

---

# Cours 0-1 : Introduction

jean-michel Douin, douin au cnam point fr  
version : 7 Janvier 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;  
    }  
}
```

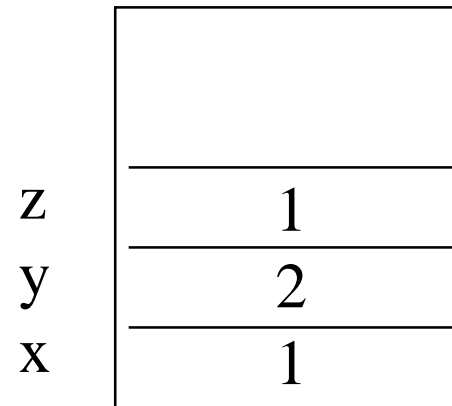
Note : C# apparenté Java ...

# Sûr par l'absence de pointeurs accessibles au programmeur

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

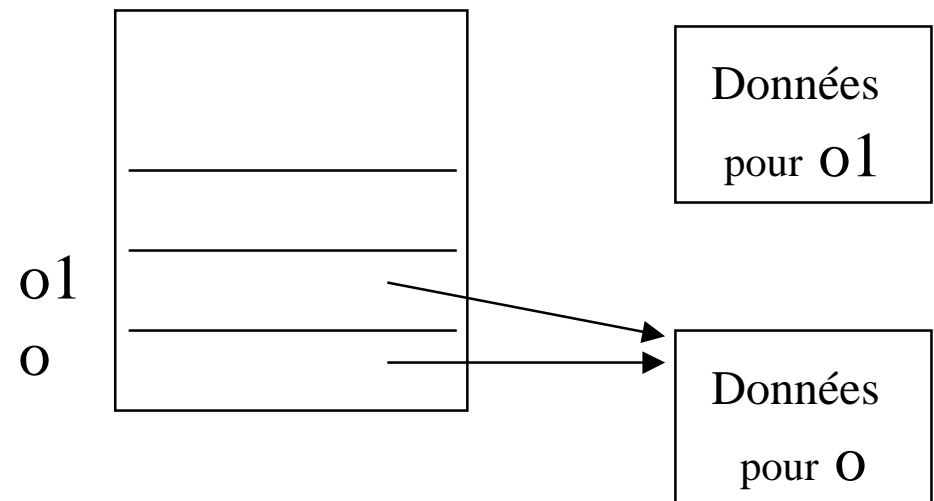
- **primitif :**

- `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
- **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

Le source Java  
*Num.java*

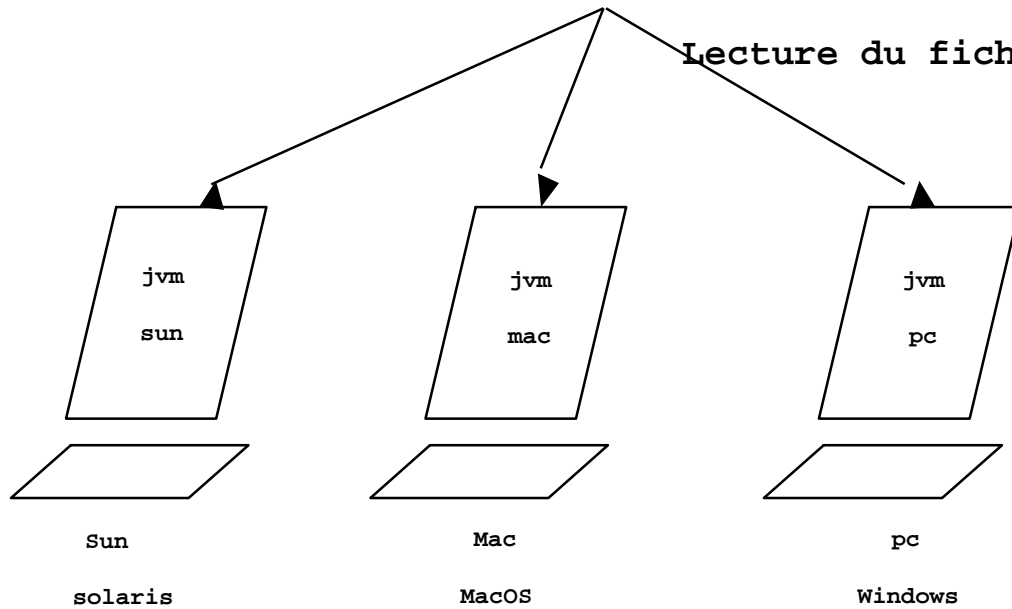
```
public class Num {  
  ...  
}
```

1) **compilation**

Le fichier compilé  
*Num.class*

```
1100 1010 1111 1110 1011 1010 1011 1110  
0000 0011 0001 1101 .....
```

Lecture du fichier locale ou distante



2) **interprétation, exécution**



# 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)**
- **Etat 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
- **A** + **D** = **P**      langage modulaire, Ada, modula-2
- **a** + **D** = **P**      langage Orienté Objet

$$A + d = P$$

---

- **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)**

$$A + D = P$$

---

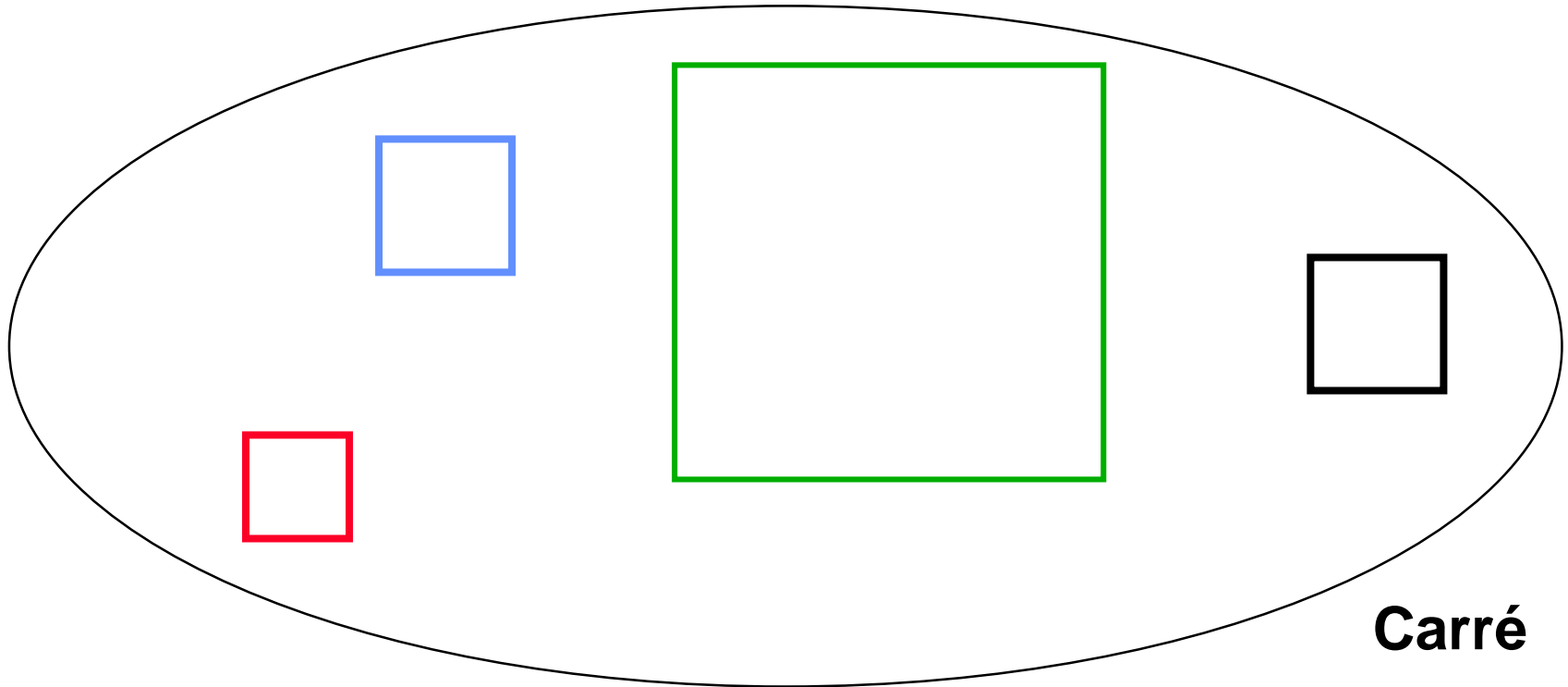
- classe **Carré** =
- longueurDuCote ...
- surface ( ) =
- perimetre ( ) =
- ..... ( ) =
- fin\_classe;
- *usage : import de la classe Carré puis*
  - carré unCarré; // une instance de la classe Carré
- **y = unCarré.surface ( )**

# Classe et objet (instance d'une classe)

---

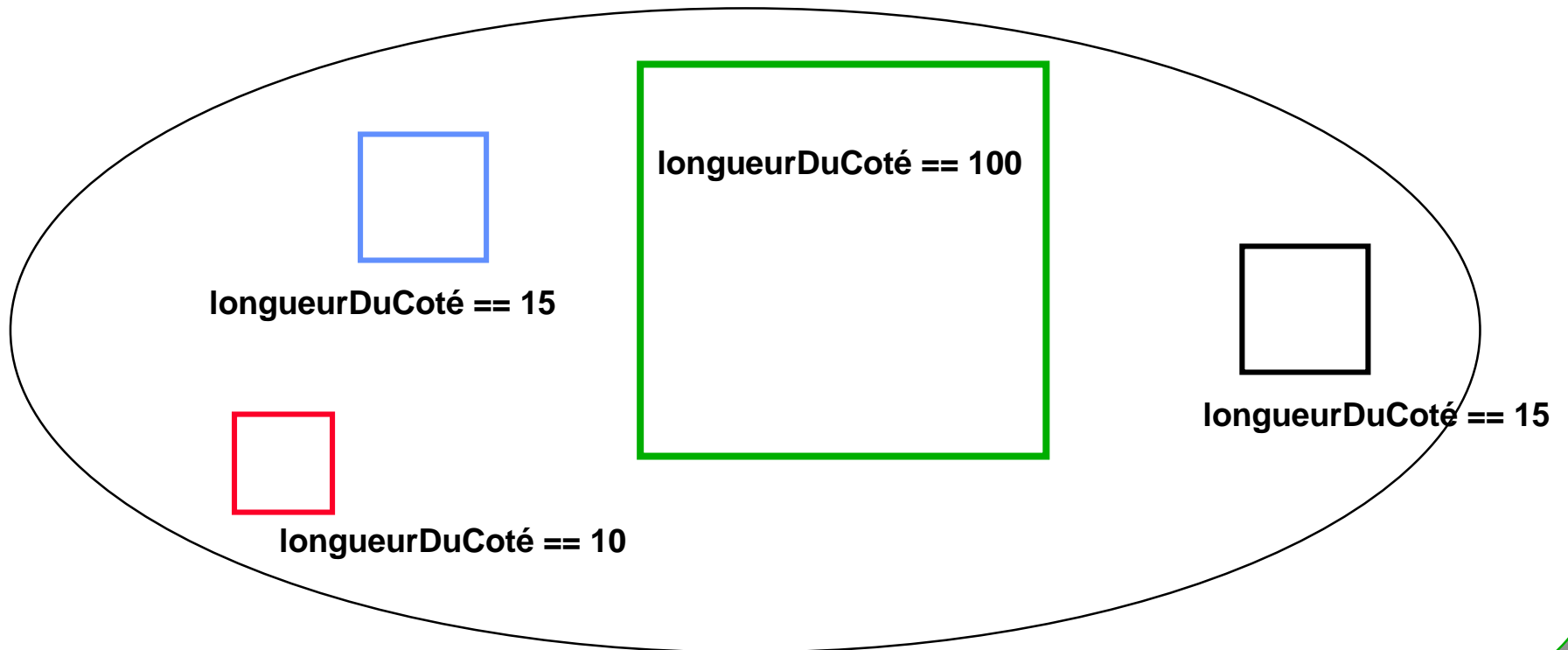
**class Carré{**

**}**



# Etat d'un objet et données d'instance

