# NSY102
# Conception de logiciels Intranet
# Quelques Patrons

Cnam Paris
jean-michel Douin, douin au cnam point fr
22 Février 2016

**Notes de cours**

---

**Avertissement** la plupart des diapositives ont été extraites des supports de l'unité NFP121

# Sommaire

- **Historique**
  - **Gains escomptés**
- **Les fondamentaux …**
- **Les patrons de base**
  - **Adapter, Proxy, Template Method, Composite, Observer**
- **Les structurels**
- **Les comportementaux**

# Principale bibliographie utilisée

- [Grand00]
  - Patterns in Java le volume 1

    http://www.mindspring.com/~mgrand/

- [head First]
  - Head first : http://www.oreilly.com/catalog/hfdesignpat/#top

- [DP05]
  - L'extension « Design Pattern » de BlueJ : http://hamilton.bell.ac.uk/designpatterns/

- [divers]
  - Certains diagrammes UML : http://www.dofactory.com/Patterns/PatternProxy.aspx
  - informations générales http://www.edlin.org/cs/patterns.html

# Design Patterns Pourquoi ?

# Patrons/Patterns pour le logiciel

- **Origine C. Alexander un architecte**
  - **1977, un langage de patrons pour l 'architecture 250 patrons**
- **Abstraction dans la conception du logiciel**
  - **[GoF95] la bande des 4 : Gamma, Helm, Johnson et Vlissides**
    - **23 patrons/patterns**
- **Une communauté**
  - **PLoP Pattern Languages of Programs**
    - **http://hillside.net**

# Introduction

- **Classification habituelle**
  - **Créateurs**
    - **Abstract Factory, Builder, Factory Method Prototype Singleton**
  - **Structurels**
    - **Adapter Bridge Composite Decorator Facade Flyweight Proxy**
  - **Comportementaux**

    **Chain of Responsability. Command Interpreter Iterator**
    **Mediator Memento Observer State**
    **Strategy Template Method Visitor**

- **Quelques patterns seulement !**

# Patron défini par J. Coplien

- *Un pattern est une **règle** en trois parties exprimant une relation entre un **contexte**, un **problème** et une **solution** ( **Alexander**)*

- **Summary by Jim Coplien:**

  ***Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.***

# Définition d 'un patron

- **Contexte**
- **Problème**
- **Solution**

- **Patterns and software :**
  - **Essential Concepts and Terminology par Brad Appleton http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html**

- **Différentes catégories**
  - **Conception (Gof)**
  - **Architecturaux(POSA/GoV, POSA2 [Sch06])**
  - **Organisationnels (Coplien www.ambysoft.com/processPatternsPage.html)**
  - **Pédagogiques(http://www.pedagogicalpatterns.org/)**
  - **……**

# Les fondamentaux [Grand00]

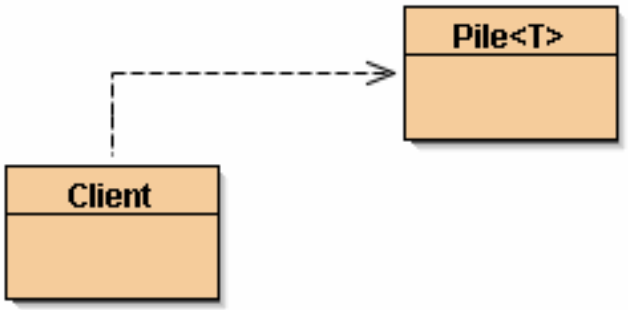
- **Constructions**
  - **Delegation**
  - **Interface**
  - **Abstract superclass**
  - **Immutable**
  - **Marker interface**

# Delegation

- **Ajout de fonctionnalités à une classe**
- **Par l'usage d'une instance d'une classe**
  - **Une instance inconnue du client**
- **Gains**
  - **Couplage plus faible**
  - **Sélection plus fine des fonctionnalités souhaitées**

# Delegation : un exemple classique…

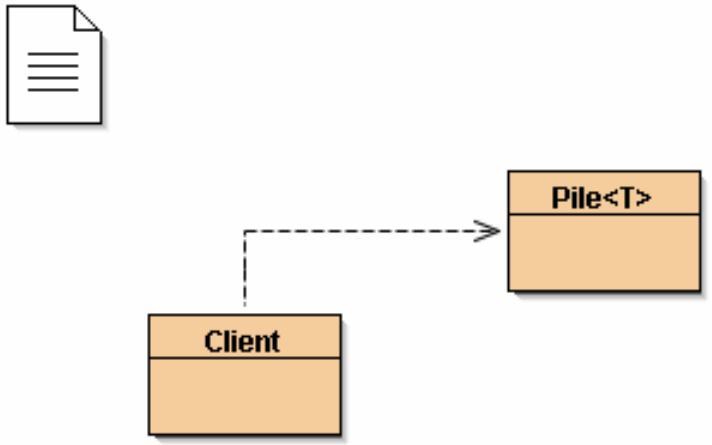
```
import java.util.Stack;
public class Pile<T>{
  private final Stack<T> stk;
  public Pile(){
   stk = new Stack<T>();
  }
  public void empiler(T t){
    stk.push(t);
   }
 …}
```




```
public class Client{
  public void main(String[] arg){
    Pile<Integer> p = new Pile<Integer>();
    p.empiler(4);
   …
  }
}
```

# Delegation : souplesse …, Client inchangé

```
import java.util.List;
import java.util.LinkedList;
public class Pile<T>{
  private final List<T> stk;
  public Pile(){
   stk = new LinkedList<T>();
  }
  public void empiler(T t){
    stk.addLast(t);
   }
 …}
```




```
public class Client{
  public void main(String[] arg){
    Pile<Integer> p = new Pile<Integer>();
    p.empiler(4);
   …
  }}
```

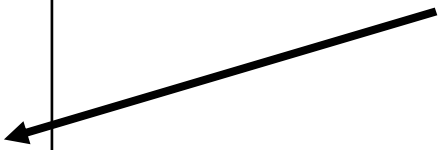
# Délégation / Héritage

- **Discussion…**

# Délégation : une critique tout de même

```
public class Pile<T>{

  private final List<T> stk;

  public Pile(){
   stk = new LinkedList<T>();
  }

 …}
```

L'utilisateur
n'a pas le choix de
l'implémentation de la Liste .



# Interface

- **La liste des méthodes à respecter**
  - **Les méthodes qu'une classe devra implémenter**
  - **Plusieurs classes peuvent implémenter une même interface**
  - **Le client choisira en fonction de ses besoins**
  - **Exemple**
    - **Collection<T>  est une interface**
    - **ArrayList<T>, LinkedList<T> sont des implémentations de Collection<T>**
    - **Iterable<T>  est une interface**
    - **L'interface Collection « extends » cette interface et propose  la méthode**
      - **public Iterator<T> iterator();**

# Interface : java.util.Iterator<E>

```
interface Iterator<E>{
    E next();
    boolean hasNext();
    void remove();
}
```

**Exemple :**

**Afficher le contenu d'une Collection<E> nommée *collection***

```
Iterator<E>  it = collection.iterator();
while( it.hasNext()){
  System.out.println(it.next());
}
```

# Iterable<T> et foreach

- **Parcours d ’une Collection c**
  - **exemple une Collection<Integer> c = new ….;**
    - for( Integer i : c)
    - System.out.println(" i = " + i);

    <==>

    - for(Iterator<Integer> it = c.iterator(); it.hasNext();)
    - System.out.println(" i = " + it.next());

    - **syntaxe**
      *for( element e : collection*)*

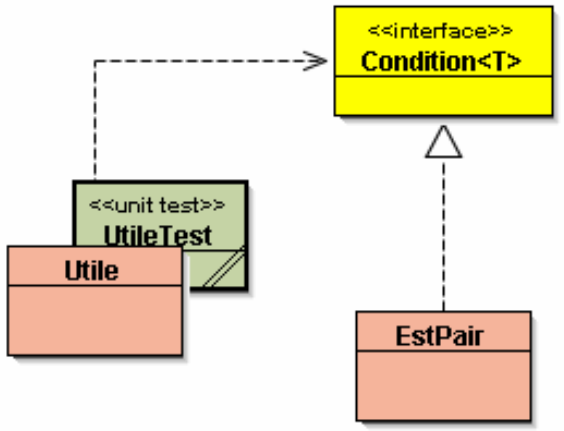
*Collection : une classe avec un iterator,* *(ou un tableau…voir les ajouts en 1.5)*
*La boucle for s’emploie sur toute implémentation de l’interface Iterable et les tableaux*

# Interface : un exemple

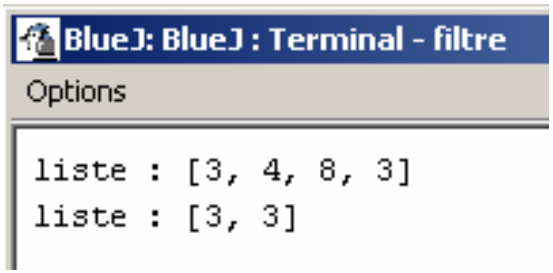
```
public static
<T> void filtrer( Iterable<T> iterable,
                  Condition<T> condition){

    Iterator<T> it = iterable.iterator();
    while (it.hasNext()) {
       T t = it.next();
       if (condition.isTrue(t)) {
          it.remove();
       }
    }
  }

 public interface Condition<T>{
    public boolean isTrue(T t);
  }
```

# Exemple suite

- **Usage de la méthode filtrer**
  - **retrait de tous les nombres pairs d'une liste d'entiers**

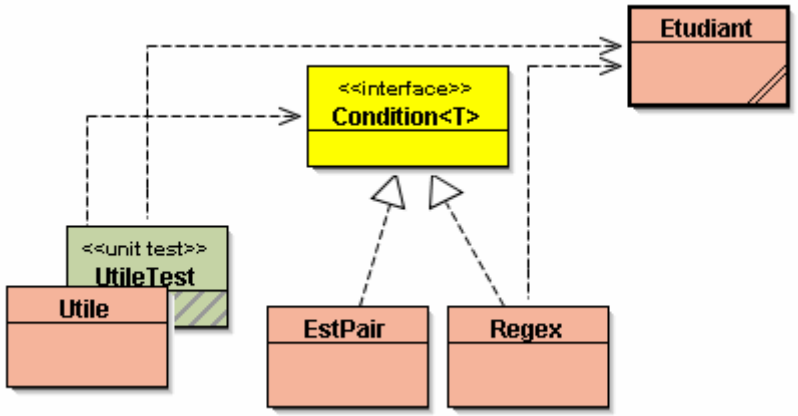



```
Collection<Integer> liste = new ArrayList<Integer>();
liste.add(3);liste.add(4);liste.add(8);liste.add(3);
System.out.println("liste : " + liste);

Utile.filtrer(liste,new EstPair());
System.out.println("liste : " + liste);
```



# Exemple suite bis

- **Usage de la méthode filtrer**
  - **retrait de tous les étudiants à l'aide d'une expression régulière**




```
Collection<Etudiant> set = new HashSet<Etudiant>();
set.add(new Etudiant("paul"));
set.add(new Etudiant("pierre"));
set.add(new Etudiant("juan"));
System.out.println("set : " + set);

Utile.filtrer(set,new Regex("p[a-z]+"));
System.out.println("set : " + set);
```

```
BlueJ: BlueJ : Terminal - filtre
Options
set : [juan, paul, pierre]
set : [juan]
```

discussion

# Delegation + Interface

- **Délégation**
  - **Usage en interne d'une classe existante : la délégation**
  - **La délégation peut-être changée, y compris dynamiquement, sans que le client s'en aperçoive**
- **Couplage encore plus faible**
  - **En laissant le choix de la classe de délégation au Client**
  - **Tout en garantissant les compatibilités ultérieures**
  - **Mise en Pratique : La Pile …**
  - **Que  l'on nomme injection de dépendance …**

# Délégation + interface = souplesse …

```
public class Pile<T>{

   private final List<T> stk;

  public Pile(){
   stk = new LinkedList<T>();
  }
 …}
```

L'utilisateur **n'a pas le choix** de l'implémentation …

```
public class Pile<T>{
   private final List<T> stk;

  public Pile(List<T> l){
   stk = l;
  }
  public Pile(){
   stk = new LinkedList<T>();
  }
 …}
```

Ici l'utilisateur a le choix de l'implémentation de la Liste …

# Delegation + interface = souplesse ?

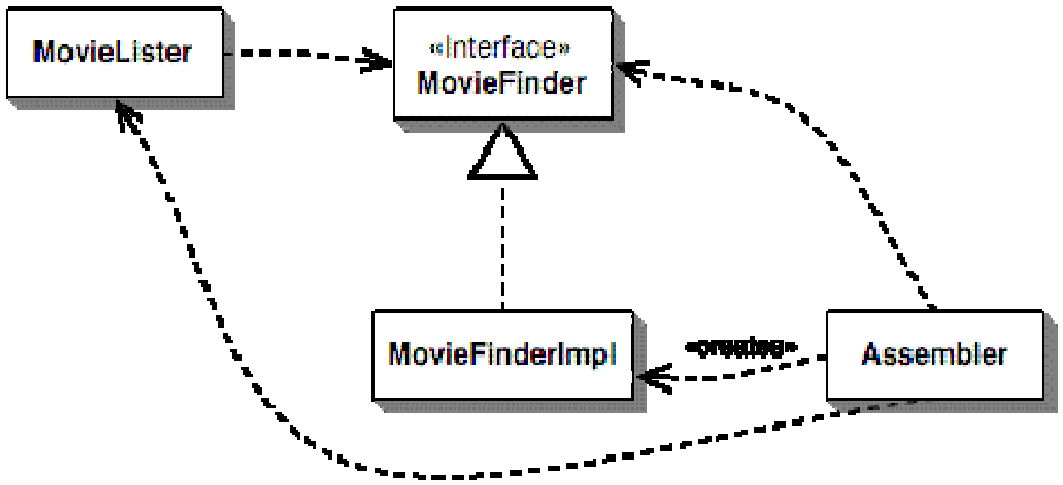
```
public class Client{
  public void main(String[] arg){
    Pile<Integer> p;
    p = new Pile<Integer>(new LinkedList<Integer>());
    p.empiler(4);
   …
  }}
```

- discussion

# Injection de dépendance

- **Délégation + interface = injection de dépendance**

- **Voir Martin Fowler**
  - **« Inversion of Control Containers and the Dependency Injection pattern »**
  - **http://martinfowler.com/articles/injection.html**




  - **L'injection de dépendance est effectuée à la création de la pile …**
  - **Voir le paragraphe « Forms of Dependency Injection »**
  - **Inversion de contrôle  utilisé par les canevas/framework**

# Injection de dépendance, un outil possible

- **La classe à injecter est décrite dans un fichier**
  - **Un fichier texte en XML par exemple**
- <injections>
- <injection classe= "Pile" injection= "java.util.LinkedList" />
- <injection …. />
-
- </injections>

- **Un outil pourrait se charger de**
  - **Lire le fichier XML et interpréter la balise** <injection
  - **Créer une instance de la classe à injecter :** java.util.LinkedList
  - **Créer une instance de la classe** Pile
  - **D'exécuter le constructeur de la classe** Pile

  - **Cet outil permettrait alors une séparation configuration / Implémentation**

- **À la recherche du couplage faible**
  - **Ici induite par l'usage d'interface …**

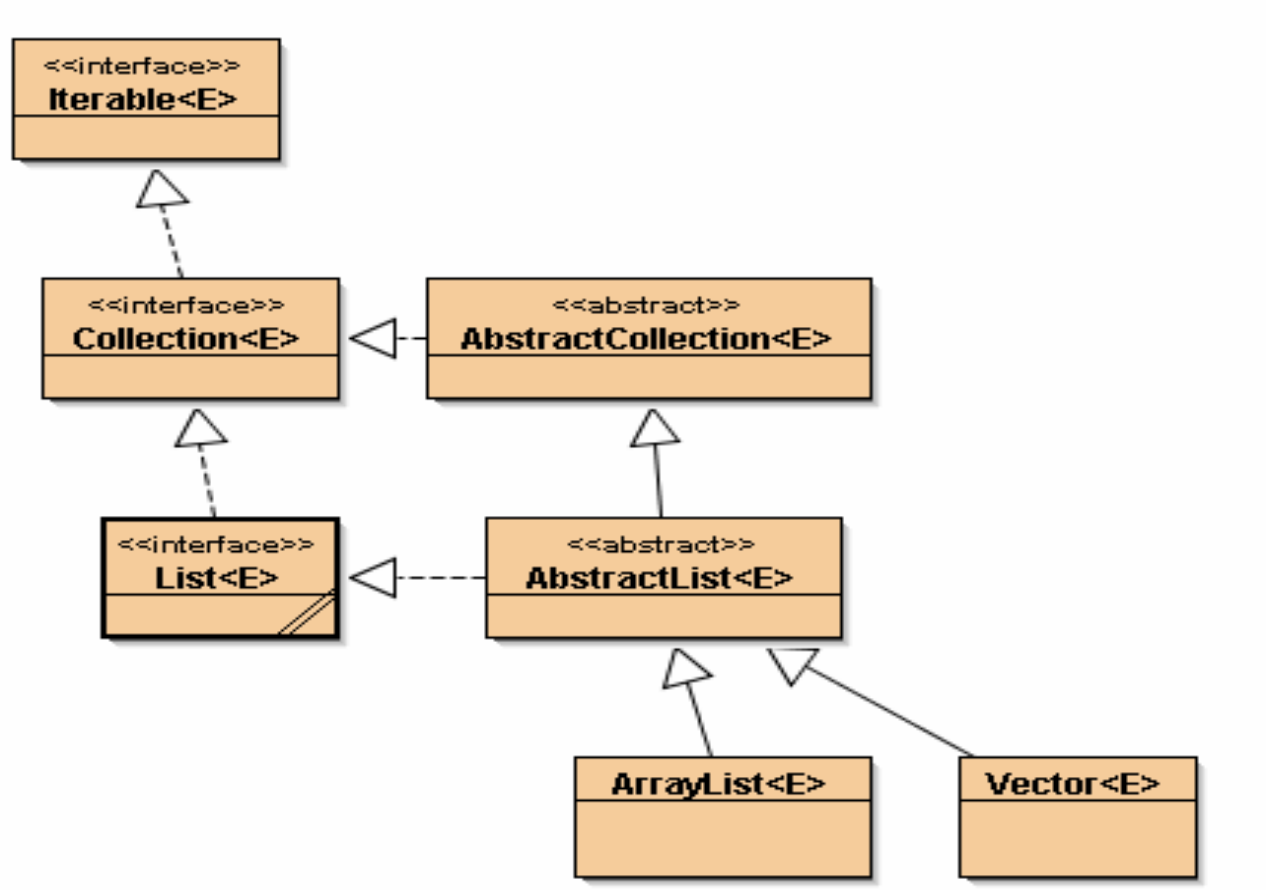
# Abstract superclass

- **Construction fréquemment  associée à l'Interface**
  - **Une classe propose une implémentation incomplète**
    - **abstract class en Java**
  - **Apporte une garantie du « bon fonctionnement » pour ses sous-classes**
  - **Une sous-classe doit être proposée**
  - **Souvent liée à l'implémentation d'une interface**
  - **Exemple extrait de java.util :**
    - **abstractCollection<T>  propose 13 méthodes sur 15**

