les patrons Proxy

jean-michel Douin, douin au cnam point fr
version : 9 Octobre 2009

Le Patron Procuration/ Pattern Proxy

Sommaire

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**

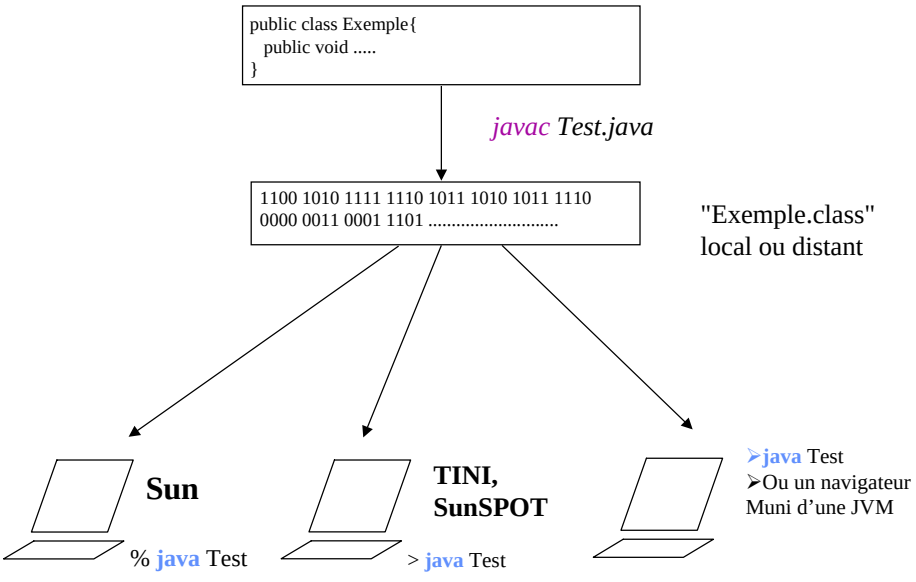
Sommaire pour les Patrons

- **Classification habituelle**
  - **Créateurs**
    - **Abstract Factory, Builder,** Factory Method **Prototype** Singleton
  - **Structurels**
    - **Adapter Bridge** Composite Decorator **Facade Flyweight** Proxy
  - **Comportementaux**
    Chain of Responsability. Command Interpreter Iterator
    Mediator Memento Observer State
    Strategy Template Method Visitor

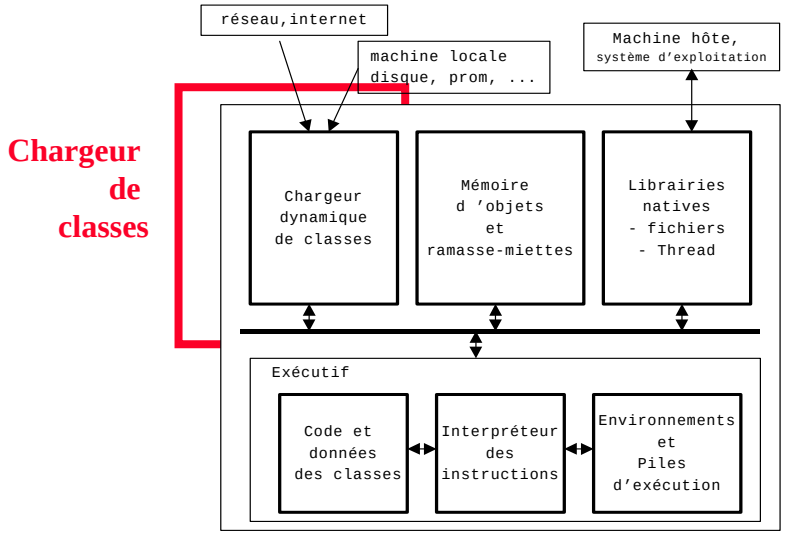
Présentation rapide

- **JVM, ClassLoader & Introspection**
  - **Rapide ?**
    - **Juste ce qu'il faut pour lire les exemples présentés**

La JVM



JVM : architecture simplifiée



- **Java Virtual Machine**
  - **Chargeur de classes et l'exécutif**
  - **Extrait de http://www.techniques-ingenieur.fr/dossier/machine_virtuelle_java/H1588**

