Construction d'une application MVC distribuée avec Spring Remoting

Partie 1 : HttpInvoker, Hessian, Burlap, RMI

serge.tahe@istia.univ-angers.fr, juillet 2005

1 Introduction

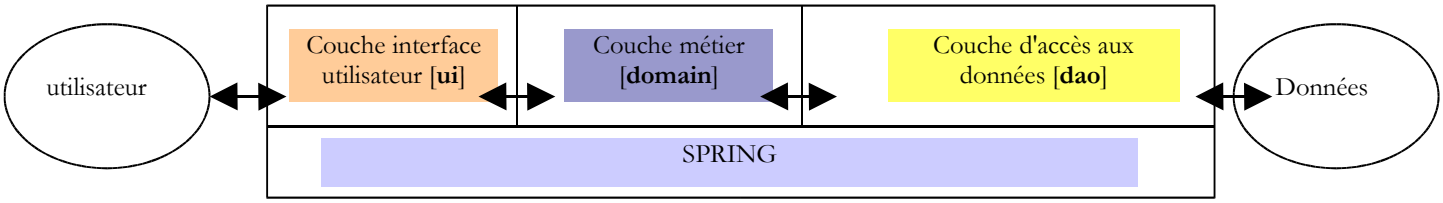
Nous poursuivons ici les articles :

1. [Variations autour d'une application web à trois couches], disponible à l'url [http://tahe.developpez.com/java/web3tier]. Nous le nommerons par la suite [article1]. Cet article présentait une application simplifiée d'achats de produits sur le web. Son architecture MVC était implémentée de trois façons différentes :
   - avec une servlet contrôleur et des pages JSP pour les vues
   - avec le framework [**Struts**]
   - avec le framework [**Spring MVC**]

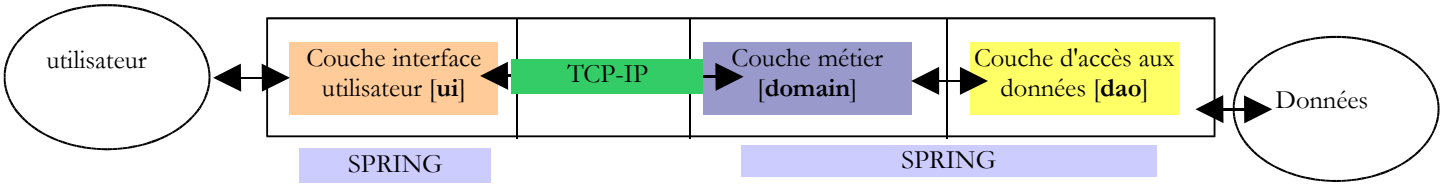
2. [M2VC - un moteur MVC pour les applications swing], disponible à l'url [http://tahe.developpez.com/java/m2vc]. Nous le nommerons par la suite [article2]. [M2VC] est un framework MVC pour des applications Swing inspiré de [Struts]. M2VC signifie **M**oteur **MVC**. On peut utiliser M2VC lorsqu'on veut donner une architecture MVC à une application swing.

3. [Construction d'une application swing MVC à trois couches avec Spring], disponible à l'url [http://tahe.developpez.com/java/swing3tier/]. Nous le nommerons par la suite [article3]. Cet article reprenait l'application web de [article1] pour en faire une application swing "standalone". L'architecture MVC initiale de l'application web était reproduite grâce au moteur M2VC décrit dans [article2].

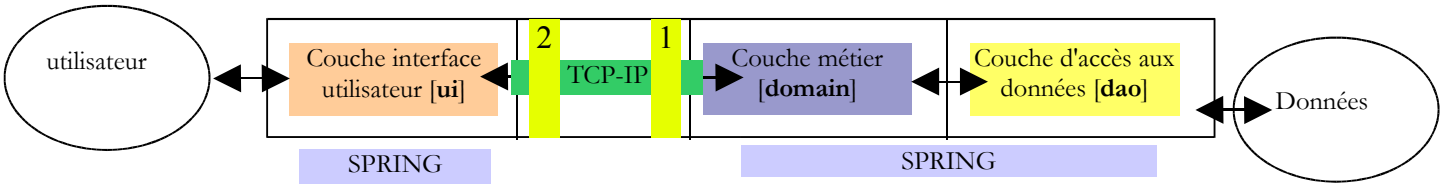
Le présent article reprend l'application swing développée dans [article3] pour en faire une application client-serveur. L'application [swingarticles] développée dans [article3] avait l'architecture suivante :



L'application que nous allons développer ici et que nous appellerons désormais [magasin-client-serveur] aura, elle, l'architecture client-serveur suivante :



Nous voulons réutiliser autant que possible les couches [ui, domain, dao] développées dans les articles précédents. L'idéal serait qu'on utilise les archives de ces couches sans toucher au code source. Il nous faut, pour cela, créer des couches d'adaptation entre le client et le serveur. L'architecture précédente devient alors la suivante :



Nous ajoutons deux couches notées ci-dessus 1 et 2 :

- la couche 1 va exposer un service distant d'accès à la couche [domain]. Cette couche a pour rôle de cacher le réseau à la couche [domain] qui ne saura pas qu'elle est utilisée par des clients distants. Nous l'appellerons couche [proxyServeur].
- la couche 2 va implémenter les proxy d'accès aux services web de la couche 1, de façon transparente pour la couche [ui] qui ne saura pas que les données qu'elle reçoit et envoie vont sur le réseau. Nous l'appellerons couche [proxyClient].

Un article analogue a été écrit pour le monde [dotnet] et est disponible à l'url [http://tahe.developpez.com/dotnet/web3tier-part3]. Le présent document s'en inspire seulement en partie. La version [dotnet] ne présentait qu'une seule implémentation de la couche

[proxyServeur], celle d'un service web SOAP. Ici, cette implémentation n'est pas présentée. Elle le sera dans un futur article. Nous présentons trois technologies différentes pour l'implémentation du service distant [proxyServeur] :

- Spring RMI
- Spring HttpInvoker
- Spring Hessian
- Spring Burlap

Outils utilisés :

- **Eclipse 3.02** pour le développement des applications Java, disponible à l'url [http://www.eclipse.org/downloads/index.php].
- **Plugin Eclipse** : Sysdeo Tomcat pour développer des applications web avec le conteneur de servlets Tomcat, disponible à l'url [http://www.sysdeo.com/eclipse/tomcatplugin].
- **Spring 1.2**, notamment **Spring IoC** pour l'instanciation des objets nécessaires à l'architecture n-tier de l'application ainsi que **Spring Remoting** pour l'implémentation du dialogue client-serveur. Spring est disponible aux url [http://www.springframework.org/download] et [http://prdownloads.sourceforge.net/springframework/spring-framework-1.2.2-with-dependencies.zip?download]. On prendra la version de [Spring] avec dépendances car [Spring] utilise de nombreux outils tiers dont les archives sont contenues dans la version "avec dépendances".
- **Tomcat 5** comme conteneur de servlets, disponible à l'url [http://jakarta.apache.org/site/downloads/downloads_tomcat-5.cgi].
- **Ibatis SqlMap** pour la couche d'accès aux données du SGBD disponible à l'url [http://ibatis.apache.org/downloads.html]
- le moteur **M2VC** disponible à l'url [http://tahe.developpez.com/java/m2vc]
- une base de données avec un pilote JDBC ou un pilote ODBC. L'exemple livré avec cet article contient une base ACCESS accédé via un pilote JDBC-ODBC parce que la plupart des lecteurs de cet article disposeront probablement du SGBD ACCESS. Ceci dit, toute base de données avec un pilote JDBC ou un pilote ODBC fait l'affaire. Un fichier de configuration permet d'ajuster ce point.

Tous ces outils sont gratuits, excepté ACCESS.

Dans une échelle [débutant-intermédiaire-avancé], ce document est plutôt dans la partie [avancé]. Sa compréhension nécessite divers pré-requis. Certains d'entre-eux peuvent être acquis avec des documents que j'ai écrits. Dans ce cas, je les cite. Il est bien évident que ce n'est qu'une suggestion et que le lecteur peut utiliser ses documents favoris.

- [**article1**] - cité plus haut
- [**article2**] - cité plus haut
- [**article3**] - cité plus haut
- utilisation de l'aspect IoC de **Spring** : [http://tahe.developpez.com/java/springioc]
- utilisation d'**Eclipse** avec **Tomcat 5** : [ftp://ftp2.developpez.biz/developpo/users/tahe/fichiers/progwebjavaavececlipseettomcat.pdf]. Ce document présente la version de Tomcat 5.0. Celle actuellement disponible est la 5.5.9 (juillet 2005). Dans cette version, le lien [Tomcat administration] de la page d'accueil [http://localhost:8080] de Tomcat, ne fonctionne plus comme dans la version 5.0. En-dehors de ce point, les explications du document précédent restent valides. Le présent article n'utilise pas le lien [Tomcat administration].
- documentation **Ibatis SqlMap** : [http://ibatis.apache.org/downloads.html]
- documentation **Spring** : [http://www.springframework.org/documentation]

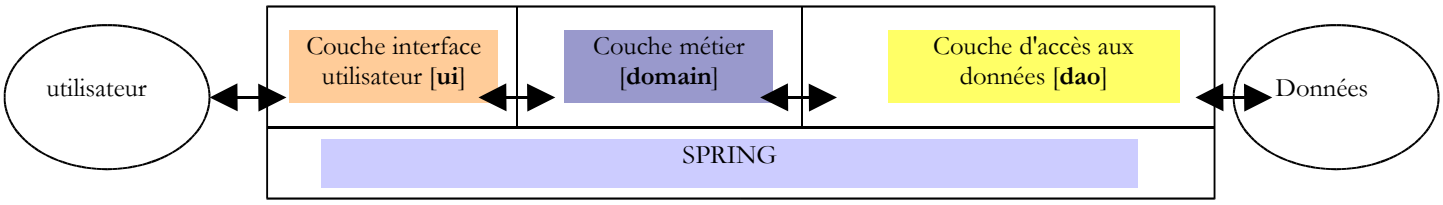
2 L'application [swingarticles] - Rappels

Nous présentons ici les éléments de l'application simplifiée de commerce électronique étudiée dans [article3]. Celle-ci permet à des clients :

- de consulter une liste d'articles provenant d'une base de données
- d'en mettre certains dans un panier électronique
- de valider celui-ci. Cette validation a pour seul effet de mettre à jour, dans la base de données, les stocks des articles achetés.

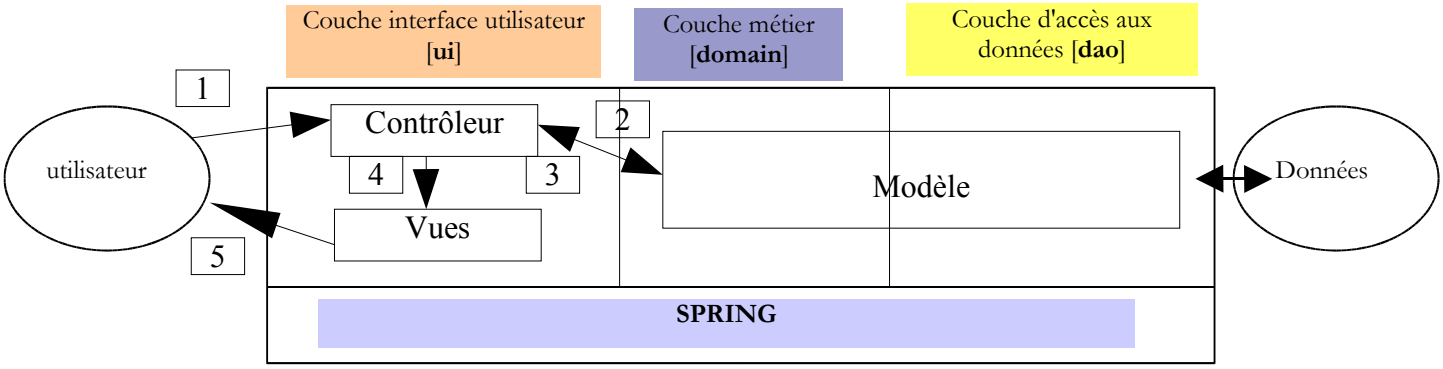
2.1 L'architecture 3-tier de l'application [swingarticles]

L'application [swingarticles] présente l'architecture à trois couches suivante :



- les trois couches sont rendues indépendantes grâce à l'utilisation d'interfaces
- l'intégration des différentes couches est réalisée avec **Spring**

L'application respecte une architecture MVC (Modèle - Vue - Contrôleur). Si nous reprenons le schéma en couches ci-dessus, l'architecture MVC s'y intègre de la façon suivante :

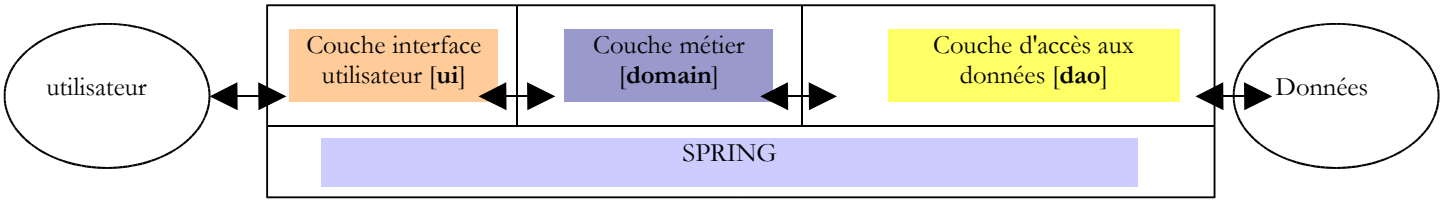


Le traitement d'une demande d'un client se déroule selon les étapes suivantes :

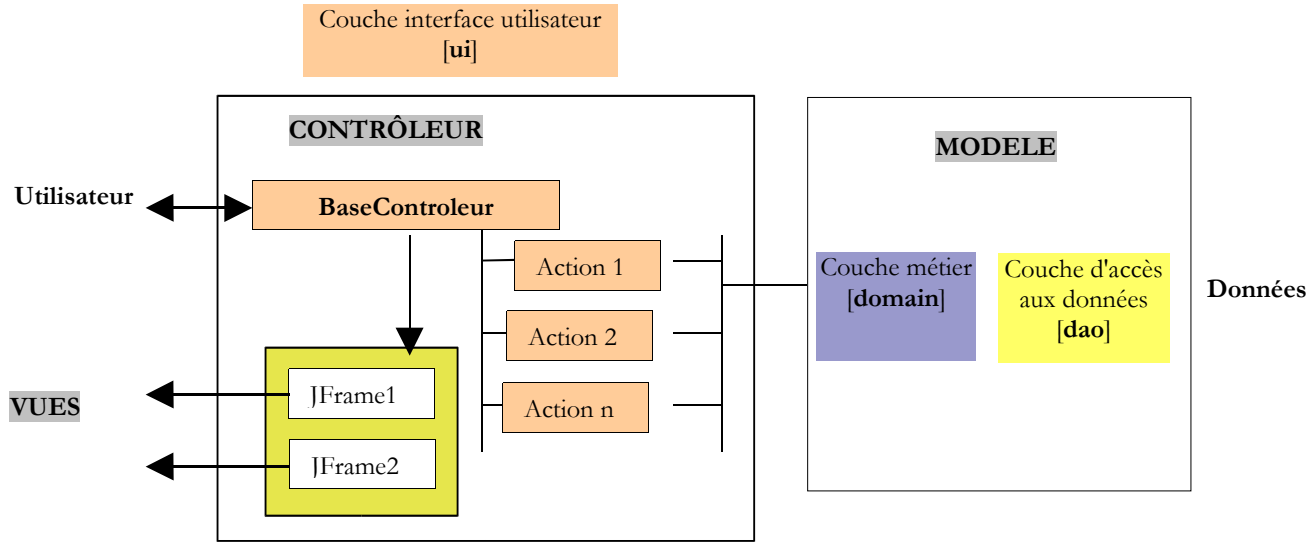
1. le client fait une demande au contrôleur à partir d'une fenêtre. Ce contrôleur est une classe qui voit passer toutes les demandes des clients. C'est la porte d'entrée de l'application. C'est le C de MVC.
2. le contrôleur traite cette demande. Pour ce faire, il peut avoir besoin de l'aide de la couche métier, ce qu'on appelle le modèle M dans la structure MVC.
3. le contrôleur reçoit une réponse de la couche métier. La demande du client a été traitée. Celle-ci peut appeler plusieurs réponses possibles. Un exemple classique est
   - une page d'erreurs si la demande n'a pu être traitée correctement
   - une page de confirmation sinon
4. le contrôleur choisit la réponse (= vue) à afficher. Celle-ci est le plus souvent une fenêtre contenant des éléments dynamiques. Le contrôleur fournit ceux-ci à la vue.
5. la vue est affichée. C'est le V de MVC.

2.2 La couche [ui] de l'application [swingarticles]

Revenons au schéma général de notre application [swingarticles] :



La couche [ui] est la couche d'interface avec l'utilisateur. Pour l'implémenter, nous avons utilisé le moteur MVC [M2VC]. Celui-ci nous a amenés à implémenter la couche [ui] de la façon suivante :



M=modèle	les classes métier, les classes d'accès aux données et la base de données
V=vues	les formulaires Windows
C=contrôleur	le contrôleur [BaseControleur] de traitement des requêtes clientes, les objets [Action]

Les grands principes de fonctionnement du moteur [M2VC] sont les suivants :

1. le contrôleur [BaseControleur] est le coeur de l'application. Toutes les demandes du client transitent par lui. C'est une classe fournie par [M2VC]. On peut dans certains cas être amené à la dériver. Pour les cas simples, ce n'est pas nécessaire.
2. [BaseControleur] prend les informations dont il a besoin dans un fichier appelé [**m2vc.xml**]. Il y trouve la liste des objets [Action] destinés à exécuter les demandes du client, la liste des vues de l'application, une liste d'objets [InfosAction] décrivant chaque action. [InfosAction] a les attributs suivants :
   - ✗ [vue] : désigne une vue [JFrame] à afficher si l'action ne consiste qu'à changer de vue.
   - ✗ [action] : désigne un objet [Action] à exécuter si l'action demandée nécessite l'exécution d'un code
   - ✗ [états] : un dictionnaire associant une vue à chacun des résultats possibles de l'objet [Action]. Le contrôleur affichera la vue associée au résultat renvoyé par l'action.
3. l'utilisateur a devant lui un formulaire windows. Celui-ci traite certains événements lui-même, ceux qui ne nécessitent pas la couche métier. Les autres sont délégués au contrôleur. On dit alors que la vue demande l'exécution d'une action au contrôleur. Le contrôleur reçoit cette demande sous la forme d'un nom d'action.
4. [BaseControleur] récupère alors l'instance [InfosAction] liée au nom de l'action qu'on lui demande d'exécuter. Pour cela, il a un dictionnaire associant le nom d'une action à une instance [InfosAction] rassemblant les informations nécessaires à cette action.
5. si l'attribut [vue] de [InfosAction] est non vide, alors la vue associée est affichée. On passe ensuite à l'étape 9.
6. si l'attribut [action] de [InfosAction] est non vide, alors l'action est exécutée. Celle-ci fait ce qu'elle a à faire puis rend au contrôleur une chaîne de caractères représentant le résultat auquel elle est parvenue.
7. le contrôleur utilise le dictionnaire [états] de [InfosAction] pour trouver la vue V à afficher. Il l'affiche.
8. ici une vue a été affichée. Le contrôleur s'est synchronisé avec et attend que la vue déclenche une nouvelle action. Celle-ci va être déclenchée par une action particulière de l'utilisateur sur la vue, qui à cette occasion va repasser la main au contrôleur en lui donnant le nom de l'action à exécuter.
9. le contrôleur reprend à l'étape 4

2.3 Le modèle de l'application [swingarticles]

Le modèle M du MVC est ici constitué des éléments suivants :

1. les classes métier
2. les classes d'accès aux données
3. la base de données

2.3.1 La base de données

La base de données ne contient qu'une table appelée ARTICLES générée avec les commandes SQL suivantes :

```
CREATE TABLE ARTICLES (
    ID              INTEGER NOT NULL,
    NOM             VARCHAR(20) NOT NULL,
    PRIX            NUMERIC(15,2) NOT NULL,
```

```
    STOCKACTUEL   INTEGER NOT NULL,
    STOCKMINIMUM  INTEGER NOT NULL
);
/* contraintes */
ALTER TABLE ARTICLES ADD CONSTRAINT CHK_ID check (ID>0);
ALTER TABLE ARTICLES ADD CONSTRAINT CHK_PRIX check (PRIX>=0);
ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCKACTUEL check (STOCKACTUEL>=0);
ALTER TABLE ARTICLES ADD CONSTRAINT CHK_STOCKMINIMUM check (STOCKMINIMUM>=0);
ALTER TABLE ARTICLES ADD CONSTRAINT CHK_NOM check (NOM<>'');
ALTER TABLE ARTICLES ADD CONSTRAINT UNQ_NOM UNIQUE (NOM);
/* clé primaire */
ALTER TABLE ARTICLES ADD CONSTRAINT PK_ARTICLES PRIMARY KEY (ID);
```

`id`	clé primaire identifiant un article de façon unique
`nom`	nom de l'article
`prix`	son prix
`stockactuel`	son stock actuel
`stockminimum`	le stock au-dessous duquel une commande de réapprovisionnement doit être faite

2.3.2 Les paquetages du modèle

Le modèle M est ici fourni sous la forme de trois archives :

- **istia.st.articles.dao** : contient les classes d'accès aux données de la couche [dao]
- **istia.st.articles.exception** : contient une classe d'exception pour cette gestion d'articles
- **istia.st.articles.domain** : contient les classes métier de la couche [domain]

archive	*contenu*	*rôle*
istia.st.articles.dao	- contient le paquetage [istia.st.articles.dao] qui lui-même contient les éléments suivants : - [IArticlesDao]: l'interface d'accès à la couche Dao. C'est la seule interface que voit la couche [domain]. Elle n'en voit pas d'autre. - [Article] : classe définissant un article - [ArticlesDaoSqlMap] : classe d'implémentation de l'interface [IArticlesDao] avec l'outil SqlMap	couche d'accès aux données - se trouve entièrement dans la couche [dao] de l'architecture 3-tier de l'application web
istia.st.articles.domain	- contient le paquetage [istia.st.articles.domain] qui lui-même contient les éléments suivants : - [IArticlesDomain]: l'interface d'accès à la couche [domain]. C'est la seule interface que voit la couche web. Elle n'en voit pas d'autre. - [AchatsArticles] : une classe implémentant [IArticlesDomain] - [Achat] : classe représentant l'achat d'un client - [Panier] : classe représentant l'ensemble des achats d'un client	représente le modèle des achats sur le web - se trouve entièrement dans la couche [domain] de l'architecture 3-tier de l'application web
istia.st.articles.exception	- contient le paquetage [istia.st.articles.exception] qui lui-même contient les éléments suivants : - [UncheckedAccessArticlesException]: classe définissant une exception de type [RuntimeException]. Ce type d'exception est lancée par la couche [dao] dès qu'un problème d'accès aux données se produit.	

2.3.3 Le paquetage [istia.st.articles.dao]

La classe définissant un article est la suivante :

```
package istia.st.articles.dao;
import istia.st.articles.exception.UncheckedAccessArticlesException;

/**
 * @author ST - ISTIA
 *
 */
public class Article {
  private int id;
  private String nom;
  private double prix;
  private int stockActuel;
  private int stockMinimum;

  /**
   * constructeur par défaut
   */
  public Article() {
  }

  public Article(int id, String nom, double prix, int stockActuel,
      int stockMinimum) {
    // init attributs d'instance
    setId(id);
    setNom(nom);
    setPrix(prix);
    setStockActuel(stockActuel);
    setStockMinimum(stockMinimum);
  }

  // getters - setters
  public int getId() {
    return id;
  }

  public void setId(int id) {
    // id valide ?
    if (id < 0)
      throw new UncheckedAccessArticlesException("id[" + id + "] invalide");
    this.id = id;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    // nom valide ?
    if(nom==null || nom.trim().equals("")){
      throw new UncheckedAccessArticlesException("Le nom est [null] ou vide");
    }
    this.nom = nom;
  }

  public double getPrix() {
    return prix;
  }

  public void setPrix(double prix) {
    // prix valide ?
    if(prix<0) throw new UncheckedAccessArticlesException("Prix["+prix+"]invalide");
    this.prix = prix;
  }

  public int getStockActuel() {
    return stockActuel;
  }

  public void setStockActuel(int stockActuel) {
    // stock valide ?
    if (stockActuel < 0)
      throw new UncheckedAccessArticlesException("stockActuel[" + stockActuel + "] invalide");
    this.stockActuel = stockActuel;
  }

  public int getStockMinimum() {
    return stockMinimum;
  }

  public void setStockMinimum(int stockMinimum) {
    // stock valide ?
    if (stockMinimum < 0)
      throw new UncheckedAccessArticlesException("stockMinimum[" + stockMinimum + "] invalide");
    this.stockMinimum = stockMinimum;
```

```
  }

  public String toString() {
    return "[" + id + "," + nom + "," + prix + "," + stockActuel + ","
        + stockMinimum + "]";
  }
}
```

Cette classe offre :

1. un constructeur permettant de fixer les 5 informations d'un article
2. des accesseurs appelés souvent getters/setters servant à lire et écrire les 5 informations. Les noms de ces méthodes suivent la norme **JavaBean**. L'utilisation d'objets JavaBean dans la couche DAO pour faire l'interface avec les données du SGBD est classique.
3. une vérification des données insérées dans l'article. En cas de données erronées, une exception est lancée.
4. une méthode **toString** qui permet d'obtenir la valeur d'un article sous forme de chaîne de caractères. C'est souvent utile pour le débogage d'une application.

