

# Design patterns

François Schwarzentruher  
ENS Cachan – Antenne de Bretagne

## Example

### **Problem**

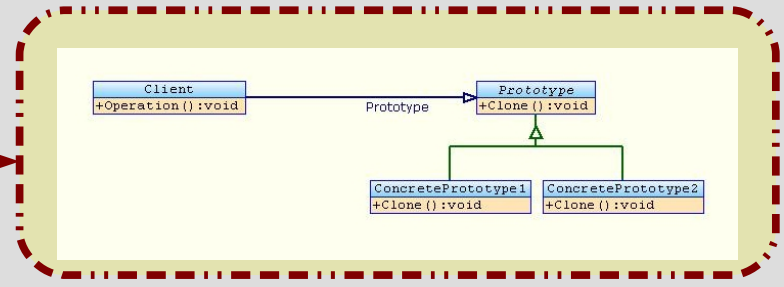
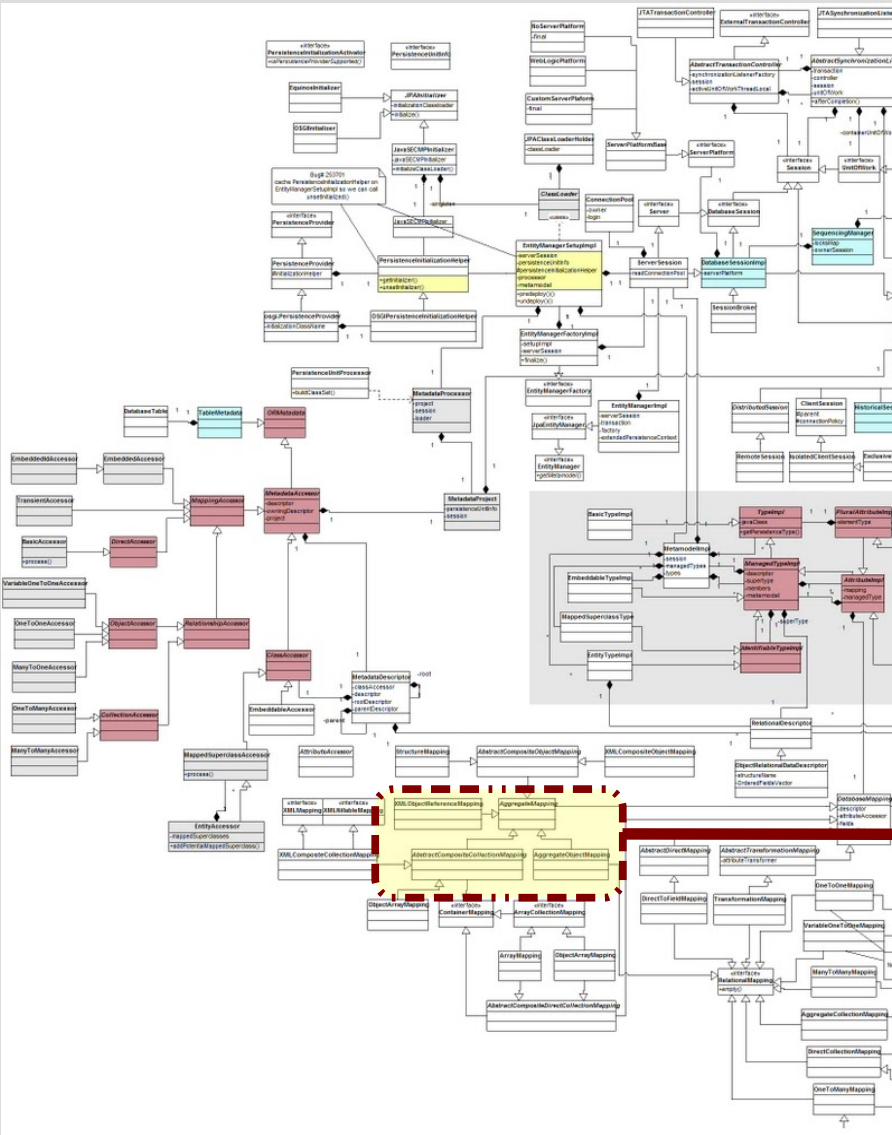
We need those actions to be extended and undone.



### **Solution**

We apply locally a suitable micro-architecture.

# Solution: Design pattern



+ sequence diagram

## Design pattern is about...



### ~~Functional properties:~~

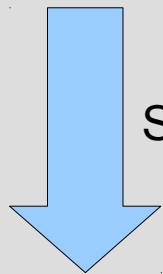
- ~~Correction of an algorithm~~
- ~~Complexity in time/space~~
- ~~...~~

### Non functional property

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

## Wisdom: do not reinvent the wheel

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



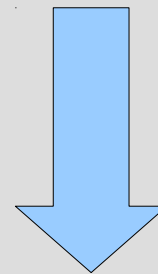
Same concepts

### **Group theory**

(Évariste Galois)

- Subgroup
- Order of an element

- UML Editor
- Pong algorithm suite
- Drawing software

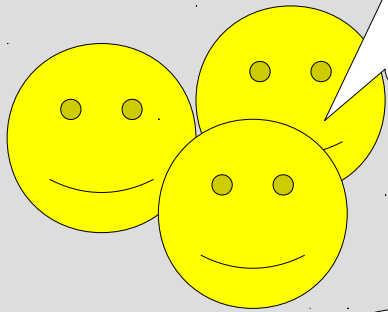


Same solutions

### **Design patterns**

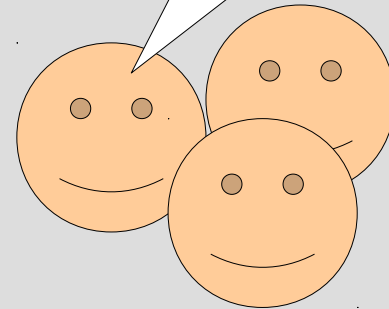
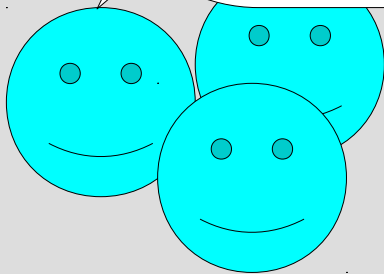
- « Façade »
- « Visitor »

# Wisdom: some vocabulary!



As the extension is Galois....

The crème anglaise is perfect with with this cake.



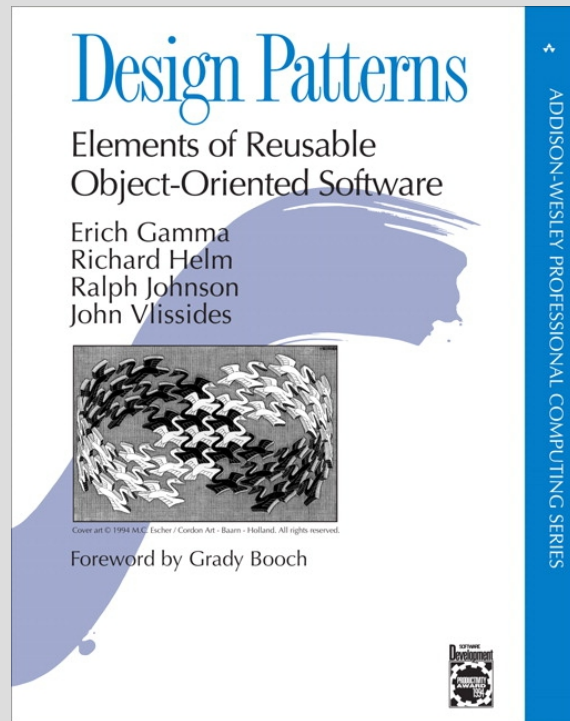
Let us apply the Visitor pattern.

## 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

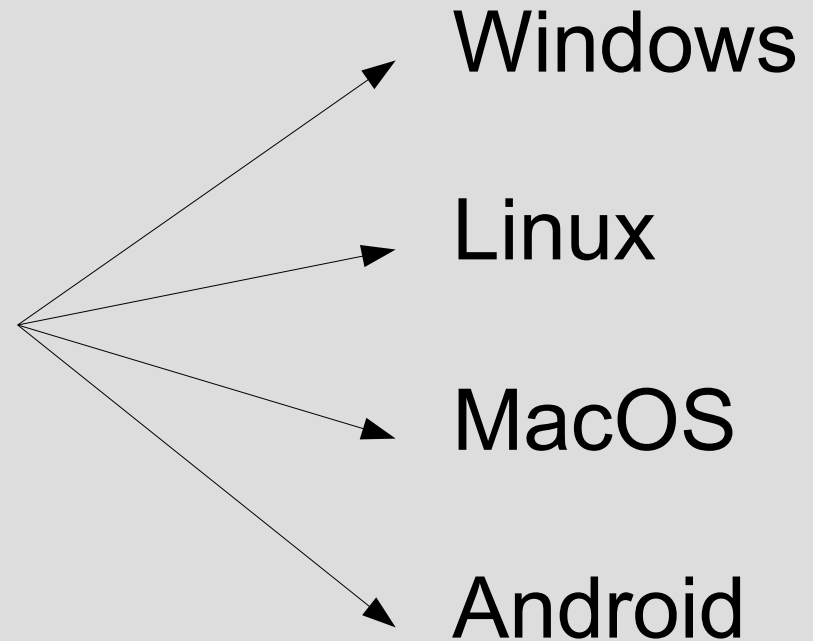
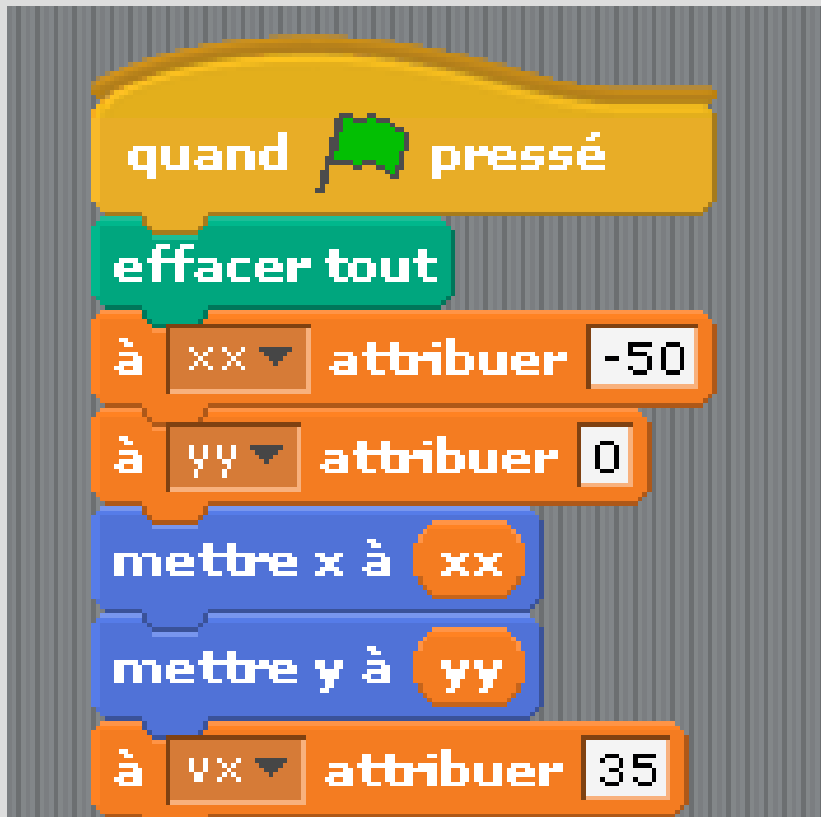
## Creational patterns

- Abstract factory
- Prototype

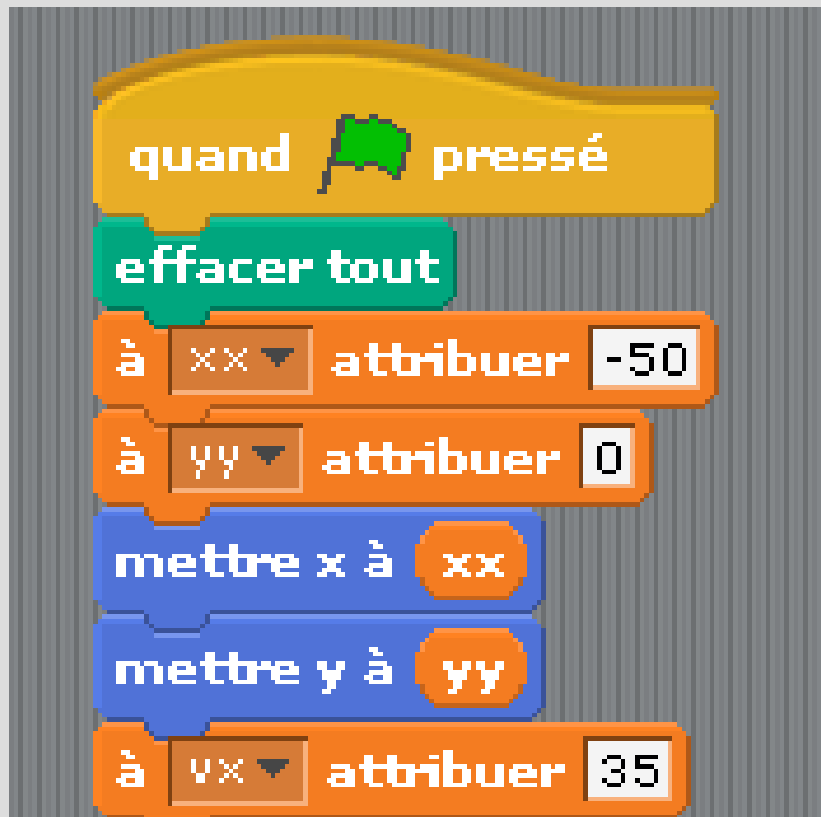
YOU want to design a software for learning algorithmic



# Need: to adapt the software to different environments



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!

```
import com.lauchenauer.istockhelper.  
import com.lauchenauer.lib.ui.Vertic  
public class AboutDialog extends JDia  
protected CardLayout mLayout;  
protected JButton mCredits;  
protected  
public AboutDialog(JDialog owner) {  
    super(owner);  
    setModal(true);  
    setUndecorated(true);  
    initUI();  
}  
protected void initUI() {  
    setSize(440, 600);  
    Container cont = getContentPane();  
    JPanel p =
```

new

```
@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.getIntegerProperty("stocks");  
            com.sun.ca.mdb.OperationRequest req1 = new com.sun.ca.mdb.OperationRequest();  
            req1.setId(ide);  
            req1.setPrice(price);  
            req1.setNoOfStocks(noOfStocks);  
            com.sun.ca.mdb.OperationResponse resp = port.operationA(req1);  
            System.out.println("Result = "+resp.getReturnValue());  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

new

new

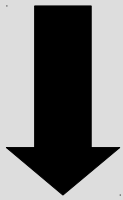
```
1 package test;  
2  
3 import java.io.IOException;  
4  
5 import jb2b.petitlien.facade.LittleLinkException;  
6 import jb2b.petitlien.facade.LittleLinkRequest;  
7  
8 import junit.framework.TestCase;  
9  
10 public class TestPetitLien extends TestCase {  
    public void testPetitLien() throws 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.println(petitLien);  
            System.out.println(petitLien2);  
        } catch (LittleLinkException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

new

new

## Non maintainable solution

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

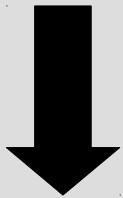


```
if(forKids)
{
    b = new brickCleanAllForKids() ;
}
else
{
    b = new brickCleanAllForHighSchoolStudents() ;
}
```

## Non maintainable solution

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

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



```
if(forKids)
```

```
{
```

```
    b = new brickCleanAllForKids() ;
```

```
}
```

```
else
```

```
{
```

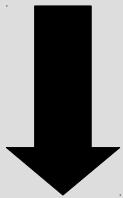
```
    b = new brickCleanAllForHighSchoolStudents() ;
```

```
}
```



## 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() ;
```

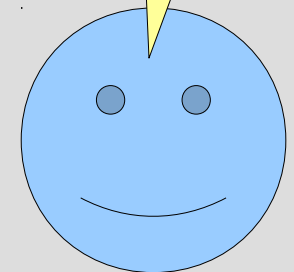
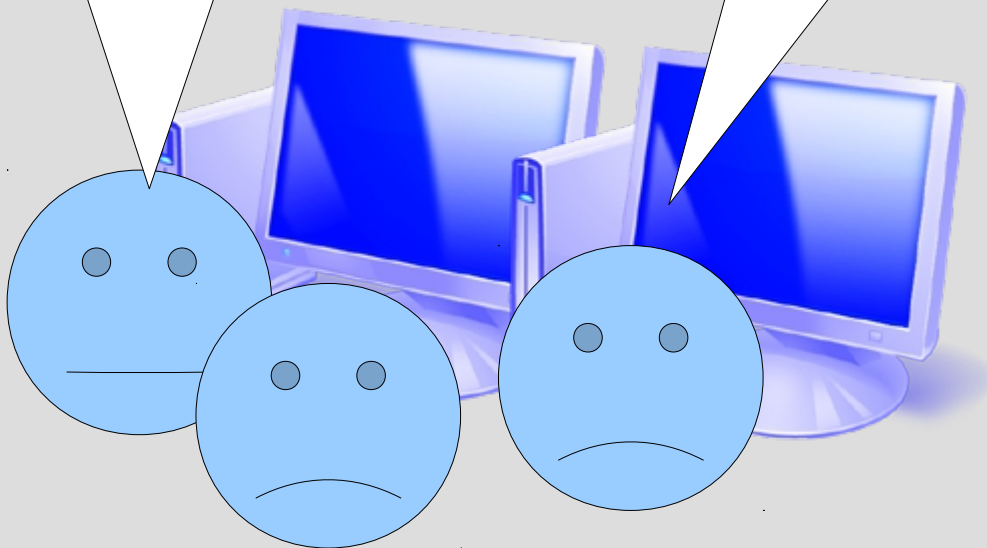
```
}
```

# Abstract factory

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

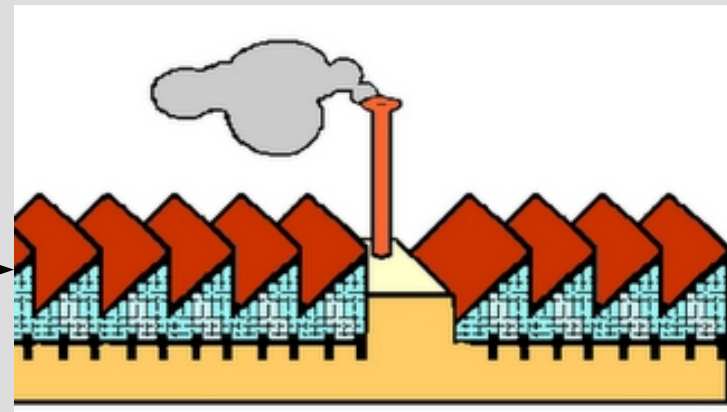
How to do this?

We apply the abstract factory pattern.



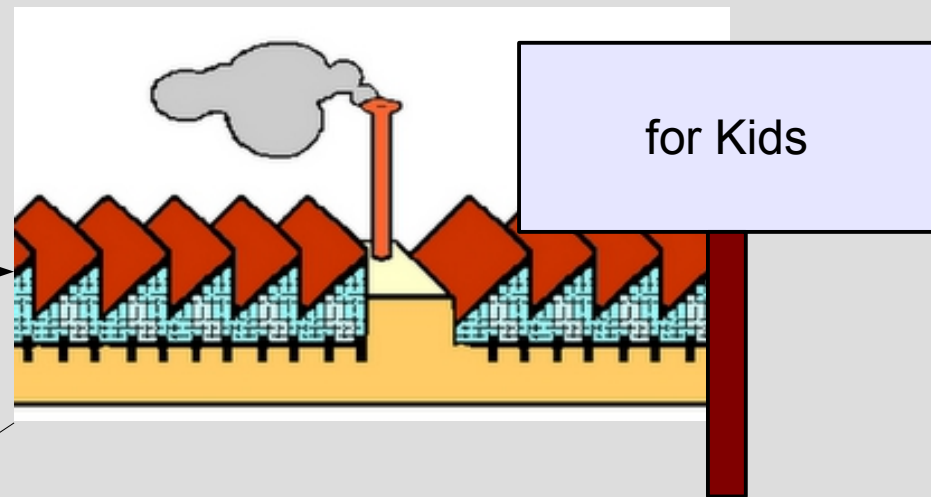
## Solution: to isolate the object creation in factories

please...  
create  
a "CleanAll" brick.



## Solution: to isolate the object creation in factories

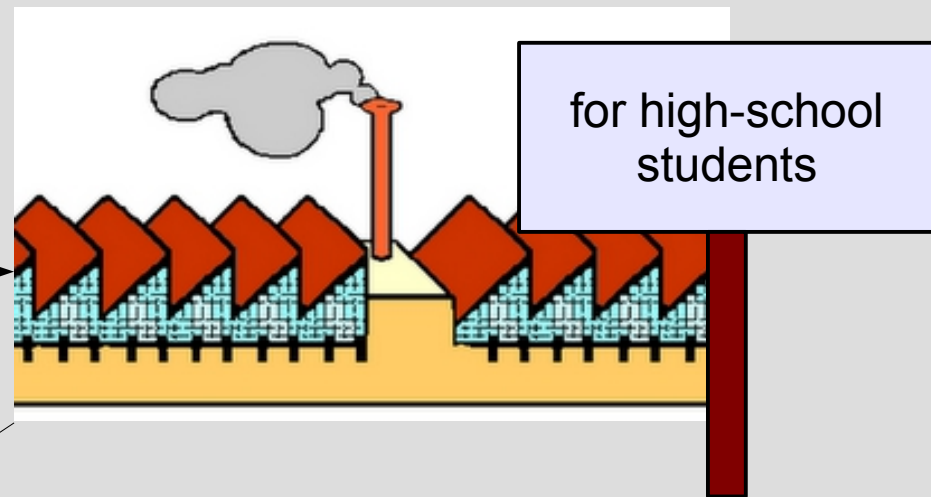
please...  
create  
a "CleanAll" brick.



clean all

## Solution: to isolate the object creation in factories

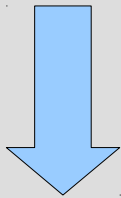
please...  
create  
a "CleanAll" brick.



clean all

# Abstract Factory pattern

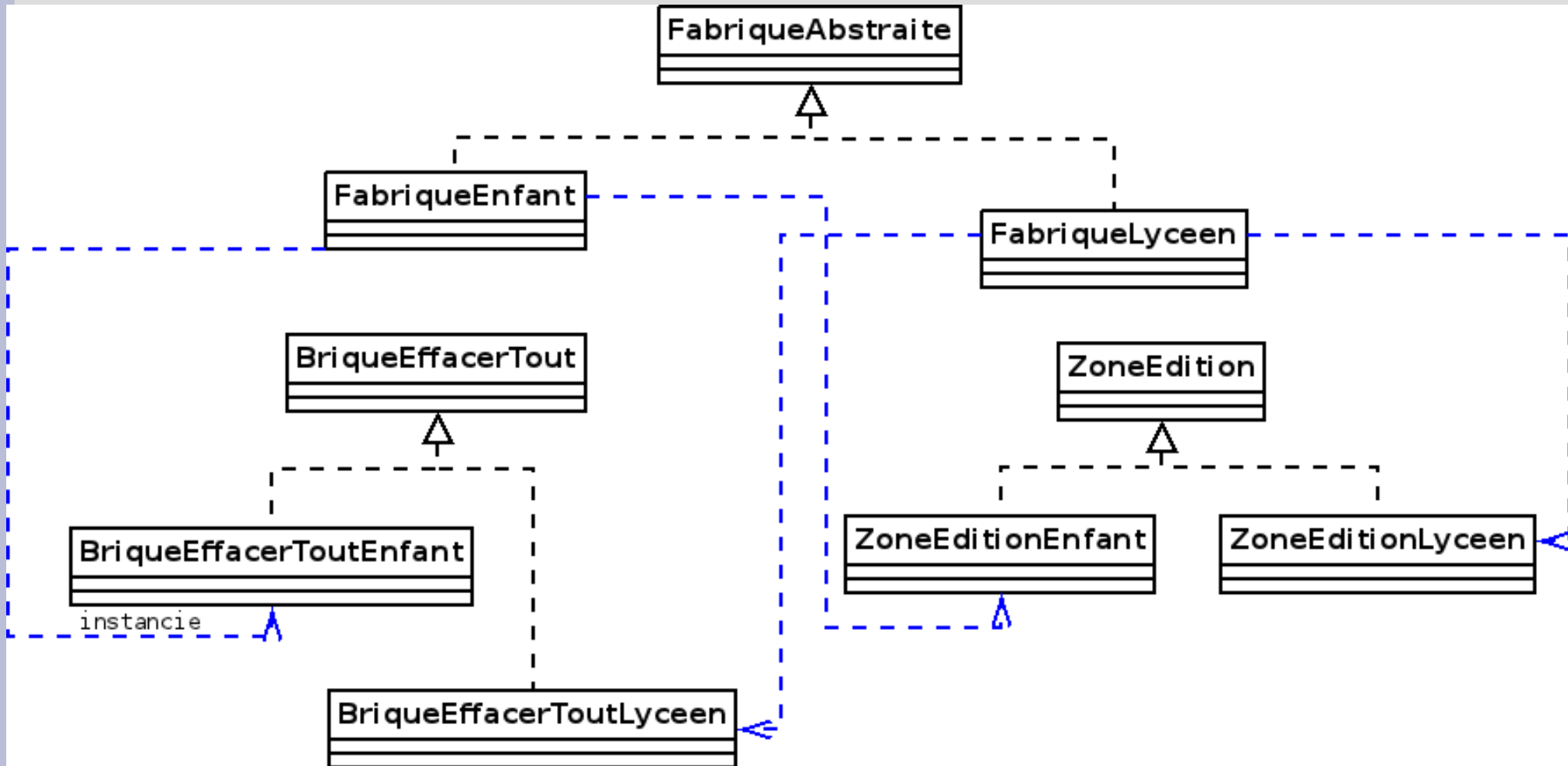
```
b = new BrickCleanAll() ;
```



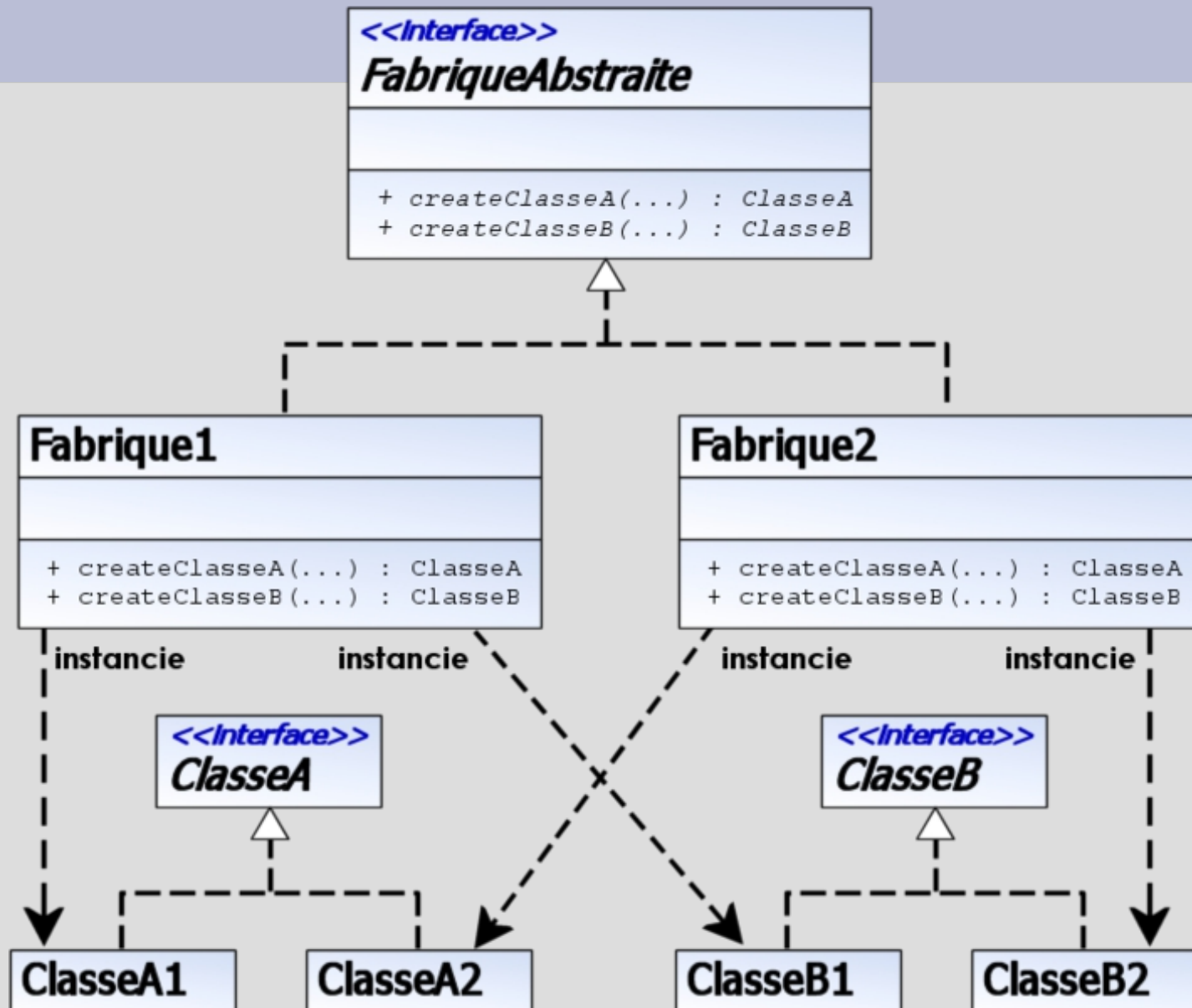
```
AbstractFactory  
factory  
= new FactoryForKids()
```

