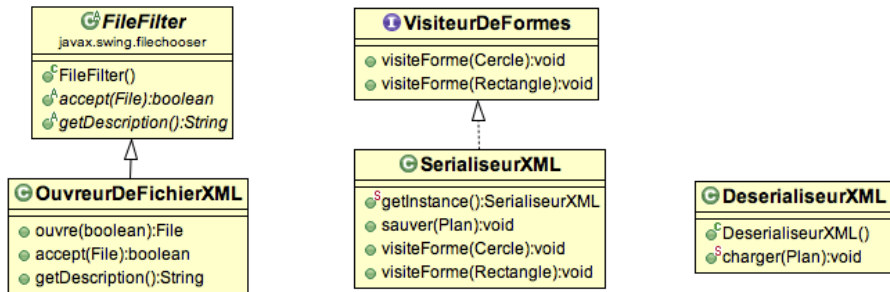
# Architecture actuelle de PlaCo



Il manque encore une chose ?

→ Charger/sauver un plan à partir d'un fichier XML

# Diagramme de classes du package xml



**Problème : Comment accéder à `OuvreurDeFichierXML.ouvre()` depuis n'importe quelle classe du package ?**

- Rendre `ouvre()` statique ?

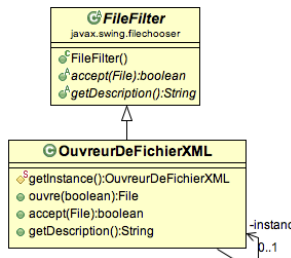
Impossible si `OuvreurDeFichierXML` est son propre `FileFilter`

```
public class OuvreurDeFichierXML extends FileFilter {
    public File ouvre(boolean lecture) throws ExceptionXML{
        JFileChooser jFileChooserXML = new JFileChooser();
        jFileChooserXML.setFileFilter(this);
    }
}
```

- Utiliser un Singleton

# Patron GoF : Singleton

```
public class OuvreurDeFichierXML extends FileFilter {  
    private static OuvreurDeFichierXML instance = null;  
    private OuvreurDeFichierXML(){}  
    protected static OuvreurDeFichierXML getInstance(){  
        if (instance == null) instance = new OuvreurDeFichierXML();  
        return instance;  
    }  
    public File ouvre(boolean lecture) throws ExceptionXML{  
    }  
    public boolean accept(File f) {  
    }  
    public String getDescription() {  
    }  
    private String getExtension(File f) {  
    }  
}
```



OuvreurDeFichierXML ne peut avoir qu'une seule instance et cette instance est accessible à toutes les classes du package

→ `OuvreurDeFichierXML.getInstance()`

## Attention :

Parfois considéré comme un anti-pattern... à utiliser avec modération !

# Plan de la partie 2

## Principes et patrons de conception orientée objet

- 1 Introduction
- 2 Illustration de design patterns avec PlaCo
- 3 Encore quelques patrons du GoF

# 23 patrons du Gang of Four (GoF)

[E. Gamma, R. Helm, R. Johnson, J. Vlissides]

## Patrons qu'on vient d'illustrer avec PlaCo :

- Création : *Factory, Singleton*
- Comportement : *Iterateur, Etat, Observateur, Commande, Visiteur*
- Structure : *PoidsPlume*

## Patrons qu'on va voir maintenant :

- Création : *Abstract factory*
- Comportement : *Stratégie*
- Structure : *Décorateur, Adaptateur, Facade, Composite*

## Patron introduit pour le projet :

- Comportement : *Template*

## Patrons qui ne seront pas vus dans ce cours :

- Création : *Prototype, Builder*
- Comportement : *Chaine de resp., Interpreteur, Mediateur, Memento*
- Structure : *Pont, Proxy*

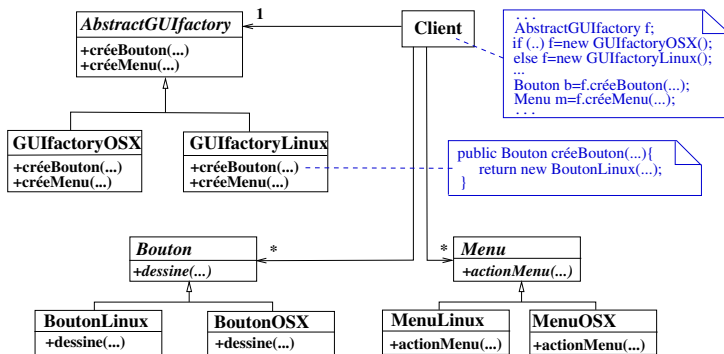
# Patron GoF : Abstract factory (1/2)

