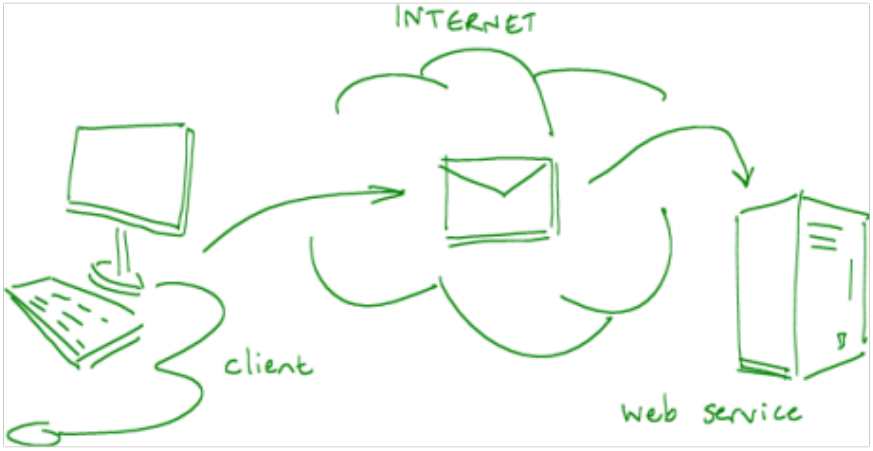
Arnaud Duplat

Mickaël Goletto

Raphaël Lumbroso

Projet web services

Rapport



Université Paris-Dauphine

Année universitaire 2012-2013

Contenu

Introduction

Afin de mettre en œuvre les compétences acquises dans l'enseignement Web Service, il nous est demandé de mettre en place un service de vente en mode Service Web. Dans ce cadre, nous avons réalisé un projet composé de plusieurs applications utilisant des technologies variées : Web Service, RMI, Java, HTML, etc…

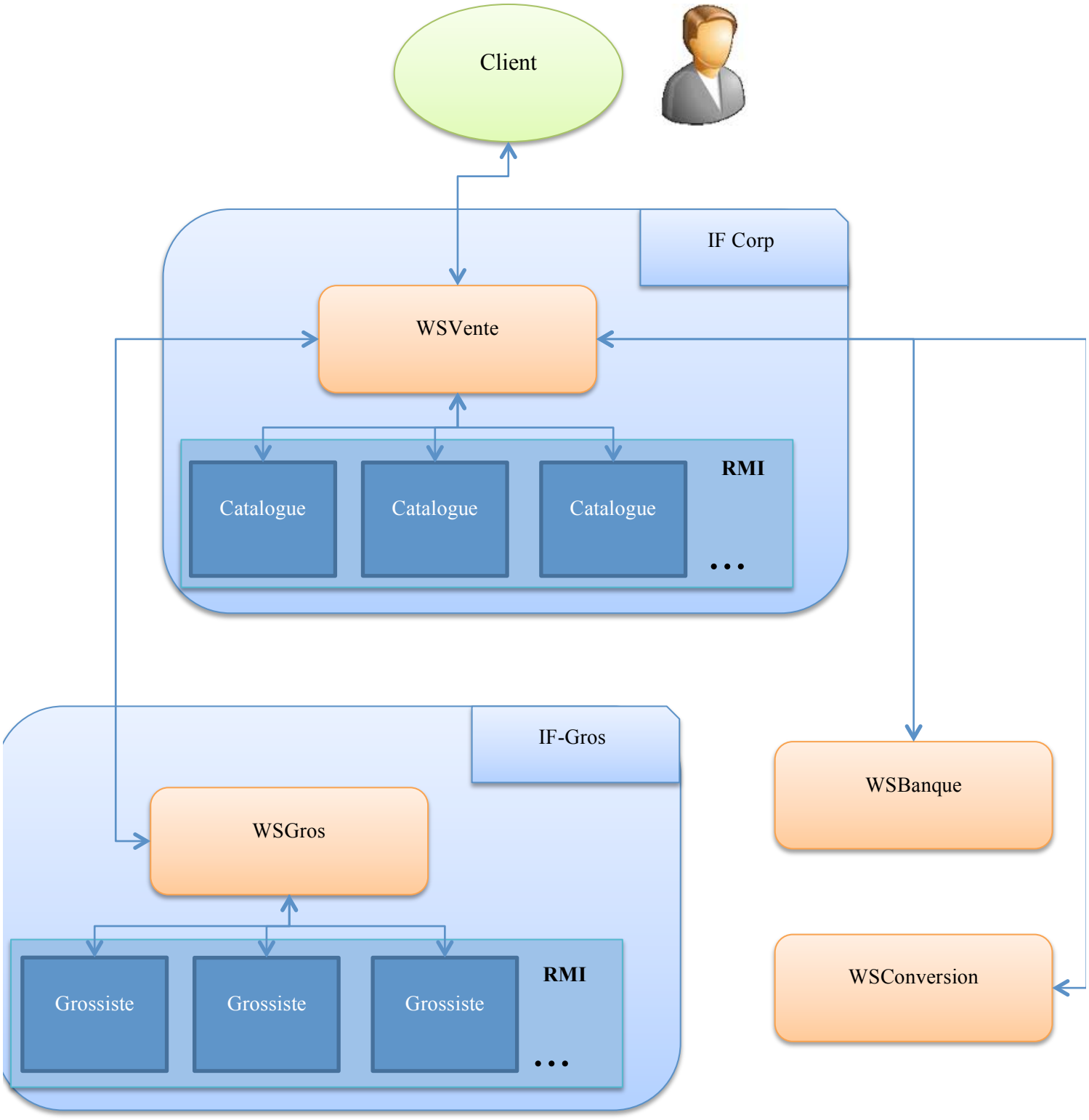
Nous allons détailler dans ce rapport les détails de conception et le modèle de donnée des différents composants du projet. Nous mettrons également en avant les particularités de notre projet et les initiatives prises vis-à-vis du sujet. Dans une dernière partie, nous développerons les impressions, les difficultés et les apports de ce projet d'un point de vue personnel.