Chargeur de classe

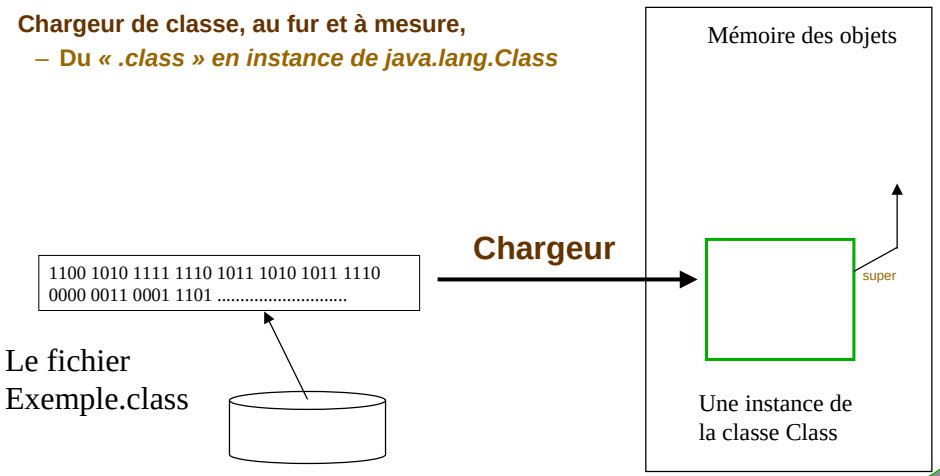
- **Chargement dynamique des *.class***
  - **Au fur et à mesure, en fonction des besoins**
    - **Chargement paresseux, tardif, *lazy***
- **Le chargeur**
  - **Engendre des instances de *java.lang.Class***
  - **Maintient l'arbre d'héritage**
- **Les instances de la classe *java.lang.Class***
  - **« Sont des instances comme les autres »**
  - **Gérées par le ramasse-miettes**

Sommaire : Classes et *java.lang.Class*

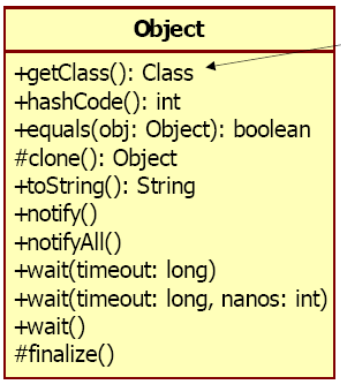
- **Le fichier *.class***
  - Une table des symboles et bytecode
  - Une décompilation est toujours possible …
    - Du .class en .java …
    - Il existe des « obfuscateurs »
- **Le chargeur de *.class***
  - Les chargeurs de classes → de .class en classe Class

Sommaire suite

- A chaque classe un fichier .class,
  - Classes standard, classes anonymes, classes internes, …
- Chargeur de classe, au fur et à mesure,
  - Du *« .class » en instance de java.lang.Class*



Obtention de l'instance Class : getClass()



A l'exécution il est possible de demander à tout objet quelle est sa classe

Exemple

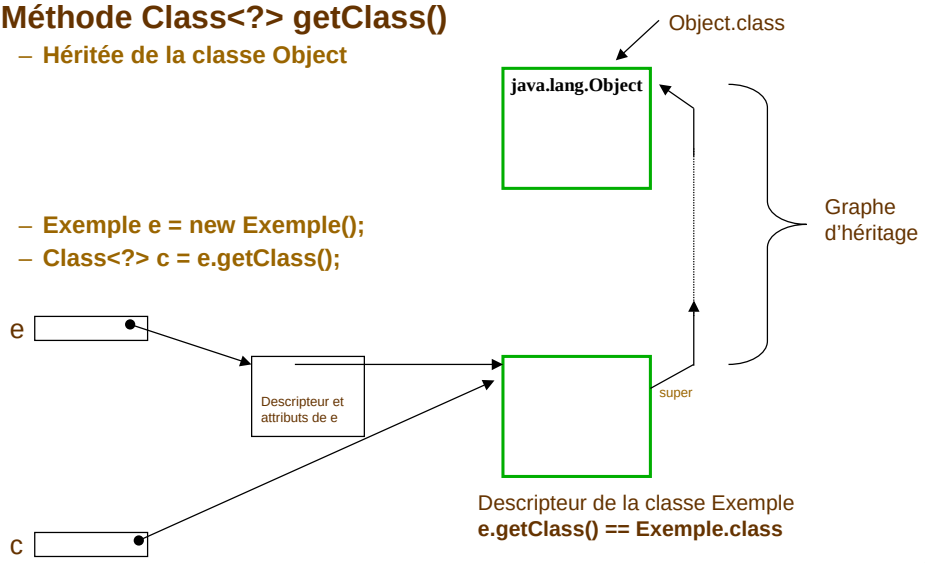
v.getClass()==C1.class

String.class.getClass() == Class.class

- **Extrait de http://jfod.cnam.fr/NFP121/07-introspection-dynamique-JavaBeans/NFP121_07_007_1.pdf**

