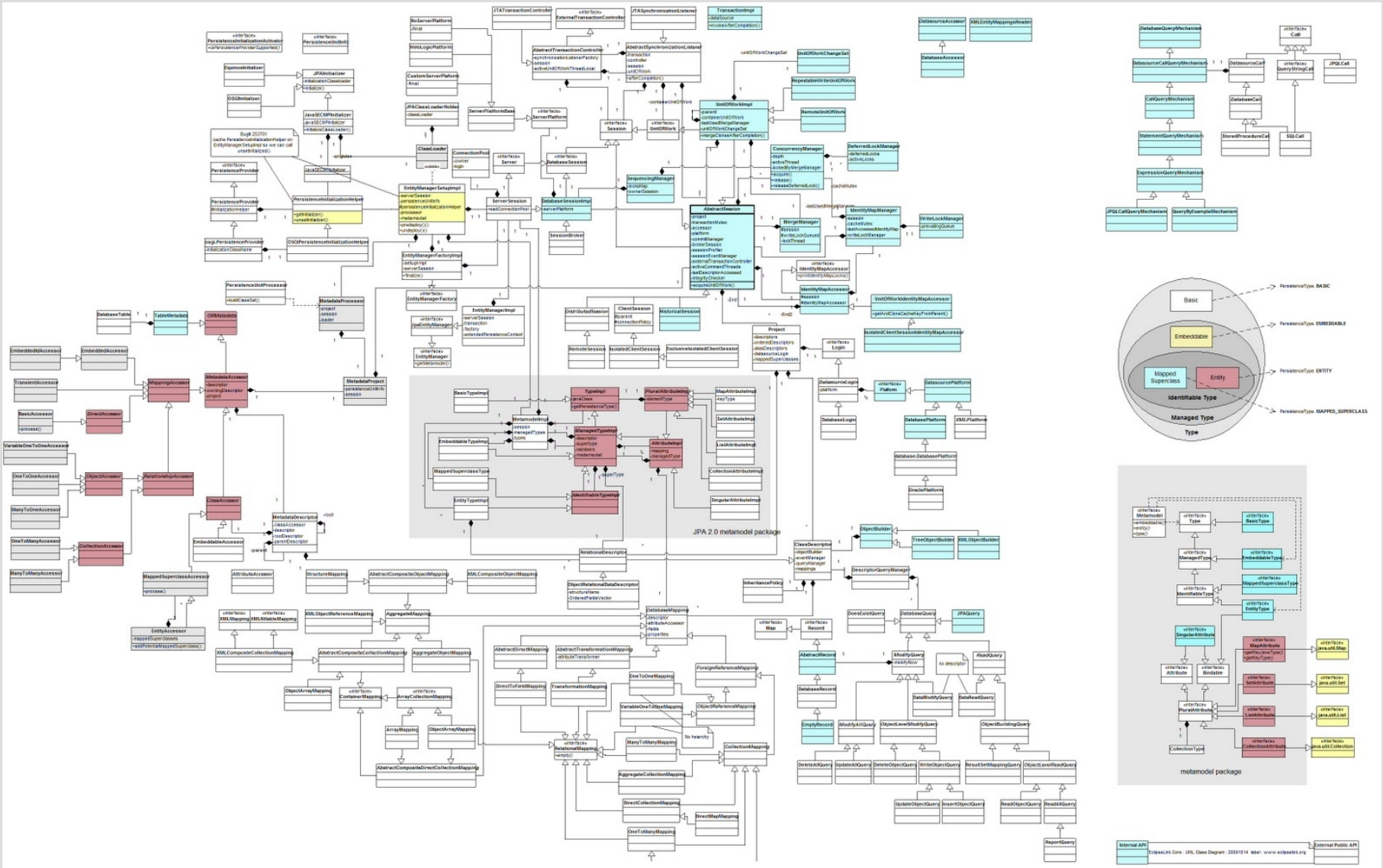
Design patterns

François Schwarzentruber
ENS Cachan – Antenne de Bretagne

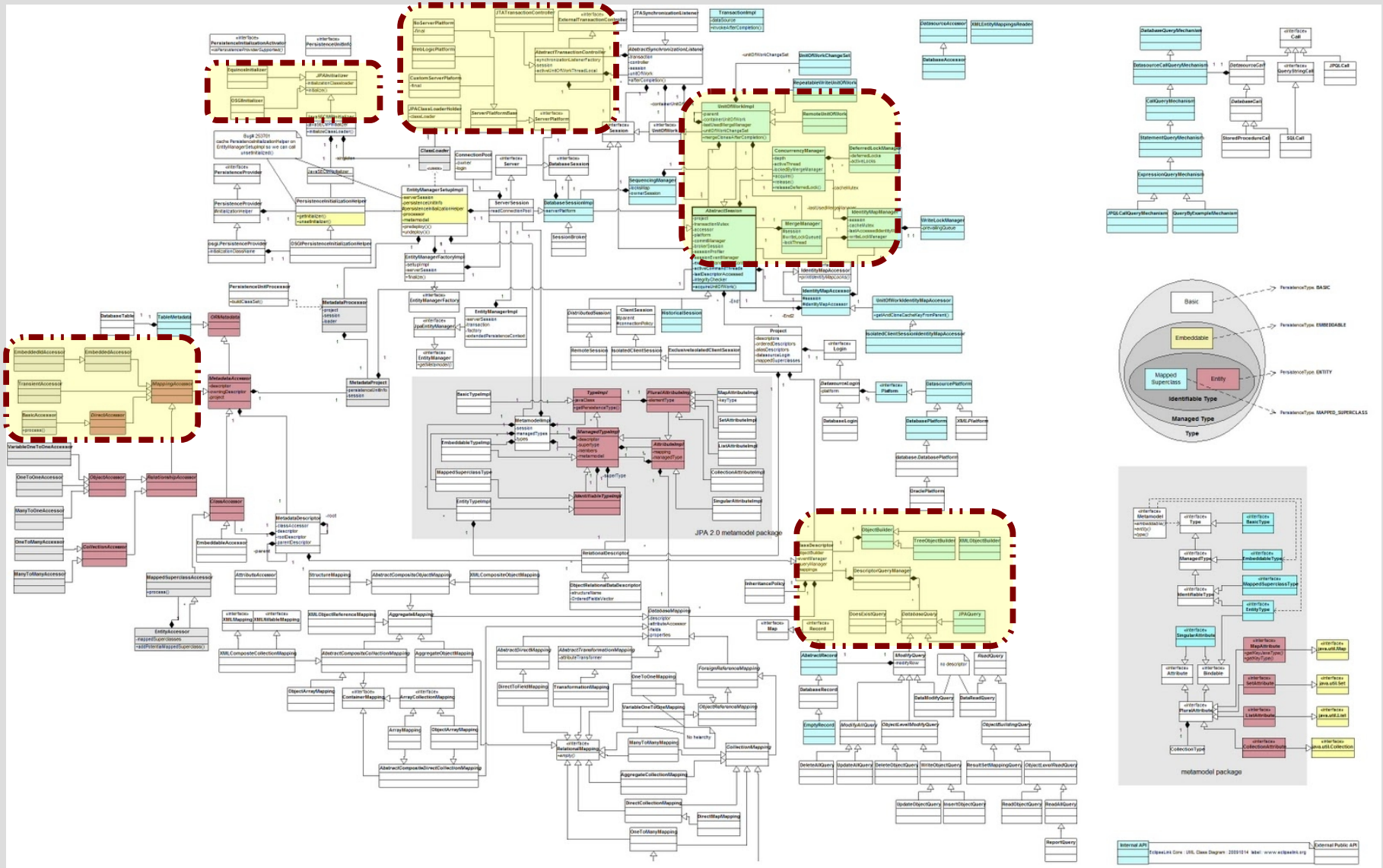
Recall of the object designs

- One class = one responsibility
- Better extend than modify
- Liskov substitution principles
- One interface per client
- Dependency inversion principle

Problem: a system is big...

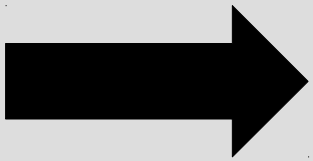


Solution concepts are local.



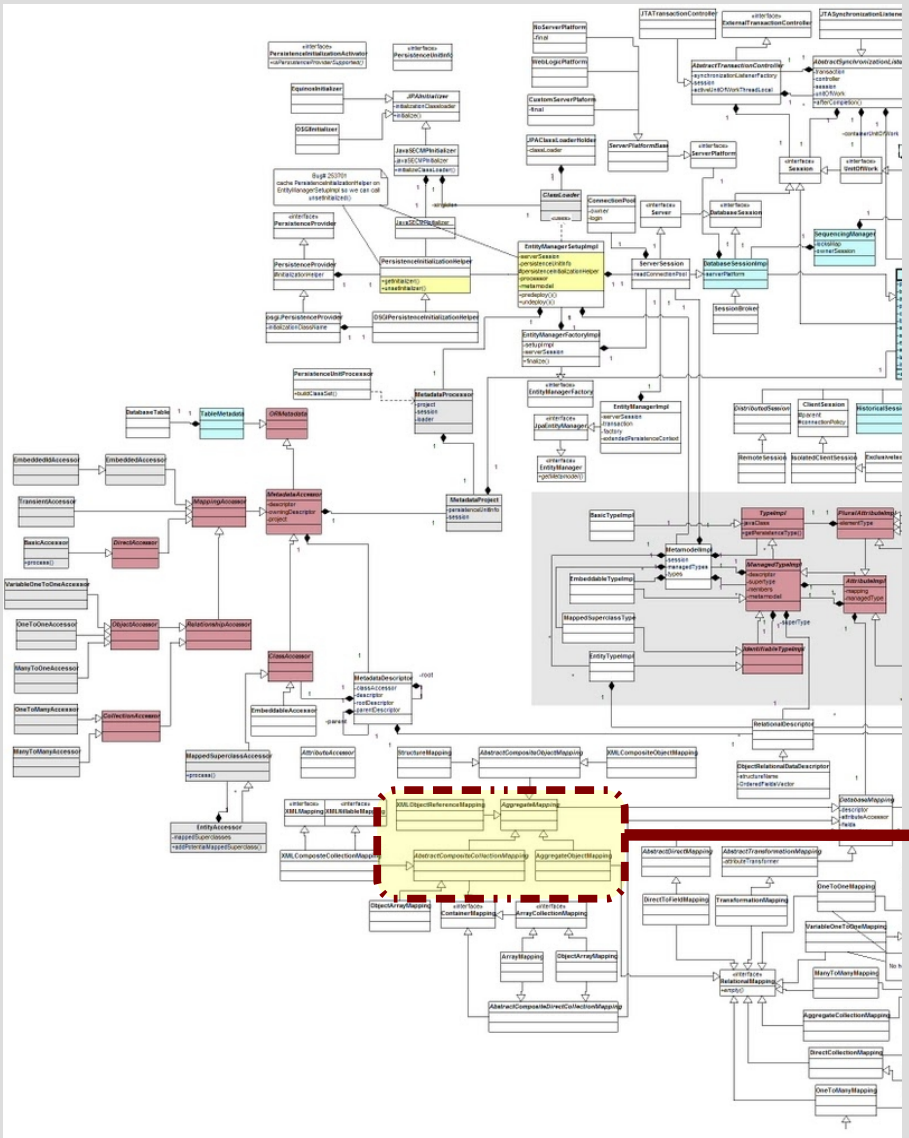
Example

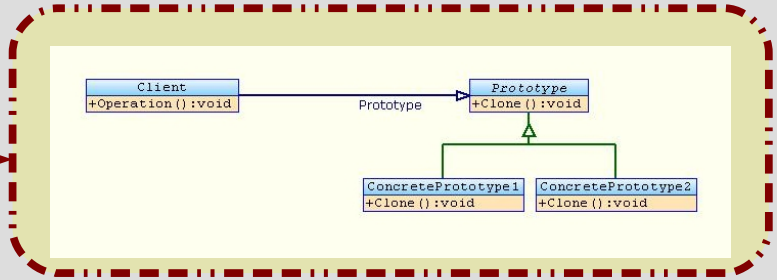
Problem
We need those actions to be extended and undone.



Solution
We apply locally a suitable micro-architecture.

Solution: Design pattern





+ sequence diagram

Design pattern is about...

Functional properties:

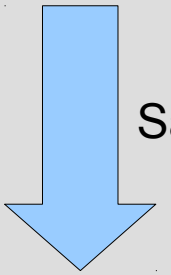
- Correction of an algorithm
- Complexity in time/space
- ...

Non functional property

- Easy to understand
- Easy to maintain
- Easy to test

Wisdom: do not reinvent the wheel

- Integers
- Matrices
- Rubik's cube...



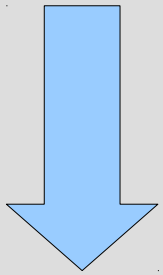
Same concepts

Group theory

(Évariste Galois)

- Subgroup
- Order of an element

- UML Editor
- Pong algorithm suite
- Drawing software

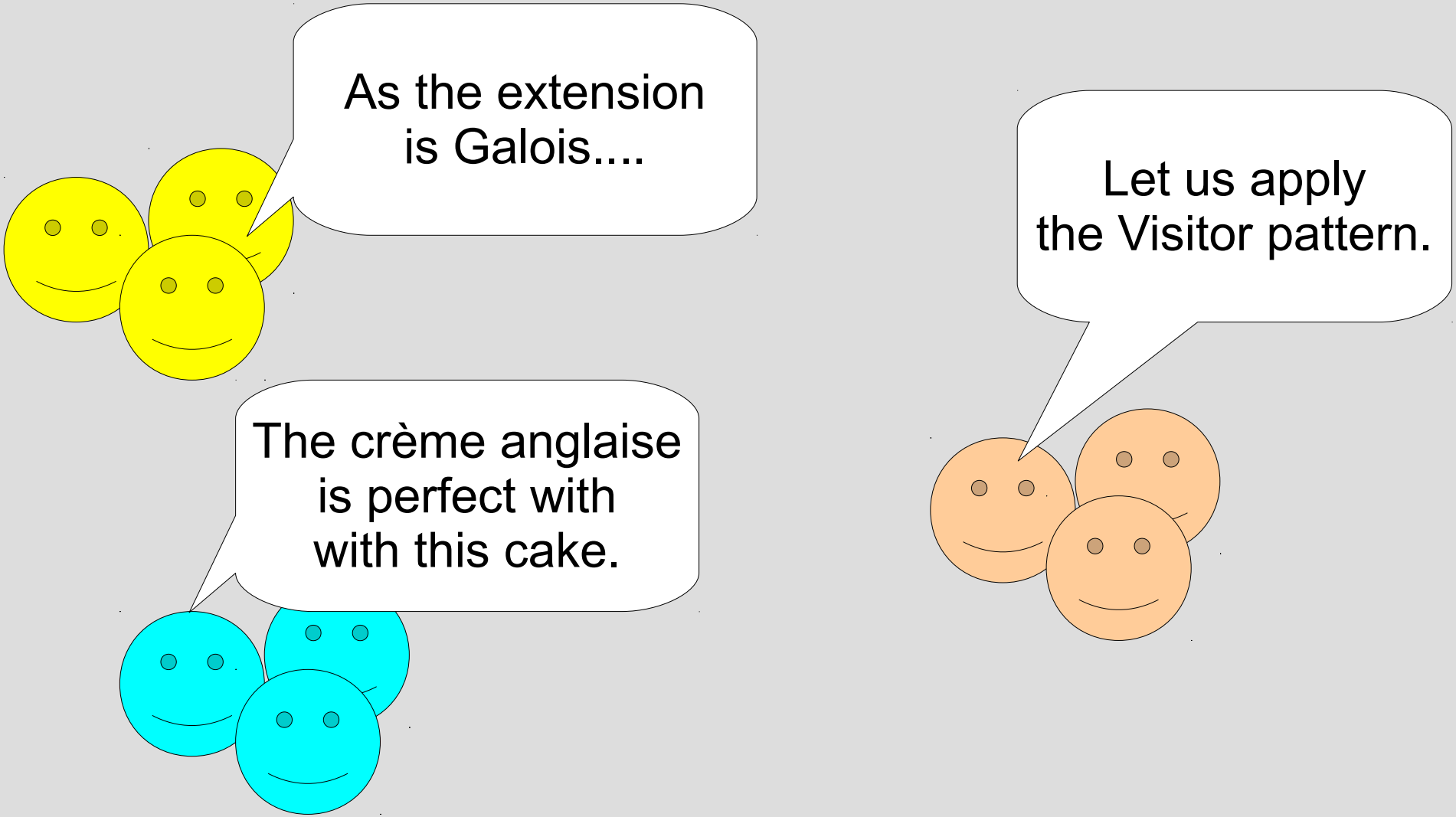


Same solutions

Design patterns

- « Façade »
- « Visitor »

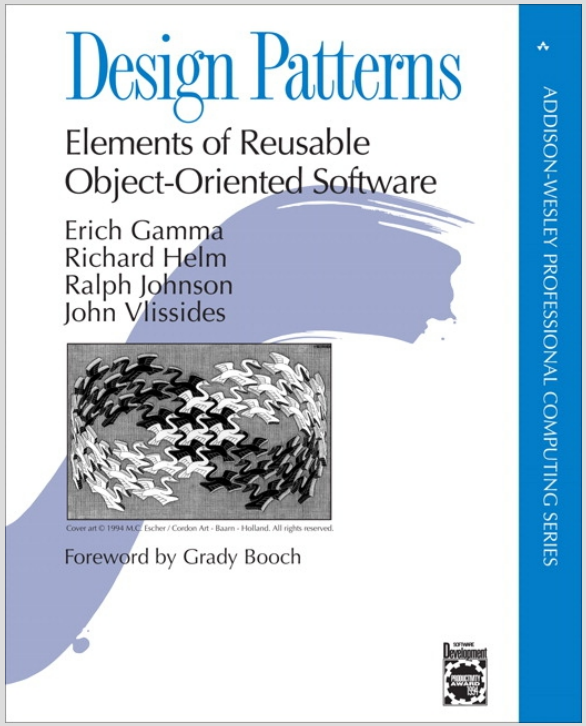
Wisdom: some vocabulary!



The idea of design patterns comes from architecture

- Christopher Alexander : anthropologist and architect
- Idea of reusable concepts

Design patterns



1995 : Gamma, Helm, Johnson et Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software

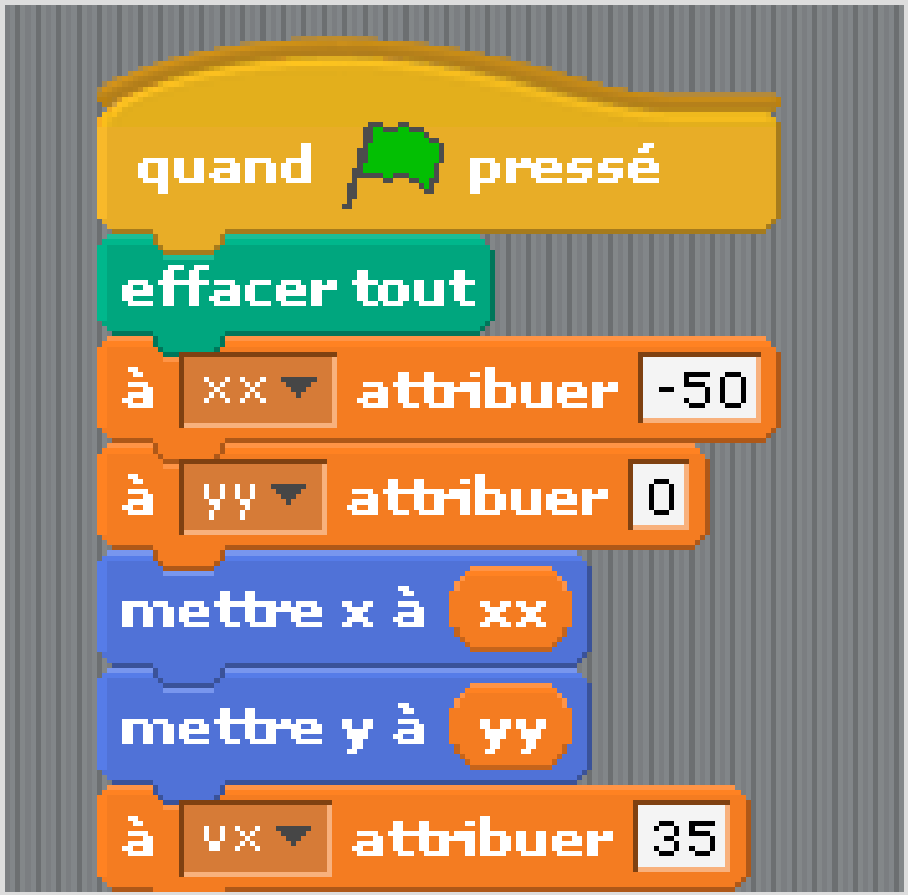
Outline: a trip in the design pattern countryside!

- Creational patterns
- Structural patterns
- Behavioural patterns

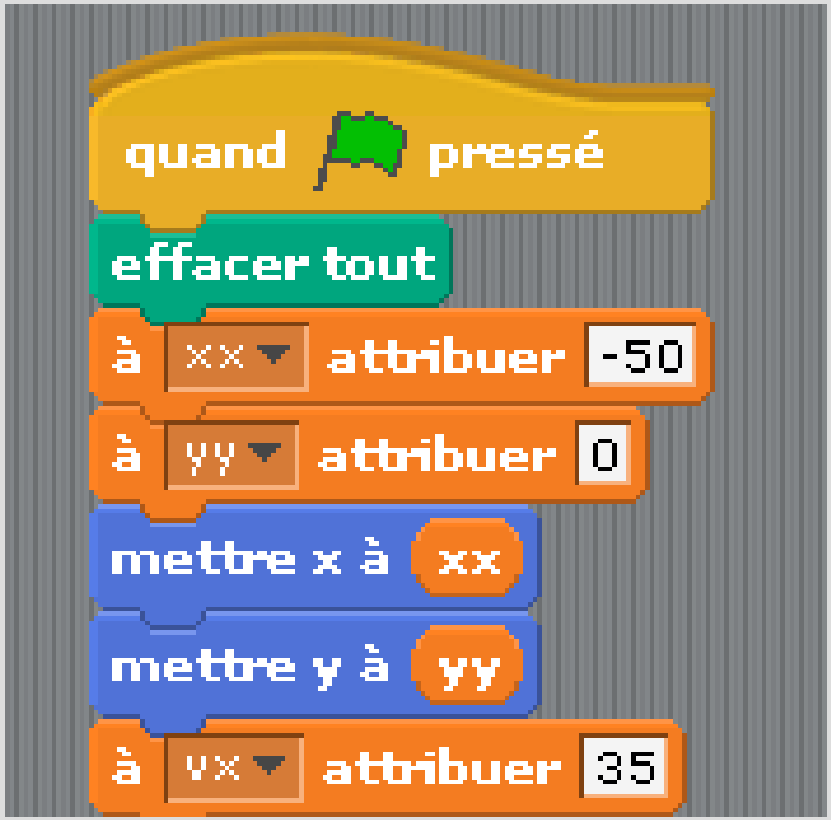
Creational patterns

- Abstract factory
- Prototype

YOU want to design a software for learning algorithmic



Need: to adapt the software to different environments



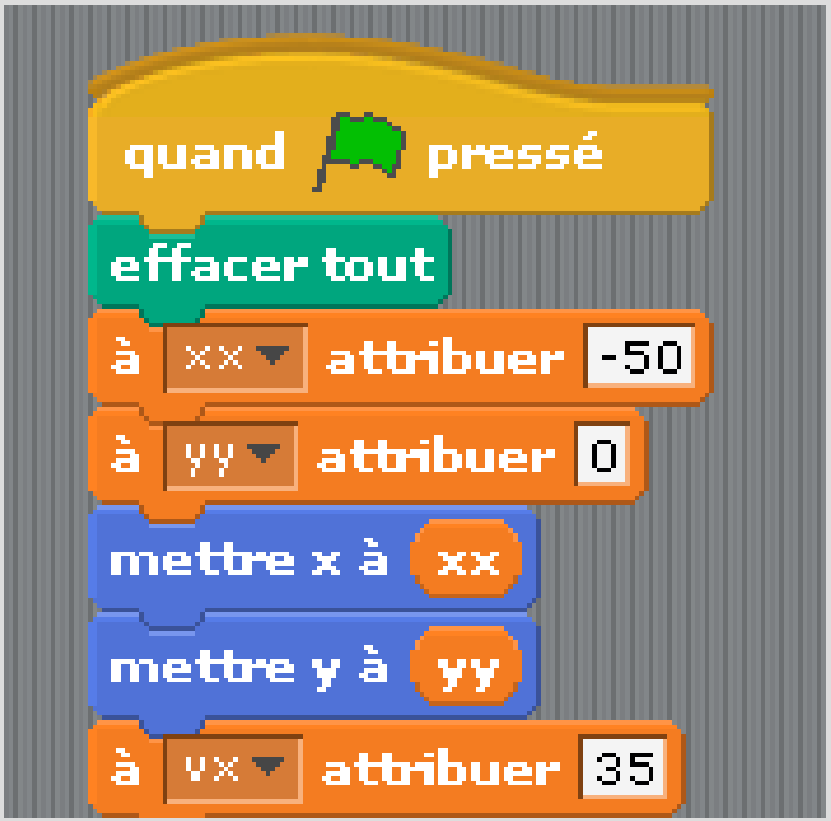
Windows

Linux

MacOS

Android

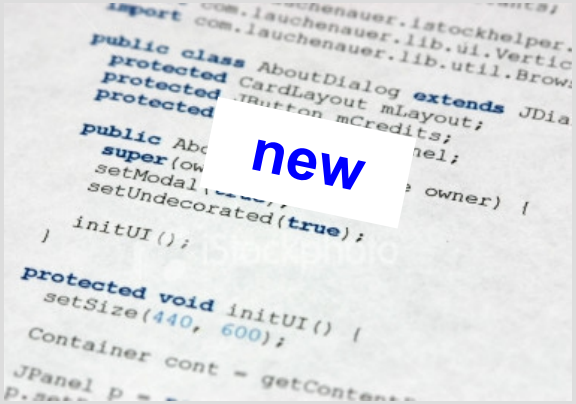
Need: to design the software for different kinds of user



Kids

High-school students

The hell: you have created objects at any part of your software!




```
@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 */
    ssListenerMDB() {
    }
    new
    ssage(Message message) {
        try {
            com.sun.ca.mdb.MyPortType port = service.getSynchronousSamplePortName();
            String ide = message.getStringProperty("id");
            double price = message.getDoubleProperty("price");
            int noOfStocks = message.ge                stocks");
            com.sun.ca.mdb.OperationARe                om.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();
        }
    }
```

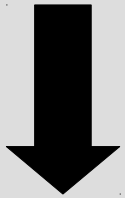
new

```
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 {

       pu    itLien() throws IOException {
             st request =
             inkRequest("MonUrlTresLong", "Alias");
   new
           LittleLinkRequest request2 =
               new LittleLinkRequest("MonUrlTresLong", "4"); // Alias automatique
           try {
               String petitLien = request.getLittleLink();
               String petitLi   = r      tLittleLink();
               System.out.pri
               System.out.pri
           } catch (LittleLin
               e.printStackTrace();
           }
       }

   }
```

new

Non maintainable solution

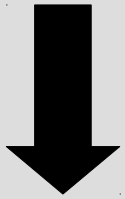
```
b = new brickCleanAll() ;
```



```
if(forKids)
{
        b = new brickCleanAllForKids() ;
}
else
{
        b = new brickCleanAllForHighSchoolStudents() ;
}
```

Non maintainable solution

```
b = new brickCleanAll() ;
```

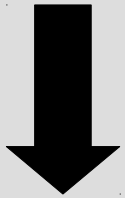


```
if(forKids)
{
        b = new brickCleanAllForKids() ;
}
else
{
        b = new brickCleanAllForHighSchoolStudents() ;
}
```

It violates
1) the open closed principle
(better extend than modify)

Non maintainable solution

```
b = new brickCleanAll() ;
```



It violates
1) the open closed principle
(better extend than modify)

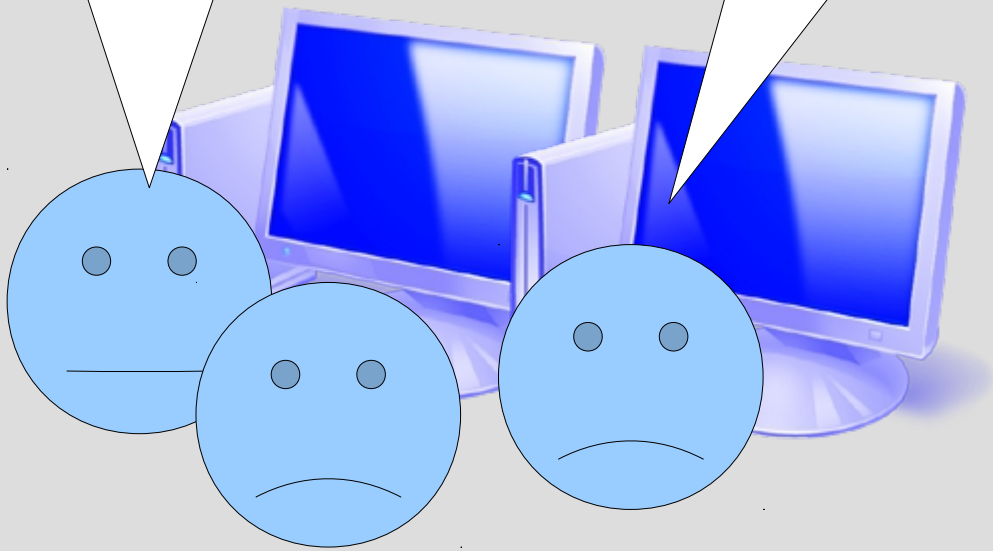
2) The dependency inversion principle
(do not depend on concretions)

```
if(forKids)
{
        b = new brickCleanAllForKids() ;
}
else
{
        b = new brickCleanAllForHighSchoolStudents() ;
}
```

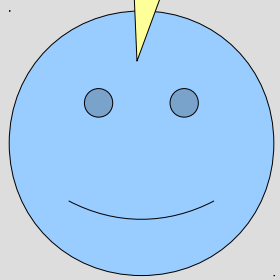
Abstract factory

We need to adapt our software to two different kinds of users.

