NSY102
Conception de logiciels Intranet
les patrons Proxy

Cnam Paris
jean-michel Douin, douin au cnam point fr
version du 6 mars 2017

Le Patron Procuration/ Pattern Proxy

Sommaire

- **Présentation ou Rappels …**
  - **ClassLoader et introspection**

- **Le patron Proxy**
  - **L ’original [Gof95]**
    - **Proxy**

  - **Les variations**
    - **VirtualProxy**
    - **RemoteProxy**
    - **SecureProxy**
    - **ProtectionProxy**
    - **…**

    - **DynamicProxy**

- **Retour sur le patron Interceptor**
  - **Apparenté Chain of Responsability**

Bibliographie utilisée

- Design Patterns, catalogue de modèles de conception réutilisables
  de Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides [Gof95]
  International thomson publishing France

- http://www.ida.liu.se/~uweas/Lectures/DesignPatterns01/panas-pattern-hatching.ppt

- http://www.mindspring.com/~mgrand/pattern_synopses.htm

- http://research.umbc.edu/~tarr/dp/lectures/DynProxies-2pp.pdf
- http://java.sun.com/products/jfc/tsc/articles/generic-listener2/index.html

- http://www.javaworld.com/javaworld/jw-02-2002/jw-0222-designpatterns.html

- et Vol 3 de mark Grand. Java Enterprise Design Patterns,
  - **ProtectionProxy**

Présentation rapide

- **JVM, ClassLoader & Introspection**
  - **Rapide ?**
    - **afin de pouvoir lire les exemples présentés**

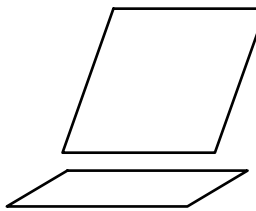
La JVM

```
public class Exemple{
   public void .....
}
```

javac Test.java

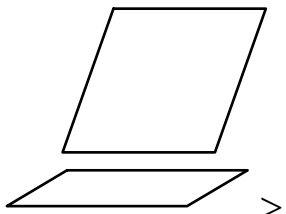
```
1100 1010 1111 1110 1011 1010 1011 1110
0000 0011 0001 1101 ..........................
```

"Exemple.class"
local ou distant



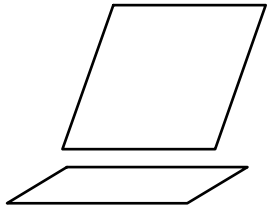
Sun

% **java** Test



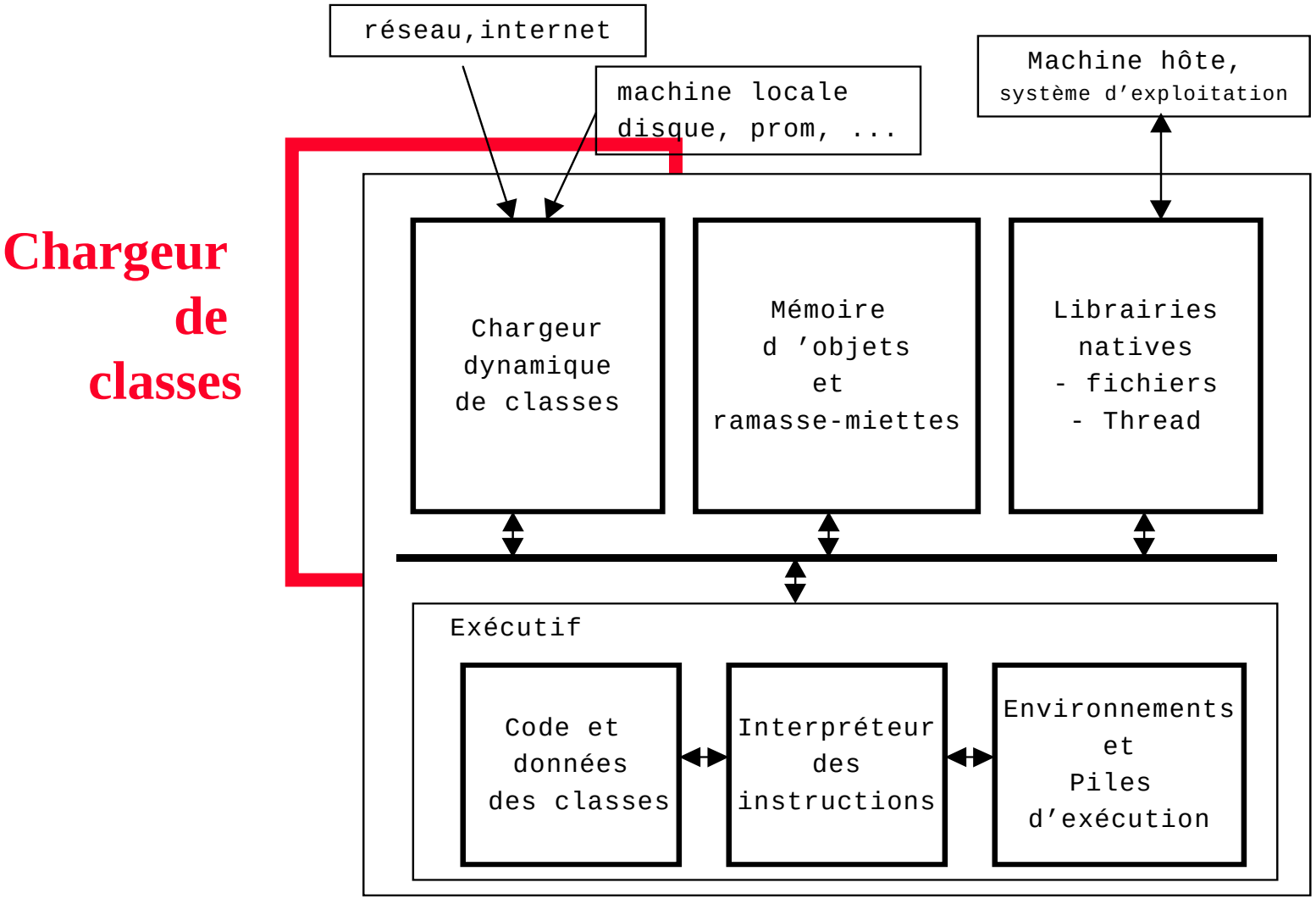
TINI, SunSPOT

> **java** Test



- **java** Test
- Ou un navigateur Muni d'une JVM

JVM : architecture simplifiée



- **Java Virtual Machine**
  - **Chargeur de classes et l'exécutif**
  - **Extrait de http://www.techniques-ingenieur.fr**

Chargeurs de classe

- **Chargement dynamique des *.class***
  - **Au fur et à mesure, en fonction des besoins**
    - **Chargement paresseux, tardif, *lazy***

- **Le chargeur**
  - **Engendre des instances de *java.lang.Class***
  - **Maintient l'arbre d'héritage en interne (instanceof, super,…)**

- **Les instances de la classe *java.lang.Class***
  - **« Sont des instances comme les autres »**
  - **Gérées par le ramasse-miettes**

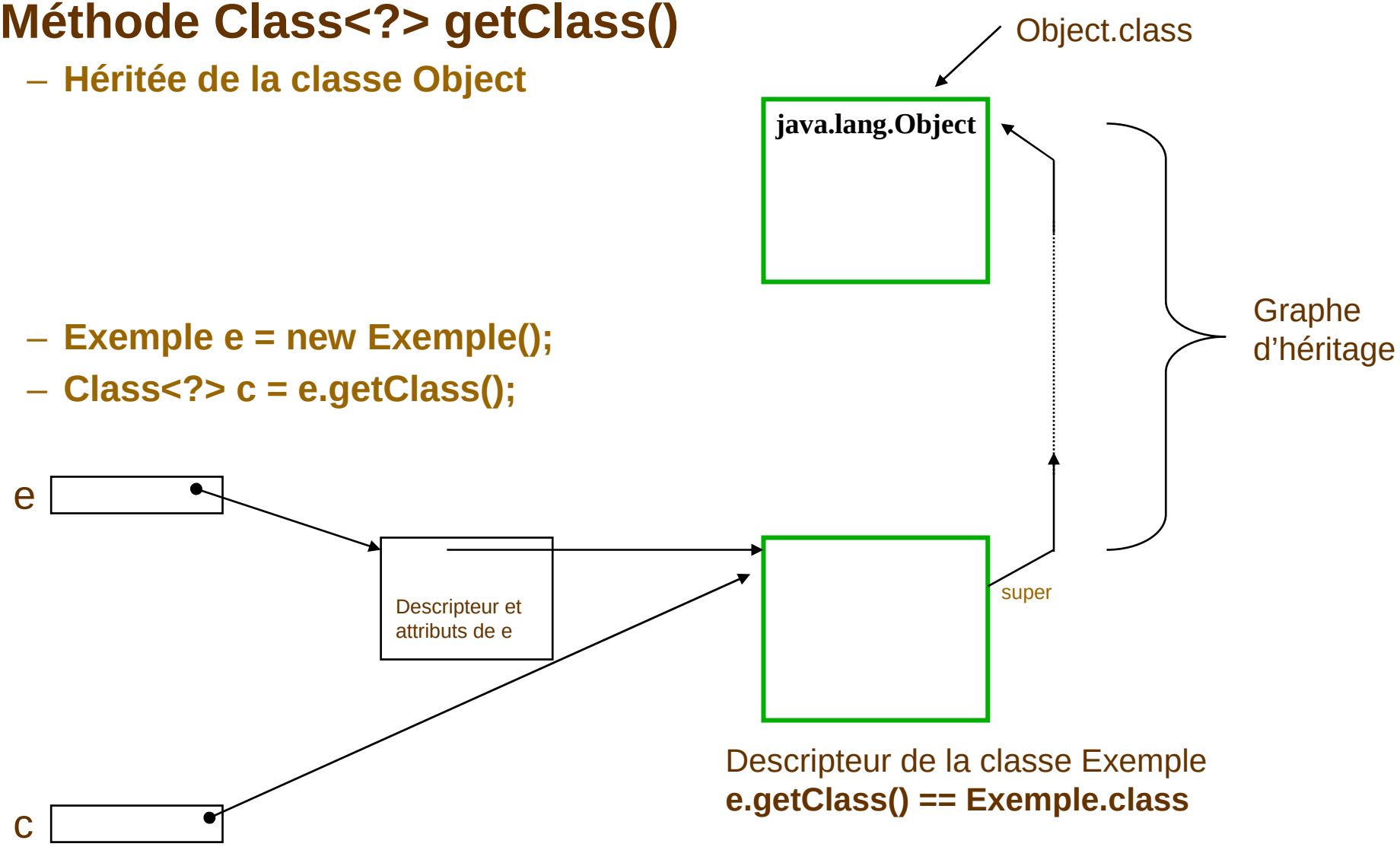
Sommaire : Classes et *java.lang.Class*

- **Le fichier *.class***
  - **Une table des symboles et des instructions (bytecode)**
  - **Une décompilation est toujours possible …**
    - **Du .class en .java …**
    - **Il existe des « obfuscateurs »**

- **Le chargeur de *.class***
  - **Les chargeurs de classes → de .class en classe Class**

Classe Class, getClass

- **Méthode Class<?> getClass()**
  - **Héritée de la classe Object**
  - **Exemple e = new Exemple();**
  - **Class<?> c = e.getClass();**



Introspection

- **Classe Class et Introspection**
  - **java.lang.Class;**
  - **java.lang.reflect.*;**

Les méthodes

Constructor[] getConstructors()
Field[] getFields()
…
Method[] getMethods()
…
Class<?>[] getInterfaces()

- **static Class<?> forName(String name);**
- **static Class<?> forName(String name, boolean init, ClassLoader cl);**
- **ClassLoader getClassLoader()**

Chargement d'une classe

- **Implicite** *et tardif*
  - **Exemple e;** *// pas de chargement*
  - **Exemple e = new Exemple();** *// chargement (si absente)*
  - **Class<Exemple> classe = Exemple.class;** *// chargement (si absente)*
    - **Équivalent à Class<?> classe = Class.forName("Exemple") ;**

- **→ Il existe un chargeur de classes par défaut**

Chargement d'une classe

- **Explicite** *et immédiat*
  - **String unNomdeClasse = XXXXX**

  - ***//* avec le chargeur de classes par défaut**
  - **Class.forName(unNomDeClasse)**

  - ***//* avec un chargeur de classes spécifique**
  - **Class.forName(unNomDeClasse, unBooléen, unChargeurDeClasse)**

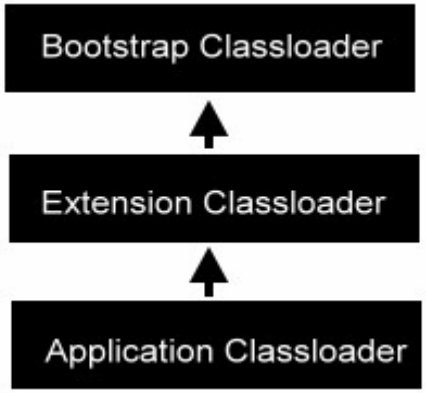
  - **unChargeurDeClasse.loadClass (unNomDeClasse)**

ClassLoader de base

- **ClassLoader**
  - **Par défaut celui de la JVM**
    - ***Bootstrap ClassLoader*** **en natif (librairies de base, rt.jar)**
    - ***Extension ClassLoader*** **en Java (lib/ext)**
    - ***Application/System ClassLoader*** **par défaut**