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") ;**

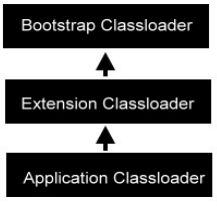
- **Il existe un chargeur de classes par défaut**

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
  - Écrire son propre *ClassLoader*
    - *Possible mais dans un autre cours*



ClassLoader prédéfinis

- **SecureClassLoader**
  - la racine

java.net
Class URLClassLoader

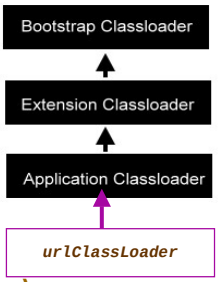
java.lang.Object
  └ java.lang.ClassLoader
    └ java.security.SecureClassLoader
      └ java.net.URLClassLoader

Direct Known Subclasses:
MLet

- **URLClassLoader**
  - Utilisé depuis votre navigateur

URLClassLoader : un exemple

- **Chargement distant de fichier** *.class* **et exécution**
  - *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 ou non*
4. *Recherche de la méthode main par introspection*

URLClassLoader : un exemple

```
public class Exemple1{

  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);


     // 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);


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

 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 Exemple1 UneClasse param1 param2 param3

UneClasse n'est connue qu'à l'exécution

Présentation rapide

- **Terminée !**

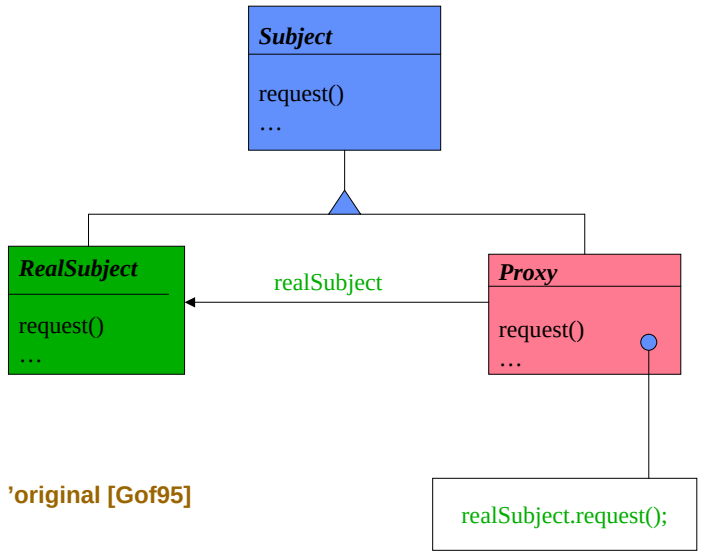
- **C'était juste pour lire les sources associés aux**

  patrons Proxy et DynamicProxy

Objectifs

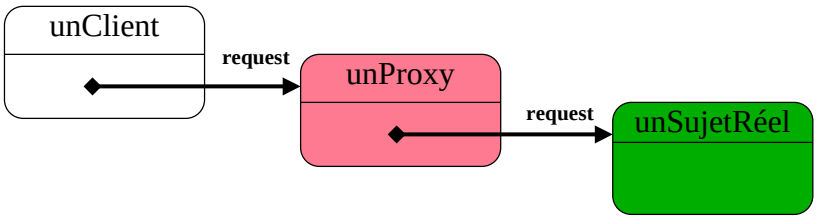
- **Proxy comme Procuration**
  - **Fournit à un tiers objet un mandataire, pour contrôler l 'accès à cet objet**
- **Alias**
  - **Subrogé (surrogate)**
- **Motivation**
  - **contrôler**
  - **différer**
  - **optimiser**
  - **sécuriser**
  - **accès distant**
  - **...**

Diagramme UML



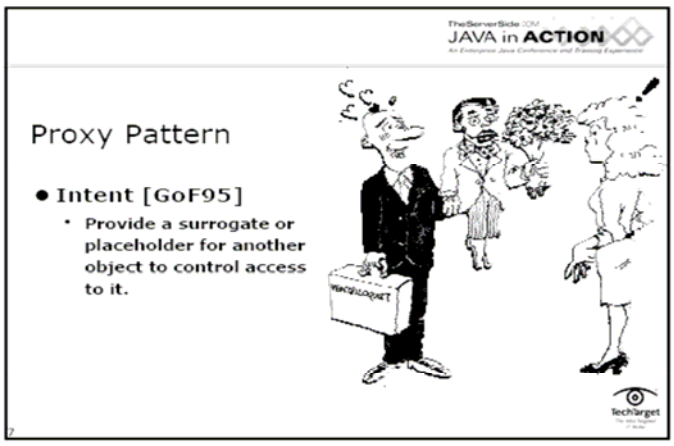
– L 'original [Gof95]

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

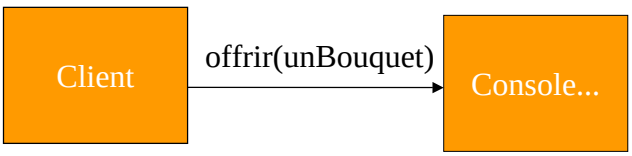
Service
offrir(Bouquet b) …

