Patrons de conception

François Schwarzentruber
ENS Cachan – Antenne de Bretagne

Qu'est ce qu'un patron de conception ?

une problématique :
pouvoir annuler

une façon de
structurer
le programme

Ce cours parle de ça :

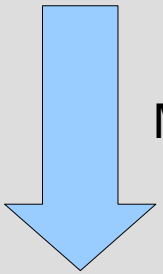
Propriétés fonctionnelles

- Correction d'un algorithme
- Complexité temps/espace
- Vivacité

- Propriétés non fonctionnelles
- Facile à comprendre
- Facile à partager
- Facile à étendre

Sagesse des anciens : ne pas réinventer la roue

- Nombres entiers
- Matrices
- Rubik's cube...



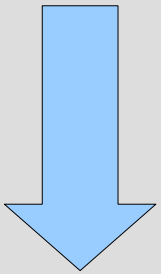
Même concepts

Théorie des groupes

(Évariste Galois)

- Sous-groupes
- Ordre d'un élément

- Prouveur
- Jeu
- Logiciel de dessin

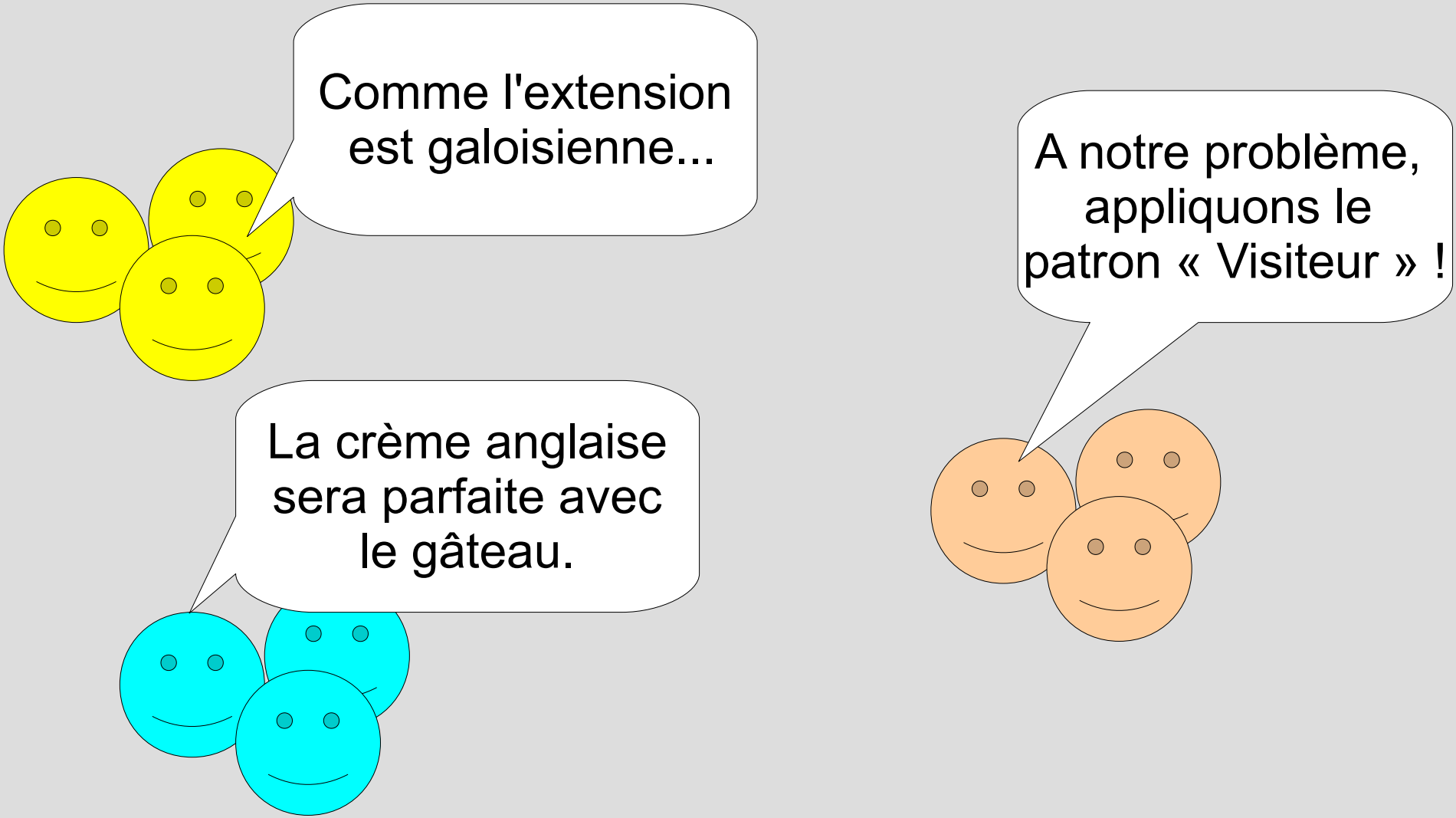


Même problématiques
de génie logiciel

Patrons de conception

- « Façade »
- « Visiteur »

Sagesse des anciens : du vocabulaire !



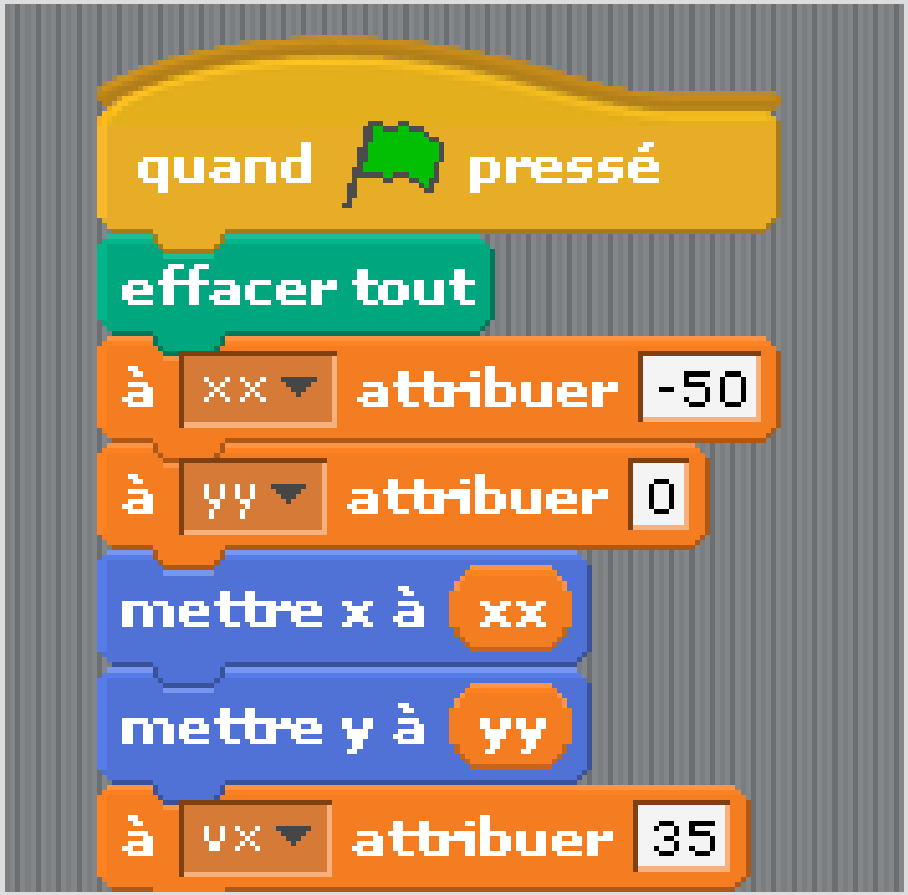
Plan (un cours « catalogue de bonnes recettes de cuisine »)

- Patrons de création
- Patrons de structure
- Patrons de comportement

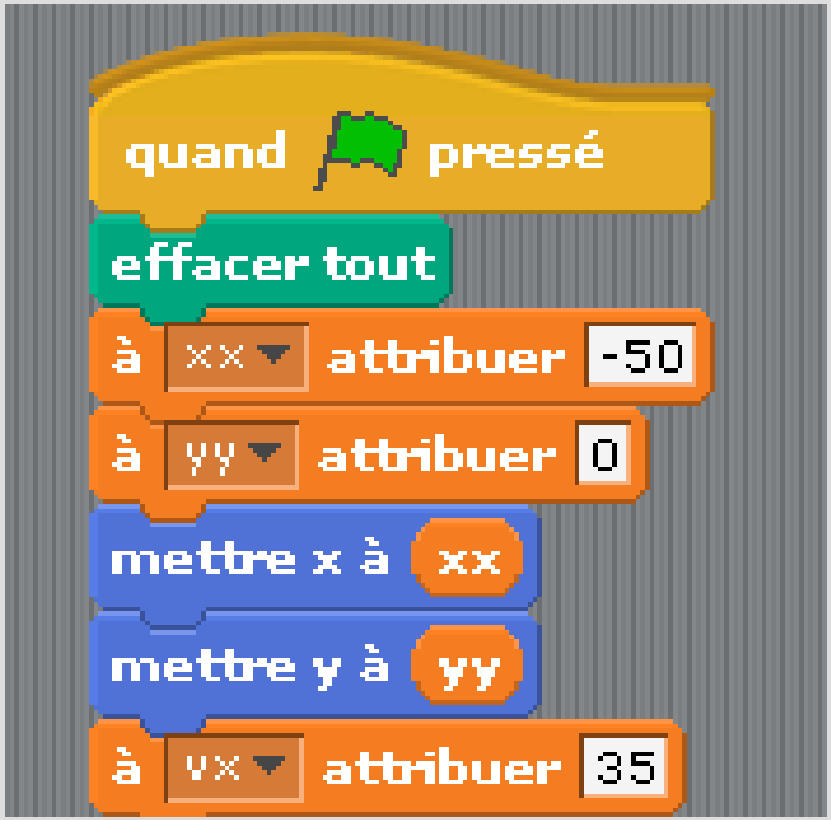
Patrons de création

- Patron « Fabrique abstraite »
- Patron « Prototype »

VOUS créez une application pour apprendre l'algorithmique

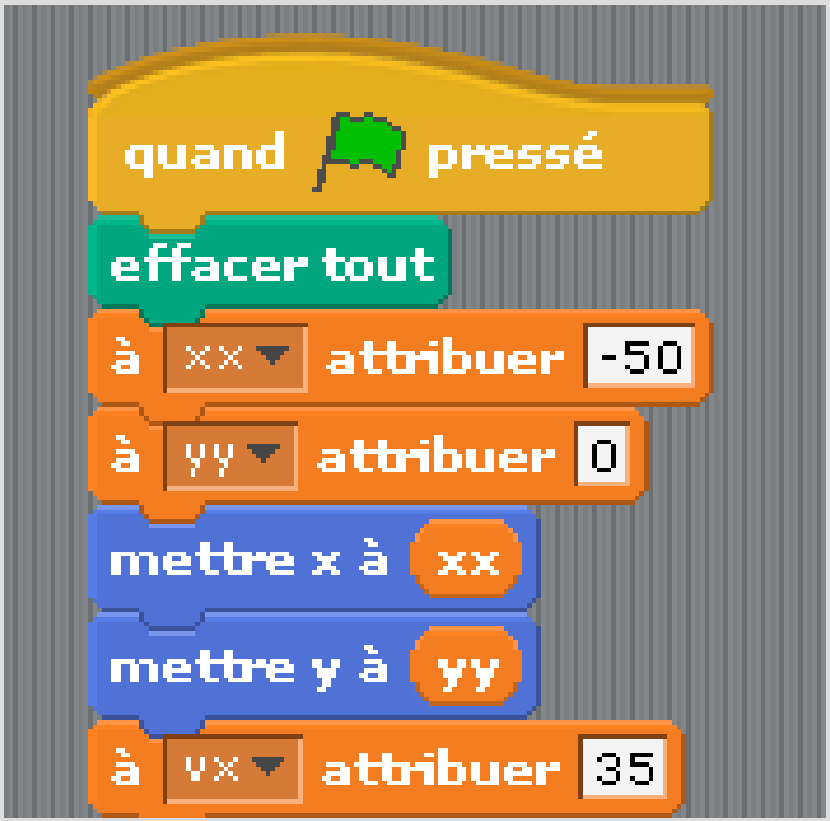


Besoin : adapter le logiciel pour différentes plate-formes



- Windows
- Linux
- MacOS
- Android

Besoin : adapter le logiciel pour différents publics



enfants

lycéens

Constat : on a créé des objets un peu partout !


```
@MessageDriven(mappedName = "jms/myQueue")
public class StockProcessListenerMDB implements MessageListener {
    @WebServiceRef(name="sun-web.serviceref/SynchronousSampleService")
    com.sun.ca.mdb.SynchronousSampleService service;
    /** Creates a new instance of StockProcessListenerMDB */
    public StockProcessListenerMDB() {
    }

    public void onMessage(Message message) {
        try {
            com.sun.ca.mdb.MyPortType port = service.getSynchronousSamplePortName();
            String ide = message.getStringProperty("id");
            double price = message.getDoubleProperty("price");
            int noOfStocks = message.getIntProperty("stocks");
            com.sun.ca.mdb.OperationARequest reql = new com.sun.ca.mdb.OperationARequest();
            reql.setId(ide);
            reql.setPrice(price);
            reql.setNoOfStocks(noOfStocks);
            com.sun.ca.mdb.OperationAResponse resp = port.operationA(reql);
            System.out.println("Result = "+resp.getReturnValue());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

new

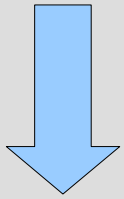
new

new

```
1  package test;
2
3  import java.io.IOException;
4
5  import jb2b.petitlien.facade.LittleLinkException;
6  import jb2b.petitlien.facade.LittleLinkRequest;
7
8  import junit.framework.TestCase;
9
10 public class TestPetitLien extends TestCase {

       public void testPetitLien() throws IOException {
           LittleLinkRequest request =
               new LittleLinkRequest("MonUrlTresLong", "Alias");

           LittleLinkRequest request2 =
               new LittleLinkRequest("MonUrlTresLong", "4"); // Alias automatique
           try {
               String petitLien = request.getLittleLink();
               String petitLien2 = request2.getLittleLink();
               System.out.pri
               System.out.pri
           } catch (LittleLin
               e.printStackTrace();
           }
       }

