# Chapitre 5

## Service et interface

L'objectif de ce chapitre est de présenter le concept de la notion de service et interface dans les applications intranet.

# 1. Qu’est ce qu’un service

## 1.1. Introduction

Un service (de type RPC : Remote Process Call) est, en informatique, mise en œuvre par un intergiciel (middleware)

Fonctions attendues de l ’intergiciel :

- Proposer une API de haut Niveau utilisé par le client et par le serveur.
- Masquer l ’hétérogénéité des informations manipulées
- Faciliter la programmation répartie et donc rendre invisible la répartition des données et des programmes (Annuaire, Factory : getService(« NOM DU SERVICE »))

## 1.2. Définition

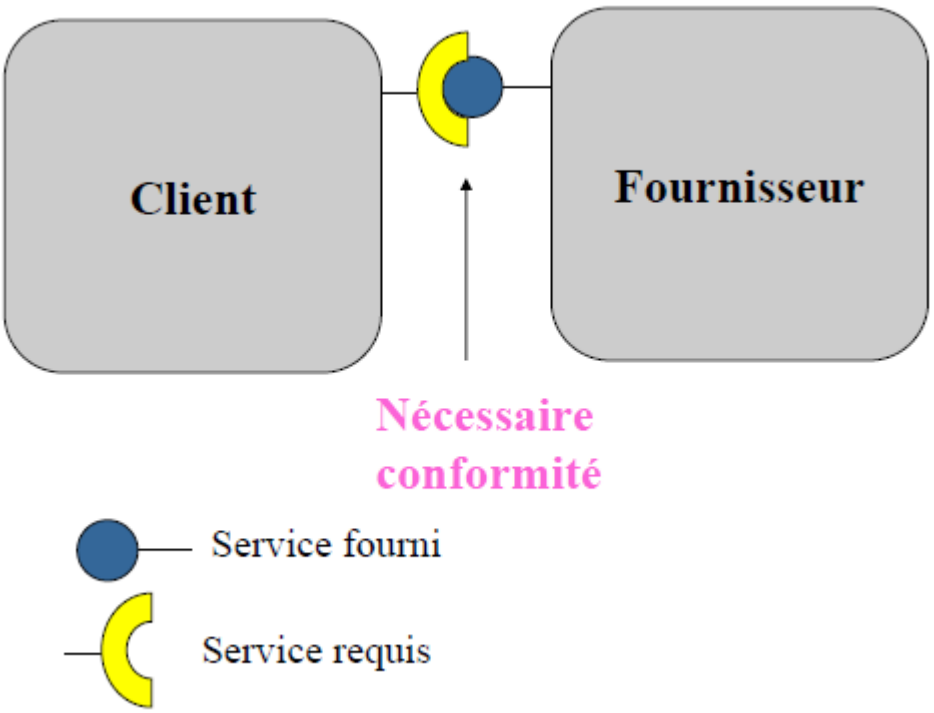
Un service est un comportement défini par un **contrat**, qui peut être implémenté et fourni par un composant afin d’être utilisé par un autre composant sur la base exclusive du contrat

Un service est accessible via une **interface**.

L’interface décrit l’interaction entre le client et le fournisseur de service.

*Remarquer qu'à ce stade nous ne parlons pas de « serveur » mais de « fournisseur d’un service ». Bien sur, dans l’aspect « système », il existe toujours un serveur. Mais le serveur devient une structure d’accueil "générique" de fournisseur d’un service.*
*Exemple : un serveur est composé de plusieurs factory qui sont des objets distribués*
*Ces "serveurs" (ex: Serveur de Socket, Web Service, Restful, …) sont des composants du Middleware.*

## 1.3. Description




Le contrat doit être conforme = compatibilité entre le « Service fourni » et le « Service requis »

.

- I1 =  qui est un composant déployé sur le Client
- I2 =  qui est un composant déployé sur le Fournisseur de Service

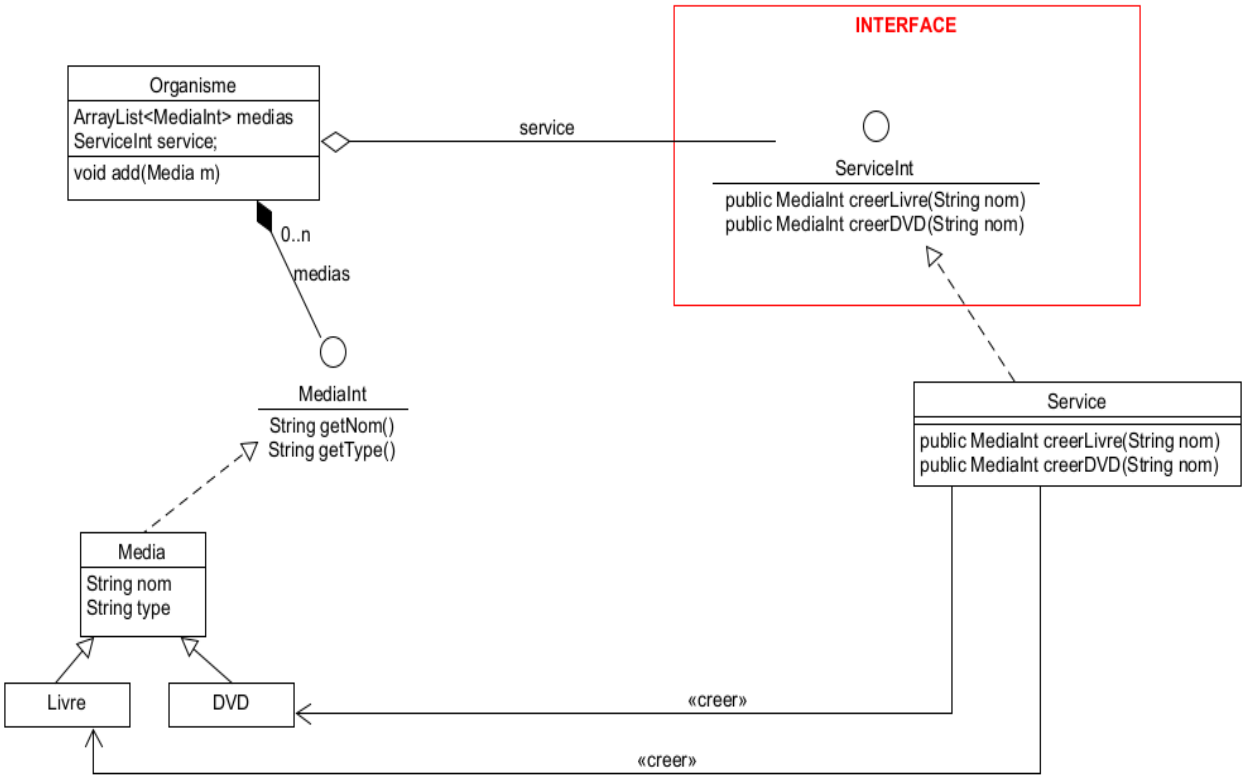
## 1.4. Exemple d'une interface locale

Exemple d'une interface client / fournisseur dont le service est utilisé **en local** d'un programme Java.

Cet exemple illustre les points suivants:

- le service est créé dans le même programme que l'utilisateur du service
- le service crée des objets de natures différentes et les retourne à l'utilisateur
- l'utilisateur récupère ces objets créés et les stocke dans ses collections
- le service est vu par l'utilisateur sous la forme d'une interface
- les objets créés sont vus par l'utilisateur sous la forme d'une interface


Voir sur le site http://jacques.laforgue.free.fr l'exemple
**ExempleCh05_01_InterfaceService1**




## 1.5. Exemple d'une interface distante

On fait évoluer l'exemple précédent afin que le service soit distant. Pour cela, on définit un proxy de communication afin de limiter l'impact sur le code précédent.
Cela n'est possible que parce que précédemment on a utilisé le DP Interface.

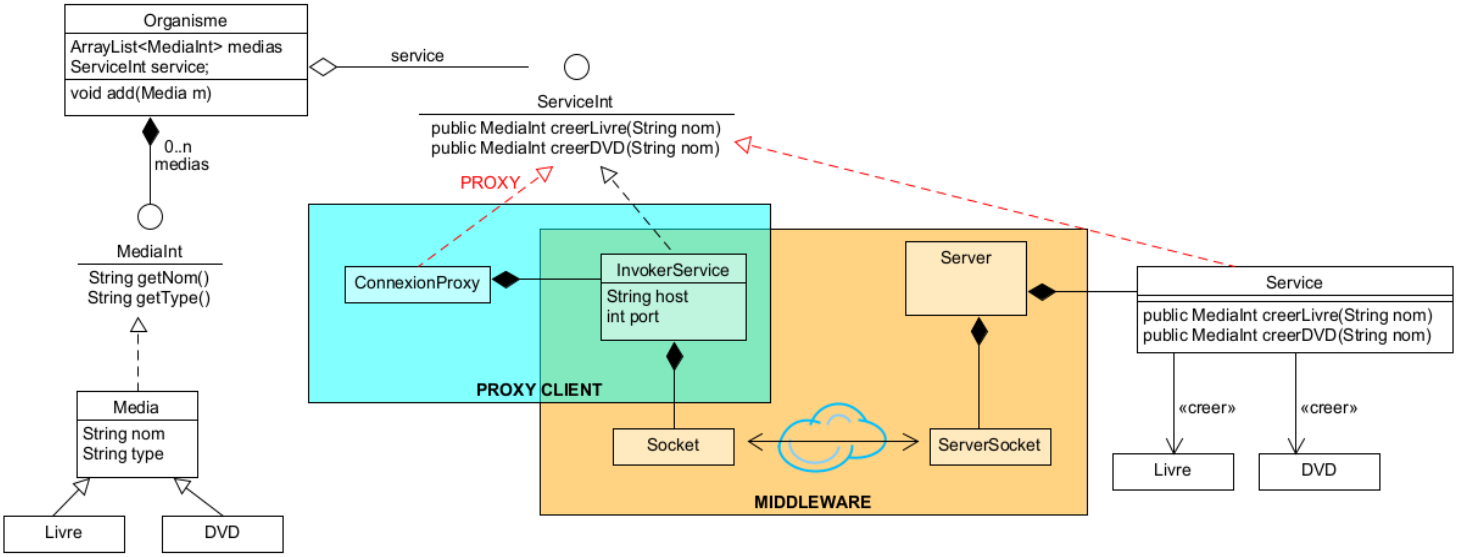
De plus, pour bien démontrer l'aspect distant, on implémente la communication entre l'exemple et le serveur avec des sockets.

