
NFP121, Cnam/Paris

Cours 3-2 : Introduction aux patrons de conception

jean-michel Douin, douin au cnam point fr
version : 7 Octobre 2019

Notes de cours

Sommaire

- Conception à l'aide de patrons (*design pattern*).
- BlueJ un *plug-in* www.patterncoder.org

Principale bibliographie utilisée pour ces notes

- [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>

Design Pattern

En quelques mots ...

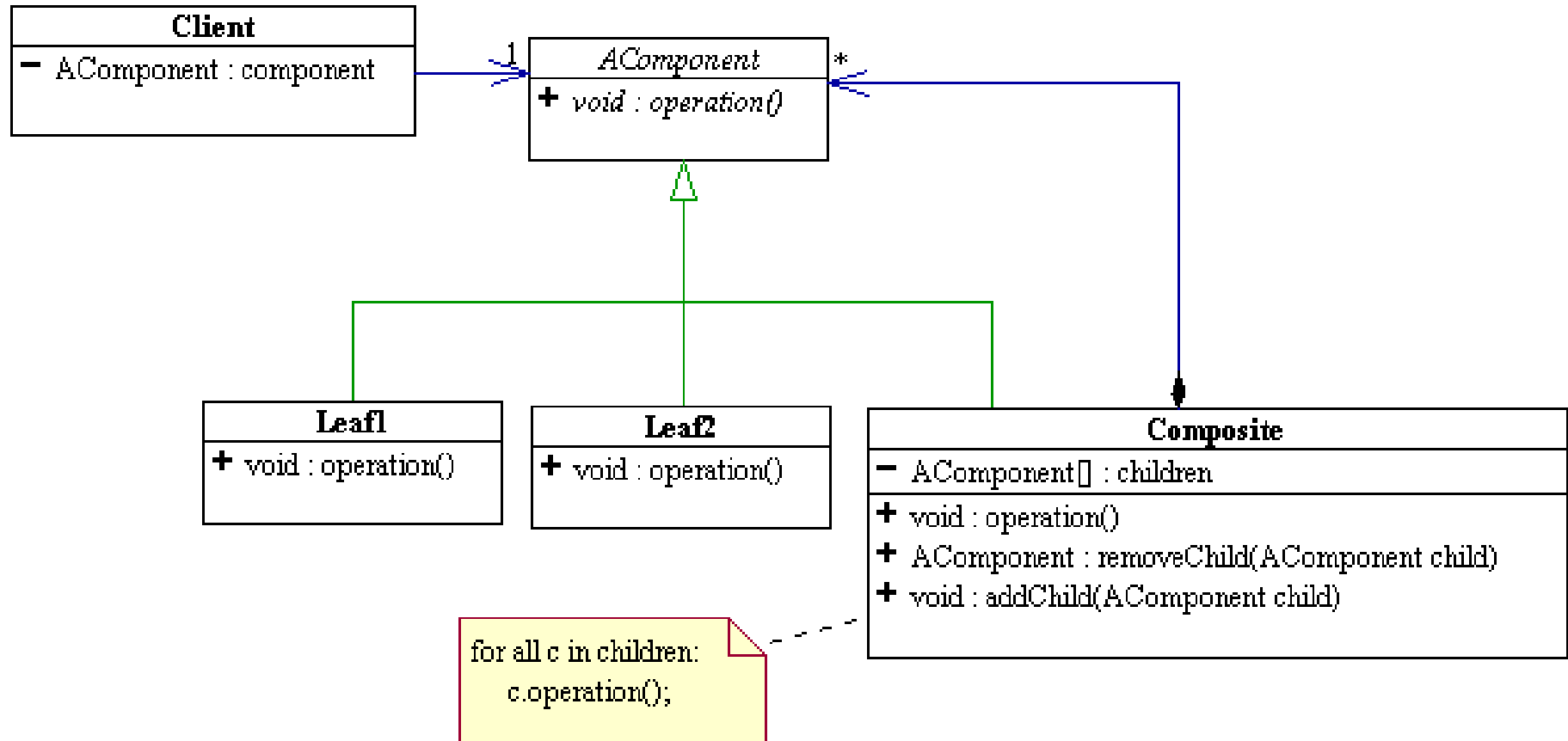
- **Moyen d'accomplir quelque chose,**
- **Une méthode éprouvée, réutilisée,**
- **Un code simple, « propre et peu perfectible »,**
- **Un jargon pour discuter du savoir faire,**
- **Quelque soit le langage à Objet,**
- **Intra discipline ...**

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** ???
 - Les entrées sorties utilisent le **patron Décorateur** ???
 - etc. ...
- **Une application = un assemblage de plusieurs patterns**
 - Un rôle ?

La bibliothèque graphique du JDK utilise un composite ?