   }
```

Solution non maintenable

```
b = new BriqueEffacerTout() ;
```



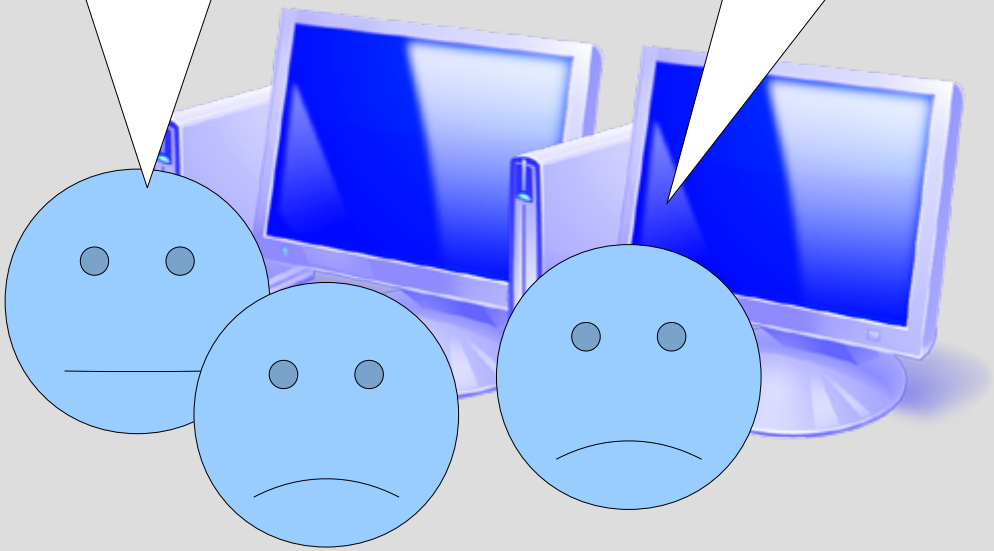
```
if(pourlesEnfants)
{
        b = new briqueEffacerToutPourEnfant() ;
}
else
{
        b = new briqueEffacerToutPourLyceen() ;
}
```

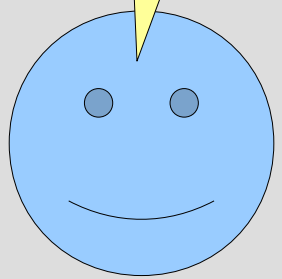
Patron de conception : fabrique abstraite

On va devoir adapter notre logiciel d'algorithmique à deux publics.

Comment faire ?

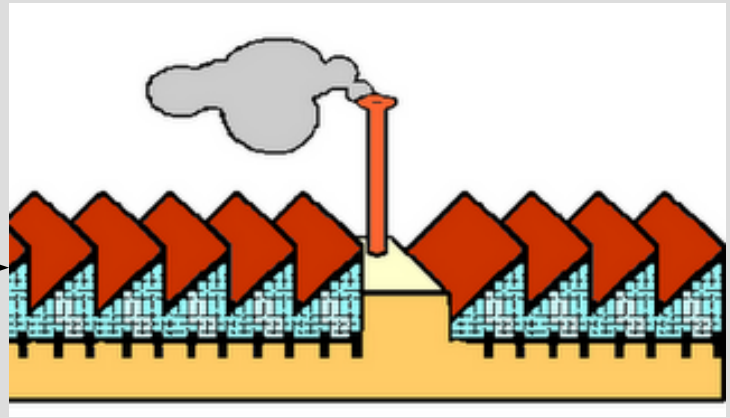
On applique le patron de conception « fabrique abstraite ».





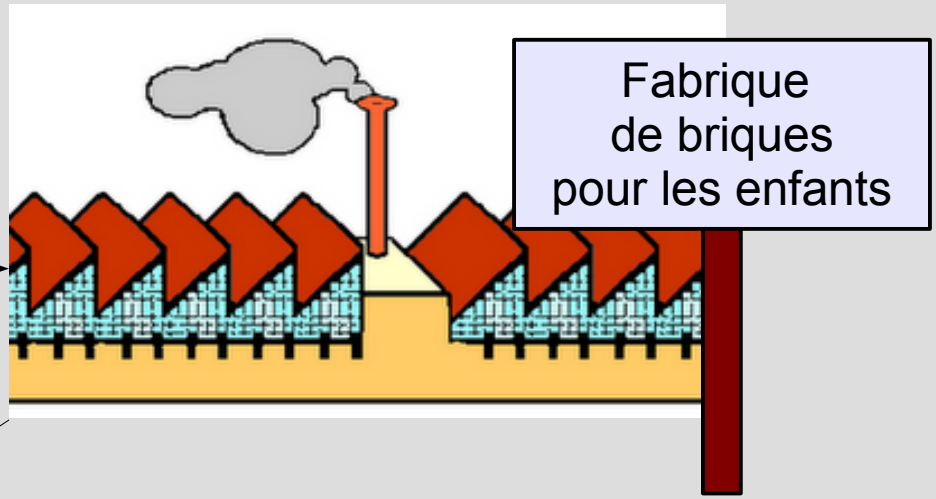
Solution : isoler la création des objets dans des « fabriques abstraites »

stp, fabrique moi une brique « EffacerTout »



Solution : isoler la création des objets dans des « fabriques abstraites »

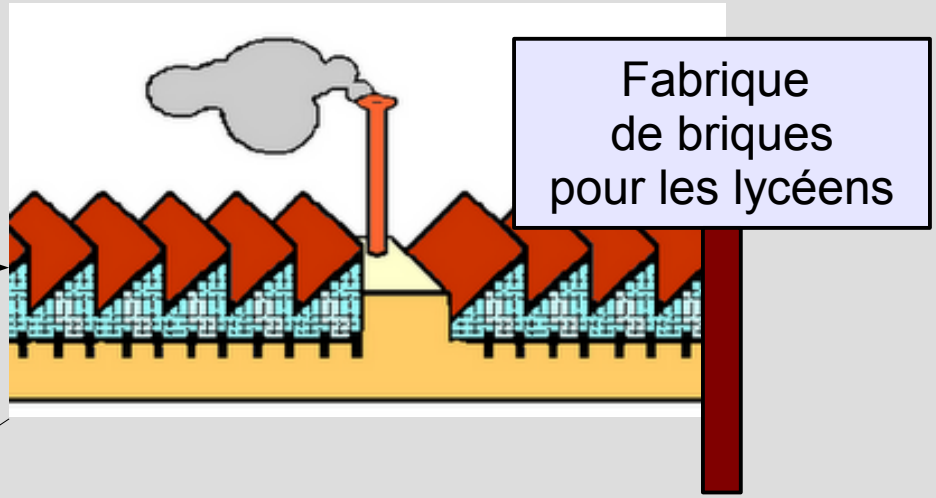
stp, fabrique moi une brique « EffacerTout »



effacer tout

Solution : isoler la création des objets dans des « fabriques abstraites »

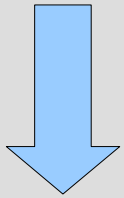
stp, fabrique moi une brique « EffacerTout »

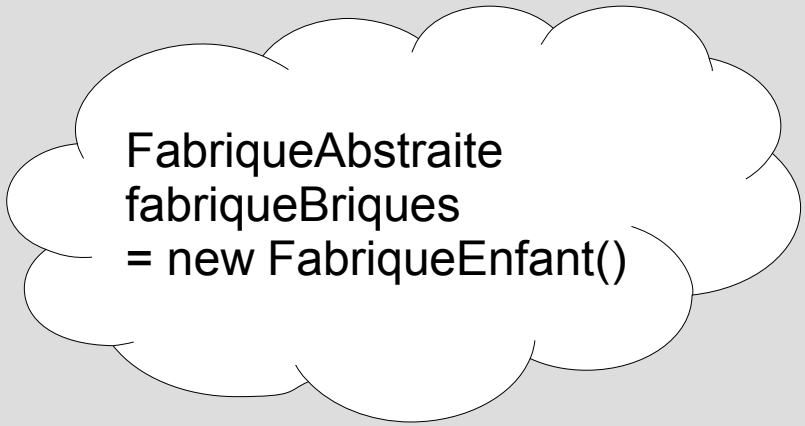


effacer tout

Bonne solution

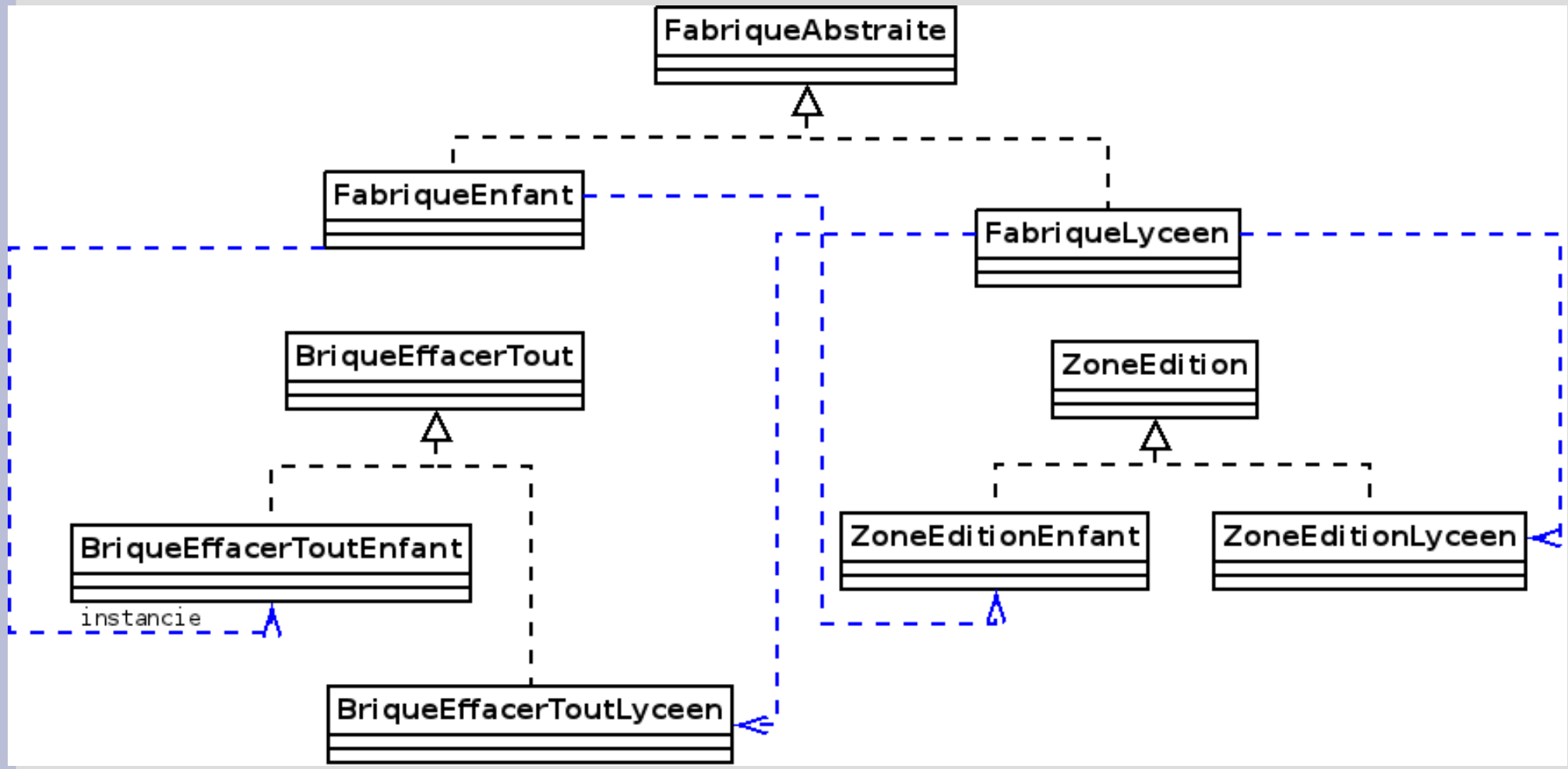
b = new BriqueEffacerTout() ;





fabriqueBriques.getNouvelleBriqueEffacerTout()

Patron de conception : fabrique abstraite



Patron de conception : fabrique abstraite

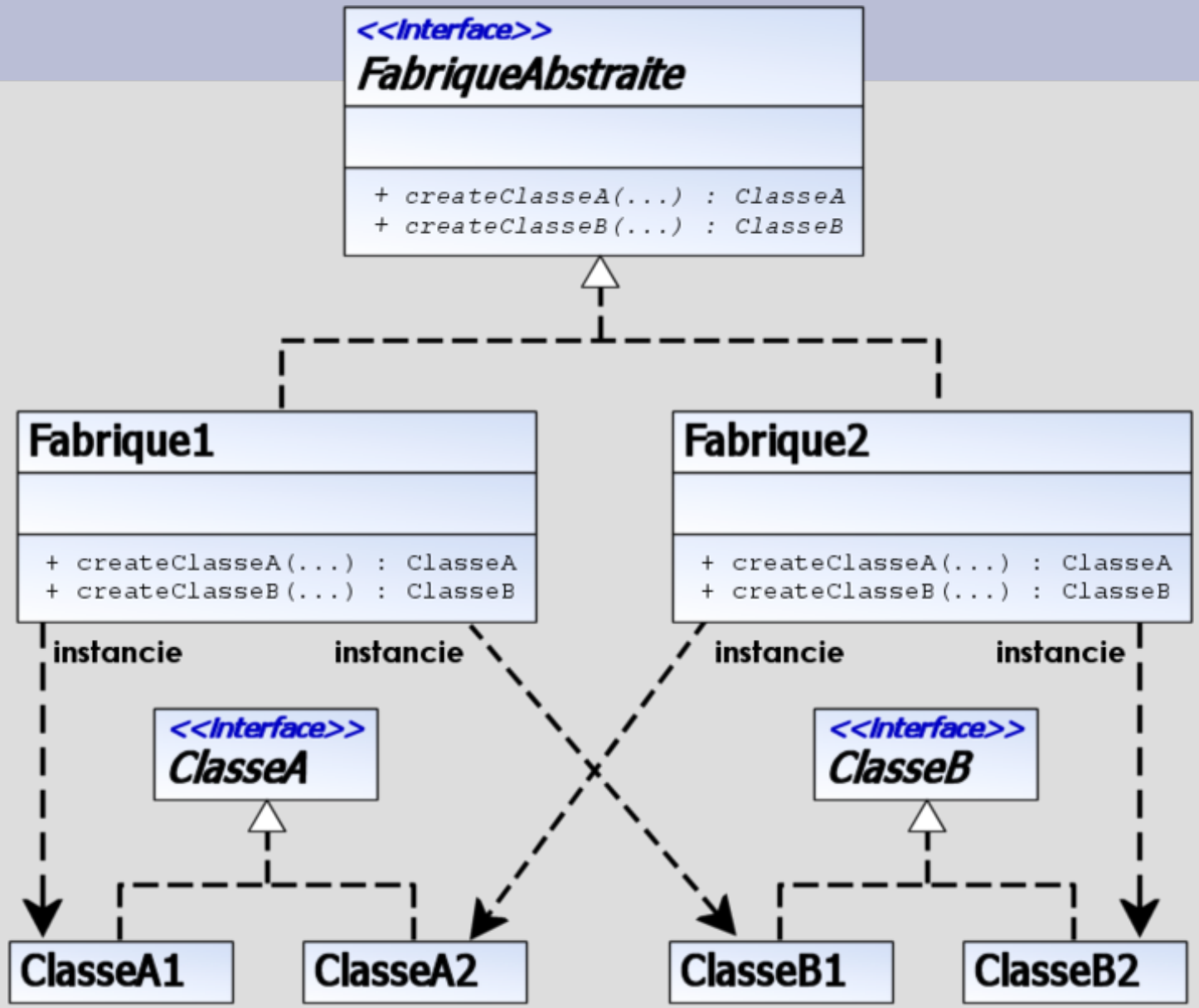
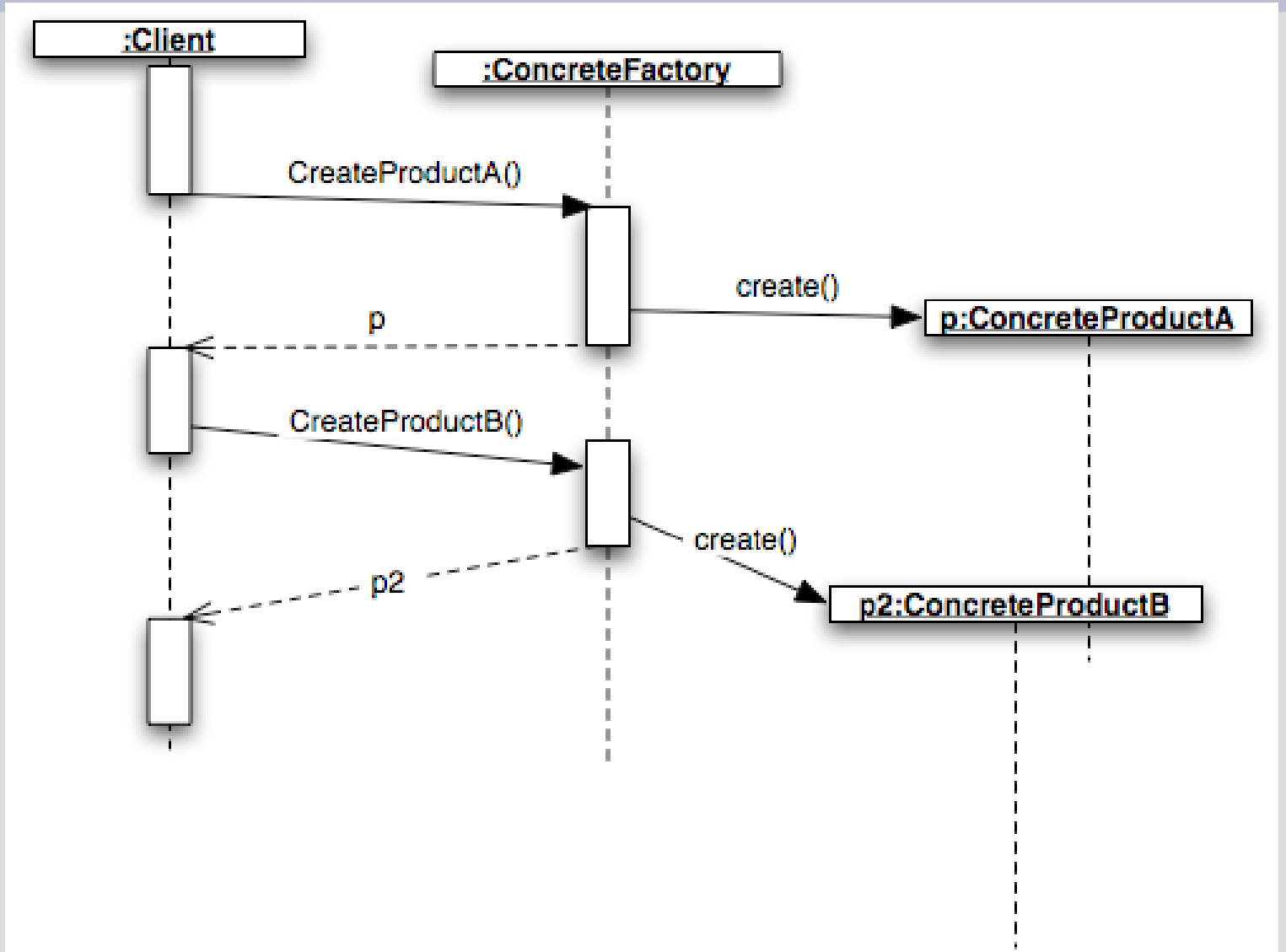


Diagramme de séquence de la fabrique abstraite

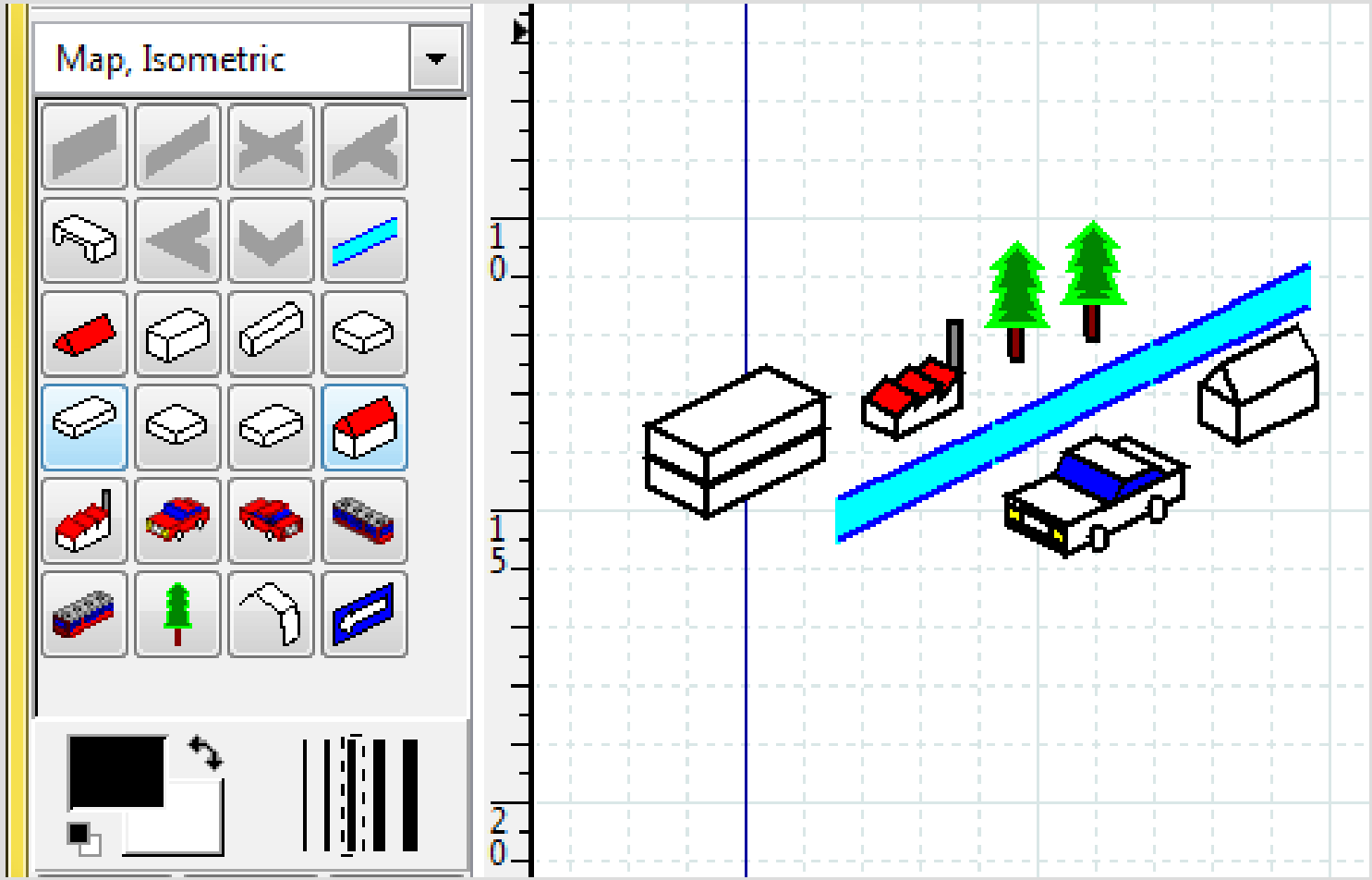


Source : cours de Noël Plouzeau

Conclusion sur la fabrique abstraite

- + Ajouter une nouvelle plate-forme
- + Assurer la cohérence du logiciel (deux plate-formes ne sont pas mélangées)
- + Cloisonne les besoins du client et les soucis dus aux plate-formes
- - Rajout d'un nouveau composant (brique, bouton...) difficile

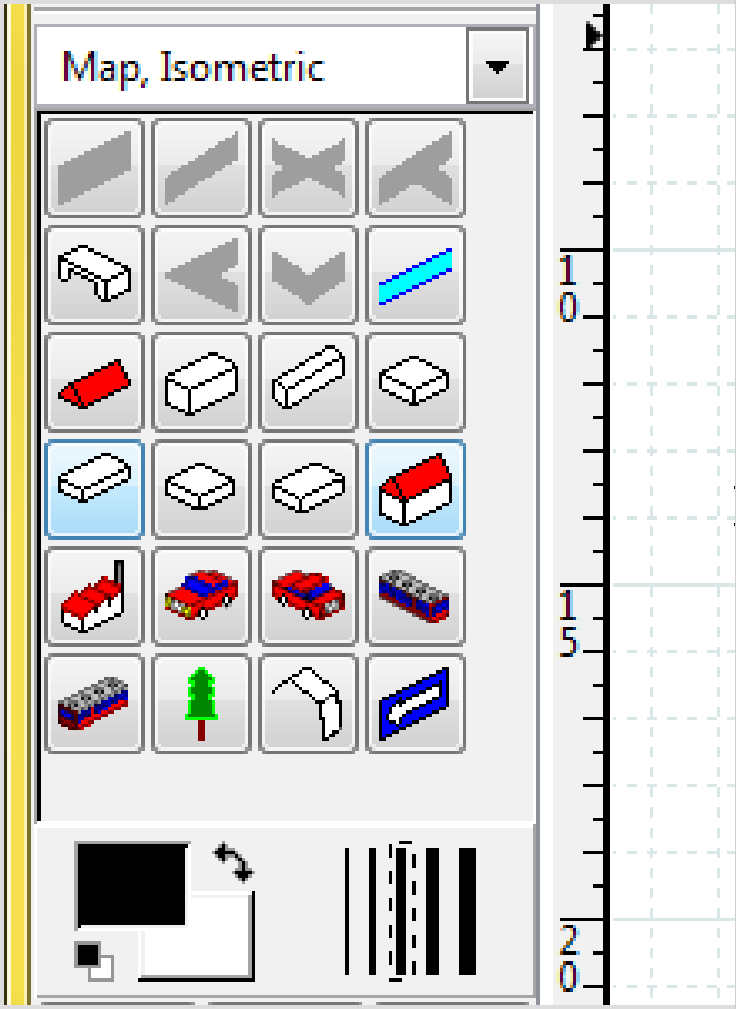
Créer des objets selon un modèle : le patron « prototype »



Source : logiciel Dia

Besoins

- Créer selon des modèles
- Enrichir la liste des modèles



A ne pas faire : une création différente au sein de chaque bouton...

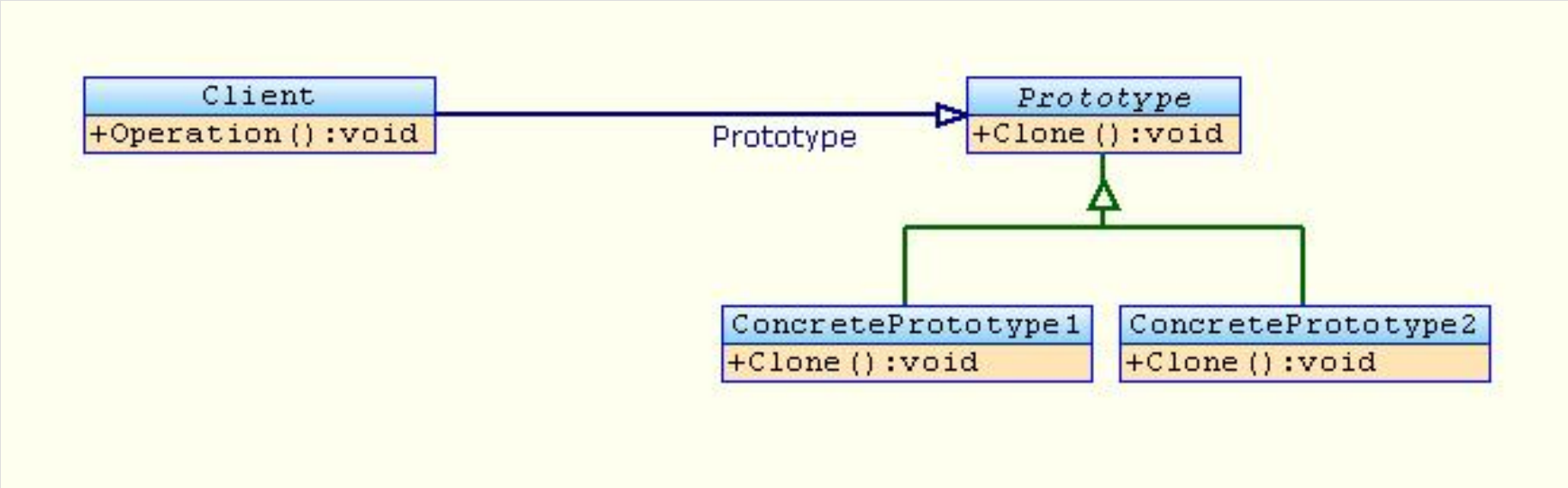
```
new Maison(Color.RED,
Color.WHITE, 16, 32, 16)
```