Le service **est distant** et est vu comme un PROXY.

Cet exemple illustre les points suivants:

- L'utilisateur (Organisme) du service utilise le service à travers un proxy de communication
- Cette utilisation lui est transparente grâce au choix précédent de l'interface
- Ici, l'utilisateur récupère toujours une copie des objets créés par le serveur et les stocke dans ses collections


Voir sur le site http://jacques.laforgue.free.fr l'exemple
**ExempleCh05_02a_InterfaceService2**




Nous pouvons remarquer que :

- Le ConnexionProxy assure la transparence de communication avec le service. Il est un **DP Proxy de communication** "logique" avec Serveur
- Il est aussi un DP Proxy dit "Client" avec la classe InvokerService qui assure la communication "technique".
- Les classes InvokerService et Server représente le Middleware de communication.
- L'interface ServiceInt est un composant qui est utilisé à la fois par le "client" et par le "serveur".

## 1.6. Exemple de "stub"

Nous avons vu que dans l'exemple précédent les objets sont créés sur le serveur. Puis une copie de ces objets sont envoyés au client.

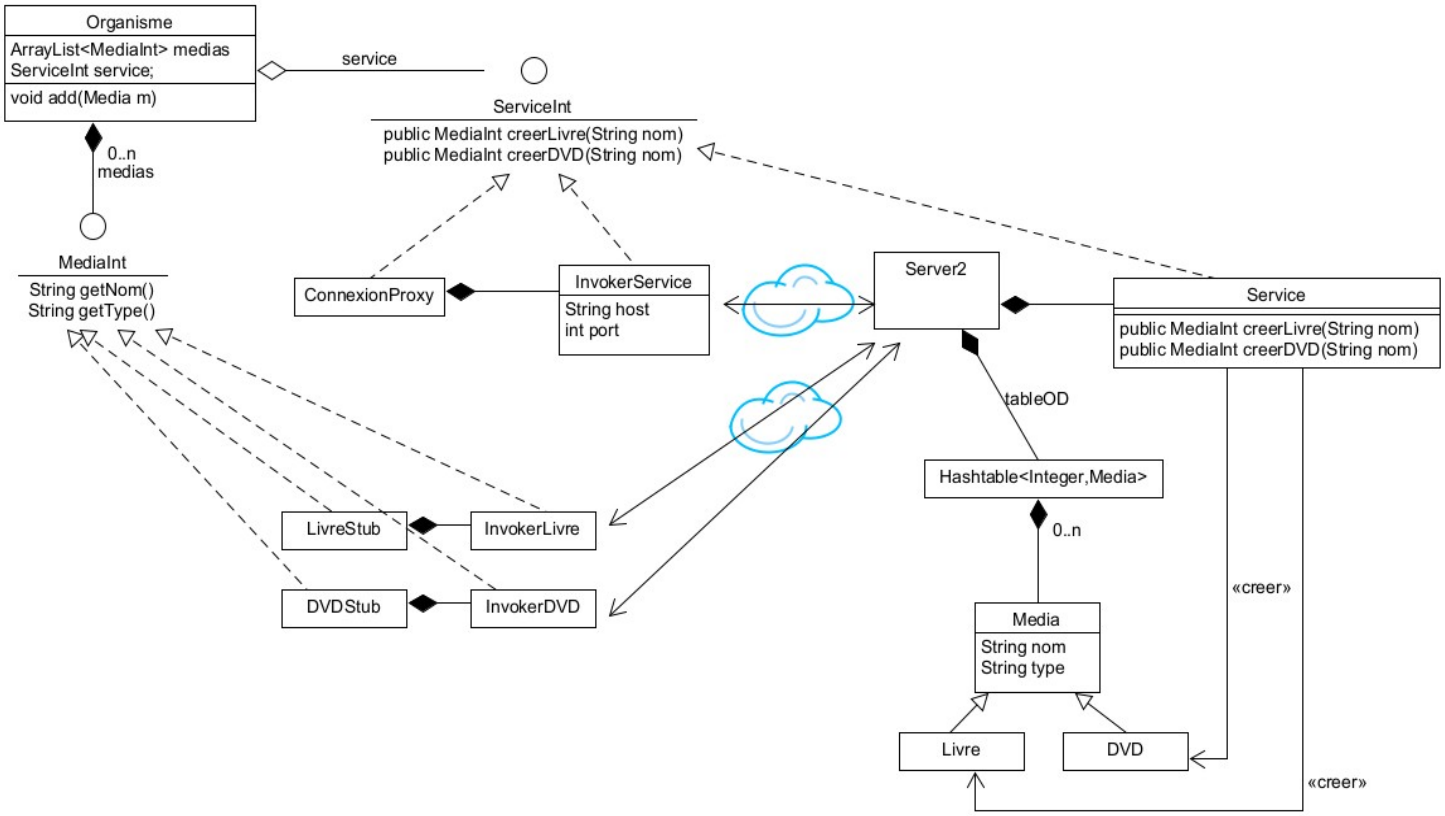
Nous voulons ici faire en sorte que le serveur retourne un "stub" et non l'objet lui-même. Ainsi les objets sont créés sur le serveur et le client les utilise de manière distante.

Nous allons implémenter ce "stub" sous la forme d'un **Proxy de communication**.

Cet exemple illustre les points suivants :

- Les objets Livre et DVD sont devenus des objets distants interfacés avec le client par un Proxy de communication
- Chaque type d'objet a son Proxy client (ou stub)


Voir sur le site http://jacques.laforgue.free.fr l'exemple
**ExempleCh05_02b_InterfaceService2**

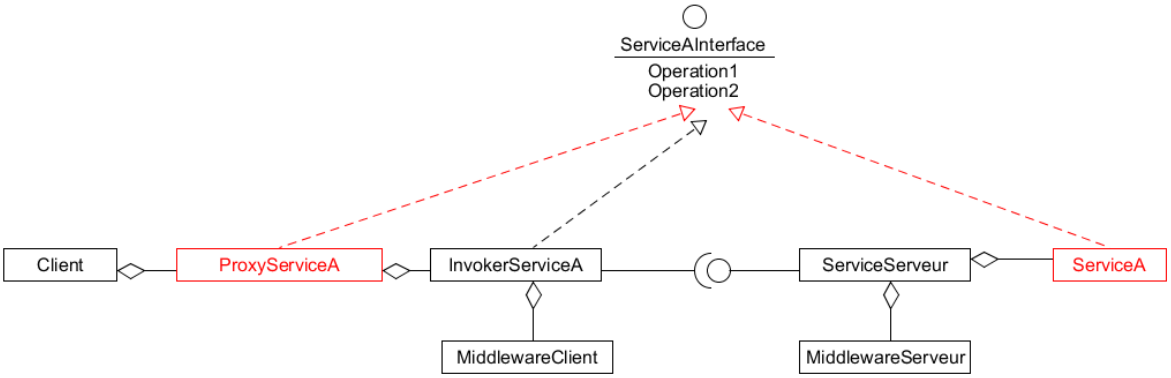



Nous pouvons remarquer que :

- Tous les invokers communiquent avec un même serveur de socket
- Pour retrouver les différents objets Livre et DVD, une identification unique de ces derniers est mise en place (hashCode).

## 1.7. Le DP Proxy de communication

Fort de la présentation précédente, on peut dégager le DP de Proxy de Communication :




Le ProxyServiceA sert de proxy de communication avec le service ServiceA. Son rôle est de rendre transparent pour le Client le fait qu'il va utiliser le ServiceA ne sachant pas que ce service est distant.

Ce proxy utilise un autre proxy InvokerServiceA qui a pour rôle d'invoquer à distance les méthodes de ServiceA. Ce proxy va implémenter chaque méthode de l'interface afin de réaliser une **requête de communication** en fonction des propriétés du Middleware utilisé (dans notre exemple précédent le middleware utilisé est le socket).