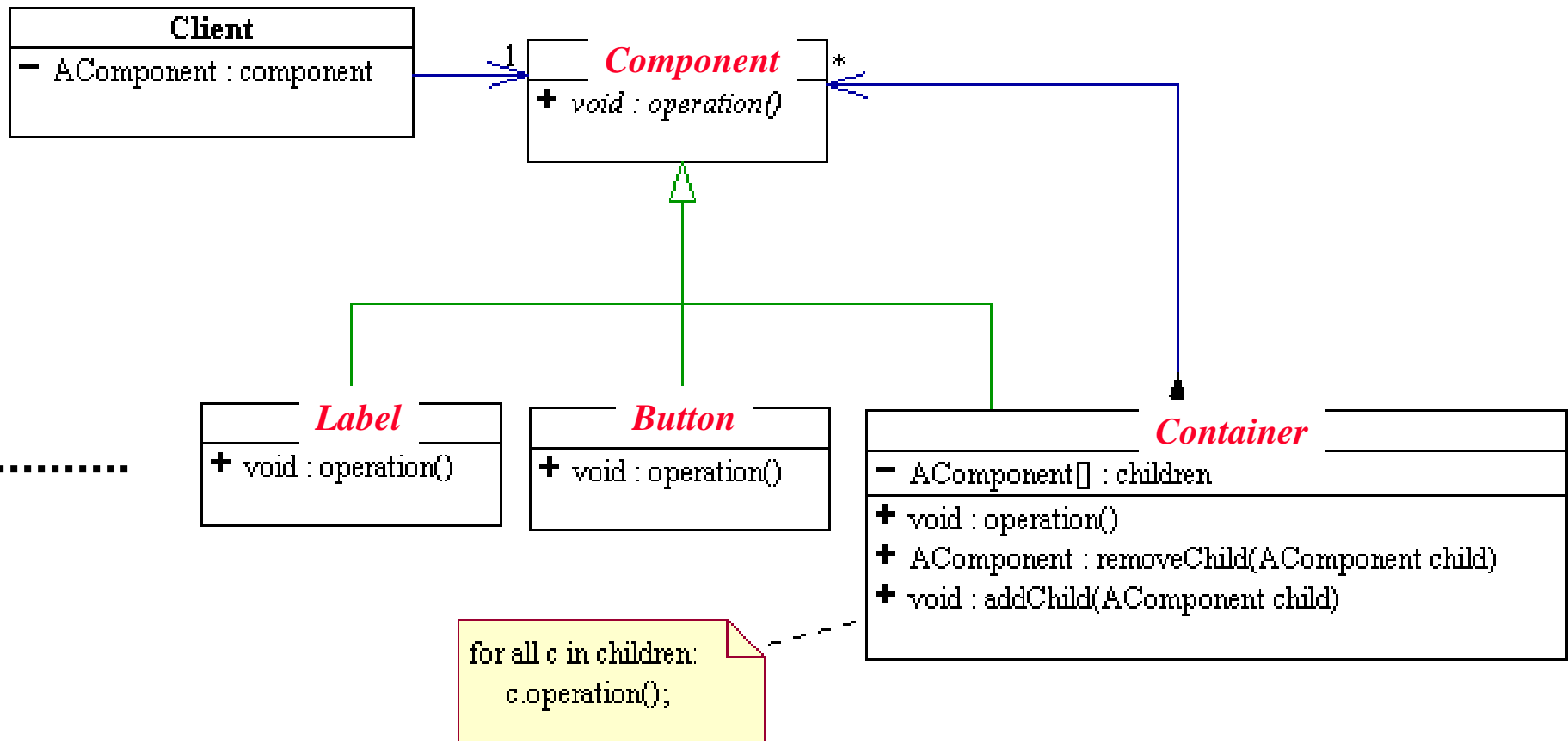
- Le pattern Composite ?, usage d'un moteur de recherche sur le web ...



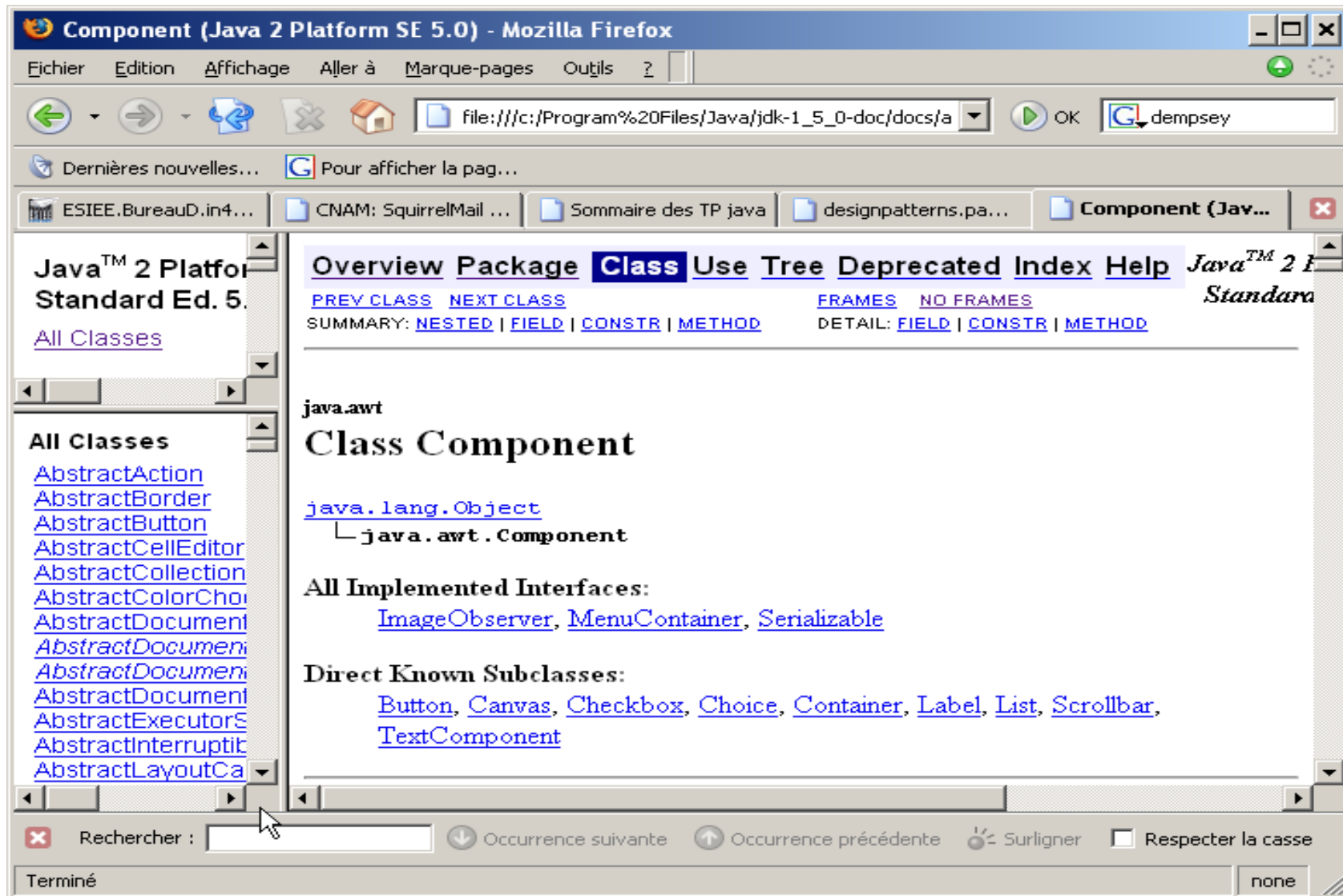
??? Le composite est un composant, les feuilles aussi ???

La bibliothèque graphique utilise bien un Composite :

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



Premier gain : À la place de ceci



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
- **Adapté à 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, les noms des 23 patrons**
 - **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
- **La carte mémo à se procurer**
 - http://ifod.cnam.fr/NFP121/supports/extras_designpatternscard.pdf

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

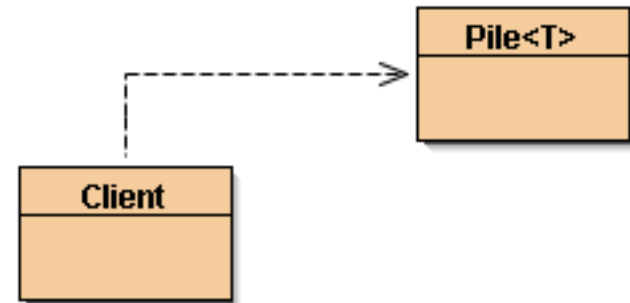
Delegation

- **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 offertes aux clients
- **Note syntaxique <T> type générique**
 - « le type T est un paramètre de la classe »
 - `java.util.List<T>` est une interface,
 - `Java.util.Stack<T>` et `java.util.LinkedList<T>` sont deux implémentations

Délégation : un exemple classique...