```
new Arbre(Color.GREEN,
Color.BROWN, 8, 32, 8)
```

A faire

Chaque bouton contient un prototype à cloner.

Diagramme de classe de « Prototype »



Source : wikipedia

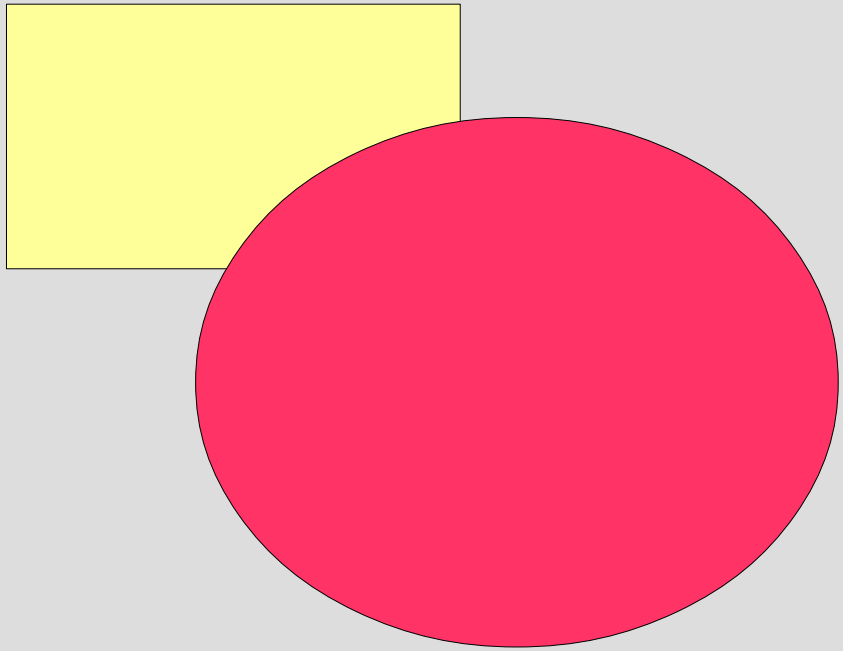
Conclusion sur le patron « Prototype »

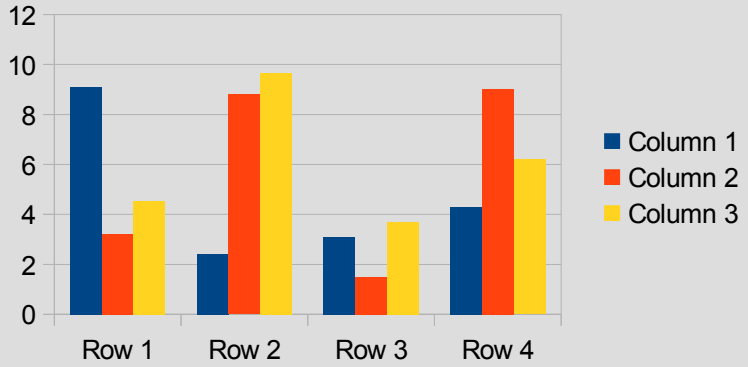
- + Copier un objet sans se soucier de sa classe exacte
- + Ajouter dynamiquement des objets à une palette
- - le mécanisme de clonage à créer

Patrons de structure

- Adapter une interface (~ adaptateur)
- Rendre un programme lisible (interface simplifiée ~ façade)
- Structures récursives (~ composite)
- Utilisation de la délégation (~ décoration)

Patron de conception « Adaptateur »



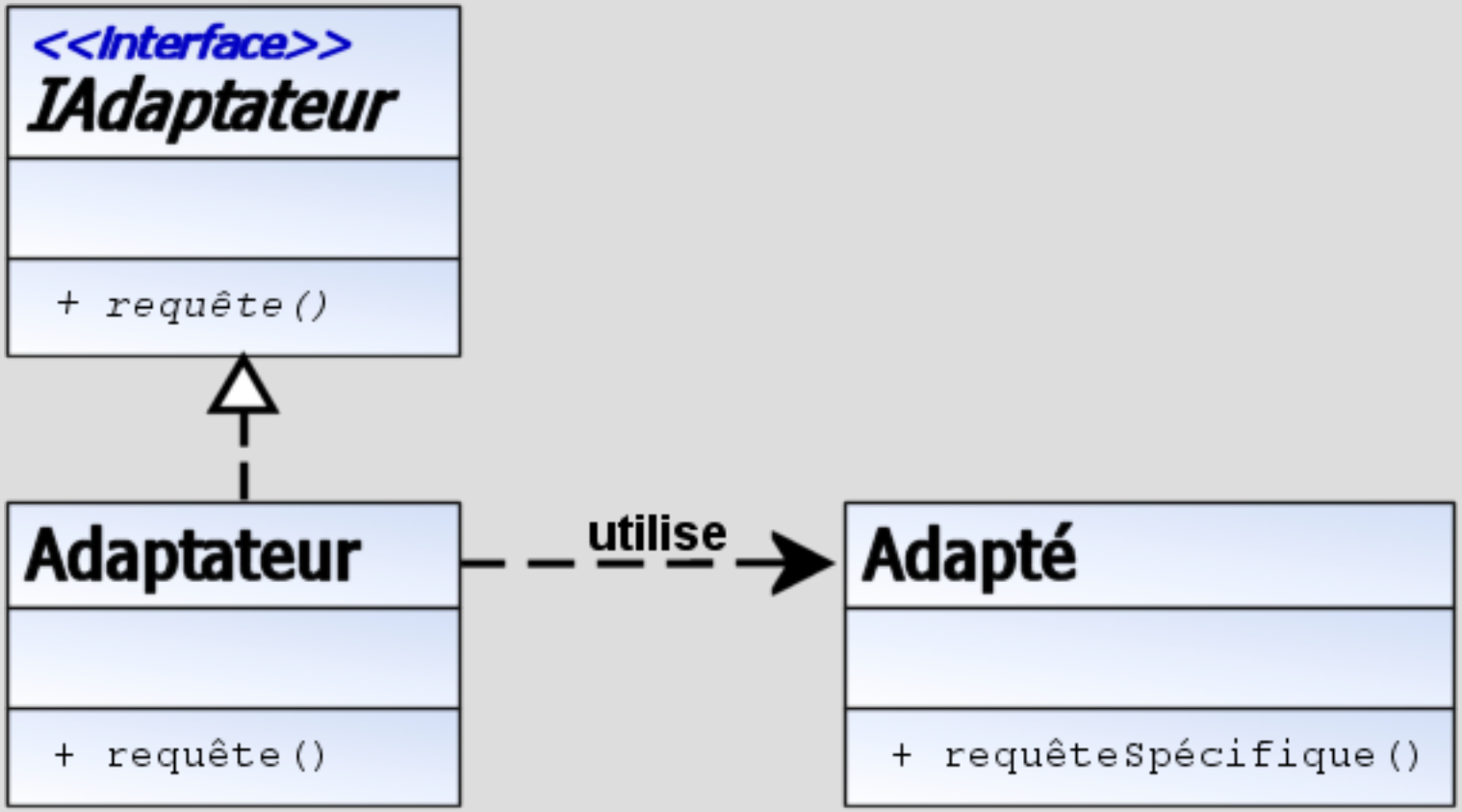


ObjetGraphique
getBounds() : Rect

OpenOfficeGraphics
width height

(interface utilisée par mon logiciel)

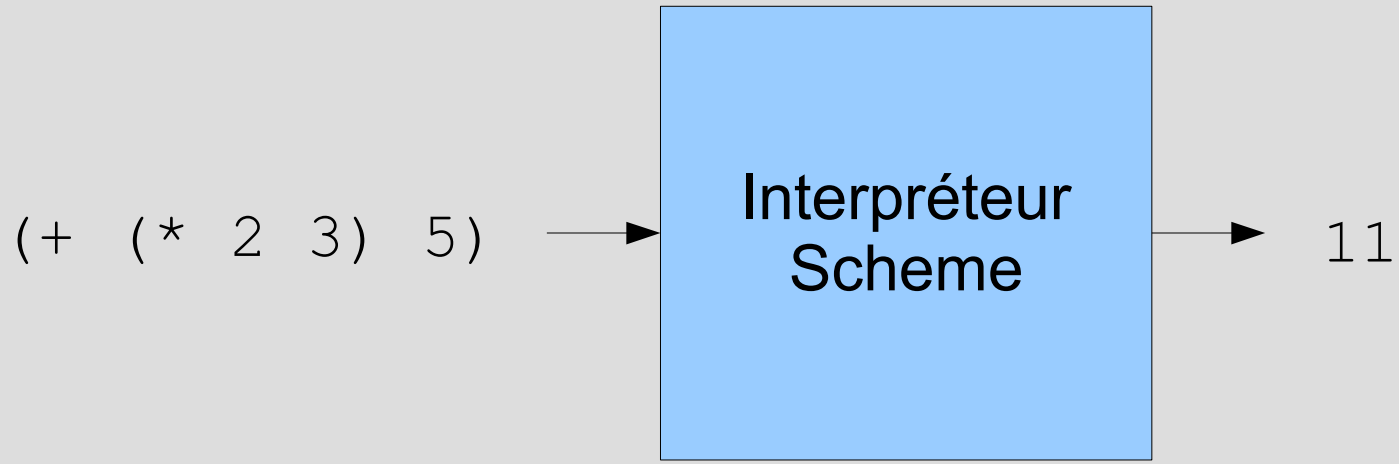
Patron de conception « Adaptateur »



source : Wikipedia

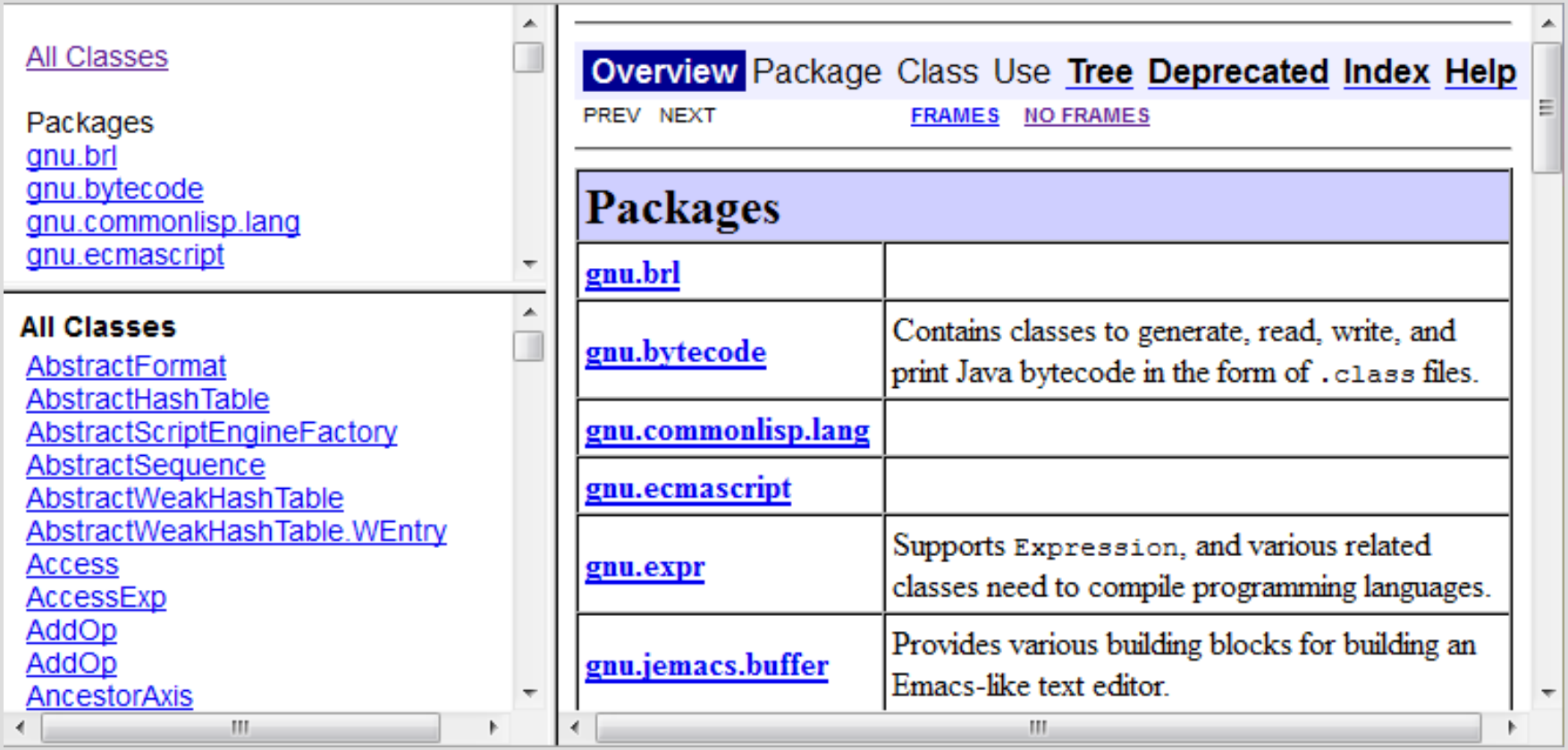
Patron « façade » : à un besoin simple, une interface simple

Mon besoin :

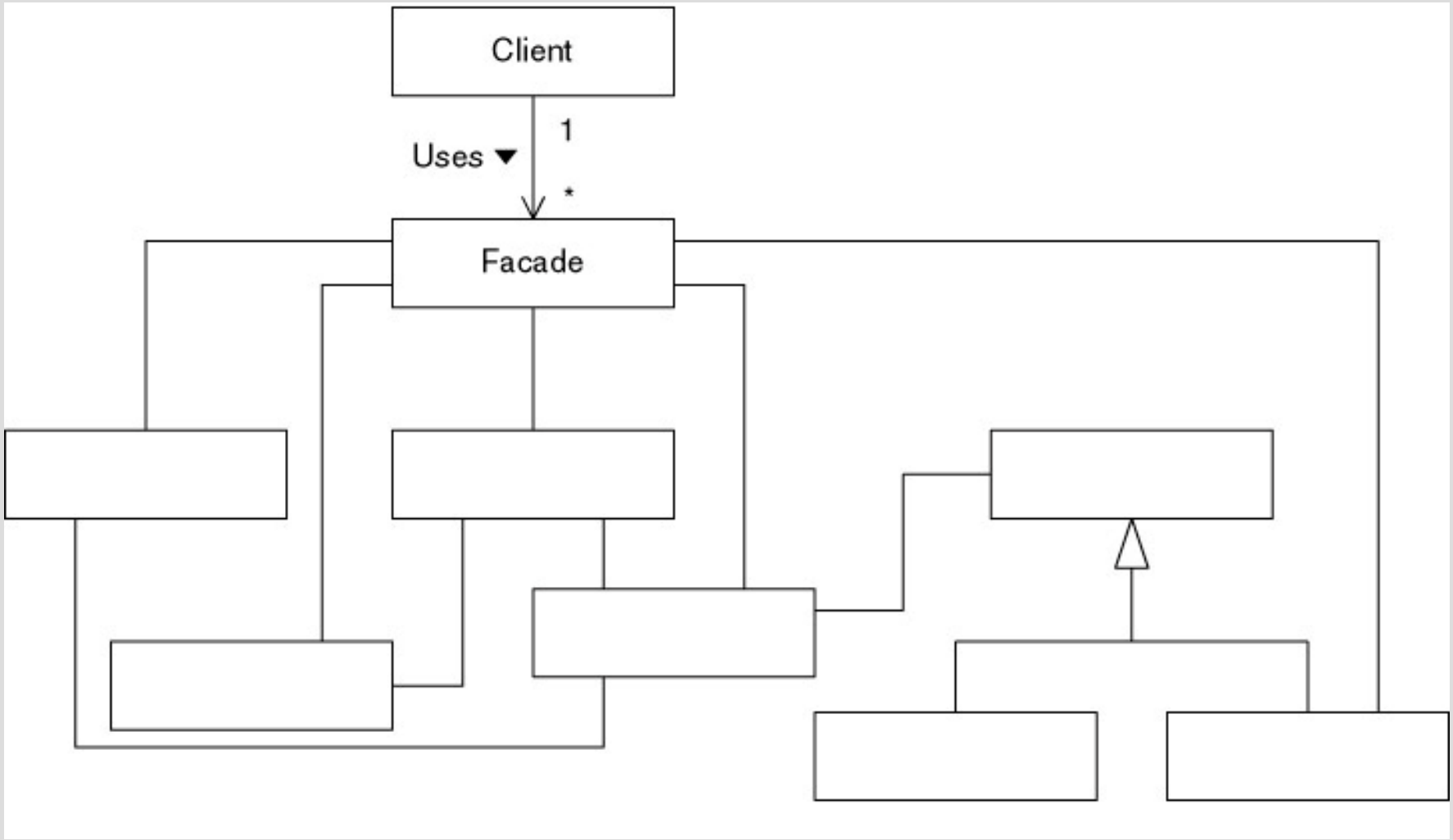


Patron « façade » : à un besoin simple, une interface simple

Et j'ai à ma disposition kawa, une bibliothèque JAVA pour parser/compiler/interpréter du code Scheme et créer/enregistrer/restaurer des environnements :



Patron « façade »



Source : Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition by Mark Grand

Différence entre le patron « Façade » et « Adaptateur »

Façade

- But : créer une interface simplifiée à une bibliothèque complexe à utiliser

Adaptateur

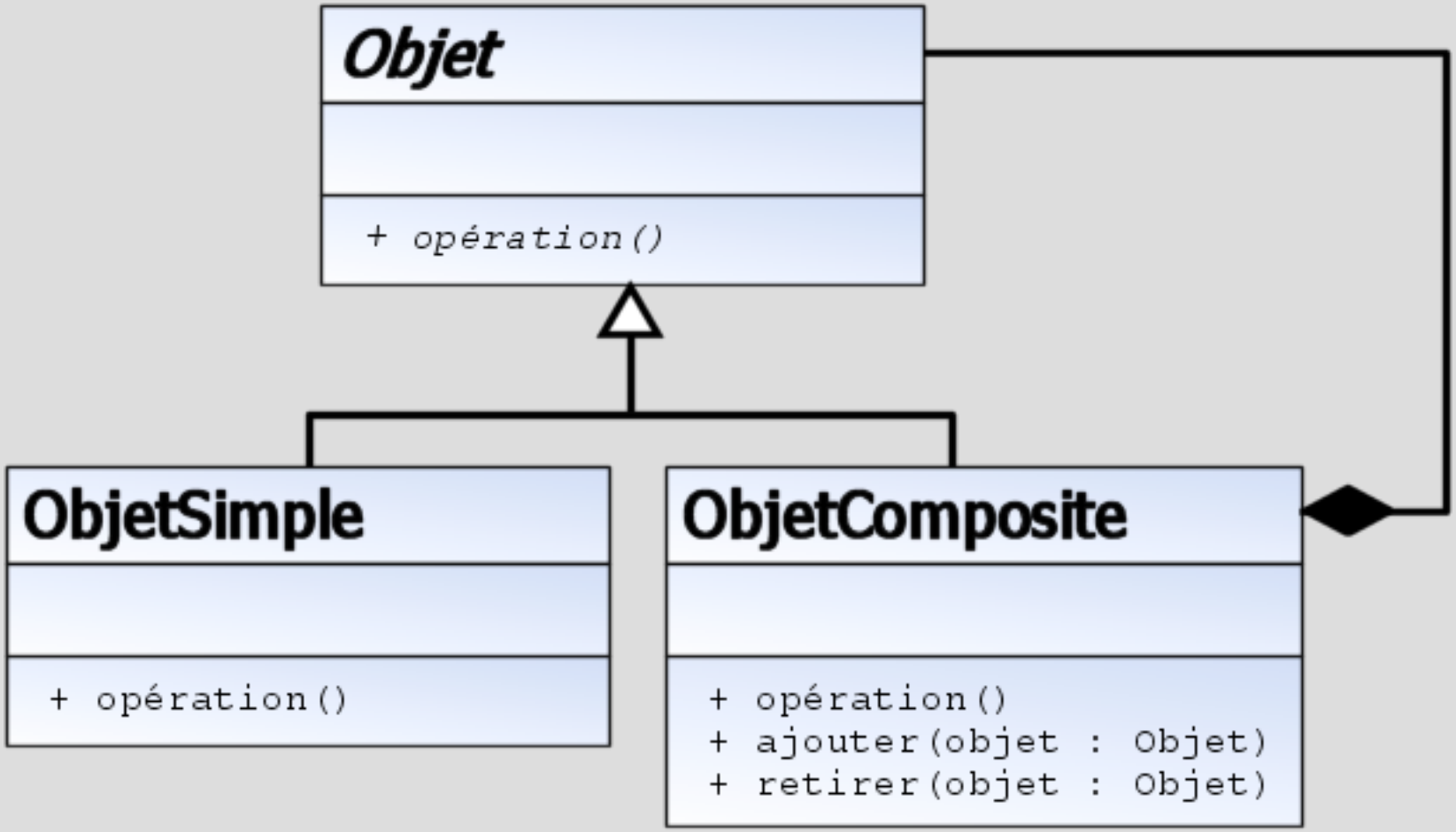
- But : offrir une interface conforme et adapter une classe

Patron « Composite »

Objets récursifs :

- Arborescence de fichiers
- Expressions
- Structure d'un document
- Commande (on verra plus tard)
- Etc.

Patron « Composite »



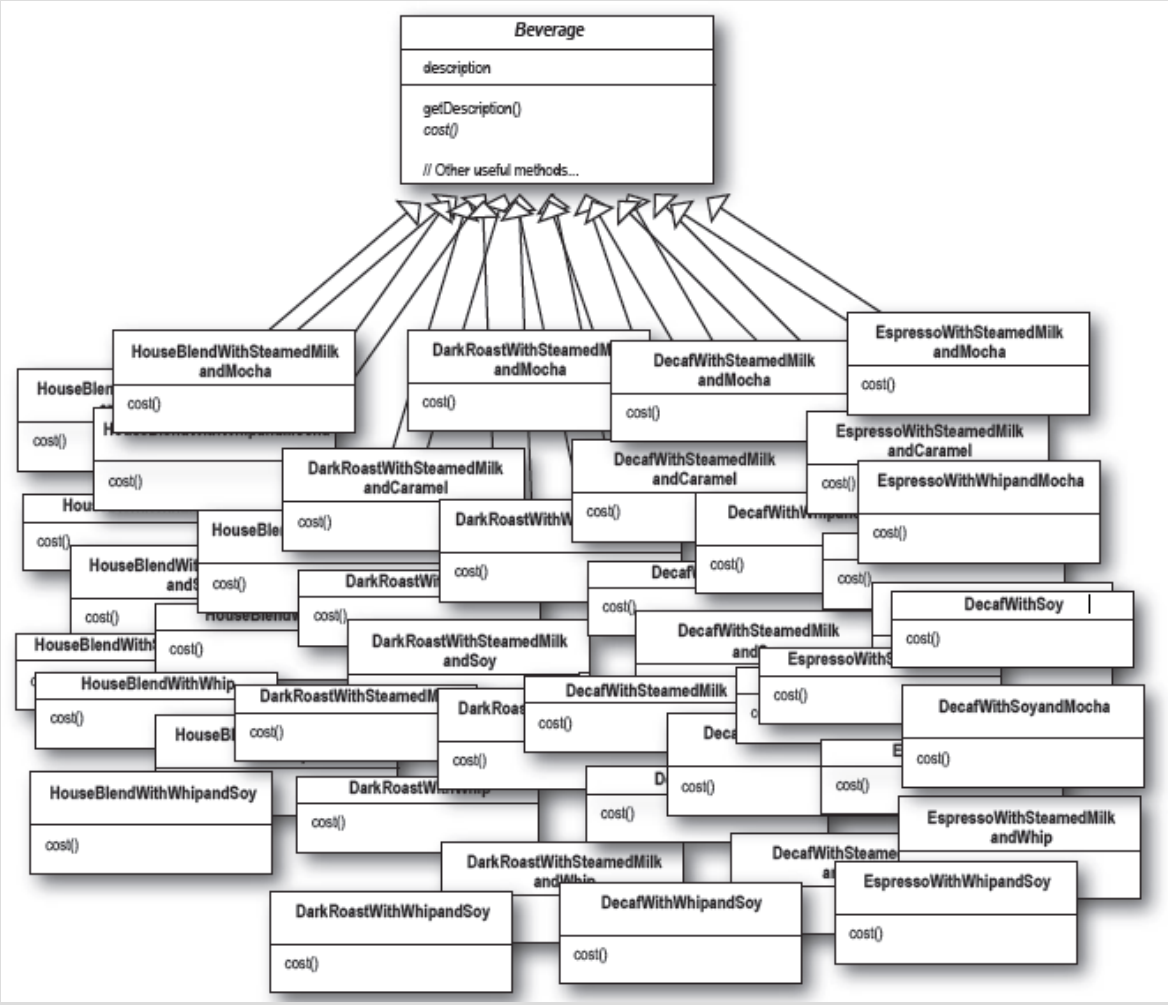
Source : wikibooks

Patron « Décoration »

Une fenêtre peut :

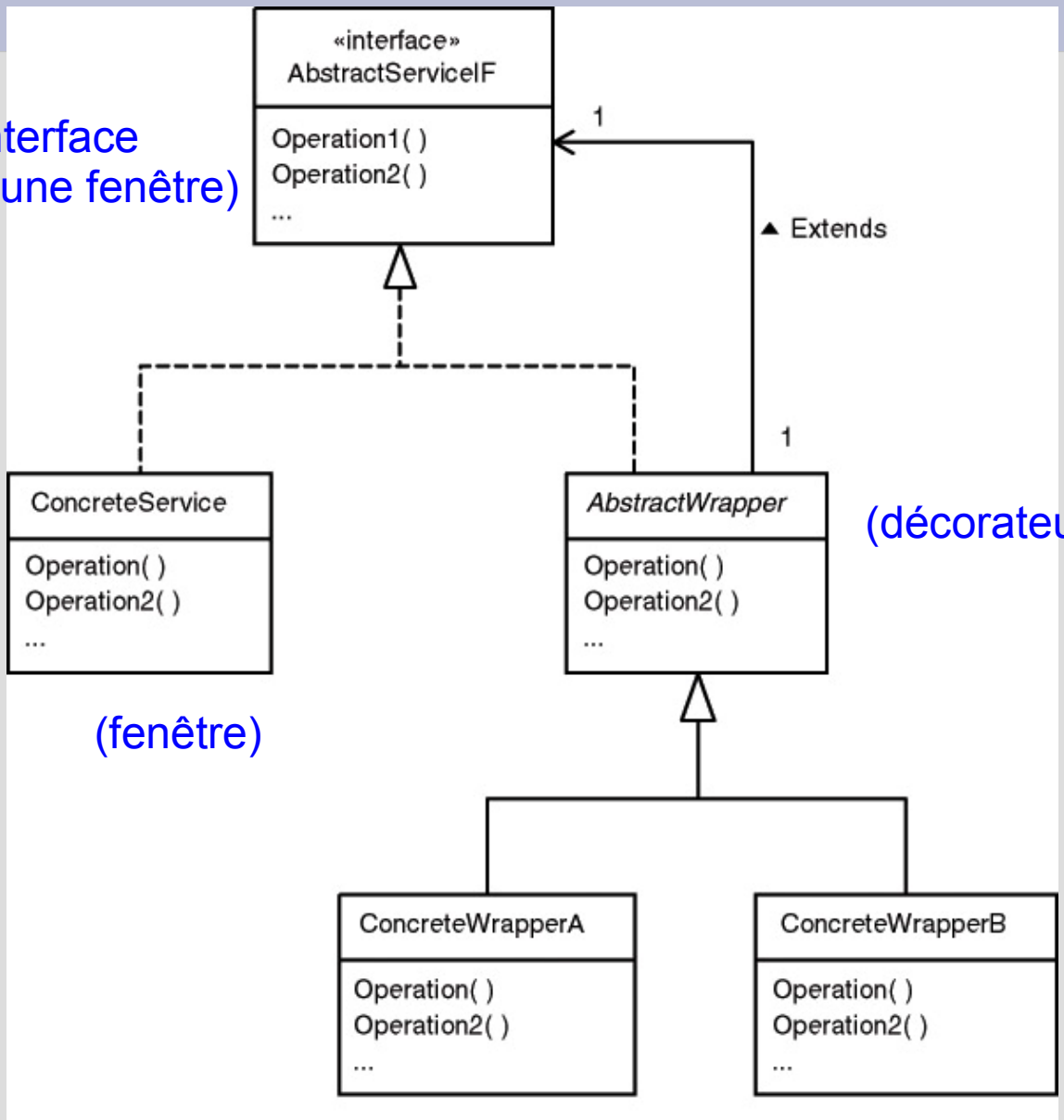
- Avoir / ne pas avoir une bordure
- Avoir / ne pas avoir des barres de défilement
- Avoir / ne pas avoir un fond
- Avoir / ne pas avoir une gestion de zoom
- Avoir / ne pas avoir d'effet lors du déplacement

Avec l'héritage



on aurait 2^n classes différentes !