How to do this?

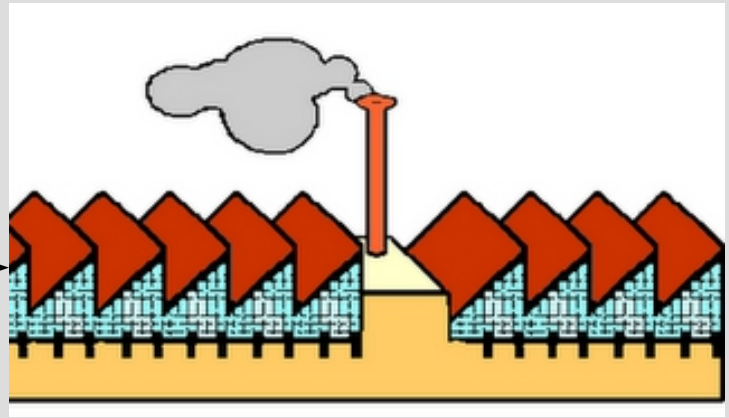


We apply the abstract factory pattern.



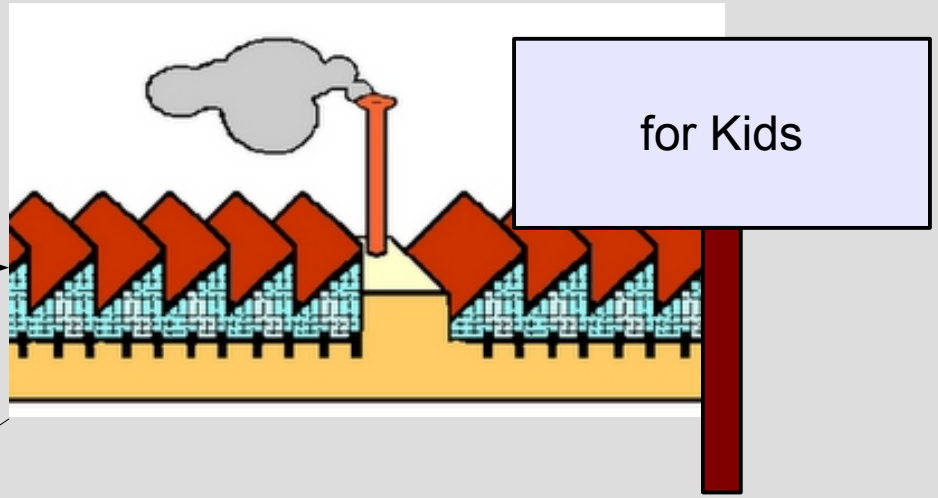
Solution: to isolate the object creation in factories

please...
create
a “CleanAll” brick.



Solution: to isolate the object creation in factories

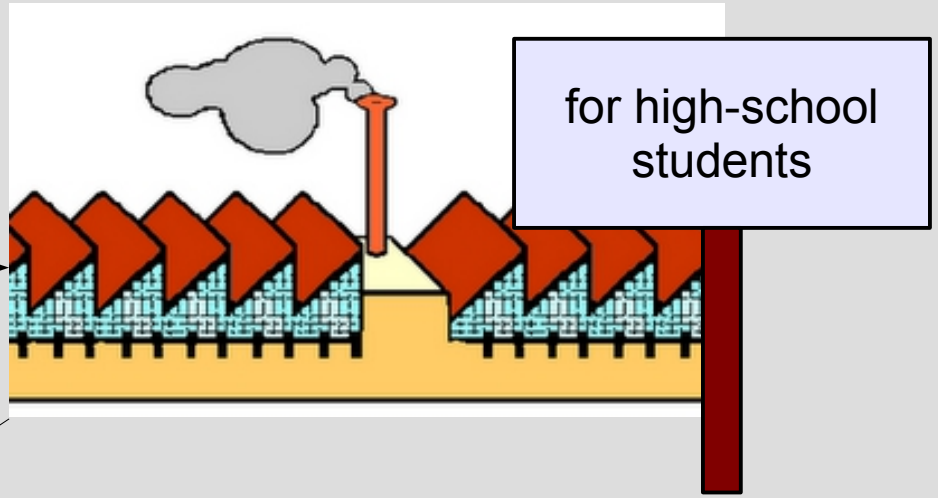
please...
create
a “CleanAll” brick.



clean all

Solution: to isolate the object creation in factories

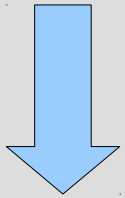
please...
create
a “CleanAll” brick.

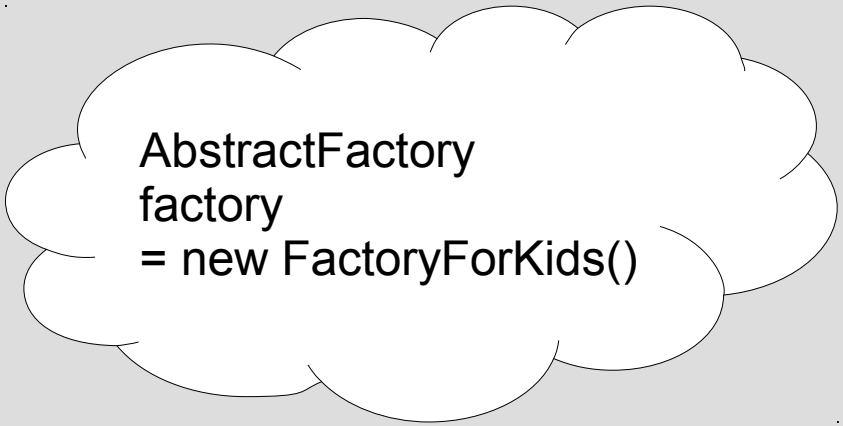


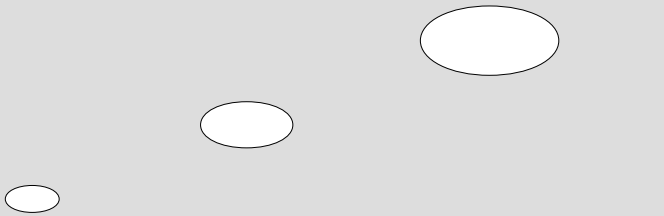
clean all

Abstract Factory pattern

b = new BrickCleanAll() ;

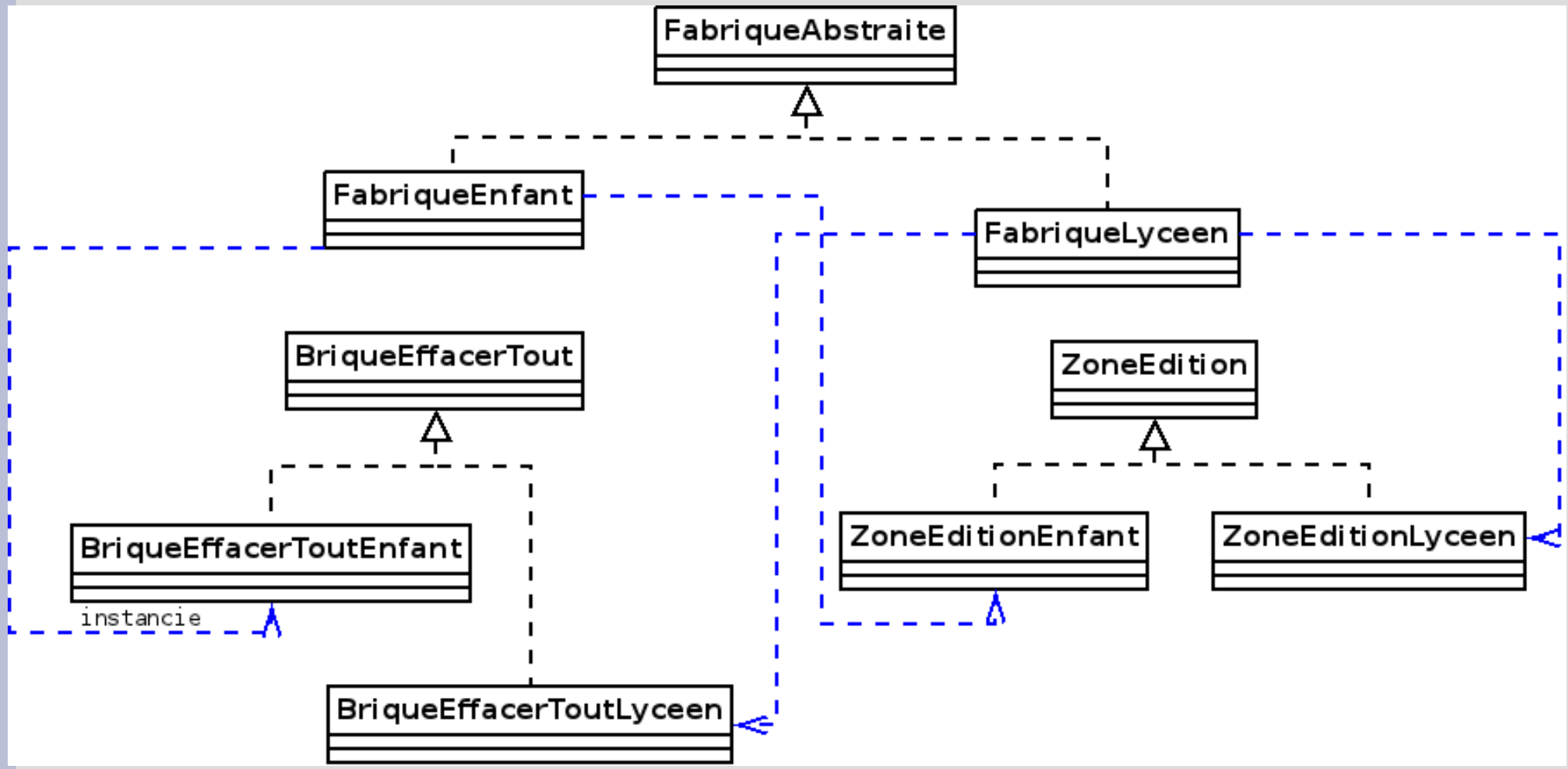




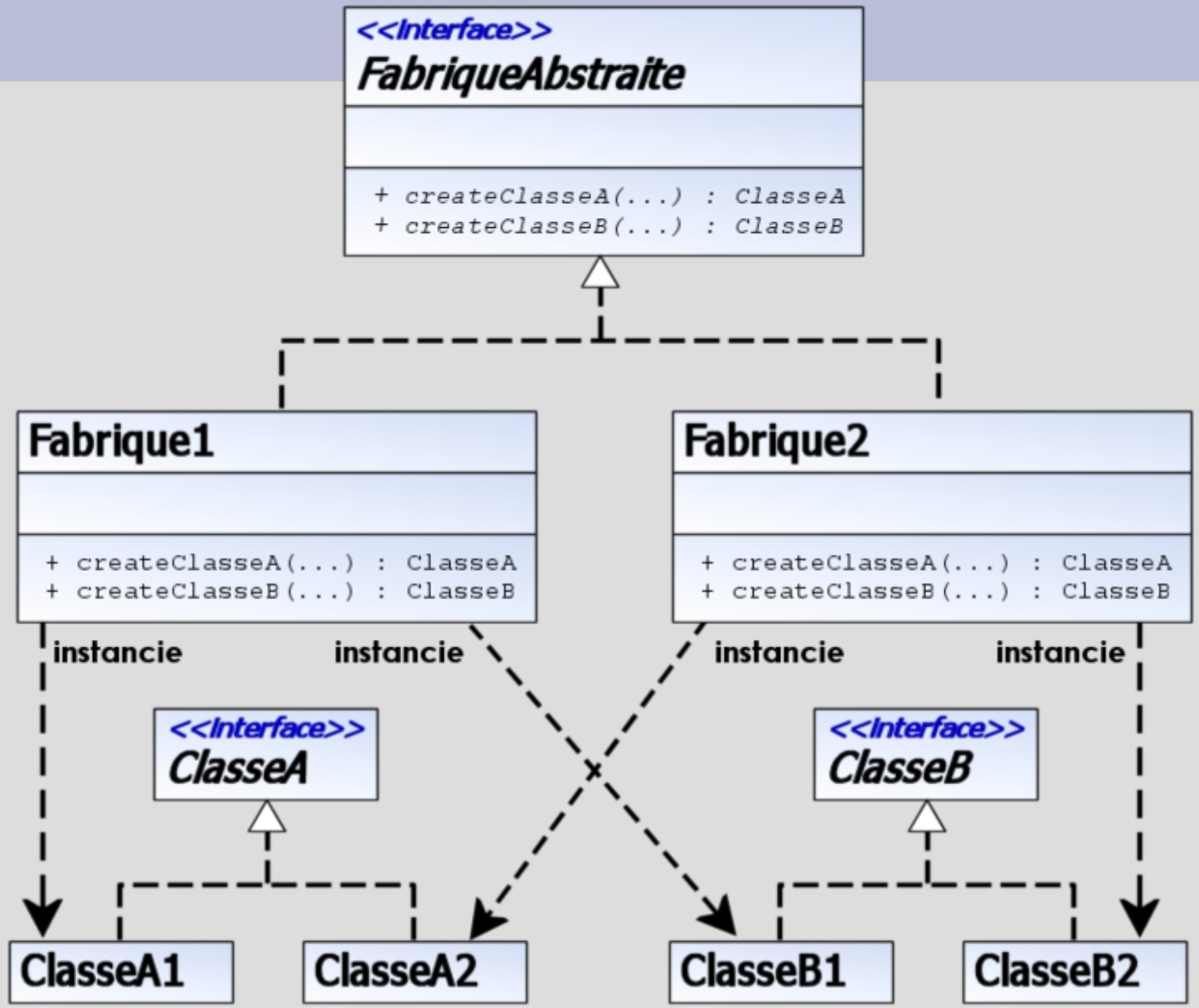


factory.getNewBrickCleanAll()

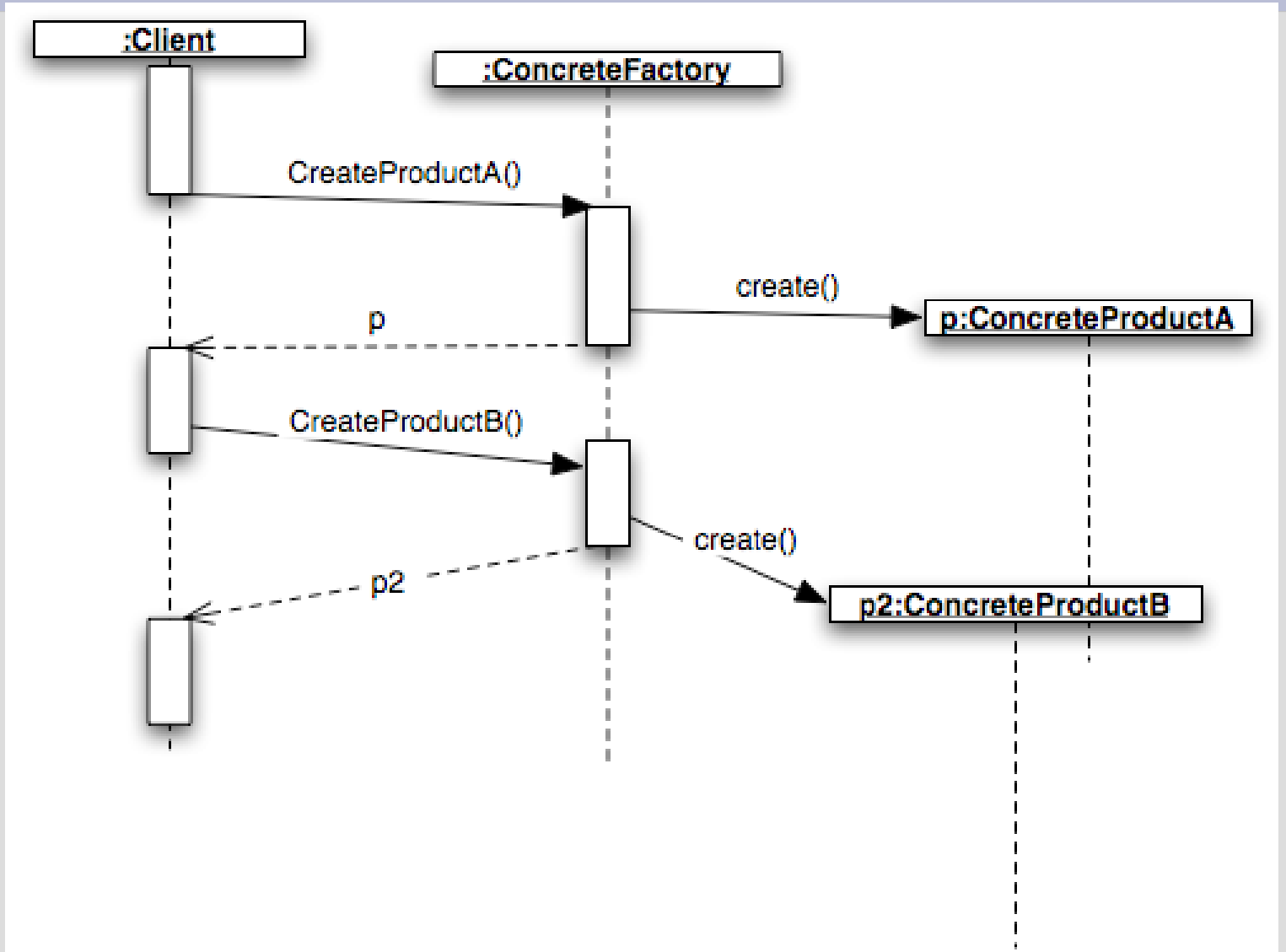
Abstract Factory pattern



Patron de conception : fabrique abstraite



Sequence diagram for the Abstract Factory pattern



Source : cours de Noël Plouzeau

Conclusion on the abstract factory

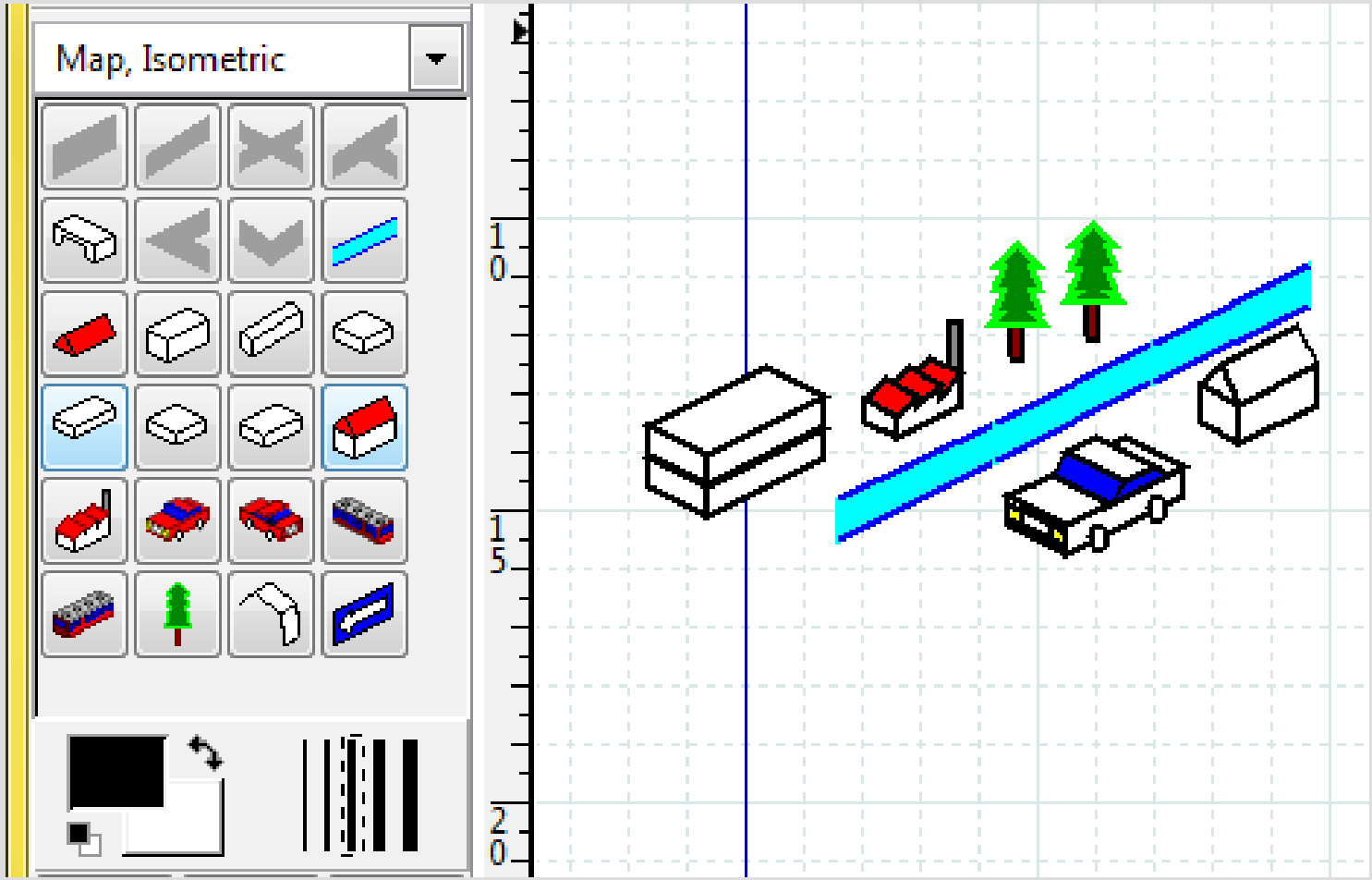
Good points

- depends on abstraction
- platforms are isolated (less coupling)

Bad points

- Factory is difficult to maintain ~ but in fact it would worse with the abstract factory pattern!

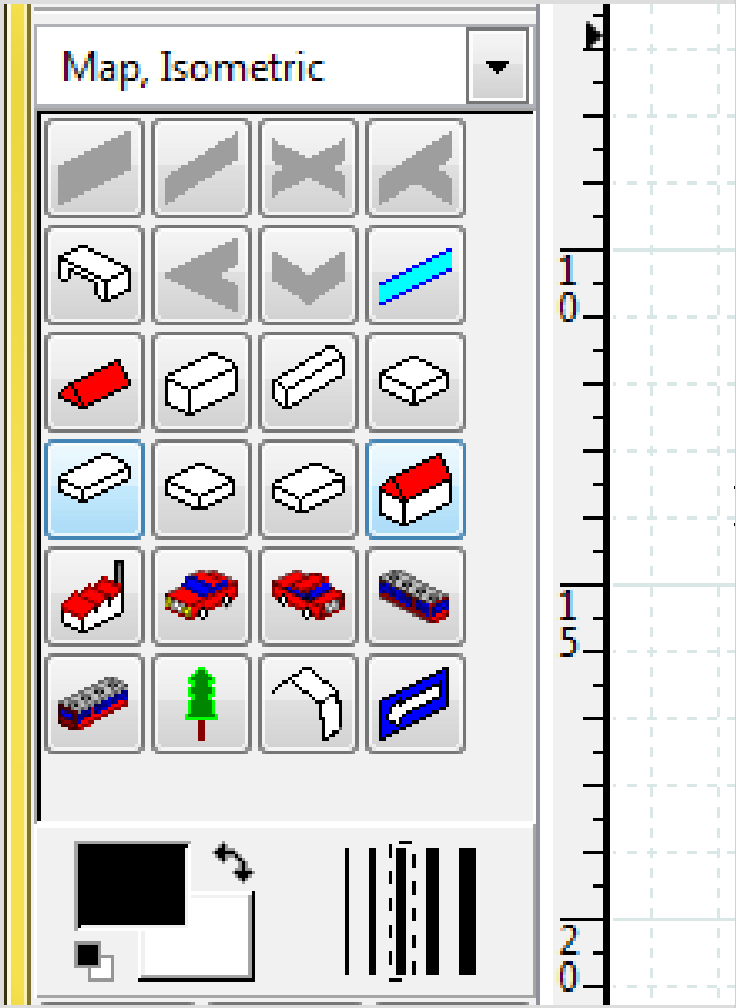
Build objects from a model: prototype pattern



Source : logiciel Dia

Needs

- Copy objects
- Being able to add elements to the palette



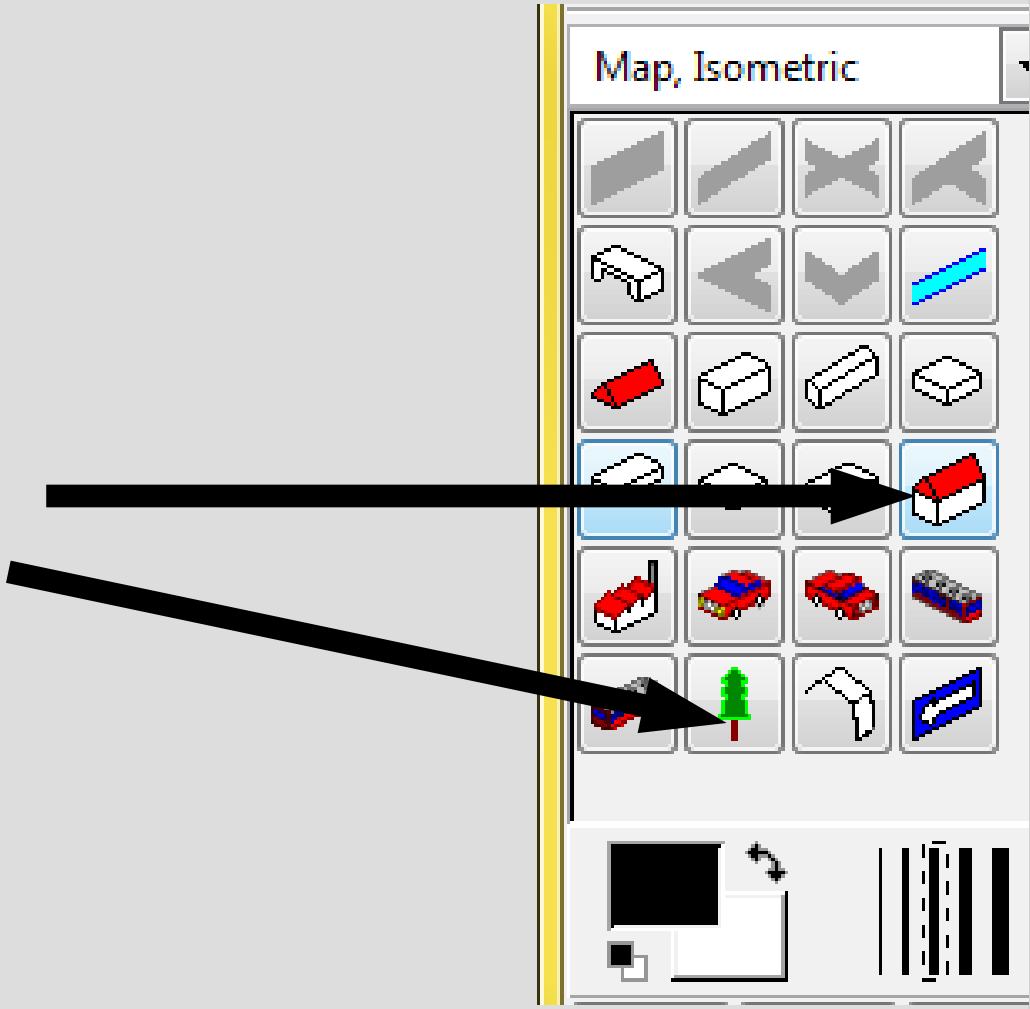
DO NOT DO THAT: one creation per button!

