



# Projet eMagine

Etude technique

## FICHE D'APPROBATION

Version	Date	Rédacteurs	
1.0		L. BARBISAN	A. OGIER
		Chef de projet	Responsable Documentaire
		J.-B. RENAUDIN	Responsable Client L. BARBISAN R. FORAX

## FICHE DE RÉVISION

Numéro de version actuel : 1.0

Version	Date	Auteur	Libellé
0.1	23/10/05	L.BARBISAN	Création du document
0.2	23/10/05	L.BARBISAN	Création du sommaire & des titres
0.3	26/10/05	L.BARBISAN	Ajout de la partie « Gestion du développement ».
0.4	26/10/05	A.OGIER	Ajout de la partie « Architecture Générale de l'application eMagine ».
0.5	28/10/05	A.OGIER	Ajout de la partie « Outils de développement Java ».
0.6	28/10/05	L.BARBISAN	Ajout de la partie « Test unitaire ».
0.7	01/11/05	A.OGIER	Ajout de la partie « Tests unitaires d'applications J2EE ».
0.8	03/11/05	A.OGIER	Ajout de la partie « Environnement de développement J2EE ».
0.9	03/11/05	A.OGIER	Ajout de la partie « Serveur d'application ».
0.10	05/11/05	A.OGIER	Ajout de la partie « Mise à jour dynamique d'une page Internet ».
0.11	06/11/05	A.OGIER	Ajout de la partie « Bibliothèque de génération de graphiques ».
0.12	06/11/05	A.OGIER	Ajout de la partie « Bibliothèque de génération d'emails ».
0.13	07/11/05	A.OGIER	Ajout de la partie « Bibliothèque de génération de feuilles excel ».
0.14	09/11/05	L.BARBISAN	Ajout de la partie « Outils de déploiements ».
0.15	09/11/05	L.BARBISAN	Ajout de la partie « Système de gestion de base de données ».
0.16	09/11/05	L.BARBISAN	Ajout de la partie « Persistance ».
0.17	10/11/05	L.BARBISAN	Ajout de la partie « Loggeur ».
0.18	12/11/05	L.BARBISAN	Ajout de la partie « Gestion de version ».
0.19	14/11/05	L.BARBISAN	Ajout de la partie « Test de performance ».
0.20	15/11/05	A.OGIER	Ajout de la partie « Implication dans le développement eMagine ».
0.21	15/11/05	L.BARBISAN	Ajout de la partie « Implication dans l'architecture eMagine ».
0.22	16/11/05	L.BARBISAN	Correction & Harmonisation des parties
0.1	01/12/05	J.RENAUDIN	Livraison version 1.0

## SOMMAIRE

<b>1. ARCHITECTURE GÉNÉRALE.....</b>	<b>6</b>
<b>1.1. GESTION DU DÉVELOPPEMENT.....</b>	<b>6</b>
a. Organisation de l'information.....	6
b. Outils de développement.....	8
<b>1.2. ARCHITECTURE GÉNÉRALE DE L'APPLICATION eMAGINE.....</b>	<b>9</b>
a. Diagramme.....	10
b. Détails de l'architecture.....	10
c. Interface Serveur d'application/Base de données.....	13
<b>2. ÉTUDE TECHNIQUE.....</b>	<b>13</b>
<b>2.1. OUTILS DE DÉVELOPPEMENT JAVA.....</b>	<b>13</b>
a. Choix d'Eclipse : Justification.....	13
b. Installation.....	14
c. Liens utiles.....	19
d. Installation des plugins d'aide au développement J2EE.....	19
<b>2.2. TEST UNITAIRE.....</b>	<b>22</b>
a. Justification.....	22
b. Utilisation.....	23
c. Tests et exemples.....	26
<b>2.3. TESTS UNITAIRES D'APPLICATIONS J2EE.....</b>	<b>32</b>
a. Choix de Cactus : Justification.....	32
b. Concept.....	33
c. Utilisation.....	35
d. Liens utiles.....	40
<b>2.4. ENVIRONNEMENT DE DÉVELOPPEMENT J2EE.....</b>	<b>41</b>
a. Choix de Struts : Justification.....	41
b. Utilisation.....	41
c. Liens utiles.....	49
<b>2.5. SERVEUR D'APPLICATION.....</b>	<b>50</b>
a. Choix de Tomcat : Justification.....	50
b. Utilisation.....	50
<b>2.6. MISE À JOUR DYNAMIQUE D'UNE PAGE INTERNET.....</b>	<b>51</b>
a. Choix d'AJAX : Justification.....	51
b. Utilisation.....	52
<b>2.7. BIBLIOTHÈQUE DE GÉNÉRATION DE GRAPHIQUES.....</b>	<b>57</b>
a. Choix de Cewolf : Justification.....	57
b. Utilisation.....	57
<b>2.8. BIBLIOTHÈQUE DE GÉNÉRATION D'EMAILS.....</b>	<b>63</b>
a. Choix de JavaMail : Justification.....	63
b. Concept.....	63
c. Liens utiles.....	68
<b>2.9. BIBLIOTHÈQUE DE GÉNÉRATION DE FEUILLES EXCEL.....</b>	<b>69</b>
a. Choix de Java POI : Justification.....	69
b. Concept.....	70
<b>2.10. OUTILS DE DÉPLOIEMENTS.....</b>	<b>73</b>
a. Comparatif.....	73
b. Justification.....	73
c. Tests et exemples.....	75
<b>2.11. SYSTÈME DE GESTION DE BASE DE DONNÉES.....</b>	<b>76</b>
a. Comparatif.....	76
b. Justification.....	77
c. Utilisation.....	77
d. Tests et exemples.....	79
<b>2.12. PERSISTANCE.....</b>	<b>88</b>
a. Comparatif.....	88
b. Justification.....	89
c. Tests et exemples.....	89
<b>2.13. LOGGEUR.....</b>	<b>96</b>
a. Comparatif.....	96
b. Justification.....	96
c. Utilisation.....	96
<b>2.14. GESTION DE VERSION.....</b>	<b>100</b>
a. Comparatif.....	100
b. Justification.....	101

c. Concept.....	101
d. Tests et exemples.....	109
<b>2.15. TEST DE PERFORMANCE.....</b>	<b>122</b>
a. Justification.....	122
b. Utilisation.....	122
c. Tests et exemples.....	144
<b>3. ARCHITECTURE FINALE.....</b>	<b>152</b>
<b>3.1. IMPLICATION DANS LE DÉVELOPPEMENT eMAGINE.....</b>	<b>152</b>
a. IDE : Eclipse.....	153
b. Tests unitaires d'applications J2EE : Cactus.....	153
c. Outils de déploiement : Ant.....	153
d. Outils de gestion de version : SubVersion.....	153
e. JUnit .....	154
f. JMeter .....	154
<b>3.2. IMPLICATION DANS L'ARCHITECTURE eMAGINE.....</b>	<b>155</b>
a. Environnement de développement J2EE : Struts.....	156
b. Serveur d'applications : Tomcat.....	156
c. Mise à jour dynamique d'une page : AJAX.....	156
d. Génération de graphiques : Cewolf.....	156
e. API mails : JavaMail.....	157
f. Génération de fichiers Excel : Java POI et HSSF.....	157
g. Logueur : Log4j.....	157
h. Hibernate.....	157

## INTRODUCTION

Ce document a pour but de décrire les solutions techniques générales envisagées pour la mise en place de l'application.

Tout d'abord, une présentation de l'ensemble des solutions techniques étudiées sera effectuée. Le second chapitre décrit le comparatif de ces solutions et désigne celles qui seront retenues définitivement.

Finalement, la dernière partie explique l'intégration des solutions techniques retenues, ainsi que les choix et les contraintes que cela impose pour conclure sur la présentation de l'architecture finale.

# 1. ARCHITECTURE GÉNÉRALE

---

L'architecture se décompose en deux sous parties :

**Gestion du développement** : présente les moyens mis en oeuvre pour la gestion du développement au sein de l'équipe eMagine.

**Architecture d'eMagine** : présente l'architecture générale de l'application eMagine.

## 1.1. GESTION DU DÉVELOPPEMENT

Pour le développement de eMagine, plusieurs outils ont été mis en place afin de permettre un développement rapide et efficace. Le plus important à noter est l'IDE<sup>1</sup> Eclipse qui permet de centraliser à peu près tous les applications nécessaires au développement. L'organisation de l'information sera présentée pour expliquer où se trouvent les ressources de développement.

### a. Organisation de l'information

L'organisation de l'information explique où trouver les ressources nécessaires au développement.

La gestion de l'information peut être découpée comme suit :

#### ➤ Gestion du projet

Permet de gérer toutes les données liées au projet.

#### ➤ Gestion de la communication extérieure

Permet de communiquer l'état d'avancement du projet au public.

#### ➤ Gestion des ressources

Permet de gérer toutes les ressources utilisées pour créer le logiciel.

#### ➤ Gestion de la documentation

Permet d'effectuer le suivi de la documentation.

#### ➤ Gestion des sources

Permet d'effectuer le suivi des sources.

#### ➤ Gestion de la communication Équipe

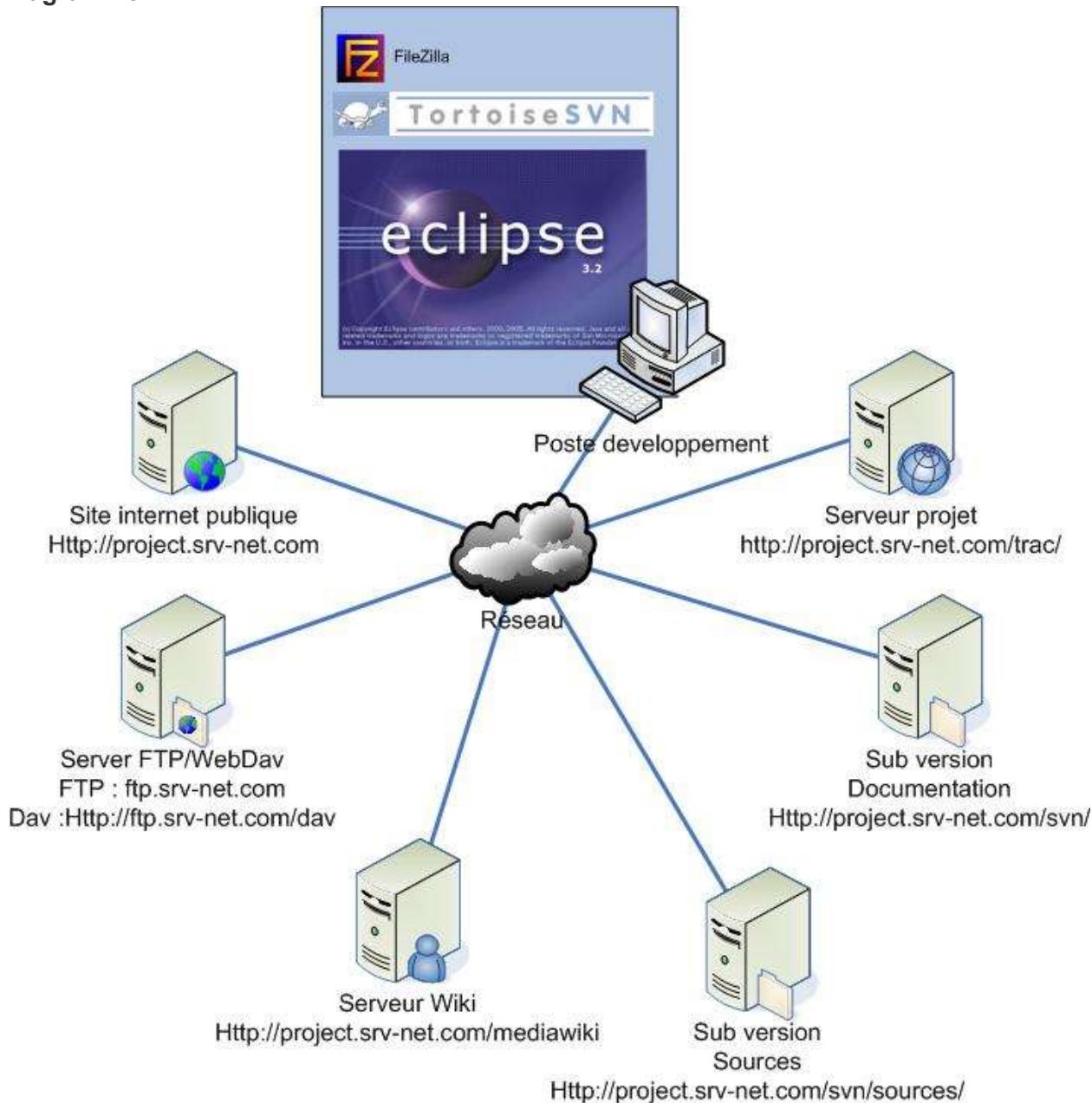
Permet de faire communiquer les membres de l'équipe entre eux.

