Cours 0-1 : Introduction

jean-michel Douin, douin au cnam point fr
version : 10 Septembre 2008

Notes de cours

Sommaire

- Les objectifs des concepteurs
- Présentation des concepts de l'orienté Objet
- Conception par les patrons (*pattern*)

- Outils et développement
  - BlueJ
  - JNEWS

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/
  - Ou bien en http://www.patterncoder.org/
- [Liskov]
  - Program Development in Java,

  Abstraction, Specification, and Object-Oriented Design, B.Liskov avec J. Guttag

  Addison Wesley 2000. ISBN 0-201-65768-6

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

Java : les objectifs

- « Simple »
  - syntaxe " C "
- « sûr »
  - pas de pointeurs, vérification du code à l 'exécution et des accès réseau et/ou fichiers
- Orienté Objet
  - (et seulement !), pas de variables ni de fonctions globales, types primitifs et objet
- Robuste
  - ramasse miettes, fortement typé, gestion des exceptions
- Indépendant d 'une architecture
  - Portabilité assurée par la présence d 'un interpréteur de bytecode sur chaque machine
- Environnement riche
  - Classes pour l 'accès Internet
  - classes standard complètes
  - fonctions graphiques évoluées

Simple : syntaxe apparentée C,C++

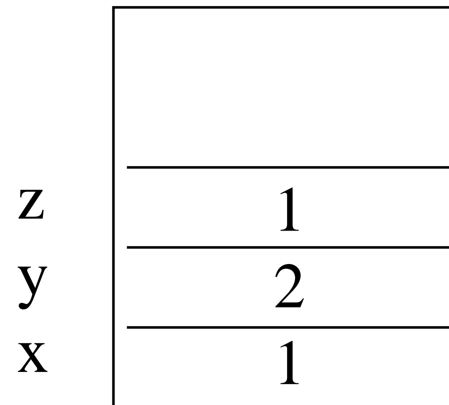
```
public class Num{

    public static int max( int x, int y){
      int max = y;
      if(x > y){
             max = x;
       }
       return max;
    }
}
```

Note : C# apparenté Java

Sûr par l 'absence de pointeurs (accessibles au programmeur)

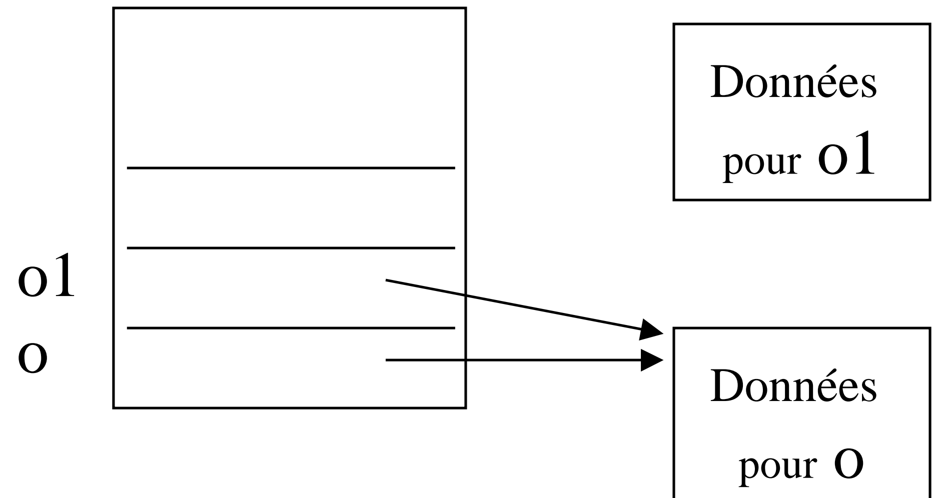
- **Deux types : primitif ou Object** (et tous ses dérivés)

- **primitif :**
  - **int x = 1;**
  - **int y = 2;**
  - **int z = x;**



- **Object**
  - **Object o = new Object();**
  - **Object o1 = new Object();**

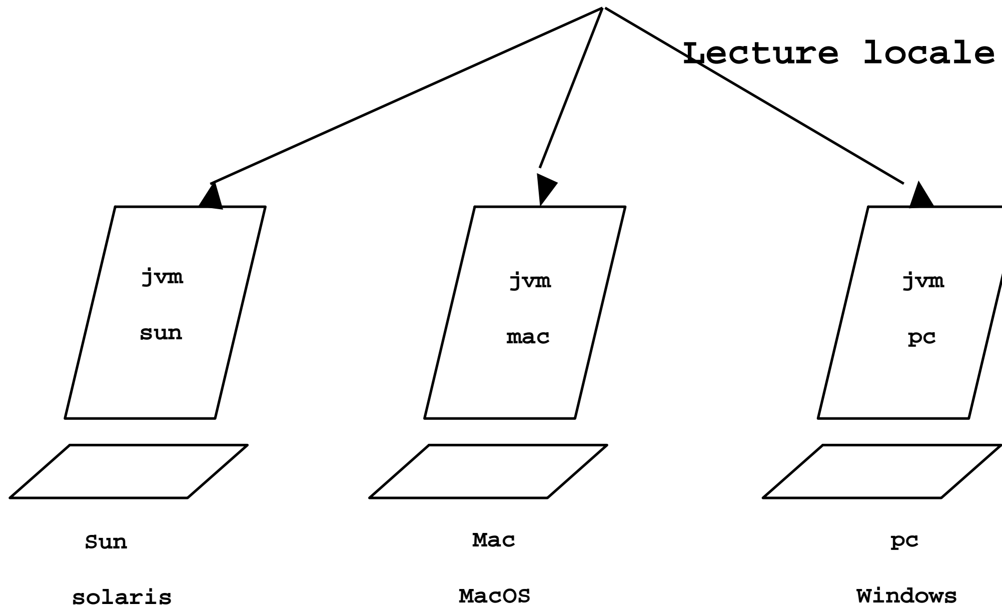
  - **Object o1 = o;**



Robuste

- **Ramasse miettes ou gestionnaire de la mémoire**
  - **Contrairement à l'allocation des objets, leur dé-allocation n'est pas à la charge du programmeur**
  - **(Ces dé-allocations interviennent selon la stratégie du gestionnaire)**
- **Fortement typé**
  - **Pas d'erreur à l'exécution due à une erreur de type**
  - **Un changement de type *hasardeux* est toujours possible**
- **Généricité**
  - **Vérification statique du bon « typage »**
- **Exceptions**
  - **Mécanisme de traitements des erreurs,**
  - **Une application ne devrait pas s'arrêter à la suite d'une erreur,**
    - **(ou toutes les erreurs possibles devraient être prises en compte …)**

Portable



Environnement (très) riche

- java.applet
- java.awt
- java.beans
- java.io
- java.lang
- java.math
- java.net
- java.rmi
- java.security
- java.sql
- java.text
- java.util
- javax.accessibility
- javax.swing
- org.omg.CORBA
- org.omg.CosNaming

- Liste des principaux paquetages de la plate-forme JDK 1.2

soit environ 1500 classes !!! Et bien d 'autres A.P.I. JSDK, JINI, ...

- le JDK1.3/1850 classes,
- Le JDK1.5 ou j2SE5.0 3260 classes
- Le J2SE 1.6 ….

Concepts de l'orienté objet

- **Le vocable Objet :**

- **Un historique …**
- **Classe et objet (instance d ’une classe)**
- **État d'un objet et données d ’instance**
- **Comportement d'un objet et méthodes**
  - **liaison dynamique**
- **Héritage**
- **Polymorphisme**

Un historique

- **Algorithm + Data Structures = Program**

A + d = P **langage de type pascal,** *années 70*

A + D = P **langage modulaire, Ada, modula-2,** *années 80*

a + D = P **langage Orienté Objet** *années 90, simula67…*

A + d = P

- **surface** (triangle t) =
- **surface** (carré c) =
- **surface** (polygone_régulier p) =
- ....
- **perimetre** (triangle t) =
- **perimetre** (carré c) =
- **perimetre** (polygone_régulier p) =
- ....

- *usage : import de la* ***librairie*** *de calcul puis*
  - **carré unCarré;** ***// une variable de type carré***
- **y = surface (unCarré)**

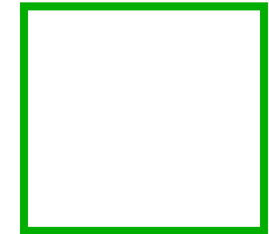
A + D = P

- type carré = structure
-                          longueurDuCote
- fin_structure;
- >>>---<<<

- surface (carré c) =
- perimetre (carré c) =
- ………… (carré c) =

- *usage : import du* ***module carré*** *puis*
  - **carré unCarré;**   ***// une variable de type carré***
- **y = surface (unCarré)**

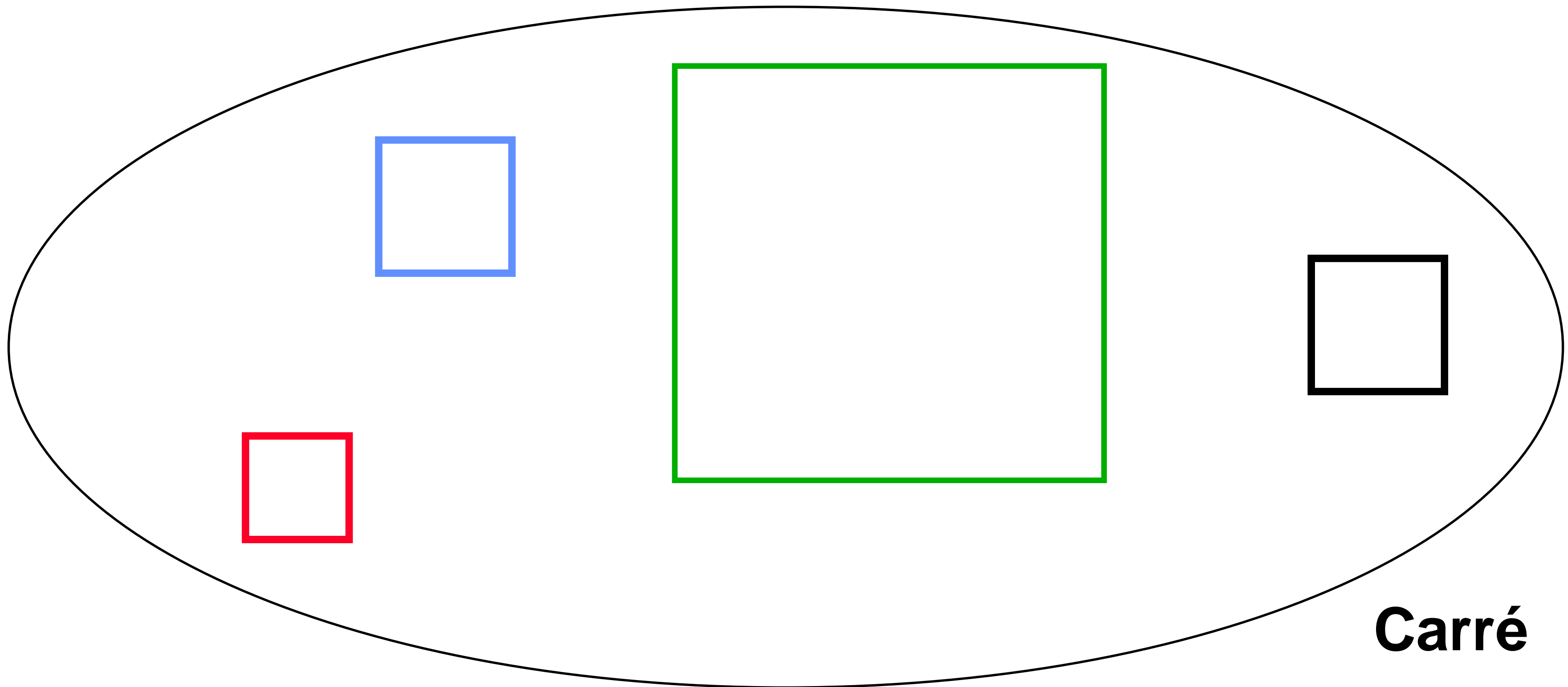
a + D = P



- classe **Carré** =
- longueurDuCote …
- surface ( ) =
- perimetre ( ) =
- ………… () =
- fin_classe;
- *usage : import de la **classe carré** puis*
  - **carré unCarré;** ***// une instance de la classe Carré***
- **y = unCarré.surface ( )**