```
factory.getNewBrickCleanAll()
```

# Abstract Factory pattern

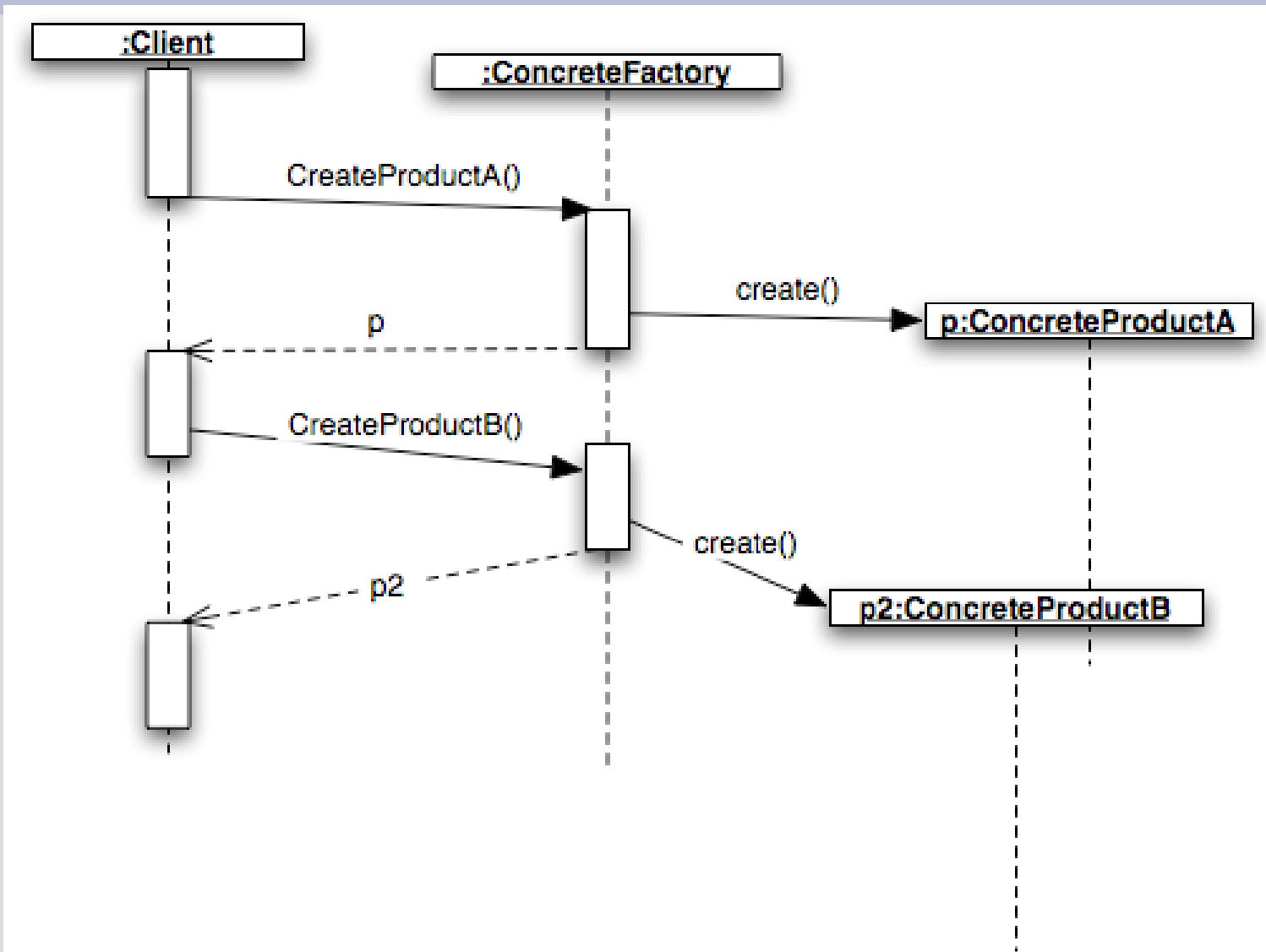


# Design pattern: abstract factory





# Sequence diagram for the Abstract Factory pattern



## Conclusion on the abstract factory

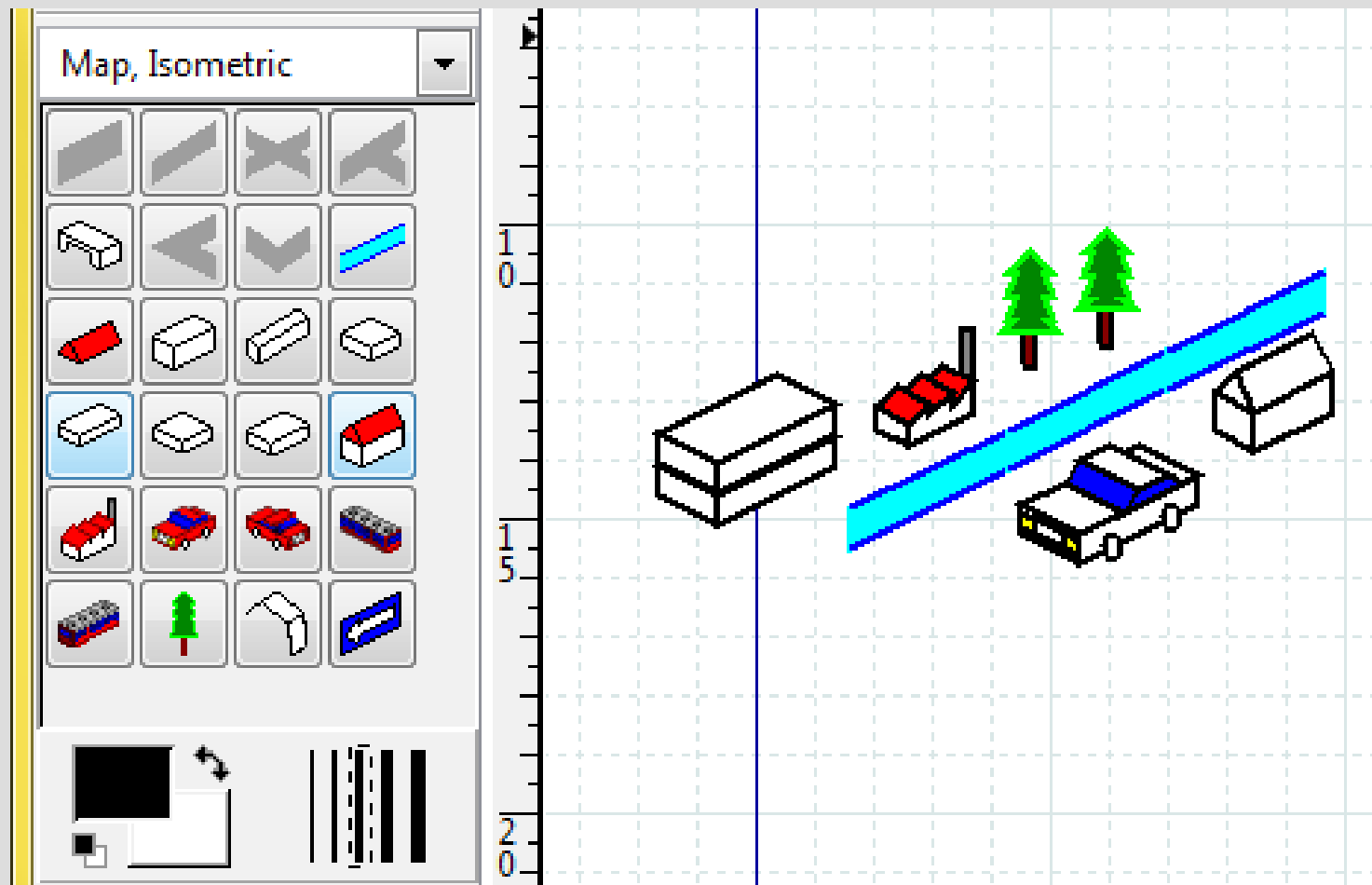
### Good points

- 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!

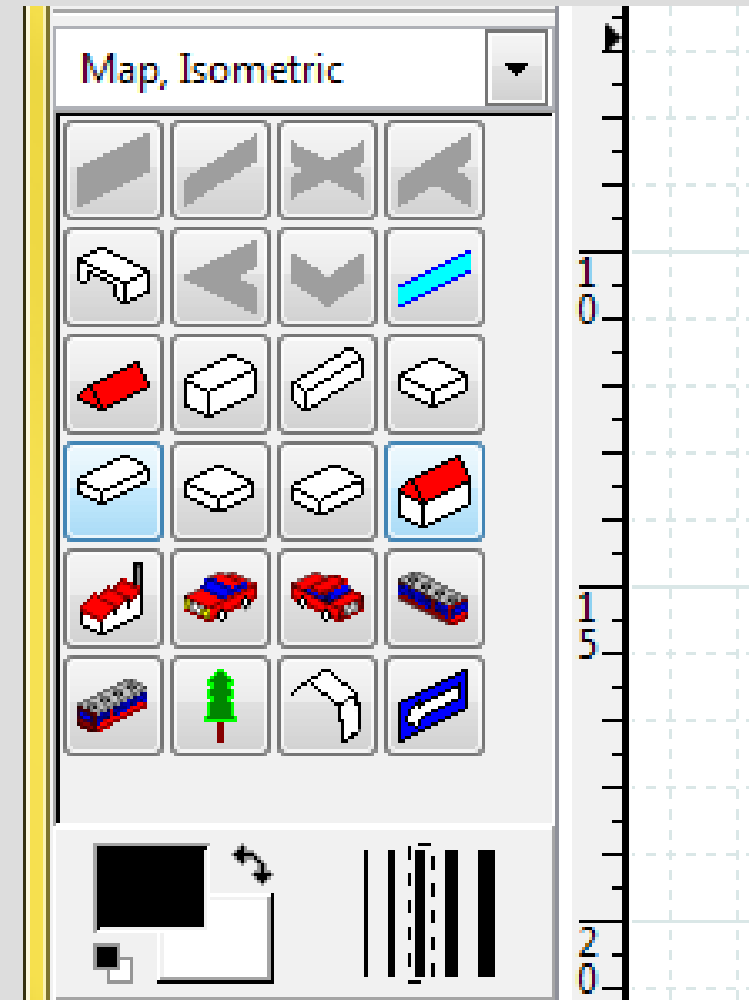
# Build objects from a model: prototype pattern



Source : logiciel Dia

# Needs

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



## DO NOT DO THAT: one creation per button!

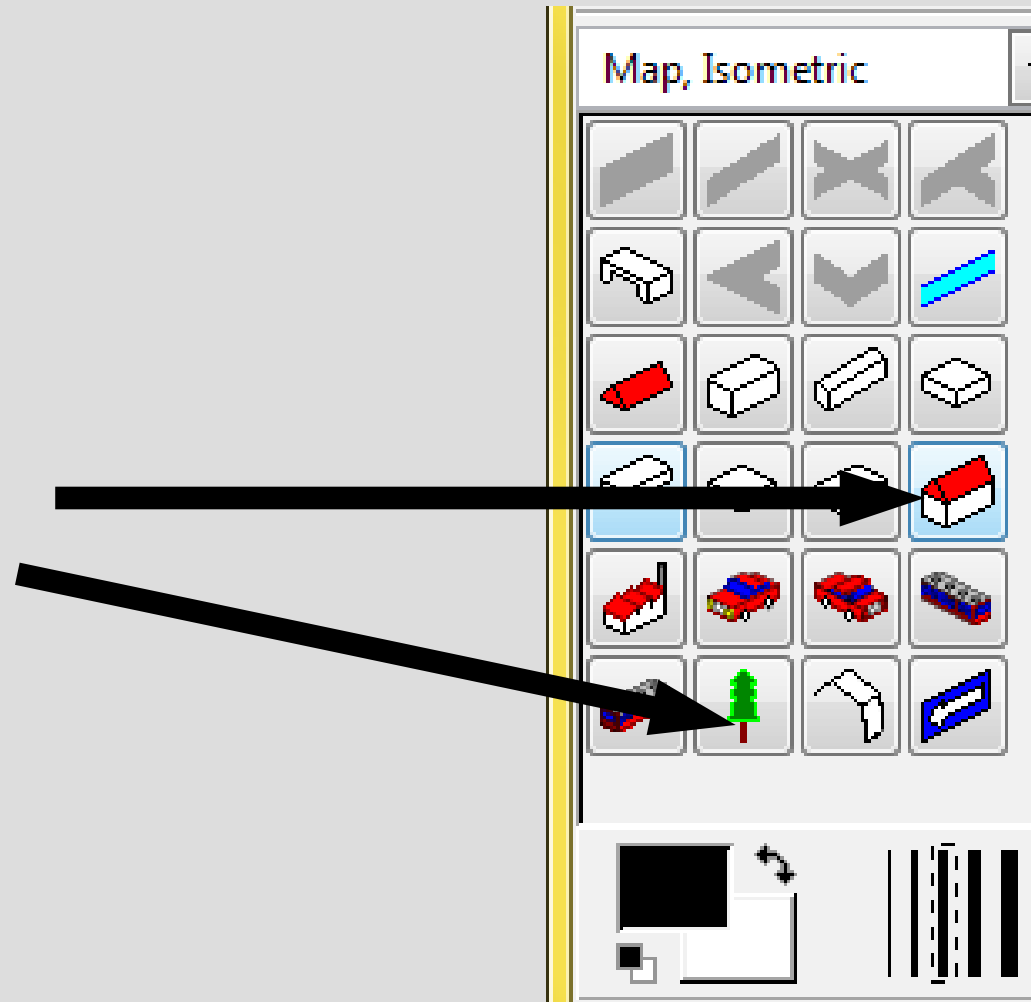
```
new House(Color.RED,  
Color.WHITE, 16, 32, 16)
```

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

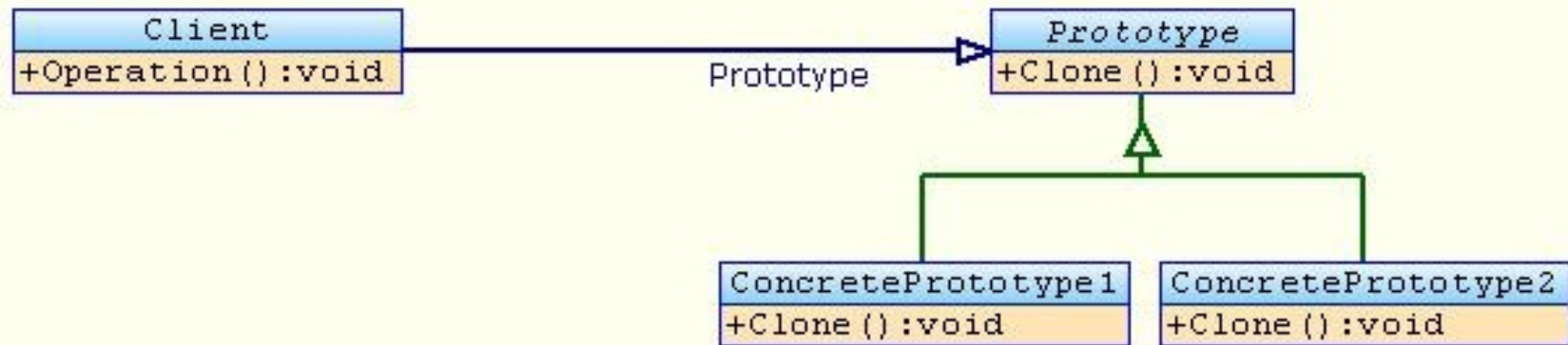


# TO DO!

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



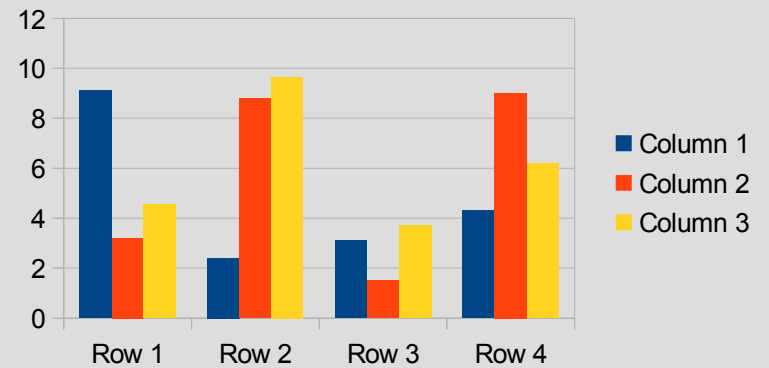
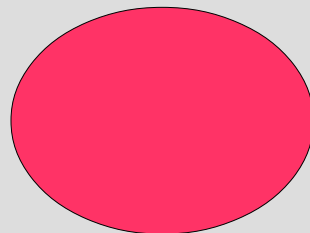
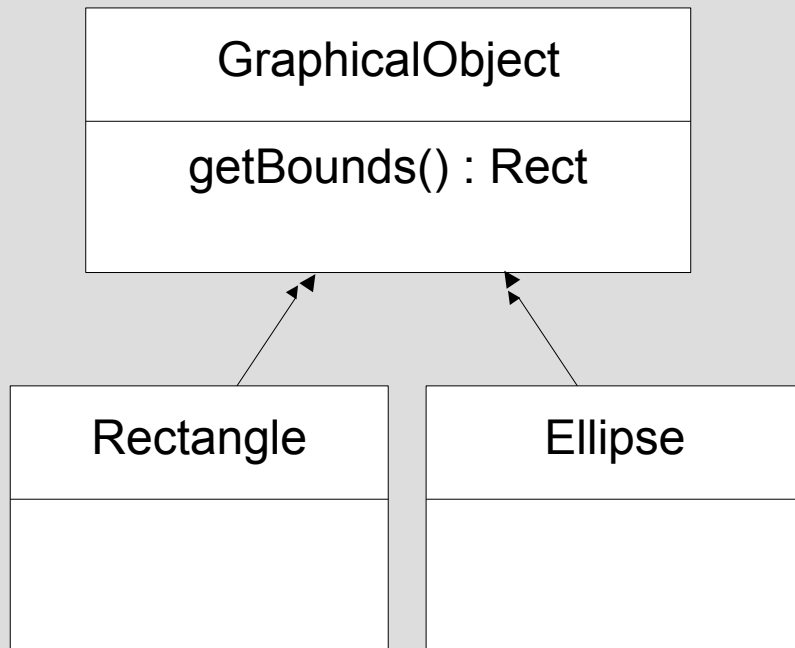
## Structural pattern

- 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...

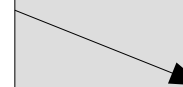
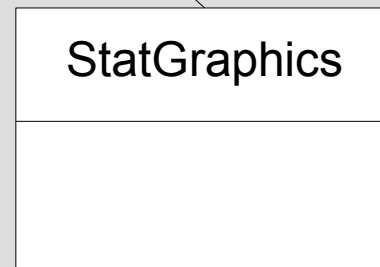
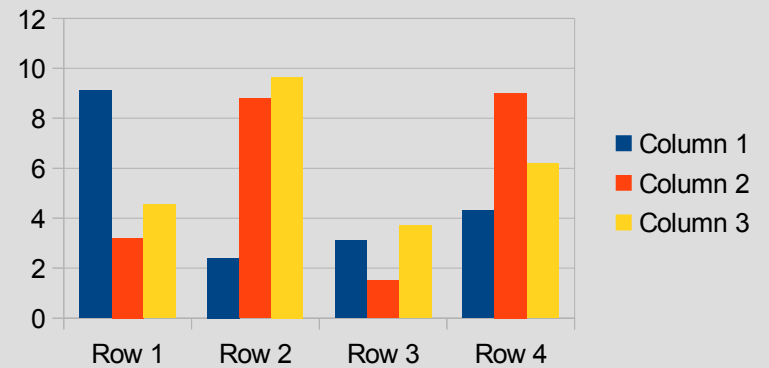
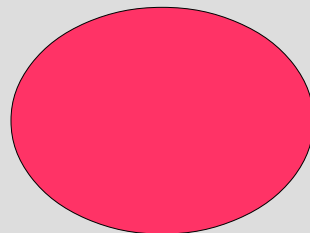
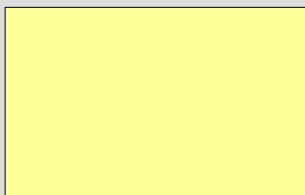
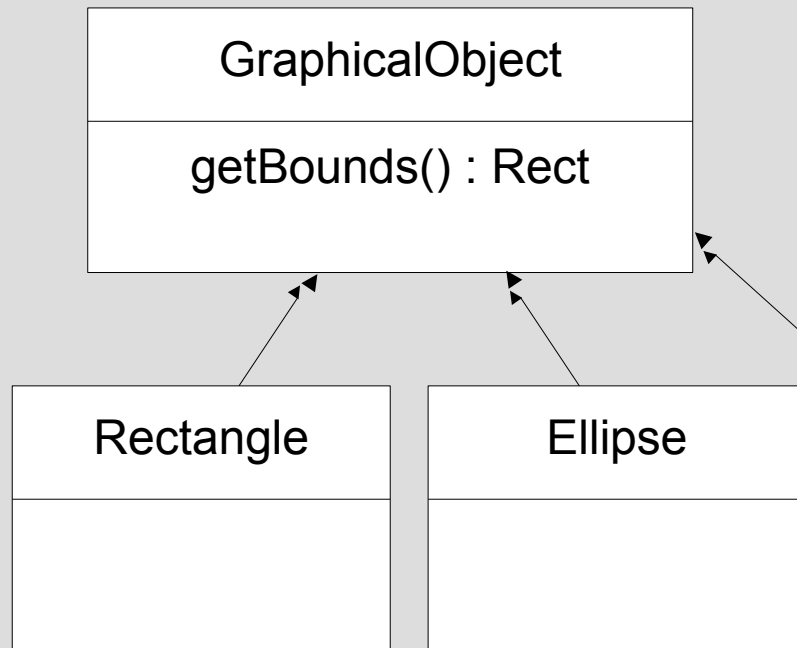


LibreOffice proposes such graphics... But the interface is different...

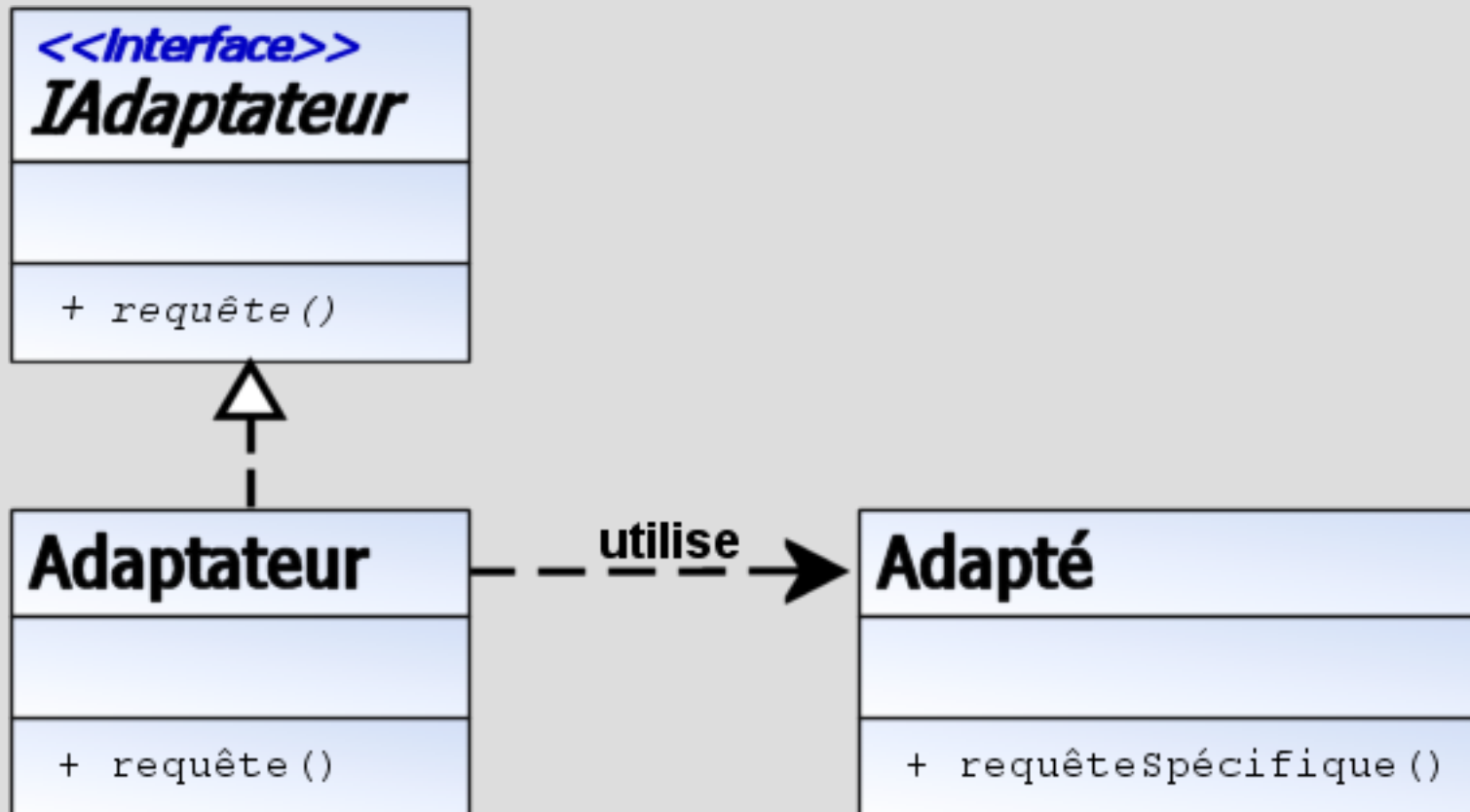
# Solution

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