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

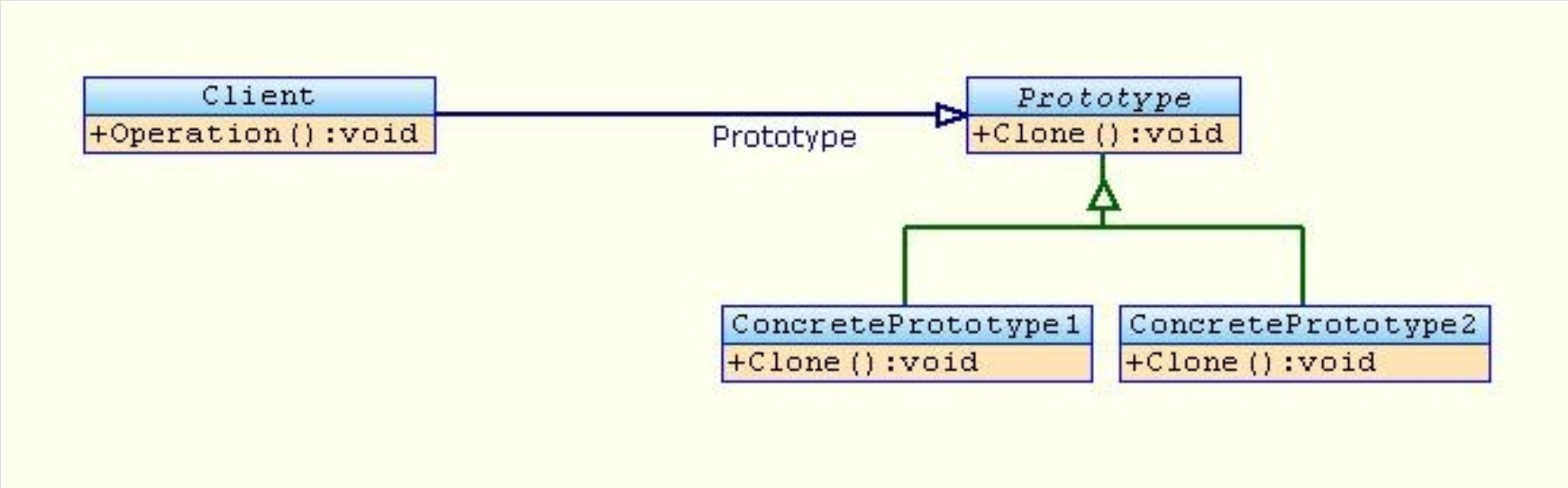
```
new Tree(Color.GREEN,
Color.BROWN, 8, 32, 8)
```

TO DO!

Each button contains a prototype to clone.



Prototype pattern



Source : wikipedia

Conclusion on the Prototype pattern

- + Buttons depend on abstraction
- + Only one class for buttons
- - Clone to implement

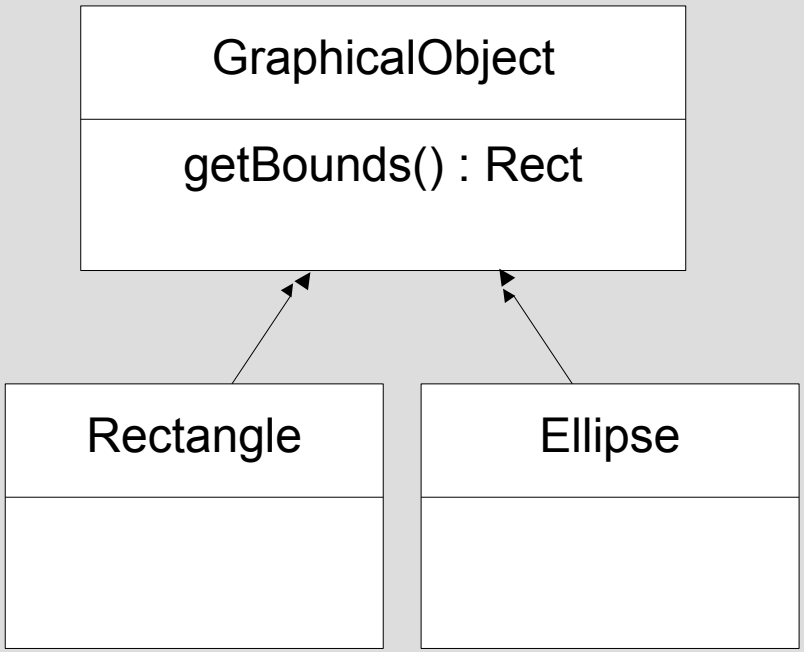
Structural pattern

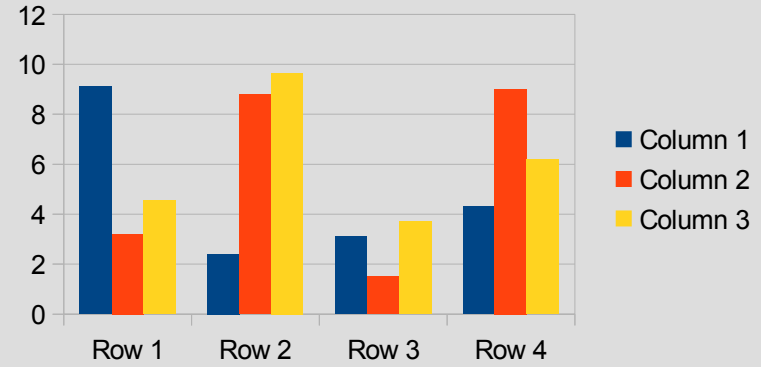
- To adapt an object to a given interface (~ adaptor)
- To propose a simplified interface (~ façade)
- Divide responsibilities (~ bridge)
- Recursive structures (~ composite)
- Add many features (~ decoration)

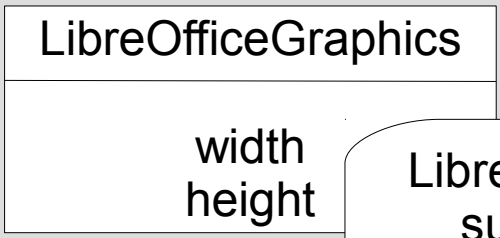
Need

I have implemented rectangles, ellipses in my software...

But I also want to use graphics for statistics...





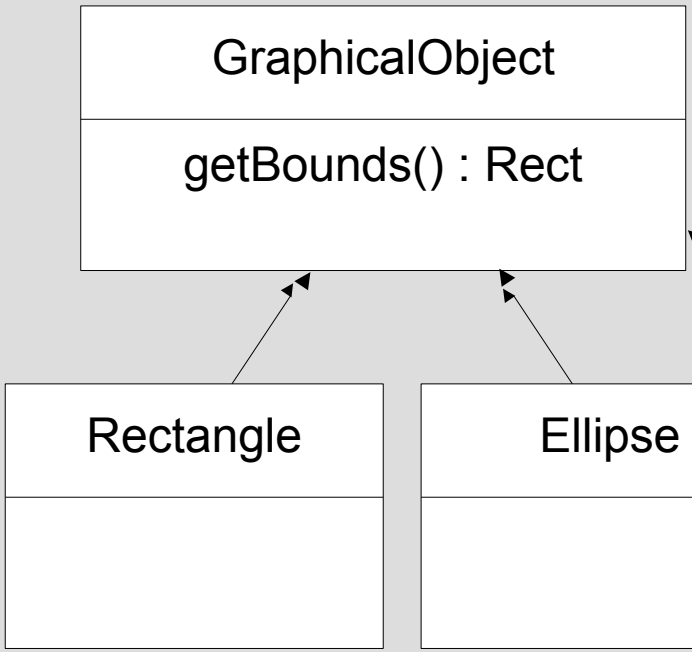


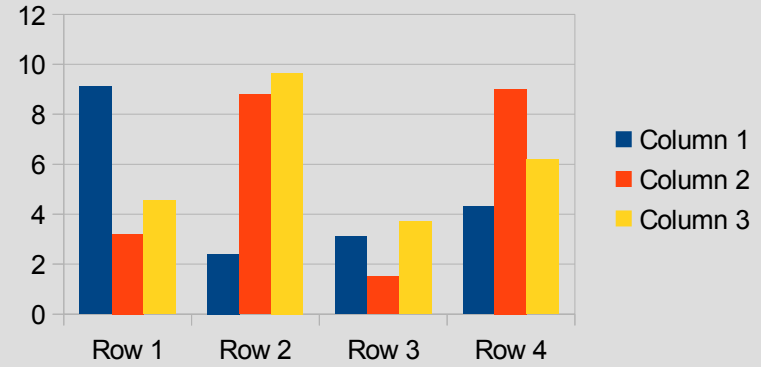
LibreOffice proposes such graphics... But the interface is different...

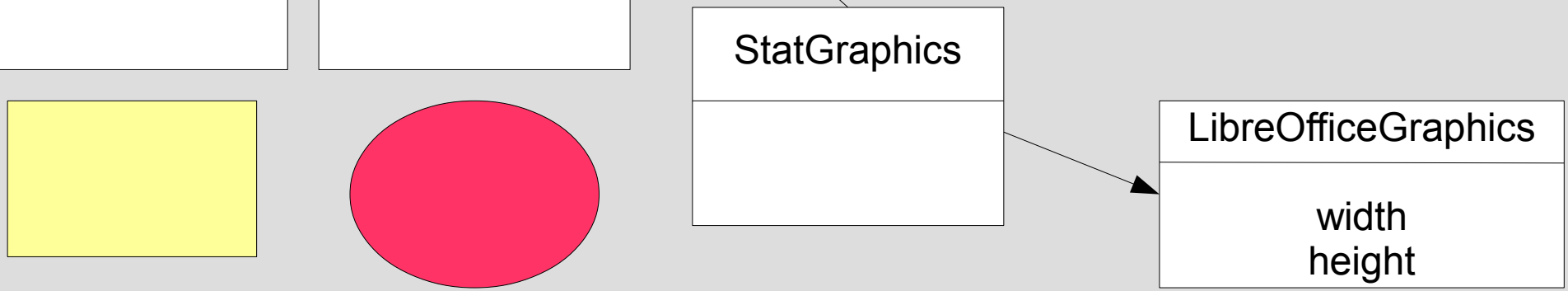
Solution

I have implemented rectangles, ellipses in my software...

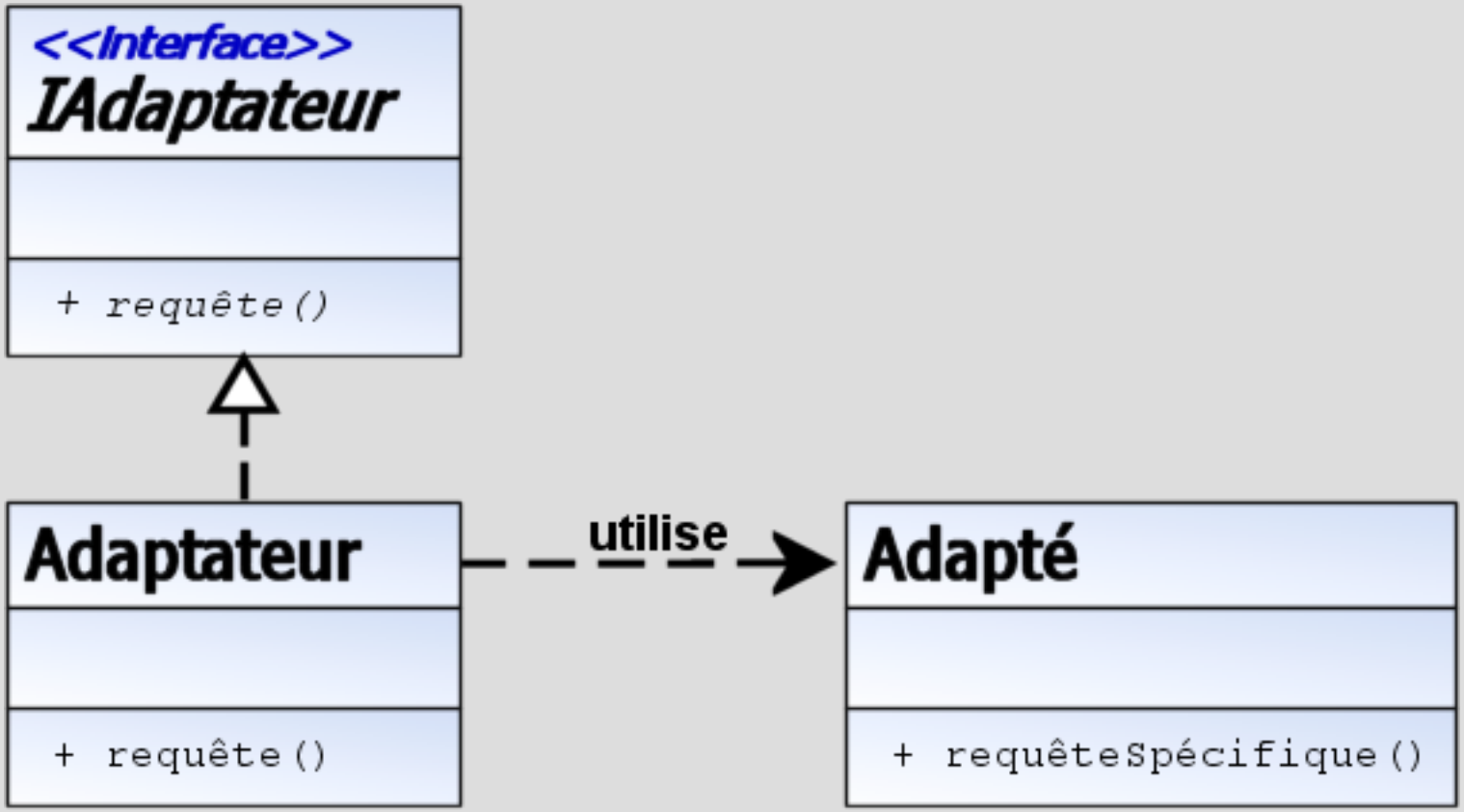
But I also want to use graphics for statistics...





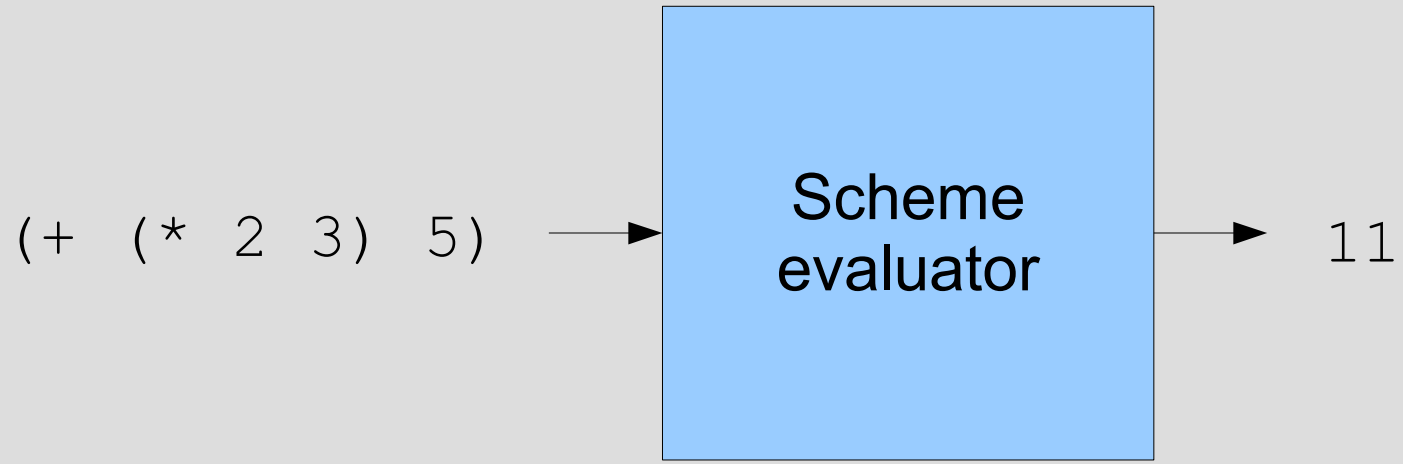


Adaptor



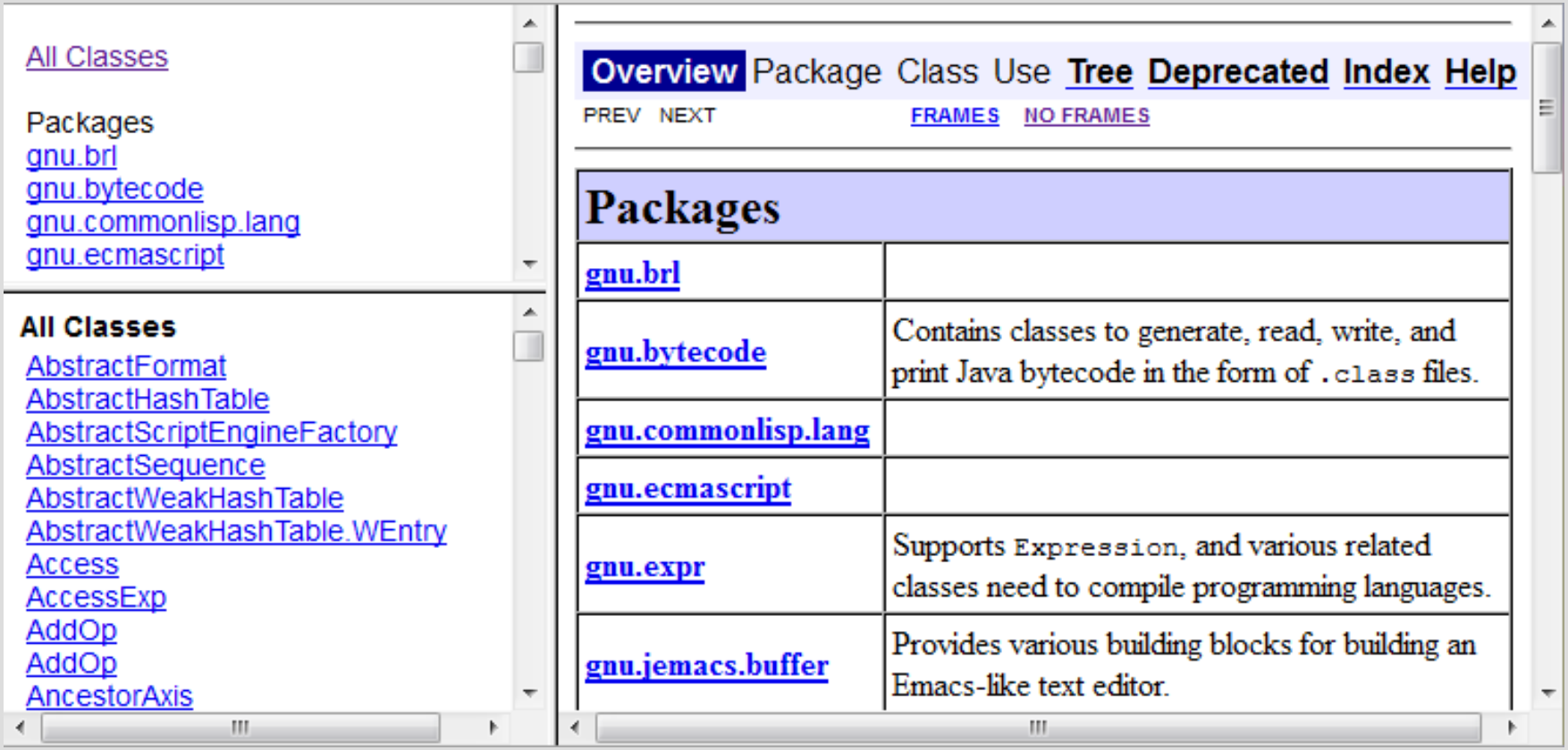
source : Wikipedia

Other story: new need!

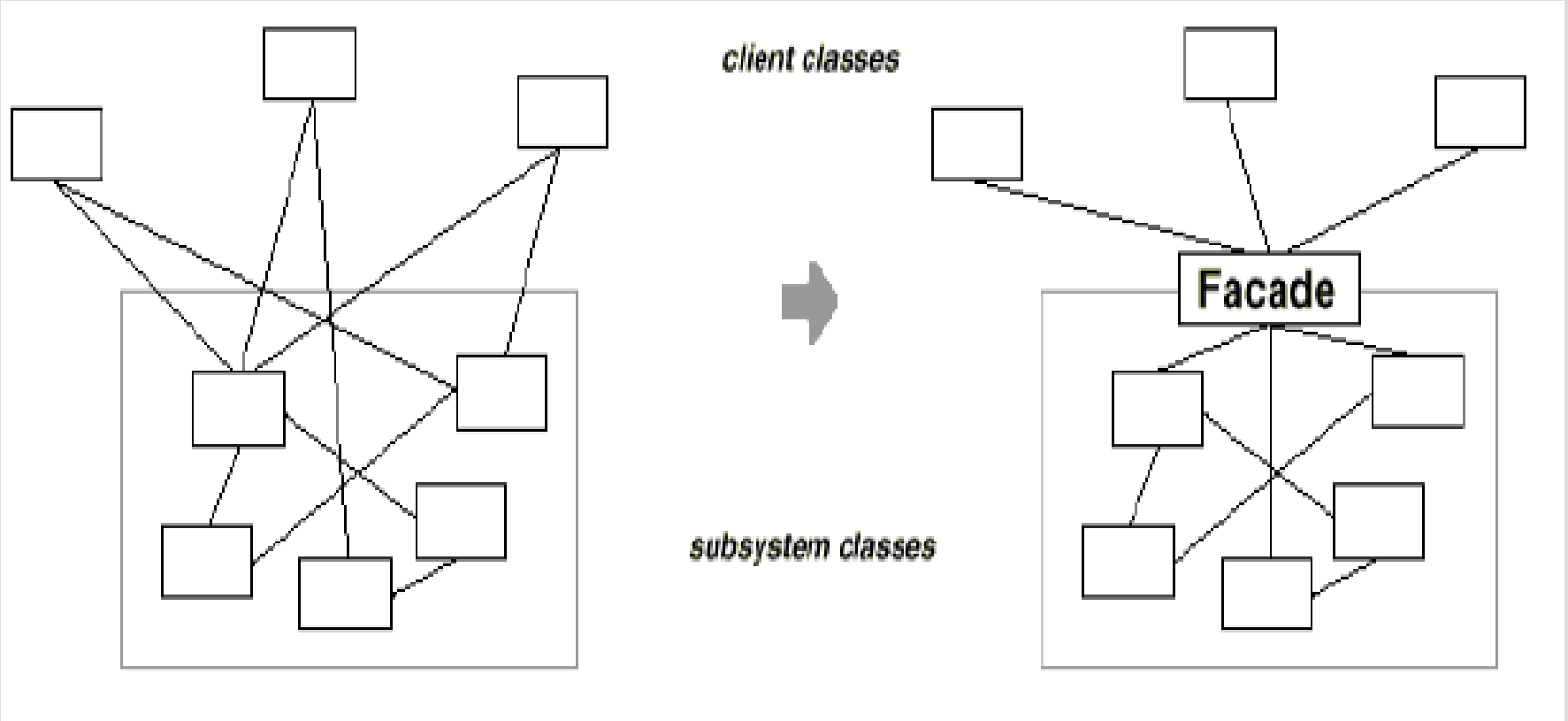


What I have...

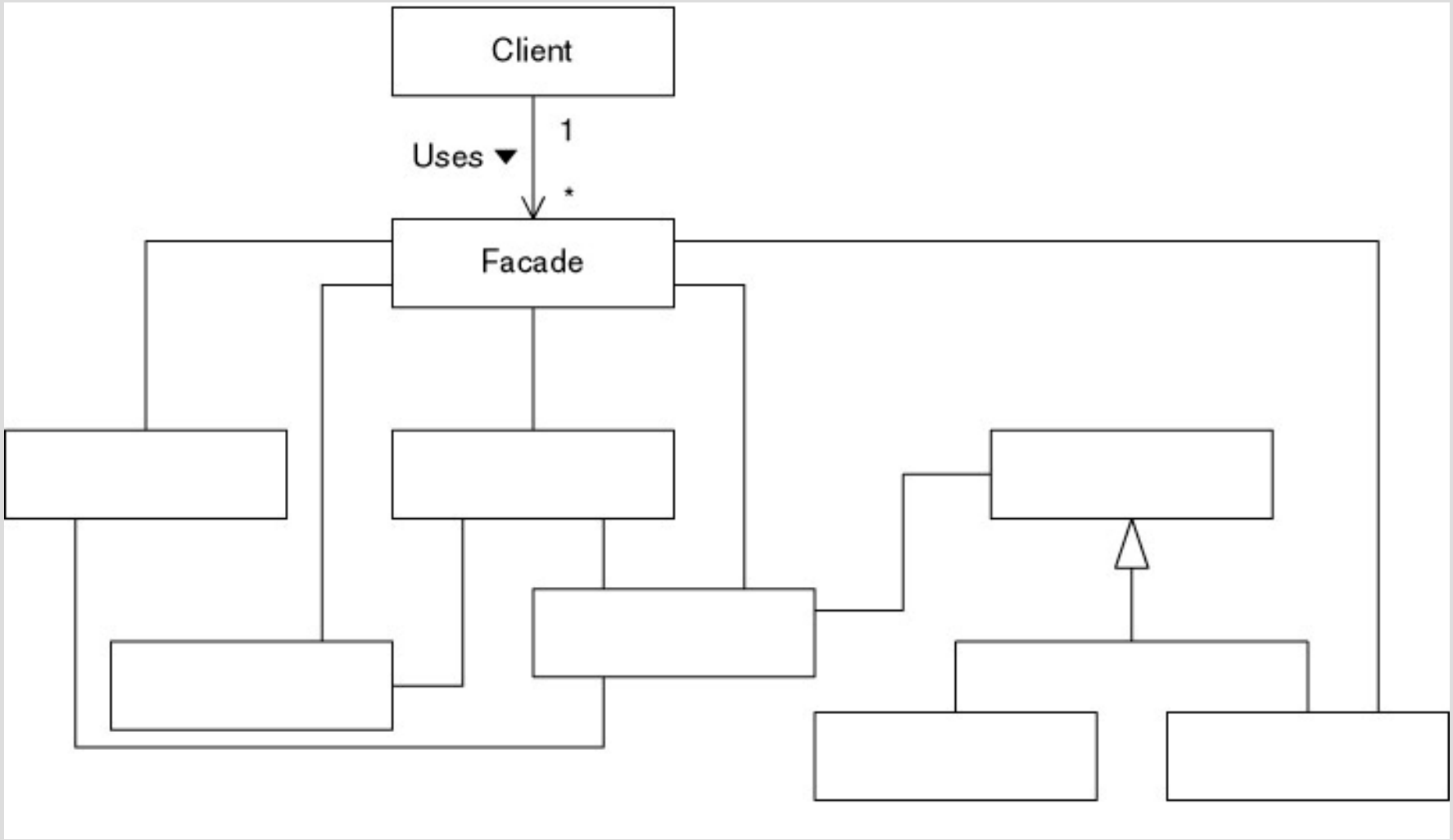
I may use the library `kawa` (a JAVA library for parse/compile/interpret Scheme code and handle Scheme environments)



