Chapitre 8

Le DynamicProxy

L'objectif de ce chapitre est de montrer le design pattern DynamicProxy. De profiter pour faire un rappel sur le chargement dynamique des classes en Java et sur la réflexivité du langage Java.

1. LE DYNAMICPROXY 2

1.1. NOM ET ROLE 2
1.2. DESCRIPTION DE LA SOLUTION 3

2. LA MISE EN ŒUVRE EN JAVA 3

3. EXEMPLE SIMPLE 4

4. LE CHARGEMENT DYNAMIQUE DE CLASSE 4

4.1. JVM, CLASSLOADER & INTROSPECTION 5
4.2. EXEMPLE EXEMPLECH08_02_CLASSLOADER 6

5. SERVICEVIRTUALPROXY, DYNAMICPROXY ET CLASSLOADER 8

5.1. L'OBJECTIF 8
5.2. L'EXEMPLE EXEMPLECH08_03 8
5.2.1. LE PROXY 8
5.2.2. LE SERVICEVIRTUALPROXY 9
5.2.3. LE DYNAMICPROXY 10

6. APPLICATION A RMI 12

6.1. LE PROXY DE COMMUNICATION 12

7. APPLICATION DU DYNAMICPROXY A L'EXEMPLE CH08_04 : LE PACKAGE MYRMI 14

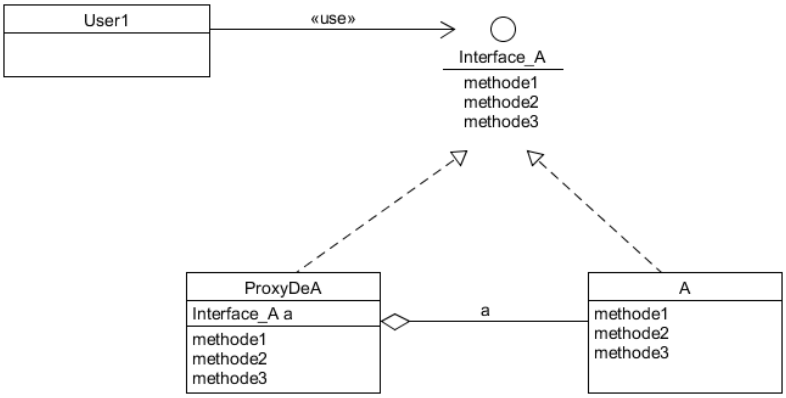
7.1. PRESENTATION 14
7.2. LES DIAGRAMMES DE CLASSE DU PACKAGE MYRMI 14
7.3. COMMENTAIRE DU CODE 16
7.4. PROGRAMME DE TEST DU PACKAGE MYRMIL 19

1. Le DynamicProxy

1.1. Nom et rôle

Le **DynamicProxy** est un Design Pattern.

Le DynamicProxy est un Proxy. Rappel du DP Proxy :



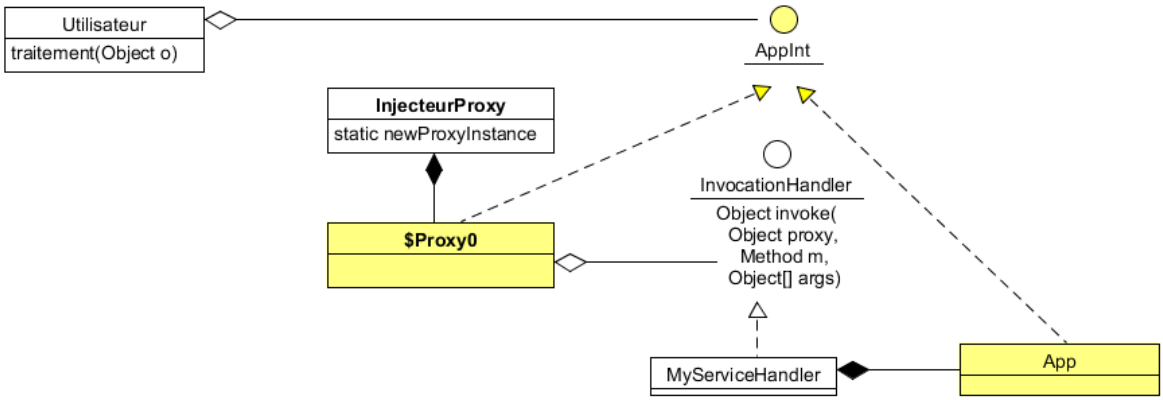
Quand on regarde le code de beaucoup de Proxy, on s'aperçoit qu'il y a :

- en fonction du rôle du proxy, une plus ou moins grande répétitivité dans le codage de chaque méthode de ProxyDeA
- chacune de ces méthodes, à terme, fait l'appel du même nom de méthode à la classe A
- par principe d'un Proxy, n'importe quelle classe qui implémente l'interface peut être utilisée à travers le proxy

Or quand l'interface A évolue (et la classe A avec) alors il faut impacter la classe ProxyDeA => **C'est ce que l'on veut éviter**.

Le rôle du DynamicProxy est de créer **un proxy qui s'adapte automatiquement à l'interface** (et même à plusieurs interfaces).

1.2. Description de la solution



La solution consiste à demander à un "injecteur de proxy" (Ex: Java) de créer au moins automatiquement sinon dynamiquement une classe de proxy ($Proxy0) qui implémente toutes les méthodes de l'interface AppInt.

La classe $Proxy0 implémente chacune des méthodes de la manière suivante :

- appel de la méthode **invoke**.

A la charge du développeur de réaliser le code de la méthode invoke.
Pour cela, il doit créer une classe, (Ex: MyServiceHandler), qui implémente l'interface InvocationHandler (méthode **invoke**).

Toute la difficulté est de créer la classe InjecteurProxy (et la classe Method).
En fonction des langages cela est plus ou moins difficile.

En JAVA, grâce à la réflexivité (introspection) du langage, cela est directement géré par le langage Java.

Et c'est d'autant plus facile, en JAVA, car la méthode **newProxyInstance** existe nativement.