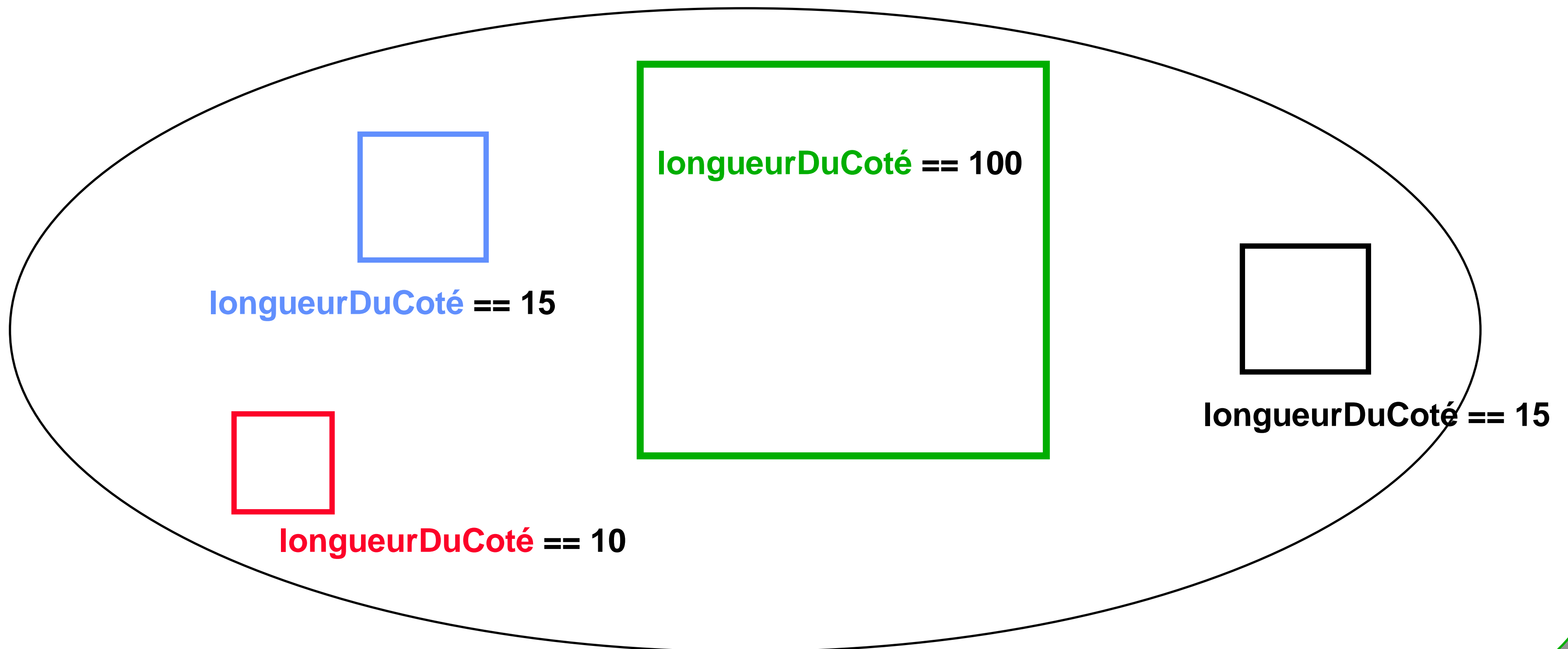
Classe et objet (instance d 'une classe)

```
class Carré{

}
```



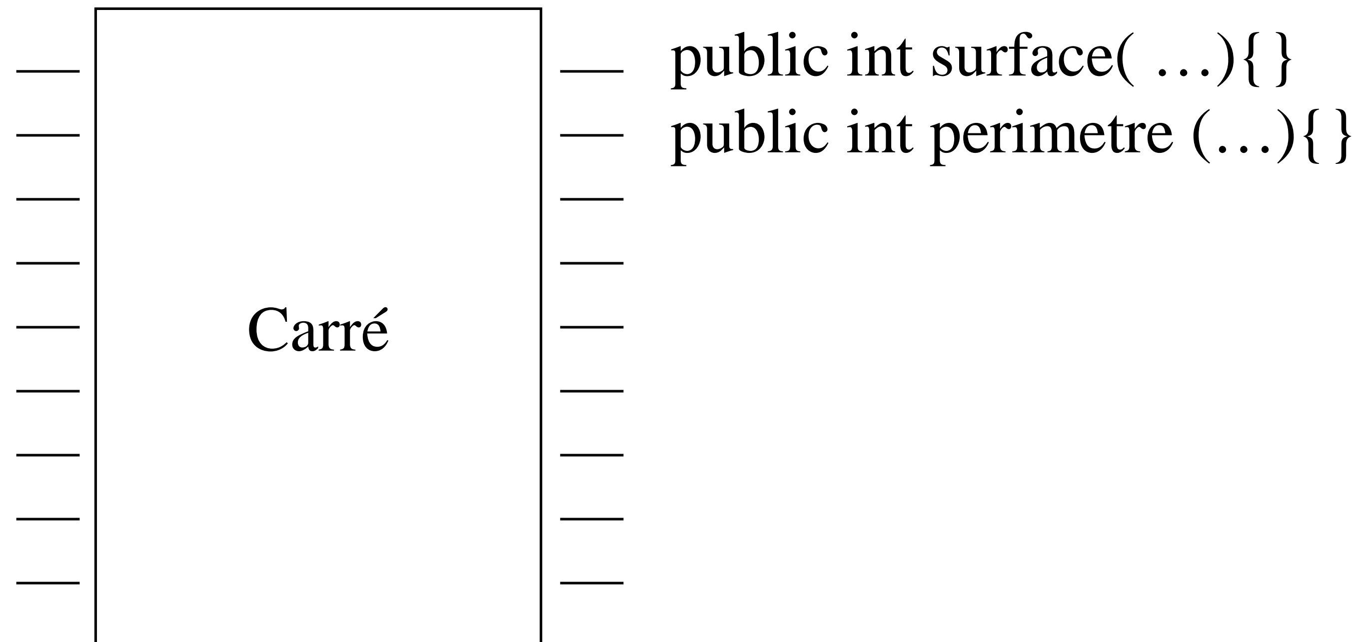
État d'un objet et données d 'instance

```
class Carré{
  int longueurDuCoté;

}
```



Classe et Encapsulation

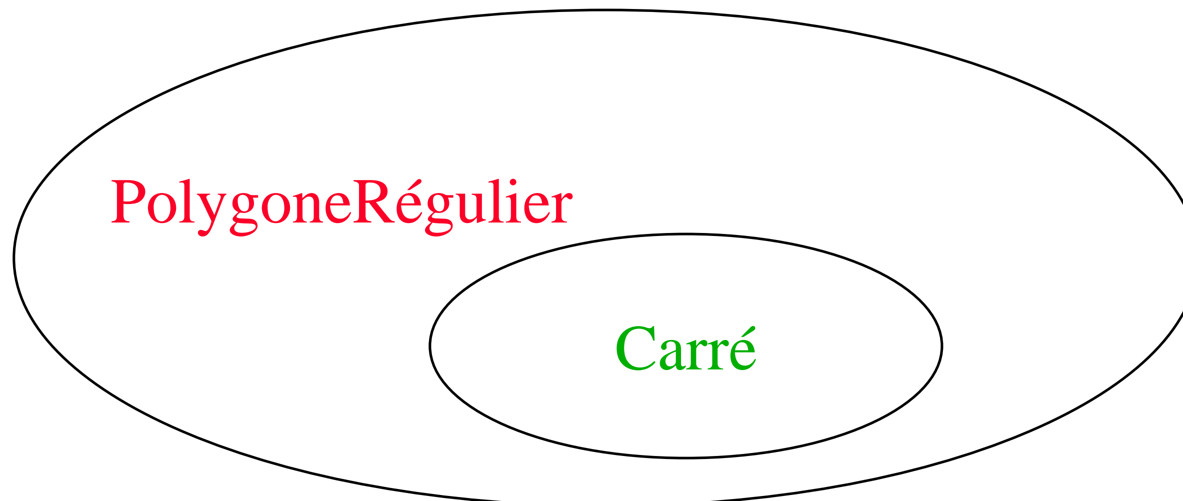
- **contrat avec le client**
  - **interface publique, une documentation, le nom des méthodes**
  - **implémentation privée**
    - **Les données d'instances**



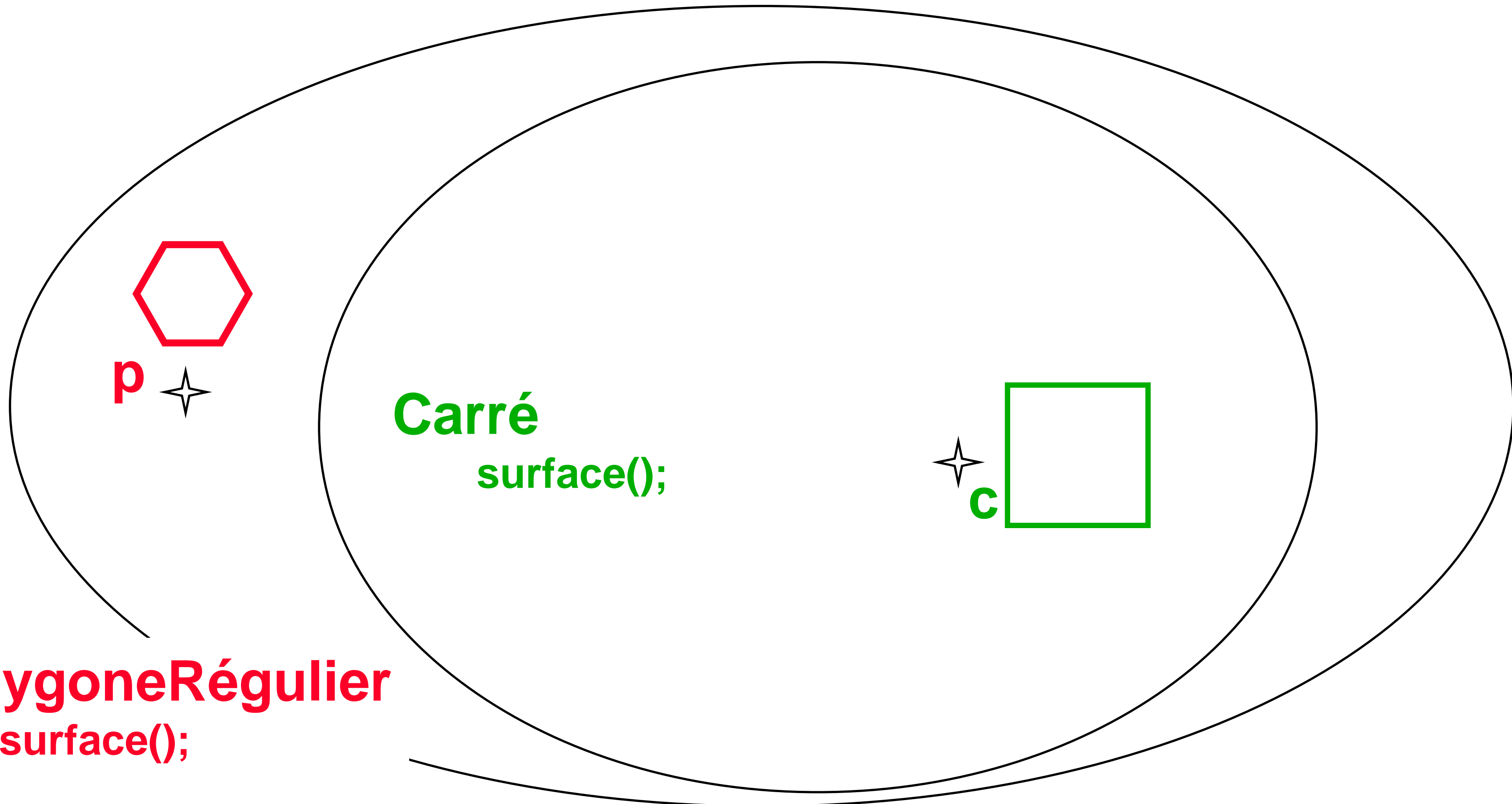
Héritage et classification

- **définir une nouvelle classe en ajoutant de nouvelles fonctionnalités à une classe existante**
  - **ajout de nouvelles fonctions**
  - **ajout de nouvelles données**
  - **redéfinition de certaines propriétés héritées (masquage)**

- **Une approche de la classification en langage naturel**
- Les carrés **sont** des polygones réguliers *(ce serait l 'idéal...)*

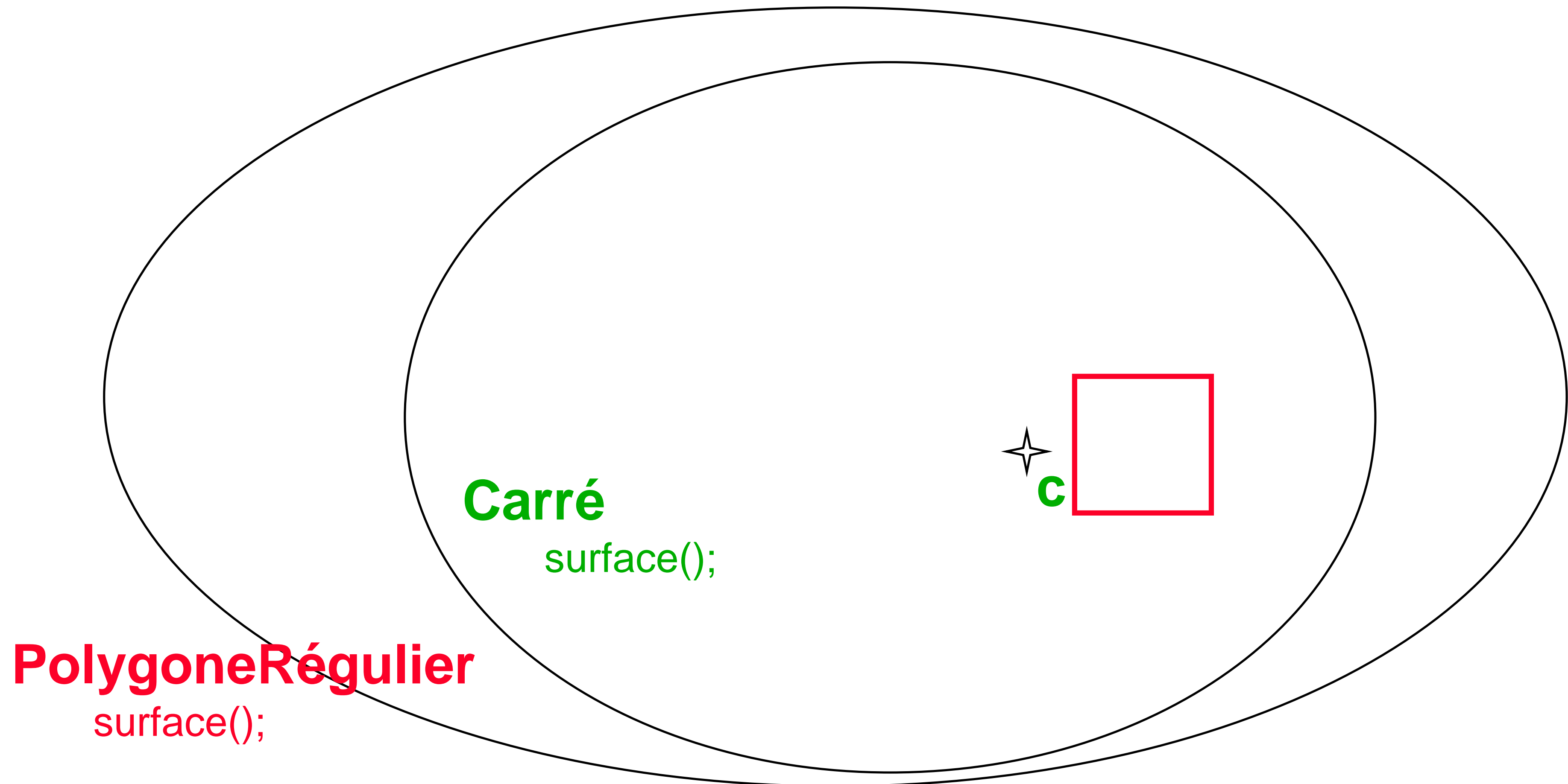


Héritage et redéfinition (masquage)