## Problème :

Créer une famille d'objets sans spécifier leurs classes concrètes

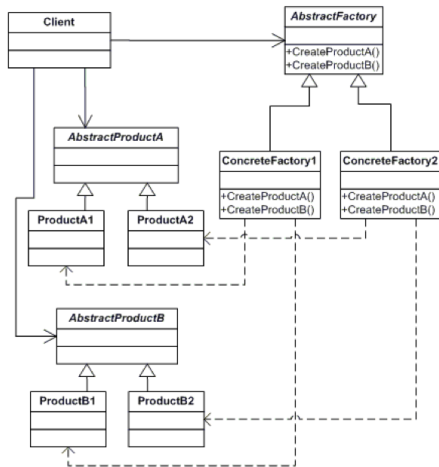
## Illustration sur un exemple :

- Créer une interface graphique avec widgets (boutons, menus, ...)
- Point de variation : OS (Linux, OSX, Windows)



# Patron GoF : Abstract factory (2/2)

## Solution Générique [Wikipedia] :



## Remarques :

- **AbstractFactory** et **AbstractProduct** sont généralement des interfaces  
~ Programmer pour des interfaces
- Les méthodes *createProduct...()* sont des *factory methods*

## Avantages du pattern :

- Indirection : Isole Client des implémentations des produits
- Protection des variations : Facilite la substitution de familles de produits
- Maintien automatique de la cohérence

Mais l'ajout de nouveaux types de produits est difficile...

# Patron GoF : Strategy (1/4)

## Problème :

Changer dynamiquement le comportement d'un objet

## Illustration sur un exemple :

- Dans un jeu vidéo, des personnages combattent des monstres...  
    ~ méthode `combat (Monstre m)` de la classe `Perso`  
...et le code de `combat` dépend du niveau du personnage
  - Sol. 1 : `combat` contient un cas pour chaque niveau possible
  - Sol. 2 : La classe `Perso` est spécialisée en sous-classes qui redéfinissent `combat`
- Représenter ces solutions en UML. Peut-on facilement :
  - Ajouter un nouveau niveau ?
  - Changer le niveau d'un personnage ?



# Patron GoF : Strategy (1/4)

## Problème :

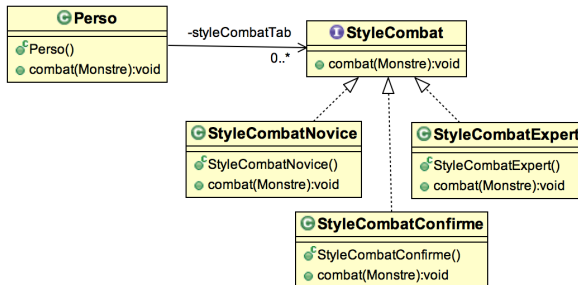
Changer dynamiquement le comportement d'un objet

## Illustration sur un exemple :

- Dans un jeu vidéo, des personnages combattent des monstres...  
    ~ méthode `combat (Monstre m)` de la classe `Perso`  
...et le code de `combat` dépend du niveau du personnage
  - Sol. 1 : `combat` contient un cas pour chaque niveau possible
  - Sol. 2 : La classe `Perso` est spécialisée en sous-classes qui redéfinissent `combat`
  - Sol. 3 : La classe `Perso` délègue le combat à des classes encapsulant des codes de combat et réalisant toutes une même interface
- Représenter ces solutions en UML. Peut-on facilement :
  - Ajouter un nouveau niveau ?
  - Changer le niveau d'un personnage ?

# Patron GoF : Strategy (2/4)

## Diagramme de classes de la solution 3 :



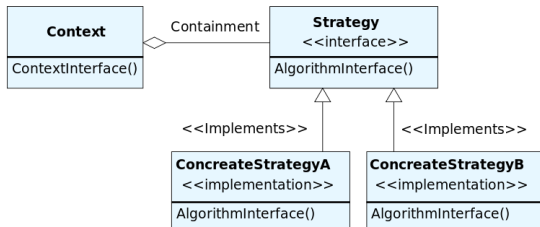
## Code Java de la classe Perso :

```
public class Perso{
    private int niveau;
    private final StyleCombat[] styleCombatTab = new StyleCombat[]{
        new StyleCombatNovice(),
        new StyleCombatConfirme(),
        new StyleCombatExpert()};
    public void combat(Monstre m){
        styleCombatTab[niveau].combat(m);
    }
}
```

# Patron GoF : Strategy (3/4)

## Solution générique :

[Wikipedia]