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

```



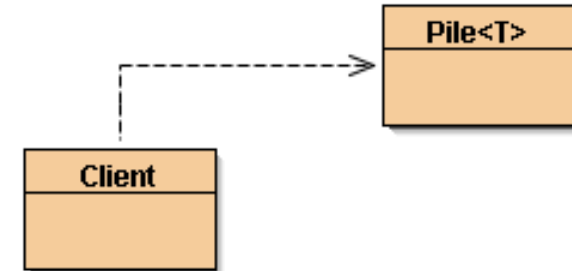
```
public class Client{
    public void main(String[] arg){
        Pile<Integer> p = new Pile<Integer>();
        p.empiler(4);
    }
}

```


Délégation : souplesse ... Client inchangé

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

```

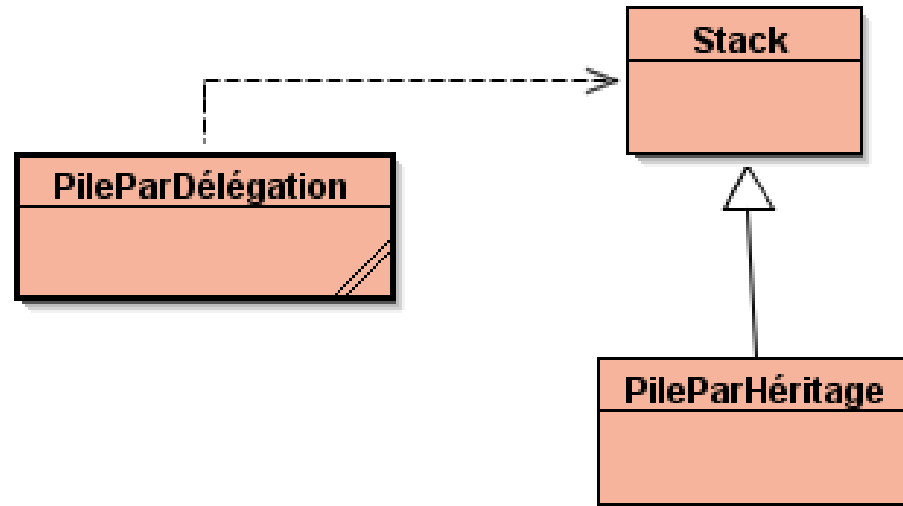


```
public class Client{
    public void main(String[] arg){
        Pile<Integer> p = new Pile<Integer>();
        p.empiler(4);
    }
}

```

Délégation / Héritage

- Petite discussion...



- Avantages/inconvénients
- Délégation préférée

mais

Délégation : mais

```
public class Pile<T>{  
  
    private List<T> stk;  
  
    public Pile(){  
        stk = new LinkedList<T>() ;  
    }  
  
    ...  
}
```

Mais sur cet exemple,
l'utilisateur
n'a pas le choix de
l'implémentation
LinkedList est imposé de fait
...



Interface

- **La liste des méthodes à respecter**
 - Les méthodes qu'une classe devra implémenter
 - Plusieurs classes peuvent implémenter une même interface
 - Le client choisira une implémentation en fonction de ses besoins

–Exemple

- **Collection<T>** est une interface
- **ArrayList<T>, LinkedList<T>**
sont des implémentations de **Collection<T>**

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());  
}
```

Interface java.lang.Iterable<T>

- Tout objet que l'on peut parcourir, c'est vaste...

```
public interface Iterable<T>{  
    Iterator<T> iterator();  
}
```

java.lang

Interface Iterable<T>

All Known Subinterfaces:

[BeanContext](#), [BeanContextServices](#), [BlockingQueue](#)<E>, [Collection](#)<E>, [List](#)<E>, [Queue](#)<E>, [Set](#)<E>, [SortedSet](#)<E>

All Known Implementing Classes:

[AbstractCollection](#), [AbstractList](#), [AbstractQueue](#), [AbstractSequentialList](#), [AbstractSet](#), [ArrayBlockingQueue](#), [ArrayList](#), [AttributeList](#),
[BeanContextServicesSupport](#), [BeanContextSupport](#), [ConcurrentLinkedQueue](#), [CopyOnWriteArrayList](#), [CopyOnWriteArraySet](#), [DelayQueue](#), [EnumSet](#),
[HashSet](#), [JobStateReasons](#), [LinkedBlockingQueue](#), [LinkedHashSet](#), [LinkedList](#), [PriorityBlockingQueue](#), [PriorityQueue](#), [RoleList](#), [RoleUnresolvedList](#), [Stack](#),
[SynchronousQueue](#), [TreeSet](#), [Vector](#)

Usage des interfaces

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

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

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

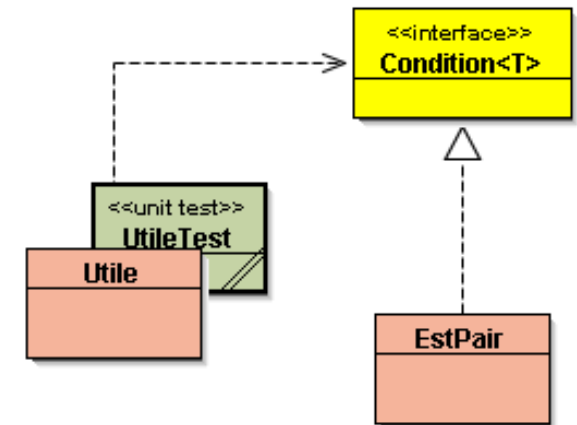
public interface Condition<T>{
    public boolean isTrue(T t);
}
```