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

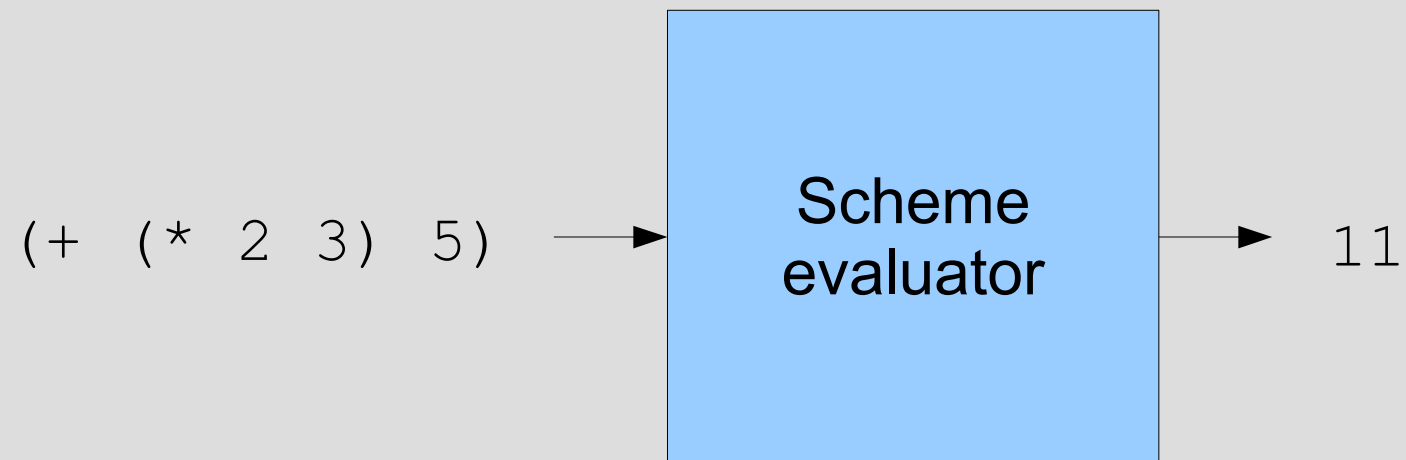


# Adaptor



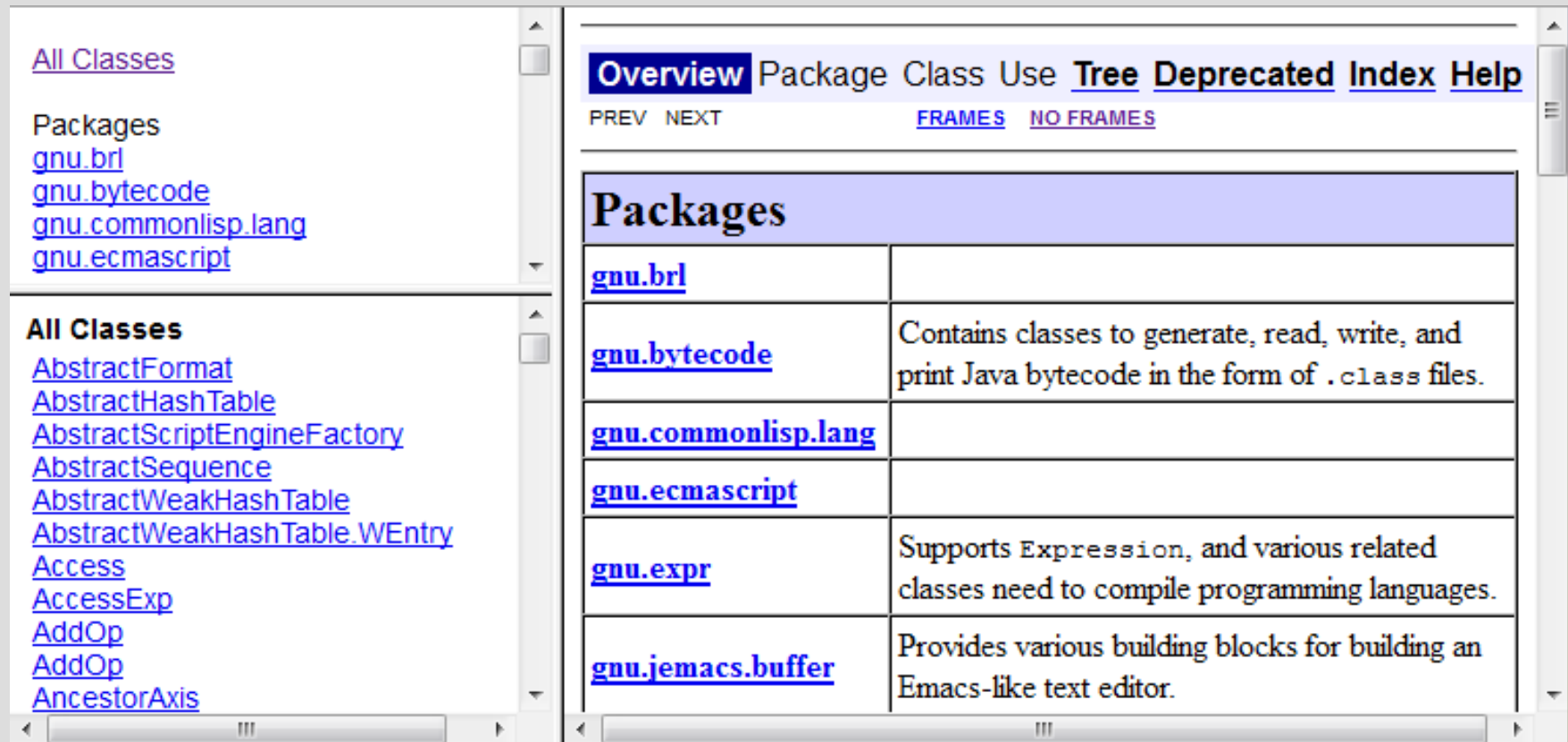
source : Wikipedia

## 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)

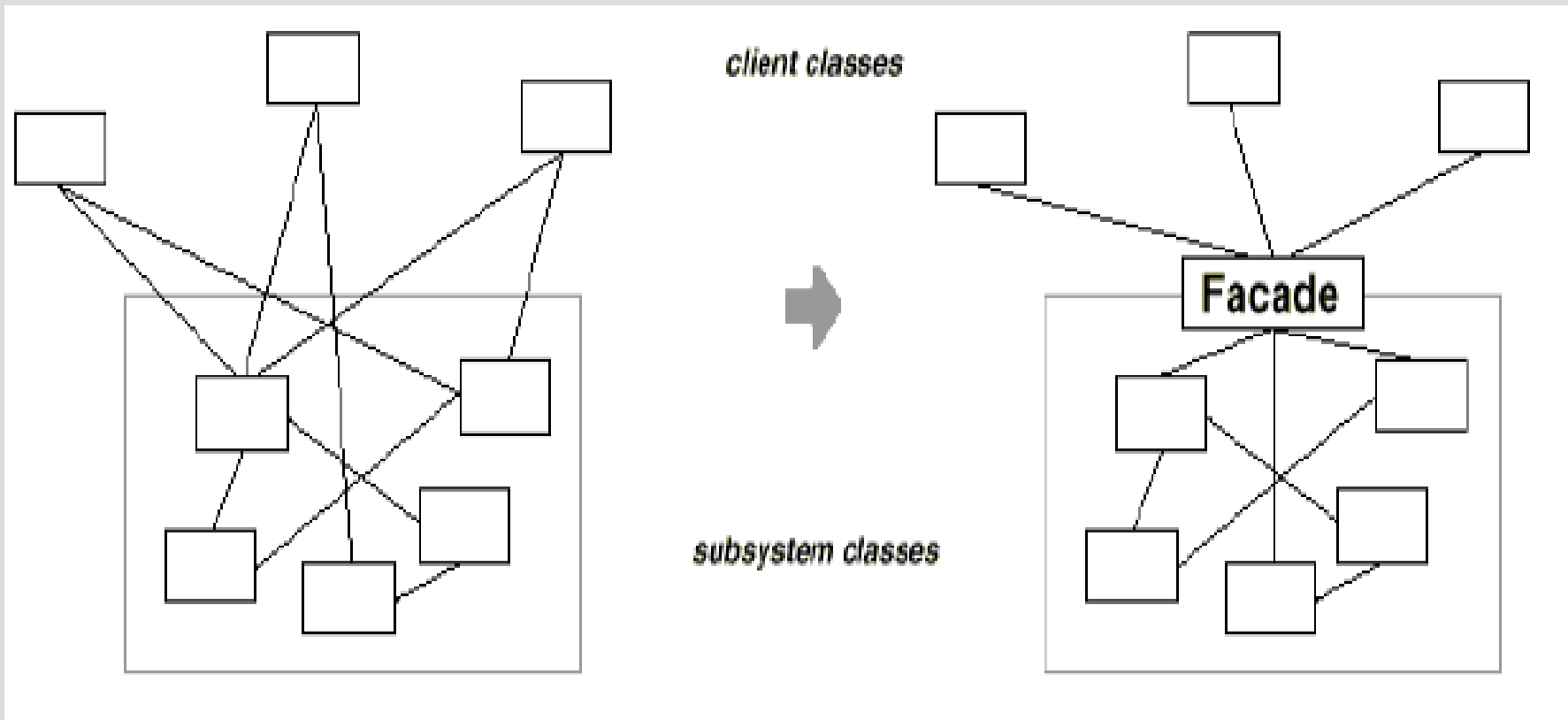


The screenshot shows a Java class browser interface. On the left, there are two panels: 'All Classes' and 'All Packages'. The 'All Packages' panel lists the following packages: `gnu.brl`, `gnu.bytecode`, `gnu.commonlisp.lang`, and `gnu.ecmascript`. The 'All Classes' panel lists various classes including `AbstractFormat`, `AbstractHashTable`, `AbstractScriptEngineFactory`, `AbstractSequence`, `AbstractWeakHashTable`, `AbstractWeakHashTable.WEntry`, `Access`, `AccessExp`, `AddOp`, `AddOp`, and `AncestorAxis`.

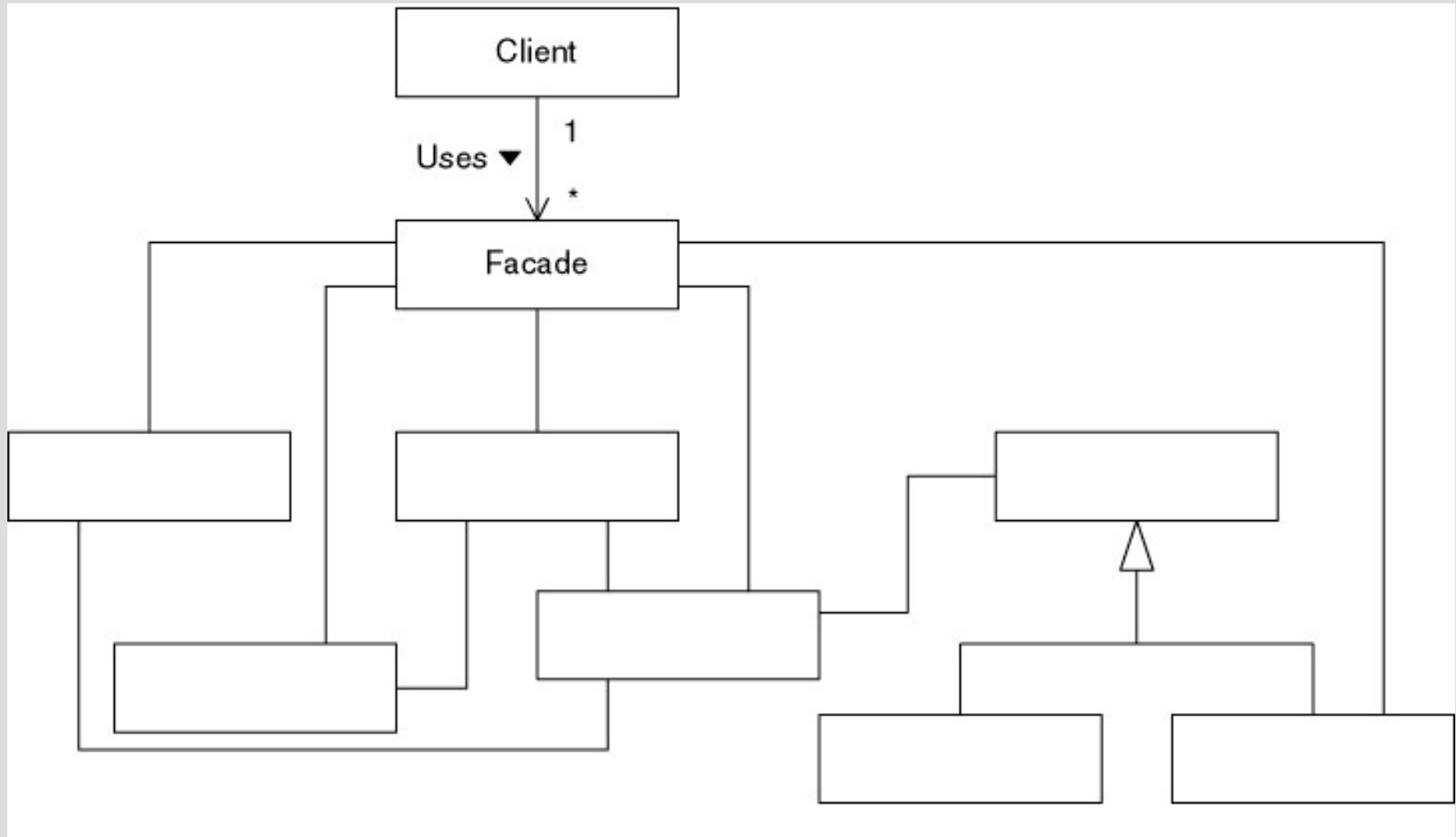
The main window displays the 'Overview' page for the `gnu` package. The navigation bar includes 'Overview', 'Package', 'Class', 'Use', 'Tree', 'Deprecated', 'Index', and 'Help'. Below the navigation bar, there are links for 'PREV', 'NEXT', 'FRAMES', and 'NO FRAMES'. The main content area is titled 'Packages' and contains a table with the following entries:

Packages	
<a href="#">gnu.brl</a>	
<a href="#">gnu.bytecode</a>	Contains classes to generate, read, write, and print Java bytecode in the form of <code>.class</code> files.
<a href="#">gnu.commonlisp.lang</a>	
<a href="#">gnu.ecmascript</a>	