Patron « Décoration » utilise la délégation



(interface d'une fenêtre)

(fenêtre)

(décorateur abstrait)

(décorateur concret comme « fenêtre avec un fond »)

- Avec la délégation on a O(n) classes.

Conclusion sur le patron « Décorateur »

- + Meilleure maintenance que des « if »
- + Moins de classes qu'avec l'héritage
- + Modifier dynamiquement les décorations
- - Comportement inattendu...

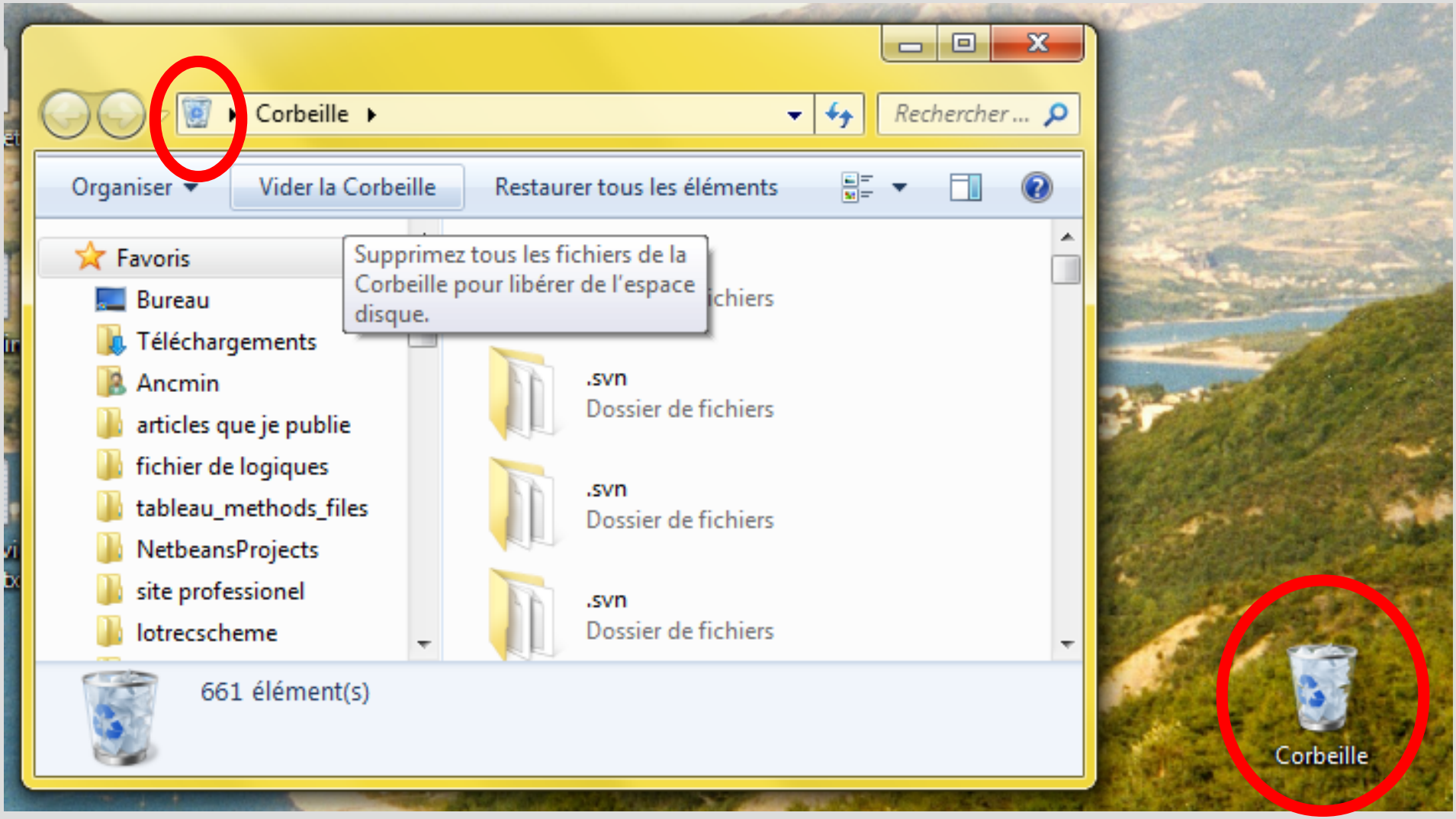
  new FenetreBordure(new FenetreFond(f))

  ~ new FenetreFond(new FenetreBordure(f)) ?

Patrons de comportement

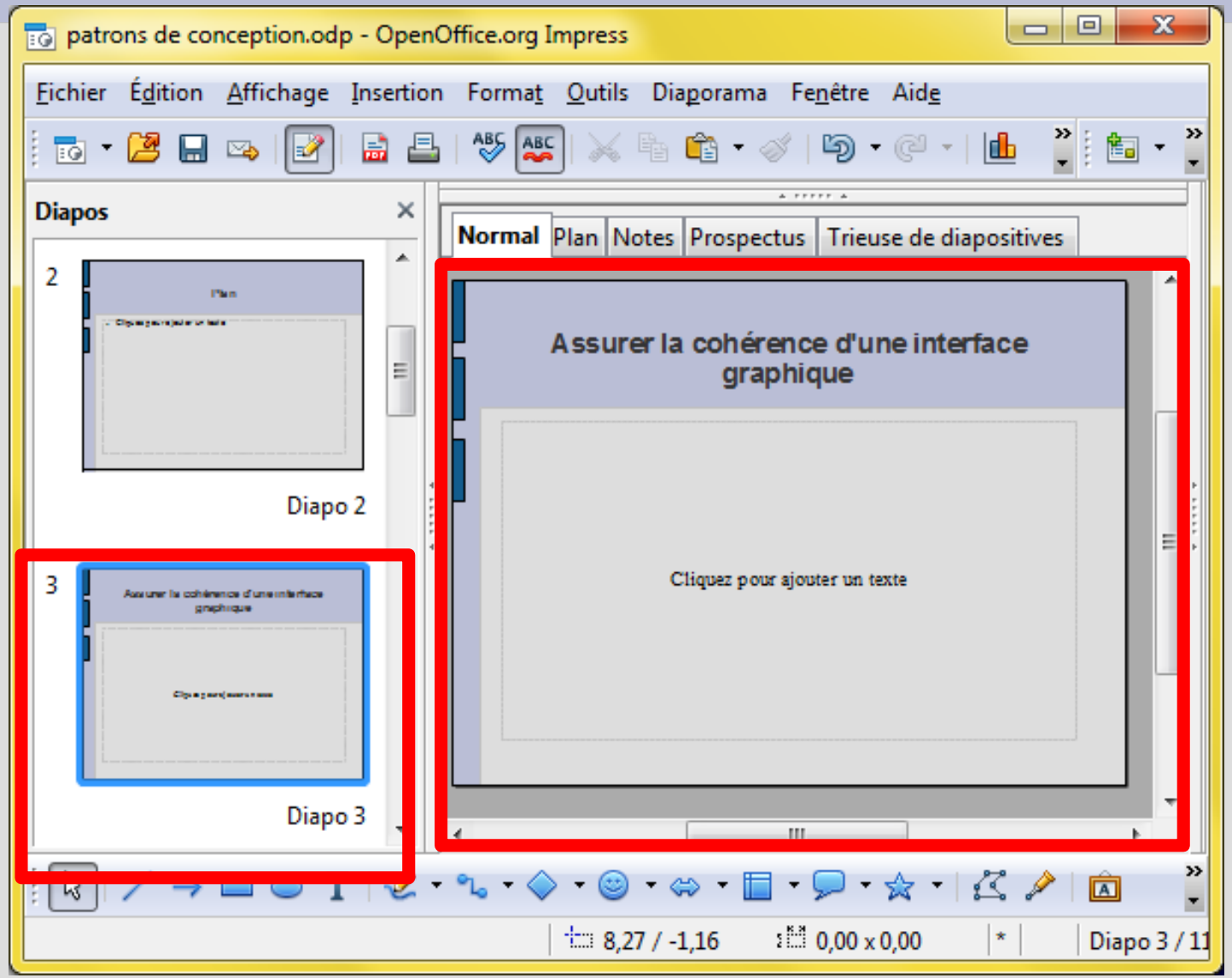
- Interface graphique (rafraîchissement, souris, clavier...)
- Conversion de données
- Pouvoir annuler des commandes
- Implémenter plusieurs algorithmes

Rafraîchissement de l'interface graphique : le patron « observateur »



Source : interface de Windows 7

Rafraîchissement de l'interface graphique



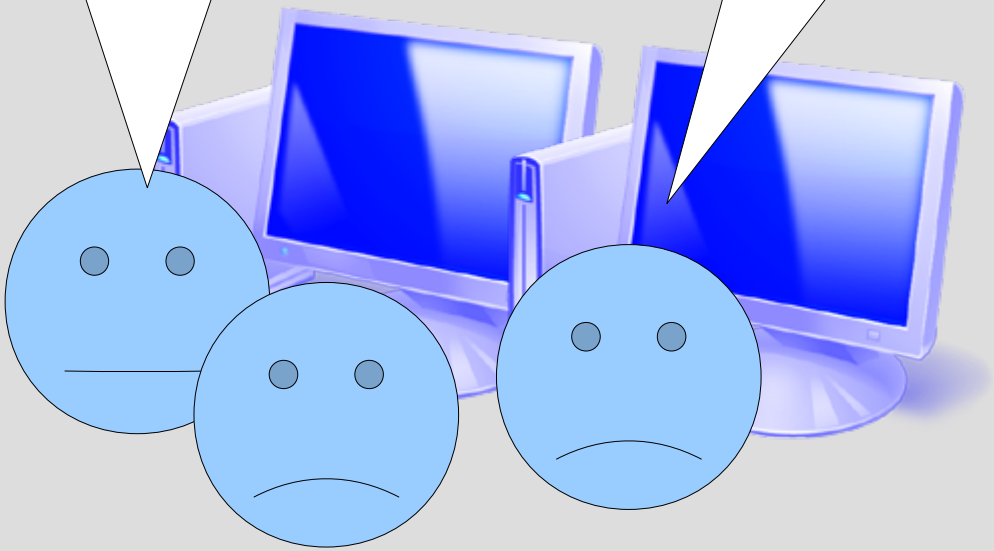
Source : OpenOffice.org Impress

Patron de conception : observateur

On veut assurer la cohérence de l'interface graphique.

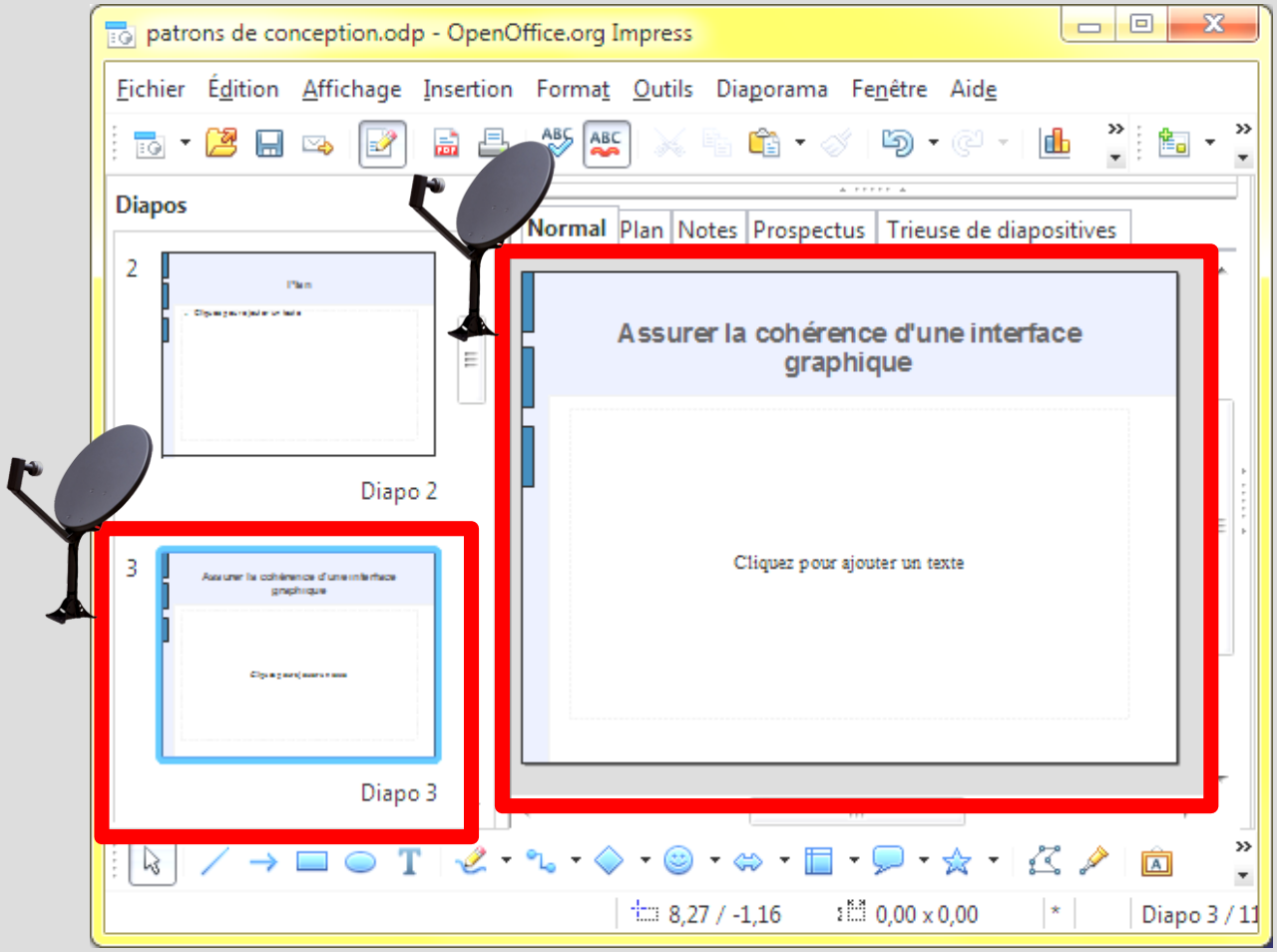
Comment faire ?

On applique le patron de conception « observateur ».



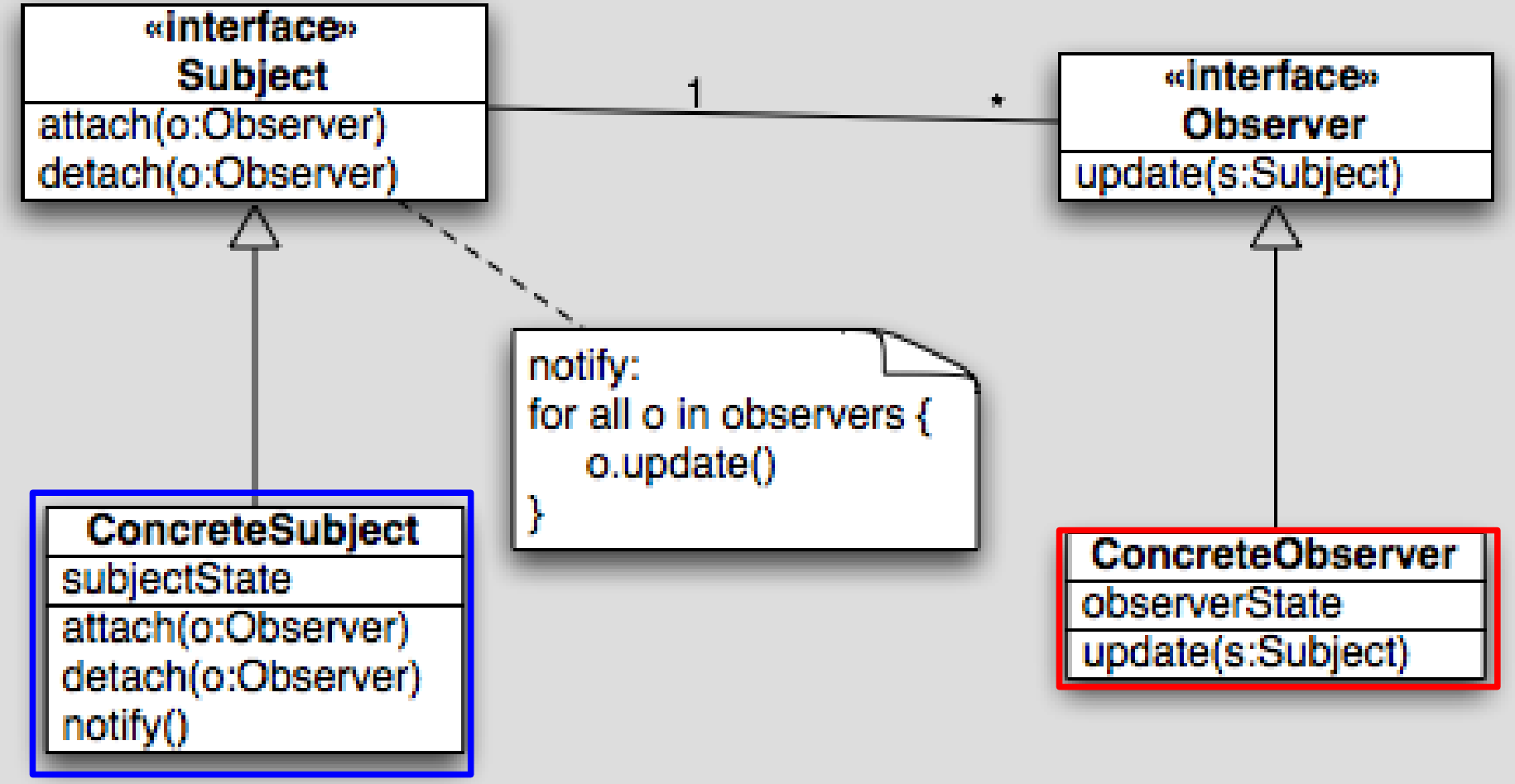
Fonctionnement du patron « observateur » : les deux zones de l'écran observent les données.

Données

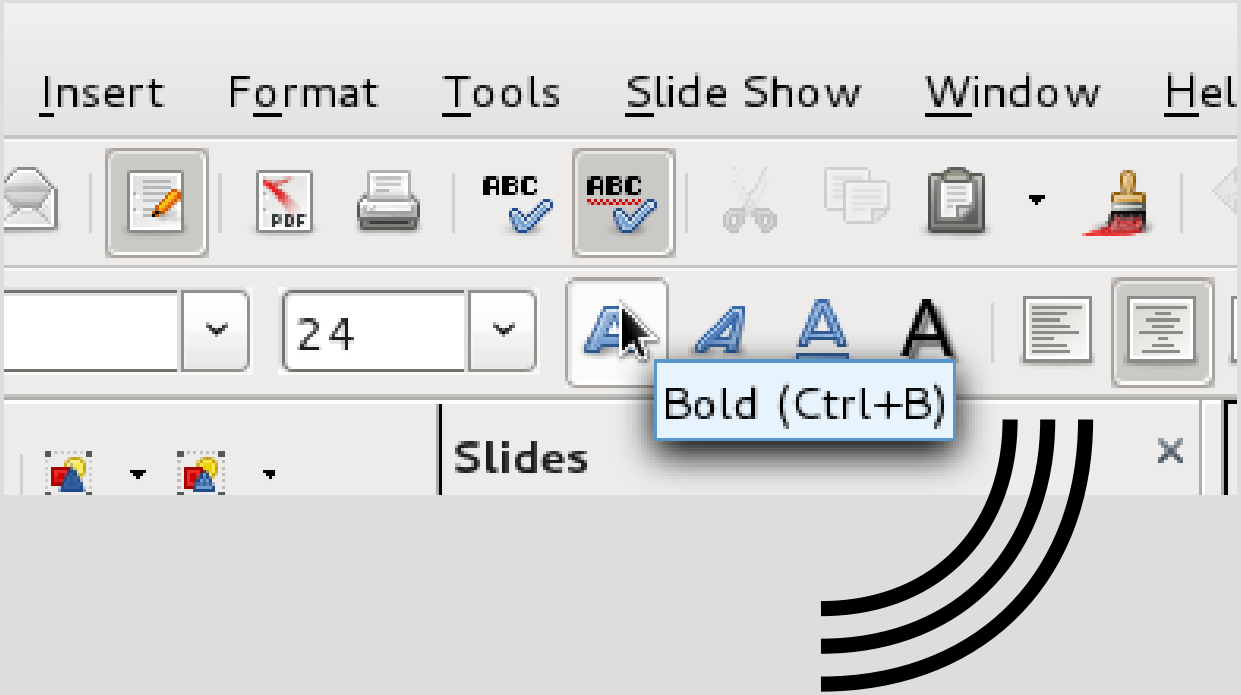


Ces deux zones de l'écran (observateurs) **écoutent** les données (sujets).

Diagramme de classe du modèle Observateur

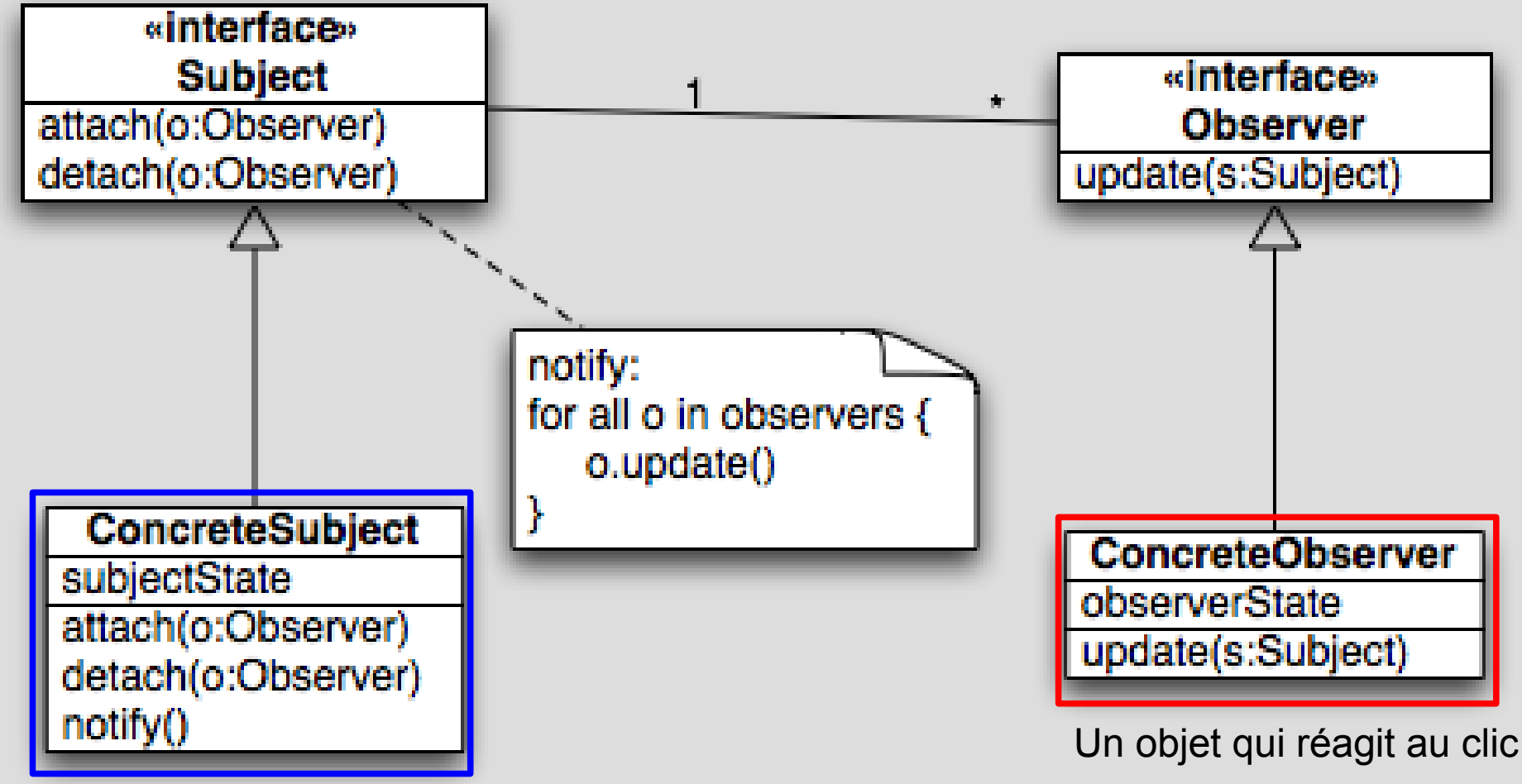


Autre application de « Observateur » : gestion des entrées/sorties



objet qui écoute et effectue une action

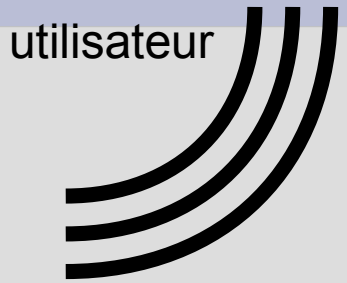
Autre application de « Observateur » : gestion des événements souris en JAVA



Un bouton par exemple

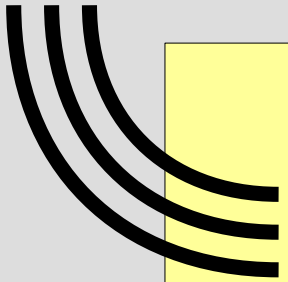
Un objet qui réagit au clic

Modèle-vue-contrôleur



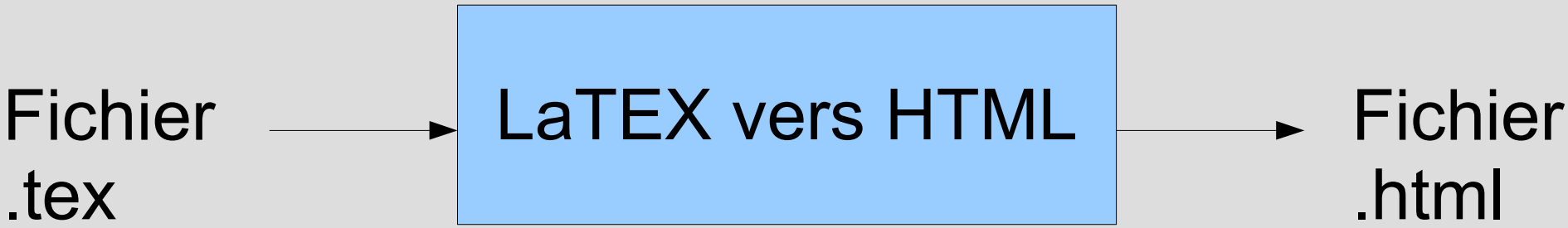
contrôleur

vue

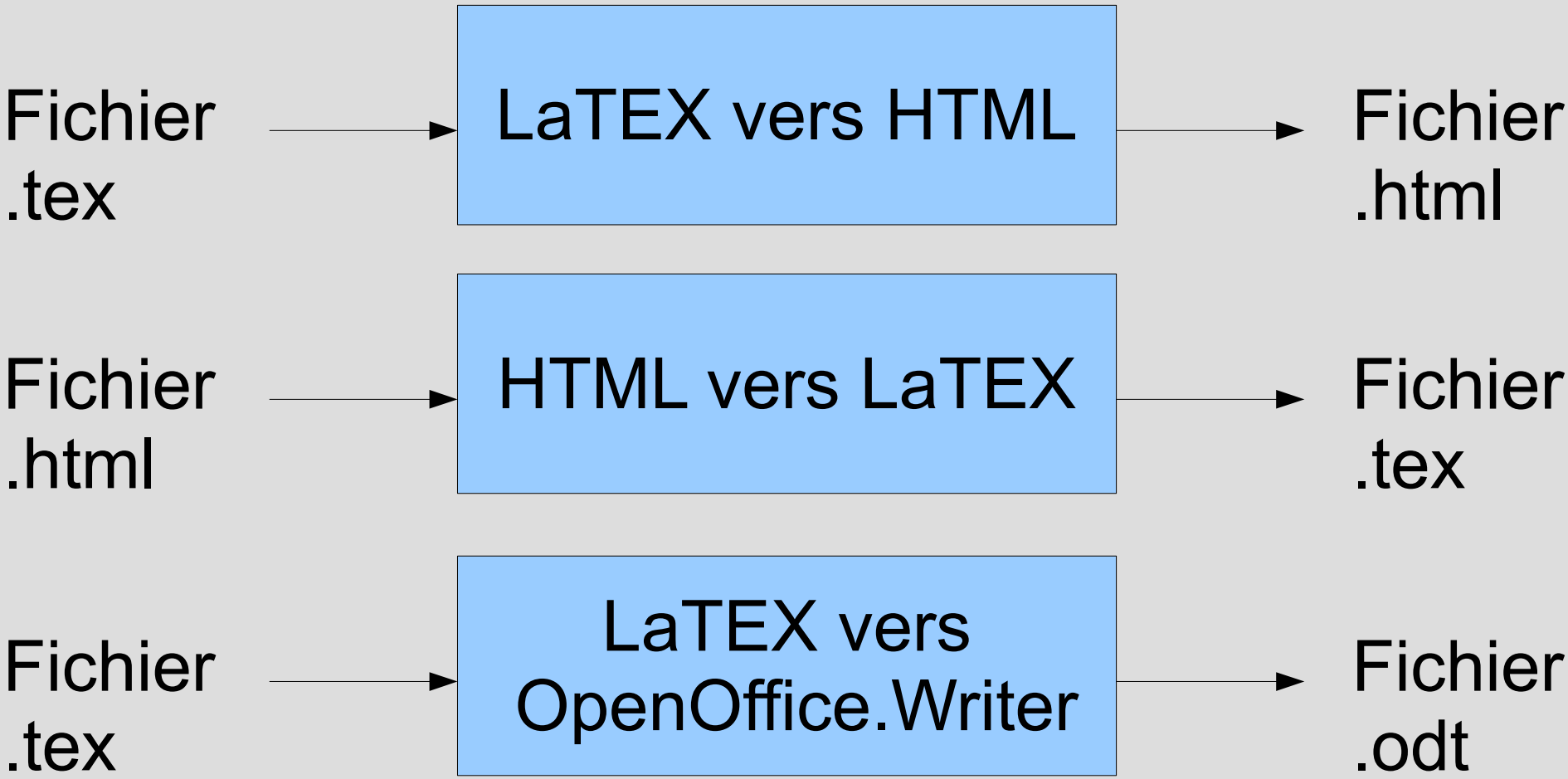


modèle

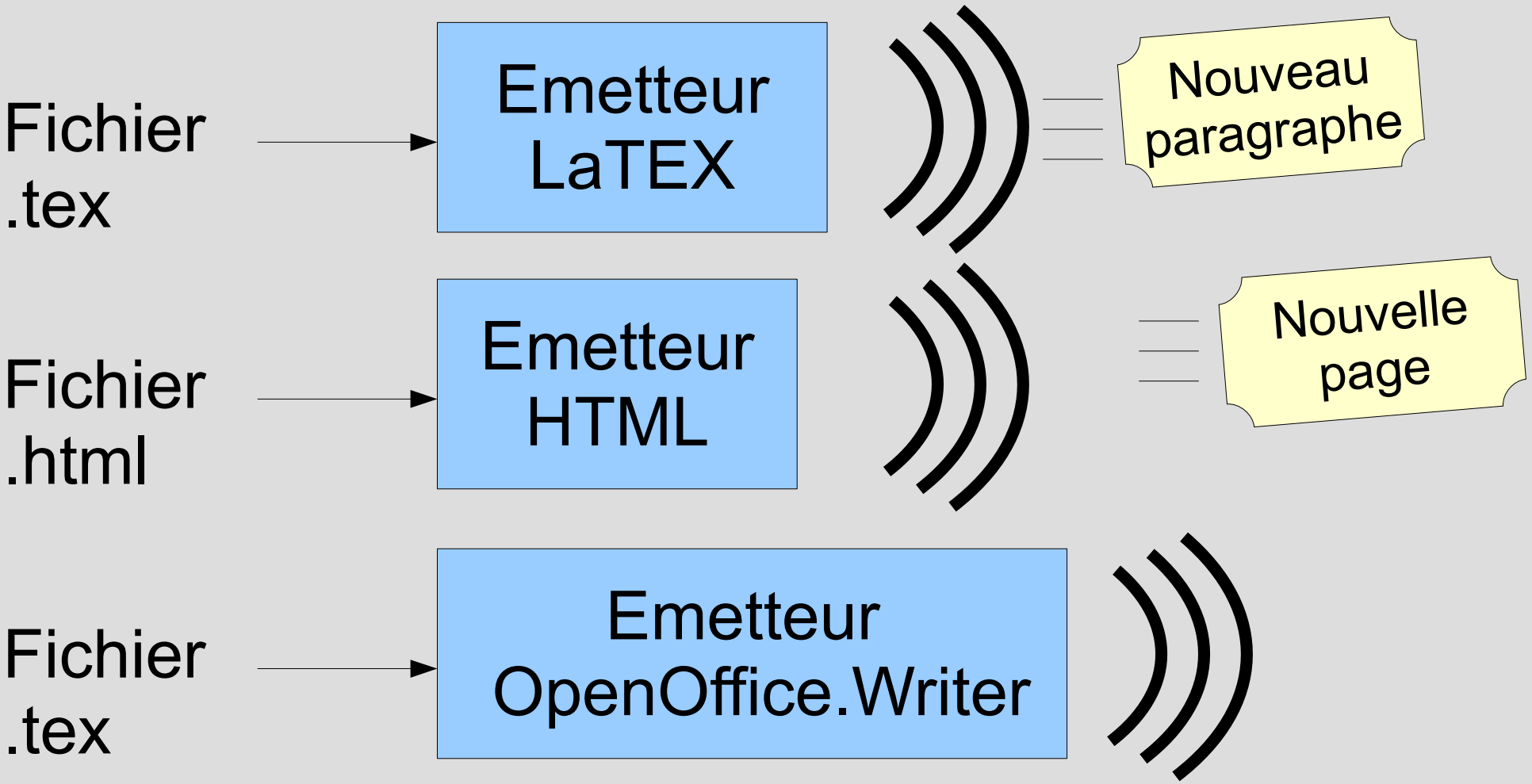
Autre application de « Observateur » : conversions de données d'un format vers un autre



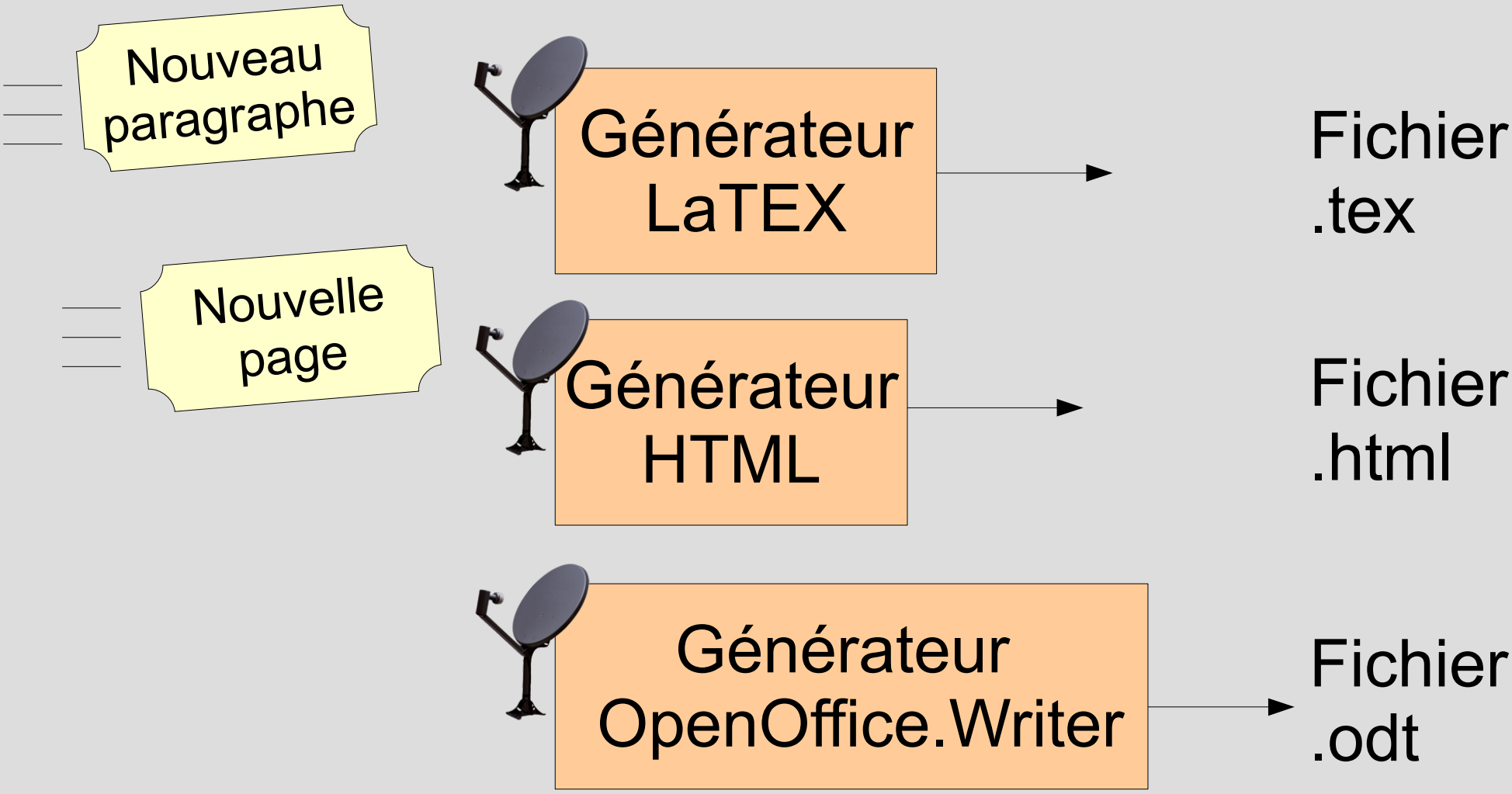
Autre application de « Observateur » : conversions de données d'un format vers un autre