    **et implémente Collection<T> …**

# Abstract superclass exemple

– **java.util.Collection un extrait**

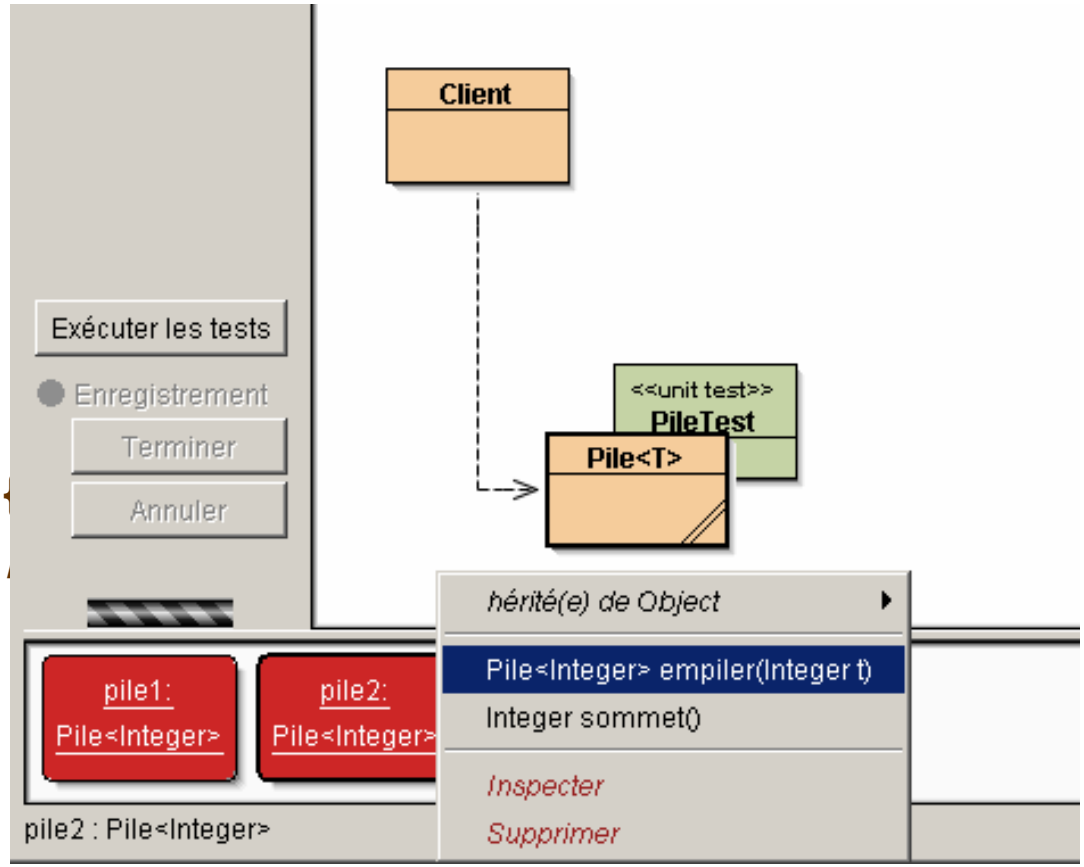


# Immutable

- **La classe, ses instances ne peuvent changer d'état**
  - **Une modification engendre une nouvelle instance de la classe**
- **Robustesse attendue**
- **Partage de ressource facilitée**
  - **Exclusion mutuelle n'est pas nécessaire**
- **Voir java.lang.String / java.lang.StringBuffer**

# Immutable : exemple

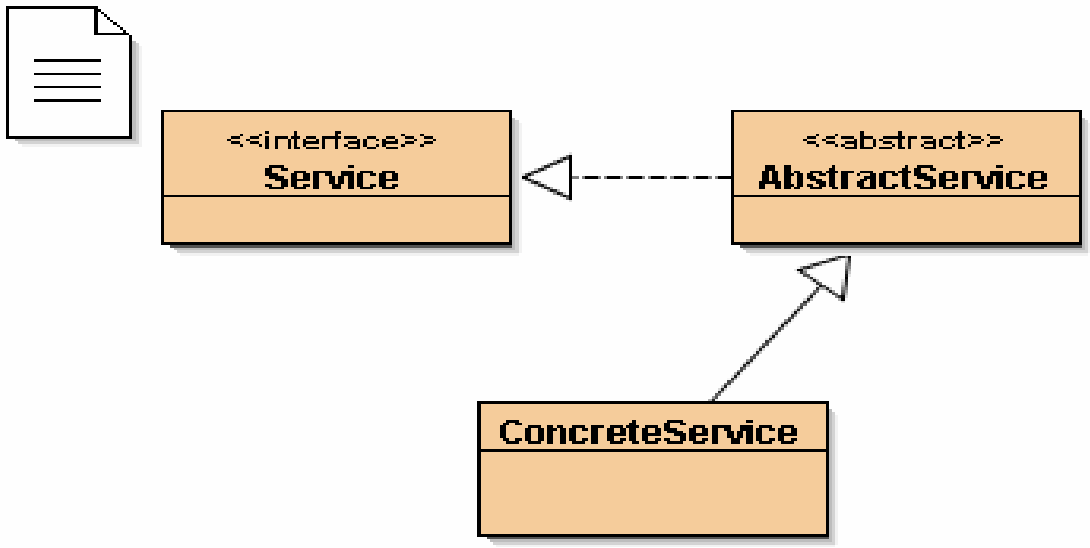
```
public class Pile<T>{

  private final Stack<T> stk;

  public Pile(){
   stk = new Stack<T>();
  }

  public Pile<T> empiler(T t){
    Pile<T> p = new Pile<T>();
    p.stk.addAll(this.stk);
    p.stk.push(t);
    return p;
  }

  public T sommet(){
    return stk.peek();
   }
 …
}
```



# Marker Interface

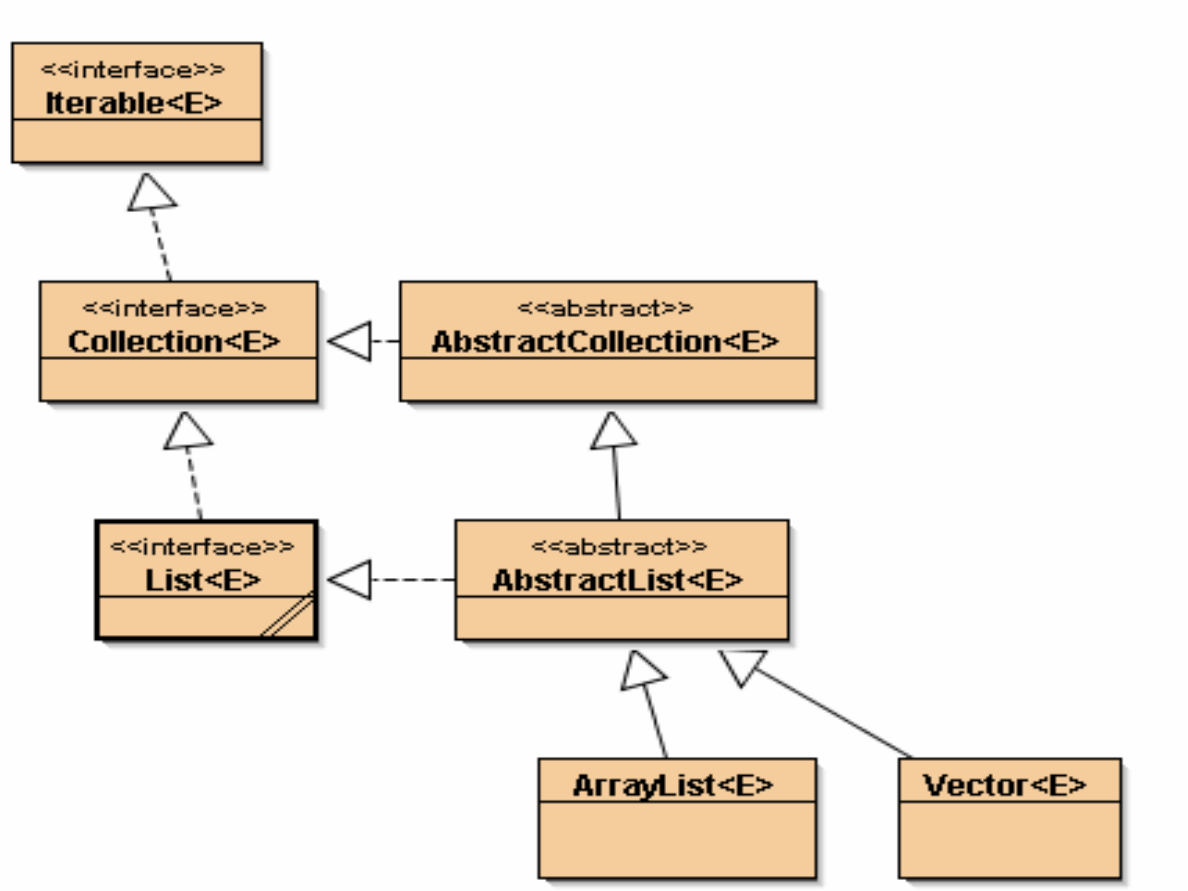
- **Une interface vide !**
  - **Classification fine des objets**
  - **Une fonctionnalité attendue d'une classe**
  - **Exemples célèbres**
    - **java.io.Serializable, java.io.Cloneable**
      - **Lors de l'usage d'une méthode particulière une exception sera levée si cette instance n'est pas du bon « type »**

# Interface & abstract




- **Avantages cumulés !**
  - **Collection<T>  interface**
  - **AbstractCollection<T>**
  - **ArrayList<T>**

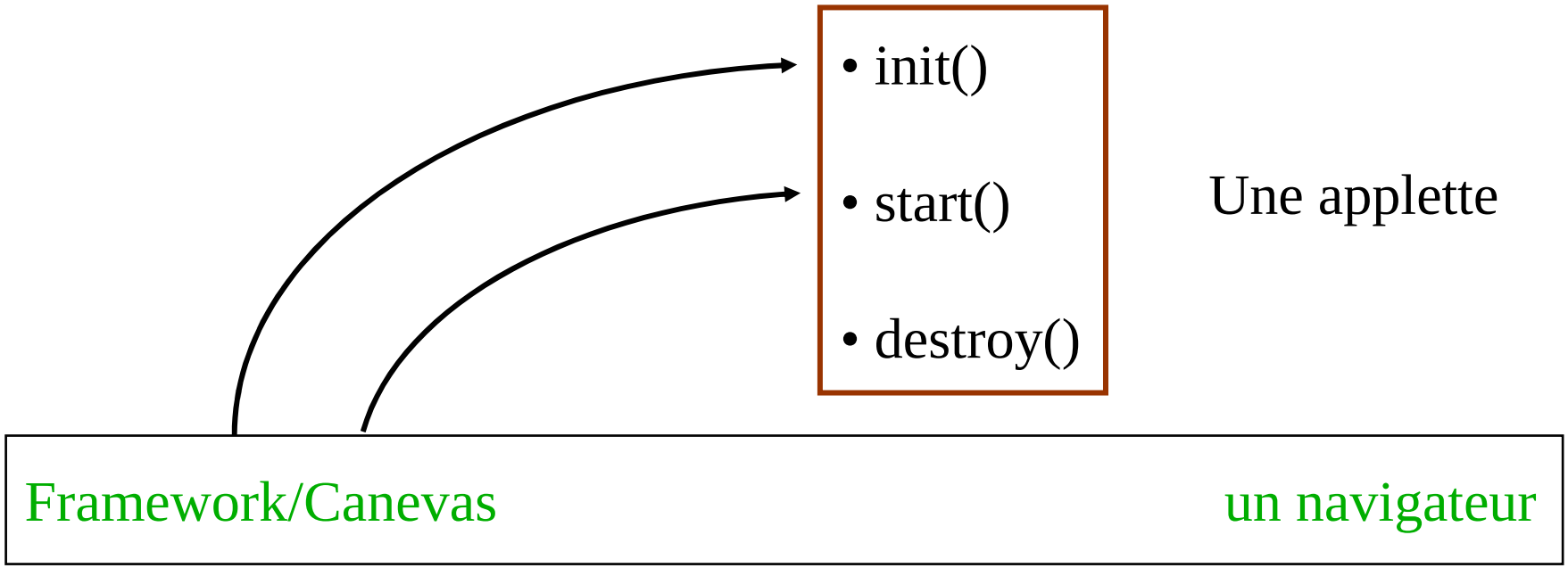
# Interface & abstract




– **Déjà vu …**

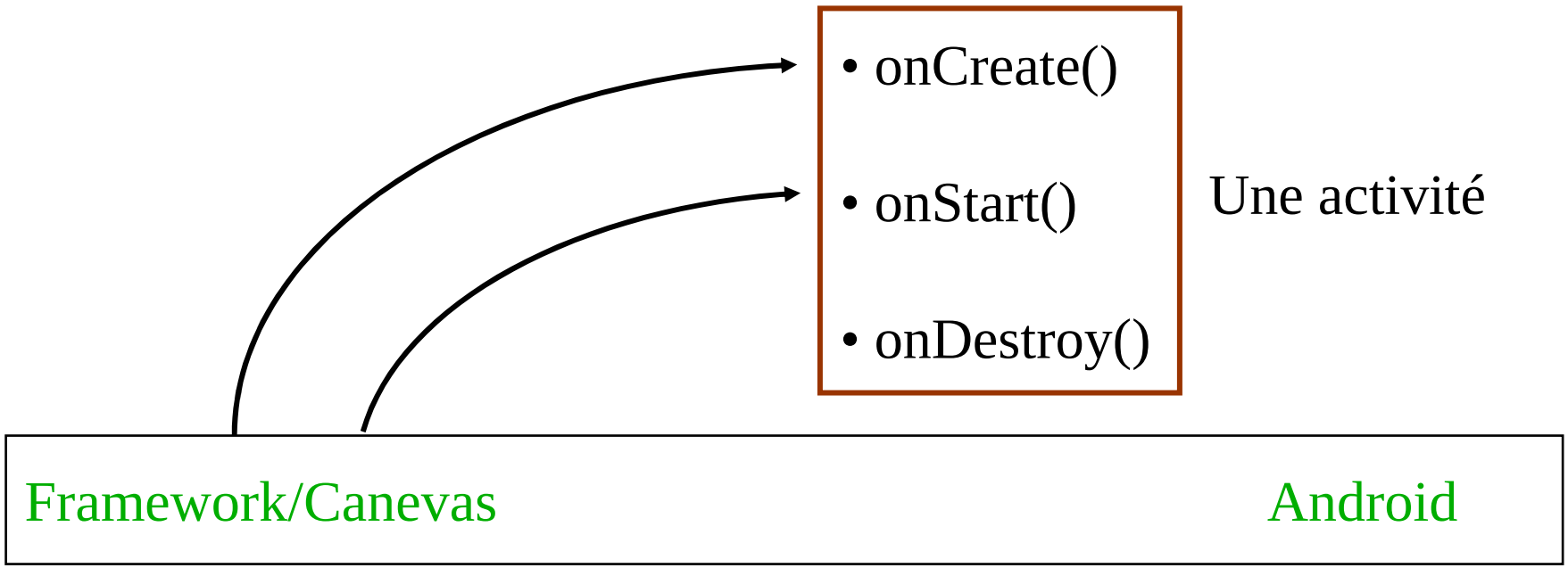
# Vocabulaire

- **Encore Martin Fowler**
  - **Injection de dépendance**
  - **Inversion de contrôle ?**



# Vocabulaire

- **Encore Martin Fowler**
  - **Injection de dépendance**
  - **Inversion de contrôle**



# Inversion de contrôle/Injection de dépendance

- **« Baies d'accueil » de programme,**
  - **Framework, canevas**
  - **Navigateur et Applet**
  - **Eclipse et Plug-in**
  - **JVM et classes**
  - **Serveur Web et servlets**
  - **Android et applications**
  - **…**
- **À lire : Martin Fowler**
  - **« Inversion of Control Containers and the Dependency Injection pattern »**
  - **http://martinfowler.com/articles/injection.html**

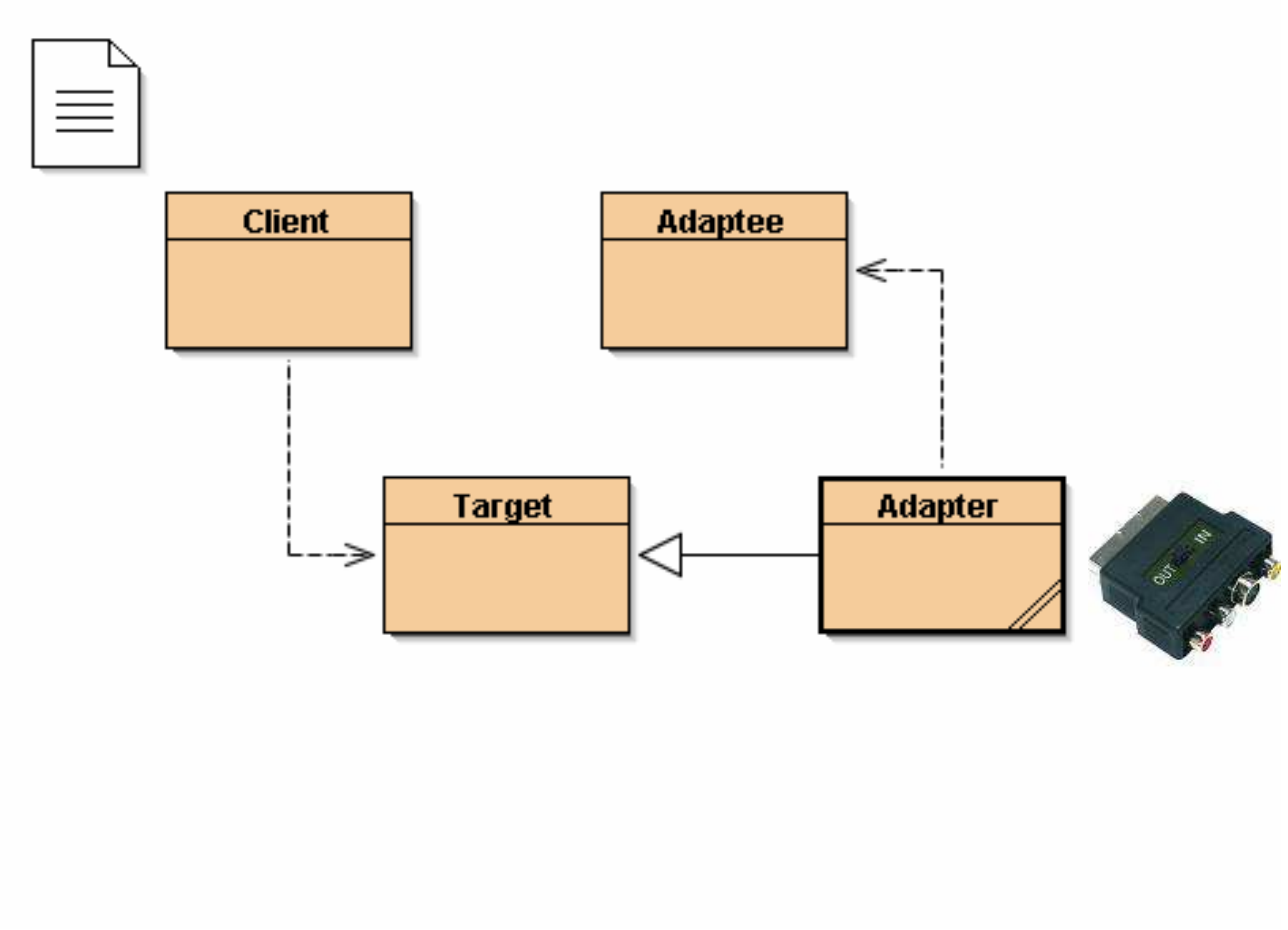
# Les patrons de base, les omni-présents…

- **Adapter**
  - Adapte l'interface d'une classe conforme aux souhaits du client
- **Proxy**
  - Fournit un mandataire au client afin de contrôler/vérifier ses accès
- **Template Method**
  - Définit une partie de l'algorithme complété dans les sous classes
- **Composite**
  - Structuration des objets en arborescence
- **Observer**
  - Notification d'un changement d'état d'une classe aux observateurs inscrits

# Adaptateurs


- **Adaptateurs**
  - **prise US/ adaptateur / prise EU**
  - **Client RCA / adaptateur / Prise Péritel**

# Pattern Adapter [DP05]

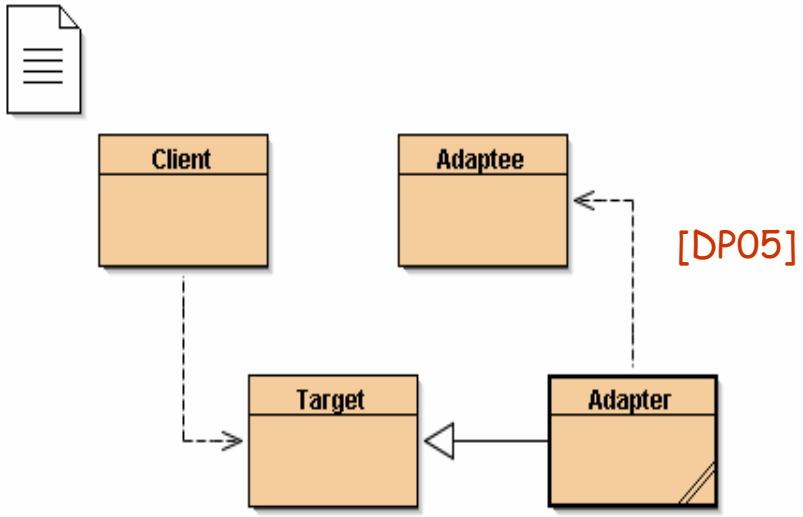


# Adaptateur de prise …

```
public interface Prise {
   public void péritel();
}

public class Adapté {
   public void RadioCorporationAmerica(){…}
}

public class Adaptateur implements Prise {
   public Adapté adapté;
   public Adaptateur(Adapté adapté){
       this.adapté = adapté;
   }

   public void péritel(){
     adapté.RadioCorporationAmerica();
   }
}
```