A la charge du serveur qui reçoit ces requêtes d'appeler les méthodes de Service A.

## 1.8. Le DP Adaptateur de communication

Dans l'exemple précédent, le serveur du service n'était absolument pas générique : toutes les requêtes de chaque méthode de chacun des services :

- Connexion serveur à distance,
- Livre à distance,
- DVD à distance

étaient à plat dans un switch unique.

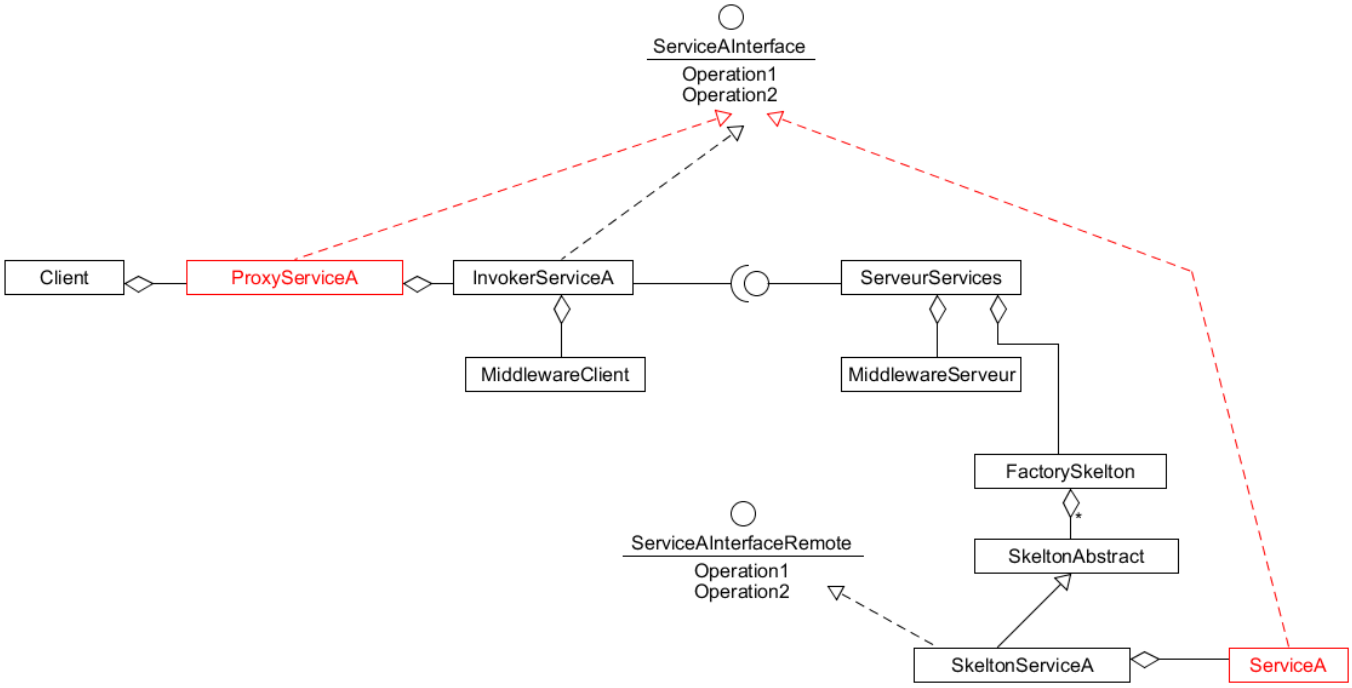
Pour résoudre cette problématique, il faut créer un serveur qui crée un objet dédié à chaque service. Cet objet dédié prend en charge la prise en compte des requêtes propre au service. Cet objet est appelé un **Skelton.**

Ce skelton est décrit par le serveur à travers une interface distante.

Le rôle de ce skelton est de convertir cette interface distante en l'interface du service.

Ce skelton est donc un Adptateur. C'est un **Adptateur de Communication**.

On obtient la description UML suivante :




Le serveur de services (ServeurServices) gère dans un factory de skeltons.

Chaque service a son skelton qui implémente l'interface distante afin d'appeler les méthodes du service : il est un Adaptateur de l'interface distante du skelton à l'interface du service.

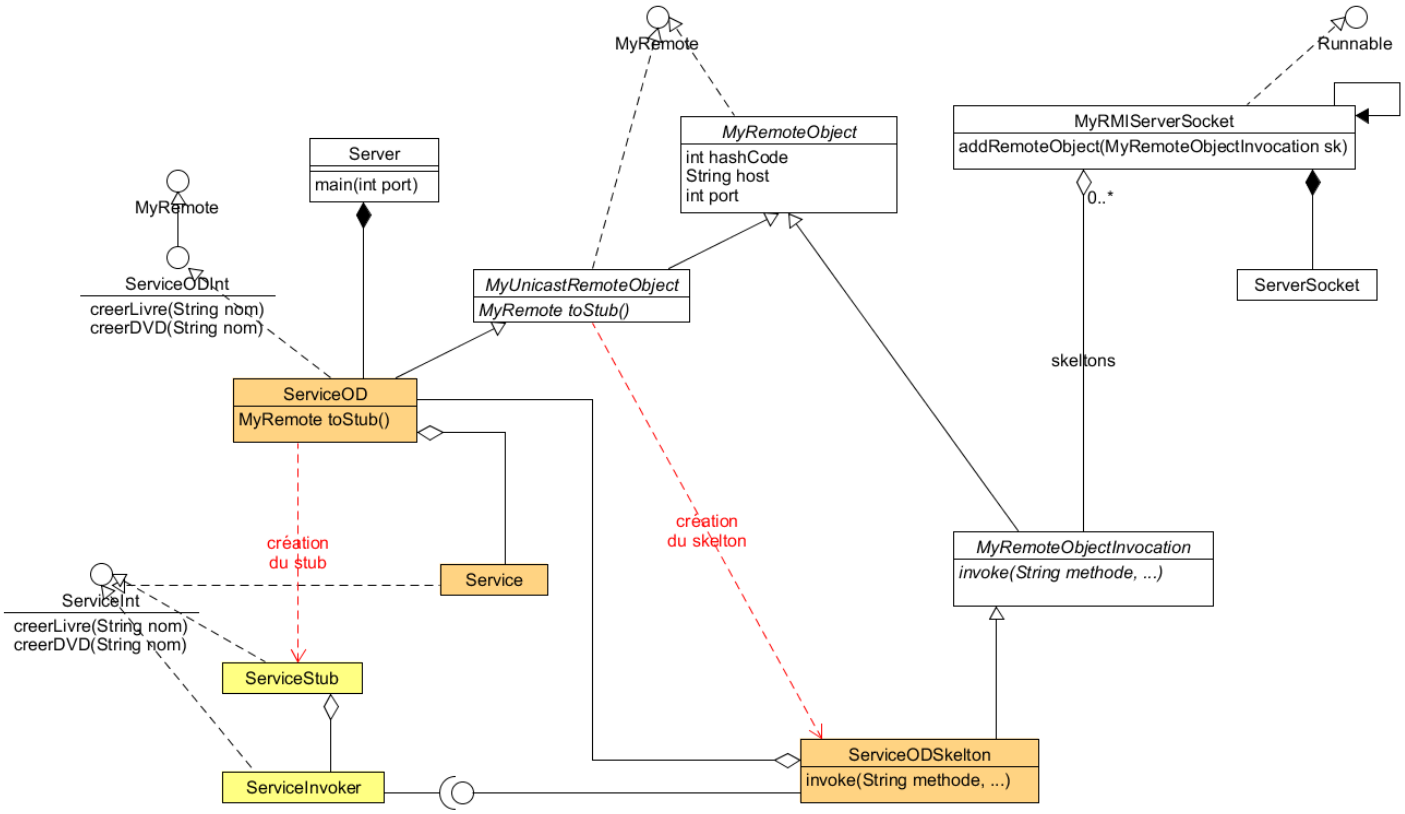
## 1.9. Exemple complet


Voir sur le site http://jacques.laforgue.free.fr l'exemple
**ExempleCh05_02c_InterfaceService2**

Cet exemple est le même que précédemment mais avec une implémentation de classe qui implémente les services sur le serveur sous la forme de skeltons.

Dans cette implémentation, nous avons fait le choix de se rapprocher de ce que fait RMI dans on implémentation.

On obtient le diagramme UML suivant, pour 1 seul service « service de connexion »:




Commentaires en cours

Dans cette implémentation, il est nécessaire de créer soi-même les classes XXXStub et XXXSkelton pour créer un service.

Il est à noter que ces deux éléments ont toujours été générés automatiquement en Java ou dans les framework.. Mais ce n'est pas le cas, dans de nombreux langages.

On peut remarquer que toutes les classes qui mettent en œuvre le « mécanisme » sont abstraites. Cela veut dire que ce diagramme UML et son code associé (qu'il suffit de porter dans un autre langage) est une solution générique et un cadre (donc un DP d'architecture) pour développer une communication aussi puissante que celle de RMI en Java.

De plus, le middleware utilisé ici est la technologie socket, qui existe dans tous les langages de programmation.

# 2. Designs Patterns d'un Objet Distant

L'objectif de ce paragraphe est de montrer les différentes conceptions qu'il est possible d'envisager pour créer un objet distant tout en les formalisant sous la forme de Design pattern en utilisant la technologie RMI.