Iterable et Condition
sont des interfaces

discussion

Démonstration, 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);
```

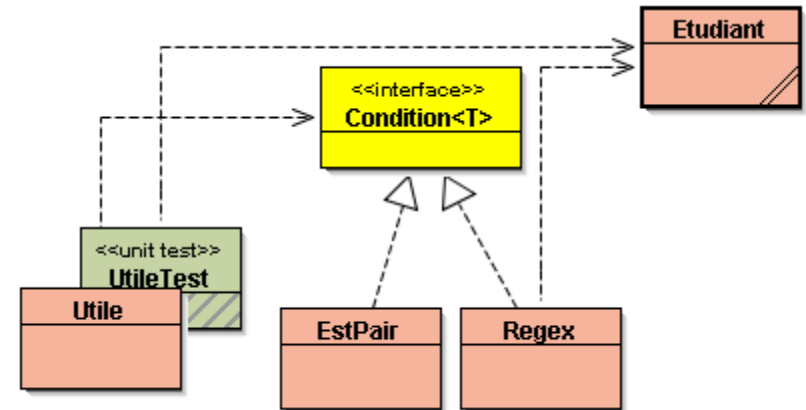
BlueJ: BlueJ : Terminal - filtre

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);
```

discussion

```
BlueJ: BlueJ : Terminal - filtre
Options

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

Délégation : une critique, bis

```
public class Pile<T>{  
  
    private final List<T> stk;  
  
    public Pile(){  
        stk = new LinkedList<T>();  
    }  
    ...}
```

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

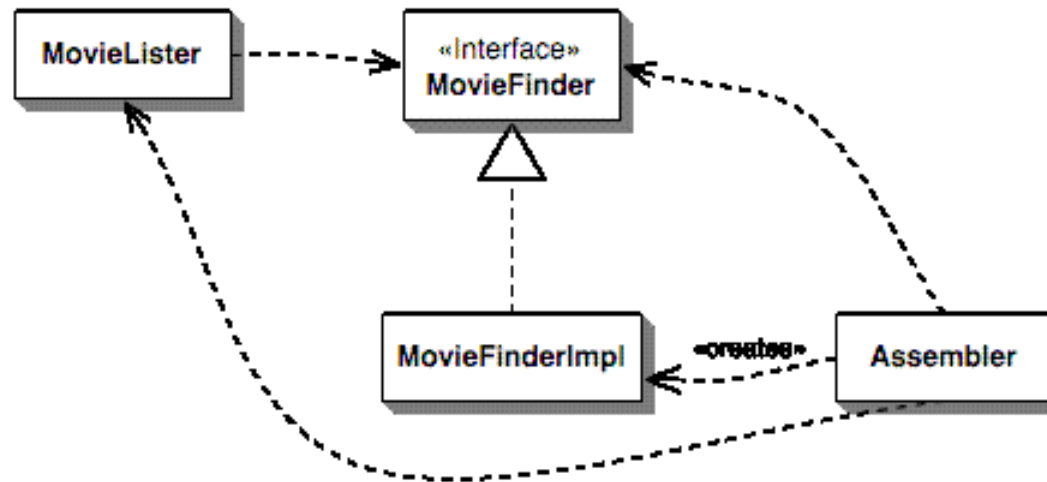
```
public class Pile<T>{  
    private final List<T> stk;  
  
    public Pile(List<T> l){  
        stk = l;  
    }  
    public Pile(){  
        stk = new LinkedList<T>();  
    }  
    ...}
```

Ici l'utilisateur
a le choix de
l'implémentation de la liste ...

En injectant une implémentation (l)
créant une dépendance envers celle-ci

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 »

Délégation : une critique de critique, mais de mais

```
public class Pile<T>{  
    private final List<T> stk;  
  
    public Pile(List<T> l){  
        stk = l;  
    }...}
```

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

Mais rien n'empêche ici une utilisation *malheureuse* de **l à l'extérieur de Pile**