NB : Le manuel utilisateur est disponible en annexe du document.

I) Conception générale du projet

1) Fonctionnement général

Voici un schéma représentant le fonctionnement général du projet, avec ses différents composants :



D'une manière générale, le projet fonctionne de la façon suivante :

- Les services d'IF Corp et IF-Gros sont lancés en préalable ; les catalogues sont lancés par RMI et IF Corp initialise sa liste des catalogues. Les catalogues grossistes sont lancés et IF-Gros initialise sa liste des catalogues grossistes.

- L'utilisateur se connecte via une application web développé en HTML. Il peut voir les produits, les ajouter au panier, acheter… Nous verrons la liste de fonctionnalités qui lui sont proposées dans la partie II.

- L'application HTML communique toujours directement avec le service web d'IF Corp, WSVente.
  WsVente se charge de communiquer avec :

  - Les différents catalogues connectés pouvant être distribuées sur des machines différentes, pour toute interaction concernant les produits.

  - Le service web WSGros, qui dispose d'un système de catalogues similaire, pour tout ce qui concerne le réapprovisionnement.

  - Le service web WSBanque, pour vérifier les informations bancaires fournies par le client au moment des achats.

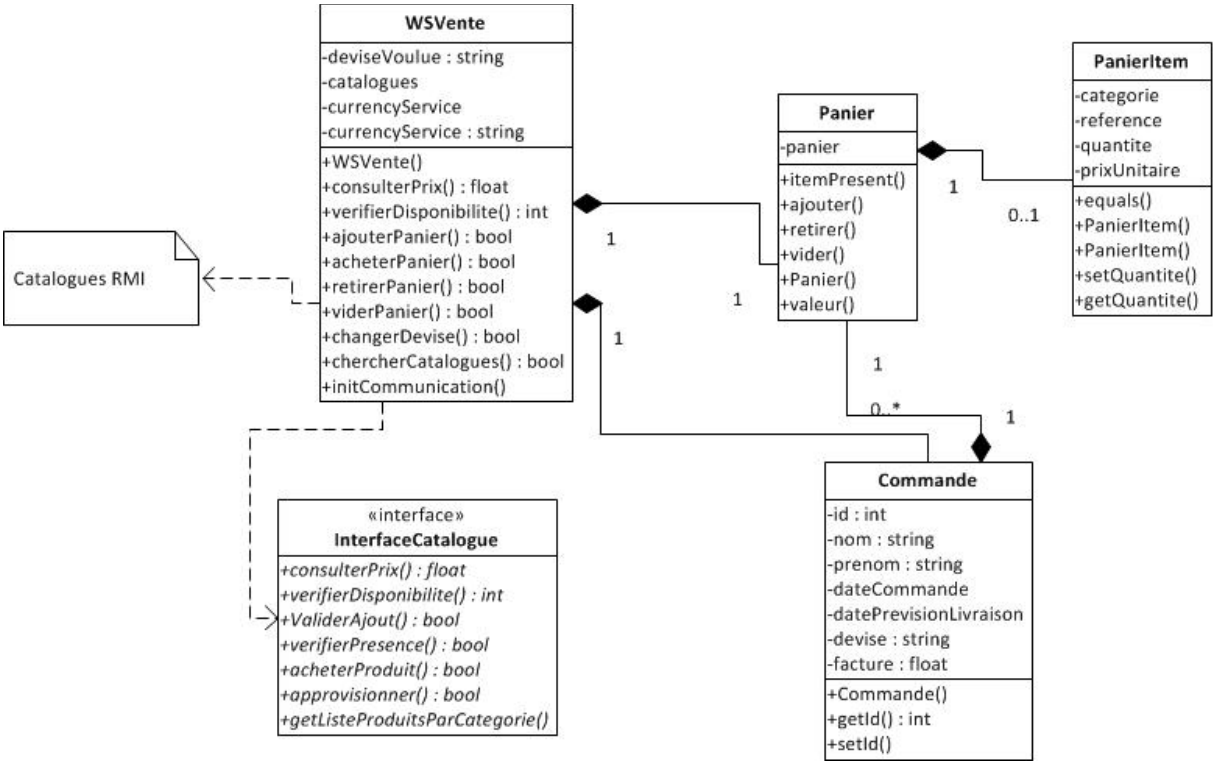
  - Le service de conversion de devises. Ce service n'a pas été développé par nos soins ; nous utiliserons un service web disponible sur internet à l'adresse http://www.currencyserver.de/webservice/currencyserverwebservice.asmx

  - Le client pour lui retourner les informations nécessaires à l'affichage.

Chacun de ces composants du projet va être détaillé dans les parties suivantes de ce rapport.

2) Le web service de IF Corp

a) La modélisation du service



Le service de vente est composé de 5 composants :

- La classe WSVente, point d'entrée unique de l'application web de l'utilisateur vers IF Corp. Cette classe possède en attribut la liste des catalogues récupérés depuis le registry RMI. La gestion des catalogues est détaillée dans la partie *La gestion des catalogues en RMI* du rapport.

- La classe Panier permettant une gestion simple du panier.

- La classe PanierItem permettant de manipuler les articles du panier.

- La classe Commande qui enregistre les achats du client sur le web service.