---

<sup>1</sup> Environnement de Développement Intégré (EDI ou IDE en anglais pour *Integrated Development Environment*)

Le diagramme ci-après montre la répartition des différentes informations.

## ➤ Diagramme



## ➤ Gestion de projet : Serveur projet

Le serveur projet est hébergé par le responsable site de l'équipe. Il permet de gérer les tickets de bogue, afin de répertorier et historier chaque problème relevé par les programmeurs. L'applicatid utilisé sera trac. Un forum est aussi à leur disposition pour poser des questions, résoudre des problèmes, etc. Il permet aussi de gérer le versionning de sources.

### ➤ **Gestion de la communication extérieure : Serveur Site internet**

Le serveur internet est hébergé par le responsable site de l'équipe. Le site internet permet de suivre le déroulement du développement du projet par le client. Il contient les informations officielles et les téléchargements divers. C'est l'interface entre les acheteurs potentiels et eMagine.

### ➤ **Gestion des ressources : Serveur FTP/WebDav**

Le serveur FTP est utilisé pour stocker les ressources qui n'ont pas besoin d'être gérées à l'aide de Subversion. Notamment les documents figés, comme le sujet du projet, les logiciels utilisés (fileZilla, etc).

### ➤ **Gestion de la documentation : Serveur Subversion documentation**

Le serveur de Subversion pour la documentation permet de stocker tous les documents livrables (les cahiers des charges entre autres). Cela permet d'avoir un historique complet de la rédaction du fichier et de gérer les problèmes de modifications simultanées.

### ➤ **Gestion des sources : Serveur Subversion sources**

Le serveur de Subversion pour les sources permet de stocker toutes les sources d'eMagine. Cela permet d'avoir un historique complet des sources et de gérer les problèmes de modifications simultanées.

### ➤ **Gestion de la communication Équipe : Serveur Wiki**

Un wiki est un site Web dynamique permettant à tout individu d'en modifier les pages à volonté. Il permet non seulement de communiquer et de diffuser des informations rapidement, mais aussi de structurer cette information pour permettre d'y naviguer commodément. Il réalise donc une synthèse des forums Usenet, des FAQ dans le World Wide Web en une seule application intégrée. Actuellement, mediawiki est utilisé par l'équipe eMagine.

## b. Outils de développement

Les outils de développement permettent d'augmenter la rentabilité. Les principaux outils sont les bibliothèques qui permettent de réutiliser des fonctions écrites par un autre programmeur.

### ➤ **Langage de programmation**

Le langage utilisé pour la programmation sera Java. Ce langage de programmation est orienté objet ce qui permet un développement rapide grâce à l'aspect objet. De plus, l'engouement des programmeurs pour le langage Java engendre la création de nombreuses bibliothèques pour simplifier les développements. En effet, l'utilisation d'une bibliothèque évite le développement de l'outil nécessaire, et donc tous les tests.

## ➤ Logiciels

### **Logiciel de développement**

L'IDE (Integrated Developpement Environment) doit offrir une interface très intuitive, ce qui permet de gagner du temps dans toutes les tâches fastidieuses de développement. Il doit permettre le développement java et intégrer de nombreux modules.

### **Logiciel de gestion des sources**

La gestion des sources est faite à l'aide de Subversion. Ce logiciel est le successeur de CVS déjà très réputé dans le monde du développement Open Source. Actuellement, il comble la plupart des failles de CVS. La gestion des sources permet de garder un historique complet des développements. Ainsi s'il est nécessaire de revenir en arrière Subversion le permet.

## ➤ Plugins Eclipse

### **Déploiement de l'application**

Le déploiement de l'application est géré avec Ant. Celui-ci est un projet développé sous l'égide de Apache par la communauté Jakarta. Jakarta est un des acteurs les plus actifs dans le monde Java. Il est connu pour de nombreuses participations. Notamment POI (gestion des outils Excel et Word). Ant permet d'automatiser les tâches de compilation, de distribution, et d'exécution. De plus, il permet aussi d'effectuer de nombreuses autres actions.

### **Mise en place des tests unitaires**

Les tests unitaires permettent de tester l'application, et d'éviter les régressions. Il devra exister une bibliothèque intégrée à l'environnement de développement qui permettra de générer et d'exécuter des tests unitaires. Le test est valide si tous les tests se sont bien déroulés sinon il a échoué.

### **Mise en place des tests de performance**

Les tests de performance permettent d'évaluer le temps de réponse du logiciel et d'apporter des corrections sur les sections lentes de l'application (goulots d'étranglements ou autres).

### **Aide au débogage**

Une aide au débogage est nécessaire, notamment lors du déroulement d'un programme. Elle doit signaler au programmeur dans quelle phase se trouve le programme Java. Cela permettra d'obtenir de précieuses informations quant au développement de l'application.

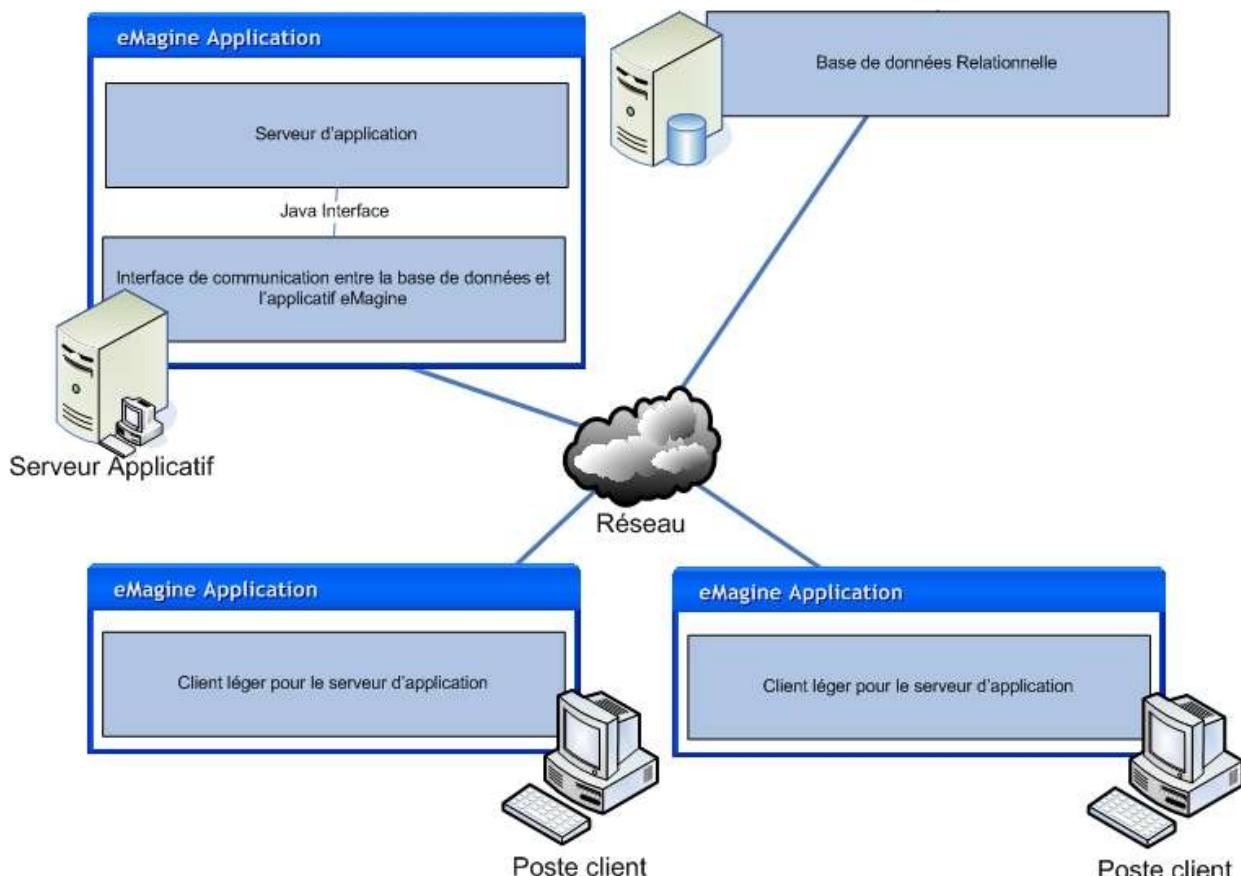
## **1.2. ARCHITECTURE GÉNÉRALE DE L'APPLICATION eMAGINE**

Afin de trouver des solutions techniques, une description générale de l'architecture de eMagine est nécessaire.

## a. Diagramme

eMagine sera une application du type client/serveur avec une base de données relationnelle permettant le stockage des données. Chaque utilisateur à l'aide d'un navigateur pourra accéder au serveur pour faire ses modifications.

Le diagramme suivant montre le principe de base :



**Illustration 1 : Architecture générale eMagine**

## b. Détails de l'architecture

Ce chapitre étudie les différentes technologies utilisables pour répondre à cette architecture.

### ➤ Base de données

La base de données contient les informations du logiciel, il existe sur le marché quatre grands Système de Gestion de Bases de Données, à savoir :

**Microsoft SQL Server** : Utilisation simple, mais propriétaire, et qui demande l'achat d'une licence.

**Oracle** : Utilisation professionnelle qui demande des connaissances pointues pour être maîtrisé.

**MySql** : Utilisation orientée web.

**PostgreSQL** : Serveur de base de données relationnelle le plus abouti.

JDBC (Java DataBase Connection) permet de dialoguer avec différentes bases de données, en utilisant la même interface, quelque soit sa structure.

## ➤ Serveur d'application

Le serveur d'application permet de mettre en place une architecture client/serveur avec une application centralisée sur un serveur.

Il existe sur le marché deux principaux serveurs d'application :

**JBoss** : Il a de nombreuses particularités. La principale concerne son architecture qui est de type micro-kernel. L'intégralité est développée sous forme de module JMX.