    - ***Bootstrap*** **parent-de** ***Extension*** **parent-de** ***Application***

  - ***ClassLoader*** **prédéfinis**



ClassLoader prédéfinis

- **SecureClassLoader**
  - **la racine**

java.net

Class URLClassLoader

```
java.lang.Object
  └ java.lang.ClassLoader
      └ java.security.SecureClassLoader
          └ java.net.URLClassLoader
```

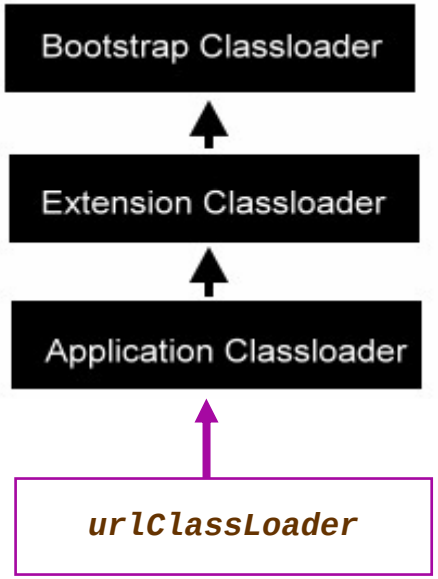
Direct Known Subclasses:

MLet

- **URLClassLoader**
  - **Utilisé depuis votre navigateur, par exemple**

URLClassLoader : un exemple

- **Chargement distant de fichier *.class* et exécution de la méthode main**
  - *Depuis cette archive*
    - http://jfod.cnam.fr/progAvancee/classes/utiles.jar
  - Ou bien un .class à cette URL
    - http://jfod.cnam.fr/progAvancee/classes/



1. *Création d'une instance de URLClassLoader*
2. *Son parent est le classLoader par défaut*
3. *Class<?> classe = forName(nom,init,urlClassLoader)*
   1. *nom le nom de la classe*
   2. *init : exécution des blocs statiques,*
      false: retardée à la première création d'instance
      true : au chargement,
      et dans les deux cas si ceux-ci n'ont pas déjà été exécutés
4. *Recherche de la méthode main par introspection*

URLClassLoader : un exemple

```
public class Execute{
 public static void main(String[] args)throws Exception{

  URL urlJars    =
    new URL("http://jfod.cnam.fr/progAvancee/classes/utiles.jar");

  URL urlClasses =
    new URL("http://jfod.cnam.fr/progAvancee/classes/");

  // par défaut le classloader parent est celui de la JVM
  URLClassLoader classLoader =
    URLClassLoader.newInstance(new URL[]{urlJars,urlClasses});


   Class<?> classe = Class.forName(args[0], true, classLoader);


    // exécution de la méthode  main ?  Comment ?
    // page suivante
}
```

Méthode main par introspection …

```
// page précédente
 URL urlJars    = new URL("http://jfod.cnam.fr/progAvancee/classes/utiles.jar");
URL urlClasses = new URL("http://jfod.cnam.fr/progAvancee/classes/");

// par défaut le classloader parent est celui de la JVM
URLClassLoader classLoader;
classLoader = URLClassLoader.newInstance(new URL[]{urlJars,urlClasses});
Class<?> classe = Class.forName(args[0], true, classLoader);
```

```
// à la recherche de la méthode main
 Method m = classe.getMethod("main",new Class[]{String[].class});

// recopie des paramètres
 String[] paramètres = new String[args.length-1];
 System.arraycopy(args,1,paramètres,0,args.length-1);

// exécution de la méthode main
 m.invoke(null, new Object[]{paramètres});
```

usage java Execute UneClasse param1 param2 param3

Ce fût une présentation rapide …

- **Terminée !**

- **Rappel : Une aide à la lecture des sources associés aux patrons Proxy, DynamicProxy et Interceptor**

Patron Proxy Objectifs

- **Proxy comme Procuration**
  - **Fournit à un tiers objet un mandataire, pour contrôler l 'accès à cet objet**

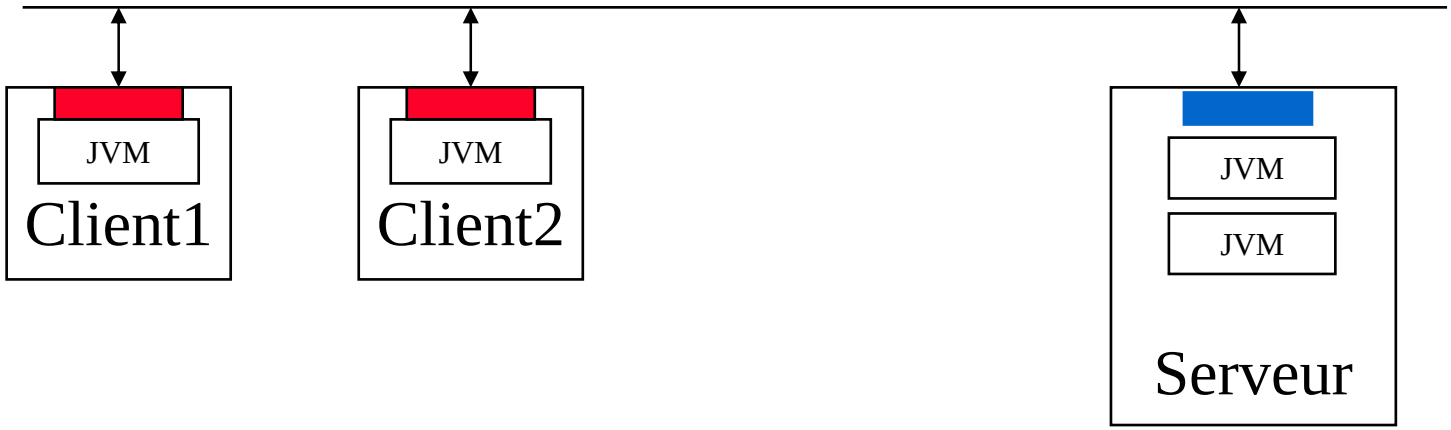
- **Alias**
  - **Subrogé (surrogate)**

- **Motivation**
  - **contrôler**
  - **différer**
  - **optimiser**
  - **sécuriser**
  - **accès distant**
  - **...**

Un exemple ad'hoc : appels distants

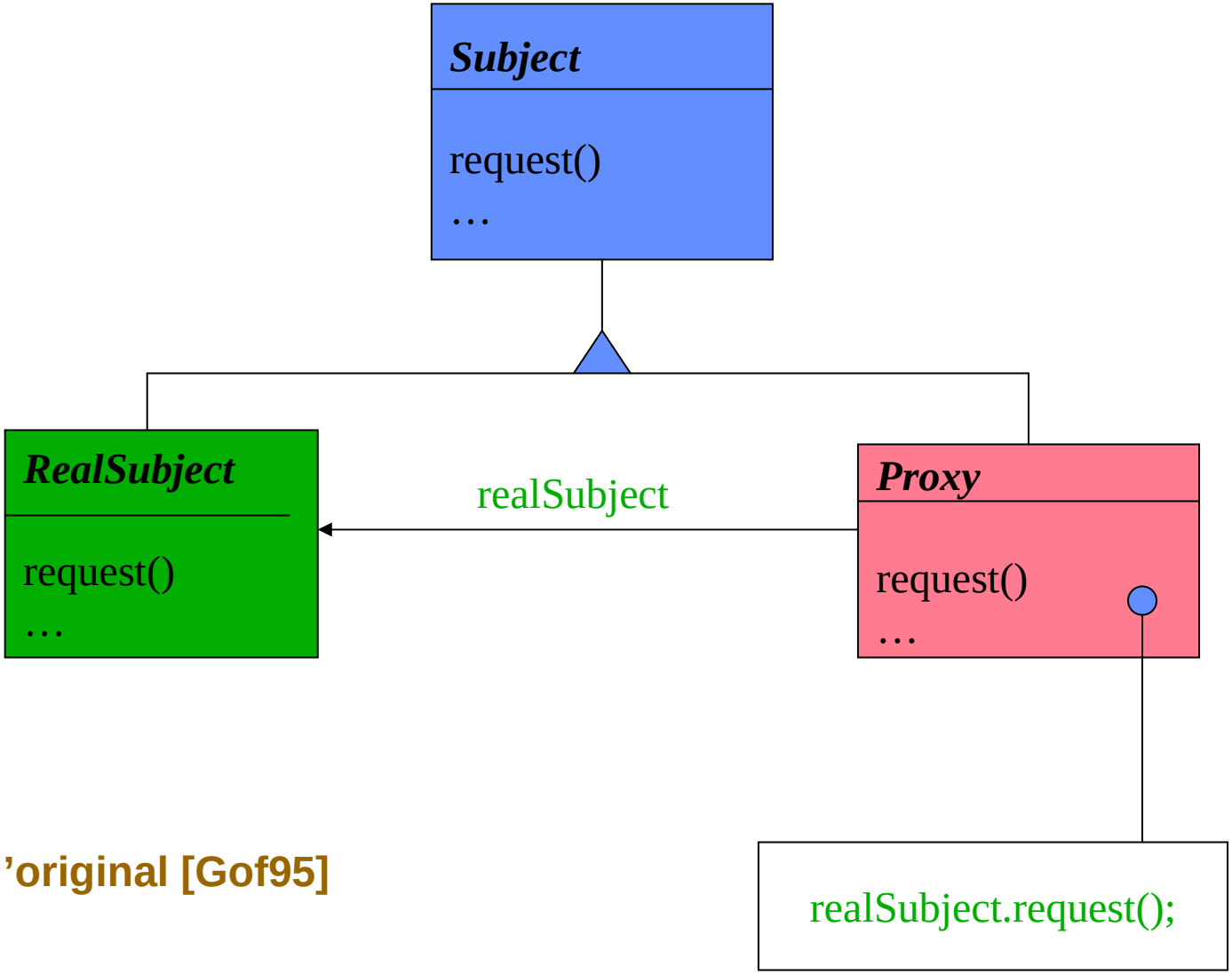
- **Les clients déclenchent des méthodes côté serveur**
- **Comment**
  - **Assurer la transmission des paramètres et du résultat ?**
  - **Gérer les exceptions levées côté serveur et à destination d 'un client ?**
  - **Être le plus « transparent » possible pour le programmeur ?**
- **Un mandataire s'en charge !**

Exemple : appels distants



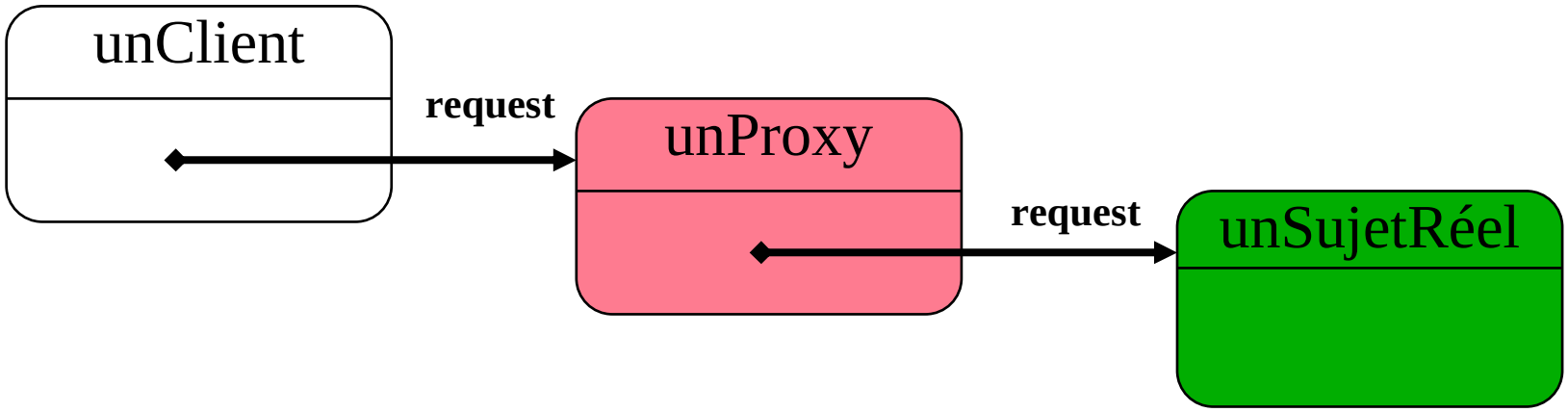
– **Un mandataire est téléchargé par les clients depuis le serveur**
  - **Appelé parfois Client_proxy, voir Zdun et Schmidt…**
  - **Une souche, un stub …**
  - **Un service se charge de sélectionner la bonne méthode**