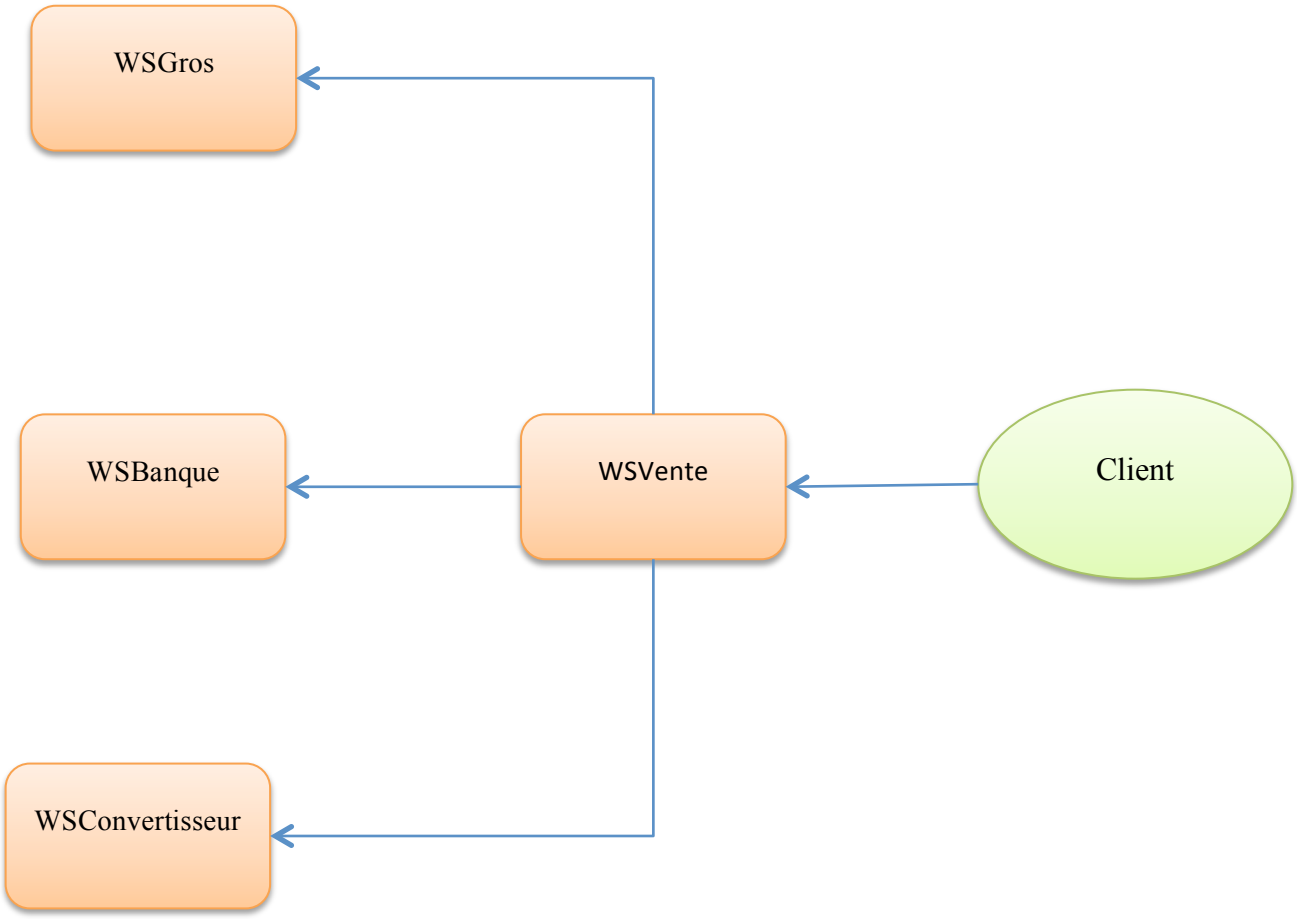
- L'interface InterfaceCatalogue qui permet à WSVente de connaitre l'implémentation des catalogues pour pouvoir dialoguer avec eux.

Aussi, la classe WSVente possède la liste des catalogues connectés sous RMI dans un objet de type Hashtable :

```
private Map<String,CatalogueInterface> catalogues;
```

Cette variable est initialisée lors du passage dans le constructeur de WSVente.

b) Les services web impliqués



Le service web WSVente est appelé par :

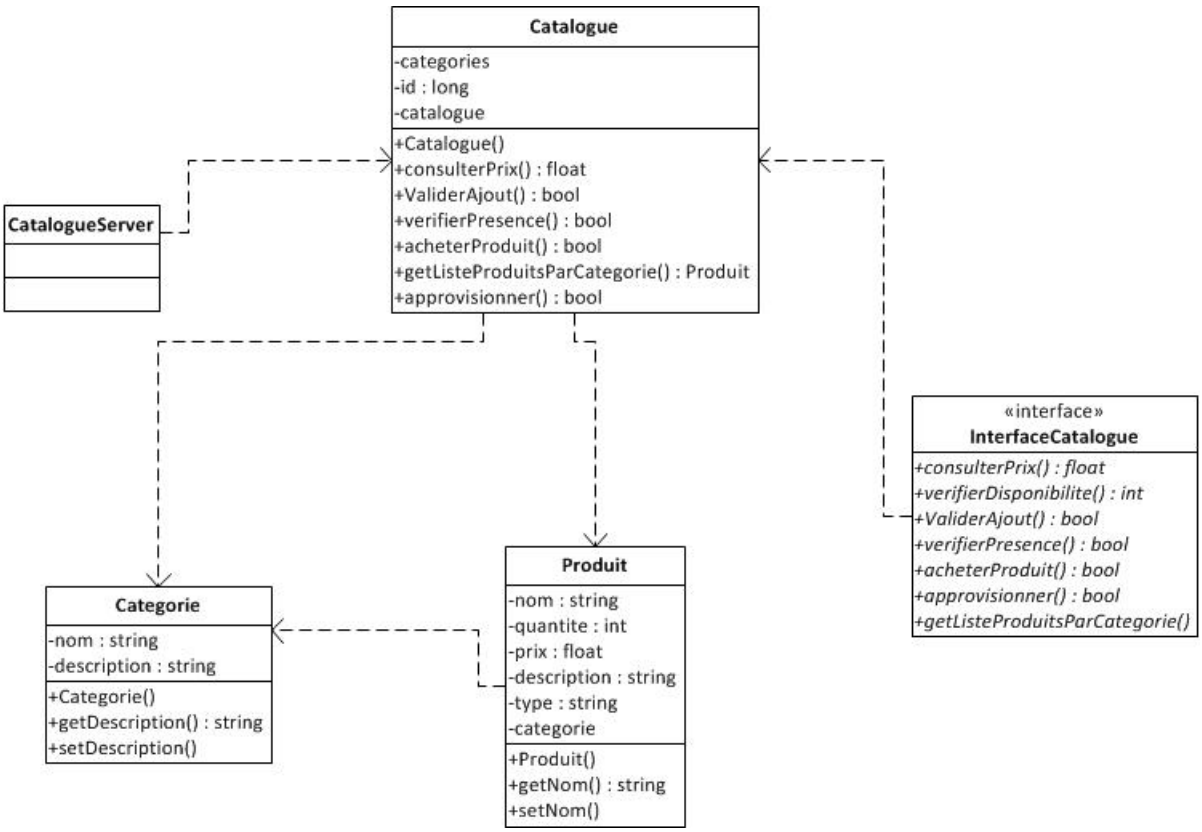
- L'application web du client pour toutes les communications entre le client et le service.