```
class Carré{  
  int longueurDuCoté;  
  Color couleur;  
}
```

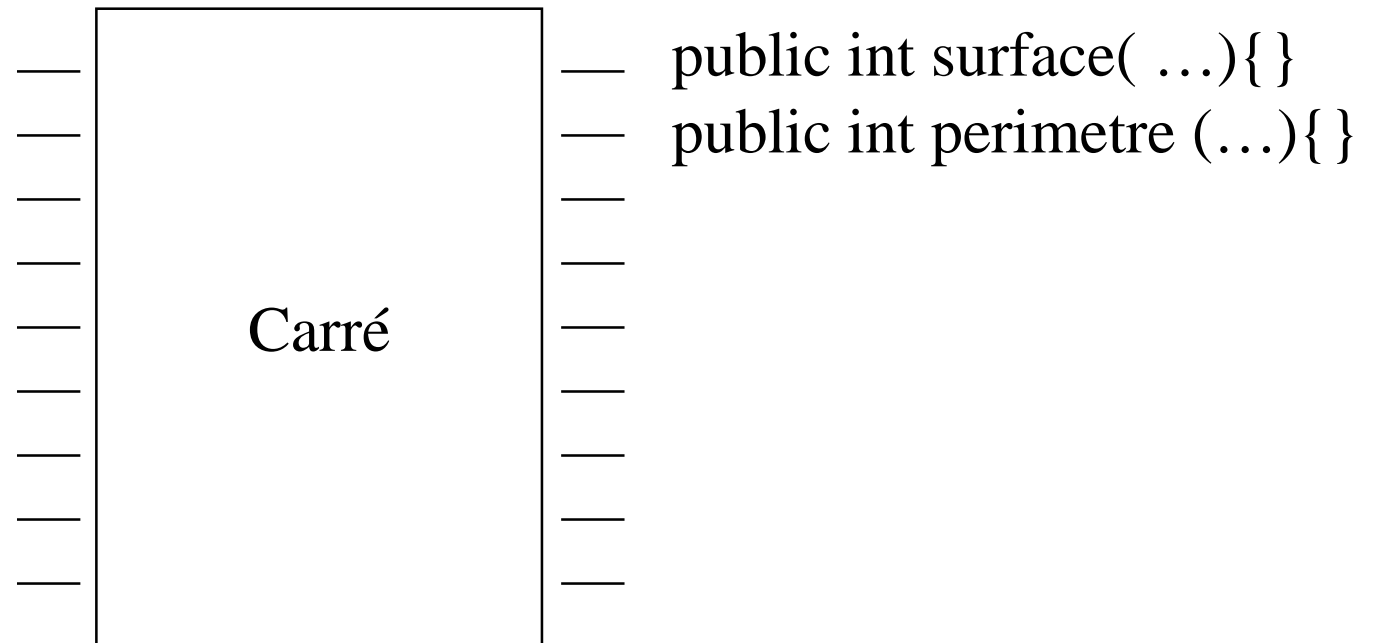




# Classe et Encapsulation

---

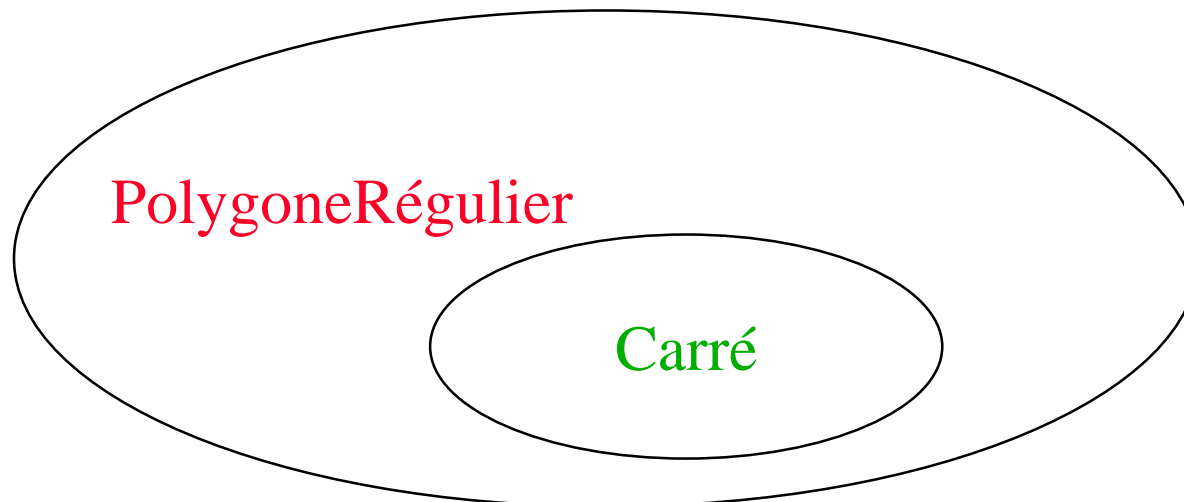
- **contrat avec le client**
  - interface publique
  - implémentation privée



# Héritage et classification

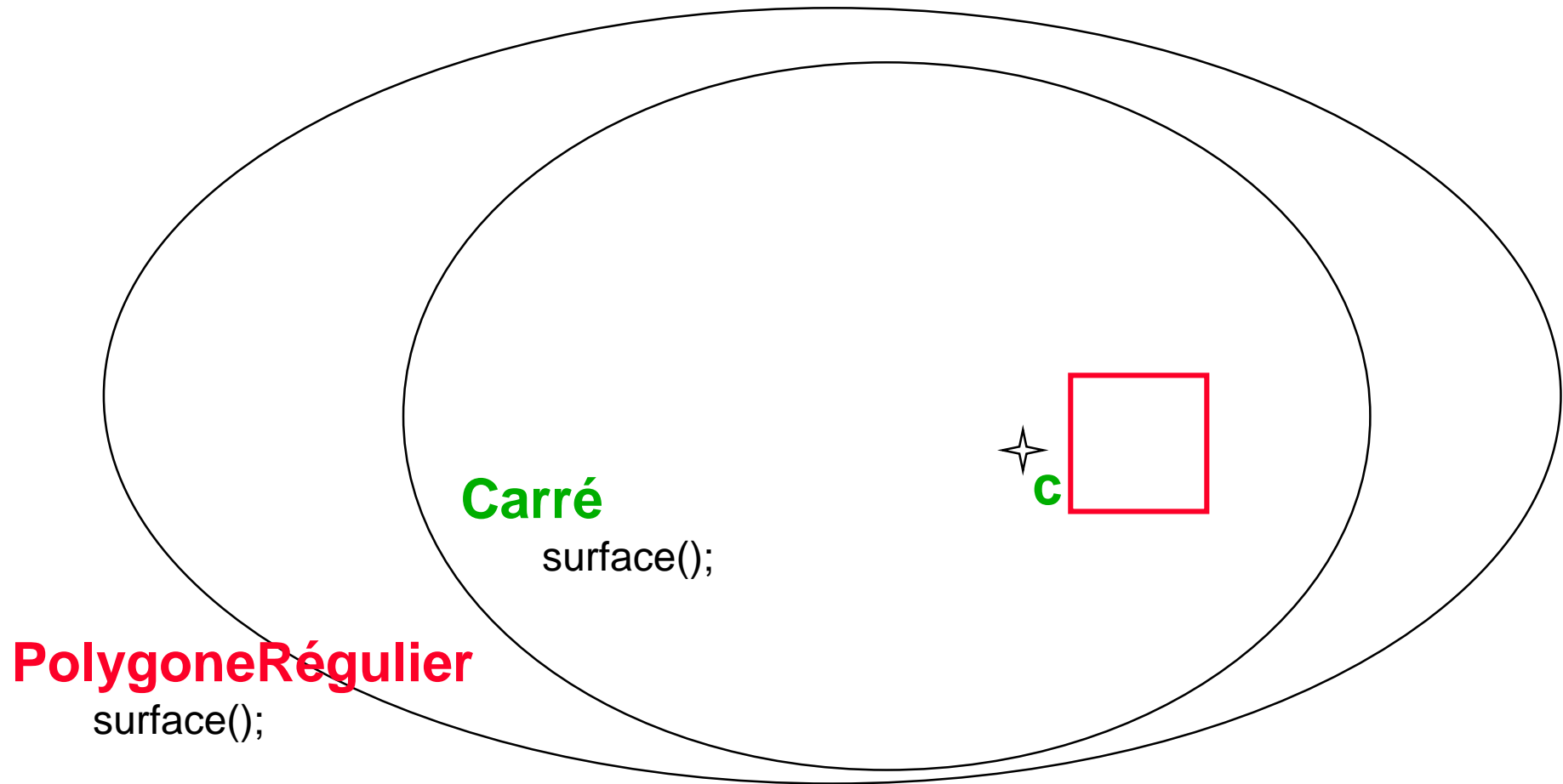
---

- **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...)



# Comportement d'un objet et méthodes

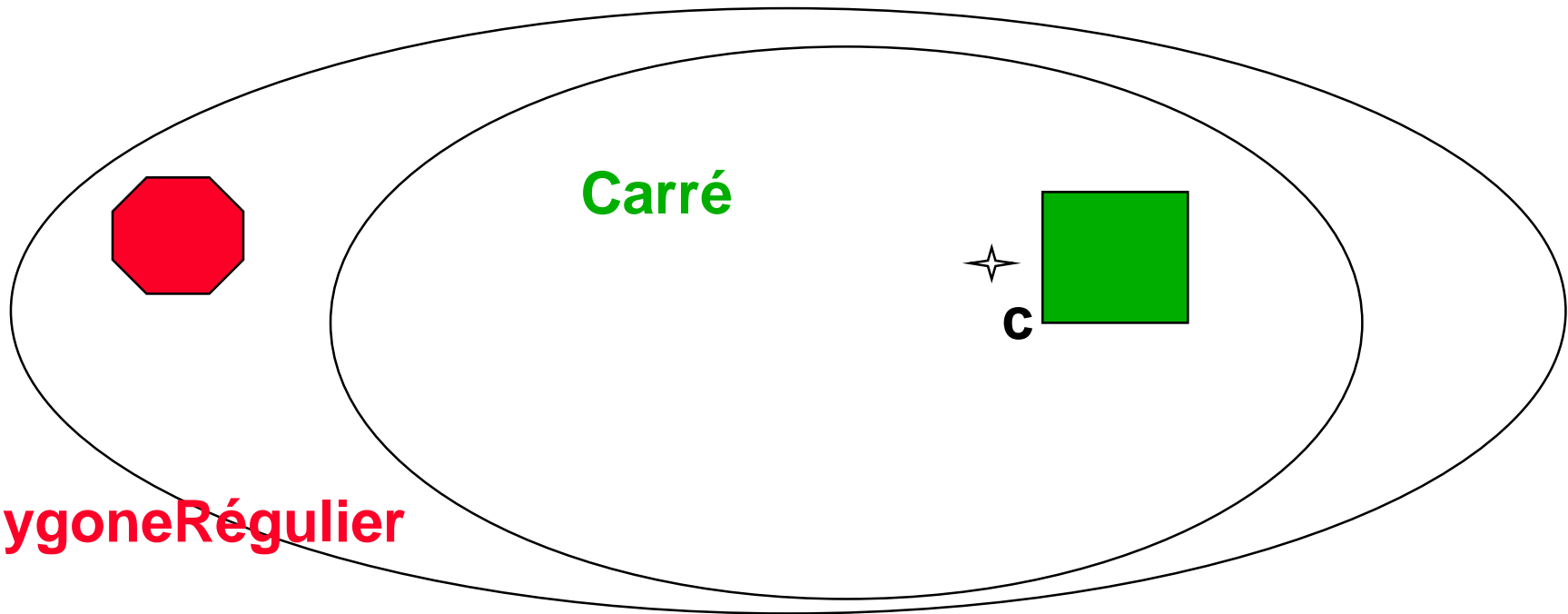
---



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

# Les carrés sont des polygones, attention

---



- Les carrés sont de couleur verte
- Les polygones réguliers sont rouges
- ? Couleur d'un Polygone Régulier 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 (overriding),**
- 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(3);`    `out.print(3.0);`    `out.print("bonsoir");`

– *le choix de la méthode est **résolu statiquement** dès la compilation*

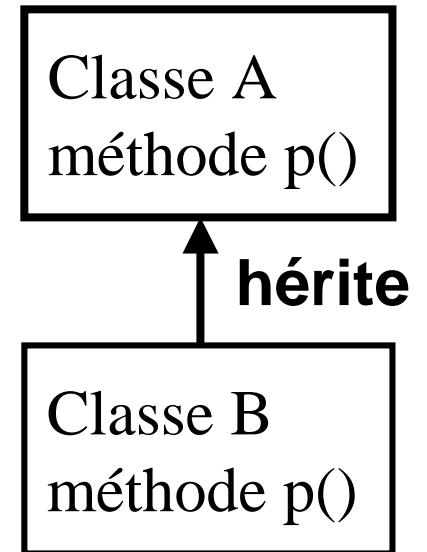
# Polymorphisme d'inclusion

- **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

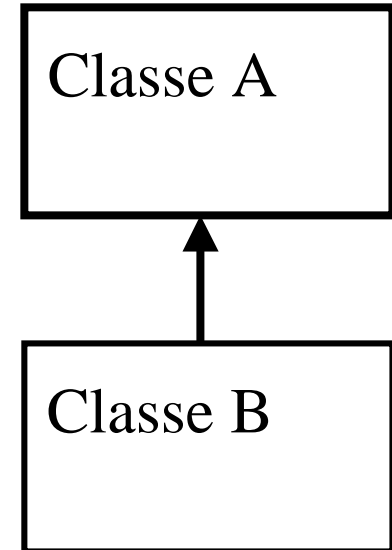
- Une liste homogène

- ```
public 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, c2 est un autre nom pour c1*

- **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.
- Java guarantees that the apparent type of any expression is a supertype of its actual type.