# Pattern Adapter

```
public interface Target {
   public void serviceA();
}

public class Adaptee {
   public void serviceB(){…}
}

public class Adapter implements Target
   public Adaptee adaptee;
   public Adapter(Adaptee adaptee){
       this.adaptee = adaptee;
   }

   public void serviceA(){
     adaptee.serviceB();
   }

}
```



# Adapter et classe interne java

- **Souvent employé …**

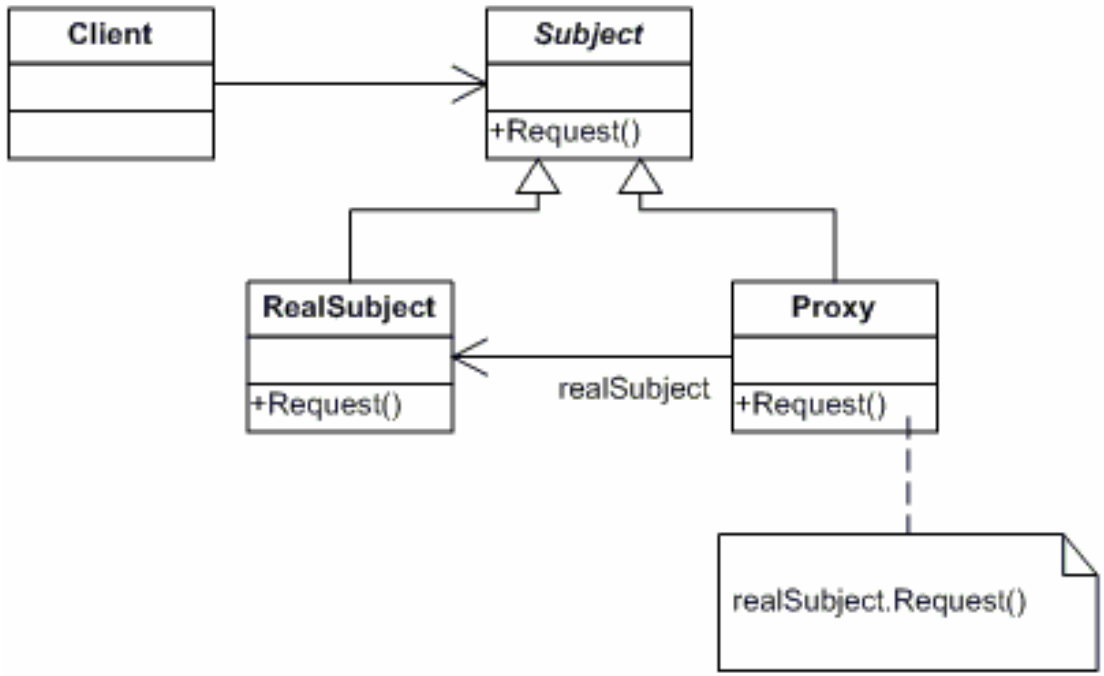
```
public Target newAdapter(final Adaptee adaptee){
    return
      new Target(){
        public void serviceA(){
          adaptee.serviceB();
        }
      };
    }
```

- **Un classique …**

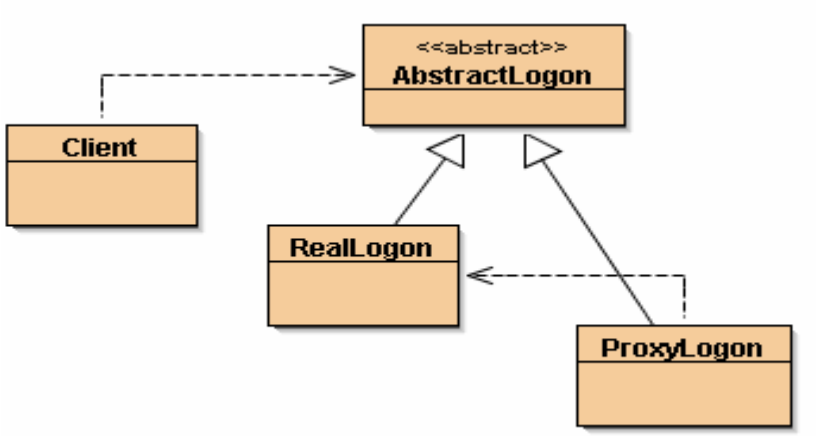
```
WindowListener w = new WindowAdapter(){
        public void windowClosing(WindowEvent e) {
          System.exit(0);
         }
       };
```

# Proxy

- **Fournit un mandataire au client afin de**
  - **Contrôler/vérifier les accès**

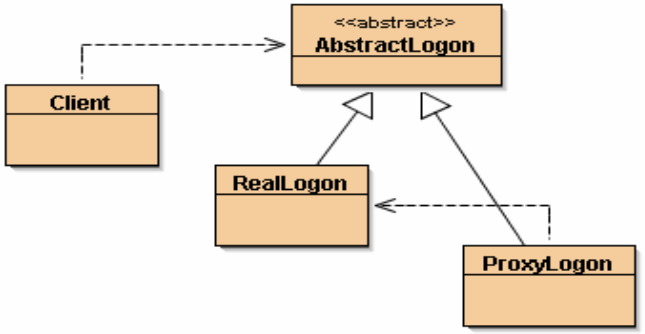


# Proxy : un exemple




```
public abstract class AbstractLogon{
   abstract public boolean authenticate( String user, String password);
}

public class Client{
 public static void main(String[] args){
    AbstractLogon logon = new ProxyLogon();
    …
   }
}
```

# Proxy : exemple suite

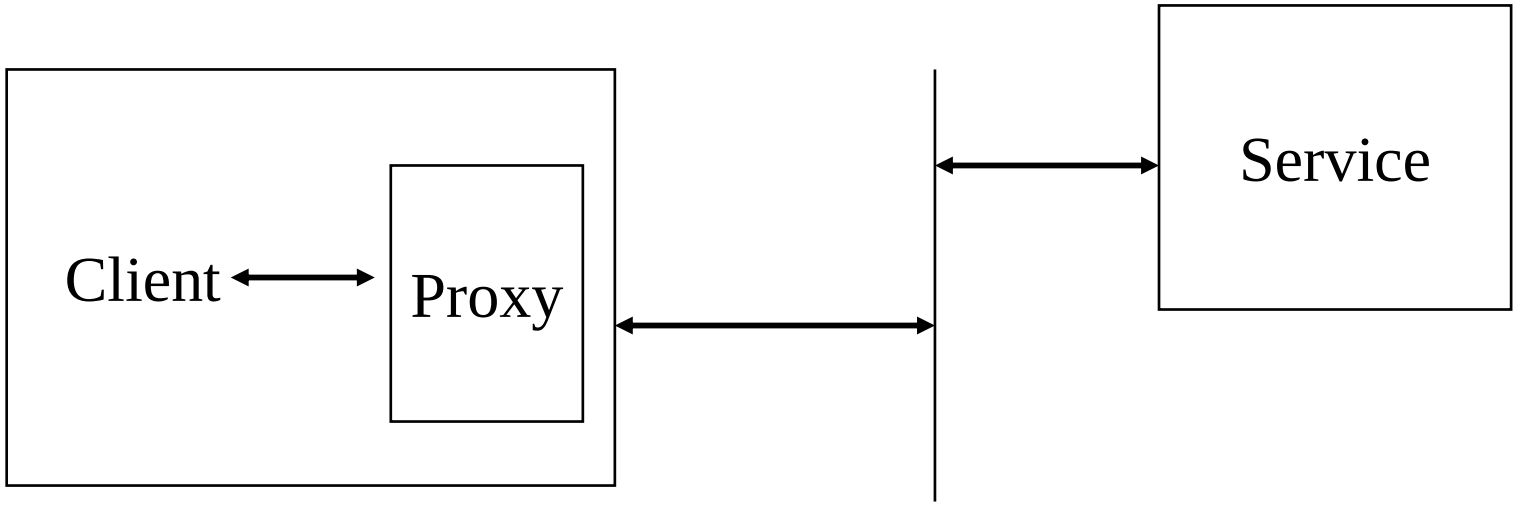



```
public class ProxyLogon extends AbstractLogon{
  private AbstractLogon real = new RealLogon();

  public boolean authenticate(String user, String password){
    if(user.equals("root") && password.equals("java"))
      return real.authenticate(user, password);
    else
      return false;
  }
}
```

```
public class RealLogon extends AbstractLogon{
   public boolean authenticate(String user, String password){
      return true;
  }
}
```

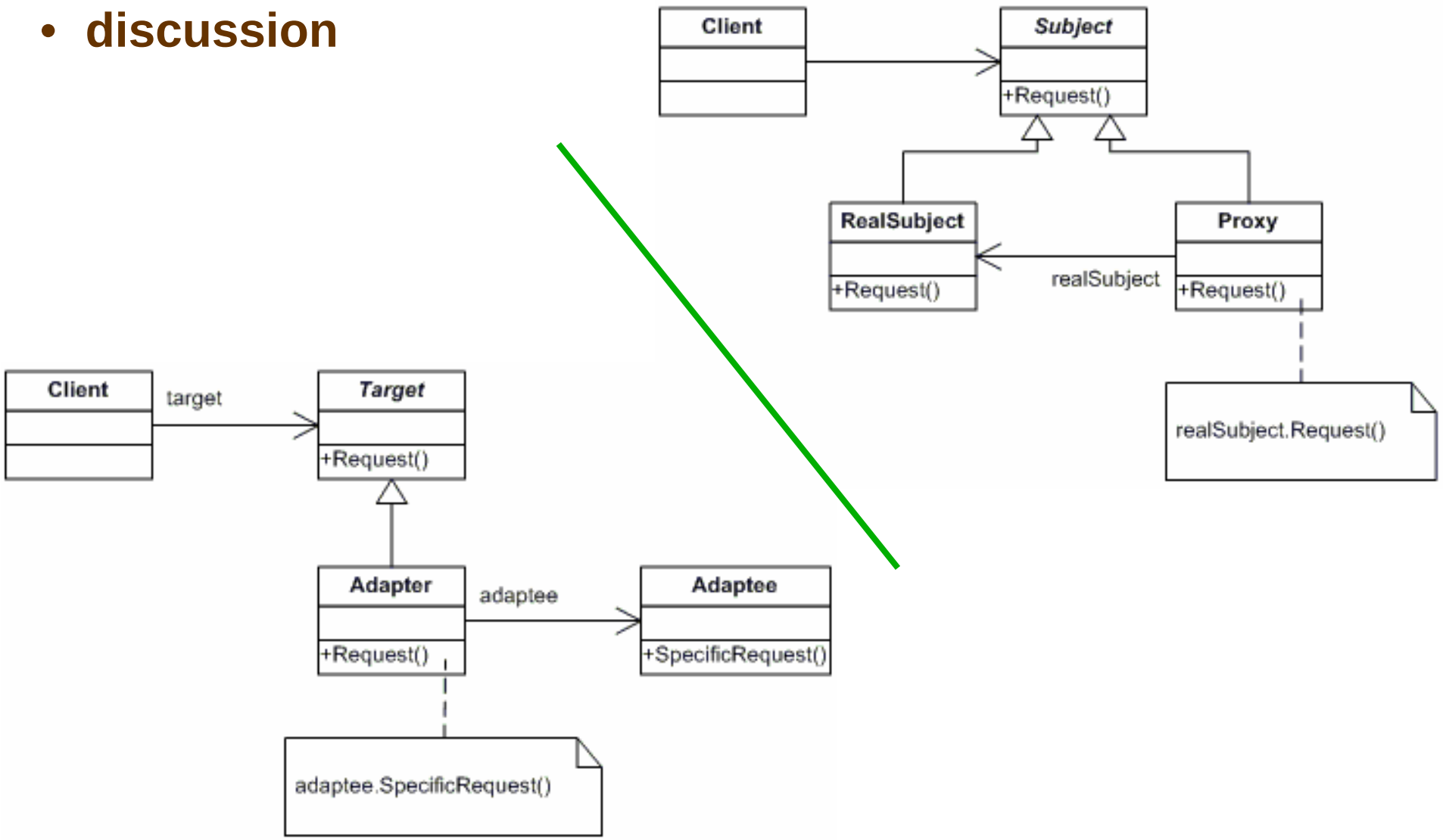
# Proxy et RMI

- **Abstraire la communication**
  - **Par un proxy/mandataire pour le client**
  - **Transmettre les valeurs des objets**
    - **Sérialisation en java**
  - **Recevoir les résultats ou Exceptions**



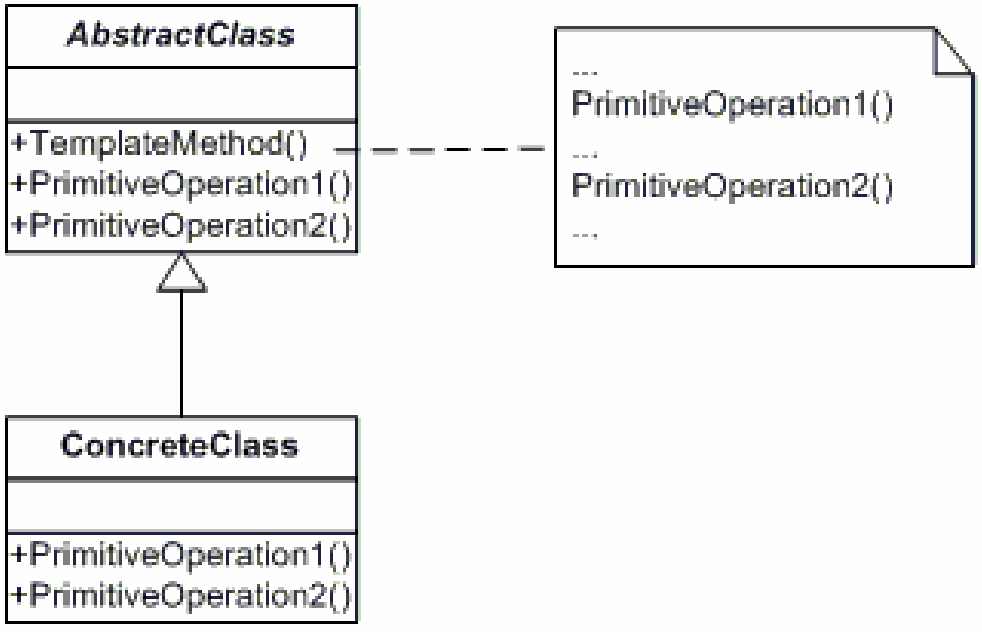
# Adapter\Proxy

- **discussion**

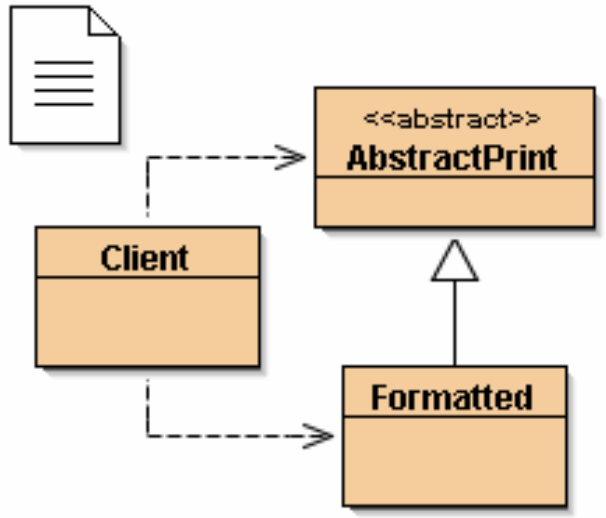


# Template Method

- **Laisser à la responsabilité des sous-classes**
  - **La réalisation d'une partie de l'algorithme**

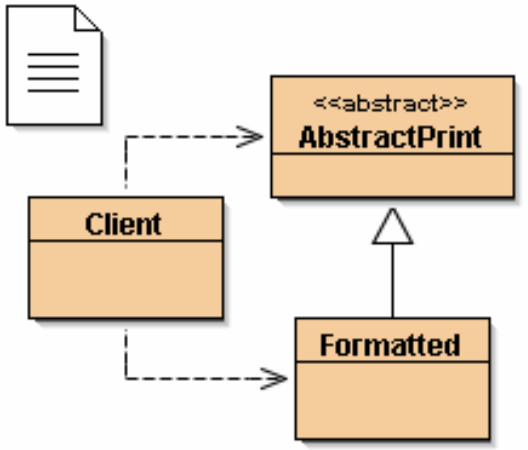


# TemplateMethod : exemple




```
public abstract class AbstractPrint{
   public void print(String str){
    setUp();
     System.out.print(str);
    tearDown();
     System.out.println();
    }

    public abstract void setUp();  // noms empruntés à junit …
    public abstract void tearDown();
}
```

# Template Method




```
public class Formatted extends AbstractPrint{
    public void setUp(){System.out.print("<PRE>");}
    public void tearDown(){System.out.print("</PRE>");}
}
public class Client{
  public static void main(String[] args){
    AbstractPrint out = new Formatted();
    out.print("un texte");
  }
}
```

<PRE>un texte</PRE>

# Template Method atomique

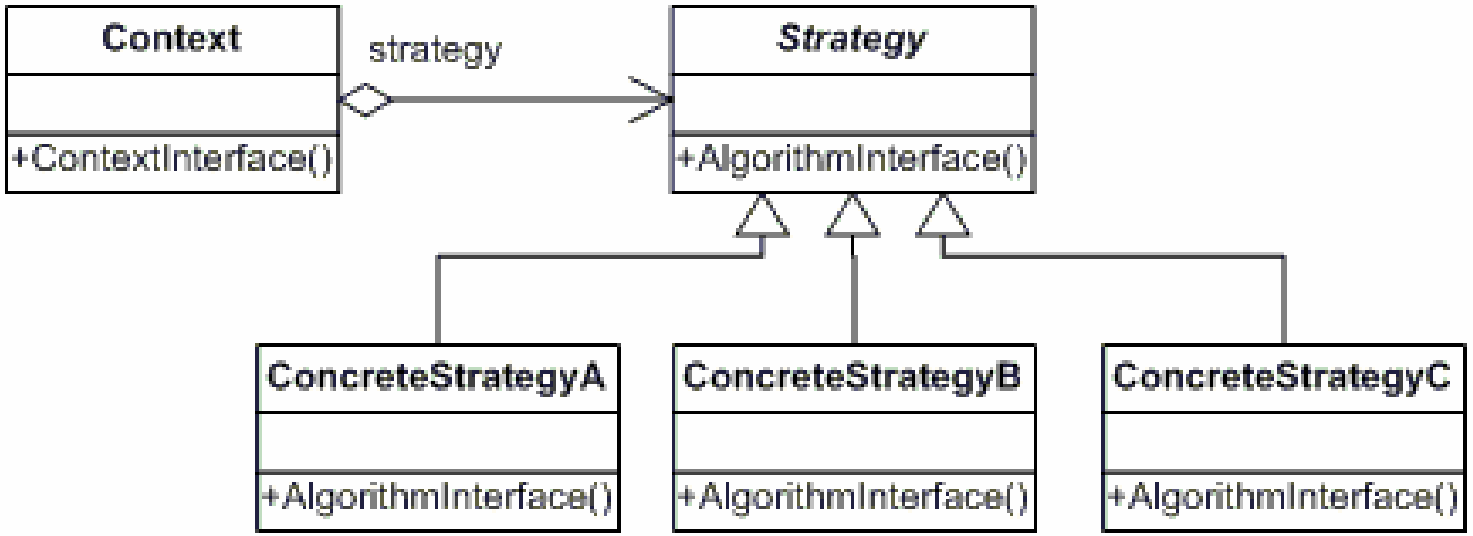
```
public abstract class Transaction{

   public void operation(){
    beginTransaction();
      performOperation();
    endTransaction();
   }

    public abstract void performOperation();

}
```

# Template Method et Transaction

```
public abstract class Transaction{

   public void operation(){
      try{
        beginTransaction();
        performOperation();
        endTransaction();
      }catch(Exception e){
        rollbackTransaction();
      }
   }

    public abstract void performOperation();
    public abstract void beginTransaction();
    public abstract void endTransaction();
    public abstract void rollbackTransaction();
```