Le service web WSVente appelle :

- Le web service WSGros pour les demandes de réapprovisionnement des catalogues.
- Le web service WSBanque pour la vérification du paiement au moment de l'achat du panier.
- Le web service WSConvertisseur pour toutes les situations ou un changement de devise est nécessaire.

c) La gestion des catalogues en RMI

Les catalogues sont un bloc à part du service web de IF Corp. Ils ont leur propre schéma de donnée :



Les catalogues sont composés de 5 composants :

- La classe Catalogue qui référencie par une Hashtable un ensemble de produits liée à une référence. Ces produits viennent d'un fichier XML, chaque catalogue ayant le sien. A l'initialisation du catalogue, le fichier est lue et la Hashtable est créée. Cette classe contient également l'ensemble des catégories autorisées dans une variable finale et statique.

- L'interface CatalogueInterface qui sera distribuée à l'application WSVente pour permettre la communication entre les deux composants.

- La classe Produit permettant l'instanciation finale des articles du service. Le type d'un produit correspond au fait qu'il appartienne à IF-Corp (type « IF ») ou qu'il soit rajouté par un utilisateur (type « UTILISATEUR ») .

- La classe Catégorie qui permet de mieux spécifier les articles et qui sera notamment utilisé lord de la présentation de l'interface graphique.

- La classe CatalogueServer qui permet de lancer le catalogue et de l'inscrire sur le RMI registry.

Lorsque que l'on lance un Catalogue, celui-ci s'enregistre sur le registry concerné. Cela se passe durant le traitement suivant :

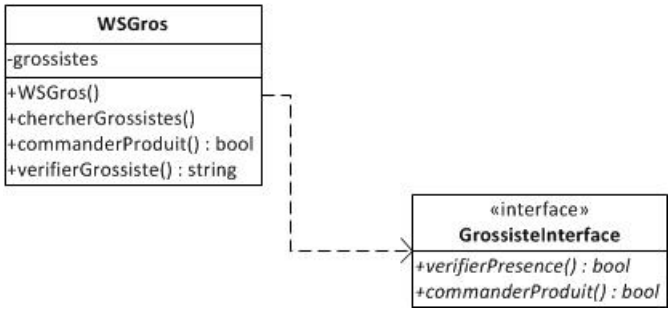
```
Naming.rebind("rmi://localhost:1099/Catalogue_"+((Catalogue)catalogue).getId(), catalogue);
```

On l'enregistre avec le mot clé Catalogue suivi d'un identifiant généré à partir du hashcode du catalogue. On aura ainsi sur le registry (ici 1099), une série de catalogue et leurs identifiants qui seront récupérés par le web service WSVente à son initialisation.

3) Les autres services web

a) Le web service de IF-Gros

Le mode de fonctionnement du service de IF-Gros est semblable à celui de If-Corp ; il est aussi nettement plus simple :



WSGros est composé de 2 composants :

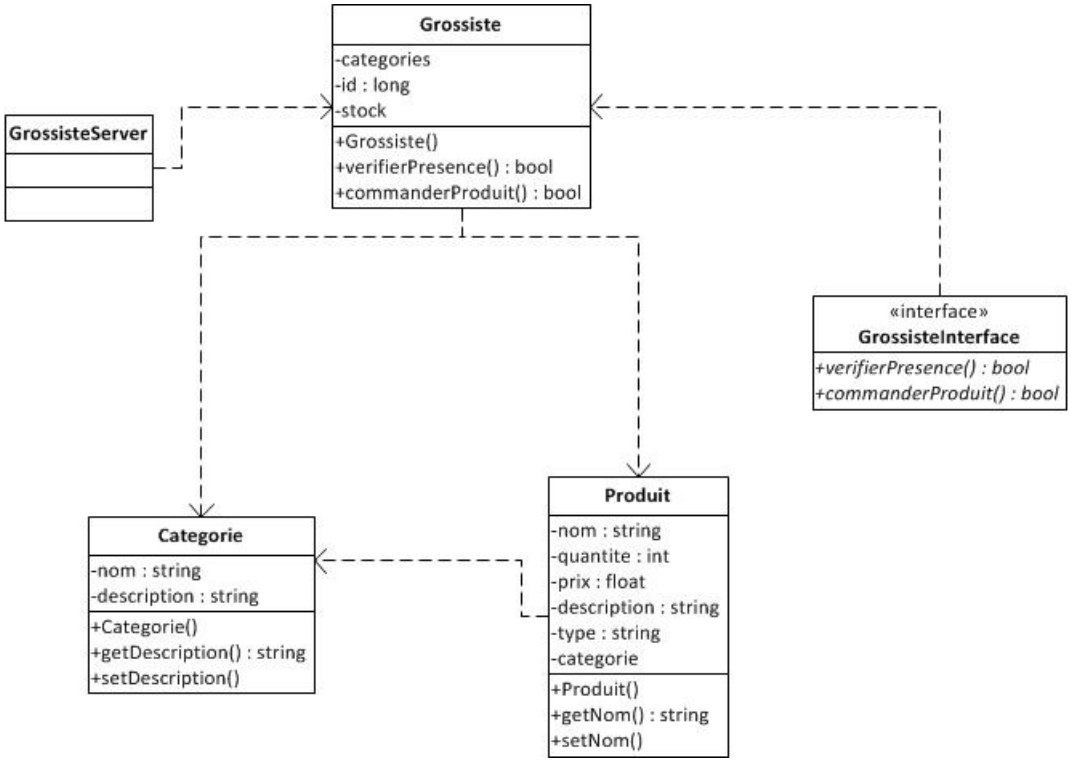
- La classe WSGros, qui est le point d'entrée du service WSVente. Le système de catalogue grossiste est très comparable à celui des catalogues. Les méthodes disponibles sont également semblables.

