Le patron Décorateur

jean-michel Douin, douin au cnam point fr
version : 2 Décembre 2019

Notes de cours

Sommaire pour les Patrons

- **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

Les patrons déjà vus en quelques lignes …

- **Adapter**
  - Adapte l'interface d'une classe conforme aux souhaits du client
- **Proxy**
  - Fournit un mandataire au client afin de contrôler/vérifier ses accès
- **Observer**
  - Notification d'un changement d'état d'une instance aux observateurs inscrits
- **Template Method**
  - Laisse aux sous-classes une bonne part des responsabilités
- **Iterator**
  - Parcours d'une structure sans se soucier de la structure interne choisie
- **Composite**
  - Définition d'une structure de données récursives
- **Interpreter**
  - Un calcul, une interprétation du noeud d'un composite
- **Visitor**
  - Parcours d'une structure Composite
- **Command, Memento,…**

Sommaire

- **Le patron Décorateur**
  - Comportement dynamique d 'un objet
    - Trois exemples
      - Un texte décoré de balises HTML
      - Un source java décoré de pré et post assertions et d'invariant
      - Une pizza + garniture = une pizza décorée…
- **Alternative à l'héritage ?**
- **Comment ajouter ou retirer des fonctionnalités**
- **Décorateur et femtoContainer**
  - Vers une séparation de la configuration de l'utilisation

Principale bibliographie

- **GoF95**
  - **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**
  - **Design Patterns, Elements of Reusable Object-oriented software Addison Wesley 1995**
- **+**
  - **http://www.eli.sdsu.edu/courses/spring98/cs635/notes/composite/composite.html**
  - **http://www.patterndepot.com/put/8/JavaPatterns.htm**
- **+**
  - **http://www.javaworld.com/javaworld/jw-12-2001/jw-1214-designpatterns.html**

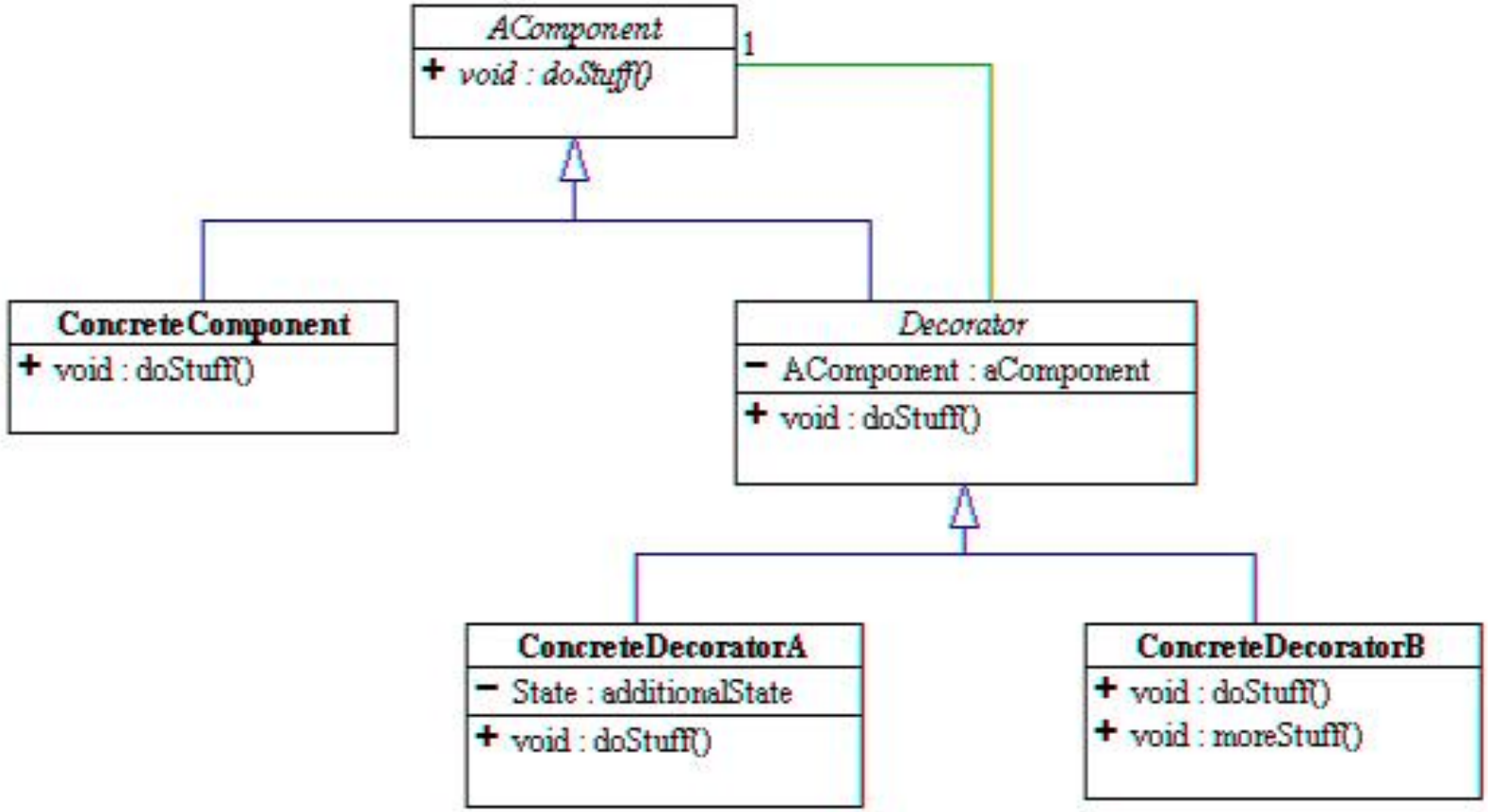
http://jfod.cnam.fr/NFP121/Chapter03_Head_First.pdf

Le Pattern Décorateur

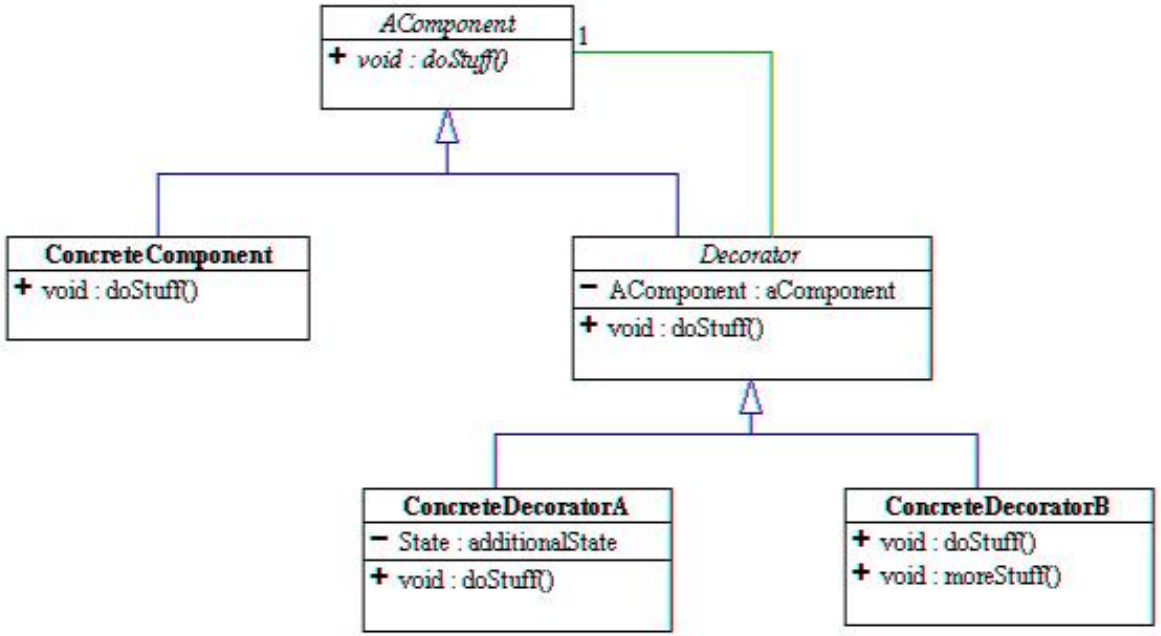
- **Ajout dynamique de responsabilités**
- **Alternative à l'héritage ?**
- **Transparent au client ?**
- **Wikipédia :**
  - <<Un décorateur permet d'attacher dynamiquement de nouvelles responsabilités à un objet. Les décorateurs offrent une alternative assez souple à l'héritage pour composer de **nouvelles fonctionnalités**.>>

le Pattern Décorateur

- **Ajout dynamique de responsabilités à un objet**



Le Pattern : mise en œuvre



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

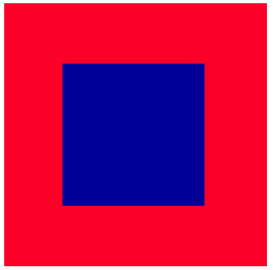
- *Decorator* implémente *AComponent* et contient une instance de *AComponent*
- *Cette instance est décorée*
- **ConcreteDecoratorA, ConcreteDecoratorB** héritent de *Decorator*

- *** implémente ou hérite de**

Quelques déclarations

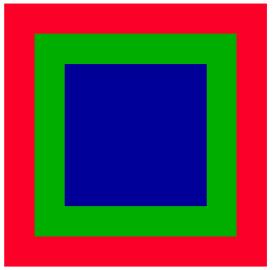
Un composant standard

- AComponent a = new ConcreteComponent();
- a.doStuff();



Un composant décoré

- AComponent a = new ConcreteDecoratorA(new ConcreteComponent());
- a.doStuff();

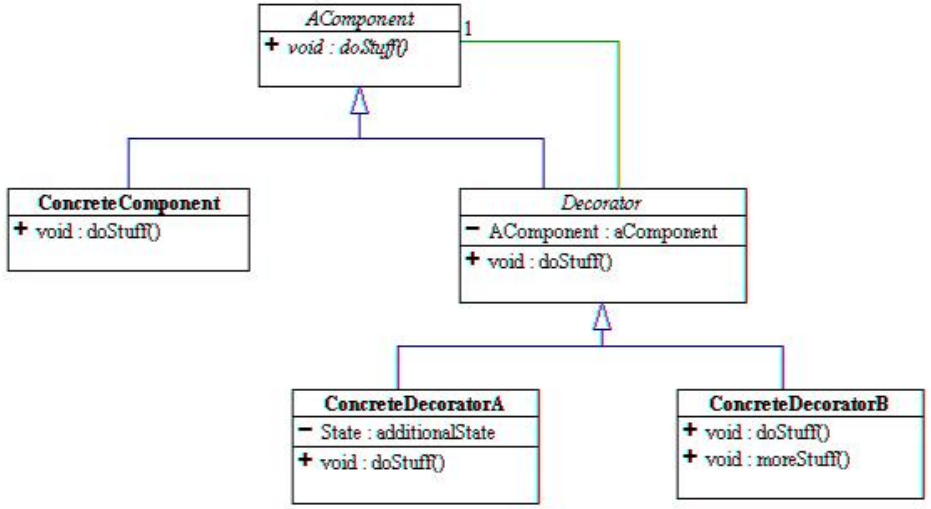


Un composant décoré deux fois

- AComponent a = new ConcreteDecoratorA(
- new ConcreteDecoratorB(new ConcreteComponent()));
- a.doStuff();

public abstract class Decorator

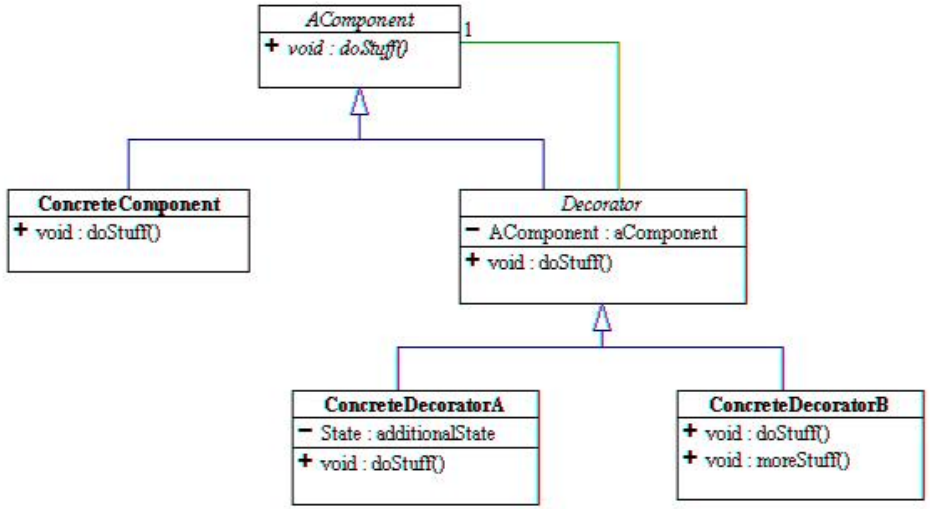
La classe Decorator contient
L'instance à décorer :
aComponent




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

  public void doStuff(){
    aComponent.doStuff();
  }
}     // un exemple concret, vite …
```

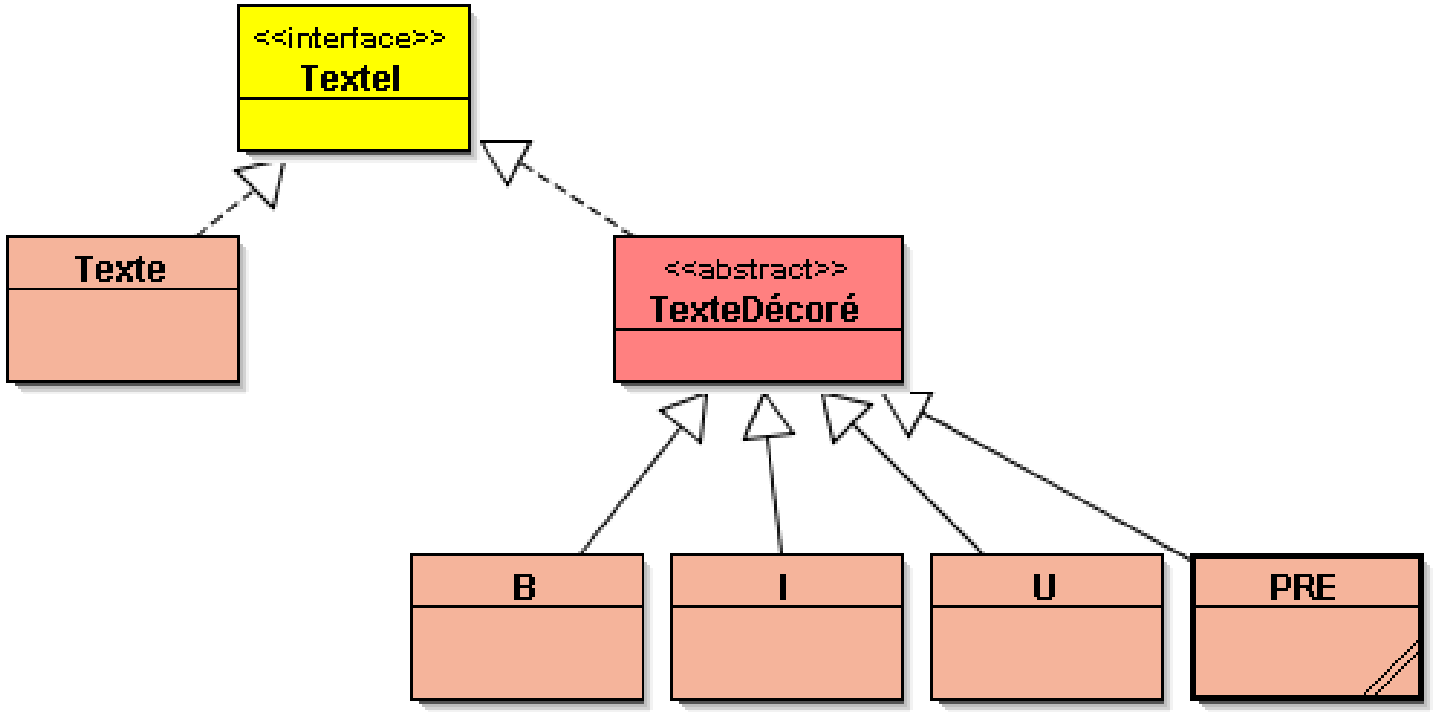
Suite en exemples

- **Un exemple de texte décoré**
  - **<B>** un texte en caractères gras **</B>** un texte en caractères gras
  - **<U> <I> un texte en italique et souligné </I> </U>** *un texte en italique et souligné*
  - **ConcreteComponent : le texte**
  - **ConcreteDecoratorA : les balises**



Un exemple de texte décoré

- **Un exemple : un texte décoré par des balises HTML**
  - **<b>exemple<b>**



Le TexteI, Texte et TexteDécoré

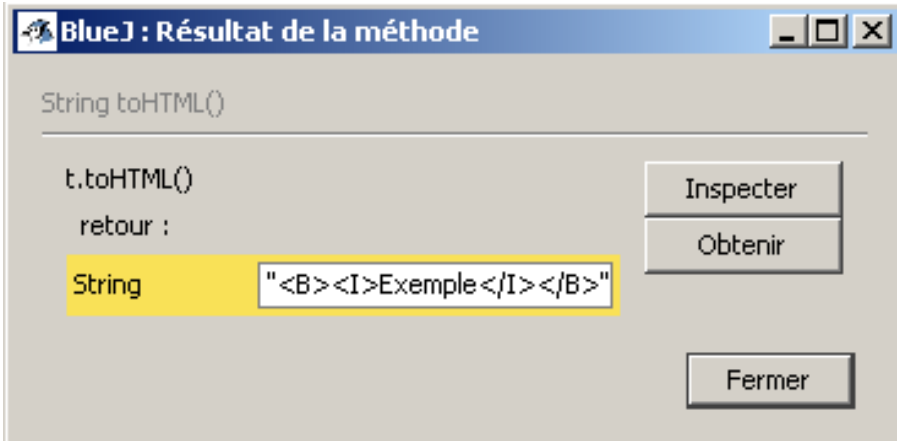
```
public interface TexteI{
  public String toHTML();
}