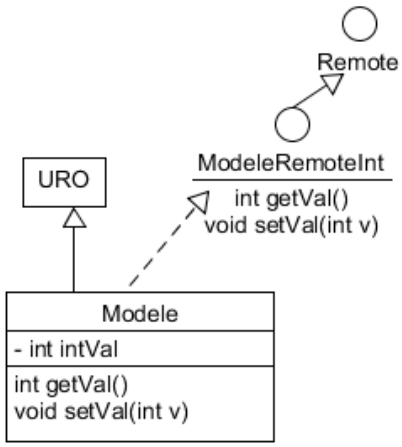
Il existe 5 approches :
- par héritage
- par composition ou agrégation
- par adaptateur
- par double adaptateur
- par proxy

Nous avons au départ une classe Modele définie comme suit, et nous voulons la "transformer" en un objet distant :

```
class Modele
{
    private int val;
    public Modele(int valInit){ val=valInit; }
    public int getVal(){return val;}
    public void setVal(int val){this.val = val;}
}
```

## 2.1. Conception d'un OD par héritage

Dans ce cas, la classe Modele est profondément modifiée afin qu'elle devienne un objet distribué par héritage de la classe UnicastRemoteObject.




```
import java.rmi.*;
import java.rmi.server.*;

// Conception par heritage


class Modele extends UnicastRemoteObject implements ModeleInt
{
    private int val;
    public Modele(int port,int valInit) throws RemoteException
```

```
    {
        super(port);
        val=valInit;
    }
    public int getVal() throws RemoteException
    {return val;}
    public void setVal(int val) throws RemoteException
    {this.val = val;}
}

interface ModeleInt extends Remote
{
    public int getVal() throws RemoteException;
    public void setVal(int val) throws RemoteException;
}

public class CodeParHeritage
{
    public void main(String[] args) throws Exception
    {
        ModeleOD modod = new ModeleOD(9100,100);
    }
}
```

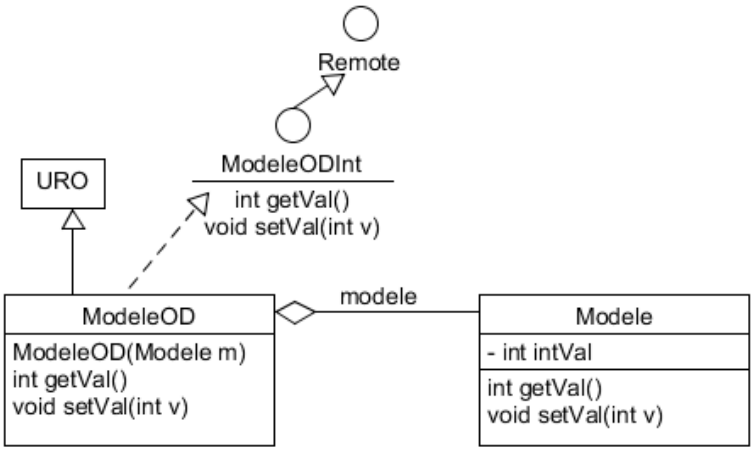
## 2.2. Conception d'un OD par composition ou agrégation

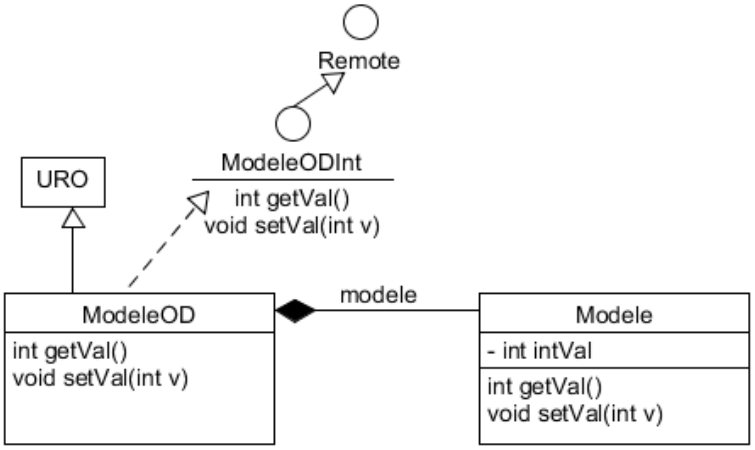
Dans ce cas on crée un objet distant à part entière qui va être liée à la classe Modele.

Dans le cas de la composition les deux classes sont intimement liées par leurs constructeurs, le modèle étant créé par l'OD.

Dans le cas de l'agrégation les deux classes sont créées indépendamment.

La classe Modele reste par contre inchangée.



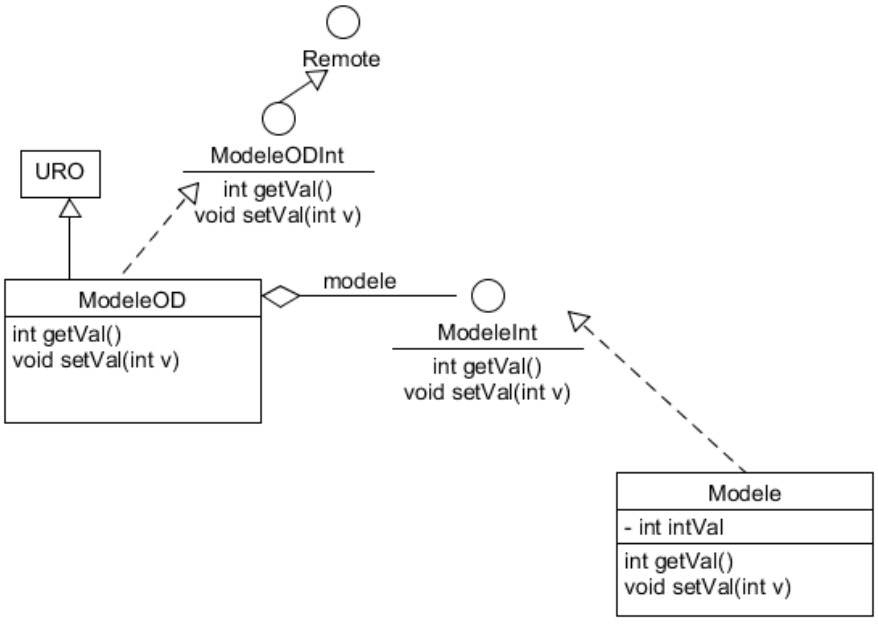

Exemple de code : par composition

```
import java.rmi.*;
import java.rmi.server.*;

class Modele
{
    private int val;
    public Modele(int valInit){ val=valInit; }
    public int getVal(){return val;}
    public void setVal(int val){this.val = val;}
}

// Conception par composition
class ModeleOD extends UnicastRemoteObject implements ModeleODInt
{
    private Modele modele;
    public ModeleOD(int port,int valInit) throws RemoteException
    {
        super(port);
        modele = new Modele(valInit);
```

```
    }
    public int getVal() throws RemoteException
    {
        return modele.getVal();
    }
    public void setVal(int val) throws RemoteException
    {
        modele.setVal(val);
    }
}

interface ModeleODInt extends Remote
{
    public int getVal() throws RemoteException;
    public void setVal(int val) throws RemoteException;
}

public class CodeParComposition
{
    public void main(String[] args) throws Exception
    {
        ModeleOD modod = new ModeleOD(9100,100);
    }
}
```

## 2.3. Conception d'un OD par adaptateur

Ce cas est très proche du précédent. La différence est l'utilisation d'une interface.
Ce DP a l'avantage que l'OD s'adapte à toute classe qui implémente l'interface métier.




ModelOD est un adaptateur de conversion de l'interface ModelInt à l'interface ModeleODInt.