Ces notes de cours utilisent

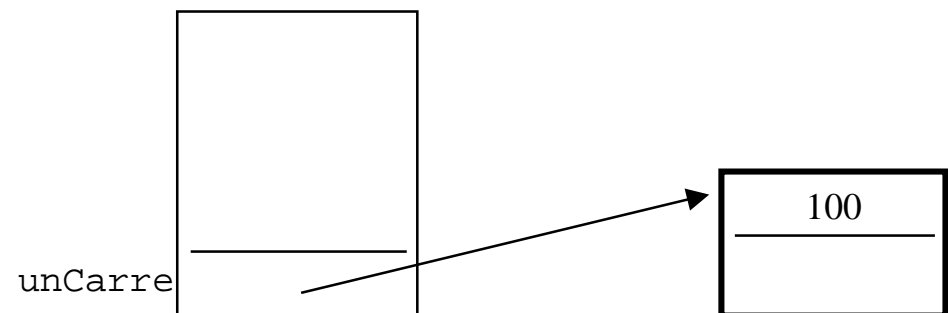
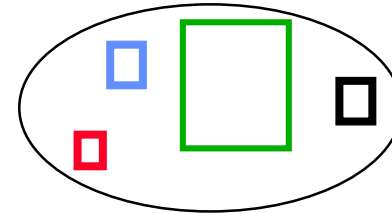
- type **déclaré** pour apparent type et
- type **constaté** pour actual type

# Un exemple en Java : la classe Carre

```
public class Carre {  
    private int longueurDuCote;  
  
    public void initialiser(int longueur){  
        longueurDuCote = longueur;  
    }  
  
    public int surface(){  
        return longueurDuCote * longueurDuCote;  
    }  
  
    public int perimetre(){  
        return 4*longueurDuCote;  
    }  
}
```

// un usage de cette classe

```
Carre unCarre = new Carre();  
unCarre.initialiser(100);  
int y = unCarre.surface();
```

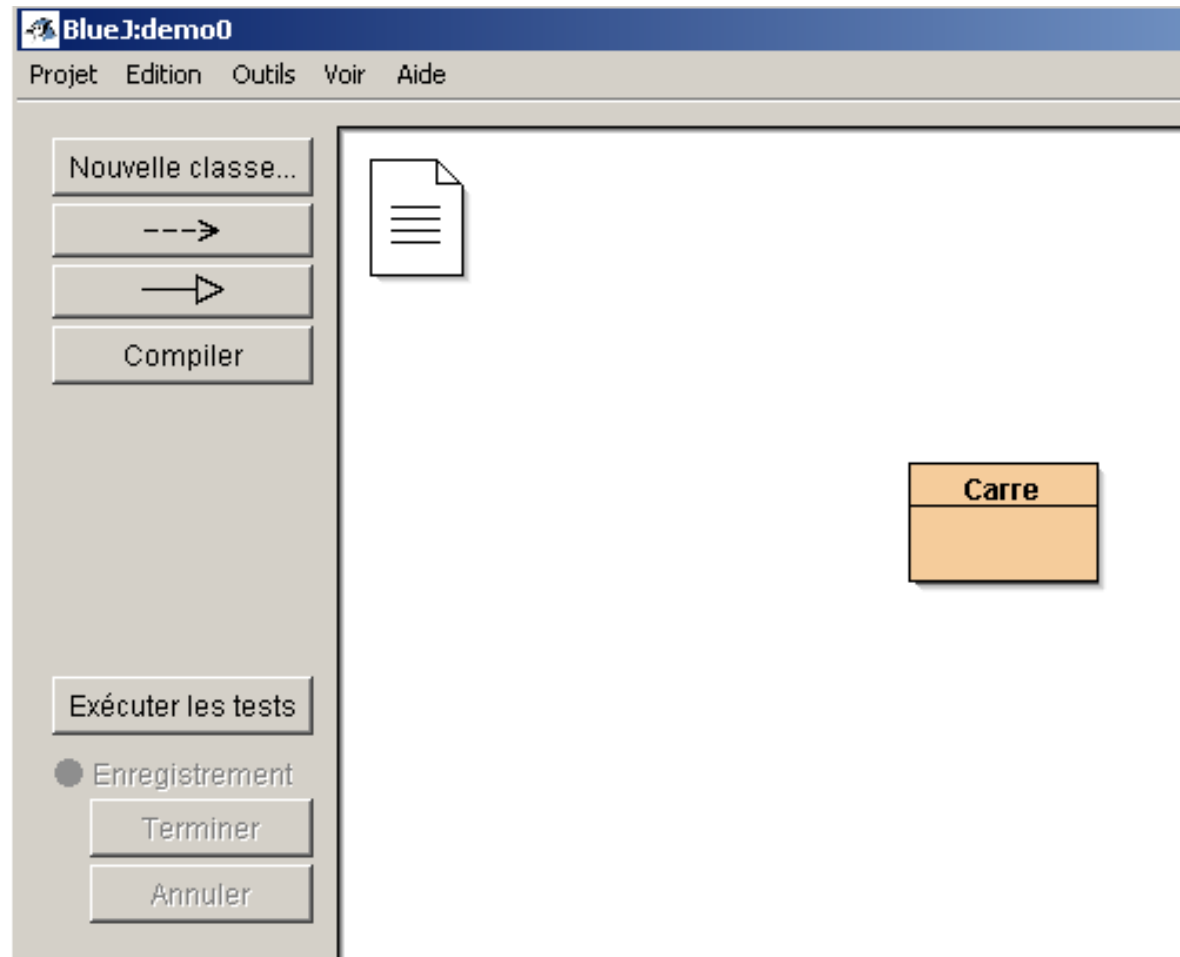


# Démonstration

---

- **Outil Bluej** <http://www.bluej.org>
  - La classe Carré
  - Instances inspectées
  
- **Tests Unitaires** <http://www.junit.org>
  - La Classe CarréTest
    - Ou la mise en place d'assertions pertinentes
  
- **Outil JNEWS** <http://jfod.cnam.fr/jnews/>
  - Java New Evaluation Web System

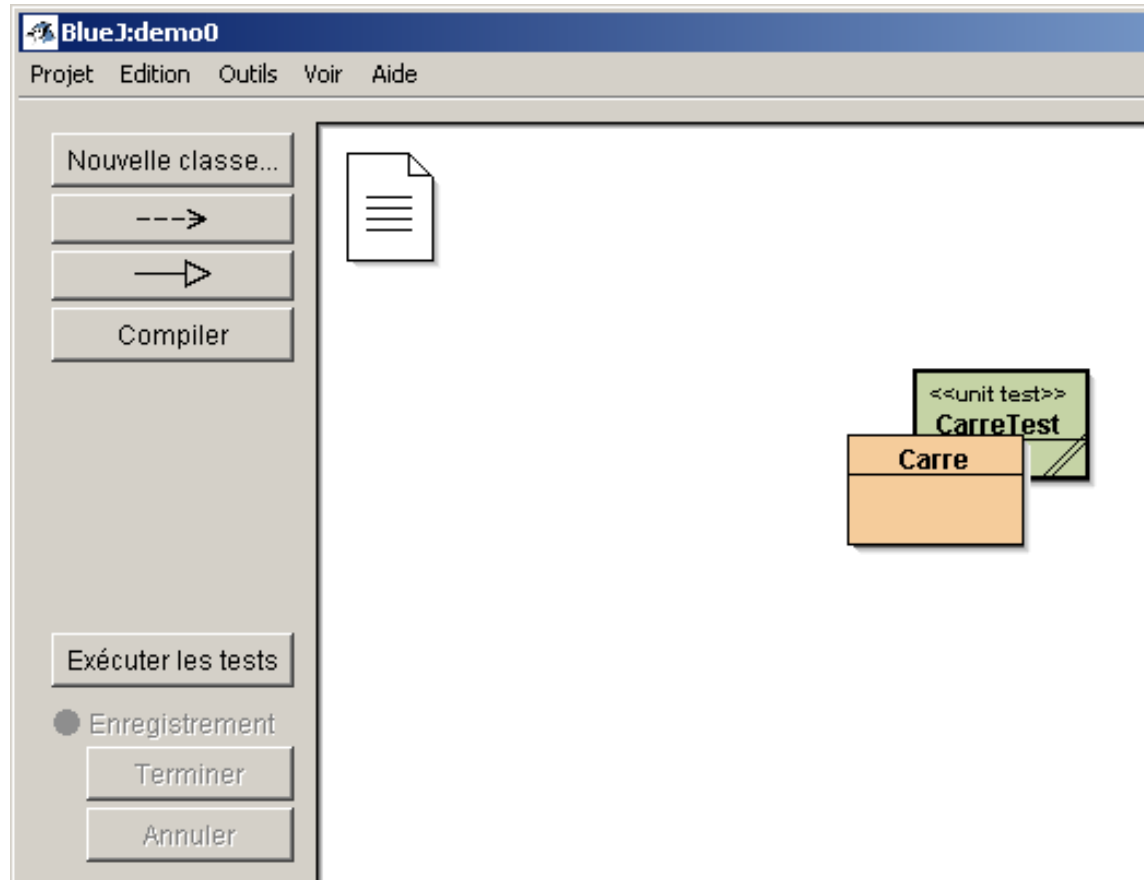
# Demo : Bluej



- **Instances et leur inspection**



# Demo : Bluej + tests unitaires



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

# Tests unitaires : outil *junit* intégré

- <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);
```

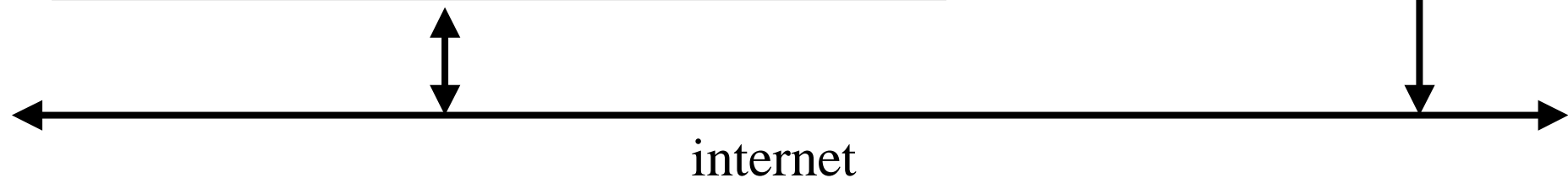
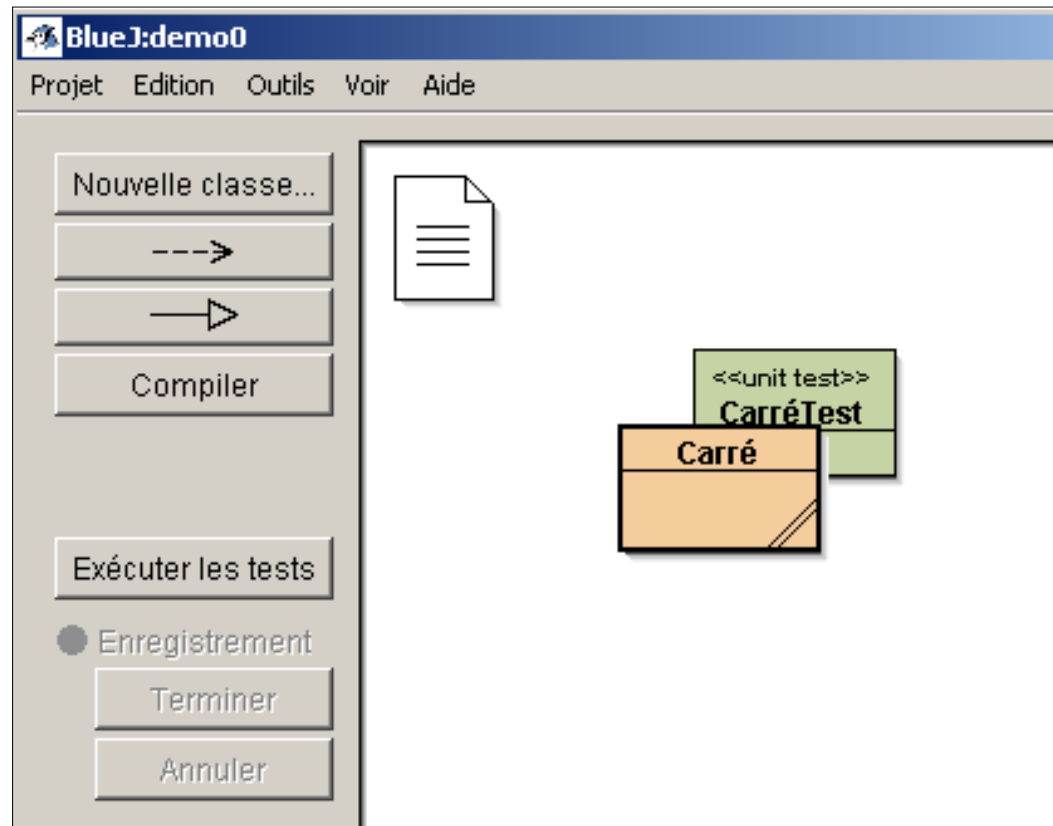
```
assertNotNull(" un commentaire ???" ,un object);
```