- L'interface GrossisteInterface, distribuée à WSVente et qui permettra la communication des deux composants.

Le service web WSGros est appelé par :

- Le service web WSVente pour les demandes de réapprovisionnement des catalogues.

Du côté des grossistes en RMI, le fonctionnement et la représentation des données est quasiment équivalente à celles des catalogues :



Les catalogues grossistes sont composés de 5 composants :

- La classe Grossiste qui référencie par une Hashtable un ensemble de produits liée à une référence ; comme le catalogue.

- L'interface GrossisteInterface qui sera distribuée à l'application WSVente pour permettre la communication entre les deux composants.

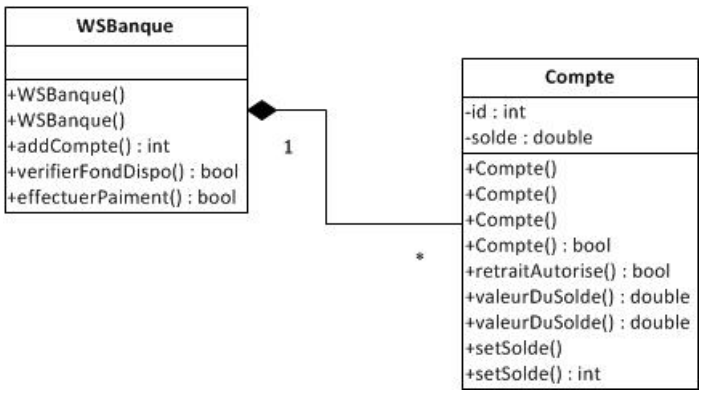
- Les classe Produit et Catégorie, identiques à celles du catalogue.

- La classe GrossisteServer qui permet de lancer le catalogue grossiste.

Le lancement des serveurs grossiste est similaire à celle des catalogues à l'exception que le registry est exécuté sur le port 2099 :

```
Naming.rebind("rmi://localhost:2099/Grossiste_"+((Grossiste)grossiste).getI
d(), grossiste);
```

b) Le service web Banque



Le service web banque est composée de deux classes :

- La classe WSBanque centralisant les appels au web service. Celle-ci est responsable de la gestion des comptes du client et propose les fonctions de vérification des disponibilités des fonds nécessaire pour les achats sur le service Web WSVente.

- La classe Compte, permettant la gestion des fonds de chaque compte client.

A noté que la classe WSBanque stocke les comptes client avec un composant de type NavigableMap :

```
private Map<Integer, Compte> comptes;
```

La gestion de compte est réalisée pour tous les clients en même temps : la banque contenant tous les comptes des clients, il est inutile de l'instancier à chaque appel. On utilisera donc le « scope » à la valeur « application ». Ainsi, ce web service est initialisé une seule fois dans le constructeur de WSVente.

Le service web WSBanque est appelé par :

- Le service web WSVente pour les demandes de vérifications des disponibilités des fonds au moment des achats.

c) L'appel au service web de conversion de devises

Pour la conversion des devises, nous utilisons le service web lié au fichier WSDL suivant : http://www.currencyserver.de/webservice/currencyserverwebservice.asmx?WSDL.

Les appels à ce service extérieur ce font de la manière suivante :

```
currencyService.getCurrencyValue("AVERAGE", "EUR", deviseVoulue));
```

La notion « AVERAGE » signifie que l'on prend le taux moyen entre 2 taux de conversions légèrement différents.

Dans ce projet, nous avons décidé que la gestion de la conversion des devises est entièrement centralisée par le web service WSVente. Ainsi, seul celui-ci fait les appels extérieurs au convertisseur. De leur côté, les composants des catalogues et des grossistes traitent toujours les sommes d'argents sous forme d'euros (€). Ces sommes transitant toujours par le service web WSVente, celui-ci se charge de convertir ces données avant de les renvoyer sous la bonne devise vers l'utilisateur.

Nous avons également décidé, pour des raisons d'ergonomies de ne pas proposer l'ensemble des devises à l'utilisateur. Seule une dizaine des devises les plus courantes sont proposées pour permettre un affichage du choix de la devise sous forme de liste suffisamment courte.

4) Le client Html

a) Détails de conception

Nous avons opté pour une interface utilisation les technologies suivantes : PHP, HTML et CSS. À travers ce choix, nous avons essayé de montrer l'interopérabilité entre les différents langages de programmation grâce à l'utilisation de WebService.

Pour cela, nous disposons aussi d'un serveur web (LAMP dans notre cas) qui se connecte au WebService une fois que nous lui avons du renseigner l'adresse du WSDL. Celui-ci effectue des appels simples aux WebService grâce à la classe SoapClient de PHP.

Un des points sensibles du développement de l'interface a été le maintien de la session de l'utilisateur vis-à-vis du web service. En effet, lorsque l'on initialise le scope en mode session, le web service fourni un ID à l'application cliente dès le premier échange. Cependant, il est nécessaire de conserver cet ID et le renvoyer à chaque connexion pour être identifié par le service web. Nous avons donc implémenté une fonction :

```
public void initCommunication(){
}
```

Celle-ci est exécuté dès la création du WebService et va permettre de conserver l'id afin qu'il soit communiqué par le client pour les futurs échanges. Une fois ce premier échange effectue, nous analysons la réponse HTTP pour en extraire l'id (nommé JSESSIONID) et nous le stockons dans une variable session toujours du côté client. Grâce à cela nous pouvons conserver un id unique même en cas de changement ou rechargement de page. Pour que cet id soit correctement transmis au webservice nous avons dû utiliser la méthode suivante :

```
$service = new SoapClient($wsdl, array('trace' => 1));

$service->__setCookie('JSESSIONID', $_SESSION['soapCookieJSESSIONID']);
```