2. La mise en œuvre en JAVA

API Java :

- java.lang.reflect
  - *InvocationHandler*
  - **Proxy**

```
public static Object newProxyInstance(
                      ClassLoader loader,
                      Class<?>[] interfaces,
                      InvocationHandler h)
                              throws IllegalArgumentException
```

Cette méthode est une méthode static de la classe Proxy.

Cette méthode prend en paramètre :

- loader : un chargeur de classe. Il est utilisé par le proxy pour, éventuellement, charger dynamiquement les classes (*) utilisées dans le InvocationHandler
- **les** interfaces que le proxy doit implémenter
- le InvocationHandler qui est un objet qui implémente l'interface InvocationHandler dont la classe **code le rôle de votre proxy**.

Cette méthode crée un objet, dynamiquement grâce à la réflexivité du langage.

Cet objet est une instance d'une classe dynamique $Proxy0 qui implémente toutes les méthodes de toutes les interfaces. L'implémentation de chacune de ces méthodes est l'appel à la méthode **invoke** :

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

Cette méthode prend en paramètre :

- le proxy qui a été créé,
- la méthode concernée,
- les paramètres passés en paramètre par l'appelant de la méthode concernée.
  .

A la charge du programmeur de réaliser le traitement de cette méthode.

(*) Nous verrons cela plus loin car cela a son importance en Java.

3. Exemple simple

Cet exemple est dérivé de l'exemple Ch04_05_DPProxy que nous avons vu dans le cours sur les Designs Patterns mais en utilisant un proxy dynamique.

Voir sur le site http://jacques.laforgue.free.fr l'exemple
ExempleCh08_01_DPProxyDynamic

Comparer le code de cet exemple avec l'exemple ExempleCh04_05_DPProxy des DP.

4. Le chargement dynamique de classe

La création d'un proxy dynamique prend en entrée un chargeur de classe dynamique.

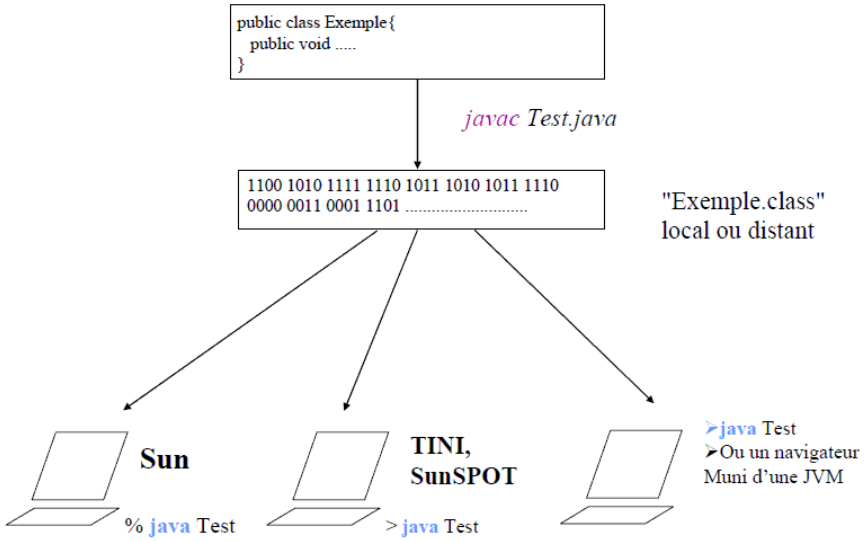
Cela est utilisé pour pouvoir créer dynamiquement, dans le invoker handler, les classes (dont la classe sur laquelle on fait le proxy) et pouvoir indiquer la localisation de ces classes.

Avant d'aller plus loin faisons un rappel.

4.1. JVM, ClassLoader & Introspection

Faisons un rappel sur les notions de **chargement de classe dynamique** et d'introspection en JAVA car ces notions nous sont utiles pour expliquer le fonctionnement du design pattern **DynamicProxy**, base de la communication RMI depuis la version 1.5 de JAVA que nous verrons plus loin, et très utilisé dans les Frameworks pour faire de l'injection de dépendance.

Le chargeur dynamique de classe permet de télécharger à distance du P-CODE de classe Java afin qu'il puisse s'exécuter dans une autre JVM.



Le « ClassLoader » est le mécanisme au sein de la JVM qui permet de charger les fichiers contenant la définition de typage (les classes) et le code (les méthodes).

Le chargeur engendre des instances de **java.lang.Class** : ainsi il convertit un fichier .class(chargé en mémoire : bytes) en une classe Class.

La méthode **getClass()** retourne une classe de type **Class**.

Il existe différents chargeurs de classe :

- un chargeur qui charge la classe à travers le **CLASSPATH** (extension de la JVM : nécessaire si on veut éviter, une recompilation et réexécution du programme)
- un chargeur qui lit des **répertoires** contenant des fichiers .class
- un chargeur qui charge les classes se trouvant dans le WWW d'un **serveur http**.

Le ClassLoader prédéfini de base, la méthode prédéfinie **forName** qui charge les classes à travers le classpath (mais pas que) :

```
String unNomdeClasse XXXX;
```

```
// avec le chargeur de classes par défaut
Class<?> classe = Class.forName(String unNomDeClasse)
```

Un ClassLoader spécifique :

```
// avec un chargeur de classes spécifique
Class<?> classe = Class.forName(unNomDeClasse, unBooléen,
                                                unChargeurDeClasse)
```

ou

```
Class<?> classe = unChargeurDeClasse.loadClass ( unNomDeClasse )
```

Ainsi, tout chargeur de classe retourne une **Class**.