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; // ← le texte à décorer

  public TexteDécoré(TexteI unTexte){
    this.unTexte = unTexte;
  }
  public String toHTML(){
    return unTexte.toHTML(); // ← le texte décoré
  }
}
```

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();



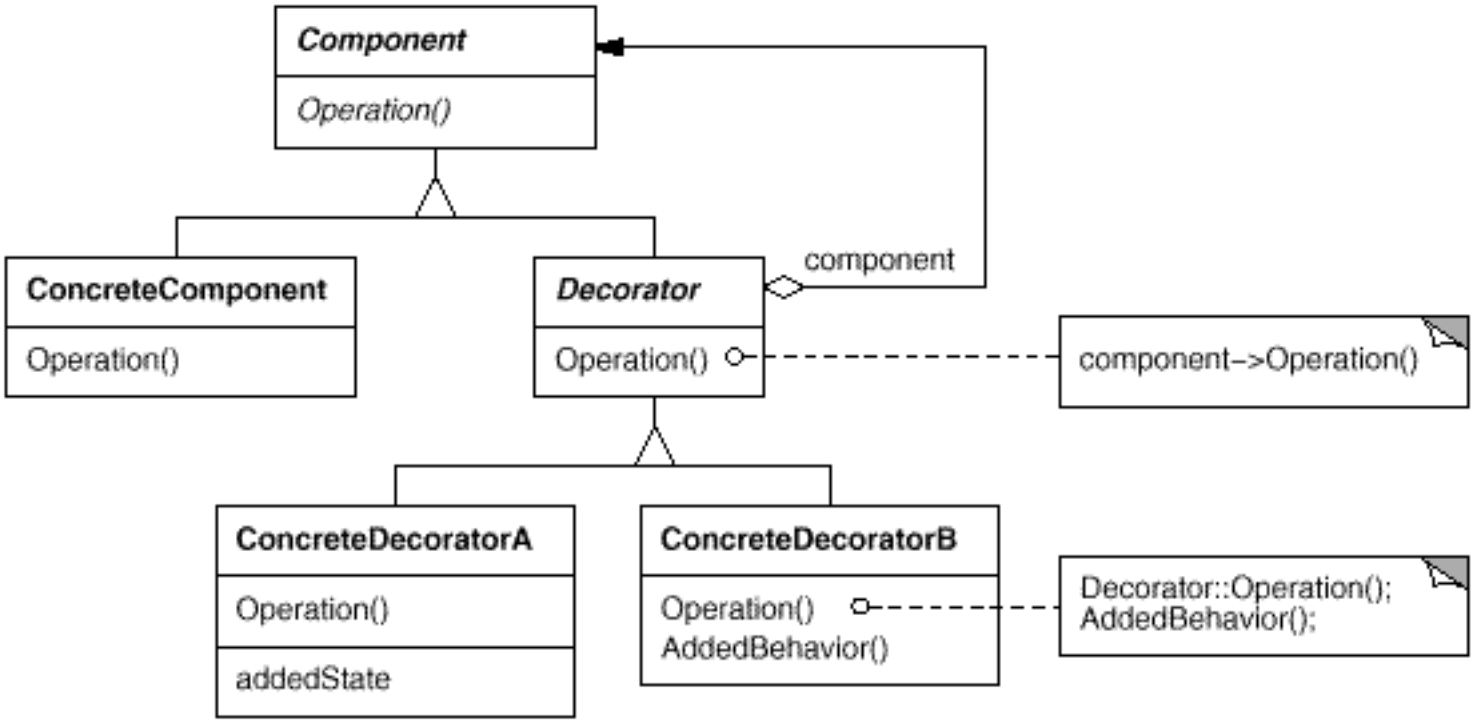
- t = new U(new B(new I(new Texte("Exemple"))));
- s = t.toHTML();

- Démonstration/ Discussion
  - Liaison dynamique ….

Démonstration, Discussion

- **Est-ce une alternative à l'héritage ?**
  - **Ajout de nouvelles fonctionnalités ?**
  - **Comportement enrichi de méthodes héritées**
- **Instances de classe au comportement dynamique**
  - **En fonction du contexte**

Decorator UML



- **Rappel du diagramme**

Décorateur et femtoContainer

Le patron Décorateur + conteneur de beans

- Les décorateurs pourraient être injectés selon une configuration

- Le décorateur « HTML » revu pour femtoContainer

  1) Ajout du constructeur par défaut et des mutateurs

  2) Création du fichier de configuration

  3) Démonstration, cf. femtoContainer.jar
     (paquetage decorator)

TexteDécoré devient, modifications mineures

```
public abstract class TexteDecore implements TexteI{
  private TexteI unTexte; // ? le texte à décorer

  public TexteDecore(){}
  public void setUnTexte(TexteI unTexte){
      this.unTexte = unTexte;
    }

  public TexteDecore(TexteI unTexte){
    this.unTexte = unTexte;
  }
  public String toHTML(){
    return unTexte.toHTML(); // ? le texte décoré
  }
}
```

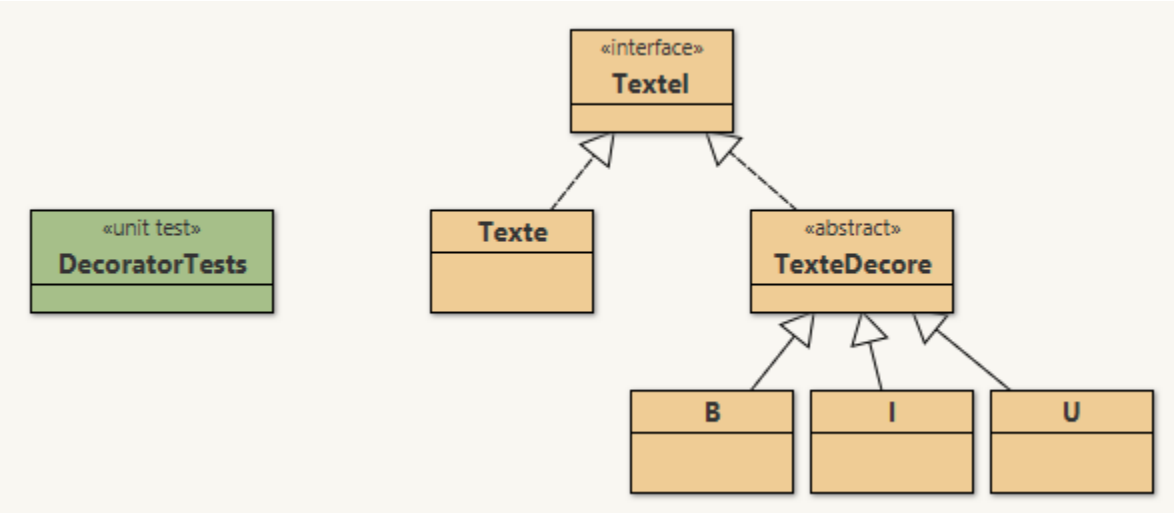
Le test en quelques lignes

```
public void testAvecInjection() throws Exception{
    ApplicationContext ctx =Factory.createApplicationContext("./decorator/README.TXT");

    TexteI texteDecore = ctx.getBean("texteDecore");
    System.out.println("texteDecore: " + texteDecore.toHTML());

    assertEquals("<U><B><I>Exemple</I></B></U>",texteDecore.toHTML());

}

     public void testSansInjection() throws Exception{
       TexteI texte = new Texte("Exemple");
       TexteI texteDecore = new U(new B(new I( texte)));
       assertEquals("<U><B><I>Exemple</I></B></U>",texteDecore.toHTML());
     }
}
```

Le fichier de configuration

```
bean.id.1=texte
texte.class=decorator.Texte
texte.property.1=texte
texte.property.1.param.1=Exemple