Cette méthode est propre à l'utilisation de la classe SoapClient, elle permet qu'à chacun des échanges effectués cet id soit transmis.

b) Fonctionnalités proposées

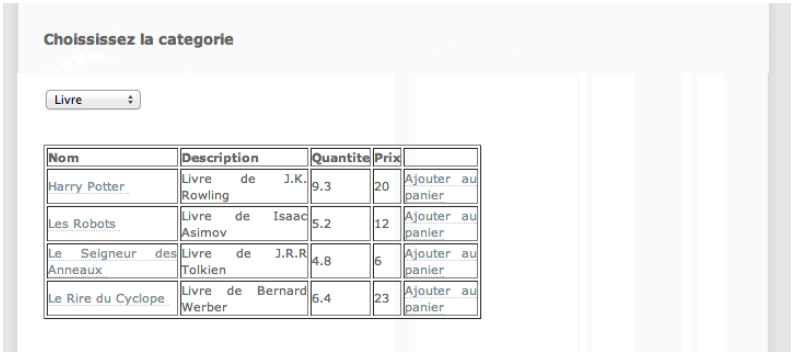
À travers cette interface graphique, nous offrons la possibilité à l'utilisateur de lister les produits avec des filtres sur la catégorie et le type si besoin. La gestion du panier est aussi effectuée, l'utilisateur peut ajouter, supprimer des éléments du panier au bien modifier la quantité.

Voici la liste des principales fonctionnalités :

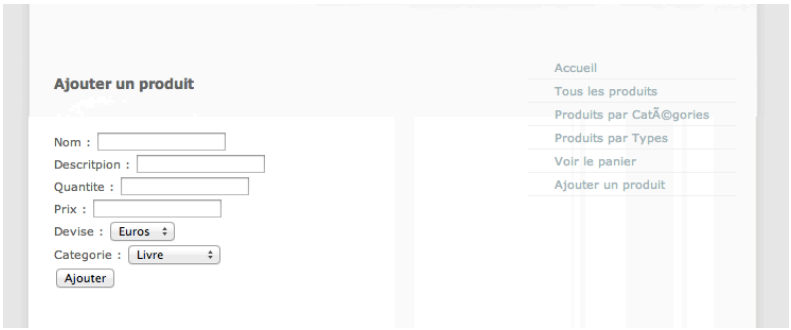
- Afficher un filtre par catégorie ou par type
- Afficher tous les produits
- Ajouter, modifier ou supprimer un élément du panier
- Changer la devise du panier, ou du produit afficher
- Etre averti d'un de la de livraison
- Gestion des erreurs lors de l'ajout d'un produit
- Ajout d'un produit

Voici quelques captures d'écrans de la vue de l'utilisateur final sur l'interface client.

Page de sélection



Filtre sur les Types



Ajout d'un produit

II) Nos innovations

1) Une architecture catalogue réellement dynamique en RMI

Une réflexion particulière a été portée sur la conception et l'architecture du fonctionnement catalogues vis-à-vis de leur partage grâce à RMI.

Lors de nos premières lectures du sujet, nous avons imaginé une solution dans laquelle chaque ordinateur se connectant à RMI apporte un nouveau type de catégorie d'objets. Cette solution était facile à implémenter puisqu'il suffit pour le service web central de contacter la bonne machine en fonction de la catégorie de l'objet. Cependant, après relecture du sujet, nous avons compris que les machines et les catégories ne sont pas nécessairement liées. Une nouvelle machine doit pouvoir apporter de nouveaux « livres » même si des « livres » sont déjà présents sur une machine RMI.

Nous avons donc organisé notre architecture de manière à ce que :

- Le service web central (que ce soit WSVente pour les catalogues ou WSGros pour le grossiste) puisse demander à chaque catalogue (ou grossiste) s'il possède la référence d'un produit de manière à ce que le catalogue et la catégorie soit totalement dissociée.

- Chaque catalogue s'initialise sur le registry d'une même machine.

- Lors de l'initialisation du service web central, celui-ci interroge le RMI registry correspondant pour obtenir la liste des catalogues qui sont connectés. Ainsi, on peut enregistrer le nombre de catalogues que l'on souhaite.
  Ceci se fait par la fonction : `private void chercherCatalogues(String[] args)` de la classe WSVente qui extrait du registry la liste des objets et les références dans un attribut *catalogues* de la classe. Cet objet est une Map<String, CatalogueInterface>, la clé de cette Map étant l'adresse de binding d'un objet catalogue.

- Avec cette architecture, la mise à jour de la liste des articles pour l'application n'est pas instantanée. Cependant, elle ne crée pas de bug. Les catalogues sont remis à jour du point de vue client à chaque fois que sa session est réinitialisée (lors d'une nouvelle instanciation de la classe WSVente). Cela sera donc transparent pour lui et ne lui posera pas de problème. Cela permet une totale flexibilité pour l'ajout d'autres catalogues côté serveur.

Ainsi, la gestion des catalogues et des grossistes est entièrement indépendante des autres composants et l'on peut à tout moment ajouter ou enlever un catalogue ou un grossiste sans dommage pour l'intégrité du système.

Par rapport à l'architecture prévue initialement, les temps d'exécution seront plus longs car on passe par une recherche de références dans les catalogues pour savoir sur quel catalogue se trouve le produit que l'on recherche lorsqu'on traite des objets. Cependant, l'adaptabilité du service est maintenant effective.

2) Une utilisation du XML

a) Principale utilisation

Durant ce projet, nous avons décidé de stocker les données (produits d'un catalogue, produit d'un grossiste, comptes de la banque) dans fichiers XML. Ce choix a été fait car c'est un format de données facile à utiliser et qui permet une meilleure flexibilité dans l'ajout de données qu'une base de données. De plus, cela permet de lier à chaque objet ses propres données.