Ensuite, l'introspection permet de :

- créer une instance de cette classe (**newInstance**) à partir d'une **Class**

Puis,

Si la classe n'implémente pas d' interface :

- faire de l'introspection sur l'instance créée qui est un **Object**, afin d'obtenir les méthodes, les constructeurs, les attributs, … (en nom et en nombre)
- exécuter les méthodes de la classe

Si la classe implémente une interface

- appeler les méthodes à travers l'interface

Dans ce cas, l'interface doit être dans le CLASSPATH (pour passer à la compilation).

```
Exemple :
        Class classeToto = Class.forName("Toto");
                          // qui contient une métode truc(…)

         Object obj = classeToto.newInstance(…..);
          // IMPOSSIBLE  :   new Toto()

          // IMPOSSIBLE :  Toto obj = classeToto.newInstance(…..);

        Method m = classeToto.getMethod("truc", ……..);
       m.invoke( obj, args);

      ////
     TotoInterface  obj = classeToto.newInstance(…..);
     obj.truc(…….);
```

4.2. Exemple ExempleCh08_02_ClassLoader

Cet exemple aborde les cas de :

- chargement dynamique de classes qui sont dans un répertoire et accessible par l'OS (via NFS par exemple)
- chargement dynamique de classes via une communication socket qui est en communication avec un serveur de socket
- chargement dynamique de classes qui sont sur un serveur http.
- chargement dynamique de classes à travers le protocole RMI

Voir **ExempleCh08_02_ClassLoader**

Remarque : RMI utilise la technique du serveur de socket pour échanger les classes nécessaires à sa mise en œuvre entre le client et l'Objet Distant.

5. ServiceVirtualProxy, DynamicProxy et ClassLoader

5.1. L'objectif

Une classe, appelée "applicative", est décrite par une interface.

La classe "applicative" n'est pas connue de la JVM.

Une autre classe, appelée "utilisateur", utilise cette classe à travers son interface.

L'objectif est que l'utilisateur utilise la classe applicative à travers un proxy.

Nous allons explorer 2 cas de figure :

- le proxy n'est pas dynamique et est donc spécifique à l'interface applicative. La classe n'est pas connue de la JVM. C'est le cas du **ServiceVirtualProxy**.
- le proxy s'adapte à n'importe quelle interface applicative. Le proxy est créé dynamiquement par la JVM. C'est le cas du **DynamicProxy**.

  La classe n'est pas connue de la JVM.

5.2. L'exemple ExempleCh08_03

Pour illustrer, ces 2 cas de proxy, l'exemple ExempleCh08_03 (cas a, b et c)

Le cas a est l'exemple de départ qui correspond à un simple

- **Proxy**.

Ensuite les cas b et c correspondent à

- **ServiceVirtualProxy** et
- **DynamicProxy**.

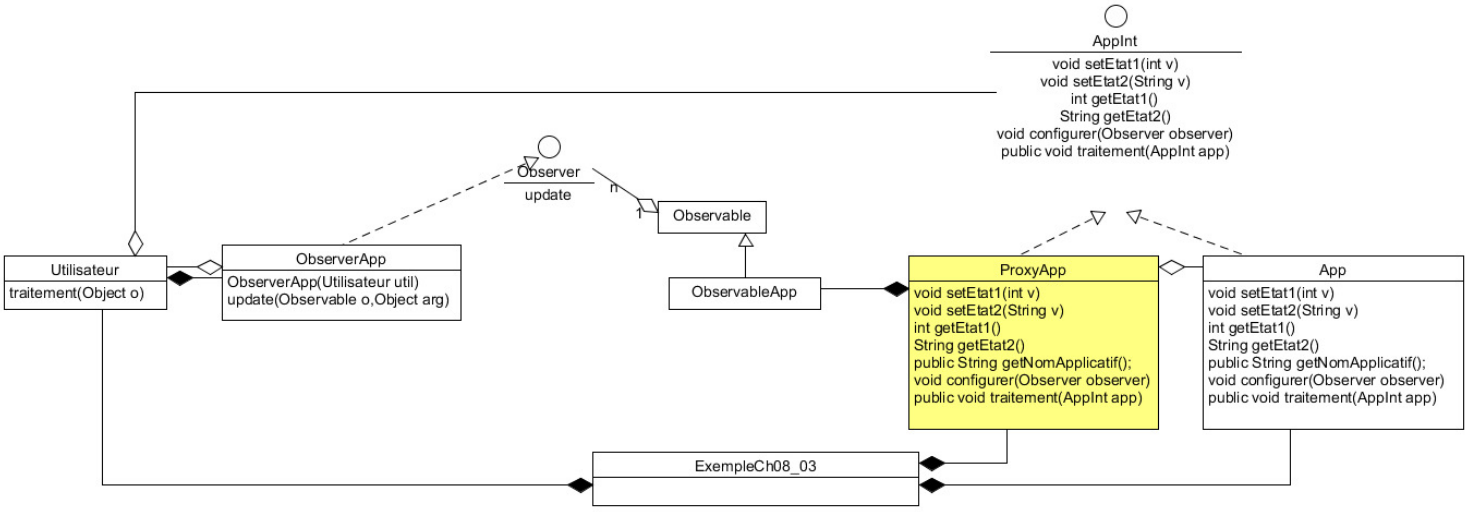
5.2.1. Le Proxy

Notre démonstration part de cet exemple que nous avons vu dans le cours des Designs Patterns (avec quelques améliorations) :
Exemple ExempleCh04_10c_DPObserverProxy.

Rappel du rôle ce proxy :

- créer un observable ;
- à chaque fois que le traitement utilise un **setteur** du proxy, il appelle le setteur de l'applicatif **et notifie** le paramètre du setteur aux observer de l'observable;
- permet à une classe utilisatrice de s'abonner à l'observable afin de recevoir les notifications.