```
...
```

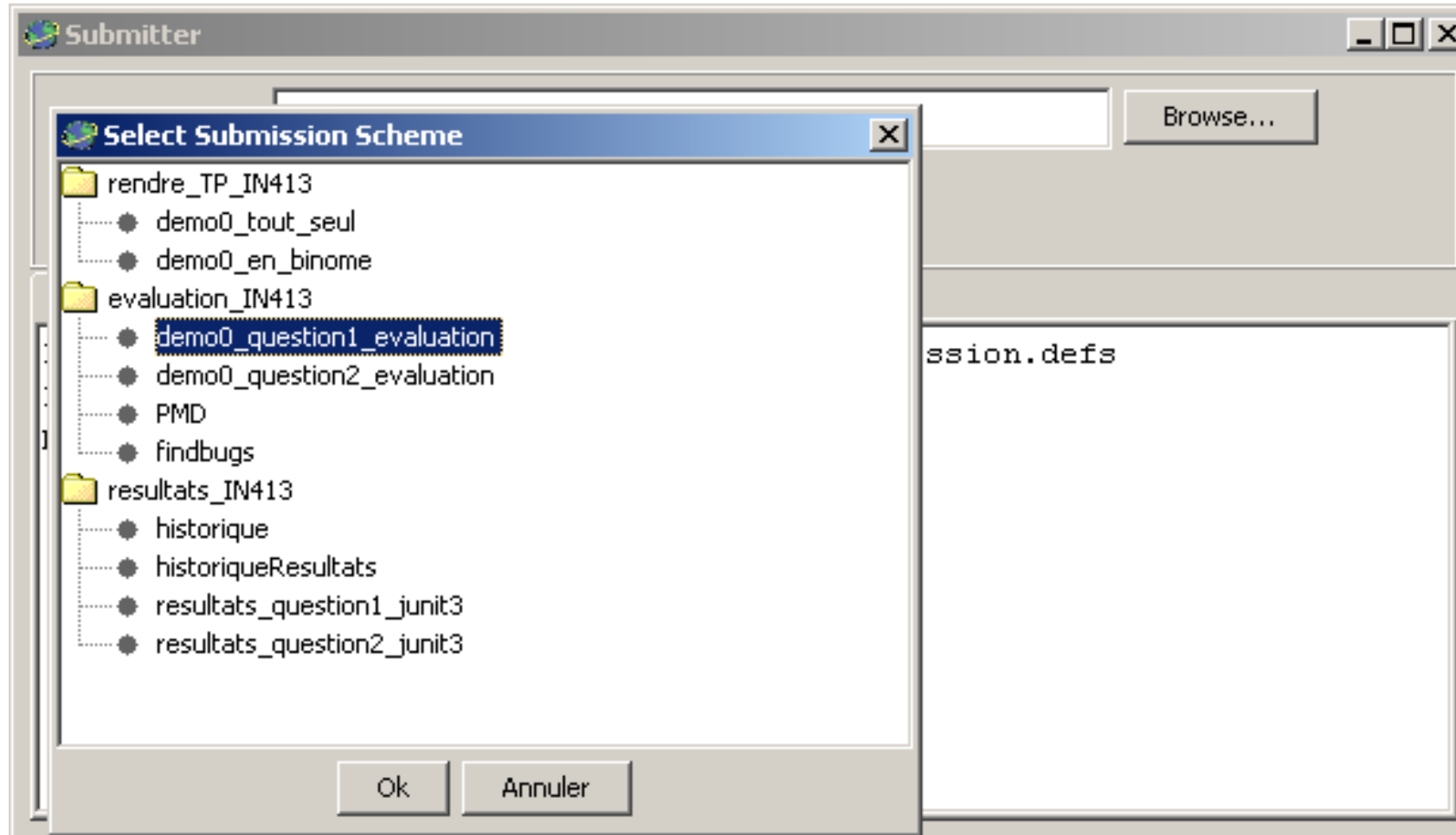
Le *commentaire* est affiché lorsque l'assertion échoue

# JNEWS contient des Tests unitaires distants

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



# JNEWS : outil *Submitter...* intégré



- **Un clic suffit**

# Démonstration : JNEWS Bravo!!

The screenshot shows a 'Submitter' window with a 'Scheme' field containing 'evaluation\_IN413/demo0\_question1\_evaluation' and buttons for 'Submit' and 'Annuler'. Below this, a 'Submission Result' dialog box is open, displaying the following information:

M/Mme/Melle test (IN413, demo0q1) le 03-09-07 à 14:47

**Résultat : 0 échec aux 2 tests<sup>1</sup> référents**

**Bravo !!**

**(0/6)<sup>2</sup>**

site : CEP. Au demo0q1, à ce jour : 1 succès / 1 utilisateur.

*JNEWS\_Cnam, pour Java New Evaluation Web Services est un service d'aide au développement attendu. Vos travaux sont évalués par l'exécution de classes de tests référentes sur le serveur <http://jod.cnam.fr/jnews/>, ces classes de tests utilisent ici junit3, tout comme votre environnement bluej.*

*Le nombre d'échecs aux tests référents représente le résultat de l'évaluation et un historique vous rappelle les derniers résultats obtenus. Un historique détaillé est également accessible depuis BlueJ, (menu Outils>Item submit puis Browse et JNEWS\_IN413/historique)*

*JNEWS\_Cnam est bien entendu perfectible, n'hésitez pas à nous envoyer vos critiques, souhaits d'améliorations, mesures supplémentaires, etc...*

*Bonnes soumissions avec JNEWS !*

[JNEWS\\_Cnam](#), Java New Evaluation Web Services du Cnam.

<sup>1</sup> Un test comporte plusieurs assertions, une assertion fautive induit un échec.

<sup>2</sup> Soit le (nombre d'essais antérieur / nombre maximal de tentatives).

+++ notez qu'il est préférable de fermer cette fenêtre par la case [X] en haut à droite plutôt que d'utiliser le bouton Ok...+++

Ok

# JNEWS à l'ESIEE, en intranet

---

- **Une aide à la réponse attendue**
  - +
  - **Outils en ligne comme PMD & findbugs**
  - +
  - **Remise planifiée des sources**
- <http://www.esiee.fr/~bureaud/Unites/In413/0708/Proc/procedure.htm>

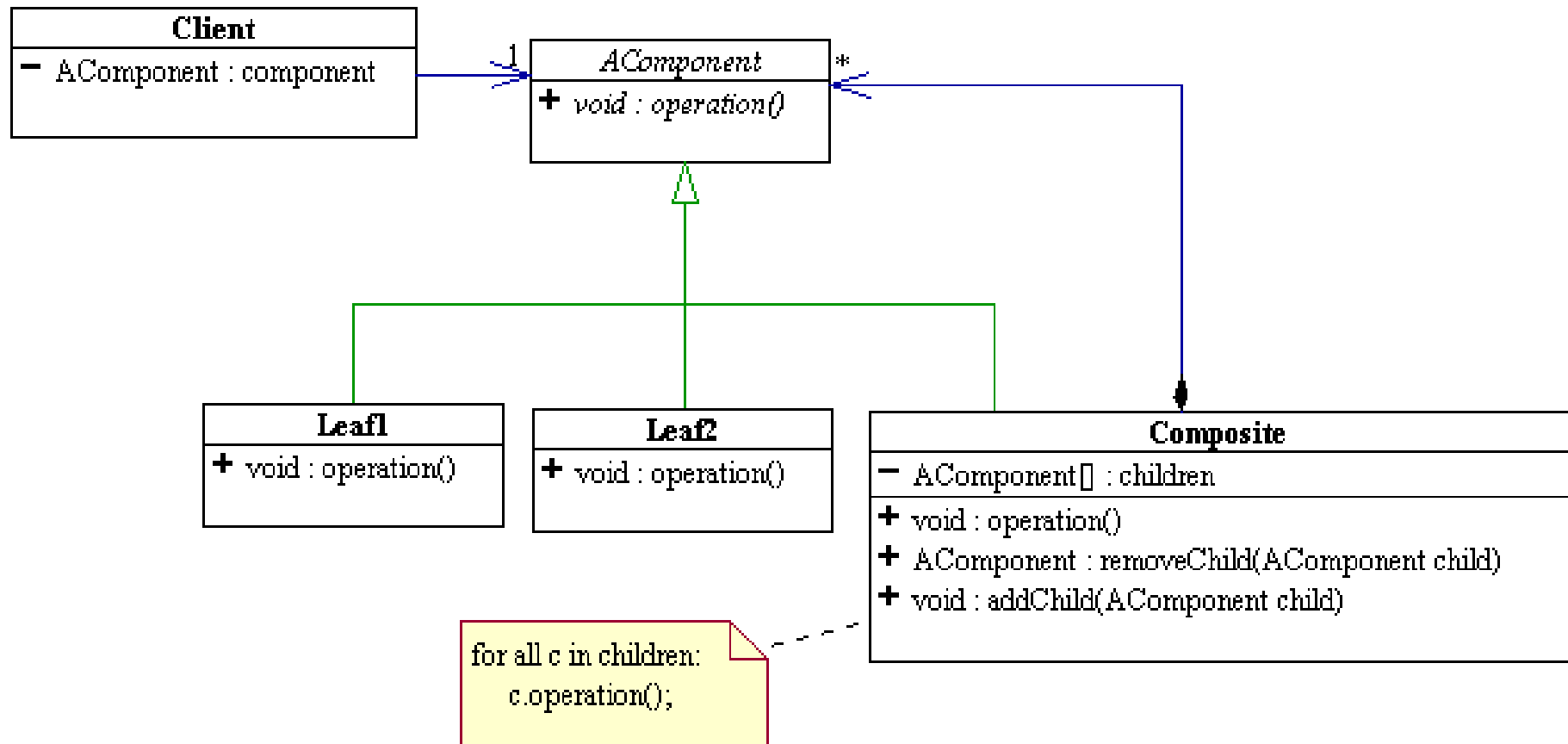
# Pattern pourquoi ?

---

- **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 événements de Java utilisent le **patron Observateur** ???
  - ...
  - etc. ...
- **Une application = un assemblage de plusieurs patterns**
  - Un rêve ?

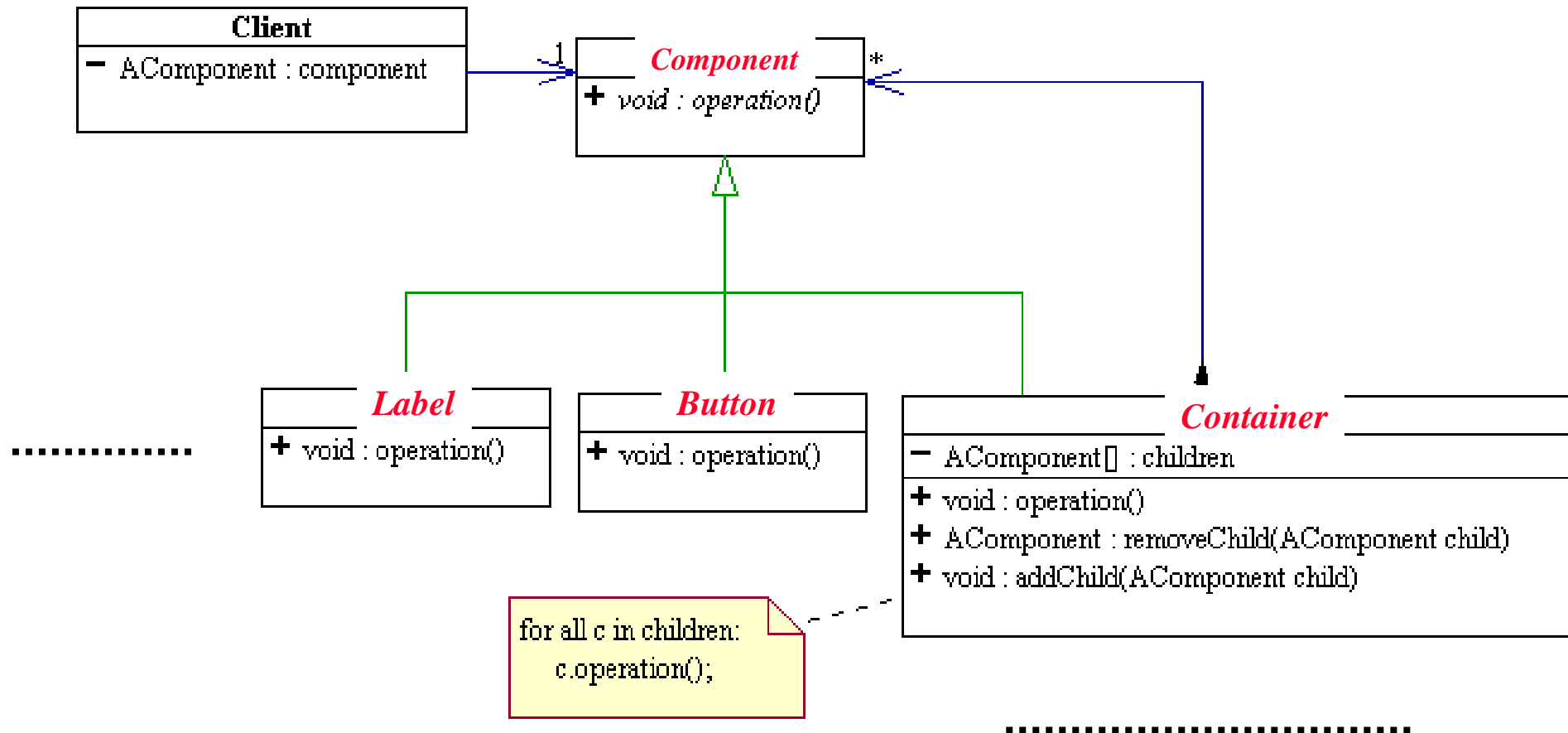
# La bibliothèque graphique du JDK utilise un composite ?

- Le pattern Composite ??? recherche sur le web ...





# la bibliothèque graphique utilise bien un Composite :



# À la place de

The screenshot shows a Mozilla Firefox browser window titled "Component (Java 2 Platform SE 5.0) - Mozilla Firefox". The address bar shows the file path: `file:///c:/Program%20Files/Java/jdk-1_5_0-doc/docs/a`. The browser is displaying the Java 2 Platform SE 5.0 documentation for the `java.awt.Component` class. The left sidebar shows a list of "All Classes" including `AbstractAction`, `AbstractBorder`, `AbstractButton`, `AbstractCellEditor`, `AbstractCollection`, `AbstractColorCho`, `AbstractDocument`, `AbstractDocument`, `AbstractDocument`, `AbstractDocument`, `AbstractExecutors`, `AbstractInterrupti`, and `AbstractLayoutCa`. The main content area shows the "Class Component" page with the following information:

- Navigation links: [PREV CLASS](#), [NEXT CLASS](#), [FRAMES](#), [NO FRAMES](#)
- Summary: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) | [DETAIL: FIELD](#) | [CONSTR](#) | [METHOD](#)
- Package: `java.awt`
- Class: `Class Component`
- Parent class: `java.lang.Object`
- Subclass: `java.awt.Component`
- All Implemented Interfaces: [ImageObserver](#), [MenuContainer](#), [Serializable](#)
- Direct Known Subclasses: [Button](#), [Canvas](#), [Checkbox](#), [Choice](#), [Container](#), [Label](#), [List](#), [Scrollbar](#), [TextComponent](#)

The bottom of the browser window shows a search bar with the text "Rechercher :" and a search button. The search results are "Terminé" and "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 Responsibility. 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](http://www.ambysoft.com/processPatternsPage.html))
  - Pédagogiques(<http://www.pedagogicalpatterns.org/>)
  - .....

# Les fondamentaux [Grand00] avant tout

---

- **Constructions**

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



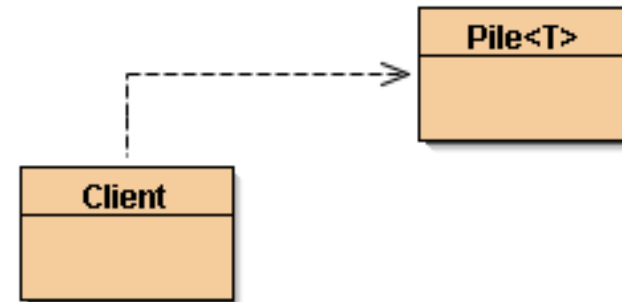
# Delegation

---

- **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...

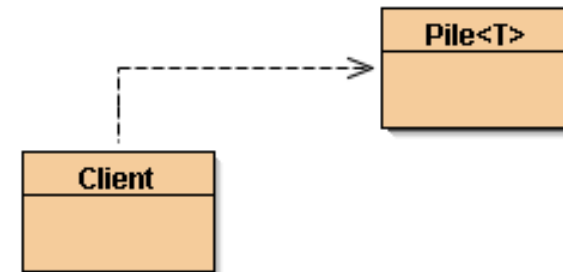
```
import java.util.Stack;
public class Pile<T>{
    private final Stack<T> stk;
    public Pile(){
        stk = new Stack<T>();
    }
    public void empiler(T t){
        stk.push(t);
    }
    ...}
}
```