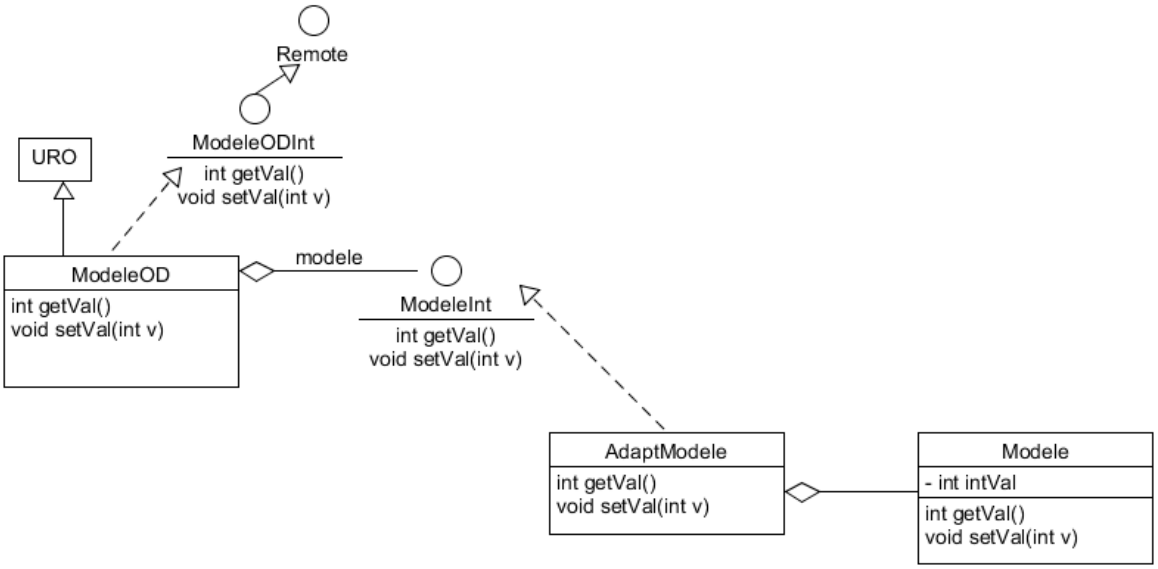
```
import java.rmi.*;
import java.rmi.server.*;
```

```
  // Conception par interface


  class Modele implements ModeleInt
  {
      private int val;
      public Modele(int valInit){ val=valInit; }
      public int getVal(){return val;}
      public void setVal(int val){this.val = val;}
  }

  class ModeleOD extends UnicastRemoteObject implements ModeleODInt
  {
      private ModeleInt modele;
      public ModeleOD(int port,ModeleInt modele) throws RemoteException
      {
          super(port);
          this.modele = modele;
      }
      public int getVal() throws RemoteException
      {
          return modele.getVal();
      }
      public void setVal(int val) throws RemoteException
      {
          modele.setVal(val);
      }
  }

  interface ModeleInt
  {
      public int getVal();
      public void setVal(int val);
  }

  interface ModeleODInt extends Remote
  {
      public int getVal() throws RemoteException;
      public void setVal(int val) throws RemoteException;
  }

  public class CodeParInterface
  {
      public void main(String[] args) throws Exception
      {
          ModeleInt mod = new Modele(100);
          ModeleOD modod = new ModeleOD(9100,mod);

          ModeleInt mod2 = new Modele2(100);   // Modele2 qui implémente
l'interface ModeleInt
          ModeleOD modod2 = new ModeleOD(9100,mod);

      }
  }
```

## 2.4. Conception d'un OD par double adaptateur

On est dans le cas précédent mais on veut utiliser un modèle qui n'est pas compatible avec l'interface métier ModeleInt.

Dans ce cas il faut utiliser un adaptateur à l'interface ModelInt.




AdaptModele est un adaptateur à l'interface ModelInt de Modele

ModelOD est un adaptateur de conversion de l'interface ModeleInt en l'interface ModelODInt.

```
import java.rmi.*;
import java.rmi.server.*;

// Conception par adaptateur

class Modele
{
    private int val;
    public Modele(int valInit){ val=valInit; }
    public int getVal(){return val;}
    public void setVal(int val){this.val = val;}
}

class AdaptModele implements ModeleInt
{
    private Modele modele;
    public AdaptModele(Modele modele){ this.modele = modele; }
    public int getVal(){return modele.getVal();}
    public void setVal(int val){modele.setVal(val);}
}

class ModeleOD extends UnicastRemoteObject implements ModeleODInt
{
    private ModeleInt modele;
    public ModeleOD(int port,ModeleInt modele) throws RemoteException
    {
        super(port);
        this.modele = modele;
    }
    public int getVal() throws RemoteException
```

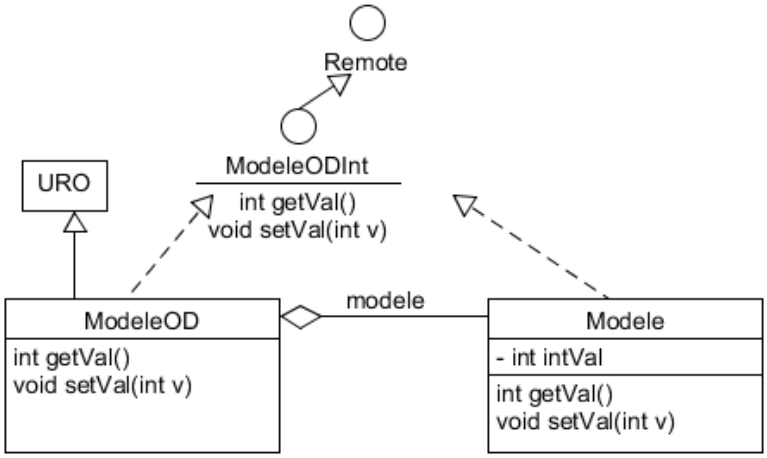
```
    {
        return modele.getVal();
    }
    public void setVal(int val) throws RemoteException
    {
        modele.setVal(val);
    }
}

interface ModeleInt
{
    public int getVal();
    public void setVal(int val);
}

interface ModeleODInt extends Remote
{
    public int getVal() throws RemoteException;
    public void setVal(int val) throws RemoteException;
}

public class CodeParAdaptateur
{
    public void main(String[] args) throws Exception
    {
        Modele modele = new Modele(100);
        ModeleInt mod = new AdaptModele(modele);
        ModeleOD modod = new ModeleOD(9100,mod);
    }
}
```

## 2.5. Conception d'un OD par proxy

Dans ce cas, l'interface de définition du modele est déjà exprimée sous la forme d'une interface distante (potentiel).

Remarque : on peut s'apercevoir que l'interface Remote ne contient pas de méthode. (C'est le cas de l'interface Serializable).

On pourrait donc systématiquement, pour toutes nos classes dont les objets pourraient être utilisés à terme d'une manière distante, les faire implémenter l'interface Remote.




Il suffit alors

```
import java.rmi.*;
import java.rmi.server.*;

// Conception par proxy

class Modele implements ModeleInt
{
    private int val;
    public Modele(int valInit){ val=valInit; }
    public int getVal(){return val;}
    public void setVal(int val){this.val = val;}
}

class ModeleOD extends UnicastRemoteObject implements ModeleInt
{
    private ModeleInt modele;
    public ModeleOD(int port,ModeleInt modele) throws RemoteException
    {
        super(port);
        this.modele = modele;
    }
    public int getVal()
    {
        try{
            return modele.getVal();
        }catch(RemoteException ex){ };
        return 0;
    }
    public void setVal(int val)
    {
         try{
             modele.setVal(val);
         }catch(RemoteException ex){ };
    }
}

interface ModeleInt extends Remote
{
    public int getVal() throws RemoteException;
    public void setVal(int val) throws RemoteException;
```

```
}

public class CodeParProxy
{
    public void main(String[] args) throws Exception
    {
        ModeleInt mod = new Modele(100);
        ModeleOD modod = new ModeleOD(9100,mod);
    }
}
```

# 3. Le patron "Proxy" et son application ( Application à RMI et http : Exemple Ch05_03_ServiceRMI )


Voir sur le site http://jacques.laforgue.free.fr l'exemple **ExempleCh05_03_ServiceRMI**

Dans ce répertoire vous trouverez le fichier pdf **ExempleCh05_03.pdf** qui décrit la conception de cet exemple et sa mise en œuvre (compilation et exécution).

Son objectif est de démontrer l'intérêt à utiliser une interface de description des services informatiques utilisés par un autre composant informatique.

Cet exemple est composé de plusieurs cas d'implémentation successifs du même « programme ». Chaque cas successif correspond à une architecture logicielle de plus en plus complexe jusqu'à la séparation physique de l'utilisation des services et des services eux-mêmes.

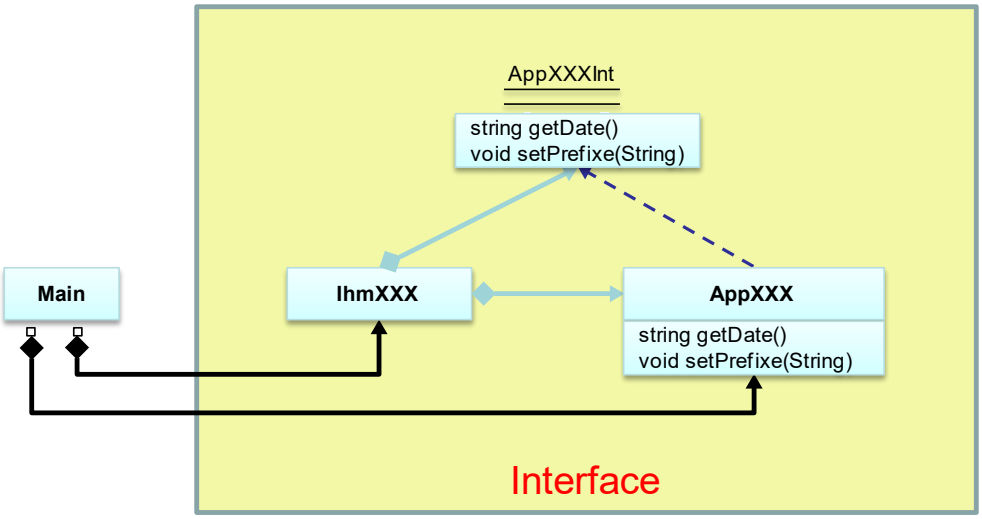
Cet exemple démontre que :

- L'interface joue un rôle important dans l'architecture des SI
- L'interface permet d'utiliser un composant informatique indépendamment de son implémentation par le choix de la création d'un Proxy
- Elle permet l'interopérabilité des composants (interopérabilité entre http et RMI)
- Elle permet d'abstraire les moyens de communication à mettre en place au sein du SI.