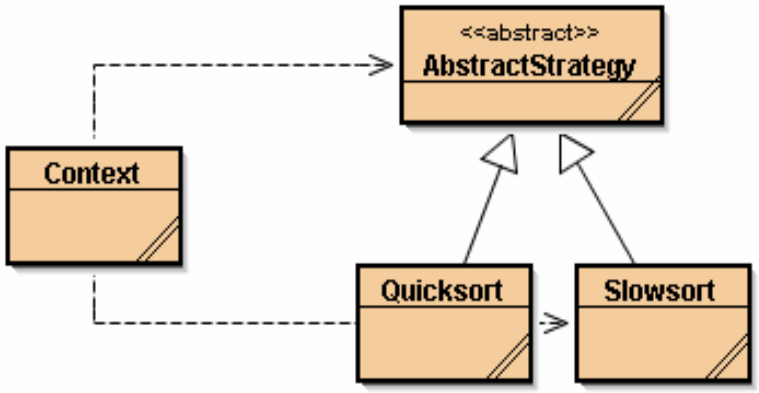
# Le pattern Strategy

- **Une même famille d'algorithme**
  - **Une super-classe**
  - **Des sous-classes interchangeable sans que le client s'en aperçoive**



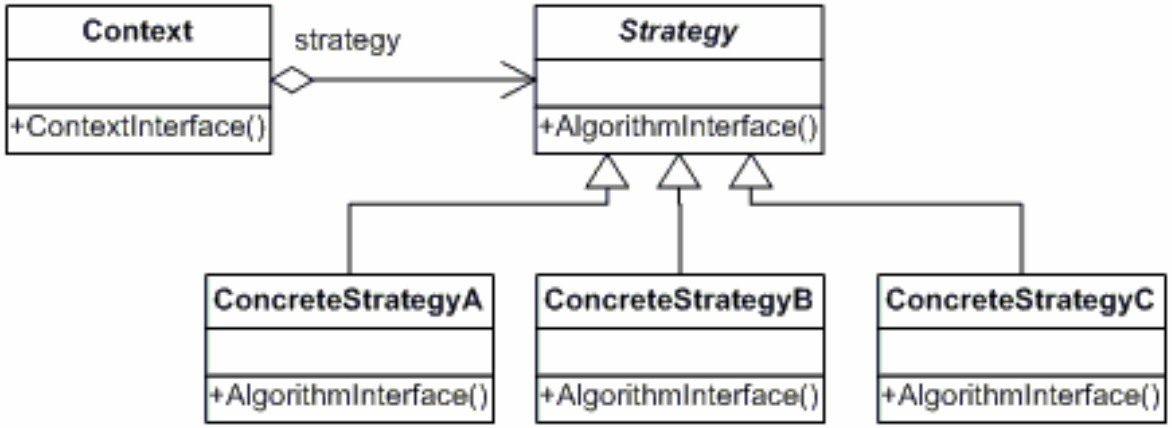
# Le pattern Strategy exemple classique

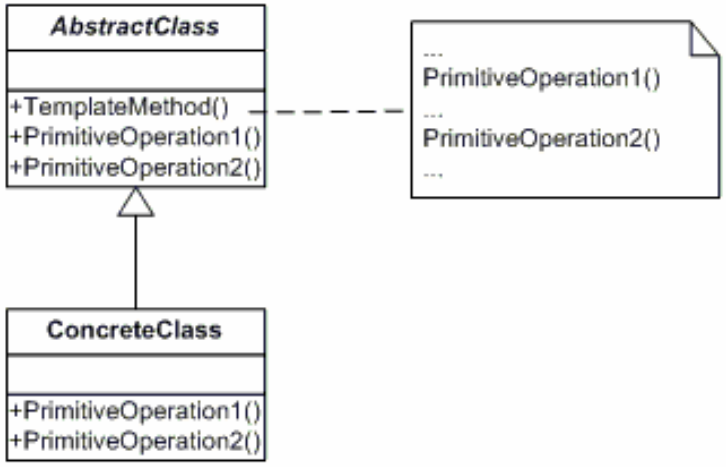
```
public abstract class AbstractStrategy{
   public abstract <T extends Comparable<T>> void sort(T... list);
}

public class Context{
  private AbstractStrategy strategy;
  public <S extends AbstractStrategy> Context(S strategy){
    this.strategy = strategy;
  }
  public void operation(Integer... t){
    strategy.sort(t);
  }


  public static void main(){
    Integer[] t = new Integer[]{3,2,1,6};
    Context ctxt = new Context(new Slowsort());
    // ou bien new Context(new Quicksort());
    ctxt.operation(t);
  }
}
```



# Template Method \ Strategy

- **discussion**



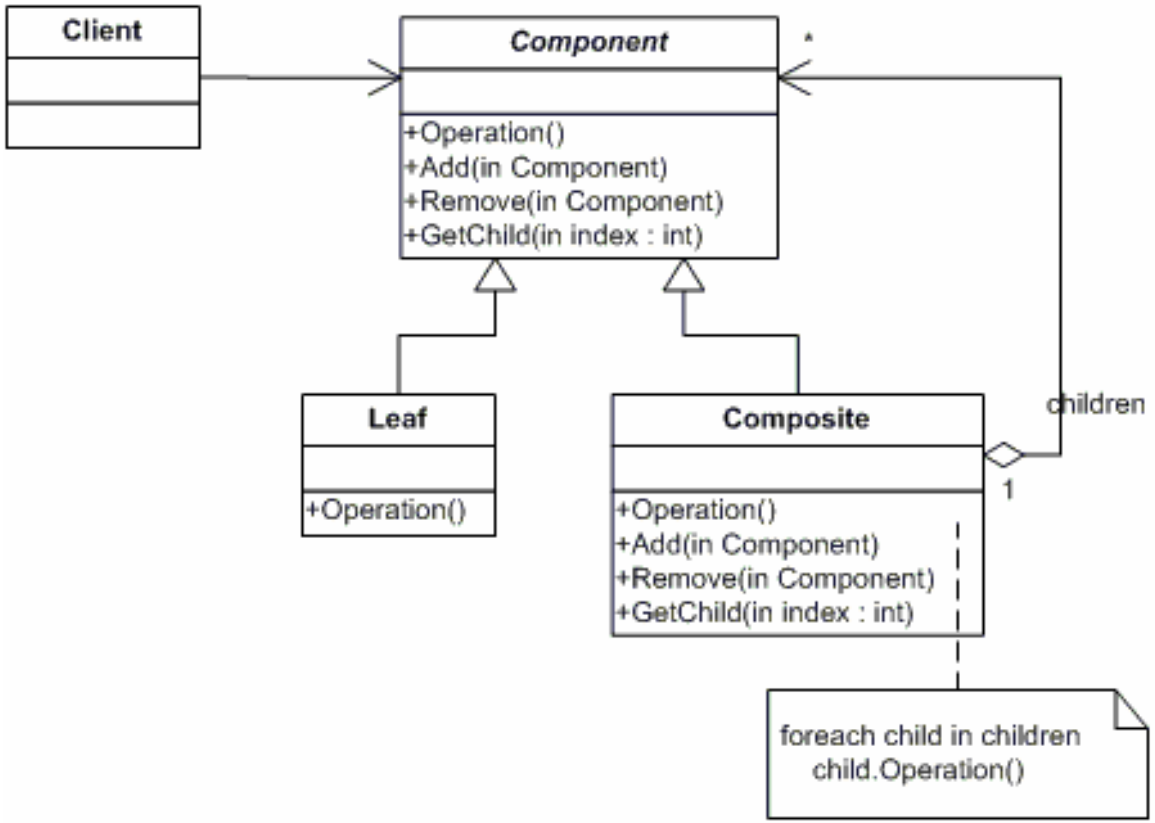




**delegation**

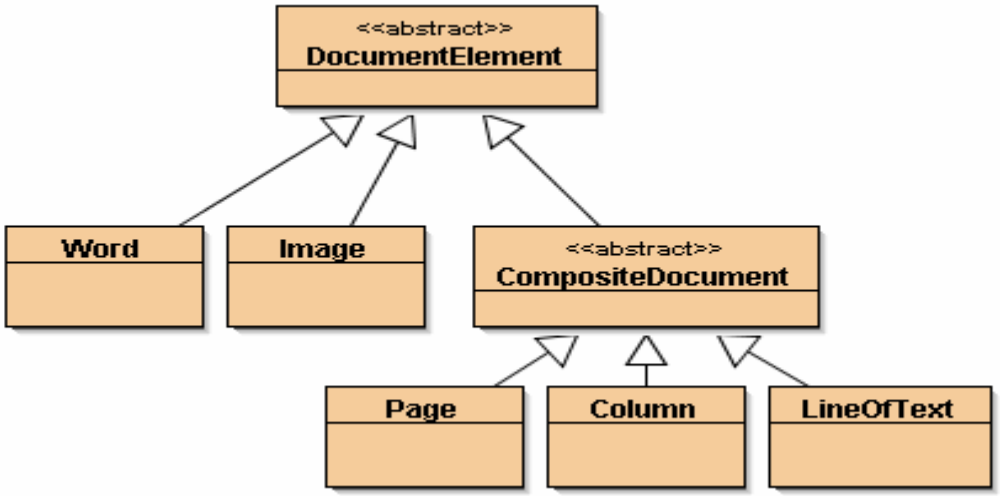
**héritage**

# Composite

- **Structures de données récursives, arborescentes**
  - **Le client utilise des « Component » quelque soit leur complexité**

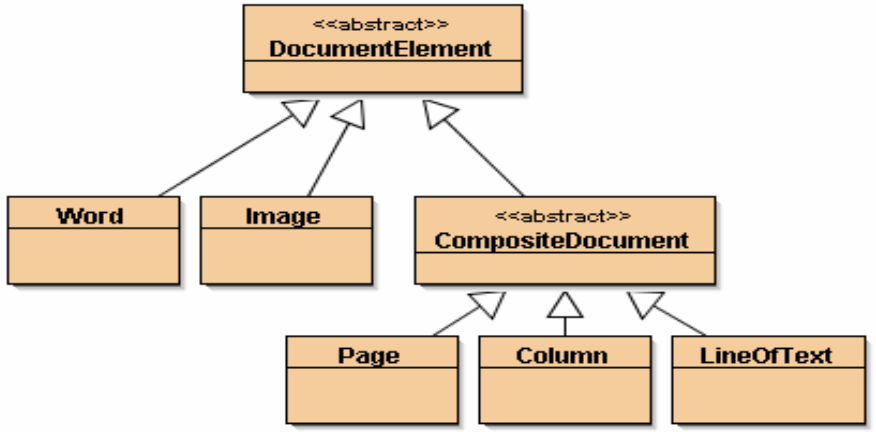


# Composite, un exemple [Grand00]




```
public abstract class DocumentElement{
  protected CompositeDocument parent;
  public CompositeDocument getParent(){
    return parent;
   }
  public abstract int size();
}

public class Word
  extends DocumentElement{
    public int size(){ return 1;}
}
```

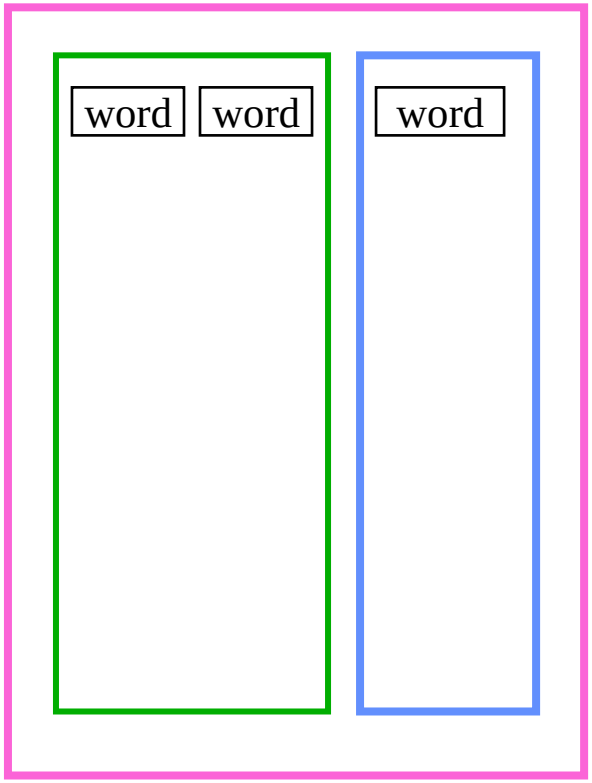
# Composite, suite




```
public abstract class CompositeDocument
  extends DocumentElement{
  private List<DocumentElement> children = …

  public void add(DocumentElement doc){
    children.add(doc);
    doc.parent=this;
  }
  public int size(){
    int sz = 0;
    for( DocumentElement child : children){sz = sz + child.size();}
    return sz;
   }}
```

# Composite, fin

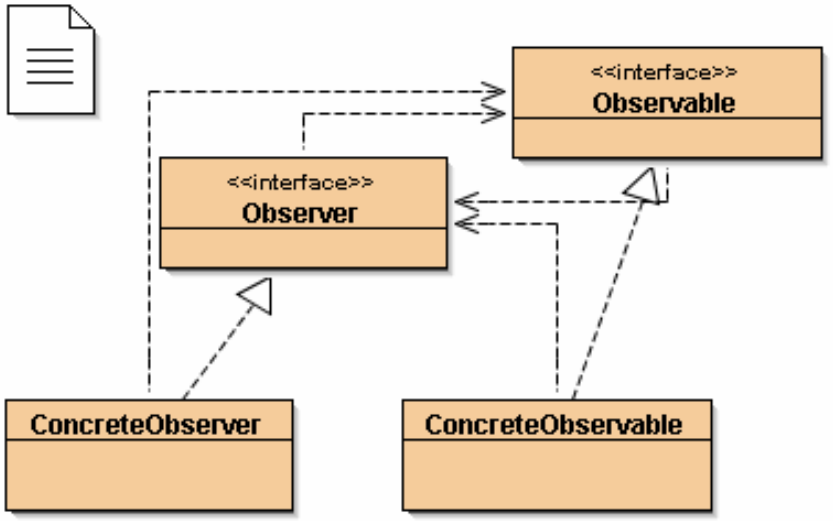
```
CompositeDocument col1 = new Column();
col1.add(new Word());col1.add(new Word());

CompositeDocument col2 = new Column();
col2.add(new Word());

CompositeDocument page = new Page();
page.add(col1);page.add(col2);

System.out.println(page.size());
```



# Observer

– **Lors d'un changement d'état notification aux observateurs inscrits**

```
public interface Observable{

  public void addObserver(Observer o);
  public void removeObserver(Observer o);
  public void notifyObservers();

  public int getState();
  public void setState(int state);
}

public interface Observer{
    public void update(Observable o);
}
```



# ConcreteObservable

```
public class ConcreteObservable implements Observable{
  private Collection<Observer> observers = new ……
  private int state = 0;
  public void addObserver(Observer observer){
    observers.add(observer);
  }
  public void removeObserver(Observer observer){
    observers.remove(observer);
  }

  public void notifyObservers(){
    for(Observer obs : observers)
      obs.update(this);
    }

   public int getState(){return this.state;}

   public void setState(int state){
        this.state = state;
        notifyObservers();
    }
}
```

# Observer : mise en oeuvre

```
Observable o = new ConcreteObservable();
Observer obs1= new ConcreteObserver();

o.addObserver(obs1);

o.setState(3);  // obs1 est réveillé, notifié


Observer obs2= new ConcreteObserver();

o.addObserver(obs2);

o.setState(33);  // obs1 et obs2 sont réveillés, notifiés …
```

# Observer : affinage, EventObject …

– **A chaque notification un « event object » est transmis**

```
public interface Observer{
    public void update(java.util.EventObject evt);
}
```

```
package java.util;
public class EventObject extends Object implements Serializable{
  public EventObject(Object source){ …}

  public Object getSource(){ …}

  public String toString(){ …}

}
```

# Concrete Observer reçoit un « EventObject »

```
// notification persistente … ( EventObject est « Serializable »)

public class ConcreteObserver implements Observer{

  public void update(EventObject event){
    try{
      ObjectOutputStream oos =
              new ObjectOutputStream(
                      new FileOutputStream("event.ser"));

      oos.writeObject(event);
      oos.close();
    }catch(Exception e){
        e.printStackTrace();
    }
  }

}
```

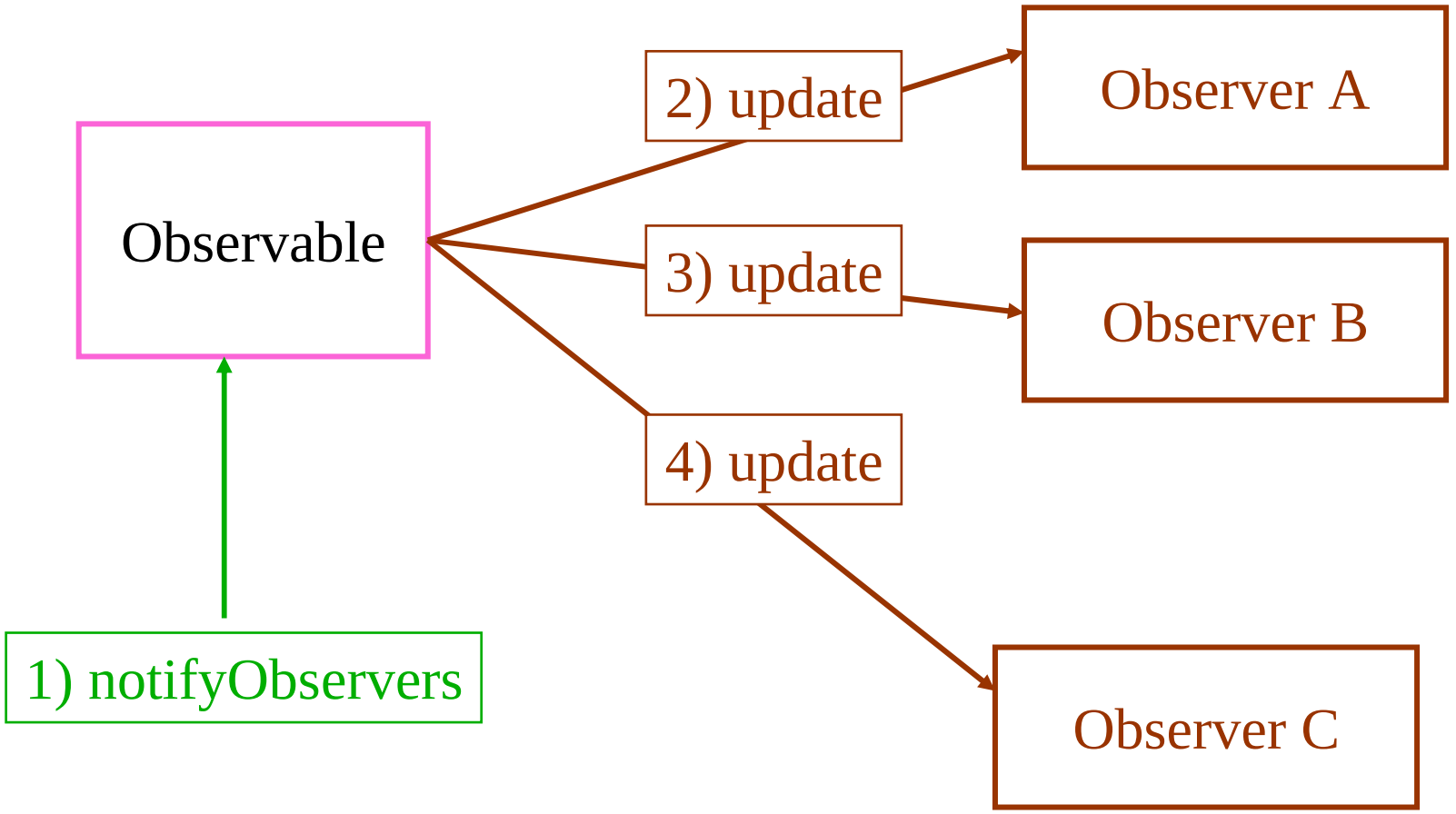
# java.util et plus

- **java.util.Observer** & **java.util.Observable**
  - **update** — **addObserver**
- **java.awt.event.ActionListener**
  - **« update »** — **addActionListener**
- **XXXXXListener**
  - **« update »** — **addXXXXXListener**