Les différents modules couvrent la norme J2EE :

- Jboss MQ gère les messages Java (Java Messaging System),
- JBoss MX gère la messagerie électronique,
- JBoss TX gère les transactions JTA/JTS,
- JBoss SX gère la sécurité (basée sur JAAS),
- JBoss CX gère la connectivité JCA,
- Jboss CMP gère la persistance CMP.

Chacun de ces modules peut être remplacé par un module au format JMX. Trois types de configuration sont installés en standard : minimum, default et all.

**Tomcat** : Le serveur Tomcat est un serveur Open Source qui agit comme un conteneur de servlet J2EE. Il fait partie du projet Jakarta, au sein de la fondation Apache. Tomcat implémente les spécifications des servlets et des JSP de Sun Microsystems. Incluant un serveur HTTP interne, il est aussi considéré comme un serveur HTTP.

## ➤ Client léger

Les clients légers sont des interfaces web accessibles via n'importe quel navigateur. Différentes contraintes techniques apparaissent lors de la mise en place.

### **Bibliothèque AJAX**

Le client léger, doit avoir un temps de réponse convenable, c'est pour cette raison que la technologie AJAX va être utilisée. Pour éviter d'avoir à tout re-développer, de nombreux framework existent :

- Dojo

- DWR
- JSON-RPC-JAVA
- MochiKit
- Prototype
- Rico
- SAJAX
- Scriptaculous
- Xajax
- Sack

### **JavaMail**

JavaMail est une bibliothèque de gestion des mails, elle supporte le protocole POP et STMP.

### **Java POI**

POI est un projet de Jakarta d'Apache Software Foundation permettant de manipuler divers types de fichiers créés par Microsoft avec le langage Java. Ainsi avec POI, nous avons la possibilité de travailler avec des fichiers Excel et Word.

### **Corba**

CORBA, acronyme de Common Object Request Broker Architecture, est une architecture logicielle, pour le développement de composants et d'Object Request Broker ou ORB. Ces composants, qui sont assemblés afin de construire des applications complètes, peuvent être écrits dans des langages de programmation distincts, être exécutés dans des processus séparés, voire être déployés sur des machines distinctes.

### **RMI**

**Remote Method Invocation**, plus connu sous l'acronyme RMI est une interface de programmation (API) pour le langage Java qui permet d'appeler des objets distants. L'utilisation de cette API nécessite l'emploi d'un registre RMI sur la machine distante. Cette machine héberge les objets que l'on désire appeler et utiliser. Cette API est utilisée très souvent en parallèle avec l'API d'annuaire JNDI ou encore avec la spécification de composants distribués transactionnels EJB du langage Java.

### **Applet**

Une applet est un programme Java qui s'exécute dans la fenêtre d'un navigateur Web.

Cette approche offre le moyen de fournir à l'utilisateur, sans installation d'un logiciel ad-hoc (souvent appelé client lourd), une application ergonomique et réactive. Au final, le code sera exécuté sur le poste client et non pas sur le poste hébergeant.

Un navigateur Web basique n'offre qu'une interface de présentation d'informations, il est incapable de les traiter. Lui assigner certaines tâches réduit les communications réseau ainsi que le volume de travaux (charge) imposé au serveur, donc mobilise les ressources (CPU, mémoire informatique...) du poste client afin d'améliorer la fluidité en réduisant les latences. De surcroît, une applet offre au développeur un moyen d'employer certaines ressources du poste client qui demeurent hors de portée du

HTML et améliorent l'ergonomie de l'application, par exemple grâce à des éléments d'interface graphique. Tout navigateur contemporain abrite pour cela un environnement d'exécution d'applets.

### **Struts**

Apache Struts est un cadre d'applications open-source pour développer des applications web J2EE. Il utilise et étend l'API Servlet Java afin d'encourager les développeurs à adopter l'architecture MVC (Modèle Vue Contrôleur). Apache Struts a été créé par Craig McClanahan et donné à la fondation Apache en mai 2000.

Un plugin eclipse est disponible pour l'utilisation de struts.

### **JfreeChart**

JFreeChart est une librairie graphique qui permet de créer des graphiques de formes très variées à partir de données.

On notera parmi les fonctionnalités la possibilité d'insérer un deuxième axe ou d'effectuer des exports sous divers formats : PNG, JPEG mais aussi PDF ou SVG.

### **c. Interface Serveur d'application/Base de données**

L'interface devra permettre d'éviter les accès intempestifs à la base de données. Il existe actuellement une bibliothèque réputée pour cela : Hibernate. Les deux autres solutions seraient soit de créer l'interface, mais les délais de développement deviendraient beaucoup trop élevés; soit d'utiliser les EJBs. Ces solutions ne seront pas étudiées.

### **Hibernate**

Hibernate utilise une base de données et des données de configuration pour fournir un service de persistance (et des objets persistants) à l'application. L'architecture la plus complète abstrait l'application des APIs JDBC/JTA sous-jacentes et laisse Hibernate s'occuper des détails. Cette bibliothèque permet donc de manipuler les éléments d'une base de données sans ses contraintes.

## **2. ÉTUDE TECHNIQUE**

---

Java 2 Platform, Enterprise Edition (J2EE) est une spécification pour le langage de programmation Java de Sun plus particulièrement destinée aux applications d'entreprise. Dans ce but, il contient un ensemble de technologies standardisées permettant de faciliter la création d'applications réparties.

### **2.1. Outils de développement Java**

#### **a. Choix d'Eclipse : Justification**

Eclipse est un IDE (*Integrated Development Environment*) très complet. Il est spécialisé pour le développement d'application Java. Il possède également un système pour accueillir des « plugins » et ainsi augmenter ses fonctionnalités.

Nous l'avons choisi pour plusieurs raisons. Tout d'abord, c'est un logiciel libre (*open source*) et gratuit. Tous les membres de l'équipe peuvent donc l'avoir à disposition.

D'autre part, il possède tous les plugins dont nous avons besoins pour le développement de nos différentes parties, notamment des plugins permettant de gérer Tomcat, Hibernate ou encore Subversion.

### b. Installation

Eclipse 1.0 peut être installé sur les plate-formes Windows ( 98ME et SE / NT / 2000 / XP) et Linux.

Eclipse 2.0 peut être installé sur les plate-formes Windows ( 98ME et SE / NT / 2000 / XP), Linux (Motif et GTK), Solaris 8, QNX, AIX 5.1, HP-UX 11.

Eclipse 2.1 peut être installé sur toutes les plate-formes citées précédemment mais aussi sous MAC OS X.

Quelle que soit la plate-forme, il faut obligatoirement qu'un JDK 1.3 minimum y soit installé. La version 1.4 est fortement recommandée pour améliorer les performances et pouvoir utiliser le remplacement de code lors du débogage (technologie JPDA).

Lors de son premier lancement, Eclipse crée par défaut un répertoire nommé Workspace qui va contenir les projets et les éléments qui les composent.

Le principe pour l'installation de toutes les versions d'Eclipse reste le même et d'une grande simplicité puisqu'il suffit de décompresser le contenu de l'archive d'Eclipse dans un répertoire du système.

#### ➤ Installation d'Eclipse SDK 3.1.1 sous Windows

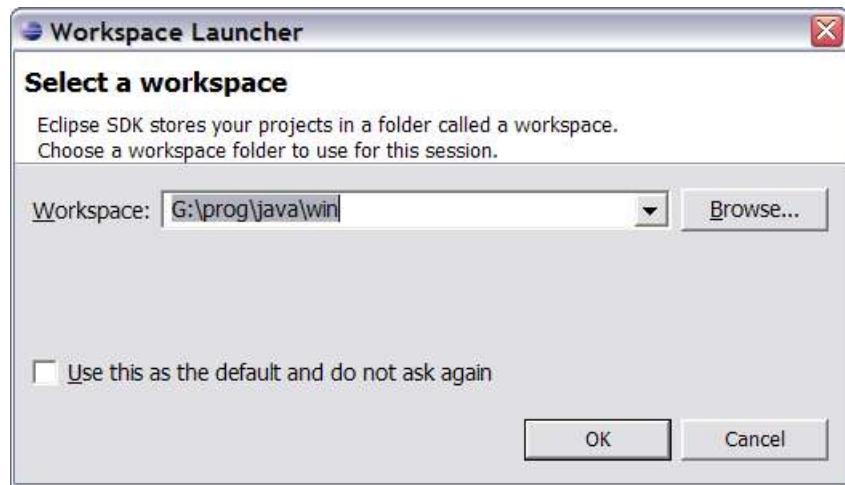
Il faut télécharger le fichier eclipse-SDK-3.1.1-win32.zip sur le site :

<http://www.eclipse.org/downloads/index.php>

L'installation d'Eclipse 3.1 se fait de la même façon que pour les versions antérieures. Il suffit d'extraire l'archive dans un répertoire, par exemple c:\Program Files\.

Pour lancer Eclipse, il suffit de lancer le fichier eclipse.exe.

Durant la phase de lancement, Eclipse 3 propose de sélectionner l'espace de travail à utiliser. Par défaut, c'est celui présent dans le répertoire workspace du répertoire d'Eclipse qui est proposé.



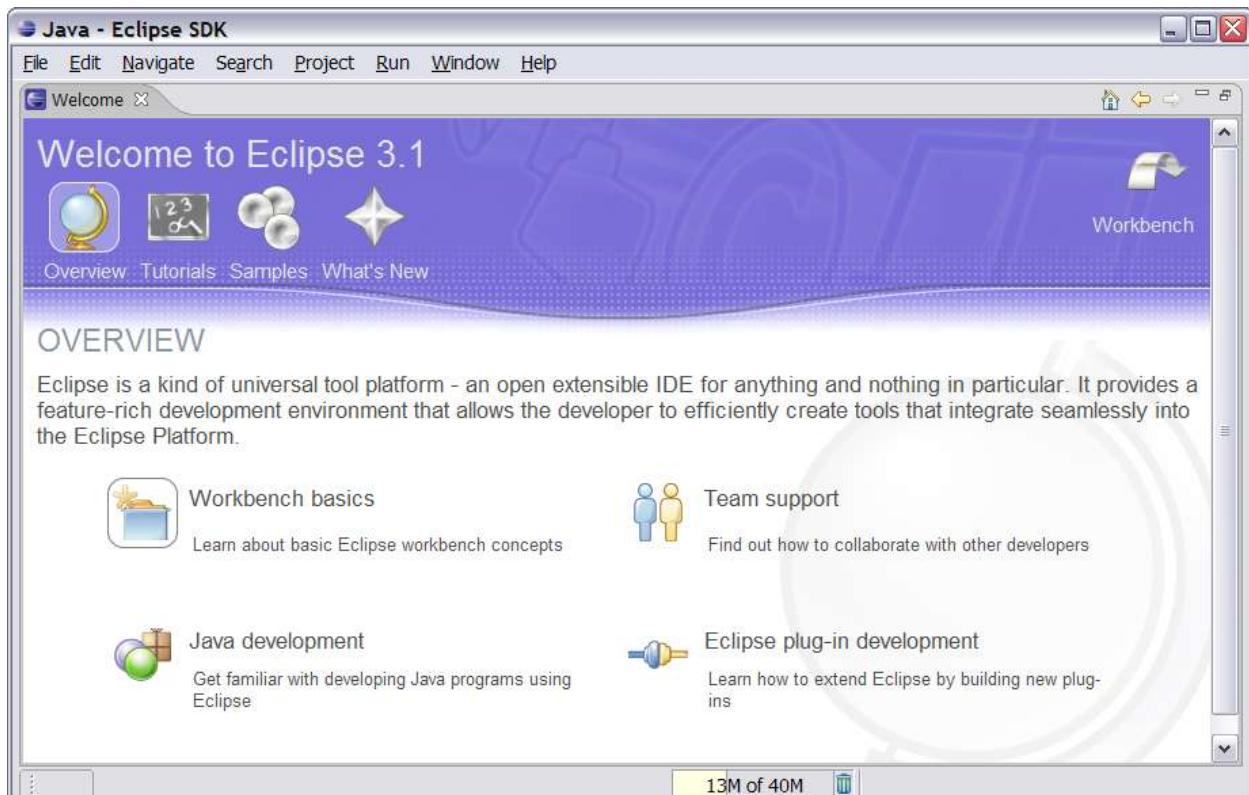
Cette demande est particulièrement utile lors de l'utilisation de plusieurs espaces de travail. Si un seul et unique workspace est utilisé, le plus simple est de cocher la case « Utiliser par défaut et ne plus poser la question » avant de cliquer sur le bouton « OK ». L'espace précisé deviendra alors celui par défaut.