Nous rappelons que nous avons le diagramme de classe suivant :



Voir sur le site http://jacques.laforgue.free.fr cours NSY102
l'exemple **ExempleCh08_03a_SimpleProxy**

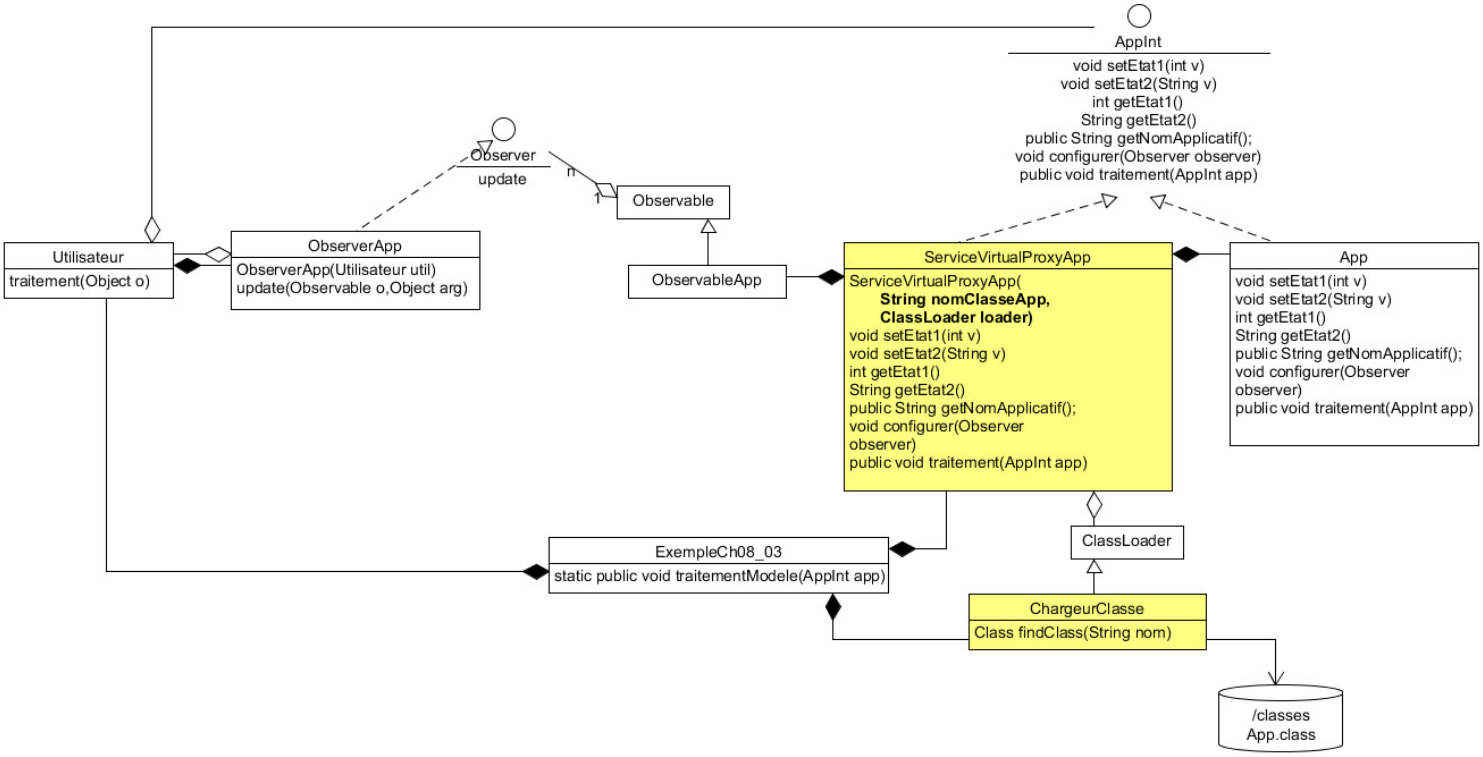
Cet exemple montre les points suivants :

- des utilisateurs utilisent un applicatif à travers une interface
- l'implémentation de l'interface est un PROXY qui réalise les notifications
- l'observable (mécanisme de notification) est encapsulé par le proxy (amélioration par rapport à l'exemple du cours sur les DP)
- ~~le traitement est une méthode "auto-référencé" de l'applicatif (amélioration par rapport à l'exemple du cours sur les DP)~~
- des utilisateurs utilisent l'applicatif sans passer par un proxy

5.2.2. Le ServiceVirtualProxy

Dans le cas du ServiceVirtualProxy, la classe applicative n'est pas connue du classpath de la JVM.

Le Service Virtual Proxy prend en entrée un chargeur de classe qui gère la localisation et l'accès à la classe applicative.



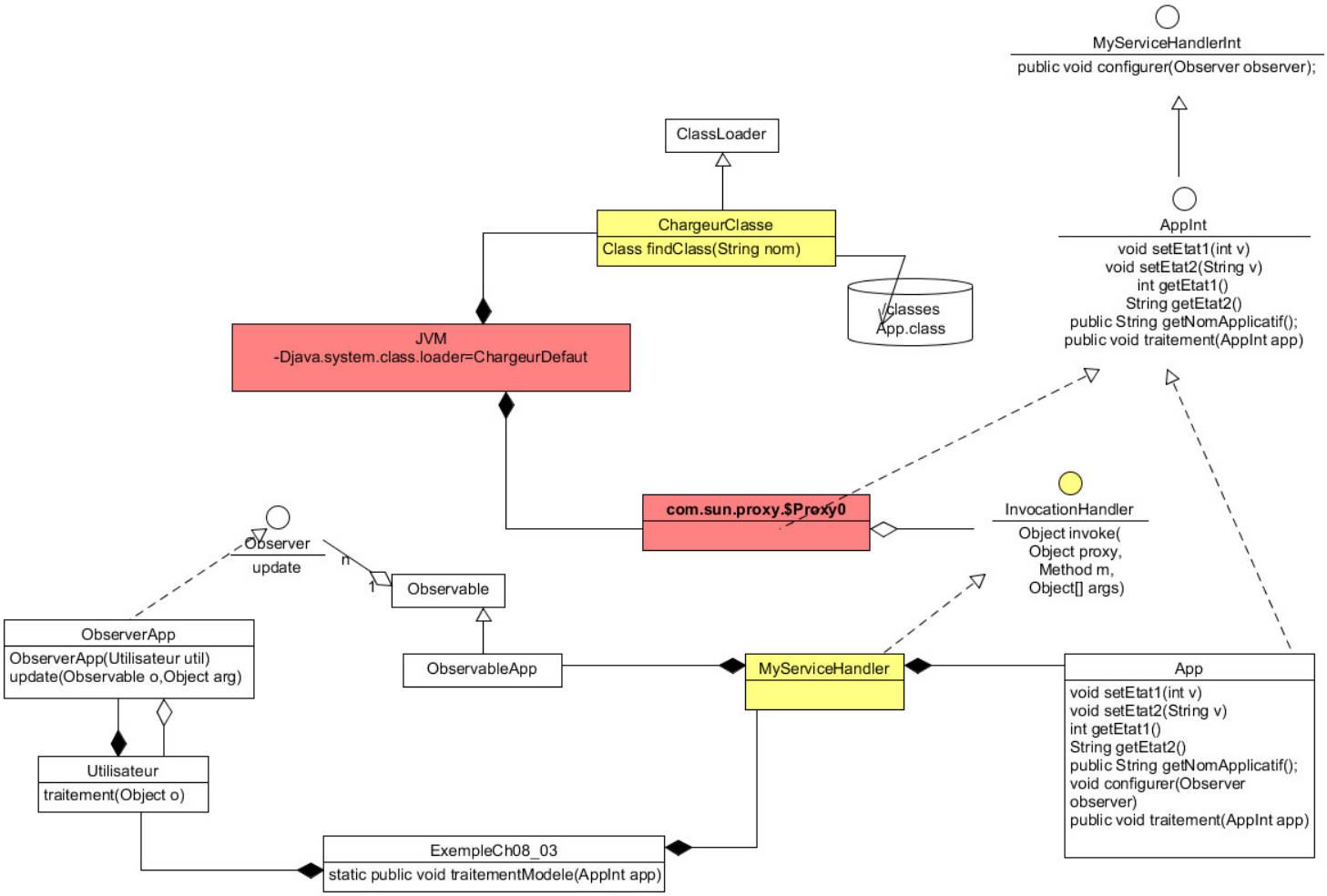
Voir sur le site http://jacques.laforgue.free.fr cours NSY102
l'exemple **ExempleCh08_03b_ServiceVirtualProxy**