- **La méthode surface est redéfinie pour les Carrés**
- **Carré c … int s = c.surface();**
- **PolygoneRégulier p … int s1 = p.surface();**

Comportement d'un objet et méthodes



- PolygoneRégulier p = c;
- p.surface(); ??? surface(); ou surface(); ???
-

Liaison dynamique

- **PolygoneRégulier p = c;**
- **p.surface();** **???** **surface();** **ou** **surface();** **???**

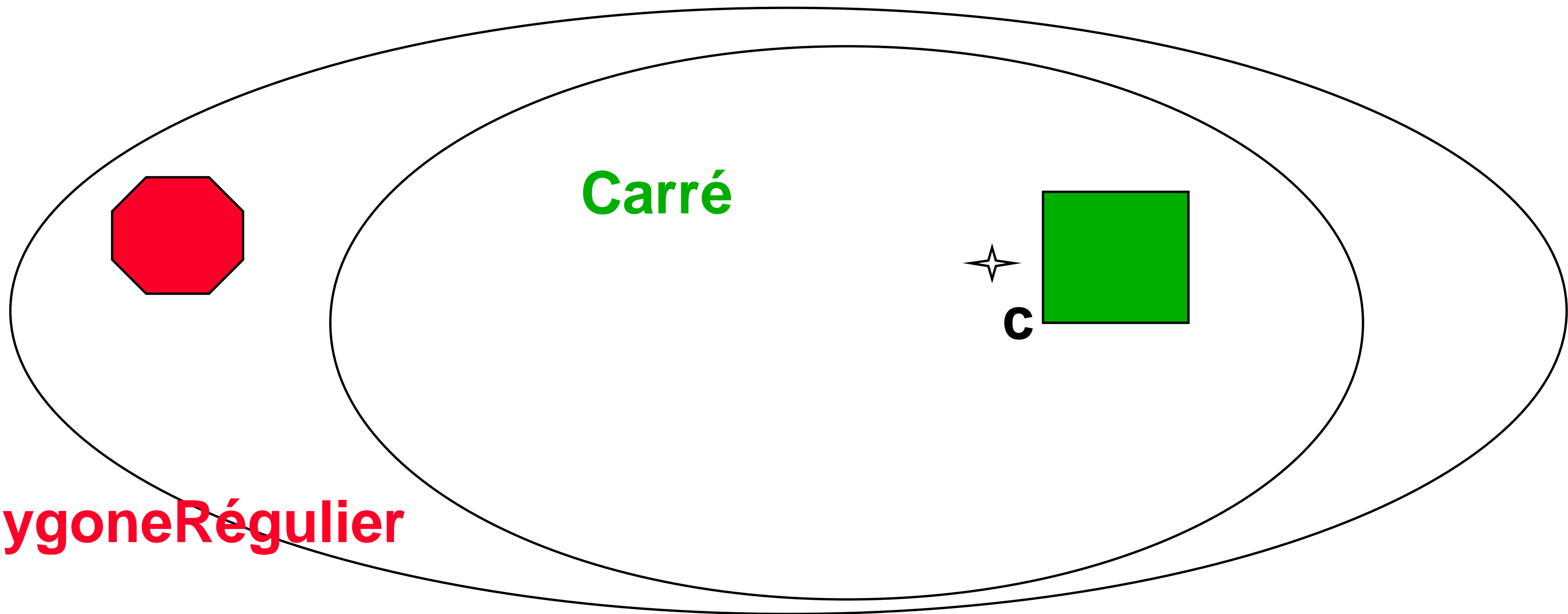
- Hypothèses
  1) La classe Carré hérite de la classe PolygoneRégulier
  2) La méthode surface est redéfinie dans la classe Carré

- **PolygoneRégulier p = c;**
  - **p doit se comporter comme un Carré**

PolygoneRégulier p1 = *… un pentagone* ;

- **p = p1** **Possible ???** **p.surface() ???**

Les carrés sont des polygones, attention



- Les carrés sont de couleur verte
- Les polygones réguliers sont rouges

- ? Couleur d'un **PolygoneRegulier** de 4 côtés ?

Polymorphisme : définitions

- **Polymorphisme ad'hoc**
  - **Surcharge(overloading),**
  - plusieurs implémentations d'une méthode en fonction des types de paramètres souhaités, le choix de la méthode est **résolu statiquement** dès la compilation

- **Polymorphisme d'inclusion**
  - **Redéfinition, masquage (overridding),**
  - est fondé sur la relation d'ordre partiel entre les types, relation induite par l'héritage. si le type B est inférieur selon cette relation au type A alors on peut passer un objet de type B à une méthode qui attend un paramètre de type A, le choix de la méthode est **résolu dynamiquement** en fonction du type de l'objet receveur

- **Polymorphisme paramétrique** ou généricité,
  - consiste à définir un modèle de procédure, ensuite incarné ou instancié avec différents types, ce choix est **résolu statiquement**

  - extrait de M Baudouin-Lafon. La Programmation Orientée Objet. ed. Armand Colin

Polymorphisme ad'hoc

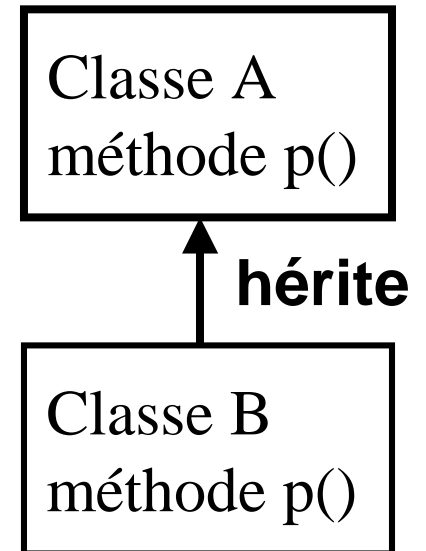
- 3 + 2   3.0 + 2.5   "bon" + "jour"

- out.print(5);   out.print(5.5);   out.print("bonjour");

  - *le choix de la méthode est **résolu statiquement** à la compilation*

Polymorphisme d'inclusion

- **A a = new A();   a.p();**
- **B b = new B();   b.p();**
- **a = new B();   a.p();**



- **void m(A a){**

  a.p();
  }

  m(new B());
  m(new(A());

  - B hérite de A, B est inférieur selon cette relation au type A
  - *le choix de la méthode est **résolu dynamiquement** en fonction du type de l'objet receveur*

Polymorphisme paramétrique

- **Une liste homogène**

- **Class Liste<T>{**

```
void add(T t) …
void remove(T t) …
…
}
```

  - **Liste<Integer> li = new Liste<Integer>();**
  - **li.add(new Integer(4));**

  - **Liste<A> la = new Liste<A>();**
  - **la.add(new A());**
  - **la.add(new B());**

  - incarné ou instancié avec différents types, ce choix est **résolu statiquement**

Affectation polymorphe

- **Création d'instances**
  - **Carre c1 = new Carre(100);**
  - **Carre c2 = new Carre(10);**
  - **PolygoneRegulier p1 = new PolygoneRegulier(4,100);**

- **Affectation**
  - **c1 = c2;** *// **synonymie** : c1 est un autre nom pour c2*
    // c1 et c2 désignent un carré de longueur 10

- **Affectation polymorphe**
  - **p1 = c1;**

- **Affectation et changement de classe**
  - **c1 = (Carre) p1;** *// Hum, Hum ...*
    - **If (p1 instanceof Carre) c1 = (Carre)p1;** *// mieux, beaucoup mieux …*

Liaison dynamique

- **Sélection de la méthode en fonction de l'objet receveur**
- **déclaré / constaté à l'exécution**

- **PolygoneRegulier p1 = new PolygoneRegulier(5,100);**
  // p1 déclarée PolygoneRegulier
- **Carre c1 = new Carre(100);**

- **int s = p1.surface();** *// p1 **constatée** PolygoneRegulier*

- **p1 = c1;** *// affectation polymorphe*

- **s = p1.surface();** *// p1 **constatée** Carre*

- Note: la recherche de la méthode s'effectue uniquement dans l'ensemble des méthodes masquées associé à la classe dérivée
  – Rappel : Dans une classe dérivée, la méthode est masquée seulement si elle possède **exactement** la même signature

En pratique... http://lmi17.cnam.fr/~barthe/OO/typage-java-2/

```
class A{
  void m(A a){  System.out.println(" m de A"); }
  void n(A a){  System.out.println(" n de A"); }
}

public class B extends A{

  public static void main(String args[]){
    A a = new B();
    B b = new B();

    a.m(b);
    a.n(b);
  }

  void m(A a){  System.out.println(" m de B"); }
  void n(B b){  System.out.println(" n de B"); }
}
```

- **Exécution de B.main : Quelle est la trace d'exécution ?**

•m de B
•n de A

En pratique : une explication

- **mécanisme de liaison dynamique en Java :**
  - **La liaison dynamique effectue la sélection d'une méthode en fonction du type constaté de l'objet receveur, la méthode doit appartenir à l'ensemble des méthodes masquées,**
  - **(la méthode est masquée dans l'une des sous-classes, si elle a exactement la même signature)**
  - Sur l'exemple,
    - nous avons uniquement dans la classe B la méthode m(A a) masquée
  - **en conséquence :**
  - **A a = new B();** *// a est déclarée A, mais constatée B*
  - **a.m --> sélection de ((B)a).m(...) car m est bien masquée**
  - **a.n --> sélection de ((A)a.n(...) car n n'est pas masquée dans B**

Choix d'implémentation de Java : compromis vitesse d'exécution / sémantique ...

Types et hiérarchie[Liskov Sidebar2.4,page 27]

- Java supports *type* ***hierarchy***, in which one type can be the ***supertype*** of other types, which are its ***subtypes***. A subtype 's objects have all the methods defined by the supertype.

- All objects type are subtypes of **Object**, which is the top of the type hierarchy. **Object** defines a number of methods, including equals and toString. Every object is guaranteed to have these methods.

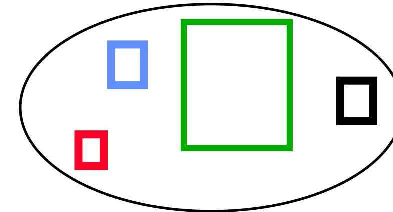
- The ***apparent type*** of a variable is the type understood by the compiler from information available in declarations. The ***actual type*** of an Object is its real type -> the type it receives when it is created.