<a href="#">gnu.expr</a>	Supports <code>Expression</code> , and various related classes need to compile programming languages.
<a href="#">gnu.jemacs.buffer</a>	Provides various building blocks for building an Emacs-like text editor.

## Solution: Façade design pattern



## Solution: Façade design pattern





## Difference between Façade and adaptor

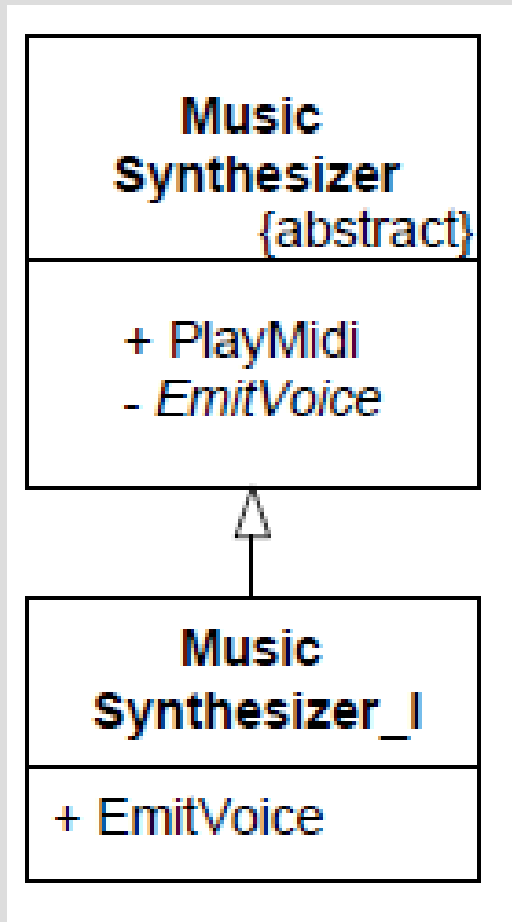
### **Façade**

- We need and create an interface

### **Adaptor**

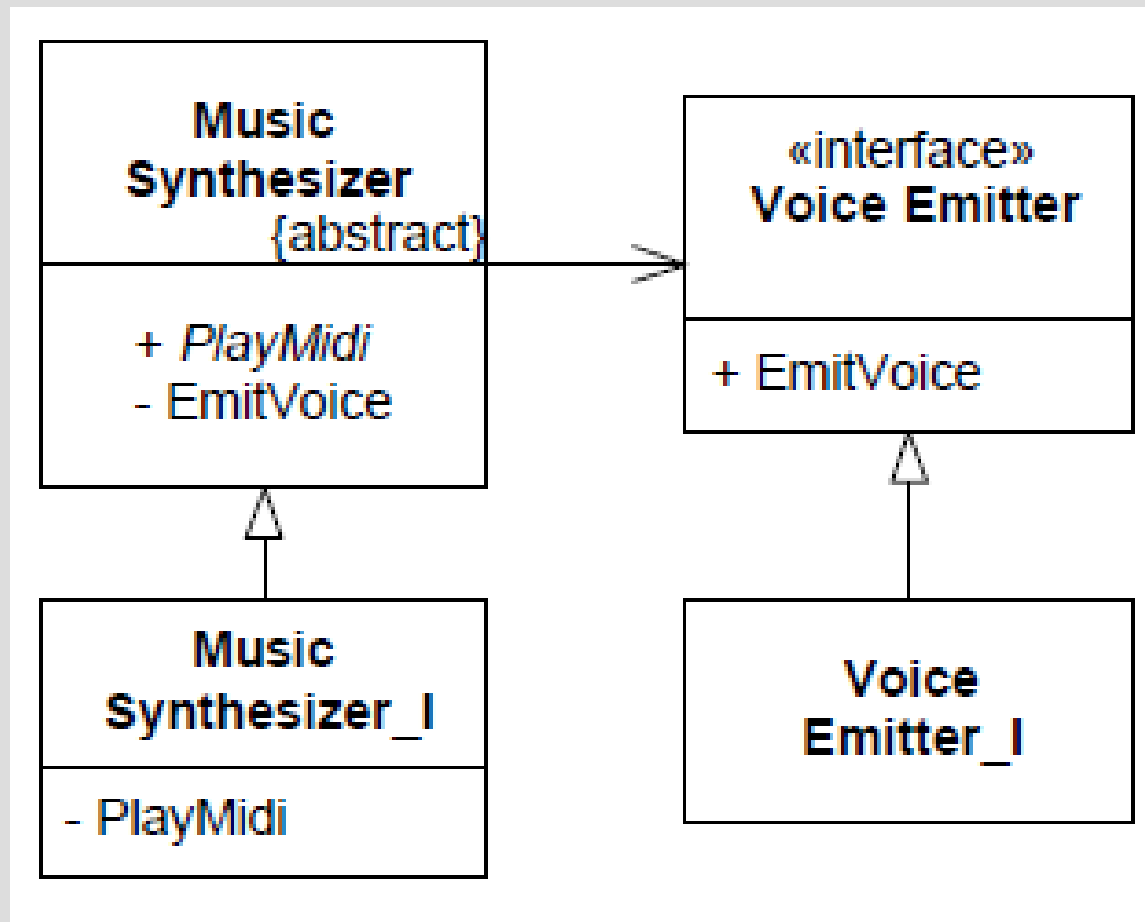
- We need to adapt an object to a given interface.

## 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

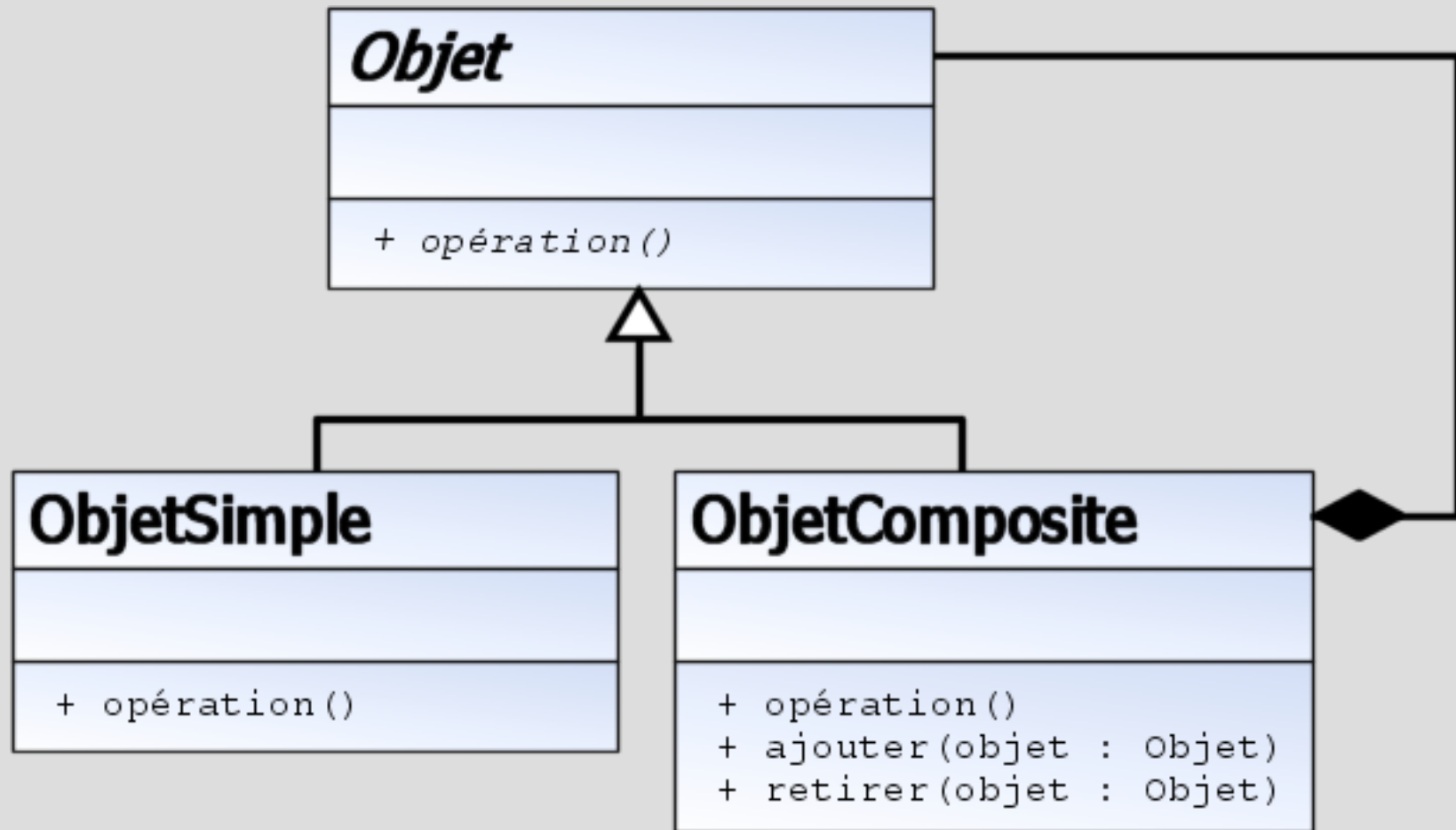


## « Composite »

Recursive objects:

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

## « Composite »



## 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:

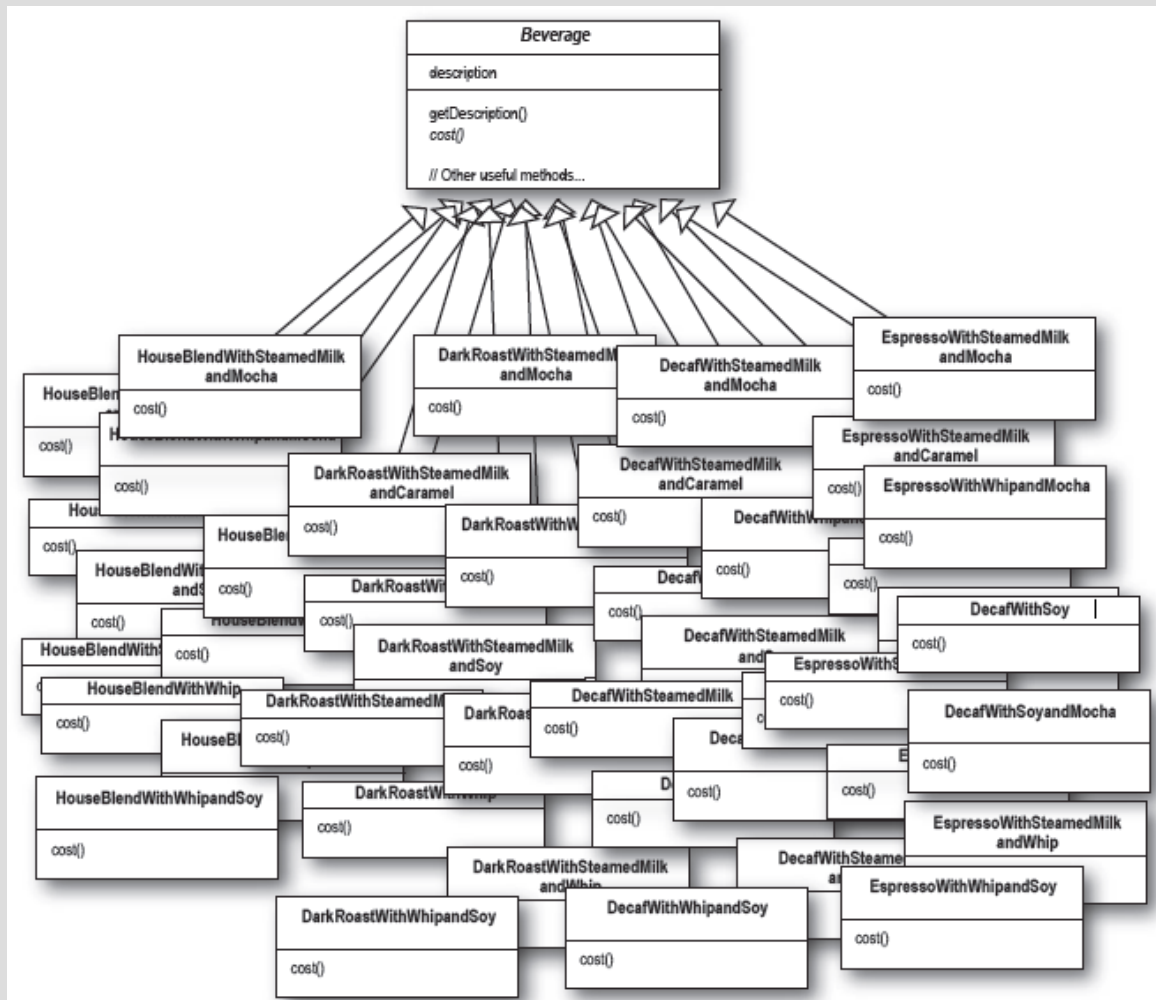
- ~ Use “if”
- ~ Use a kind of bridge?

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

- Inheritance
- The design pattern “decorator”

# With inheritance

We would have  
 $2^n$  different  
classes!

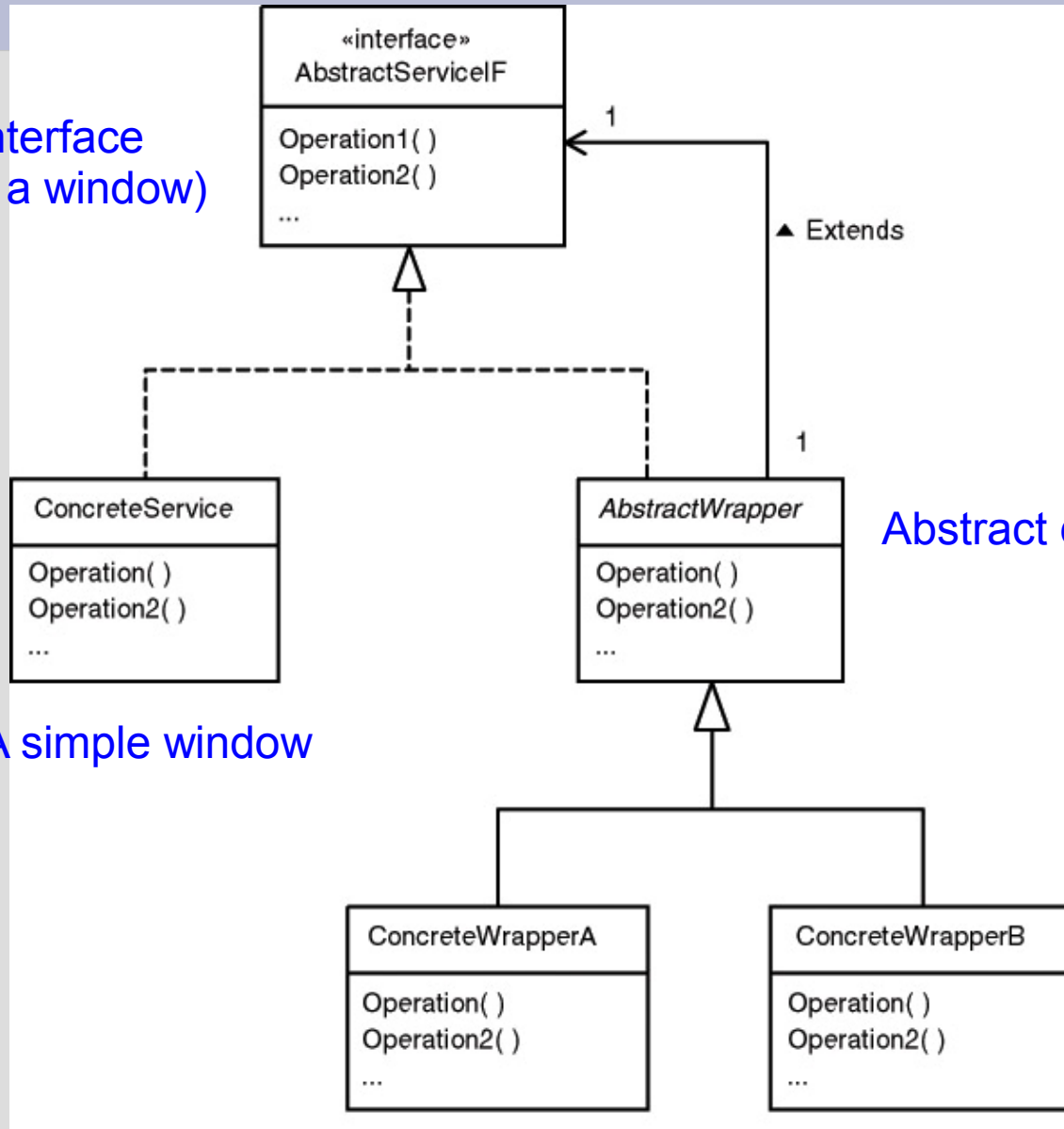




# “Decoration” design pattern

(interface  
of a window)

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



Abstract decorator

A simple window

Concrete decorator (as  
“window with special  
effects”)

## Conclusion on the pattern « Décorateur »

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