```
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


```
public class ServiceImpl implements Service{

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

}
```

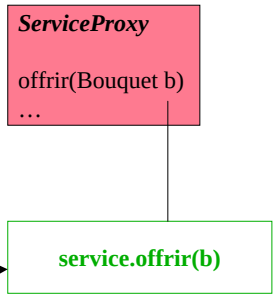

Offrir en « direct »



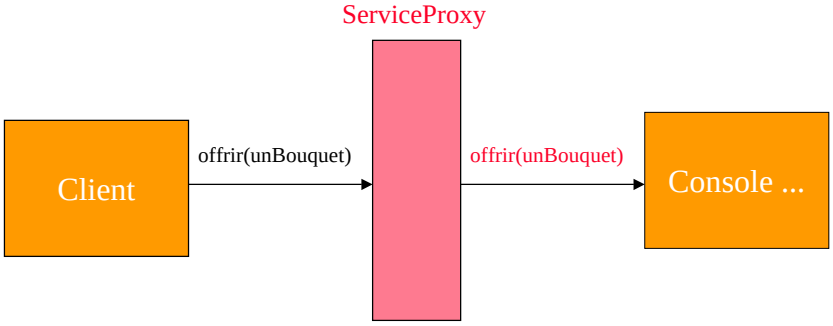

```
Service service = new ServiceImpl();

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;
   }    }
```

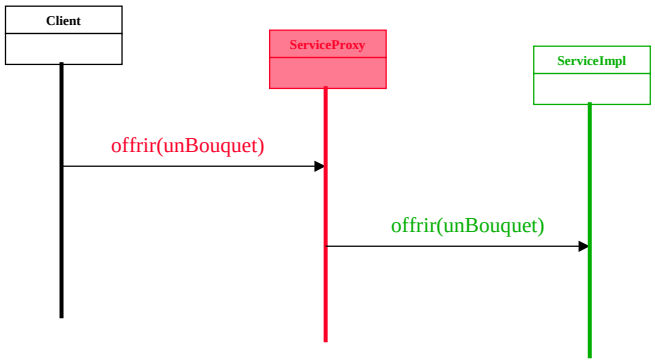


Offrir par l 'intermédiaire de




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

Séquence …



- **Sans commentaire**

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) : ");
     Service service = getServiceImpl();
     résultat = service.offrir(bouquet);

     return résultat;
   }
```

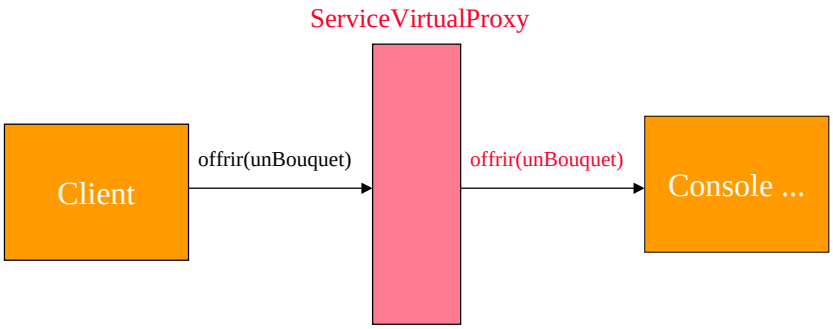

VirtualProxy, suite & Introspection