L'interface [IArticlesDao] est définie comme suit :

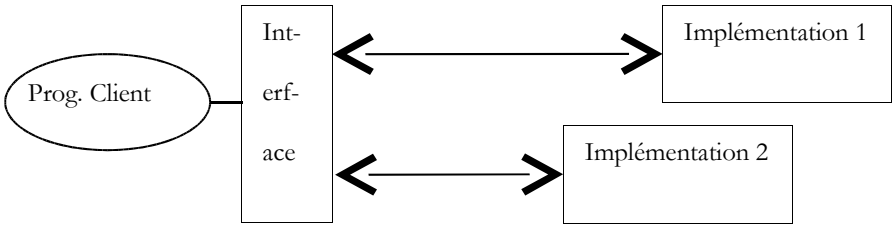
```
package istia.st.articles.dao;

import istia.st.articles.domain.Article;
import java.util.List;

/**
 * @author ST-ISTIA
 *
 */
public interface IArticlesDao {

  /**
   * @return : liste de tous les articles
   */
  public List getAllArticles();

  /**
   * @param unArticle :
   *          l'article à ajouter
   */
  public int ajouteArticle(Article unArticle);

  /**
   * @param idArticle :
   *          id de l'article à supprimer
   */
  public int supprimeArticle(int idArticle);

  /**
   * @param unArticle :
   *          l'article à modifier
   */
  public int modifieArticle(Article unArticle);

  /**
   * @param idArticle :
   *          id de l'article cherché
   * @return : l'article trouvé ou null
   */
  public Article getArticleById(int idArticle);

  /**
   * vide la table des articles
   */
  public void clearAllArticles();

  /**
   *
   * @param idArticle id de l'article dont on change le stock
   * @param mouvement valeur à ajouter au stock (valeur signée)
   */
  public int changerStockArticle(int idArticle, int mouvement);
}
```

Le rôle des différentes méthodes de l'interface est le suivant :

`getAllArticles`	rend tous les articles de la table ARTICLES dans une liste d'objets [Article]
`clearAllArticles`	vide la table ARTICLES
`getArticleById`	rend l'objet [Article] identifié par sa clé primaire
`ajouteArticle`	permet d'ajouter un article à la table ARTICLES
`modifieArticle`	permet de modidier un article de la table [ARTICLES]

`supprimerArticle`	permet de supprimer un article de la table [ARTICLES]
`changerStockArticle`	permet de modifier le stock d'un article de la table [ARTICLES]

L'interface met à disposition des programmes clients un certain nombre de méthodes définies uniquement par leurs signatures. Elle ne s'occupe pas de la façon dont ces méthodes seront réellement implémentées. Cela amène de la souplesse dans une application. Le programme client fait ses appels sur une interface et non pas sur une implémentation précise de celle-ci.



Le choix d'une implémentation précise se fait au moyen d'un fichier de configuration Spring. La classe d'implémentation [ArticlesDaoSqlMap] de l'interface **IArticlesDao**, choisie ici, utilise le produit open source I**batis SqlMap**. Celui-ci nous permet d'enlever toute instruction SQL du code java. Le squelette de la classe [ArticlesDaoSqlMap] est le suivant :

```
package istia.st.articles.dao;

// Imports
import com.ibatis.sqlmap.client.SqlMapClient;
import istia.st.articles.domain.Article;
import java.util.List;

public class ArticlesDaoSqlMap implements IArticlesDao {

  // Fields
  private SqlMapClient sqlMap;

  // Constructors
  public ArticlesDaoSqlMap(String sqlMapConfigFileName) { }

  // Methods
  public SqlMapClient getSqlMap() {}
  public void setSqlMap(SqlMapClient sqlMap) { }
  public synchronized List getAllArticles() {}
  public synchronized int ajouteArticle(Article unArticle) {}
  public synchronized int supprimeArticle(int idArticle) {}
  public synchronized int modifieArticle(Article unArticle) {}
  public synchronized Article getArticleById(int idArticle) {}
  public synchronized void clearAllArticles() { }
  public synchronized int changerStockArticle(int idArticle, int mouvement) {}
}
```

Toutes les méthodes d'accès aux données ont été synchronisées afin d'éviter les problèmes d'accès concurrents à la source de données. A un moment donné, un seul thread a accès à une méthode donnée.

La classe [ArticlesDaoSqlMap] utilise l'outil [Ibatis SqlMap]. L'intérêt de cet outil est de permettre de sortir le code SQL d'accès aux données du code Java. Il est alors placé dans un fichier de configuration. Nous aurons l'occasion d'y revenir. Pour se construire, la classe [ArticlesDaoSqlMap] a besoin d'un fichier de configuration dont le nom est passé en paramètre au constructeur de la classe. Ce fichier de configuration définit les informations nécessaires pour :

- accéder au SGBD dans lequel se trouvent les articles
- gérer un pool de connexions
- gérer les transactions

Dans notre exemple, il s'appellera [sqlmap-config-odbc.xml] et définira l'accès à une base ODBC :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMapConfig
    PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
  <transactionManager type="JDBC">
        <dataSource type="SIMPLE">
      <property name="JDBC.Driver" value="sun.jdbc.odbc.JdbcOdbcDriver"/>
      <property name="JDBC.ConnectionURL"
        value="jdbc:odbc:odbc-access-dvp-articles"/>
      <property name="JDBC.Username" value="sysdba"/>
      <property name="JDBC.Password" value="masterkey"/>
      <property name="JDBC.DefaultAutoCommit" value="true"/>
    </dataSource>
```

```
  </transactionManager>
  <sqlMap resource="articles.xml"/>
</sqlMapConfig>
```

Le fichier de configuration [articles.xml] référencé ci-dessus permet de définir comment construire une instance de la classe [istia.st.articles.dao.Article] à partir d'une ligne de la table [ARTICLES] du SGBD. Il définit également les requêtes SQL qui permettront à la couche [dao] d'obtenir les données de la source de données ODBC. Sn contenu est le suivant :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
    PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-2.dtd">
<sqlMap namespace="Articles">

  <!-- un alias sur la classe istia.st.articles.dao.Article -->
  <typeAlias alias="article" type="istia.st.articles.dao.Article"/>

  <!-- le mapping ORM  : ligne table ARTICLES - instance classe Article -->
  <resultMap id="article" class="article">
    <result property="id" column="ID"/>
    <result property="nom" column="NOM"/>
    <result property="prix" column="PRIX"/>
    <result property="stockActuel" column="STOCKACTUEL"/>
    <result property="stockMinimum" column="STOCKMINIMUM"/>
  </resultMap>

  <!-- la requête SQL pour obtenir tous les articles -->
  <statement id="getAllArticles" resultMap="article">
    select id, nom, prix,
    stockactuel, stockminimum from ARTICLES
</statement>

  <!-- la requête SQL pour supprimer tous les articles -->
  <statement id="clearAllArticles">delete from ARTICLES</statement>

  <!-- la requête SQL pour insérer un article -->
  <statement id="insertArticle">
    insert into ARTICLES (id, nom, prix,
    stockactuel, stockminimum) values
    (#id#,#nom#,#prix#,#stockactuel#,#stockminimum#)
</statement>

  <!-- la requête SQL pour supprimer un article donné -->
  <statement id="deleteArticle">delete FROM ARTICLES where id=#id#</statement>

  <!-- la requête SQL pour modifier un article donné -->
  <statement id="modifyArticle">
    update ARTICLES set nom=#nom#,
    prix=#prix#,stockactuel=#stockactuel#,stockminimum=#stockminimum# where
    id=#id#
</statement>

  <!-- la requête SQL pour obtenir un article donné -->
  <statement id="getArticleById" resultMap="article">
    select id, nom, prix,
    stockactuel, stockminimum FROM ARTICLES where id=#id#
</statement>

  <!-- la requête SQL pour modifier le stock d'un article donné -->
  <statement id="changerStockArticle">
    update ARTICLES set
    stockActuel=stockActuel+#mouvement#
    where id=#id# and stockActuel+#mouvement#&gt;=0
</statement>
</sqlMap>
```

2.3.4 Le paquetage [istia.st.articles.domain] initial

L'interface [IArticlesDomain] découple la couche [métier] de la couche [ui]. Cette dernière accède à la couche [métier/domain] via cette interface sans se préoccuper de la classe qui l'implémente réellement. L'interface [IArticlesDomain] définit les actions suivantes pour l'accès à la couche métier :

```
package istia.st.articles.domain;

// Imports
import java.util.ArrayList;
import java.util.List;

public abstract interface IArticlesDomain {

  // Méthodes
```

```
  void acheter(Panier panier);
  List getAllArticles();
  Article getArticleById(int idArticle);
  ArrayList getErreurs();
}
```

`List getAllArticles()`	rend la liste liste d'objets [Article] à présenter au client
`Article getArticleById(int idArticle)`	rend l'objet [Article] identifié par [idArticle]
`void acheter(Panier panier)`	valide le panier du client en décrémentant les stocks des articles achetés de la quantité achetée - peut échouer si le stock est insuffisant
`ArrayList getErreurs()`	rend la liste des erreurs qui se sont produites - vide si pas d'erreurs

Ici, l'interface [IArticlesDomain] sera implémentée par la classe [AchatsArticles] suivante :

```
package istia.st.articles.domain;

// Imports
import istia.st.articles.dao.IArticlesDao;
import istia.st.articles.exception.UncheckedAccessArticlesException;
import java.util.ArrayList;
import java.util.List;

public class AchatsArticles implements IArticlesDomain {

  // Champs
  private IArticlesDao articlesDao;
  private ArrayList erreurs;

  // Constructeurs
  public AchatsArticles(IArticlesDao articlesDao) { }

  // Méthodes
  public ArrayList getErreurs() {}
  public List getAllArticles() {}
  public Article getArticleById(int id) {}
  public void acheter(Panier panier) { }
}
```

Cette classe implémente les quatre méthodes de l'interface [IArticlesDomain]. Elle a deux champs privés :

`IArticlesDao articlesDao`	l'objet d'accès aux données fourni par la couche d'accès aux données
`ArrayList erreurs`	la liste des erreurs éventuelles

Pour construire une instance de la classe, il faut fournir l'objet permettant l'accès aux données du SGBD :

`public AchatsArticles(IArticlesDao articlesDao)`	constructeur

La classe [Achat] représente un achat du client :

```
package istia.st.articles.domain;

public class Achat {

  // Champs
  private Article article;
  private int qte;

  // Constructeurs
  public Achat(Article article, int qte) { }

  // Méthodes
  public double getTotal() {}
  public Article getArticle() {}
  public void setArticle(Article article) { }
  public int getQte() {}
  public void setQte() { }
  public String toString() {}
}
```

La classe [Achat] est un JavaBean avec les champs et méthodes suivants :

`article`	l'article acheté
`qte`	la quantité achetée
`double getTotal()`	rend le montant de l'achat
`String toString()`	chaîne d'identité de l'objet

La classe [Panier] représente l'ensemble des achats du client :

```
package istia.st.articles.domain;

// Imports
import java.util.ArrayList;

public class Panier {

  // Champs
  private ArrayList achats;

  // Constructeurs
  public Panier() { }

  // Méthodes
  public ArrayList getAchats() {}
  public void ajouter(Achat unAchat) { }
  public void enlever(int idAchat) { }
  public double getTotal() {}
  public String toString() { }
}
```

La classe [Panier] est un JavaBean avec les champs et méthodes suivants :

`achats`	la liste des achats du client - liste d'objets de type [Achat]
`void ajouter(Achat unAchat)`	ajoute un achat à la liste des achats
`void enlever(int idArticle)`	enlève l'achat de l'article idArticle
`double getTotal()`	rend le montant total des achats
`String toString()`	rend la chaîne d'identité du panier
`ArrayList getAchats()`	rend la liste des achats

2.3.5 Le paquetage [istia.st.articles.exception]

Ce paquetage contient la classe définissant l'exception lancée par la couche [dao] lorsqu'elle rencontre un problème d'accès à la source de données :

```
package istia.st.articles.exception;

public class UncheckedAccessArticlesException
    extends RuntimeException {

  public UncheckedAccessArticlesException() {
    super();
  }

  public UncheckedAccessArticlesException(String mesg) {
    super(mesg);
  }

  public UncheckedAccessArticlesException(String mesg, Throwable th) {
    super(mesg, th);
  }
}
```

L'exception [UncheckedAccessArticlesException] étant dérivée de [RuntimeException], le compilateur ne nous oblige pas la gérer avec une structure try-catch.

2.3.6 Le paquetage [istia.st.articles.domain] revisité

Alors que l'interface [IArticlesDomain] et sa classe d'implémentation [AchatsArticles] ont été utilisées dans plusieurs articles précédents, ils s'avèrent en fait avoir une conception incorrecte pour ce qui est de l'achat du panier. Revenons sur la classe [AchatsArticles] :

```
package istia.st.articles.domain;

// Imports
import istia.st.articles.dao.IArticlesDao;
import istia.st.articles.exception.UncheckedAccessArticlesException;
import java.util.ArrayList;
import java.util.List;

public class AchatsArticles implements IArticlesDomain {

  // Champs
  private IArticlesDao articlesDao;
```

```
  private ArrayList erreurs;

  // Constructeurs
  public AchatsArticles(IArticlesDao articlesDao) { }

  // Méthodes
  public ArrayList getErreurs() {}
  public List getAllArticles() {}
  public Article getArticleById(int id) {}
  public void acheter(Panier panier) { }
}
```

L'achat d'un panier se fait en deux temps :

1. la méthode [acheter] est tout d'abord utilisée pour décrémenter les stocks des articles achetés,
2. la méthode [getErreurs] est ensuite utilisée pour obtenir la liste des éventuelles erreurs. Celles-ci listent les articles dont les stocks sont insuffisants pour satisfaire la demande du client.

Si cette méthode est acceptable (quoique maladroite) lorsqu'il n'y a qu'un client, elle ne l'est plus lorsqu'il y en a plusieurs comme c'est le cas dans une application web où les clients sont multiples. Il faut tout d'abord comprendre qu'il n'y a qu'une seule instance de la classe [AchatsArticles] pour satisfaire les demandes des clients. On a affaire à un singleton. Prenons la situation suivante dans le cadre d'une application web :

1. le client 1 veut acheter le panier [panier1]. Il est servi par un thread T1 côté serveur. Celui-ci commence par faire appel à la méthode [acheter] du singleton [AchatsArticles]. Celle-ci se termine. Les erreurs d'achats ont été stockées dans le champ privé [erreurs] du singleton. Le thread T1 est alors interrompu pour une raison quelconque.
2. le client 2 veut acheter le panier [panier2]. Il est servi par un thread T2 côté serveur. Celui-ci commence par faire appel à la méthode [acheter] du singleton [AchatsArticles]. Celle-ci se termine. Les erreurs d'achats ont été stockées dans le champ privé [erreurs] du singleton écrasant donc les erreurs du client 1. Le thread T2 n'est pas interrompu. Il fait alors appel à la méthode [getErreurs] du singleton et obtient la liste d'erreurs des achats de [panier2]. Le thread T2 envoie sa réponse au client 2 et se termine.
3. Le thread T1 récupère alors le processeur et fait ce qu'il n'a pas pu eu le temps de faire. Il fait appel à la méthode [getErreurs] du singleton [AchatsArticles] et obtient la liste des erreurs d'achats de [panier2] et non celle de [panier1] qui n'existe plus.

On a un problème...

Lorsqu'on a affaire à un singleton servant plusieurs clients, il est important de vérifier que ses données d'état ne concernent pas un client particulier. Dans la classe [AchatsArticles], les données d'état sont les suivantes :

```
  // Champs
  private IArticlesDao articlesDao;
  private ArrayList erreurs;
```

`articlesDao`	représente l'instance d'accès à la couche [dao] - est elle-même un singleton. N'est pas lié à un client particulier.
`erreurs`	la liste des erreurs qui se sont produites sur l'achat d'un panier. Cette donnée est liée au panier d'un client particulier. Elle ne peut faire partie de l'état du singleton.

La solution à notre problème est ici relativement simple. Il faut changer la signature de la méthode [acheter]. Elle doit devenir :

```
  public ArrayList acheter(Panier panier);
```

Le résultat de type [ArrayList] est la liste des erreurs survenues lors de l'achat du panier. Celle-ci n'a donc plus à faire partie de l'état du singleton implémentant l'interface [IArticlesDomain]. Cette dernière devient la suivante :

```
1. package istia.st.articles.domain;
2.
3. import istia.st.articles.dao.Article;
4.
5. import java.util.ArrayList;
6. import java.util.List;
7.
8. /**
9.  * @author serge.tahe@istia.univ-angers.fr
10. *
11. */
12.public interface IArticlesDomain {
13.  // liste de tous les articles
14.  public List getAllArticles();
15.  // obtenir un article particulier
16.  public Article getArticleById(int id);
17.  // acheter un panier
18.  public ArrayList acheter(Panier panier);
19.}
```

La méthode [getErreurs] a disparu. La méthode [acheter] a changé de signature, ligne 18.

La classe d'implémentation [AchatsArticles] devient la suivante :

```
1. package istia.st.articles.domain;
2.
3. // Imports
4. import istia.st.articles.dao.IArticlesDao;
5. import istia.st.articles.exception.UncheckedAccessArticlesException;
6. import java.util.ArrayList;
7. import java.util.List;
8.
9. public class AchatsArticles implements IArticlesDomain {
10.
11.  // Champs
12.  private IArticlesDao articlesDao;
13.  private ArrayList erreurs;
14.
15.  // Constructeurs
16.  public AchatsArticles(IArticlesDao articlesDao) { }
17.
18.  // Méthodes
19.  public List getAllArticles() {...}
20.  public Article getArticleById(int id) {...}
21.  public ArrayList acheter(Panier panier) {...}
22.}
```

La méthode [getErreurs] a disparu. La méthode [acheter] a changé de signature et devient la suivante :

```
1.   // valider un panier d'achats
2.   public ArrayList acheter(Panier panier) {
3.     // la liste des erreurs
4.     ArrayList erreurs = new ArrayList();
5.     // on parcourt les achats
6.     ArrayList achats = panier.getAchats();
7.     Article article = null;
8.     Achat achat = null;
9.     for (int i = achats.size() - 1; i >= 0; i--) {
10.      // on récupère l'achat
11.      achat = (Achat) achats.get(i);
12.      // on tente de modifier le stock de l'article dans la base
13.      int nbarticles = articlesDao.changerStockArticle(achat.getArticle()
14.          .getId(), -achat.getQte());
15.      // a-t-on réussi ?
16.      if (nbarticles != 0) {
17.        achats.remove(i);
18.      } else {
19.        erreurs.add("Achat article [" + achat.getArticle() + ","
20.            + achat.getQte() + "] impossible - Vérifiez son stock");
21.      }
22.    }
23.    // on rend la liste des erreurs
24.    return erreurs;
25.  }
```

- la liste des erreurs est devenue une variable locale à la méthode - ligne 4
- elle est rendue comme résultat - ligne 24

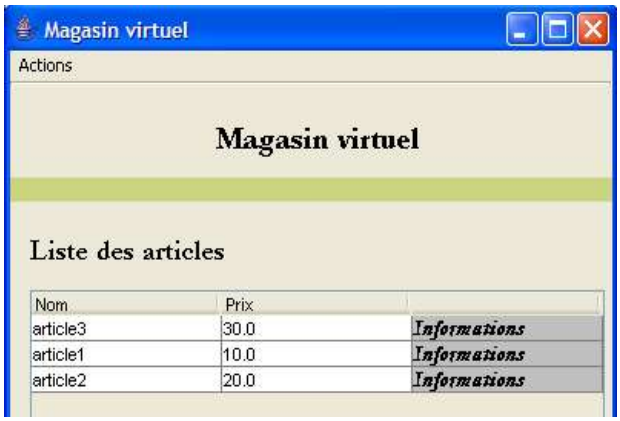
Ce code est-il suffisant pour garantir l'étanchéité entre deux clients ? Dans cette méthode, seules des variables locales à la méthode sont utilisées sauf la variable [articlesDao] qui est une variable d'instance du singleton partagée par tous les clients. La méthode [articlesDao.changerStockArticle] utilisée ligne 13 est synchronisée (synchronized). Deux threads ne peuvent l'exécuter simultanément. Il semble donc que la méthode [acheter] assure bien l'étanchéité des clients.

Nous utiliserons désormais cette nouvelle interface [IArticlesDomain] et sa nouvelle classe d'implémentation [AchatsArticles]. On peut conclure de cette erreur que les tests de la couche [domain] avaient été insuffisants. On avait négligé de tester l'étanchéité des clients entre-eux.

2.4 Les vues de l'application [swingarticles]

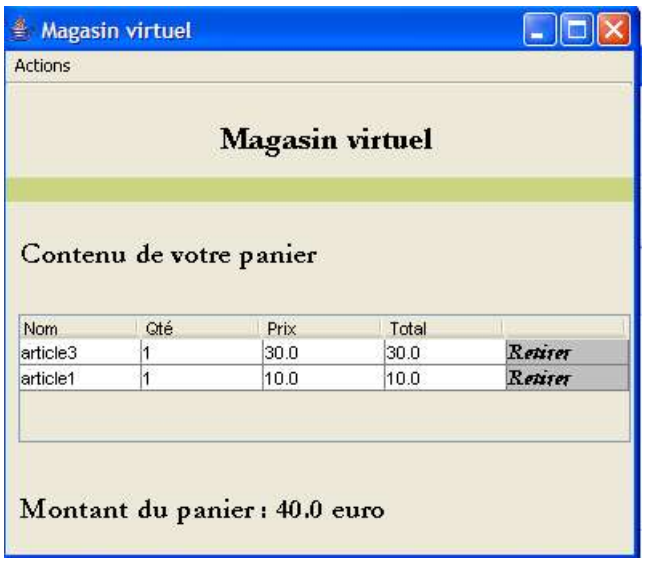
Les différentes vues présentées à l'utilisateur sont les suivantes :

- la vue "LISTE" qui présente une liste des articles en vente



- la vue [INFOS] qui donne des informations supplémentaires sur un produit :

- la vue [PANIER] qui donne le contenu du panier du client

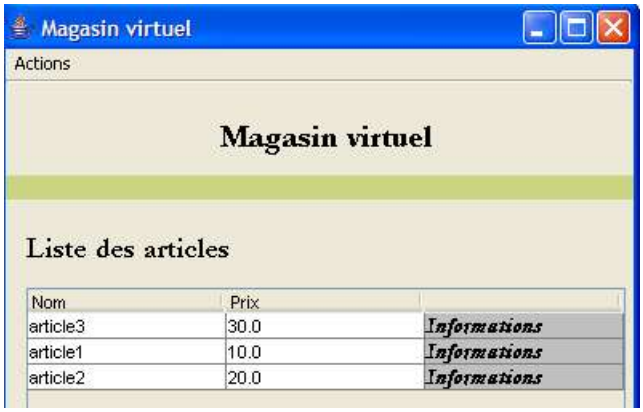


- la vue [PANIERVIDE] pour le cas où le panier du client est vide

- la vue [ERREURS] qui signale toute erreur d'achat d'articles

2.5 Fonctionnement de l'application [swingarticles]

Nous présentons ci-dessous l'enchaînement des vues lors d'une utilisation typique de l'application :



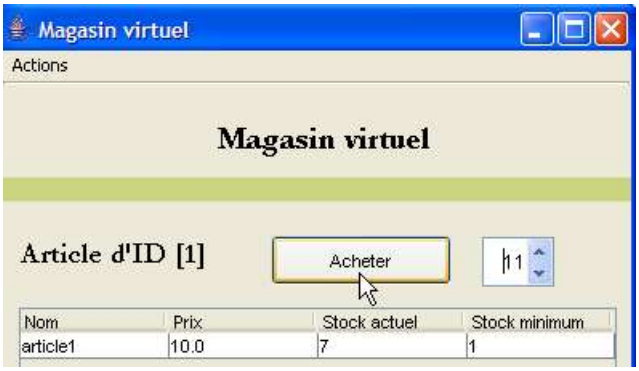
A partir de la vue ci-dessus, nous utilisons les options du menu pour faire des opérations. En voici quelques unes. La colonne de gauche représente la demande du client et la colonne de droite la réponse qui lui est faite.