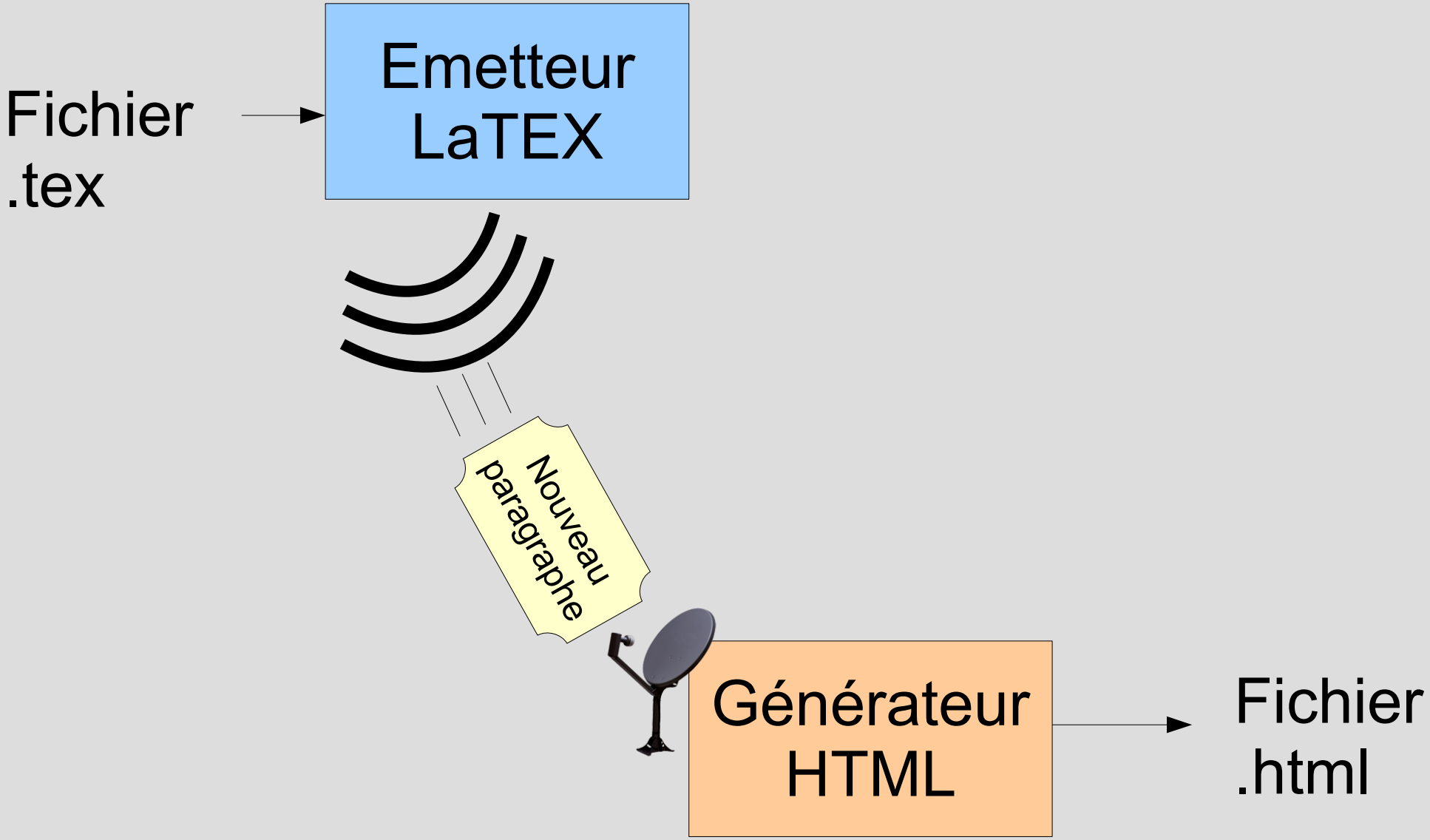
Autre application de « Observateur » : conversions de données d'un format vers un autre



Autre application de « Observateur » : conversions de données d'un format vers un autre



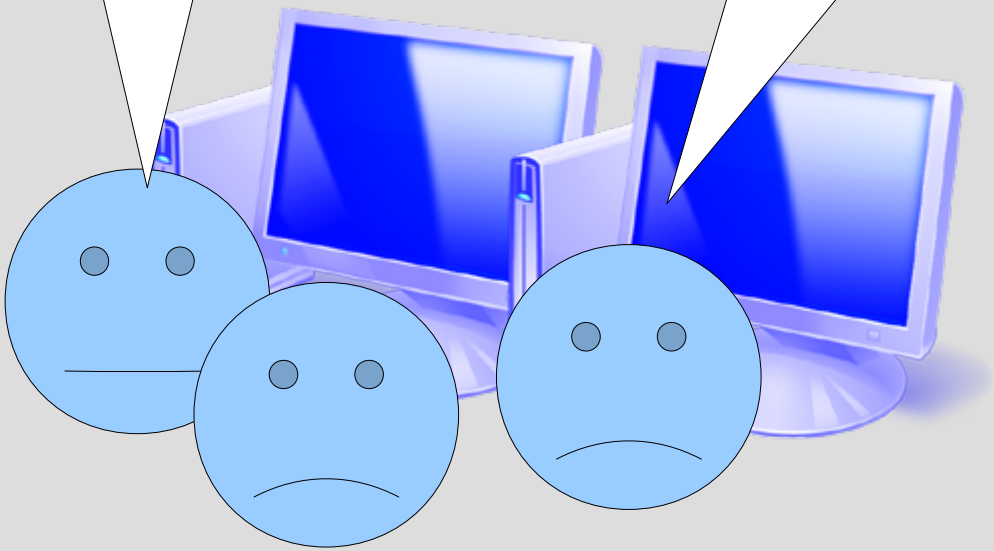
Autre application de « Observateur » : conversion de données d'un format vers un autre



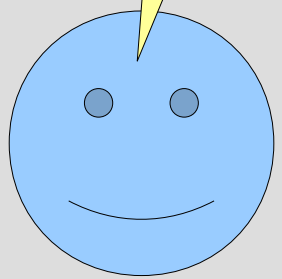
Patron de conception « Commande »

On va devoir mettre
une fonction
« annuler »
dans le logiciel.

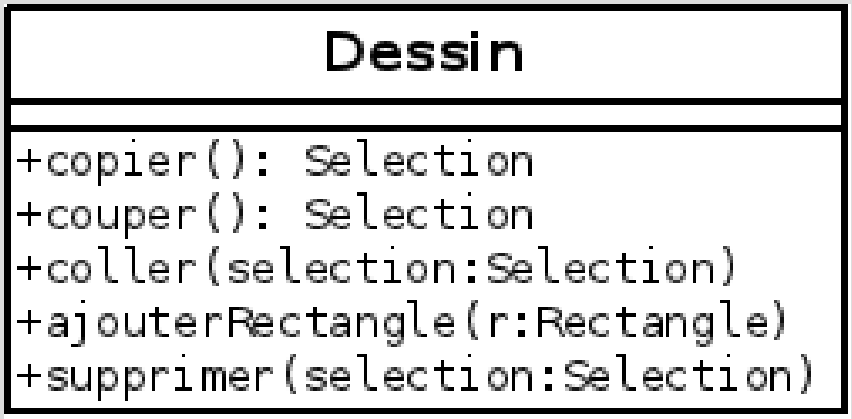
Comment faire ?



On applique
le patron de
conception
« Commande ».

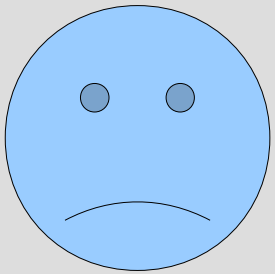


Si on suit le principe « une action = une opération »

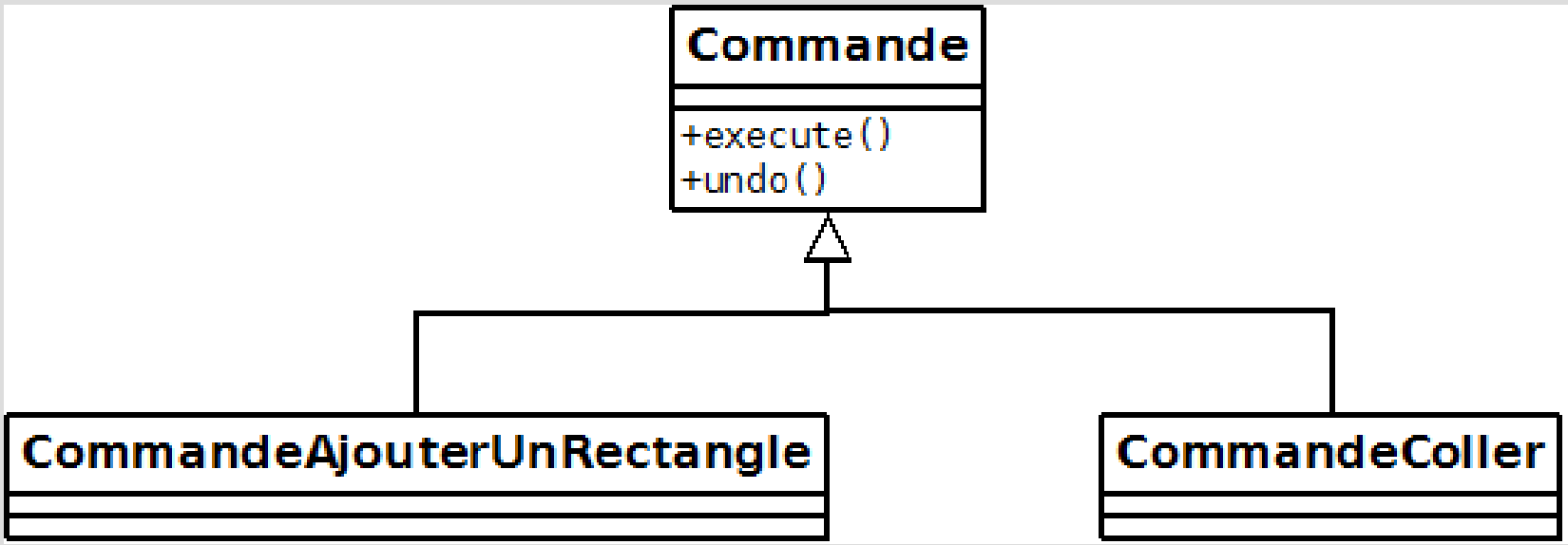


Problèmes

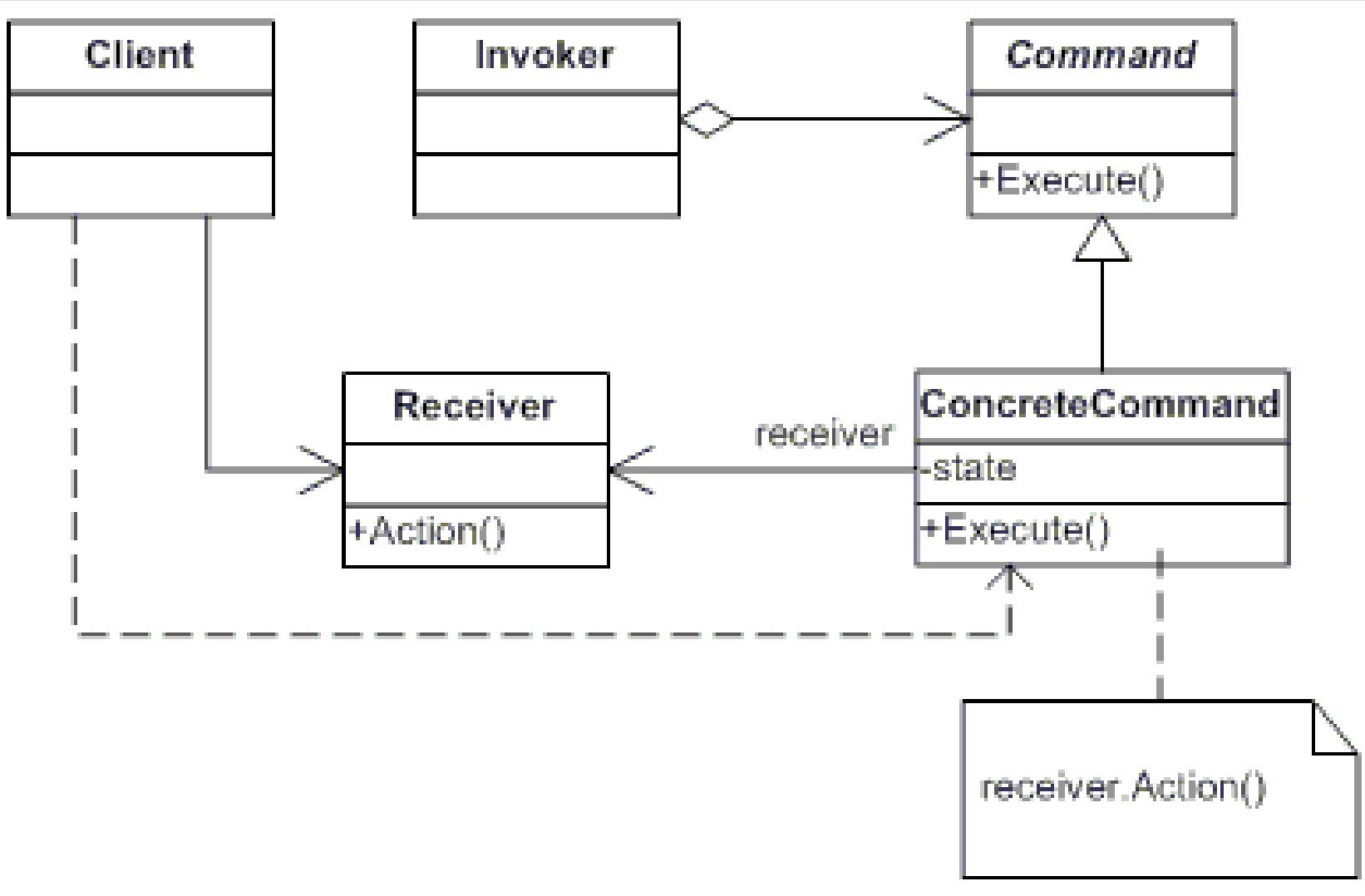
- annuler des commandes ?
- enregistrer des macros ?
- la classe « Dessin » grossit si on rajoute des opérations



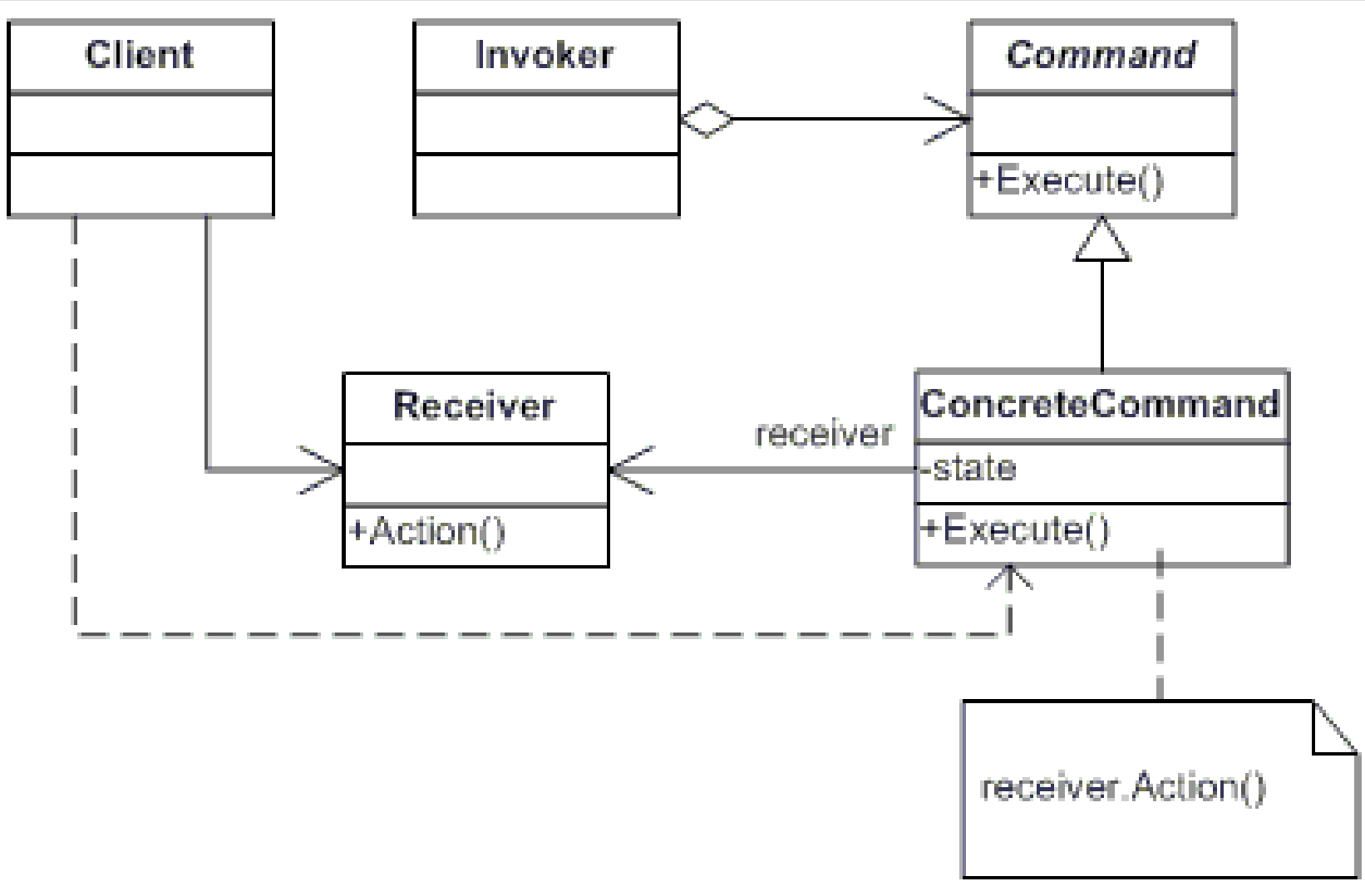
Solution : patron de conception « commande »
Une commande = un objet



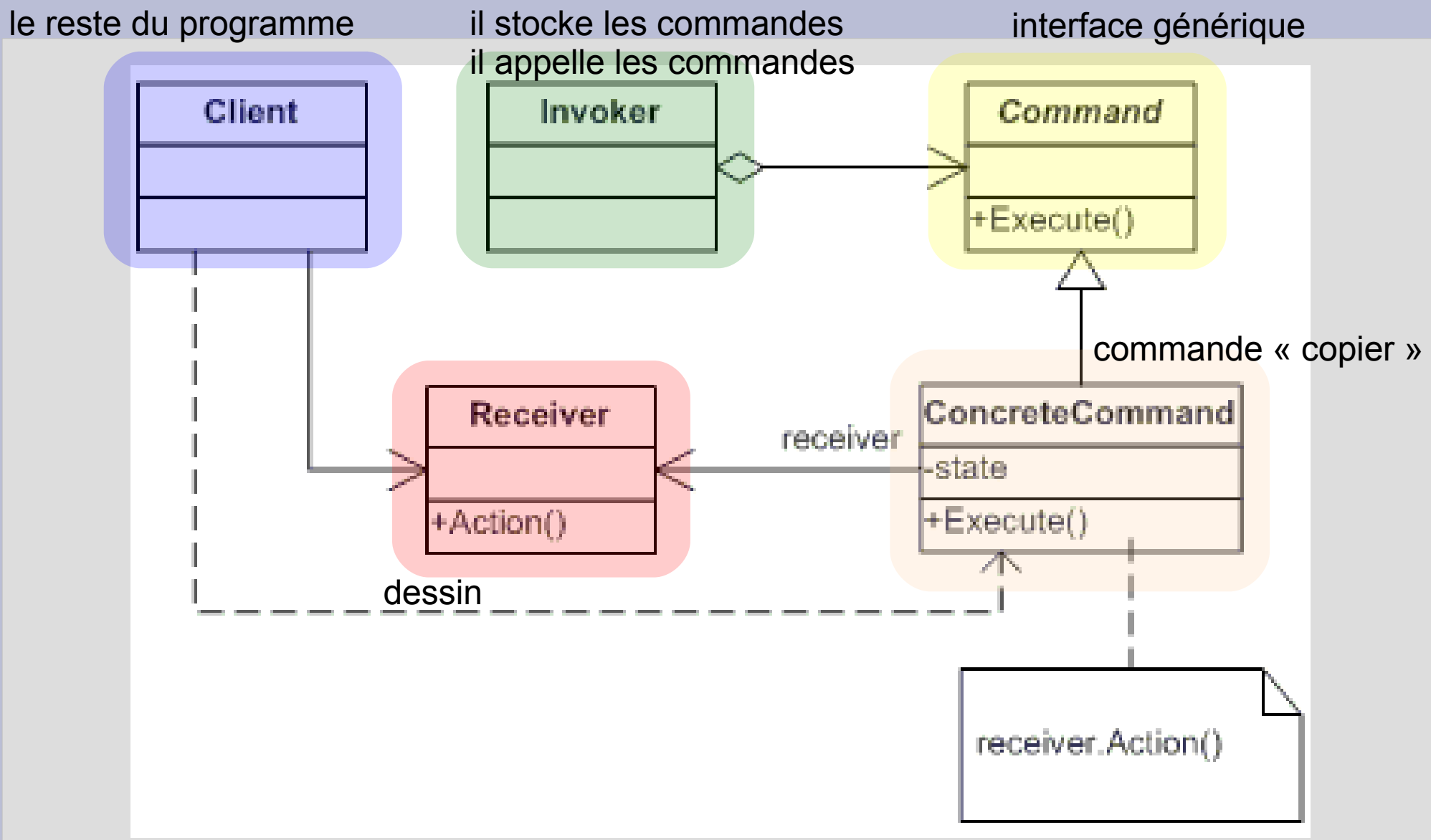
Patron de conception « Commande »
Diagramme de classe



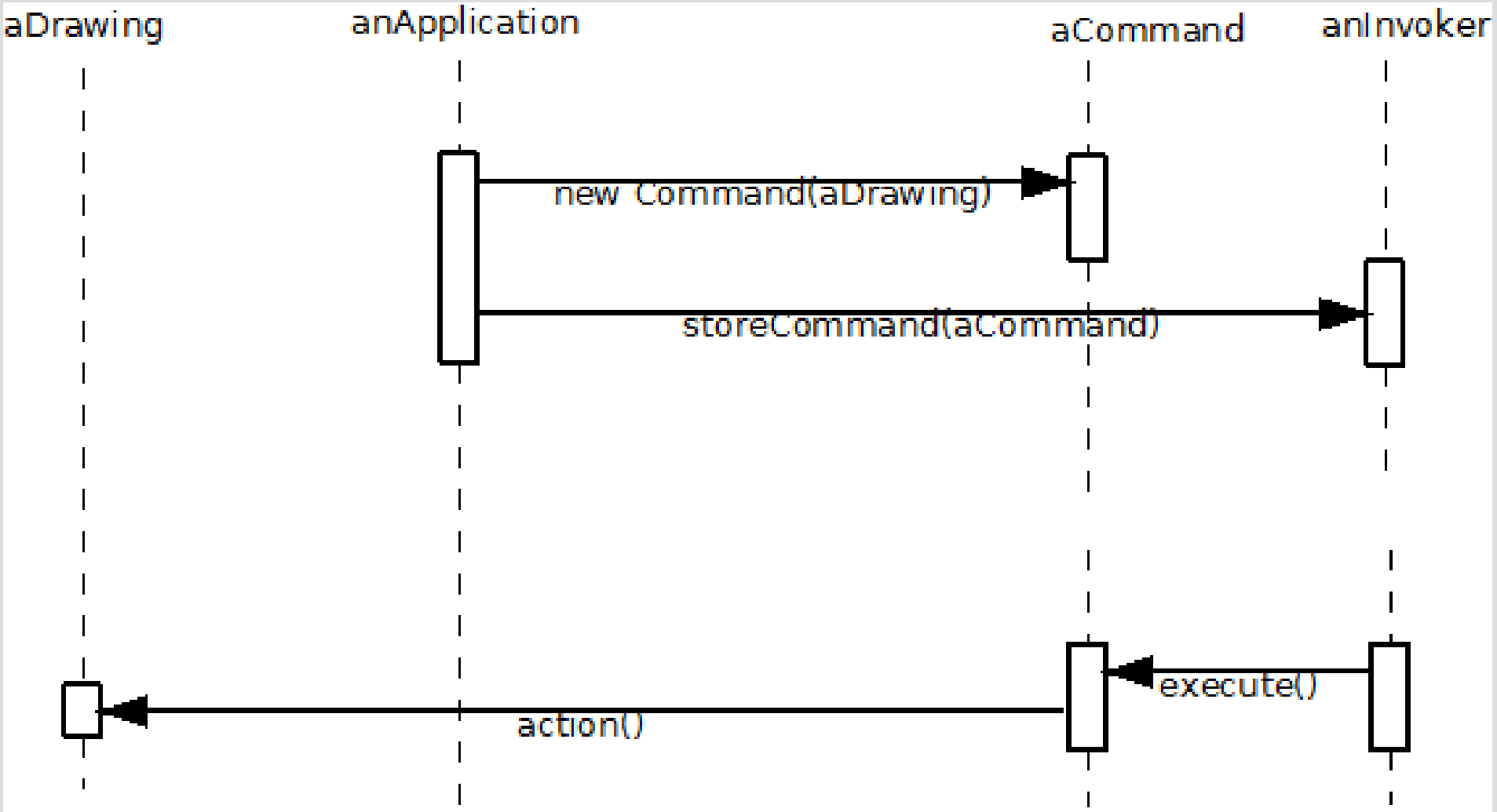
Patron de conception « Commande »
Diagramme de classe



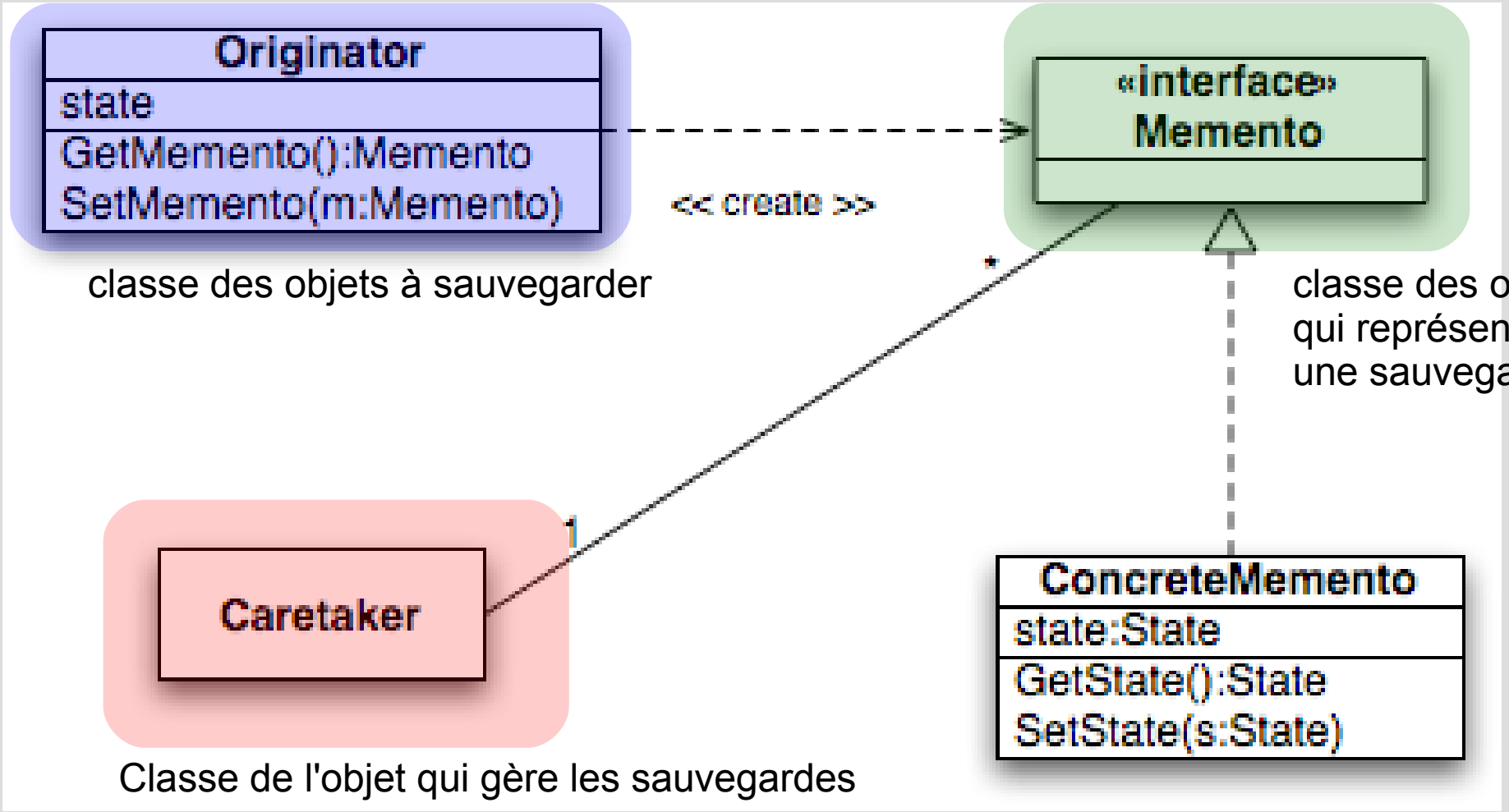
Patron de conception « Commande »
Diagramme de classe



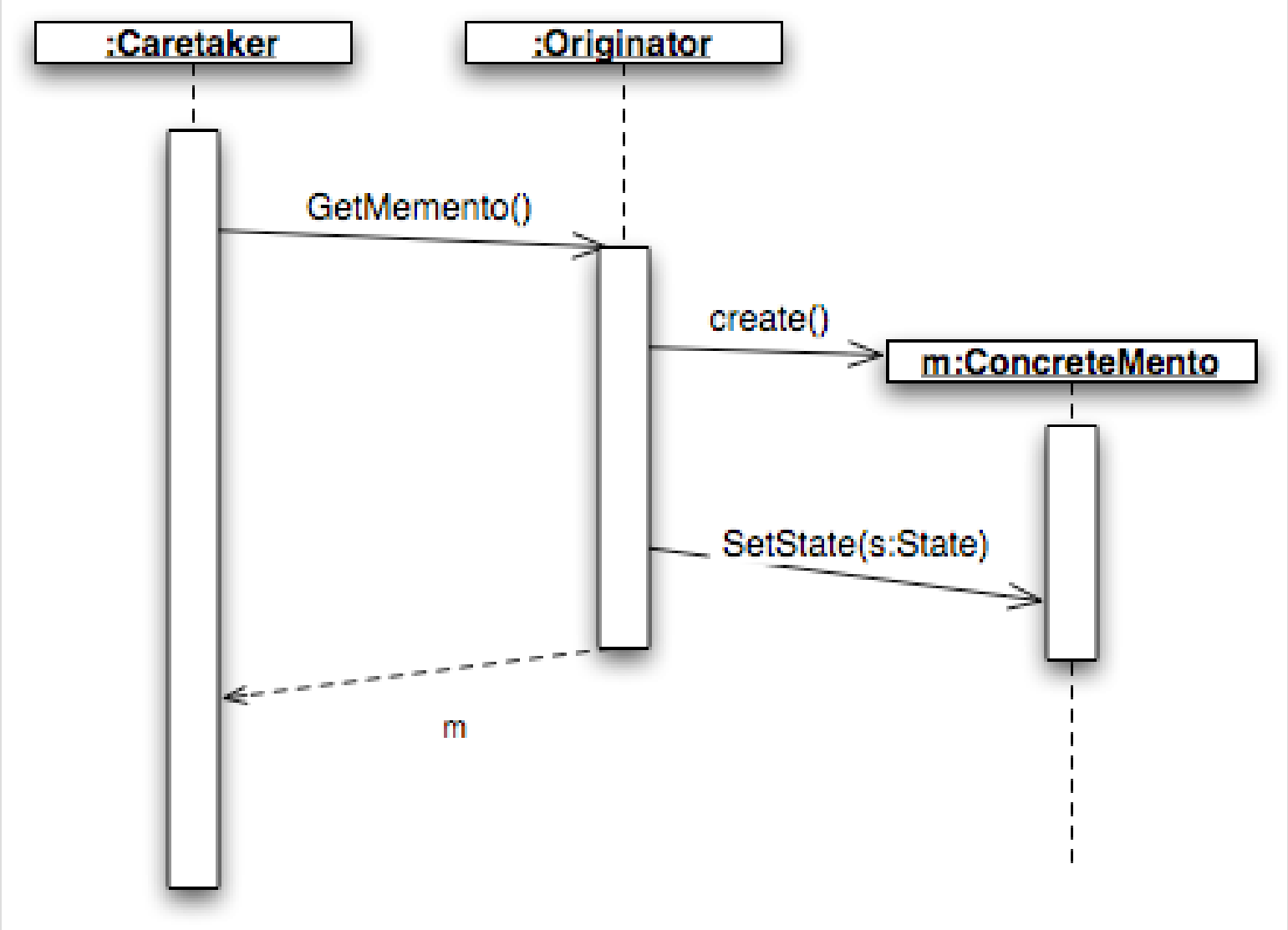
Patron de conception « Commande »
Diagramme de séquence



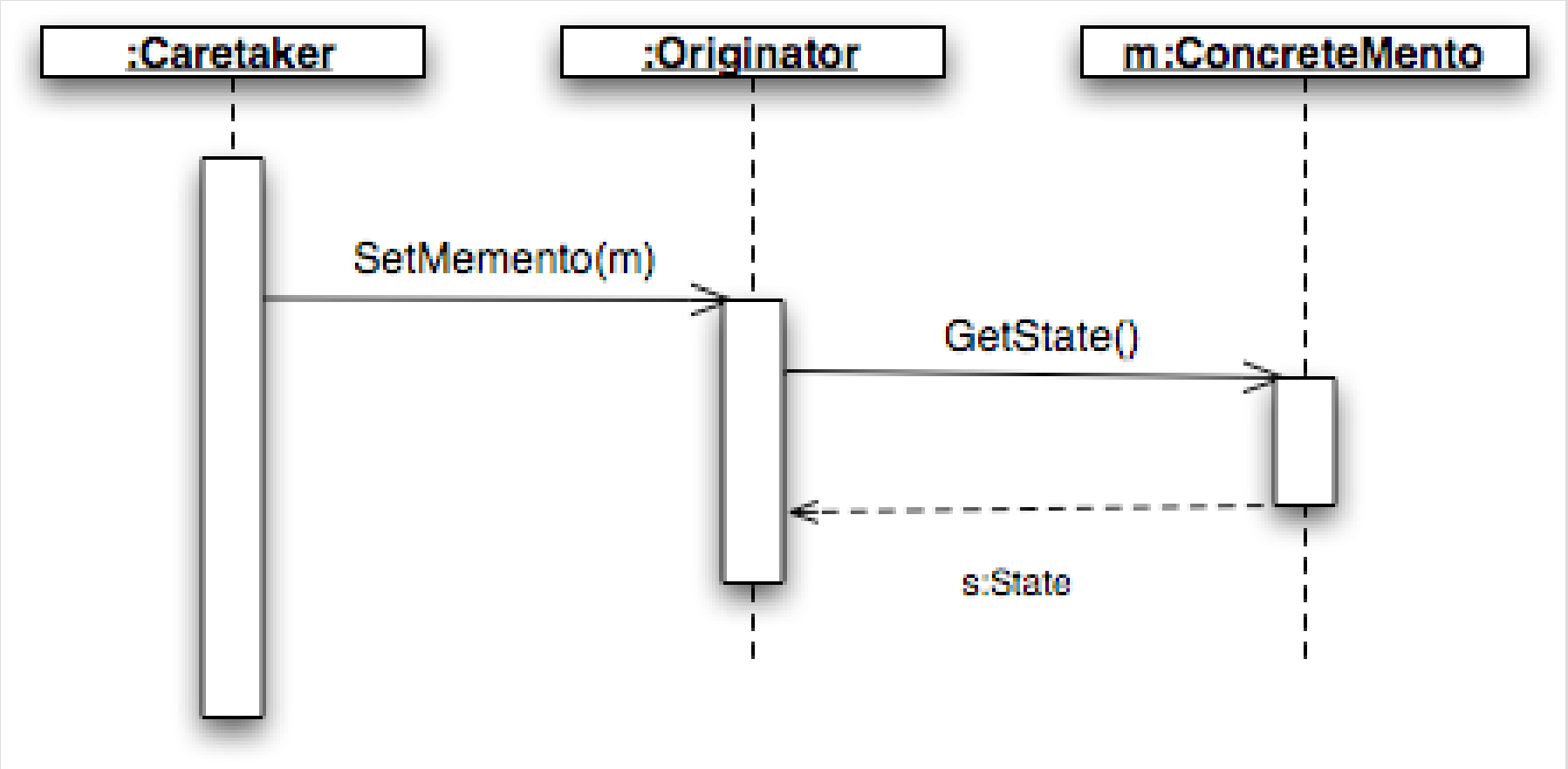
Memento


Originator
state
GetMemento():Memento
SetMemento(m:Memento)
«interface»
Memento
<< create >>
classe des objets à sauvegarder
classe des objets
qui représente
une sauvegarde
*
1
Caretaker
ConcreteMemento
state:State
GetState():State
SetState(s:State)
Classe de l'objet qui gère les sauvegardes

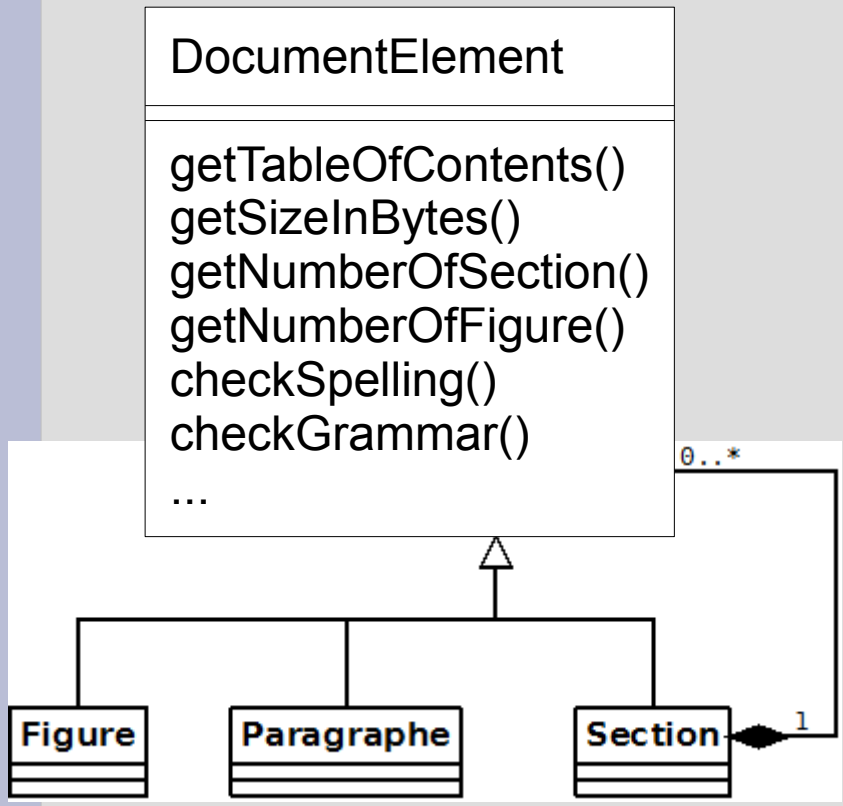
Memento : sauvegarde

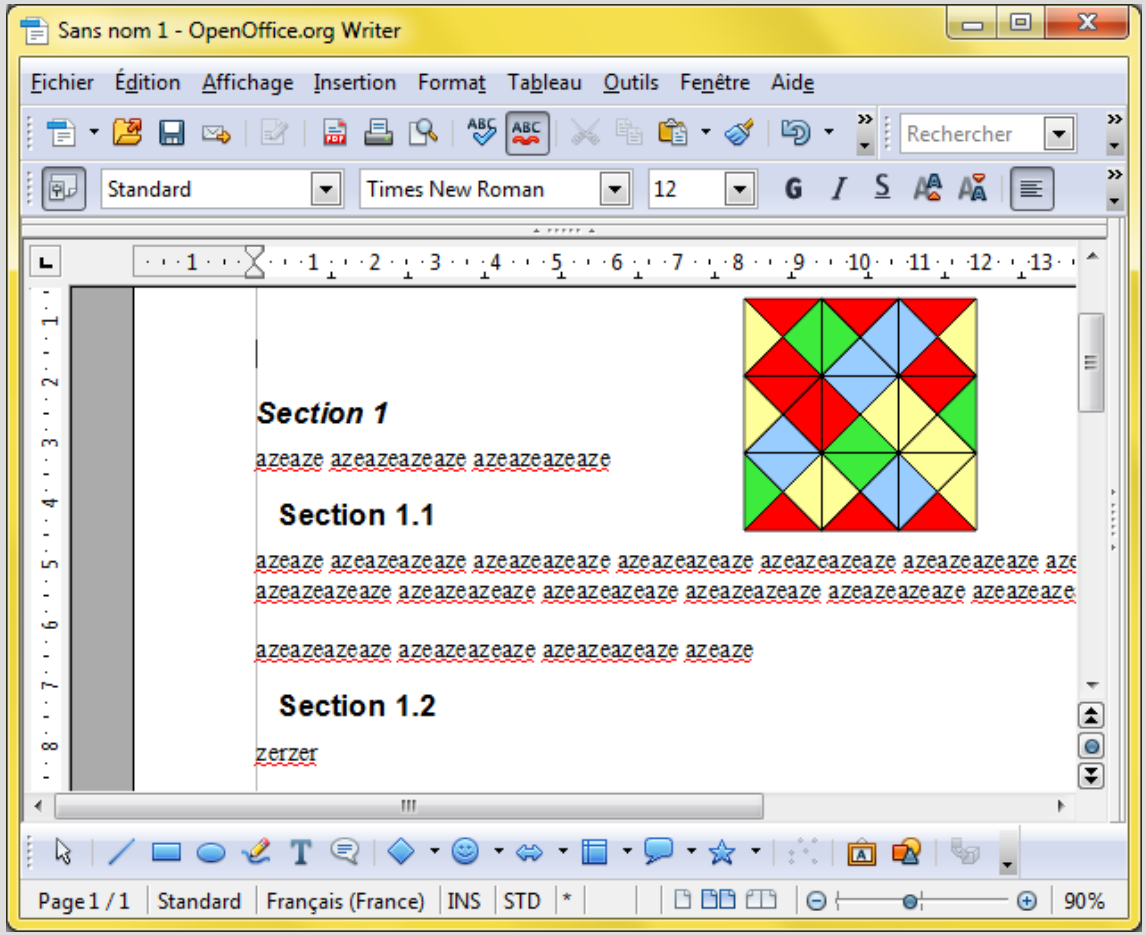


Memento : Restauration



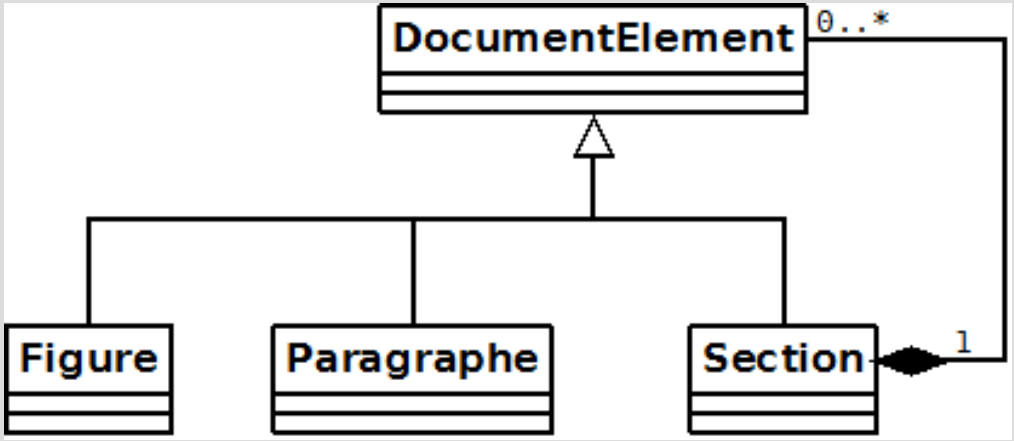
Opérations sur des structures récursives

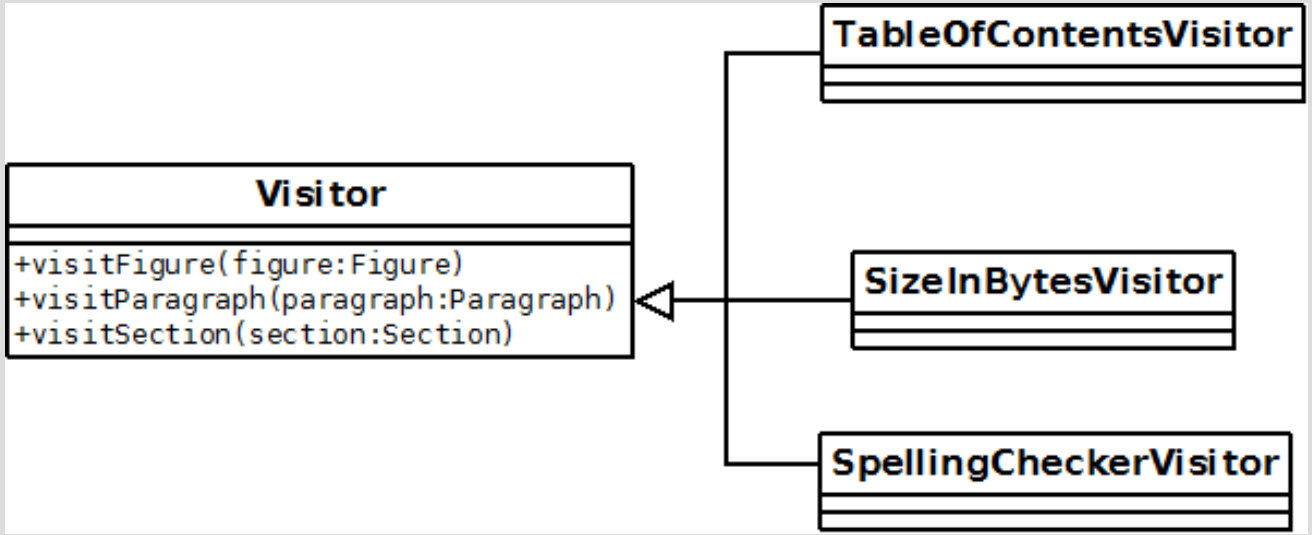




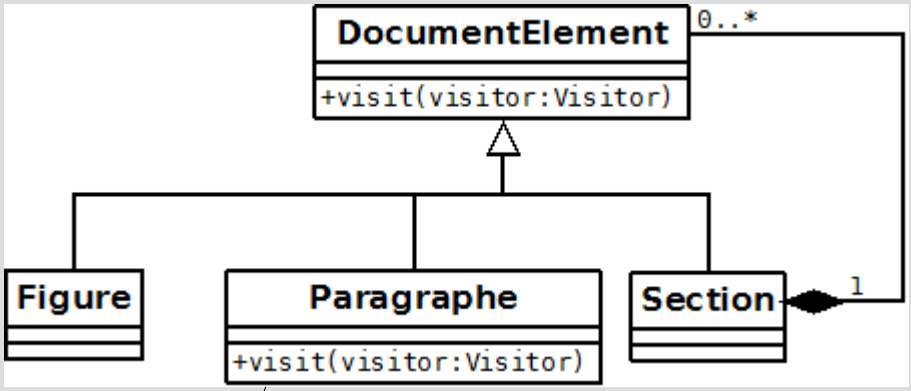
Problème : les classes deviennent **énormes**.

Solution : le patron de conception « Visiteur »

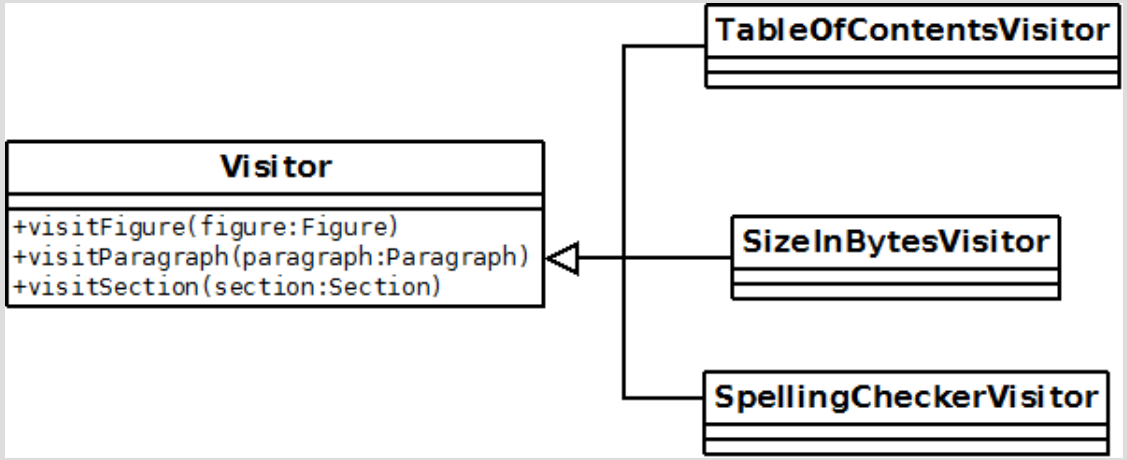




Solution : le patron de conception « Visiteur »