Ces notes de cours utilisent

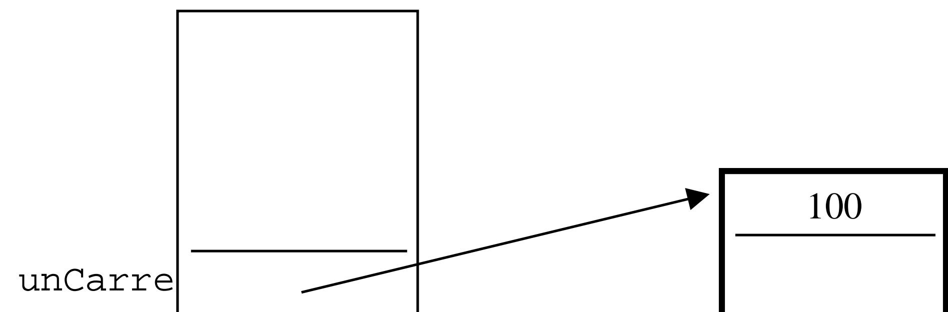
- *type déclaré pour apparent type et*
- *type constaté pour actual type*

Un exemple en Java : la classe Carre

```java
public class Carre extends PolygoneRegulier{

  private int longueurDuCote;


 public void initialiser(int longueur){
   longueurDuCote = longueur;
 }

 public int surface(){
  return longueurDuCote * longueurDuCote;
 }

 public int perimetre(){
  return 4*longueurDuCote;
 }

}
```



```java
// un usage de cette classe
    Carre unCarre = new Carre();
    unCarre.initialiser(100);
    int y = unCarre.surface();
```



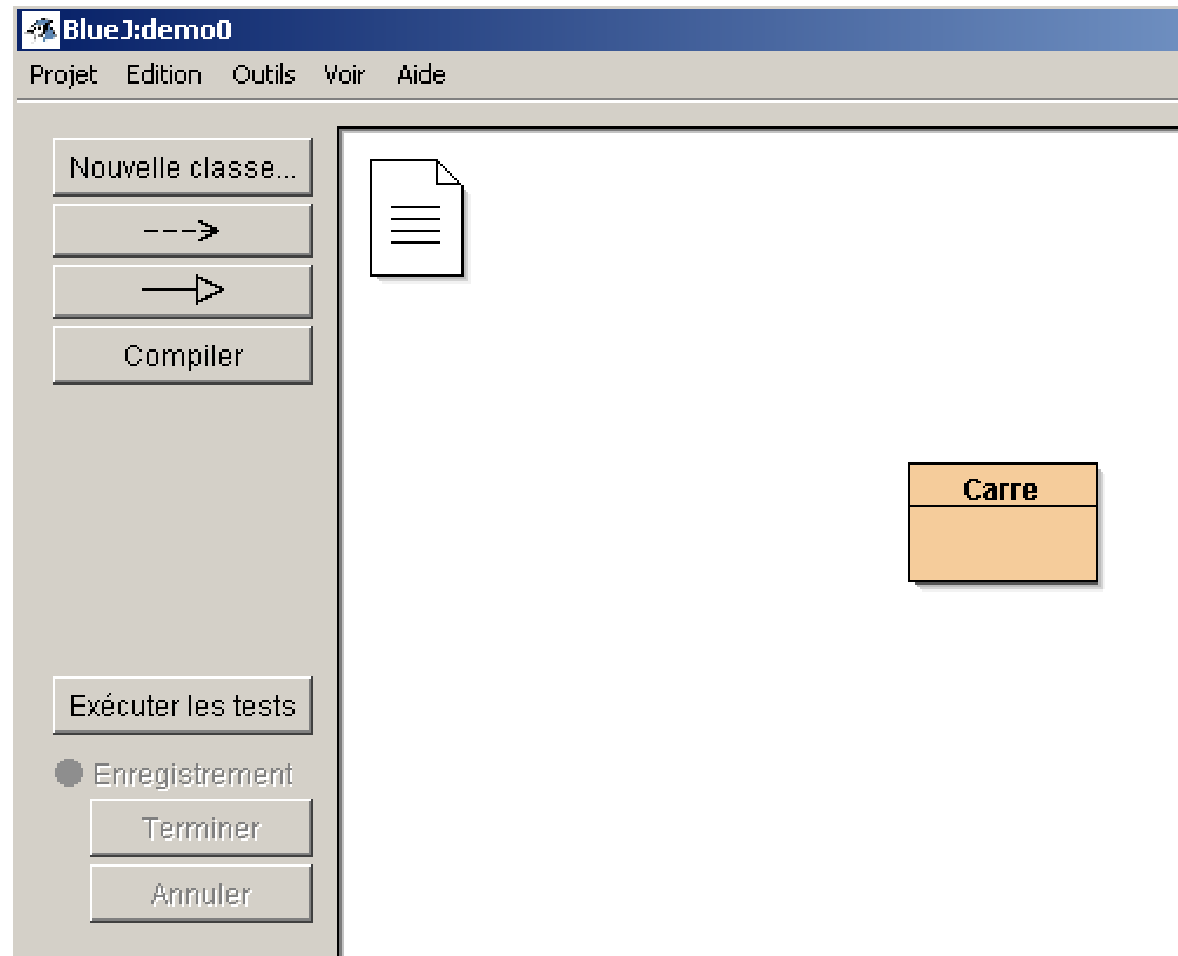
Démonstration

- **Outil Bluej**
  - La classe Carré
  - Instances inspectées

- **Tests Unitaires**
  - La Classe CarréTest
    - Ou la mise en place d'assertions pertinentes
    - En assurant ainsi des test de non régression,
      - le système ne se dégrade pas à chaque modification..
    - Augmenter le taux de confiance envers le code de cette classe …
      - Informel, mais comment obtenir des tests pertinents ?

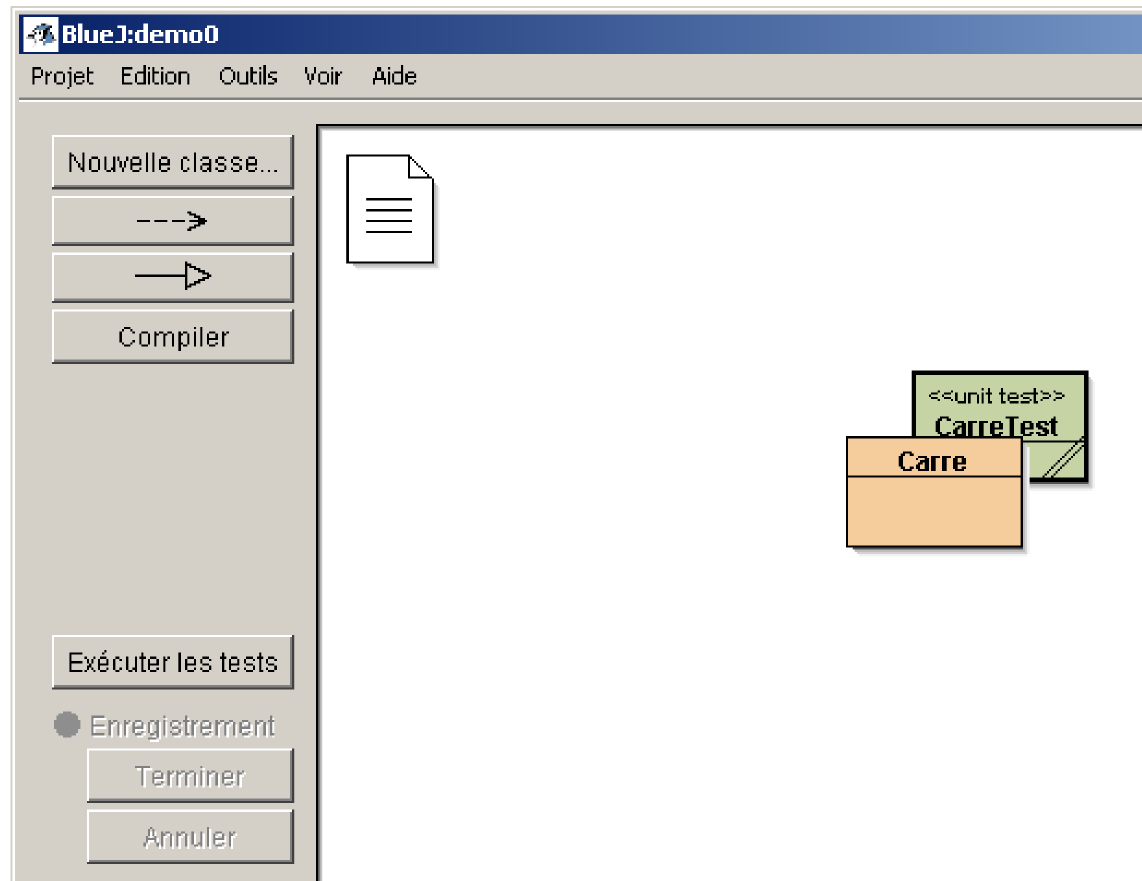
- **Tests Unitaires « référents » : Outil JNEWS**
  - Java New Evaluation Web System

Demo : Bluej



- **Instances et leur inspection**

Demo : Bluej + tests unitaires



- **Test unitaires depuis BlueJ ou en source**

Tests unitaires : outil *junit* intégré

- **www.junit.org**
- **http://junit.sourceforge.net/javadoc/junit/framework/Assert.html**
- **Un exemple :**

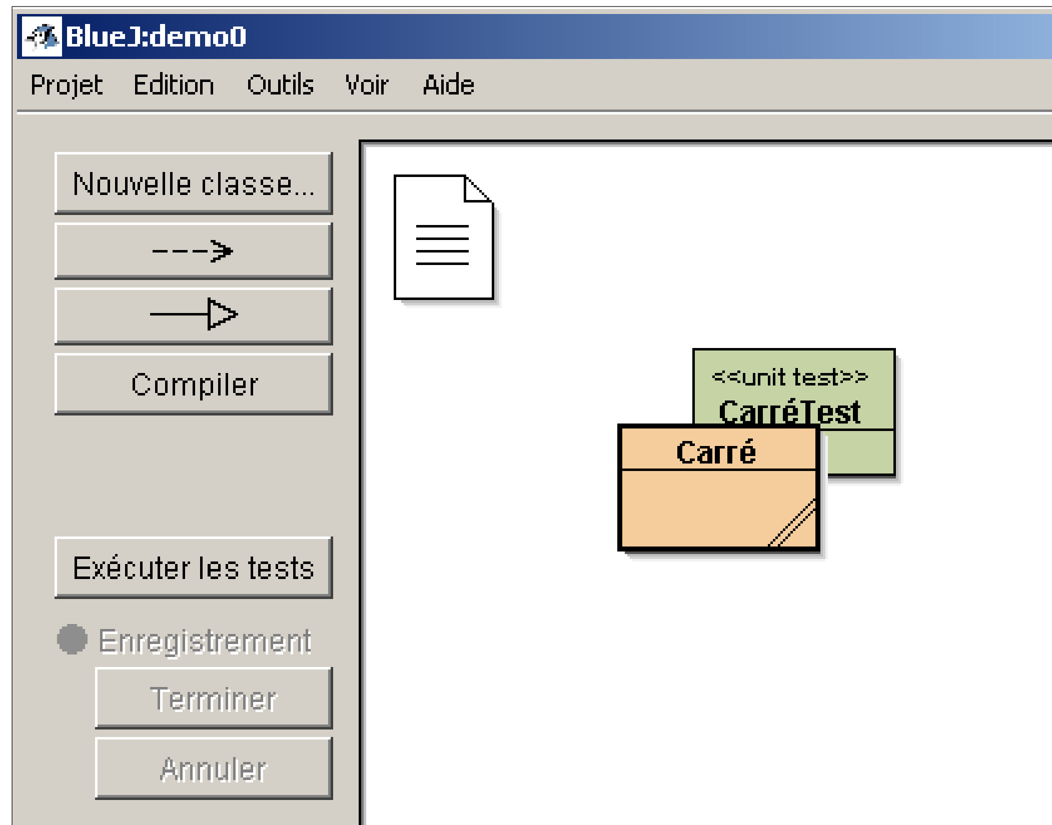
```
public class CarreTest extends junit.framework.TestCase{


  public void testDuPerimetre(){
   Carre c = new Carre();
   c.initialiser(10);
   assertEquals(" périmètre incorrect ???" ,40, c.perimetre());
  }
}
```

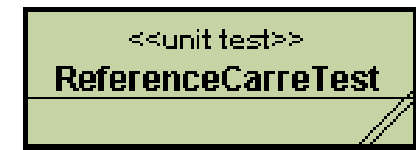
```
assertEquals(" un commentaire ???" ,attendu, effectif);
assertSame(" un commentaire ???" ,attendu, effectif);
assertTrue(" un commentaire ???" ,expression booléenne);
assertFalse(" un commentaire ???" , expression booléenne);
assertNotNull(" un commentaire ???" ,un object);
…
```

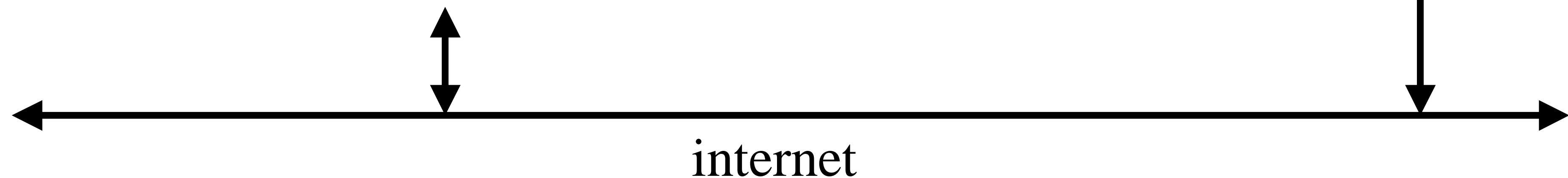
Le commentaire est affiché lorsque l'assertion échoue