Mandataire : diagramme UML



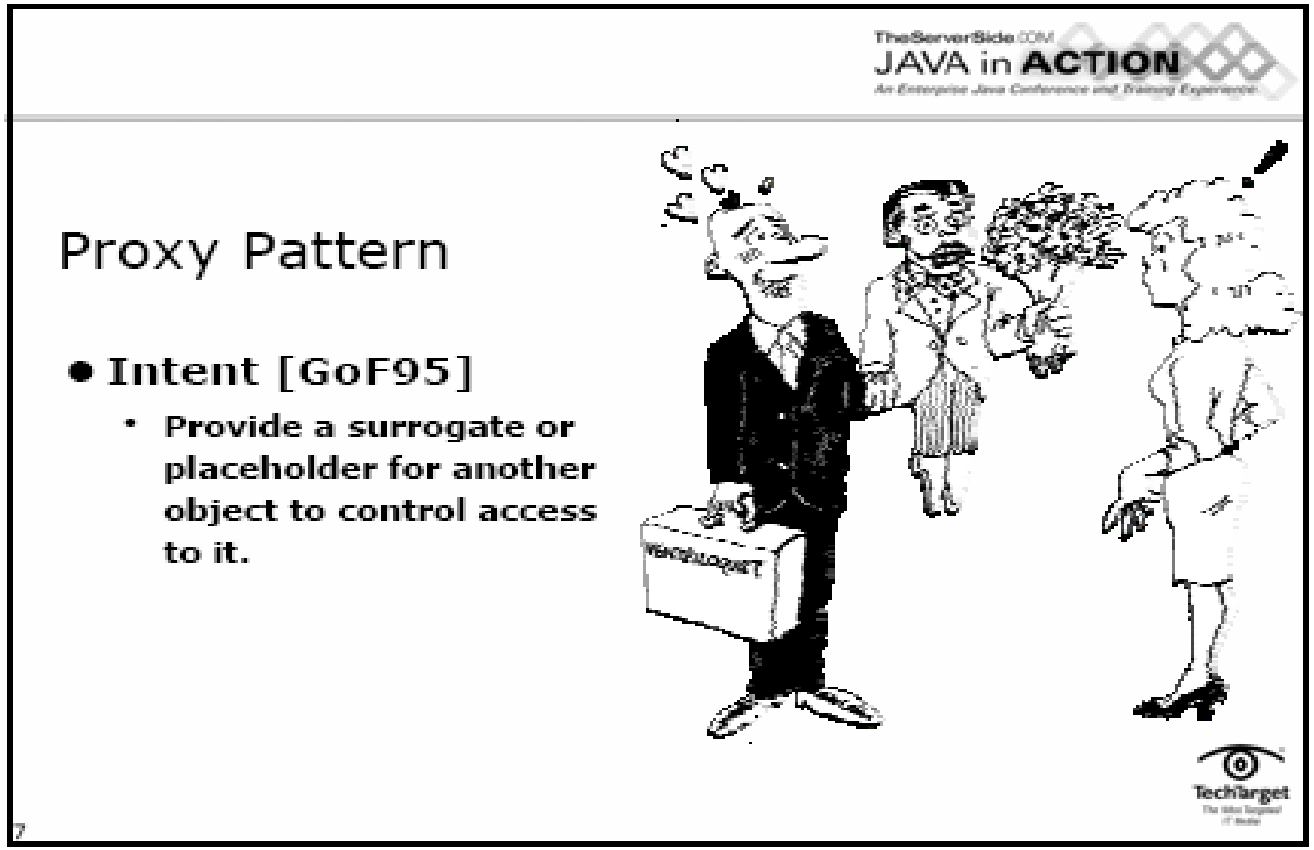
– **L 'original [Gof95]**

Un exemple possible à l'exécution



- **Une séquence et des instances possibles**
  - **unProxy.request() → unSujetRéel.request()**

Un exemple fleuri



- Un exemple de mandataire …
  - http://www.theserverside.com/tt/articles/content/JIApresentations/Kabutz.pdf

le Service/Sujet

Service

offrir(Bouquet b)
…

```
public interface Service{
   /** Offrir un bouquet de fleurs.
    * @param b le bouquet
    * @return réussite ou échec ...
    */
   public boolean offrir(Bouquet b);

}
```

Une implémentation du Service

ServiceImpl

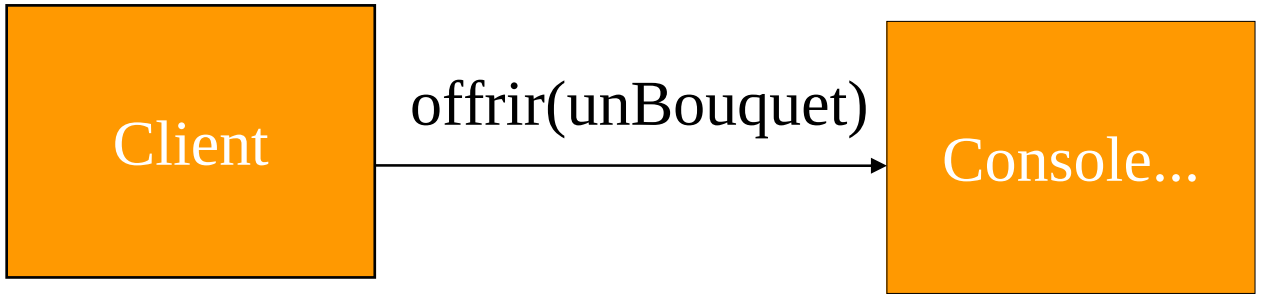
offrir(Bouquet b)
…

```
public class ServiceImpl implements Service{

    public boolean offrir(Bouquet b){
      System.out.println(" recevez ce bouquet : " + b);
      return true;
    }

}
```

Offrir en « direct »



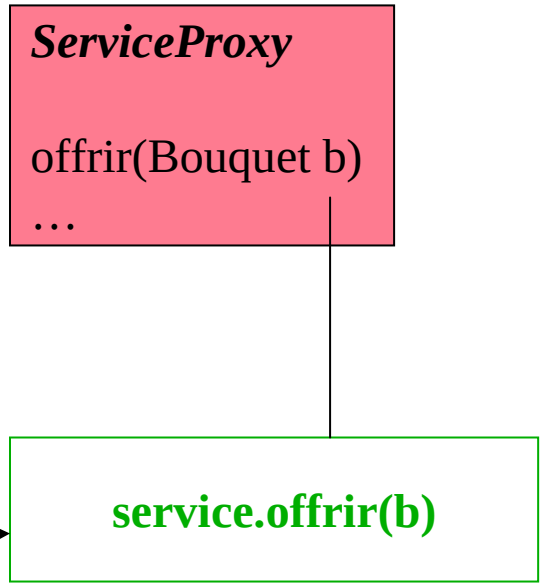

```
Service service = new ServiceImpl();
Bouquet unBouquet=…
boolean résultat = service.offrir(unBouquet);
```

Le mandataire

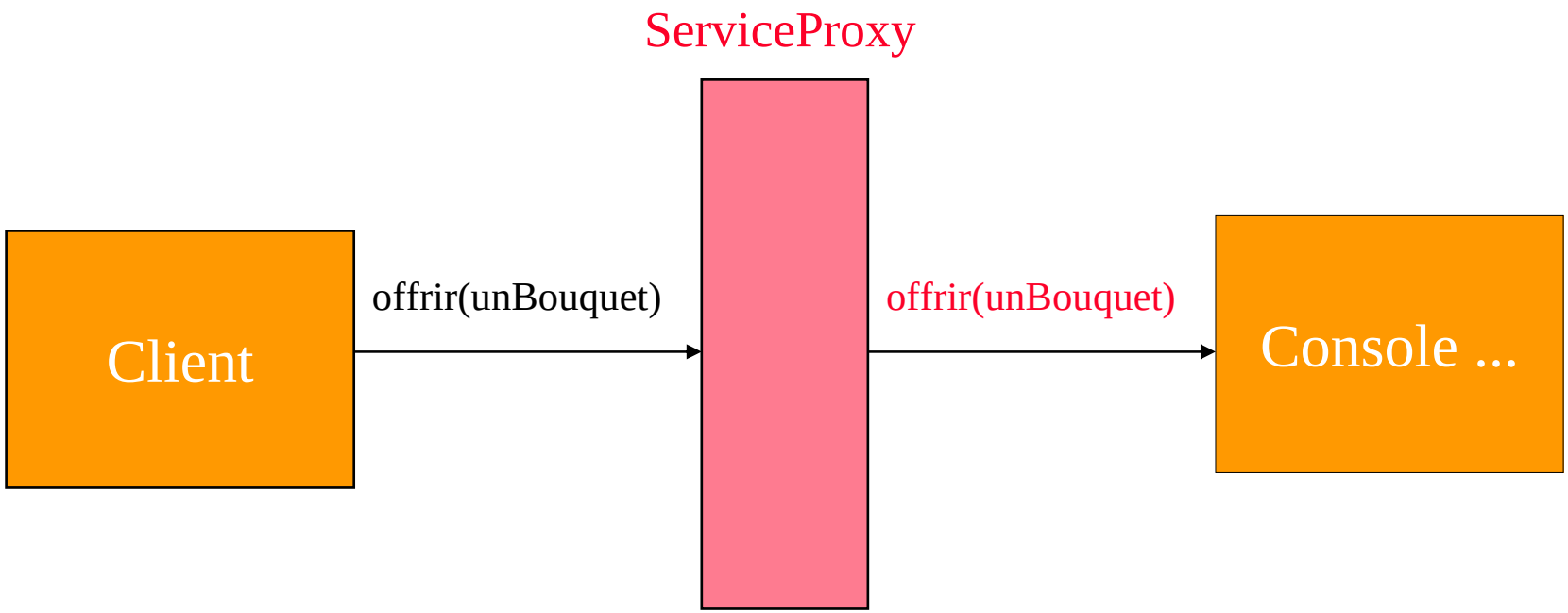
```
public class ServiceProxy implements Service{
   private Service service;

   public ServiceProxy(){
         this.service = new ServiceImpl();
   }

   public boolean offrir(Bouquet bouquet){
     boolean résultat;
     System.out.print(" par procuration : ");
     résultat = service.offrir(bouquet);
     return résultat;
   }    }
```