```
public void visit(Visitor visitor)
{
      visitor.visitParagraph(this);
}
```



Avantage du patron « Visiteur »

- Découpe des tâches
- Classes petites

- Dépend des données
- Les classes des données doivent montrer leurs contenus

Visitor

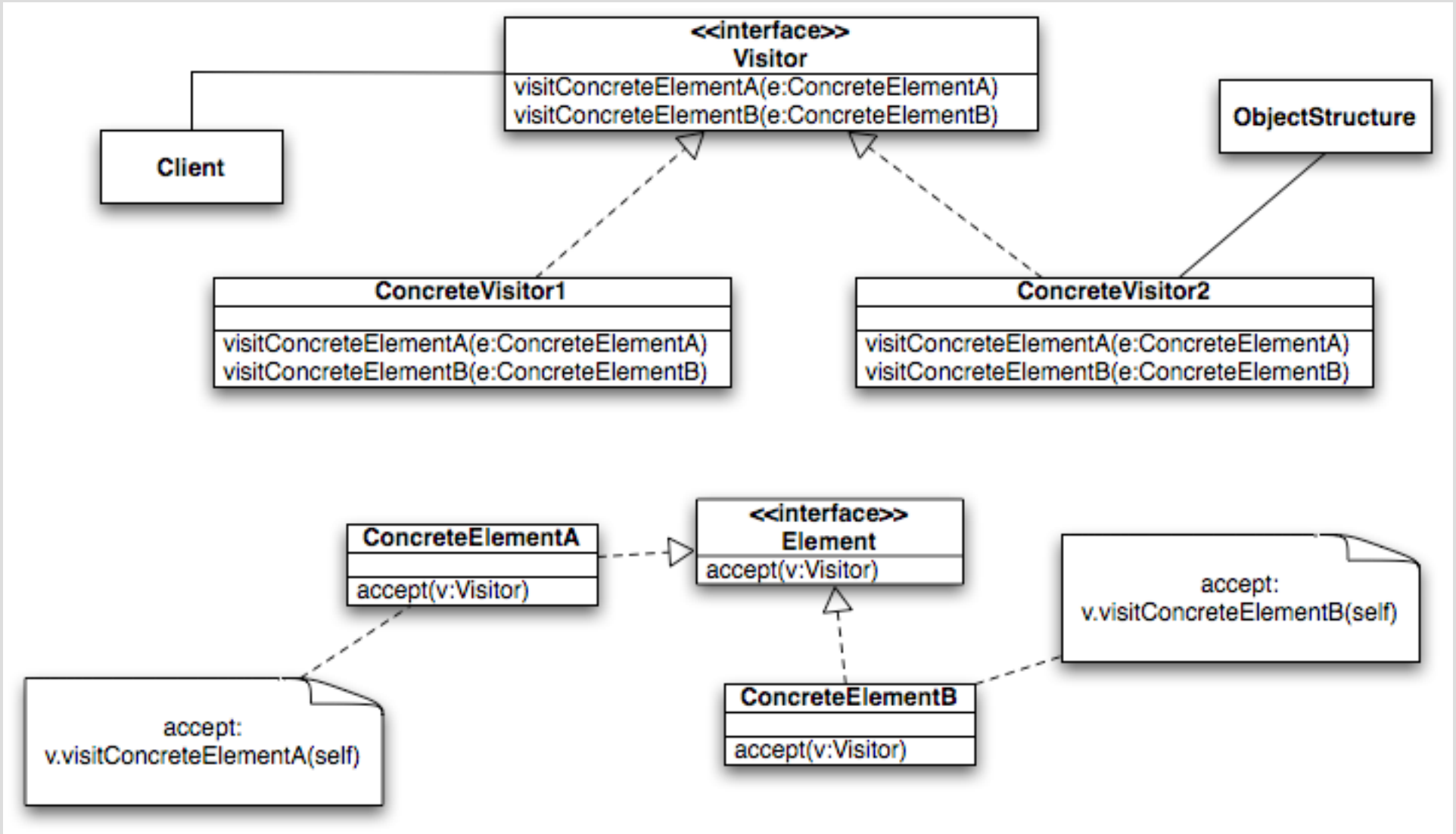
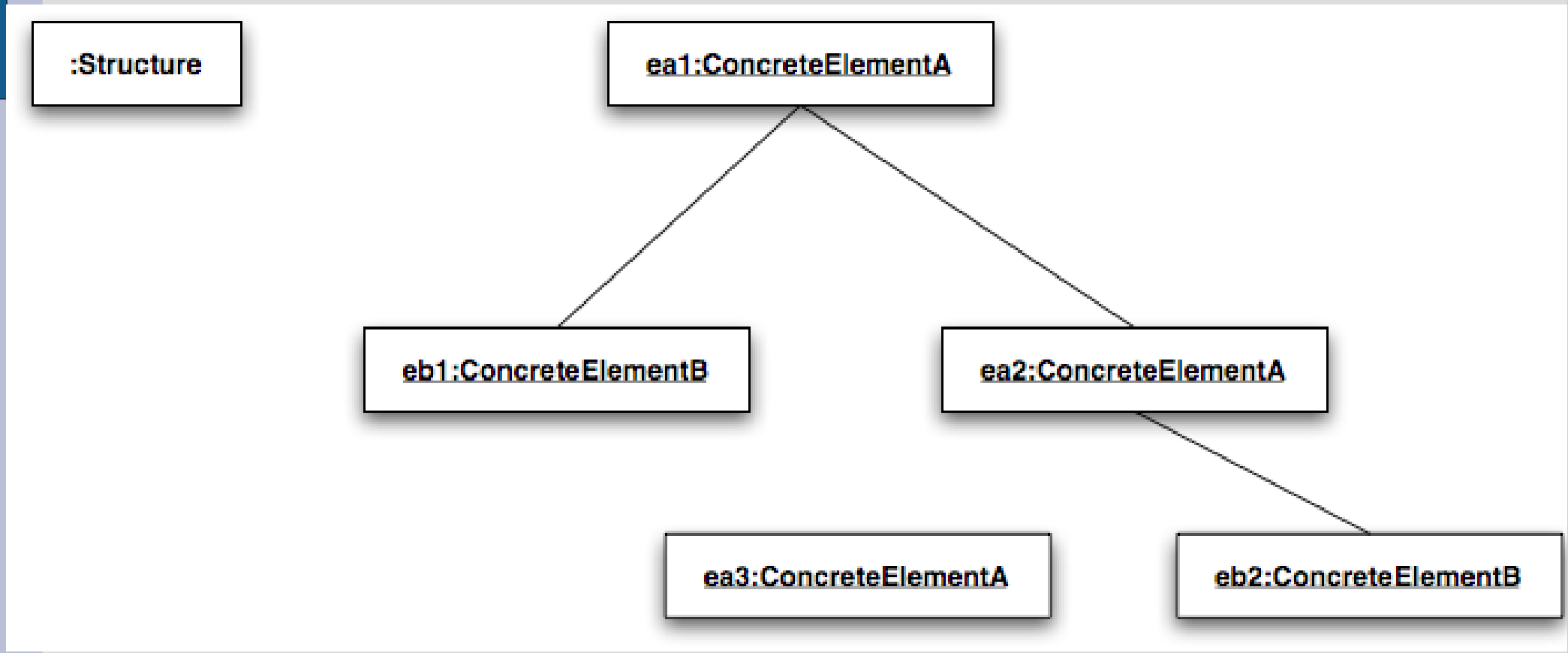
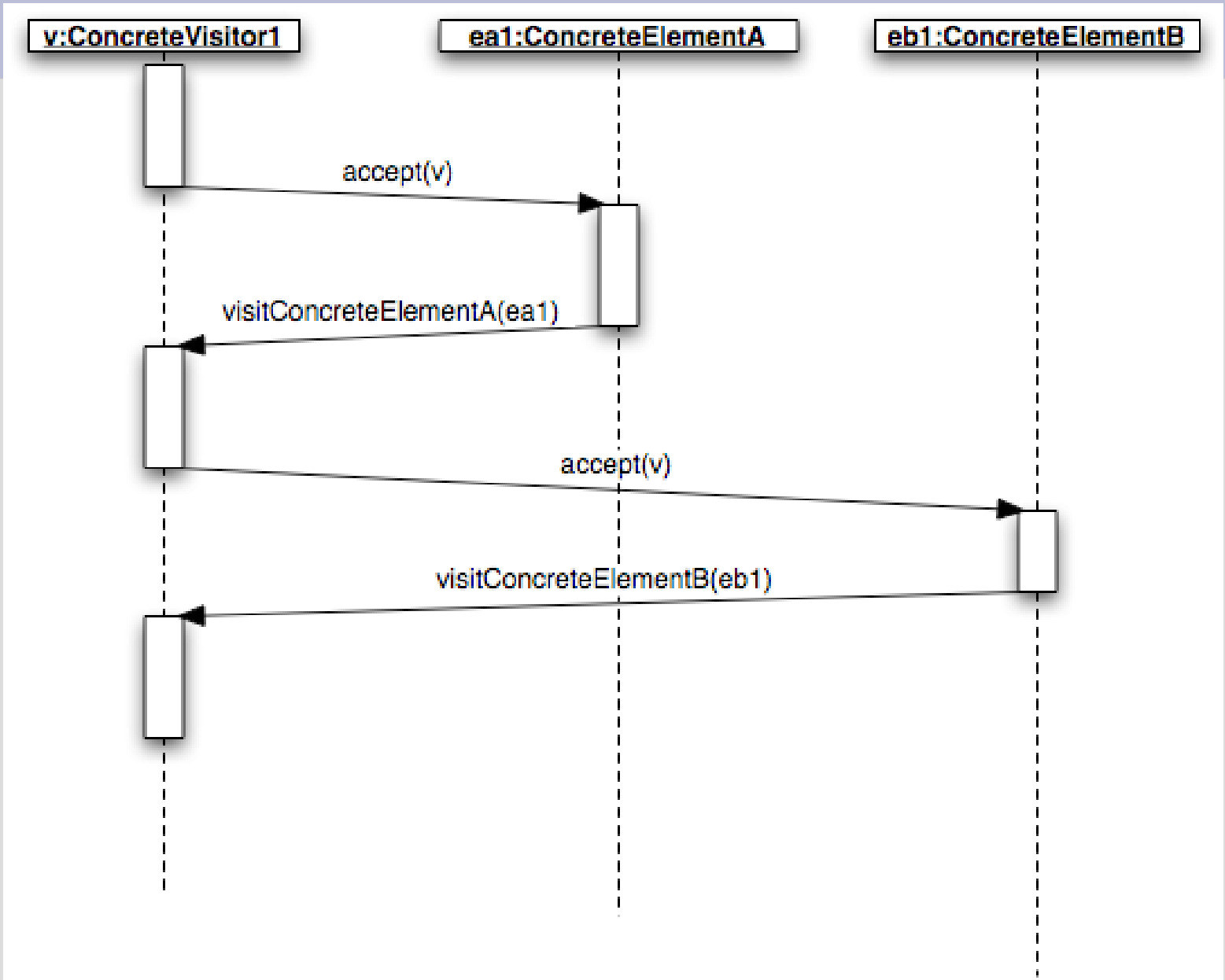

<<interface>>
Visitor
visitConcreteElementA(e:ConcreteElementA)
visitConcreteElementB(e:ConcreteElementB)
Client
ObjectStructure
ConcreteVisitor1
visitConcreteElementA(e:ConcreteElementA)
visitConcreteElementB(e:ConcreteElementB)
ConcreteVisitor2
visitConcreteElementA(e:ConcreteElementA)
visitConcreteElementB(e:ConcreteElementB)
ConcreteElementA
accept(v:Visitor)
<<interface>>
Element
accept(v:Visitor)
accept:
v.visitConcreteElementB(self)
accept:
v.visitConcreteElementA(self)
ConcreteElementB
accept(v:Visitor)

Diagramme d'objets




v:ConcreteVisitor1
ea1:ConcreteElementA
eb1:ConcreteElementB
accept(v)
visitConcreteElementA(ea1)
accept(v)
visitConcreteElementB(eb1)

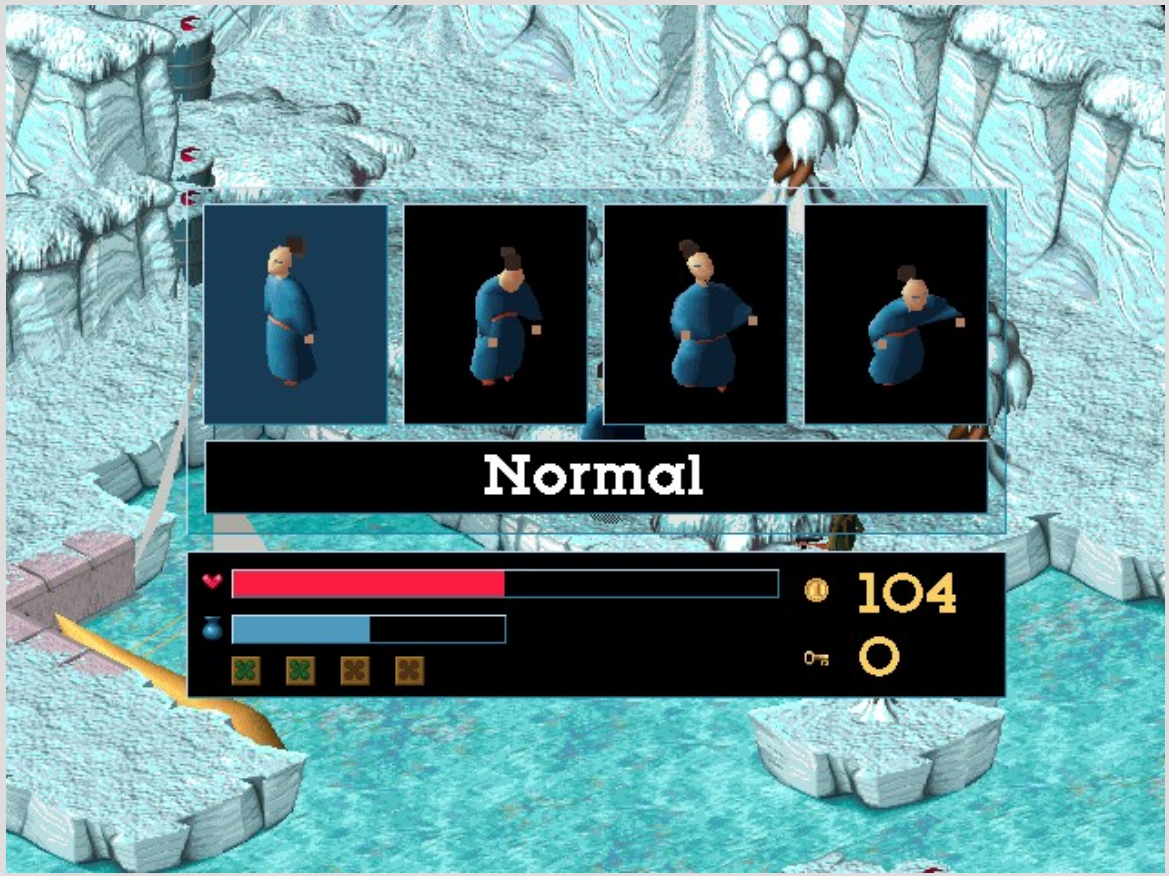
Exemple d'autres applications du patron « Visiteur » sur des structures récursives

- Logiciel de composition musicale :
  nombre de notes, parcourir la partition pour afficher, etc.
- Assistant de preuve :
  parcourir une preuve pour l'afficher, vérifier la preuve, etc.
- Logiciel 3D
  affichage squelette, affichage avec texture, calcul du poids etc.

Implémenter plusieurs algorithmes : patron de conception « Stratégie »

Exemple 1 : s'abstraire d'un algorithme qui manipule le squelette d'un personnage