```
List<String> l = new LinkedList<String>(); // ← correct  
  
Pile<String> p = new Pile(l); // ← idem  
p.empiler("ok"); // ← idem  
  
l.add("attention"); // ← attention  
// état de la pile ?
```

Une solution ? Satisfaisante ?

Injection de dépendance, un outil possible

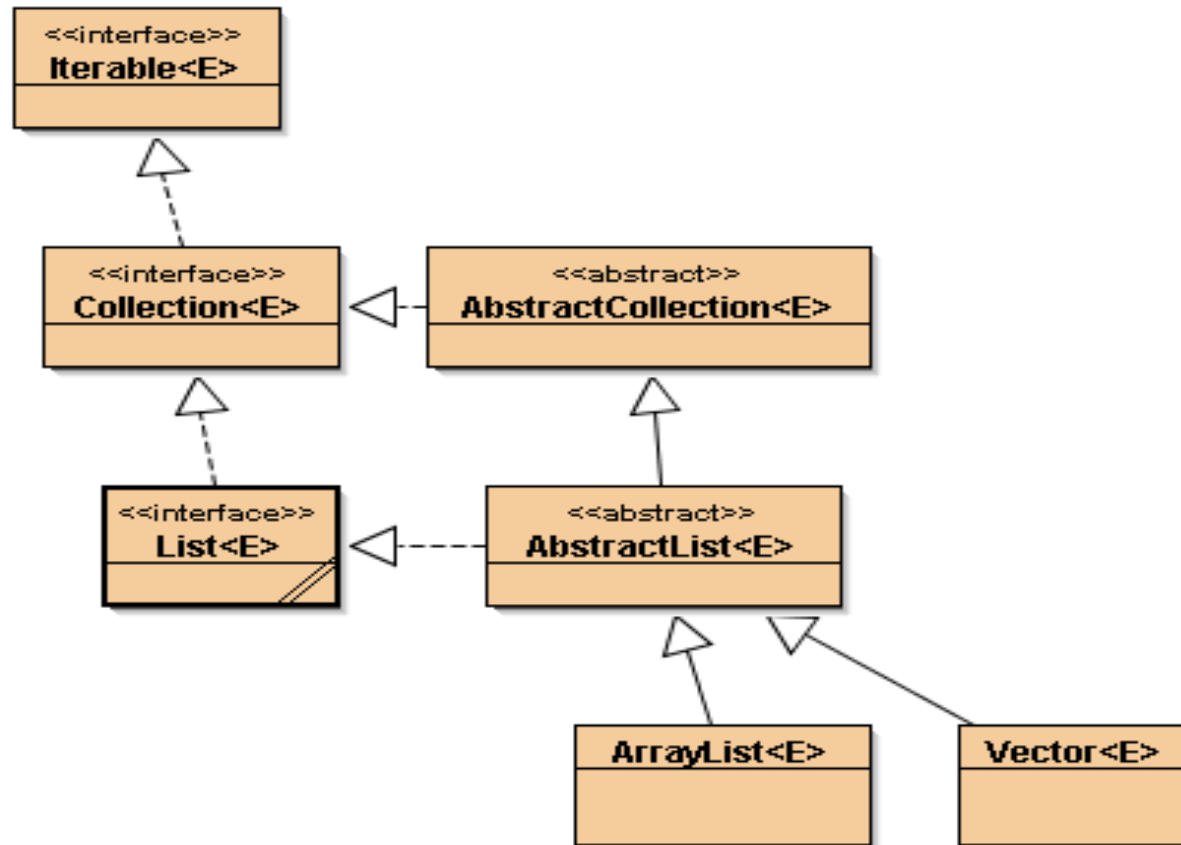
- **La classe à injecter est décrite dans un fichier**
 - Un fichier texte en XML par exemple
- `<injections>`
- `<injection classe= "Pile" injection= "java.util.LinkedList" />`
- `<injection />`
-
- `</injections>`
- **Un outil pourrait se charger de**
 - Lire le fichier XML et interpréter la balise `<injection`
 - Créer une instance de la classe à injecter : `java.util.LinkedList`
 - Créer une instance de la classe `Pile`
 - D'exécuter le constructeur et/ou les mutateurs de la classe `Pile`
 - Cet outil permettrait alors une séparation configuration / Implémentation
- **À la recherche du couplage faible**
 - Ici induite par l'usage d'interface ...
- **Affaire à suivre ...**

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 *paquetage que nous reverrons bientôt*

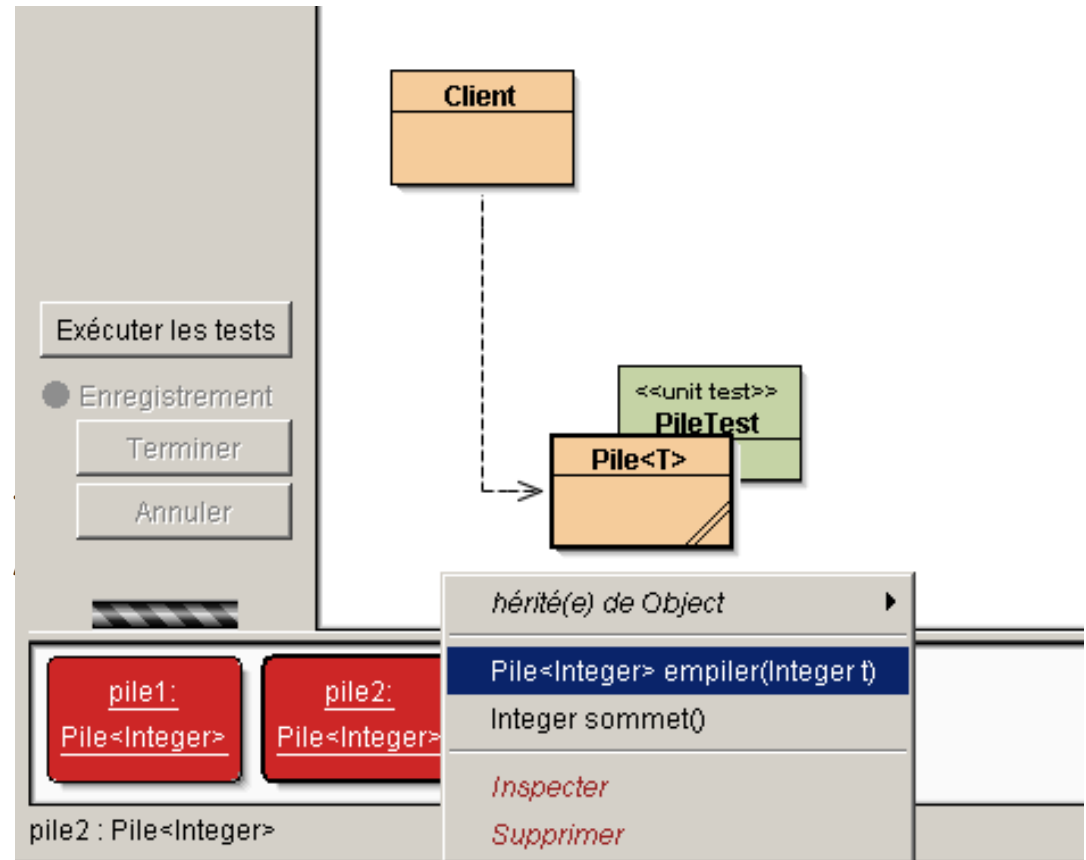


Immutable

- **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 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{}    // <- interface vide
```

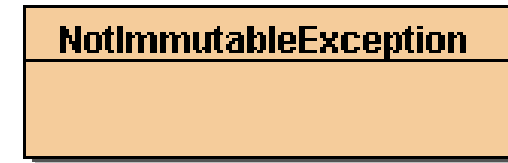
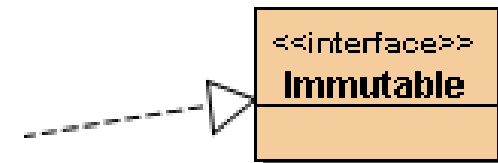
```
public class NotImmutableException
    extends RuntimeException{
    public NotImmutableException(){super();}
    public NotImmutableException(String msg){super(msg);}
}
```

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

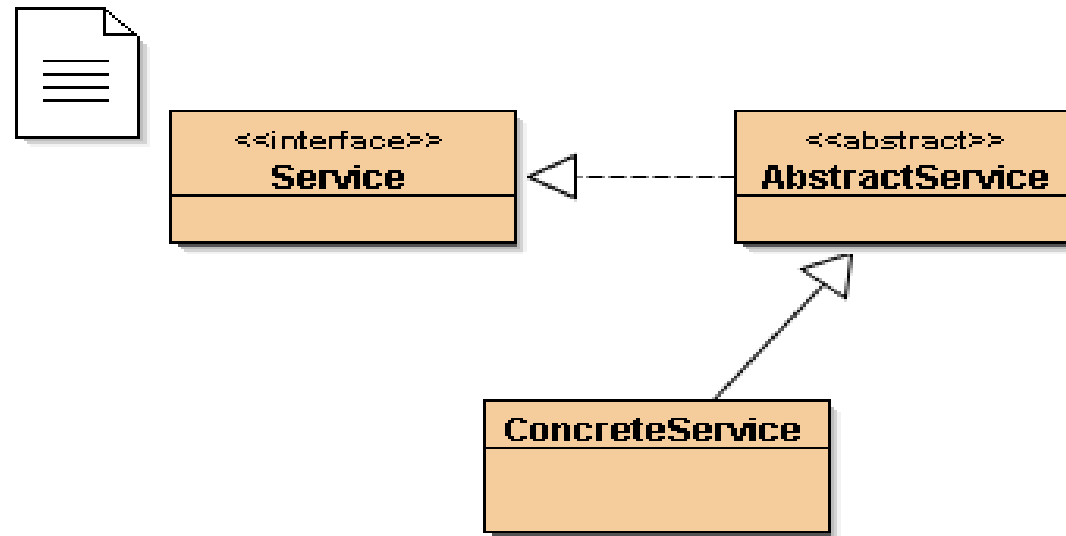
```
Pile<Integer> p = new Pile<Integer>();
if(!(p instanceof Immutable)) // <- test
```

```
    throw new NotImmutableException();
```

```
...
```