JNEWS contient des Tests unitaires distants

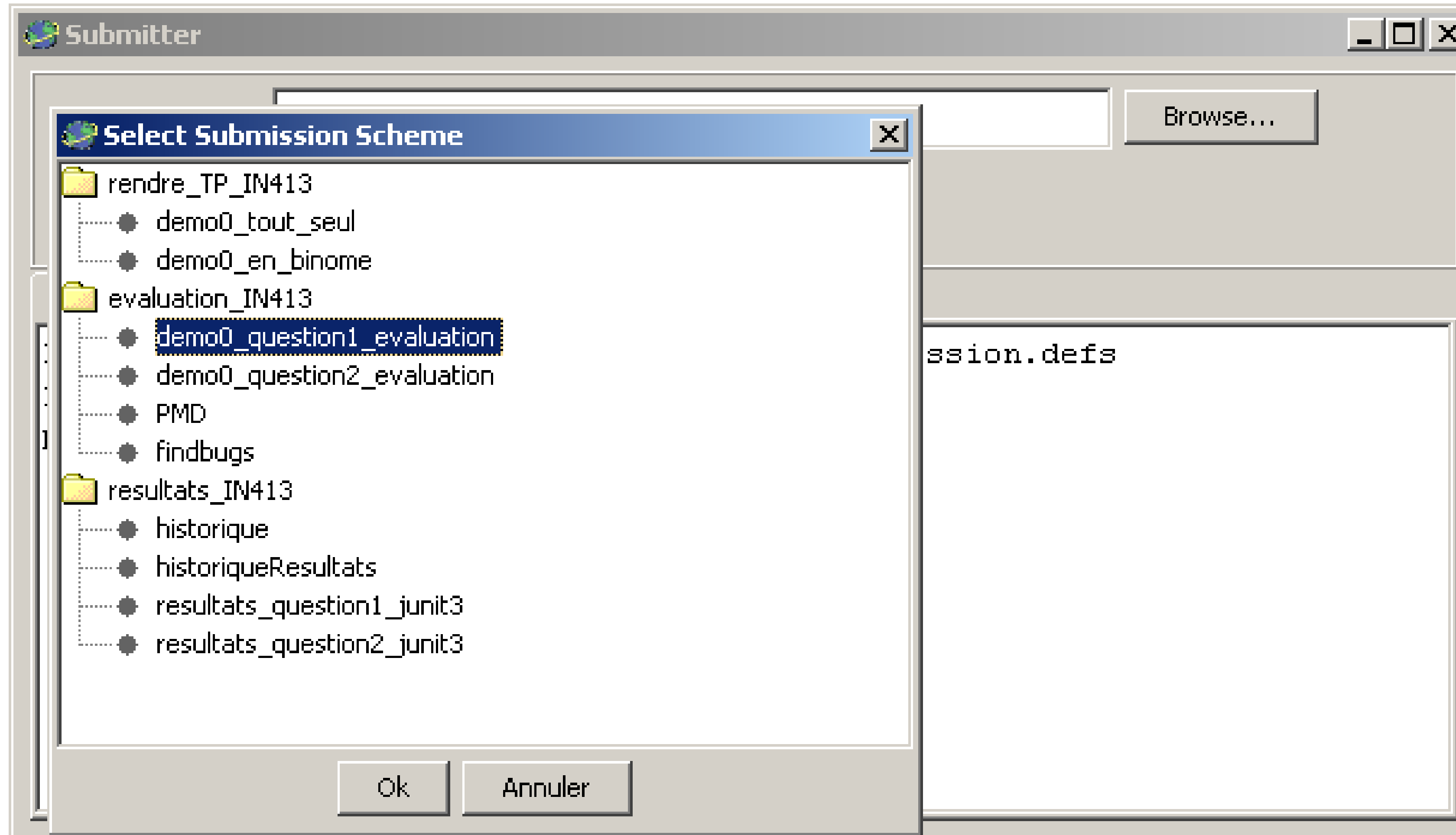
- **Tests unitaires distants et référents** *(ce qui est attendu…)*





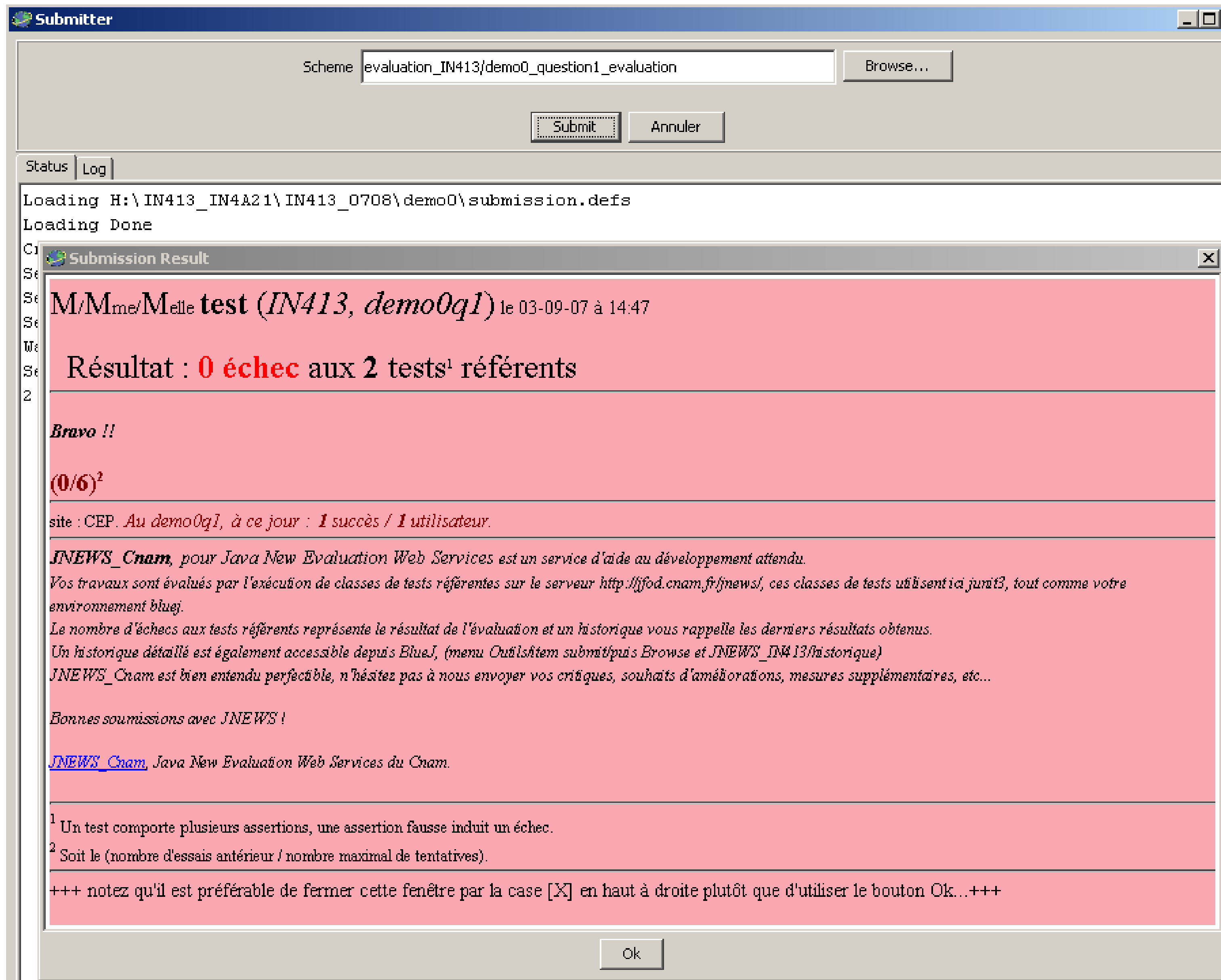


JNEWS : outil *Submitter*... intégré



- **Un clic suffit**

Demonstration : JNEWS une réponse : *Bravo!!*



JNEWS à l'ESIEE, en intranet

- Une aide à la réponse attendue
- +
- Outils en ligne comme PMD , findbugs , checkstyle
- +
- Remise planifiée des sources bien documentés

- http://jfod.cnam.fr/jnews/

Pattern pourquoi ?

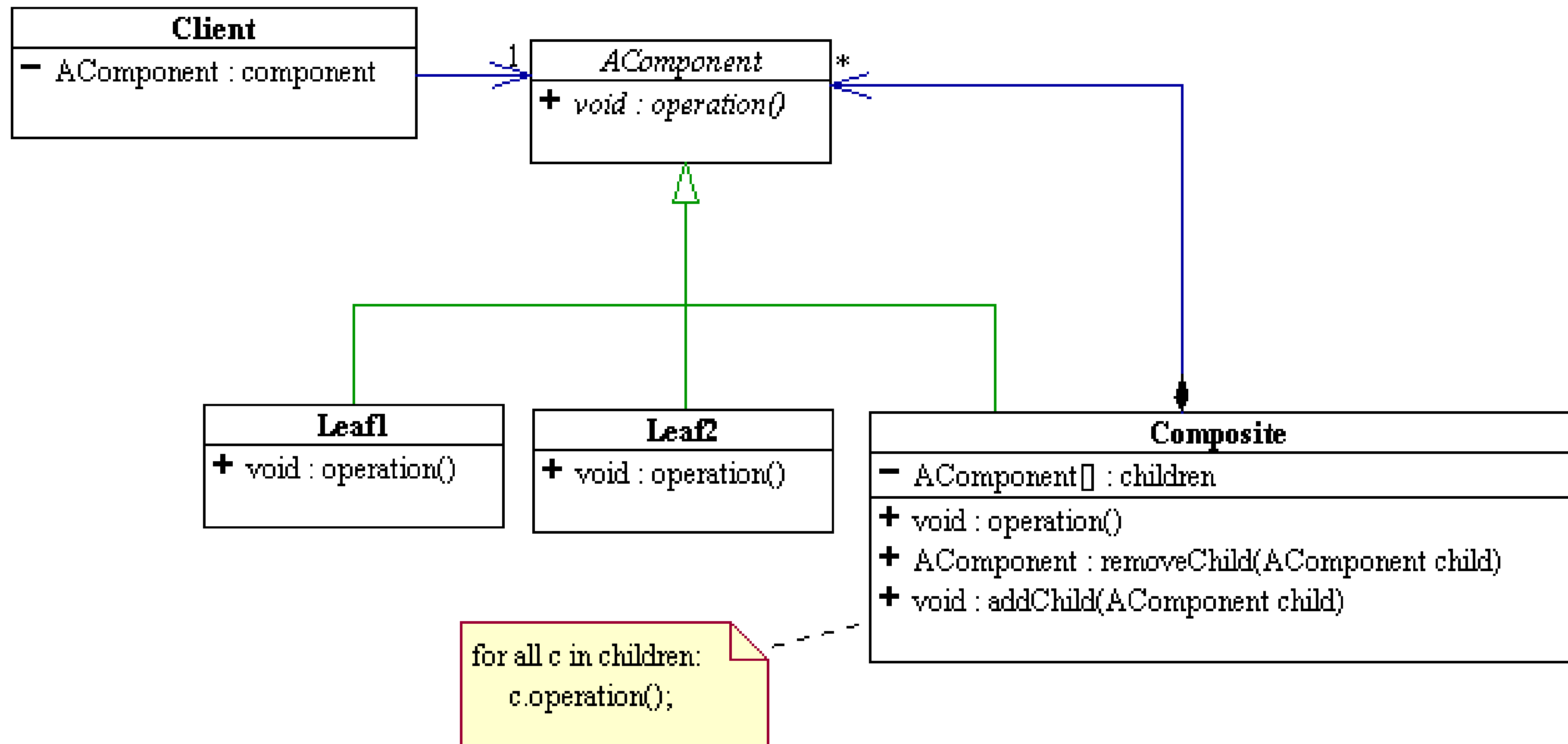
- **Patterns ou Modèles de conception réutilisables**
- **Un modèle == plusieurs classes == Un nom de Pattern**

  -> Assemblage de classes pour un discours plus clair

- **Les librairies standard utilisent ces Patterns**
  - **L'API AWT utilise le patron/pattern composite ???**
  - **Les évènements de Java utilisent le patron Observateur ???**
  - **…**
  - **etc. …**
- **Une application = un assemblage de plusieurs patterns**
  - **Un rêve ?**

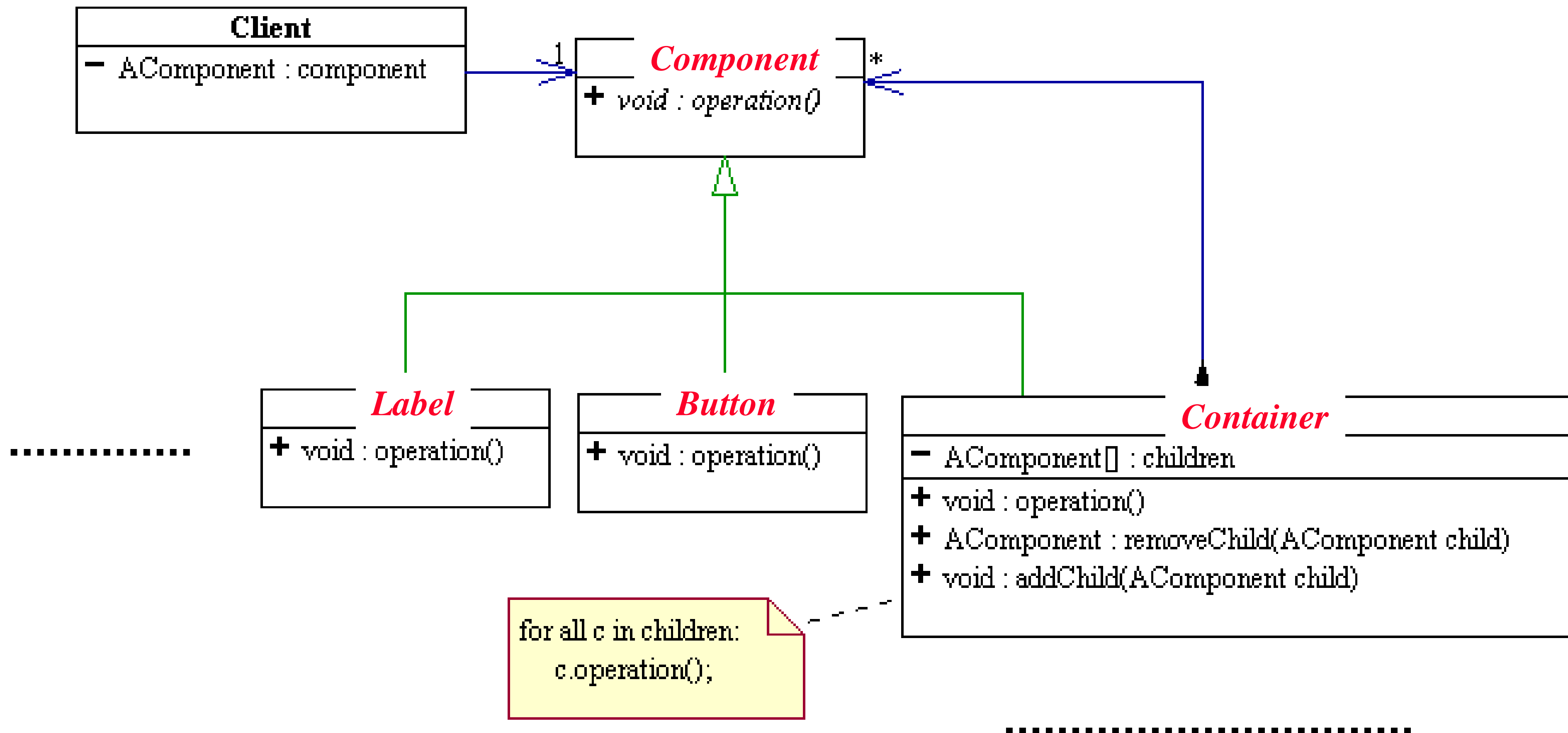
La bibliothèque graphique du JDK utilise un composite ?