```
heros.move()
```



Source : Little Big Adventure 2

Exemple 2 : s'abstraire d'un algorithme de tri

```
tableau.sort()
```

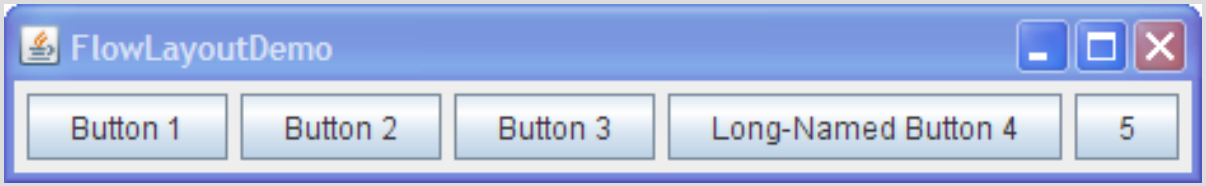
BubbleSort ?

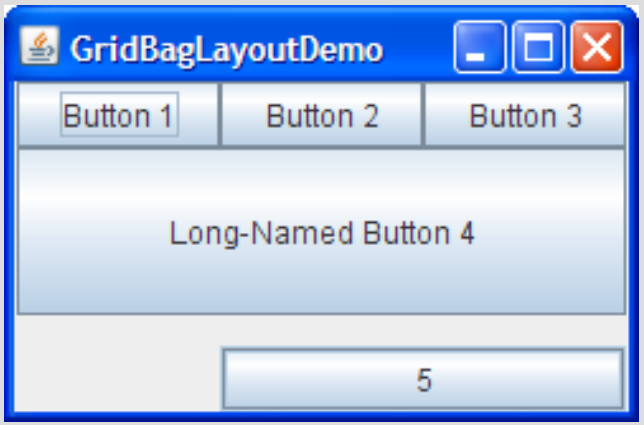
QuickSort ?

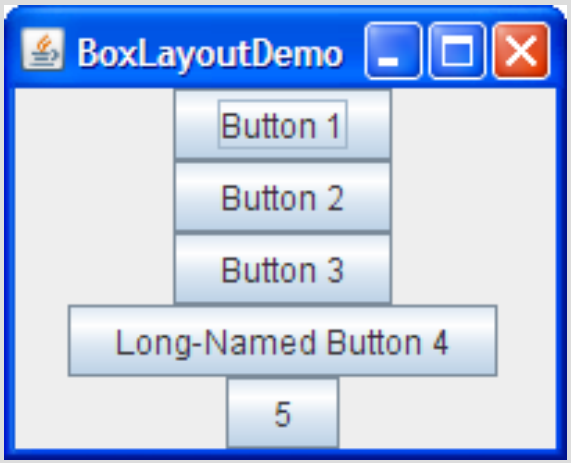
HeapSort ?

Exemple 3 : s'abstraire d'un algorithme de positionnement d'élément graphique à l'écran