## Remarques :

- Principes de conception orientée objet mobilisés :
  - Indirection : Isole **Context** des implémentations de **Strategy**  
↪ Protection des variations
  - Composer au lieu d'hériter : Changer dynamiquement de stratégie
- Passage d'informations de **Context** à **Strategy**
  - en "poussant" : l'info est un param de **AlgorithmInterface()**
  - en "tirant" : le contexte est un param. de **AlgorithmInterface()** qui utilise des getters pour récupérer l'info

# Patron GoF : Strategy (4/4)

## Lambda expressions de Java 8

Création d'objets implémentant des interfaces contenant une seule méthode

### Code de la classe Perso revisité avec des lambda expressions :

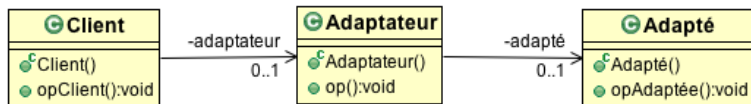
```
public class PersoAvecLambda {
    private int niveau;
    private final StyleCombat[] styleCombatTab = new StyleCombat[]{
        m->combatNovice(m),
        m->combatConfirme(m),
        m->combatExpert(m)};
    public void combat(Monstre m){
        styleCombatTab[niveau].combat(m);
    }
    private void combatNovice(Monstre m) {
        System.out.println("Je me bats comme un novice car je suis de niveau "+niveau);
    }
    private void combatConfirme(Monstre m) {
        System.out.println("Je me bats comme un confirme car je suis de niveau "+niveau);
    }
    private void combatExpert(Monstre m) {
        System.out.println("Je me bats comme un expert car je suis de niveau "+niveau);
    }
}
```

# Patron GoF : Adaptateur

## Problème :

Fournir une interface stable (Adaptateur) à un composant dont l'interface peut varier (Adapté)

## Solution générique :



→ Application des principes “indirection” et “protection des variations”

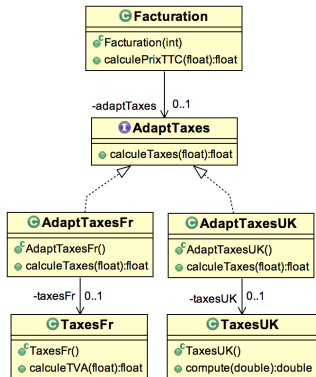
## Exercice :

Utiliser un adaptateur pour calculer les taxes en fct du pays (UK ou France) :

- Pour UK, la classe `TaxesUK` a la méthode `double compute(double prix)`
- Pour France, la classe `TaxesFr` a la méthode `float calcTVA(float prix)`

# Patron Adaptateur avec plusieurs classes adaptées

## Diagramme de classes



## Code de Facturation

```
public class Facturation {
    private AdaptTaxes adaptTaxes;

    public Facturation(int pays){
        switch (pays){
            case 1 : adaptTaxes = new AdaptTaxesUK();
                       break;
            case 2 : adaptTaxes = new AdaptTaxesFr();
                       break;
        }
    }

    public float calculerPrixTTC(float prixHT){
        return adaptTaxes.calculerTaxes(prixHT);
    }
}
```

## Questions :

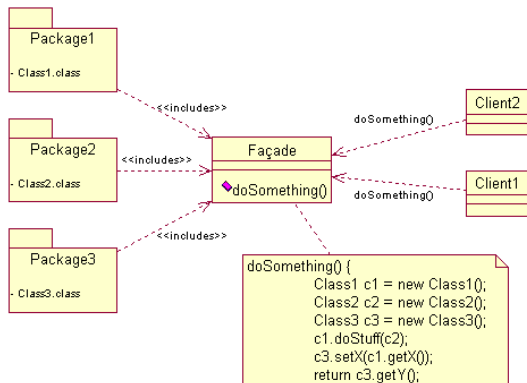
- Peut-on utiliser des lambda expressions ici ?
- Quel pattern utiliser pour créer `adaptTaxes` ?
- Points communs et différences avec le pattern Strategy ?