Eclipse 3 possède une page d'accueil permettant d'obtenir des informations sur l'outil réparties en quatre thèmes :

- overview : permet d'accéder rapidement à la partie de l'aide en ligne correspondant au thème sélectionné

- tutorials : permet d'accéder à des assistants qui permettent sous la forme de didacticiel de réaliser de simples applications ou plugins
- samples : permet de lancer des exemples d'applications avec SWT et JFace qu'il faut télécharger sur internet
- what's new : permet d'accéder rapidement à la partie de l'aide en ligne concernant les nouveautés d'Eclipse 3



Eclipse 3 possède une nouvelle interface liée à une nouvelle version de SWT.

#### ➤ Installation d'Eclipse 3.0.x sous Mandrake 10.1 pour un et un seul utilisateur

Il faut obligatoirement qu'un JDK 1.3 minimum soit installé sur le système. Dans cette section le JDK utilisé est le 1.4.2 de Sun.

Exemple :

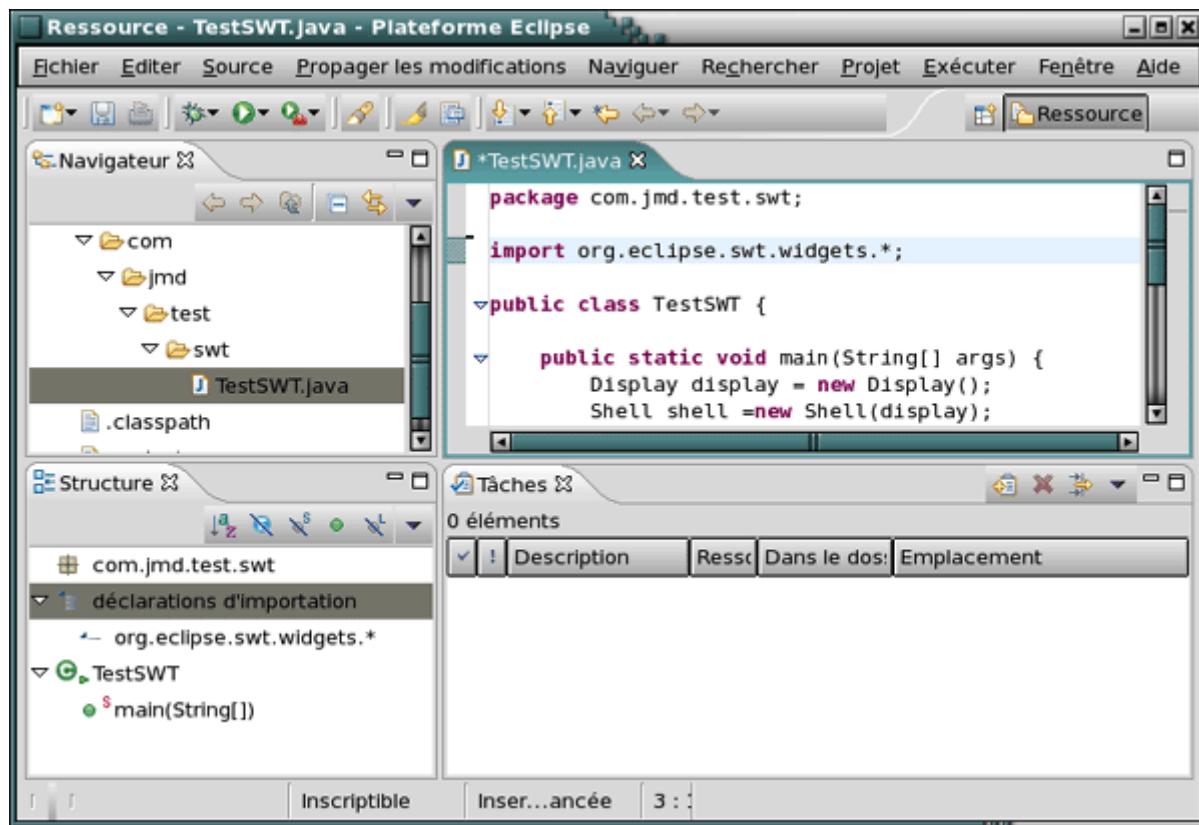
```
[java@localhost eclipse]$ java -version
java version "1.4.2_06"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_06-b03)
Java HotSpot(TM) Client VM (build 1.4.2_06-b03, mixed mode)
```

Deux versions existent pour Linux selon la bibliothèque graphique utilisée :

- une utilisant Motif

- une utilisant GTK : eclipse-SDK-3.0-linux-gtk.zip

La version GTK sera utilisée dans cette section.



Le plus simple est de décompresser le fichier `eclipse-SDK-3.0-linux-gtk.zip` dans le répertoire home de l'utilisateur.

L'inconvénient de cette méthode est que par défaut seul cet utilisateur pourra utiliser Eclipse.

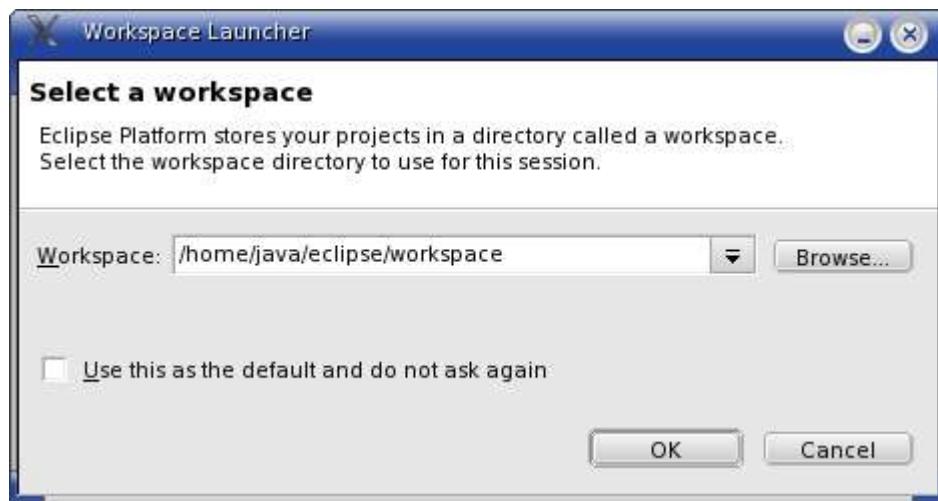
#### Exemple :

```
[java@localhost eclipse]$ cp eclipse-SDK-3.0-linux-gtk.zip ~
[java@localhost eclipse]$ cd ~
[java@localhost java]$ unzip eclipse-SDK-3.0-linux-gtk.zip
  creating: eclipse/plugins/org.eclipse.pde.build_3.0.0/
  inflating: eclipse/plugins/org.eclipse.pde.build_3.0.0/pdebuild.jar
  creating: eclipse/plugins/org.eclipse.pde.build_3.0.0/lib/
  inflating: eclipse/plugins/org.eclipse.pde.build_3.0.0/lib/pdebuild-ant.jar
  inflating: eclipse/plugins/org.eclipse.pde.build_3.0.0/.options
  creating: eclipse/plugins/org.eclipse.pde.build_3.0.0/feature/
...
  inflating: eclipse/startup.jar
  inflating: eclipse/icon.xpm
  inflating: eclipse/eclipse
[java@localhost java]$ ll
total 87140
drwxr-xr-x  3 java java    4096 oct 16 21:24 Desktop/
drwxr-xr-x  2 java java    4096 oct 16 22:34 Documents/
drwxr-xr-x  6 java java    4096 oct 18 23:23 eclipse/
```

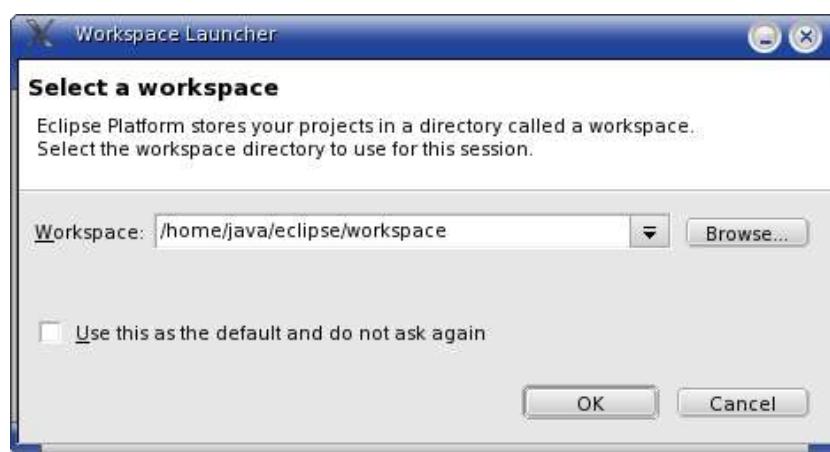
```
-rwxr--r-- 1 java java 89113015 oct 18 23:23 eclipse-SDK-3.0-linux-gtk.zip*
-rw-rw-r-- 1 java java 2 oct 16 22:23 java.sh~
drwx----- 3 java java 4096 oct 18 23:17 tmp/
[java@localhost java]$ cd eclipse
[java@localhost eclipse]$ ll
total 100
drwxrwxr-x 2 java java 4096 jun 25 18:43 configuration/
-rw-rw-r-- 1 java java 15049 jun 25 18:43 cpl-v10.html
-rwxr-xr-x 1 java java 27119 jun 25 18:43 eclipse*
drwxr-xr-x 9 java java 4096 oct 18 23:23 features/
-rw-rw-r-- 1 java java 10489 jun 25 18:43 icon.xpm
```

Le fichier `eclipse-SDK-3.0-linux-gtk.zip` peut être supprimé.