Magasin virtuel
Actions
Liste des articles
Valider le panier
Quitter
Magasin virtuel
Contenu de votre panier
Nom
Qté
Prix
Total
article3
2
30.0
60.0
Retirer

Magasin virtuel
Actions
Magasin virtuel
Liste des articles
Nom
Prix
article3
30.0
Informations
article1
10.0
Informations
article2
20.0
Informations

Magasin virtuel
Actions
Magasin virtuel
Liste des articles
Nom
Prix
article3
30.0
Informations
article1
10.0
Informations
article2
20.0
Informations

Magasin virtuel
Actions
Magasin virtuel
Article d'ID [1]
Acheter
1
Nom
Prix
Stock actuel
Stock minimum
article1
10.0
7
1

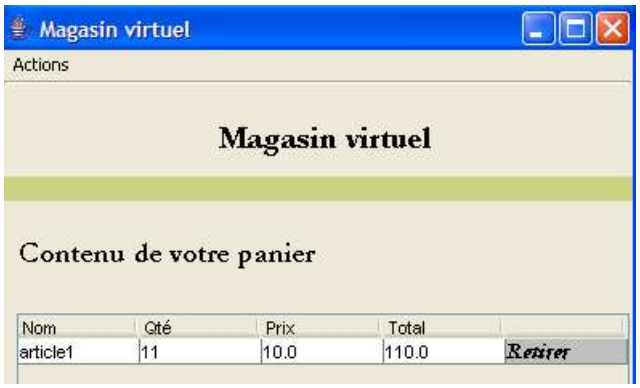

Magasin virtuel
Actions
Magasin virtuel
Article d'ID [1]
Acheter
11
Nom
Prix
Stock actuel
Stock minimum
article1
10.0
7
1

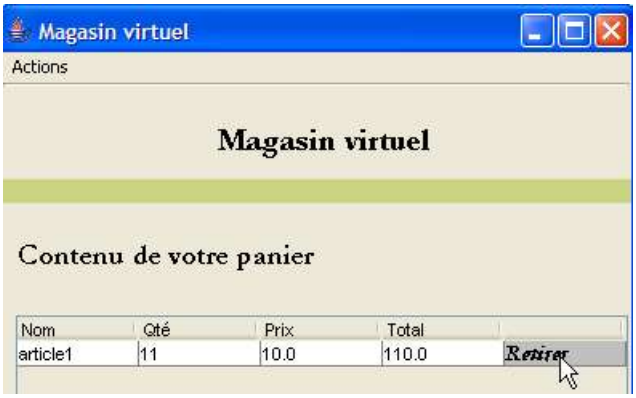
Magasin virtuel
Actions
Magasin virtuel
Contenu de votre panier
Nom
Qté
Prix
Total
article3
2
30.0
60.0
Retirer
article1
11
10.0
110.0
Retirer
Montant du panier : 170.0 euro

Magasin virtuel
Actions
Liste des articles
Valider le panier
Quitter
Magasin virtuel
Contenu de votre panier
Nom Qté Prix Total
article3 2 30.0 60.0 Retirer
article1 11 10.0 110.0 Retirer

Magasin virtuel
Actions
Magasin virtuel
Les erreurs suivantes se sont produites
1 - Achat article [[1,article1,10.0,7,1],11] impossible - Vérifiez son stock

Magasin virtuel
Actions
Liste des articles
Voir le panier
Quitter
Magasin virtuel
Les erreurs suivantes se sont produites
1 - Achat article [[1,article1,10.0,7,1],11] impossible - Vérifiez son stock


Magasin virtuel
Actions
Magasin virtuel
Contenu de votre panier
Nom Qté Prix Total
article1 11 10.0 110.0 Retirer


Magasin virtuel
Actions
Magasin virtuel
Contenu de votre panier
Nom Qté Prix Total
article1 11 10.0 110.0 Retirer

Magasin virtuel
Actions
Magasin virtuel
Votre panier est vide !

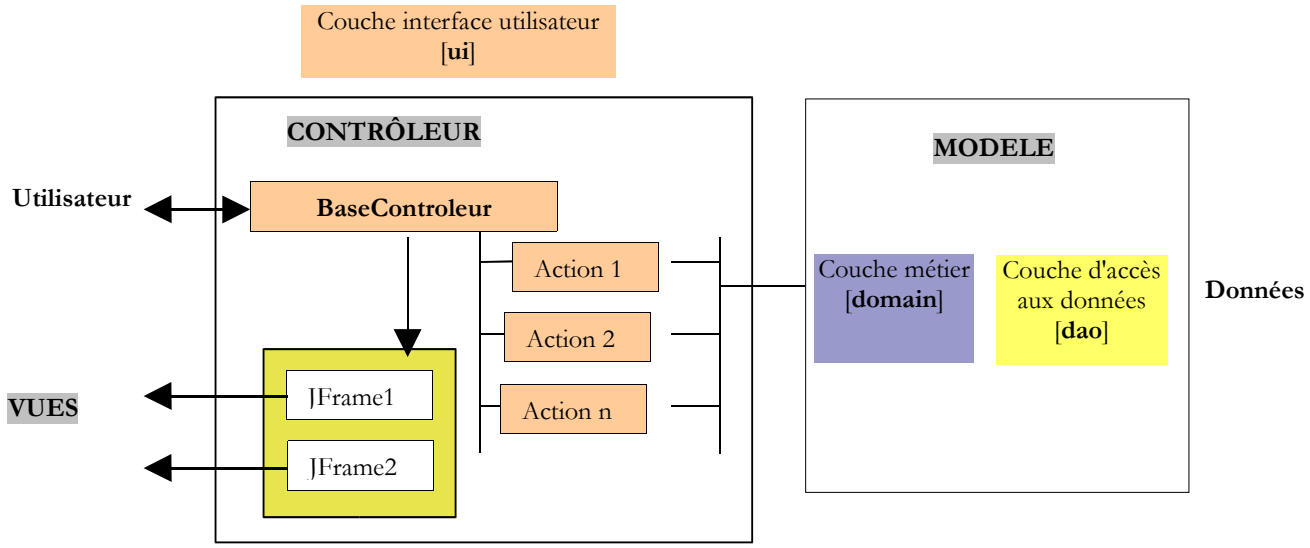
Magasin virtuel
Actions
Liste des articles
Quitter
Magasin virtuel
Votre panier est vide !

Magasin virtuel
Actions
Magasin virtuel
Liste des articles
Nom Prix
article3 30.0 Informations
article1 10.0 Informations
article2 20.0 Informations

2.5.1 Modification de l'application [swingarticles]

L'application [swingarticles] décrite dans [article3] utilisait les anciennes versions de l'interface [IArticlesDomain] et sa classe d'implémentation [AchatsArticles], celles qui pouvaient "boguer" de façon aléatoire en présence de plusieurs clients simultanés. L'application [swingarticles] décrite dans [article3] étant une application mono-utilisateur, le problème de clients simultanés ne se pose pas.

Dans le contexte client-serveur dans lequel nous allons nous placer, nous sommes obligés d'utiliser les nouvelles versions de l'interface [IArticlesDomain] et sa classe d'implémentation [AchatsArticles] qui assurent l'étanchéité des clients entre-eux. Si nous reprenons l'architecture de [swingarticles], nous voyons que la couche [ui] implémentée par [swingarticles] parle à la couche [domain] implementée par la classe [AchatsArticles] :



Il faut donc que l'application [swingarticles] sache parler à la nouvelle interface [IArticlesDomain]. Nous avons vu que seule la méthode [acheter] de l'interface avait changé. Donc seules les classes [Action] utilisant cette méthode dans [swingarticles] sont à modifier. Il n'y en a qu'une, l'action [ActionValiderPanier] :

```
1. package istia.st.m2vc.magasin.actions;
2.
3. import istia.st.m2vc.magasin.bases.*;
4. import java.util.ArrayList;
5.
6. /**
7.  * @author serge.tahe@istia.univ-angers.fr
8.  *
9.  */
10.public class ActionValiderPanier
11.    extends AbstractBaseAction {
12.
13.  // le client valide son panier
14.  public String execute() {
15.    // on récupère la session
16.    Session session = getSession();
17.    // on tente de valider le panier
18.    try {
19.      ArrayList erreurs=session.getArticlesDomain().acheter(session.getPanier());
20.      // on note les éventuelles erreurs d'achats
21.      session.setErreurs(erreurs);
22.    }
23.    catch (Exception ex) {
24.      // on note l'erreur
25.      ArrayList erreurs = session.getErreurs();
26.      erreurs.clear();
27.      erreurs.add(ex.toString());
28.    }
29.    // on rend le résultat
30.    if (session.getErreurs().size() == 0) {
31.      return "succès";
32.    }
33.    else {
34.      return "échec";
35.    }
```

```
36.   }
37.}
```

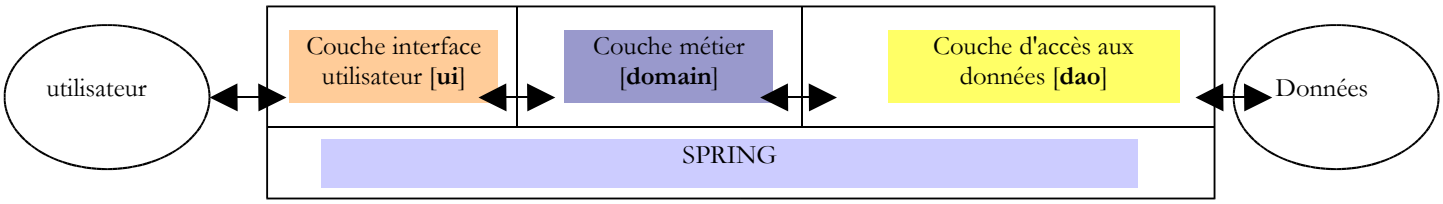
- ligne 19 : le panier est acheté et les erreurs d'achats récupérées en même temps. Dans la version précédente, il y avait deux appels successifs : un premier à la méthode [acheter], le second à la méthode [getErreurs].

3 Introduction à l'application client-serveur [magasin-client-serveur]

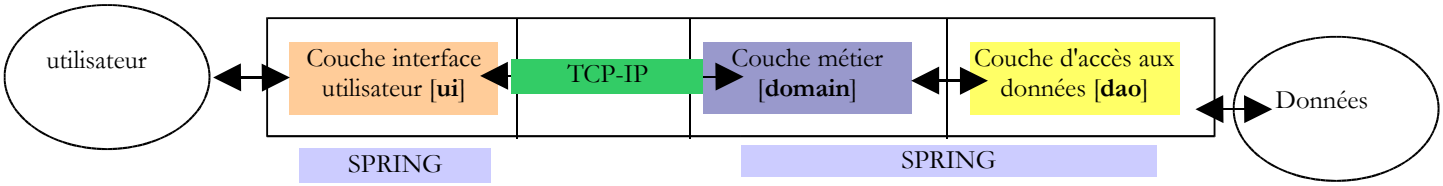
Nous commençons ici le portage de l'application [swingarticles] décrite précédemment et qui s'exécutait sur une seule machine, vers une application client-serveur que nous appellerons désormais [magasin-client-serveur] et qui elle s'exécutera sur deux machines.

3.1 Architecture de l'application client-serveur [magasin-client-serveur]

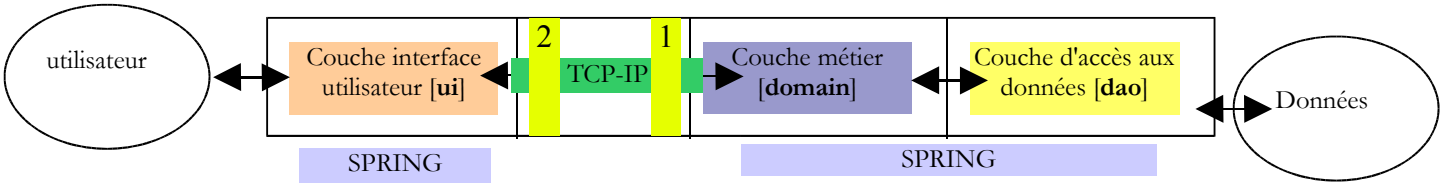
Rappelons ici ce qui a été dit dans l'introduction. L'application [swingarticles] a l'architecture suivante :



L'application que nous allons développer ici et que nous appellerons désormais [magasin-client-serveur] aura, elle, l'architecture client-serveur suivante :



Nous voulons réutiliser autant que possible les couches [ui, domain, dao] développées dans les articles précédents. L'idéal serait qu'on utilise les archives de ces couches sans toucher au code source. Il nous faut, pour cela, créer des couches d'adaptation entre le client et le serveur. L'architecture précédente devient alors la suivante :



Nous ajoutons deux couches notées ci-dessus 1 et 2 :

- la couche 1 va exposer un service distant d'accès à la couche [domain]. Cette couche a pour rôle de cacher le réseau à la couche [domain] qui ne saura pas qu'elle est utilisée par des clients distants. Nous l'appellerons couche [proxyServeur].
- la couche 2 va implémenter les proxy d'accès aux services web de la couche 1, de façon transparente pour la couche [ui] qui ne saura pas que les données qu'elle reçoit et envoie vont sur le réseau. Nous l'appellerons couche [proxyClient].

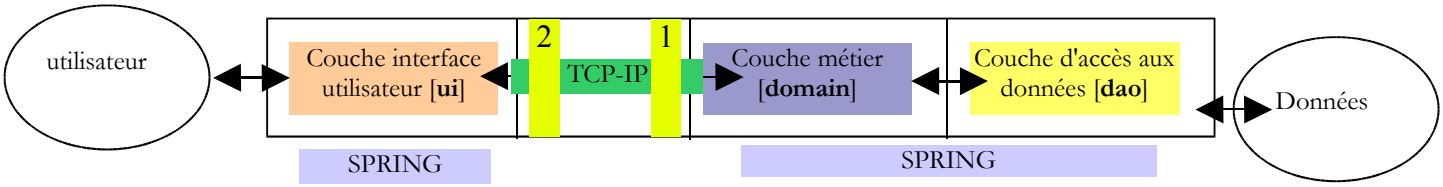
Spring permet d'implémenter les couches d'adaptation ci-dessus de différentes façons :

1. avec la technologie Spring RMI
2. avec la technologie Spring RMI-IIOP
3. avec la technologie Spring HttpInvoker
4. avec la technologie Spring Caucho Hessian
5. avec la technologie Spring Caucho Burlap
6. avec la technologie JaxRpc

Nous présentons dans cet article, des implémentations utilisant successivement les technologies Spring HttpInvoker, Spring Caucho Hessian, Spring Caucho Burlap et Spring RMI.

3.2 Les méthodes exposées dans la couche [proxyServeur]

Reprenons l'architecture de l'application [magasin-client-serveur] :



Le fait que les couches [ui] et [domain] communiquent désormais via un réseau TCP-IP a des conséquences sur l'interface présentée par la couche 1 [Proxyserveur]. Rappelons qu'actuellement la couche [domain] est accédée via l'interface [IArticlesDomain] suivante :

```
20.package istia.st.articles.domain;
21.
22.import istia.st.articles.dao.Article;
23.
24.import java.util.ArrayList;
25.import java.util.List;
26.
27./**
28. * @author serge.tahe@istia.univ-angers.fr
29. *
30. */
31.public interface IArticlesDomain {
32.  // liste de tous les articles
33.  public List getAllArticles();
34.  // obtenir un article particulier
35.  public Article getArticleById(int id);
36.  // acheter un panier
37.  public ArrayList acheter(Panier panier);
38.}
```

`List getAllArticles()`	rend la liste liste d'objets [Article] à présenter au client
`Article getArticleById(int idArticle)`	rend l'objet [Article] identifié par [idArticle]
`ArrayList acheter(Panier panier)`	valide le panier du client en décrémentant les stocks des articles achetés de la quantité achetée - peut échouer si le stock est insuffisant. Rend une liste de messages d'erreurs éventuellement vide.

Supposons que cette interface soit celle exposée par la couche [proxyServeur]. Listons les différentes actions d'un utilisateur et voyons comment celles-ci peuvent être traitées avec la nouvelle architecture :

- **liste des articles**

L'utilisateur demande la liste des articles.

La méthode

```
List getAllArticles();
```

du service distant va être interrogée par la couche 2 [proxyClient]. Elle va rendre une liste d'articles éventuellement vide. Elle peut également lancer une exception si l'accès aux données n'a pu se faire. Les méthodes des services web propagent correctement les exceptions via le réseau. Aussi la couche [proxyClient] pourra gérer l'éventuelle exception. Il n'y a là besoin que d'un échange.

- **demande d'un article particulier**

La méthode

```
Article getArticleById(int idArticle);
```

du service [proxyServeur] va être interrogée par la couche 2 [proxyClient]. Elle va rendre l'article demandé s'il existe ou une exception s'il n'existe pas ou si l'accès aux données s'est révélé impossible. Il n'y a là besoin que d'un échange.

- **gestion du panier**

Autour de la gestion du panier on trouve les actions suivantes :

1. voir le panier
2. ajouter un article dans le panier
3. retirer un article du panier
4. acheter le panier

L'applications [swingarticles] fait gérer les opérations 1, 2 et 3 par sa couche [ui]. L'opération 4 nécessite un accès au service distant. La méthode

```
ArrayList acheter(Panier panier);
```

du service [proxyServeur] va être interrogée par la couche 2 [proxyClient]. Le contenu du panier est modifié par le service distant [proxyServeur]. Après son achat, il est soit vide, soit il contient les articles qui n'ont pu être achetés pour cause de stocks insuffisants. Les erreurs éventuelles sont placées dans le [ArrayList] résultat de la méthode.

La nouvelle architecture ne permet pas de réaliser l'achat du panier correctement : à l'issue de l'échange client-serveur via la méthode [acheter], la couche [ui] s'attend à avoir un panier vidé des articles achetés. Dans l'application [swingarticles], les couches [ui] et [domain] travaillaient sur le même panier car la couche [ui] transmettait à la couche [domain] une **référence** du panier. Ici ce n'est plus le cas. Les échanges client-serveur se font par **valeur** et non par **référence**. La couche [ui] ne peut transmettre à la couche [proxyServeur] qu'une **copie de la valeur du panier**. Après l'échange, le service web sait ce qu'est devenu le contenu du panier. Il doit donc transmettre la nouvelle valeur de celui-ci dans sa réponse.

La couche [proxyServeur] offrira donc la méthode suivante pour l'achat d'un panier :

```
  // achat d'un panier
  public PanierAchete acheterPanier(Panier panier);
```

- le paramètre d'entrée est le panier à acheter. Il a été envoyé par le client via le réseau.
- la réponse est de type [PanierAchete] suivant :

```
1. package istia.st.magasin.remoting.httpinvoker;
2.
3. import istia.st.articles.domain.Panier;
4.
5. import java.io.Serializable;
6. import java.util.ArrayList;
7.
8. /**
9.  * @author serge.tahe@istia.univ-angers.fr
10. *
11. */
12.public class PanierAchete implements Serializable{
13.  // le panier acheté
14.  private Panier panier;
15.
16.  public Panier getPanier() {
17.    return panier;
18.  }
19.
20.  public void setPanier(Panier panier) {
21.    this.panier = panier;
22.  }
23.
24.  // les erreurs d'achat du panier
25.  private ArrayList erreurs;
26.
27.  public ArrayList getErreurs() {
28.    return erreurs;
29.  }
30.
31.  public void setErreurs(ArrayList erreurs) {
32.    this.erreurs = erreurs;
33.  }
34.
35.  // constructeurs
36.  public PanierAchete(){
37.  }
38.
39.  public PanierAchete(Panier panier, ArrayList erreurs){
40.    // init champs privés
41.    setPanier(panier);
42.    setErreurs(erreurs);
43.  }
44.}
```

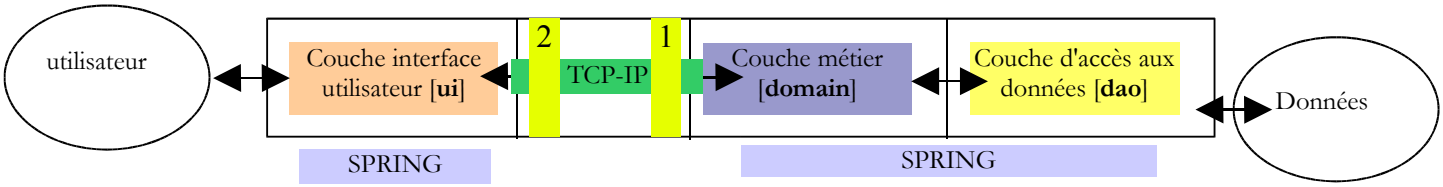
- ligne 12 : la classe implémente l'interface [Serializable]. Le rôle de celle-ci est décrit un peu plus loin paragraphe 3.4. On utilisera également une version de la classe [PanierAchete] n'implémentant pas l'interface [Serializable]. Nous verrons dans quels cas.
- une instance de [PanierAchete] représente le panier d'un client après sa validation. Après celle-ci, le panier a été débarrassé des articles achetés. On retrouvera celui-ci dans le champ privé [panier] de la ligne 14. La validation a pu générer des messages d'erreurs pour les articles dont les stocks étaient insuffisants pour satisfaire la demande du client. On trouvera ces messages dans la liste [erreurs] ligne 25.
- lignes 39-43, on définit un constructeur initialisant les deux champs privés [panier] et [erreurs] de la classe.

La méthode [acheterPanier] permet bien au client de récupérer les deux informations qu'il attend :

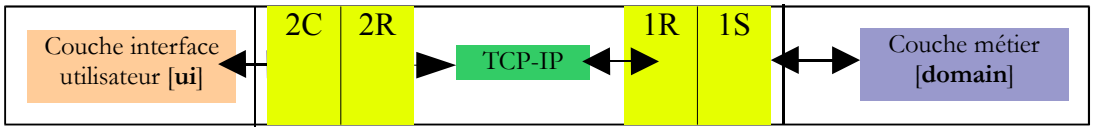
- le nouveau panier après validation
- la liste des erreurs d'achats

3.3 Les couches d'interfaçage avec le réseau

Reprenons l'architecture de l'application [magasin-client-serveur] :



Parce qu'on veut conserver les couches [ui] et [domain] déjà construites, il faut que les couches 1 [proxyServeur] et 2 [proxyClient] soient construites de telle sorte que la couche [ui] "parle" à la couche [domain] comme si elles étaient toutes les deux sur la même machine. Les couches 1 [proxyServeur] et 2 [proxyClient] ont pour rôle de "cacher" le réseau. Spring va nous y aider en nous fournissant des couches d'interfaçage avec le réseau. Les dialogues des couches [ui] et [domain] se feront dans l'architecture suivante :



- la couche 1 [proxyServeur] sera divisée en deux sous-couches :
  - la couche 1R est en contact avec le réseau. Elle nous sera fournie par Spring.
  - la couche 1S est en contact avec la couche [domain]. Nous l'écrirons.
- la couche 2 [proxyClient] sera divisée en deux sous-couches :
  - la couche 2R est en contact avec le réseau. Elle nous sera fournie par Spring.
  - la couche 1C est en contact avec la couche [ui]. Nous l'écrirons.

Au final, Spring va nous fournir les interfaces de communication avec le réseau, celles qui sont les plus délicates à écrire.