Il nous a paru plus judicieux d'utiliser ce type de représentation des données plutôt que de mettre des produits en dur dans le code Java car, comme décrit ci-dessus avec notre architecture dynamique, afin de rajouter un catalogue par exemple, il suffit simplement de dupliquer un dossier « Catalogue » avec toutes les classes compilées et de remplacer le fichier XML présent par son exemplaires des données.

b) Partie technique

Afin de *parser* les fichiers XML dans les constructeurs de *Catalogue.java*, *Grossiste.java* et *WSBanque.java*, nous avons utilisé l'API 'DOM' fournit avec Java. Il s'agit de représenter le fichier XML en entier sous forme d'un arbre et de le parcourir depuis la racine, ou par nom de balise par exemple.

```
NodeList comptesNode = doc.getElementsByTagName("compte");

for(int i=0; i<comptesNode.getLength(); i++){
        Element compteNode = (Element) comptesNode.item(i);
        int numeroCompte = Integer.parseInt(compteNode.getAttribute("numeroCompte"));
        String passwordCompte = compteNode.getAttribute("password");
        double solde = Double.parseDouble(compteNode.getElementsByTagName("solde").item(0).
getTextContent());

        Compte compte = new Compte(numeroCompte, passwordCompte, solde);
        comptes.put(compte.getNumeroCompte(), compte);
}
```

Par exemple ici, on récupère la liste de tous les nœuds dont le nom de la balise est 'compte' puis pour chacune d'elle, on récupère les différentes sous-balises pour créer finalement un objet Compte que l'on stocke ensuite dans la Map globale `comptes`.

Chaque type de fichier XML est associé au fichier de définition DTD qui lui correspond afin de cadrer et de donner une structure unique au fichier XML. (*catalogue.xml* + *catalogue.dtd*, *stock.xml* + *stock.dtd*, *compte.xml* + *compte.dtd*).

3) Des fonctionnalités clientes poussées

Une attention particulière a été portée aux interactions entre le client et le web service de vente.

Nous avons décidé d'implémenter un panier, paradigme en vogue dans tous les sites internet de commerce actuel. Le client peut interagir avec celui-ci pour en ajouter les articles dans son panier, en retirer, vider le panier ou acheter le panier. Ceci est rendu possible par le fait que l'on est en « session state» sur notre service web : les informations sont traitées client par client.

De plus, nous ne passons pas par un système de session interne à notre logiciel. Le but est ici de faire profiter à l'utilisateur de l'avantage des services web : pouvoir l'utiliser directement sans contrainte. Ainsi, l'utilisateur peut directement à son arrivée sur le site consulter les produits, remplir un panier. Ses informations personnelles ne sont demandées seulement lors du payement.

III) Retours d'expérience

1) Les apports

Ce projet a été pour certains d'entre nous l'occasion de mettre en place pour la première fois les notions de services web dans un projet conséquent. Ce premier exercice nous aura permis de mettre en pratique et de consolider nos connaissances, notamment sur les architectures possibles d'application centrées autour des services web et la façon dont plusieurs services peuvent s'entre-appeler pour fournir un service spécifique.

Concernant la partie RMI, nous avions tous eu une première expérience universitaire l'année dernière. Cependant, celle-ci a été variée et nous a donc permis d'améliorer nos connaissances sur le domaine. Par exemple, le fait d'avoir plusieurs types d'objets repartis (catalogues et grossistes) et le fait d'avoir à récupérer et travailler dynamiquement sur les objets du registry a été pour nous une expérience nouvelle et formatrice.

Enfin, la mise en place d'une interface graphique nous a permis de bien comprendre les possibilités d'utilisation concrètes de l'utilisation de web services sur les applications web, ainsi que la possibilité d'utiliser des technologies différentes pour le serveur et pour le client, consommateur du web service. Les tests en environnements réels sont plus explicites que sur l'environnement de test d'Eclipse.

2) Les difficultés rencontrées

Au cours du développement du web service, nous avons dû faire face à plusieurs problèmes. Voici une liste des principaux et les solutions trouvées :

- Comme nous l'avons déjà évoqué, l'architecture technique des catalogues RMI a dû évoluer durant le projet, suivant notre meilleure compréhension du projet. Nous étions partis sur une solution où chaque catalogue était une catégorie, ce qui était simple à implémenter et avons dû en cours de développement revoir la conception pour que les catégories et les catalogues soient indépendants.
  La refonte du code déjà existant est toujours quelque chose de difficile mais nous avons pu arriver à une solution satisfaisante.

- Le choix de modélisation des données a été assez complexe. Dans un souci de ne pas alourdir le projet et de mettre d'avantage en avant les aspects services web et RMI, nous avons décidé de ne pas intégrer de bases de données en tant que telles dans le projet. Nous avons donc dû trouver des solutions alternatives, que ce soit par des attributs de classes ou par l'utilisation assez poussée des fichiers XML. La difficulté étant d'aboutir à une modélisation des données cohérentes. Nous sommes toutefois satisfaits de cette opportunité d'utiliser le

XML, que nous avons pu apprendre au cours de ce projet ainsi que dans d'autres enseignements au cours de l'année.

- L'interface utilisateur a également été un point sensible de notre développement. La partie critique du projet étant l'architecture web service et RMI, la partie d'interface n'a pas été en premier lieu une priorité pour nous. Cependant, la création d'interface ne peut se faire en demi-teinte ; il a donc fallu se reconcentrer sur ce point-là en fin de projet, quitte à ne pas développer toute nos idées de service web pour pouvoir proposer une interface correcte.

- Nous avons expérimenté les désagréments de l'utilisation de services web externe à notre projet. En effet, le service de conversion initialement choisie (www.webservicex.com/currencyconvertor.asmx?WSDL) est devenu instable en fin de projet. Le service marchait avec alternance, et nous ne comprenions pas forcément d'où venait le problème. Nous avons donc dû changer de service en cours de projet et nous diriger vers http://www.currencyserver.de/webservice/currencyserverwebservice.asmx?WSDL.

- Comme déjà mentionné dans la partie de l'interface HTML, nous avons rencontré des problèmes majeurs dans la conservation de la session utilisateur sur le service web. Nous avons pour corriger cela dû implémenter une fonction d'initialisation de l'ID session et conserver cet ID sur du côté de l'application HTML pour qu'il soit renvoyé à chaque requête.