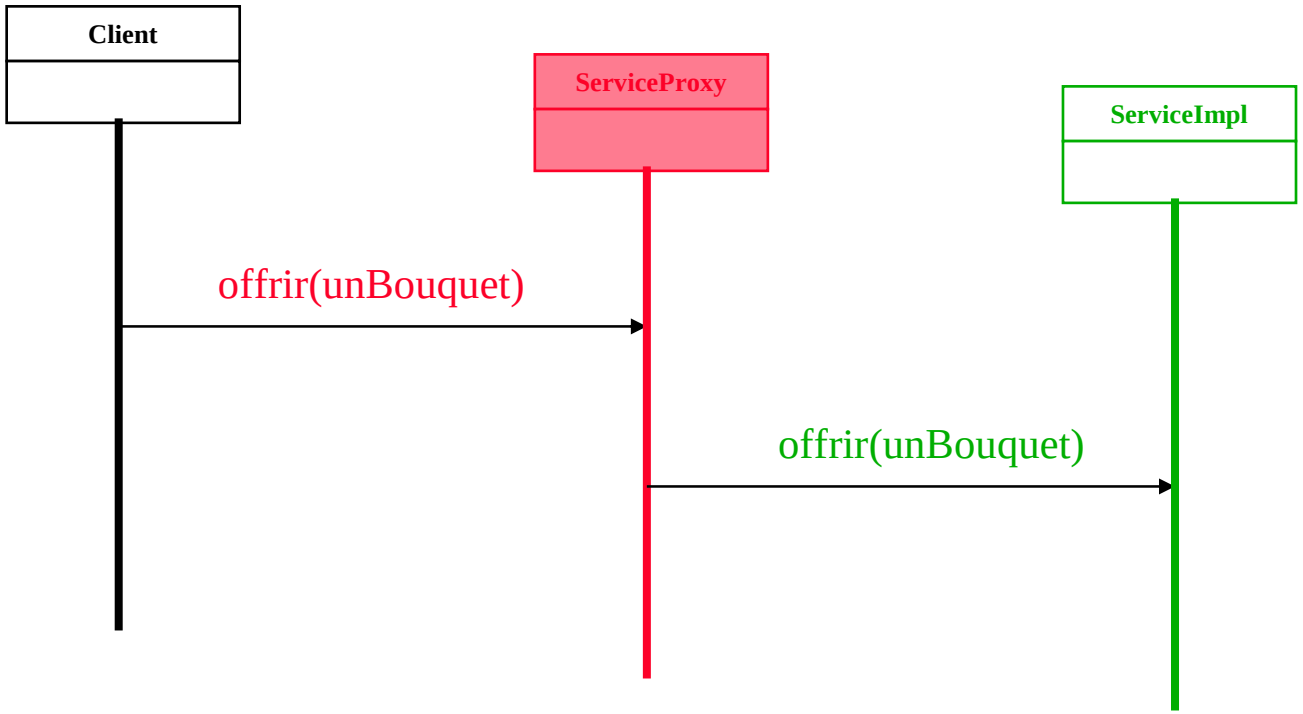


Offrir par l'intermédiaire de



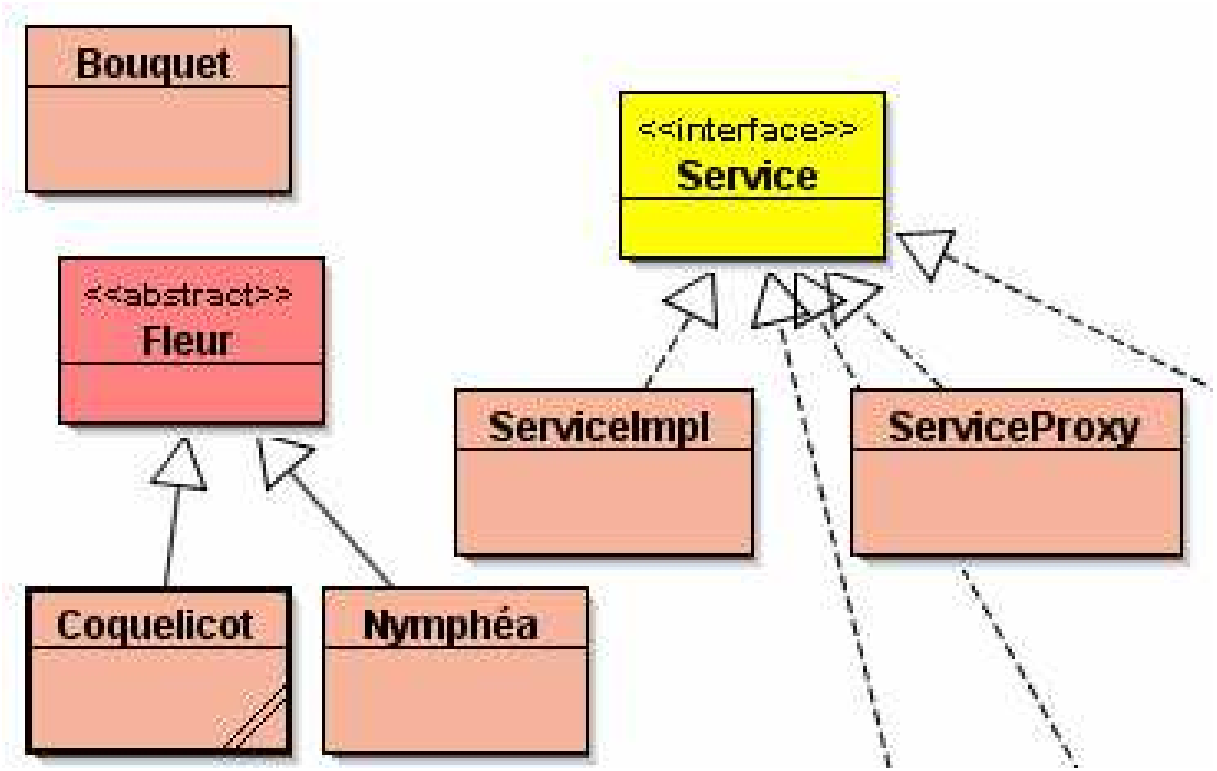

```
Service service = new ServiceProxy();
Bouquet unBouquet=…
boolean résultat = service.offrir(unBouquet);    // idem
```

Séquence …



- **Sans commentaire**

Démonstration



VirtualProxy

- **Création d ’objets « lourds » à la demande**
  - **Coût de la création d ’une instance …**
  - **Chargement de la classe et création d ’un objet seulement si nécessaire**
    - **Au moment ultime**
  - **Exemple des fleurs revisité**
    - **Seulement lors de l’exécution de la méthode offrir le service est « chargé »**

Virtual (lazy) Proxy

```
public class VirtualProxy implements Service{
   private Service service;

  public VirtualProxy(){
        // this.service = new ServiceImpl(); en commentaire … « lazy » … ultime
   }

 public boolean offrir(Bouquet bouquet){
    boolean résultat;
    System.out.print(" par procuration, virtualProxy (lazy) : ");
    this.service = getServiceImpl();
    résultat = service.offrir(bouquet);

    return résultat;
   }
```

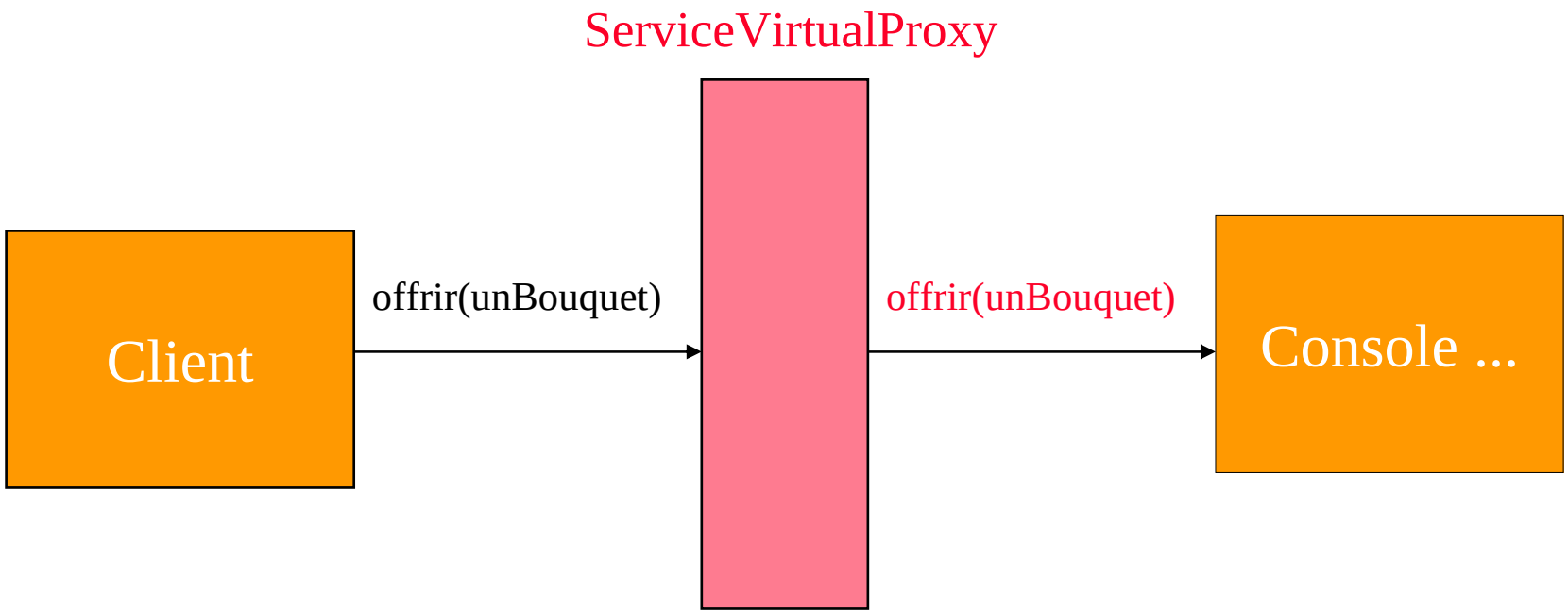
VirtualProxy, suite & Introspection

```
private Service getServiceImpl(){
      if(service==null){
        try{
          Class classe = Class.forName("ServiceImpl");                    // si classe absente alors chargement en JVM
         Class[] typeDesArguments = new Class[]{};                        // à la recherche du constructeur sans paramètre
          Constructor cons = classe.getConstructor(typeDesArguments);   // le constructeur
          Object[] paramètres = new Object[]{};                            // sans paramètre
          Service service = (Service)cons.newInstance(paramètres);                    // exécution
         // ou service = (Service) classe.newInstance(); de la classe Class
        }catch(Exception e){
        }
       }
       return service;
 }
```

- **ou bien sans introspection ...** *Plus simple …*

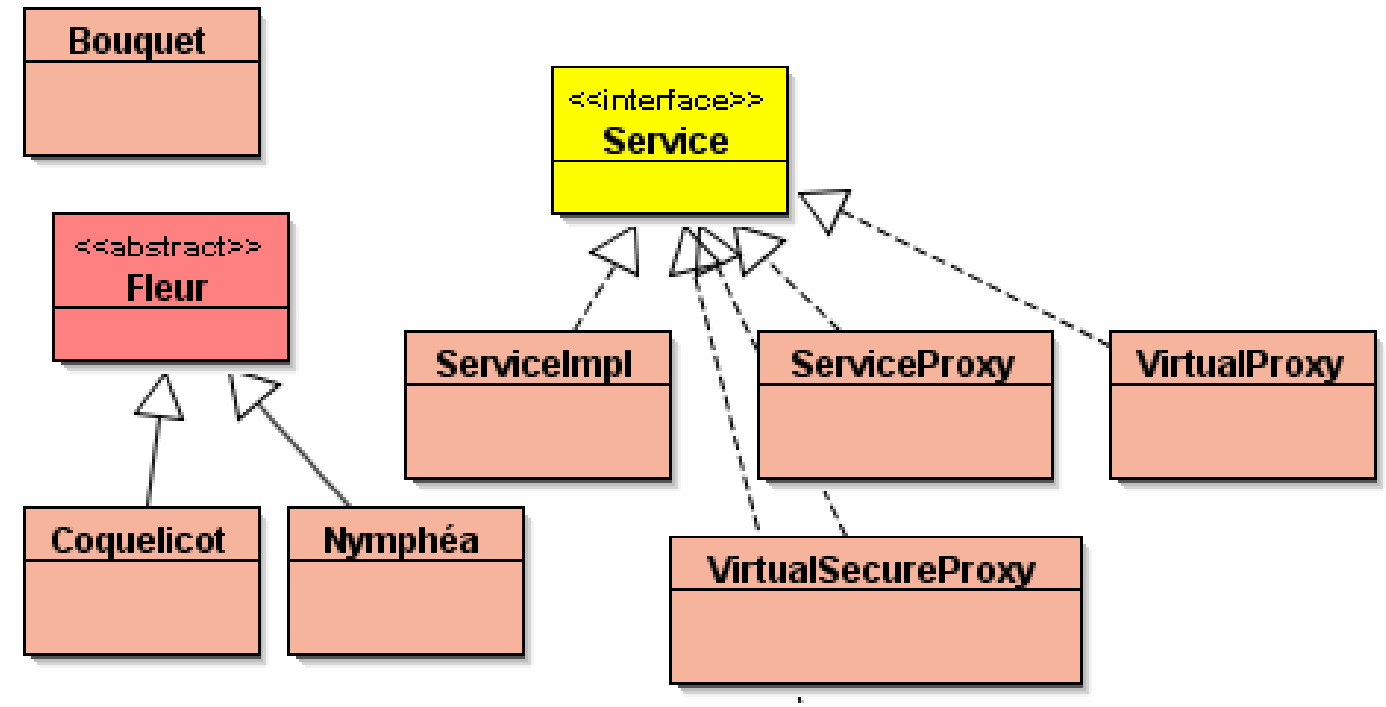
```
private Service getServiceImpl(){
      if(service==null)
         service = new ServiceImpl();
       return service;
 }
```

Offrir par l'intermédiaire de




```
Service service = new ServiceVirtualProxy();
Bouquet unBouquet=…
boolean résultat = service.offrir(unBouquet);
```

La « famille proxy» avec BlueJ



La famille s'agrandit

- **SecureProxy**
  - sécuriser les accès
- **ProtectionProxy**
  - se protéger
- **Introspection + Proxy = DynamicProxy**
  - création dynamique de mandataires
- **RemoteProxy …**
  - accès distant

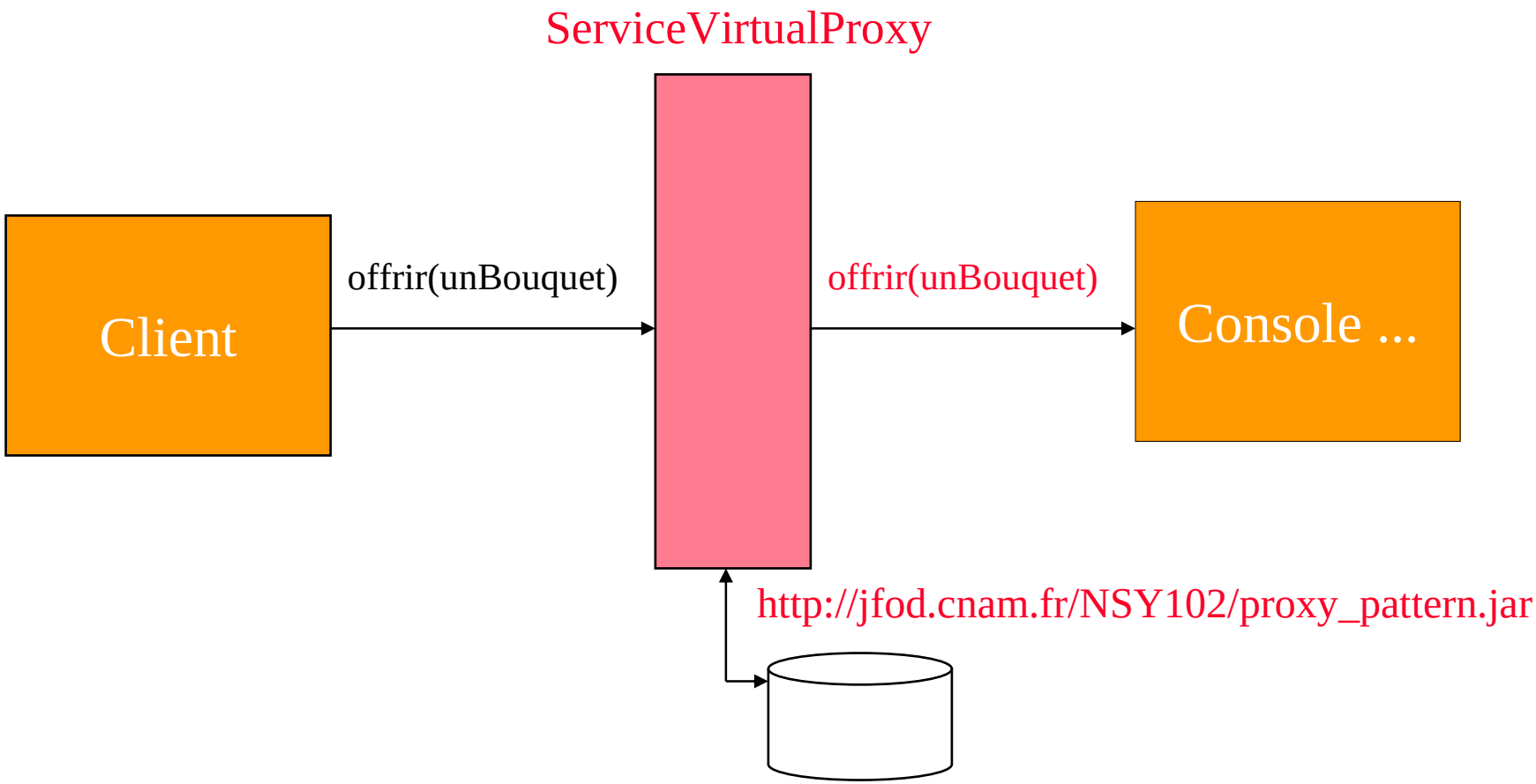
Sécurité … VirtualProxy bis

- **URLClassLoader hérite de SecureClassLoader**
  - **Il peut servir à « vérifier » un code avant son exécution**
  - **associé à une stratégie de sécurité (« java.policy »)**

```
private Service getServiceImpl(){                    // source complet voir VirtualProxy
    if(service==null){
     try{
       ClassLoader classLoader = new URLClassLoader(
              new URL[]{new URL("http://jfod.cnam.fr/nsy102/proxy_pattern.jar")});
       Class classe = Class.forName("ServiceImpl",true, classLoader);
      service = (Service) classe.newInstance();

     }catch(Exception e){
      //e.printStackTrace();
     }
    }
     return service;
 }
```