bean.id.2=i
i.class=decorator.I
i.property.1=unTexte
i.property.1.param.1=texte

bean.id.3=b
b.class=decorator.B
b.property.1=unTexte
b.property.1.param.1=i

bean.id.4=u
u.class=decorator.U
u.property.1=unTexte
u.property.1.param.1=b

bean.id.5=texteDecore
texteDecore.class=decorator.U
texteDecore.property.1=unTexte
texteDecore.property.1.param.1=b
```

Démonstration



- **Deux nouvelles décorations avec femtoContainer …**

Un autre exemple extrait de Head first

- inspiré de http://jfod.cnam.fr/NFP121/Chapter03_Head_First.pdf

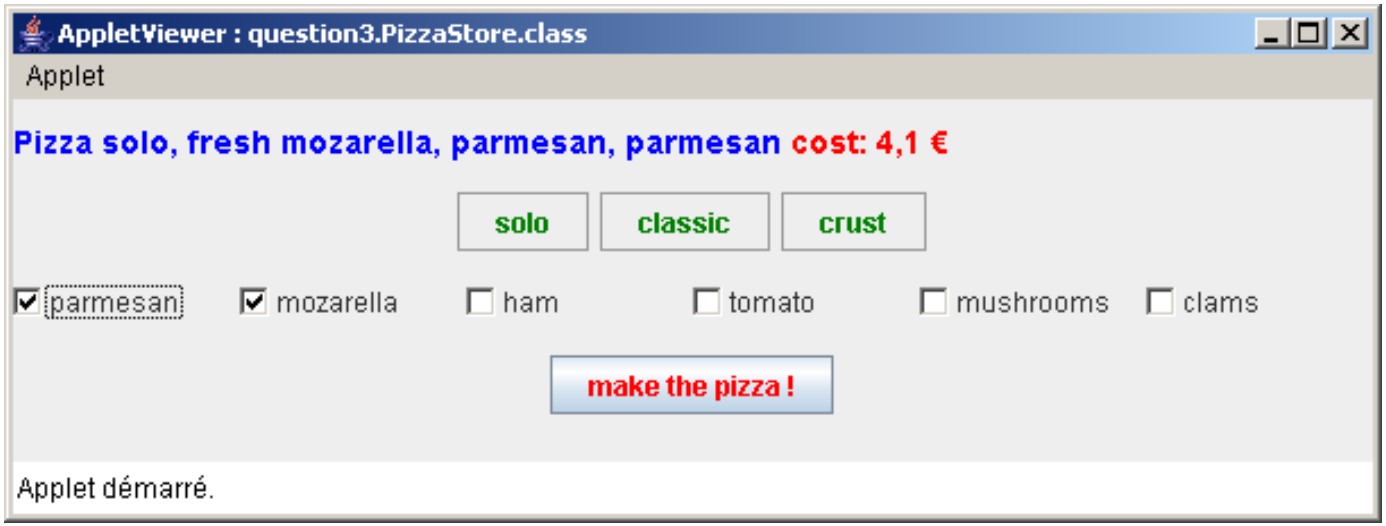
- **Confection d 'une Pizza à la carte**
  - 3 types de pâte
  - 12 ingrédients différents, (dont on peut doubler ou plus la quantité)
  - si en moyenne 5 ingrédients, soit 792* combinaisons !

  - ? Confection comme décoration ?

  - Une description de la pizza commandée et son prix

* n parmi k, n! / k!(n-k)!

Un dernier exemple avec une IHM



>appletviewer http://jfod.cnam.fr/progAvancee/tp8/tp8.html

3 types de pâte

- **Pâte solo, (très fine…)**
- **Pâte Classic**
- **Pâte GenerousCrust©**

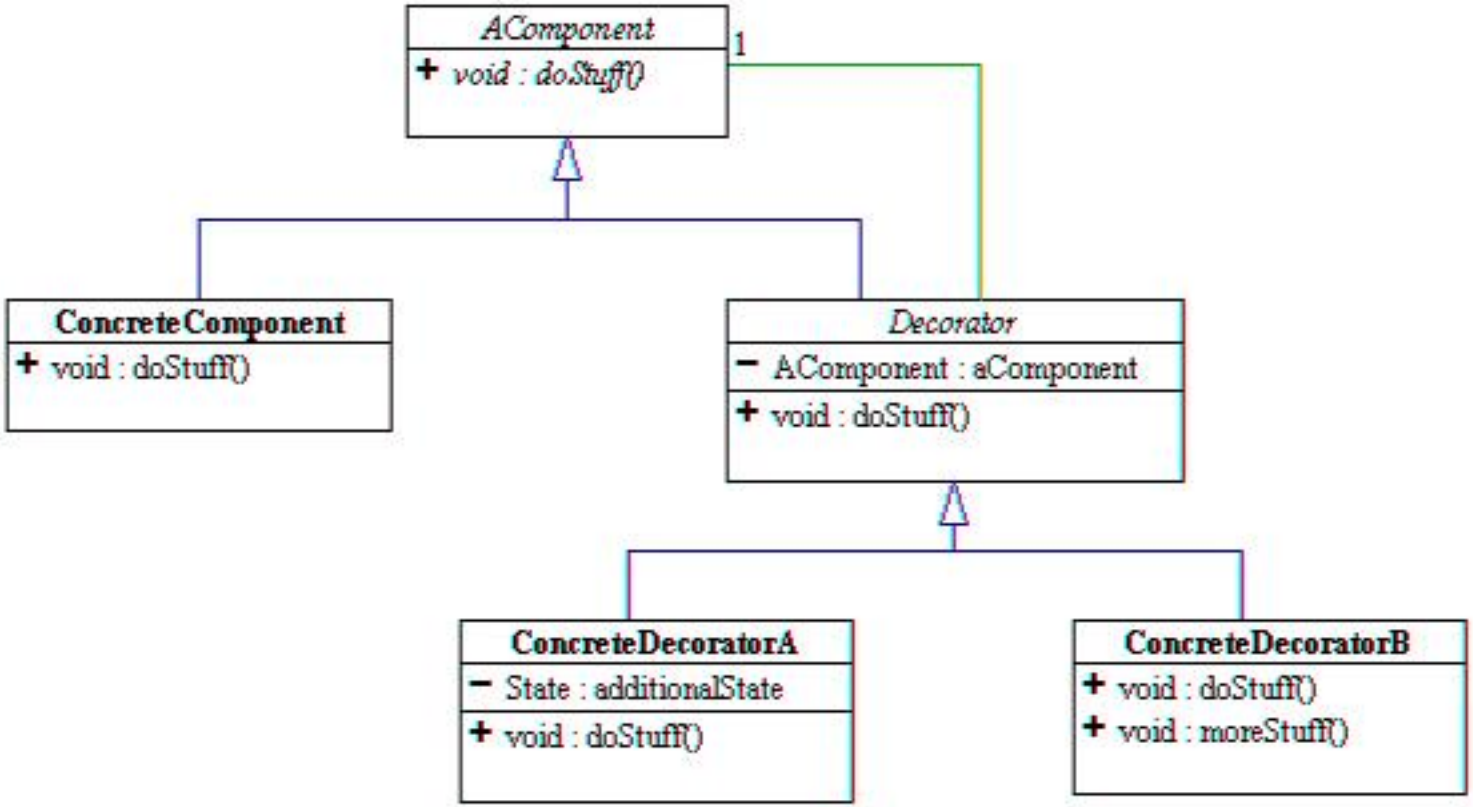
12 ingrédients différents

- **Mozarella, parmesan, Ham, Tomato, Mushrooms, diced onion, etc...**

Quelles choix de conception ?

- **3 classes de Pâtes ?**
- **Une valeur entière par ingrédient ?**
- **Un patron : le bien nommé Décorateur**

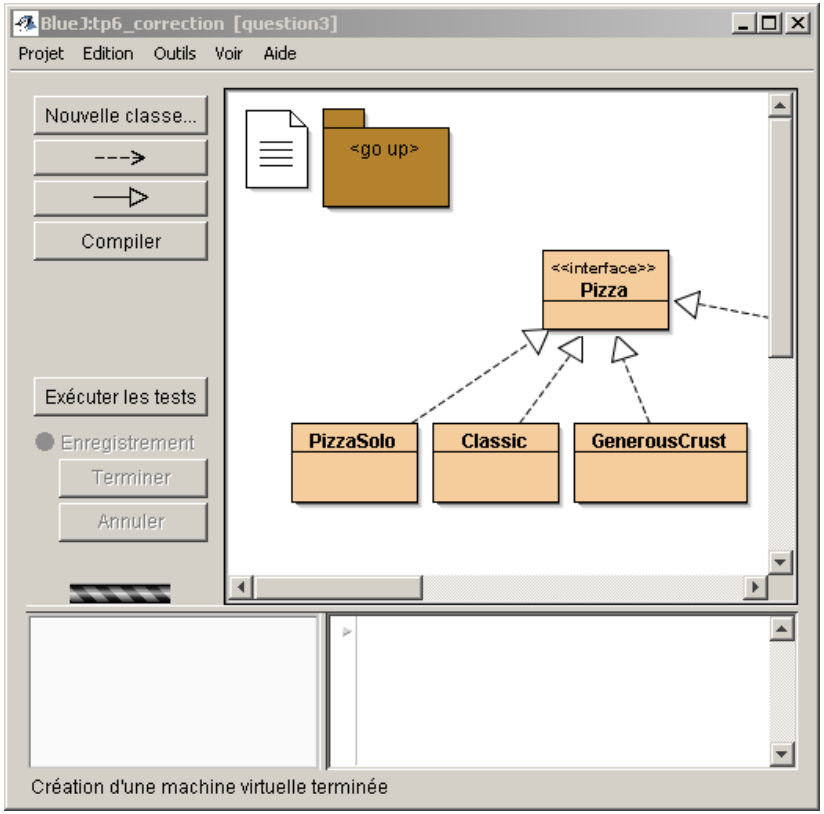
Le décorateur de pizza



- AComponent --> une interface Pizza
- ConcreteComponent --> les différentes pâtes

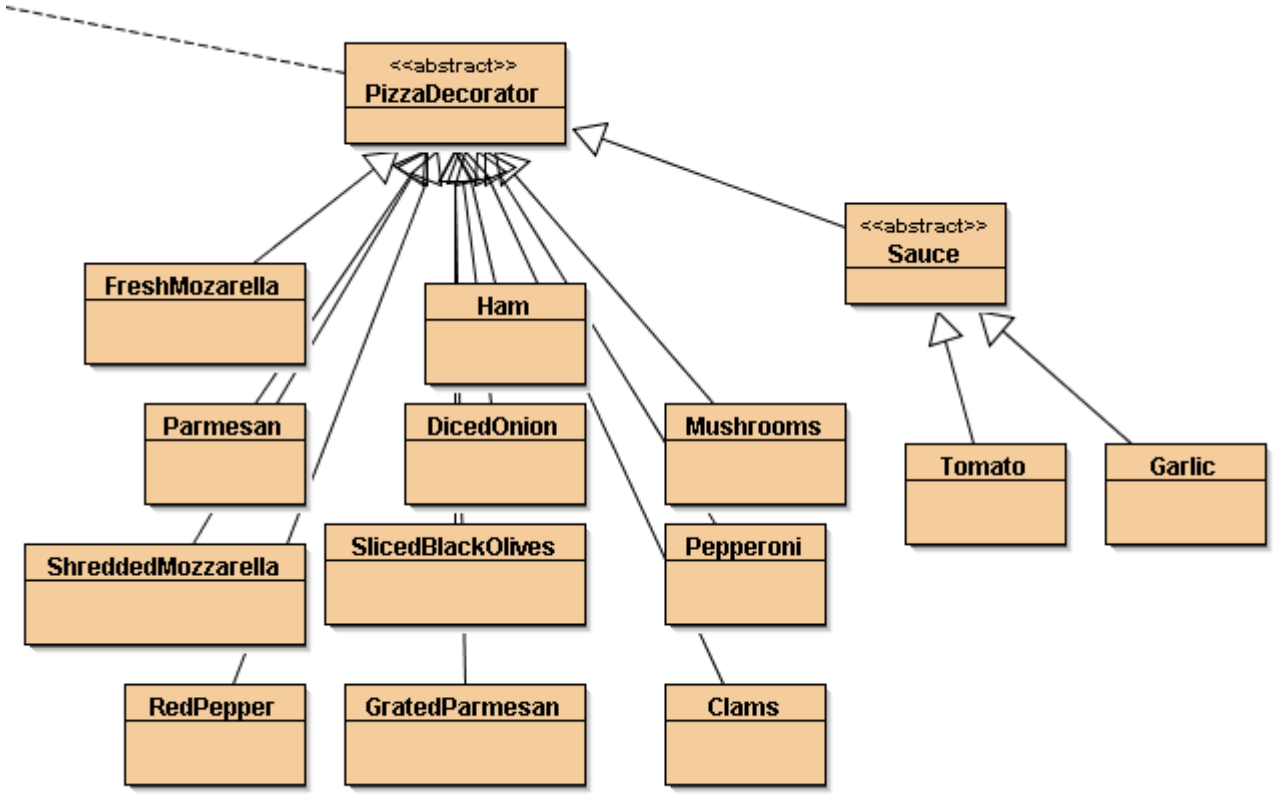
- Decorator l'ingrédient, la décoration
- ConcreteDecorator Parmesan, Mozarella, ...

3 types de pâte




```
public interface Pizza{
   abstract public String getDescription();
   abstract public double cost();
}
```

Les ingrédients



- !
- **Discussion : comment faire sans le patron décorateur ?**

PizzaDecorator

```
public abstract class PizzaDecorator implements Pizza{
  protected Pizza pizza;
  public PizzaDecorator(Pizza pizza){
     this.pizza = pizza;
  }
  public abstract String getDescription();
  public abstract double cost();
}
```

Ham & Parmesan

```
public class Ham extends PizzaDecorator{
  public Ham(Pizza p){super(p);}
  public String getDescription(){
    return pizza.getDescription() + ", ham";
  }
  public double cost(){return pizza.cost() + 1.50;}
}


public class Parmesan extends PizzaDecorator{
  public Ham(Pizza p){super(p);}
  public String getDescription(){
    return pizza.getDescription() + ", parmesan";
  }
  public double cost(){return pizza.cost() + 0.75;}
}
```

Pizza Solo + Mozarella + quel coût ?

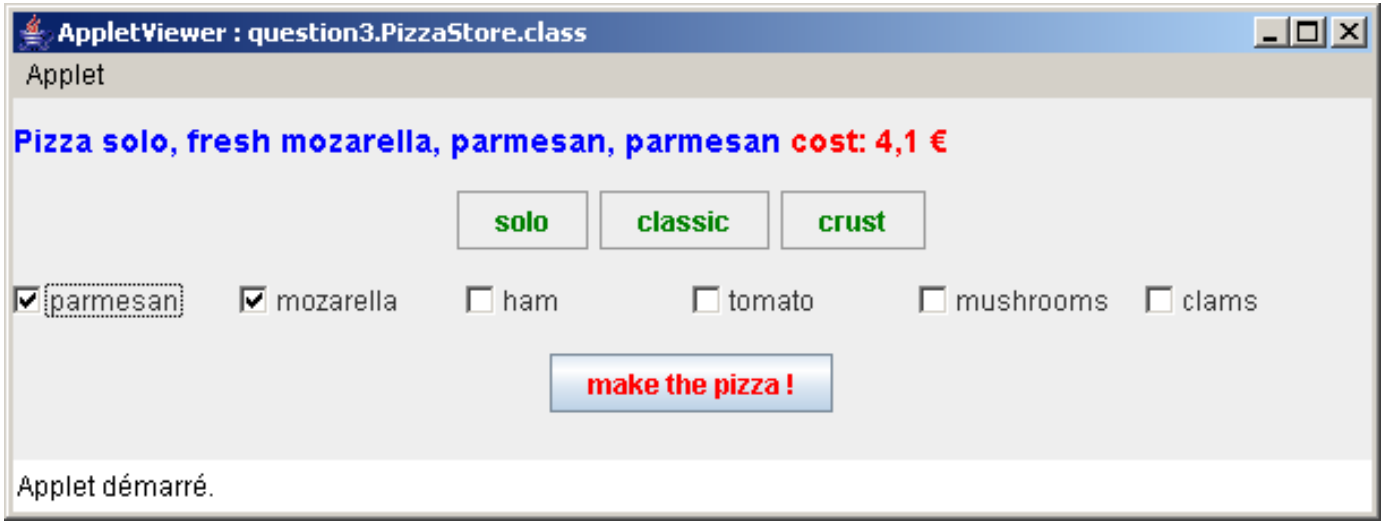
```
double coût = new Parmesan(
                new FreshMozarella(
                  new PizzaSolo())).cost();
assert coût == 5.8;
```

Une pizza aux 2 fromages

```
Pizza p=new Mozarella(new Mozarella(new Parmesan(new PizzaSolo())));
```

Magique ou liaison dynamique ???

L 'IHM du pizzaiolo



- **Pizza p; // donnée d 'instance de l 'IHM**
- **choix de la pâte, ici solo**

```
boutonSolo.addActionListener(
  new ActionListener(){
    public void actionPerformed(ActionEvent ae){
      pizza = new PizzaSolo();
      validerLesDécorations();
    }
  });
```

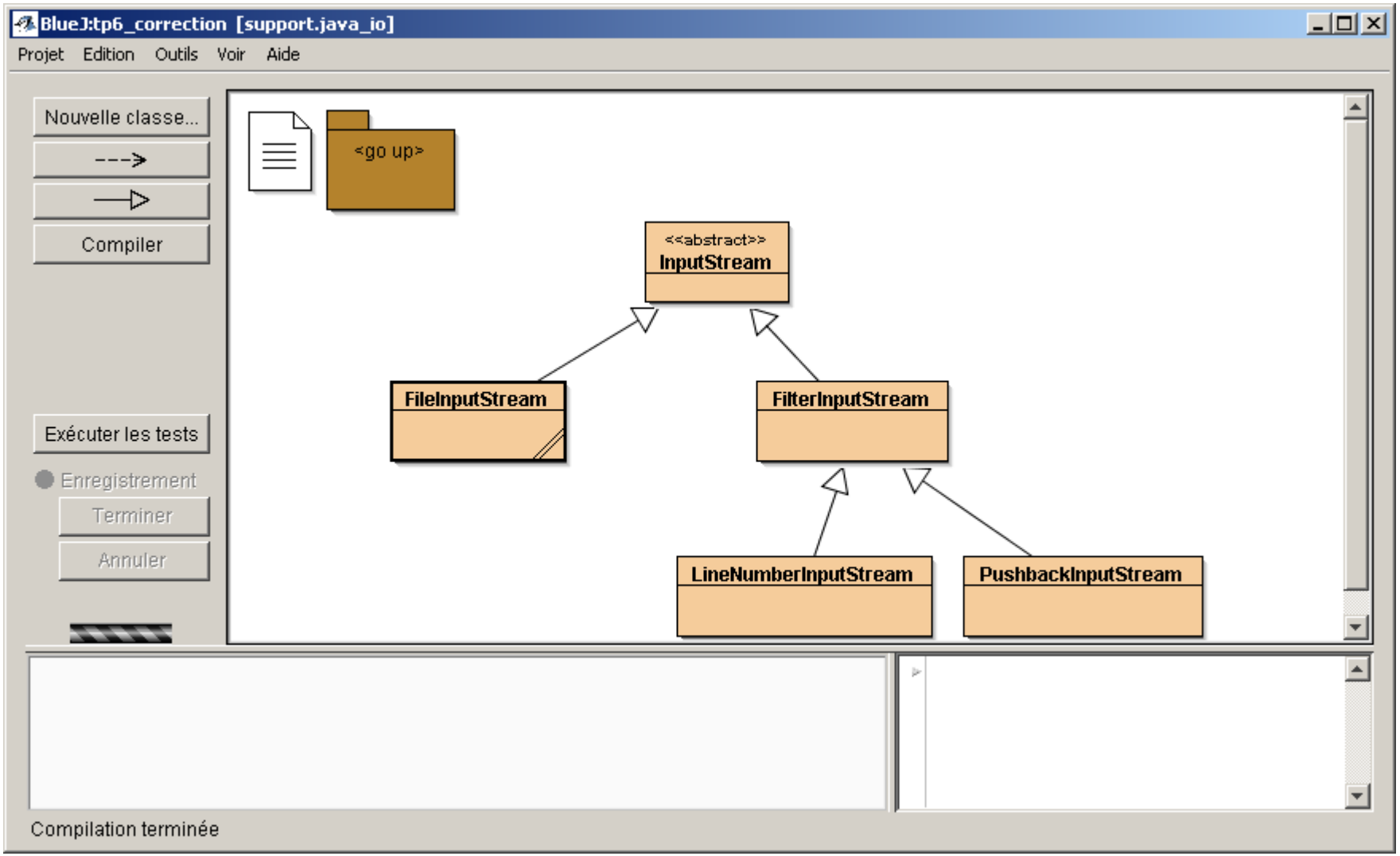
L 'IHM : les ingrédients ici Ham*2