Solution: Façade design pattern



Solution: Façade design pattern



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

Difference between Façade and adaptor

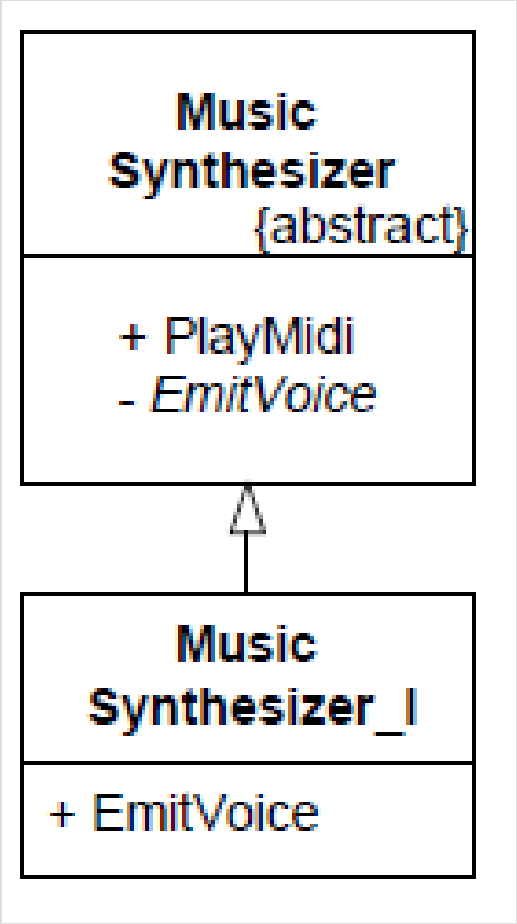
Façade

- We need and create an interface

Adaptor

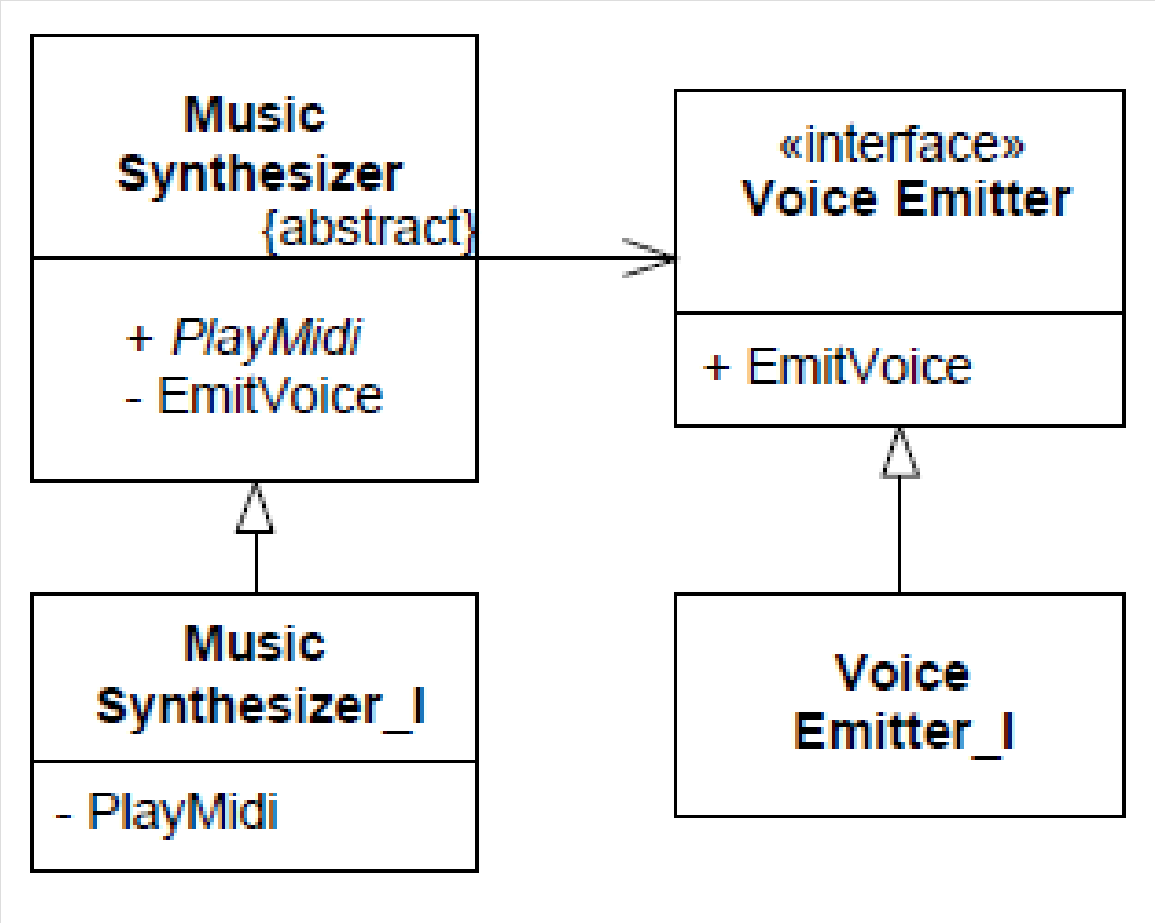
- We need to adapt an object to a given interface.

New problem: to divide responsibilities



- Good: we are able to modify the way we emit voice (soft, hard, 8bit like...)
- Bad: we want also to modify the way a sound is played (strict rhythm, rubato...)

Solution: bridge design pattern

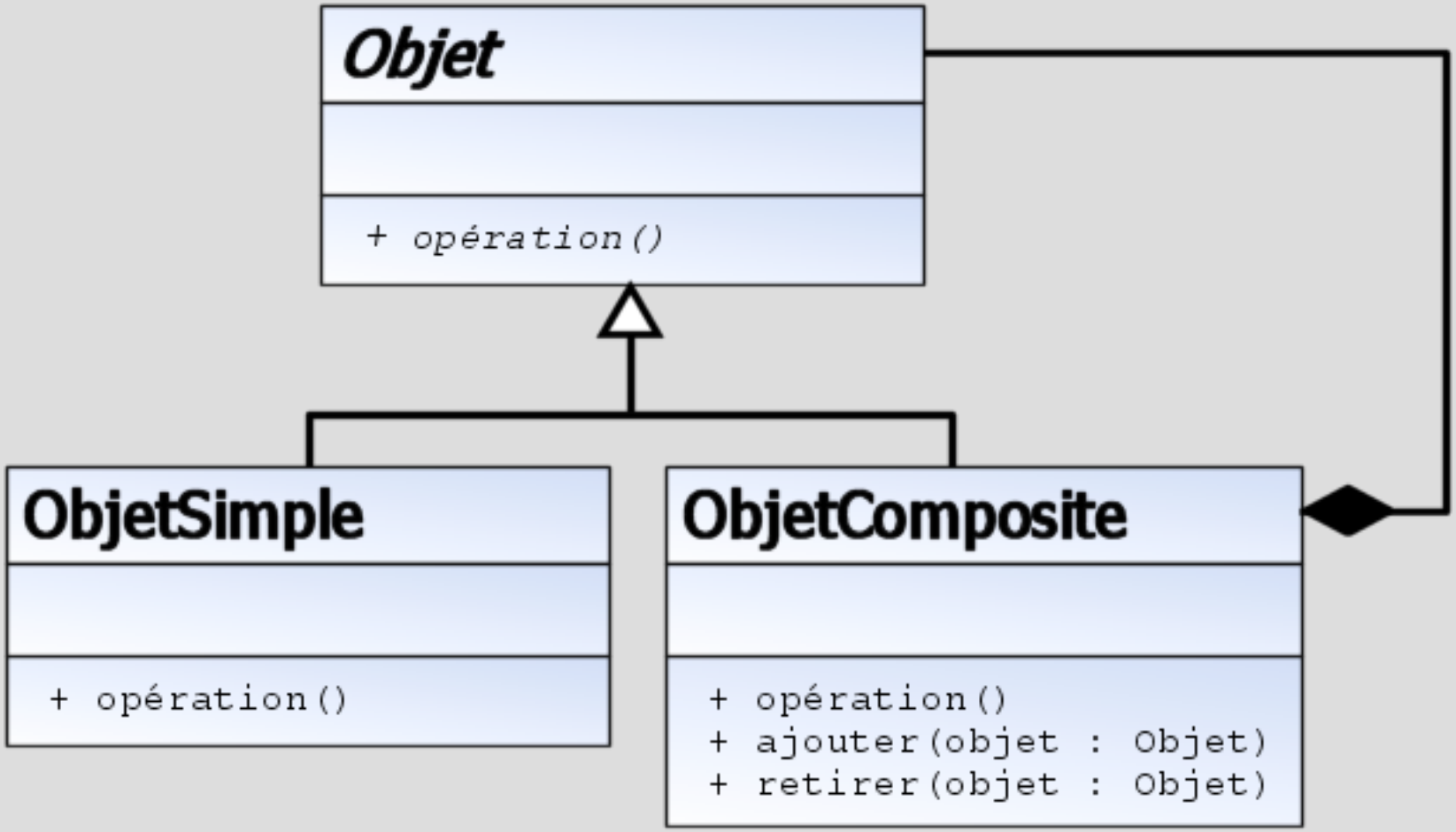


« Composite »

Recursive objects:

- File and folders
- Expressions
- Structure of a document
- Commands (we will see in a few minutes)
- Etc.

« Composite »



Source : wikibooks

To handle decorations

A window may:

- Have / not have a border
- Have / not have scrollbars
- Have / not have a background
- Handle / not handle zoom
- Have / not have special effects
- Etc.

Multiple solutions

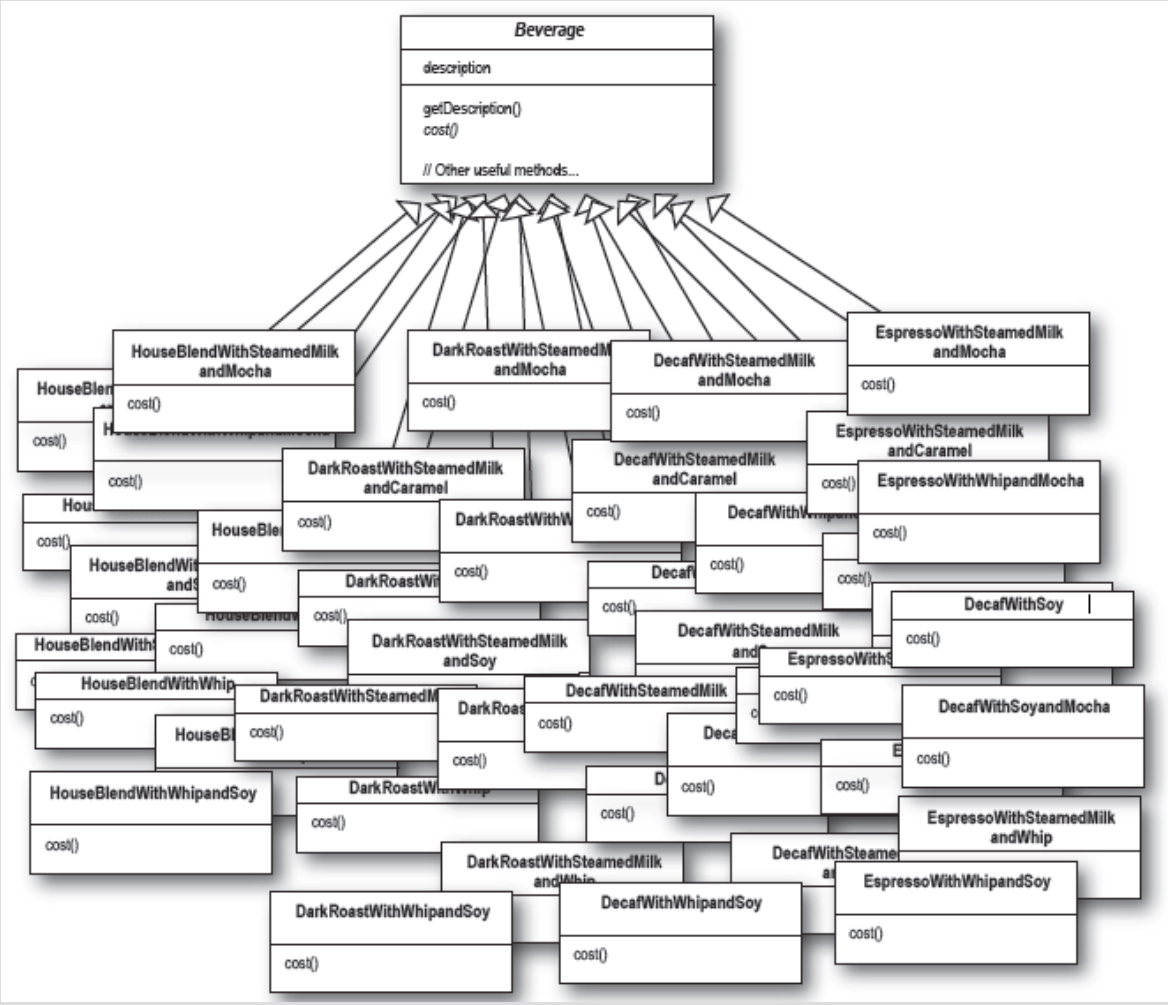
If you can modify the interface and the window class:

- Use “if”
- Use a kind of bridge?

If you can NOT modify the interface and the window class:

- Inheritance
- The design pattern “decorator”

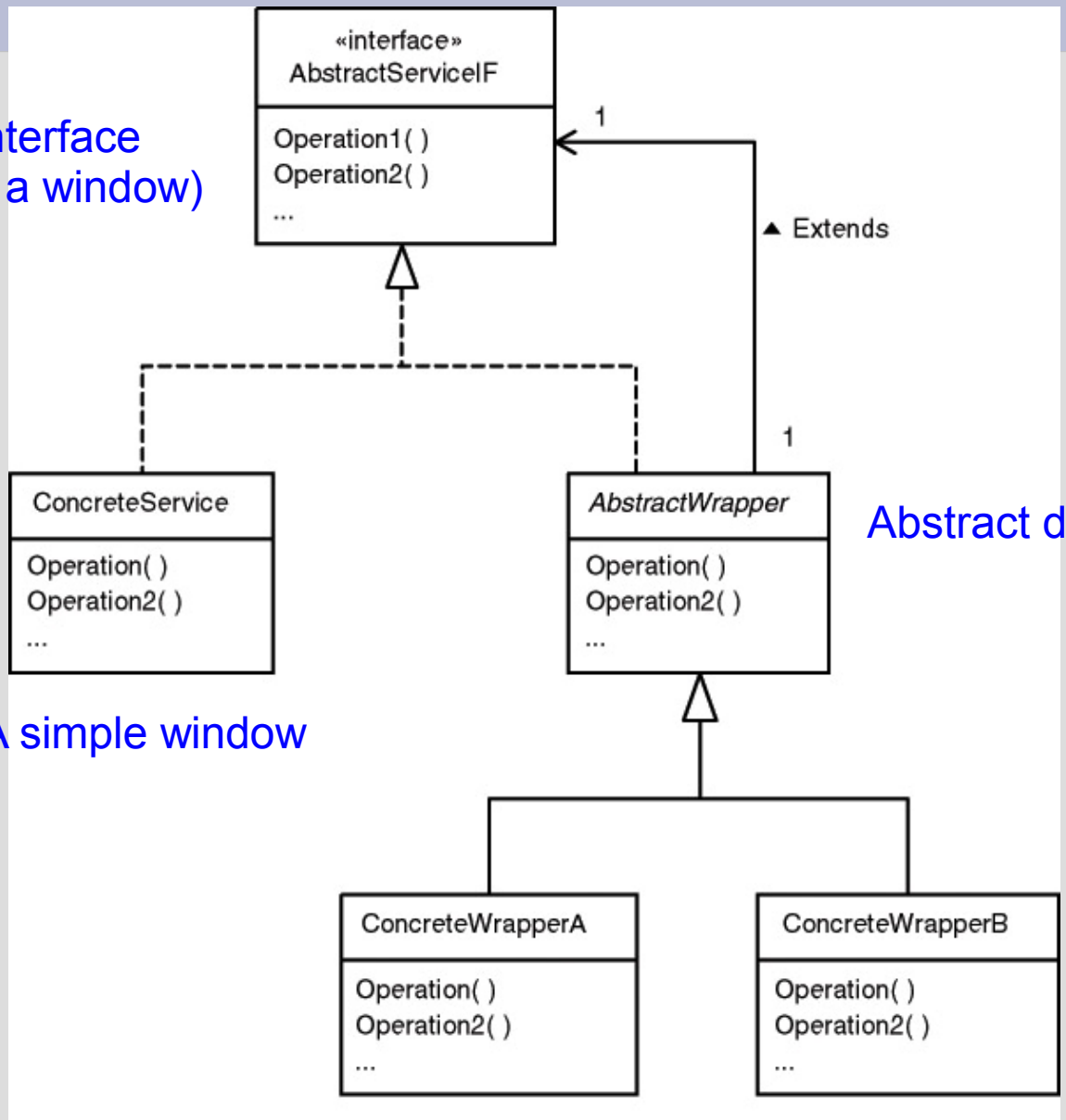
With inheritance



on aurait 2^n classes différentes !

“Decoration” design pattern

(interface of a window)



A simple window

Abstract decorator

Concreate decorator (as “window with special effects”)

- With “decoration” design pattern: O(n) classes.

Conclusion sur le patron « Décorateur »

- + Meilleure maintenance que des « if »
- + Less classes than with inheritance
- + Possible to modify a decoration dynamically
- - Behavior not clear...

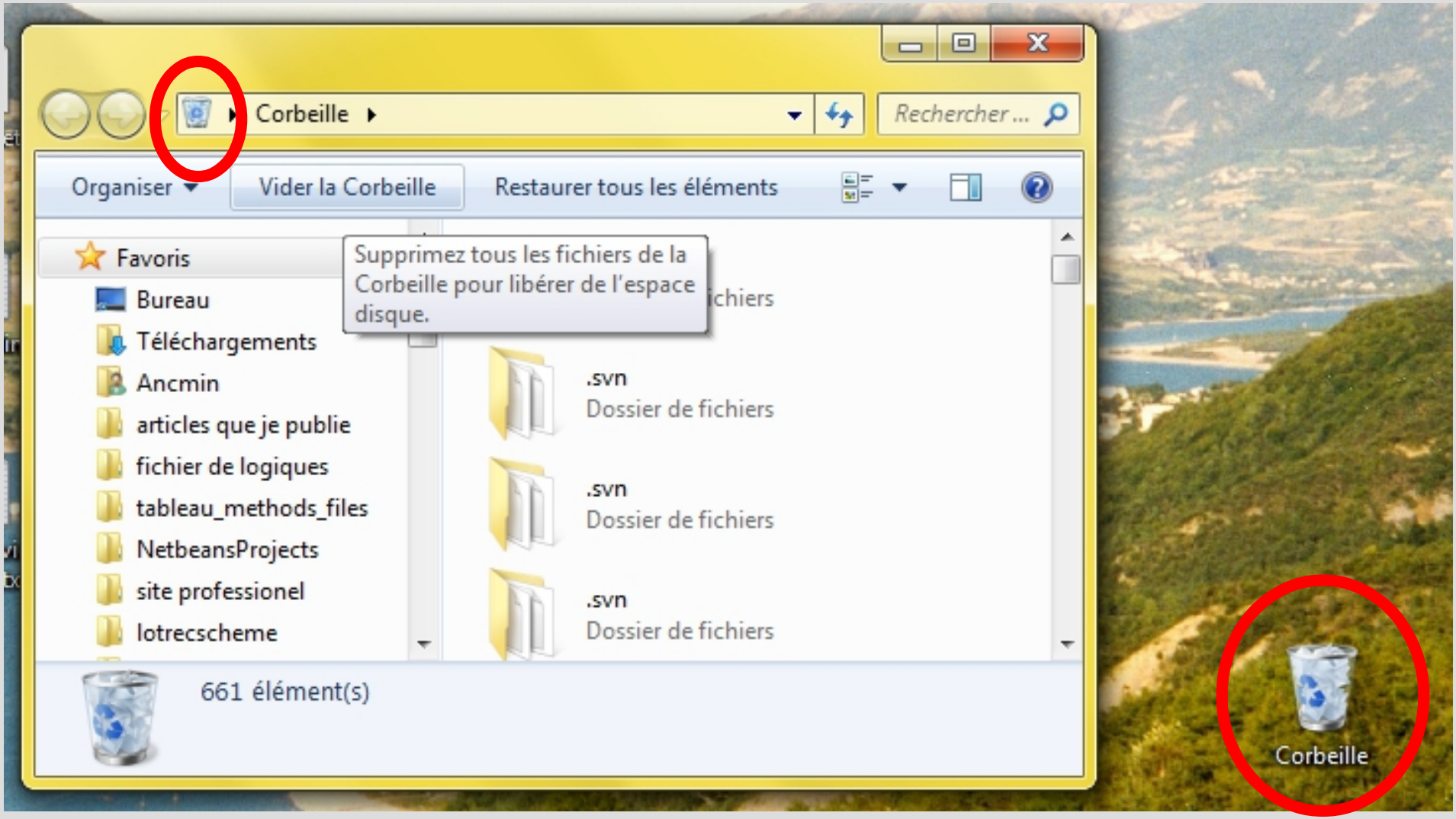
  new FenetreBordure(new FenetreFond(f))

  ~ new FenetreFond(new FenetreBordure(f)) ?

Behaviour design patterns

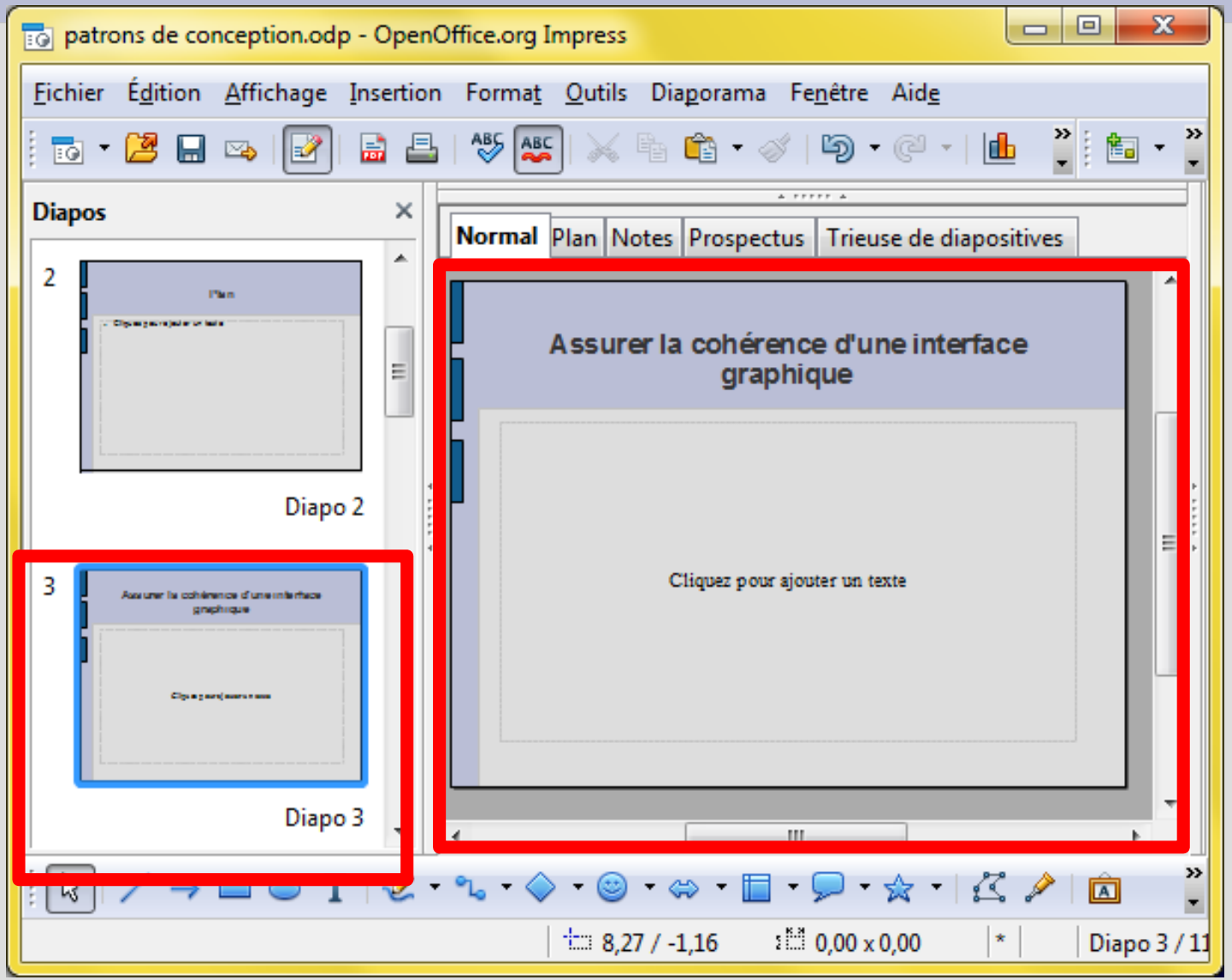
- Dependency inversion principle
- Data converter
- Cancel feature
- Change the behaviour of an object