```
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...**

# 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 : un exemple

---

```
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);
}
```

# Delegation + Interface

---

- **Délégation**

- Usage en interne d'une classe existante : la délégation
- La délégation peut-être changée 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 ...

## Delegation + interface = souplesse ...

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

```

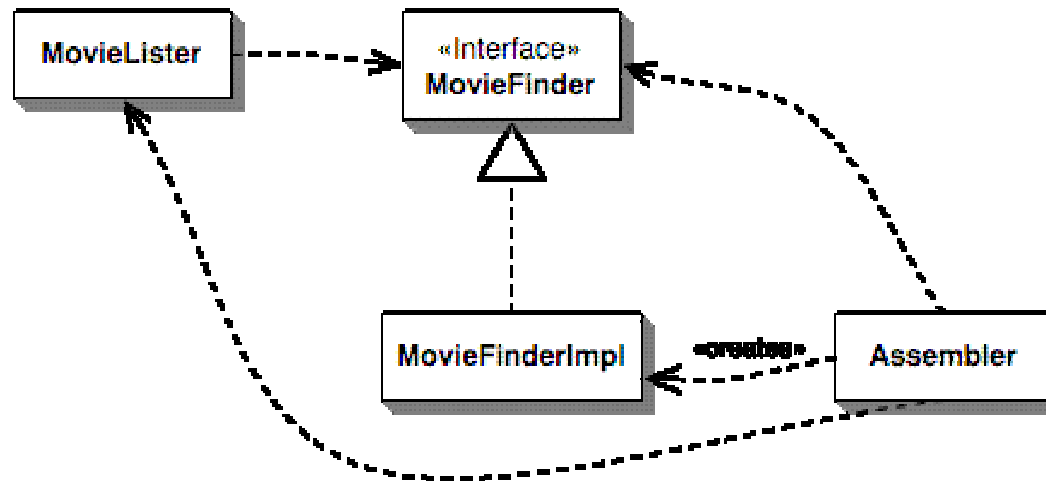
```
public class Client{
    public void main(String[] arg){
        Pile<Integer> p;
        p = new Pile<Integer>(new LinkedList<Integer>());
        p.empiler(4);
        ...
    }}

```



# Injection de dépendance

- Délégation + interface = injection de dépendance
- À lire : 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 »

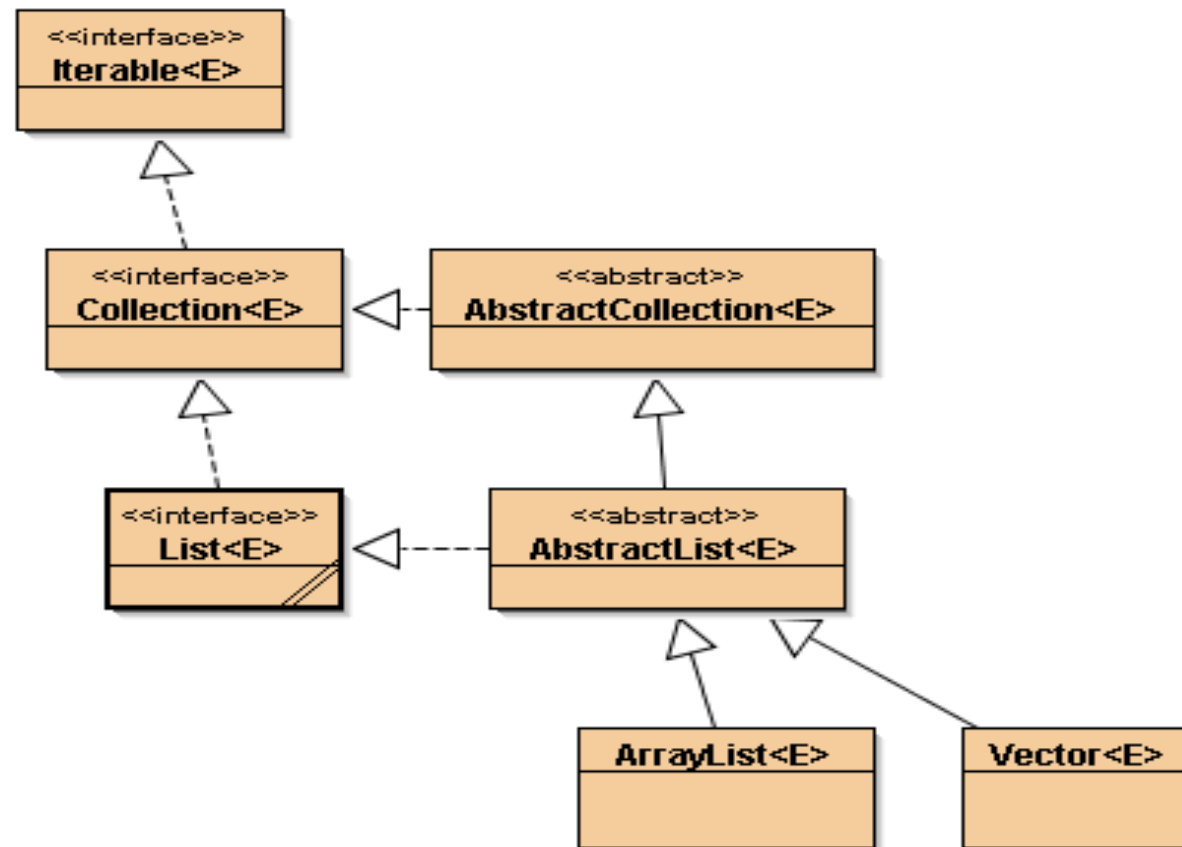
# Abstract superclass

---

- **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



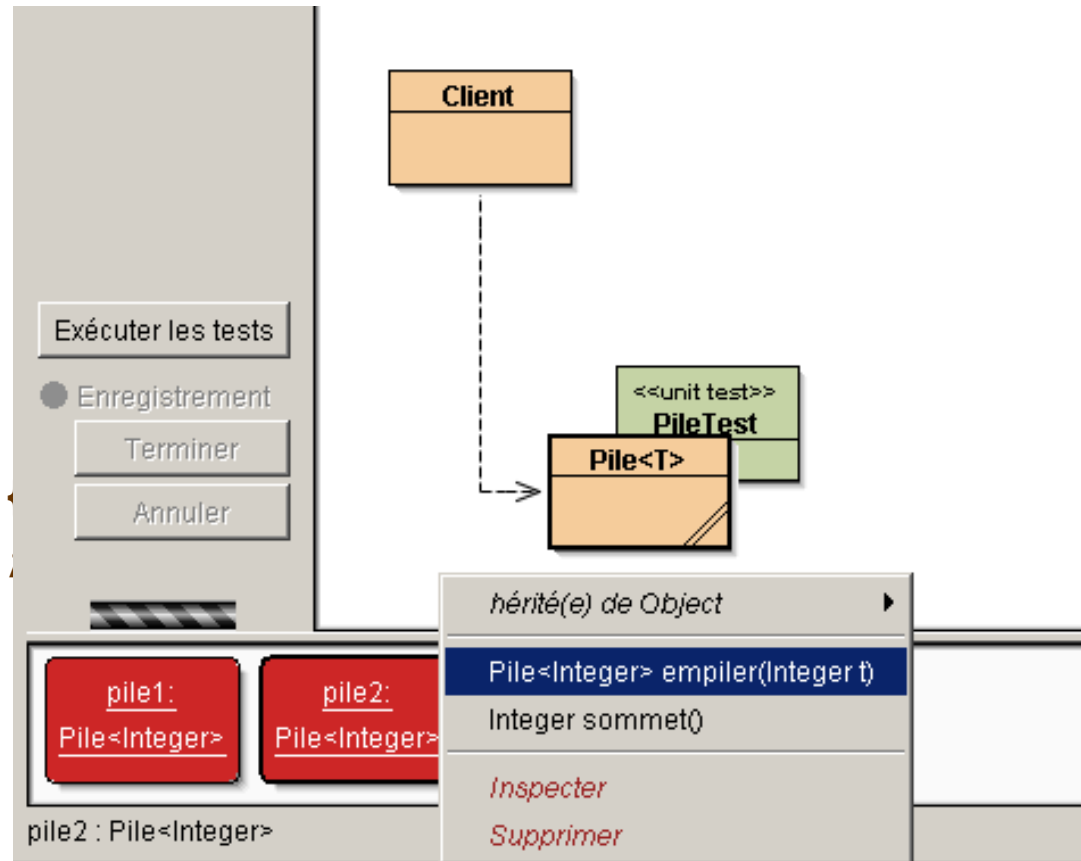
# Immutable

---

- **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
- 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 »

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

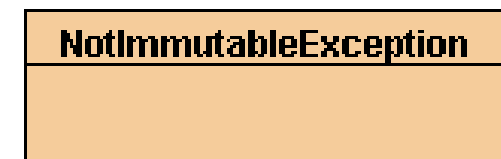
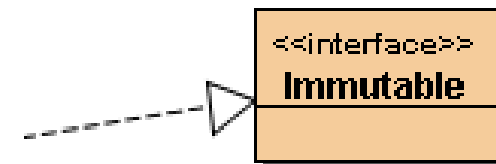
# Marker Interface : exemple