3.4 Sérialisation des objets sur le réseau

Les couches [proxyClient] et [proxyServeur] vont échanger des objets :
- liste d'objets [Article] avec la méthode distante **getAllArticles**
- objet [Article] avec la méthode distante **getArticleById**
- objet [Panier], [Achat] et [Article] avec la méthode distante **acheterPanier**
- objet [ArrayList] d'objets [String] avec la méthode distante **getErreurs**

Un objet a une certaine représentation dans la mémoire de l'ordinateur. Lorsqu'il faut le mettre sur le réseau, il faut émettre une suite de bits qui le représente correctement. Cette opération est appelée **sérialisation**. A l'autre bout de la communication, celui qui reçoit l'objet doit faire l'opération inverse. A partir d'une suite de bits, il doit reconstituer l'objet initial. On parle alors de **désérialisation**. Le protocole de sérialisation peut être quelconque. Il faut simplement que les deux partenaires soient d'accord sur celui-ci. Java a son propre protocole de sérialisation. Pour qu'un objet Java soit sérialisable, il faut qu'il implémente l'interface [java.io.Serializable] comme dans la déclaration suivante :

```
public class Panier implements Serializable{
...
}
```

La présence du mot clé [Serializable] est suffisante. Il n'y a pas de méthode à implémenter. C'est un simple marqueur qui indique que les instances de la classe doivent pouvoir être sérialisées.

On peut avoir d'autres types de sérialisation, par exemple une sérialisation XML :

- pour sérialiser un objet sur le réseau, on envoie sur celui-ci sa description XML
- à l'autre bout, la désérialisation XML consiste à lire le flux XML envoyé et à recréer un objet en mémoire à partir de celui-ci.

Les classes [Article], [Panier] et [Achat] utilisées dans [article1] et [article2] n'implémentent pas l'interface [java.io.Serializable]. Ils ne peuvent donc être utilisés tels quels dans notre application client-serveur. C'est dommage. Si on les avait déclarés tout de suite [Serializable], cela aurait été transparent pour les applications des articles [article1, article2, article3] et les couches [domain] et [dao] initiales auraient pu être réutilisées ici sans retouches. Ici, nous ne le pouvons pas.

Pour notre application client-serveur, les sources des classes [Article], [Panier] et [Achat] sont donc modifiées de la façon suivante :

```
public class Article implements Serializable{
... inchangé
}
```

```
public class Achat implements Serializable{
... inchangé
}

public class Panier implements Serializable{
... inchangé
}
```

Ceci fait, les classes composant les deux couches [dao] et [domain] ont été placées dans trois archives :

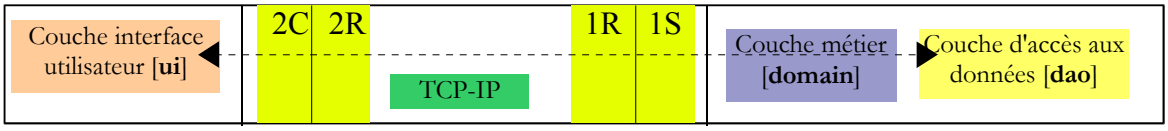
- **istia.st.articles.domain.jar**
- **istia.st.articles.dao.jar**
- **istia.st.articles.exception.jar**

Ce sont ces trois archives qui sont utilisées par la suite.

4 [magasin-client-serveur] avec Spring HttpInvoker

4.1 Architecture

Reprenons l'architecture de l'application [magasin-client-serveur] :



Les couches d'adaptation 1R [proxyServeur] et 2R [proxyClient] sont implémentées avec la technologie **Spring HttpInvoker**. Derrière ce sigle, se cache l'architecture suivante :

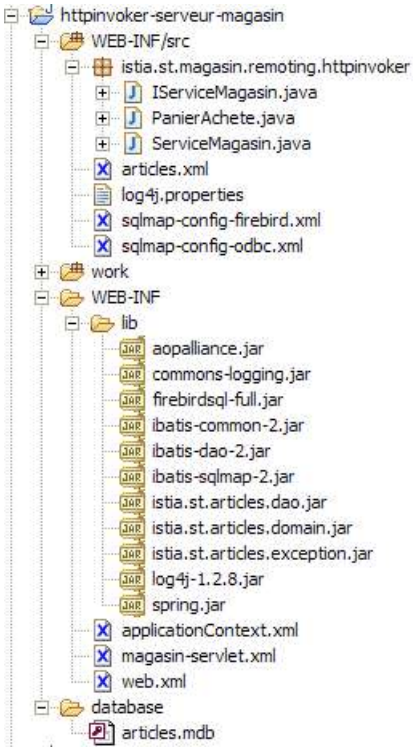
- les couches réseau client [2R] et serveur [1R] communiquent selon le protocole HTTP.
- la couche [1R] qui interface le serveur avec le réseau est assurée par une servlet Spring [HttpInvokerServiceExporter]. Elle se tient donc derrière un serveur HTTP, plus exactement un conteneur de servlets. Ici ce sera Tomcat 5.0. La servlet est paramétrée par deux informations :
  - l'interface qui est exportée. Elle sait ainsi quelles demandes elle doit satisfaire. Les autres seront rejetées.
  - la classe d'implémentation de cette interface. On trouvera celle-ci dans la couche [1S] que nous avons à écrire.
  - la servlet [1R] déléguera le traitement des appels clients à la couche [1S]. Elle le fera sous la forme d'appels de méthodes. Ainsi la classe d'implémentation de la couche [1S] n'a pas à se soucier du réseau. La demande d'accès aux données sera transmise à la couche [domain] puis à la couche [dao]. Les résultats de cette demande feront le chemin inverse pour arriver jusqu'à la couche [1R] qui les mettra sur le réseau où elles seront transportées selon le protocole HTTP.
- la couche [2R] interface le client avec le réseau. C'est une classe, là encore, fournie par Spring : [HttpInvokerProxyFactoryBean]. Elle est paramétrée par deux informations :
  - l'url du service HTTP à atteindre.
  - l'interface implémentée par cette couche. C'est la même que celle exportée par la couche serveur [1R]
- la couche [2C] est à construire par nous. Elle doit nécessairement implémenter l'interface [IArticlesDomain] définie dans [article1] parce que la couche [ui] a été construite pour parler à cette interface. Pour implémenter [IArticlesDomain] et donc obtenir des données venant de la couche [dao], la couche [2C] fera appel à la couche [2R] qui enverra la requête au serveur HTTP. Elle rendra les données qu'elle trouvera dans la réponse HTTP reçue, à la couche [2C] sous la forme des objets que celle-ci attend. Ainsi la couche cliente [2C] est ignorante de la présence du réseau.

Voyons maintenant la mise en oeuvre de cette solution sous Eclipse.

4.2 Le serveur

4.2.1 Le projet Eclipse du serveur

Le projet Eclipse du serveur est le suivant :

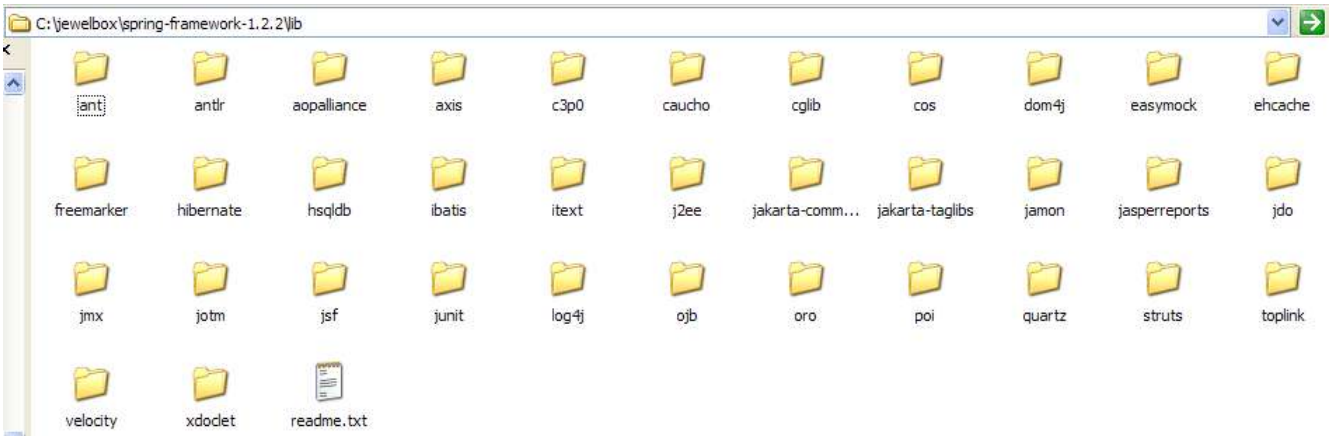


C'est un projet [Sysdeo Tomcat] créé avec le plugin [Sysdeo Tomcat] qui permet de développer des solutions Tomcat sous Eclipse. Nous détaillons tout d'abord les éléments que nous allons rencontrer dans toutes les implémentations de l'application [magasin-client-serveur] :

- [articles.xml], [sqlmap-config-odbc.xml] sont les fichiers nécessaires à l'implémentation [Ibatis SqlMap] de la couche [dao]. Ils ont été décrits : [articles.xml] page 10 et [sqlmap-config-odbc.xml] page 9. [sqlmap-config-odbc.xml] est configuré pour gérer une source ODBC appelée [odbc-access-dvp-articles]. Au moment des tests, le lecteur devra créer cette source ODBC au format indiqué paragraphe 2.3.1, page 5. Une base ACCESS exemple est fournie dans le dossier [database].
- [sqlmap-config-firebird.xml] est un autre fichier de configuration d'Ibatis SqlMap permettant de gérer une source Firebird. Ce fichier est donné à titre d'exemple. Il n'est pas utilisé par l'application [magasin-client-serveur]. Le choix du fichier de configuration d'Ibatis SqlMap à utilliser est fait dans le fichier [applicationContext.xml].
- [log4j.properties] est réclamé par Spring pour ses logs. Il n'est pas indispensable. Son absence n'empêche pas le serveur de fonctionner. Son contenu est le suivant :

```
# Global logging configuration
log4j.rootLogger=INFO, stdout

# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

- l'archive [spring.jar] contient toutes les classes de Spring. La distribution de Spring vient avec cette archive qui contient tout, ainsi qu'avec plusieurs archives plus petites qui implémentent chacune un aspect de Spring (aop, core, remoting, ...). Cette solution permet de n'utiliser que les seules archives dont on a besoin. Elle est préférable mais n'a pas été utilisée ici. La version de Spring utilisée est la 1.2.2.
- les archives [aopalliance.jar, commons-logging.jar, log4j-1.2.8.jar] sont nécessaires à Spring et sont trouvées dans le dossier [lib] de la distribution de Spring :



- les archives [ibatis*.jar] sont nécessaires au framework [Ibatis SqlMap]. On les trouve dans la distribution d'Ibatis.
- les archives [istia.st.articles.*.jar] sont les archives des couches [domain] et [dao] de l'application.
- l'archive [firebirdsql-full.jar] contient le pilote JDBC du SGBD Firebird, pour le cas où l'utilisateur disposant de ce SGBD voudrait utiliser le fichier de configuration [sqlmap-config-firebird.xml].

Les éléments spécifiques à l'implémentation [Spring HttpInvoker] sont les suivants :

- **IServiceMagasin** : l'interface exposée par le serveur à ses clients distants
- **ServiceMagasin** : la classe d'implémentation de cette interface
- **PanierAchete** : l'objet renvoyé au client après l'achat d'un panier
- **web.xml** : le descripteur de l'application web qui assure le service distant
- **magasin-servlet.xml** : le descripteur de la servlet à qui le service va être délégué
- **applicationContext.xml** : la description des objets serveur manipulés par Spring

Nous décrivons maintenant ces différents éléments.

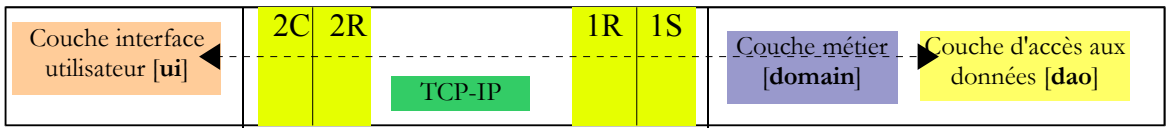
4.2.2 Classes et interfaces du serveur

La classe [**PanierAchete**] a été décrite au paragraphe 3.2, page 23.

L'interface [**IServiceMagasin.java**] est la suivante :

```
1. package istia.st.magasin.remoting.httpinvoker;
2.
3. import istia.st.articles.dao.Article;
4. import istia.st.articles.domain.Panier;
5.
6. import java.util.List;
7. /**
8.  * @author serge.tahe@istia.univ-angers.fr
9.  *
10. */
11.public interface IServiceMagasin {
12.  // liste des articles
13.  public List getAllArticles();
14.  // un article particulier
15.  public Article getArticleById(int idArticle);
16.  // achat d'un panier
17.  public PanierAchete acheterPanier(Panier panier);
18.}
```

Rappelons l'architecture de l'application :



L'interface [IServiceMagasin] est l'interface que présente la couche [1S] à la couche [1R]. Par ailleurs, on sait que la couche [domain] présente l'interface [IArticlesDomain] à la couche [1S]. Pourquoi ne pas exposer directement l'interface [IArticlesDomain] à la couche [1R], rendant ainsi inutile la couche [1S]. Cela a été discuté au paragraphe 3.2, page 21. Nous y indiquions que la méthode [acheter(Panier panier)] de l'interface [IArticlesDomain] ne convenait pas pour être exposée aux clients distants. Ceux-ci s'attendent

en effet à avoir en retour le même panier débarrassé des achats validés. Or la méthode [acheter(Panier panier)] ne rend aucun résultat. L'interface [IServiceMagasin] est donc là pour remédier à ce problème.

- ligne 13 : la méthode permettant d'avoir tous les articles
- ligne 15 : la méthode permettant d'avoir un article particulier
- ligne 17 : la méthode permettant d'acheter un panier

La classe [**ServiceMagasin.java**] d'implémentation de l'interface [**IServiceMagasin.java**] est la suivante :

```
1. package istia.st.magasin.remoting.httpinvoker;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. import istia.st.articles.dao.Article;
7. import istia.st.articles.domain.IArticlesDomain;
8. import istia.st.articles.domain.Panier;
9.
10./**
11. * @author serge.tahe@istia.univ-angers.fr
12. *
13. */
14.public class ServiceMagasin implements IServiceMagasin {
15.
16.  // objet d'accès à la couche [domain]
17.  private IArticlesDomain articlesDomain;
18.
19.  public IArticlesDomain getArticlesDomain() {
20.    return articlesDomain;
21.  }
22.
23.  public void setArticlesDomain(IArticlesDomain articlesDomain) {
24.    this.articlesDomain = articlesDomain;
25.  }
26.
27.  // achat d'un panier
28.  public PanierAchete acheterPanier(Panier panier) {
29.    // achat à la couche domain
30.    ArrayList erreurs=articlesDomain.acheter(panier);
31.    // création du panier résultat
32.    PanierAchete remotePanier=new PanierAchete(panier,erreurs);
33.    // on rend le résultat
34.    return remotePanier;
35.  }
36.
37.  // liste des articles
38.  public List getAllArticles() {
39.    return articlesDomain.getAllArticles();
40.  }
41.
42.  // un article particulier
43.  public Article getArticleById(int idArticle) {
44.    return articlesDomain.getArticleById(idArticle);
45.  }
46.}
```

- ligne 14 : la classe implémente l'interface [IServiceMagasin]
- lignes 16-25 : on définit une référence privée [articlesDomain] à un objet permettant d'accéder à la couche [domain]. Cet objet sera créé par Spring et injecté par configuration dans la classe [ServiceMagasin]. Les méthodes get et set associées sont définies. L'objet [articlesDomain] implémente l'interface [IArticlesDomain] de la couche [domain].
- ligne 39 : la liste des articles est demandée à la couche [domain]
- ligne 44 : l'article demandé par le client est demandé à la couche [domain]
- ligne 28-35 : le client demande l'achat d'un panier. Il veut en retour le panier sans les articles qui ont pu être achetés (stocks suffisants) ainsi que la liste des erreurs éventuelles.
  - ligne 30 : on demande à la couche [domain] l'achat du panier. Celle-ci va supprimer du panier les articles achetés. Elle rend par ailleurs la liste des erreurs d'achats éventuelles.
  - ligne 32 : le nouveau panier et la liste des erreurs sont placés dans un objet de type [PanierAchete]
  - ligne 24 : on rend cet objet au client

4.2.3 Configuration de l'application web du serveur

Rappelons que le service est rendu par une application web. Les fichiers de configuration de celle-ci sont dans le dossier [WEB-INF] du projet Eclipse :

4.2.3.1 Le descripteur de déploiement [web.xml]

Son contenu est le suivant :

```
1. <?xml version="1.0" encoding="ISO-8859-1"?>
2.
3. <!DOCTYPE web-app PUBLIC
4.  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5.  "http://java.sun.com/dtd/web-app_2_3.dtd">
6. <web-app>
7.  <!-- le chargeur du contexte spring de l'application -->
8.  <listener>
9.    <listener-class>
10.     org.springframework.web.context.ContextLoaderListener</listener-class>
11. </listener>
12. <!-- la servlet -->
13. <servlet>
14.   <servlet-name>magasin</servlet-name>
15.   <servlet-class>
16.     org.springframework.web.servlet.DispatcherServlet</servlet-class>
17. </servlet>
18. <!-- le mapping des url -->
19. <servlet-mapping>
20.   <servlet-name>magasin</servlet-name>
21.   <url-pattern>/run</url-pattern>
22. </servlet-mapping>
23.</web-app>
```

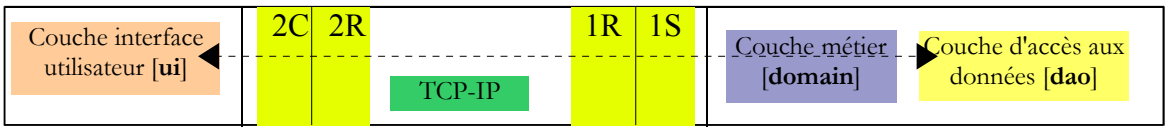
- lignes 8-11, un listener est défini. Un listener est averti de différents événements de l'application web à laquelle il appartient. Il sera ainsi averti du démarrage initial de l'application. Le listener est ici une classe fournie par [Spring]. Son rôle est de lire le fichier [WEB-INF/applicationContext.xml] et d'instancier les beans Spring qu'il va y trouver.
- lignes 13-17 : on définit une servlet appelée [magasin]. Cette servlet est [DispatcherServlet], une servlet prédéfinie de [Spring]. Une fois chargée, cette servlet va lire le fichier [magasin-servlet.xml] qui configure la servlet [magasin]. De façon générale, une servlet X est configurée par un fichier [WEB-INF/X-servlet.xml]. La servlet [DispatcherServlet] reçoit toutes les demandes des clients et délègue leur traitement à d'autres classes dont elle trouve la définition et la configuration dans le fichier [magasin-servlet.xml].
- lignes 19-22 : on dit que lorsqu'un client demande à l'application web, une URL terminée par /run, alors cette demande doit être traitée par la servlet [magasin] définie aux lignes 13-17. Les autres demandes seront refusées.

4.2.3.2 Le contexte de l'application web Spring [applicationContext.xml]

Une application web a un contexte qui est un ensemble d'objets partagés par tous les clients de l'application. Ce contexte est accessible à tous les autres objets de l'application web. Une application web Spring permet d'initialiser ce contexte avec le fichier [applicationContext.xml]. Celui de notre application est le suivant :

```
1. <?xml version="1.0" encoding="ISO-8859-1"?>
2. <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3. <beans>
4.  <!-- la classe d'accès aux données -->
5.  <bean id="articlesDao" class="istia.st.articles.dao.ArticlesDaoSqlMap">
6.    <constructor-arg index="0">
7.      <value>sqlmap-config-odbc.xml</value>
8.        <!-- avec une base Firebird
9.      <value>sqlmap-config-firebird.xml</value>
10.       -->
11.   </constructor-arg>
12. </bean>
13. <!-- la classe métier -->
14. <bean id="articlesDomain" class="istia.st.articles.domain.AchatsArticles">
15.   <constructor-arg index="0">
16.     <ref bean="articlesDao"/>
17.   </constructor-arg>
18. </bean>
19. <!-- la classe proxy -->
20. <bean id="serviceMagasin" class="istia.st.magasin.remoting.httpinvoker.ServiceMagasin">
21.   <property name="articlesDomain">
22.     <ref bean="articlesDomain"/>
23.   </property>
24. </bean>
25.</beans>
```

On a là un fichier de configuration Spring classique qui définit l'instanciation de trois objets. Chacun d'eux correspond à l'une des couches [1S, domain, dao] du serveur :



- **lignes 5-11, articlesDao** : l'objet d'accès à la couche [dao], implémenté ici par la classe [ArticlesDaoSqlMap]. Celle-ci a un constructeur à un paramètre : le nom du fichier de configuration d'Ibatis SqlMap. Le fichier choisi ici, est celui qui utilise une source ODBC. Il a été défini page 9.
- **lignes 14-18, articlesDomain** : l'objet d'accès à la couche [domain], implémenté ici par la classe [AchatsArticles]. Celle-ci a un constructeur à un paramètre : l'objet permettant l'accès à la couche [dao], ici l'objet [**articlesDao**], défini lignes 5-11.
- **lignes 20-24, serviceMagasin** : la classe d'implémentation de la couche [1S], implémenté ici par la classe [ServiceMagasin] décrite page 28. On sait que cette classe est un Javabean ayant une propriété **IArticlesDomain articlesDomain**. Celle-ci est initialisée avec l'objet [**articlesDomain]** défini lignes 14-18.

Ces trois objets sont instanciés par Spring avant toute autre objet de l'application web.

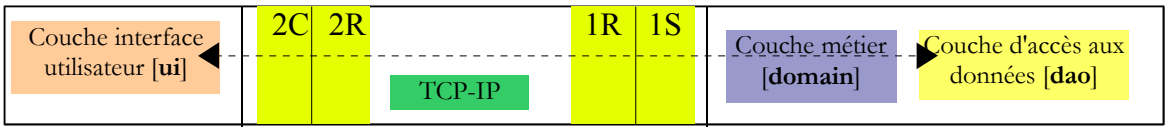
4.2.3.3 Le fichier de configuration [magasin-servlet.xml] de la servlet [magasin]

Le fichier [magasin-servlet.xml] sert à paramétrer la servlet [magasin] qui est utilisée pour traiter l'url /run. Son contenu est le suivant :