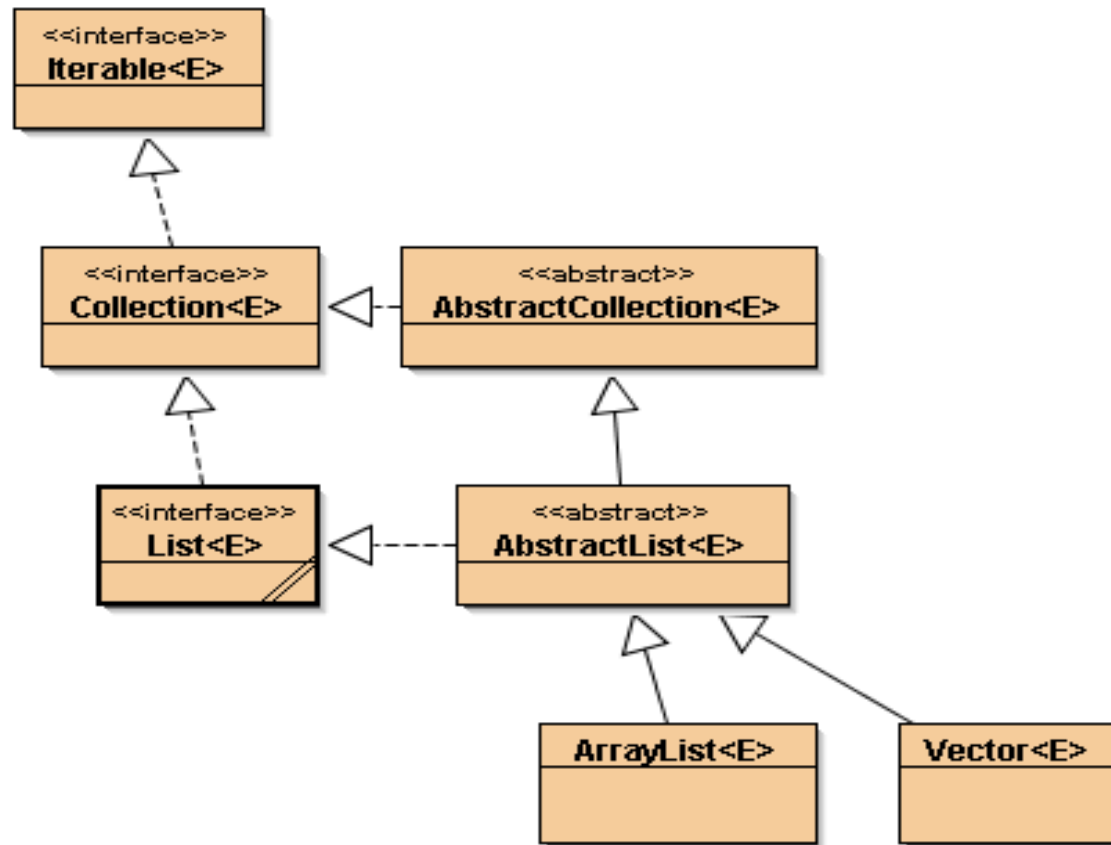
Interface & abstract



- **Avantages cumulés !**

- `java.util.Collection<T>` est une interface
- `java.util.AbstractCollection<T>` est une classe incomplète
- `java.util.ArrayList<T>` est une classe concrète

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 Responsibility. Command Interpreter Iterator
Mediator Memento Observer State
Strategy Template Method Visitor

- **La carte mémo à télécharger**

- http://ifod.cnam.fr/NFP121/supports/extras_designpatternscard.pdf

Deux patrons pour l'exemple...

- **Dans la famille “Patrons Structurels”**
- **Adapter**
 - **Adapte** l'interface d'une classe afin d'être conforme aux souhaits du client
- **Proxy**
 - **Fournit** un mandataire au client afin de contrôler/vérifier ses accès

Adaptateurs : exemples



- **Adaptateurs**

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

Adaptateur exemple : Péritel \leftrightarrow RCA

Ce que nous avons : RCA

```
public interface Plug {  
    public void RCA();  
}
```

Ce que le client souhaite : une prise Péritel

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

- Il faut s'adapter aux souhaits du client



Adaptateur (implements Prise)

```
public class Adaptateur implements Prise {  
    public Plug adapté;  
  
    public Adaptateur(Plug adapté) {  
        this.adapté = adapté;  
    }  
  
    public void p  ritel() {  
        adapté.RCA();  
    }  
}
```

Adaptateur : Le client est satisfait

```
public class UnClient {
```

```
    Prise prise = new Adaptateur(new PlugRCA());
```

```
    prise.péritel(); // il est satisfait ...
```