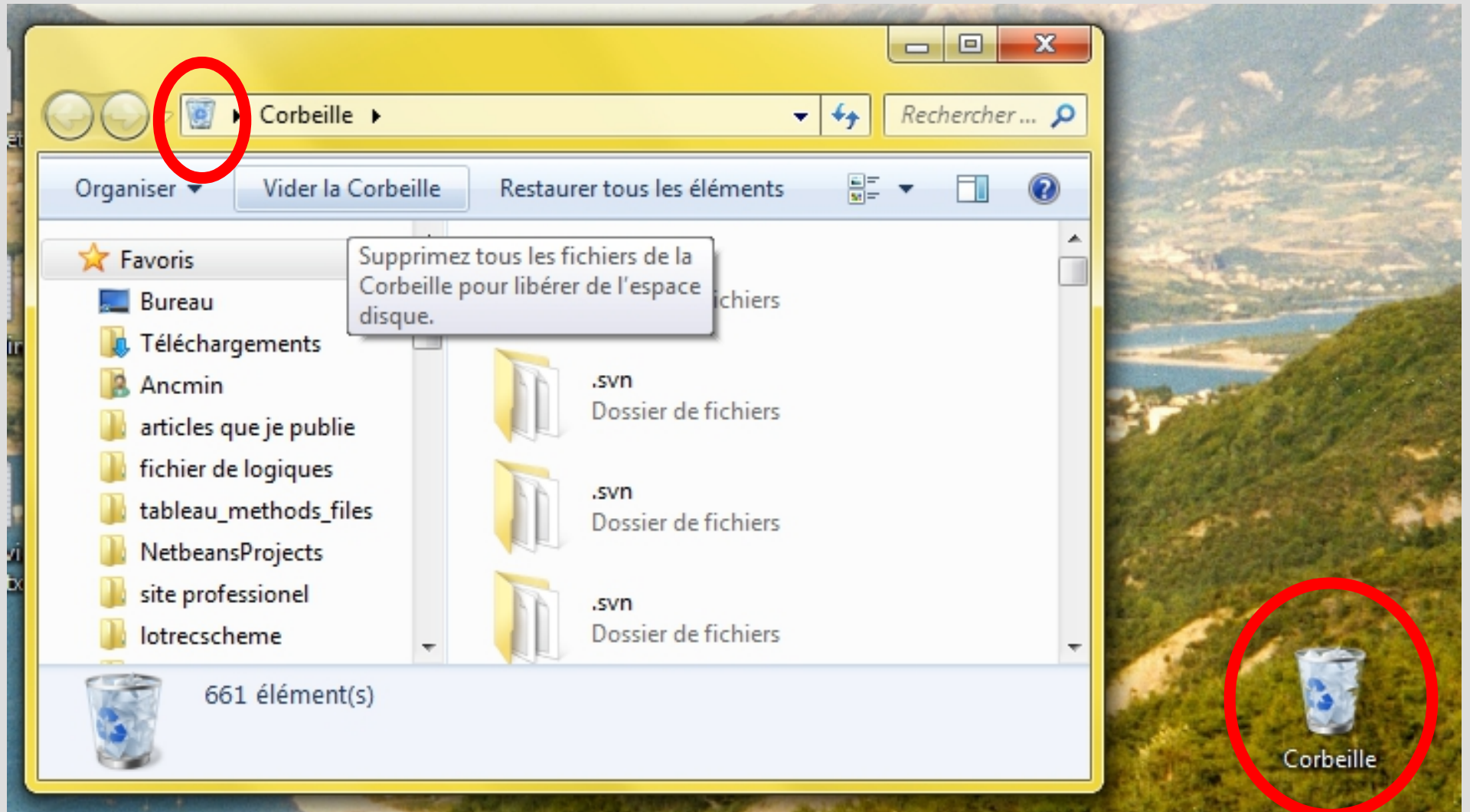
new FenetreBordure(new FenetreFond(f))

~ new FenetreFond(new FenetreBordure(f)) ?

## Behaviour design patterns

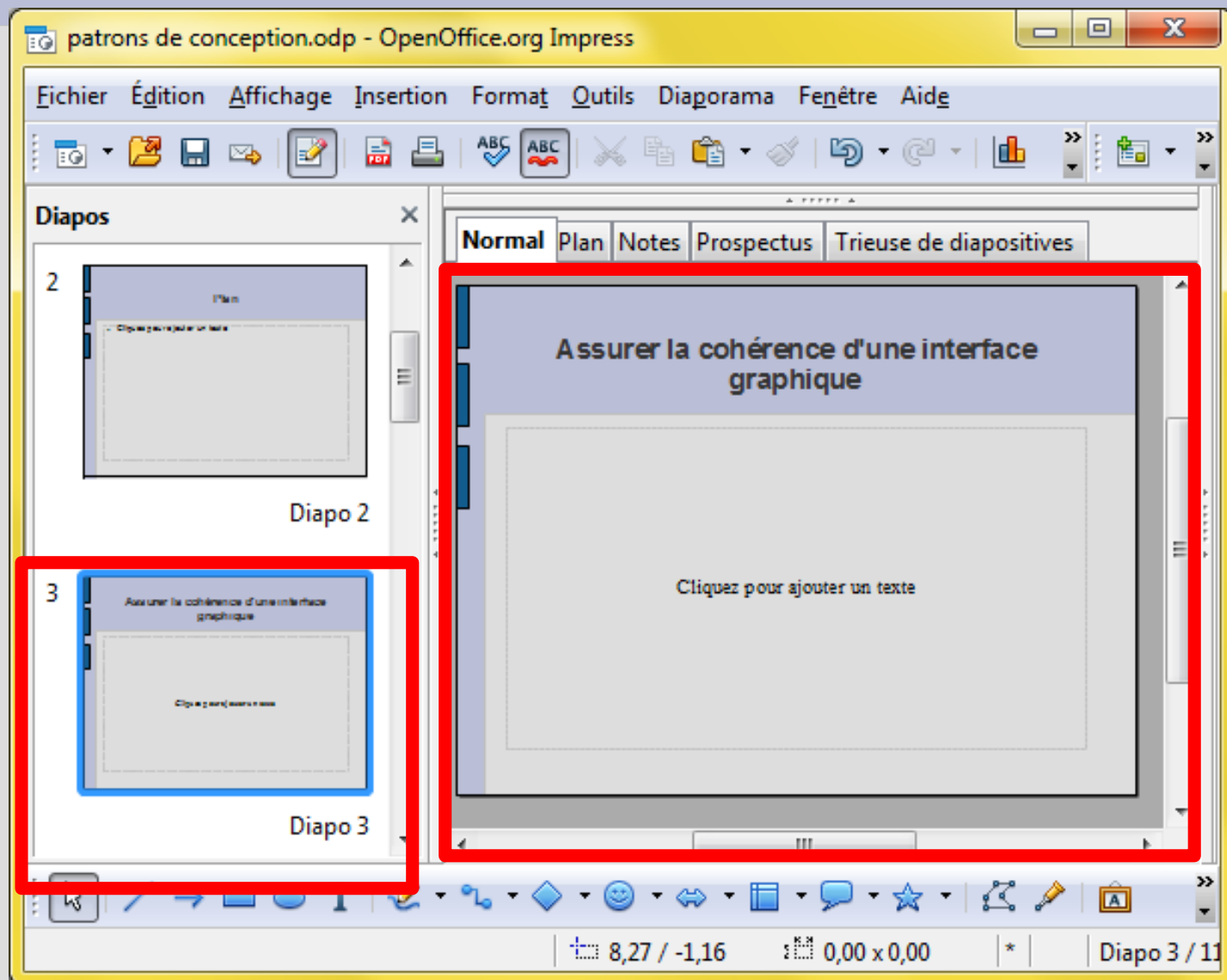
- Dependency inversion principle
- Data converter
- Cancel feature
- Change the behaviour of an object

## Need: to refresh the graphical user interface

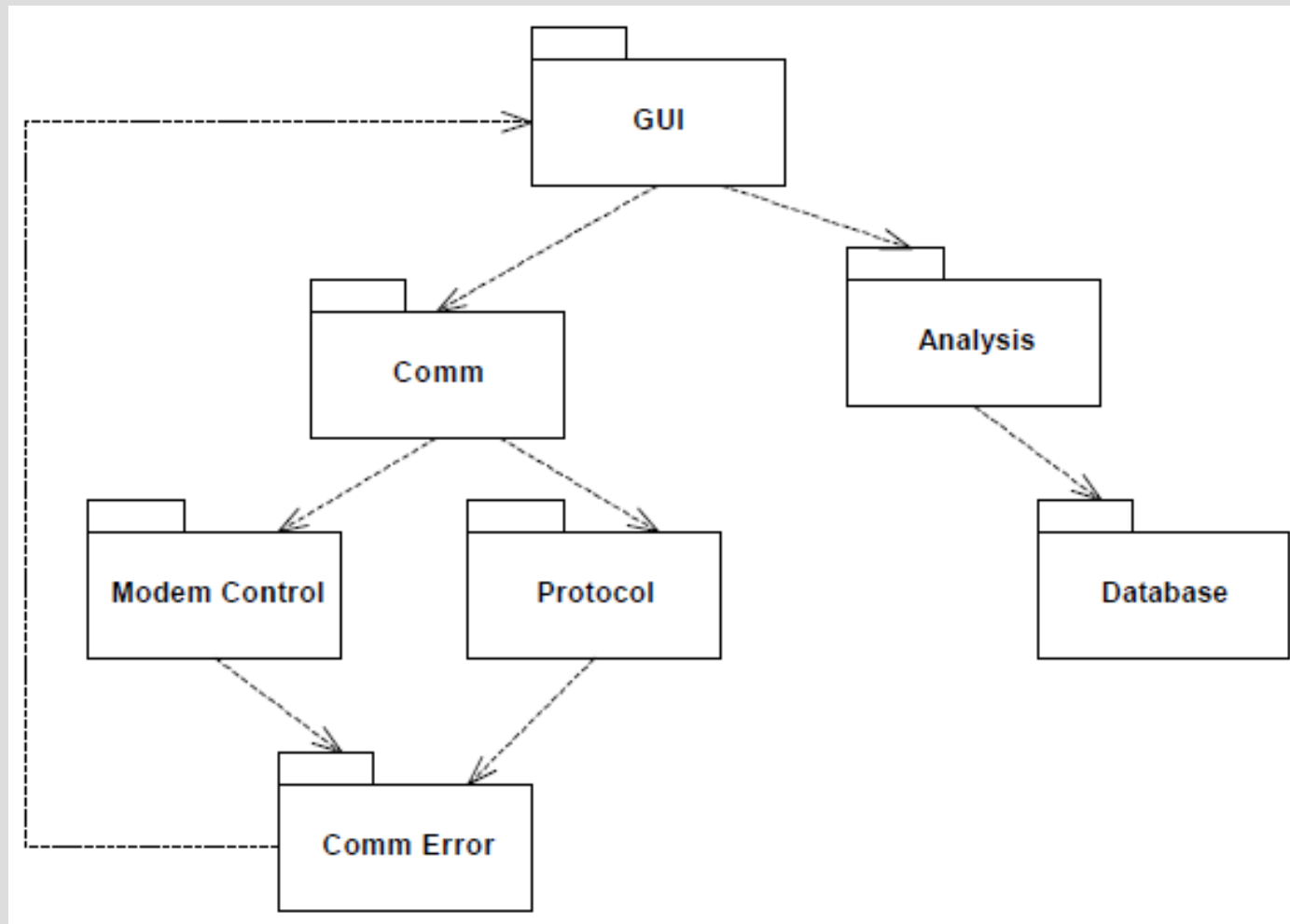


Source: Windows 7 graphical interface

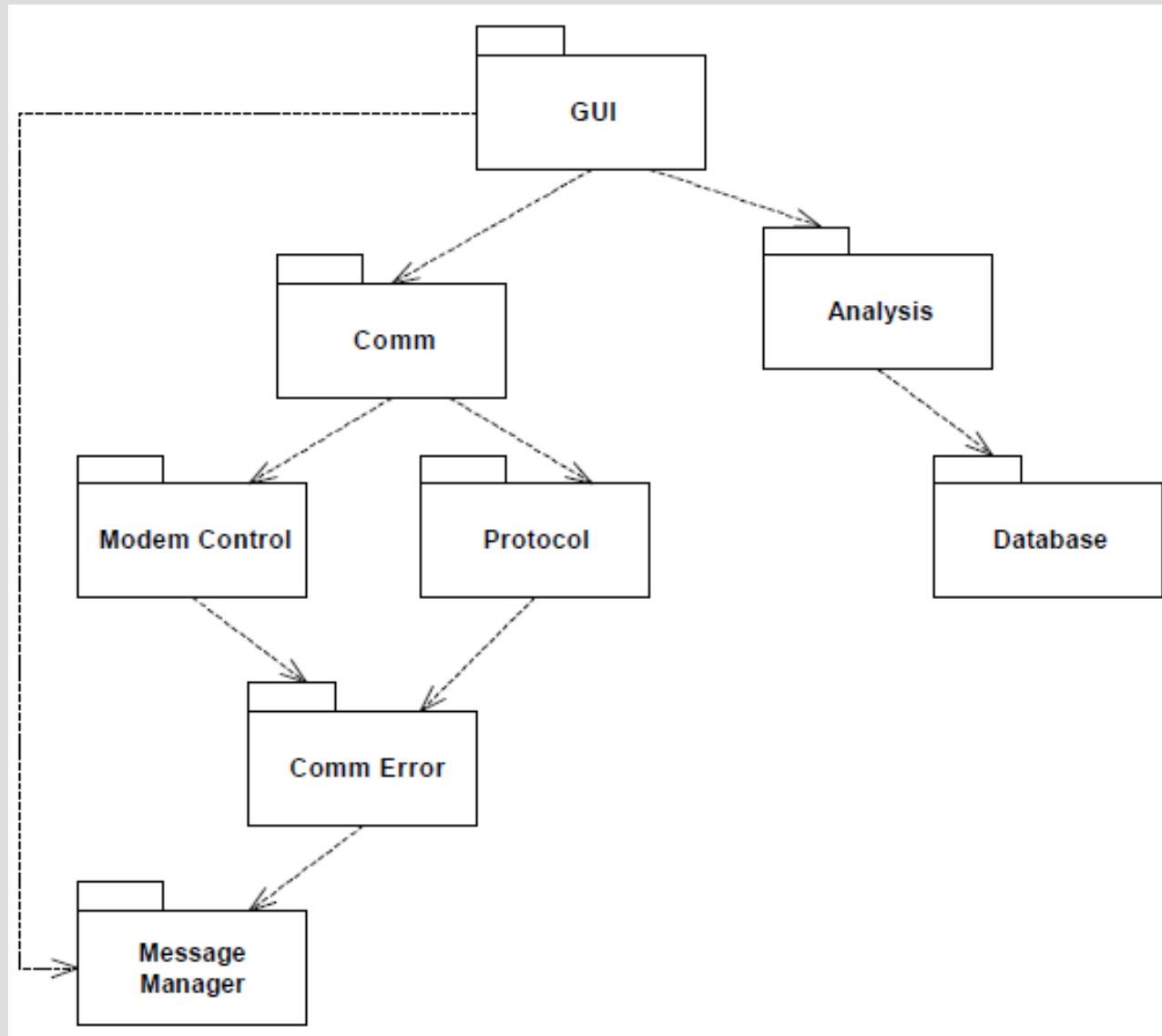
## Need: to refresh the graphical user interface



# Problem



# Solution

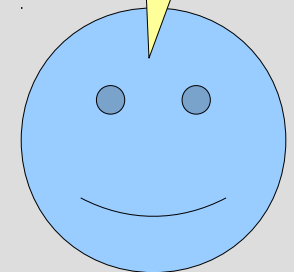
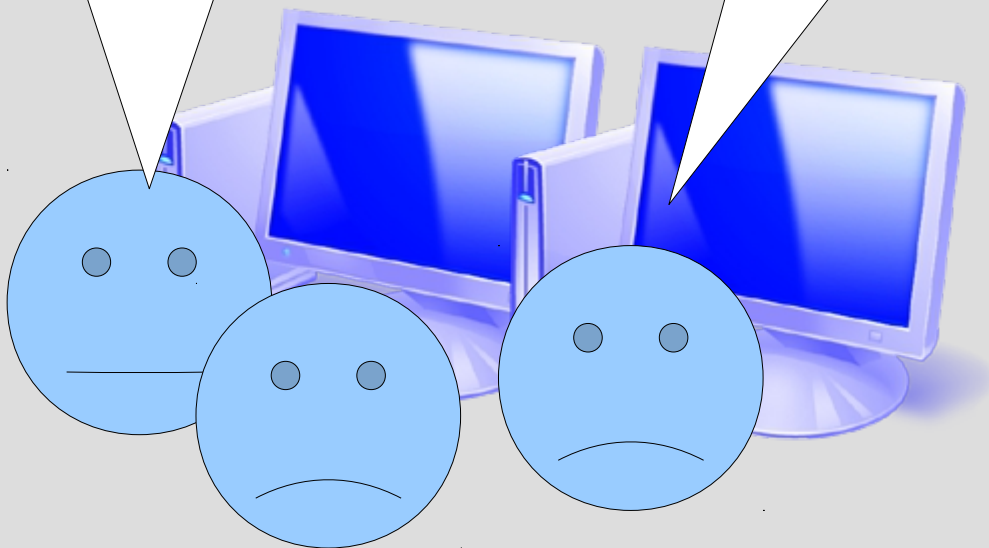


## Solution: Listener

Need: to refresh the graphical user interface

How to do that?

We may apply the pattern Listener.



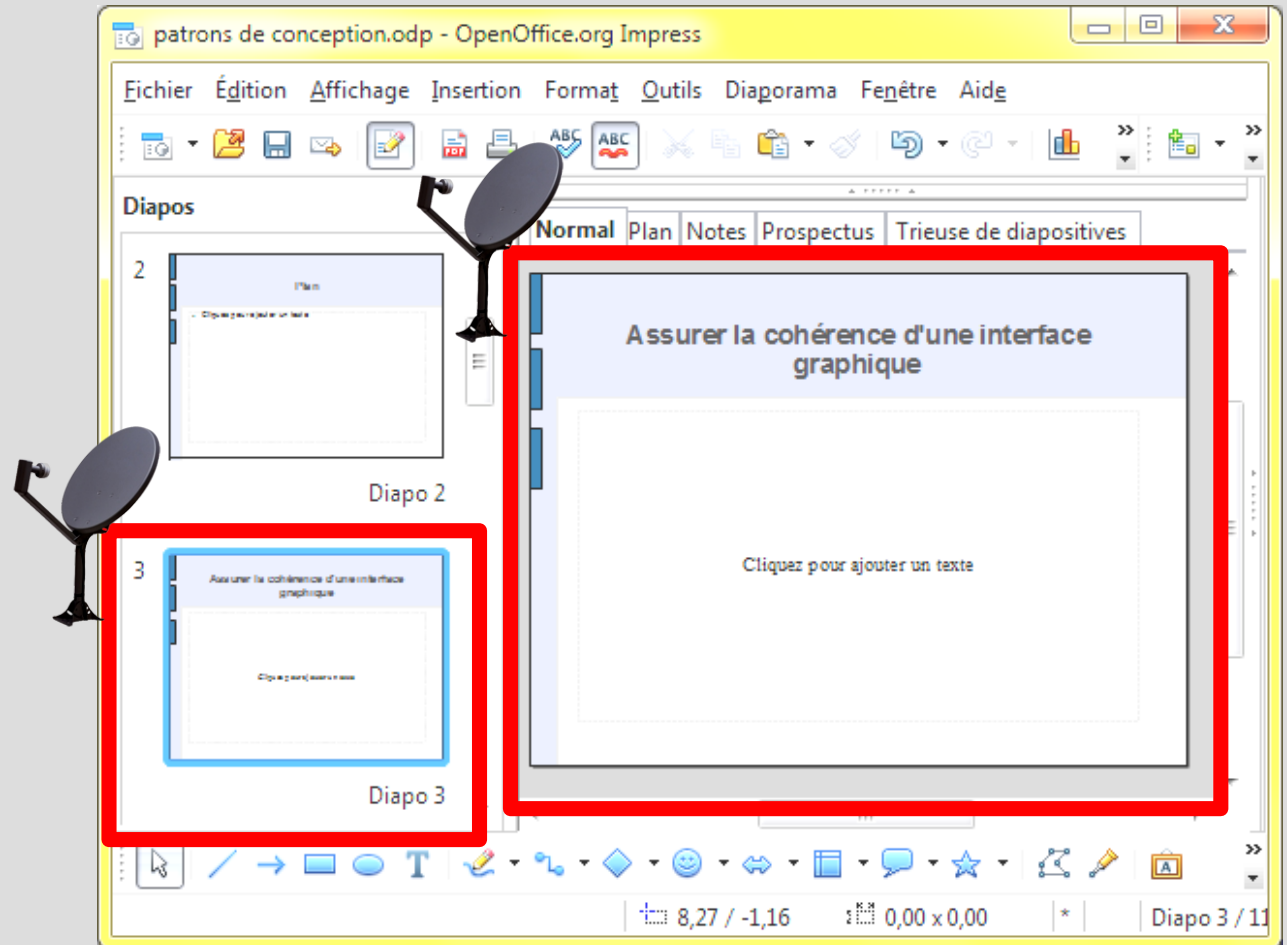


# Listener

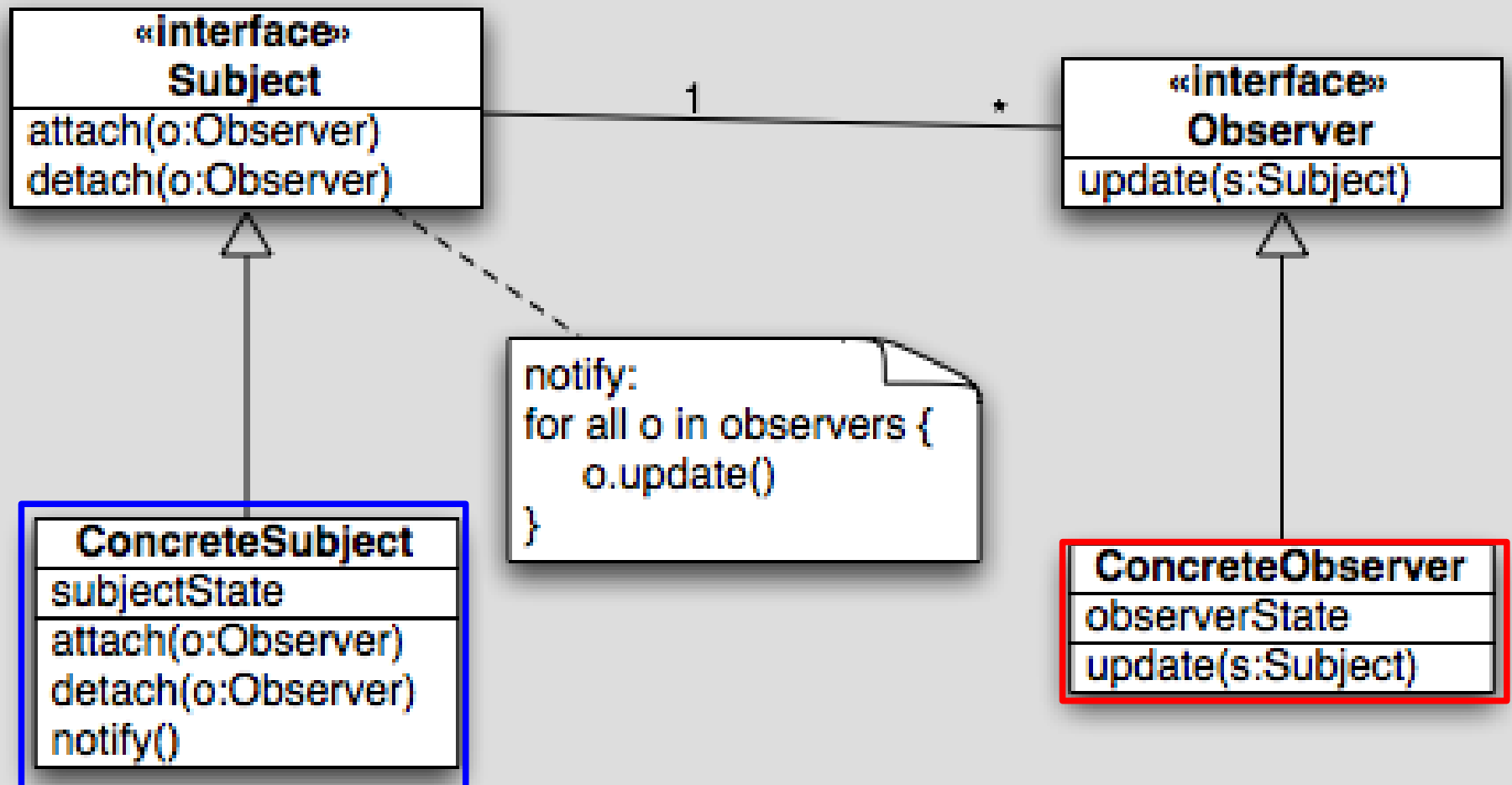
Data



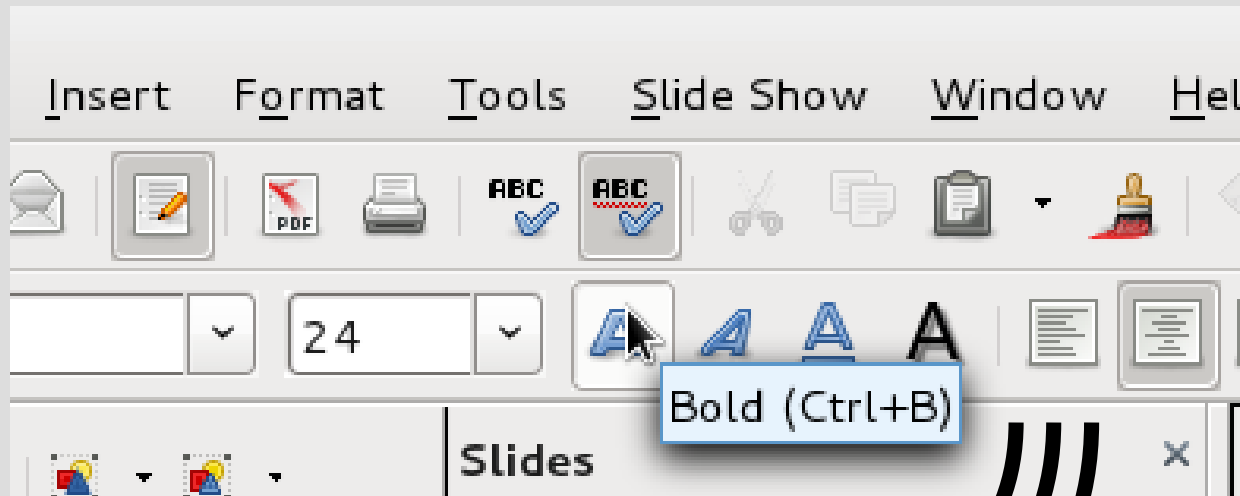
The two zones  
(listeners)  
listen the data  
(subject).



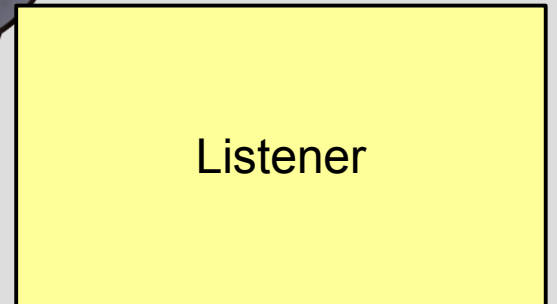
# Listener



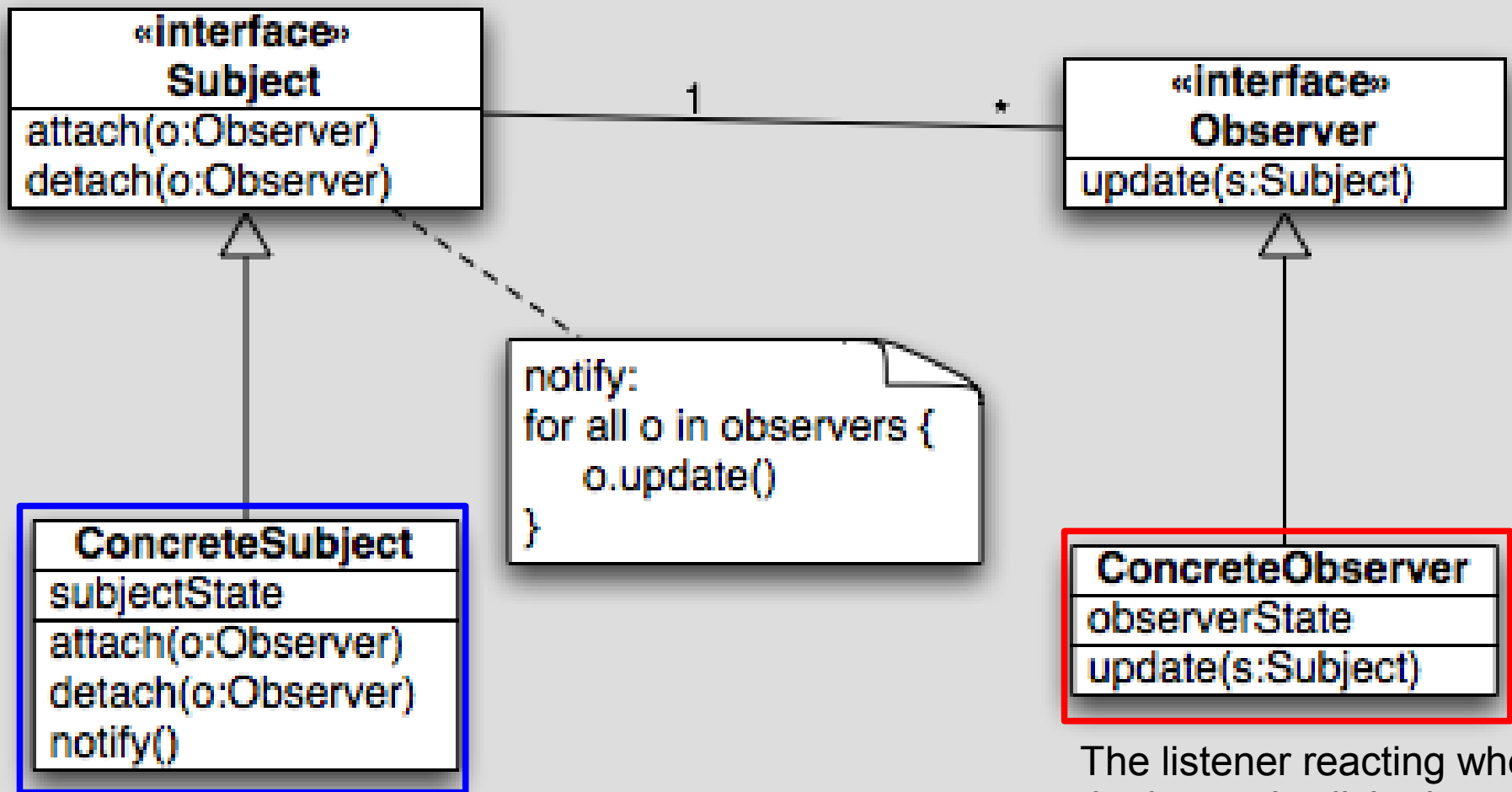
# Other application of the Listener design pattern: input/output



Click!



# Other application of the Listener design pattern: mouse events in JAVA Swing

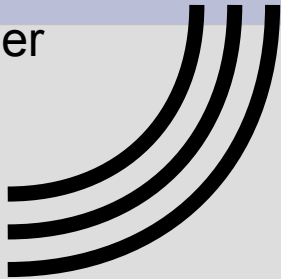


A button for instance

The listener reacting when the button is clicked

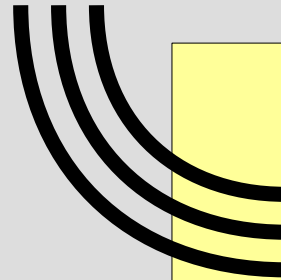
# Model - view - controller

user



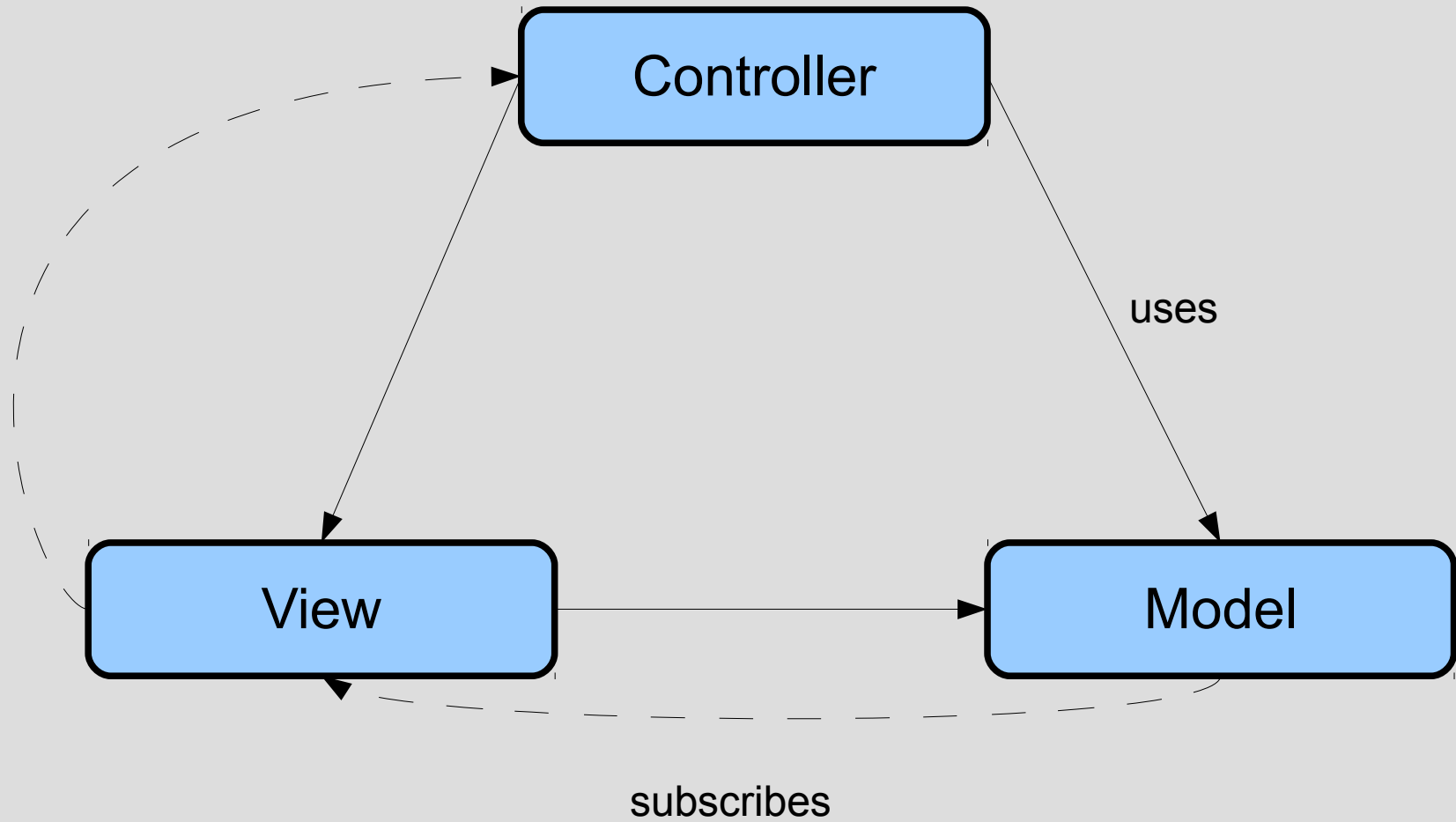
Controller