```
1. <?xml version="1.0" encoding="ISO-8859-1"?>
2. <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
3. <beans>
4.  <bean id="defaultHandlerMapping"
5.    class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
6.  <bean name="/run"
7.    class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
8.    <property name="service">
9.      <ref bean="serviceMagasin"/>
10.   </property>
11.   <property name="serviceInterface">
12.     <value>istia.st.magasin.remoting.httpinvoker.IServiceMagasin</value>
13.   </property>
14. </bean>
15.</beans>
```

On a de nouveau affaire à un fichier de configuration Spring.

- lignes 4-5 : elles définissent un mapping URL <--> classe de traitement de cette URL. Supposons que le nom de l'application web soit APPLIWEB et que l'url demandée par le client soit [http://machine:8080/APPLIWEB/x/y/z]. L'URL demandée est ici [/x/y/z]. Les lignes 4-5, disent que cette url doit être traitée par le bean appelé [/x/y/z]. Spring a d'autres types de mapping. Rappelons que n'est traitée par la servlet [magasin] que l'url [/run] (cf fichier web.xml). Donc un seul bean est défini ici, le bean [/run], ligne 6.
- lignes 6-14 : un bean appelé [/run] est défini. Sa classe d'implémentation est une classe prédéfinie de Spring [org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter]. C'est elle qui assure l'implémentation de la couche [1R] du serveur :



- la classe [HttpInvokerServiceExporter] a besoin de connaître deux choses :
  - lignes 8-10 : la propriété [service] de la classe [HttpInvokerServiceExporter] est une référence sur la classe chargée de traiter la demande du client. Pour nous ici, il s'agit de la classe [ServiceMagasin] d'implémentation de la couche [1S] définie dans le fichier [applicationContext.xml] sous le nom de bean [serviceMagasin]. On indique donc cette référence en ligne 9. Dans la balise [<ref bean="serviceMagasin"/>], le mot clé [bean] indique que le bean peut être cherché dans d'autres fichiers de configuration de [Spring]. Ici, il sera trouvé dans [applicationContext.xml]. Si on avait écrit [<ref local="serviceMagasin"/>], le bean aurait été cherché uniquement dans [magasin-servlet.xml] et n'aurait alors pas été trouvé.
  - lignes 11-13 : le nom de l'interface qui doit être exposée aux clients distants. Ici c'est l'interface [IServiceMagasin] définie page 27.
  - si on regarde le schéma ci-dessus, on voit que les deux informations nécessitées par la classe [HttpInvokerServiceExporter] permettent à la classe de savoir ce qu'elle doit présenter comme interface à ses clients sur sa gauche et à quelle classe elle doit passer la requête du client sur sa droite.

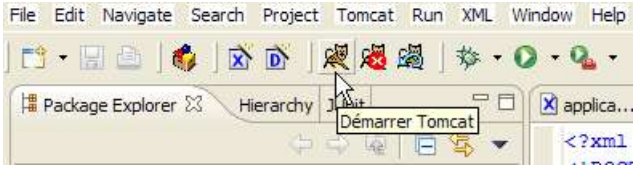
4.2.4 Lancement du serveur

Nous sommes prêts pour les premiers tests. Pour lancer le service [ServiceMagasin], il nous faut tout d'abord lancer un serveur HTTP conteneur de servlets. Comme nous l'avons dit, nous utiliserons ici un serveur Tomcat 5.0.27. Nous pouvons utiliser deux méthodes pour lancer ce serveur :

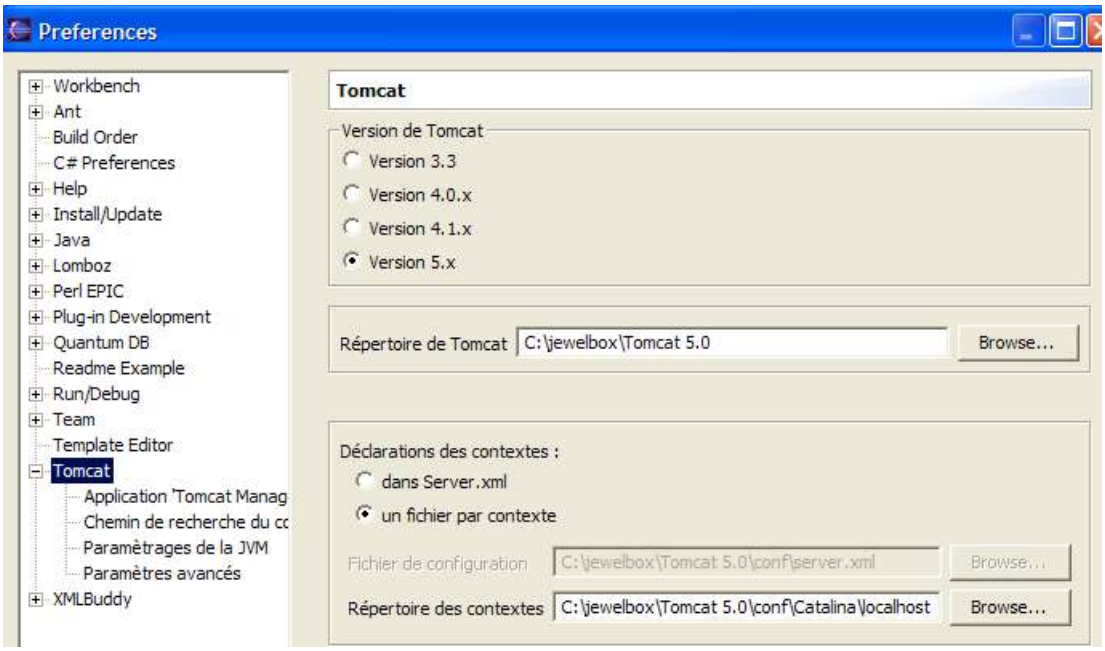
- depuis Eclipse
- directement sans passer par Eclipse

4.2.4.1 Lancement du serveur Tomcat à partir d'Eclipse

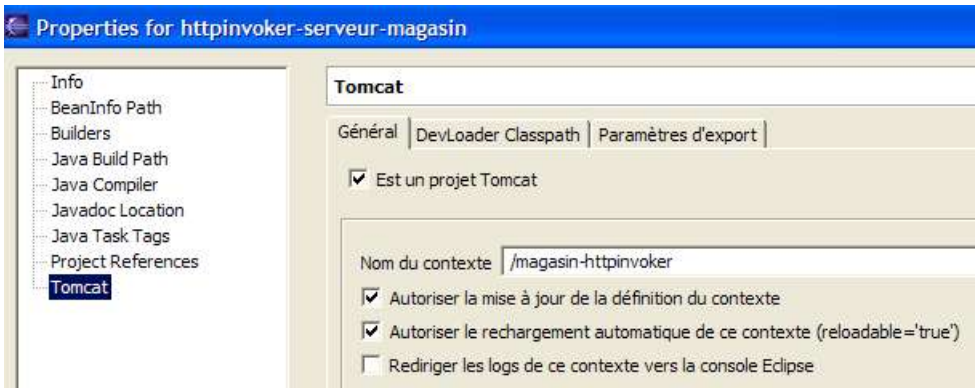
Nous lançons Tomcat à partir d'Eclipse de la façon suivante :



Ceci fait, Tomcat est lancé selon la configuration du plugin [Sysdeo Tomcat] :



Tomcat lancé, il charge toutes les applications web pour lesquelles il a été configuré. Le nom du contexte web de l'application [magasin-client-serveur] est obtenu dans la page de propriétés du projet Eclipse [httpinvoker-serveur-magasin] :



Le nom de l'application web est donc [/magasin-httpinvoker]. Cela signifie que le client devra demander des URL de la forme [http://machine:8080/magasin-httpinvoker/url]. Nous avons vu dans le fichier [web.xml] que la seule url acceptée par l'application était l'url [/run]. Un client sur la même machine que le serveur demandera donc l'url [http://localhost:8080/magasin-httpinvoker/run].

Une fois lancé sous Eclipse, Tomcat affiche des logs dans la fenêtre [Console] d'Eclipse. Dans les logs, les lignes concernant l'application web [/magasin-httpinvoker] sont les suivantes :

```
1. INFO: Processing Context configuration file URL file:C:\jewelbox\Tomcat
   5.0\conf\Catalina\localhost\magasin-httpinvoker.xml
2.  INFO [main] - Root WebApplicationContext: initialization started
3.  INFO [main] - Loading XML bean definitions from ServletContext resource [/WEB-
   INF/applicationContext.xml]
4.  INFO [main] - Bean factory for application context [Root WebApplicationContext]:
   org.springframework.beans.factory.support.DefaultListableBeanFactory defining beans
   [articlesDao,articlesDomain,serviceMagasin]; root of BeanFactory hierarchy
5.  INFO [main] - 3 beans defined in application context [Root WebApplicationContext]
6.  INFO [main] - JDK 1.4+ collections available
7.  INFO [main] - Unable to locate MessageSource with name 'messageSource': using default
   [org.springframework.context.support.DelegatingMessageSource@c9f997]
8.  INFO [main] - Unable to locate ApplicationEventMulticaster with name 'applicationEventMulticaster':
   using default [org.springframework.context.event.SimpleApplicationEventMulticaster@17c7988]
9.  INFO [main] - No ThemeSource found for [Root WebApplicationContext]: using ResourceBundleThemeSource
10. INFO [main] - Pre-instantiating singletons in factory
   [org.springframework.beans.factory.support.DefaultListableBeanFactory defining beans
   [articlesDao,articlesDomain,serviceMagasin]; root of BeanFactory hierarchy]
11. INFO [main] - Creating shared instance of singleton bean 'articlesDao'
12. INFO [main] - Creating shared instance of singleton bean 'articlesDomain'
```

```
13. INFO [main] - Creating shared instance of singleton bean 'serviceMagasin'
14. INFO [main] - Using context class [org.springframework.web.context.support.XmlWebApplicationContext]
   for root WebApplicationContext
15. INFO [main] - Root WebApplicationContext: initialization completed in 1812 ms
```

- ligne 1 : Tomcat traite le fichier de configuration de l'application [/magasin-httpinvoker]. Ce fichier a été généré par le plugin [Sysdeo Tomcat]. Si on n'utilise pas ce plugin, ce fichier doit être généré à la main et placé dans le dossier [<tomcat5>\conf\Catalina\localhost] :

Son contenu est le suivant :

```
1. <Context path="/magasin-httpinvoker" reloadable="true"
2. docBase="D:\data\serge\travail\2004-2005\webjava\client-serveur-magasin\httpinvoker-serveur-magasin"
3. workDir="D:\data\serge\travail\2004-2005\webjava\client-serveur-magasin\httpinvoker-serveur-magasin\work" />
```

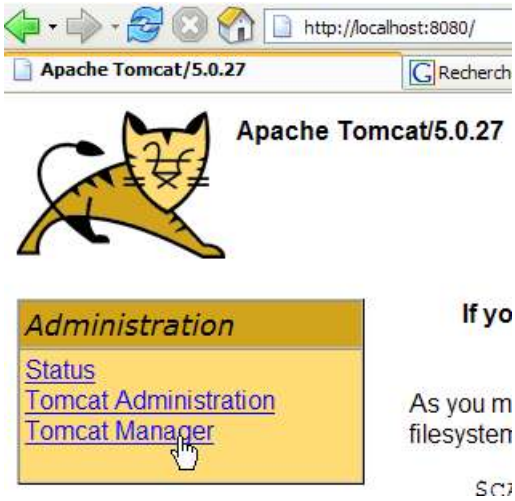
- ligne 1 : **path** désigne le nom de l'application web configurée
- ligne 1 : **reloadable=true** indique que l'application web peut être rechargée à chaud, c.a.d. sans avoir à relancer Tomcat
- ligne 2 : **docBase** désigne le dossier physique de l'application web
- ligne 3 : **workDir** désigne un répertoire de travail où Tomcat pourra stocker un certain nombre de fichiers, nécessaires à l'exécution de l'application web.

Revenons aux logs de Tomcat dans la console d'Eclipse :

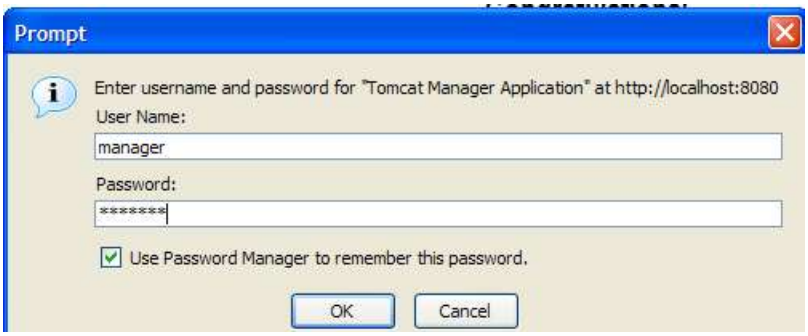
- ligne 2 : le listener [org.springframework.web.context.ContextLoaderListener] défini dans [web.xml] est averti par Tomcat du lancement de l'application. Il signale son chargement par un log.
- ligne 3 : le listener signale qu'il va exploiter le fichier [applicationContext.xml].
- ligne 4 : il indique les beans qu'il a trouvés dans le fichier [applicationContext.xml].
- lignes 5-9 : divers logs
- lignes 10-13 : instanciation des trois singletons trouvés dans le fichier [applicationContext.xml].

Si le fichier [applicationContext.xml] est erroné, les erreurs seront indiquées dans les logs. Les messages d'erreurs de Spring sont précis et permettent de localiser aisément les erreurs du fichier de configuration.

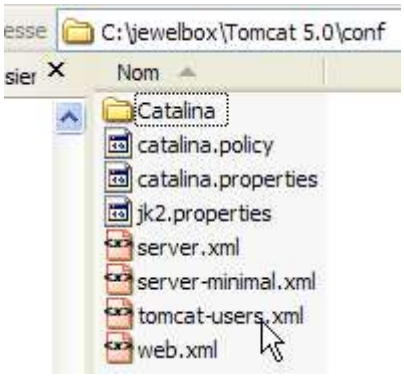
Le déploiement de l'application peut être vérifié avec un navigateur à l'url [http://localhost:8080/] :



Suivons le lien [Tomcat Manager] :



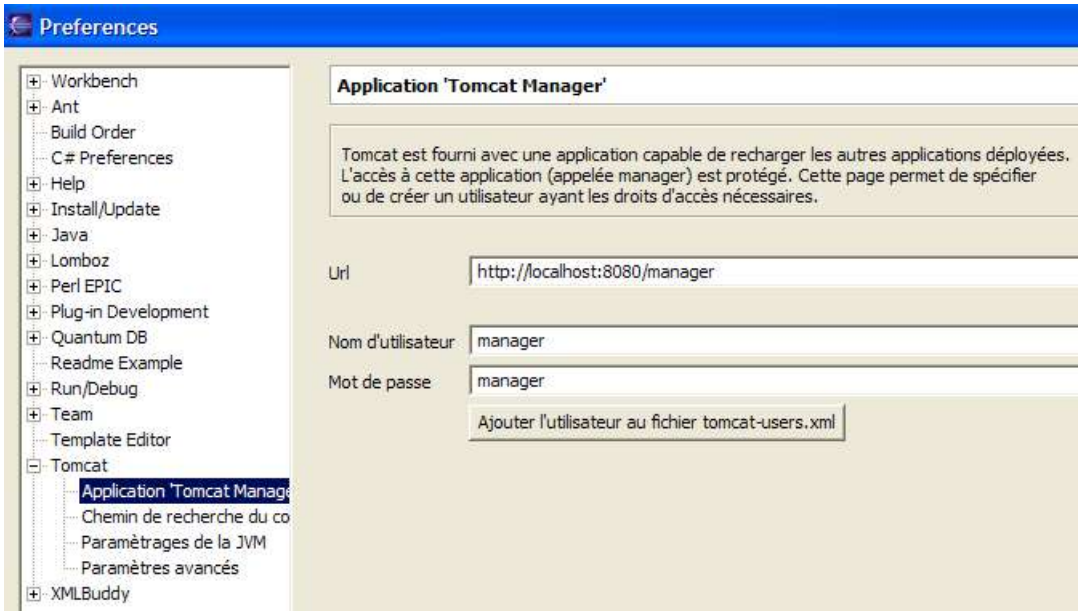
Le couple login - mot de passe demandé ici est défini dans le fichier [tomcat-users.xml] du dossier <tomcat5>\conf :



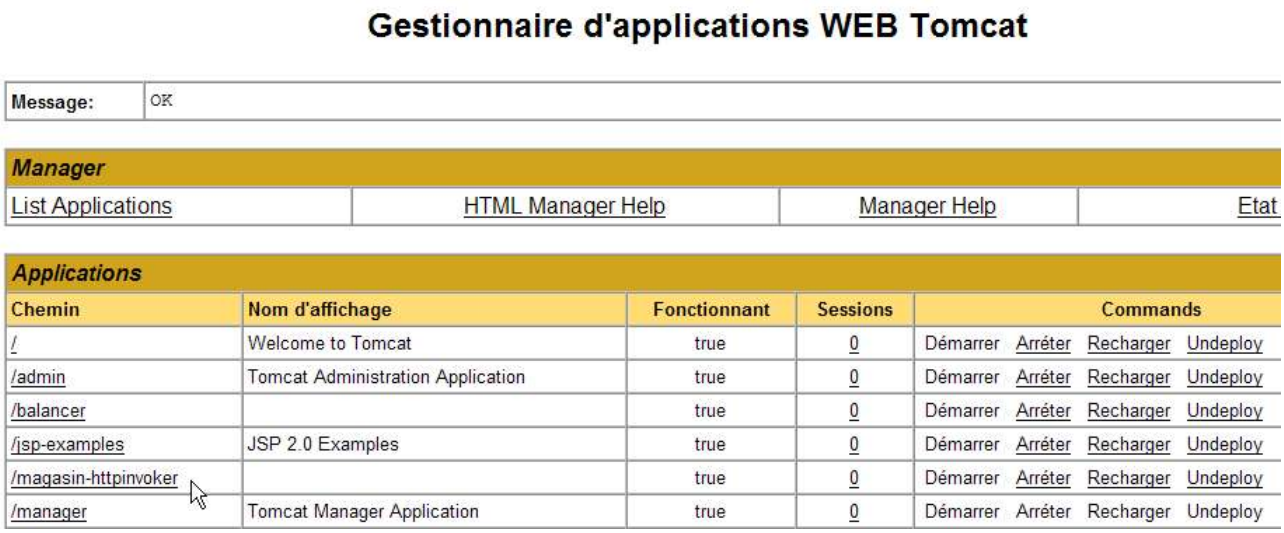
Dans notre exemple, son contenu est le suivant :

```
1. <?xml version='1.0' encoding='utf-8'?>
2. <tomcat-users>
3.   <role rolename="tomcat"/>
4.   <role rolename="role1"/>
5.   <role rolename="manager"/>
6.   <role rolename="admin"/>
7.   <user username="tomcat" password="tomcat" roles="tomcat"/>
8.   <user username="both" password="tomcat" roles="tomcat,role1"/>
9.   <user username="role1" password="tomcat" roles="role1"/>
10.  <user username="manager" password="manager" fullName="" roles="manager"/>
11.  <user username="admin" password="admin" roles="admin,manager"/>
12.</tomcat-users>
```

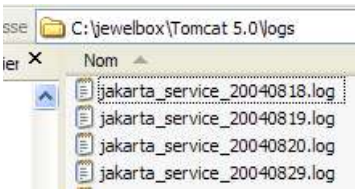
Nous devons utiliser un utilisateur disposant du rôle [manager] défini en ligne 5. La ligne 10 définit un tel utilisateur. C'est celui qui a été utilisé sur la copie d'écran plus haut. On aurait pu également utiliser l'utilisateur [admin] de la ligne 11. Le plugin [Sysdeo Tomcat] d'Eclipse utilise cette information afin de pouvoir recharger les applications web "à chaud". Seul le rôle [manager] peut faire cela. Cette information fait partie de la configuration du plugin :



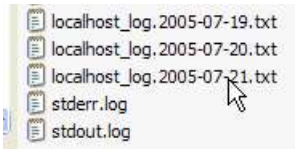
Après nous être identifié suite au clic sur le lien [Tomcat Manager], nous obtenons la page suivante :



On voit que l'application web [/magasin-httpinvoker] a été correctement déployée. Elle va maintenant pouvoir répondre aux demandes des clients. Diverses choses peuvent alors mal se passer côté serveur. Le client reçoit un message généraliste du genre [Internal server error] qui le laisse perplexe. Il faut alors aller voir du côté du serveur. On ne trouvera le plus souvent rien dans la console d'Eclipse. Il faut alors aller voir les logs du serveur Tomcat dans le dossier <tomcat5>\logs :



Il y a là plusieurs fichiers de logs. Celui qui nous intéresse est celui qui contient la date du jour :



Ouvrons-le :

```
1. 2005-07-21 09:13:29 StandardContext[/balancer]org.apache.webapp.balancer.BalancerFilter: init():
   ruleChain: [org.apache.webapp.balancer.RuleChain: [org.apache.webapp.balancer.rules.URLStringMatchRule:
   Target string: News / Redirect URL: http://www.cnn.com],
   [org.apache.webapp.balancer.rules.RequestParameterRule: Target param name: paramName / Target param
   value: paramValue / Redirect URL: http://www.yahoo.com],
   [org.apache.webapp.balancer.rules.AcceptEverythingRule: Redirect URL: http://jakarta.apache.org]]
2. 2005-07-21 09:13:30 StandardContext[/magasin-httpinvoker]Loading Spring root WebApplicationContext
3. 2005-07-21 09:13:33 StandardContext[/jsp-examples]ContextListener: contextInitialized()
4. 2005-07-21 09:13:33 StandardContext[/jsp-examples]SessionListener: contextInitialized()
5. 2005-07-21 09:13:33 StandardContext[/servlets-examples]ContextListener: contextInitialized()
6. 2005-07-21 09:13:33 StandardContext[/servlets-examples]SessionListener: contextInitialized()
7. 2005-07-21 09:44:39 StandardContext[/manager]HTMLManager: init: Associated with Deployer 'localhost'
8. 2005-07-21 09:44:39 StandardContext[/manager]HTMLManager: init: Global resources are available
9. 2005-07-21 09:44:39 StandardContext[/manager]HTMLManager: list: Listing contexts for virtual host
   'localhost'
```

Tout cela a un côté délicieusement obscur mais c'est quand même là qu'il faut chercher lorsqu'on obtient côté client la fameuse erreur [Internal server error]. Une erreur fréquente est que l'application web a eu besoin d'une classe qu'il n'a pas trouvée. Son archive a été oubliée dans [WEB-INF/lib]. Comme cette classe n'est pas chargée au moment du démarrage de l'application, celui-ci se passe bien et on a l'impression que l'application est correctement déployée. C'est le cas. Seulement lorsqu'elle va avoir besoin de la classe manquante, elle va générer l'erreur [Internal server error] au client. C'est dans les logs de Tomcat qu'on trouvera le nom de la classe manquante qui a provoqué la défaillance de l'application.

Que se passe-t-il maintenant que le serveur est lancé, si nous demandons l'url [http://localhost:8080/magasin-httpinvoker/run] ? Regardons :

Ca ne se passe pas très bien... Regardons d'abord les logs dans la console d'Eclipse :

```
1.  INFO [http-8080-Processor23] - Initializing servlet 'magasin'
2.  INFO [http-8080-Processor23] - FrameworkServlet 'magasin': initialization started
3.  INFO [http-8080-Processor23] - Loading XML bean definitions from ServletContext resource [/WEB-
   INF/magasin-servlet.xml]
4.  INFO [http-8080-Processor23] - Bean factory for application context [WebApplicationContext for
   namespace 'magasin-servlet']: org.springframework.beans.factory.support.DefaultListableBeanFactory
   defining beans [defaultHandlerMapping,/run]; parent:
   org.springframework.beans.factory.support.DefaultListableBeanFactory defining beans
   [articlesDao,articlesDomain,serviceMagasin]; root of BeanFactory hierarchy
5.  INFO [http-8080-Processor23] - 2 beans defined in application context [WebApplicationContext for
   namespace 'magasin-servlet']
6.  INFO [http-8080-Processor23] - Unable to locate MessageSource with name 'messageSource': using default
   [org.springframework.context.support.DelegatingMessageSource@1c8f91e]
7.  INFO [http-8080-Processor23] - Unable to locate ApplicationEventMulticaster with name
   'applicationEventMulticaster': using default
   [org.springframework.context.event.SimpleApplicationEventMulticaster@15b6116]
8.  INFO [http-8080-Processor23] - No ThemeSource found for [WebApplicationContext for namespace 'magasin-
   servlet']: using ResourceBundleThemeSource
9.  INFO [http-8080-Processor23] - Pre-instantiating singletons in factory
   [org.springframework.beans.factory.support.DefaultListableBeanFactory defining beans
   [defaultHandlerMapping,/run]; parent:
   org.springframework.beans.factory.support.DefaultListableBeanFactory defining beans
   [articlesDao,articlesDomain,serviceMagasin]; root of BeanFactory hierarchy]
10. INFO [http-8080-Processor23] - Creating shared instance of singleton bean 'defaultHandlerMapping'
11. INFO [http-8080-Processor23] - Creating shared instance of singleton bean '/run'
12. INFO [http-8080-Processor23] - CGLIB2 not available: proxyTargetClass feature disabled
13. INFO [http-8080-Processor23] - Using context class
   [org.springframework.web.context.support.XmlWebApplicationContext] for servlet 'magasin'
14. INFO [http-8080-Processor23] - Unable to locate MultipartResolver with name 'multipartResolver': no
   multipart request handling provided
```