```
ham.addItemListener(new ItemListener(){
    public void itemStateChanged(ItemEvent ie){
      if(ie.getStateChange()==ItemEvent.SELECTED)
         pizza = new Ham(pizza);
      afficherLaPizzaEtSonCoût();
    }
});
```

L'applette est ici, mais la livraison n'est pas garantie

appletviewer http://jfod.cnam.fr/progAvancee/tp8/tp8.html

java.io

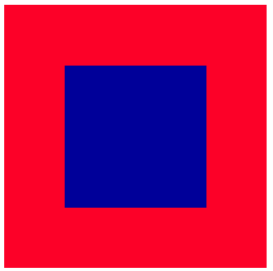


- **Le Décorateur est bien là**

Quelques déclarations

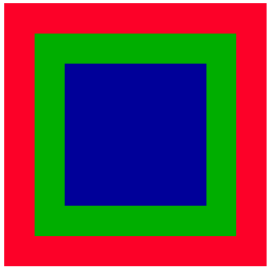
Un fichier standard

- InputStream a = new FileInputStream("fichier.txt");
- a.read();



Un fichier décoré

- InputStream a = new PushbackInputStream (new FileInputStream("fichier.txt"));
- a.read();

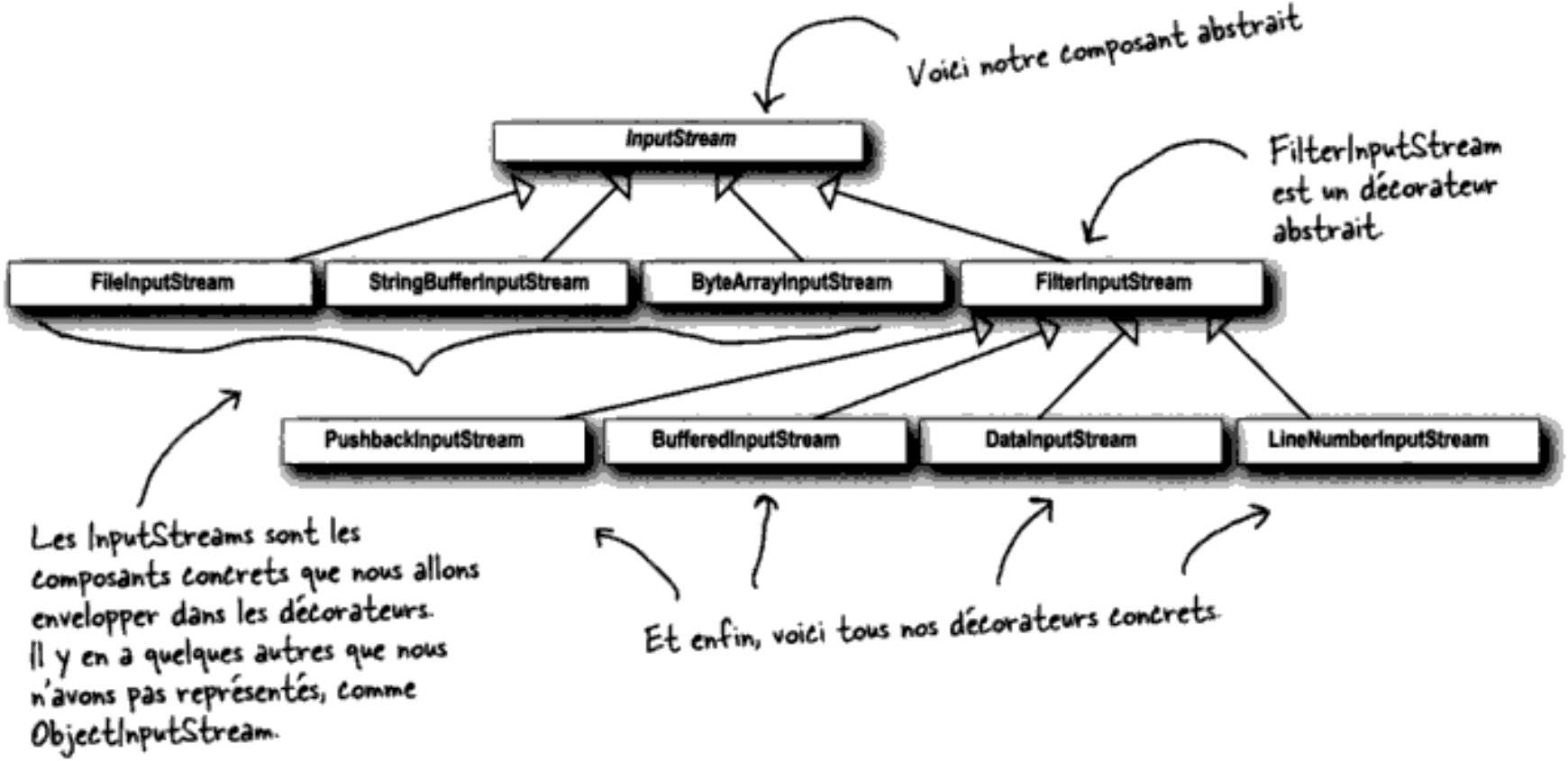


Un autre fichier décoré

- InputStream a = new LineNumberInputStream(
- new PushbackInputStream(new FileInputStream("fichier.txt"))
- a.read();

Extrait de java : tête la première

Décoration des classes de java.io



La suite …

- **De nouveaux décorateurs**
  - **Incursion douce dans la programmation par contrat**
    - **Pré et post assertion**
    - **Invariant de classe**
    - **Fonction d'abstraction**
    - **Héritage de pré et post assertions**
    - **Avec femtoContainer ? : une bonne idée**

Programmation par contrats ? (en quelques lignes…)

- **Propriétés, assertions comme**
  - **Pré-post conditions**
    - **Propriété vraie à l'appel (pré) et garantie au retour pour l'appelant (post)**
  - **Invariant de classe**
    - **Une propriété vraie avant et après l'appel de toute méthode d'instance et à la sortie du ou des constructeurs**
- **Assertions :**
  - **que l'on pourrait installer dans le source java**
    - **Par exemple à l'aide de l'instruction assert**

Exemple : Une décoration de source Java

require
précondition

Example 2.2 precondition of a method

```
public Object remove() {
  /** require !empty(); **/
  ...
}
```

Example 2.3 postcondition of a method using `Old` and `changeonly`

```
public Object remove() {
    ...
    return o;
    /** ensure changeonly{count,out};
          Old.contains(Result);
          Result.equals(Old.store[Old.out]); **/
}
```

ensure
postcondition

Example 2.4 class invariant

```
public class Buffer {
  ...
  /** invariant 0 <= count && count <= capacity(); **/
}
```

invariant

- **JASS http://csd.informatik.uni-oldenburg.de/~jass/**
- **exemples extraits de http://jfod.cnam.fr/NFP121/jass_presentation.pdf**

Precondition, postcondition et invariant, ici sous forme de commentaires, Jass les extrait et produit un source java instrumenté

Décoration de sources Java, Google cofoja

```
@Invariant("size() >= 0")
interface Stack<T> {
  public int size();

  @Requires("size() >= 1")
  public T peek();

  @Requires("size() >= 1")
  @Ensures({
    "size() == old(size()) - 1",
    "result == old(peek())"
  })
  public T pop();

  @Ensures({
    "size() == old(size()) + 1",
    "peek() == old(obj)"
  })
  public void push(T obj);
}
```

- **@Invariant**
  - Invariant de classe
- **@Requires**
  - Pré assertion
- **@Ensures**
  - Post assertion

- **Cofoja (Contracts for java), JML, JASS-modern,…**
  - https://github.com/nhatminhle/cofoja
  - Ici à l'aide d'annotations, génération de sources instrumentés et parfois de tests unitaires
  - Voir aussi OCL / UML
    - Object Constraint Language

Programmation par contrats

- **Design by contracts**
  - **Bertrand Meyer 1992**
    - **https://archive.eiffel.com/doc/manuals/technology/contract/**

The benefits of Design by Contract include the following:


A better understanding of the object-oriented method and, more generally, of software construction.


A systematic approach to building bug-free object-oriented systems.


An effective framework for debugging, testing and, more generally, quality assurance.


A method for documenting software components.


Better understanding and control of the inheritance mechanism.


A technique for dealing with abnormal cases, leading to a safe and effective language construct for exception handling.

- **B.Liskov & J. Guttag Program Development in Java**
  - **Chapitres 6 et 7**

- Vers une proposition de Décoration comme documentation ou l'inverse

Design by contracts

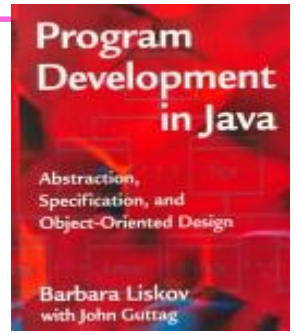
Bertand Meyer 1992

Le langage Eiffel

```
class ACCOUNT create
   make
feature
    ... Attributes as before:
        balance , minimum_balance , owner , open ...
   deposit (sum: INTEGER) is
           -- Deposit sum into the account.
      require
        sum >= 0
      do
        add (sum)
      ensure
        balance = old balance + sum
     end
   withdraw (sum: INTEGER) is
           -- Withdraw sum from the account.
      require
        sum >= 0
        sum <= balance - minimum_balance
      do
        add (-sum)
      ensure
        balance = old balance - sum
     end
   may_withdraw ... -- As before
feature {NONE}
   add ... -- As before
   make (initial: INTEGER) is
           -- Initialize account with balance initial.
      require
        initial >= minimum_balance
      do
        balance := initial
      end
invariant
   balance >= minimum_balance
end -- class ACCOUNT
```

- https://archive.eiffel.com/doc/online/eiffel50/intro/language/invitation-07.html

B. Liskov & J. Guttag : repOK ou invariant, af :



- **Documentation d'une classe ?**
  - **B. Liskov & Guttag proposent deux notions**
    - **Fonction d'abstraction af()**

    et
    - **Invariant de représentation repOk**
  - **Comme commentaires ou bien en méthodes de la classe**
  - **Page 99 du livre est en page suivante …**

Aids to Understanding Implementations

In this section, we discuss two pieces of information, the abstraction function and the representation invariant, that are particularly useful in understanding an implementation of a data abstraction.

af

The *abstraction function* captures the designer's intent in choosing a particular representation. It is the first thing you decide on when inventing the rep: what instance variables to use and how they relate to the abstract object they are intended to represent. The abstraction function simply describes this decision.

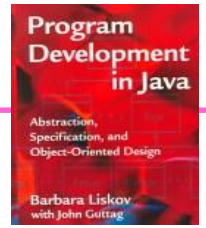
repOk

The *rep invariant* is invented as you investigate how to implement the constructors and methods. It captures the common assumptions on which these implementations are based; in doing so, it allows you to consider the implementation of each operation in isolation of the others.

The abstraction function and rep invariant together provide valuable documentation, both to the original implementor and to others who read the code. They capture the reason why the code is the way it is: for example, why the implementation of `choose` can return the $zero^{th}$ element of `els` (since the elements of `els` represent the elements of the set), or why `size` can simply return the size of `els` (because there are no duplicates in `els`).

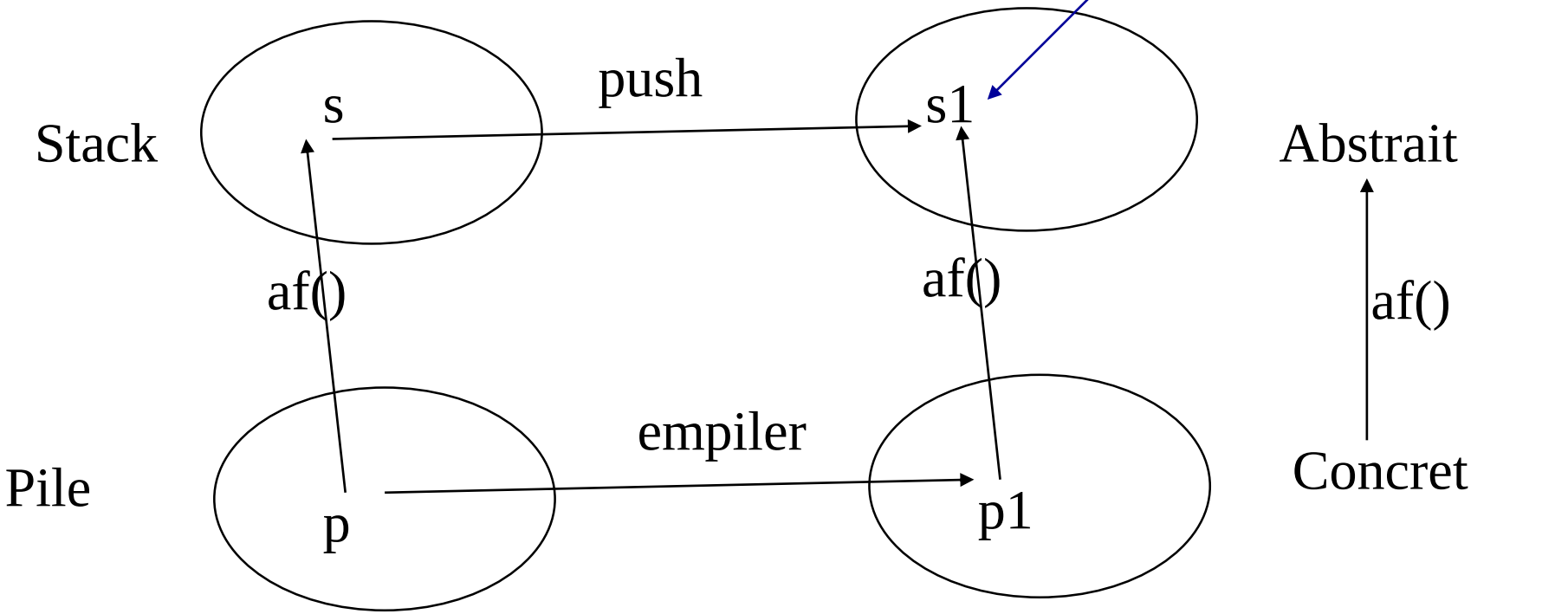
Because they are so useful, both the abstraction function and rep invariant should be included as comments in the code. This section describes how to define them and also how to provide them as methods.

repOk == invariant, af, pre, post



- **repOk**
  - **Invariant de représentation, invariant de classe**
- **af**
  - **Fonction d'abstraction**

*Le diagramme commute, s1 désigne la même instance :**c'est correct***

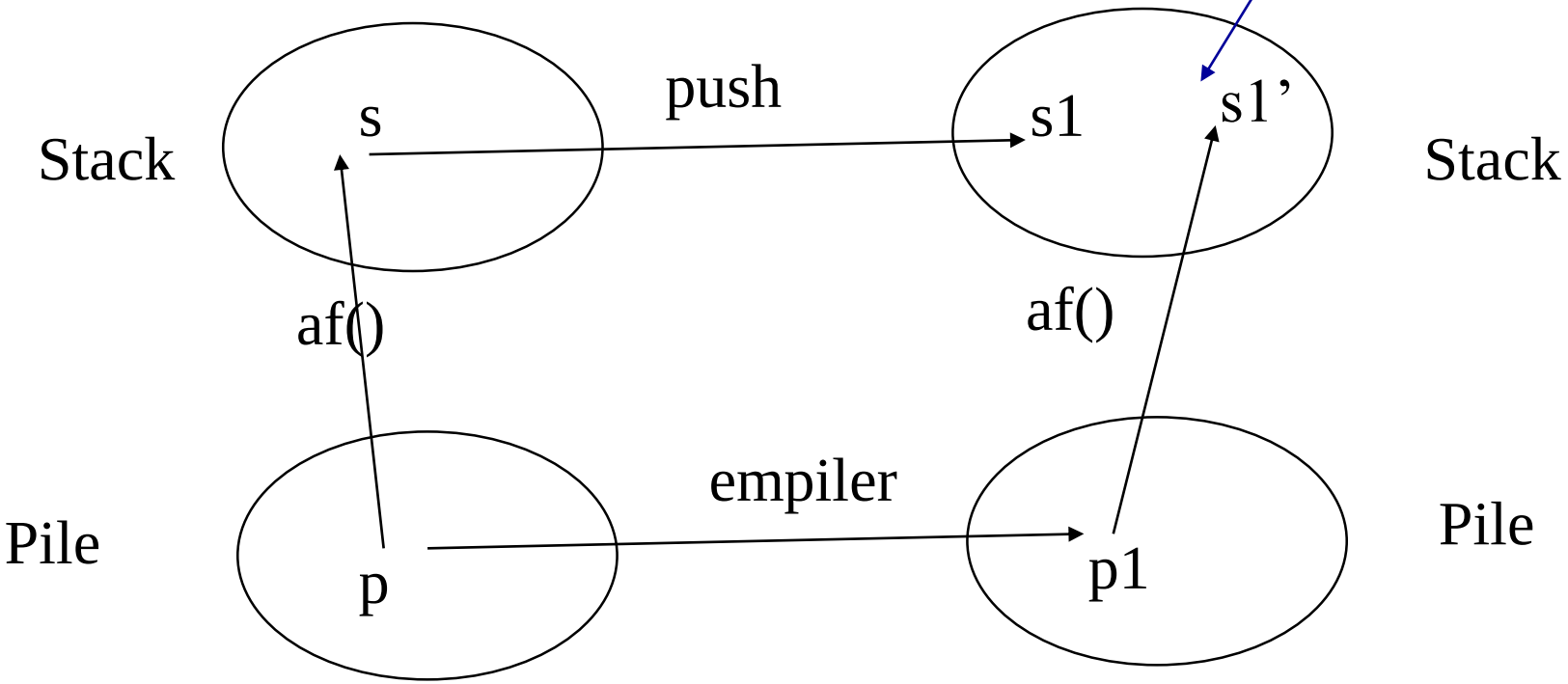


s1 == p.af().push(44) et s1== p.empiler(44).af()

af incorrect

- **af**
  - **Fonction d'abstraction**

Le diagramme ne commute pas,
C'est incorrect



Nous pouvons vérifier cette assertion par

assert p.af().push(44).equals(p.empiler(44).af());

repOk, af, Pre Post assertions

- **pre, post**
  - **pre_assertion vrai à l'appel de la méthode,**
  - **post_assertion vrai après l'exécution de la méthode,**
    - **Pre, post : un contrat pour l'appelant**

- **En prédéfini nous avons**
  - **assert Expression Booléenne : " un commentaire " ;**
    - **Une option du compilateur java, javac -ea**

Un résumé

Invariant, pre et post assertion, héritage

- Un invariant de classe doit être vrai avant et après l'appel de chaque méthode de la classe et à la sortie du constructeur (repOk)
- La pré assertion précise le contrat que doit respecter l'appelant
- La post assertion garantit un contrat en retour pour l'appelant
  - Framework existants
    - Cf. cofoja, JML, Contract4j, Jass-Modern, UML + OCL …
      - Par annotations @Requires,@Ensures, @Invariant
      - http://www.ifi.uzh.ch/seal/teaching/courses/archive/hs10-1/ase/07-jmldbc.pdf

pré et post assertions comme décorations

- **Exemple : ajout d'un élément dans une liste**