Need: to refresh the graphical user interface



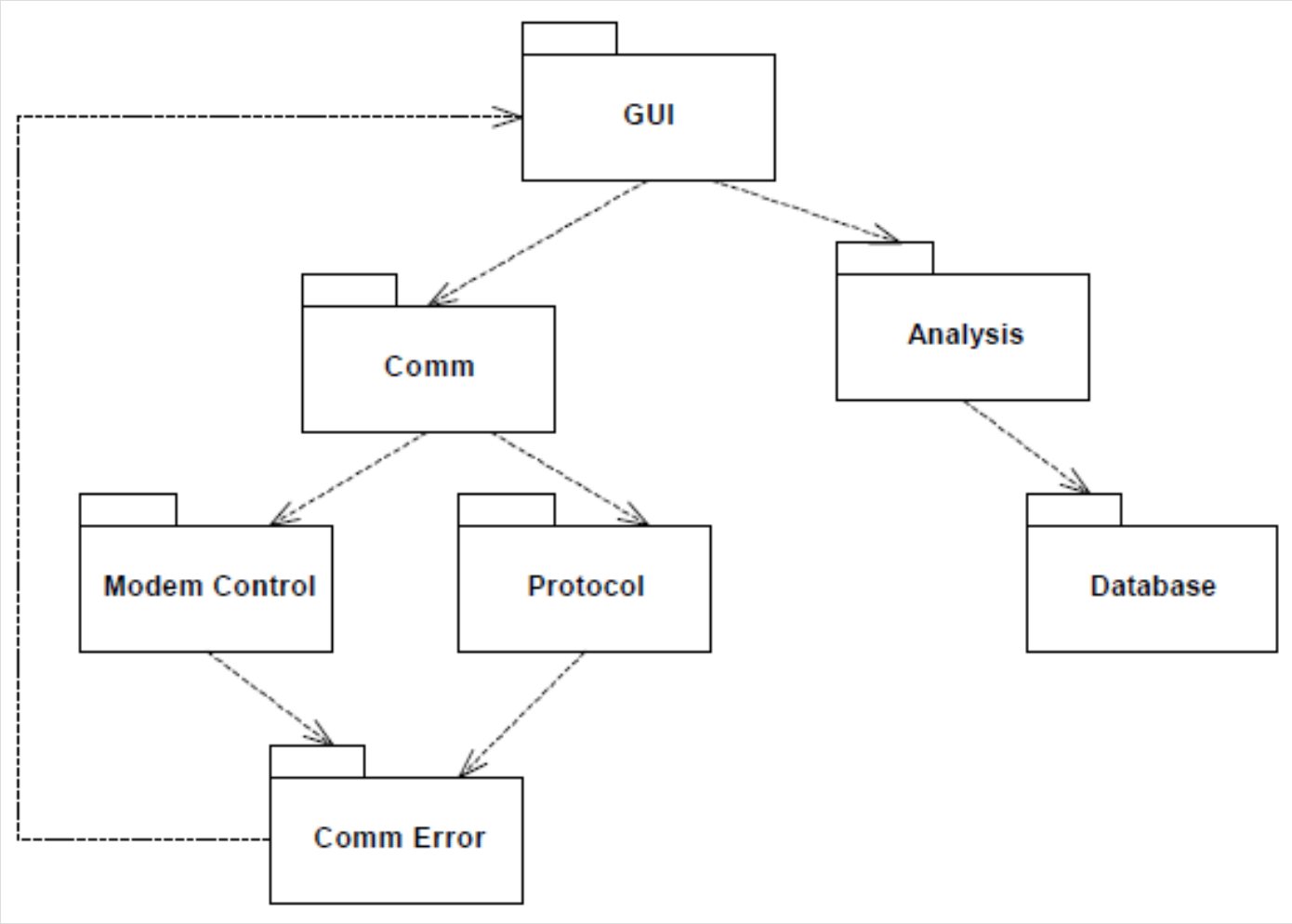
Source : interface de Windows 7

Need: to refresh the graphical user interface

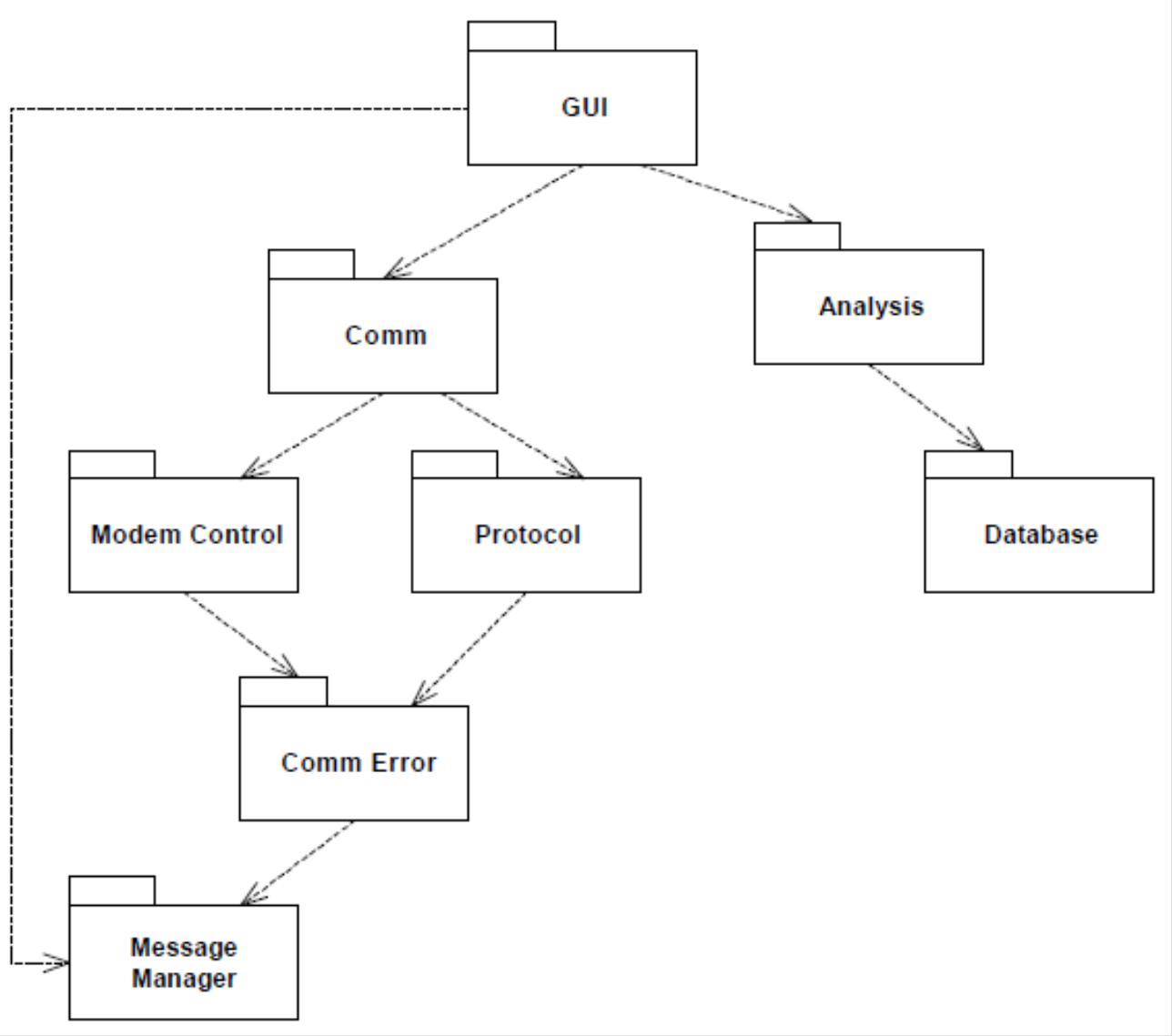


Source : OpenOffice.org Impress

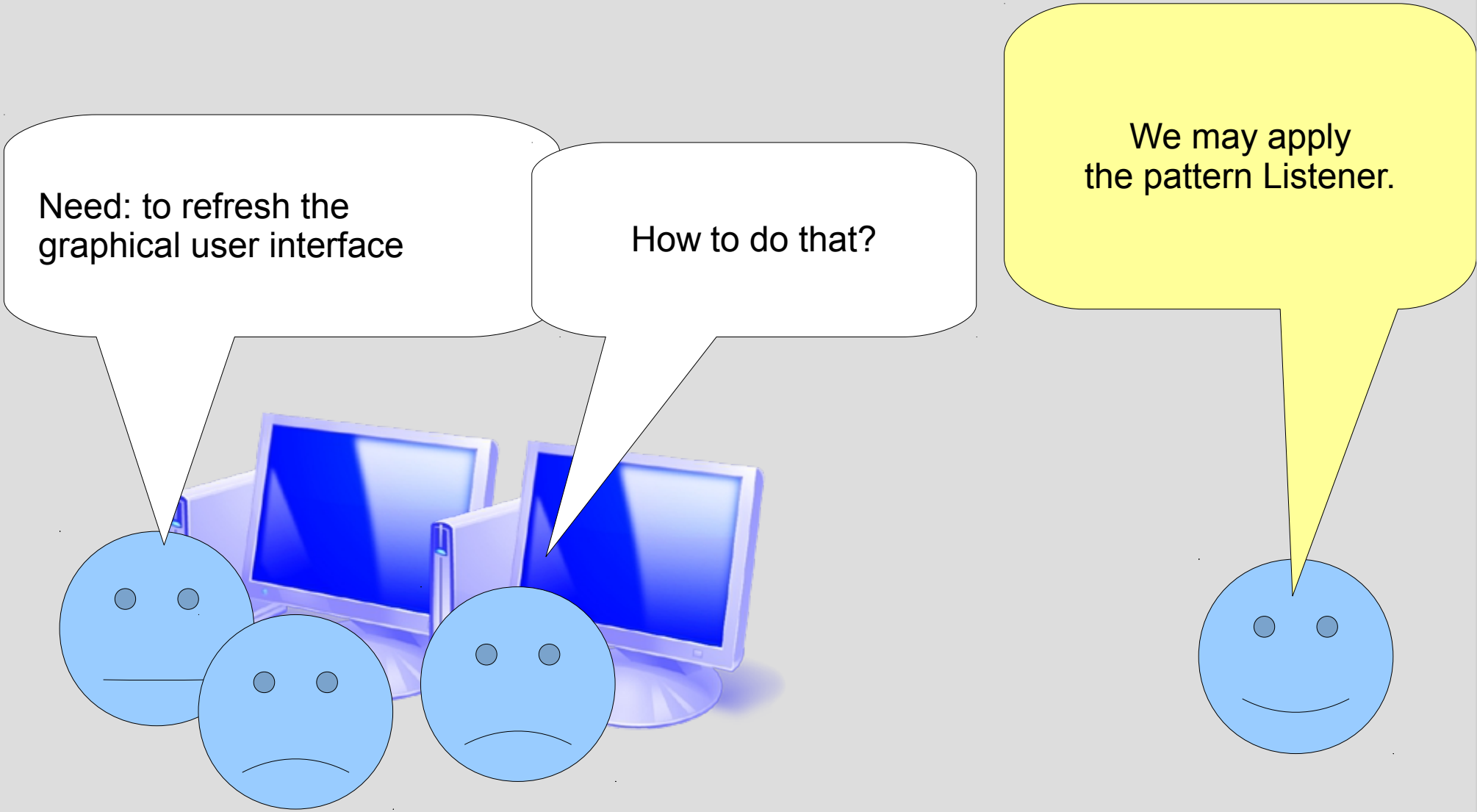
Problem


GUI
Comm
Analysis
Modem Control
Protocol
Database
Comm Error

Solution


GUI
Comm
Analysis
Modem Control
Protocol
Database
Comm Error
Message Manager

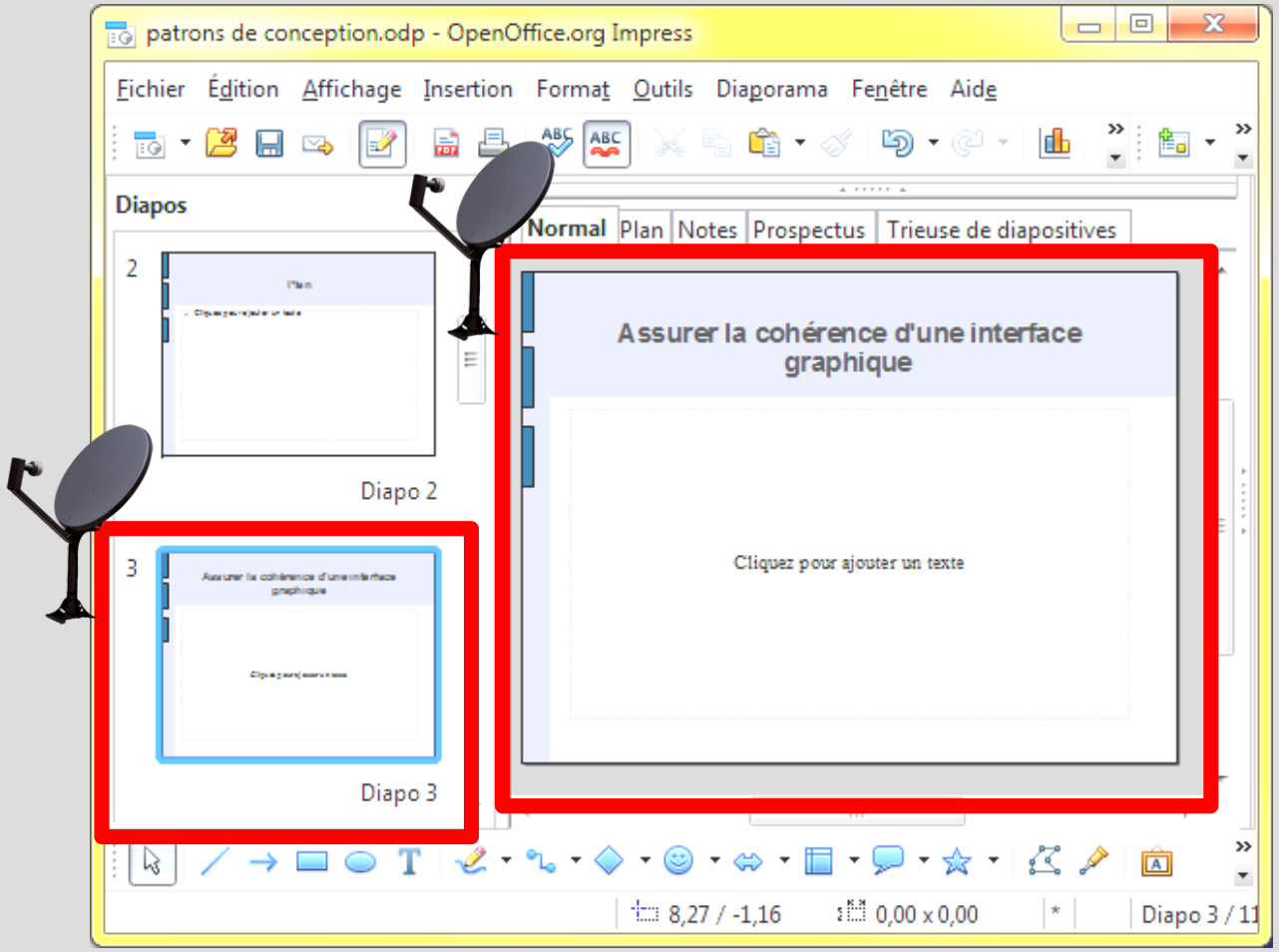
Solution: Listener



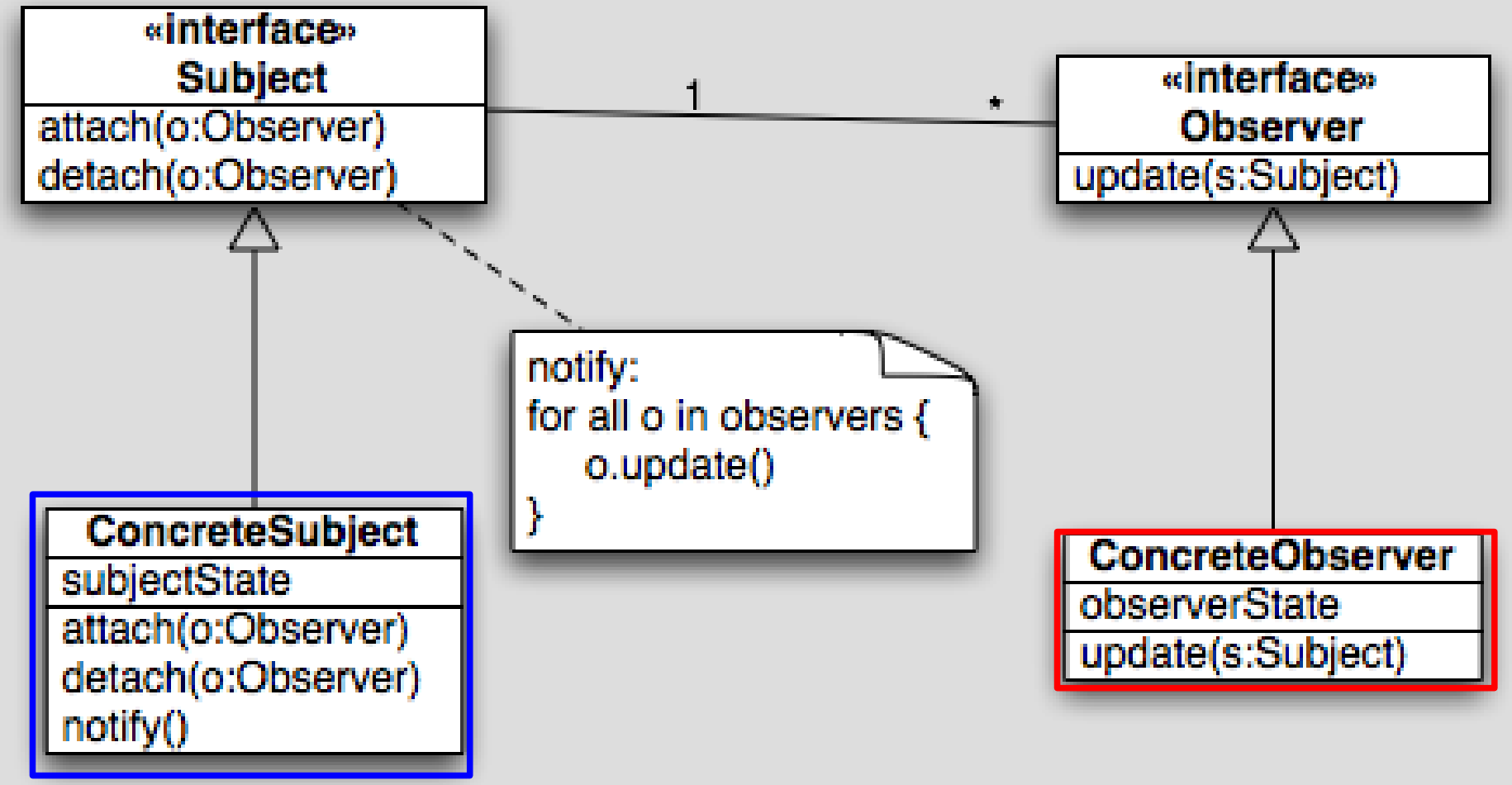
Listener

Data

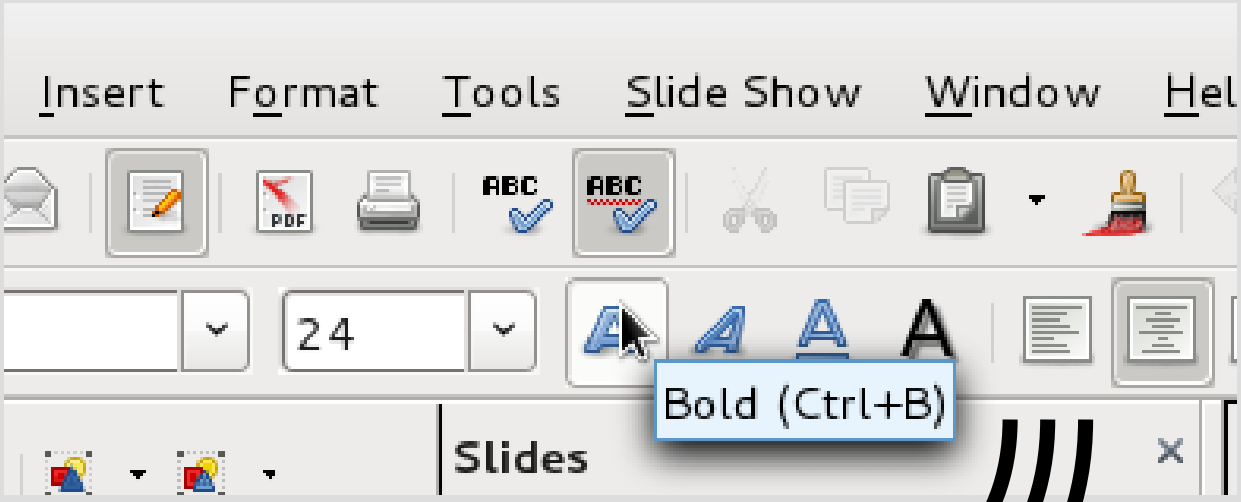
The two zones (listeners) **listen** the data (subject).



Listener


«interface»
Subject
attach(o:Observer)
detach(o:Observer)
1
*
«interface»
Observer
update(s:Subject)
notify:
for all o in observers {
o.update()
}
ConcreteSubject
subjectState
attach(o:Observer)
detach(o:Observer)
notify()
ConcreteObserver
observerState
update(s:Subject)

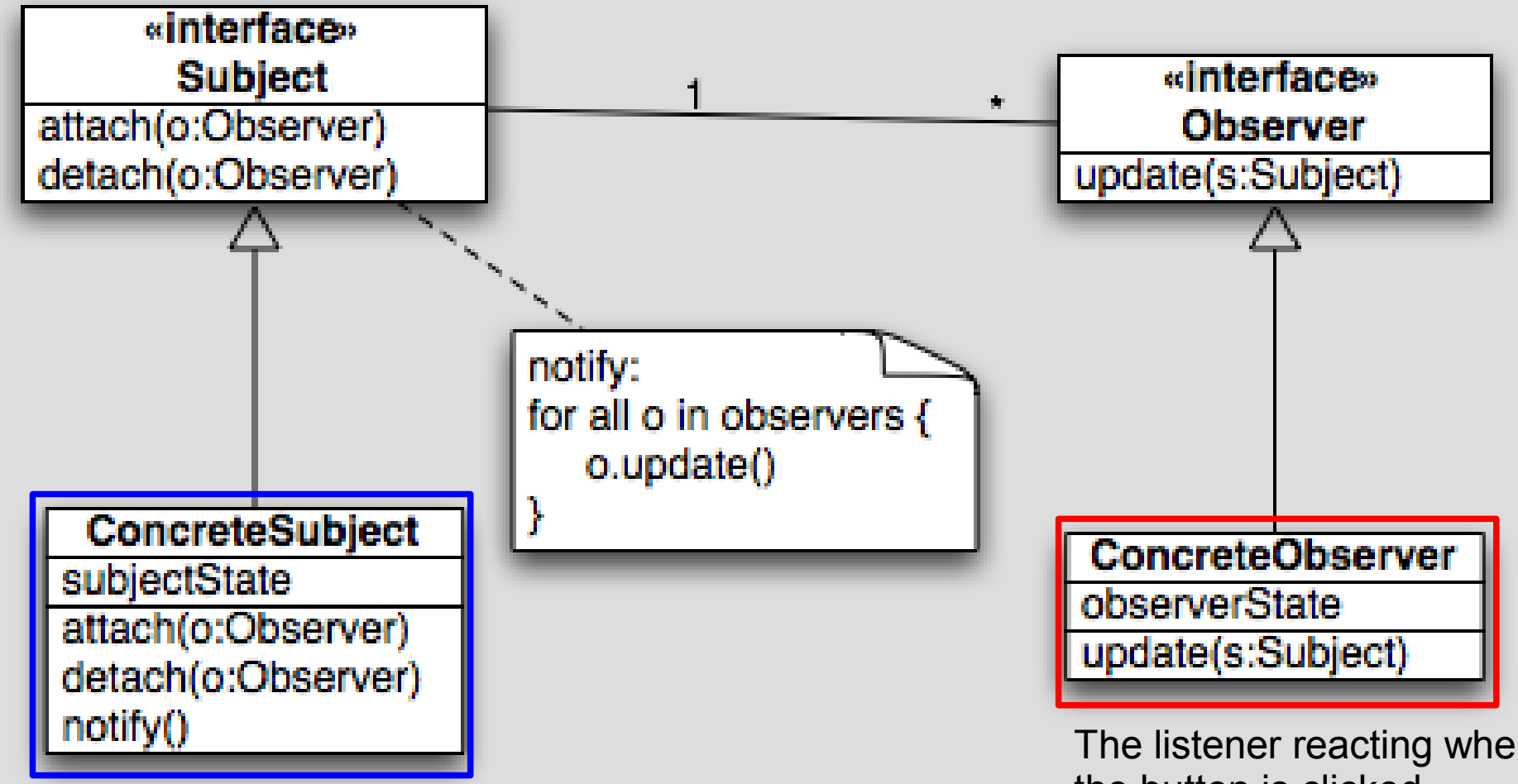
Other application of the Listener design pattern: input/output



Click!

Listener

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



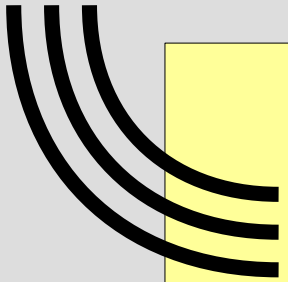
A button for instance

The listener reacting when the button is clicked

Model - view - controller

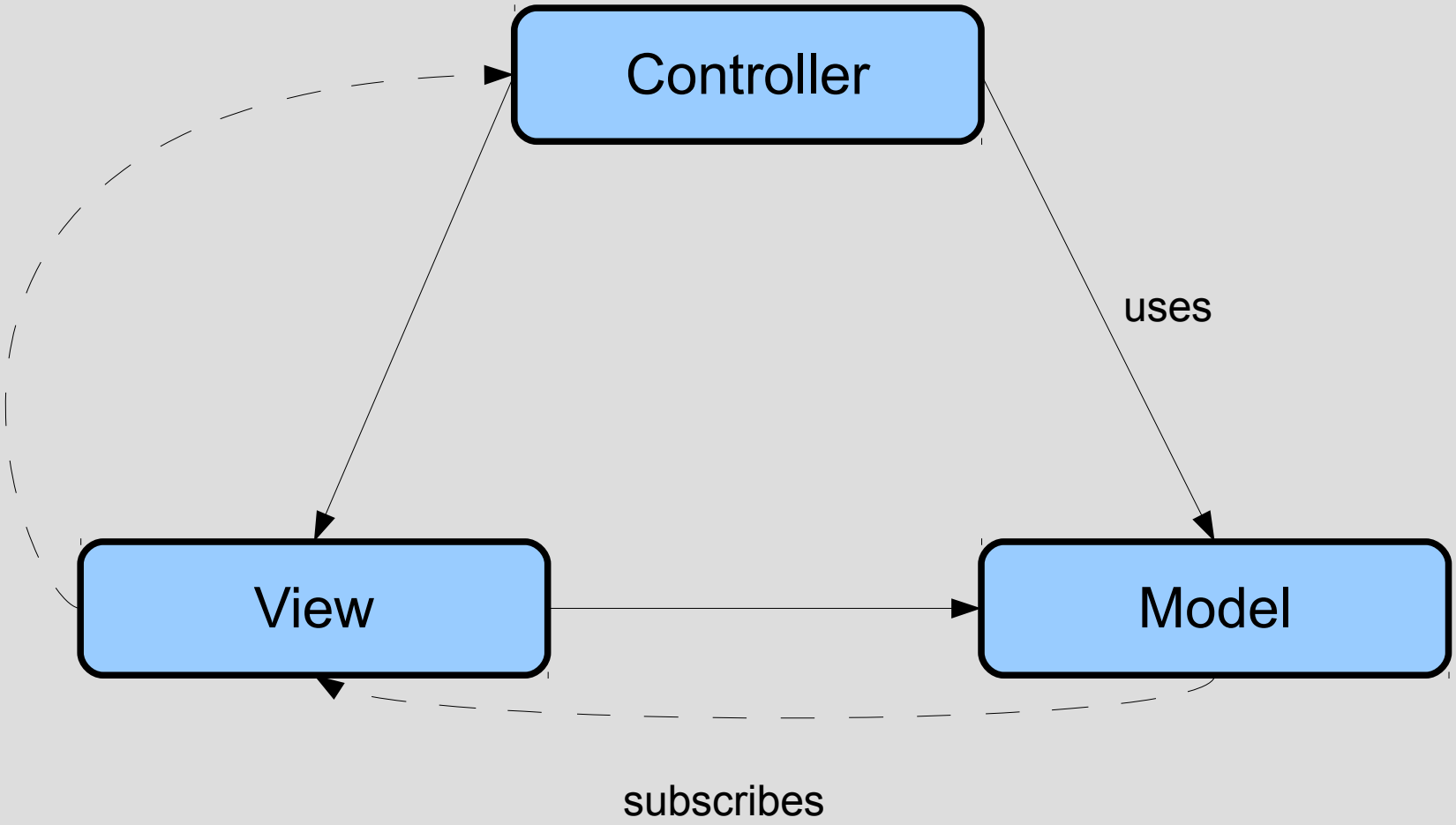
Controller

View

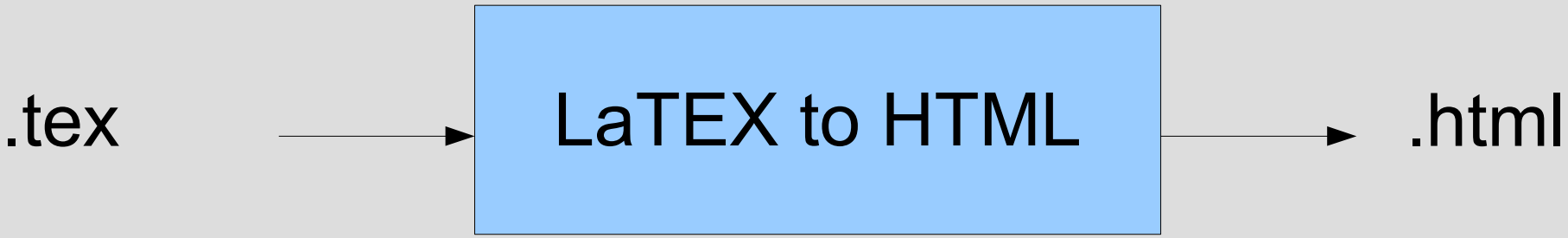


Model
(data)