Le workspace à utiliser peut être celui proposé par défaut (celui dans le répertoire d'installation d'Eclipse)



L'apparence d'Eclipse dépend du thème et de l'environnement graphique utilisé, ici sous KDE :



### c. Liens utiles

Nous verrons dans les diverses parties abordées par la suite, quels plugins apporter à Eclipse. Un site regroupant beaucoup de tutoriels consacrés à Eclipse se trouve à l'adresse suivante :

<http://perso.wanadoo.fr/jm.doudoux/java/dejae/indexavecframes.htm>

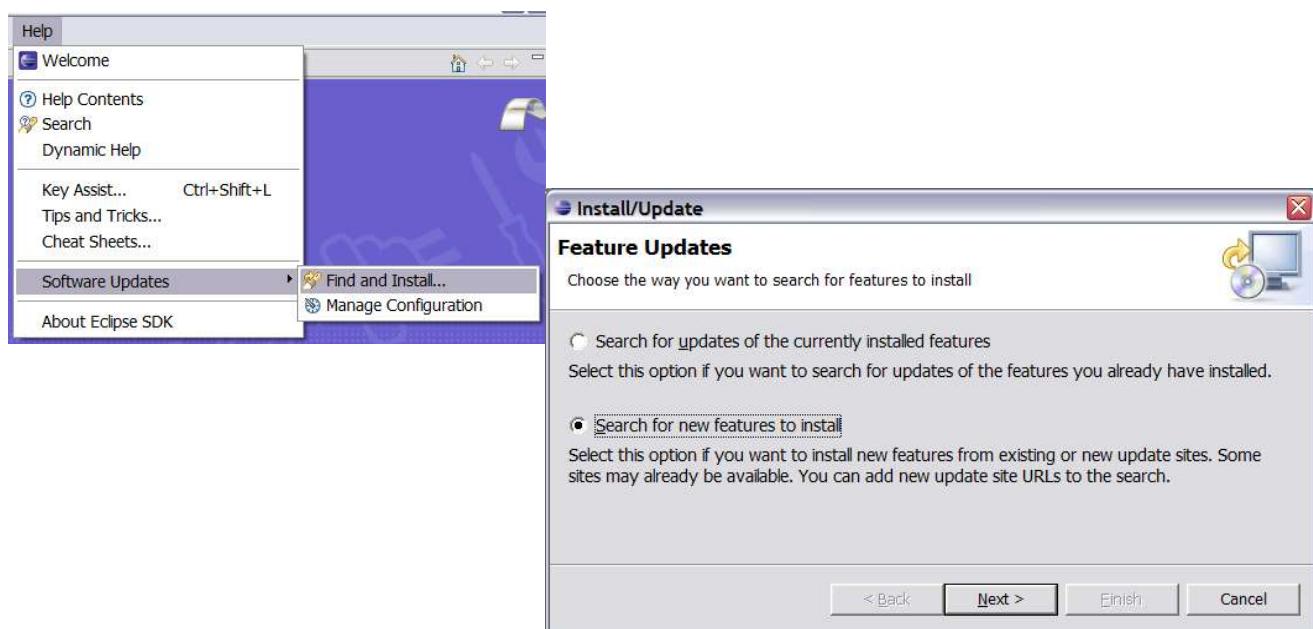
### d. Installation des plugins d'aide au développement J2EE

Le projet eclipse regroupe de nombreux autres petits projets créés autour de l'IDE et qui permettent de le rendre encore plus utile et pratique.

Pour le développement d'applications web J2EE, il existe deux projets : JST (J2EE Standard Tools) et WST (Web Standard Tools). Ceux-ci permettent notamment d'avoir une saisie assistée pour bon nombres de fichiers et de technologies standards J2EE.

Voici un tutoriel expliquant comment installer ces deux plugins.

Dans un premier temps, il faut accéder à l'assistant Eclipse pour les mises à jour.



Il faut ensuite choisir le site des mises à jour. Par défaut, le site d'eclipse.org est satisfaisant.

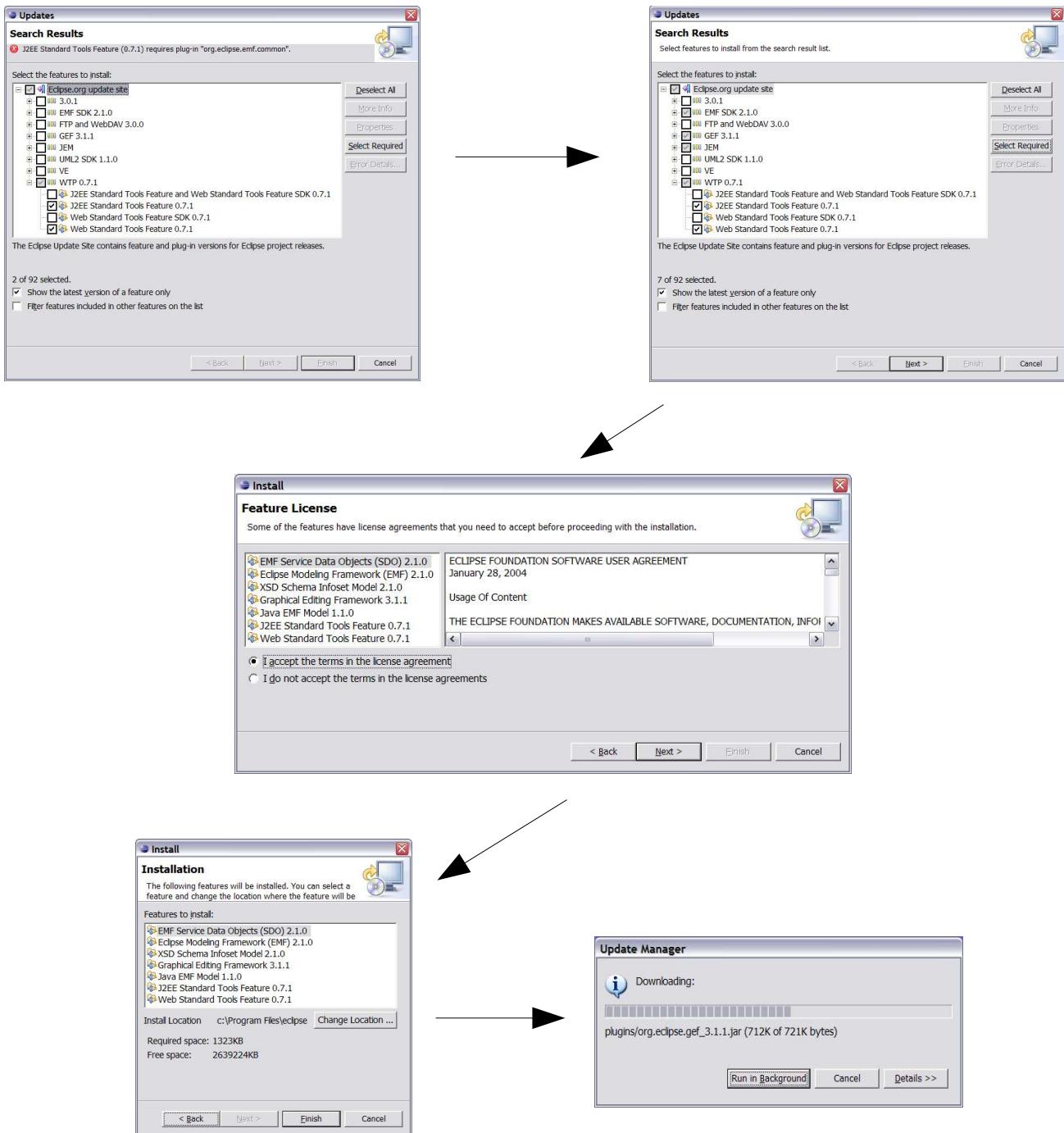


Puis nous choisissons un miroir de téléchargement. Pour la France, le miroir de suisse est assez rapide :

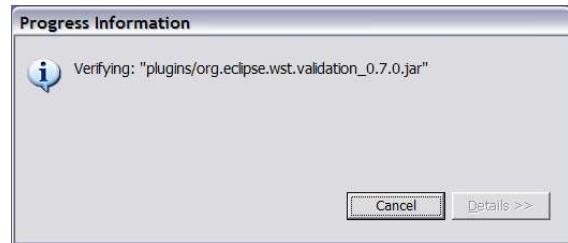
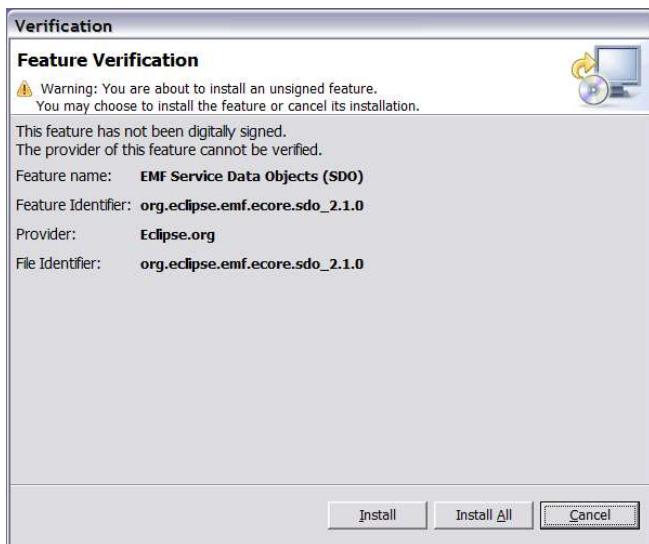


Il s'agit ensuite de sélectionner les 2 plugins JST et WST, puis de cliquer sur « Select Required » afin de sélectionner les plugins dont dépend JST et WST :

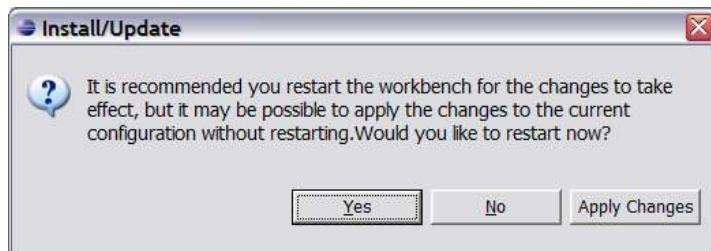
Après l'acceptation des licences, et un récapitulatif de ce qui va être installé, l'installateur télécharge les librairies des plugins.



Ensuite, un message nous avertit que des librairies ne sont pas vérifiées. Il faut les installer tout de même (« Install All »), elles sont vérifiées par la suite.



Enfin, un message vous invite à redémarrer Eclipse, ce qui est conseillé de faire.



## 2.2. TEST UNITAIRE

Les tests unitaires permettent de s'assurer de ne pas régresser dans le développement (correction d'un bogue qui en fait apparaître un autre).

### a. Justification

- Parce que pour une raison ou pour une autre, nous devons programmer efficacement. nous ne pouvons pas nous permettre de sortir des sentiers battus en programmant "hors sujet" ou en oubliant une fonctionnalité. Notre seul objectif en programmant est alors de valider les tests unitaires.
- Toujours dans un soucis d'efficacité, nous ne pouvons pas passer non plus notre temps à chercher un bogue. En développant un test unitaire, nous ajoutons une marge de sécurité et nous laissons moins de bogues derrière nous.

- Parce que nous voulons savoir si nous ne sommes pas en train de faire régresser notre code. JUnit nous dira tout de suite si nous avons perdu une fonctionnalité en cours de route, par exemple après avoir supprimé 200 lignes de code en pensant bien faire.

Réaliser des tests unitaires avec JUnit n'est pas seulement une alternative aux "println", c'est aussi une plus grande chance de satisfaire un objectif dans le but d'améliorer la qualité du logiciel.