```
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)cons.newInstance(paramètres);      // exécution
        // ou service = (Service) classe.newInstance(); de la classe Class
       }catch(Exception e){
       }
      }
      return service;
 }
```

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

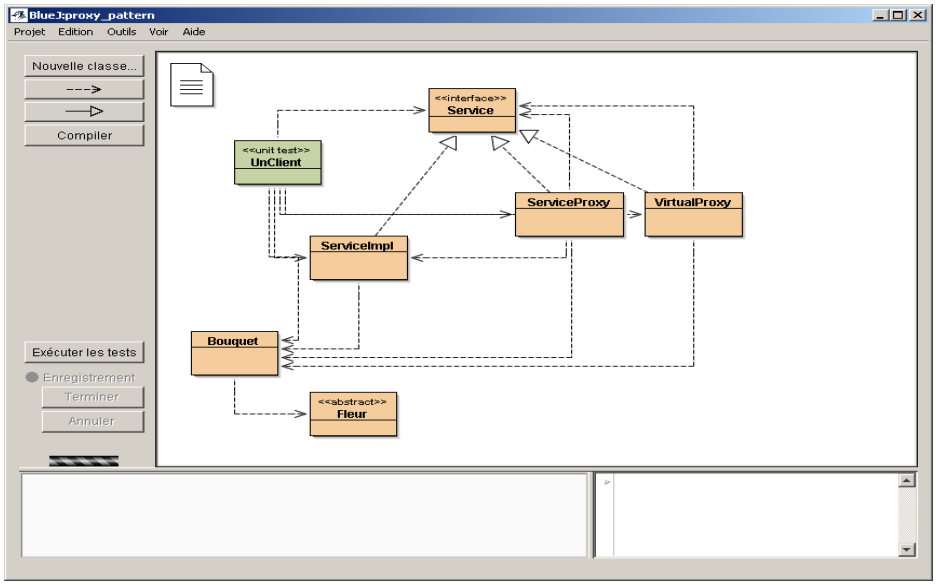
```
private Service getServiceImpl(){
      if(service==null)
         service = new ServiceImpl();
      return service;
 }
```

Offrir par l 'intermédiaire de



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

La « famille proxy» avec BlueJ



La famille s'agrandit

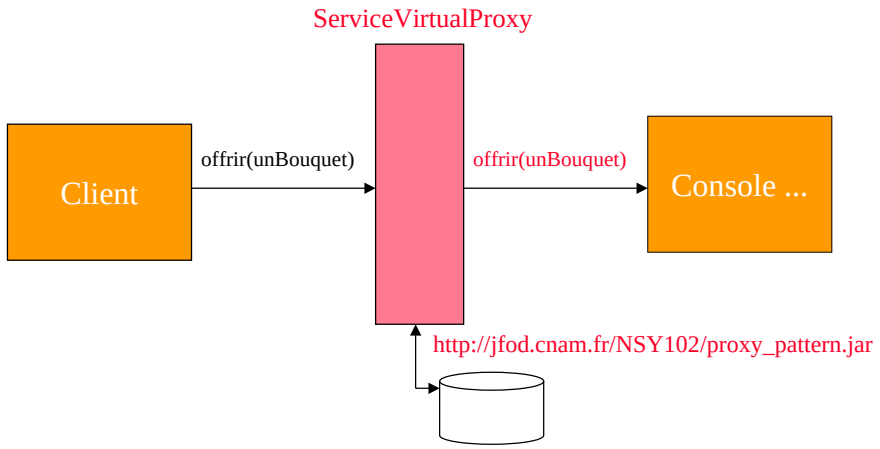
- **SecureProxy**
  - sécuriser les accès
- **Introspection + Proxy = DynamicProxy**
  - création dynamique de mandataires
- **RemoteProxy …**
  - accès distant

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

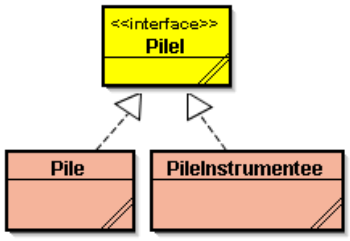



```
Service service = new ServiceVirtualProxy();
boolean résultat = service.offrir(unBouquet);
```


Un exemple moins fleuri : la classe Pile !

- **Prémisses**
  - Une pile (dernier entré, premier sorti)
  - En java une interface : PileI
  - Une ou plusieurs implémentations
- **Comment tester le bon comportement ?**
  - JUnit tests associés à chaque implémentation, ok
  - Ou bien
  - Un mandataire instrumenté par des assertions

Une pile instrumentée



- **PileI p = new PileInstrumentee(10);**
- **exécuterUneSéquence(p);**