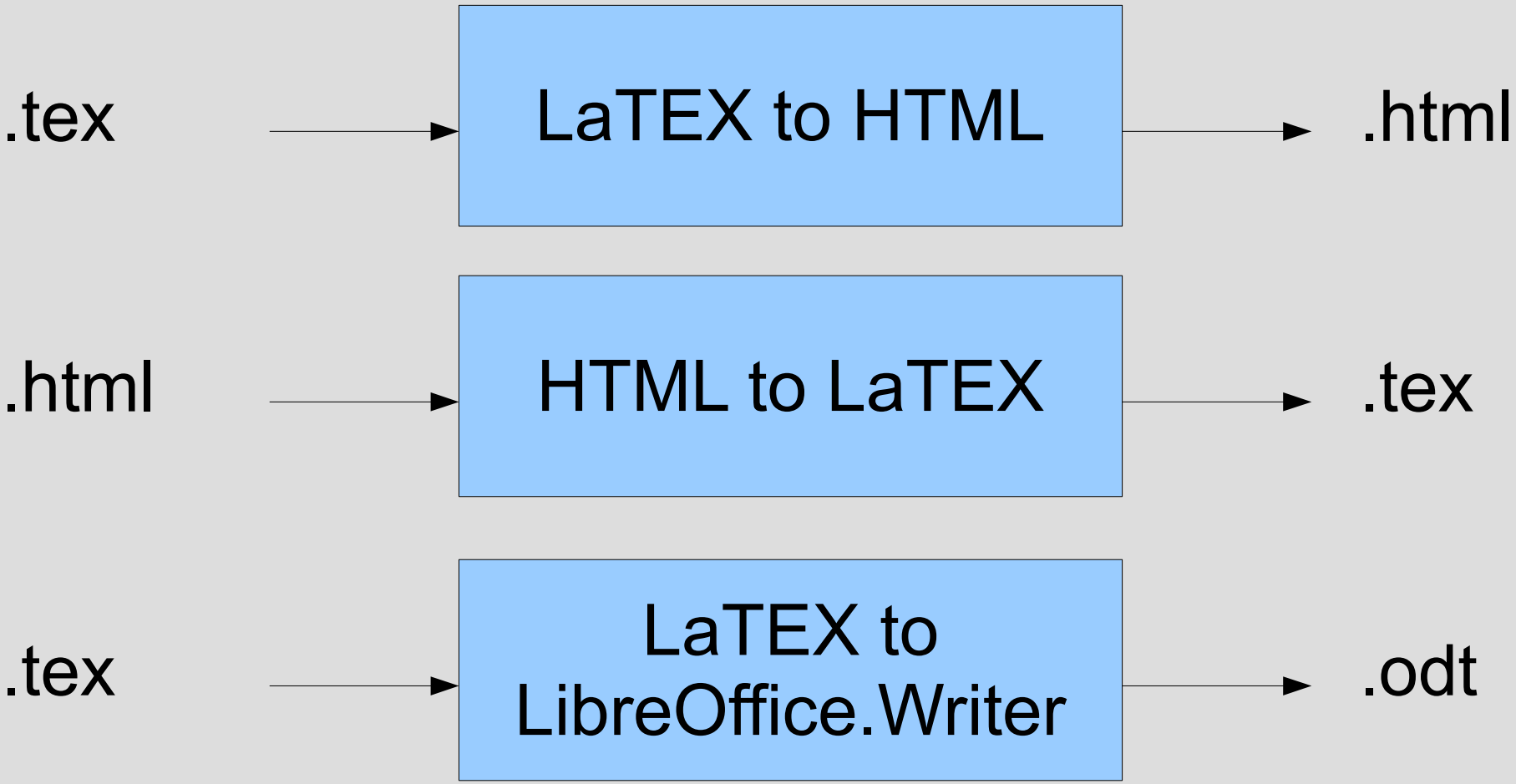
Model - view - controller



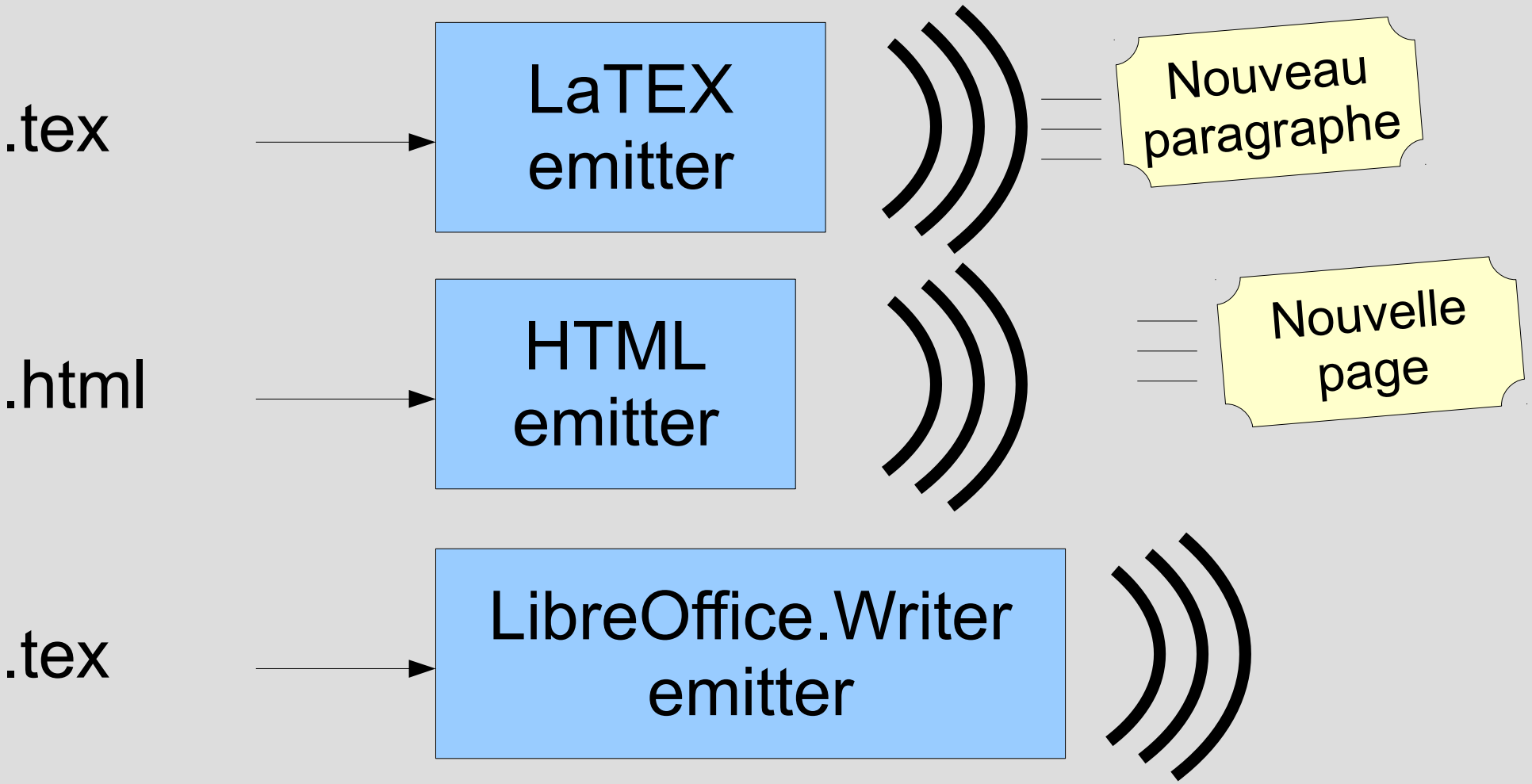
Another application for Listener: data converter



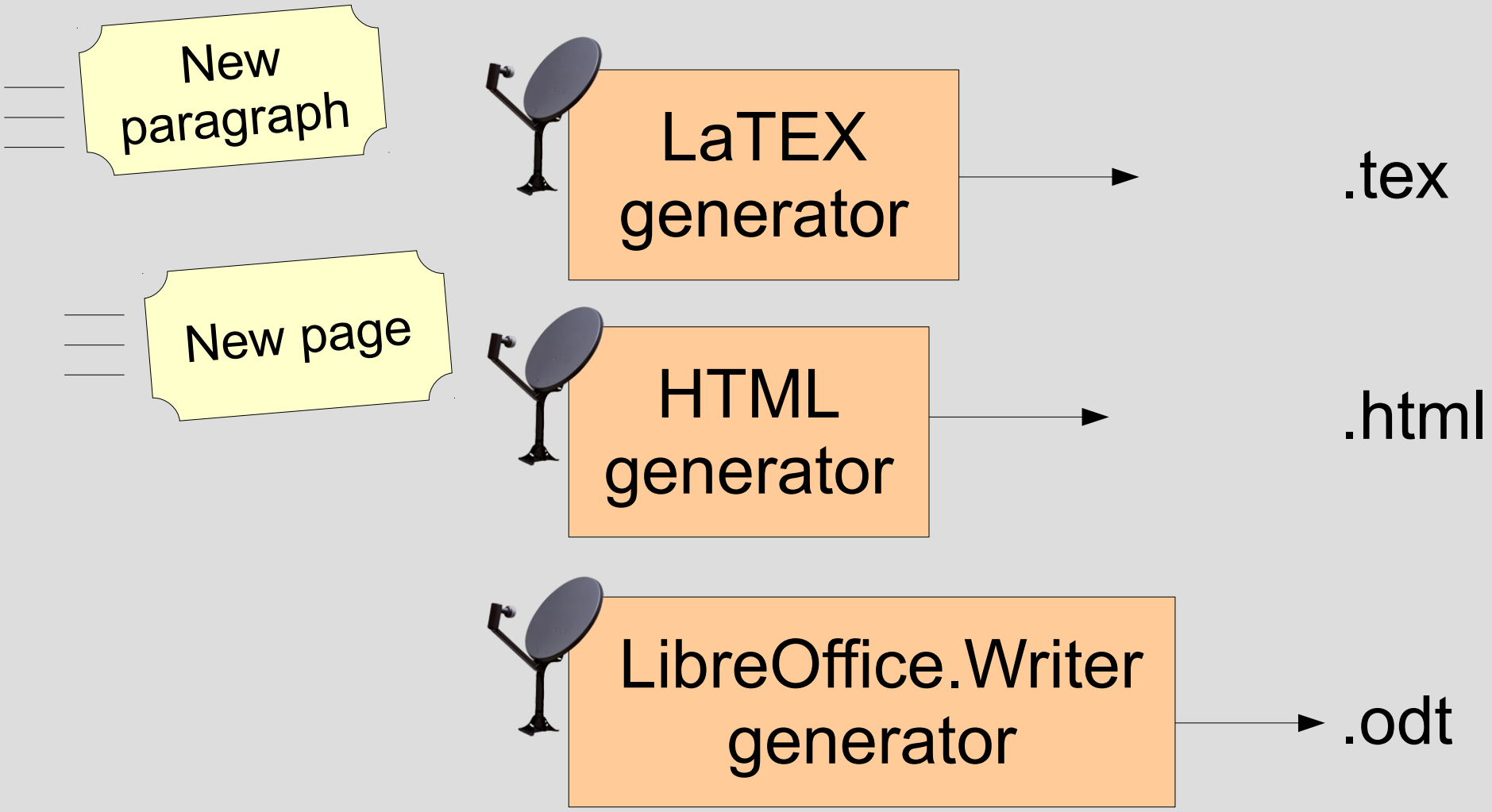
Another application for Listener: data converter



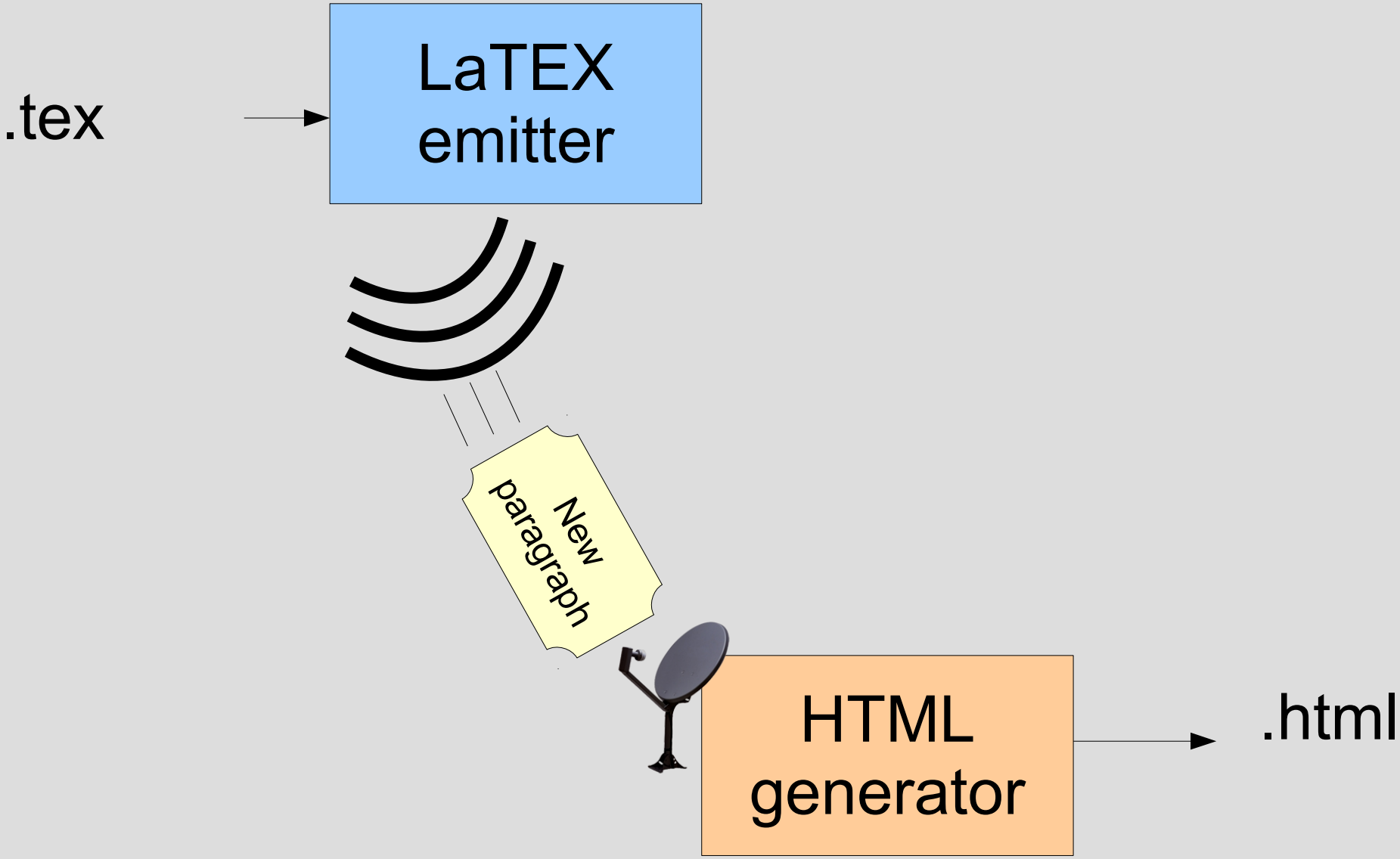
Another application for Listener: data converter



Another application for Listener: data converter



Another application for Listener: data converter

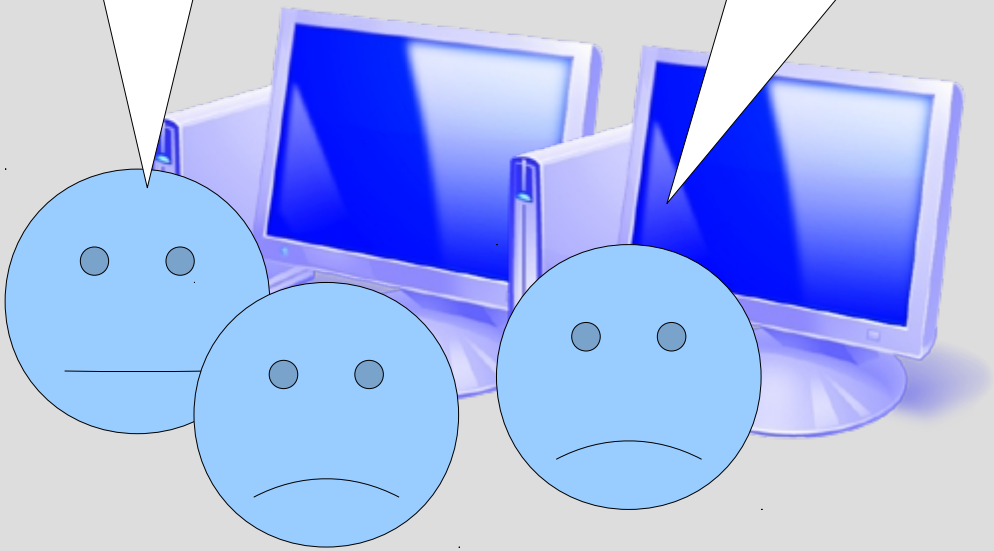


Design pattern “Command”

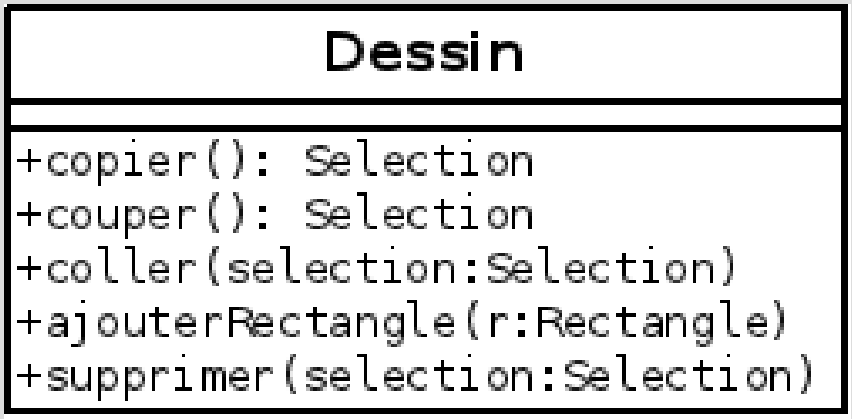
We want to be able to cancel.

How to do that?

We may apply the design pattern “command”.

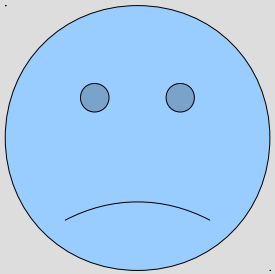


Is the principle “action = operation” good?

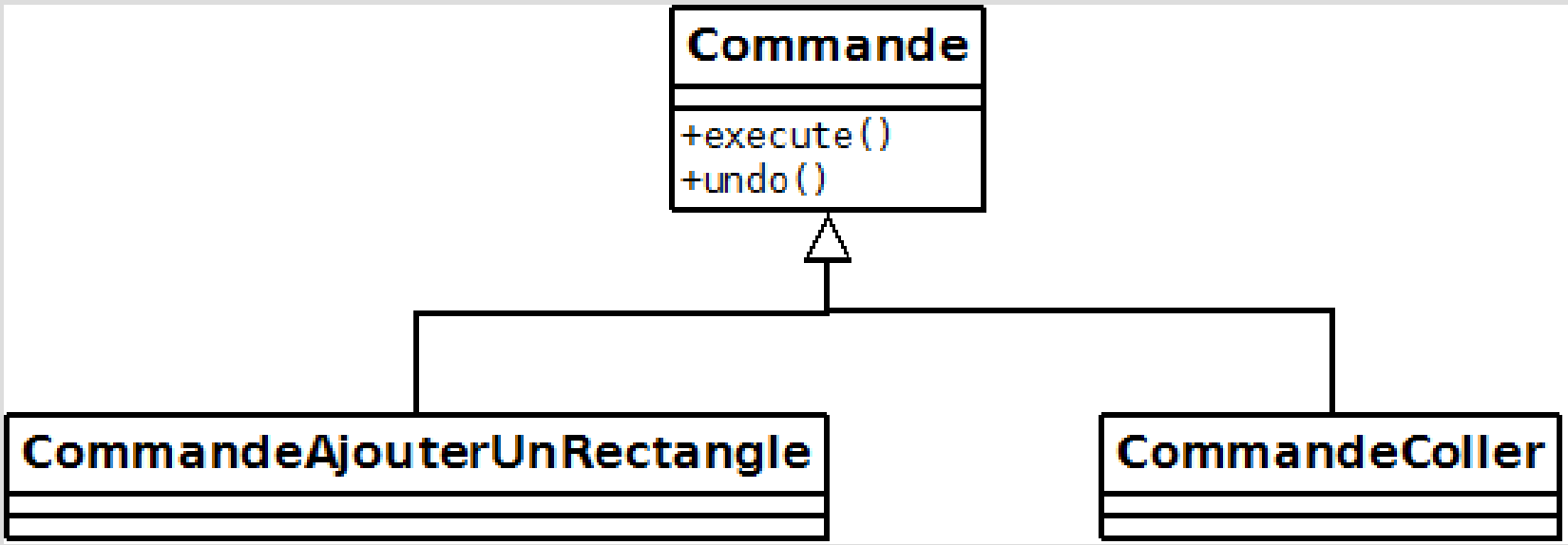


Problem

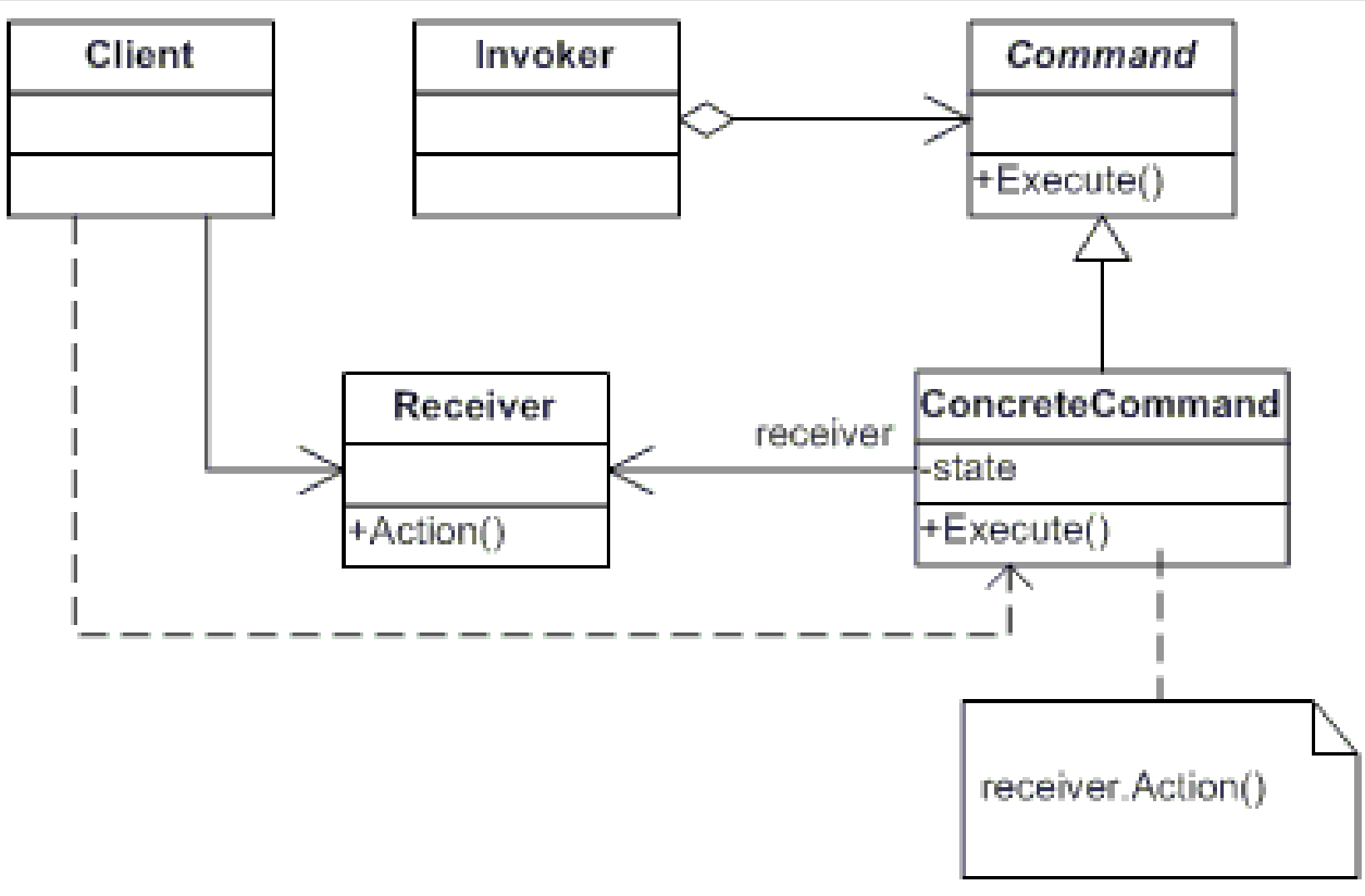
- Cancel?
- Save macros?
- Too many responsibilities for the class “Dessin”



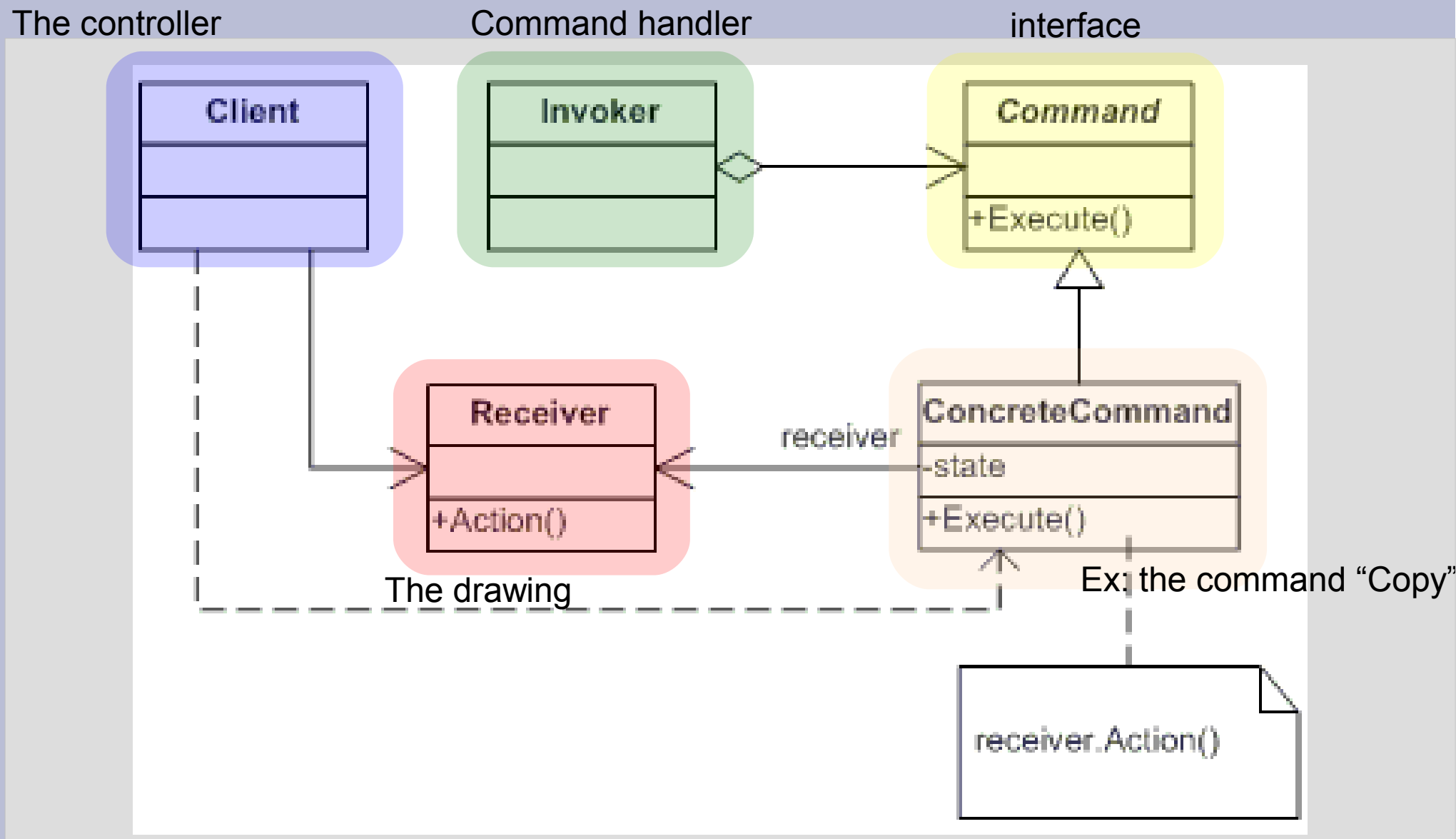
Solution: design pattern “command”