Ci-dessous des schémas UML de description des différents cas d'architecture de ce SI.

**Dans les 3 cas, à aucun moment le code de l'IHM et de l'Applicatif ne sont modifiés car on utilise une interface. (Si on n'avait pas d'interface on ferait des adaptateurs de chaque côté).**

## *3.1. Cas 1 : programme unique*




Les classes d'IHM et d'Applicatif sont créées par le programme principal.

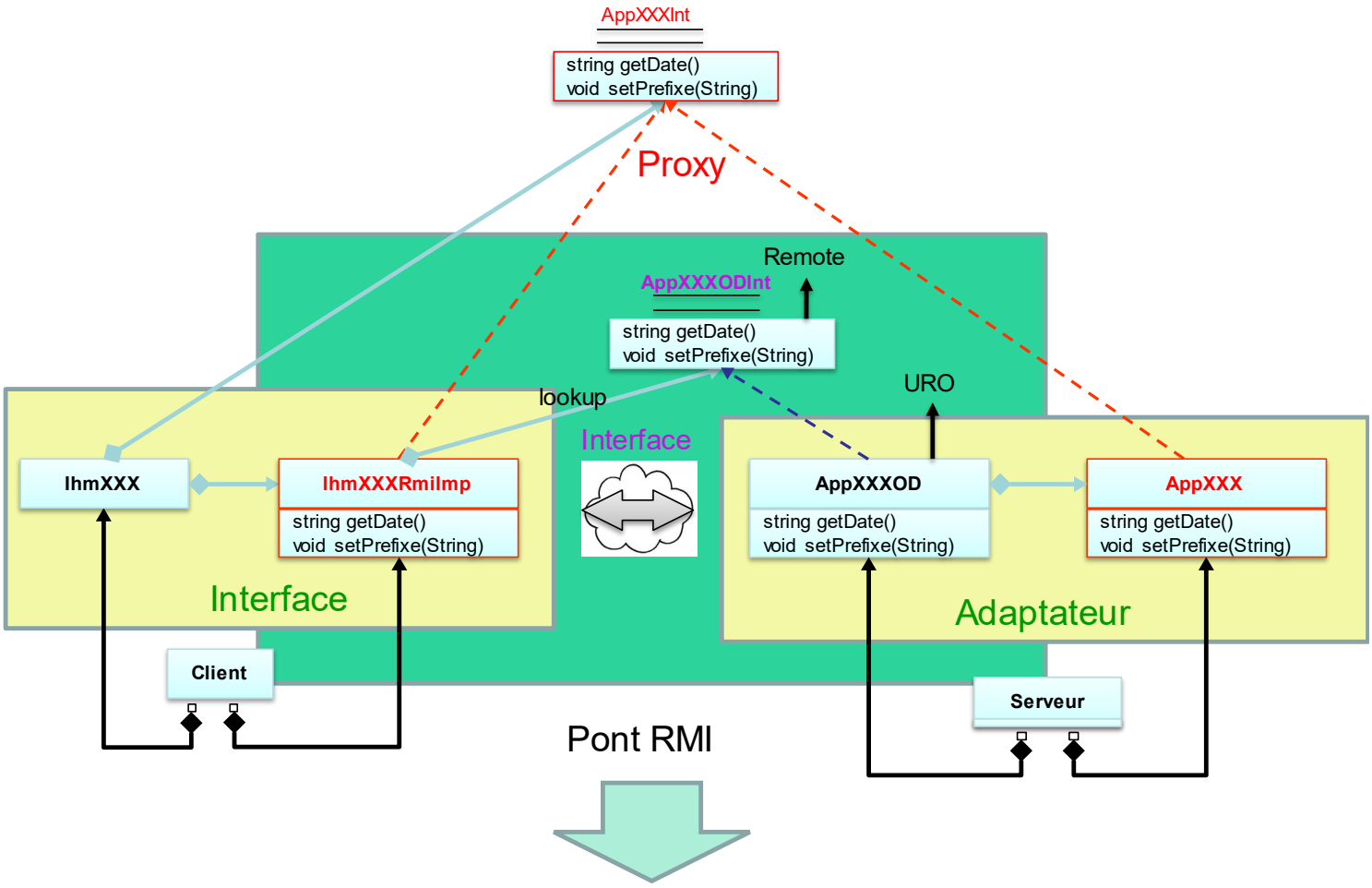
L'IHM utilise en local l'Applicatif via une interface.

ApppXXXInt est une interface locale.

Nous utilisons ici le DP "**Interface de traitement**".

Voir le code.

## 3.2. Cas 2 : architecture client / serveur en RMI




Ce DP encadré correspond à un "pont RMI" entre l'IHM et l'Applicatif

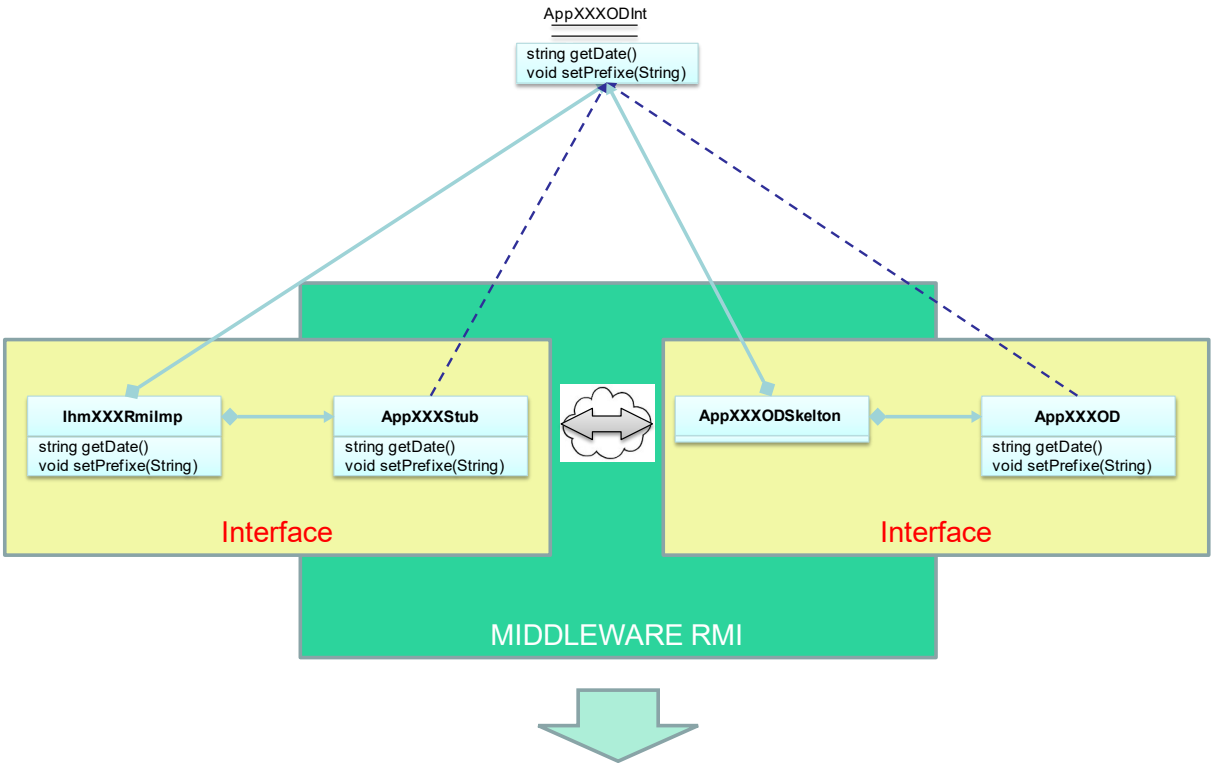
AppXXXODInt est l'interface distante.

La classe IhmXXXRMIImp réalise le lookup sur l'OD.
Elle est un **proxy** logique de AppXXX.
L'interface de communication se fait entre IhmXXXRMIImp et AppXXXOD.
Ces 2 classes définissent le protocole de communication qui est ici le protocole RMI.