```
assert  element != null ;

    ajouter(element);

assert( tailleAvant + 1 == tailleAprès);
```

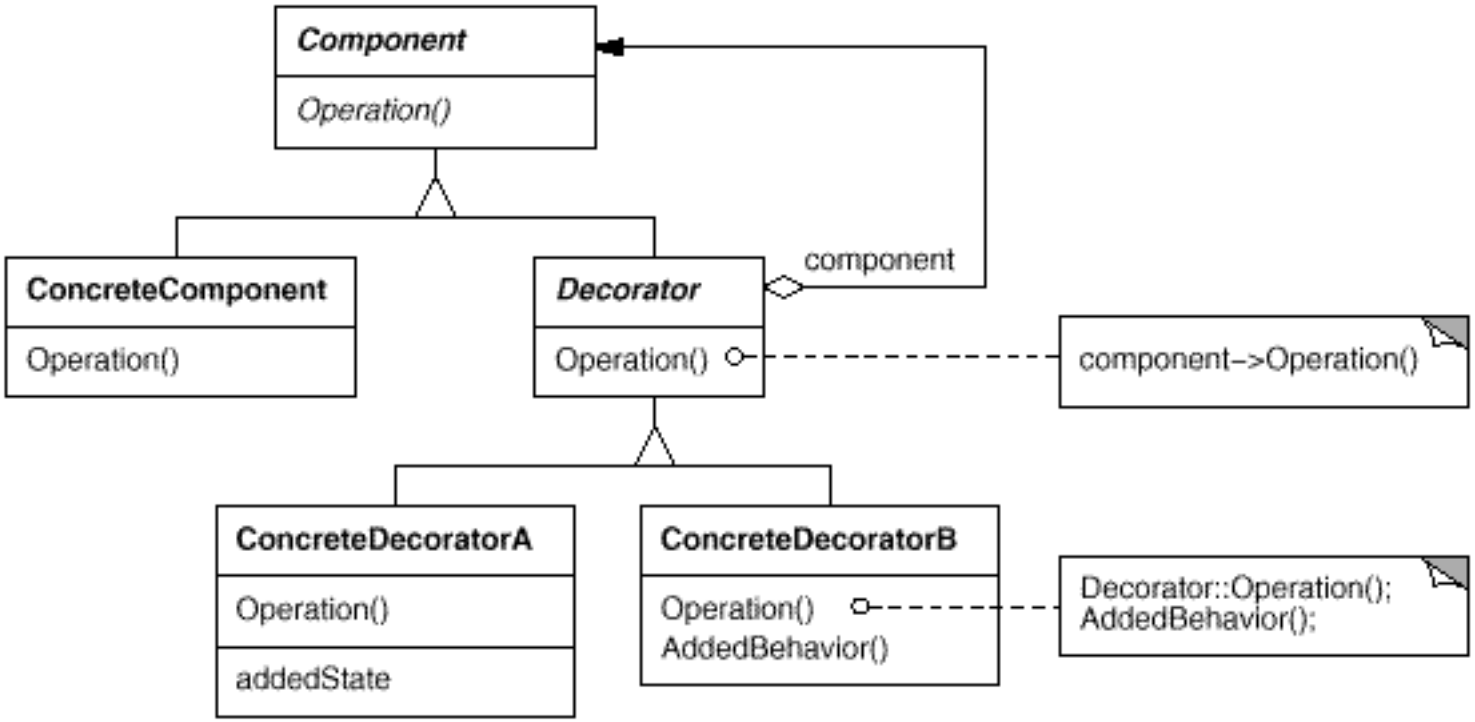
- **Serait-ce une décoration de la méthode ajouter ?**

pré et post assertions, de simples décorations…

- **En java, ajout d'un élément dans une liste**

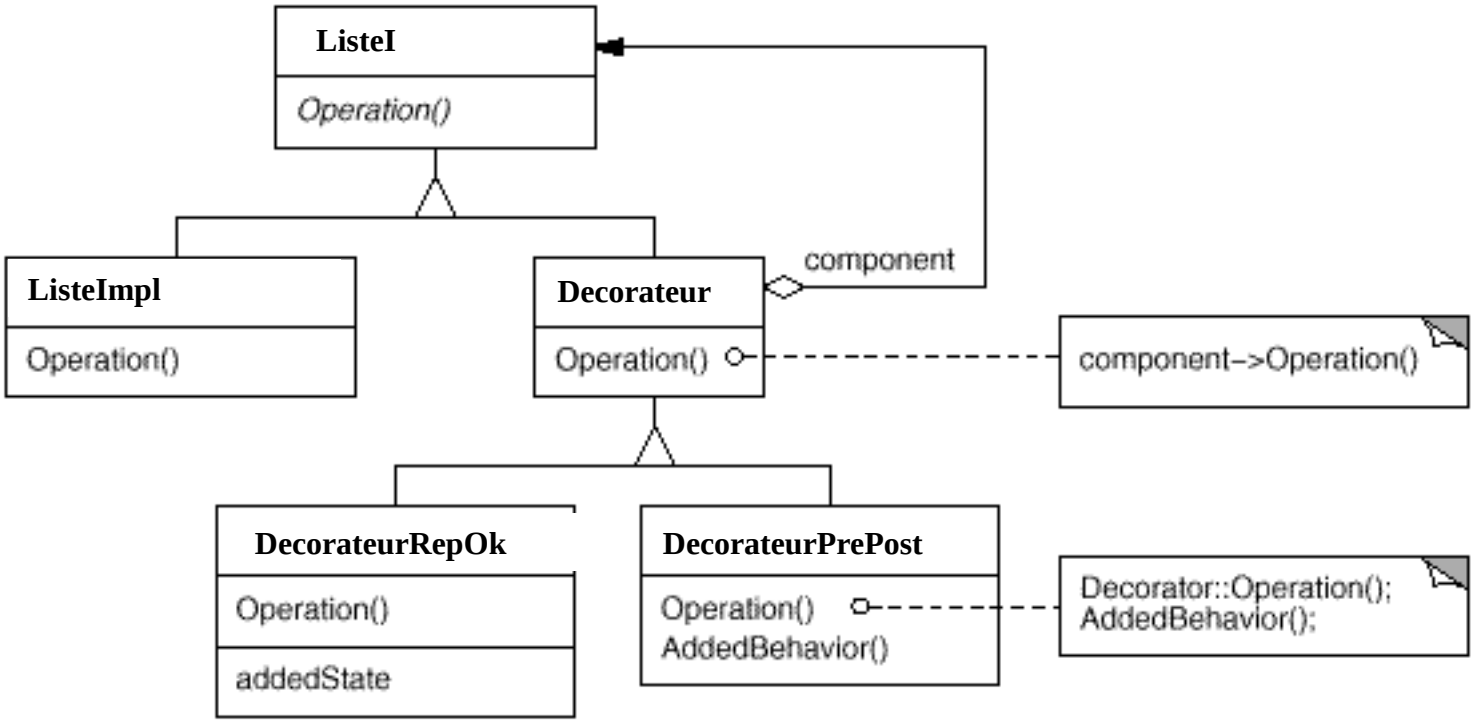
```
protected boolean pre_assertion( int element){
   return element != null;
}
protected boolean post_assertion(){
   return tailleAvant +1 == liste.taille();
}

public void ajouter(int element){
   InitialisationDesVariables(); // tailleAvant, etc …
   assert pre_assertion(element);
   liste.add(element); // ← le décoré
   assert post_assertion();
}
```

Usage du patron Décorateur



- **La liste et ses décorateurs**

Usage du patron Décorateur



DecorateurPrePost

- La liste décorée de ses pré et post assertions
  - Ici Operation ←→ ajouter un élément à cette liste

DecorateurRepOk

- Vérification de l'invariant

Pre, Post en tant que décorateur…

```
public class DecorateurPrePost extends Decorateur{

  protected boolean pre_assertion( int element){
    return element != null;
  }
  protected boolean post_assertion(){
    return (tailleAvant +1 == super.taille())
  }

  public void ajouter(int element){
      InitialiserDesVariables();
      assert pre_assertion(element);

      super.ajouter(element); // ← le décoré

      assert post_assertion();
    }
}
```

Pre, Post avec exception…

```
public class DecorateurPrePost extends Decorateur{

  protected boolean pre_assertion( int element){
    return element != null;
  }
  protected boolean post_assertion(Exception e){
    return (e==null && tailleAvant +1 == super.taille()) ||
           (e!=null && e instanceof ListePleineException && tailleAvant==super.taille());
  }

  public void ajouter(int element){
     InitialiserDesVariables();
     assert pre_assertion(element);
     Exception cause =null;
     try{
       super.ajouter(element);// la liste est bornée …
                              // alors une exception est possible
     }catch(Exception e){
       cause = e;
       throw e;  // l'exception est propagée
     }finally{
        assert post_assertion(cause);
     }
  }
```

Invariant de classe, repOk==invariant

- **Exemple : la liste suite**

```
 private E[] tableau = ……

Liste(){ …}
 invariant( tableau != null)
```

```
invariant( tableau != null)
   ajouter(element);
invariant( tableau != null)
```

Vrai à la sortie du constructeur
à l'entrée et la sortie de chaque méthode

- **Serait-ce aussi une décoration du constructeur et de la méthode ajouter ?**

Invariant devient lui aussi un décorateur concret

```
public class DecorateurRepOk extends Decorateur{

  public DecorateurRepOk(ListeI l){
    super(l);
    assert super.repOk();
  }

  public void ajouter(int element){
    assert super.repOk()
    super.ajouter(element);  // ← le décoré
    assert super.repOk();
  }
```

Invariant, repOk, doit être vrai à la sortie du constructeur, avant et après l'appel de chaque méthode

Deux exemples

1) La classe Nombre

- Le nombre est borné entre min et max

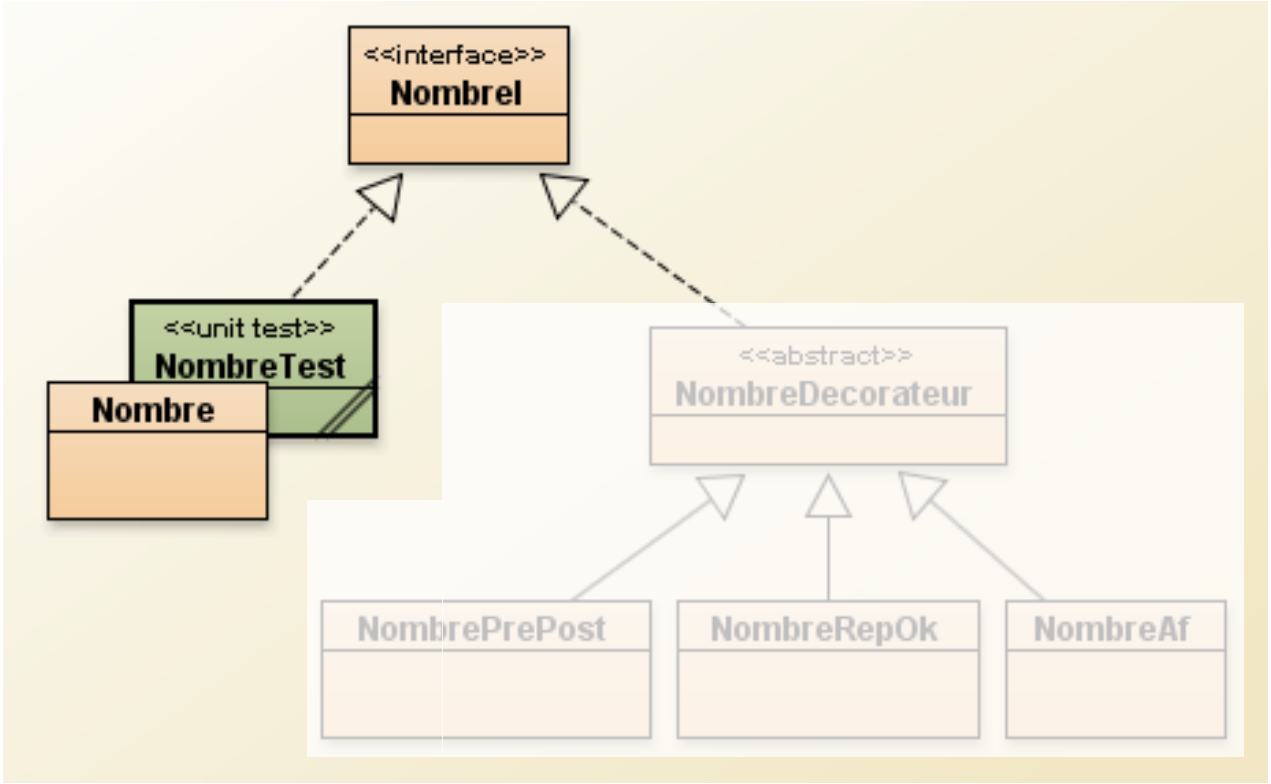
2) La classe Ensemble, sa sous-classe EnsembleOrdonné

- Un ensemble d'entiers

- Ainsi chaque classe peut être décorée
  - De pré et post assertions
  - De la vérification de l'invariant
  - De la vérification de la fonction d'abstraction
    - *« vérification » pour un jeu de tests*

- **Le projet Bluej à télécharger**
  - http://jfod.cnam.fr/NFP121/assertions_et_decorateur.jar

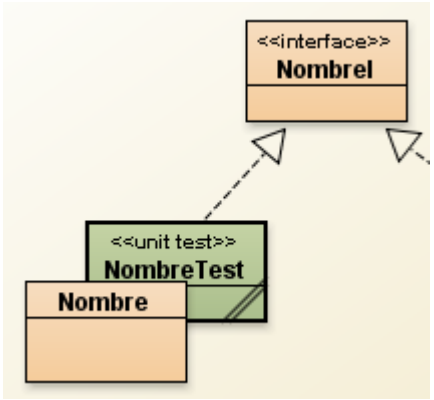
La classe Nombre



- **Une interface NombreI**
- **Une implémentation Nombre**
  - **Une classe de tests unitaires NombreTest**

NombreIl, l'interface et son ombre