View

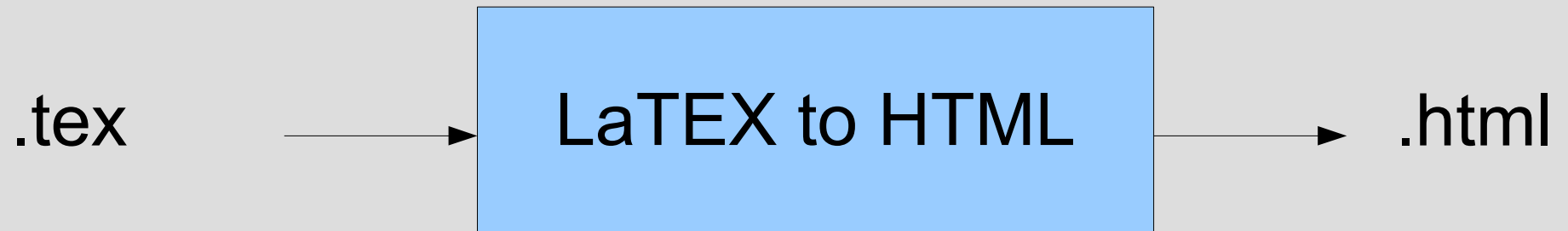


Model  
(data)

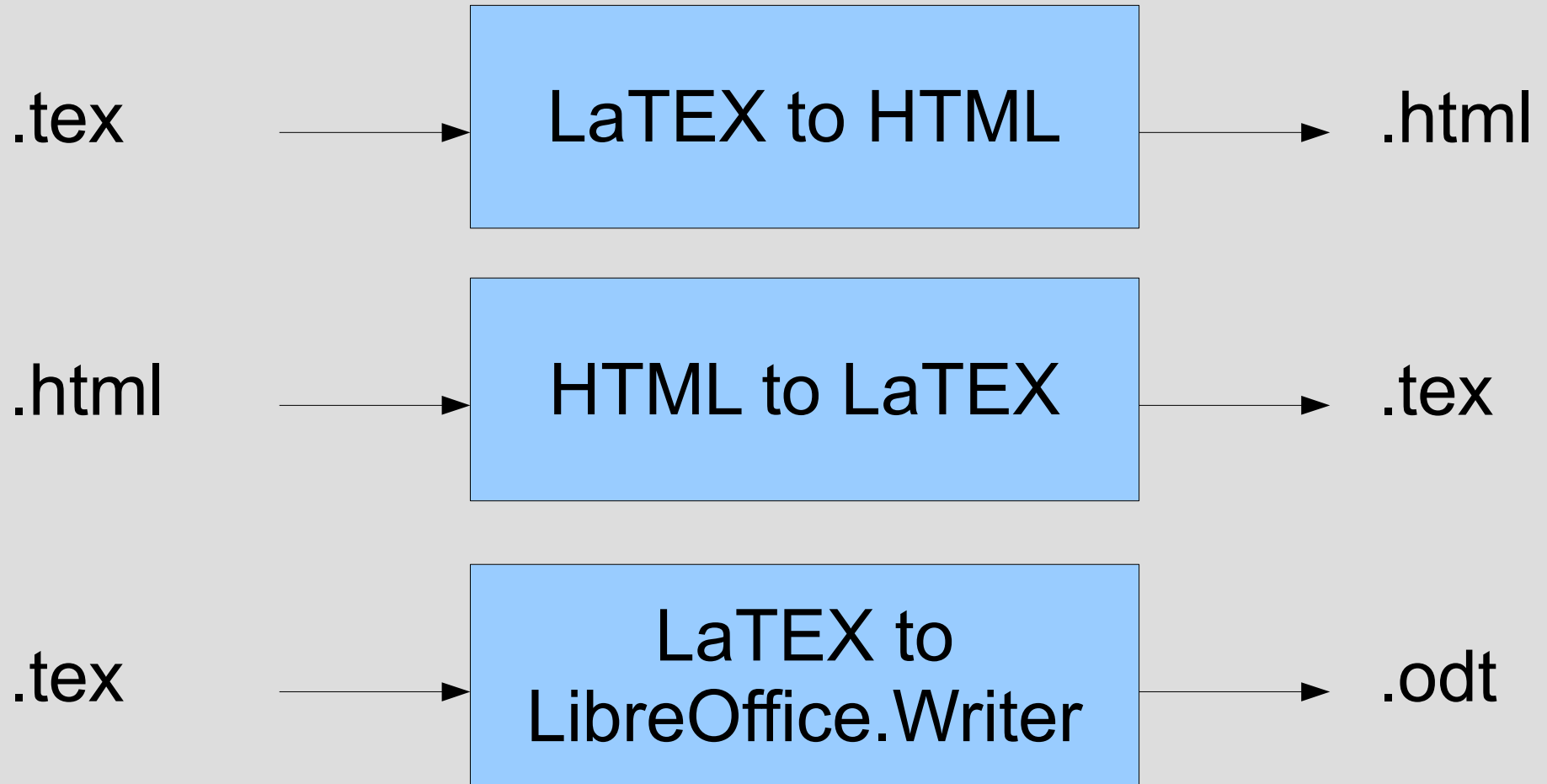
# Model - view - controller



## Another application for Listener: data converter

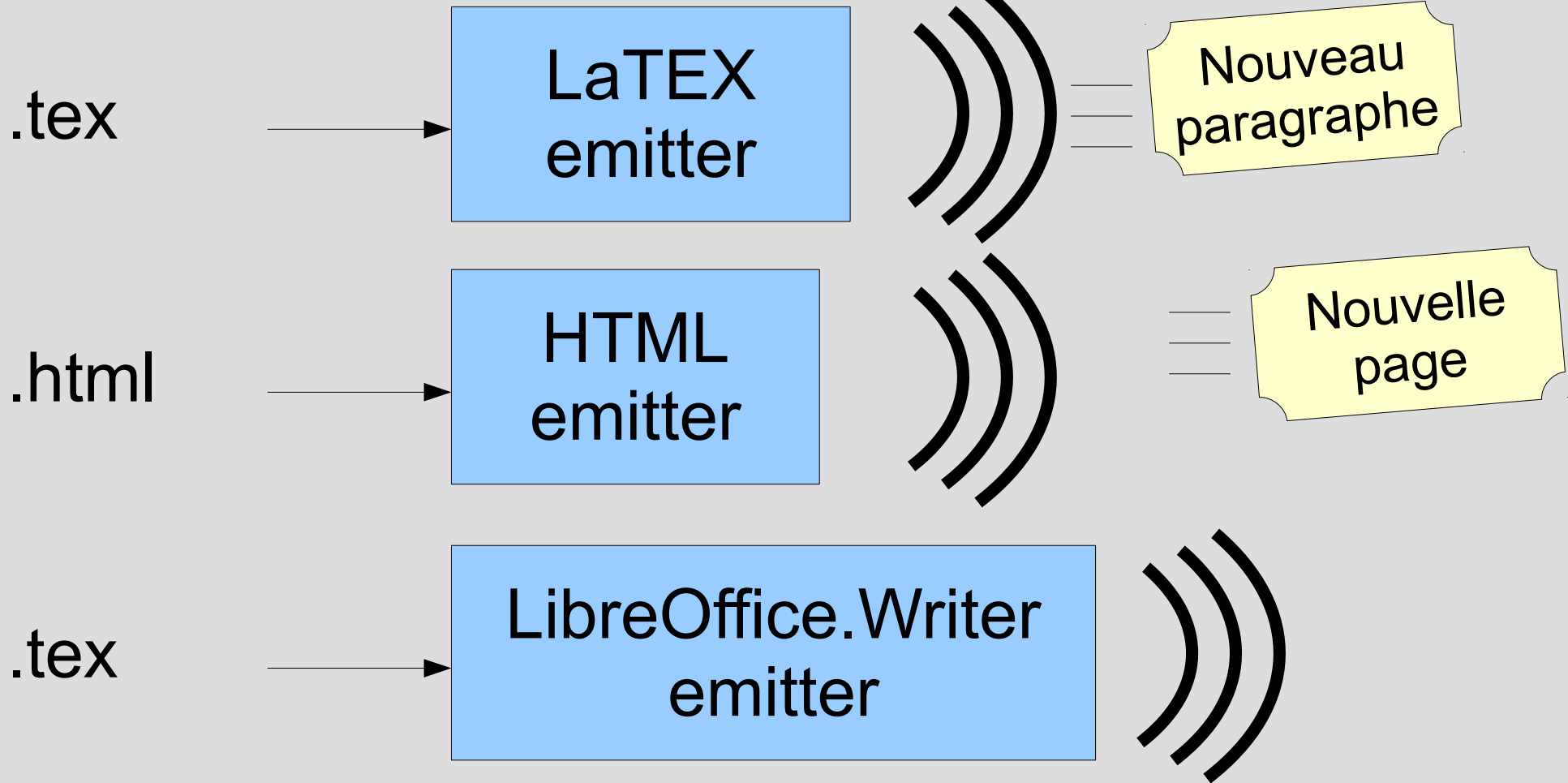


## Another application for Listener: data converter

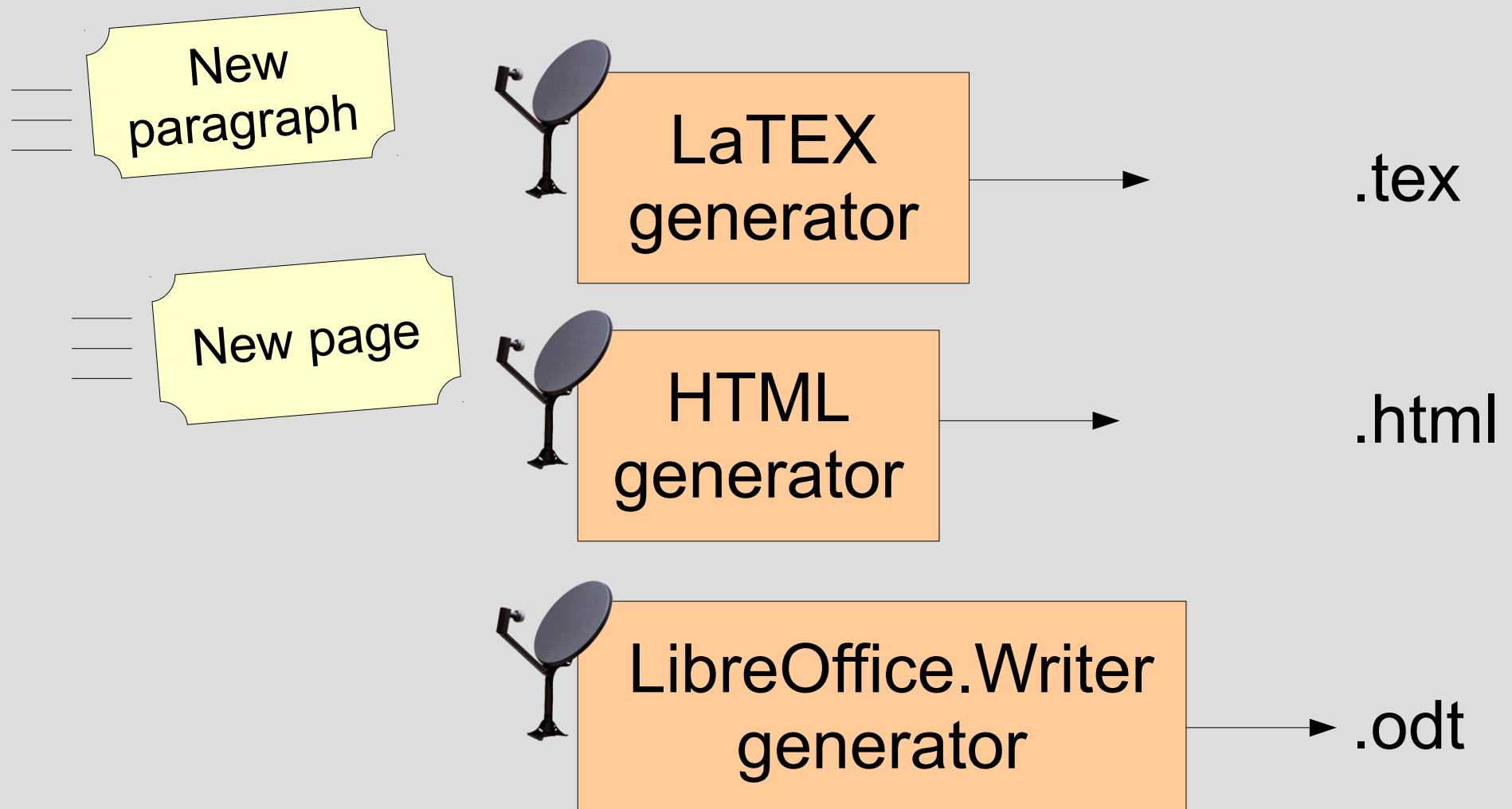




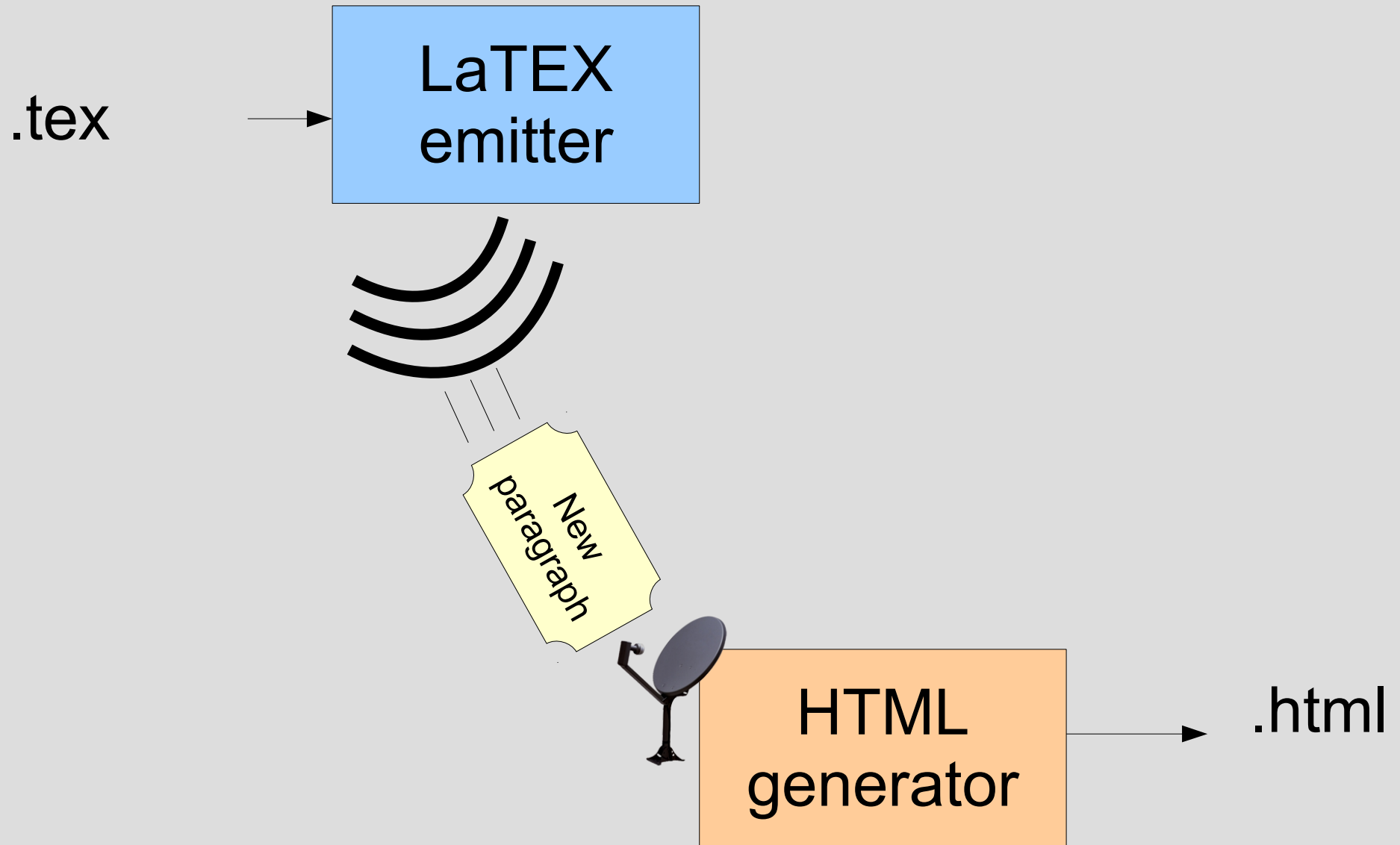
## Another application for Listener: data converter



## Another application for Listener: data converter



## Another application for Listener: data converter

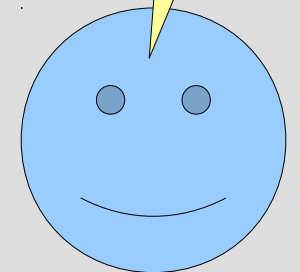
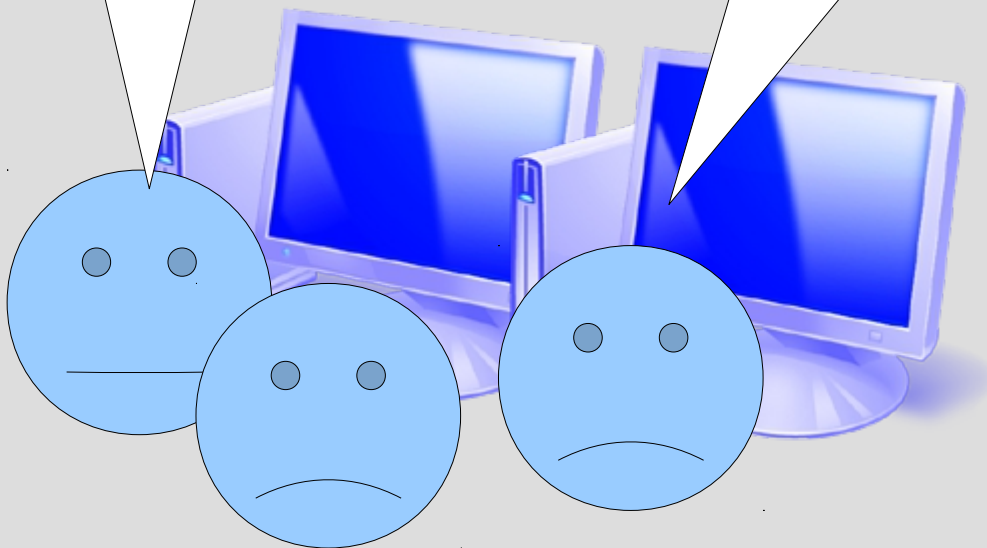


## Design pattern "Command"

We want to be able to cancel.

How to do that?

We may apply the design pattern "command".



Is the principle “action = operation” good?

### Dessin

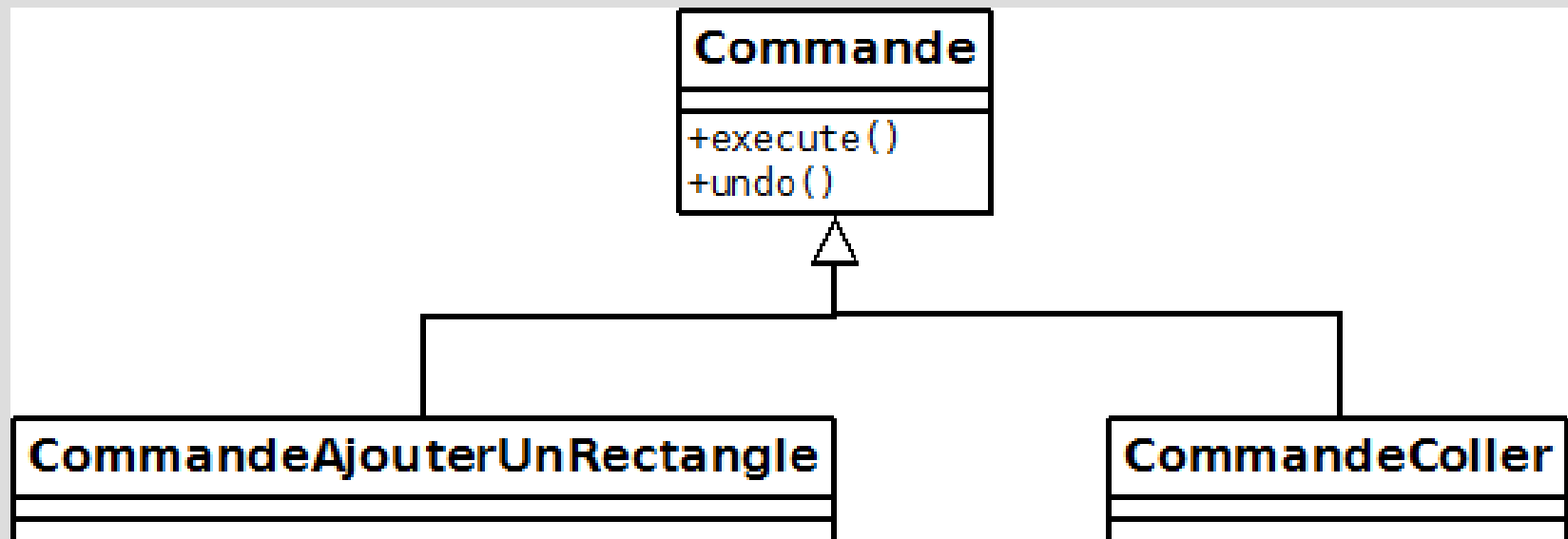
```
+copier(): Selection  
+couper(): Selection  
+coller(selection:Selection)  
+ajouterRectangle(r:Rectangle)  
+supprimer(selection:Selection)
```

## Problem

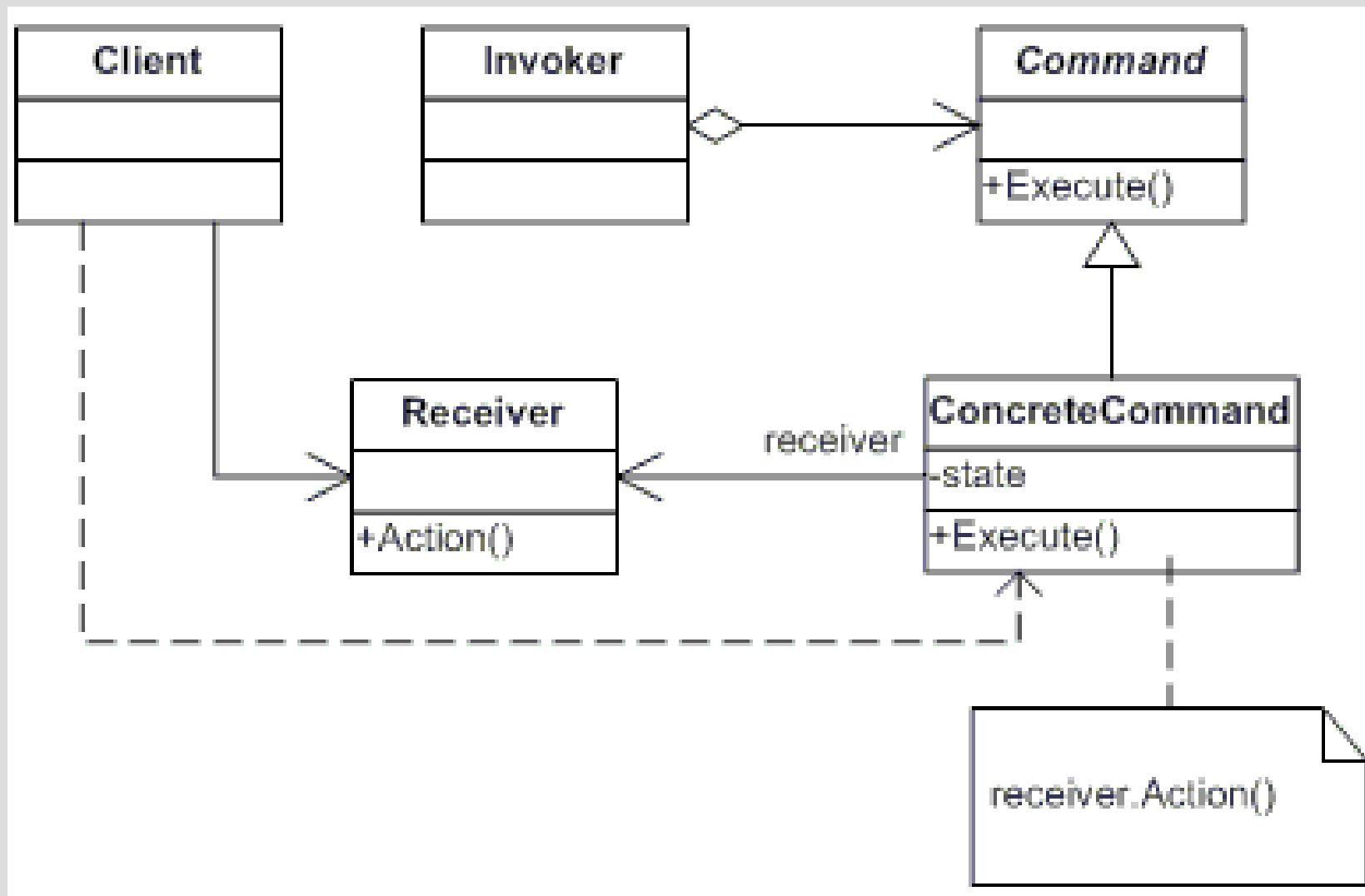
- Cancel?
- Save macros?
- Too many responsibilities for the class “Dessin”



## Solution: design pattern “command”



## Solution: design pattern “command”



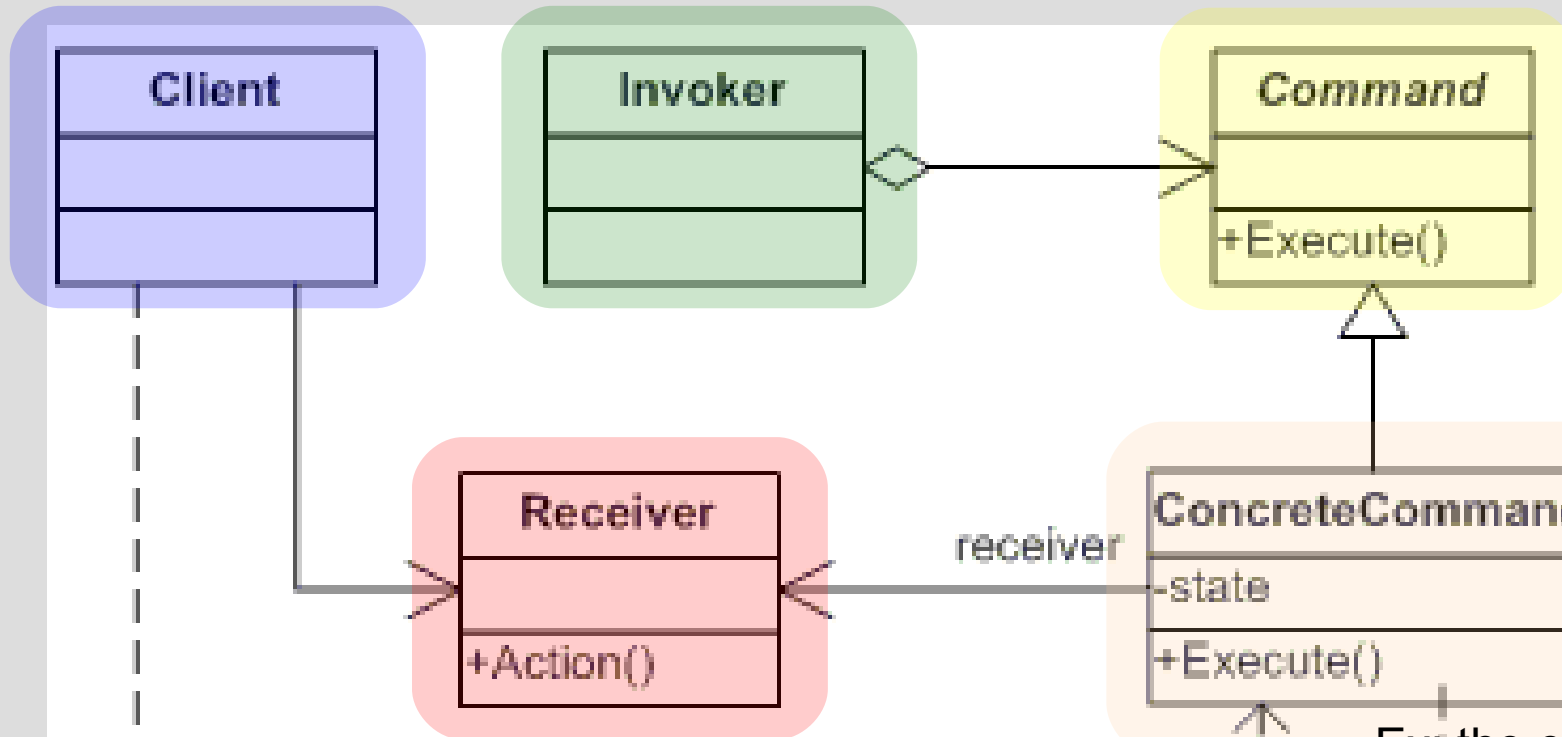


# Solution: design pattern "command"

The controller

Command handler

interface

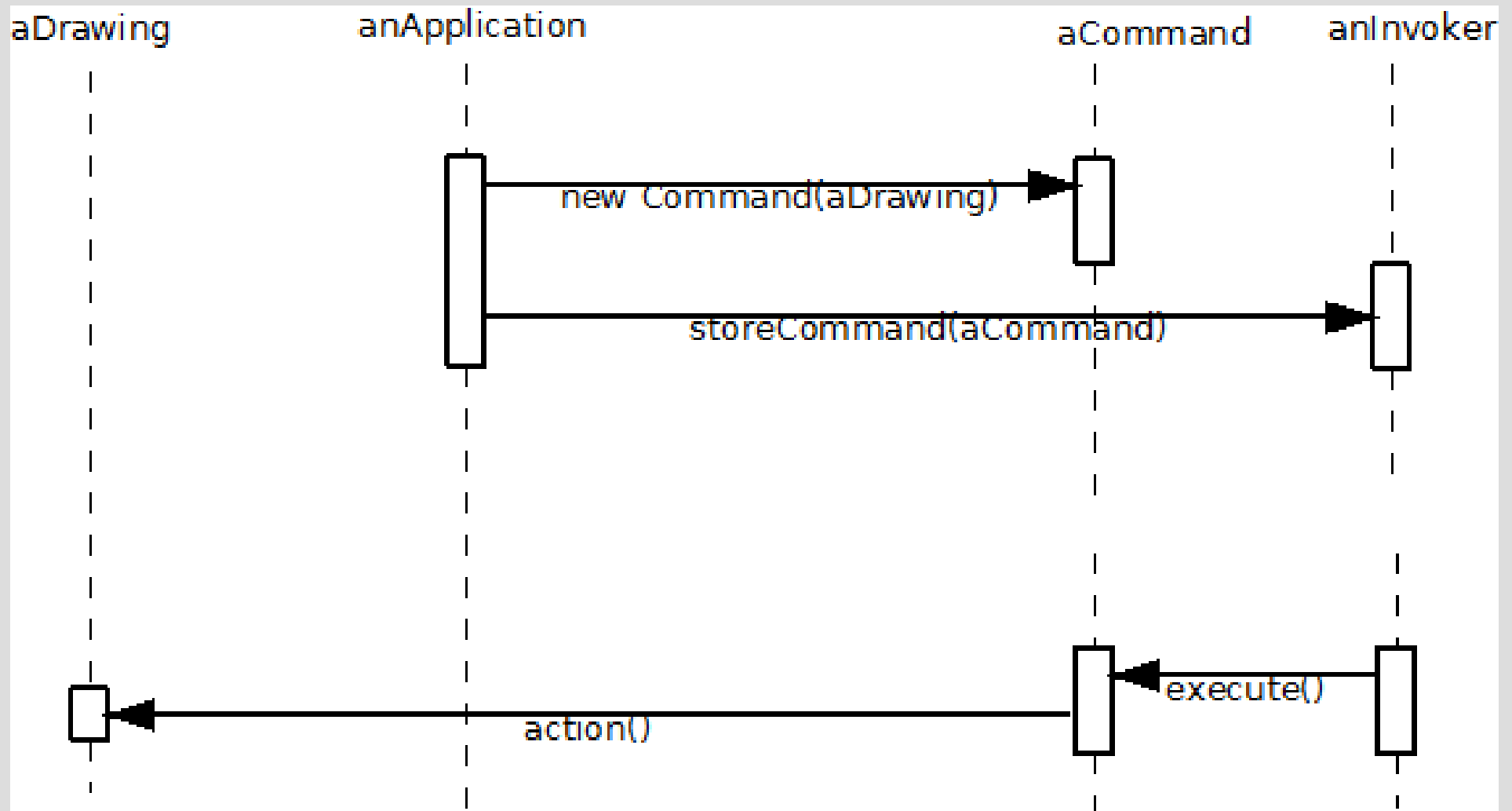


The drawing

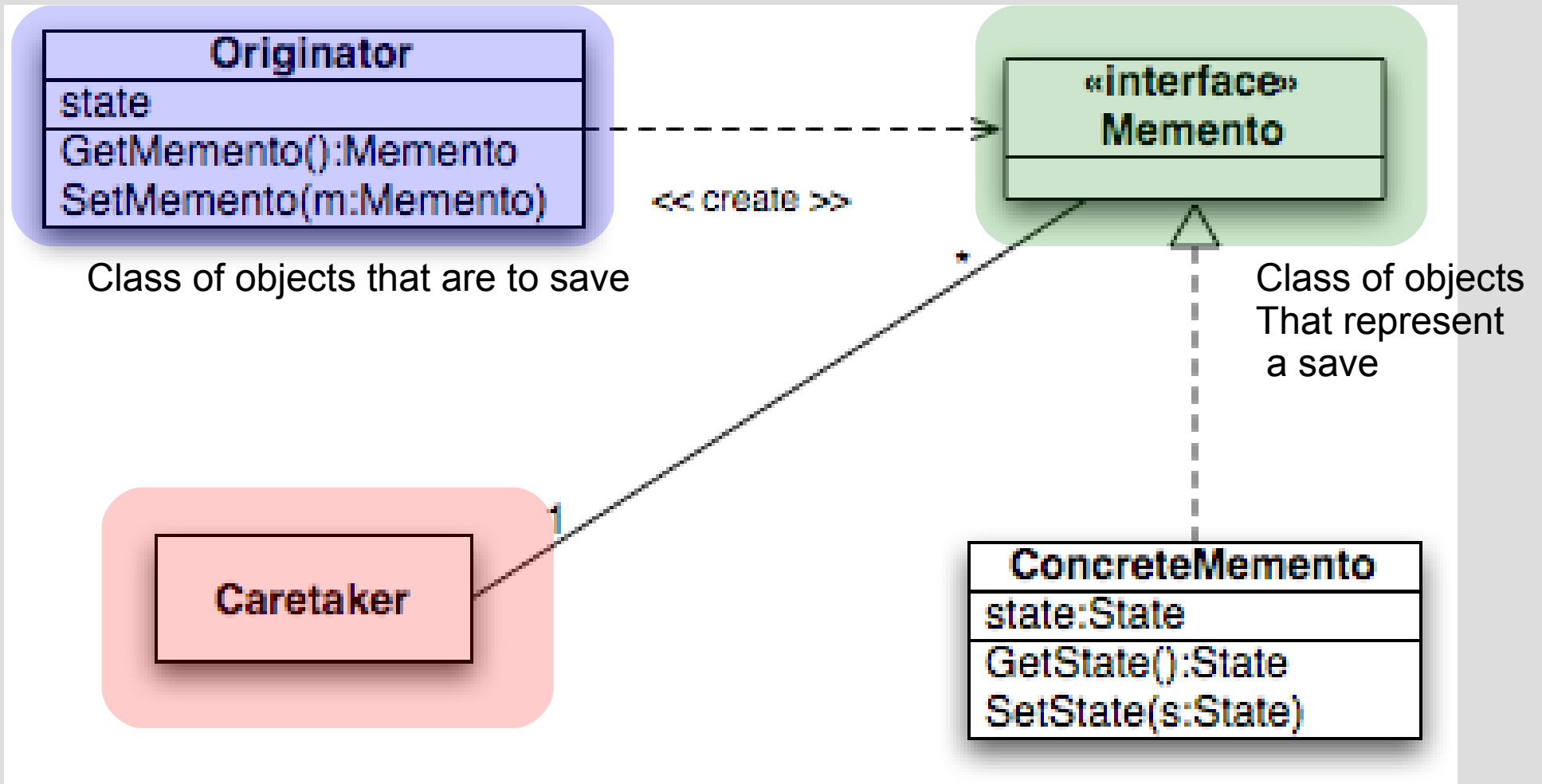
Ex: the command "Copy"