```
public static void exécuterUneSéquence(PileI p)  throws Exception{
   p.empiler("b");
   p.empiler("a");
   System.out.println(" la pile : " + p);
  …
  }
```


PileInstrumentée : un extrait

```
public class PileInstrumentee implements PileI{
  private PileI p;


/** @Pre taille > 0;
 *  @Post p.capacite() == taille;
 */
public PileInstrumentee(int taille){
  assert taille > 0 : "échec : taille <= 0";
  this.p = new Pile(taille);
  assert p.capacite() == taille : "échec : p.capacite() != taille";
}


@Pre et @Post pourraient être dans l'interface PileI …
PileInstrumentee pourrait être générée automatiquement …
```

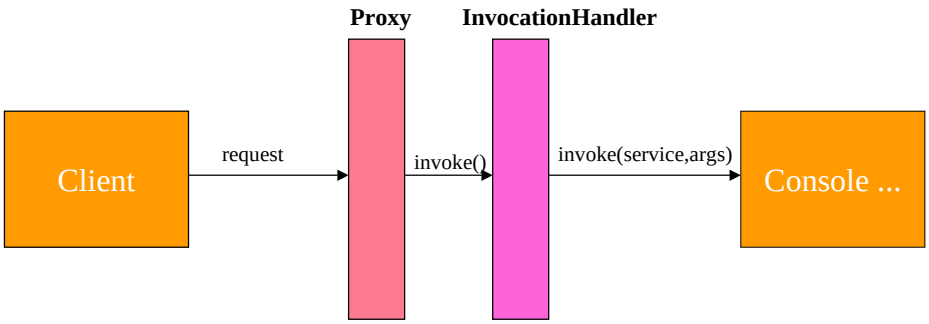
Résumé, bilan intermédiaire

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

Un cousin plutôt dynamique et standard

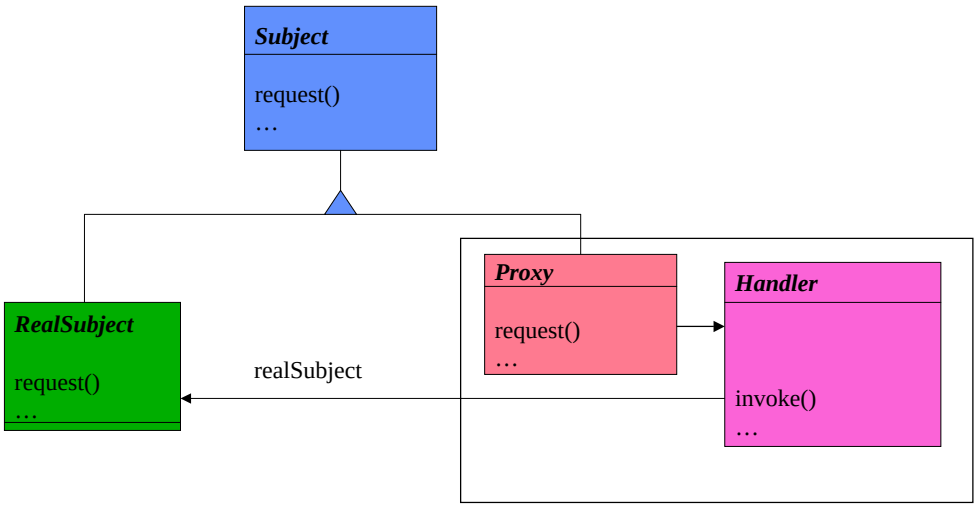
- **extrait de l 'API du J2SE Dynamic Proxy** *(depuis 1.3)*
  - **http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/InvocationHandler.html**
  - **http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/Proxy.html**
- **Génération de byte code à la volée**
  - **Rare exemple en java**
- **Paquetages**
- **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



Handler implements InvocationHandler

interface java.lang.reflect.InvocationHandler;

- **ne contient qu 'une seule méthode**

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

- **proxy : le proxy généré**
- **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
  }

}
```

Il nous manque le mandataire qui se charge de l'appel de invoke
ce mandataire est 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
….