Cet exemple montre les points suivants :

- des utilisateurs utilisent un applicatif à travers une interface
- l'implémentation de l'interface est un SERVICE VIRTUAL PROXY qui réalise les notifications
- Le SERVICE VIRTUAL PROXY qui utilise un chargeur de classe pour créer dynamiquement la classe applicative
- La classe applicative est ici dans un répertoire séparé du programme
- Le chargeur de classe charge des classes se trouvant dans un répertoire

5.2.3. Le DynamicProxy

Voir sur le site http://jacques.laforgue.free.fr cours NSY102
l'exemple **ExempleCh08_03c_DynamicProxy**



Cet exemple montre les points suivants :

- des utilisateurs utilisent un applicatif à travers une interface
- l'implémentation de l'interface est un DYNAMIC PROXY qui réalise les notifications
- Le DYNAMIC PROXY utilise un chargeur de classe pour créer dynamiquement la classe applicative
- Le DYNAMIC PROXY utilise un INVOCATION HANDLER qui implémente le mécanisme de notification
- On démontre que le même DYNAMIC PROXY peut être utilisé pour une toute autre interface et un tout autre applicatif
- Toutes les interfaces applicatives HERITENT d'une interface commune qui contient les méthodes propres au rôle du PROXY
- Le chargeur de classe est déclaré lors du lancement de la JVM
- La classe applicative est ici dans un répertoire séparé du programme
- Le chargeur de classe charge des classes se trouvant dans un répertoire
- On peut empiler plusieurs proxy : proxy de log, proxy d'authentification

6. Application à RMI

6.1. Le Proxy de communication

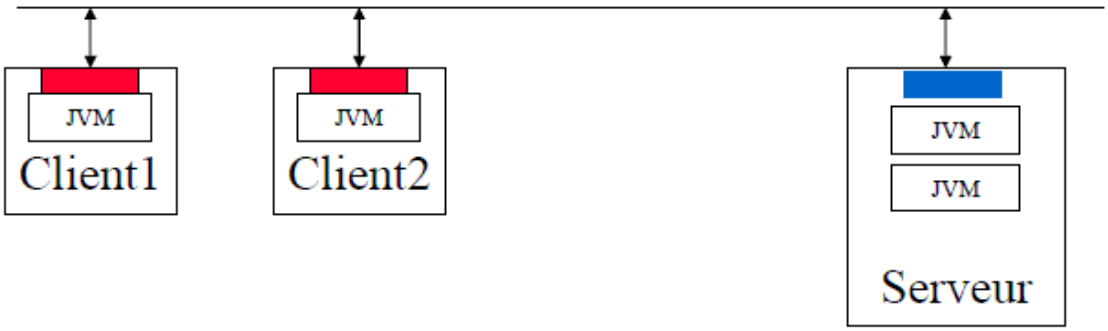
Objectif :
L'utilisation d'un Proxy afin de fournir à un tiers, un mandataire qui permet de contrôler l'accès d'un objet.

Le domaine d'application n'est pas uniquement la communication car les motivations sont de :

- contrôler
- différer
- optimiser
- sécuriser
- **accès distant** (communication) ➔ le domaine de prédilection

Les objectifs dans un appel distant sont :

- 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 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

Il existe la classe :

```
public class RemoteObjectInvocationHandler extends RemoteObject
                                    implements InvocationHandler
```