```
2
3  public interface NombreI{
4
5      public void setValeur(int valeur);
6      public void inc();
7      public void dec();
8
9      public int getValeur();
10     public int getMin();
11     public int getMax();
12
13     public Object af();
14     public boolean repOk();
15 }
16
```



- **Notez la fonction d'abstraction af et l'invariant repOk**

La classe Nombre, un extrait

```
3  public class Nombre implements NombreI {
4     private int valeur;
5     private final int min;
6     private final int max;
7
8     public Nombre(int min, int max){
9        if(min>=max) throw new RuntimeException(" min >= max");
10       this.valeur=min;
11       this.min = min;
12       this.max = max;
13    }
14    public void setValeur(int valeur){
15       if((valeur<min || valeur>max)) throw new RuntimeException("valeur<min
16       this.valeur = valeur;
17    }
18    public void inc(){
19       if(valeur==max)throw new RuntimeException("inc: valeur == max");
20       this.valeur++;
```

- **Le nombre est borné [min..max]**
- **Une exception est levée si les bornes sont dépassées**

La classe Nombre : repOk et af

```
public int getValeur(){return this.valeur;}
public int getMin(){ return this.min;}
public int getMax(){ return this.max;}

public Integer af(){
  return new Integer(valeur);
}
public boolean repOk(){
  return this.valeur>=min && this.valeur<=max;
}
}
```

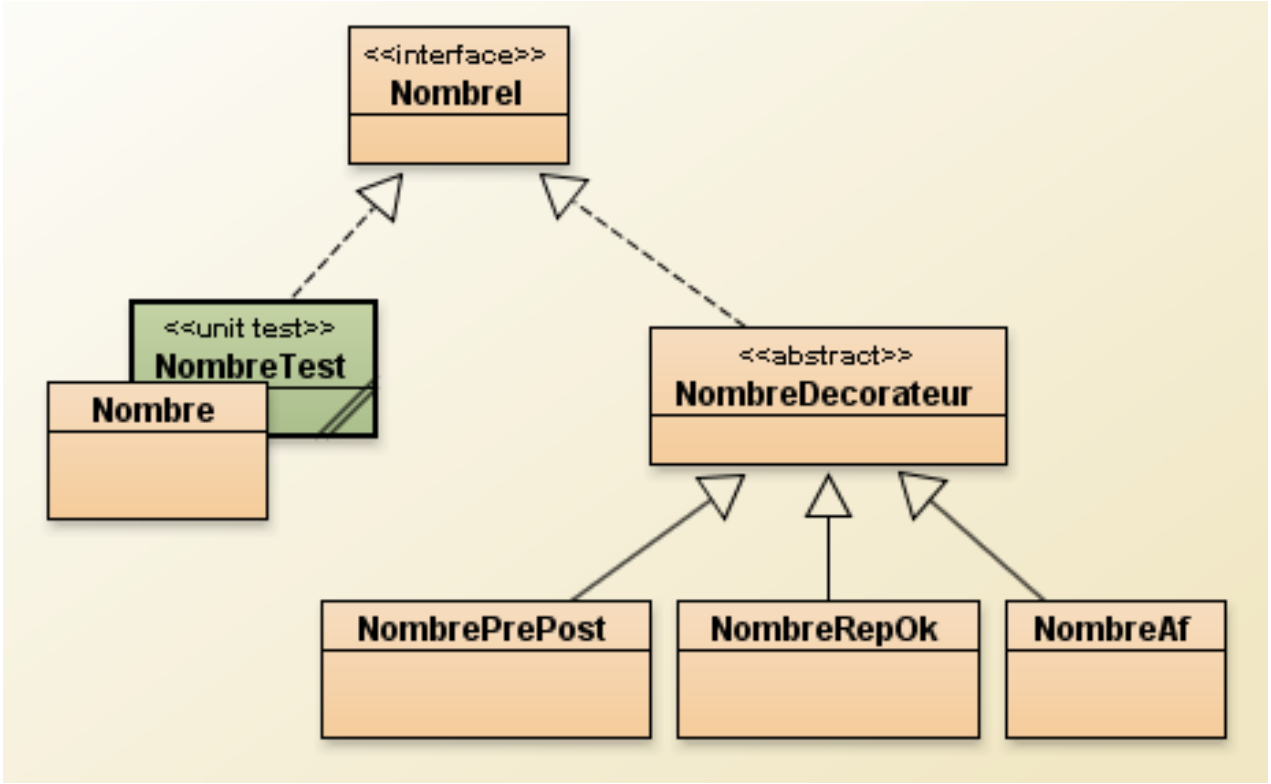
- **af : Ce nombre a ici une représentation abstraite par un Integer**
  - **Notez la Covariance utilisée pour la fonction d'abstraction**
- **repOk : invariant de classe**

La classe de test

```
14  public void testNombre(){
15      assertEquals(0, n.getMin());
16      assertEquals(4, n.getMax());
17      assertTrue(n.repOk());
18      assertEquals(0, n.getValeur());
19      assertEquals(0, ((Integer)n.af()).intValue());
20      n.inc();
21      assertEquals(1, n.getValeur());
22      n.inc();n.inc();n.inc();
23      assertEquals(4, n.getValeur());
24      try{
25          n.inc();
26          fail(" une exception est attendue ");
27      }catch(Exception e){
28          assertTrue( e instanceof RuntimeException);
29      }
30      n.dec();
31      assertEquals(3, n.getValeur());
32      n.dec();n.dec();n.dec();
```

- **Tests classiques …**
  - **Avec n = new Nombre(0,4);**

La classe Nombre et ses décorateurs



- **Depuis la classe de tests unitaires NombreTest**
  - **Vérification des décorations d'une instance de classe Nombre …**
  - **NombreI n = new NombrePrePost(new NombreRepOk(new NombreAf(new Nombre(0,10)))**

Decorateur NombrePrePost : la méthode inc

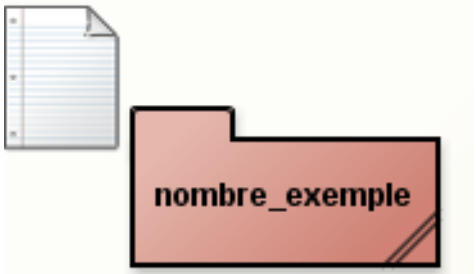
```
protected boolean pre_inc(){
  return true;
}
protected boolean post_inc(Exception cause){
  return (cause == null && super.getValeur()==valeurAvant+1) ||
         (cause != null && cause instanceof RuntimeException &&
          super.getValeur()==valeurAvant);
}
public void inc(){
  valeurAvant = super.getValeur();
  assert pre_inc(): " pre assertion inc ???";
  Exception cause = null;
  try{
    super.inc();
  }catch(Exception e){
    cause = e;
    throw e;
  }finally{
    assert post_inc(cause):" post assertion dec ???";
  }
}
```

- **Un ajout d'un décorateur : la classe de tests unitaires inchangée ou presque**
  - **n = new DecorateurPrePost(new Nombre(0,4));**

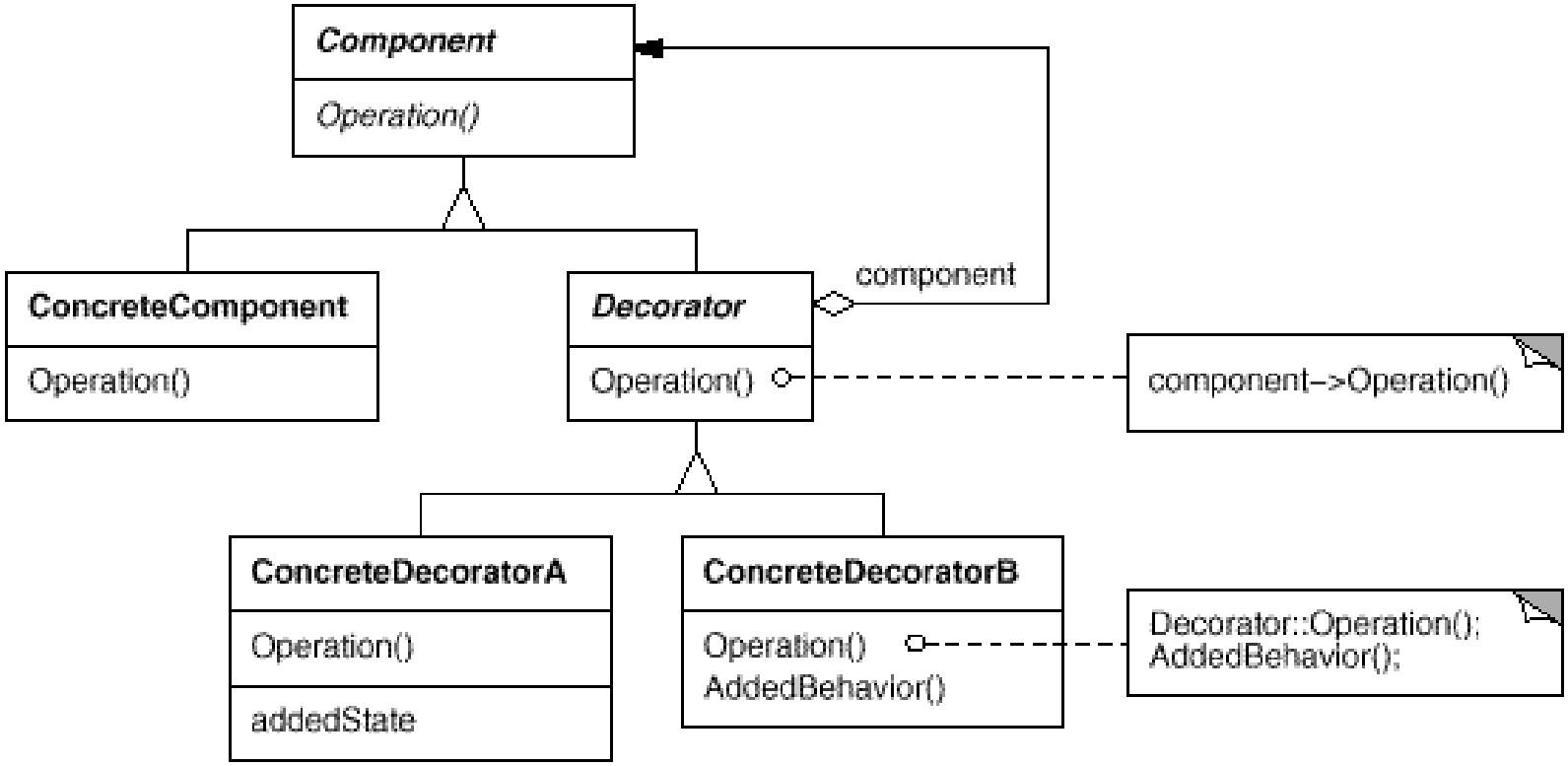
Démonstration collaborative

- **A vos bluej**

- **Le projet Bluej à télécharger**
  - **http://jfod.cnam.fr/NFP121/assertions_et_decorateur.jar**
  - **Il reste des assertions à faire …**
  - **Rendez vous classe NombrePrePost package nombre_exemple**



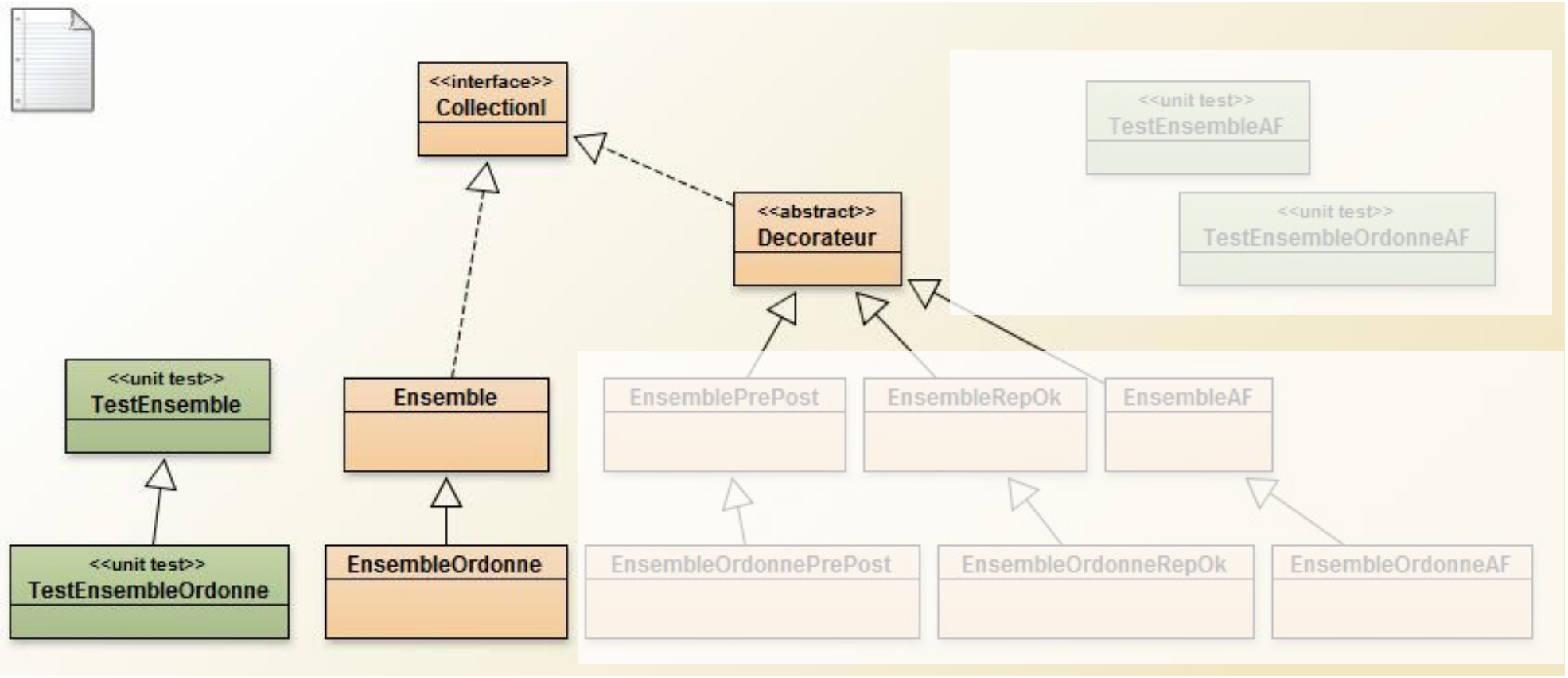
Un autre exemple : la classe Ensemble



- **La classe Ensemble : ConcreteComponent**
- **repOk, pre-post assertion : ConcreteDecoratorA, B**

Un exemple : la classe Ensemble d'entiers

- **class Ensemble implements CollectionI**



- **Un Ensemble implémente CollectionI** *(du déjà vu …)*
- **Un ensemble ordonné est un ensemble**
  - pour lequel la relation d'ordre des éléments est conservée

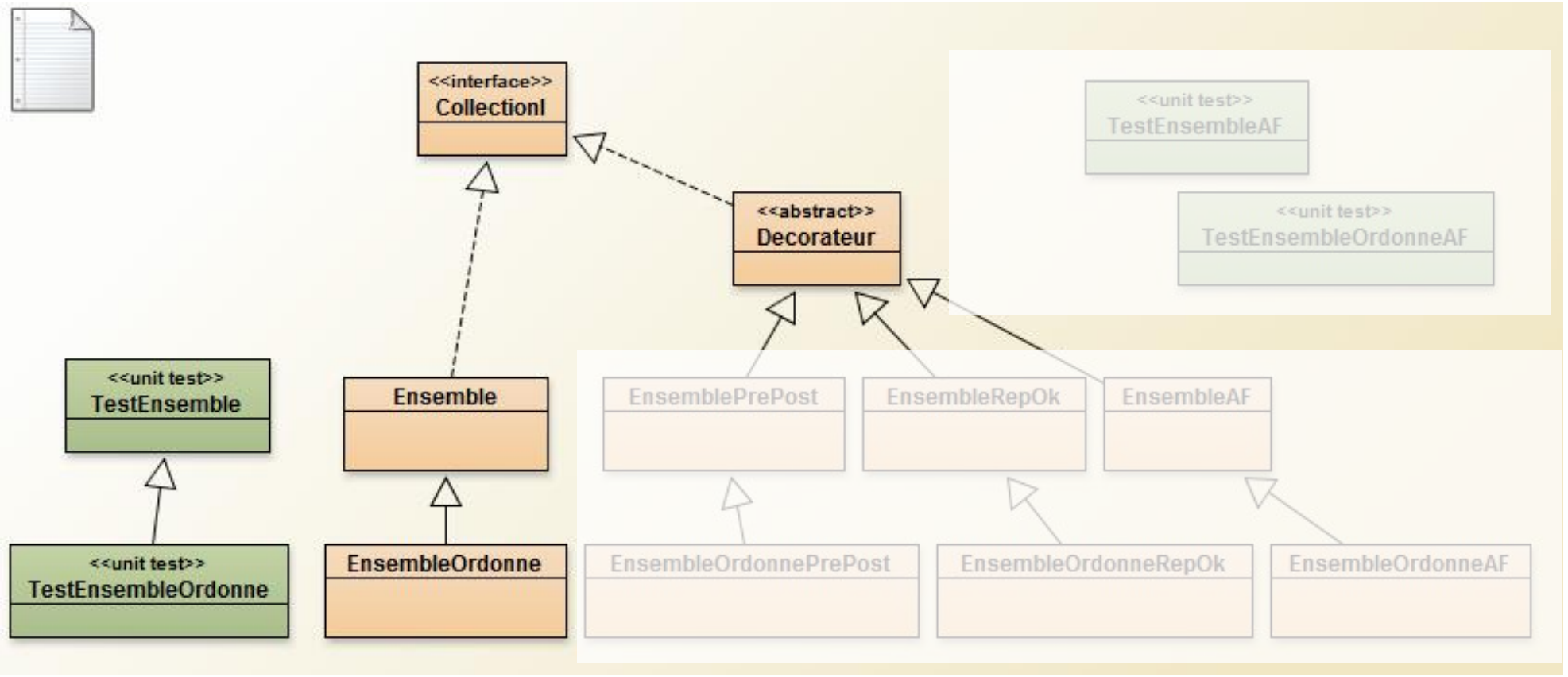
CollectionI