```
public interface Immutable{}
```

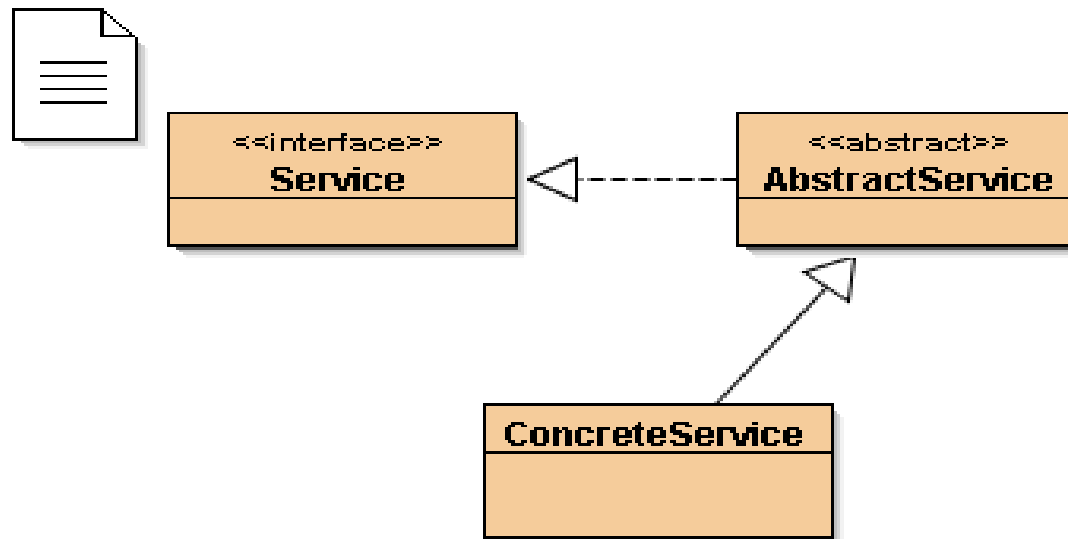
```
public class NotImmutableException  
    extends RuntimeException{  
    public NotImmutableException(){super();}  
    public NotImmutableException(String msg){super(msg);}  
}
```

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

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



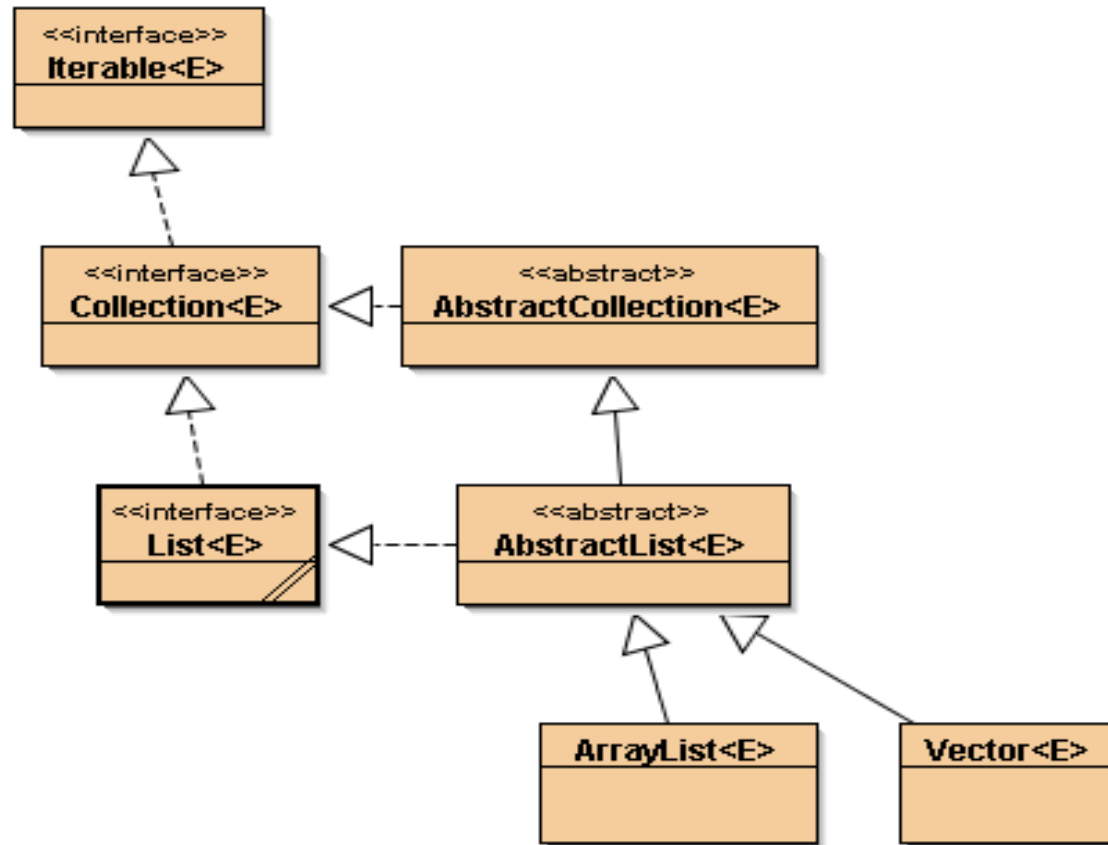
# Interface & abstract



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



# Interface & abstract



– Déjà vu ...

# Deux patrons pour l'exemple...

---

- Dans la famille des “Patrons Structurels”
- “je demande:”
- **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

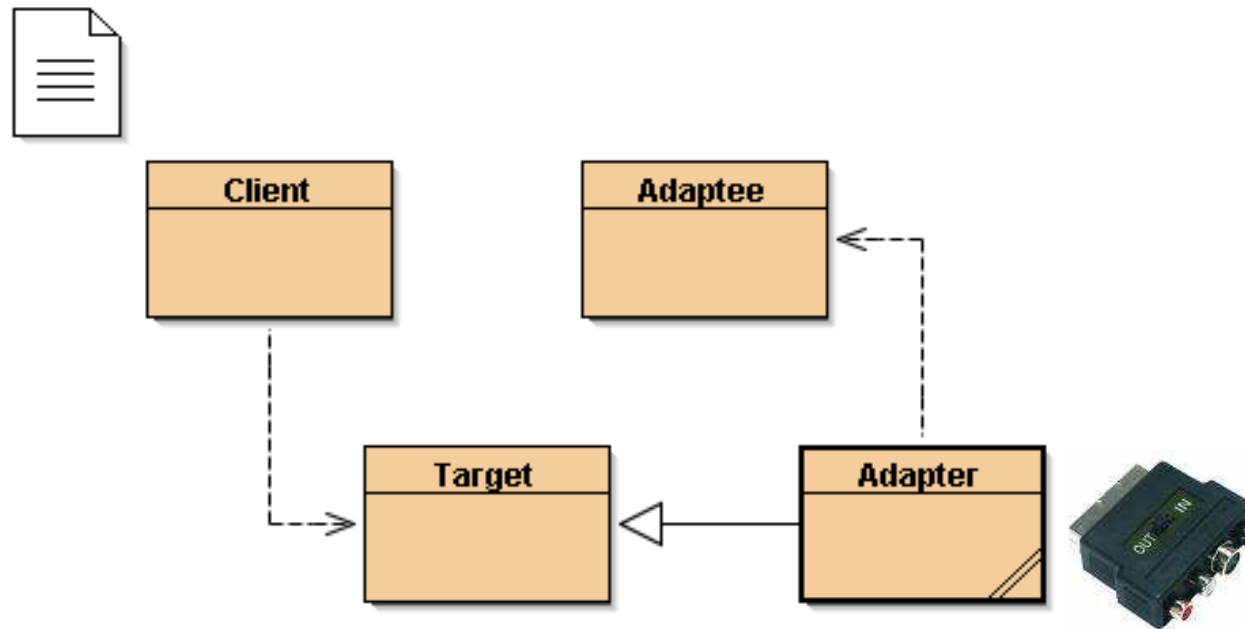
---



- **Adaptateurs**

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

# Pattern Adapter [DP05]



- DP05 ou [www.patterncoder.org](http://www.patterncoder.org)
  - [http://www.ics.heacademy.ac.uk/italics/vol6iss4/paterson\\_haddow.pdf](http://www.ics.heacademy.ac.uk/italics/vol6iss4/paterson_haddow.pdf)

# Adaptateur de prise ...

---

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

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

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

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



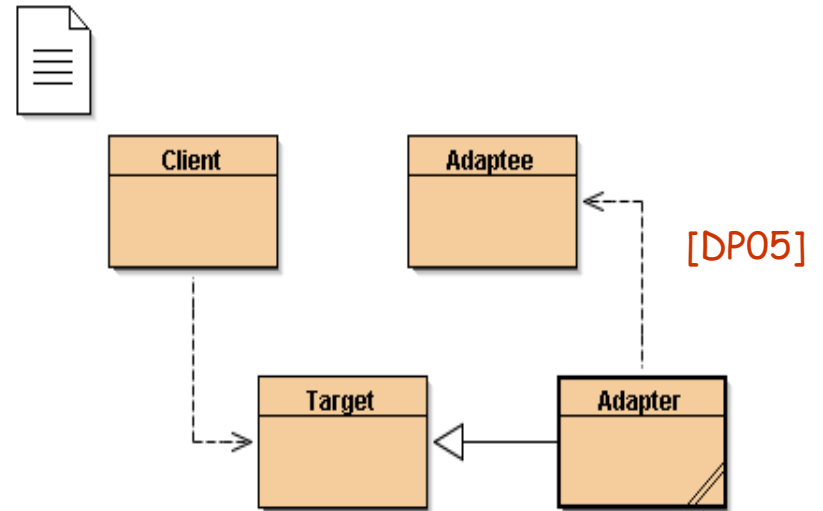
# Pattern Adapter