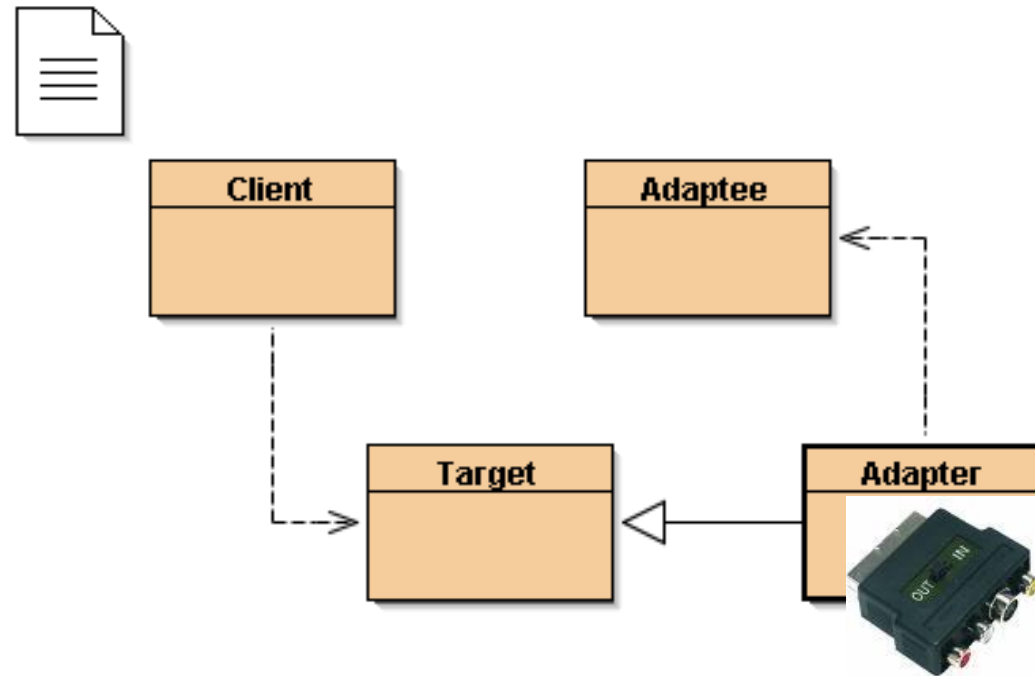
```
}
```

```
public class PlugRCA implements Plug {
```

```
    public void RCA(){ ....}
```

```
}
```

Pattern Adapter [DP05]



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

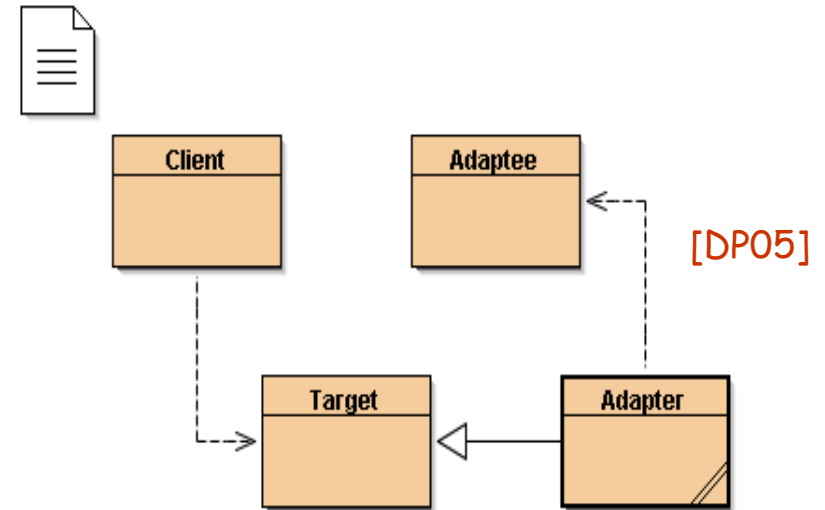
Pattern Adapter [DP05]

```
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 ...**

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

Une question possible ...

3) Soit l'interface PileI ci-dessous, sans aucune implémentation,

```
1 public interface PileI<E>{  
2  
3     public void empiler(E e);  
4     public E dépiler();  
5     public boolean estVide();  
6 }
```

par contre nous disposons de plusieurs implémentations de l'interface StackI,

```
3 public interface StackI<E>{  
4  
5     public void push(E e);  
6     public E pop();  
7     public boolean isEmpty();  
8 }
```

L'utilisateur est francophone et souhaite vivement continuer d'appeler les méthodes définies dans l'interface PileI.

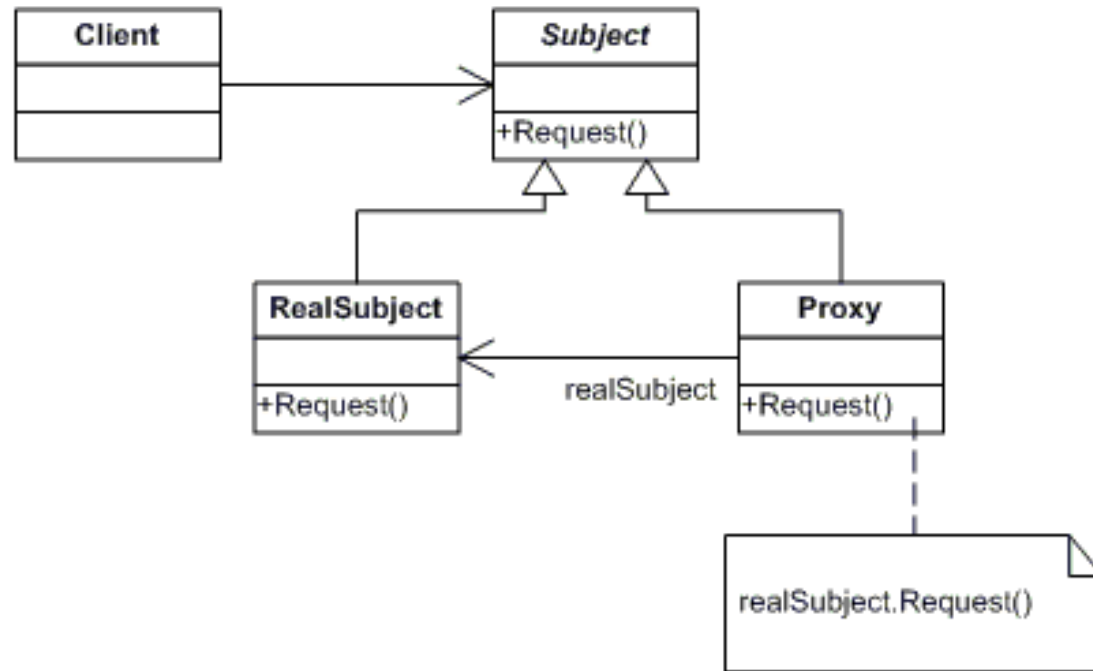
Choisissez un patron permettant à cet utilisateur de respecter ses souhaits et implémentez complètement la solution. Bien entendu empiler à la même sémantique que push, idem pour dépiler/pop et estVide/isEmpty



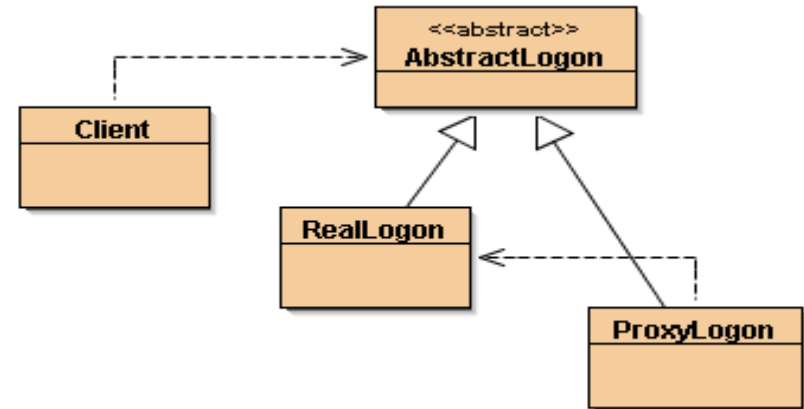
Proposez un scenario d'utilisation pour notre ClientFrancophone

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);
}
```