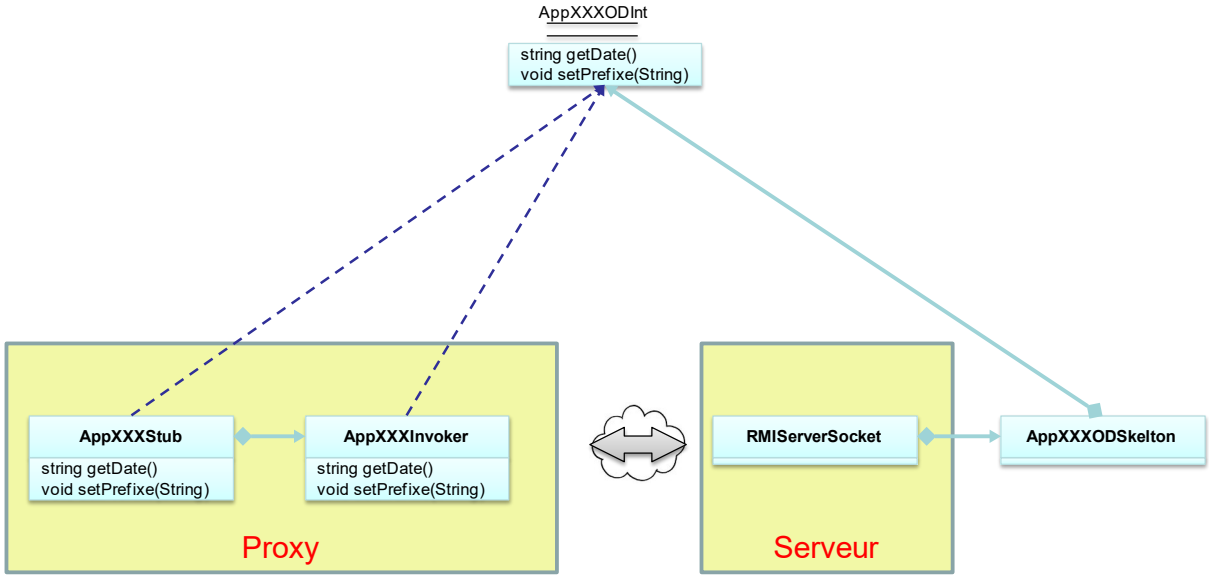
La classe AppXXXOD implémente les méthodes de AppXXX sous la forme de méthodes distantes. Il est un **adaptateur** de l' interface distante.

Voir le code.

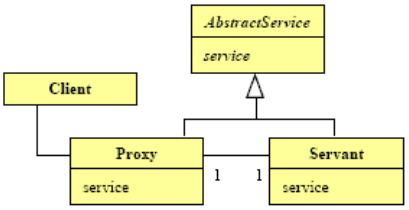
On détaille le "pont RMI" :

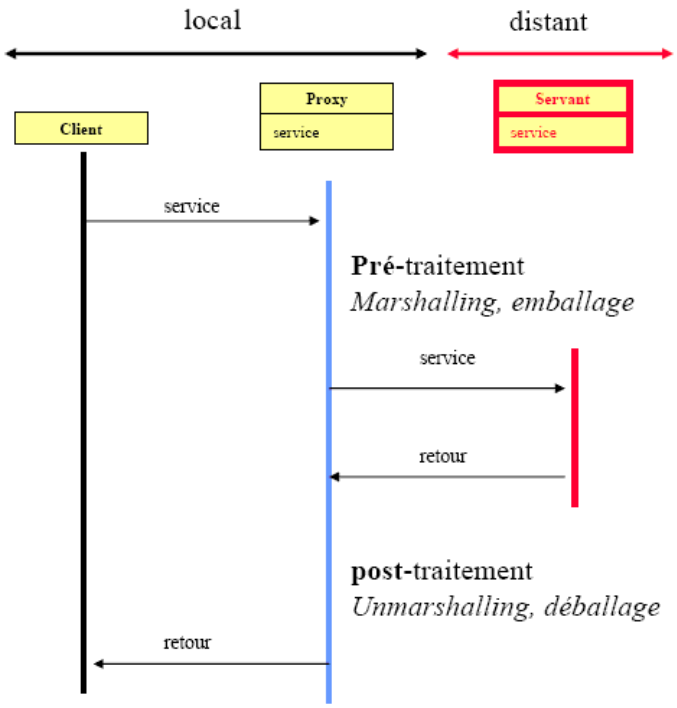



On détaille le Middleware RMI :




La communication se fait entre un proxy et son serveur (appelé "servant" dans les applications distribués)



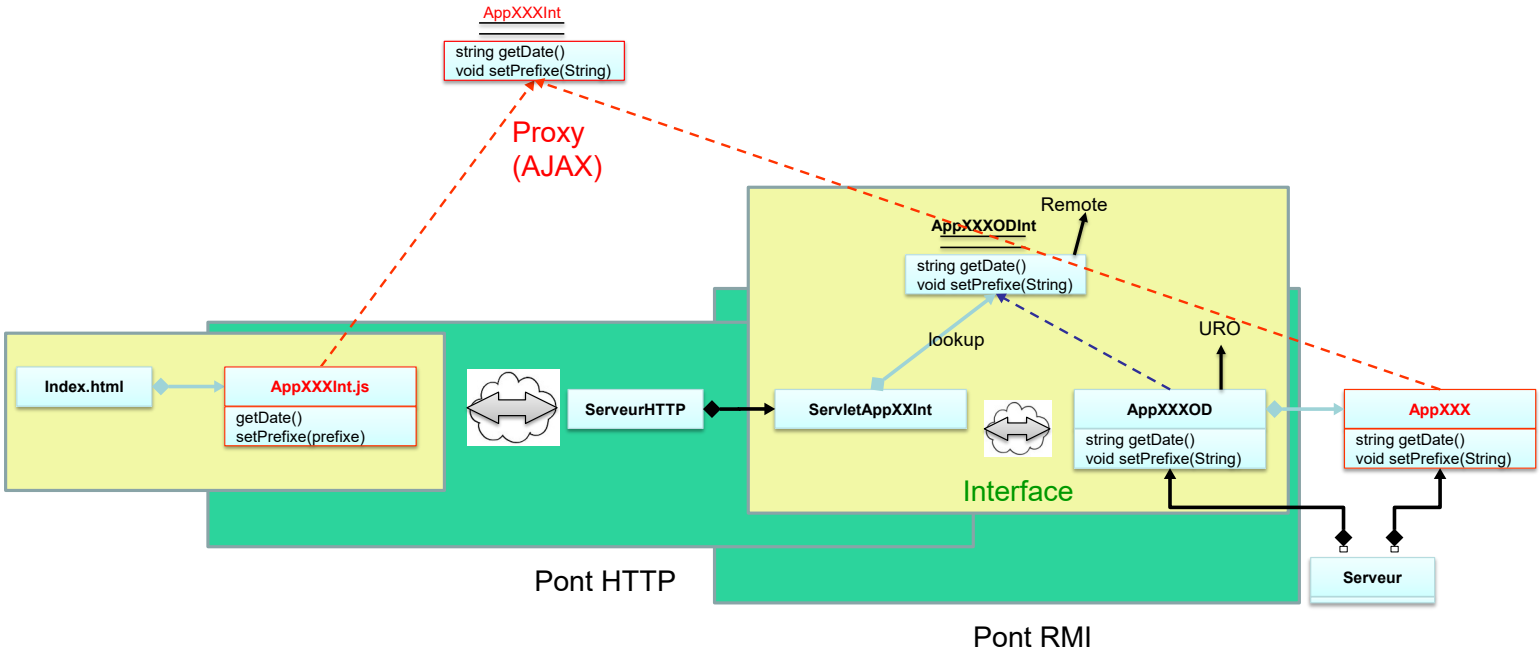




- « Transparent » pour le client

## 3.3. Cas 3 : architecture web-services

On va changer le protocole de communication précédent en remplaçant le proxy IhmXXXRMIImp en un autre (AppXXXInt.js) qui utilise en javascript le protocole : **HTTP**

**Toutes les autres classes ne sont pas modifiées.**




Le code javascript AppXXXInt.js implémente les méthodes de l'interface. Elle les traduit chacune en un appel à un servlet qui est exécuté par le Serveur http.

Les servlets appellent les méthodes distantes de l'objet distribué. (Elles font ce que faisait IhmXXRMIImp).

Nous sommes en face d'une architecture 3 tiers :
- la couche cliente (Vue)
- la couche serveur http (Contrôleur)
- la couche applicative (Modèle)

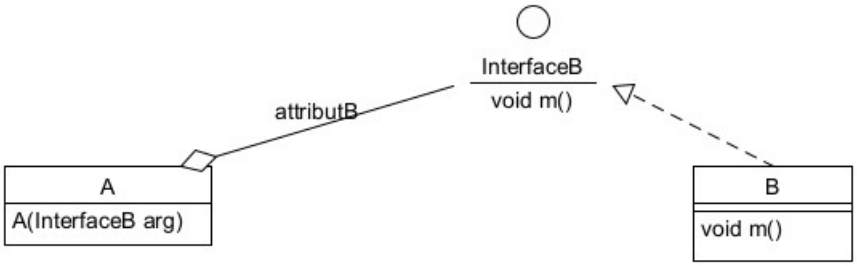
On remarque que la couche applicative est la seule à utiliser le protocole RMI (côté serveur).

Voir le code.

# 4. Conclusion : Rendre distant deux composants

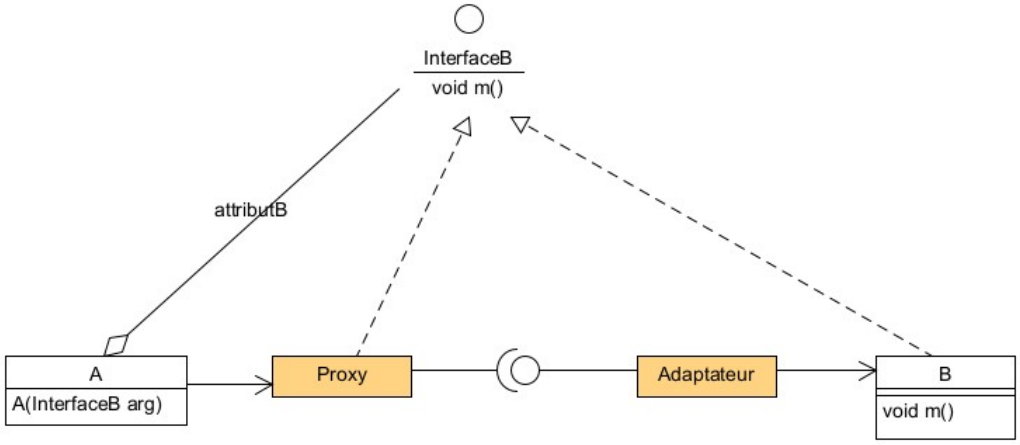
## *4.1. Le Design Pattern de communication*

Soit deux composants A et B dont la dépendance est décrite à travers une interface :




Nous voulons rendre distant la dépendance entre A et B.