Offrir par l'intermédiaire de




```
Service service = new ServiceVirtualProxy();
Bouquet unBouquet=…
boolean résultat = service.offrir(unBouquet);
```

Résumé, petit bilan intermédiaire

- **Procuration à un tiers,** *en général connu à la compilation*
  - **Proxy**
    - **Un contrôle complet de l 'accès au sujet réel**
  - **VirtualProxy**
    - **Une prise en compte des contraintes de coûts en temps d 'exécution**
  - **ProtectionProxy en annexe**
    - **liée à une stratégie de sécurité**

- **Et si**
  - **Le sujet réel n'est connu qu'à l'exécution ?**
  - **Les méthodes d'une classe ne sont pas toutes les bienvenues ?**
    - **cette liste est dynamique …**

Un cousin plutôt dynamique et standard

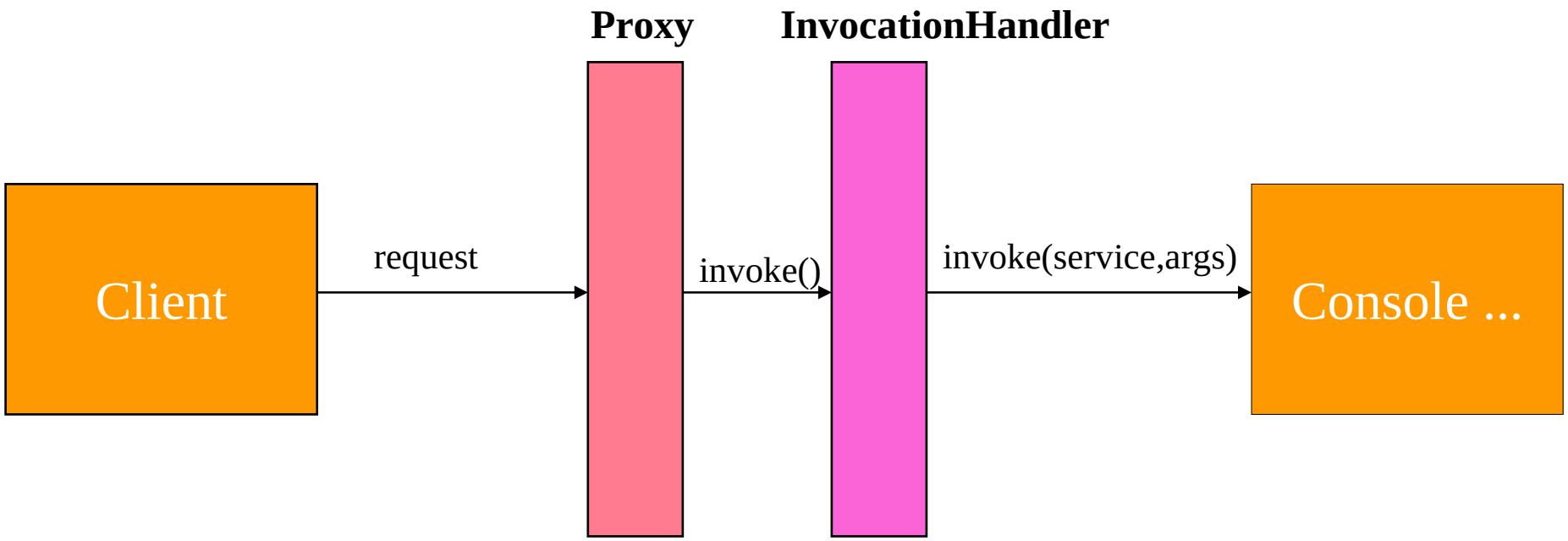
- **extrait de l 'API du J2SE Dynamic Proxy** *(depuis 1.3)*
  - **http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/InvocationHandler.html**
  - **http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/Proxy.html**

- **Génération de byte code à la volée**
  - **Rare exemple en java**

- **Paquetages**

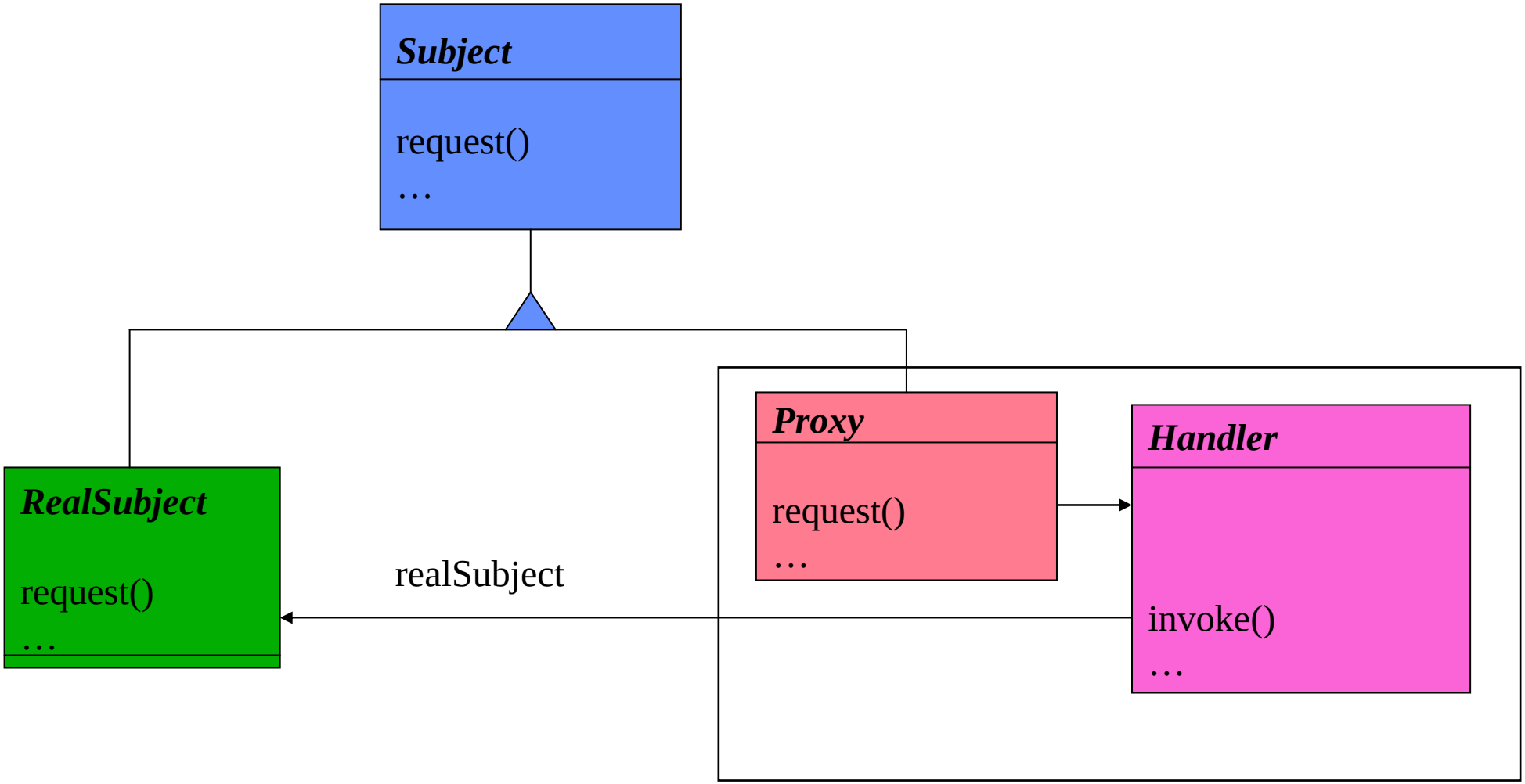
- **java.lang.reflect et java.lang.reflect.Proxy**

Proxy + InvocationHandler



- **1) InvocationHandler**
- **2) Création dynamique du Proxy, qui déclenche la méthode invoke de « InvocationHandler »**

Le diagramme UML revisité



- **Vite un exemple …**

L'interface InvocationHandler

– L 'interface à implémenter pour chaque instance de *proxy* dynamique :

interface java.lang.reflect.InvocationHandler;

– ne contient qu 'une seule méthode !

Object invoke(Object proxy, Method m, Object[] args) throws Throwable;

- proxy : l'instance proxy invoquée
- m : la méthode choisie
- args : les arguments de cette méthode

Handler implements InvocationHandler

```
public class Handler implements InvocationHandler{

    private Service service;

    public Handler(){
        this.service = new ServiceImpl();      // les fleurs : le retour
    }


  public Object invoke(Object proxy, Method method, Object[] args)
                                                    throws Exception{

    return method.invoke(service, args); // par introspection ici
  }

}
```

Note : La méthode invoke est déclenchée par le mandataire dynamique …

mandataire créé par une méthode ad'hoc toute prête newProxyInstance

Création dynamique du Proxy/Mandataire

public static Object **newProxyInstance**(ClassLoader *loader*,
Class[] interfaces,
InvocationHandler h) throws
….

crée <u>dynamiquement un mandataire</u>

spécifié par le chargeur de classe *loader*

lequel implémente les interfaces *interfaces,*

la méthode h.invoke sera appelée par l'instance du Proxy retournée

retourne une instance du *proxy*

Méthode de classe de la classe java.lang.reflect.Proxy;

Exemple

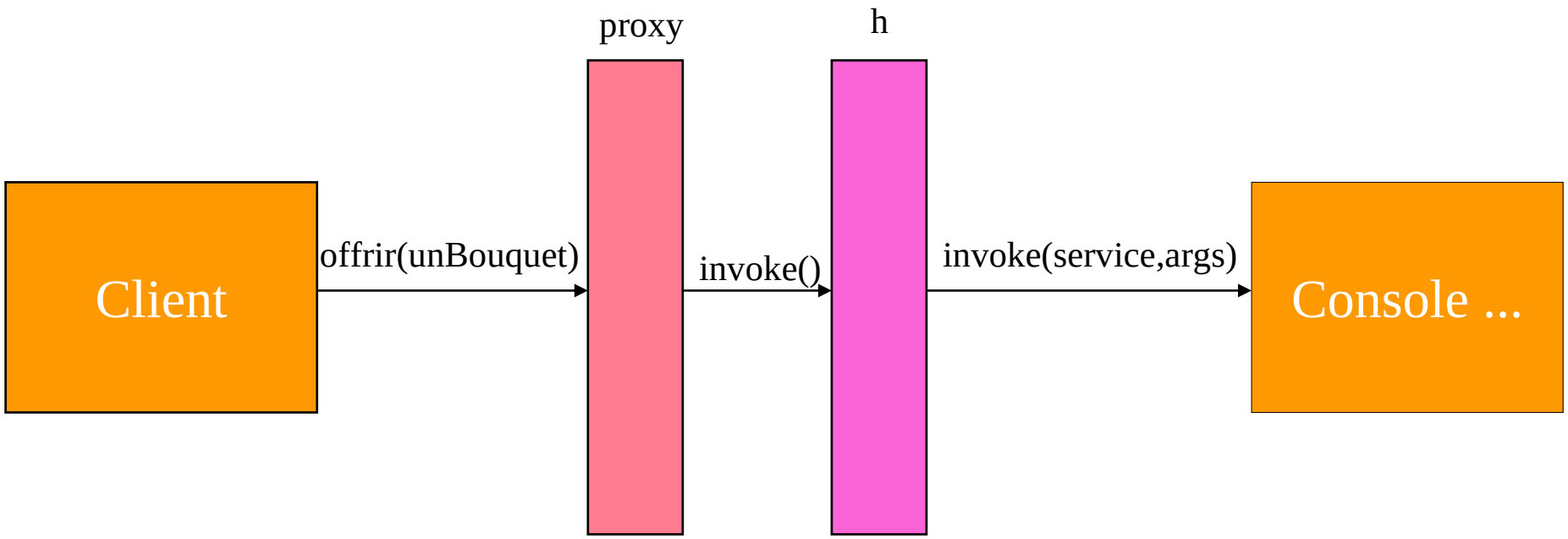
```
// obtention du chargeur de classes
  ClassLoader cl = Service.class.getClassLoader();

//  l'interface implémentée par le futur mandataire
  Class[] interfaces = new Class[]{Service.class};

//  le mandataire Handler
  InvocationHandler h = new Handler();


Service proxy = (Service)
  Proxy.newProxyInstance(cl,interfaces,h);
```