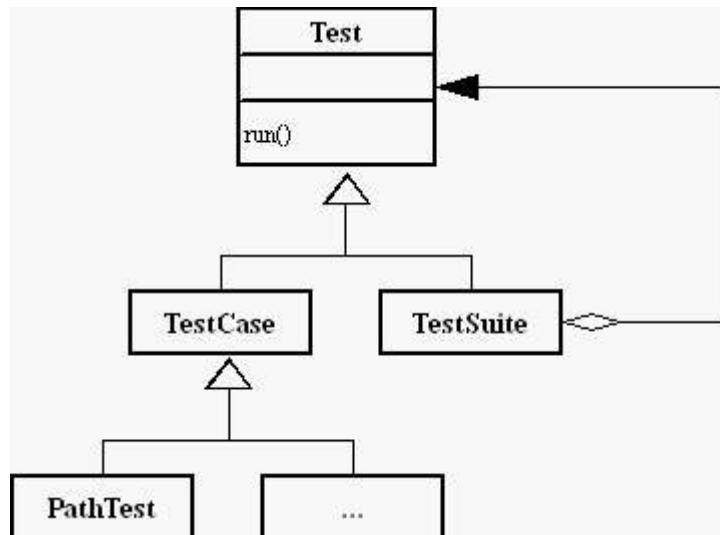
### b. Utilisation

Puisque le nombre de tests unitaires croît rapidement au cours du projet, il est impensable de les écrire à l'intérieur d'une simple fonction main(). Nous organisons plutôt ces tests en classes, en créant une classe de test par classe à tester.

Toutes nos classes de test sont gérées par un framework dont la seule fonction est d'exécuter en séquence les tests ainsi définis. Le framework de référence en la matière se nomme **JUnit**, il est l'oeuvre conjointe de Kent Beck (créateur de XP) et Erich Gamma (auteur des *Design Patterns*). Cet outil est gratuit et téléchargeable à partir du site junit.org. Il est écrit en Java, mais des frameworks gratuits inspirés de celui-ci sont également disponibles pour de nombreux autres langages (C++, Python...).

JUnit définit trois notions principales :

- Un **TestCase** est typiquement une classe de test. Chacune de nos classes de test dérive de cette classe, qui elle-même dérive de la classe **Test**.
- Une **TestSuite** est un ensemble de tests. Elle dérive également de la classe **Test** et contient des instances de la classe **Test** (suivant le *Design Pattern* "Composite"), ce qui lui permet de contenir indifféremment des **TestCase** ou d'autres **TestSuite** :



- Un **TestRunner** permet de lancer l'exécution d'une liste de **Test**.

Pour utiliser ce framework, il suffit donc d'écrire des classes de test dérivées de `TestCase`, placer une instance de chaque test dans une hiérarchie de `TestSuite` et enfin donner le tout à un `TestRunner` pour qu'il l'exécute. Les pages suivantes montrent comment cela se fait en pratique.

## ➤ Écriture d'une classe de test

Chaque classe de test dérive de `TestCase` :

```
public class PathTest extends TestCase {
```

Le constructeur rappelle celui de la classe de base en lui passant le nom du test :

```
public PathTest(String name) {
    super(name);
}
```

Les tests proprement dits sont écrits sous forme de méthodes de cette classe, par exemple :

```
public void testLastComponent() {
    Path p = new Path("/usr/include/stdcuh.h");
    assertEquals("stdcu.h", p.getLastComponent());
}

public void testLastComponentWithEmptyPath() {
    Path p = new Path("/");
    assertEquals("", p.getLastComponent());
}
```

A l'intérieur de ces méthodes les tests sont principalement réalisés à l'aide des méthodes suivantes :

- `assert(boolean expression)` vérifie que l'expression logique est vraie,
- `assertNotNull(Object ref)` vérifie qu'une référence d'objet est valide,
- `assertEquals(Object value, Object expression)` vérifie que l'expression renvoie une valeur donnée (la comparaison est bien sûr faite avec la méthode `Object.equals()`),
- `fail(String message)` fait échouer le test en affichant un message. Cette méthode est utile pour s'assurer qu'on ne traverse pas une section de code donnée, en particulier lorsqu'on s'attend à recevoir une exception.

Il arrive parfois que toutes les méthodes d'une classe de test partagent des portions identiques de code pour l'initialisation des ressources et leur libération. Ces fonctions peuvent être factorisées en surchargeant des méthodes dédiées de `TestCase` : `setUp()` et `tearDown()`, qui seront appelées respectivement avant et après l'exécution de chaque méthode de test. Il faut bien noter qu'**une instance de test est créée pour chaque méthode à exécuter**, cela peut avoir une grande importance si la classe comporte des données membres par exemple.

Une fois la classe écrite, on ajoute par convention une méthode statique `suite()` dont le rôle est de construire les instances à exécuter. Notre méthode `suite()` construit les instances et les place dans une `TestSuite` commune :

```
public static Test suite() {  
    TestSuite suite= new TestSuite();  
    suite.addTest(new PathTest("testLastComponent"));  
    suite.addTest(new PathTest("testLastComponentWithEmptyPath"));  
    return suite;  
}
```

Chaque instance est construite avec le nom de la méthode associée, ensuite JUnit utilise les outils d'introspection Java pour appeler la méthode en question. Il est également possible de s'appuyer davantage sur l'introspection en laissant à JUnit le soin de construire les instances :

```
public static Test suite() {  
    return new TestSuite(PathTest.class);  
}
```

Dans ce cas JUnit construit une instance de la classe pour chaque méthode dont le nom commence par "test". Nous préférons en général cette approche puisque la méthode `suite()` n'a plus besoin d'être mise à jour lorsqu'une nouvelle méthode est ajoutée.

#### ➤ Ecriture du main() et lancement

Une fois la classe de test écrite, il reste à la déclarer au framework pour qu'il l'exécute. Nous créons pour cela une classe `AllTests` qui portera le `main()` de l'application. Cette classe comporte également une méthode `suite()` qui construit une `TestSuite` contenant les suites de toutes les classes de tests à intégrer :

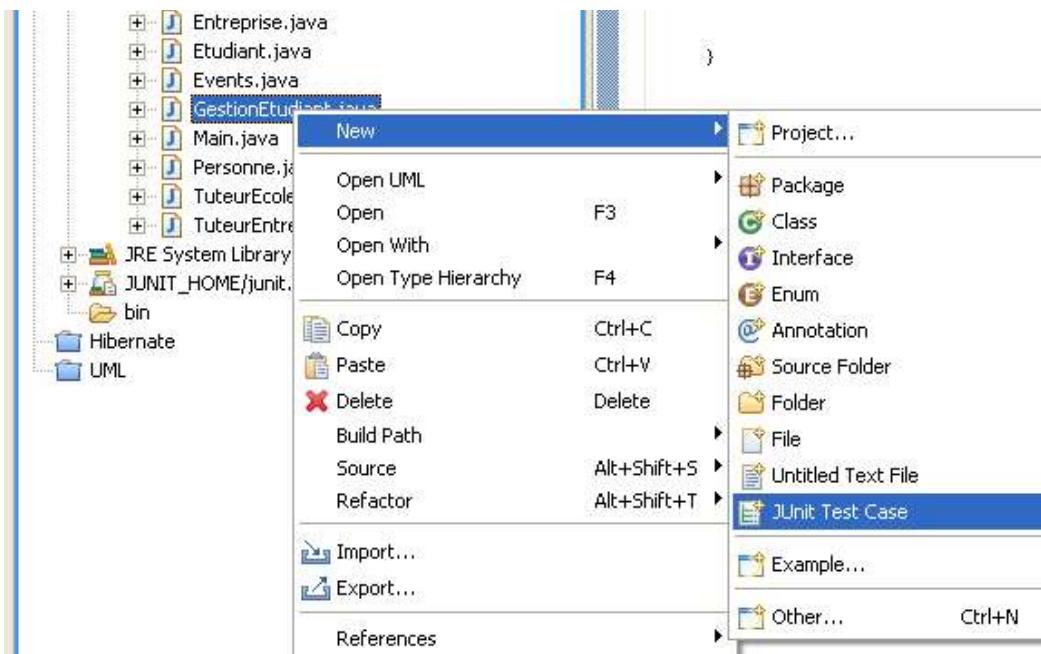
```
public class AllTests {  
    public static Test suite() {  
        TestSuite suite = new TestSuite();  
        suite.addTest(PathTest.suite());  
        suite.addTest(...);  
        ...  
        return suite;  
    }  
    public static void main(String[] args) {  
        junit.textui.TestRunner.run(suite());  
    }  
}
```

Le `main()` utilise un `TestRunner` du package `textui`, qui affiche les résultats des tests sur la sortie standard. Vous pourrez découvrir d'autres types de `TestRunner` dans la documentation de Junit.

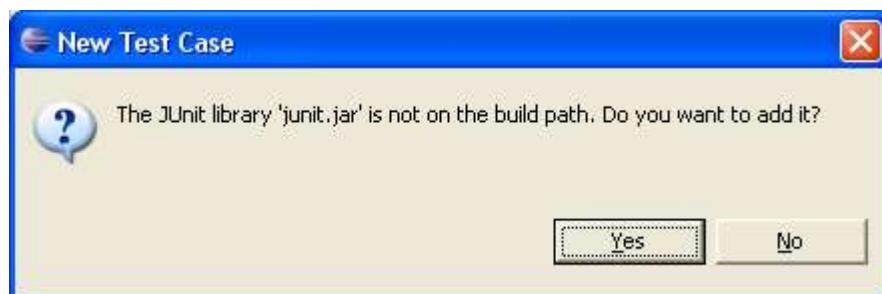
### c. Tests et exemples

L'exemple possède une classe gestion d'étudiant qui permet d'ajouter à un étudiant, un enseignant, etc...

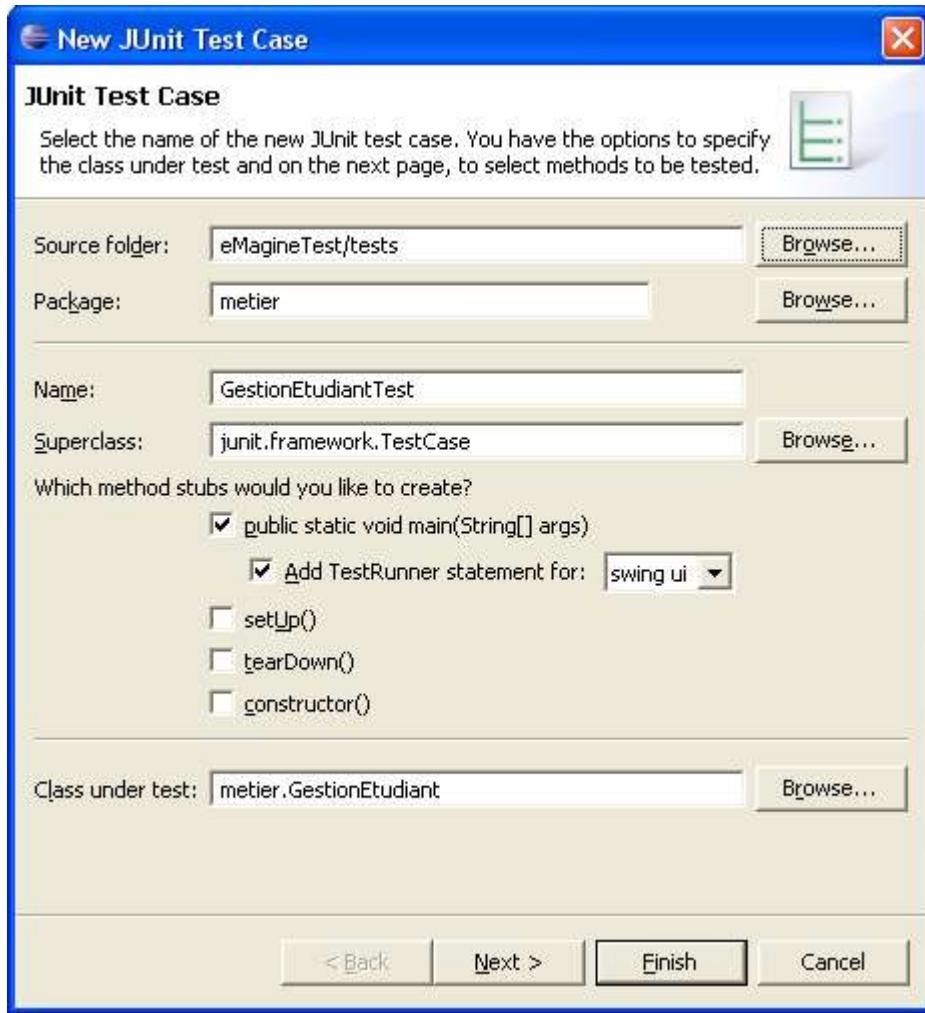
Sélectionner la classe à tester :



L'exemple va se faire avec le plugin fourni par eclipse :



La fenêtre suivante s'ouvre :



**Source folder** : Répertoires où seront stockées les classes de test. Ici un répertoire source spécial a été créé pour stocker les classes de test : « tests »

**Name** : Nom de la classe de test, la classe de test porte par défaut le nom de la classe testé suivie de 'Test', ici la classe testée étant *GestionEtudiant*, la classe test est donc : *GestionEtudiantTest*

**Superclass** : Classe parente pour les tests, toujours *junit.framework.TestCase*

**Case 'public static void main (String[] args)'** : Créer une méthode main, la classe de test peut alors être exécutée en dehors de l'environnement JUnit d'eclipse.