```
receiver.Action()
```

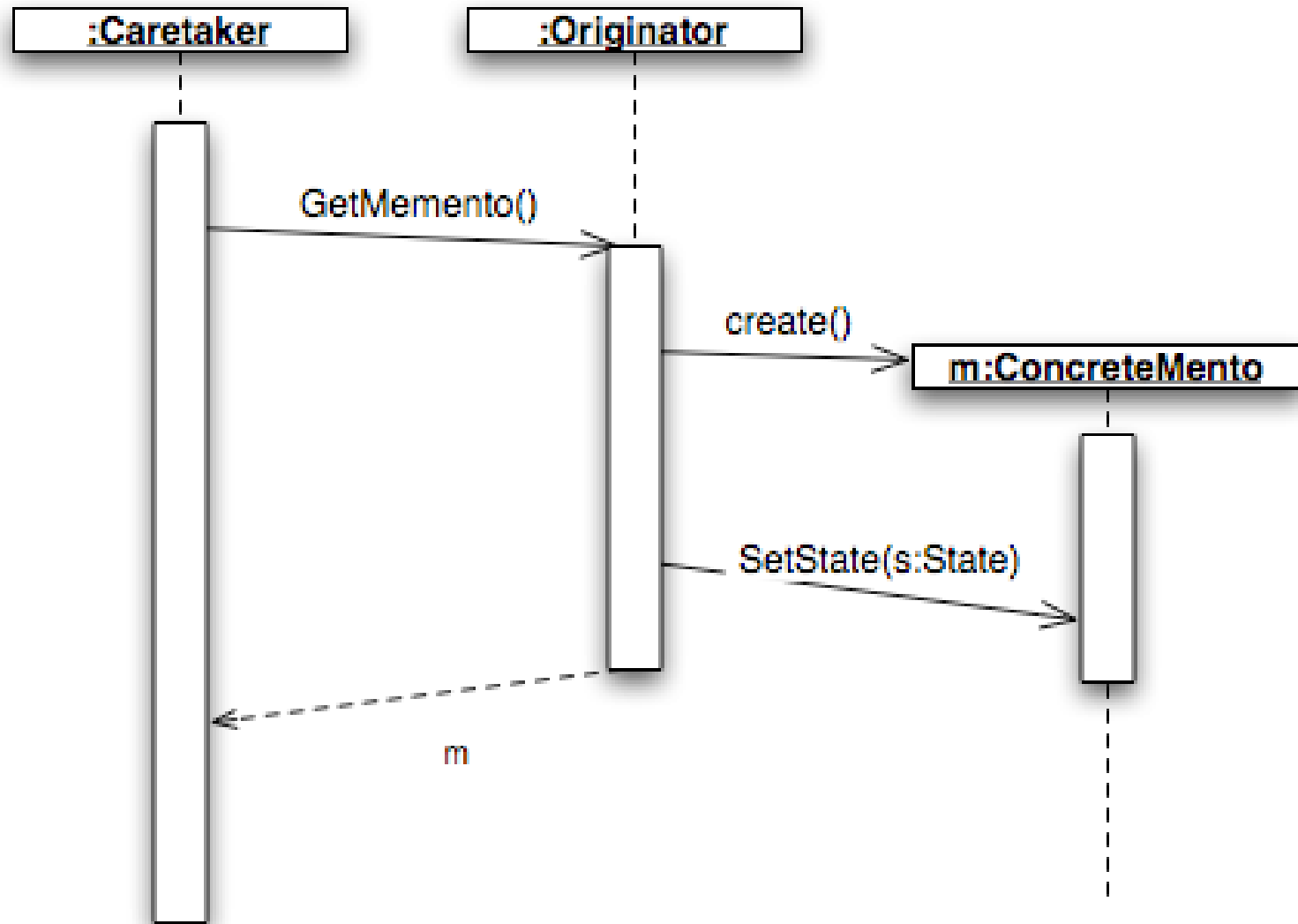
## Solution: design pattern “command”



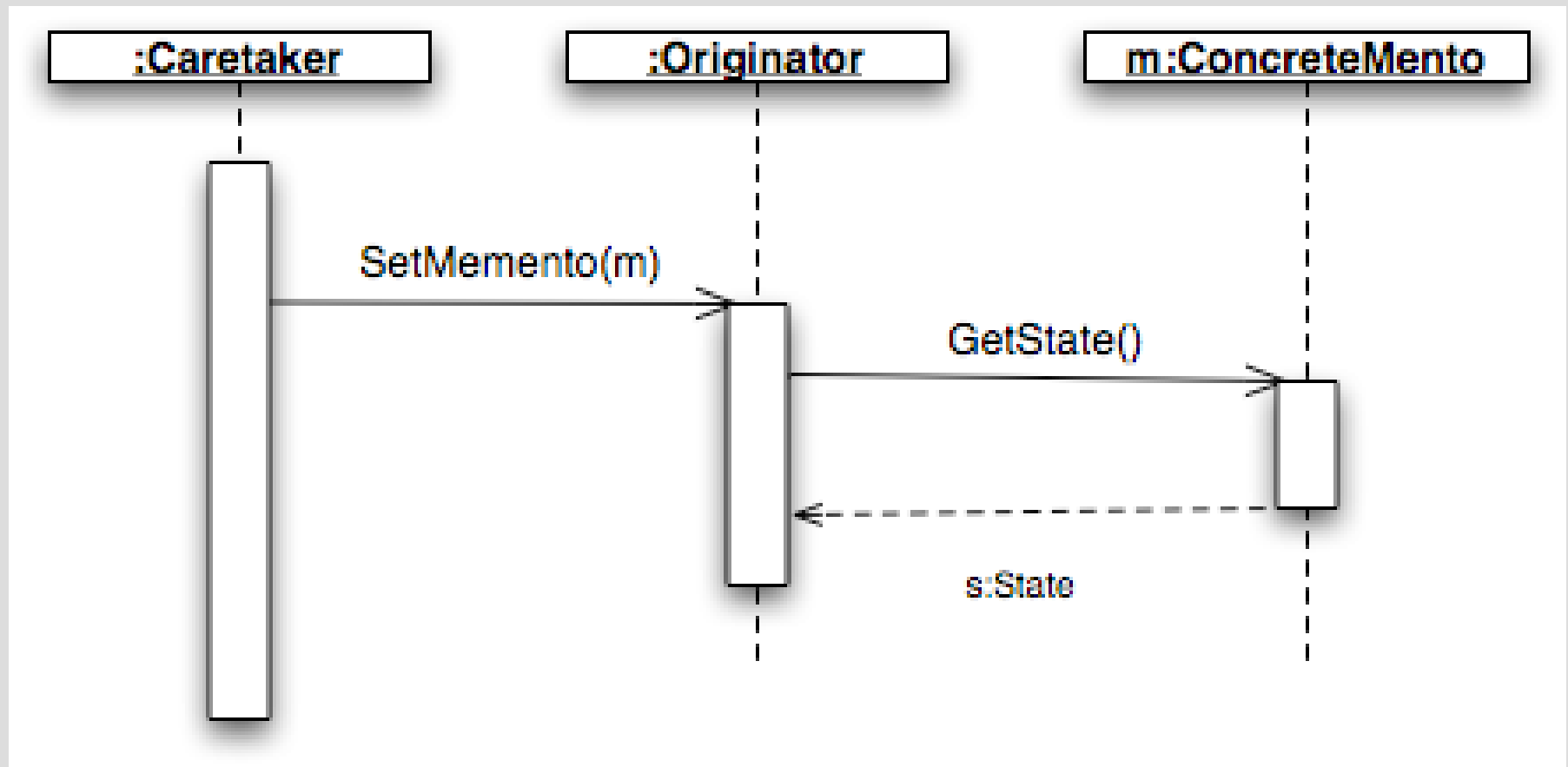
# Memento



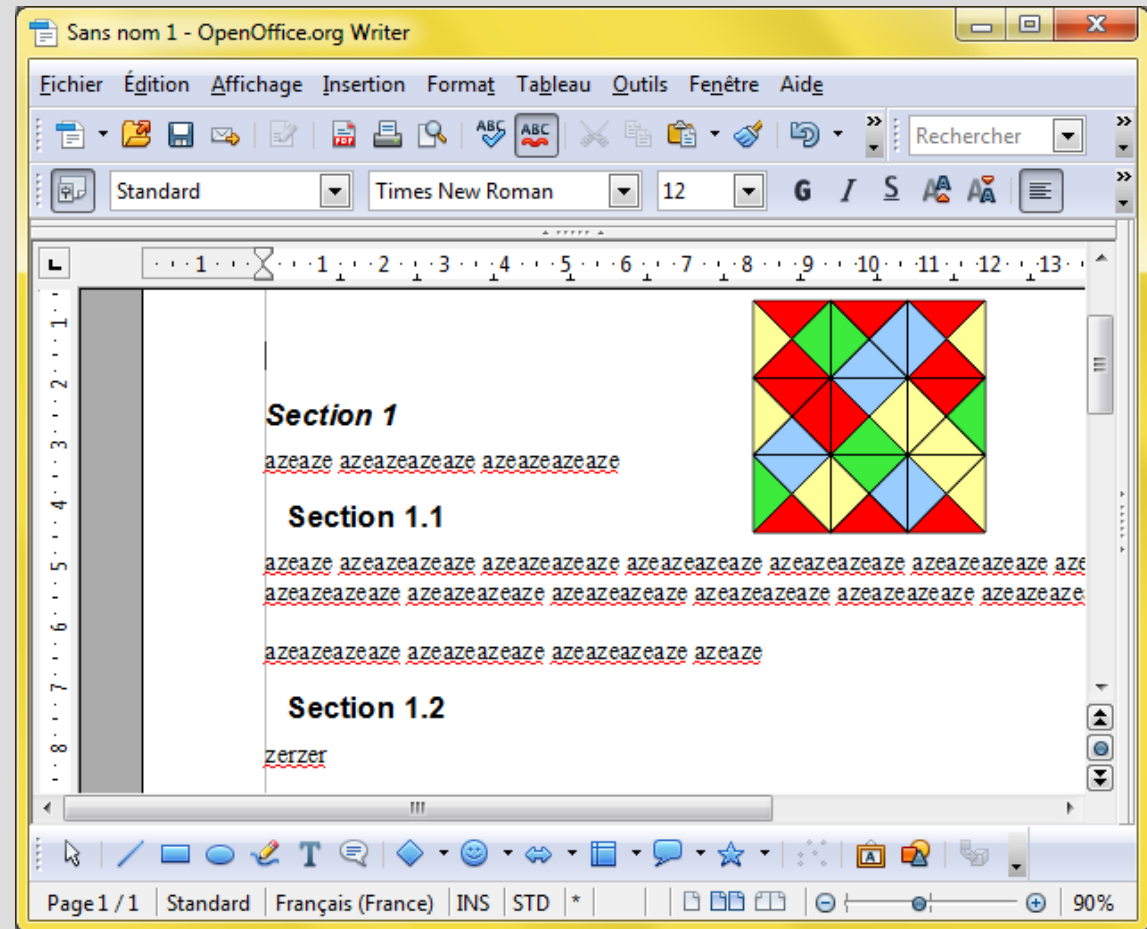
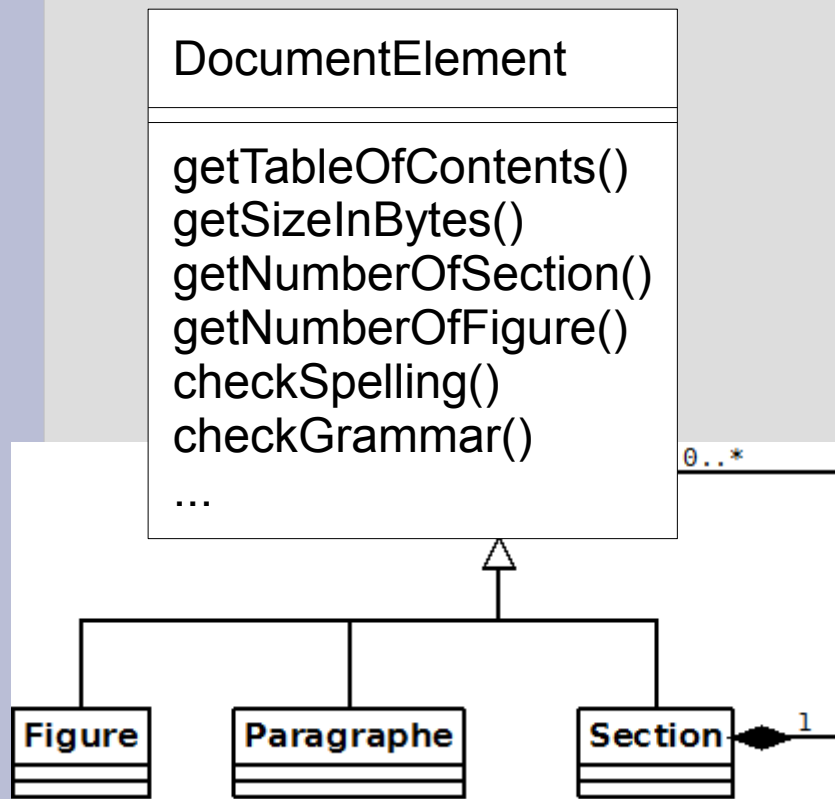
## Memento : saving



## Memento : restoration



# Operations on recursive structures

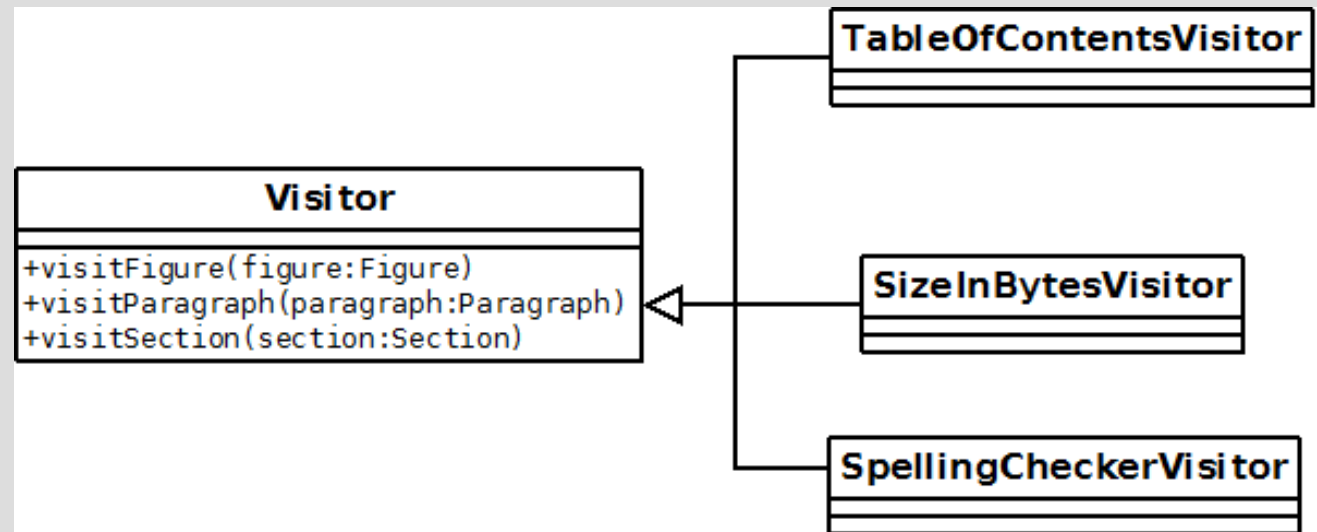
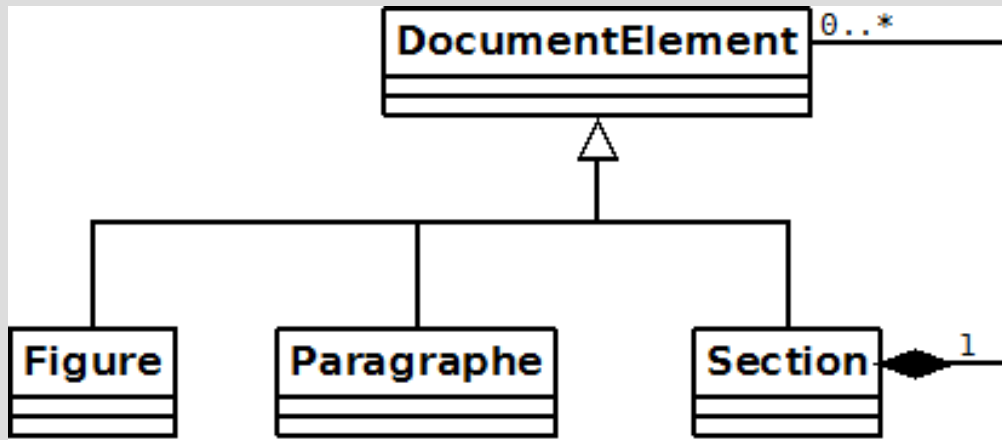


Problem: classes become really **big!**

## Example of other applications of the Visitor pattern

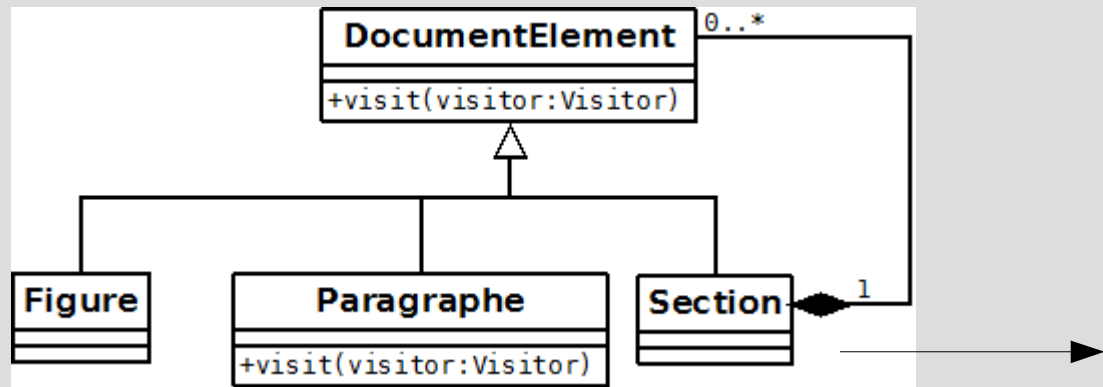
- Music score editor:  
Number of notes, display the score, etc.
- Proof assistant:  
Display the proof, check the proof, etc.
- 3D software  
to display the skeleton, compute the weight, etc.

# Solution: Visitor





# Solution: Visitor



## Class Section

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

## Class Paragraphe

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

## Solution: Visitor

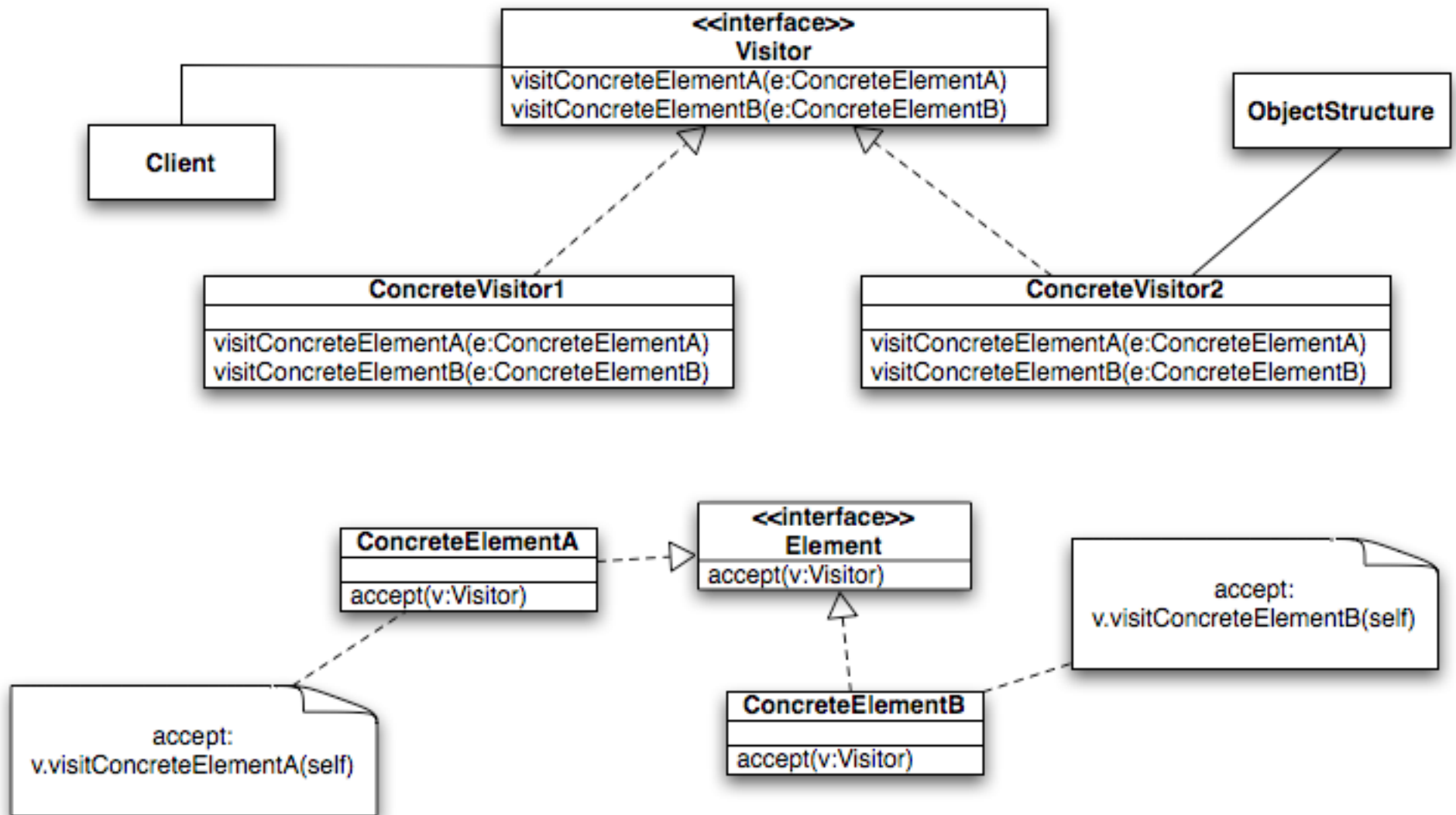


- Each class has a responsibility

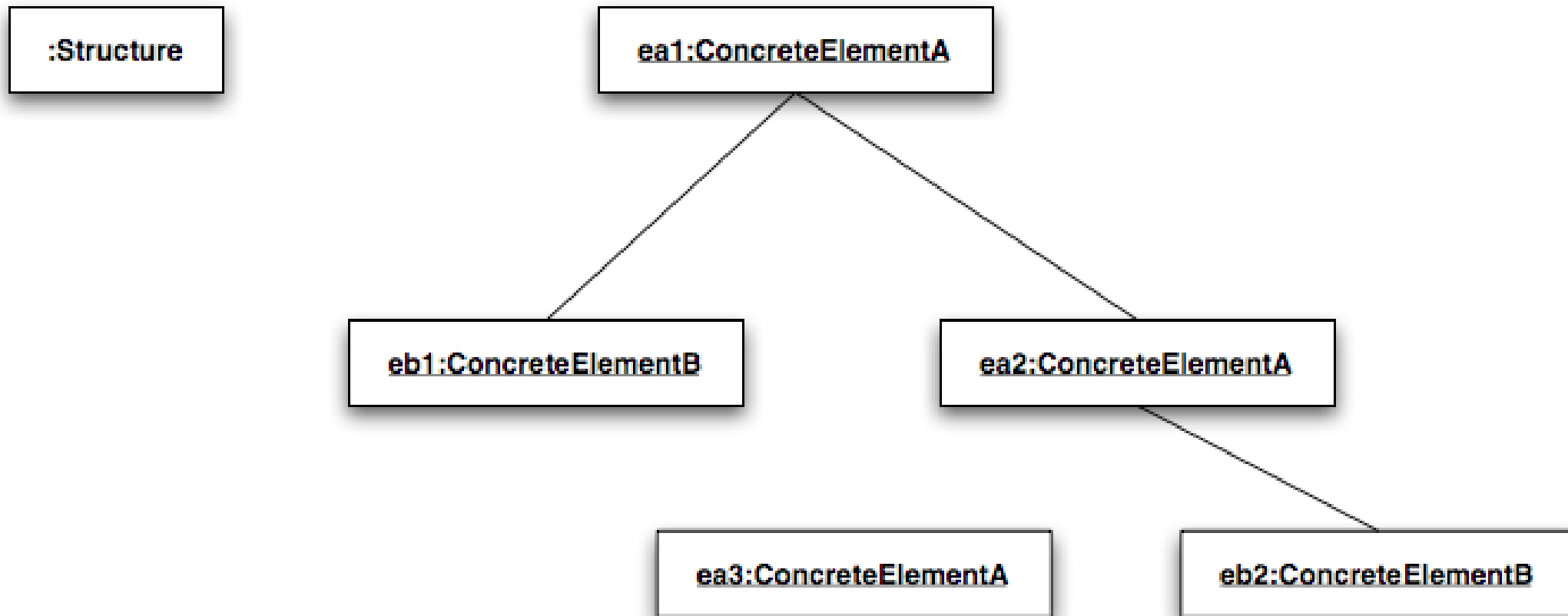


- Depends on the data
- Data classes must have public accessors.

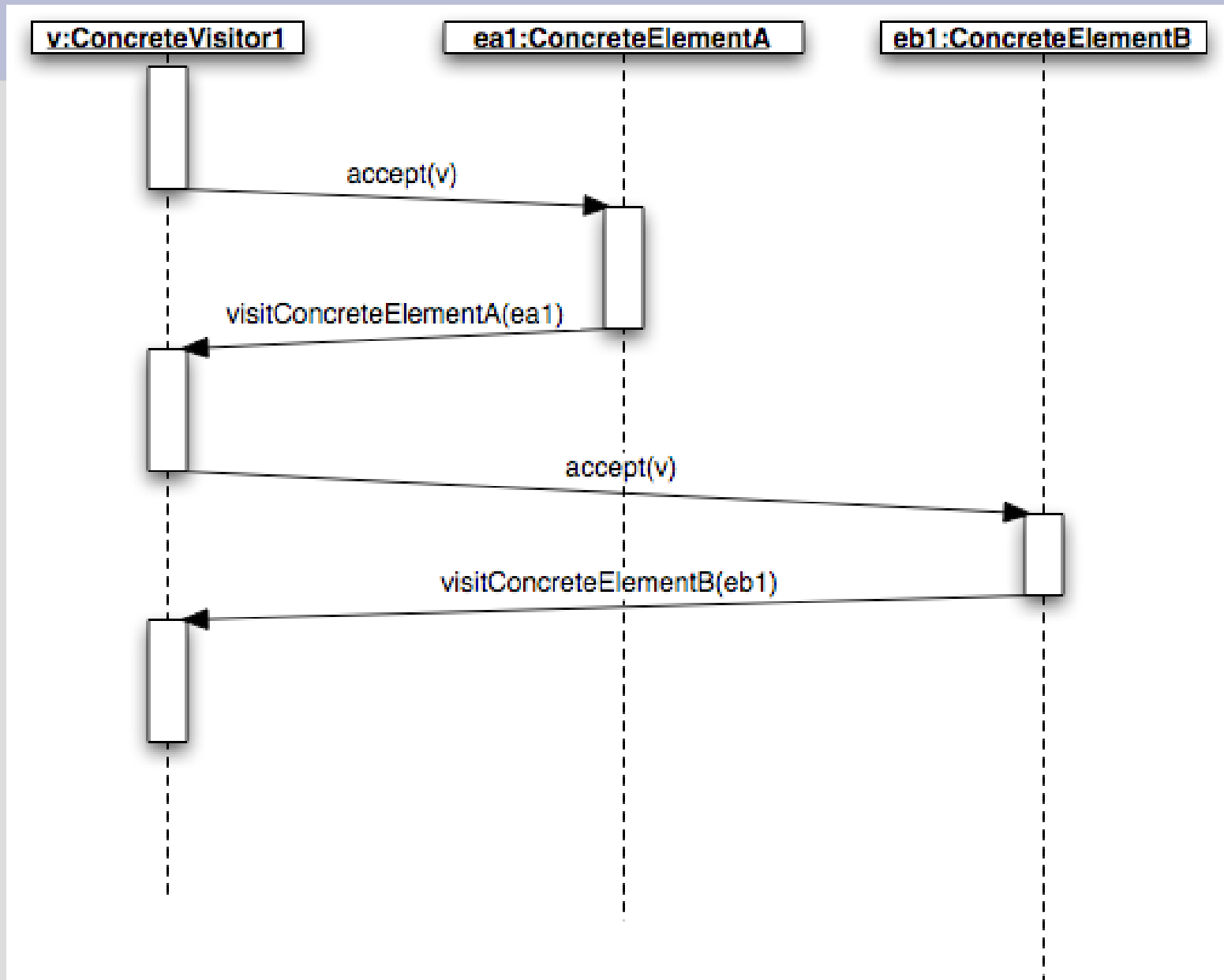
# Visitor



# Object diagram



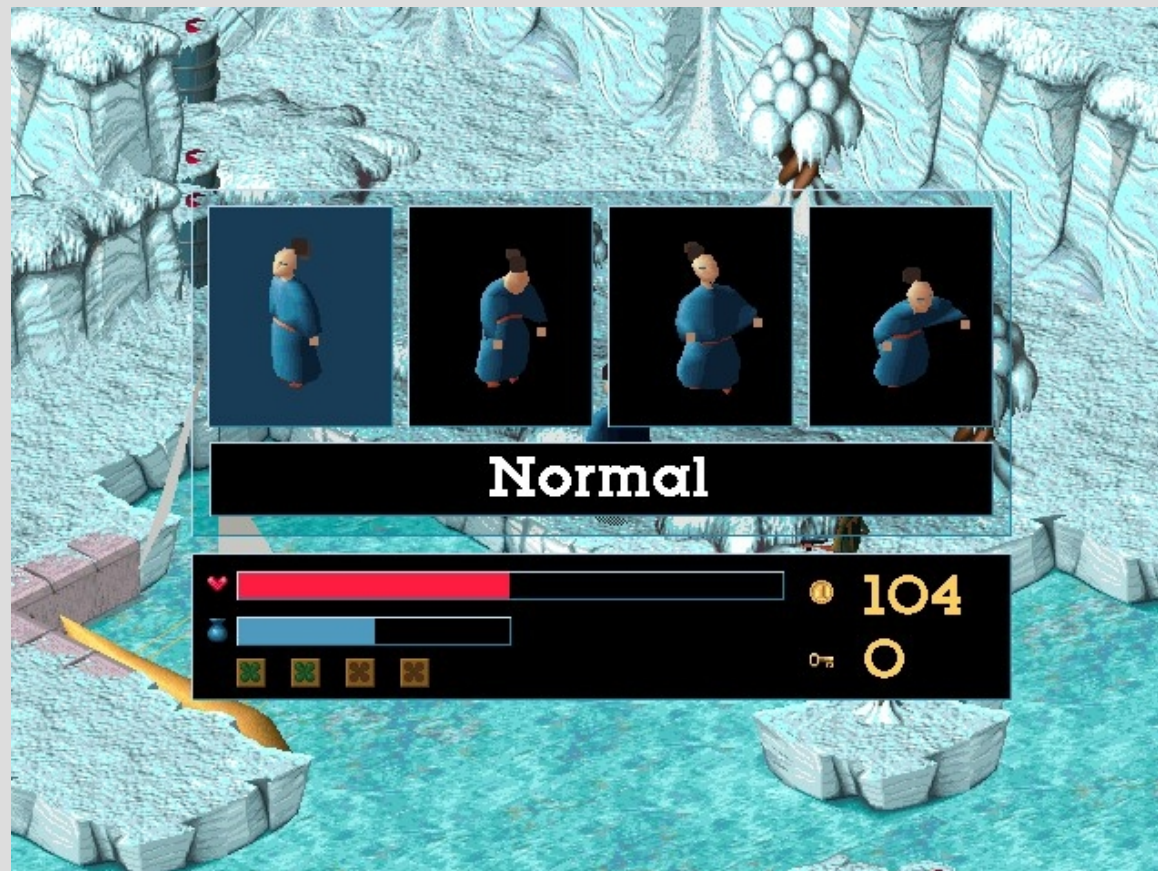
# Visitor



## Implement different algorithms: design pattern « Strategy »

## Need of several behaviour of the hero

hero.move ()



Source : Little Big Adventure 2

## Need of several sorting algorithms