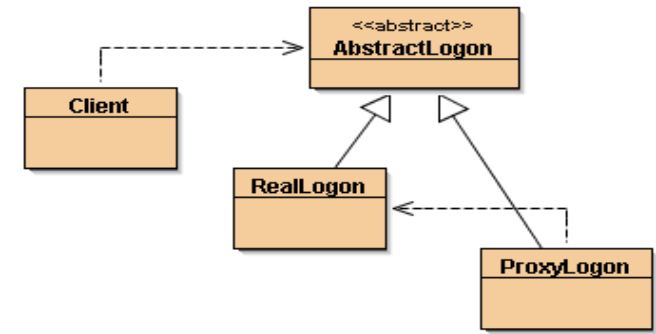
```
public class Client{
    public static void main(String[] args){
        AbstractLogon logon = new ProxyLogon();
        ...
    }
}
```

Proxy : exemple suite

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

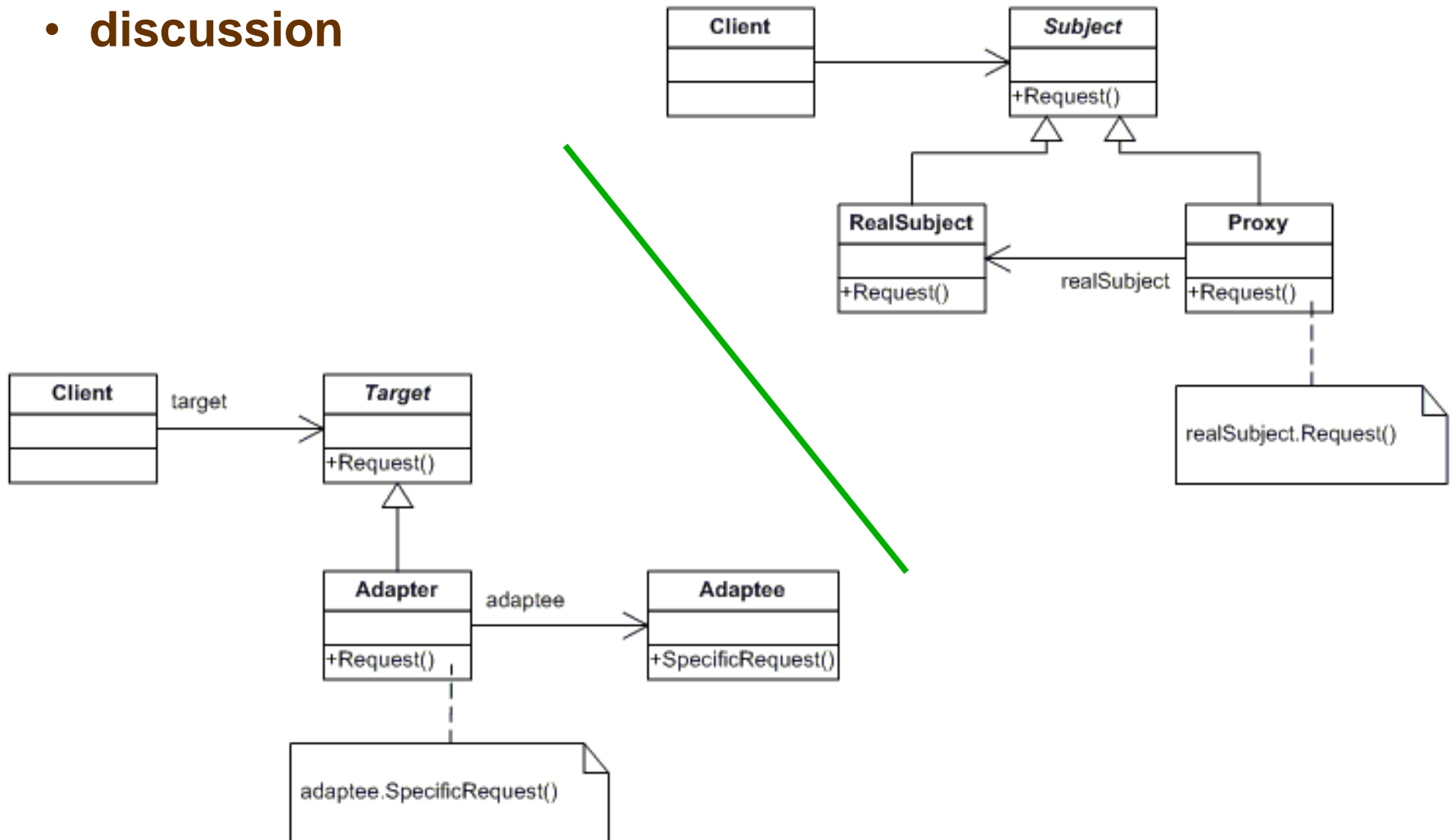
    public boolean authenticate(String user, String password){
        if(user.equals("root") && password.equals("java"))
            return real.authenticate(user, password);
        else
            return false;
    }
}
```

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




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 ?**

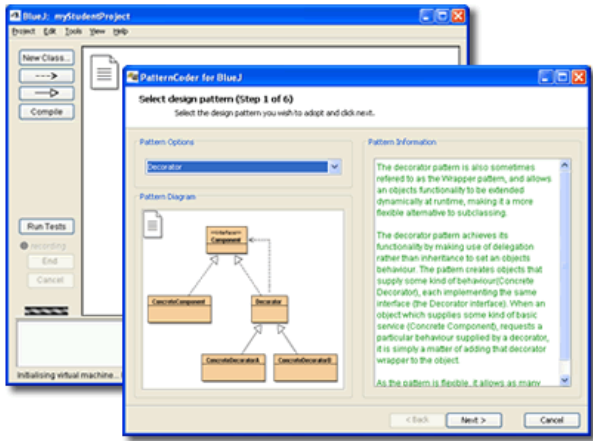


[Home](#) [About](#) [Download](#) [Resources](#) [Contact](#)

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](#)



- Démonstration : le patron Adapter