Dynamic Proxy



- Service proxy = (Service) Proxy.newProxyInstance(**cl**,**interfaces**,**h**);
- **boolean résultat = proxy.offrir(unBouquet);**

Proxy.newProxyInstance : du détail

// Appel de cette méthode

```
public static Object newProxyInstance(
                ClassLoader loader,
                Class[] interfaces,
                InvocationHandler h)   throws .... {
```

// ou en bien moins simple …

```
   ClassLoader loader = Service.class.getClassLoader();
   Class[] interfaces = new Class[]{Service.class};
   InvocationHandler h = new Handler();
   Class cl = Proxy.getProxyClass( loader, interfaces);
   Constructor cons = cl.getConstructor(new Class[] { InvocationHandler.class } );
  return  cons.newInstance(new Object[] { h });
```

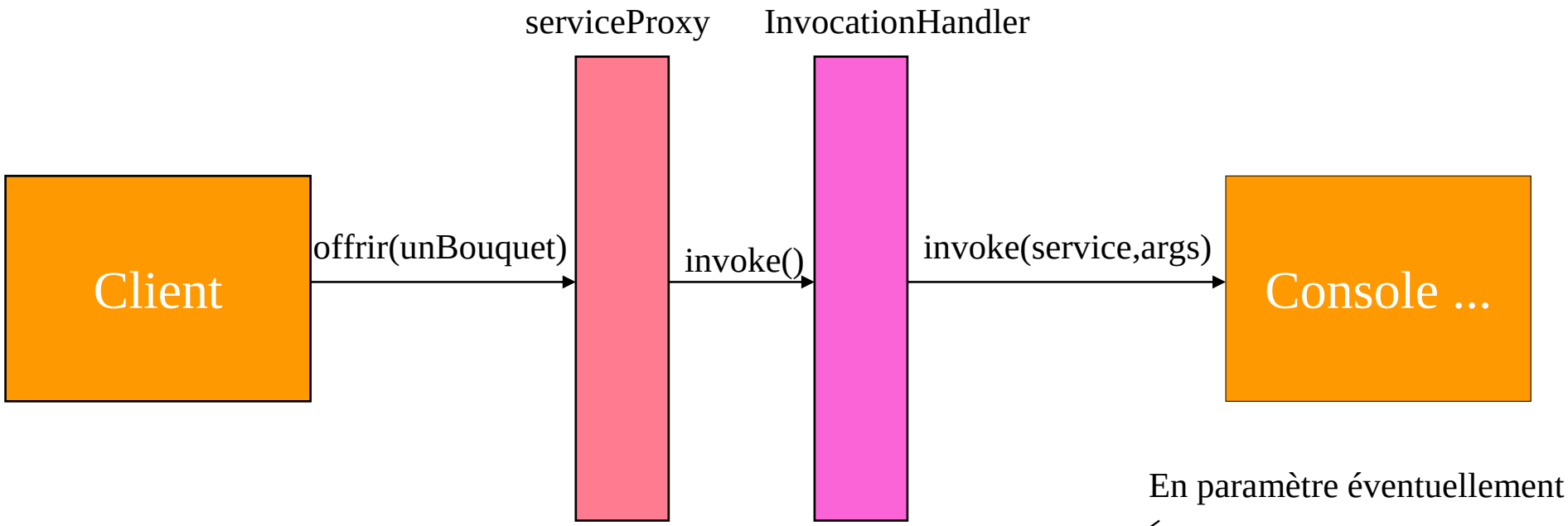
note: un « fichier.class » est généré « à la volée »

nom de ce fichier : proxyPkg + "$Proxy" + num;

voir également la méthode isProxyClass(Class c)

par curiosité java.lang.reflect.Proxy.java

VirtualSecureProxy le retour




```
ClassLoader cl =
    new URLClassLoader(
            new URL[]{new URL("http://jfod.cnam.fr/nsy102/proxy_pattern.jar")});

    Class classe = Class.forName("Handler",true, cl);
    InvocationHandler handler = (InvocationHandler)classe.newInstance();
    Service service = (Service) Proxy.newProxyInstance(cl, new Class[]{Service.class},handler);

    boolean résultat = service.offrir(unBouquet);
}
```

Critiques / solutions

- **Connaissance préalable des classes invoquées**
  - **Ici le Service de fleurs…**
- **Ajout nécessaire d ’un intermédiaire du mandataire**
  - **Lequel implémente InvocationHandler**
- *Compliqué … alors*
- **Rendre la délégation générique**
- **Abstraire le programmeur de cette étape …**

Plus générique ?

Le « Service » devient un paramètre du constructeur, Object target

```
public class GenericProxy implements InvocationHandler{

  private Object target;

  public GenericProxy(Object target){
    this.target = target;
  }

  public Object invoke(Object proxy, Method method, Object[] args)
                                                    throws Throwable{
    return method.invoke(target, args);
  }
}
```

- *Serait-ce un décorateur ?*

Service de fleurs : Nouvelle syntaxe

```
ClassLoader cl = Service.class.getClassLoader();

Service service = (Service) Proxy.newProxyInstance(
                                                    cl,
                                                    new Class[]{Service.class},
                                                    new GenericProxy(new ServiceImpl()));

résultat = service.offrir(unBouquet);
```

La délégation est effectuée par la classe GenericProxy, compatible avec n 'importe quelle interface ...

Donc un mandataire dynamique d'un mandataire dynamique,
ou plutôt une simple décoration

Une autre syntaxe avec un DebugProxy…

```
public class DebugProxy implements InvocationHandler{
  private Object target;

  public DebugProxy(Object target){ this.target = target;}

// tout ce qu'il a de plus générique
  public static Object newInstance(Object obj){return
   Proxy.newProxyInstance(obj.getClass().getClassLoader(),
                                obj.getClass().getInterfaces(),
                                new DebugProxy(obj));
  }

public Object invoke(Object proxy, Method method, Object[] args)
   throws Throwable{
    Object résultat=null;
    // code avant l'exécution de la méthode, pré-assertions,
   vérifications du nom des méthodes ...
    résultat = method.invoke(target, args);
    // code après l'exécution de la méthode, post-assertions,
   vérifications, ...
    return résultat;
   }
}
```

DebugProxy invoke, un exemple

```
public Object invoke(Object proxy, Method method, Object[] args) throws
   Throwable{
     Object résultat=null;
     long top = 0L;
     try{
       top = System.currentTimeMillis();

       résultat = method.invoke(target, args);
       return résultat;

     }catch(InvocationTargetException e){  // au cas où l'exception se produit
                                           // celle-ci est propagée
        throw e.getTargetException();

     }finally{
        System.out.println(" méthode appelée : " + method +
                           " durée : "  + (top - System.currentTimeMillis()));
     }
}
```

Un autre usage de DebugProxy

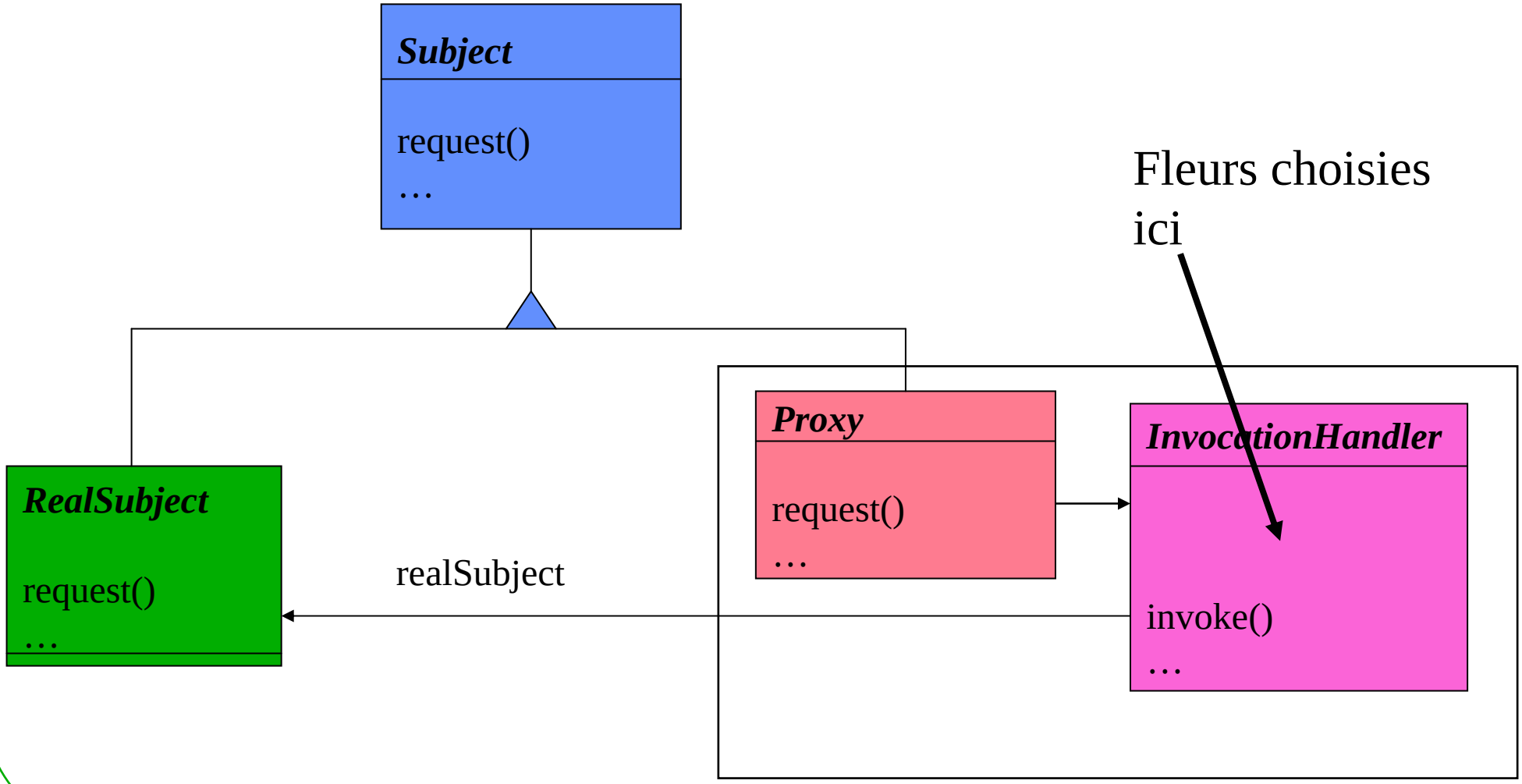
```
Service service = (Service) DebugProxy.newInstance(new ServiceImpl());
résultat = service.offrir(unBouquet);
```

ou bien avec 2 mandataires

```
Service service2 = (Service) DebugProxy.newInstance(new ServiceProxy());
résultat = service2.offrir(unBouquet);
```

Un mandataire particulier

- **Retrait par le mandataire de certaines fleurs du bouquet…**



Démonstration

DynamicProxy se justifie ...