3) Les axes d'amélioration

Plusieurs points de ce projet peuvent encore être amélioré ou optimisés avec un délai supplémentaire. Voici la liste de nos principales idées :

- L'intégration d'une gestion de base de données pour les clients afin de gérer les données des utilisateurs plus en profondeurs. On pourrait par exemple stocker définitivement l'historique des achats sur le serveur, les articles qu'il a ajouté personnellement dans les catalogues, etc…

- Eventuellement la possibilité de suggérer des articles du catalogue en fonction des achats précédents.

- Un développement visuel et ergonomique plus poussé de l'interface. L'interface développée permet de mettre en œuvre les fonctionnalités développées mais peut-être améliorer avec un temps de développement plus conséquents ; cette activité étant particulièrement chronophage.

Conclusion

Notre impression concernant la réalisation de ce projet est plutôt bonnes ; nous sommes parvenus à implémenter les fonctionnalités que nous souhaitions, nous avons trouvé des solutions satisfaisantes pour les problèmes que nous avons rencontrés. Nous sommes particulièrement satisfaits par notre gestion dynamique des catalogues et la mise en place d'une interface graphique crédible pour un utilisateur.

Un des grands avantages de ce projet aura été pour nous l'occasion de manipuler en parallèle trois technologies complémentaire : les web services, RMI et l'interface HTML. Les compétences acquises ont donc été bien plus complètes que sur un projet classique portant sur une seule technologie.

Nul doute que les compétences acquises nous servirons dans nos futurs parcours professionnels.

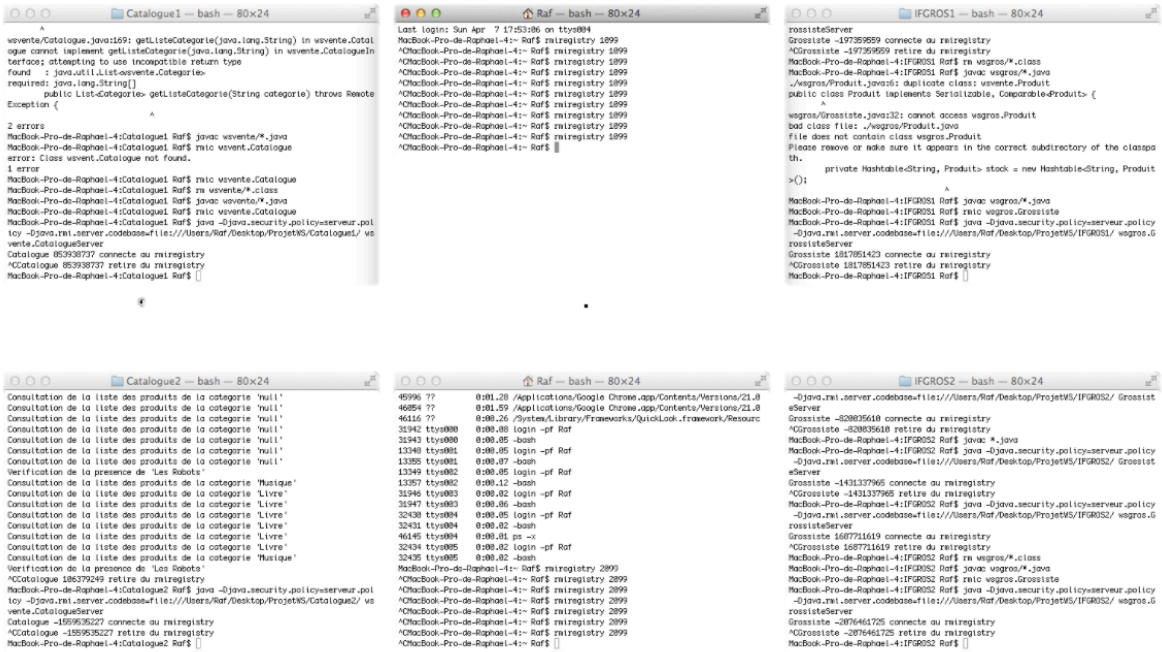
Annexe : Le manuel utilisateur

Pour ce projet, la mise en place de l'ensemble des web services a été assez compliquée, surtout le fait de lier RMI et web services générés avec Eclipse.

Voici la démarche à suivre afin de faire fonctionner notre projet.

1) Démarrer autant de terminaux que de Catalogue et de Grossistes différents.

2) Démarrer 2 autres terminaux en plus chargé d'héberger les registry sur les différents ports (1099 et 2099).



3) Pour les catalogues, dans le dossier 'Catalogue', taper *javac wsvente/*.java*. Puis ensuite taper *rmic wsvente.Catalogue* afin de générer le stub.

4) De la même manière pour les grossistes, dans le dossier 'IFGROS', taper *javac wsgros/*.java* puis *rmic wsgros.Grossiste*

5) Dans une des deux fenêtres du registry, taper *rmiregistry 1099* et dans l'autre *rmiregistry 2099*.

Ensuite pour lancer les serveurs, il faut taper *java -Djava.security.policy=serveur.policy -Djava.rmi.server.codebase=file:///Users/Raf/Desktop/ProjetWS/Catalogue1/ wsvente.CatalogueServer* pour les Catalogues et *java -Djava.security.policy=serveur.policy -Djava.rmi.server.codebase=file:///Users/Raf/Desktop/ProjetWS/IFGROS1/ wsgros.GrossisteServer* pour les Grossistes

C’est terminé pour la partie RMI.

Il suffit ensuite de lancer les 3 projets Java ‘WSGros’, ‘WSBanque’ et ‘WSVenterServeur’ sur un serveur Tomcat sur Eclipse afin de lancer les 3 web services que l’on a décrit plus haut et d’avoir accès à :

http://localhost:8080/WSGros/services/WSGros?wsdl

http://localhost:8080/WSBanque/services/WSBanque?wsdl

http://localhost:8080/WSVenteServeur/services/WSVente?wsdl