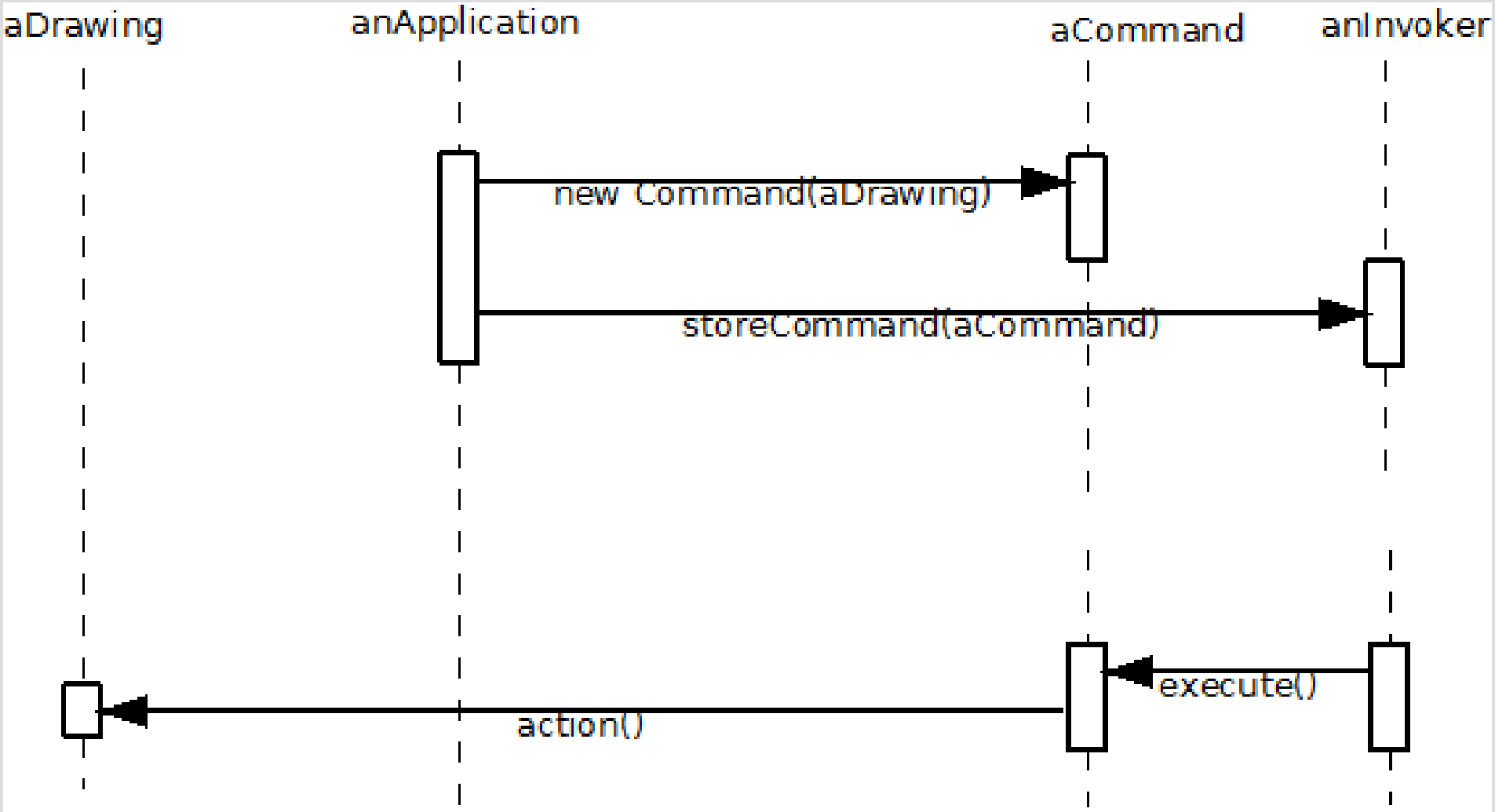
Solution: design pattern “command”



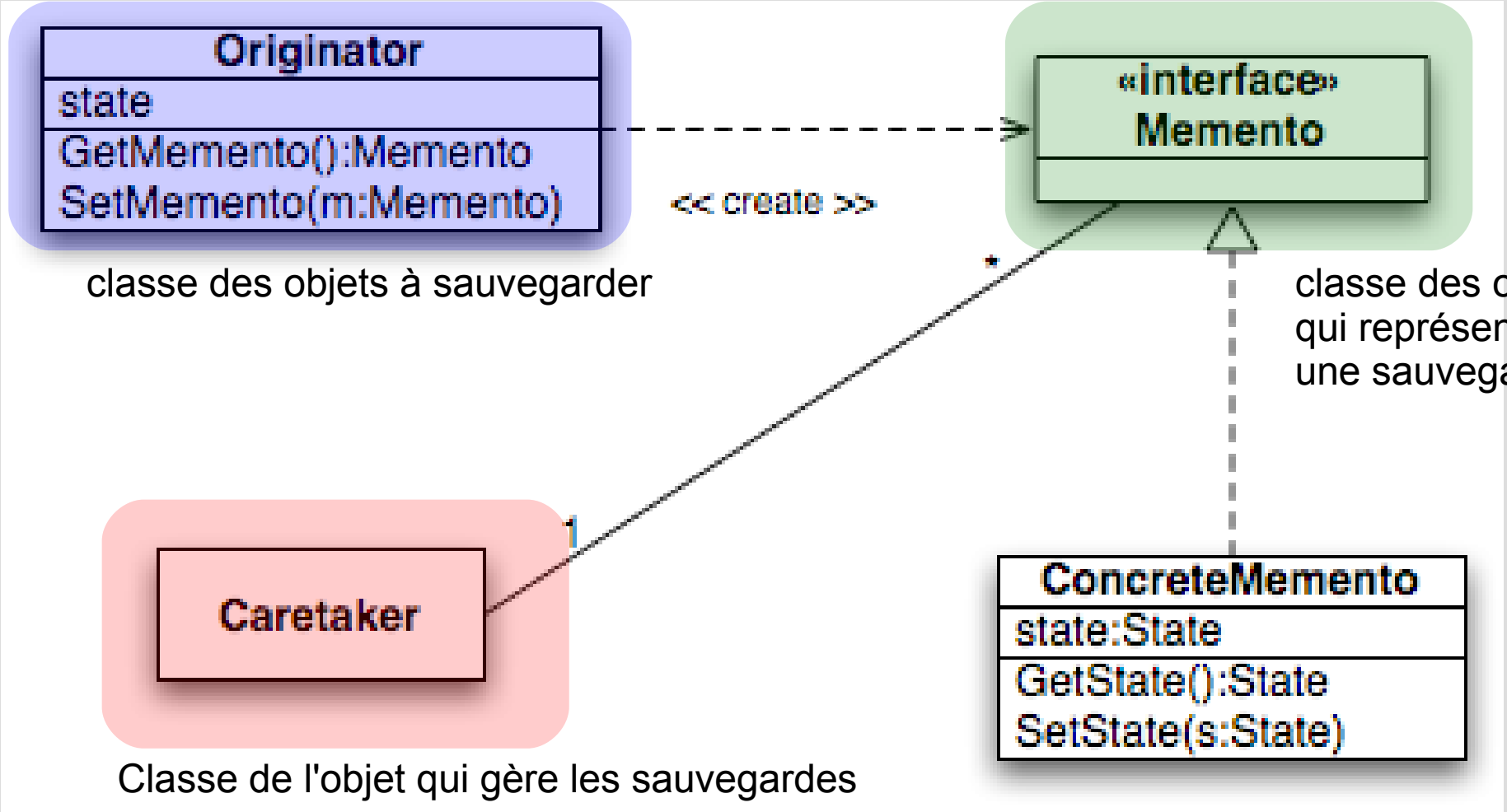
Solution: design pattern “command”



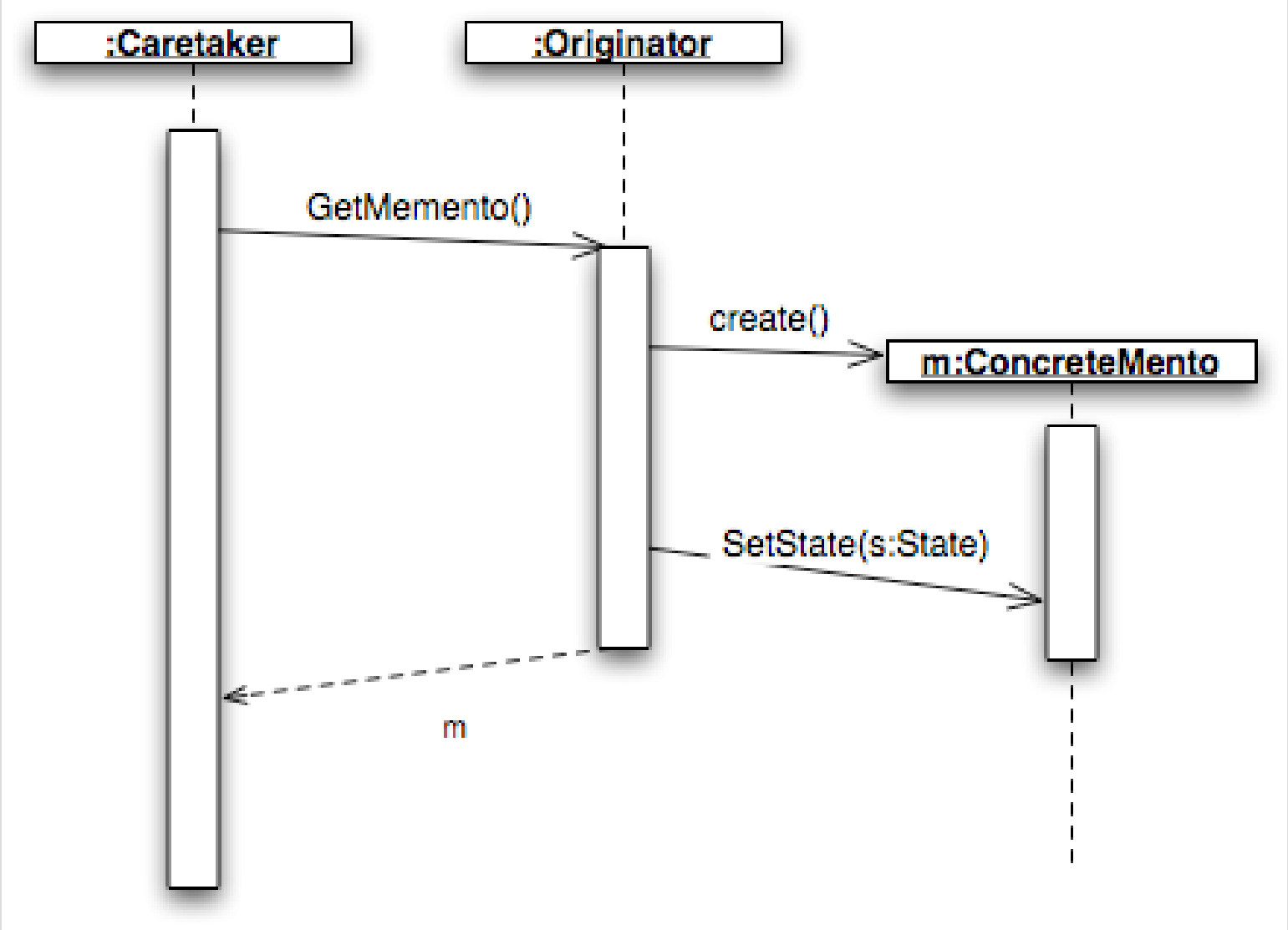
Solution: design pattern “command”



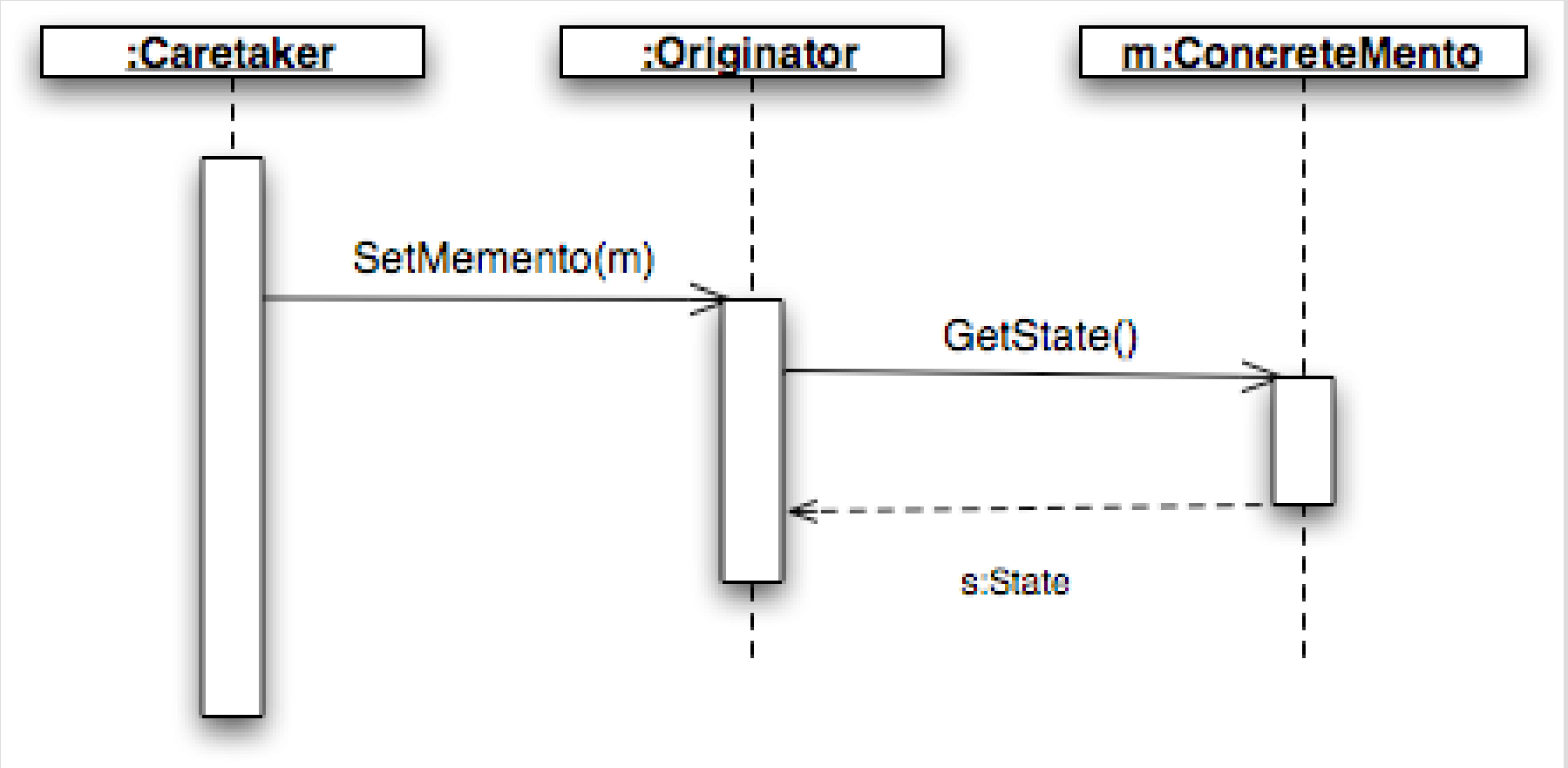
Memento


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

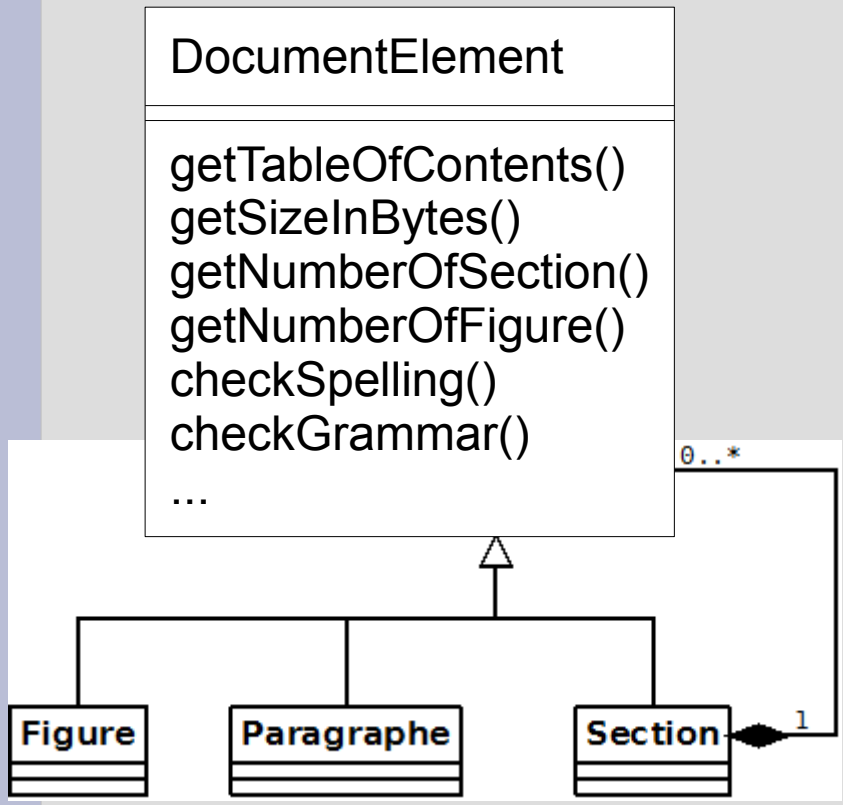
Memento : saving

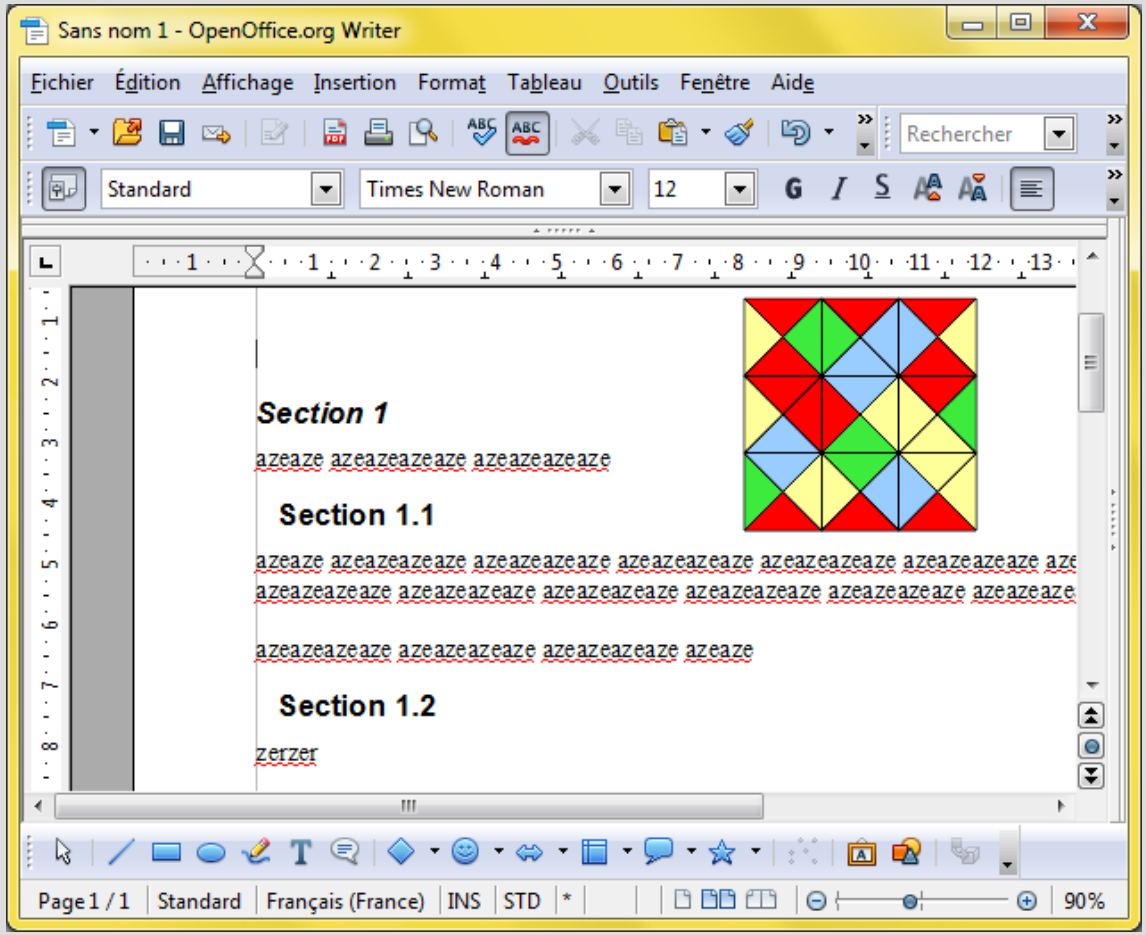


Memento : restoration



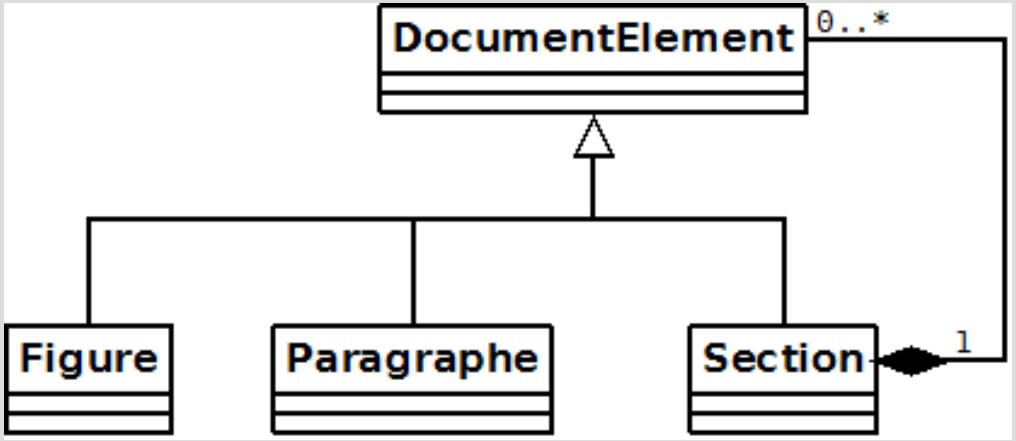
Operations on recursive structures

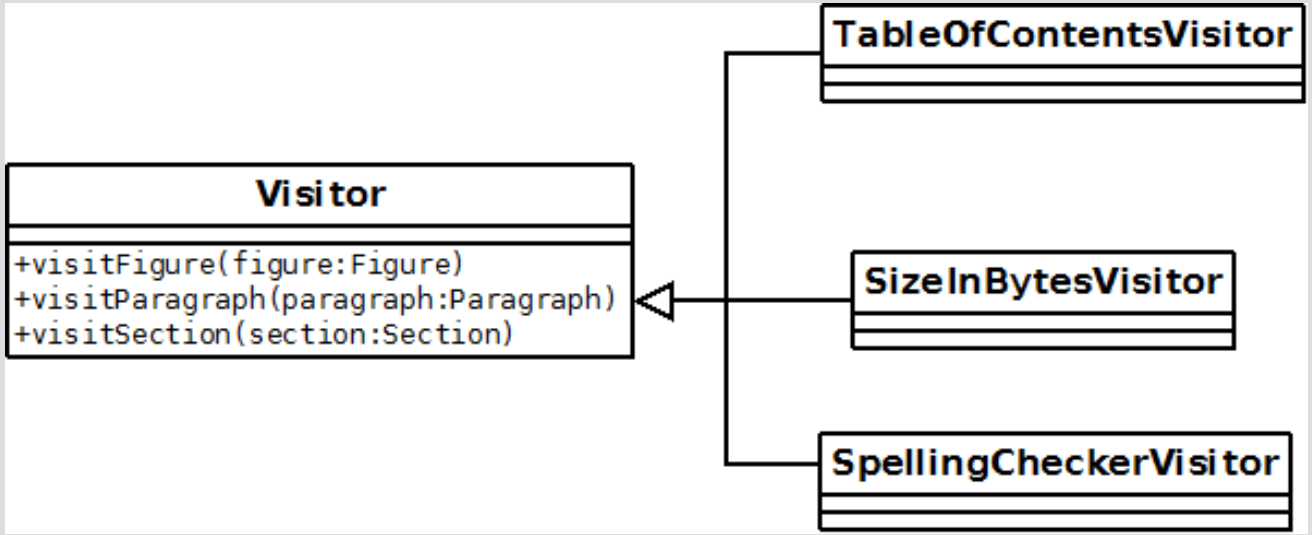




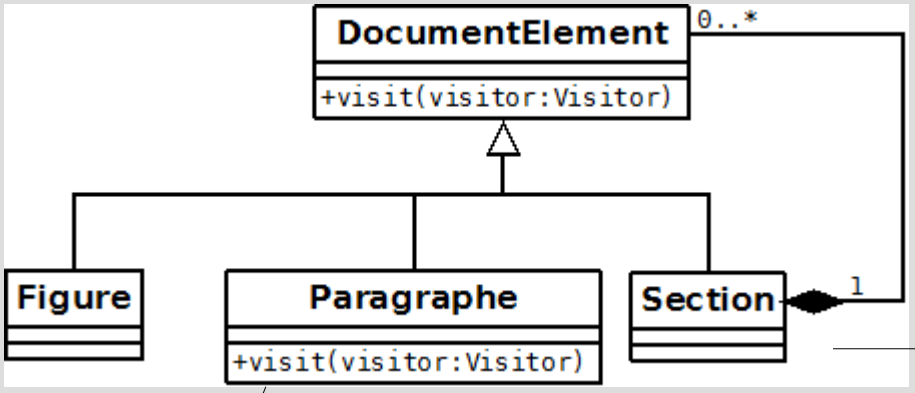
Problem: classes become really **big!**

Solution: Visitor





Solution: Visitor




```
Class Paragraph
{

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

}
```

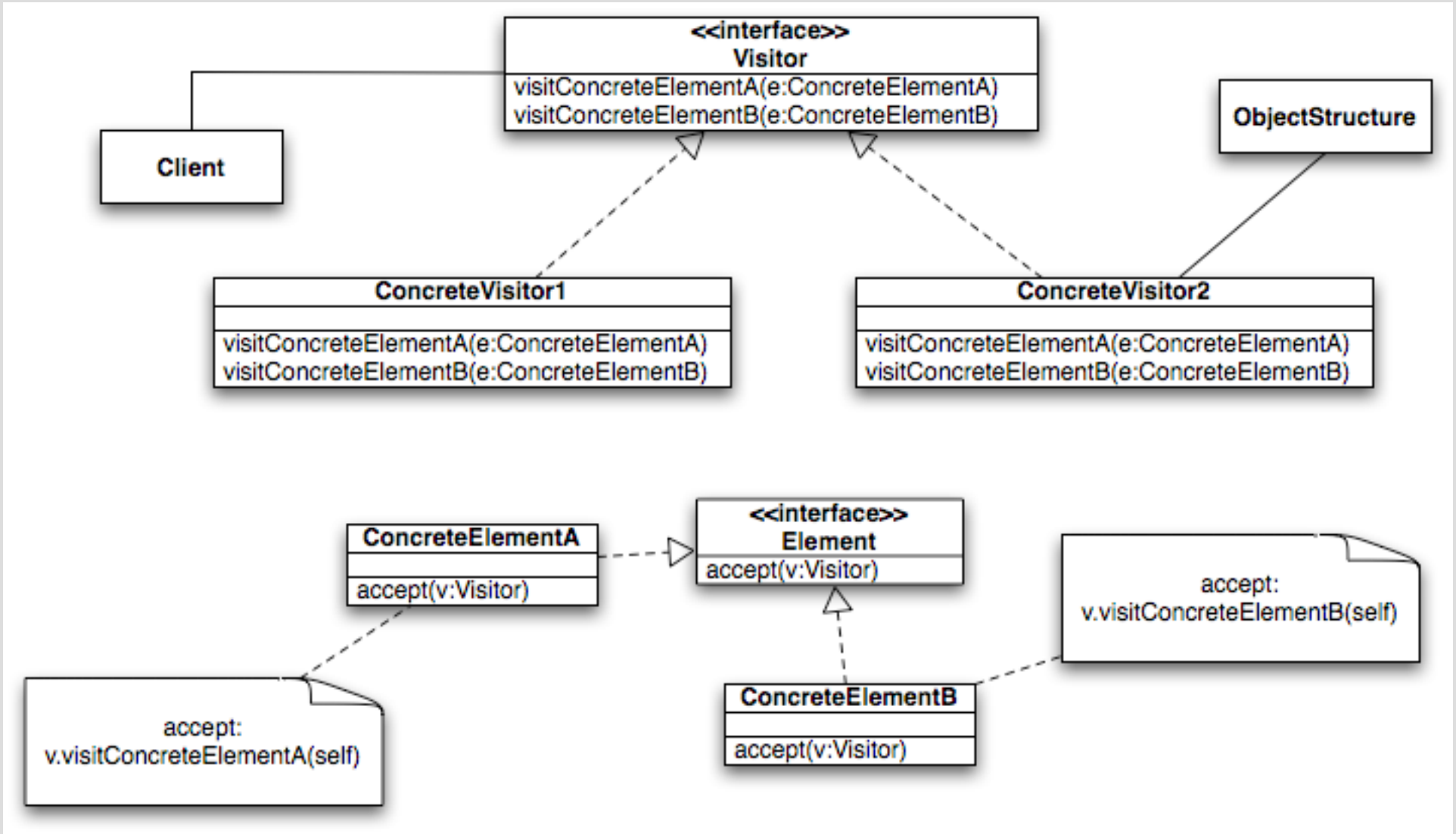
```
Class Section
{

   public void visit(Visitor visitor)
   {
      visitor.visitSection(this);
      for(DocumentElement el : this)
      {
            el.visit(visitor);
      }
   }

}
```

Solution: Visitor

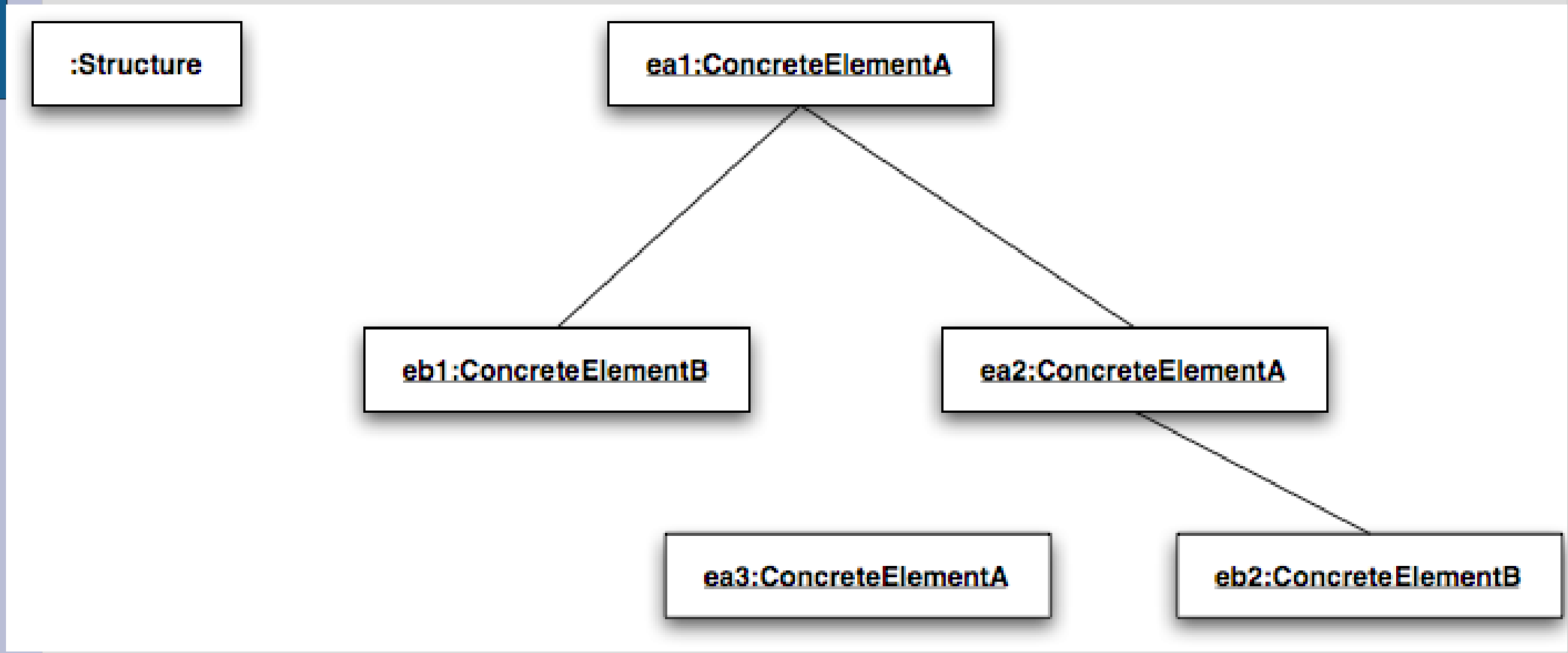
- Each class has a responsibility

- Depends on the data
- Data classes must have public accessors.