```
15. INFO [http-8080-Processor23] - Unable to locate LocaleResolver with name 'localeResolver': using
   default [org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver@1f1e666]
16. INFO [http-8080-Processor23] - Unable to locate ThemeResolver with name 'themeResolver': using default
   [org.springframework.web.servlet.theme.FixedThemeResolver@135877f]
17. INFO [http-8080-Processor23] - No HandlerAdapters found in servlet 'magasin': using default
18. INFO [http-8080-Processor23] - No ViewResolvers found in servlet 'magasin': using default
19. INFO [http-8080-Processor23] - FrameworkServlet 'magasin': initialization completed in 360 ms
20. INFO [http-8080-Processor23] - Servlet 'magasin' configured successfully
21.ERROR [http-8080-Processor23] - Could not complete request
22.java.io.EOFException
23. at java.io.ObjectInputStream$PeekInputStream.readFully(ObjectInputStream.java:2165)
24. at java.io.ObjectInputStream$BlockDataInputStream.readShort(ObjectInputStream.java:2631)
25. at java.io.ObjectInputStream.readStreamHeader(ObjectInputStream.java:734)
26....
```

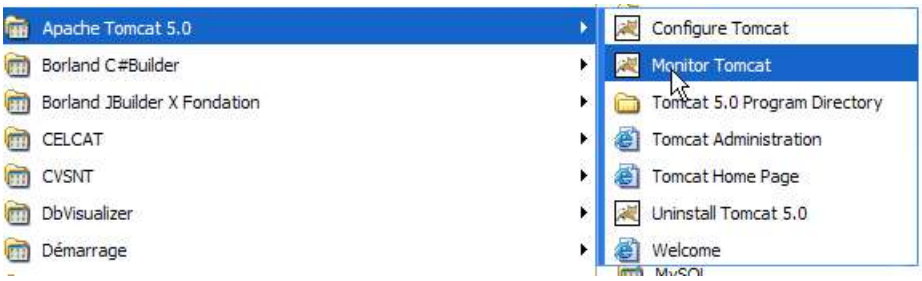
Les logs ci-dessus sont faits par la servlet de l'application web [/magasin-httpinvoker]. Rappelons que la servlet principale de cette application est [DispatcherServlet], une servlet de Spring. Les logs ci-dessus sont donc faits par Spring.

- ligne 1 : l'application web a reconnu que l'url était destinée à la servlet [magasin]. Elle a trouvé cette information dans le fichier [web.xml]. Elle initialise donc cette servlet qui n'avait pas encore été chargée.
- ligne 3 : Spring indique qu'il va exploiter le fichier [magasin-servlet.xml], le fichier associé à la servlet [magasin].
- ligne 4 : Spring indique qu'il a trouvé deux beans [defaultHandlerMapping,/run] dans [magasin-servlet.xml] et que ce fichier a un parent dans lequel sont définis les beans [articlesDao,articlesDomain,serviceMagasin]. Ce sont les beans du fichier [applicationContext.xml] qui eux ont été créés lors du démarrage de l'application. Cette relation parent - enfant entre les fichiers [applicationContext.xml] et [magasin-servlet.xml] signifie que dans [magasin-servlet.xml], on peut référencer des beans définis dans [applicationContext.xml]. C'est ce qui a été fait pour le bean [/run].
- lignes 5-8 : divers logs
- lignes 9-11 : les beans [defaultHandlerMapping,/run] sont instanciés
- lignes 12-18 : divers logs dont certains pourraient être vus comme signalant des erreurs
- lignes 19-20 : on nous dit en fait que la servlet [magasin] a pu être initialisée correctement. Bonne nouvelle.
- lignes 21-22 : catastrophe...

En fait, il n'y a rien d'anormal dans tout cela. On a simplement un client HTTP (le navigateur) qui n'est pas adapté au serveur qu'il a en face de lui. Le serveur attend des requêtes HTTP particulières que ne peut pas lui envoyer un navigateur. Il nous faut un client HTTP particulier que nous allons construire prochainement.

4.2.4.2 Lancement du serveur Tomcat en mode autonome

Plutôt que de lancer Tomcat à partir d'Eclipse, on peut vouloir le lancer directement :

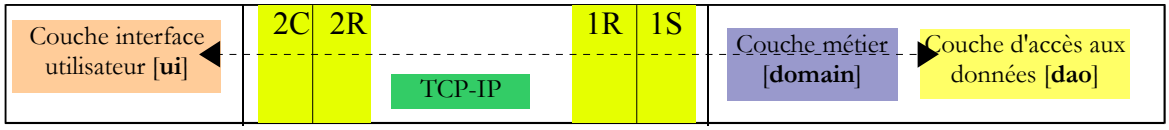


Le moniteur [Tomcat] est un programme qui permet de lancer - arrêter le serveur Tomcat. Son icône se place dans la barre d'état. De là, on lance le service Tomcat :

Les pré-requis nécessaires au lancement réussi de l'application web [/magasin-httpinvoker] sont les mêmes que ceux décrits dans le paragraphe précédent. L'inconvénient de ce mode de lancement est qu'on ne dispose plus de la console d'Eclipse pour les logs de Tomcat et de Spring. Les logs de Spring peuvent être obtenus dans un fichier. Il suffit de configurer dans ce sens le fichier [WEB-INF/src/log4j.properties] du projet Eclipse. Les logs de Tomcat eux restent disponibles dans le dossier <Tomcat5>\logs. Dans la pratique, nous n'utiliserons ce mode qu'après avoir vérifié sous Eclipse que le déploiement de l'application web se faisait correctement.

4.3 Le client

Maintenant que la partie serveur de l'application [magasin-client-serveur] a été construite, nous construisons la partie cliente. Reprenons l'architecture de l'application [magasin-client-serveur] :

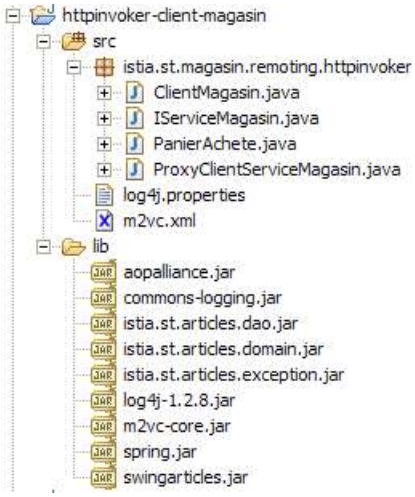


Il s'agit ici de construire la couche 2 qui interface la couche [ui] avec le réseau. La couche 2 a deux composantes :

- la couche [2R] qui s'occupe de la communication réseau. Elle sera implémentée par une classe fournie par Spring. Elle présente à la couche [2C] la même interface [IServiceMagasin] qu'exposée par la couche serveur [1R].
- la couche [2C] sert d'adaptateur entre la couche [ui] qui parle à une interface [IArticlesDomain] et la couche [2R] qui elle présente une interface [IServiceMagasin]. C'est nous qui construirons cette partie de la couche 2.

4.3.1 Le projet Eclipse du client

Le projet Eclipse du client est le suivant :



C'est un projet Java classique. Détaillons son contenu :

- le projet a été configuré pour utiliser les archives du dossier [lib]. On retrouve parmi celles-ci certaines des archives déjà décrites pour le serveur. Nous ne revenons pas dessus. Nous trouvons deux archives directement liées au client [swingarticles] décrit dans [article3] et qui sera notre client dans toutes les applications client-serveur décrites dans ce document :
  - **m2vc-core.jar** : archive du moteur MVC décrit dans [article2] : [M2VC - un moteur MVC pour les applications swing], disponible à l'url [http://tahe.developpez.com/java/m2vc].
  - **swingarticles.jar** : archive du client graphique décrit dans [article3] et modifié au paragraphe 2.3.6 : [Construction d'une application swing MVC à trois couches avec Spring], disponible à l'url [http://tahe.developpez.com/java/swing3tier/]. Ce client est configuré par le fichier [**m2vc.xml**].
- dans le dossier [src], on trouve les éléments suivants :
  - **IServiceMagasin** : l'interface à implémenter par la couche [2R]
  - **ProxyClientServiceMagasin** : la classe implémentant la couche [2C]
  - **ClientMagasin** : le lanceur du client
  - **log4j.properties** : configure les logs de Spring. C'est le même que celui déjà décrit côté serveur.
  - **m2vc.xml** : le fichier de configuration du client riche [swingarticles].

4.3.2 Classes et interfaces du client

La classe [**PanierAchete**] a été décrite au paragraphe 3.2, page 23.

L'interface [**IServiceMagasin.java**] est la même que pour le serveur :

```
1. package istia.st.magasin.remoting.httpinvoker;
```

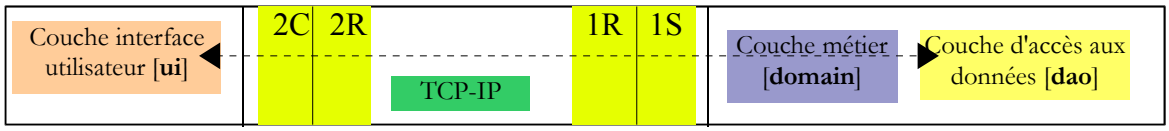
```
2.
3. import istia.st.articles.dao.Article;
4. import istia.st.articles.domain.Panier;
5.
6. import java.util.List;
7. /**
8.  * @author serge.tahe@istia.univ-angers.fr
9.  *
10. */
11.public interface IServiceMagasin {
12.  // liste des articles
13.  public List getAllArticles();
14.  // un article particulier
15.  public Article getArticleById(int idArticle);
16.  // achat d'un panier
17.  public PanierAchete acheterPanier(Panier panier);
18.}
```

Rappelons que cette interface va être ici implémentée par une classe prédéfinie de Spring obtenue par configuration dans [m2vc.xml].

La classe [**ProxyClientServiceMagasin]** est la suivante :

```
1. package istia.st.magasin.remoting.httpinvoker;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. import istia.st.articles.dao.Article;
7. import istia.st.articles.domain.Achat;
8. import istia.st.articles.domain.IArticlesDomain;
9. import istia.st.articles.domain.Panier;
10.
11./**
12. * @author serge.tahe@istia.univ-angers.fr
13. *
14. */
15.public class ProxyClientServiceMagasin implements IArticlesDomain {
16.
17.  // attributs
18.  private IServiceMagasin proxy;
19.  public IServiceMagasin getProxy() {
20.    return proxy;
21.  }
22.  public void setProxy(IServiceMagasin proxy) {
23.    this.proxy = proxy;
24.  }
25.
26.  // achat du panier
27.  public ArrayList acheter(Panier panier) {
28.    // on demande au proxy d'acheter le panier
29.    PanierAchete remotePanier=proxy.acheterPanier(panier);
30.    // on vide le panier originel
31.    for(int i=panier.getAchats().size()-1;i>=0;i--){
32.      panier.enlever(((Achat)panier.getAchats().get(i)).getArticle().getId());
33.    }
34.    // on le remplit avec les achats de remotePanier
35.    for(int i=0;i<remotePanier.getPanier().getAchats().size();i++){
36.      panier.ajouter((Achat)remotePanier.getPanier().getAchats().get(i));
37.    }
38.    // on rend le résultat
39.    return remotePanier.getErreurs();
40.  }
41.
42.  // liste des articles
43.  public List getAllArticles() {
44.    // liste fournie par le proxy
45.    return proxy.getAllArticles();
46.  }
47.
48.  // un article particulier
49.  public Article getArticleById(int idArticle) {
50.    // fourni par le proxy
51.    return proxy.getArticleById(idArticle);
52.  }
53.
54.}
```

Pour comprendre le code de cette classe, revenons à l'architecture de l'application :



La classe [ProxyClientServiceMagasin] implémente la couche [2C]. Elle a deux contraintes à respecter :

- elle doit présenter une interface [IArticlesDomain] à la couche [ui]. C'est fait en ligne 15.
- elle doit passer les demandes de la couche [ui] à la couche [2R] qui présente une interface [IServiceMagasin]. Une référence sur l'objet implémentant la couche [2R] sera injectée par Spring dans la classe [ProxyClientServiceMagasin]. Les lignes 18-24 permettent cette injection de dépendance.

Ceci expliqué, le code de la classe [ProxyClientServiceMagasin] consiste essentiellement à passer d'une interface [IArticlesDomain] à l'interface [IServiceMagasin]. Les deux interfaces ont deux méthodes identiques : getAllArticles, getArticleById.

- ligne 15 : la classe implémente l'interface [IArticlesDomain]
- lignes 18-24 : l'objet implémentant la couche [2R] est défini. Il s'appelle [proxy] et implémente l'interface [IServiceMagasin].
- lignes 43-46 : la liste de tous les articles est demandée à l'objet [proxy]
- lignes 49-52 : l'article demandé par le client est demandée à l'objet [proxy]
- lignes 27-40 : on fait l'achat d'un panier. Ce n'est que sur cette action que les interfaces [IArticlesDomain] et [IServiceMagasin] diffèrent.
  - ligne 29 : **panier** est acheté via l'objet [proxy]. Celui-ci rend un panier **remotePanier** de type [**PanierAchete**] qui contient **panier** vidé des articles dont l'achat a pu se faire ainsi que la liste des erreurs sur les achats qui n'ont pu se faire.
  - lignes 31-33 : *panier* est vidé
  - lignes 35-37 : *panier* est rempli avec le contenu de *remotePanier*
  - ligne 39 : la méthode [acheter] de l'interface [IArticlesDomain] doit rendre la liste des erreurs d'achats.

La classe [**ClientMagasin**] est chargée de lancer le client :

```
1. package istia.st.magasin.remoting.httpinvoker;
2.
3. import java.io.BufferedReader;
4. import java.io.InputStreamReader;
5. import org.springframework.beans.factory.xml.XmlBeanFactory;
6. import org.springframework.core.io.ClassPathResource;
7. import istia.st.m2vc.core.IControleur;
8. import javax.swing.UIManager;
9.
10./**
11. * @author serge.tahe@istia.univ-angers.fr
12. *
13. */
14.public class ClientMagasin {
15.
16.  public static void main(String[] args) throws Exception {
17.    // on fixe le look and feel des vues avant leur création
18.    try {
19.      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
20.    }
21.    catch (Exception ex) {
22.      // on s'arrête
23.      abort("Erreur lors de l'initialisation du look and feel", ex, 4);
24.    }
25.    // variables locales
26.    IControleur monControleur = null;
27.    // message de patience
28.    System.out.println("Application en cours d'initialisation. Patientez...");
29.    try {
30.      // on instancie le contrôleur de l'application
31.      monControleur = (IControleur) (new XmlBeanFactory(new ClassPathResource(
32.          "m2vc.xml"))).getBean("controleur");
33.    }
34.    catch (Exception ex) {
35.      // on s'arrête
36.      abort("Erreur lors de l'initialisation du contrôleur", ex, 1);
37.    }
38.    // exécution application
39.    System.out.println("Application lancée...");
40.    try {
41.      monControleur.run();
42.    }
43.    catch (Exception ex) {
44.      // on affiche l'erreur et on s'arrête
45.      abort("Erreur d'exécution", ex, 2);
46.    }
47.    // fin normale
```

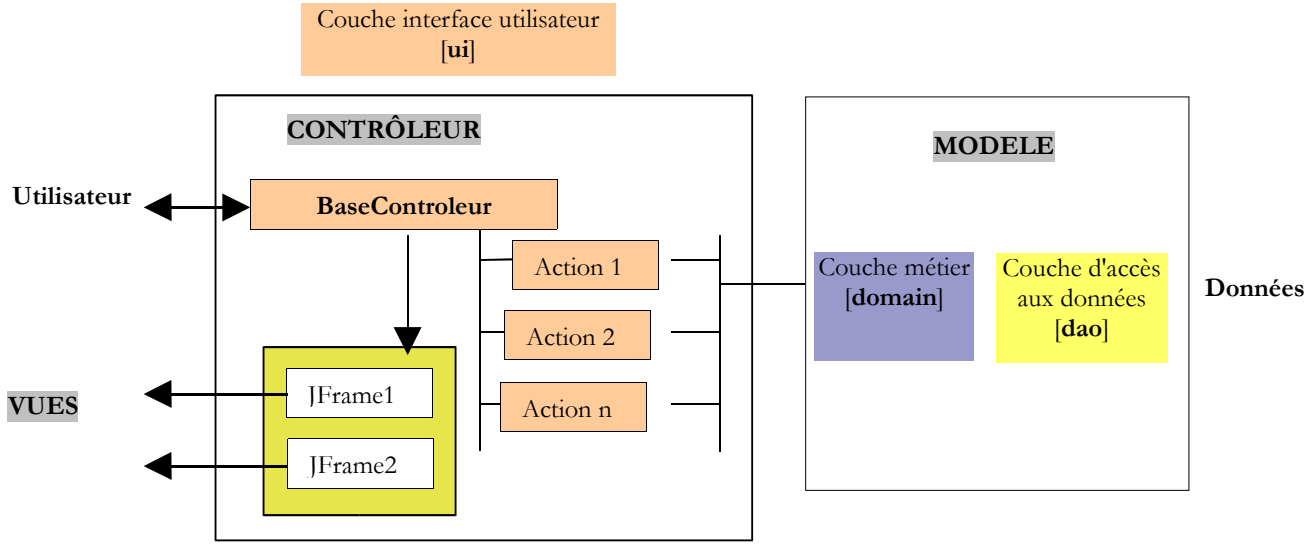
```
48.     System.out.println("Application terminée...");
49.     System.exit(0);
50.   }
51.
52.   // fin anormale
53.   private static void abort(String message, Exception ex, int exitCode) throws
54.       Exception {
55.     // on affiche l'erreur
56.     System.out.println(message);
57.     System.out.println("----------------------------------------");
58.     System.out.println(ex.toString());
59.     System.out.println("----------------------------------------");
60.     // on laisse l'utilisateur voir le message
61.     System.out.println("Tapez [enter] pour terminer l'application...");
62.     BufferedReader IN = new BufferedReader(new InputStreamReader(System.in));
63.     IN.readLine();
64.     // on s'arrête
65.     System.exit(exitCode);
66.   }
67.}
```

Ce code reprend à l'identique celui du lanceur de [swingarticles] décrit dans [article3]. Nous nous apesantirons peu dessus. Rappelons simplement les points importants :

- lignes 17-24 : on fixe l'aspect des fenêtres graphiques du client
- lignes 29-37 : on instancie le contrôleur du client. Ce contrôleur est configuré par le fichier [m2vc.xml].
- lignes 40-46 : on lance le contrôleur qui va alors afficher la fenêtre d'accueil, celle qui affiche la liste des articles.

4.3.3 Configuration du client [swingarticles]

Rappelons l'architecture du client [swingarticles] :



M=modèle	les classes métier, les classes d'accès aux données et la base de données
V=vues	les formulaires Windows
C=contrôleur	le contrôleur [BaseControleur] de traitement des requêtes clientes, les objets [Action]

Rappelons les grands principes de fonctionnement du moteur [M2VC] :

- le contrôleur [BaseControleur] est le coeur de l'application. Toutes les demandes du client transitent par lui. C'est une classe fournie par [M2VC]. On peut dans certains cas être amené à la dériver. Pour les cas simples, ce n'est pas nécessaire.
- [BaseControleur] prend les informations dont il a besoin dans un fichier appelé [**m2vc.xml**]. Il y trouve la liste des objets [Action] destinés à exécuter les demandes du client, la liste des vues de l'application, une liste d'objets [InfosAction] décrivant chaque action. [InfosAction] a les attributs suivants :
  - [vue] : désigne une vue [JFrame] à afficher si l'action ne consiste qu'à changer de vue.
  - [action] : désigne un objet [Action] à exécuter si l'action demandée nécessite l'exécution d'un code
  - [états] : un dictionnaire associant une vue à chacun des résultats possibles de l'objet [Action]. Le contrôleur affichera la vue associée au résultat renvoyé par l'action.

- l'utilisateur a devant lui un formulaire windows. Celui-ci traite certains événements lui-même, ceux qui ne nécessitent pas la couche métier. Les autres sont délégués au contrôleur. On dit alors que la vue demande l'exécution d'une action au contrôleur. Le contrôleur reçoit cette demande sous la forme d'un nom d'action.
- [BaseControleur] récupère alors l'instance [InfosAction] liée au nom de l'action qu'on lui demande d'exécuter. Pour cela, il a un dictionnaire associant le nom d'une action à une instance [InfosAction] rassemblant les informations nécessaires à cette action.
- si l'attribut [vue] de [InfosAction] est non vide, alors la vue associée est affichée. On passe ensuite à l'étape 9.
- si l'attribut [action] de [InfosAction] est non vide, alors l'action est exécutée. Celle-ci fait ce qu'elle a à faire puis rend au contrôleur une chaîne de caractères représentant le résultat auquel elle est parvenue.
- le contrôleur utilise le dictionnaire [états] de [InfosAction] pour trouver la vue V à afficher. Il l'affiche.
- ici une vue a été affichée. Le contrôleur s'est synchronisé avec et attend que la vue déclenche une nouvelle action. Celle-ci va être déclenchée par une action particulière de l'utilisateur sur la vue, qui à cette occasion va repasser la main au contrôleur en lui donnant le nom de l'action à exécuter.
- le contrôleur reprend à l'étape 4

La configuration du client [swingarticles] est faite dans le fichier [m2vc.xml]. Celui-ci a près de 250 lignes et a été décrit abondamment dans [article3]. Nous ne présentons ici que les modifications apportées au fichier par la nouvelle architecture de l'application.

Dans l'application [swingarticles], les couches [ui] et [domain] étaient sur la même machine et pouvaient se parler directement. Ici ce n'est plus le cas. Au lieu de parler à la couche [domain], la couche [ui] parle à la couche [1C]. Cela amène des modifications dans [m2vc.xml].

Dans le fichier [m2vc.xml] de [swingarticles] les objets métier étaient définis de la façon suivante :

```
1.  <!-- objets métier -->
2.  <bean id="articlesDao" class="istia.st.articles.dao.ArticlesDaoSqlMap">
3.    <constructor-arg index="0">
4.      <value>sqlmap-config-odbc.xml</value>
5.        <!-- avec une base Firebird
6.      <value>sqlmap-config-firebird.xml</value>
7.         -->
8.    </constructor-arg>
9.  </bean>
10. <bean id="articlesDomain" class="istia.st.articles.domain.AchatsArticles">
11.   <constructor-arg index="0">
12.     <ref bean="articlesDao"/>
13.   </constructor-arg>
14. </bean>
15. <bean id="panier" class="istia.st.articles.domain.Panier" />
```

- lignes 10 : on définit la classe qui va implémenter l'interface [IArticlesDomain]. Dans notre client, cette interface est désormais implémentée par la classe [ProxyClientServiceMagasin].
- ligne 12 : pour se construire la classe [AchatsArticles] avait besoin d'avoir une référence à la couche [dao], référence définie lignes 2-9. La classe [ProxyClientServiceMagasin] elle se construit différemment. L'objet [articlesDao] défini lignes 2-9 devient inutile.

Dans le fichier [m2vc.xml] de [magasin-client-serveur], les objets métier du client [swingarticles] sont désormais les suivants :

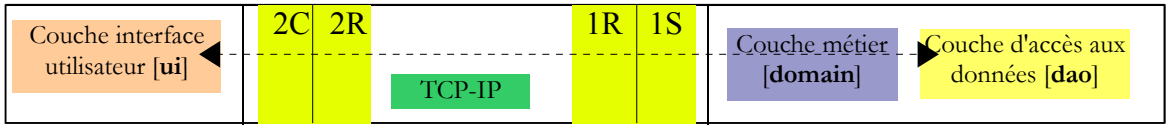
```
1.  <!-- objets métier -->
2.  <bean id="httpinvokerProxy"
3.    class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
4.    <property name="serviceUrl">
5.      <value>http://localhost:8080/magasin-httpinvoker/run</value>
6.    </property>
7.      <property name="serviceInterface">
8.      <value>istia.st.magasin.remoting.httpinvoker.IServiceMagasin</value>
9.    </property>
10. </bean>
11. <bean id="articlesDomain" class="istia.st.magasin.remoting.httpinvoker.ProxyClientServiceMagasin">
12.   <property name="proxy">
13.     <ref bean="httpinvokerProxy"/>
14.   </property>
15. </bean>
16. <bean id="panier" class="istia.st.articles.domain.Panier" />
```

- lignes 11-12 : on définit la classe implémentant l'interface [IArticlesDomain] à laquelle s'adresse notre client [swingarticles]. Il s'agit de la classe [ProxyClientServiceMagasin]. Si on se reporte à la définition de cette classe, on verra qu'elle a un attribut [proxy] défini comme suit :

```
// attributs
private IServiceMagasin proxy;
public IServiceMagasin getProxy() {
  return proxy;
}
```