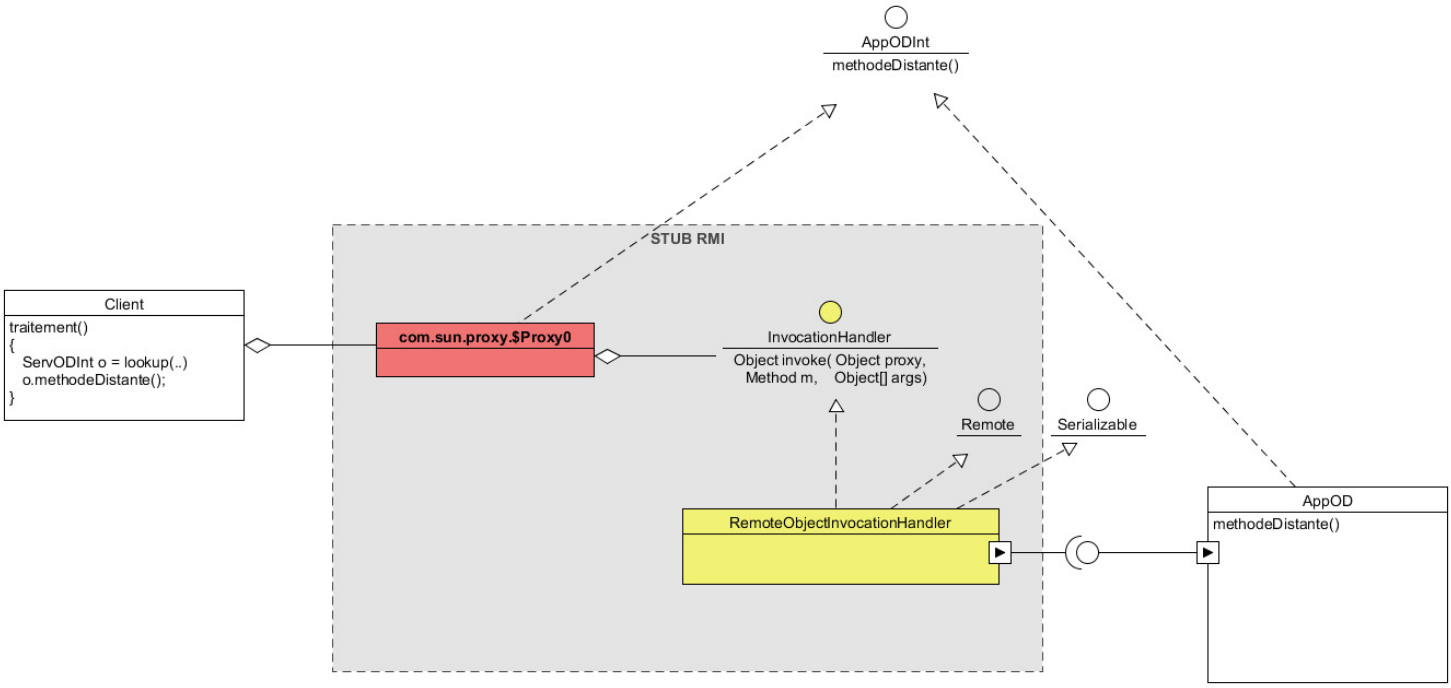
An implementation of the `InvocationHandler` *interface for use with Java Remote Method Invocation (Java RMI). This invocation handler can be used in conjunction with a dynamic proxy instance as a replacement for a pregenerated stub class.*

L'implémentation de la méthode **invoke** de l'interface InvocationHandler dans **RemoteObjectInvocationHandler** consiste à écrire et lire sur le socket (avec serialization des paramètres).

Ainsi, le lookup construit un DynamicProxy. Le mandataire réalise les accès et reste transparent pour l'utilisateur.

Le Stub est un DynamicProxy.

On obtient l'architecture de conception suivante :