```
3
4  public interface CollectionI extends Iterable<Integer>{
5
6      public void ajouter(int i);
7      public void ajouter(CollectionI I);
8      public void retirer(int i);
9
10     public boolean contient(int i);
11     public int taille();
12
13     /** Invariant de classe. cf. B. Liskov.*/
14     public boolean repOk();
15     /** Fonction d'abstraction. */
16     public Object af();
17
```

- **Une collection**
  - **Ajout, retrait, présence … habituelles …**
  - **repOk invariant de classe**
  - **af fonction d'abstraction**

Un exemple : la classe Ensemble d'entiers

- **class Ensemble implements CollectionI**



- **Un Ensemble est donc une collection sans doublon**

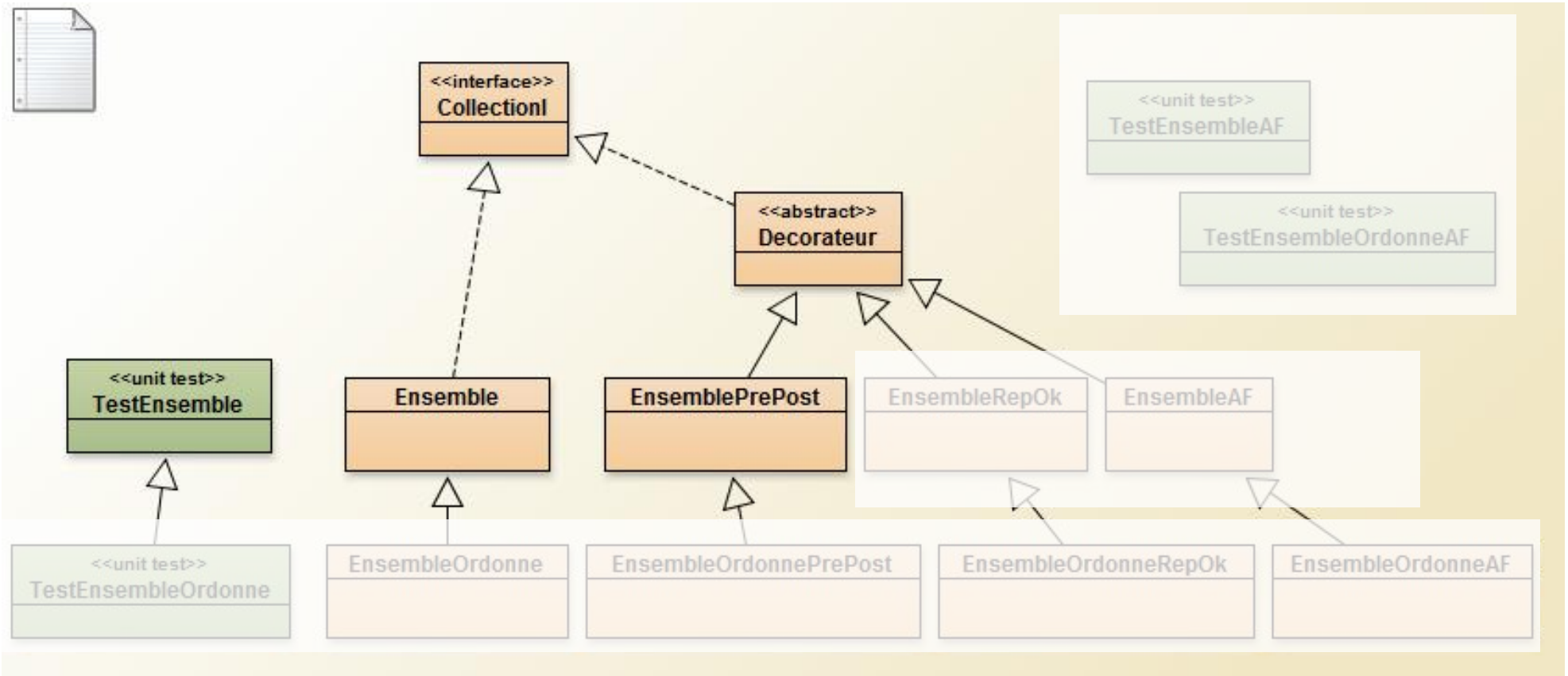
La classe Ensemble …

- **Par délégation**
  - **Usage d'une liste**
- **ajouter sans doublons**

```
2
3  public class Ensemble implements CollectionI {
4    protected List<Integer> liste;
5
6    public Ensemble(){
7      this.liste = new ArrayList<Integer>();
8    }
9
10   public void ajouter(int i){
11     if(!contient(i)) liste.add(i);
12   }
13   public void ajouter(CollectionI e){
14     for(int i : e)
15       ajouter(i);
16   }
17   public void retirer(int i){
18     liste.remove(new Integer(i));
19   }
20   public int taille(){
21     return liste.size();
22   }
23   public boolean contient(int i){
24     return liste.contains(i);
25   }
26
27   public Iterator<Integer> iterator(){
28     return liste.iterator();
```

Décorations pre-post

- **public class EnsemblePrePost extends Decorateur**



- **Décorations comme pré-post assertions**
  - **Soit la classe EnsemblePrePost**

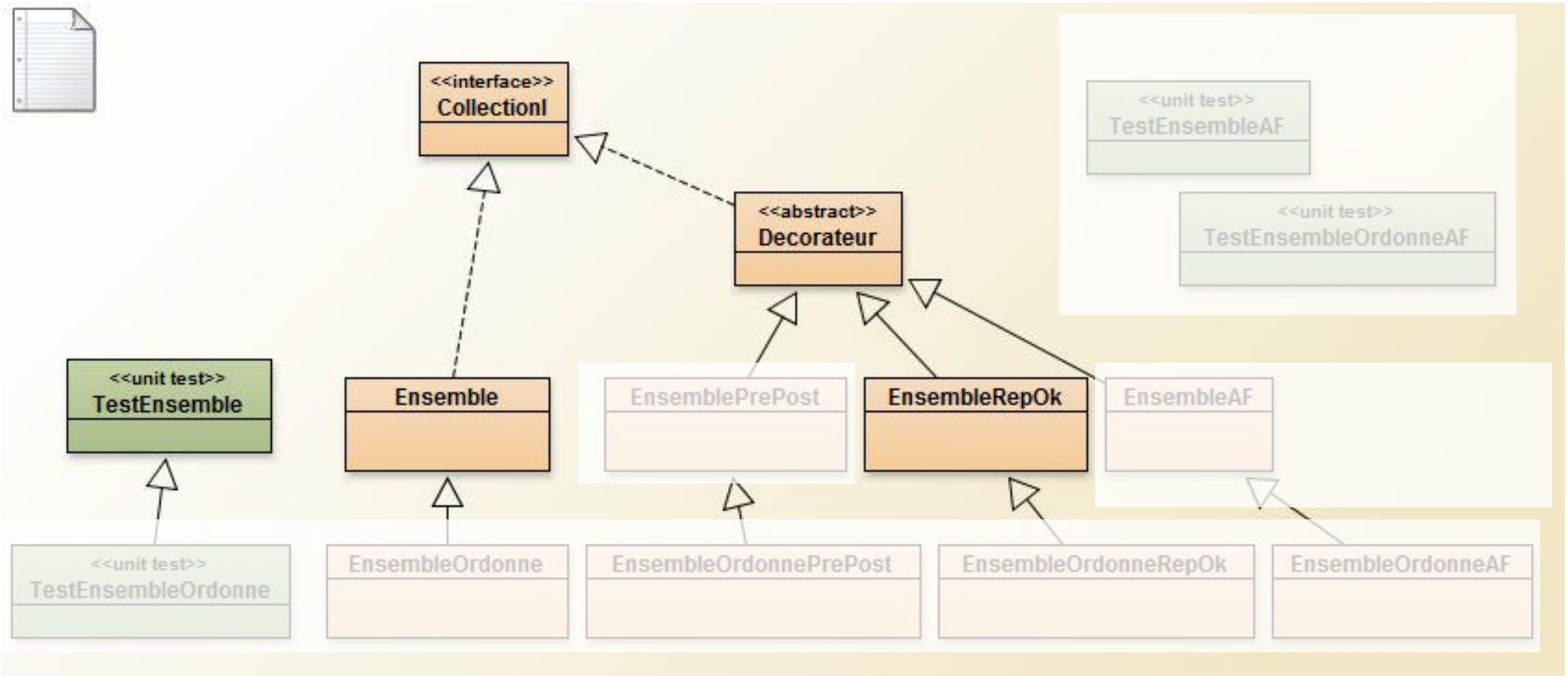
Décoration : EnsemblePrePost

```
4  public class EnsemblePrePost extends Decorateur{
5
6    protected boolean pre_ajouter(int i){
7      return true;
8    }
9    protected boolean post_ajouter(int i){
10     return (tailleAvant+1 == super.taille() && !dejaPresent && super.contient(i)) ||
11           (tailleAvant == super.taille() && dejaPresent && super.equals(ensembleAvant));
12   }
13   public void ajouter(int i){
14     initVariables(i);
15     assert pre_ajouter(i) : "pre assertion ajouter invalide !";
16     super.ajouter(i);
17     assert post_ajouter(i) : "post assertion ajouter invalide !";
18   }
19
20   public EnsemblePrePost( CollectionI c){
21     super(c);
22     assert c.repOk();
23   }
```

- *initVariables(i) :* une méthode utilitaire locale,
  - tailleAvant, …

Décoration RepOk

- **public class EnsembleRepOk extends Decorateur**



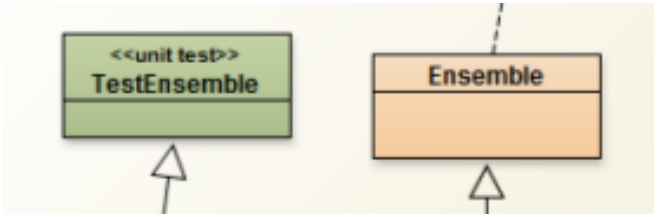
- **Décorations comme vérification de l'invariant**
  - **Soit la classe EnsembleRepOk**

Décoration : EnsembleRepOk

```
public class EnsembleRepOk extends Decorateur{

   public EnsembleRepOk( CollectionI c){
      super(c);
      assert c.repOk() : " repOk invalide !";
   }

   public void ajouter(int i){
      assert super.repOk() : " repOk invalide !";
      super.ajouter(i);
      assert super.repOk() : " repOk invalide !";
   }
   public void ajouter(CollectionI c){
      assert super.repOk() : " repOk invalide !";
      super.ajouter(c);
      assert super.repOk() : " repOk invalide !";
   }
```

- **Invariant de classe**
  - **Avant et après chaque exécution de méthode**
  - **À la sortie du constructeur**

Tests unitaires, TestEnsemble




```
public class TestEnsemble extends junit.framework.TestCase{
  protected CollectionI e,e1;

  protected void setUp(){
     // la collection est décorée
     this.e = new EnsemblePrePost( new EnsembleRepOk( new Ensemble()));
}
```