La solution : créer 2 composants supplémentaires qui servent de passerelle de communication, un proxy et un adaptateur :




L'Adaptateur est un composant qui permet de rendre distant le composant B.

Ce composant accepte des requêtes de communication qui entraîne l'appel des méthodes de B.

Deux contraintes :

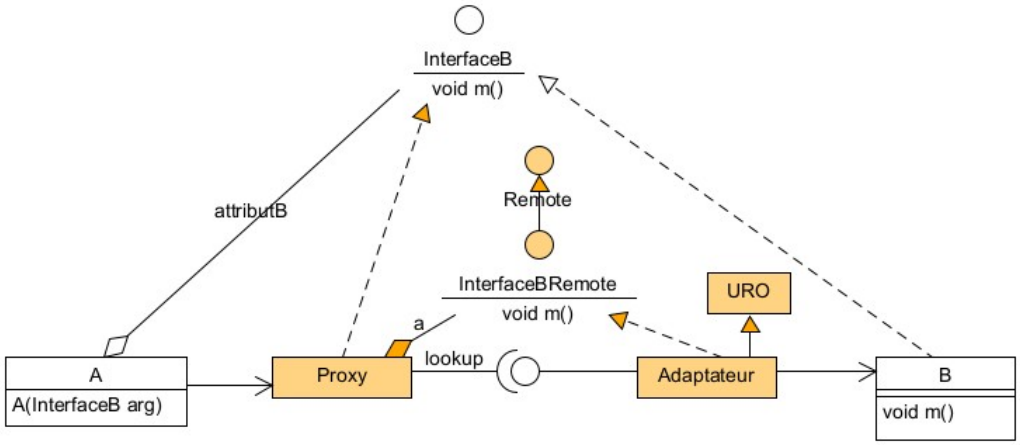
- le protocole de communication utilisé et le format des requêtes doivent permettre le passage des paramètres des méthodes (ex : sérialisation)
- dans le cas d'une communication synchrone, le protocole de communication utilisé doit permettre le retour de l'exécution de la méthode.

Le Proxy est un composant qui se fait passé pour B.

Ce proxy implémente l'interface de B, et pour chaque méthode envoie des requêtes de communication à l'adaptateur.

Ce proxy doit récupérer les moyens de communiquer avec B.

### *4.2. Le Design Pattern en RMI*

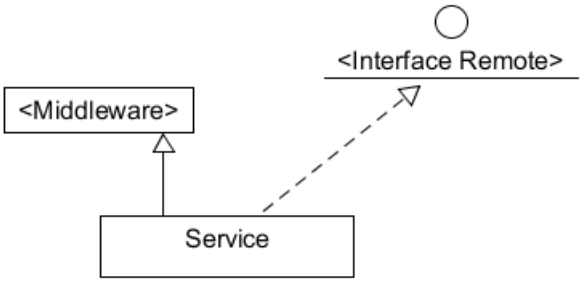



L'Adaptateur est un Objet Distant (DP OD par agrégation ou composition)

Le Proxy (Proxy de Communication) récupère le moyen de communication en faisant un lookup (ou en récupérant le stub par ailleurs), puis pour chaque méthode de l'interface de B réaliser l'appel aux méthodes distantes de l'Adaptateur.

# 5. Conclusion

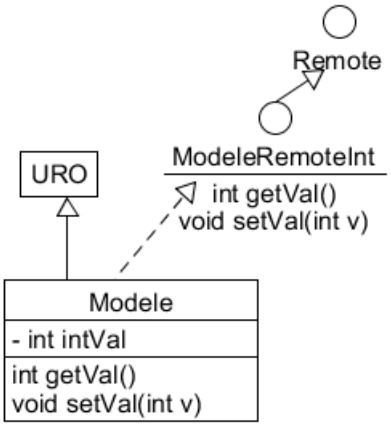
Dans la description UML d'un SI nécessitant de décrire une interface d'un service distant, on peut envisager une notation UML qui est de la forme suivante :




Le Service hérite du middleware pour réaliser son implémentation et implémente une interface distante.

### *5.1. RMI*

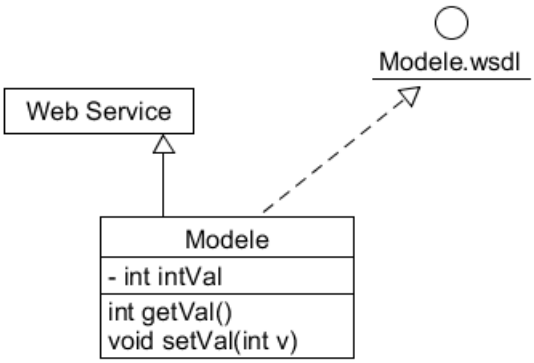
Nous avons vu qu'un Fournisseur de Service en utilisant le middleware RMI, nous avions la représentation suivante :




et toutes les variantes décrites dans précédemment.

## *5.2. WebService*

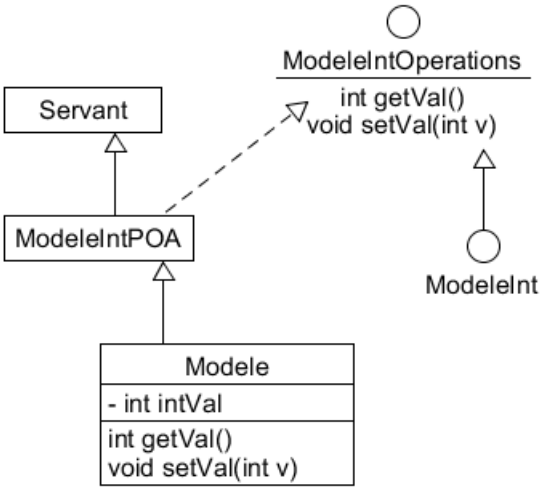
Si dans la description d'un SI, nous voudrions montrer l'utilisation d'un autre middleware que RMI, par un exemple, **un Web Service**, on utiliserait la représentation suivante :




L'interface Modele.wsdl symbolise l'interface WSDL (Web Service Definition Language) qui est un standard de la définition de l'interface d'un Web Service, décrite dans un fichier.

## *5.3. CORBA*

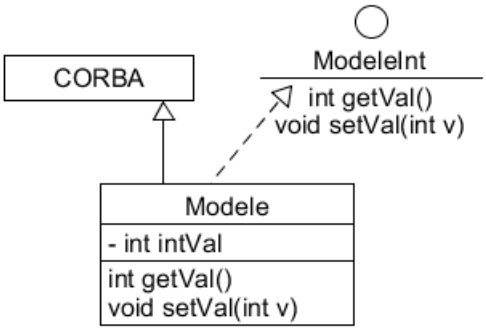
Un autre exemple en **CORBA** :




Cette représentation correspond à la réalité des classes mises en jeu dans l'implémentation d'un service CORBA (voir le cours sur CORBA).

ModeleInt est le nom de l'interface décrite en IDL.

On peut simplifier ainsi :

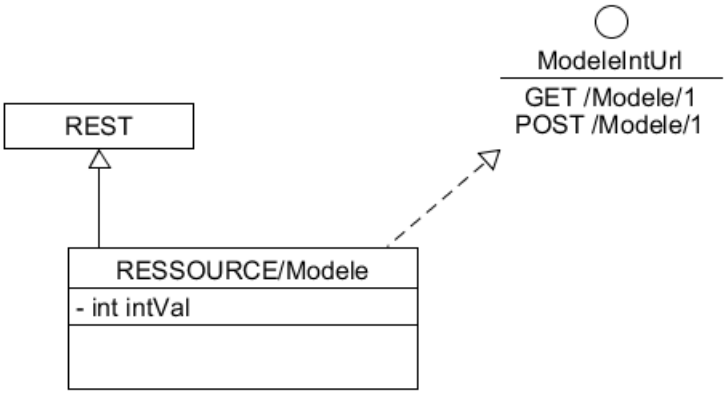



## *5.4. REST*

Dans une architecture REST, on ne parle pas d'accéder à des méthodes d'un objet mais à accéder à une RESSOURCE.

En REST, l'interface est un mapping entre des requêtes URL dont le type de la requête http (GET, HEAD, POST, PUT, DELETE) détermine le type d'action (CRUD) réalisé sur la ressource.

On peut faire la représentation suivante :




L'interface ModelIntUrl est l'interface utilisée par le client et décrit les requêtes Url possibles à faire sur la ressource.