7. Application du DynamicProxy à l'exemple Ch08_04 : le package myrmi

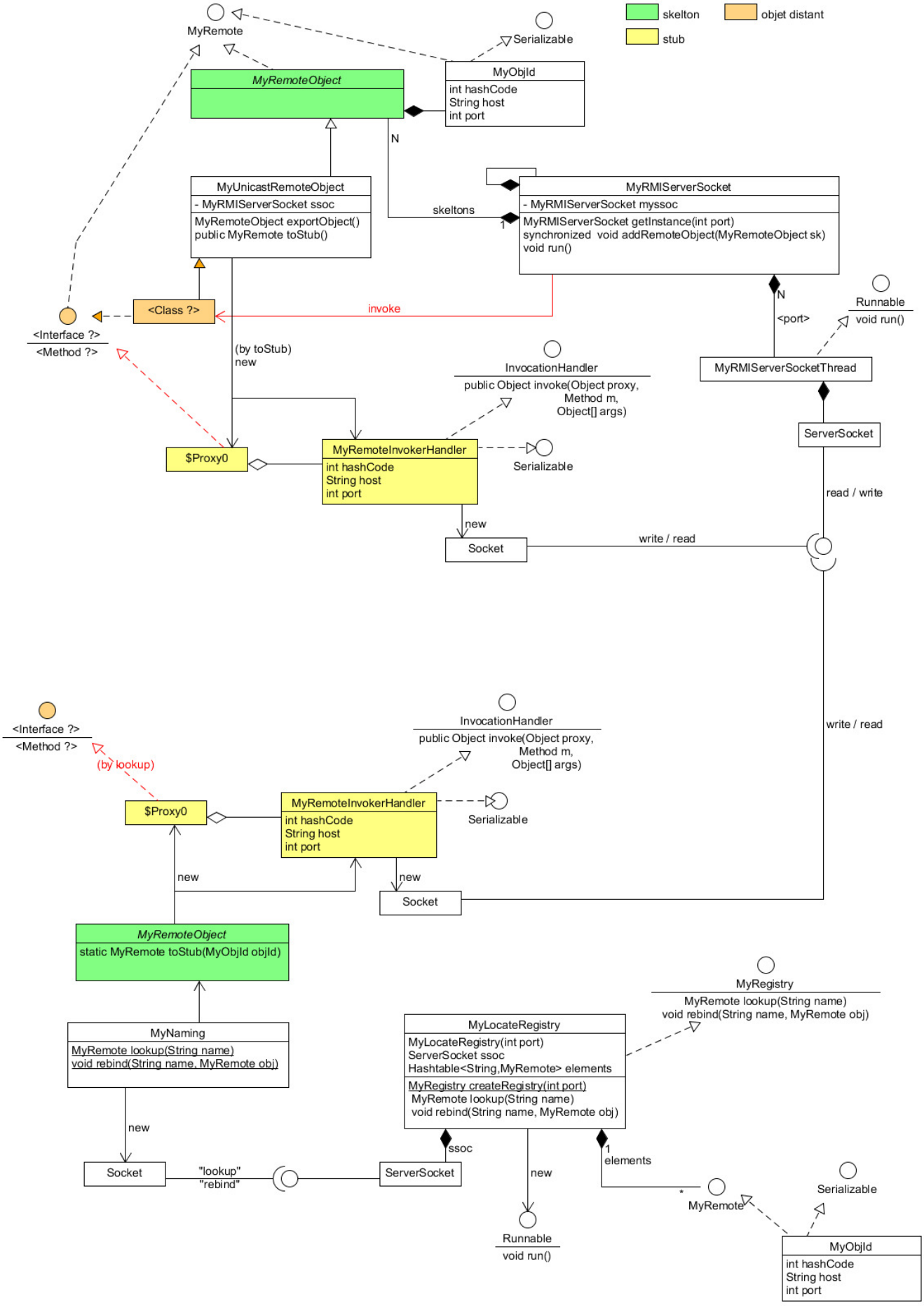
7.1. Présentation

Nous appliquons le principe du DynamicProxy sur l'exemple ExempleCh05_02c_InterfaceService2 afin que le stub ne soit plus codé en dur pour chaque classe d'OD mais afin qu'il soit pris en charge dynamiquement et génériquement par une instance du **Proxy.newProxyInstance.**

Le code complet de cet exemple est sur le site :

Voir **ExempleCh08_04_InterfaceService2**

7.2. Les diagrammes de classe du package myrmi


skelton
objet distant
stub
MyRemote
Serializable
MyRemoteObject
MyObjId
int hashCode
String host
int port
N
MyUnicastRemoteObject
- MyRMIServerSocket ssoc
MyRemoteObject exportObject()
public MyRemote toStub()
skeltons
1
MyRMIServerSocket
- MyRMIServerSocket myssoc
MyRMIServerSocket getInstance(int port)
synchronized void addRemoteObject(MyRemoteObject sk)
void run()
N
<port>
Runnable
void run()
<Class ?>
invoke
<Interface ?>
<Method ?>
(by toStub)
new
InvocationHandler
public Object invoke(Object proxy,
Method m,
Object[] args)
MyRMIServerSocketThread
ServerSocket
$Proxy0
MyRemoteInvokerHandler
int hashCode
String host
int port
Serializable
read / write
new
Socket
write / read
<Interface ?>
<Method ?>
(by lookup)
InvocationHandler
public Object invoke(Object proxy,
Method m,
Object[] args)
write / read
$Proxy0
MyRemoteInvokerHandler
int hashCode
String host
int port
Serializable
new
new
Socket
MyRemoteObject
static MyRemote toStub(MyObjId objId)
MyRegistry
MyRemote lookup(String name)
void rebind(String name, MyRemote obj)
MyNaming
MyRemote lookup(String name)
void rebind(String name, MyRemote obj)
MyLocateRegistry
MyLocateRegistry(int port)
ServerSocket ssoc
Hashtable<String,MyRemote> elements
MyRegistry createRegistry(int port)
MyRemote lookup(String name)
void rebind(String name, MyRemote obj)
new
ssoc
1
elements
Socket
"lookup"
"rebind"
ServerSocket
new
*
MyRemote
Serializable
Runnable
void run()
MyObjId
int hashCode
String host
int port

Commentaires de ce diagramme en cours.

7.3. Commentaire du code

Ainsi les classes DVDStub.java LivreStub.java et ServiceStub.java peuvent être supprimées.

De plus, en utilisant la réflexivité du langage Java, nous allons également coder le Skelton de manière dynamique et générique.

Ainsi les classes suivantes sont aussi supprimées :

DVDInvoker.java	DVDODSkelton.java
LivreInvoker.java	LivreODSkelton.java
ServiceInvoker.java	ServiceODSkelton.java

La classe MyRemoteObjectInvocation.java est également supprimée (utilisée par les XXXODSkelton).

Ainsi, cette simplification fait que le "skelton" devient l'OD lui-même.

Pour le nouveau codage du Stub, les modifications réalisées sont les suivantes.

Dans la classe MyRemoteObject dont MyUnicastRemoteObject hérite : la méthode toStub utilise le dynamic proxy pour crée le stub :

```
    static MyRemote toStub(MyObjId objId)
  {
    return (MyRemote)
java.lang.reflect.Proxy.newProxyInstance(
           objId.getClass().getClassLoader(),
           objId.getInterfaces(),
           new MyRemoteInvokerHandler(objId.getHashCode(),
                                      objId.getHost(),
                                      objId.getPort()) );
  }
```

Il est à noter que par rapport à l'exemple précèdent la classe MyObjId a été créée qui contient l'identifiant de l'objet distant.

Ceci afin que ce soit cet objet qui soit enregistré dans l'annuaire.

Le handler du Dynamic Proxy :

- il écrit sur le socket :
  - le hashcode de l'OD (qui a été transmis au stub lors de sa création)
  - le nom de la méthode
  - les paramètres de la méthode
- il attend, en lecture sur le socket, la réponse du serveur de socket.
- il retourne le résultat lu sur le socket

```
public class MyRemoteInvokerHandler implements InvocationHandler,
                                                Serializable
{
    private int hashCode;
    private String host;
    private int port;

    public MyRemoteInvokerHandler(int hashCode,String host,int port)
    {
        this.hashCode = hashCode;
```

```
            this.host = host;
            this.port = port;
        }

        public Object invoke(Object proxy,
                             Method m,
                             Object[] args) throws Throwable
        {
            Socket soc;
            if (host.equals("localhost"))
                soc = new Socket(InetAddress.getLocalHost(),port);
            else soc = new Socket(host,port);
            OutputStream os=soc.getOutputStream();
            ObjectOutputStream oos=new ObjectOutputStream(os);
            InputStream is=soc.getInputStream();
            ObjectInputStream ois=new ObjectInputStream(is);

            oos.writeObject(new Integer(hashCode));
            oos.writeObject(m.getName());  // Le nom de la methode
            if (args==null) oos.writeObject(new Integer(0));
            else oos.writeObject(new Integer(args.length));
            oos.writeObject(args);

            Object rep = ois.readObject();
            soc.close();

            return rep;
        }
    }
```

Pour le nouveau codage du skelton grâce à la réflexivité de Java, on appelle la méthode de l'OD (le skelton est l'OD) en fonction du nom de la méthode et des paramètres lus sur le socket.