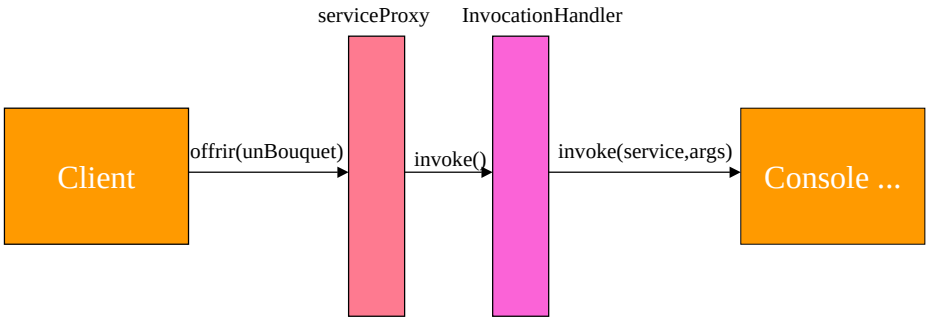
crée dynamiquement un mandataire
spécifié par le chargeur de classe *loader*
lequel implémente les interfaces *interfaces,*
la méthode h.invoke sera appelée par l'instance du Proxy retournée (le constructeur de h est alors déclenché)

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);
```

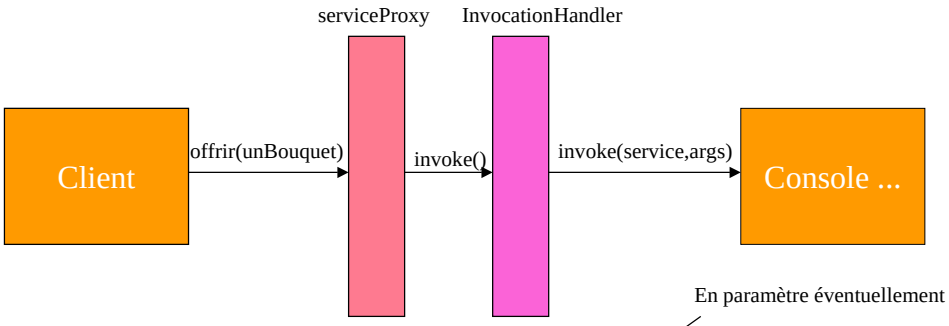
Un dessin




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

boolean résultat = proxy.offrir(unBouquet);

VirtualSecureProxy le retour




```
ClassLoader cl =
    new URLClassLoader(
            new URL[]{new URL("http://jfod.cnam.fr/csiml/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 …**
- **Mieux : généré par des outils …**

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 ? Un adaptateur ? Une délégation ?*

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.getName() +
                          " durée : "  + (top - System.currentTimeMillis()));
    }
}
```


Un autre usage de DebugProxy

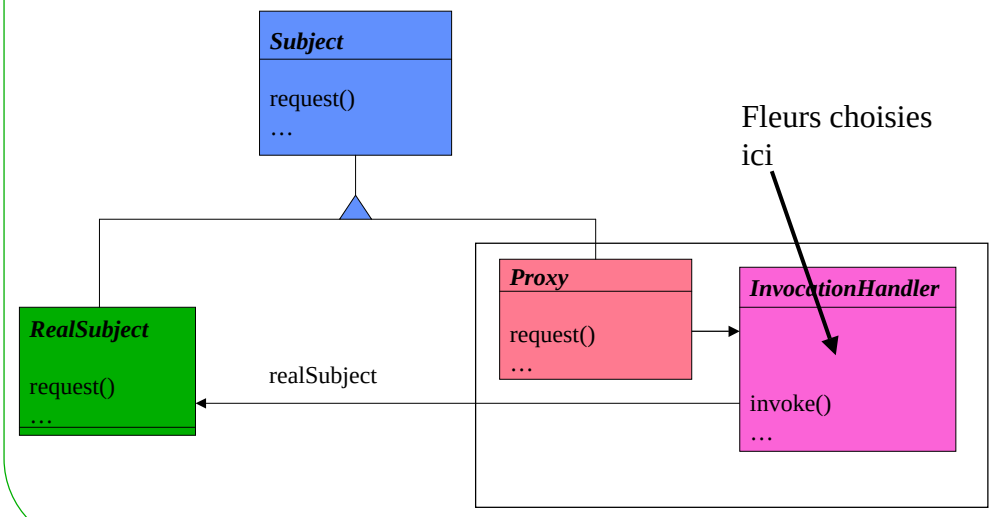
Service service = (Service) DebugProxy.newInstance(new ServiceImpl());
résultat = service.offrir(unBouquet);

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…**



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);
```