```
container.doLayout()
```

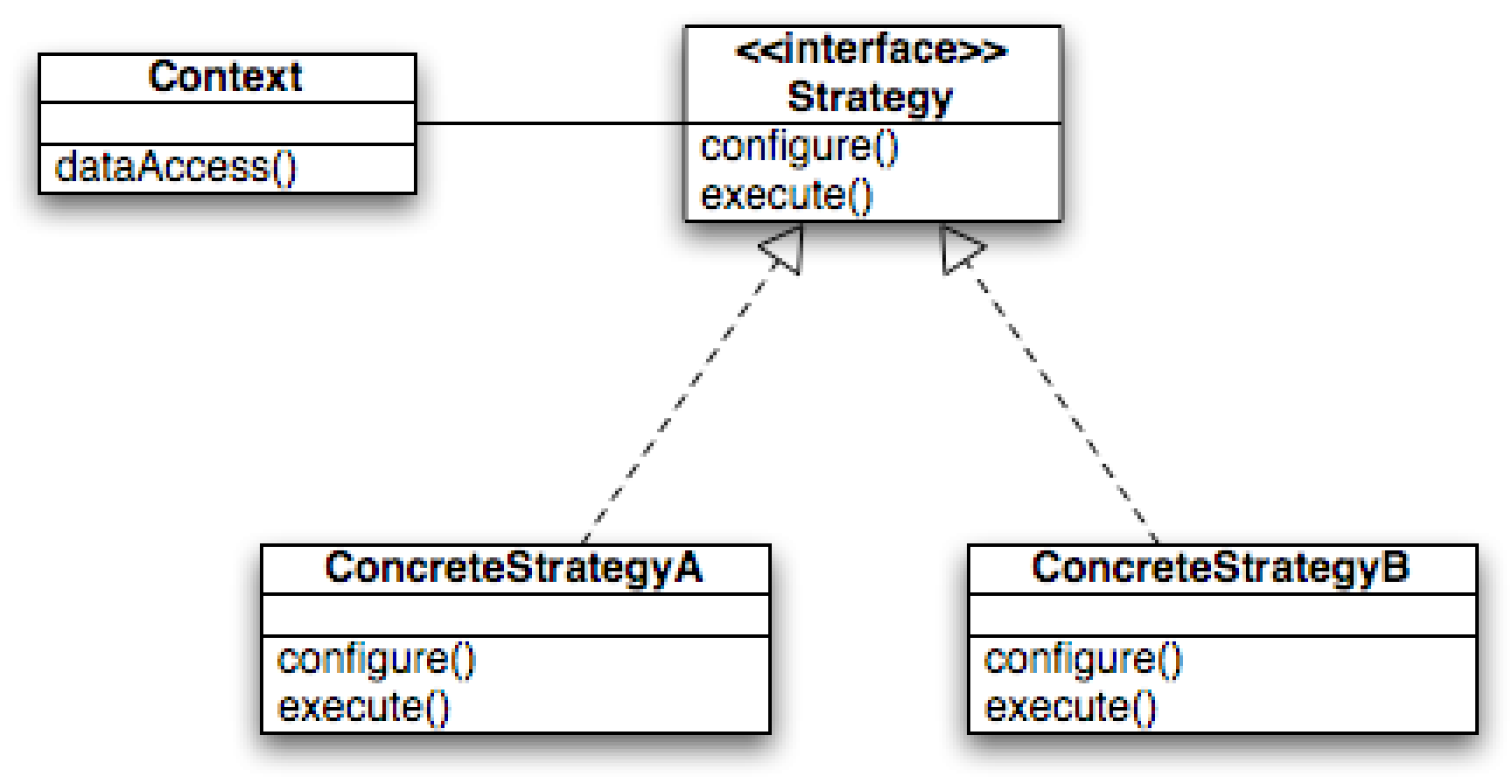




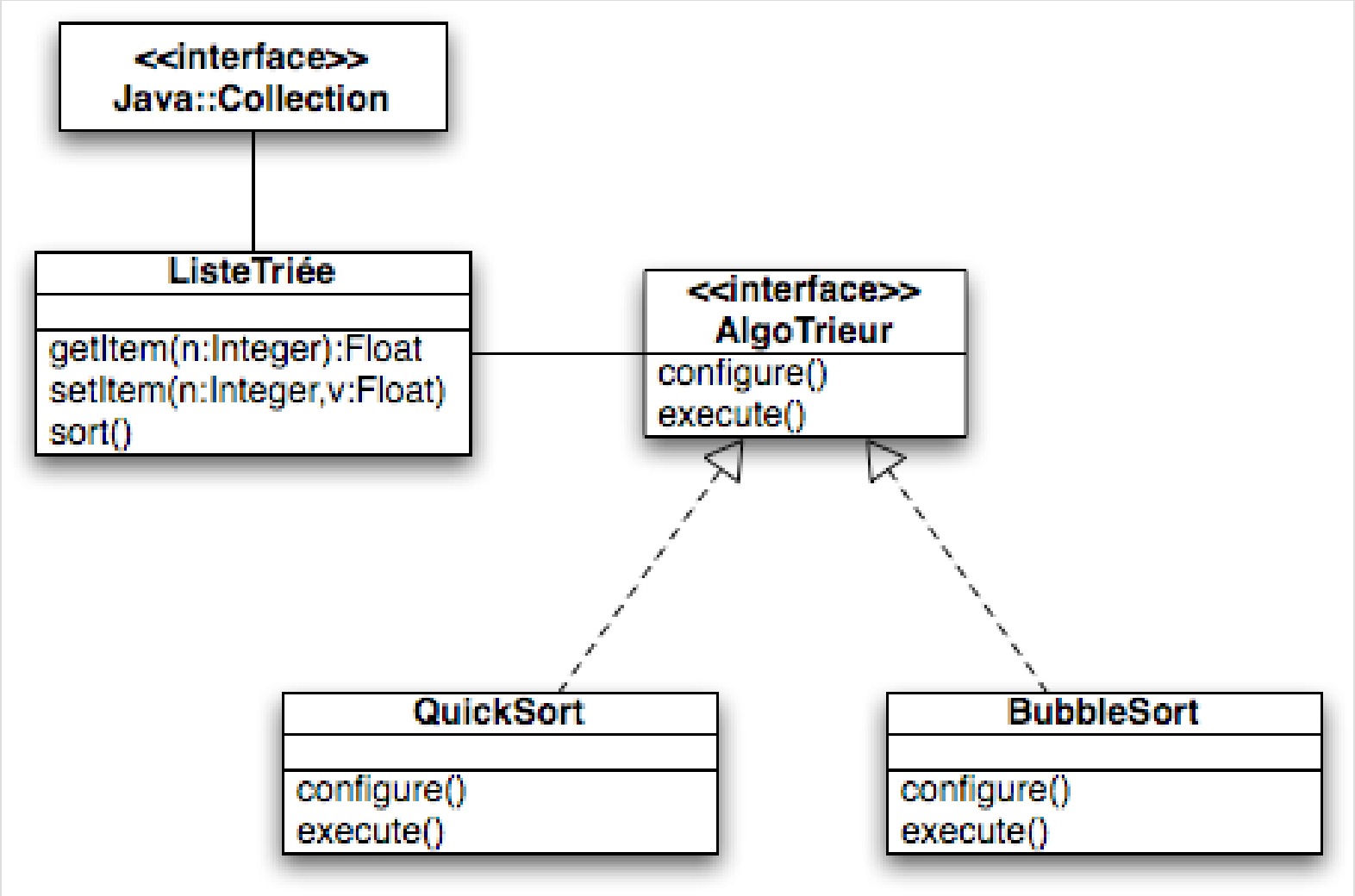


Source : http://download.oracle.com/javase/tutorial/uiswing/layout/visual.html

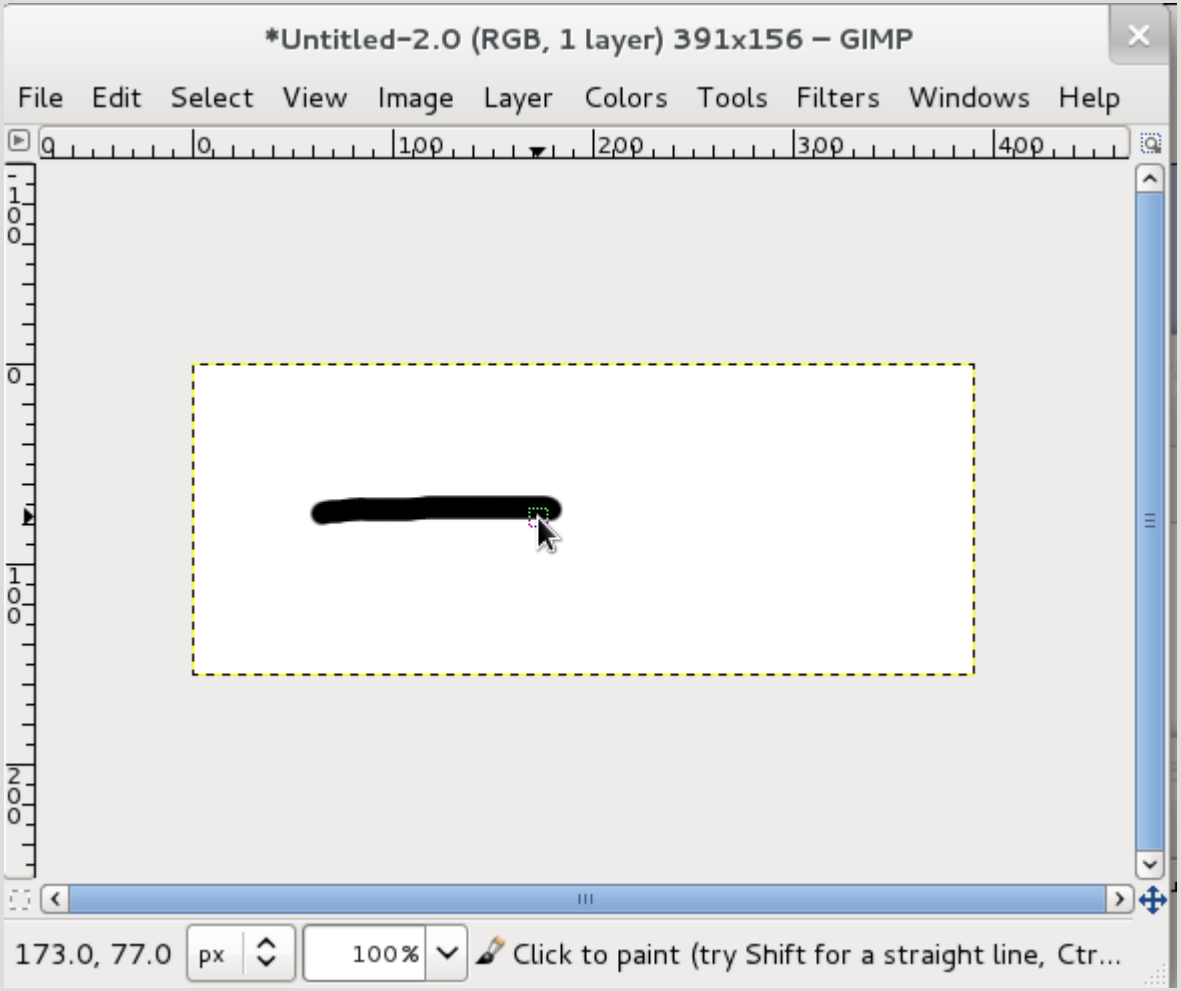
Stratégie


Context
dataAccess()
<<interface>>
Strategy
configure()
execute()
ConcreteStrategyA
configure()
execute()
ConcreteStrategyB
configure()
execute()

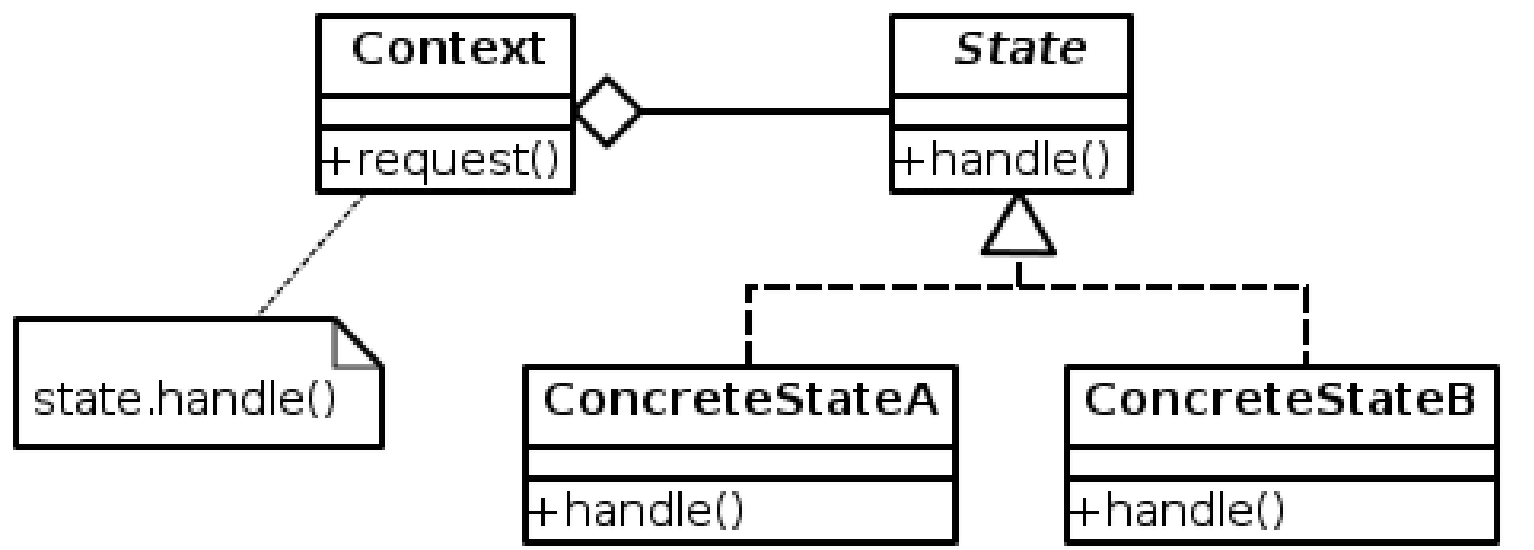
Exemple


<<interface>>
Java::Collection
ListeTriée
getItem(n:Integer):Float
setItem(n:Integer,v:Float)
sort()
<<interface>>
AlgoTrieur
configure()
execute()
QuickSort
configure()
execute()
BubbleSort
configure()
execute()

Patron de conception « état »



Patron de conception « état »

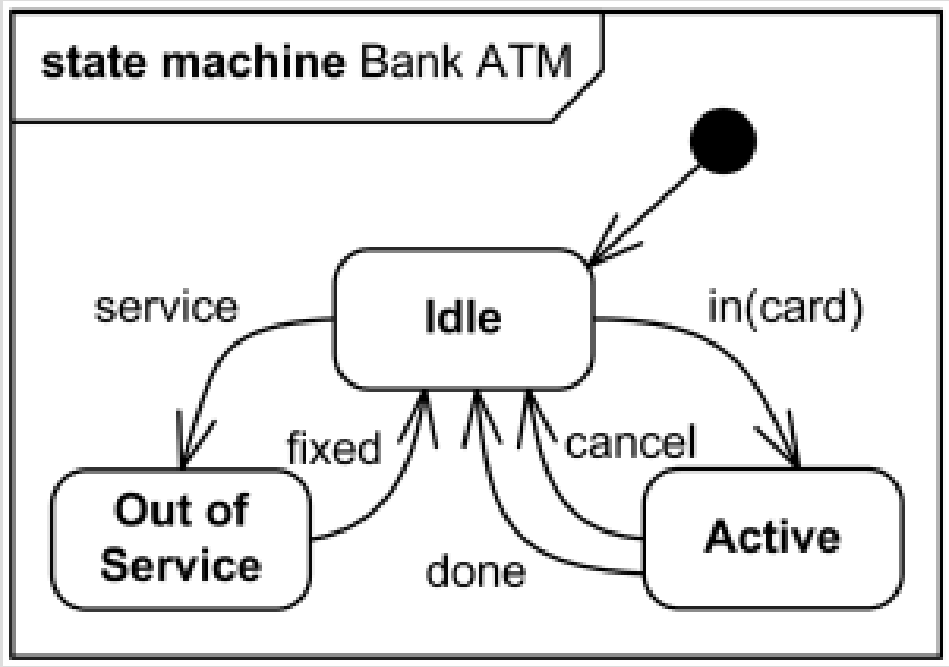


Patron de conception « état »

```
public class Dessin {
        private EtatDessin monEtat;
        public Dessin() {
                setEtat(new EtatCrayon());
        }

        :
        public void setEtat(EtatDessin nouvelEtat) {
                this.monEtat = nouvelEtat;
        }

        public void mouseUp() {
                this.monEtat.mouseUp();
        }
}
```

```
class EtatPipette implements EtatDessin {
        :
        public void mouseUp()
        {
              :
              dessin.setEtat(new EtatCrayon());
        }
}
```

Bref c'est un automate... d'ailleurs UML prévoit une « syntaxe » pour ça : les « state machine »



Conclusion

- Privilégier la structure entre les classes au lieu de test « if »
- Maintenance
- Propriétés fonctionnelles !!
- Refactoring : possibilité de modifier la structure du code « à la volée »