```
array.sort()
```

BubbleSort ?

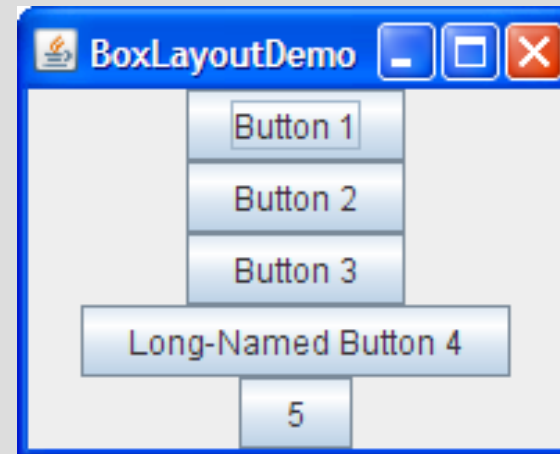
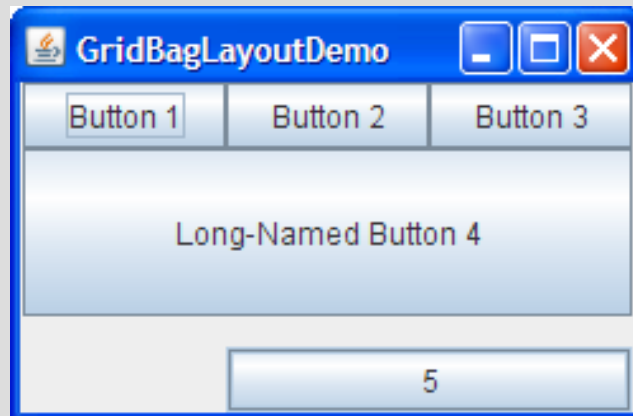
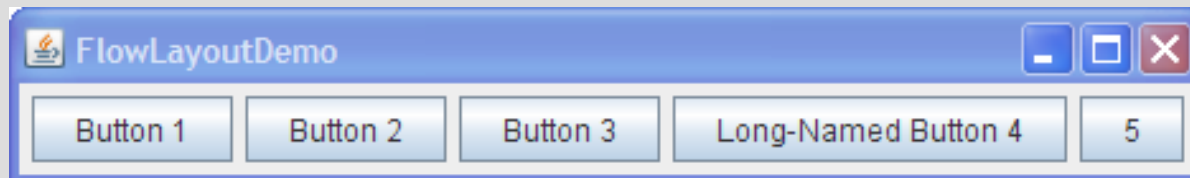
QuickSort ?

HeapSort ?

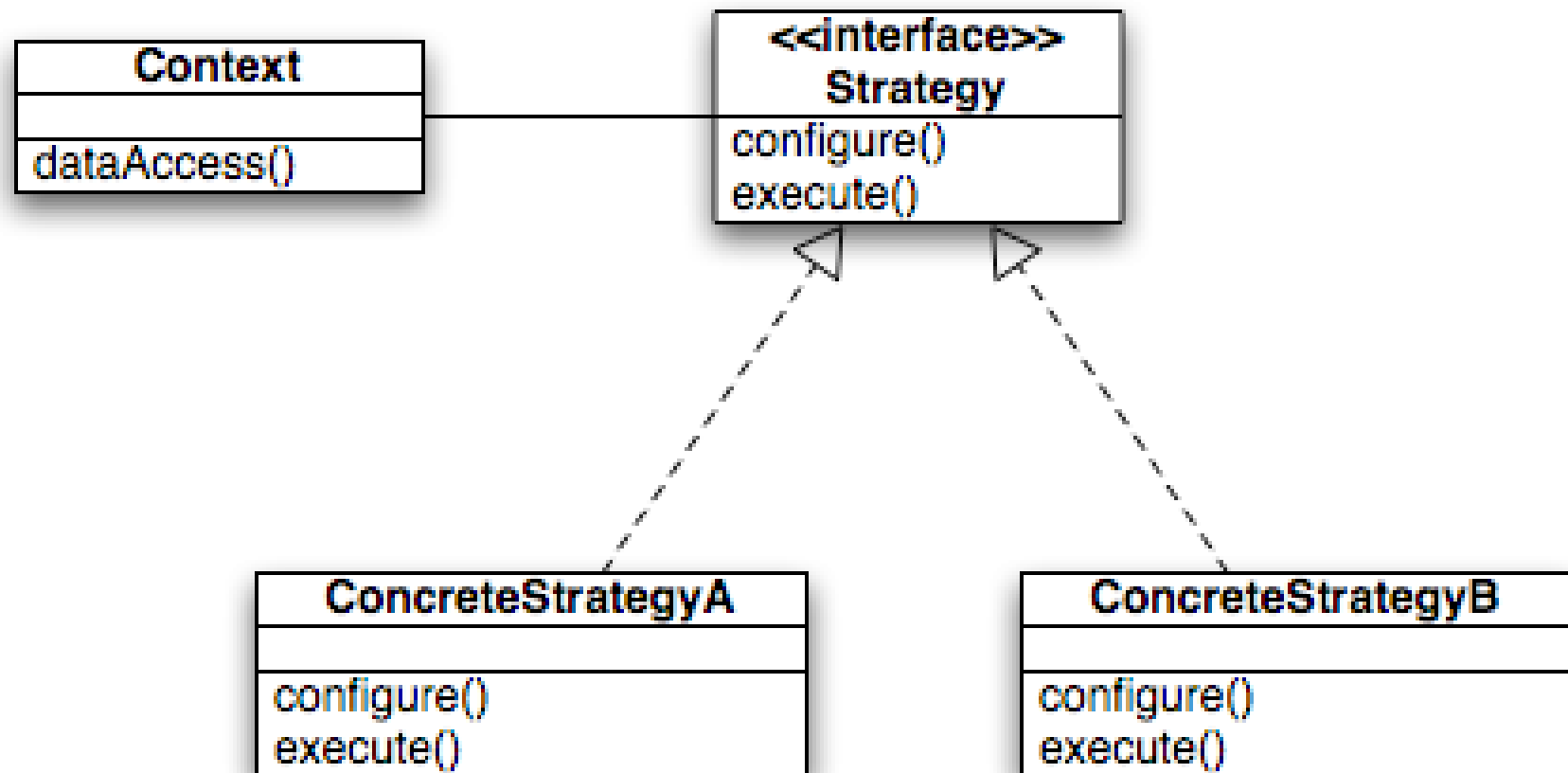


## Need of several layout algorithms

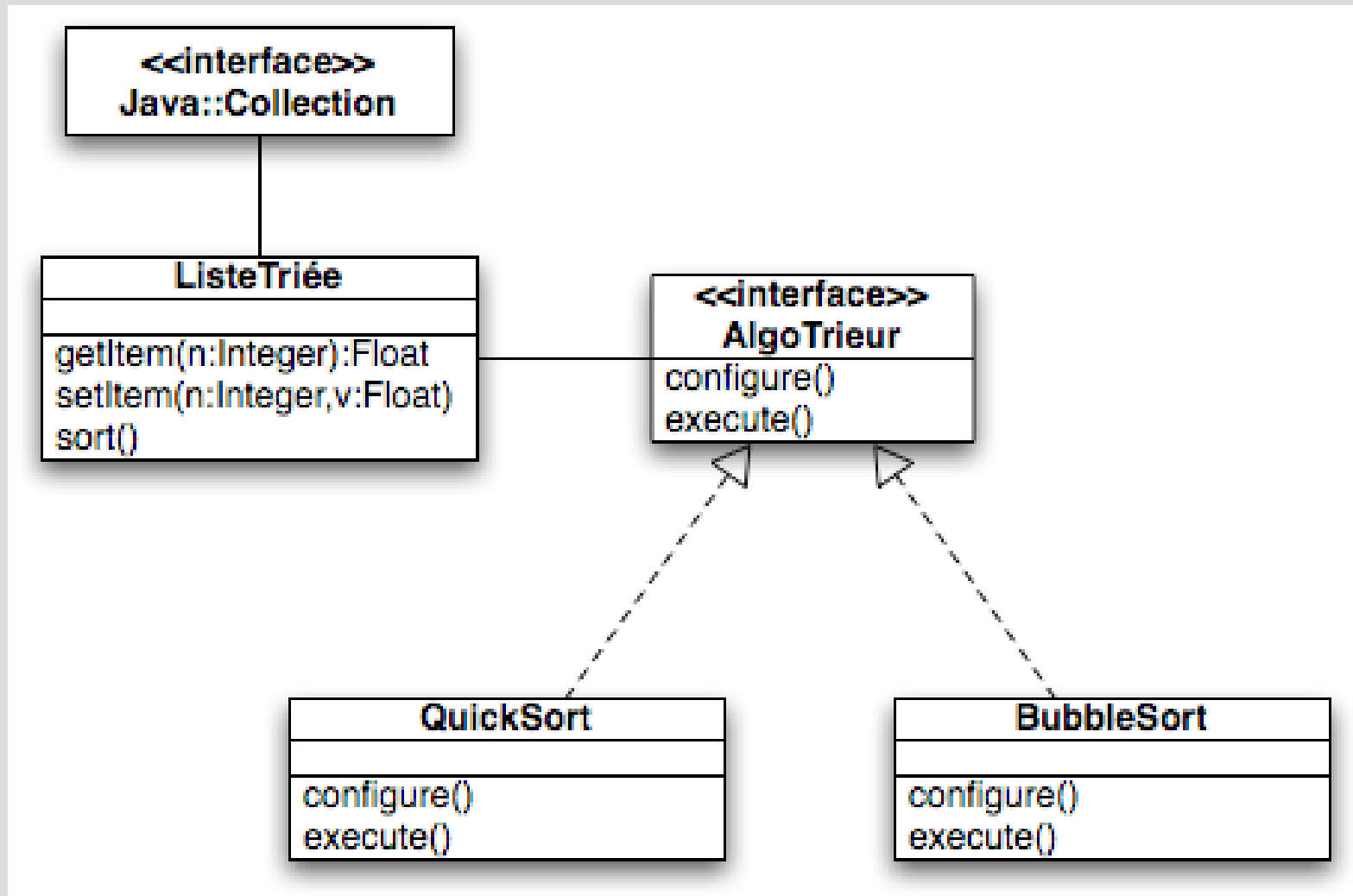
```
container.doLayout()
```



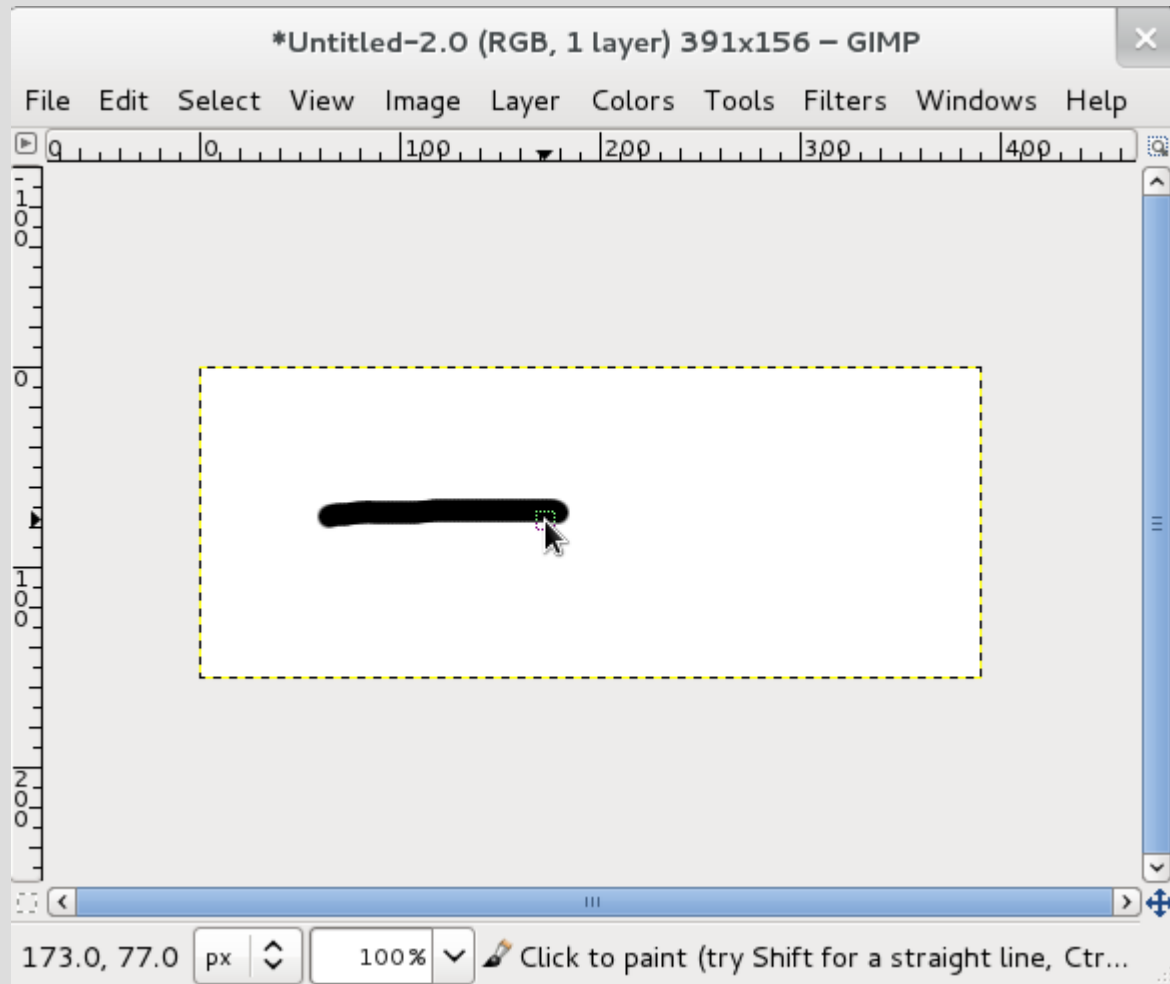
# Strategy



# Example

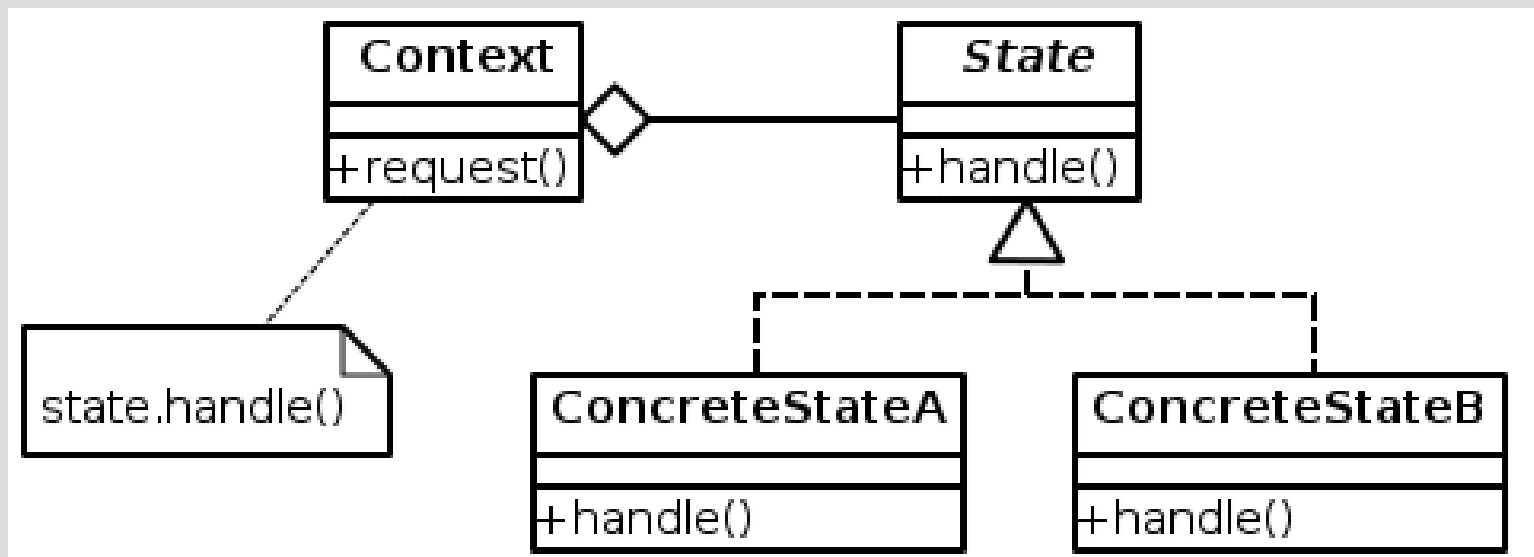


# Design pattern « state »



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

# “State” pattern



# “State” pattern

```
public class Drawing {
    private DrawingState myState;
    public Drawing() {
        setState(new DrawingStatePen());
    }

    :
    public void setState(DrawingState newState) {
        this.myState = newState;
    }

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

```
class DrawingStateSelection implements DrawingState {
    :
    public void mouseUp()
    {
        :
        drawing.setState(new DrawingStatePen());
    }
}
```