**Case 'Add TestRunner statement for'** : Ajoute le code dans la méthode main pour lancer les tests, les tests peuvent être lancés en trois modes :

- **text ui** : Affichage du résultat des tests sous forme de texte
- **awt ui** : Affichage du résultat des tests sous forme graphique awt
- **swing ui** : Affichage du résultat des tests sous forme graphique swing

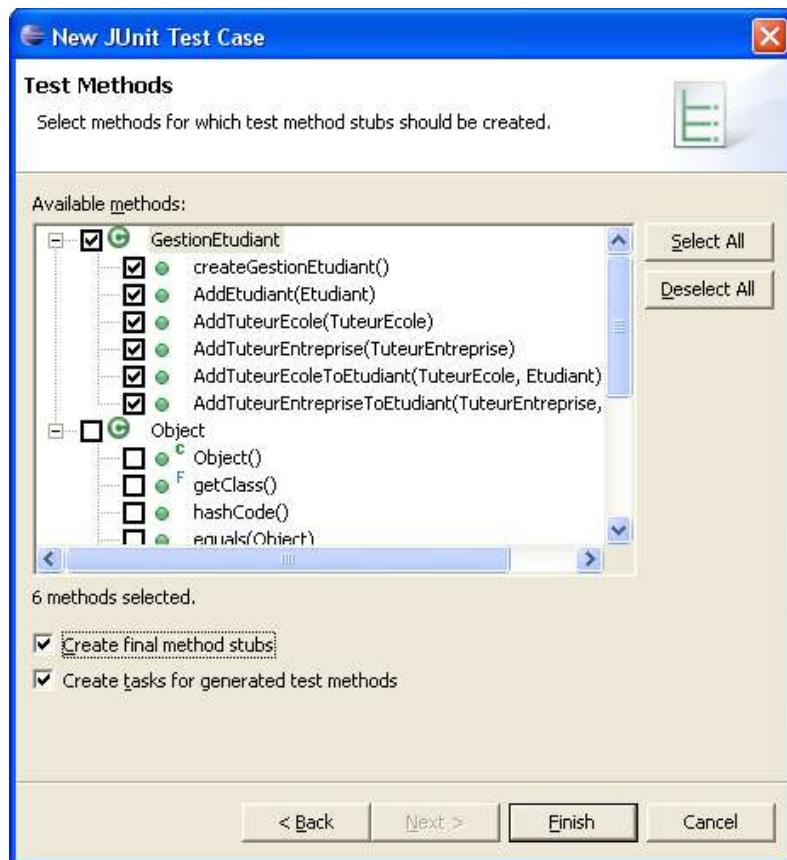
**setUp()** : Créer une méthode setUp (une seule par classe de test) qui est appelée à chaque début de test.

**tearDown()** : Créer une méthode teardown (une seule par classe de test) qui est appelée à chaque fin de test.

**constructor()** : Créer le constructeur de la classe de test, ce constructeur est appelé avant toutes les autres méthodes (tearDown et setUp). Ainsi si il y a trois méthodes de test les appels se feront comme suit :

```
Constructor
Constructor
Constructor
setUp
methode 1
tearDown
setUp
methode 2
tearDown
setUp
methode 3
tearDown
```

cliquer sur 'Next>' la fenêtre suivante apparaît :



**Case 'Create final method stubs'** : met un attribut final à toutes les fonctions à tester

**Case 'Create tasks for generated test methods'** : Rajoute le commentaire 'TODO:' dans le corps de la fonction de test.

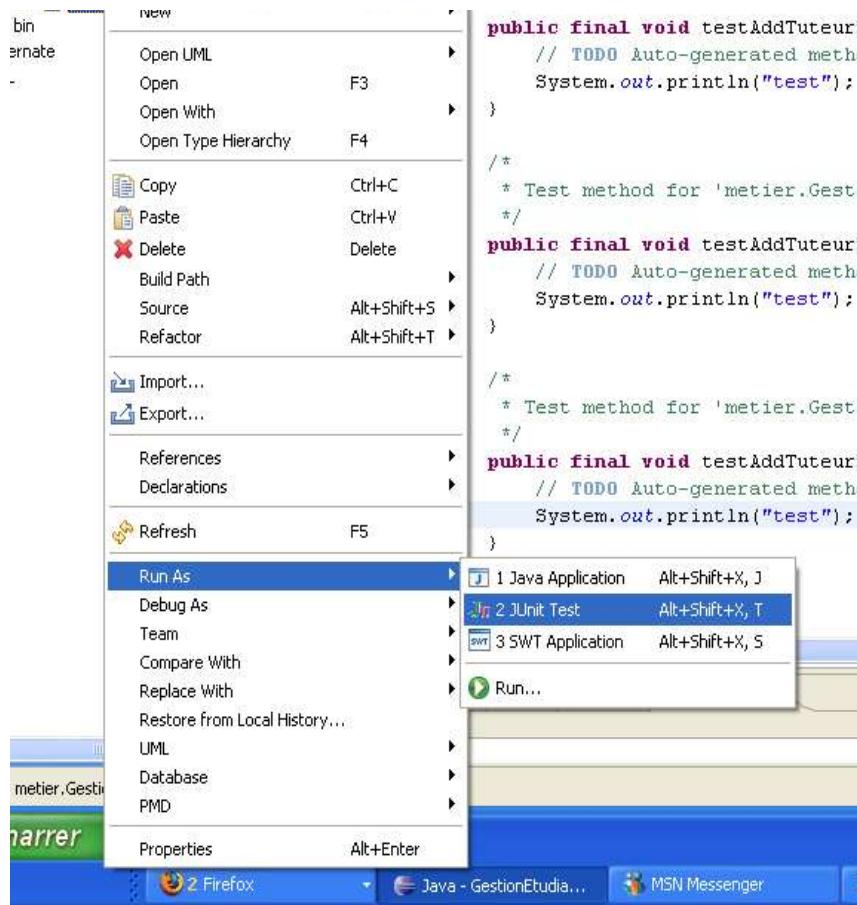
Sélectionner les méthodes à tester, ici toutes les méthodes de la classe GestionEtudiant.

Pour tester une méthode une autre méthode nommée test<nom\_de\_la\_méthode\_> sera créée dans la classe de test et c'est dans cette méthode qu'il faudra écrire le code de test

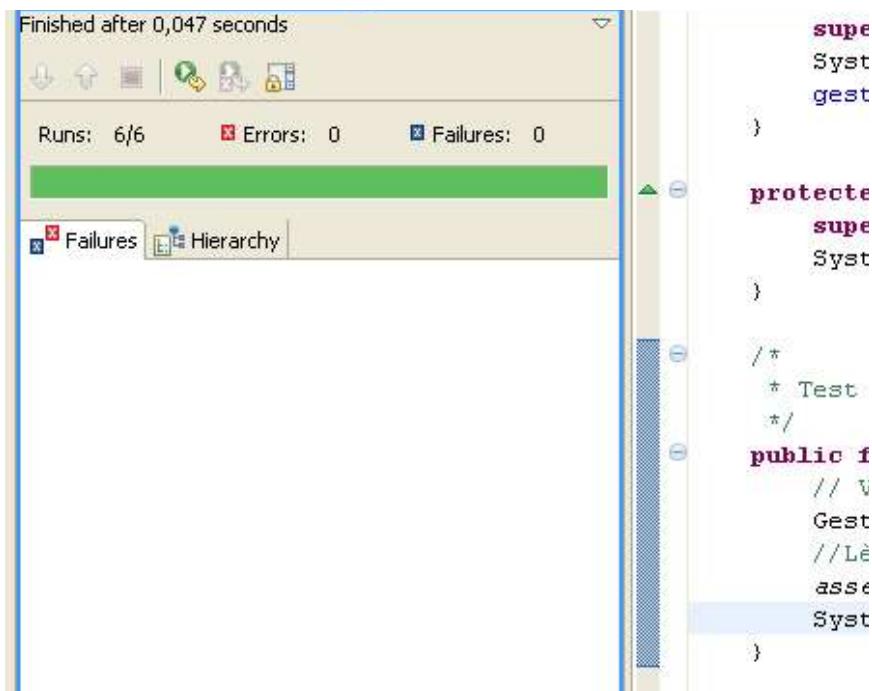
Pour chaque méthode il faut écrire les tests correspondant, par simplicité, nous n'avons tester qu'une méthode :

```
/*
 * Test method for 'metier.GestionEtudiant.createGestionEtudiant()'
 */
public final void testCreateGestionEtudiant() {
    // Vérification que la méthode Create agit bien comme un singleton
    GestionEtudiant gestionEtudiant = GestionEtudiant.createGestionEtudiant();
    //Lève une assertion si les deux classe son différentess
    assertEquals(gestionEtudiant, this.gestionEtudiant);
    System.out.println("testCreateGestionEtudiant");
}
```

l'assertEquals est le test qui va permettre de valider ou non le test.



A présent, sélectionner la classe à tester, faire un clic droit et cliquer sur « Junit Test » :



Si tous les tests sont réussis, alors la barre de progression est verte.