```
public void setProxy(IServiceMagasin proxy) {
  this.proxy = proxy;
}
```

L'attribut [proxy] est la référence sur l'objet implémentant la couche [2R] du client :



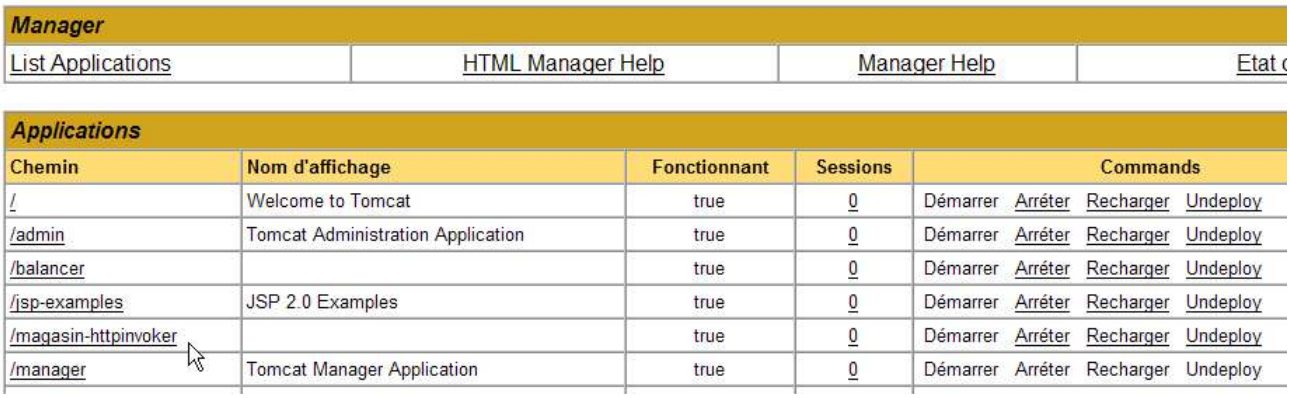
- la couche [2R] du client est implémentée par une classe prédéfinie de Spring et définie lignes 2-10 du fichier [m2vc.xml]. C'est la classe [HttpInvokerProxyFactoryBean]. Celle-ci a besoin, pour se construire de deux informations :
  - lignes 4-6 : l'URL du service distant. Nous avons indiqué à plusieurs reprises dans l'étude du serveur que le service était à l'url [http://localhost:8080/magasin-httpinvoker/run].
  - lignes 7-9 : nous avons dit que la couche [2R] devait implémenter l'interface [IServiceMagasin]. Cela, la classe prédéfinie [HttpInvokerProxyFactoryBean] ne le sait évidemment pas. On le lui dit lignes 7-9.

C'est tout. Notre client a été décrit totalement.

4.4 Les tests de l'application [magasin-client-serveur]

C'est une application client-serveur. Il faut donc lancer d'abord le serveur puis le client.

- le serveur est une application web déjà configurée au sein de Tomcat 5. Pour la lancer, on lance Tomcat 5 en mode autonome (pas à partir d'Eclipse). On vérifie ensuite qu'elle a été correctement déployée :



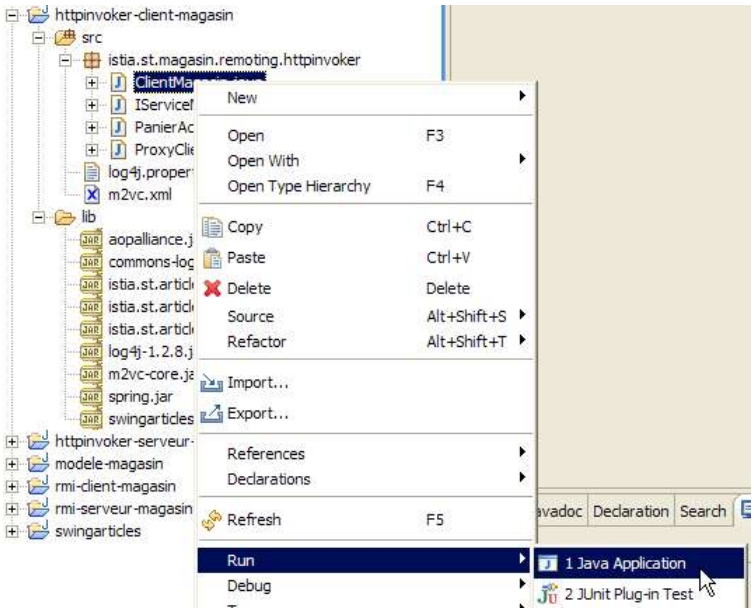
Manager

List Applications	HTML Manager Help	Manager Help	Etat c

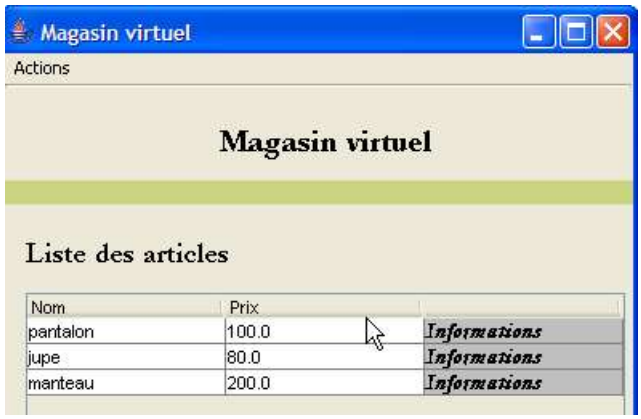
Applications

Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands
/	Welcome to Tomcat	true	0	Démarrer Arréter Recharger Undeploy
/admin	Tomcat Administration Application	true	0	Démarrer Arréter Recharger Undeploy
/balancer		true	0	Démarrer Arréter Recharger Undeploy
/jsp-examples	JSP 2.0 Examples	true	0	Démarrer Arréter Recharger Undeploy
/magasin-httpinvoker		true	0	Démarrer Arréter Recharger Undeploy
/manager	Tomcat Manager Application	true	0	Démarrer Arréter Recharger Undeploy

- on lance ensuite le client [ClientMagasin.java] directement sous Eclipse :



Si tout va bien, on obtient la première vue de l'application, celle de la liste des articles :



Spring a inscrit dans la console d'Eclipse un certain nombre de logs utiles en cas de problèmes :

```
1. Application en cours d'initialisation. Patientez...
2.  INFO [main] - Loading XML bean definitions from class path resource [m2vc.xml]
3.  INFO [main] - JDK 1.4+ collections available
4.  INFO [main] - Creating shared instance of singleton bean 'controleur'
5.  INFO [main] - Creating shared instance of singleton bean 'synchro'
6.  INFO [main] - Creating shared instance of singleton bean 'session'
7.  INFO [main] - Creating shared instance of singleton bean 'articlesDomain'
8.  INFO [main] - Creating shared instance of singleton bean 'httpinvokerProxy'
9.  INFO [main] - CGLIB2 not available: proxyTargetClass feature disabled
10. INFO [main] - Creating shared instance of singleton bean 'panier'
11. INFO [main] - Creating shared instance of singleton bean 'infosActionListe'
12. INFO [main] - Creating shared instance of singleton bean 'actionListe'
13. INFO [main] - Creating shared instance of singleton bean 'vueListe'
14. INFO [main] - Creating shared instance of singleton bean 'vueErreurs'
15. INFO [main] - Creating shared instance of singleton bean 'infosActionInfos'
16. INFO [main] - Creating shared instance of singleton bean 'actionInfos'
17. INFO [main] - Creating shared instance of singleton bean 'vueInfos'
18. INFO [main] - Creating shared instance of singleton bean 'infosActionAchat'
19. INFO [main] - Creating shared instance of singleton bean 'actionAchat'
20. INFO [main] - Creating shared instance of singleton bean 'vuePanier'
21. INFO [main] - Creating shared instance of singleton bean 'infosActionVoirPanier'
22. INFO [main] - Creating shared instance of singleton bean 'actionVoirPanier'
23. INFO [main] - Creating shared instance of singleton bean 'vuePanierVide'
24. INFO [main] - Creating shared instance of singleton bean 'infosActionRetirerAchat'
25. INFO [main] - Creating shared instance of singleton bean 'actionRetirerAchat'
26. INFO [main] - Creating shared instance of singleton bean 'infosActionValiderPanier'
27. INFO [main] - Creating shared instance of singleton bean 'actionValiderPanier'
28.Application lancée...
```

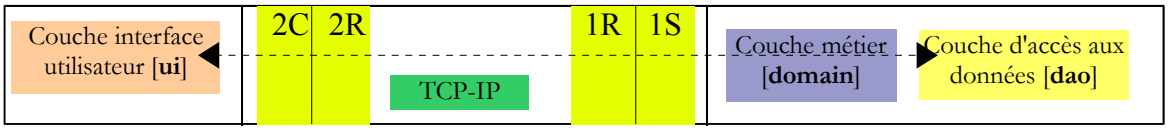
On voit clairement l'instanciation des différents objets définis dans le fichier [m2vc.xml].

Le lecteur est invité, à partir de cette première vue, à reproduire le fonctionnement décrit paragraphe 2.5, page 15. Rappelons que pour que l'application fonctionne, une source ODBC appelée [odbc-access-dvp-articles] doit être associée à une base de données ayant une table ARTICLES au format indiqué paragraphe 2.3.1, page 5. Une base ACCESS exemple est fournie dans le dossier [database] du projet Eclipse du serveur.

5 [magasin-client-serveur] avec Spring Caucho Hessian

5.1 L'architecture

Reprenons l'architecture de l'application [magasin-client-serveur] :



Les couches [1R] et [2R] étaient implémentées par des classes prédéfinies de Spring du paquetage [org.springframework.remoting.httpinvoker]. Ces couches d'interfaçage avec le réseau utilisent le protocole HTTP pour faire dialoguer deux applications Java de façon transparente pour celles-ci. Notre nouvelle implémentation de l'application [magasin-

client-serveur] va utiliser cette fois des classes prédéfinies de Spring du paquetage [org.springframework.remoting.caucho]. Les classes de ce paquetage utilisent également le protocole HTTP pour faire communiquer le client et le serveur. Sur HTTP, le paquetage [org.springframework.remoting.caucho] propose deux protocoles de transmission des objets sur le réseau :

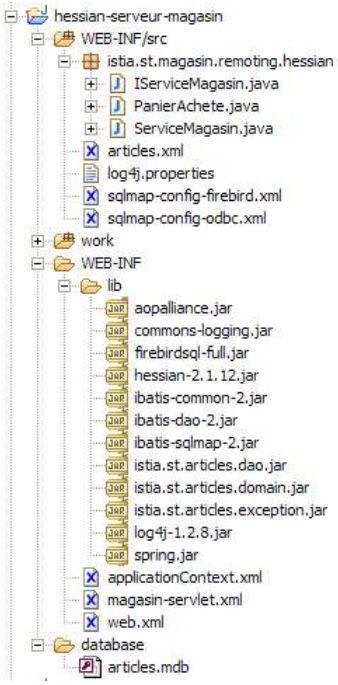
- **Hessian** : les objets sont mis sur le réseau sous forme binaire
- **Burlap** : les objets sont mis sur le réseau sous forme XML

Le protocole Hessian est disponible dans d'autres environnements que Java et c'est ce qui fait son intérêt. On consultera l'url [http://www.caucho.com/hessian] pour obtenir la liste des implémentations. Actuellement (juillet 2005) : Java, Python, C++, C#, PHP, Ruby. Le protocole Burlap semble n'avoir pour l'instant que la seule implémentation Java. Le protocole binaire Hessian est nativement plus performant que le protocole XML de Burlap, protocole plus verbeux qu'un simple échange binaire.

Pour en revenir à notre application [magasin-client-serveur], utiliser le protocole [Hessian] revient à changer les classes d'implémentation des couches [1R] et [2R] pour qu'elles parlent Hessian. C'est tout. Comme ces classes sont fournies par Spring, aucun effort ne nous est demandé. Nous décrivons la nouvelle implémentation en ne nous attardant que sur ce qui change par rapport à la solution précédente.

5.2 Le serveur

Le projet Eclipse du serveur est le suivant :



Tout est identique au projet précédent exceptés les points suivants :

- le paquetage des sources Java s'appelle désormais [istia.st.magasin.remoting.hessian]
- la classe [PanierAchete] n'implémente pas l'interface [Serializable] :

```
1. package istia.st.magasin.remoting.hessian;
2.
3. import istia.st.articles.domain.Panier;
4.
5. import java.util.ArrayList;
6.
7. /**
8.  * @author serge.tahe@istia.univ-angers.fr
9.  *
10. */
11.public class PanierAchete {
12.  // le panier acheté
13.  private Panier panier;
14.
15....
16.  // les erreurs d'achat du panier
17.  private ArrayList erreurs;
18.
19....
20.  // constructeurs
```

```
21.  public PanierAchete(){
22.  }
23.
24.  public PanierAchete(Panier panier, ArrayList erreurs){
25....
26.  }
27.}
```

- dans [WEB-INF/lib] on trouve l'archive [hessian-2.1.12.jar] qui contient les classes gérant le protocole [Caucho Hessian].
- le fichier [applicationContext.xml] devient le suivant :

```
1. <?xml version="1.0" encoding="ISO-8859-1"?>
2. <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3. <beans>
4.  <!-- la classe d'accès aux données -->
5.  <bean id="articlesDao" class="istia.st.articles.dao.ArticlesDaoSqlMap">
6.    <constructor-arg index="0">
7.      <value>sqlmap-config-odbc.xml</value>
8.        <!-- avec une base Firebird
9.      <value>sqlmap-config-firebird.xml</value>
10.        -->
11.   </constructor-arg>
12. </bean>
13. <!-- la classe métier -->
14. <bean id="articlesDomain" class="istia.st.articles.domain.AchatsArticles">
15.   <constructor-arg index="0">
16.     <ref bean="articlesDao"/>
17.   </constructor-arg>
18. </bean>
19. <!-- la classe proxy -->
20. <bean id="serviceMagasin" class="istia.st.magasin.remoting.hessian.ServiceMagasin">
21.   <property name="articlesDomain">
22.     <ref bean="articlesDomain"/>
23.   </property>
24. </bean>
25.</beans>
```

- la ligne 20 prend en compte le changement du nom du paquetage
- le fichier [magasin-servlet.xml] devient le suivant :

```
1. <?xml version="1.0" encoding="ISO-8859-1"?>
2. <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
3. <beans>
4.  <bean id="defaultHandlerMapping"
5.    class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
6.  <bean name="/run"
7.    class="org.springframework.remoting.caucho.HessianServiceExporter">
8.    <property name="service">
9.      <ref bean="serviceMagasin"/>
10.   </property>
11.   <property name="serviceInterface">
12.     <value>istia.st.magasin.remoting.hessian.IServiceMagasin</value>
13.   </property>
14. </bean>
15.</beans>
```

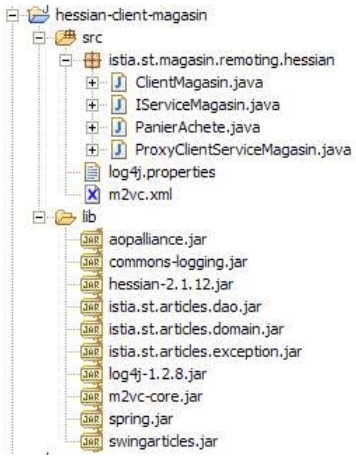
- ligne 7 : on définit la classe d'implémentation de la couche [1R] en contact avec le réseau. Ici elle est implémentée par une classe sachant parler Hessian.
- ligne 12 : le nom du paquetage change

L'application web s'appelle [/magasin-hessian] :

Ce sont les seules différences.

5.3 Le client

Le projet Eclipse du client est le suivant :



Tout est identique au projet précédent, exceptés les points suivants :

- le paquetage des sources Java s'appelle désormais [istia.st.magasin.remoting.hessian]
- la classe [PanierAchete] n'implémente pas l'interface [Serializable] comme pour le serveur
- dans [lib] on trouve l'archive [hessian-2.1.12.jar] qui contient les classes gérant le protocole [Caucho Hessian].
- le fichier [m2vc.xml] devient le suivant :

```
1. ...
2.  <!-- objets métier -->
3.  <bean id="articlesProxy"
4.    class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
5.    <property name="serviceUrl">
6.      <value>http://localhost:8080/magasin-hessian/run</value>
7.    </property>
8.      <property name="serviceInterface">
9.      <value>istia.st.magasin.remoting.hessian.IServiceMagasin</value>
10.   </property>
11. </bean>
12. <bean id="articlesDomain" class="istia.st.magasin.remoting.hessian.ProxyClientServiceMagasin">
13.   <property name="proxy">
14.     <ref bean="articlesProxy"/>
15.   </property>
16. </bean>
17. <bean id="panier" class="istia.st.articles.domain.Panier" />
18....
```

- ligne 4 : on définit la classe d'implémentation de la couche [2R] en contact avec le réseau. Elle est fournie par Spring et sait parler Hessian.
- ligne 6 : la nouvelle application web s'appelle [/magasin-hessian].
- lignes 9 et 12 : le nom du paquetage change

C'est tout.

5.4 Les tests

Le lecteur est invité à reproduire les tests décrits paragraphe 4.4, page 43. Le lien [Tomcat Manager] doit montrer que l'application [/magasin-hessian] est active :

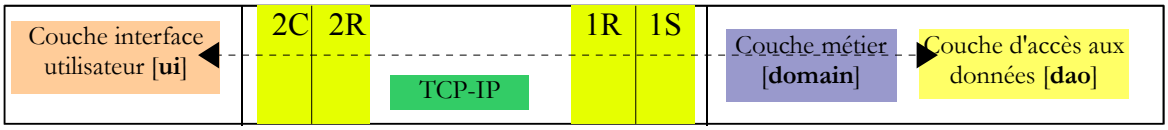
Manager			
List Applications	HTML Manager Help	Manager Help	Etat c

Applications				
Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands
/	Welcome to Tomcat	true	0	Démarrer Arréter Recharger Undeploy
/admin	Tomcat Administration Application	true	0	Démarrer Arréter Recharger Undeploy
/balancer		true	0	Démarrer Arréter Recharger Undeploy
/jsp-examples	JSP 2.0 Examples	true	0	Démarrer Arréter Recharger Undeploy
/magasin-hessian		true	0	Démarrer Arréter Recharger Undeploy
/manager	Tomcat Manager Application	true	0	Démarrer Arréter Recharger Undeploy

6 [magasin-client-serveur] avec Spring Caucho Burlap

6.1 L'architecture

Reprenons l'architecture de l'application [magasin-client-serveur] :



Nous changeons maintenant les classes d'implémentation des couches [1R] et [2R] pour qu'elles parlent **Burlap**. C'est tout. Comme ces classes sont fournies par Spring, aucun effort ne nous est demandé. Nous décrivons la nouvelle implémentation en ne nous attardant que sur ce qui change par rapport à la solution **Hessian** précédente.

6.2 Le serveur

Le projet Eclipse du serveur est le suivant :

Tout est identique au projet précédent exceptés les points suivants :

- le paquetage des sources Java s'appelle désormais [istia.st.magasin.remoting.burlap]
- la classe [PanierAchete] n'implémente pas l'interface [Serializable] comme pour le serveur Hessian

- dans [WEB-INF/lib] on trouve l'archive [burlap-2.1.12.jar] qui contient les classes gérant le protocole [Caucho Burlap]. L'archive [hessian] est supprimée.
- le fichier [applicationContext.xml] devient le suivant :

```
1. <?xml version="1.0" encoding="ISO-8859-1"?>
2. <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3. <beans>
4.  <!-- la classe d'accès aux données -->
5.  <bean id="articlesDao" class="istia.st.articles.dao.ArticlesDaoSqlMap">
6.    <constructor-arg index="0">
7.      <value>sqlmap-config-odbc.xml</value>
8.        <!-- avec une base Firebird
9.      <value>sqlmap-config-firebird.xml</value>
10.        -->
11.   </constructor-arg>
12. </bean>
13. <!-- la classe métier -->
14. <bean id="articlesDomain" class="istia.st.articles.domain.AchatsArticles">
15.   <constructor-arg index="0">
16.     <ref bean="articlesDao"/>
17.   </constructor-arg>
18. </bean>
19. <!-- la classe proxy -->
20. <bean id="serviceMagasin" class="istia.st.magasin.remoting.burlap.ServiceMagasin">
21.   <property name="articlesDomain">
22.     <ref bean="articlesDomain"/>
23.   </property>
24. </bean>
25.</beans>
```

- la ligne 20 prend en compte le changement du nom du paquetage
- le fichier [magasin-servlet.xml] devient le suivant :

```
1. <?xml version="1.0" encoding="ISO-8859-1"?>
2. <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
3. <beans>
4.  <bean id="defaultHandlerMapping"
5.    class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
6.  <bean name="/run"
7.    class="org.springframework.remoting.caucho.BurlapServiceExporter">
8.    <property name="service">
9.      <ref bean="serviceMagasin"/>
10.   </property>
11.   <property name="serviceInterface">
12.     <value>istia.st.magasin.remoting.burlap.IServiceMagasin</value>
13.   </property>
14. </bean>
15.</beans>
```

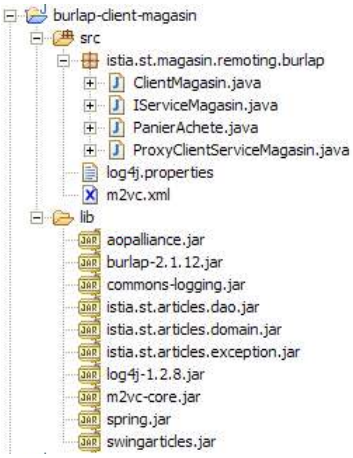
- ligne 7 : on définit la classe d'implémentation de la couche [1R] en contact avec le réseau. Ici elle est implémentée par une classe sachant parler Burlap.
- ligne 12 : le nom du paquetage change

L'application web s'appelle [/magasin-burlap] :

Ce sont les seules différences.

6.3 Le client

Le projet Eclipse du client est le suivant :



Tout est identique au client du projet précédent, exceptés les points suivants :

- le paquetage des sources Java s'appelle désormais [istia.st.magasin.remoting.burlap]
- la classe [PanierAchete] n'implémente pas l'interface [Serializable] comme pour le serveur
- dans [lib] on trouve l'archive [burlap-2.1.12.jar] qui contient les classes gérant le protocole [Caucho Burlap]. L'archive [hessian] est supprimée.
- le fichier [m2vc.xml] devient le suivant :

```
1. ...
2.  <!-- objets métier -->
3.  <bean id="articlesProxy"
4.    class="org.springframework.remoting.caucho.BurlapProxyFactoryBean">
5.    <property name="serviceUrl">
6.      <value>http://localhost:8080/magasin-burlap/run</value>
7.    </property>
8.      <property name="serviceInterface">
9.      <value>istia.st.magasin.remoting.burlap.IServiceMagasin</value>
10.   </property>
11. </bean>
12. <bean id="articlesDomain" class="istia.st.magasin.remoting.burlap.ProxyClientServiceMagasin">
13.   <property name="proxy">
14.     <ref bean="articlesProxy"/>
15.   </property>
16. </bean>
17. <bean id="panier" class="istia.st.articles.domain.Panier" />
18....
```

- ligne 4 : on définit la classe d'implémentation de la couche [2R] en contact avec le réseau. Elle est fournie par Spring et sait parler Burlap.
- ligne 6 : la nouvelle application web s'appelle [/magasin-burlap].
- lignes 9 et 12 : le nom du paquetage change

C'est tout.

6.4 Les tests

Le lecteur est invité à reproduire les tests décrits paragraphe 4.4, page 43. Le lien [Tomcat Manager] doit montrer que l'application [/magasin-burlap] est active :

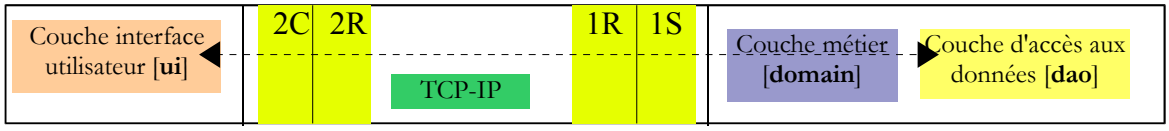
Manager			
List Applications	HTML Manager Help	Manager Help	Et

Applications				
Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands
/	Welcome to Tomcat	true	0	Démarrer Arréter Recharger Undeploy
/admin	Tomcat Administration Application	true	0	Démarrer Arréter Recharger Undeploy
/balancer		true	0	Démarrer Arréter Recharger Undeploy
/jsp-examples	JSP 2.0 Examples	true	0	Démarrer Arréter Recharger Undeploy
/magasin-burlap		true	0	Démarrer Arréter Recharger Undeploy
/magasin-hessian		true	0	Démarrer Arréter Recharger Undeploy

7 [magasin-client-serveur] avec Spring RMI

7.1 L'architecture

Reprenons l'architecture de l'application [magasin-client-serveur] :



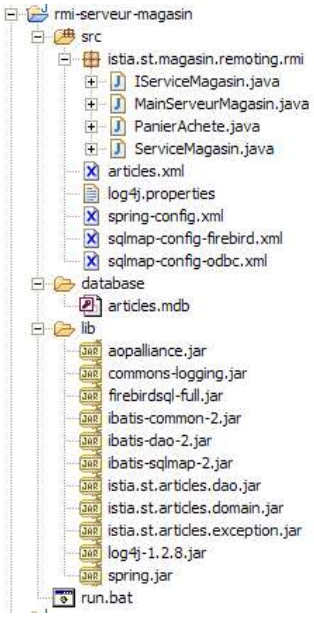
Nous changeons maintenant les classes d'implémentation des couches [1R] et [2R] pour qu'elles utilisent le protocole de communication **RMI**. **RMI** (Remote Method Invocation) est un protocole de dialogue réseau propre à Java qui se place au-dessus de TCP-IP. C'est un protocole performant mais qui a deux inconvénients :

- il traverse mal les pare-feu des entreprises et administrations. En effet ceux-ci laissent passer peu de choses en-dehors du protocole HTTP. Ainsi si on installe sur Internet une application web travaillant avec le protocole RMI, beaucoup de clients situés derrière un pare-feu d'entreprise ne pourront pas l'atteindre. Le protocole RMI est donc plutôt réservé à des applications intranet où le pare-feu n'interfère plus.
- il est propriétaire à Java. Les deux interlocuteurs doivent parler Java. Il existe une solution à ce problème. Le protocole RMI-IIOP (Internet Inter-Orb Protocol) permet de transformer un serveur RMI en serveur CORBA, un protocole supporté cette fois par d'autres langages que Java. Un serveur RMI-IIOP pourra donc avoir d'autres clients que des clients java.

Les classes nécessaires aux couches [1R] et [2R] d'interfaçage avec le réseau vont être fournies là encore par Spring. Rien ne change pour les couches [1S] et [2C]. Comme pour les projets précédents, les principales modifications seront à faire dans les fichiers de configuration.

7.2 Le serveur

Le projet Eclipse du serveur est le suivant :



Tout est identique au projet précédent exceptés les points suivants :

- ce n'est plus une application web. Sous Eclipse, le projet est un projet Java et non un projet [Tomcat]. Dans les applications précédentes, il fallait lancer Tomcat pour lancer le service. Ici le service RMI est lancé par une classe Java classique : **MainServeurMagasin**.
- les fichiers de configuration de l'application web qui étaient sous WEB-INF ont disparu pour être remplacés par le fichier [spring-config.xml].
- le paquetage des sources Java s'appelle désormais [istia.st.magasin.remoting.rmi]
- dans [WEB-INF/lib] l'archive [burlap] est supprimée.

La classe [MainServeurMagasin] chargée de lancer le service RMI est la suivante :