```
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 ...**

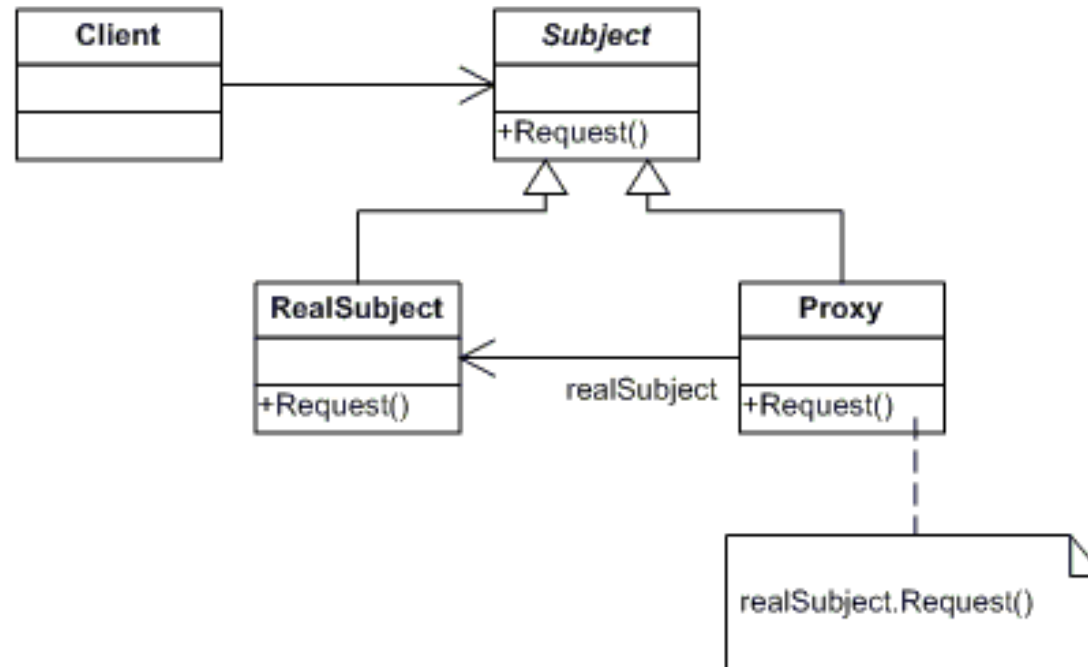
```
public Target newAdapter(final Adaptee adaptee){
    return
        new Target(){
            public void serviceA(){
                adaptee.serviceB();
            }
        };
}
```

- **Un classique ...**

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

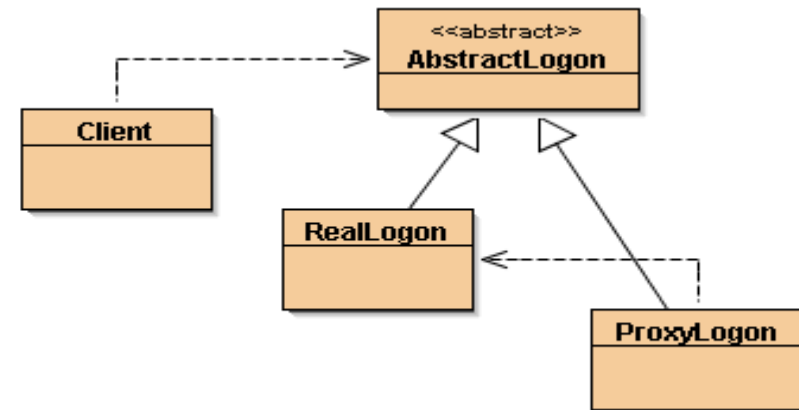
# Pattern Proxy

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





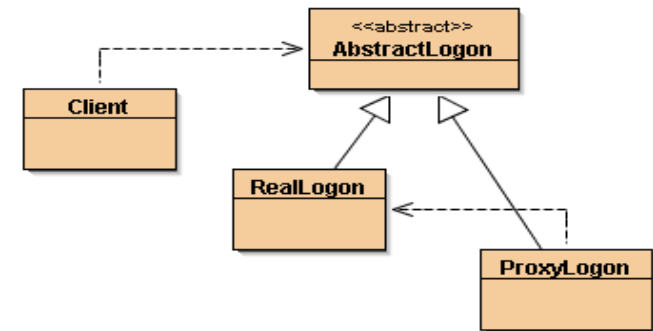
# Proxy : un exemple



```
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



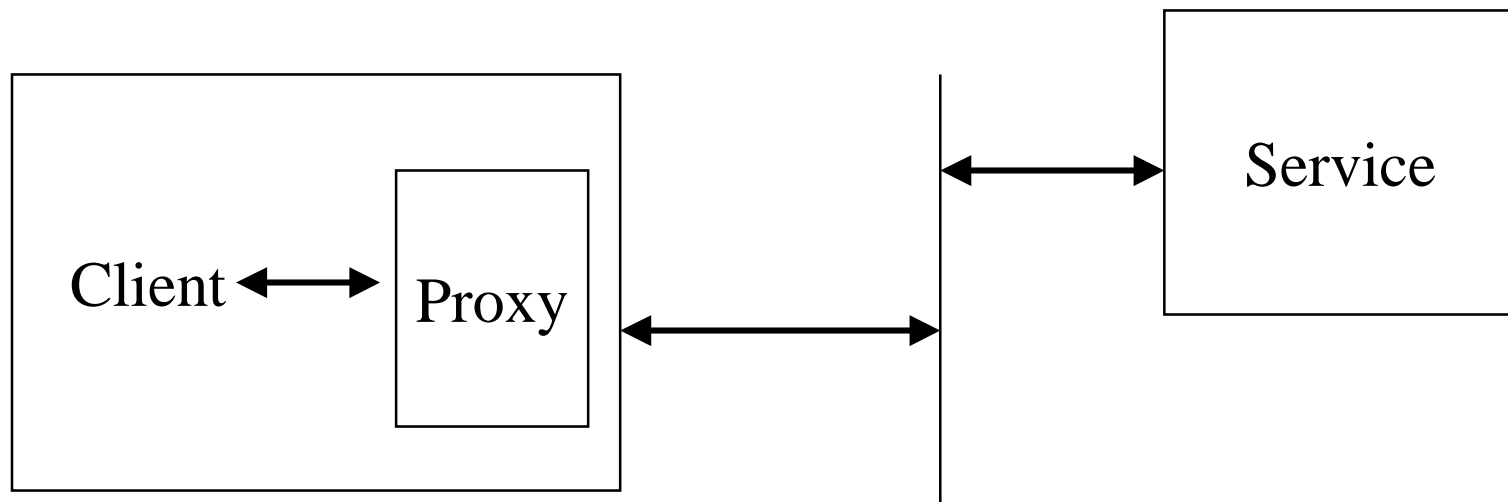
```
public class ProxyLogon extends AbstractLogon{
    private AbstractLogon real = new RealLogon();
```