# Patron GoF : Facade

## Problème :

Fournir une interface simplifiée (Facade)

## Solution générique [Wikipedia] :



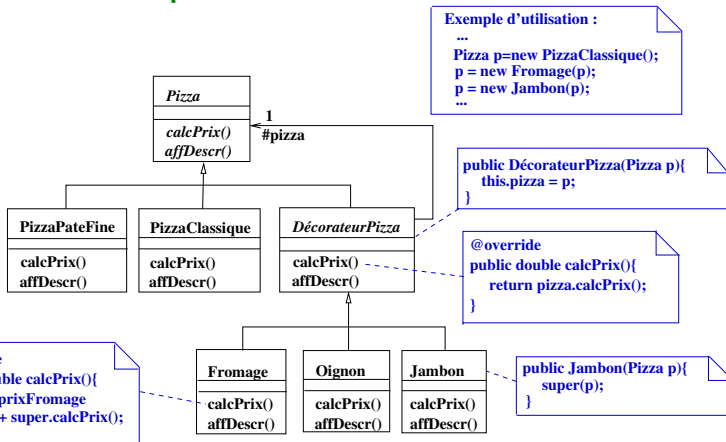
→ Application des principes “indirection” et “protection des variations”

# Patron GoF : Décorateur (1/2)

## Problème :

Attacher dynamiquement des responsabilités supplémentaires à un objet

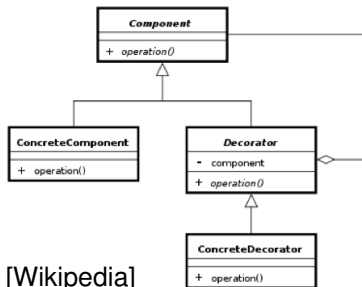
## Illustration sur un exemple :





# Patron GoF : Décorateur (2/2)

## Solution générique :



[Wikipedia]

## Remarques :

- Composer au lieu d'hériter : Ajout dynamique de responsabilités à **ConcreteComponent** sans le modifier
- $n$  décors  $\Rightarrow 2^n$  combinaisons
- **Inconvénient** : Peut générer de nombreux petits objets "enveloppes"

## Utilisation pour décorer les classes d'entrée/sortie en Java :

- **Component** : `InputStream`, `OutputStream`
- **ConcreteComponent** : `FileInputStream`, `ByteArrayInputStream`, ...
- **Decorator** : `FilterInputStream`, `FilterOutputStream`
- **ConcreteDecorator** : `BufferedInputStream`, `CheckedInputStream`, ...

# Adaptateur, Facade et Décorateur

## Points communs :

- Indirection  $\leadsto$  Enveloppe (wrapper)
- Protection des variations

## Différences :

- Adaptateur : Convertit une interface en une autre (attendue par un Client)
- Facade : Fournit une interface simplifiée
- Décorateur : Ajoute dynamiquement des responsabilités aux méthodes d'une interface sans la modifier

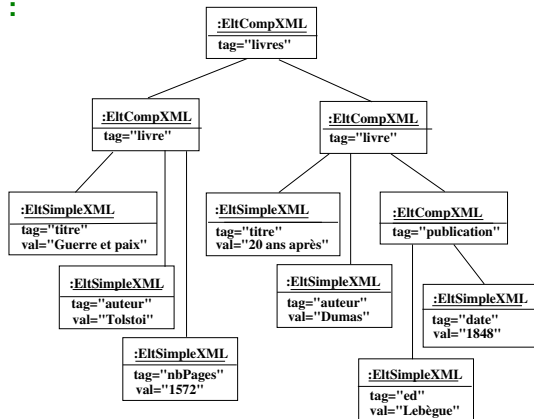
# Patron GoF : Composite (1/2)

## Problème :

Représenter des hiérarchies composant/composé et traiter de façon uniforme les composants et les composés

## Illustration sur un exemple :

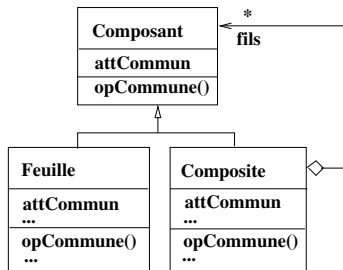
```
< ?xml version="1.0" ?>
< livres>
  < livre>
    < titre>Guerre et paix</titre>
    < auteur>Tolstoï</auteur>
    < nbPages>1572</nbPages>
  </livre>
  < livre>
    < titre>20 ans après</titre>
    < auteur>Dumas</auteur>
    < publication>
      < ed>Lebègue</ed>
      < date>1848</date>
    </publication>
  </livre>
</livres>
```



Comment compter le nombre de tags ?

# Patron GoF : Composite (2/2)

## Solution générique :



## Exercices :

- Définir les opérations permettant de :
  - Compter le nombre de fils d'un composant
  - Compter le nombre de descendants d'un composant
  - Ajouter un fils à un composant
- Comment accéder séquentiellement aux fils d'un Composite ?
- Comment accéder séquentiellement aux descendants d'un Composite ?