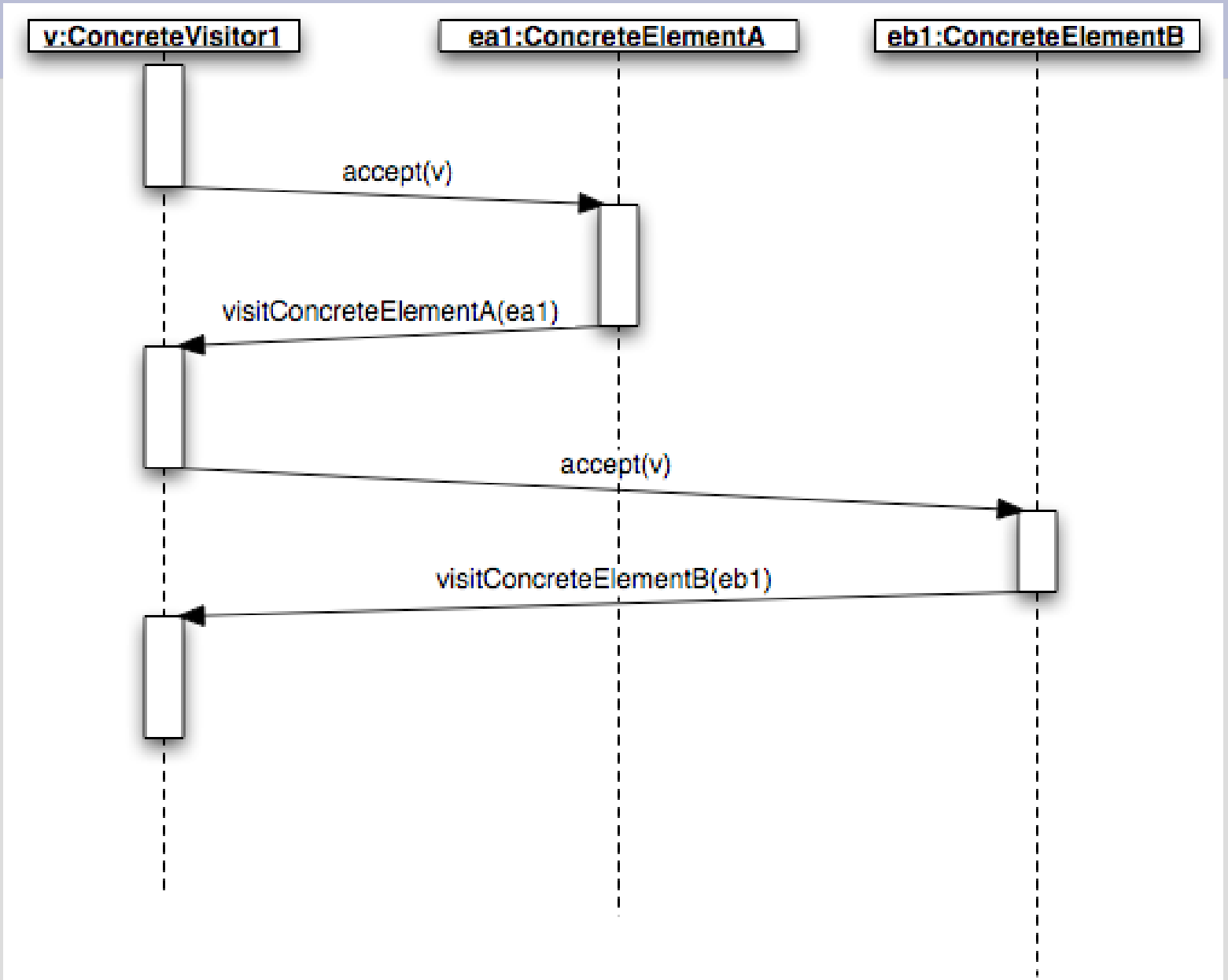
Visitor


<<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)

Object diagram



Visitor


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

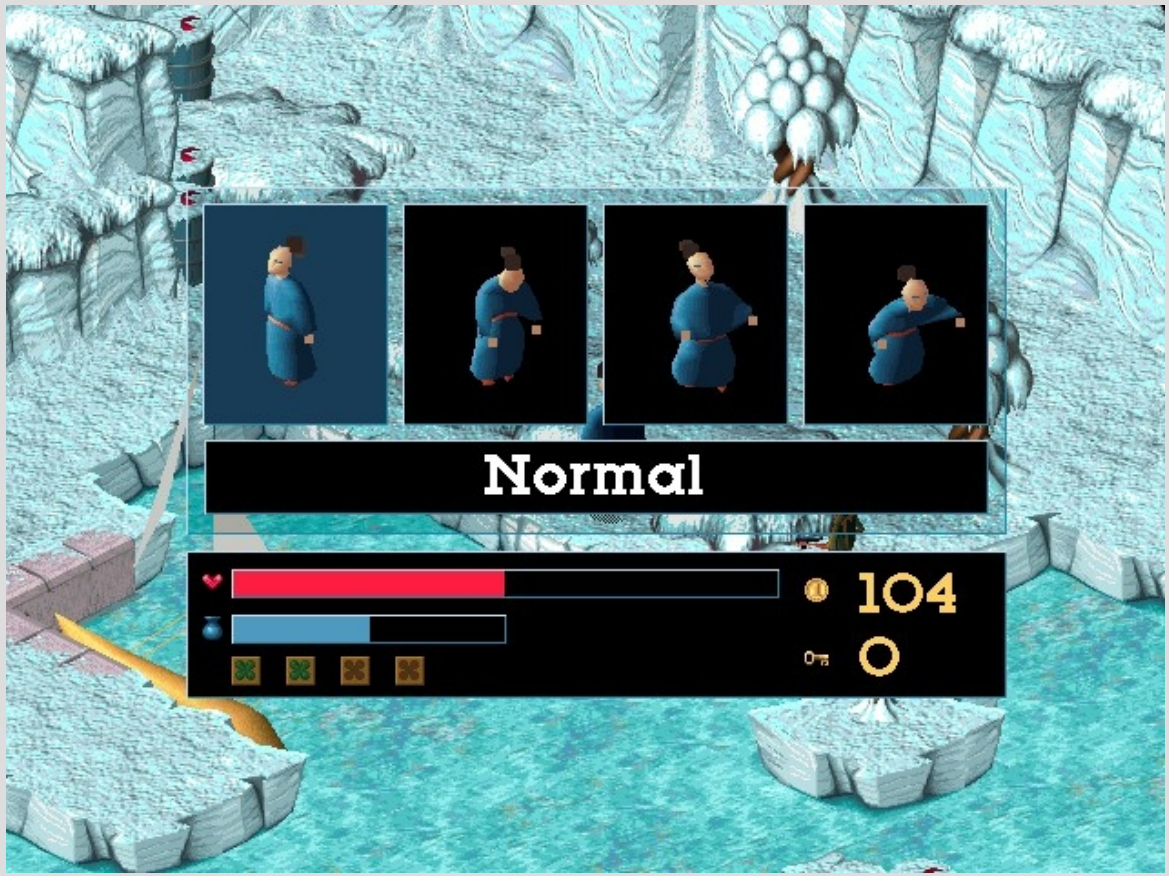
Example of other applications of the Visitor pattern

- Music score editor:
  Number of notes, display the score, etc.
- Proof assistant:
  Display the proof, check the proof, etc.
- 3D software
  to display the squeleton, compute the weight, etc.

Implémenter plusieurs algorithmes : patron de conception « Stratégie »

Need of several behaviour of the hero

```
hero.move()
```



Source : Little Big Adventure 2

Need of several sorting algorithms

```
array.sort()
```

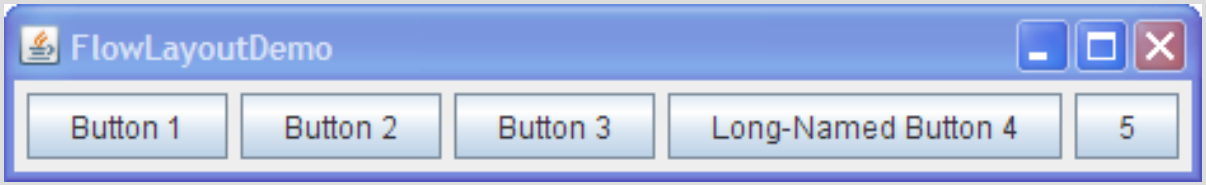
BubbleSort ?

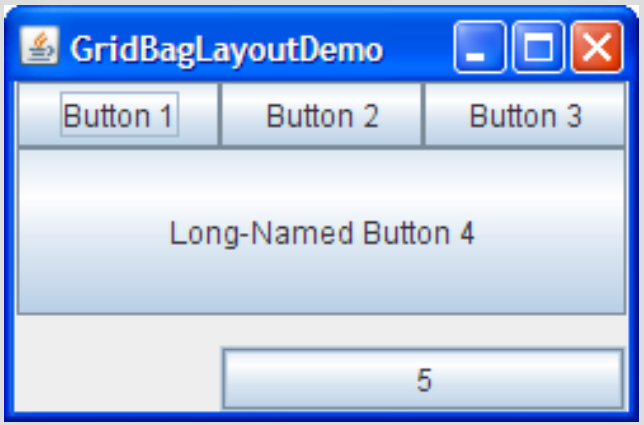
QuickSort ?

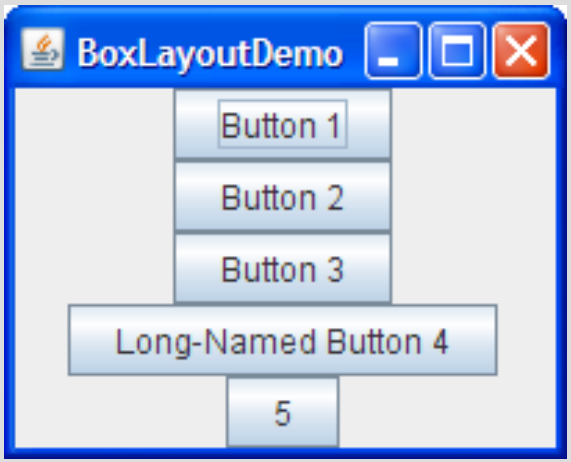
HeapSort ?

Need of several layout algorithms

```
container.doLayout()
```

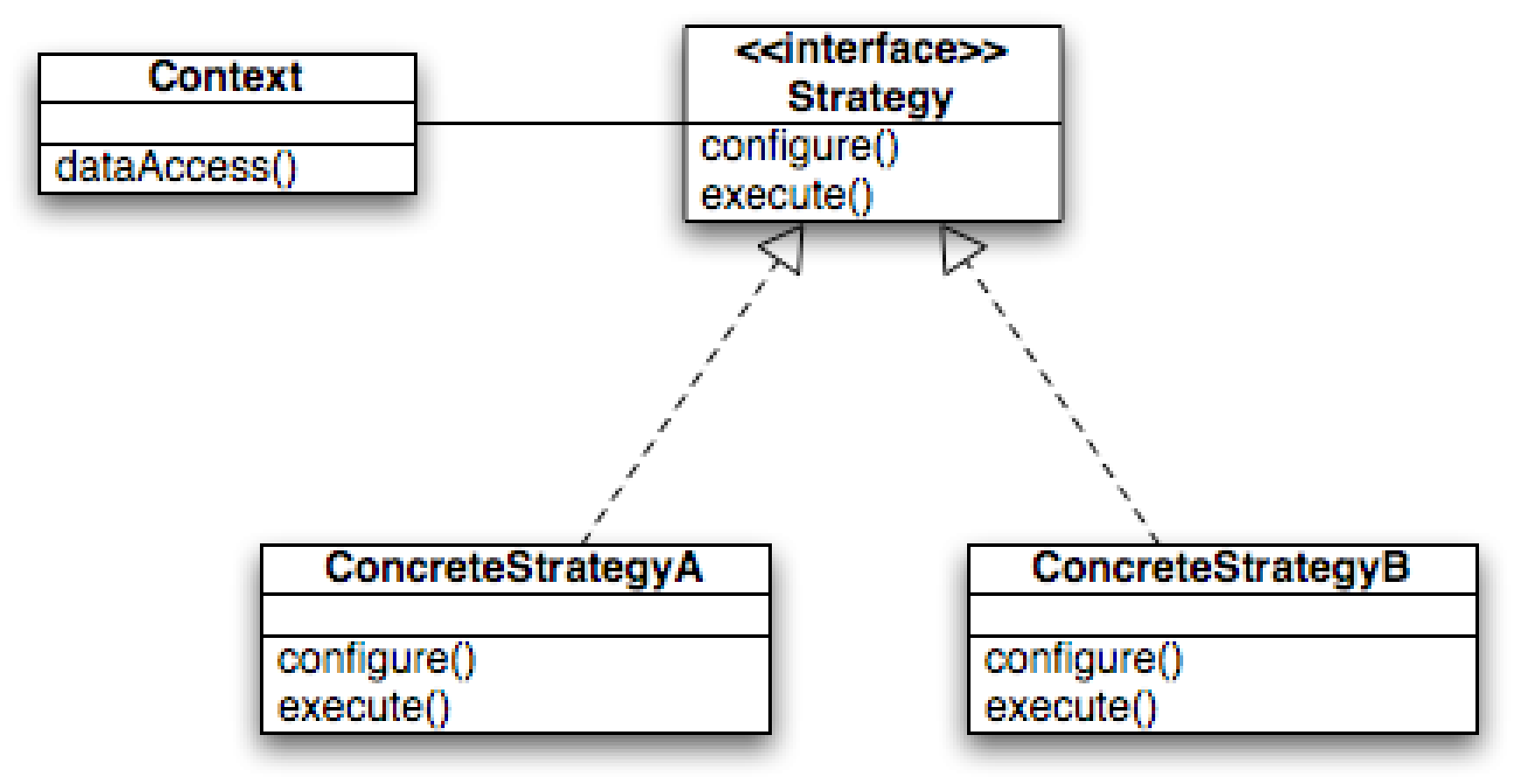




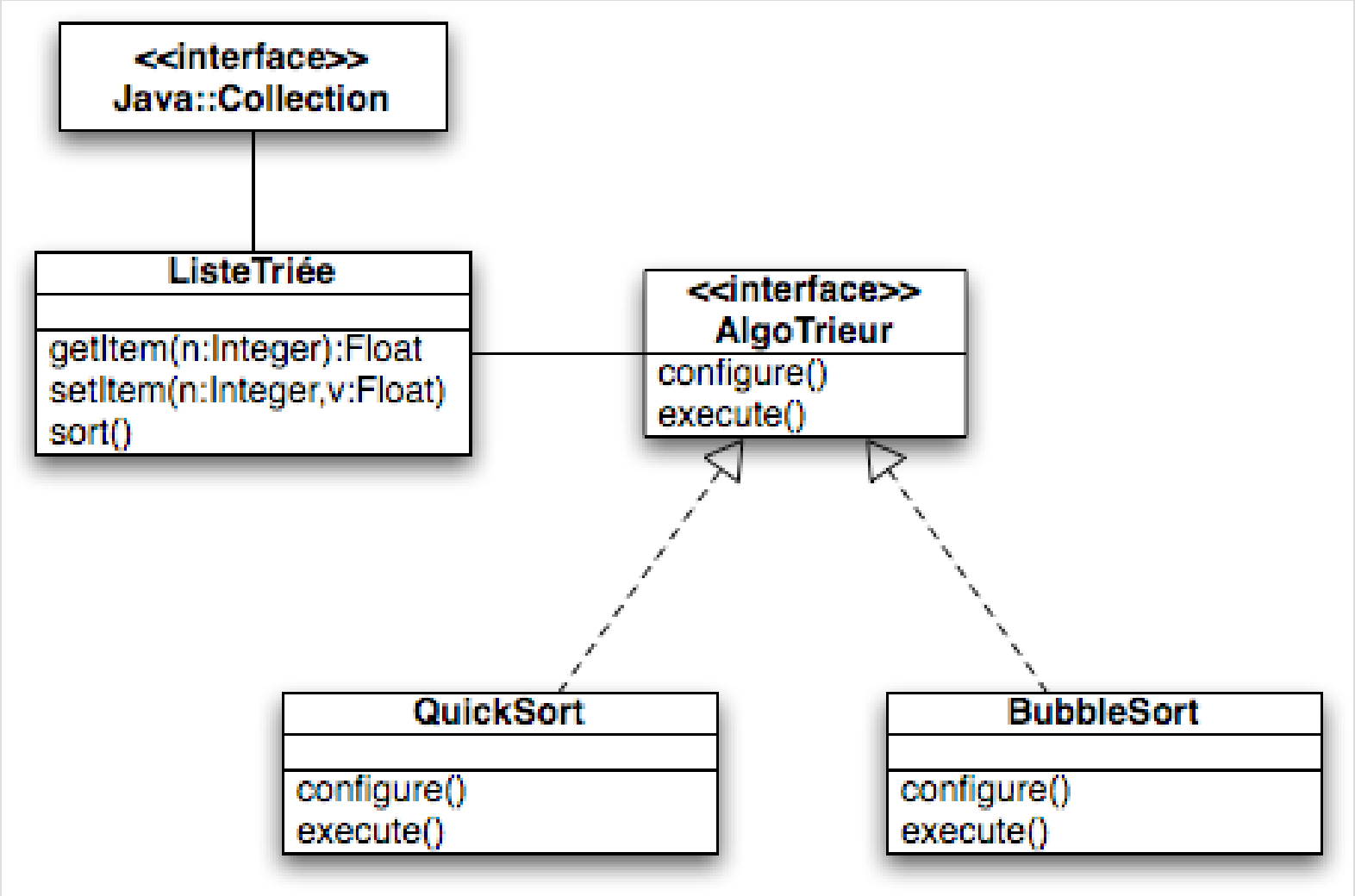


Source : http://download.oracle.com/javase/tutorial/uiswing/layout/visual.html

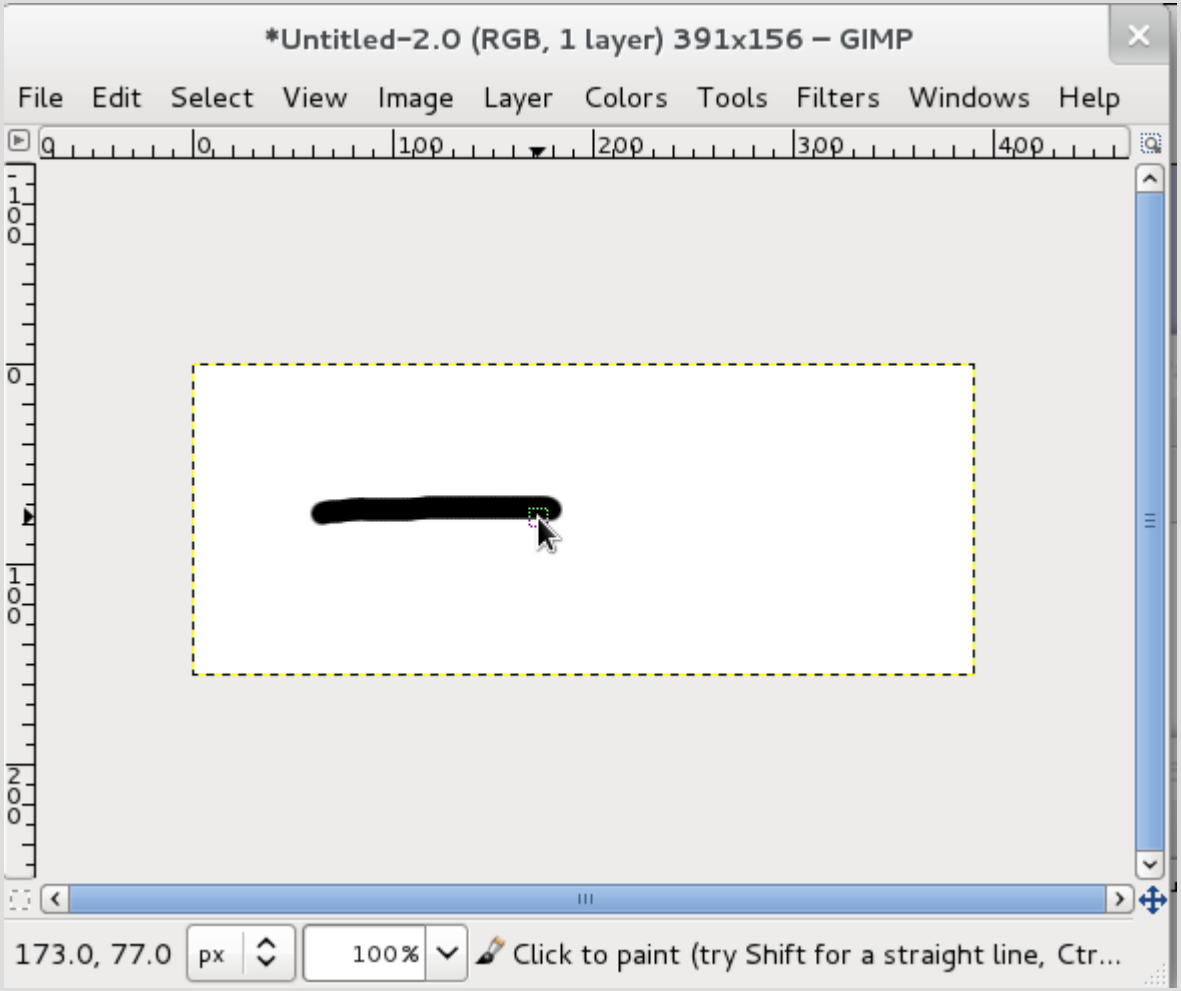
Strategy


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

Example

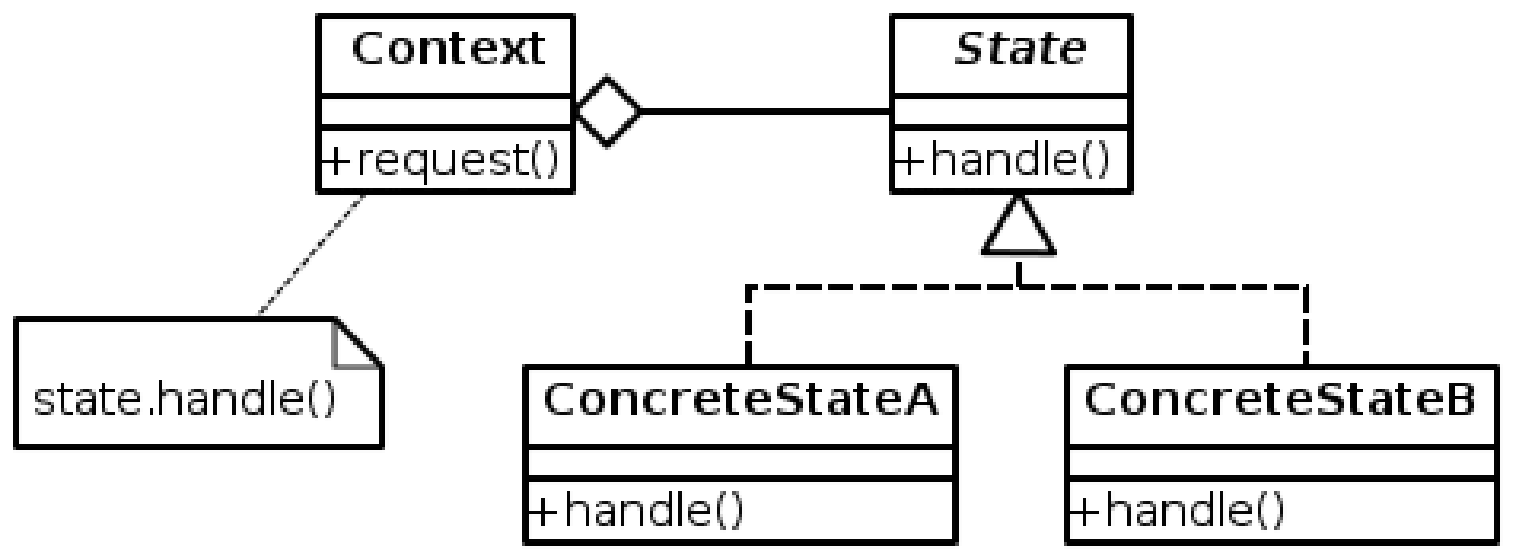

<<interface>>
Java::Collection
ListeTriée
getItem(n:Integer):Float
setItem(n:Integer,v:Float)
sort()
<<interface>>
AlgoTrieur
configure()
execute()
QuickSort
configure()
execute()
BubbleSort
configure()
execute()

Patron de conception « état »



Need: my software has many mode (edition, selection, preview...)

“State” pattern



“State” pattern

```
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());
        }
}
```

Summary

We proposed concrete and local solutions to

- One class = one responsibility
- Better extend than modify
- Liskov substitution principles
- One interface per client
- Dependency inversion principle

Remark

- Refactoring
- Finding a pattern in a graph