# Observer en résumé

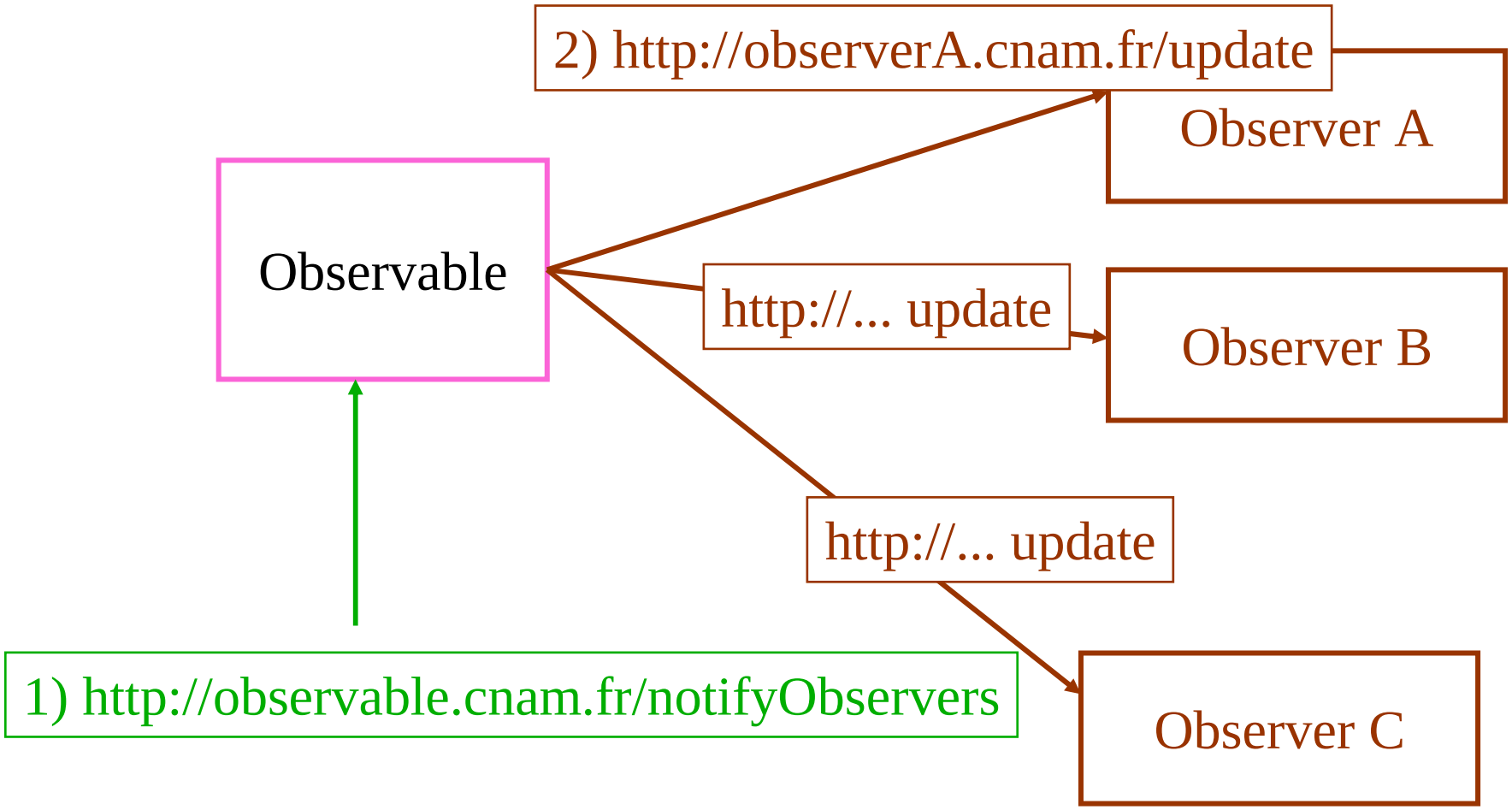
- **Ajout/retrait dynamique des observateurs**
- **L'observable se contente de notifier**
  - **Notification synchrone à tous les observteurs inscrits**
  - **API prédéfinies java.util**
  - **La grande famille des**
    - **« Listener »**
    - **« EventObject »**

# Observer distribué ?

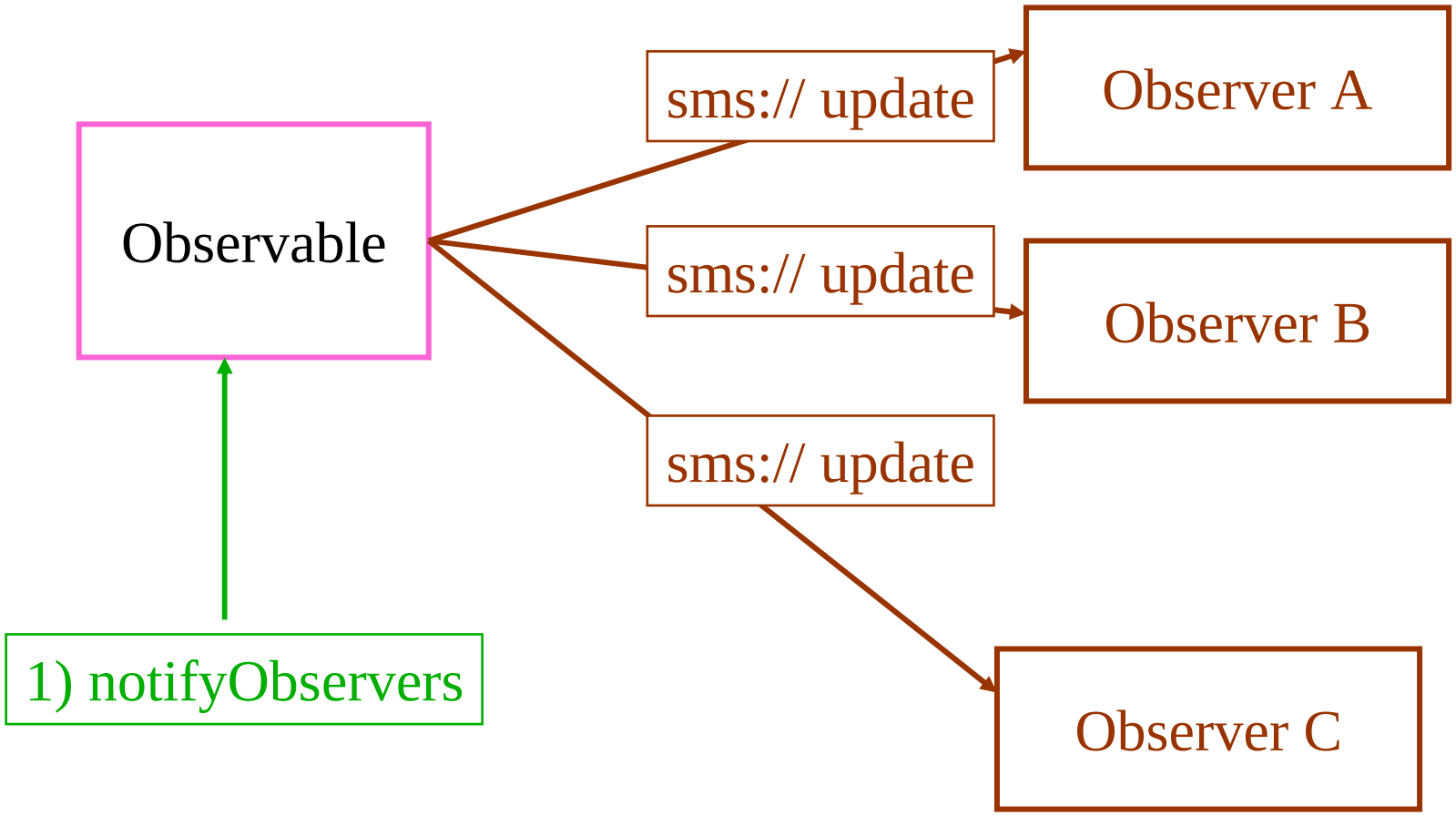



- **Naturellement …**

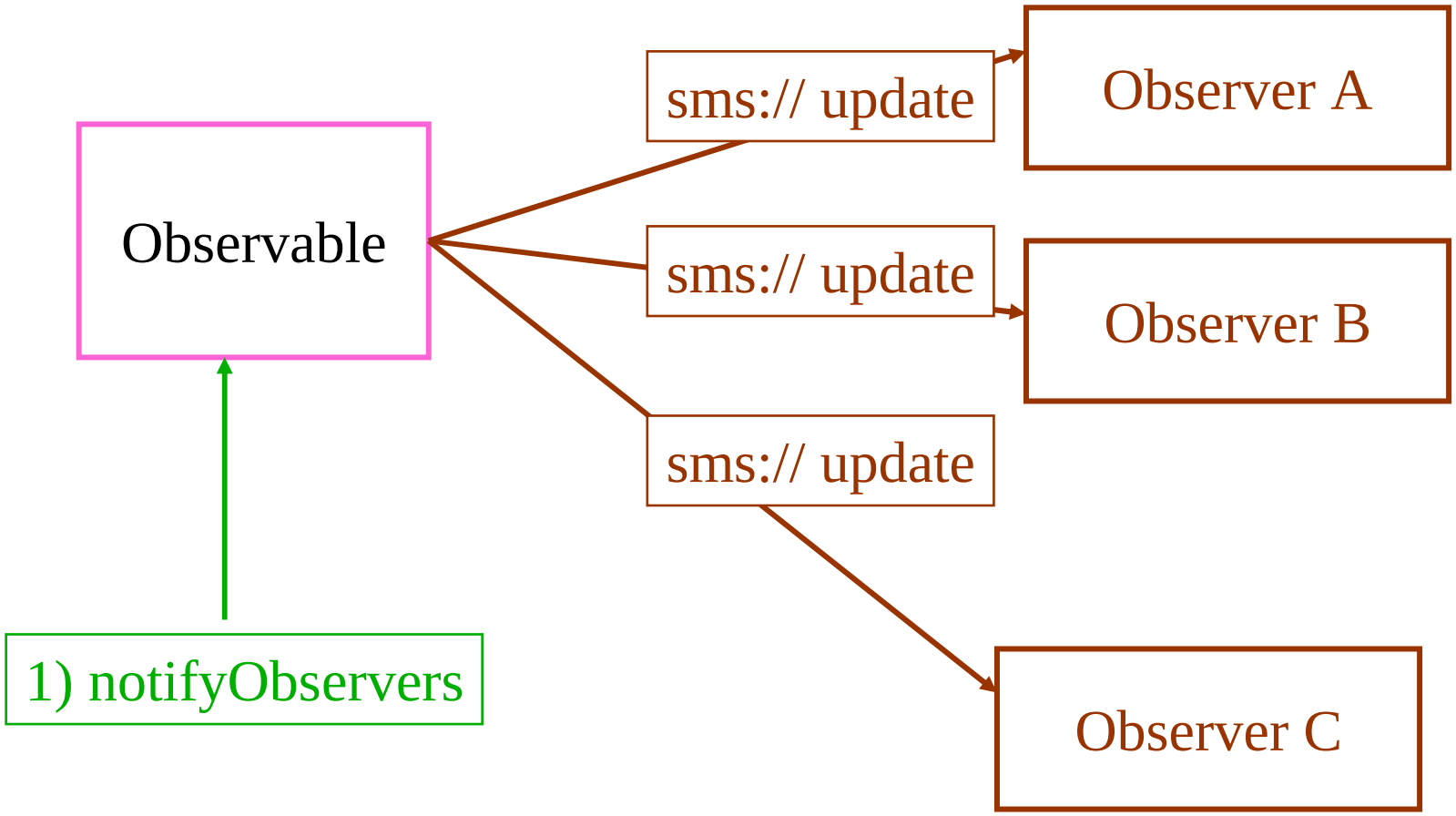
# Observer distribué ?



# Observer distribué ?



# Observer distribué ?

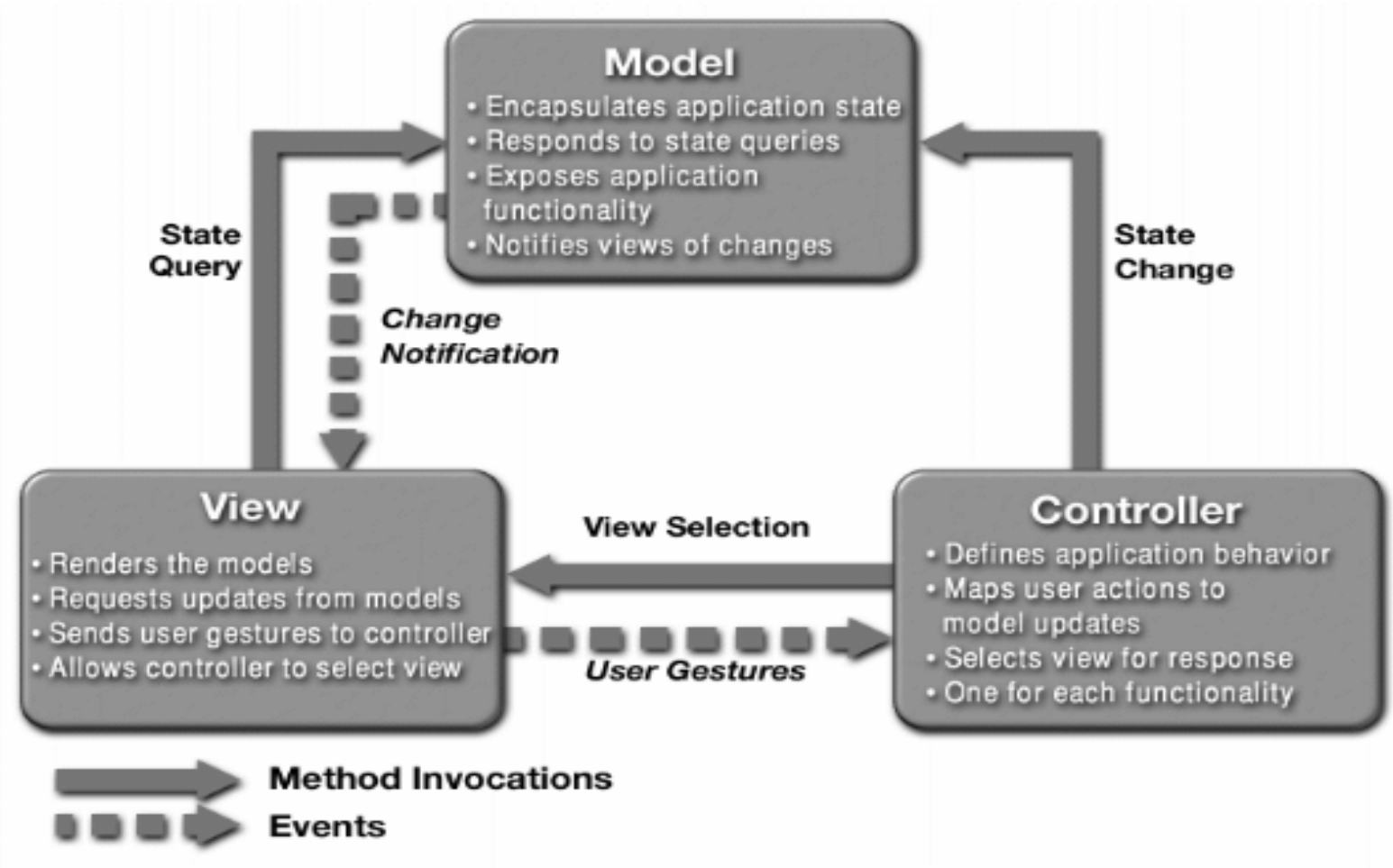


# Observer / MVC

- **Observateur/Observés**

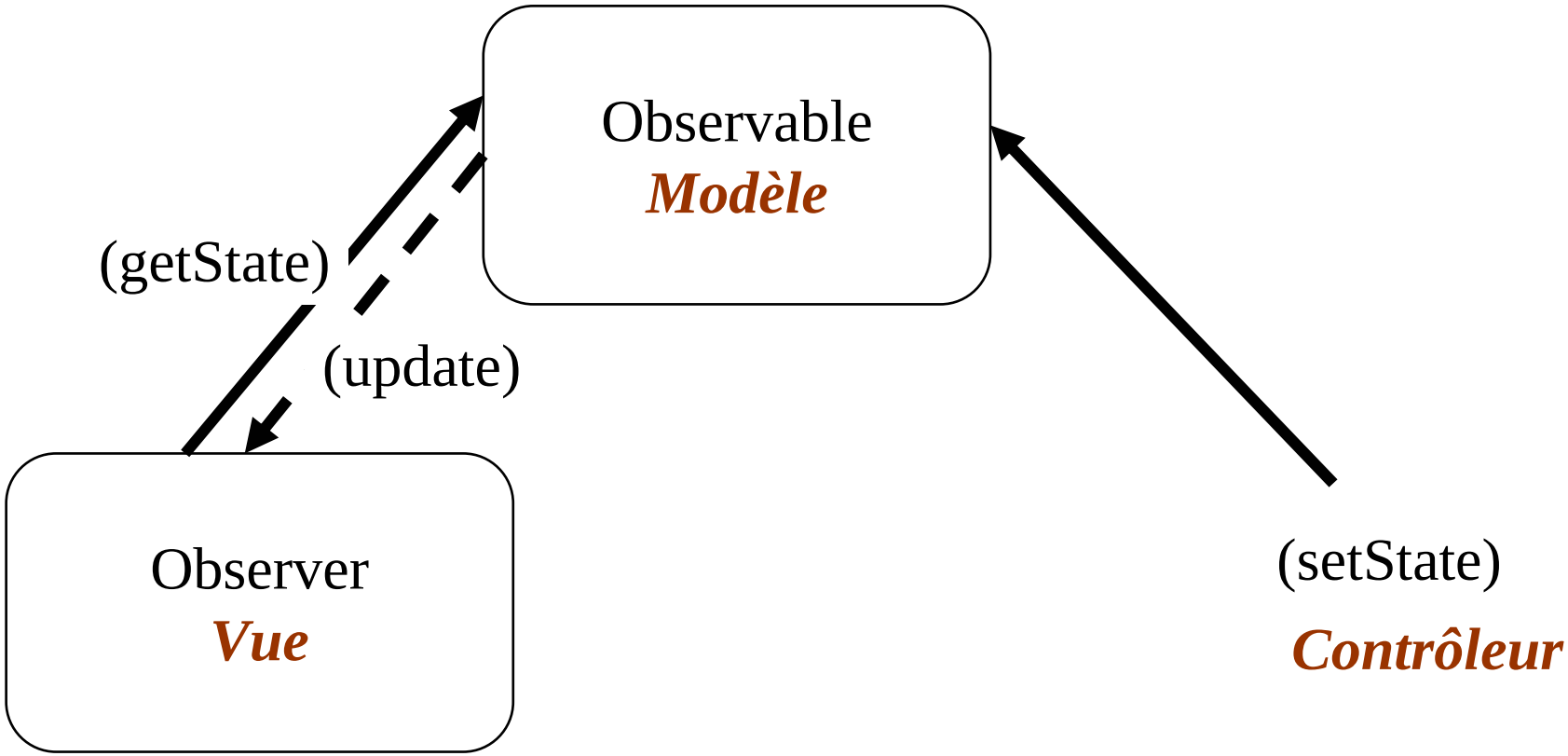
- **Modèle Vue Contrôleur**

# MVC doc de Sun




- **http://java.sun.com/blueprints/patterns/MVC-detailed.html**

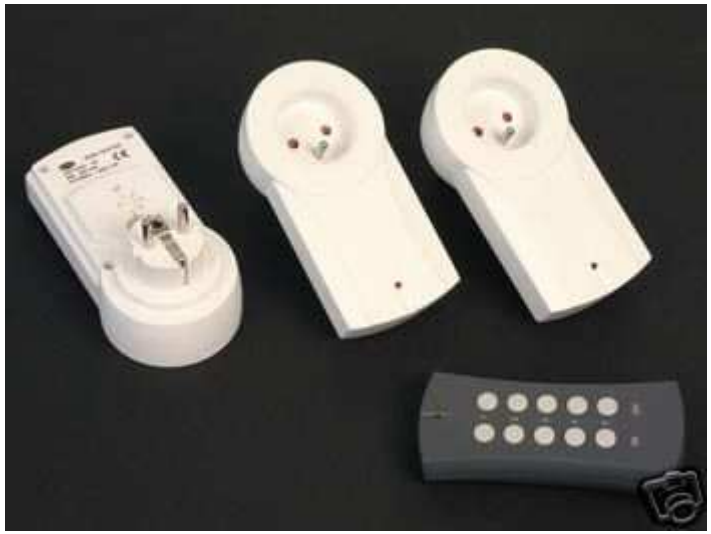
# Observer est inclus MVC



# Autres patrons entre nous

- **Patrons**
  - **Command**
  - **Strategy**
  - **Factory**
  - **Mediator**
  - **Bridge**