```
1. package istia.st.magasin.remoting.rmi;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. /**
7.  * @author serge.tahe@istia.univ-angers.fr
8.  *
9.  */
10.public class MainServeurMagasin {
11.
12.  public static void main(String[] args){
13.    // on instancie le serveur RMI
14.    ApplicationContext ac=new ClassPathXmlApplicationContext("spring-config.xml");
15.    System.out.println("Magasin RMI lancé...");
16.  }
17.}
```

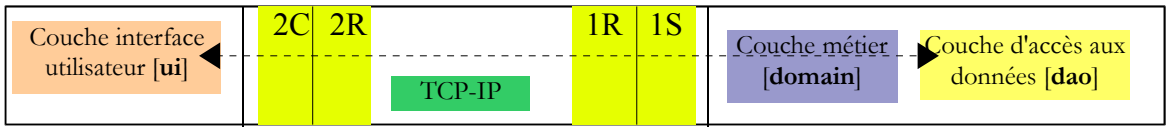
- ligne 14 : on instancie tous les beans définis dans le fichier de configuration Spring nommé [spring-config.xml]. Parmi eux, il y a le serveur RMI. Une fois celui-ci instancié, il se met à l'écoute des demandes des clients dans un thread qui lui est propre.
- ligne 15 : on affiche un message indiquant que le serveur RMI a été lancé. Après cette instruction le thread de [main] est terminé. Celui du serveur RMI est lui toujours actif. On pourra constater, au moment des tests, que la console Java de [main] ne rend pas la main indiquant par là qu'il y a encore des threads actifs.

Il nous reste à décrire le fichier [spring-config.xml] :

```
1. <?xml version="1.0" encoding="ISO-8859-1"?>
2. <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3. <beans>
4.  <!-- la classe d'accès aux données -->
5.  <bean id="articlesDao" class="istia.st.articles.dao.ArticlesDaoSqlMap">
6.    <constructor-arg index="0">
7.      <value>sqlmap-config-odbc.xml</value>
8.        <!-- avec une base Firebird
9.      <value>sqlmap-config-firebird.xml</value>
10.        -->
11.   </constructor-arg>
12. </bean>
13. <!-- la classe métier -->
14. <bean id="articlesDomain" class="istia.st.articles.domain.AchatsArticles">
15.   <constructor-arg index="0">
```

```
16.     <ref bean="articlesDao"/>
17.   </constructor-arg>
18. </bean>
19. <!-- la classe d'implémentation de l'interface du service -->
20. <bean id="serviceMagasin" class="istia.st.magasin.remoting.rmi.ServiceMagasin">
21.   <property name="articlesDomain">
22.     <ref bean="articlesDomain"/>
23.   </property>
24. </bean>
25. <!-- le service RMI qui expose l'interface du service -->
26. <bean id="serviceExporter"
27.   class="org.springframework.remoting.rmi.RmiServiceExporter">
28.   <property name="serviceName">
29.     <value>ServiceArticles</value>
30.   </property>
31.   <property name="service">
32.     <ref local="serviceMagasin"/>
33.   </property>
34.   <property name="serviceInterface">
35.     <value>istia.st.magasin.remoting.rmi.IServiceMagasin</value>
36.   </property>
37.   <property name="registryPort">
38.     <value>9000</value>
39.   </property>
40.   <property name="servicePort">
41.     <value>9010</value>
42.   </property>
43. </bean>
44.</beans>
```

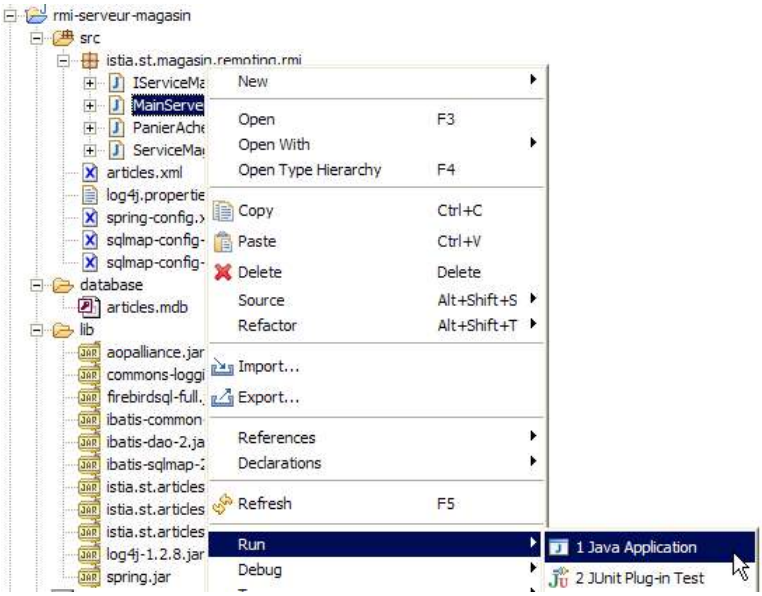
Revenons sur l'architecture de l'application :



- le fichier [spring-config.xml] définit les quatre beans implémentant les couches [dao], [domain], [1S] et [1R] du serveur
- lignes 5-12 : on définit la classe implémentant la couche [dao]
- lignes 14-18 : on définit la classe implémentant la couche [domain]
- lignes 20-24 : on définit la classe implémentant la couche [1S]
- lignes 26-43 : on définit la classe implémentant la couche [1R]. Celle-ci a besoin de plusieurs informations pour s'instancier :
  - ligne 27 : la classe à instancier. C'est une classe fournie par Spring
  - lignes 34-36 : le service RMI doit savoir quelle interface il expose à ses clients. C'est l'interface [IServiceMagasin].
  - lignes 31-33 : le service RMI a besoin d'avoir la référence d'une classe implémentant cette interface. Ici c'est le bean [serviceMagasin] défini aux lignes 20-24.
  - lignes 28-30 : un service RMI est identifié par un nom. Ici ce sera [ServiceArticles].
  - lignes 37-39 : le port sur lequel les clients devront se connecter pour demander le service. On l'appelle le port d'enregistrement. Ici c'est le port 9000. Il faut éviter bien sûr de prendre un port déjà utilisé par un autre service TCP-IP.
  - lignes 40-42 : le port de communication du service. Ici ce sera 9010. Les clients ne connaissent pas ce n° et n'ont pas à le connaître. Ils s'adressent uniquement au port d'enregistrement. Aussi peut-on ne pas forcer la valeur du port de communication. Pour cela on laisse sa valeur à 0. Le véritable n° du port de communication du service sera alors un n° aléatoire.

Il n'y a pas d'autres modifications.

Exécutons la classe [MainServeurMagasin] sous Eclipse pour lancer le service RMI :



Une fois l'application lancée, on peut voir dans la console d'Eclipse les logs de Spring et de [MainServeurMagasin] :

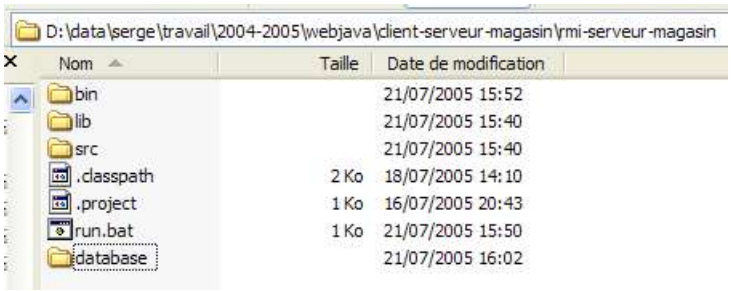
```
1.  INFO [main] - Loading XML bean definitions from class path resource [spring-config.xml]
2.  INFO [main] - Bean factory for application context
   [org.springframework.context.support.ClassPathXmlApplicationContext;hashCode=14746332]:
   org.springframework.beans.factory.support.DefaultListableBeanFactory defining beans
   [articlesDao,articlesDomain,serviceMagasin,serviceExporter]; root of BeanFactory hierarchy
3.  INFO [main] - 4 beans defined in application context
   [org.springframework.context.support.ClassPathXmlApplicationContext;hashCode=14746332]
4.  INFO [main] - JDK 1.4+ collections available
5.  INFO [main] - Unable to locate MessageSource with name 'messageSource': using default
   [org.springframework.context.support.DelegatingMessageSource@152513a]
6.  INFO [main] - Unable to locate ApplicationEventMulticaster with name 'applicationEventMulticaster':
   using default [org.springframework.context.event.SimpleApplicationEventMulticaster@f99ff5]
7.  INFO [main] - Pre-instantiating singletons in factory
   [org.springframework.beans.factory.support.DefaultListableBeanFactory defining beans
   [articlesDao,articlesDomain,serviceMagasin,serviceExporter]; root of BeanFactory hierarchy]
8.  INFO [main] - Creating shared instance of singleton bean 'articlesDao'
9.  INFO [main] - Creating shared instance of singleton bean 'articlesDomain'
10. INFO [main] - Creating shared instance of singleton bean 'serviceMagasin'
11. INFO [main] - Creating shared instance of singleton bean 'serviceExporter'
12. INFO [main] - Looking for RMI registry at port '9000'
13. WARN [main] - Could not detect RMI registry - creating new one
14. INFO [main] - RMI object 'ServiceArticles' is an RMI invoker
15. INFO [main] - CGLIB2 not available: proxyTargetClass feature disabled
16. INFO [main] - Binding RMI service 'ServiceArticles' to registry at port '9000'
17.Magasin RMI lancé...
```

- les lignes 1-11 doivent être maintenant devenues un peu connues du lecteur. On suit l'instanciation des quatre beans définis dans [spring-config.xml]
- ligne 12 : Spring vérifie si un service d'enregistrement de services RMI existe déjà sur le port 9000
- ligne 13 : il n'en a pas trouvé. Il le crée.
- ligne 16 : le service RMI "ServiceArticles" est enregistré auprès du service d'enregistrement du port 9000.

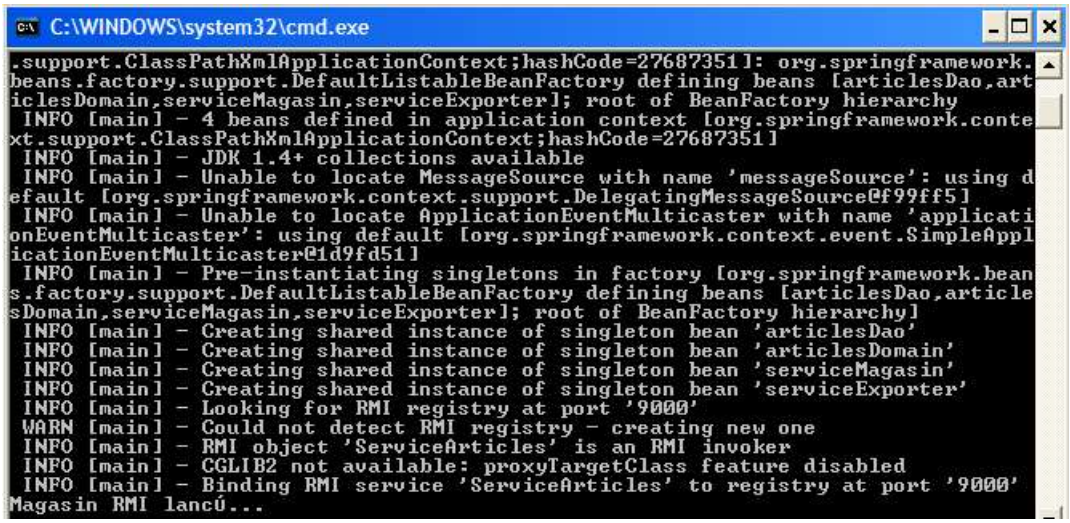
Dans la console d'Eclipse, on peut constater que le programme [MainServeurMagasin] est toujours actif :

L'icône [Terminate] ci-dessus est active indiquant par là que le programme est en cours d'exécution. En fait le serveur RMI attend des clients. Nous cliquons sur [Terminate] pour arrêter le service RMI.

Afin de pouvoir lancer le serveur RMI de l'extérieur d'Eclipse, on a créé un fichier batch appelé [run.bat] placé à la racine du dossier du serveur :



Un double-clic sur [run.bat] lance le service RMI :



Le contenu de [run.bat] est le suivant :

```
1. %java_home%\bin\java.exe -classpath
   bin;lib/spring.jar;lib/istia.st.articles.dao.jar;lib/istia.st.articles.domain.jar;lib/istia.st.articles
   .exception.jar;lib/commons-logging.jar;lib/ibatis-common-2.jar;lib/ibatis-dao-2.jar;lib/ibatis-sqlmap-
   2.jar;lib/log4j-1.2.8.jar;lib/aopalliance.jar istia.st.magasin.remoting.rmi.MainServeurMagasin
```

Avant de l'exécuter, il faut faire pointer la variable [java_home] sur la racine d'un JDK. On pourra procéder comme suit par exemple :

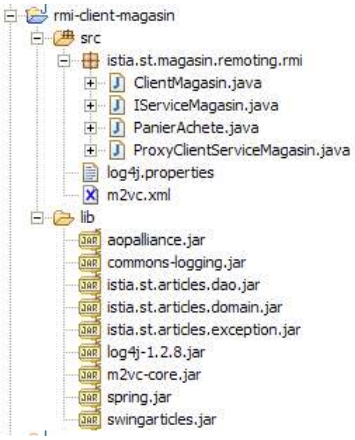
```
1. dos>set JAVA_HOME=C:\jewelbox\j2sdk1.4.2
2.
3. dos>echo %JAVA_HOME%
4. C:\jewelbox\j2sdk1.4.2
```

- ligne 1 : on fixe la valeur de JAVA_HOME
- lignes 3-4 : on la vérifie

On peut alors lancer [run.bat].

7.3 Le client

Le projet Eclipse du client est le suivant :

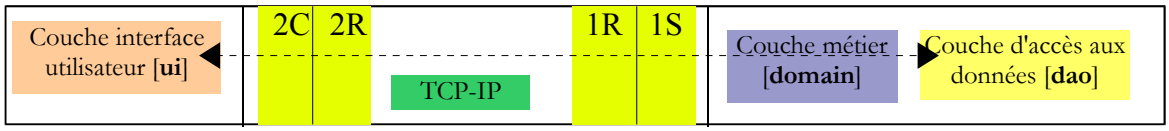


Tout est identique au client du projet précédent, exceptés les points suivants :

- le paquetage des sources Java s'appelle désormais [istia.st.magasin.remoting.rmi]
- dans [lib] l'archive [burlap-2.1.12.jar] a été supprimée.
- le fichier [m2vc.xml] devient le suivant :

```
1. ...
2.  <!-- objets métier -->
3.  <bean id="articlesProxy" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
4.    <property name="serviceUrl">
5.      <value>rmi://localhost:9000/ServiceArticles</value>
6.    </property>
7.    <property name="serviceInterface">
8.      <value>istia.st.magasin.remoting.rmi.IServiceMagasin</value>
9.    </property>
10. </bean>
11. <bean id="articlesDomain" class="istia.st.magasin.remoting.rmi.ProxyClientServiceMagasin">
12.   <property name="proxy">
13.     <ref bean="articlesProxy"/>
14.   </property>
15. </bean>
16. <bean id="panier" class="istia.st.articles.domain.Panier" />
17....
```

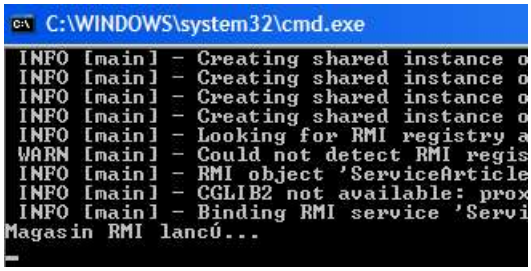
Revenons sur l'architecture de l'application :



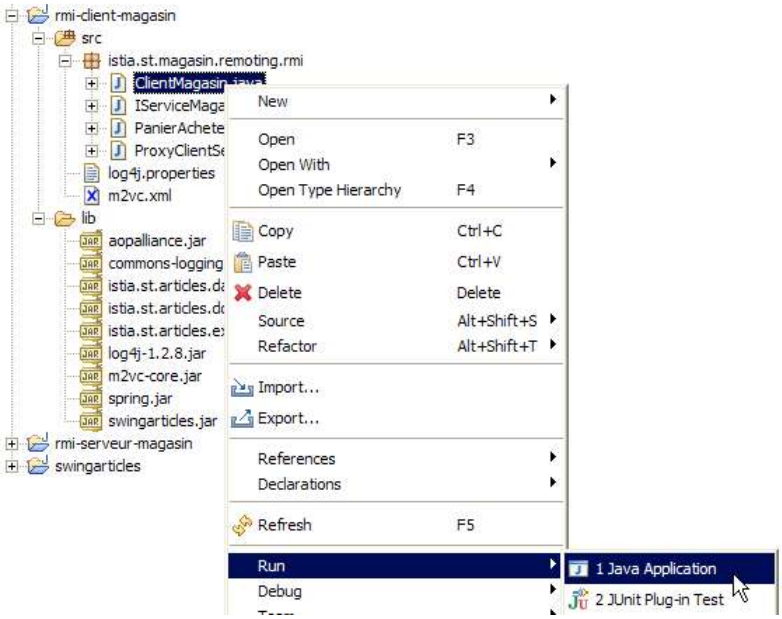
- le fichier [m2vc.xml] définit les beans implémentant les couches [2C] et [2R] du client
- lignes 3-10 : on définit la classe implémentant la couche [2R]
  - ligne 3 : on définit sa classe. C'est une classe fournie par Spring. Pour s'instancier, celle-ci a besoin de deux informations :
  - lignes 4-6 : l'url du service distant avec qui elle doit dialoguer. Cette url a la forme [rmi://machine:portEnregistrement/nomService] avec :
    - rmi : le nom du protocole de communication entre les partenaires
    - machine : machine où se trouve le service RMI
    - portEnregistrement : port à contacter pour demander un service RMI sur machine
    - nomService : le nom du service particulier que l'on veut joindre, une machine pouvant avoir plusieurs services RMI actifs en même temps.
  - lignes 7-9 : le nom de l'interface que la couche [2R] doit présenter à la couche [2C], ici l'interface [IServiceMagasin].
- lignes 14-18 : on définit la classe implémentant la couche [2C], ici la classe [ProxyClientServiceMagasin].

7.4 Les tests

Nous sommes prêts pour les tests. Tout d'abord nous lançons le serveur RMI en double-cliquant sur le fichier [run.bat] :



Puis nous lançons le client sous Eclipse :



Nous obtenons alors la première vue affichée par le client [swingarticles] :

Nous invitons le lecteur à faire les tests déjà décrits page 15.

8 Conclusion

Cet article fait suite à une série d'articles sur l'architecture des applications web multi-couches. Au fil de ces articles, nous avons découvert comment :

- découper une application en couches, que celle-ci soit dans le domaine du web, du client-serveur ou simplement une application swing "standalone" classique,
- rendre ces couches interchangeables grâce à l'utilisation d'interfaces,
- configurer l'ensemble des couches de l'application grâce à **Spring**,
- créer un client riche pour un service web,

- créer des services web avec différents protocoles client-serveur,
- donner à chaque fois à l'application une architecture MVC stricte.

Par ailleurs, pour arriver à nos fins, nous avons été amenés à créer un moteur MVC générique appelé [M2VC] utilisable par toute application swing dont le fonctionnement est réductible à celui d'une application web où seul l'utilisateur est à l'origine des événements. Ce n'est bien sûr pas le cas de toutes les applications swing.

Table des matières

1INTRODUCTION....2

2L'APPLICATION [SWINGARTICLES] - RAPPELS....3

2.1L'architecture 3-tier de l'application [swingarticles]....3
2.2La couche [ui] de l'application [swingarticles]....4
2.3Le modèle de l'application [swingarticles]....5
2.3.1La base de données....5
2.3.2Les paquetages du modèle....6
2.3.3Le paquetage [istia.st.articles.dao]....6
2.3.4Le paquetage [istia.st.articles.domain] initial....10
2.3.5Le paquetage [istia.st.articles.exception]....12
2.3.6Le paquetage [istia.st.articles.domain] revisité....12
2.4Les vues de l'application [swingarticles]....14
2.5Fonctionnement de l'application [swingarticles]....15
2.5.1Modification de l'application [swingarticles]....19

3INTRODUCTION À L'APPLICATION CLIENT-SERVEUR [MAGASIN-CLIENT-SERVEUR]....20

3.1Architecture de l'application client-serveur [magasin-client-serveur]....20
3.2Les méthodes exposées dans la couche [proxyServeur]....21
3.3Les couches d'interfaçage avec le réseau....23
3.4Sérialisation des objets sur le réseau....24

4[MAGASIN-CLIENT-SERVEUR] AVEC SPRING HTTPINVOKER....25

4.1Architecture....25
4.2Le serveur....25
4.2.1Le projet Eclipse du serveur....25
4.2.2Classes et interfaces du serveur....27
4.2.3Configuration de l'application web du serveur....28
4.2.3.1Le descripteur de déploiement [web.xml]....29
4.2.3.2Le contexte de l'application web Spring [applicationContext.xml]....29
4.2.3.3Le fichier de configuration [magasin-servlet.xml] de la servlet [magasin]....30
4.2.4Lancement du serveur....31
4.2.4.1Lancement du serveur Tomcat à partir d'Eclipse....31
4.2.4.2Lancement du serveur Tomcat en mode autonome....37
4.3Le client....38
4.3.1Le projet Eclipse du client....38
4.3.2Classes et interfaces du client....38
4.3.3Configuration du client [swingarticles]....41
4.4Les tests de l'application [magasin-client-serveur]....43

5[MAGASIN-CLIENT-SERVEUR] AVEC SPRING CAUCHO HESSIAN....44

5.1L'architecture....44
5.2Le serveur....45
5.3Le client....47
5.4Les tests....47

6[MAGASIN-CLIENT-SERVEUR] AVEC SPRING CAUCHO BURLAP....48

6.1L'architecture....48
6.2Le serveur....48
6.3Le client....49
6.4Les tests....50

7[MAGASIN-CLIENT-SERVEUR] AVEC SPRING RMI....51

7.1L'architecture....51
7.2Le serveur....51
7.3Le client....55
7.4Les tests....56

8CONCLUSION....57