- **A la volée, un mandataire « particulier » est créé**

- **Le programme reste inchangé**

la classe ProxyParticulier

```
public class ProxyParticulier implements InvocationHandler{
  private Class typeDeFleurARetirer;
   private Service service;

  public ProxyParticulier(Service service, Class typeDeFleurARetirer){
    this.service = service; this.typeDeFleurARetirer = typeDeFleurARetirer;
  }


  public Object invoke(Object proxy, Method method, Object[] args) throws
  IllegalAccessException{
    try{
      if( method.getName().equals("offrir")){
        Bouquet b = (Bouquet) args[0];
        Iterator<Fleur> it = b.iterator();
        while( it.hasNext())
          if(it.next().getClass().equals(typeDeFleurARetirer)) it.remove();
        return method.invoke(service,new Object[]{b});
     }else{ throw new IllegalAccessException();}
   }catch(InvocationTargetException e){ …}
   return null;}
```

ProxyParticulier suite

en entrée le service

en retour le mandataire

```
public static Service getProxy(Service service, Class FleurASupprimer){
      return (Service) Proxy.newProxyInstance(
                        service.getClass().getClassLoader(),
                        service.getClass().getInterfaces(),
                        new ProxyParticulier (service, FleurASupprimer));
    }
}
```

Usage du ProxyParticulier

```
Service service = ProxyParticulier.getProxy(new ServiceImpl(), Coquelicot.class);
boolean résultat = service.offrir(unBouquet);
```

```
Service service = ProxyParticulier.getProxy(new ServiceImpl(), Tulipe.class);
boolean résultat = service.offrir(unAutreBouquet);
```

Les tulipes et les coquelicots se fanent vite …

Un autre exemple, une autre syntaxe …

un mandataire de contrôle des accès à une liste (List<T>)

```
public class Controle {


 public static <T> List<T> acces(List<T> liste, String[] autorisées){
    return (List<T>) Proxy.newProxyInstance(
                              Controle.class.getClassLoader(),
                              new Class<?>[]{List.class},
                              new Filtre<T>(liste,autorisées));
                                        }
```

- // classe Filtre interne et statique page suivante
- }

Classe de filtrage

```
private static class Filtre<T> implements java.lang.reflect.InvocationHandler{
    private List<T> liste;
    private String[] autorisées;

    private Filtre(List<T> liste,String[] autorisées){
      this.liste = liste;
      this.autorisées = autorisées;
    }


     public Object invoke(Object proxy, Method m, Object[] args)throws Throwable{
        try {
          for(String s : autorisées){
            if(s.equals(m.getName())){
              return m.invoke(liste, args);
            }
          }
        } catch (InvocationTargetException e) { throw e.getTargetException();
        } catch (Exception e) {
         throw new RuntimeException("unexpected invocation exception: " +
                                    e.getMessage());
         }
       throw new  RuntimeException(m.getName() + " est une méthode inhibée ");
    }

  }
```

Un client, Une Trace

```
public class ClientControle{

  public static void main(String[] args){
      List<Integer> liste = new ArrayList<Integer>();
      liste.add(3);
    // liste reçoit le mandataire
      liste = Controle.acces(liste,new String[]{"add"});
      liste.remove(3);
  }
}
```

```
java.lang.RuntimeException: remove est une méthode inhibée
        at Controle$Filtre.invoke(Controle.java:38)
        at $Proxy0.remove(Unknown Source)
        at ClientControle.main(ClientControle.java:17)
```

Autre essai syntaxique…

```
public class ProxyFactory{

  public static <T> T create(final Class<T> type, final InvocationHandler handler){
    return type.cast(Proxy.newProxyInstance(type.getClassLoader(),
                                            new Class<?>[] {type},
                                            handler));
  }




// exemple
  public static void main(String[] args){
    List<Integer> liste1 = new ArrayList<Integer>();
    List<Integer> liste2 = ProxyFactory.create(List.class, new Handler(liste1));

    liste2.add(3);

  }

  private static class Handler implements InvocationHandler{
    private Object target;
    public Handler(Object target){this.target = target; }

    public Object invoke(Object proxy, Method m, Object[] args)throws Throwable{
      // à compléter
      return m.invoke(target,args);
    }
}
```

Discussion : Patron décorateur ou Proxy ?

```
public class Decorateur implements Service{

  private Service décoré;

  public Decorateur (Service décoré){
    this. décoré = décoré;
  }

  public boolean offrir(Bouquet bouquet){
      return décoré .offrir(bouquet);
    }
}
```

DynamicProxy et le pattern décorateur

Le patron Procuration contrôle les accès

le patron Décorateur ajoute de nouvelles fonctionnalités

les classes GenericProxy et DebugProxy sont des décorateurs ….

- **discussion**

Le petit dernier, de caractère assez réservé

- **Une procuration à distance**
  - **DynamicProxy et RMI**

- **Java RMI en quelques lignes**
  - **appel distant d 'une méthode depuis le client sur le serveur**
  - **usage de mandataires (DynamicProxy) clients comme serveur**

« DynamicProxy » à la rescousse

- **Une instance du mandataire « qui s'occupe de tout » est téléchargée par le client**
  - **1) Le serveur inscrit ce mandataire auprès de l'annuaire**
  - **2) Le client interroge l'annuaire et obtient ce mandataire**

Quelques « légères » modifications

- **RMI étudié ensuite, juste pour l'exemple …**
  - **http://java.sun.com/j2se/1.5.0/docs/guide/rmi/relnotes.html**

- **l 'interface Service « extends » java.rmi.Remote**
- **La méthode offrir adopte la clause throws RemoteException**

- **Les classes Bouquet et Fleur deviennent « Serializable »**
- **La classe ServiceImpl devient un service RMI**
  - **Ajout d'un constructeur par défaut, avec la clause throws RemoteException**

- **Un Client RMI est créé**
  - **Simple n 'est-ce pas ?**

Le client RMI

```
public class ClientRMI{

public static void main(String[] args) throws Exception {
   System.setSecurityManager(new RMISecurityManager()); // rmi sécurité
   Bouquet unBouquet = créerUnBouquet (); // avec de jolies fleurs

   // recherche du service en intranet …
   Registry registry = LocateRegistry.getRegistry("localhost");

   // réception du mandataire
   Service service = (Service) registry.lookup("service_de_fleurs");

   // appel distant
   boolean résultat = service.offrir(unBouquet);
  }
}
```

réception du dynamicProxy côté Client,
le mandataire réalise les accès distants et reste transparent pour l 'utilisateur...

Le service distant, ServiceImpl revisité

```
public class ServiceImpl implements Service{

    public boolean offrir(Bouquet bouquet) throws RemoteException{
      System.out.println(" recevez ce bouquet : " + bouquet);
      return true;
    }
    public ServiceImpl() throws RemoteException{}


    public static void main(String[] args) throws Exception{
       System.setSecurityManager(new RMISecurityManager()); // liée à une straatégie
       ServiceImpl serveurRMI = new ServiceImpl();

      Service stub = (Service)UnicastRemoteObject.exportObject(serveurRMI, 0);
      Registry registry = LocateRegistry.getRegistry();
      registry.rebind("service_de_fleurs", stub);

     System.out.print("service_de_fleurs en attente sur " + InetAddress.getLocalHost().getHostName());
    }
}
```

UnicastRemoteObject.exportObject : création du « dynamicProxy »

Une trace d'exécution

Depuis le même répertoire, sur la même machine

- start rmiregistry *un annuaire en 1099*
- start java -cp . ServiceImplRMI *le service*
  - start java -cp . –Djava.security.policy=policy.all ServiceImplRMI *liée au RmiSecurityManager*
- java -cp . ClientRMI *le client*
  - –start java -cp . –Djava.security.policy=policy.all ClientRMI *le service*

```
2 mars 2009 16:39:14 sun.rmi.server.UnicastServerRef logCall
PLUS FIN: RMI TCP Connection(2)-163.173.228.90: [163.173.228.90: ServiceImpl[-3a
6b7afd:11fc7d51785:-7fff, 1300404228195176991]: public abstract boolean Service.o
ffrir(Bouquet) throws java.rmi.RemoteException]
 recevez ce bouquet : [une tulipe, une tulipe]
```

- Avec l'option -Djava.rmi.server.logCalls=true

Le fichier policy.all

```
grant{
    permission java.security.AllPermission;
}
```

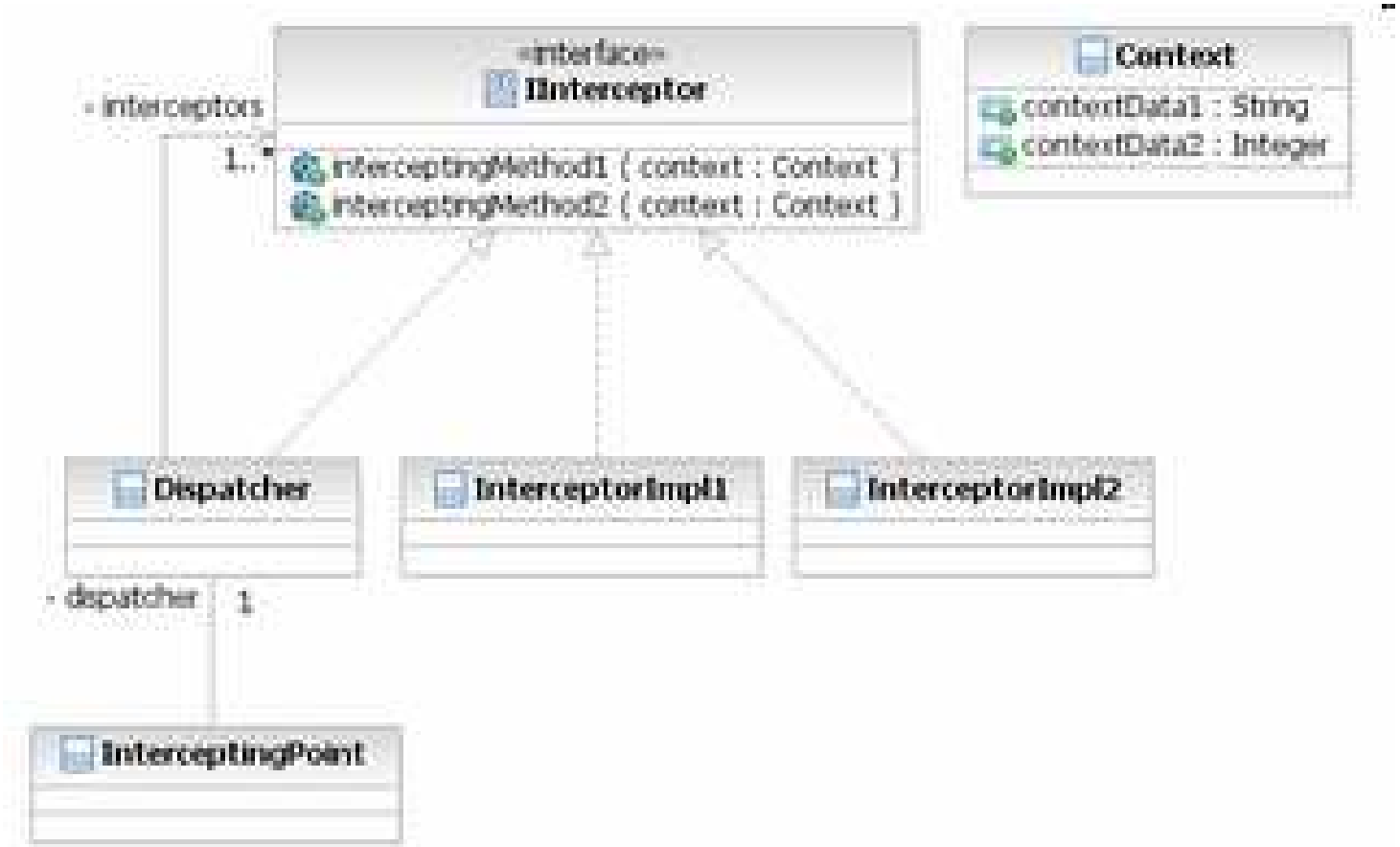
Petites discussions

- **Mandataire, Proxy**
- **Interceptor ?**
  - **En annexe**

Conclusion

- **Patron Procuration**
- **Mandataire**
  - **Introspection**
  - **DynamicProxy**
- **Interceptor**
- **Performances**

Interceptor



http://bosy.dailydev.org/2007/04/interceptor-design-pattern.html
http://struts.apache.org/release/2.0.x/docs/interceptors.html

Annexe : Interceptor : une variante de CoR

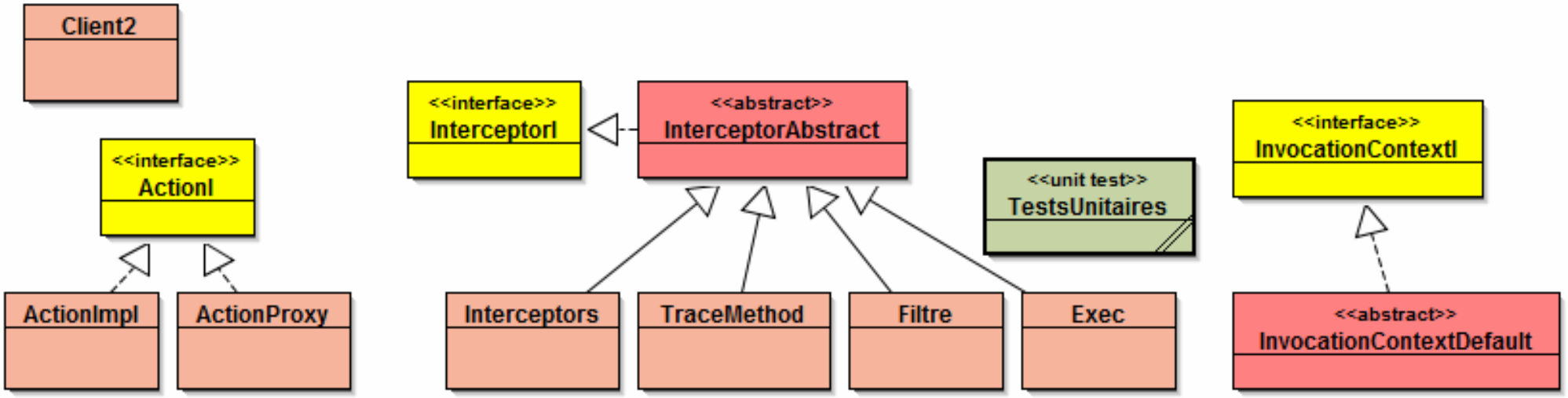
- **Interception d'appels de méthodes**
- **Filtrage de certains contenus, paramètres**
- **Trace, journal, temps d'exécution**
- **Ajout d'instructions avant et après l'exécution d'une méthode**
- **Capture des exceptions**
- **Changement des paramètres à la volée**