- **Le pattern Composite ?, usage d'un moteur de recherche sur le web …**



la bibliothèque graphique utilise bien un Composite :

java.awt.Component java.awt.Button java.awt.Container …



À la place de

Component (Java 2 Platform SE 5.0) - Mozilla Firefox

Fichier Edition Affichage Aller à Marque-pages Outils ?

file:///c:/Program%20Files/Java/jdk-1_5_0-doc/docs/a OK dempsey

Dernières nouvelles... Pour afficher la pag...

ESIEE.BureauD.in4... | CNAM: SquirrelMail ... | Sommaire des TP java | designpatterns.pa... | Component (Jav...

Java™ 2 Platfor
Standard Ed. 5.
All Classes

All Classes

AbstractAction
AbstractBorder
AbstractButton
AbstractCellEditor
AbstractCollection
AbstractColorChoo
AbstractDocument
AbstractDocumen
AbstractDocumen
AbstractDocument
AbstractExecutorS
AbstractInterruptib
AbstractLayoutCa

Overview Package Class Use Tree Deprecated Index Help *Java™ 2 P Standard*

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.awt

Class Component

```
java.lang.Object
  └java.awt.Component
```

All Implemented Interfaces:
ImageObserver, MenuContainer, Serializable

Direct Known Subclasses:
Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent

Rechercher : Occurrence suivante Occurrence précédente Surligner Respecter la casse

Terminé none

Pattern – Patrons, sommaire

- **Historique**
- **Classification**
- **Les fondamentaux …**
- **Quelques patrons en avant-première**
  - **Adapter, Proxy**

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

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] avant tout

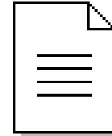
- **Constructions**
  - **Delegation**
  - **Interface**
  - **Abstract superclass**
  - **Immutable**
  - **Marker interface**

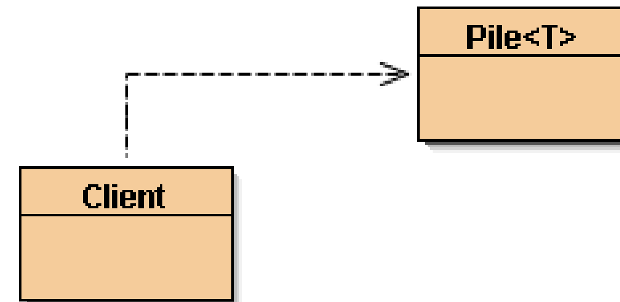
Delegation

- **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…

```java
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);
   }
 …}
```

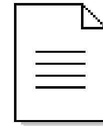


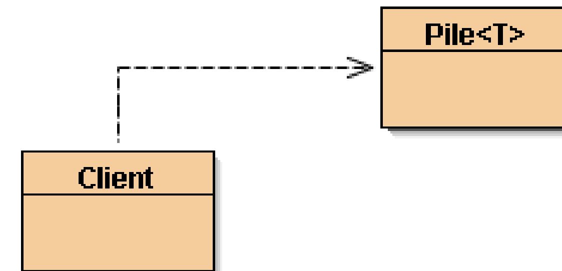



```java
public class Client{
  public void main(String[] arg){
    Pile<Integer> p = new Pile<Integer>();
    p.empiler(4);
    …
   }
 }
```

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

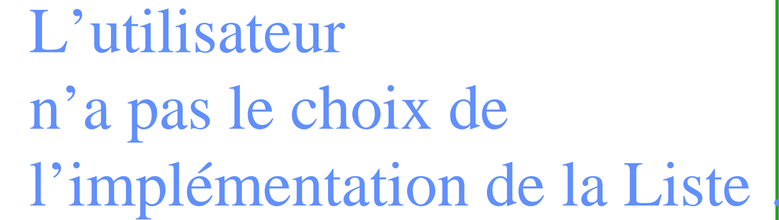
- **Discussion… entre nous**

- **Avantage à la délégation**

mais

Délégation : une critique

```
public class Pile<T>{

  private final List<T> stk;

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

 …}
```



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());
}
```

Usage des interfaces

un filtre : si la condition est satisfaite alors retirer cet élément

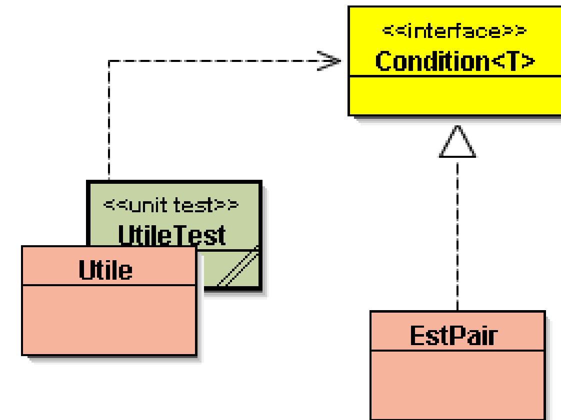
```
public static
<T> void filtrer( Collection<T> collection,
                  Condition<T> condition){

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

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

Collection et Condition sont des interfaces

discussion

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);
```

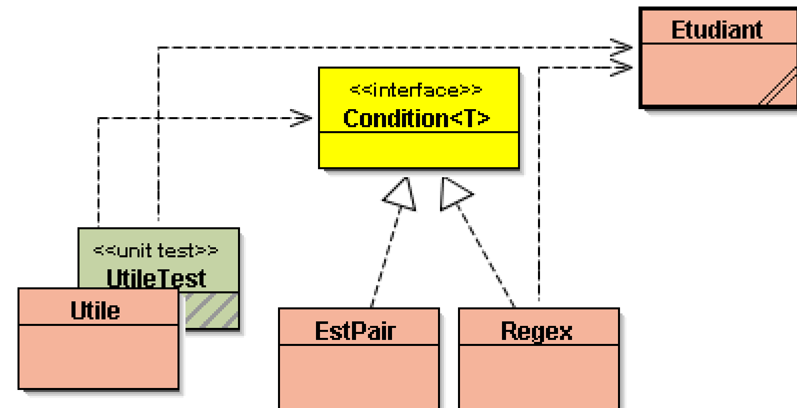
BlueJ: BlueJ : Terminal - filtre

Options

```
liste : [3, 4, 8, 3]
liste : [3, 3]
```

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

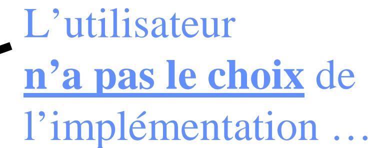
Options

```
set : [juan, paul, pierre]
set : [juan]
```

discussion

Délégation : une critique, bis

```
public class Pile<T>{

   private final List<T> stk;