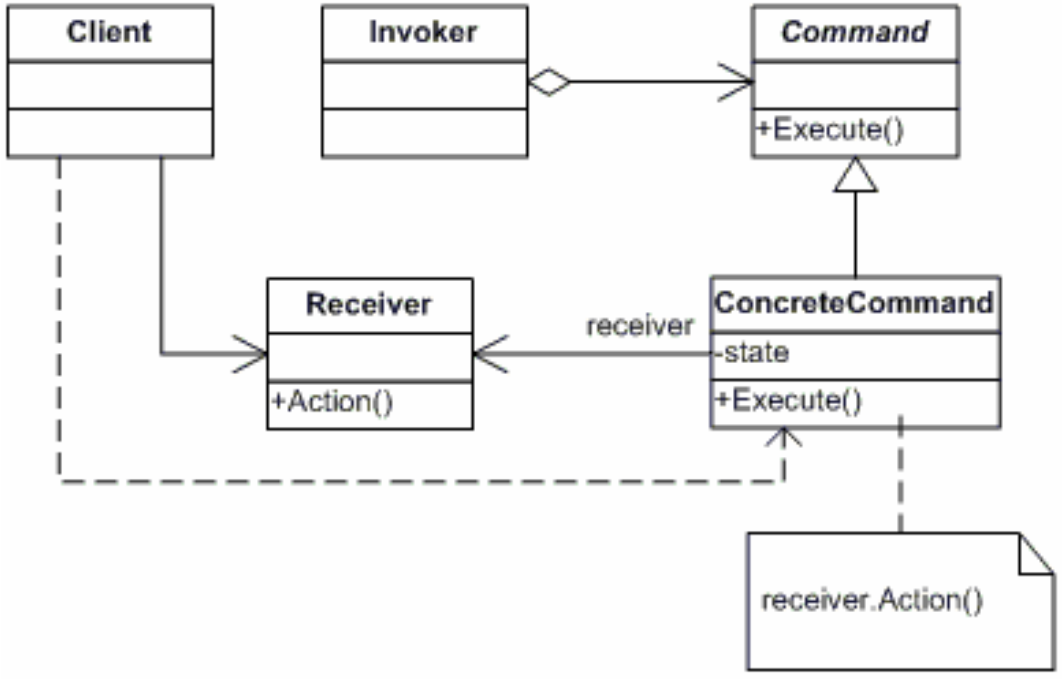
# Le pattern Command



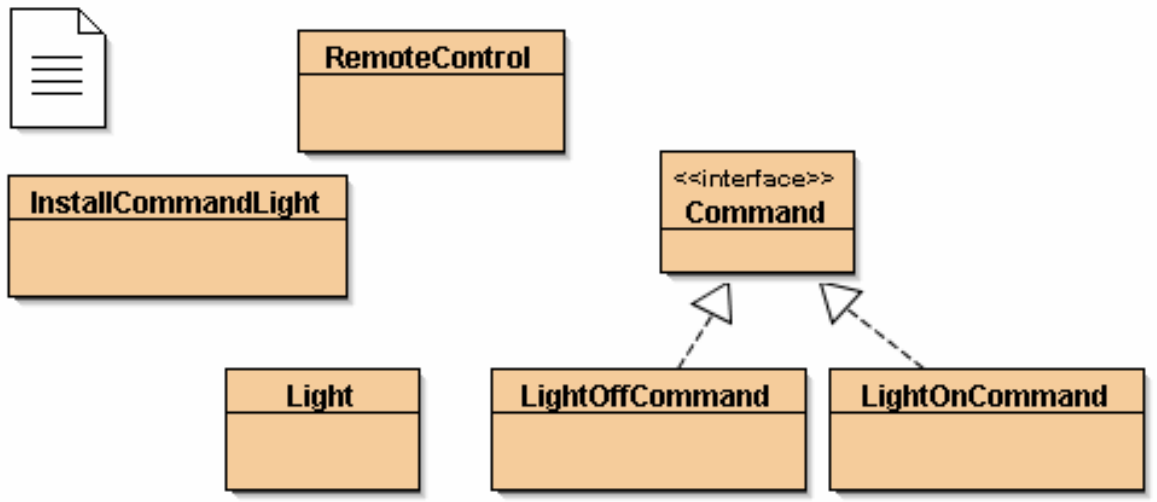
- **Une télécommande**
  - **Générique de type marche/arrêt, l'émetteur**
- **Une prise**
  - **Le récepteur**
- **Une lampe**
  - **Allumée, éteinte**
- **Exemple inspiré de  [Headfirst]**

# Command

- **Couplage faible entre l'invocateur et le récepteur**
  - **Entre l'opération invoquée et la réalisation de cette opération**



# Command exemple inspiré [headFirst]




- **Invoker**
  - RemoteControl, une télécommande on/off de n'importe quoi…
- **Receiver**
  - Light, une lampe qu s'allume et s'éteint …
- **Command**
  - Une interface qui s'exécute ou annule la dernière commande
- **ConcreteCommand**
  - LightOnCommand et LightOffCommand
- **Client**
  - InstallCommandLight : le câblage des actions de la télécommande

# Command un exemple

```
public interface Command{
   public void execute();
   public void undo();
}

public class Light{
 public void on(){
   System.out.println("on !!!");
 }
 public void off(){
   System.out.println("off !!!");
 }
}

public class LightOffCommand implements Command{
    protected Light light;                  // le « receiver »
    public LightOffCommand(Light light){
      this.light = light;
    }
    public void execute(){
      light.off();
    }
    public void undo(){
      light.on();
    }
 }
```

# Command Invoker

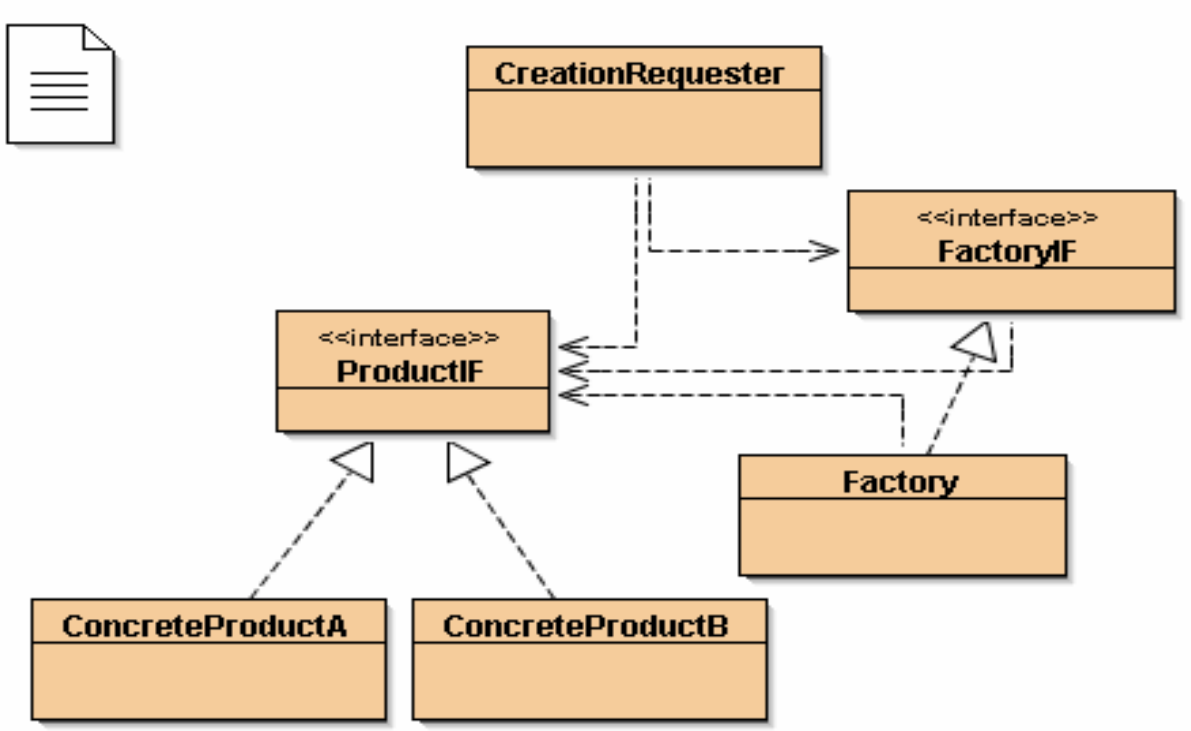
```
public class RemoteControl{ //Invoker
    private Command on,off;
    public void setCommand(Command on, Command off){
      this.on = on;
      this.off = off;
    }
    public void executeCommandOn(){
      on.execute();
    }
    public void executeCommandOff(){
      off.execute();
    }
  }
```

# Command Client

```
public static void test(){
    Light light = new Light();
    Command on = new LightOnCommand(light);
    Command off = new LightOffCommand(light);

    RemoteControl control = new RemoteControl();

    control.setCommand(on,off);
    control.executeCommandOn();
    control.executeCommandOff();
 }
```

# Factory Method

- **Laisser aux sous-classes**
  - **la responsabilité de créer l'objet**
  - **Un constructeur « abstrait », voir template method**




[DP05]

# FactoryIF, ProductIF, Factory

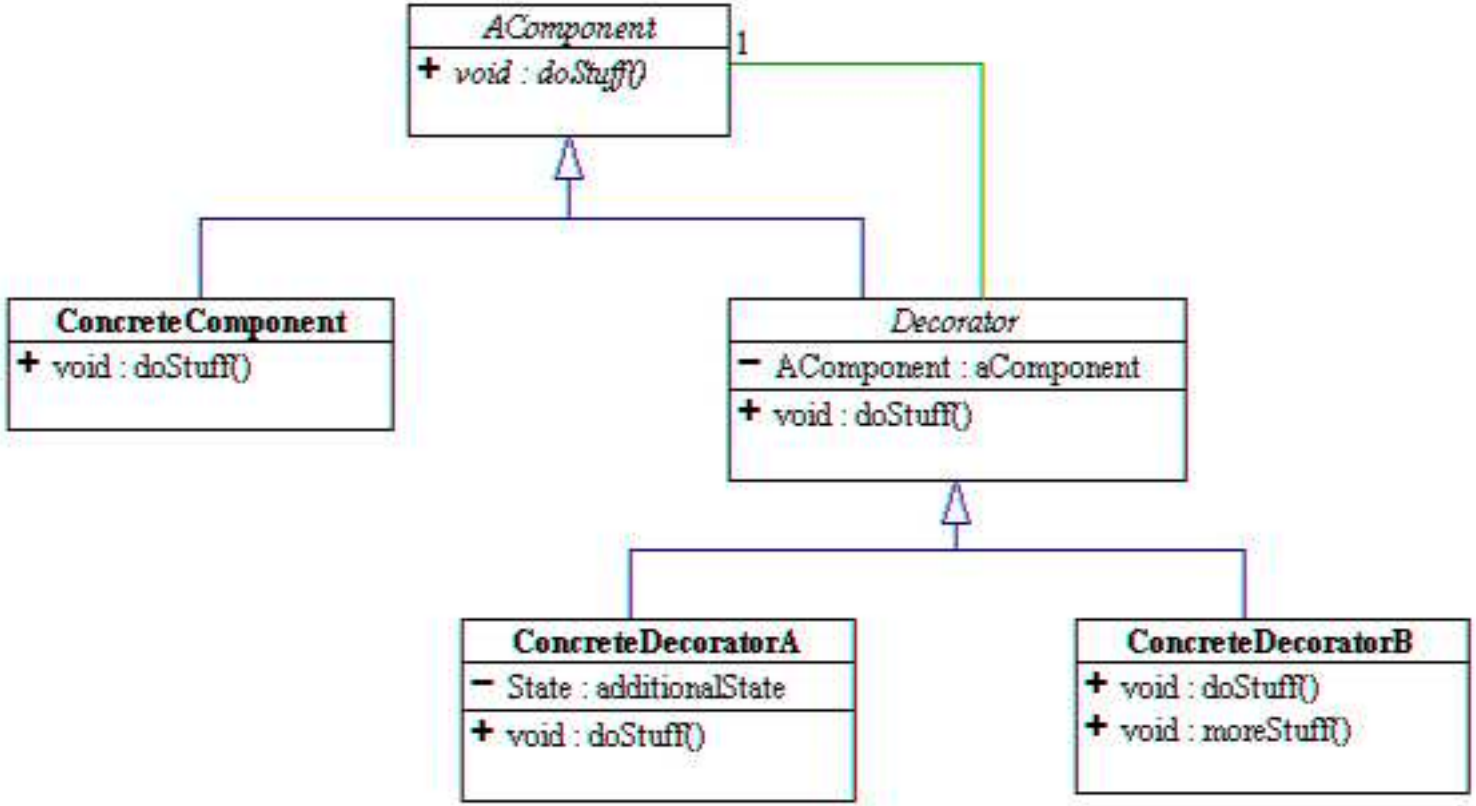
```
public interface FactoryIF {
   public abstract ProductIF CreateProduct(int discriminator);
}

public interface ProductIF {
   public abstract void operation();
}


 public class Factory implements FactoryIF{
   public ProductIF CreateProduct(int discriminator){
        ProductIF newProduct;
        if(discriminator <= 10){
            newProduct = new ConcreteProductA();
        }else{
            newProduct = new ConcreteProductB();
        }
        return newProduct;
    }
}
```

# ConcreteProduct et CreationRequest

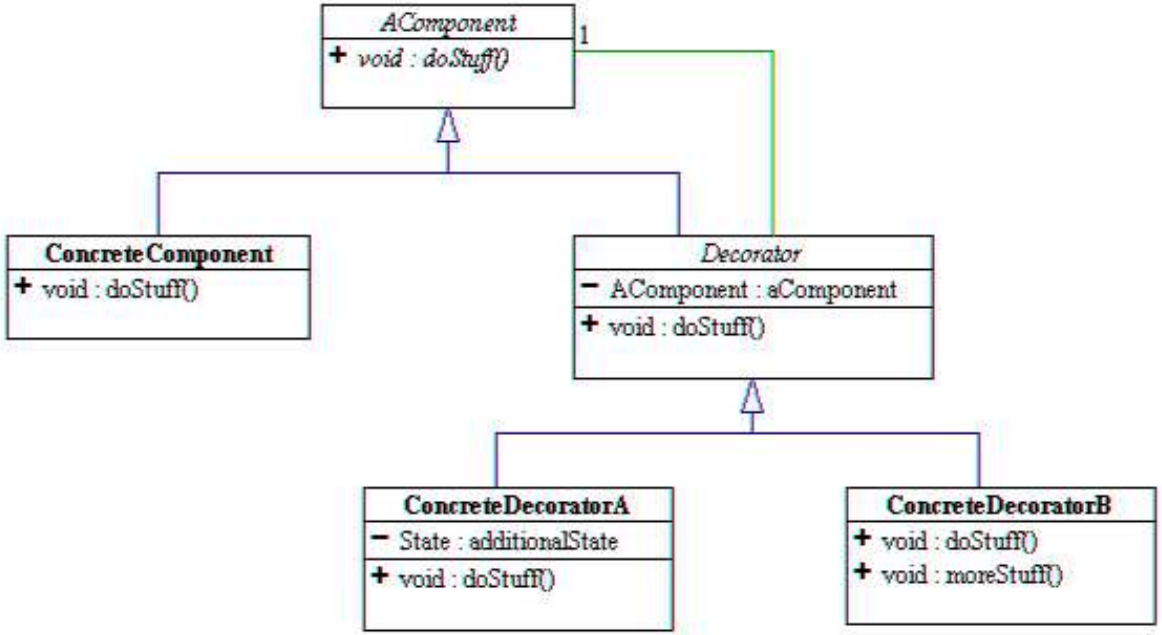
```
public class CreationRequester{

   private FactoryIF factory;

   public CreationRequester(){ factory = new Factory();}
   public CreationRequester(FactoryIF factory){
     this.factory = factory;
    }

   public ProductIF newProduct(int discriminator){
        return factory.CreateProduct(discriminator);
    }
}


public class ConcreteProductA implements ProductIF{

    public void operation(){
        System.out.println("Operation of ConcreteProductA");
    }
}
```

# le Pattern Décorateur

- **Ajout dynamique de responsabilités à un objet**
- **Une alternative à l ’héritage ?**



# Le Pattern : mise en œuvre




- *AComponent* interface ou classe abstraite
- **ConcreteComponent** implémente* *AComponent*

- *Decorator* implémente *AComponent* et contient une instance de *Acomponent*
- **ConcreteDecoratorA, ConcreteDecoratorB** héritent de *Decorator*

- *** implémente ou hérite de**

# Une mise en œuvre(1)

```
public interface AComponent{
   public abstract String doStuff();
}
public class ConcreteComponent implements AComponent{
   public String doStuff(){
      //instructions concrètes;
      return "concrete..."
   }
}

public abstract class Decorator implements AComponent{
   private AComponent aComponent;
   public Decorator(AComponent aComponent){
       this.aComponent = aComponent;
   }
   public String doStuff(){
      return aComponent.doStuff();
   }
}
```

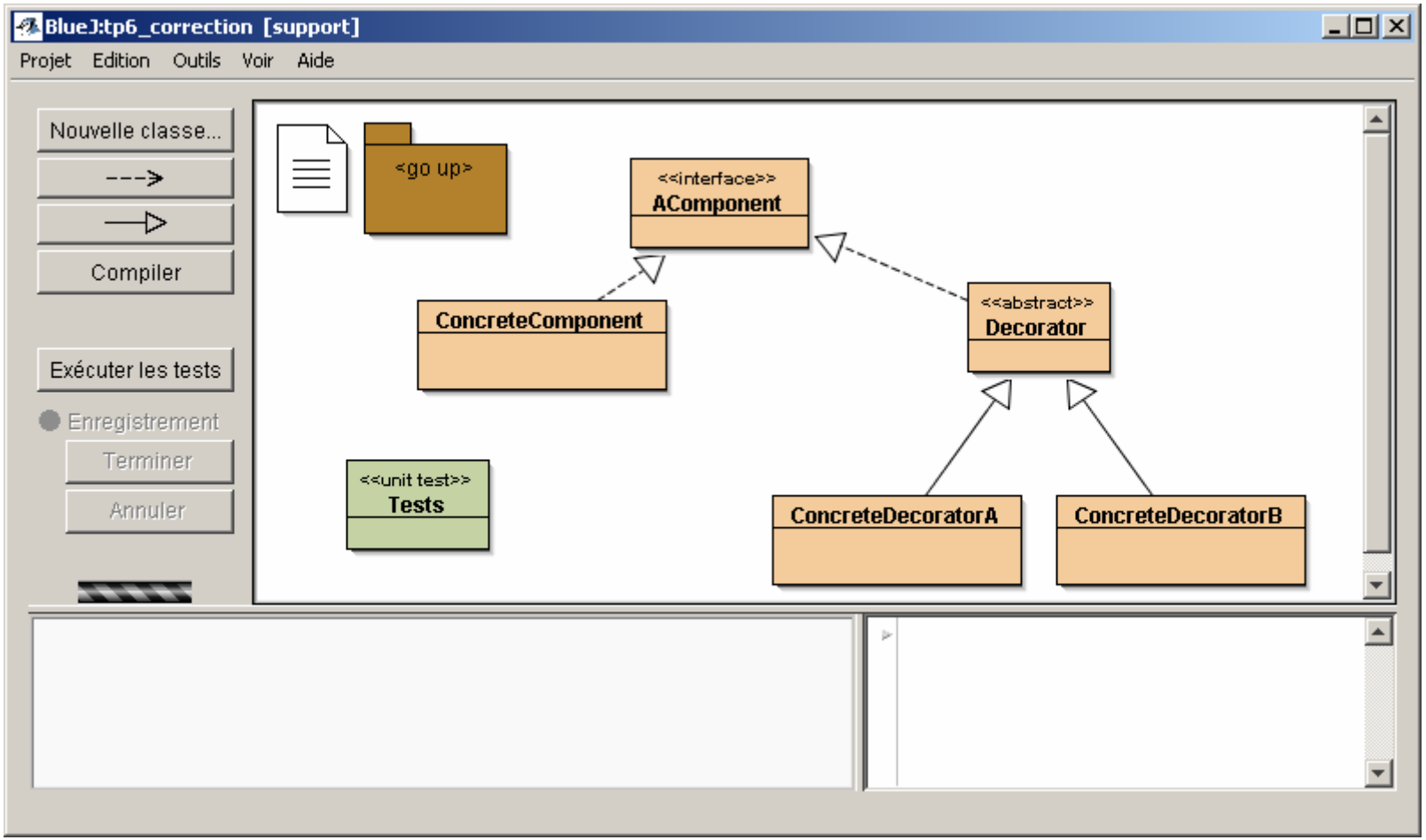
# Mise en œuvre(2)

```
public class ConcreteDecoratorA extends Decorator{
  public ConcreteDecoratorA(AComponent aComponent){
    super(aComponent);
  }
  public String doStuff(){
    //instructions decoratorA;
    return "decoratorA... "  + super.doStuff();
  }
}
```