Le patron Fabrique peut être utile

```
public class Fabrique{

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

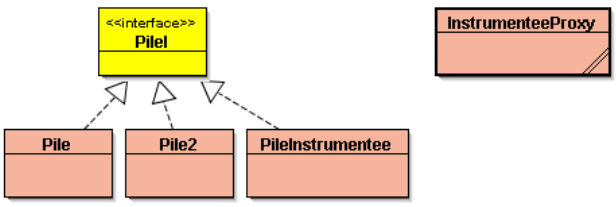
La pile : le retour, *(une valeur sûre)*

- **Hypothèses**
  - **La classe PileInstrumentee existe ,** *// déjà vue …*
  - **Cette classe implémente l'interface PileI,** *// c.f le patron proxy*
  - **Cette classe possède ce constructeur**
    - **PileInstrumentee(PileI p){ this.p = p; }**

- **Un « dynamic proxy d'instrumentation»**
  - **Crée une instance instrumentée**
  - **Intercepte tous les appels de méthodes**
  - **Déclenche si elle existe la méthode instrumentée, sinon l'orignal est appelée**
  - **Capture l'assertion en échec, et affiche la ligne du source incriminé**
  - **Propage toute exception**

- **Comment ? → DynamicProxy + introspection**

Exemple d'initialisation




```
p = (PileI)InstrumenteeProxy.newInstance(new Pile(10));
exécuterUneSéquence(p);
```

```
public static void exécuterUneSéquence(PileI p)  throws Exception{
   p.empiler("b");
   p.empiler("a");
   System.out.println(" la pile : " + p);
  …
 }
```

InstrumenteeProxy : le constructeur

```
public class InstrumenteeProxy implements InvocationHandler{
  private Object   cible;
  private Object   cibleInstrumentee;
  private Class<?> classeInstrumentee;

  public InstrumenteeProxy(Object target) throws Exception{
    this.cible = target;
// à la recherche de la classe instrumentée
    this.classeInstrumentee = Class.forName(target.getClass().getName()+"Instrumentee");
// à la recherche du bon constructeur
    Constructor cons = null;
    for(Class<?> c : target.getClass().getInterfaces()){
      try{
        cons = classeInstrumentee.getConstructor(new Class<?>[]{c});
      }catch(Exception e){}
    }
// création de la cible instrumentée
    cibleInstrumentee = cons.newInstance(target);
}
```


InstrumenteeProxy : la méthode invoke

```
public Object invoke(Object proxy, Method method, Object[] args)throws Exception{
  Method m = null;
   try{// recherche de la méthode instrumentée, avec la même signature
    m = classeInstrumentee.getDeclaredMethod(method.getName(), method.getParameterTypes());
    }catch(NoSuchMethodException e1){
      try{ // la méthode instrumentée n'existe pas, appel de l'original
        return method.invoke(cible, args);
      }catch(InvocationTargetException e2){        // comme d'habitude …
         throw e2.getTargetException();
      }}

   try{ // invoquer la méthode instrumentée
      return m.invoke(cibleInstrumentee, args);
    }catch(InvocationTargetException e){
      if(e.getTargetException() instanceof AssertionError){
        // c'est une assertion en échec
        if(e.getTargetException().getMessage() != null) // le message
          System.err.println(e.getTargetException().getMessage());
        System.err.println(e.getTargetException().getStackTrace()[0]);
      }
        throw e.getTargetException(); // propagation …
    }}
```

InstrumenteeProxy, suite et fin

```
public static Object newInstance(Object obj) throws Exception{
    return Proxy.newProxyInstance(obj.getClass().getClassLoader(),
                                  obj.getClass().getInterfaces(),
                                  new InstrumenteeProxy(obj));
  }
// déjà vu
```

- Assez générique ? Souple d'utilisation ?

Démonstration

- ?

Conclusion

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

Annexe

- **Le petit dernier proxy_rmi**
- **Appels de méthodes distantes**

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) client comme serveur**

« DynamicProxy » à la rescousse

- **Une instance du mandataire « qui s'occupe de tout » est téléchargée par le client**

Quelques « légères » modifications

- **Voir support RMI**
  - **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("vivaldi.cnam.fr");

   // réception du mandataire, en interrogeant l'annuaire
   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());
      ServiceImpl serveurRMI = new ServiceImpl();

      Service stub = (Service)UnicastRemoteObject.exportObject(serveurRMI, 0);
      Registry registry = LocateRegistry.getRegistry();  // l'annuaire
      registry.rebind("service_de_fleurs", stub);          // enregistrement auprès de l'annuaire

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

UnicastRemoteObject.exportObject : création du « dynamicProxy » le stub