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

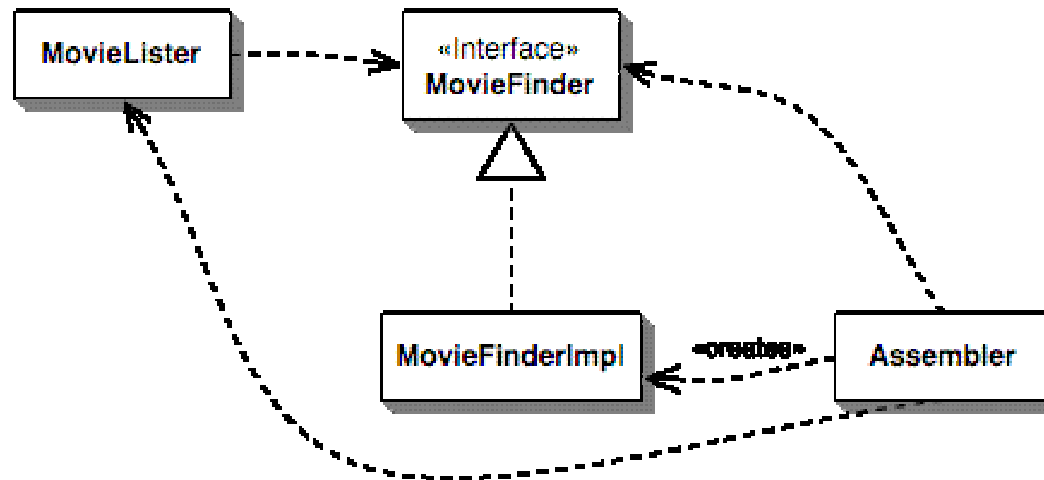



```
public class Pile<T>{
   private final List<T> stk;

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

Vocabulaire : 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 »

Abstract superclass

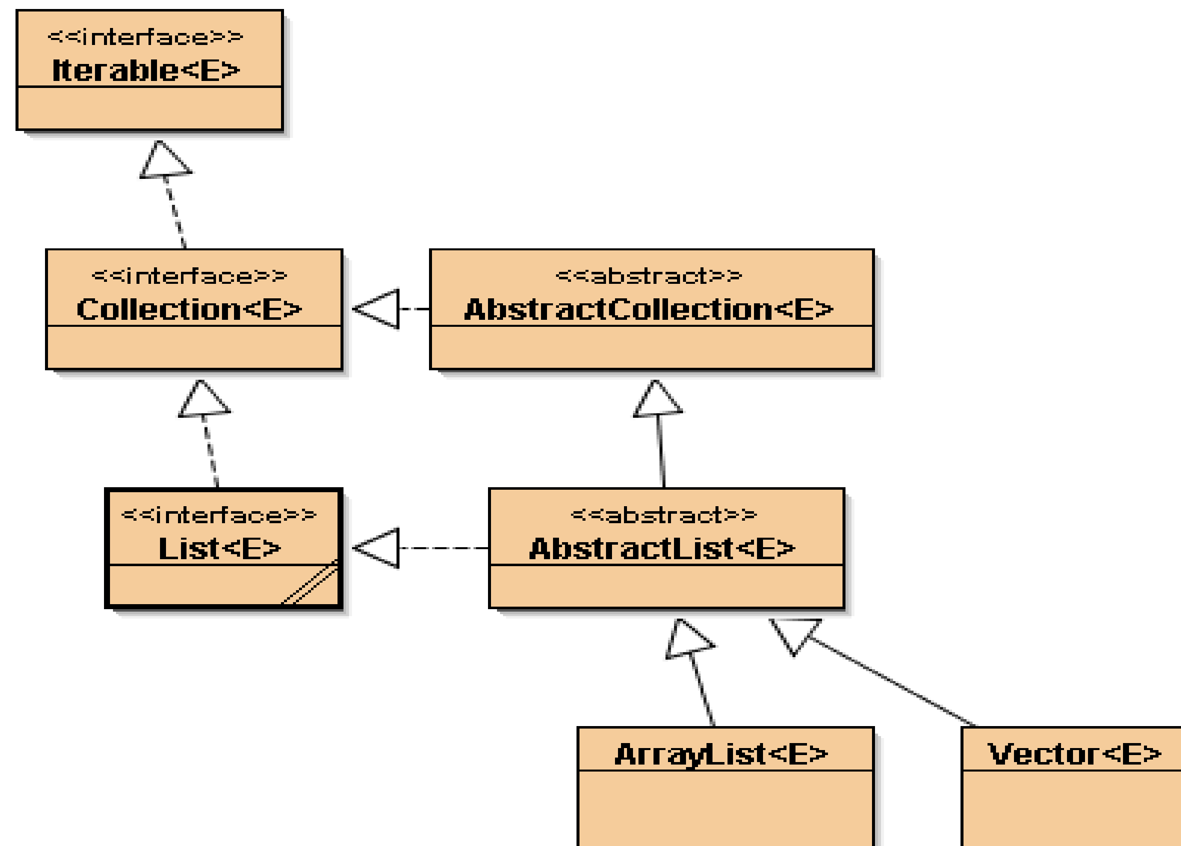
- **Construction fréquemment associée à l'Interface**
  - **Une classe propose une implémentation incomplète**
    - **abstract class en Java**
  - **Apporte une garantie du « bon fonctionnement » pour ses sous-classes**
  - **Une sous-classe doit être proposée**
  - **Souvent liée à l'implémentation 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**

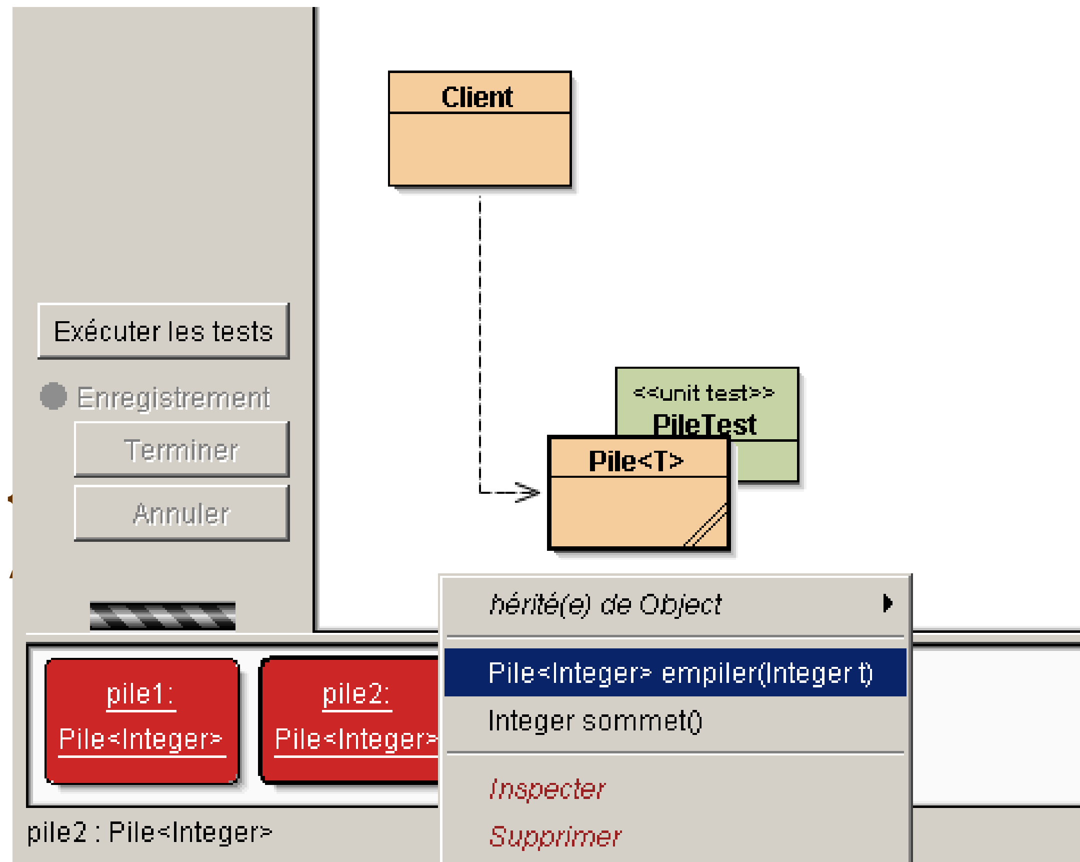


Immutable

- **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
- **java.lang.String est « Immutable »**
  - Contrairement à 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

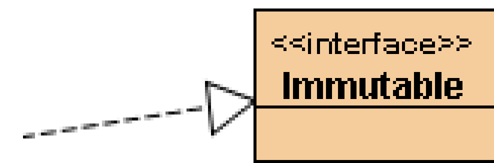
- **Une interface vide !**
  - **Classification fine des objets**
  - **implements installée sciemment par le programmeur**

  - **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 »**

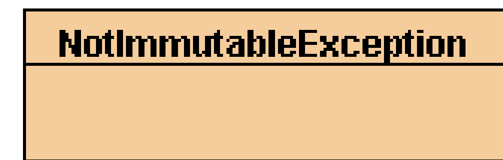
    - **Note : Les annotations de Java peuvent remplacer « élégamment » cette notion**

Marker Interface : exemple

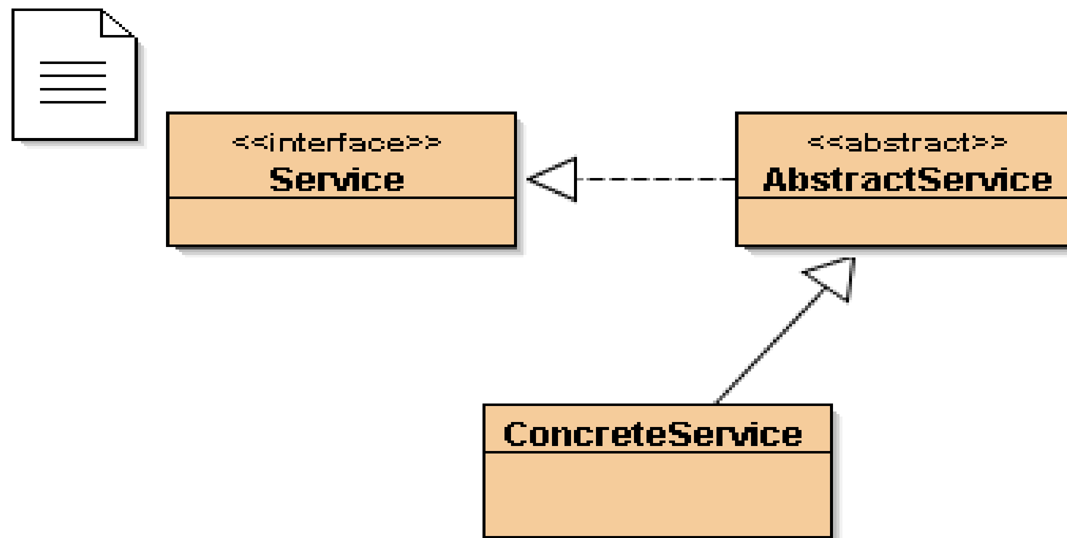
```
public interface Immutable{}

public class NotImmutableException
    extends RuntimeException{
  public NotImmutableException(){super();}
  public NotImmutableException(String msg){super(msg);}
}

public class Pile<T> implements Immutable{
...
}
```




```
Pile<Integer> p = new Pile<Integer>();
if(!(p instanceOf Immutable))
  throw new NotImmutableException();
…
```

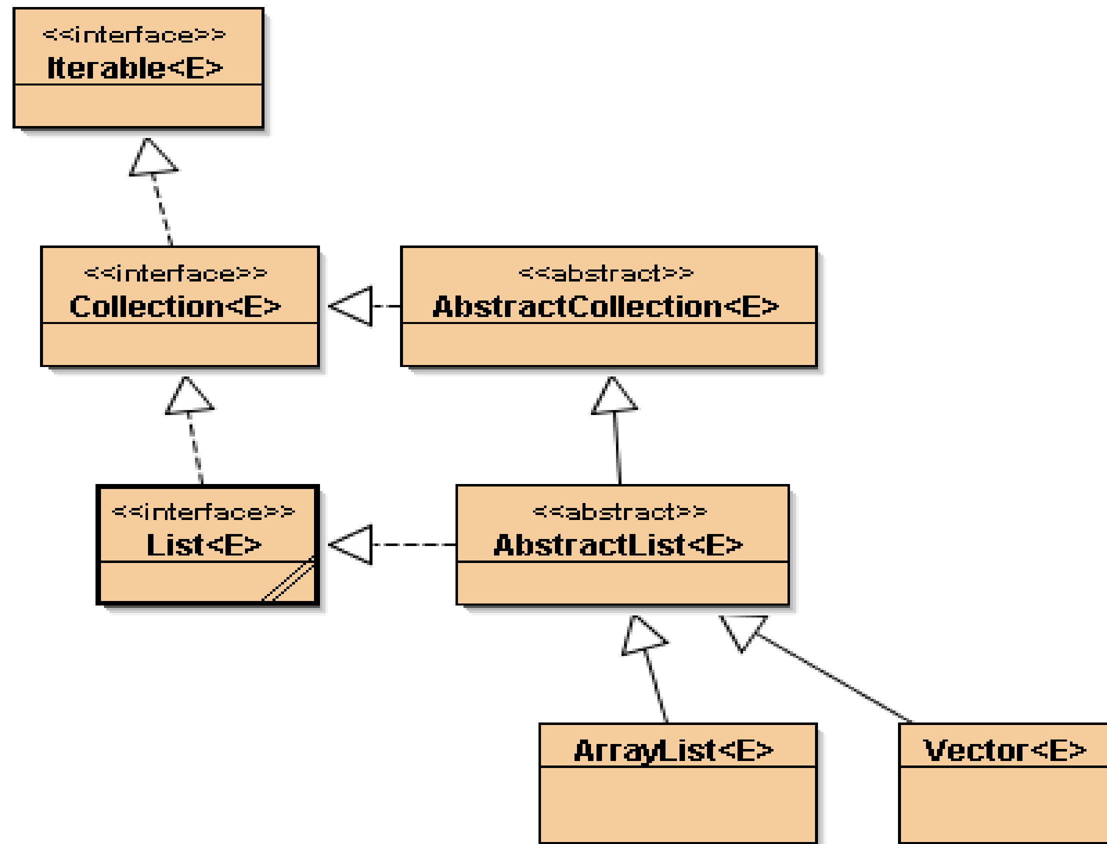


Interface & abstract



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

Interface & abstract



– **Déjà vu …**

Les 23 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

Deux patrons pour l'exemple…

- **Dans la famille "Patrons Structurels"**

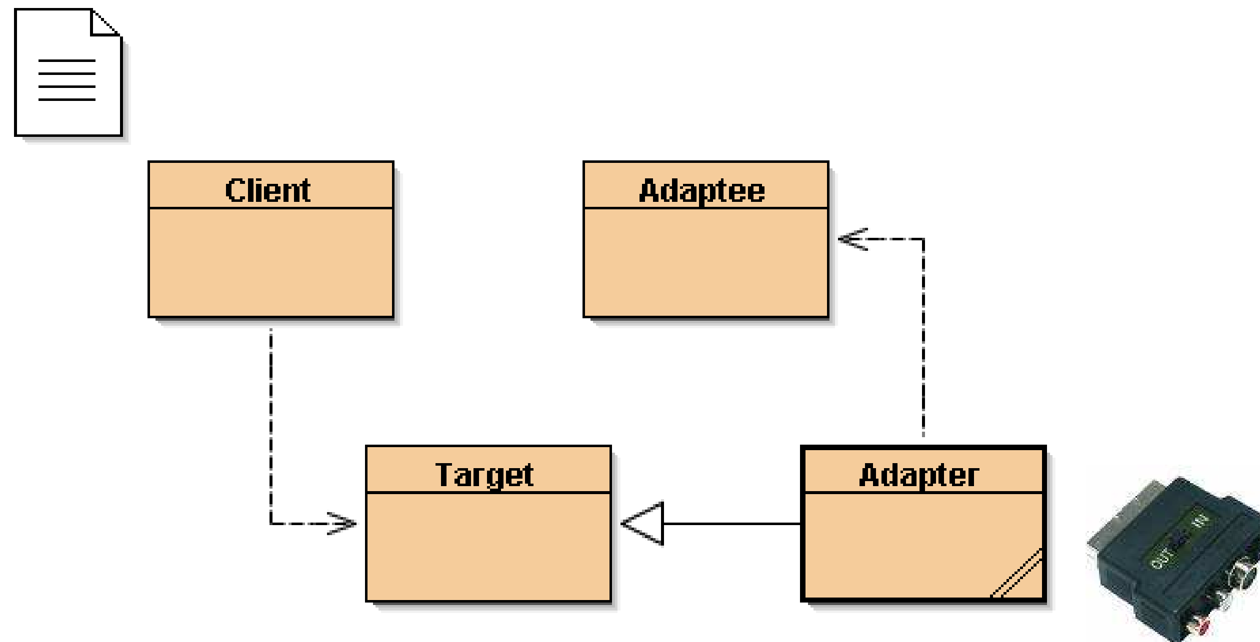
- **Adapter**
  - **Adapte l'interface d'une classe conforme aux souhaits du client**

- **Proxy**
  - **Fournit un mandataire au client afin de contrôler/vérifier ses accès**

Adaptateurs

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

Pattern Adapter [DP05]



- **Le client a une péritel, mais nous ne disposons que d'une RCA**
  - **-> Un adaptateur**

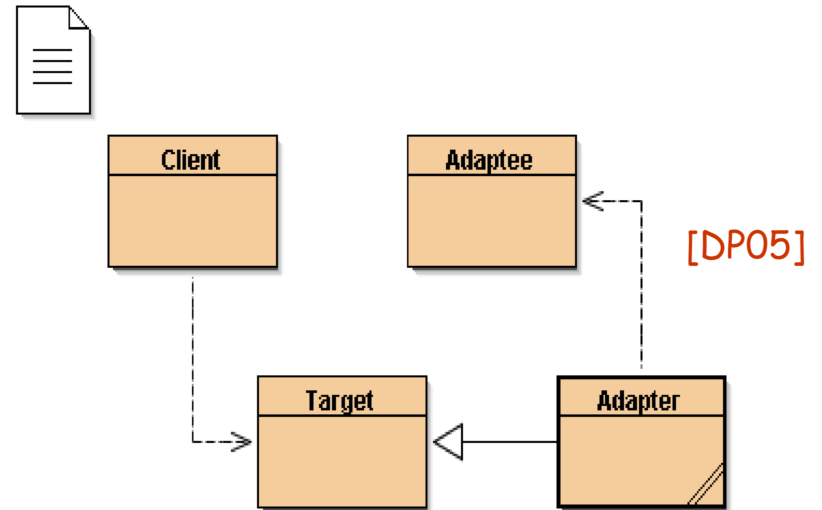
- **DP05 ou www.patterncoder.org, un plug-in de bluej**

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 …**

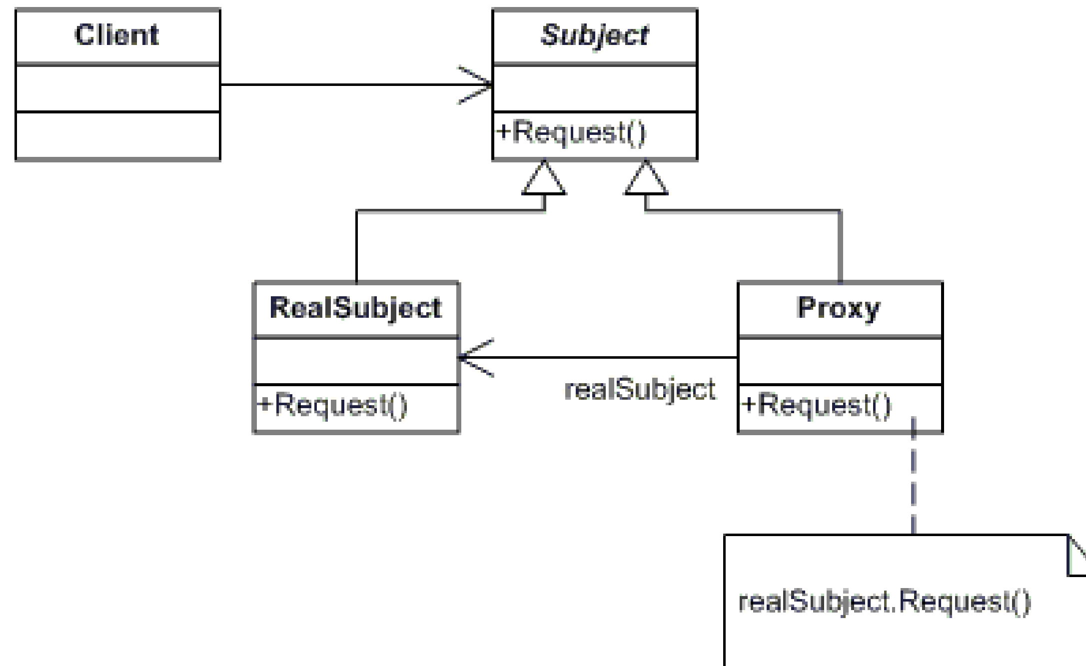
```
public Target newAdapter(final Adaptee adaptee){
    return
      new Target(){
        public void serviceA(){
          adaptee.serviceB();
        }
      };
    }
```

- **Un classique …**

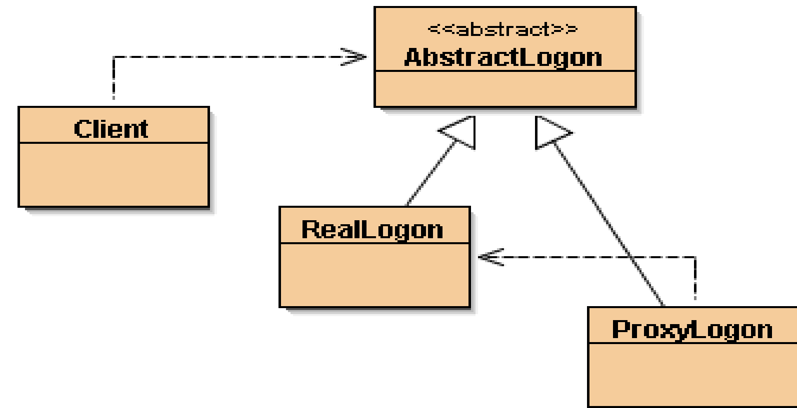
```
WindowListener w = new WindowAdapter(){
        public void windowClosing(WindowEvent e) {
          System.exit(0);
         }
       };
```

Pattern Proxy

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

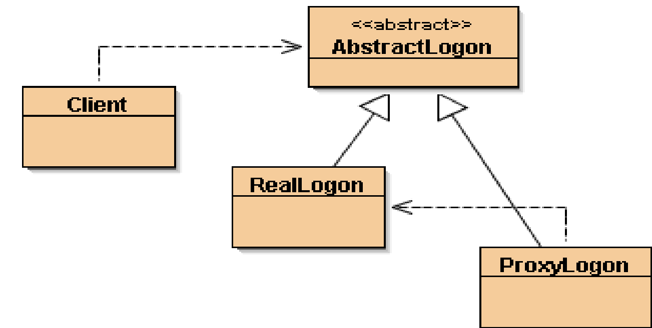


Proxy : un exemple