Déclarations

```
Acomponent concret = new ConcreteComponent();

Acomponent décoré = new ConcreteDecoratorA( concret);
décoré.doStuff();

Acomponent décoré2 = new ConcreteDecoratorA( décoré);
décoré2.doStuff();
```

# Mise en oeuvre, bluej




- **Démonstration …**

# Quelques assertions

Tests

Classe Édition Outils Options

Compiler | Défaire | Couper | Copier | Coller | Chercher... | Chercher suivant | Fermer | Implémentation

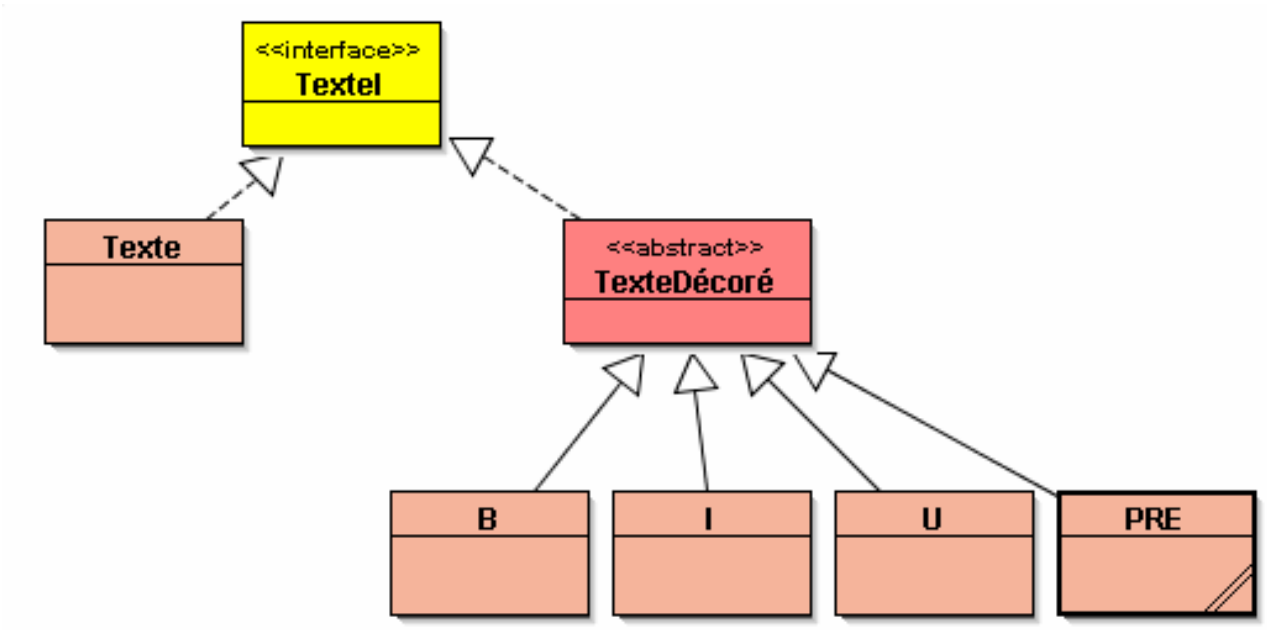
```
5
6   public void testl(){
7     AComponent concret = new ConcreteComponent();
8     assertEquals("concrete...", concret.doStuff());
9
10    AComponent decorA = new ConcreteDecoratorA(concret);
11    assertEquals("decoratorA...concrete...", decorA.doStuff());
12
13    AComponent decorBA = new ConcreteDecoratorB(decorA);
14    assertEquals("decoratorB...decoratorA...concrete...", decorBA.doStuff());
15
16    AComponent decorBB = new ConcreteDecoratorB(new ConcreteDecoratorB(concret));
17    assertEquals("decoratorB...decoratorB...concrete...", decorBB.doStuff());
18
19  }
20}
21
22
```

enregistré

- **(decoratorB (decoratorA (concrete)))**

# Instance imbriquées

- **Instances gigognes + liaison dynamique = Décorateur**
- **Un autre exemple : un texte décoré par des balises HTML**
  - **<b>exemple<b>**



# Le TexteI, Texte et TexteDécoré

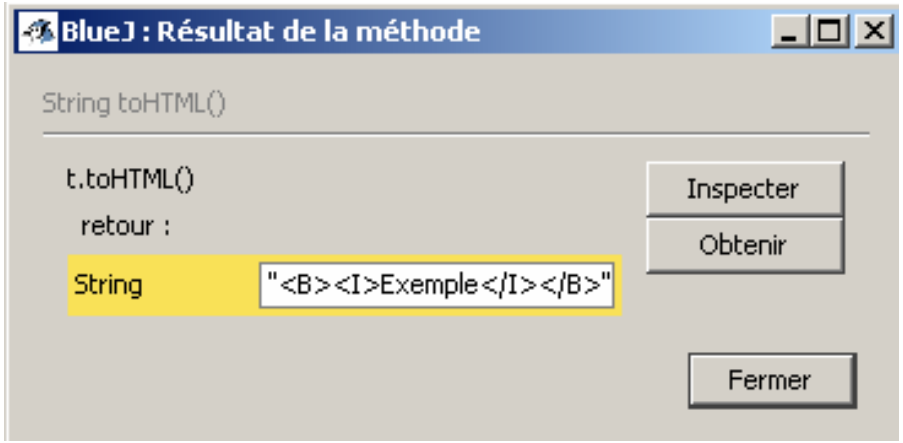
```
public interface TexteI{
  public String toHTML();
}

public class Texte implements TexteI{
  private String texte;
  public Texte(String texte){this.texte = texte;}
  public String toHTML(){return this.texte;}
}

public abstract class TexteDécoré implements TexteI{
  private TexteI unTexte;

  public TexteDécoré(TexteI unTexte){
    this.unTexte = unTexte;
  }
  public String toHTML(){
    return unTexte.toHTML();
  }
}
```

# B, I, U …

```
public class B extends TexteDécoré{

  public B(TexteI unTexte){
    super(unTexte);
  }

  public String toHTML(){
    return "<B>" + super.toHTML() + "</B>";
  }
}

public class I extends TexteDécoré{

  public I(TexteI unTexte){
    super(unTexte);
  }

  public String toHTML(){
    return "<I>" + super.toHTML() + "</I>";
  }
}
```

# Exemple

- **Textel t = new B( new I( new Texte("Exemple")));**
- **String s = t.toHTML();**




- **Démonstration**

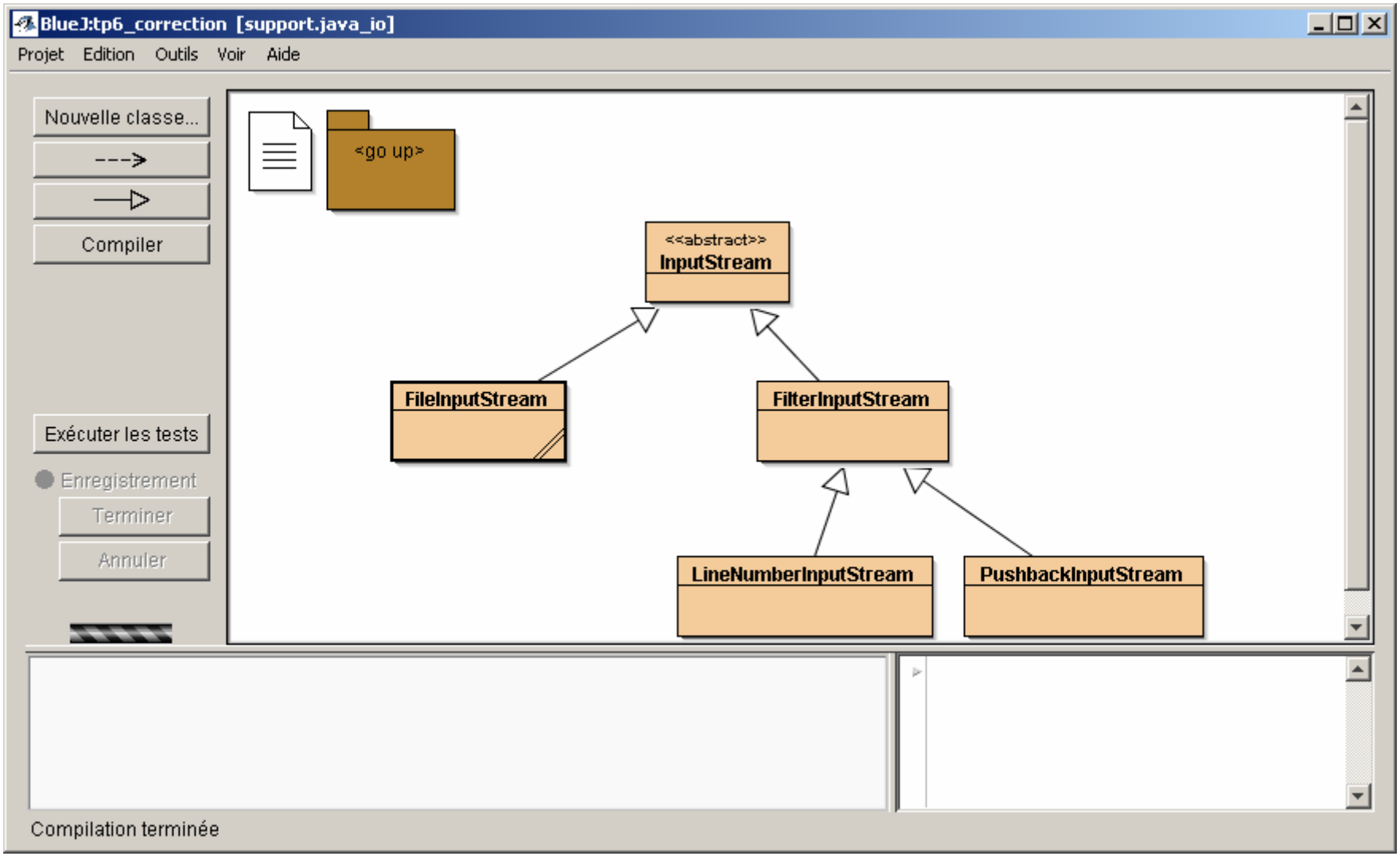
# <B><B> un texte </B></B> non merci

```
AbstractTexte texte = new B( new I( new Texte("ce texte")));
System.out.println(texte.enHTML());

AbstractTexte texte1 = new B(new I(new B( new I(new Texte("ce texte")))));
System.out.println(texte1.enHTML());

AbstractTexte texte2 = new B(new B(new B( new I(new Texte("ce texte")))));
System.out.println(texte2.enHTML());
```

```
<B><I>ce texte</I></B>
<B><I>ce texte</I></B>
<B><I>ce texte</I></B>
```

- **Comment ?**

# java.io

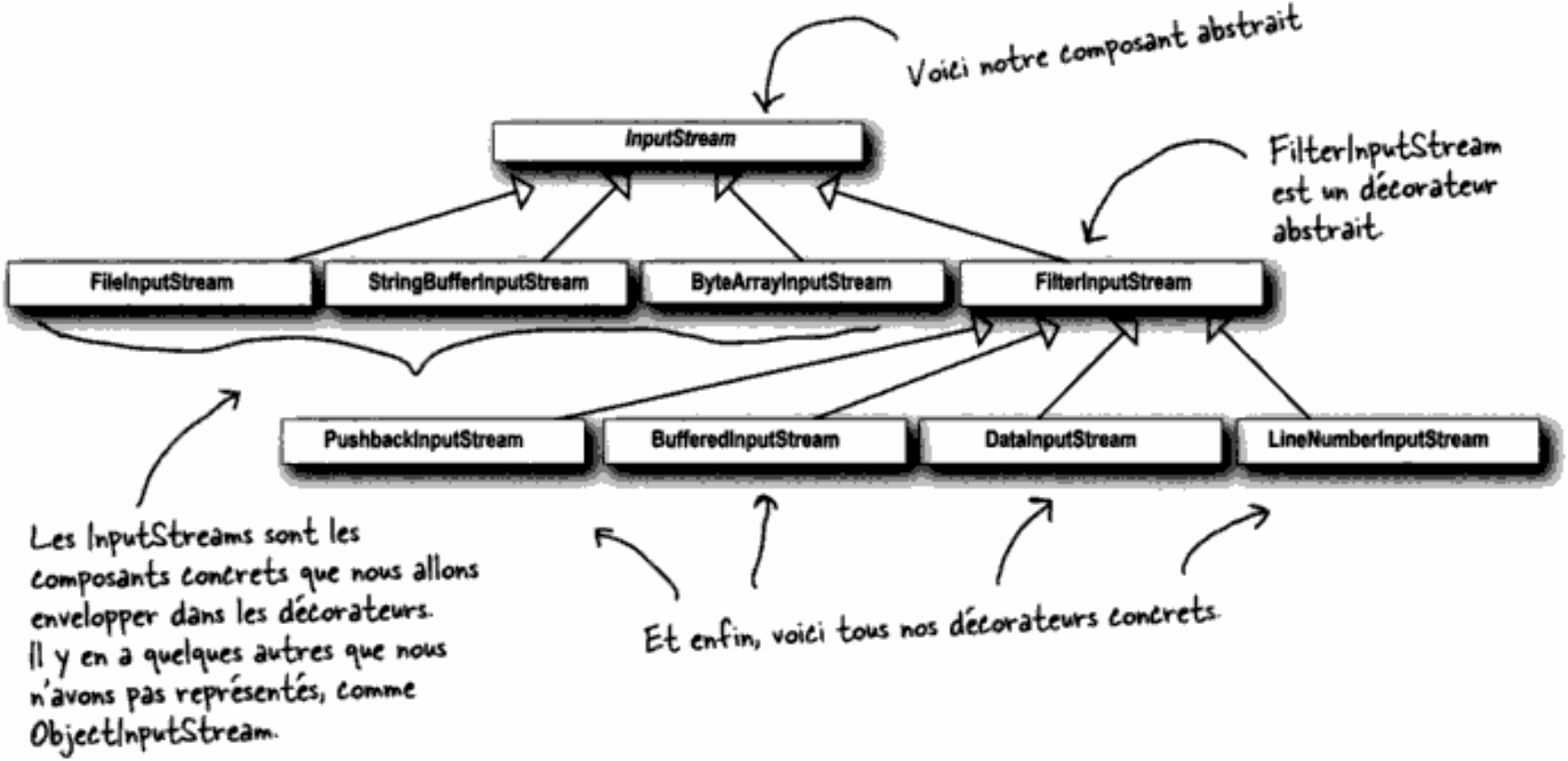



- **Le Décorateur est bien là**

# Extrait de java : tête la première



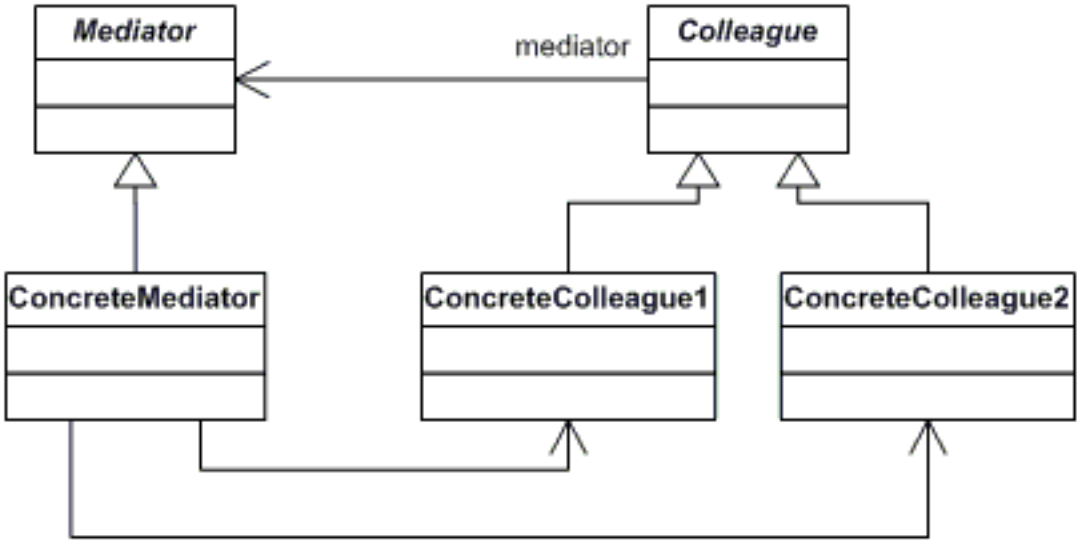
## Décoration des classes de java.io



# java.net.Socket

```
Socket socket = new Socket("vivaldi.cnam.fr", 5000);


 BufferedReader in =
   new BufferedReader(
      new InputStreamReader(socket.getInputStream()));


    System.out.println(in.readLine());
```

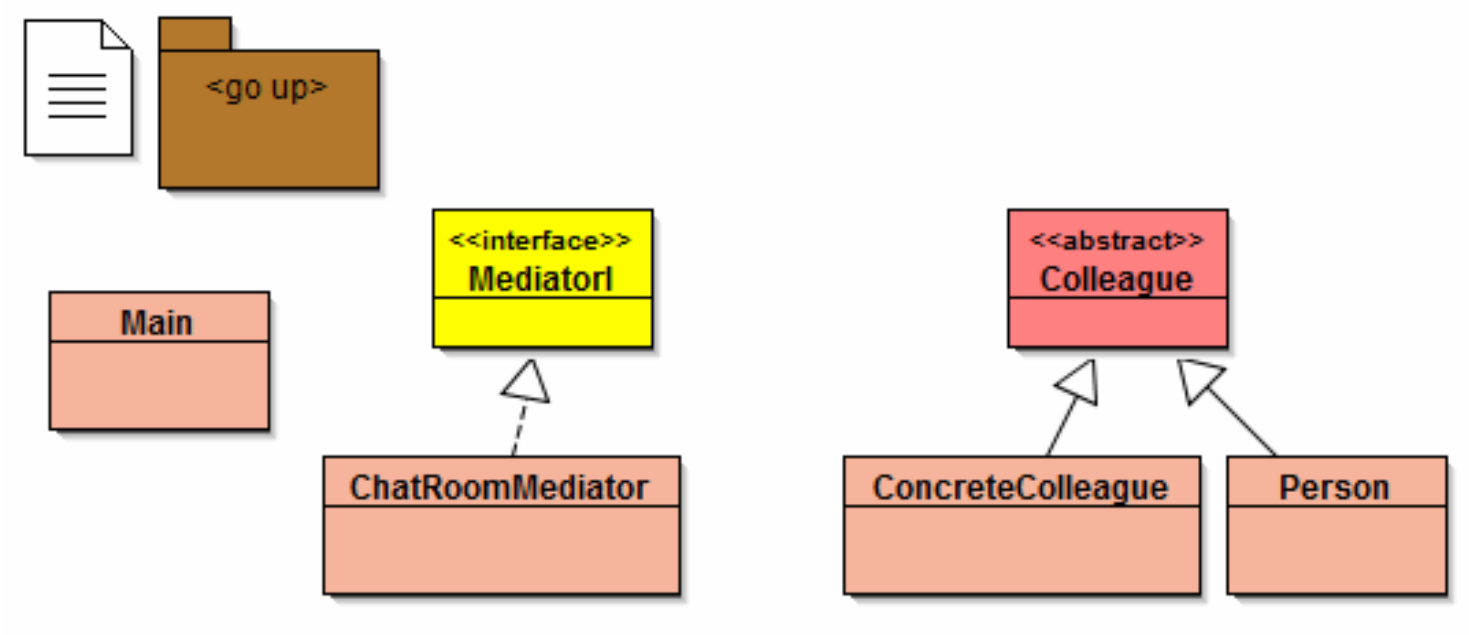
# Mediator




- **Couplage faible des participants(ConcreteColleague)**
  - **Ils ne se connaissent pas**
- **Un médiateur se charge de la communication**
  - **Les participants ne connaisse que le médiateur**

- **En exemple : un logiciel de causerie …**

# Un ChatRoom, un logiciel de causerie




- **Un exemple approprié**
  - Le *ChatRoom* est à l'écoute des participants *(Colleague)*
    - *Méthode send(String message, Colleague source)*
  - *Chaque* participant reçoit ce qui est envoyé au médiateur
    - *Méthode receive(String message)*
- **Couplage faible des participants**
  - Absence de liaison entre les participants (ils n'ont pas besoin de se connaître)