```
 public void testAjouter(){
  e.ajouter(3);
  assertTrue(e.contient(3));
  assertEquals(1, e.taille());
  e.ajouter(3);
  assertEquals(1, e.taille());
  assertTrue(e.contient(3));
  e.ajouter(2);
  assertEquals(2, e.taille());
  assertTrue(e.contient(2));
}
```

Héritage d'assertions et discussions

- **Ensemble e = new Ensemble();**
  - **e instanceof Ensemble alors ce sont les pré et post-assertions définies dans la classe Ensemble pour la méthode ajouter**

pre_ens

- **e.ajouter(33);**

post_ens

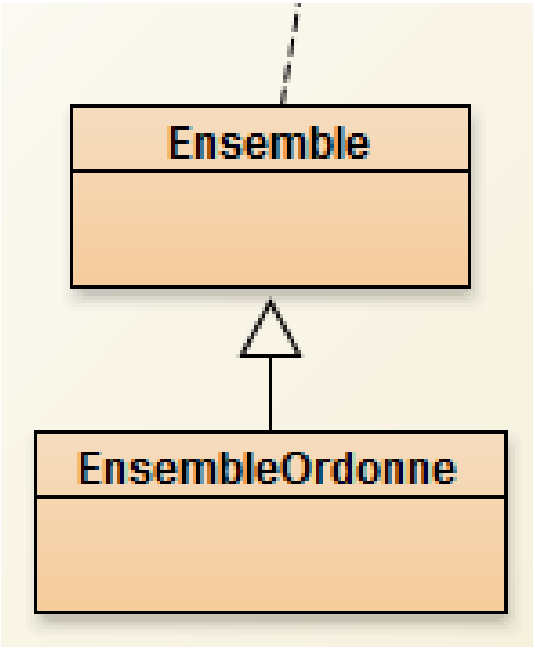
- **Ensemble e = new EnsembleOrdonne();**
  - **e instanceof EnsembleOrdonne a l'appel**

pre_ens_ord

- **e.ajouter(33);**

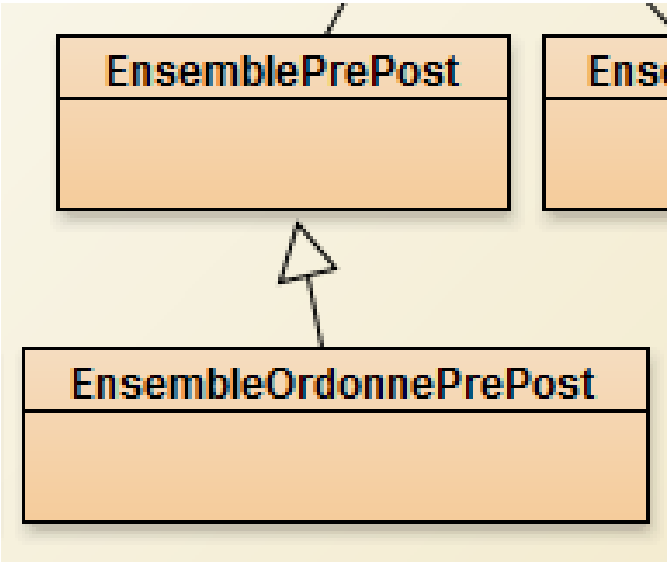
post_ens_ord

Pre post et héritage, présentation



- **pre_ens**
  - **ajouter(int i)**
- **post_ens**

- **pre_ens_ord**
  - **ajouter(int i)**
- **post_ens_ord**

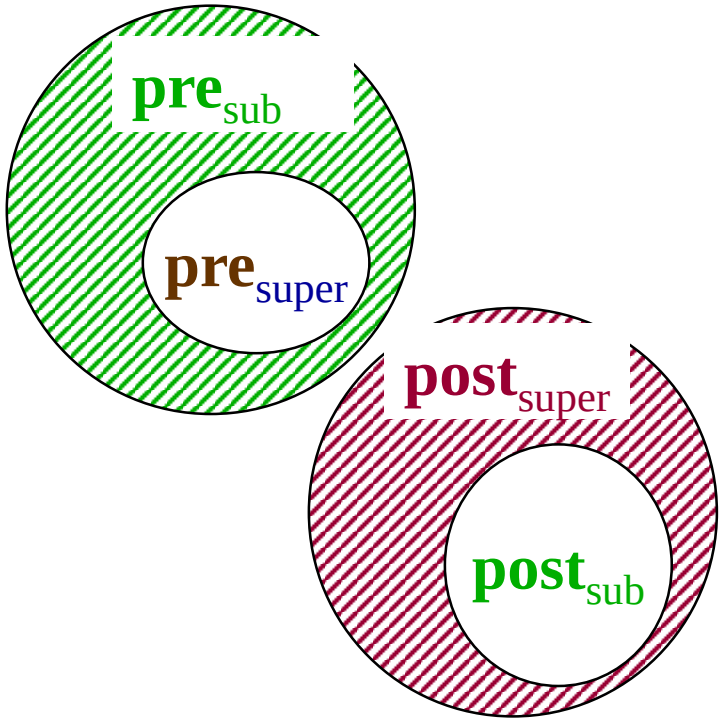


- **Ensemble e =** *new EnsembleOrdonne();*
- **pre_ens → (implique) pre_ens_ord**
- *e.ajouter(33);*
- **post_ens_ord → post_ens**

Héritage

- **Héritage d'assertions : héritage de décorations**

- **Que dit B. Liskov ? Page 176**
  - pre_{super} ➔ pre_{sub}
  - pre_{super} && $post_{sub}$ ➔ $post_{super}$



- **En Eiffel,**
  - pre_{super} || pre_{sub}
  - $post_{super}$ && $post_{sub}$

Héritage en clair

- **Liskov https://fr.wikipedia.org/wiki/Principe_de_substitution_de_Liskov**
  - **Contravariance pour les arguments**
  - **Covariance pour le résultat retourné**
  - **Aucune nouvelle exception ne doit être levée**

> Aucune nouvelle exception ne doit être générée par la méthode du sous-type, sauf si celles-ci sont elles-mêmes des sous-types des exceptions levées par la méthode du supertype

- **Liskov : parfois (trop) contraignant alors Eiffel ?**

- **Eiffel**
  - **The precondition can only become weaker than in the inherited contract.**
  - **The postcondition can only become stronger than in the inherited contract.**
    - https://docs.eiffel.com/book/platform-specifics/design-contract-and-assertions
  - In brief, the preconditions of overridden methods are or-ed, the post-conditions are and-ed.
    - https://en.wikibooks.org/wiki/Computer_Programming/Design_by_Contract

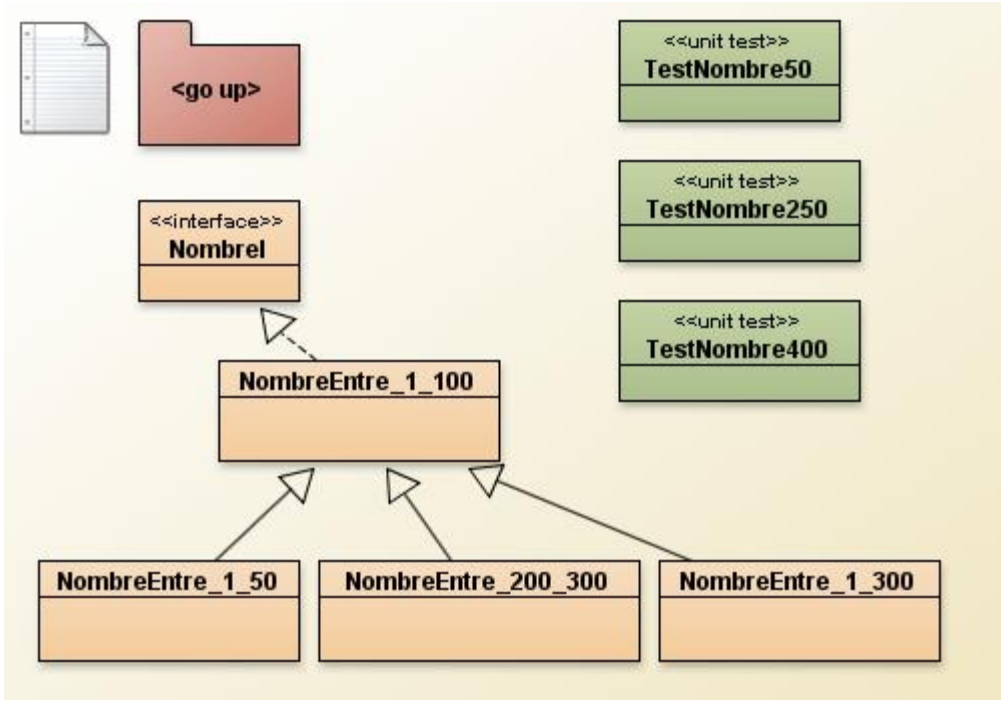
Les décorations sont héritées

```
5  public class EnsembleOrdonnePrePost extends EnsemblePrePost{
6
7    protected boolean pre_ajouter(int i){
8      return true;
9    }
0    protected boolean post_ajouter(int i){
1      return estOrdonne();
2    }
3
4    public void ajouter(int i){
5      // Eiffel, cofoja, etc ...
6      assert super.pre_ajouter(i) | this.pre_ajouter(i);
7      // pre_super ==> pre_sub   // page 176 Liskov
8      assert !super.pre_ajouter(i) | this.pre_ajouter(i) : "pre assertion ajouter invalide !";
9      super.ajouter(i);
0      // (pre_super && post_sub) ==> post_super Liskov
1      assert !(super.pre_ajouter(i) & this.post_ajouter(i)) | super.post_ajouter(i) : "post assertion ajouter
2      // Eiffel
3      assert this.post_ajouter(i) & super.post_ajouter(i) : "post assertion ajouter invalide !";
4    }
```

- pre_{super} || pre_{sub}
  $post_{super}$ && $post_{sub}$

pre_{super} ➔ pre_{sub} soit !(Pre_{super}) || pre_{sub}

Discussions



- pre_{super} ➔ pre_{sub}
- pre_{super} && $post_{sub}$ ➔ $post_{super}$

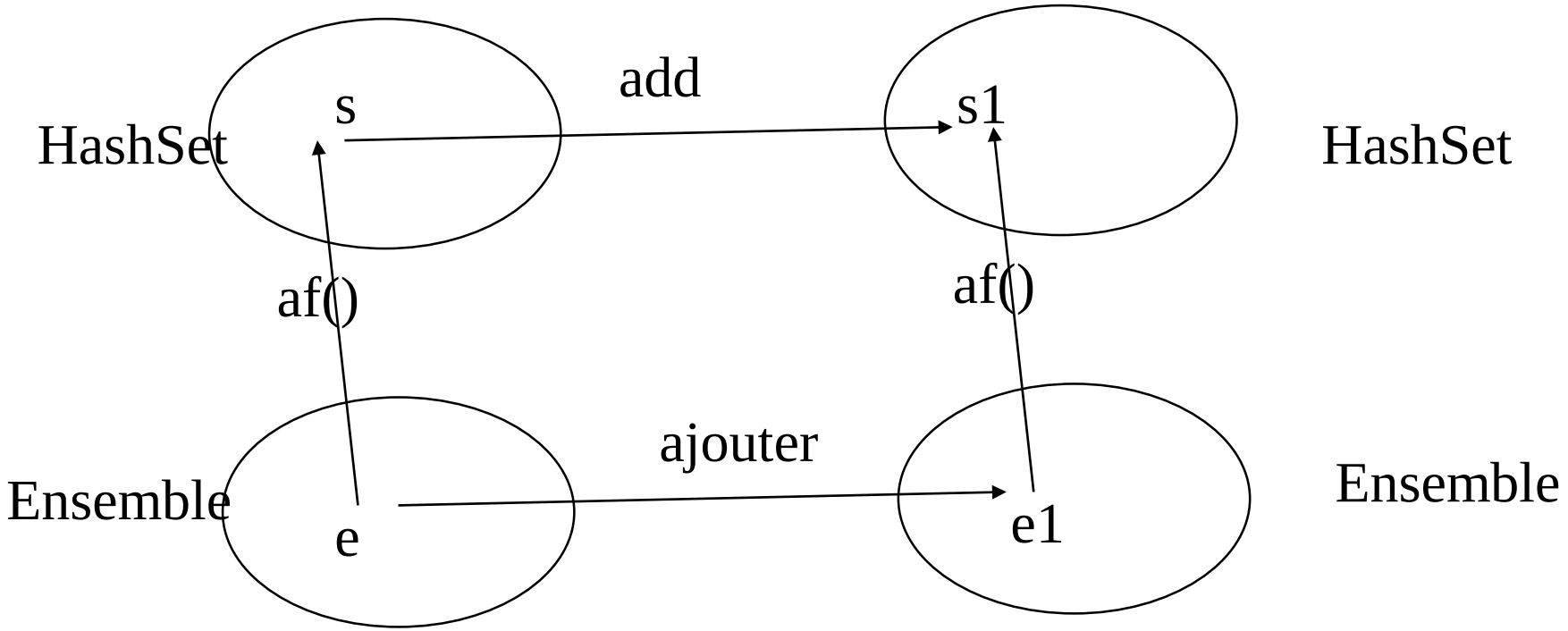
- pre_{super} || pre_{sub}
- $post_{super}$ && $post_{sub}$

- http://jfod.cnam.fr/NFP121/assertions_et_decorateur.jar

- Rendez vous package heritage_pre_post_pour_discussion

repOk == invariant, af, pre, post

- **af**
  - **Fonction d'abstraction**



Une nouvelle décoration

Fonction d'abstraction, une nouvelle décoration

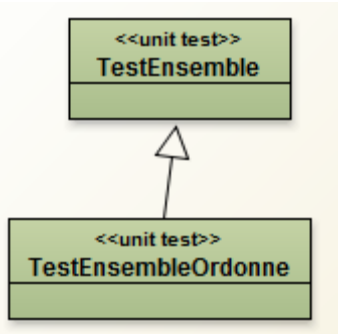
```
3  public class EnsembleAF extends Decorateur{
4
5      public EnsembleAF( CollectionI c){
6          super(c);
7      }
8
9      public void ajouter(int i){
10         Set<Integer> set = (Set<Integer>)super.af();
11         assert set != null : " ajouter af invalide !";
12         set.add(i);
13         super.ajouter(i);
14         assert super.af().equals(set) : " ajouter af invalide !";
15     }
16     public void ajouter(CollectionI c){
17         Set<Integer> set = (Set<Integer>)super.af();
18         assert set != null : " ajouter af invalide !";
19         for(int i : c) set.add(i);
20         super.ajouter(c);
21         assert super.af().equals(set) : " ajouter af invalide !";
22     }
23     public void retirer(int i){
24         Set<Integer> set = (Set<Integer>)super.af();
25         assert set != null : " retirer af invalide !";
26         set.remove(i);
27         super.retirer(i);
28         assert super.af().equals(set) : " retirer af invalide !";
```

- **class Ensemble**

```
public  Set<Integer> af(){
    return new HashSet<Integer>(this.liste);
}
```

Démonstration

```
5
6  public class TestEnsemble extends junit.framework.TestCase{
7      protected CollectionI e,e1;
8
9      @Override
10     protected void setUp(){
11             this.e  = new EnsemblePrePost(new EnsembleRepOk(new EnsembleAF(new Ensemble())));
12             this.e1 = new EnsemblePrePost(new EnsembleRepOk(new EnsembleAF(new Ensemble())));
13
```




```
4
5  public class TestEnsembleOrdonne extends TestEnsemble{
6      @Override
7      protected void setUp(){
8
9          this.e = new EnsembleOrdonnePrePost(new EnsembleOrdonneRepOk(new EnsembleOrdonneAF(new EnsembleOrdonne())));
10         this.e1 = new EnsembleOrdonnePrePost(new EnsembleOrdonneRepOk(new EnsembleOrdonneAF(new EnsembleOrdonne())));
```

Conclusion édulcorée

- **C'était un exemple de Génie logiciel …**
  - **https://en.wikipedia.org/wiki/Design_by_contract**

Annexe <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 ?**
- **En exercice ?**
  - **Une solution (peu satisfaisante) est en annexe …**

Annexe …TexteDécoré

Texte décoré mais une seule fois par décoration … une solution peu satisfaisante…

```
public class B extends TexteDécoré{
  private static boolean décoré = false;

  public B(AbstractTexte texte){
    super(texte);
  }

  public String enHTML(){

    if(B.décoré){
      return super.enHTML();
    }else{
      B.décoré = true;
      String réponse = "<B>" + super.enHTML() + "</B>";
      B.décoré = false;
      return réponse;
    }
  }
}
```