La classe **MyRMIServerSocket** gère un tableau de **thread** qui est en attente d'un client.

Chaque thread, dans la classe MyRMIServerSocketThread.java (dans l'exemple précédent cette classe n'existait pas car ici on fait en sorte de pouvoir avoir autant de serveur de socket que de port différent à gérer dans la JVM).

Sur la connexion par le client au serveur de socket (classe *ServerSocket* de JAVA), la classe réalise le traitement suivant :

- elle lit le hashcode écrit par le stub
- elle lit le nom de la méthode et les paramètres de la méthode
- elle recherche le skelton correspondant au hashcode
- puis appelle la méthode de l'OD grâce à la réflexivité du langage Java.
- écrit le retour de l'appel sur le socket.

Les paramètres et la valeur de retour peuvent être écrits sur le socket car elles implémentent l'interface *Serializable*.

```
public void run()
{
  try{
  while (true)
  {
   Socket soc = ssoc.accept();
   OutputStream os=soc.getOutputStream();
   ObjectOutputStream oos=new ObjectOutputStream(os);
   InputStream is=soc.getInputStream();
   ObjectInputStream ois=new ObjectInputStream(is);

   // LECTURE DU HASHCODE (identification de l'OD)
   Integer hashCodeRemote = (Integer)(ois.readObject());

   // LECTURE DE LA METHODE
   String methode = (String)(ois.readObject());

   //RECHERCHE SU SKELTON
   MyRMIServerSocket server = MyRMIServerSocket.getInstance(port);
   MyRemoteObject skelton = null;
   for(MyRemoteObject sk : server.skeltons)
      if (sk.getHashCode() == hashCodeRemote.intValue())
            skelton = sk;

   if(skelton!=null)
   {
      System.out.println(">>invoke de "+methode);

      // LECTURE DU NOMBRE DES PARAMETRES DE LA METHODE
      Integer onbarg = (Integer)(ois.readObject());
      int nbarg = onbarg.intValue();
      if (nbarg!=0)
          {
              // LECTURE DES PARAMETRES DE LA METHODE (1 tableau)
              Object[] args = (Object[])(ois.readObject());

              // CALCUL DE LA SIGNATURE DE LA METHODE
              Class<?>[] classes = new Class<?>[args.length];
              int i=0;
              for(Object o:args) classes[i++]=o.getClass();

              // REFLEXIVITE DE LA METHODE DE L'OD
              Method m = skelton.getClass().getMethod(methode,classes);

              // APPEL DE LA METHODE
              Object rep = m.invoke(skelton,args);

              // ECRITURE DU RESULTAT
              if (rep==null)
                  oos.writeObject(new Integer(0));
              else
                  oos.writeObject(rep);
          }
      else  // PAS DE PARAMETRE
          {
              Method m = skelton.getClass().getMethod(methode);
              Object rep = m.invoke(skelton);
              if (rep==null)
                  oos.writeObject(new Integer(0));
              else
```

```
                    oos.writeObject(rep);
                }
            }
```

Quelques commentaires sur cet exemple :

Côté client :

La classe ExempleCh08 est un client (main) qui récupère le stub (ServiceODInt) via l'annuaire :

```
  ServiceODInt serv =
(ServiceODInt)MyNaming.lookup("rmi://localhost:9100/SERVER");
```

Côté serveur :

La classe Server est un serveur (main) qui crée un OD ServiceOD et l'enregistre dans l'annuaire.

```
  ServiceOD sod = new ServiceOD(portURO);
  MyNaming.rebind("rmi://localhost:"+portRegistry+"/SERVER",sod);
```

L'annuaire est créé par :

```
  MyLocateRegistry.createRegistry(portRegistry);
```

Le fonctionnement de l'annuaire :

L'annuaire est un main (JVM) de la classe MyLocateRegistry qui crée un serveur de socket et qui permet les requête "**lookup"** et "**rebind"**.

La classe MyNaming contient 2 méthodes static :

- **rebind** permettant d'y enregistrer un MyObjId qui contient l'identifiant de l'objet :
  le **hashcode** de l'objet
  le **host** de la machine contenant l'OD
  le **port** sur lequel le server de socket RMI attend les requêtes des clients de l'OD.
- **lookup** permettant de créer le proxy dynamique de communication avec l'OD en fonction des informations de MyObjId.

Les méthodes toSub :

Il existe deux méthodes **toStub**. Toutes deux créent le proxy dynamique de communication avec l'OD

La première (non static) implémentée dans MyUnicatRemoteObject dont hérite tout OD. Cet méthode est utilisée quand on veut créer le stub sur le serveur et l'envoyer à travers une méthode distante à un client.

La deuxième (static) implémentée dans MyRemoteObject, utilisée par la classe Naming sur le lookup quand on veut créer le stub sur un client.

7.4. Programme de test du package myrmil

L'exemple de code suivant permet de tester le package myrmi.

Voir sur le site **ExempleCh08_05_TesterMyRMI**

Le premier serveur, **Server1.java**, crée l'annuaire et crée deux services (SERV1-1 et SERV1-2), sur 2 ports différentes (afin de tester la gestion de plusieurs thread de port sur le serveur RMI). Ces 2 services sont deux instances d'un même OD qui réalise la multiplication du paramètre.

Le deuxième serveur, **Server2.java**, utilise l'annuaire à travers le réseau et crée un autre service (SERV2) qui réalise la somme du paramètre. Ce service est l'instance d'un autre OD.

Le premier client, **Client1.java**, utilise les 2 services du premier serveur : SERV1-1 et SERV1-2.

Le deuxième client, **Client2.java**, utilise 1 service du premier serveur SERV1-1 et le service du deuxième serveur SERV2.

Le paramètre de retour de chacun de ces services est un objet sérialisé de classe **Resultat**.

On obtient l'architecture suivante :