```
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();
   …
  }
}
```

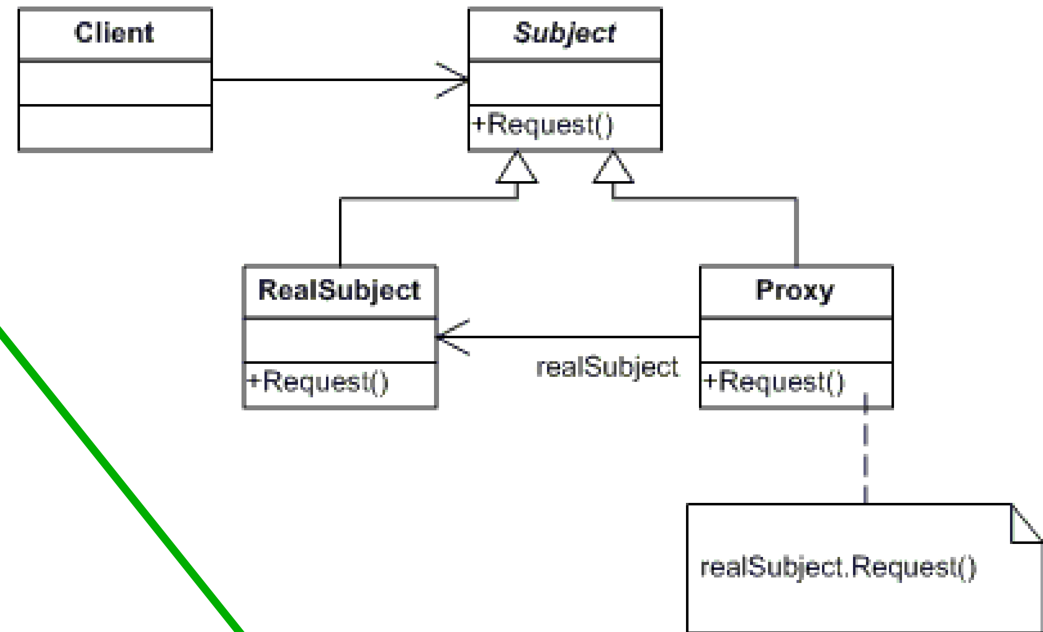
Proxy : exemple suite

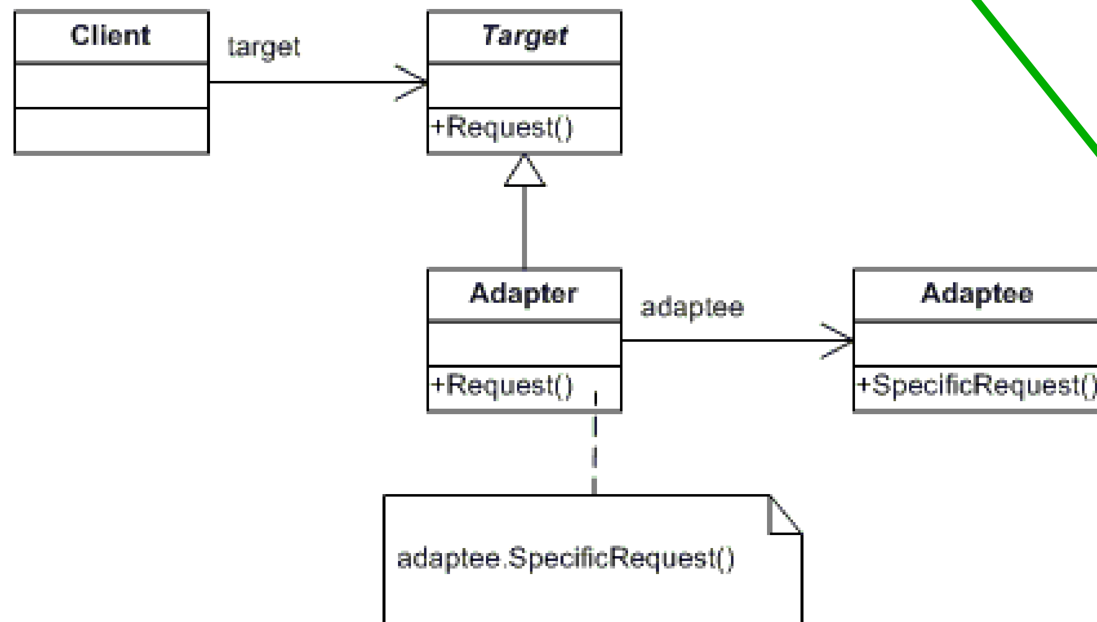



```java
public class ProxyLogon extends AbstractLogon{
  private AbstractLogon real = new RealLogon();

  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;
  }
}
```

Adapter\Proxy

- **discussion**

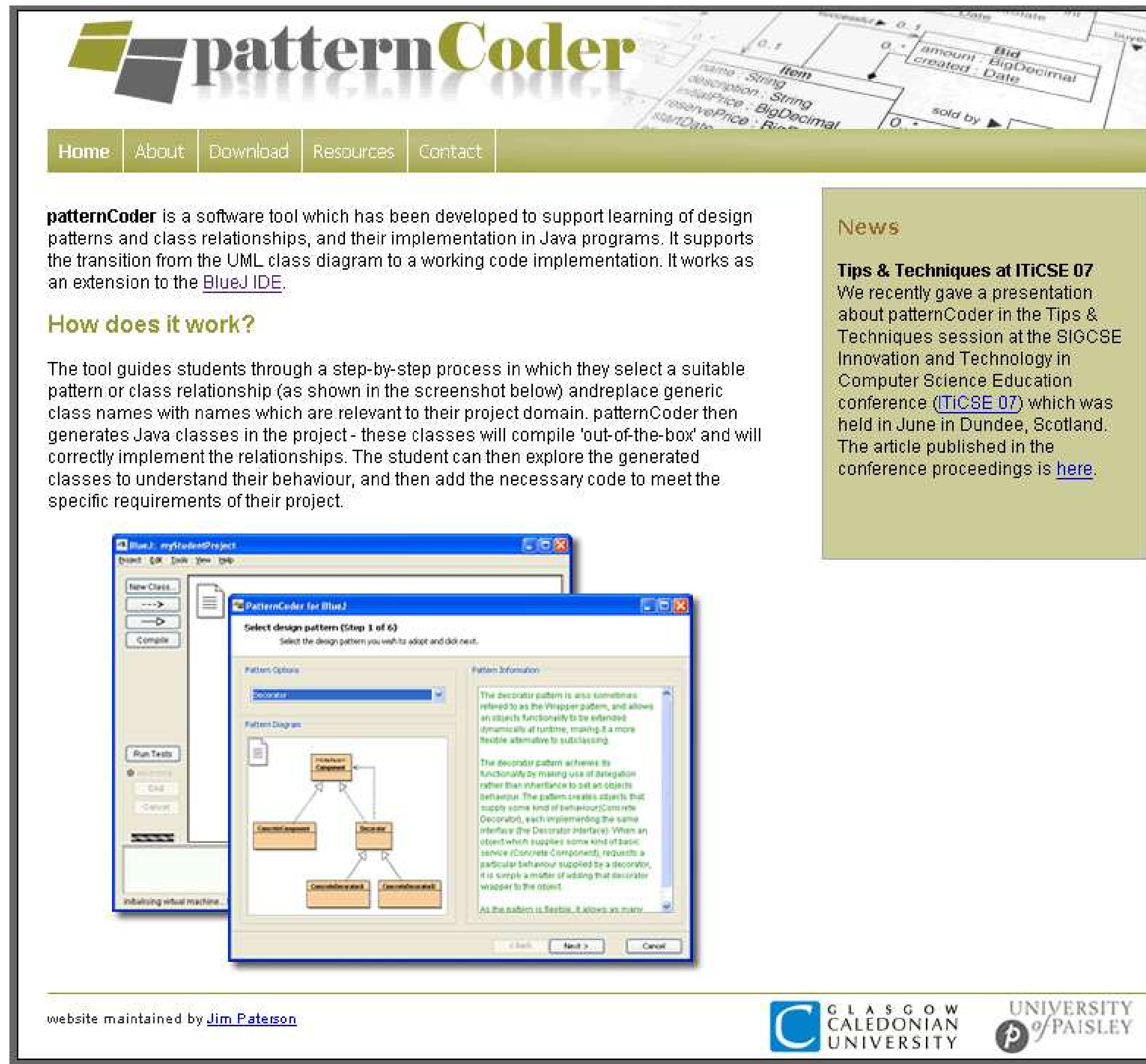




Conclusion

- **Est-ce bien utile ?**
- **Architecture décrite par les patterns ?**
- **Langage de patterns ?**
- **Comment choisir ?**
- **Trop de Patterns ?**
- **Méthodologie d'un AGL ?**

BlueJ : www.patterncoder.org



- **Démonstration : le patron Adapter**