- **framework**
  - **@interceptor pour les EJB**
    - **Configuration en XML**
      - http://longbeach.developpez.com/tutoriels/EJB3/Interceptors/
  - http://struts.apache.org/release/2.0.x/docs/interceptors.html

Framework Interceptors

Interceptor	Name	
Alias Interceptor	alias	Converts similar parar
Chaining Interceptor	chain	Makes the previous Ac type="chain"> (in the
Checkbox Interceptor	checkbox	Adds automatic checkb (usually 'false') value. value is overridable fo
Cookie Interceptor	cookie	Inject cookie with a ce

Annexe Interceptor + Proxy




```
4 public interface ActionI{
5   public boolean execute(String str) throws Exception;
6 }
```

- **ActionProxy délègue le contrôle à(aux) *l'interceptor(s)…***
  - **Interception**
    - **de la méthode, des paramètres, des exceptions …**
  - **Ajout d'une trace**

InterceptorI et InvocationContextI

```
3 public interface InterceptorI{
4
5   public Object invoke() throws Exception;
6
7   public InvocationContextI getInvocationContext();
8 }
```

```
2 import java.lang.reflect.Method;
3 public interface InvocationContextI{
4
5    public Method getMethod() throws Exception;
6    public Object[] getParameters();
7    public InvocationContextI setParameters(Object... params);
8    public Object proceed() throws Exception;
9 }
```

- **Chaque interceptor dispose du contexte afin d'effectuer les opérations sur la méthode appelée, les paramètres transmis …**

Interceptor: un exemple de filtre

```
5   public Filtre(InvocationContextI ctxt, String start){
6     super(ctxt);
7     this.start = start;
8   }
9
10  public Object invoke(InvocationContextI ctxt) throws Exception{
11    try{
12      if (ctxt.getMethod().getName().startsWith(start)){
13        System.out.println("*** les paramètres sont modifiés...");
14        ctxt.setParameters(new Object[]{"aurevoir"});
15      }
16    }finally{
17      return ctxt.proceed();
18    }
19  }
```

- **Modification des paramètres à la volée…**

ActionProxy et un contexte

```
4 public class ActionProxy implements ActionI{
5   private ActionI action = new ActionImpl();
6   private InterceptorI interceptor; //
7
8   private InvocationContextI ctxt1 = new InvocationContextDefault(){
9     {super.setParameters("bonjour");}
10    public Method getMethod() throws Exception{
11      return ActionImpl.class.getDeclaredMethod("execute",new Class<?>[]{String.class});
12    }
13    public Object proceed() throws Exception{
14      try{
15        return getMethod().invoke(action,getParameters());
16      }catch(InvocationTargetException e){
17        throw new Exception(e.getCause());
18      }
19    }
20  };
```

- La méthode proceed se contente ici de déclencher la méthode execute issue de l'implémentation, d'autres interceptors peuvent être requis

- Démonstration, voir aussi Programmation orientée aspect (AspectJ, JAC, JBoss AOP

Annexe suite

- **Outil d'assemblage de composants graphiques**
  - **Solutions possibles**
    - **Introspection**
    - **DynamicProxy**

Extrait de : Using Dynamic Proxies to Generate Event Listeners Dynamically

- **Dynamique : Nom des méthodes en paramètre**
- **Un exemple possible :**
  - ActionListener en version statique

```
ActionListener ea = new Action();
addButton.addActionListener(ea);
```

```
class Action implements ActionListener{
  public void actionPerformed( ActionEvent e ){
   // code ……
}
```

  - ActionListener avec un « dynamicproxy »

```
ActionListener ea =
  (ActionListener) GenericProxy.newInstance(
          ActionListener.class,
          "actionPerformed" ,
          this,
          "buttonAction"  );

addButton.addActionListener(ea);
```

http://java.sun.com/products/jfc/tsc/articles/generic-listener2/index.html

Annexes : ProtectionProxy

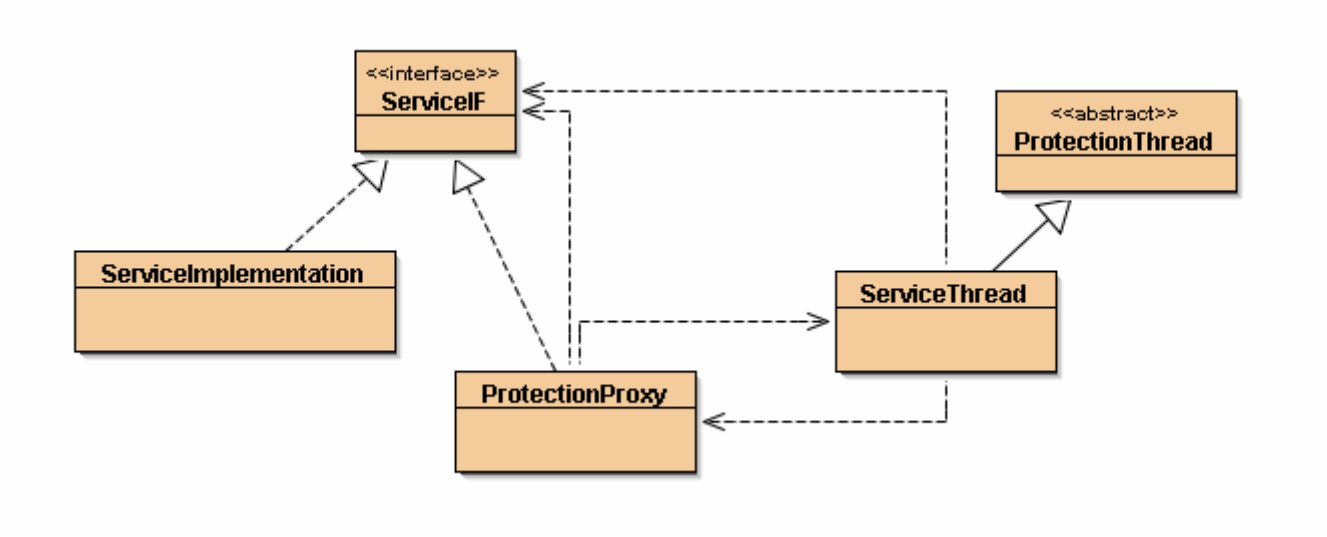
- **Extrait de Mark Grand, java Enterprise Design Patterns Vol 3**

- **Se protéger d 'autres objets usant de l 'introspection à des fins malicieuses...**

- **Un service**

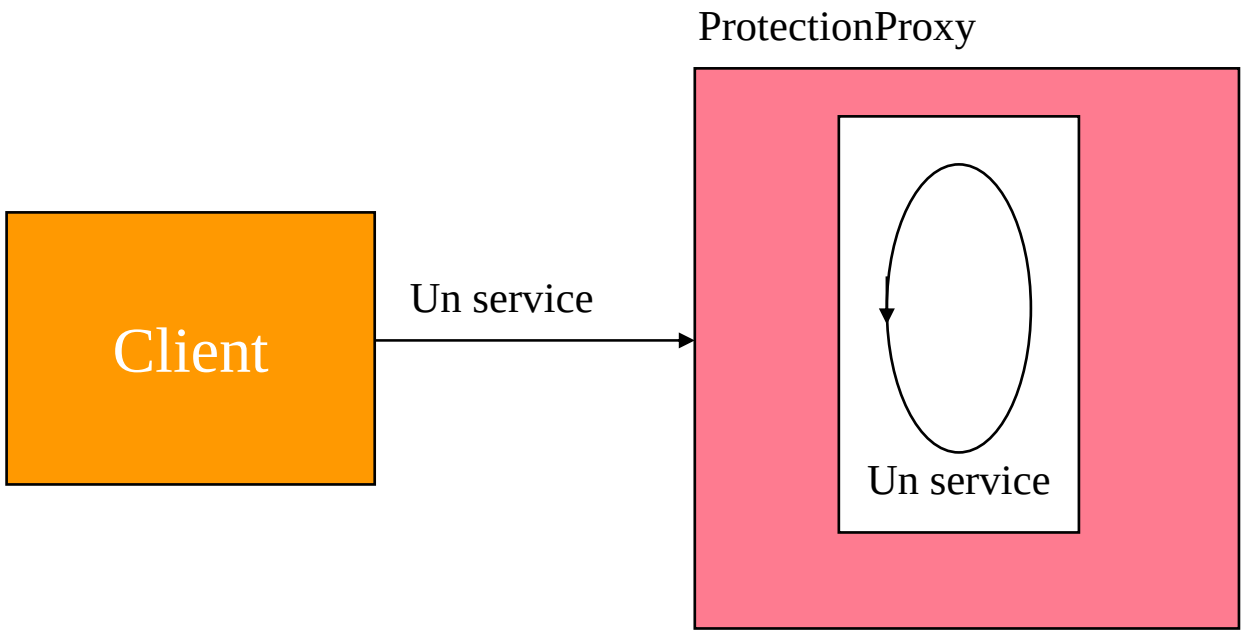
```
public interface ServiceIF{
    public int getInfo();
    public void setInfo(int x);
    public void doIt();
}
```

Les classes et leur rôle



- **ProtectionProxy**
  - **vérifie les permissions et transfère l 'exécution à un service de *Thread***
- **ServiceThread**
  - **exécute la méthode sélectionnée dans un *thread* indépendant, retourne un résultat ou une exception**
- **ProtectionThread**
  - **une classe utilitaire : Thread interne et synchronisation**

Deux contextes distincts



- **Contexte protégé pour l 'exécution du service**
- **impose une période minimale pour l 'appelant**

ProtectionProxy (1)

```
import java.security.Guard;
public class ProtectionProxy implements ServiceIF{
  private final static int GET_INFO = 1, SET_INFO = 2, DO_IT = 3;  // adéquation service / nombre

  private ServiceThread serviceThread;
  private int method, result, arg;
  private Guard guard;

  public ProtectionProxy(ServiceIF Service){
    serviceThread = new ServiceThread(service, this);
    guard = java.security.AllPermission();      // laxiste …
  }

  public int getInfo(){
      guard.checkGuard(this);                    // éventuellement SecurityException
      this.method = GET_INFO;
      serviceThread.usingTrustedThread();   // délégation de l 'exécution en toute confiance
      return result;
     }
    public void setInfo(int x){ }public void doIt(){} // même schéma que getInfo

}
```

ProtectionProxy (2)

```
// le service est enfin assuré, appelé quand tout va bien


  public void continueCall(Service service){
      if(this.method == GET_INFO)
        return service.getInfo();
      else
       ....
     }
}
```

ProtectionThread, initialisation

Mise en œuvre du service, initialisation et contexte d 'exécution

```
public abstract class ProtectionThread extends Thread {
    private static final long MIN_FREQUENCY = 2000;

    private Object service;                        // le service à contrôler

    private boolean pending = false;               // en cours d'exécution
    private boolean done = false;
    private Throwable thrownObject = null;         // pour la propagation de l'exception


    public ProtectionThread(Object myService) {
      synchronized(this){
        service = myService;
        Thread t = new Thread(this);
        t.start();                        // éligibilité du Thread
        try {
            while (service!=null) {
                 wait();  // attente, le Thread local doit démarrer
            } // while
        } catch (InterruptedException e) {
        } // try
      }
    } // constructor(Service)
```

ProtectionThread : Le Thread interne

```
public synchronized void run() {
     Object myService = service;
     service = null;
     notifyAll();
     try {
          while (true) {
               while (!pending) {
                    wait();                    // attente de synchronisation
               } pending = false;
               try {
                    continueCall(service);     // appel du service

               } catch (ThreadDeath e) {
                    throw e;
               } catch (Throwable e) {
                    thrownObject = copyThrowable(e);     // propagation de l'exception
               } finally {
                    done = true;
               } // try
               notifyAll();
               Thread.sleep(MIN_FREQUENCY);   //  délai imposé entre deux demandes

          } // while
     } catch (InterruptedException e) }}
```

–1) // synchonisation

–2) appel de continueCall(service)

public abstract void continueCall(Object myService) ; // voir la sous-classe

ProtectionThread : aie confiance …

```
public synchronized void usingTrustedThread()
                                throws RuntimeException,Error {
        try {
            pending = true;
            notifyAll();   // synchronisation
            while(!done) {
                wait();    // attente du meilleur comme du ...
            } // while
        } catch (InterruptedException e) {
        } finally {
            done = false;
            if (thrownObject != null) {// une exception s'est produite

                Throwable temp = thrownObject;
                thrownObject = null;
                if (temp instanceof RuntimeException)
                  throw (RuntimeException) temp;
                if (temp instanceof Error)
                  throw (Error) temp;
} } }
```

ServiceThread extends ProtectionThread

- **il suffit d 'implémenter la méthode abstraite …**

```
public class ServiceThread extends ProtectionThread {
   private ProtectionProxy proxy;

   public ServiceThread(ServiceIF service, ProtectionProxy proxy) {
      super(service);
      this.proxy = proxy;
   }


   public void continueCall(Object myService) {  // la  méthode abstraite de ProtectionThread
       proxy.continueCall((ServiceIF)myService);

   }
 }
```

Un client bienveillant

```
ServiceIF serviceDirect = new
 ServiceImplementation();
int resultatEnDirect =
 serviceDirect.getInfo();
```

```
ServiceIF serviceProtégé = new ProtectionProxy(new
 ServiceImplementation());
int resultatEnModeProtégé = serviceProtégé.getInfo();
```