```
    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;
    }
}
```

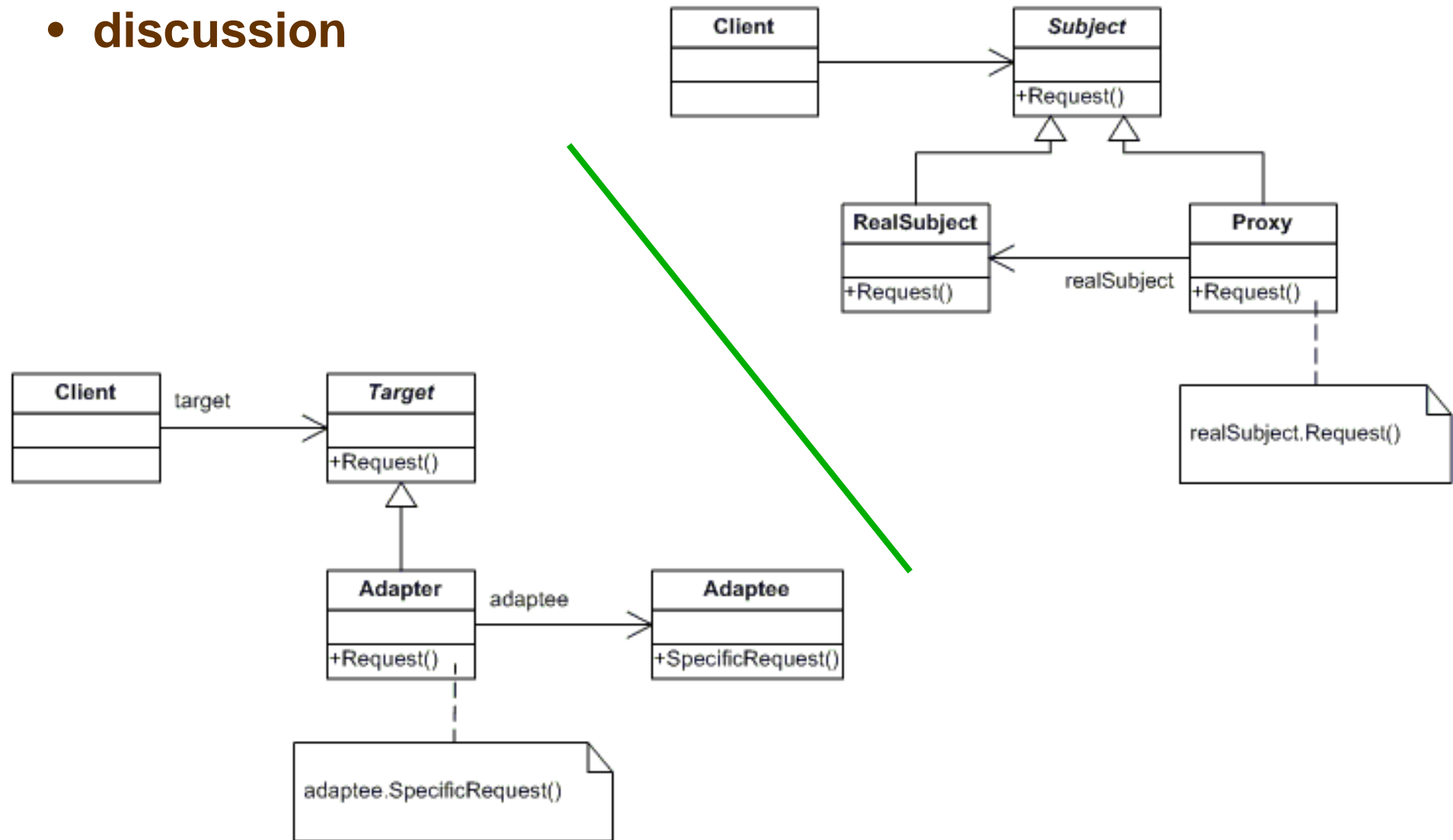
# Proxy et RMI

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



# Adapter\Proxy

- discussion

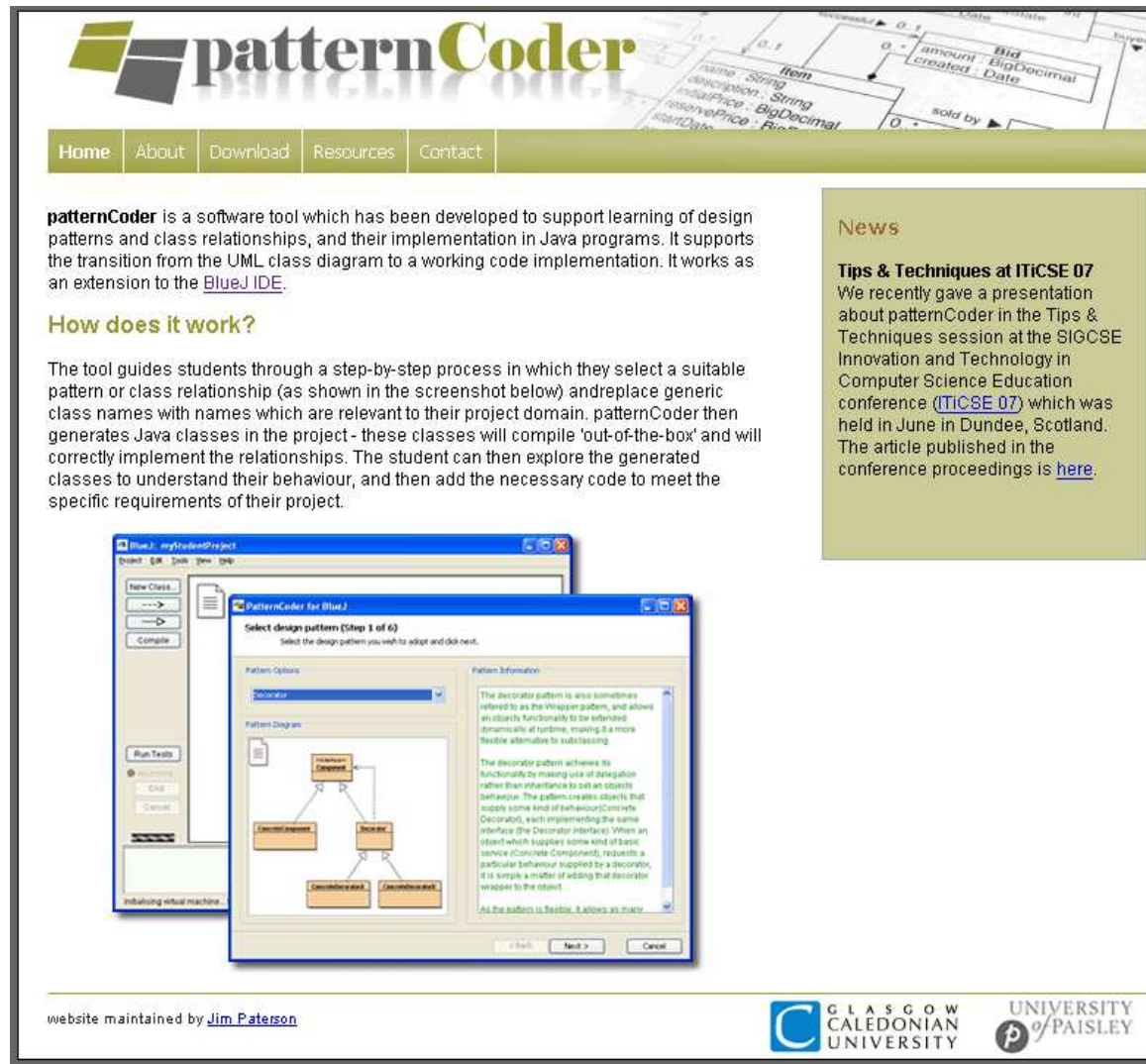


# Conclusion

---

- **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](http://www.patterncoder.org)



The screenshot shows the patternCoder website with a navigation menu (Home, About, Download, Resources, Contact) and a description of the tool. Below the text is a screenshot of the BlueJ IDE's 'PatternCoder for BlueJ' dialog box. The dialog is titled 'Select design pattern (Step 1 of 6)' and shows a 'Decorator' pattern selected. The 'Pattern Diagram' section displays a class hierarchy: 'Decorator' is the base class, with 'ConcreteDecorator' and 'Decorator' as subclasses. 'ConcreteDecorator' has two subclasses: 'ConcreteDecorator' and 'ConcreteDecorator'. The 'Pattern Information' section provides a detailed explanation of the decorator pattern, including its purpose and how it works.

**patternCoder** is a software tool which has been developed to support learning of design patterns and class relationships, and their implementation in Java programs. It supports the transition from the UML class diagram to a working code implementation. It works as an extension to the [BlueJ IDE](#).

### How does it work?

The tool guides students through a step-by-step process in which they select a suitable pattern or class relationship (as shown in the screenshot below) and replace generic class names with names which are relevant to their project domain. patternCoder then generates Java classes in the project - these classes will compile 'out-of-the-box' and will correctly implement the relationships. The student can then explore the generated classes to understand their behaviour, and then add the necessary code to meet the specific requirements of their project.

News

#### Tips & Techniques at ITICSE 07

We recently gave a presentation about patternCoder in the Tips & Techniques session at the SIGCSE Innovation and Technology in Computer Science Education conference ([ITICSE 07](#)) which was held in June in Dundee, Scotland. The article published in the conference proceedings is [here](#).

website maintained by [Jim Paterson](#)

GLASGOW CALEDONIAN UNIVERSITY UNIVERSITY OF PAISLEY

- **Démonstration : le patron Adapter**

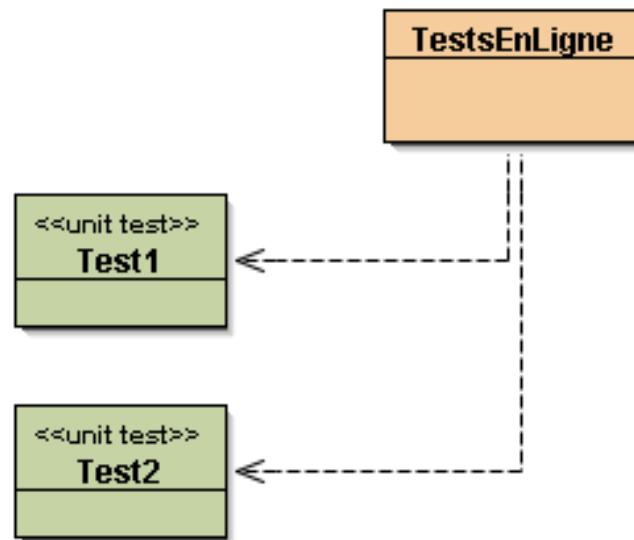
# Annexes

---

- **Architectures logicielles décrites comme un assemblage de patrons, extraites du Web**
  - **Junit**
  - **AWT**

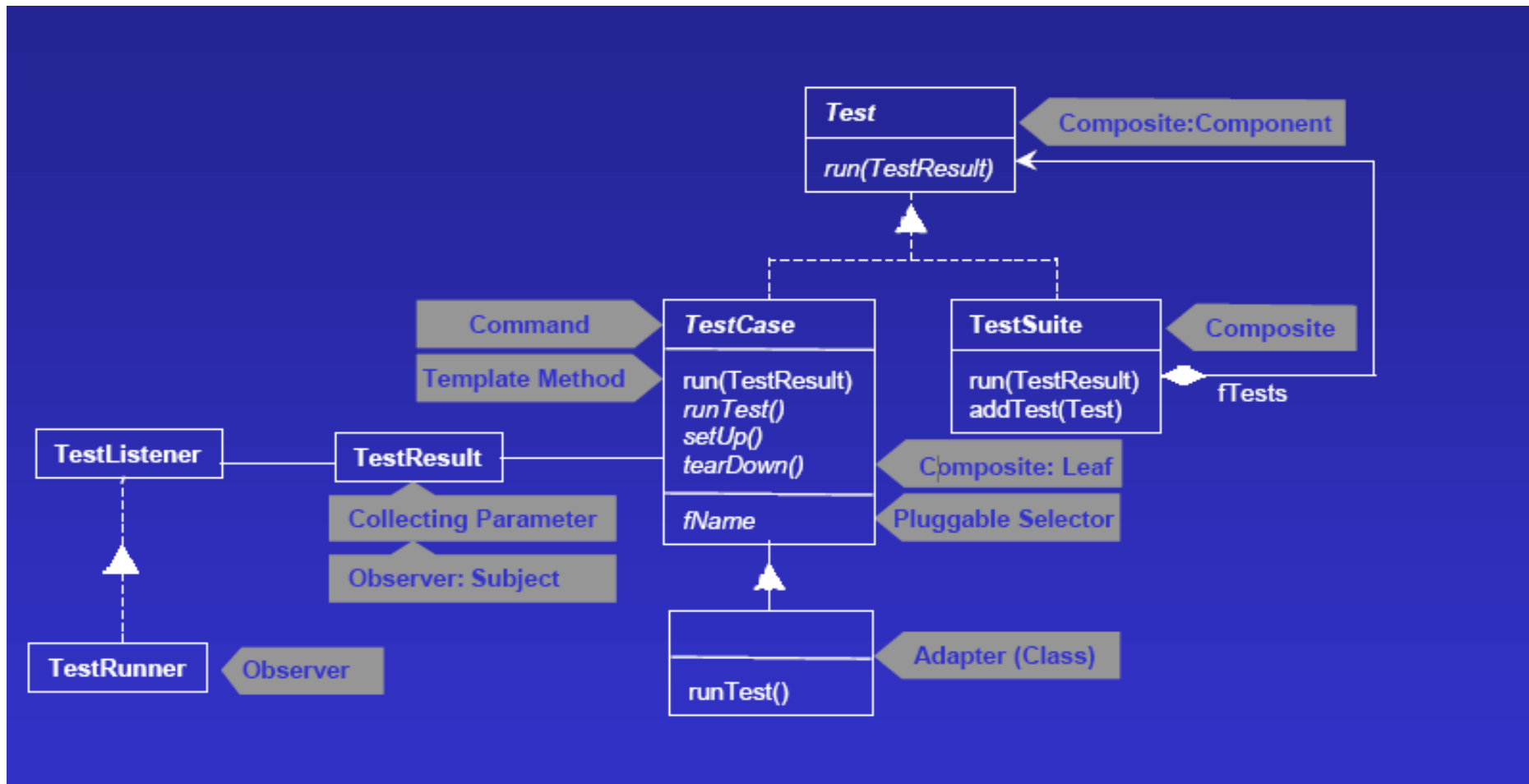
# Un exemple de conception en « patterns »

- <http://junit.sourceforge.net/doc/cookstour/cookstour.htm>
- <http://www-128.ibm.com/developerworks/java/library/j-aopwork7/index.html>
- **Tests unitaires, java, C++,**
  - Intégrés à Bluej, Eclipse,





# 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

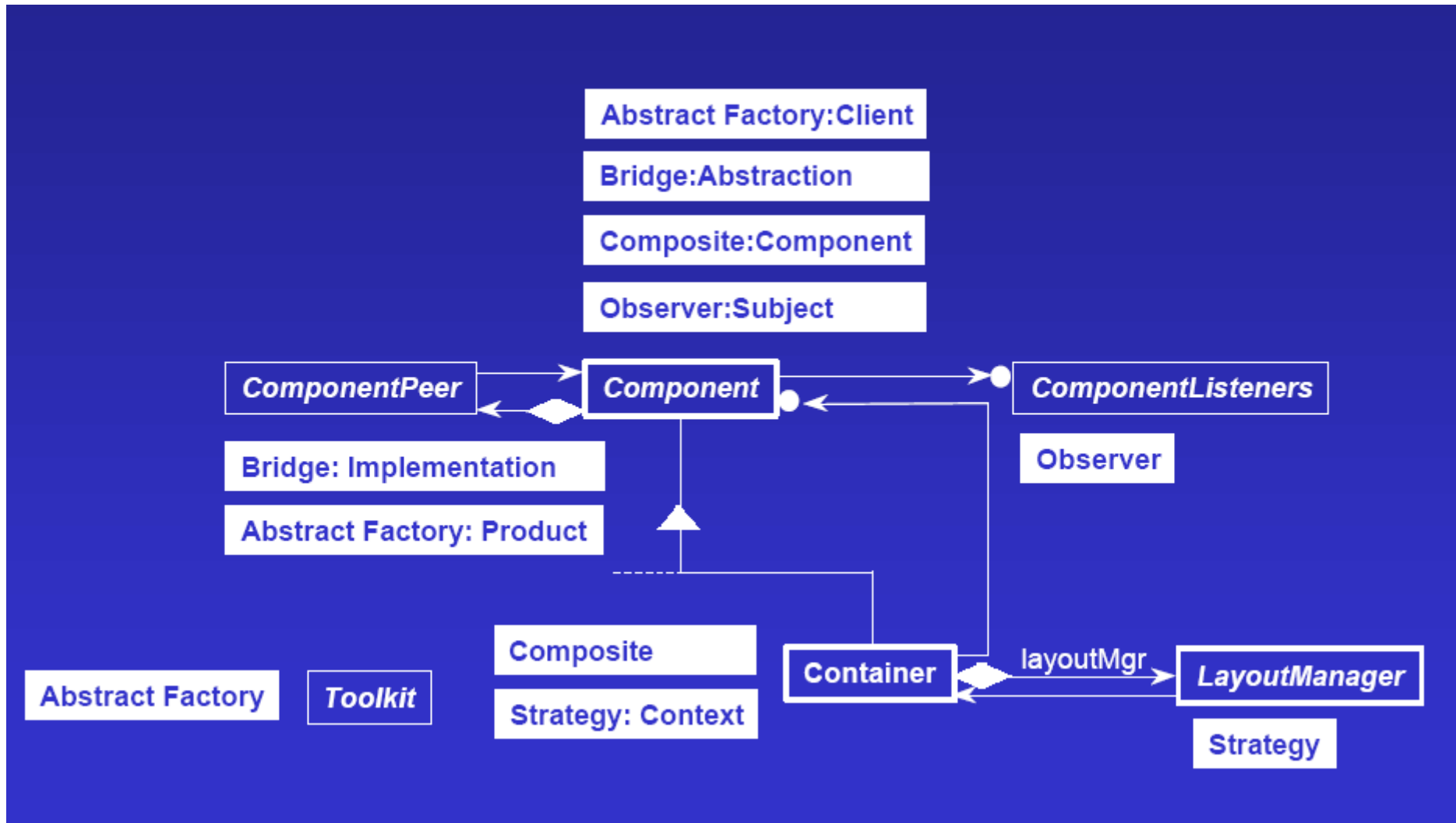
---

- **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>