# Colleague, les participants au ChatRoom

```
3  public abstract class Colleague{
4    private MediatorI mediator;
5
6    public Colleague(MediatorI mediator){
7      this.mediator = mediator;
8    }
9
10   public void send(String message){
11     mediator.send(message, this);
12   }
13
14   public MediatorI getMediator(){
15     return mediator;
16   }
17
18   public abstract void receive(String message);
```

- **Ligne 11, les participants s'adresse au médiateur**
- **Ligne 18, dans l'attente d'un message …**

# MediatorI, ChatRoom

```
4 public interface MediatorI{
5
6   public void send(String message, Colleague collegue);
7 }
```

- **Tous les participants s'adressent au médiateur**
  - **Afin ici de diffuser le message**

- **Sans médiateur,**
  - **Chaque participant devrait avoir la connaissance de ses voisins**

# ChatRoom Mediator

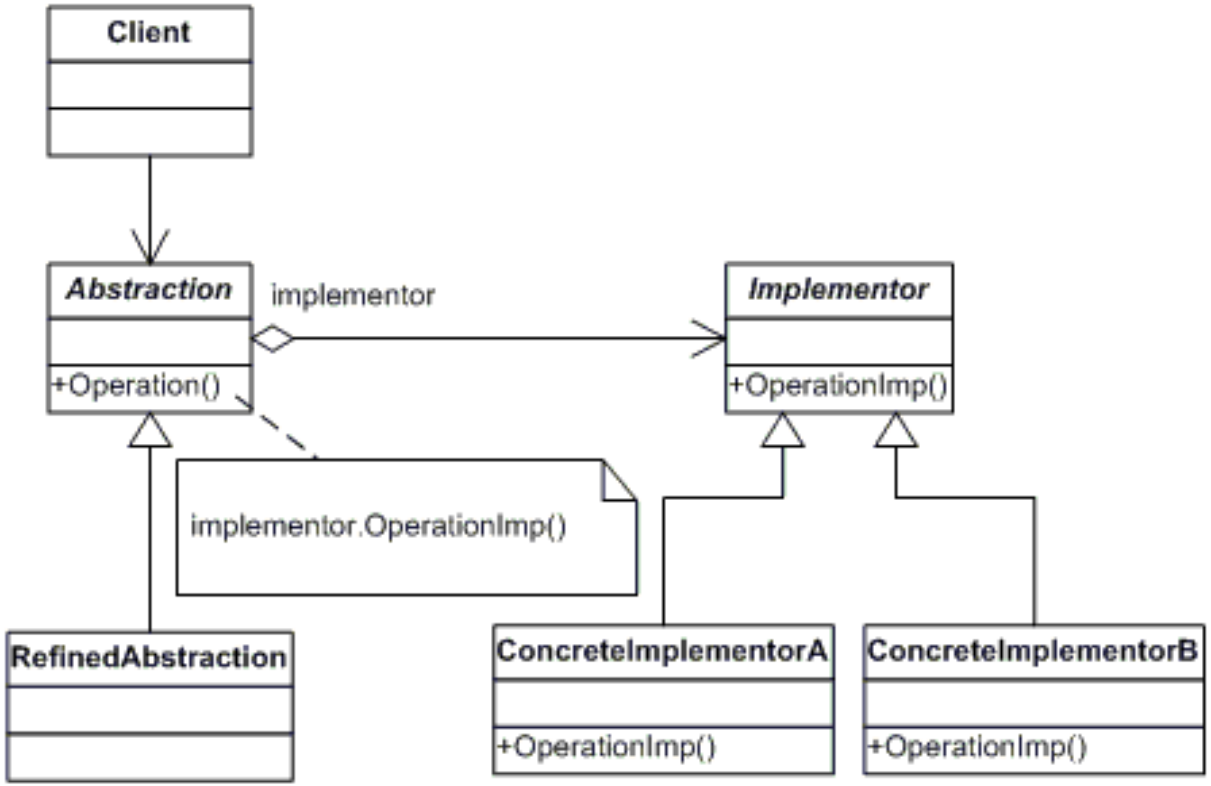
```
6  public class ChatRoomMediator implements MediatorI{
7    private List<Colleague> colleagues;
8
9    public ChatRoomMediator(){
10     this.colleagues =  new ArrayList<Colleague>();
11   }
12
13   public void addParticipant(Colleague c){
14     this.colleagues.add(c);
15   }
16
17   public void send(String message, Colleague origin){
18     for(Colleague c : colleagues){
19       if(!c.equals(origin))
20         c.receive(message);
21     }
```

- **Ligne 17: Envoi du message à tous les participants** (sauf soi-même)

# Un client : la classe Main

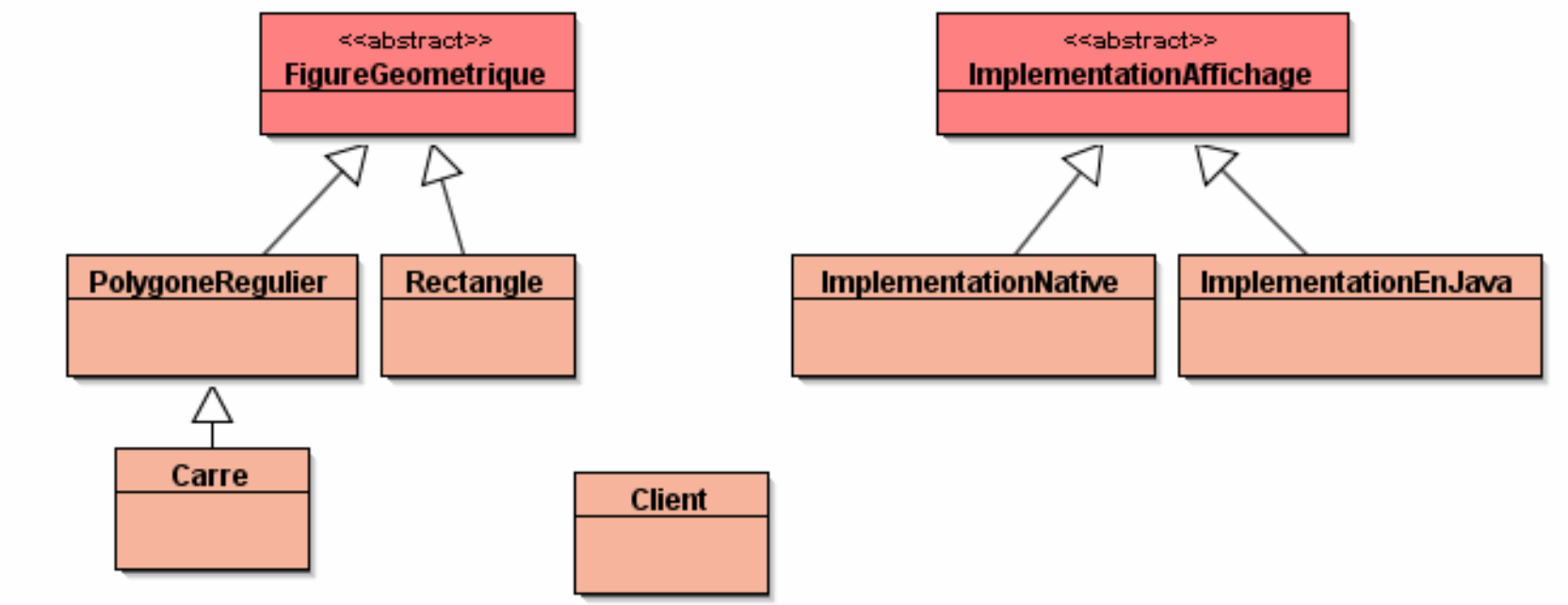
```
5   public static void main(String[] args) {
6     ChatRoomMediator mediator = new ChatRoomMediator();
7     ConcreteColleague alfred = new ConcreteColleague(mediator);
8     ConcreteColleague paul = new ConcreteColleague(mediator);
9     ConcreteColleague bill = new ConcreteColleague(mediator);
10    Person max = new Person(mediator);
11
12    mediator.addParticipant(alfred);
13    mediator.addParticipant(paul);
14    mediator.addParticipant(bill);
15    mediator.addParticipant(max);
16
17    paul.send("Bonjour à tous !!!");
18    //bill.send("Bonjour vous tous !!!");
19
20
```

# A la recherche du couplage faible




- **Patron Bridge,**
  - **Découpler l'abstraction de son implémentation**
    - **En deux niveaux indépendants (Abstraction/Implémentation)**
  - **Un exemple concret: appel de JNI, Java Native Interface**
    - **ConcreteImplementorA est écrit en C …**

# Exemple :




- **Les deux composantes évoluent séparément**
  - **Abstraction: FigureGéométriques**
    - **PolygoneRégulier, Rectangle**
  - **Implémentation: ImplementationAffichage**
    - **ImplentationNative ou en Java**

# Bridge l'exemple (Abstraction)

```
3  public abstract class FigureGeometrique{
4    private ImplementationAffichage implementation;
5
6    public FigureGeometrique(ImplementationAffichage implementation){
7      this.implementation = implementation;
8    }
9
10   public void dessiner(){
11     implementation.dessiner(this);
12   }
```

```
3  public class Rectangle extends FigureGeometrique{
4   public Rectangle(ImplementationAffichage impl){
5      super(impl);
6    }
7
8    public void dessiner(){
9      System.out.println(" dessin d'un rectangle ...");
10     super.dessiner();
11   }
```

- **Côté abstraction**
  - **FigureGeometrique**
  - **Rectangle**

# Bridge suite (Implementation)

```
5 public abstract class ImplementationAffichage{
6
7   public abstract void dessiner(FigureGeometrique f);
8 }
```

```
3 public class ImplementationNative extends ImplementationAffichage{
4   public void dessiner(FigureGeometrique f){
5     System.out.println("dessin en natif de: " + f);
6   }
7 }
```

- **Côté implementation**
  - **ImplementationAffichage**
  - **En natif… ou en java**

# Conclusion intermédiaire

- **Est-ce bien utile ?**
- **Comment choisir ?**
- **Trop de Patterns ?**
- **→ Une conception par les patrons d'une application**

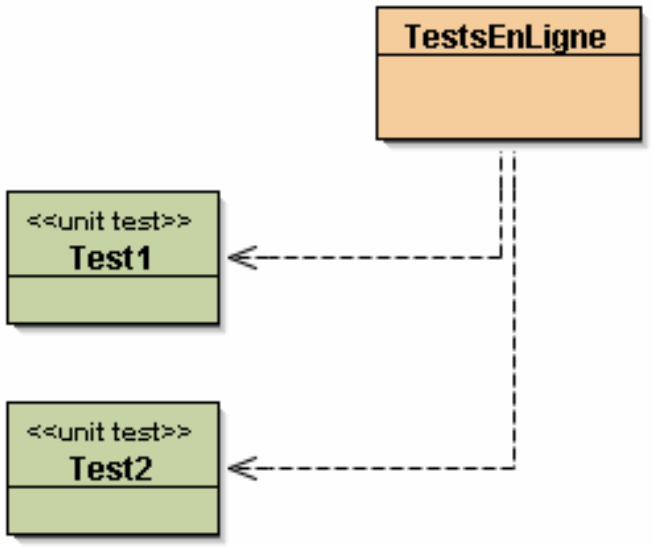
# Annexes

- **Extraits choisis**
  - **Une page web**
    - **Au hasard …**
  - **L'application JUnit**
  - **L'AWT**

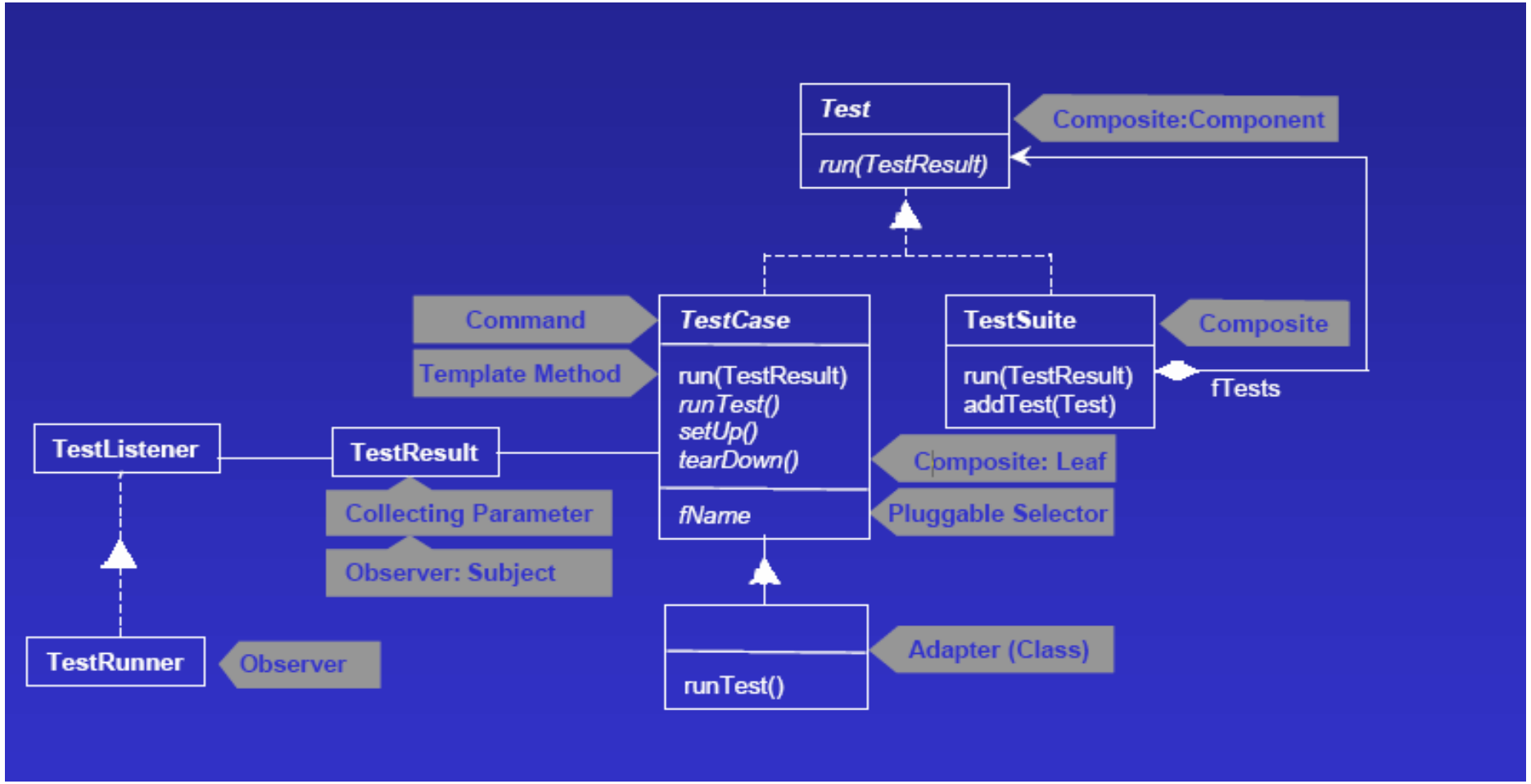
# Un exemple de conception en « patterns »

- http://junit.sourceforge.net/doc/cookstour/cookstour.htm
- http://www-128.ibm.com/developerworks/java/library/j-aopwork7/index.html

- **Tests unitaires**
  - **Intégrés à Bluej**
  - **Exécutables en ligne de commande (http://jfod.cnam.fr/JNEWS/)**



# Architecture Junit …




- **Extraite de http://www.old.netobjectdays.org/pdf/99/jit/gamma.pdf**
- **Collecting Parameter & Pluggable Selector, pages suivantes**

# Collecting Parameter

- http://c2.com/cgi/wiki?CollectingParameter

```
String[] userFiles = ...
List<String> userList = new ArrayList<String>();
for (int i=0; i < userFiles.length; i++) {
  addUsersTo(userFiles[i], userList);
}
```

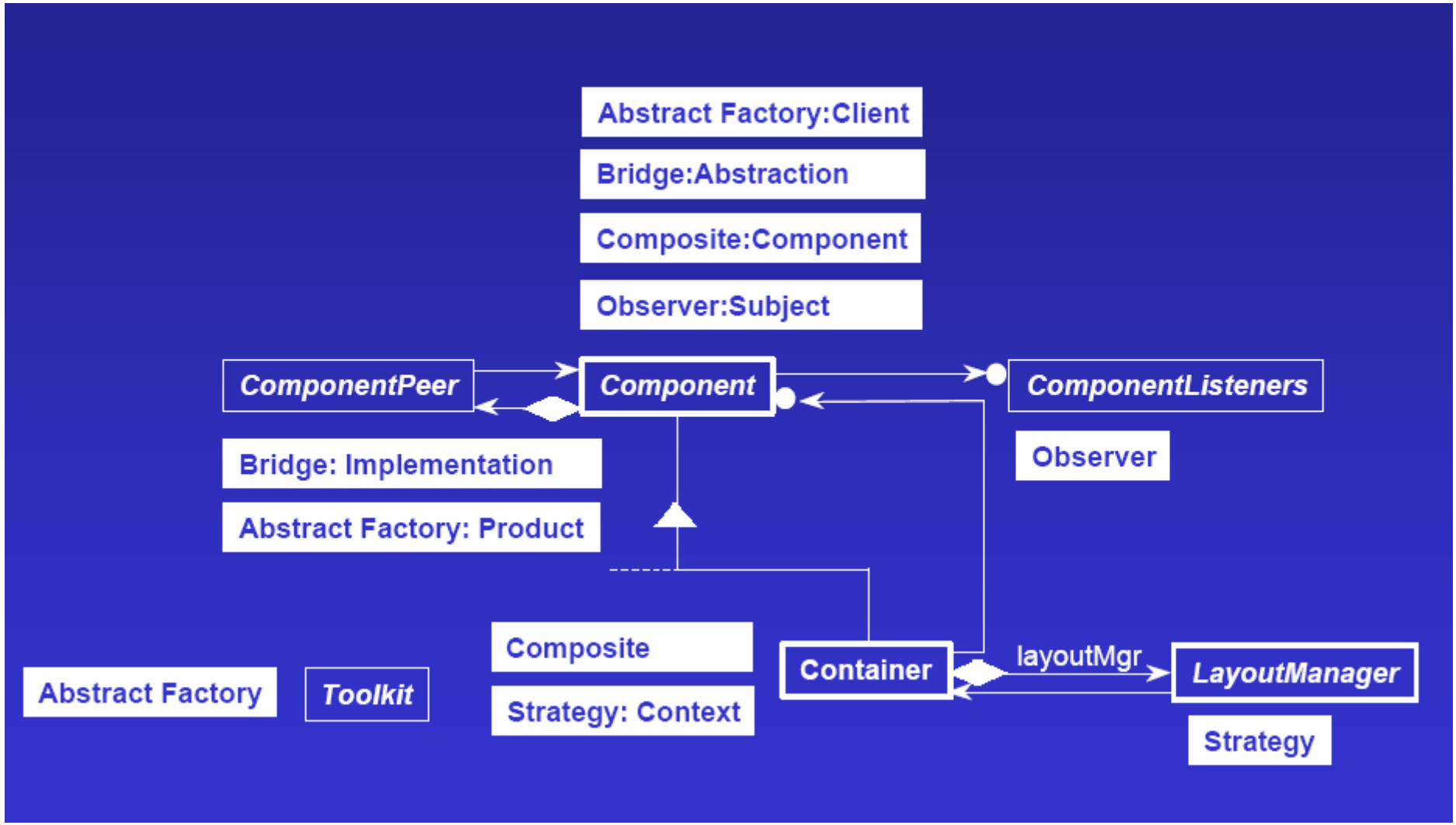
```
public void addUsersTo(String userFileName, List userList) {
   ...
}
```

# Pluggable Selector

- **Sélection d'une méthode à l'exécution**
  - **Introspection en java, les noms des méthodes choisies par l'utilisateur**

```
void runTest(String name) throws Throwable{
   Method runMethod = null;
   try{
     runMethod = getClass().getMethod(name,new Class[]{});
   }catch(Exception e){
     //NoSuchMethodException, SecurityException
   }
   try{
     runMethod.invoke(this,new Object[]{});
   }catch(Exception e){
     // IllegalAccessException, IllegalArgumentException, InvocationTargetException
   }
 }
```

# Une autre architecture (bien connue… ? )




– **Extrait de http://www.old.netobjectdays.org/pdf/99/jit/gamma.pdf**

# Conclusion

- **Architecture décrite par les patterns ?**
- **Langage de patterns ?**
- **Méthodologie d'un AGL ?**