Ci-après le code source de la classe :

```

package metier;

import junit.framework.TestCase;

public class GestionEtudiantTest extends TestCase {

    private GestionEtudiant gestionEtudiant;

    public static void main(String[] args) {
        junit.swingui.TestRunner.run(GestionEtudiantTest.class);
    }

    public GestionEtudiantTest(String name) {
        super(name);
        System.out.println("Constructor");
    }

    protected void setUp() throws Exception {
        super.setUp();
        System.out.println("setUp");
        gestionEtudiant = GestionEtudiant.createGestionEtudiant();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
        System.out.println("tearDown");
    }

    /*
     * Test method for 'metier.GestionEtudiant.createGestionEtudiant()'
     */
    public final void testCreateGestionEtudiant() {
        // Vérification que la méthode Create agit bien comme un singleton
        GestionEtudiant gestionEtudiant =
GestionEtudiant.createGestionEtudiant();
        //Lève une assertion si les deux classe son différentess
        assertNotSame(gestionEtudiant,this.gestionEtudiant);
        System.out.println("testCreateGestionEtudiant");
    }

    /*
     * Test method for 'metier.GestionEtudiant.AddEtudiant(Etudiant)'
     */
    public final void testAddEtudiant() {
        // TODO Auto-generated method stub
        System.out.println("test");
    }

}

```

```

        * Test method for 'metier.GestionEtudiant.AddTuteurEcole(TuteurEcole)'
        */
    public final void testAddTuteurEcole() {
        // TODO Auto-generated method stub
        System.out.println("test");
    }

    /*
     * Test method for 'metier.GestionEtudiant.AddTuteurEntreprise
     (TuteurEntreprise)'
     */
    public final void testAddTuteurEntreprise() {
        // TODO Auto-generated method stub
        System.out.println("test");
    }

    /*
     * Test method for 'metier.GestionEtudiant.AddTuteurEcoleToEtudiant
     (TuteurEcole, Etudiant)'
     */
    public final void testAddTuteurEcoleToEtudiant() {
        // TODO Auto-generated method stub
        System.out.println("test");
    }

    /*
     * Test method for 'metier.GestionEtudiant.AddTuteurEntrepriseToEtudiant
     (TuteurEntreprise, Etudiant)'
     */
    public final void testAddTuteurEntrepriseToEtudiant() {
        // TODO Auto-generated method stub
        System.out.println("test");
    }
}

```

## 2.3. TESTS UNITAIRES D'APPLICATIONS J2EE

### a. Choix de Cactus : Justification

Cactus est un *framework* qui permet de tester de manière quasi-exhaustive une application web conforme aux standards J2EE.

Il a donc été choisi pour tester l'ensemble de notre application, et notamment nos pages *jsp* ou nos objets EJB. De plus, il permet de générer automatiquement certains tests, ce qui raccourcit nettement le temps de développement des tests.

## b. Concept

Cactus fournit plusieurs classes `TestCase` qui étendent le `Testcase` de JUnit et plusieurs types de redirecteurs (`Servlet Redirector`, `JSP Redirector`,...).

Posons `YYYTestCase = ( ServletTestCase | FilterTestCase | JspTestCase )`.

Posons également `XXX`, le nom du cas de test. Chaque classe `YYYTestCase` contient plusieurs cas de test.

### ➤ Etapes d'un test Cactus

Parcourons les étapes successives pour comprendre comment l'ensemble fonctionne. Voici ce qu'il advient pour chaque méthode `testXXX()` dans notre classe de test (dérivée de `YYYTestCase`) :

1. Le Test Runner JUnit appelle une méthode `YYYTestCase.runTest()`. Cette méthode recherche une autre méthode `beginXXX(WebRequest)` et l'exécute si elle en trouve une. Cette méthode est exécutée côté client (pas sur un serveur). Le paramètre `WebRequest` passé à la méthode `beginXXX()` est utilisé pour positionner les en-têtes HTTP, les paramètres HTTP,... qui seront envoyés à l'étape 2 au proxy redirecteur.
2. La méthode `YYYTestCase.runTest()` ouvre alors une connexion HTTP vers le proxy redirecteur. Tous les paramètres initialisés dans la méthode `beginXXX()` sont envoyés dans la requête HTTP (en-têtes HTTP, paramètres HTTP,...)
3. Le proxy redirecteur agit comme proxy côté serveur pour notre classe de test. Cela signifie que notre classe de test est instanciée deux fois : une fois côté client (par le Test Runner JUnit) et une fois côté serveur (par le proxy redirecteur). L'instance côté client est utilisée pour exécuter les méthodes `beginXXX()` et `endXXX()` (voir les étapes 1 et 8) tandis que l'instance côté serveur est utilisée pour exécuter les méthodes `testXXX()` (voir l'étape 4).

Le proxy redirecteur exécute les opérations suivantes:

- il crée une instance de notre classe de test en utilisant la réflexion. Il initialise alors les objets implicites (les variables de classe de `YYYTestCase`). Ces objets implicites dépendent du proxy redirecteur (voir la section suivante).
- il crée des instances des "enveloppes" Cactus (wrappers) pour certains objets serveurs (`HttpServletRequest`, `ServletConfig`, `ServletContext`, ...), ceci afin d'être capable de surcharger certaines méthodes pour renvoyer des valeurs simulées. Par exemple, le framework Cactus peut simuler une URI (c'est-à-dire se comporter comme si cette URI avait été appelée au lieu de l'URI du proxy redirecteur). Donc, les méthodes `getServerName()`, `getServerPort()`, `getRequestURI()`, ... renvoient des valeurs basées sur l'URI simulée (si une telle URI a été définie par l'utilisateur).
- crée une Session HTTP si l'utilisateur en a exprimé le souhait (en utilisant le code `WebRequest.setAutomaticSession(boolean)` de la méthode `beginXXX()`; par

défaut, une session est toujours créée). Le redirecteur remplit par réflexion l'objet implicite `session`.

4. Les méthodes `setUp()`, `testXXX()` et `tearDown()` de notre classe de test sont exécutées (dans cet ordre). Elles sont appelées par le proxy redirecteur en utilisant la réflexion. Les méthodes `setUp()` et `tearDown()` sont bien sûr optionnelles (comme dans JUnit).
5. La méthode `testXXX()` appelle le code de l'application que nous voulons tester côté serveur, exécute le test et utilise l'API `assert` de JUnit pour vérifier les résultats (`assert()`, `assertEquals()`, `fail()`, ...).
6. Si le test échoue, la méthode `testXXX()` lève des exceptions, qui sont interceptées par le proxy redirecteur.
7. Si une exception a été levée, le proxy redirecteur renvoie les informations sur l'exception (son nom, sa classe, la stack trace) au côté client. Ces informations sur l'exception seront alors imprimées par JUnit dans la console de son Test Runner.
8. Si aucune exception n'a été levée, la méthode `YYYTestCase.runTest()` cherche une méthode `endXXX(org.apache.cactus.WebResponse)` ou `endXXX(com.meterware.httpunit.WebResponse)` (cette signature est utilisée par [l'intégration HttpUnit](#)) et l'exécute si elle en trouve une. À ce stade, vous avez la possibilité de vérifier les en-têtes HTTP renvoyées, les Cookies et le flux sortant de la servlet dans la méthode `endXXX()` en utilisant cette fois encore les assertions JUnit et les classes utilitaires fournies par Cactus.

## ➤ Les Proxy Redirecteurs

Cactus fournit trois implémentations de Proxys redirecteurs :

- **Servlet Redirector.** Ce redirecteur est une servlet que l'on peut utiliser pour le test unitaire de méthodes dans une servlet ou dans toute classe java qui utilise les objets de l'API Servlet (`HttpServletRequest`, ...)
- **JSP Redirector.** Ce redirecteur est une page JSP que l'on peut utiliser pour le test unitaire de code serveur qui doit accéder aux objets implicites d'une JSP (`PageContext`, ...). Le redirecteur JSP s'utilise pour le test unitaire de Taglibs.
- **Filter Redirector.** Ce redirecteur est une Servlet Filter qui s'utilise pour le test unitaire de servlets filtres ou de toute classe java qui a besoin d'accéder aux objets d'un filtre (`FilterConfig`, ...).

### **Servlet Redirector Proxy**

Le côté client ouvre deux connexions HTTP vers le redirecteur Servlet. La première pour exécuter les tests et récupérer le flux de sortie de la servlet, l'autre pour récupérer le résultat du test. Ceci permet d'accéder aux données de l'exception (message, stack trace...) si le test échoue. Les résultats du test sont stockés dans une variable qui est mise dans le `ServletContext` et récupérée par la seconde connexion HTTP.

### **JSP Redirector Proxy**

Le côté client ouvre deux connexions HTTP vers le redirecteur JSP. La première pour exécuter les tests et récupérer le flux de sortie de la servlet, l'autre pour récupérer le résultat du test. Ceci permet d'accéder aux données de l'exception (message, stack trace...) si le test échoue. Les résultats du test sont stockés dans une variable qui est mise dans le `ServletContext` et récupérée par la seconde connexion HTTP.

### **Filter Redirector Proxy**

Le côté client ouvre deux connexions HTTP vers le redirecteur Filtre. La première pour exécuter les tests et récupérer le flux de sortie de la servlet, l'autre pour récupérer le résultat du test. Ceci permet d'accéder aux données de l'exception (message, stack trace...) si le test échoue. Les résultats du test sont stockés dans une variable qui est mise dans le `ServletContext` et récupérée par la seconde connexion HTTP.

## **c. Utilisation**

### ➤ Installation de Cactus

Il n'y a pas vraiment d'installation de **Cactus**, puisque **Cactus** n'est pas une application mais un framework. Il faut toutefois comprendre comment il s'intègre dans notre environnement de travail. L'installation de **Cactus** consiste en fait à comprendre l'arborescence des jars ainsi que savoir les fichiers de configuration que nous devrons posséder et éditer. Tout ceci est exposé ci-dessous.

Il y a deux types d'environnement de développement dans lesquels nous pouvons utiliser **Cactus** :

- dans notre IDE (environnement de développement Java), et
- en ligne de commande, en utilisant Ant.

Normalement, ces deux environnements sont complémentaires. Une bonne stratégie est d'utiliser un IDE pour augmenter la productivité au quotidien, d'exécuter rapidement les tests **Cactus** dans l'IDE et d'utiliser Ant pour l'assemblage (en continu), y compris l'exécution des tests **Cactus**.

### ➤ Les fichiers de Cactus

Nous devons d'abord télécharger les fichiers de la distribution de Cactus (pour l'API J2EE que nous projetons d'utiliser), puis de la décompresser dans un répertoire `[cactus root]`.

Les jars qui constituent **Cactus** se trouvent dans `[cactus root]/lib`. Ce sont :

- **cactus.jar**: l'archive principale, qui contient les classes du framework. Ce jar se trouve dans `[cactus root]/lib`.

- **cactus-ant.jar** : un jar qui contient un certain nombre de tâches Ant pour faciliter l'intégration avec Ant. Ce jar n'est nécessaire que pour l'utilisation d'Ant pour automatiser l'exécution de tests **Cactus**. Ce jar aussi se trouve dans [**cactus root**]/lib.
- **httpclient.jar** : **Cactus** utilise le framework Jakarta Commons HttpClient pour la gestion des cookies.
- **junit.jar** : **Cactus** étend JUnit et nécessite donc le jar de JUnit.
- **aspectjrt.jar** : **Cactus** utilise AspectJ pour un certain nombre de tâches (logs à l'entrée et à la sortie des méthodes, vérification de la configuration, etc.).
- **commons-logging.jar** : **Cactus** utilise le framework façade Jakarta Commons Logging pour fournir de manière transparente les fonctionnalités de log en utilisant n'importe lequel des frameworks de log existants (log4j, LogKit, JDK 1.4 Logging, etc). Cette bibliothèque est aussi nécessaire à Commons HttpClient.
- **logging framework(optional)** : Le framework de log à utiliser (log4j, LogKit,...). Une option qui n'est nécessaire que pour les logs internes de **Cactus**. De plus, le framework Commons Logging fournit un logger simple qui écrit à la console.

## ➤ Écrire un cas de test en utilisant Cactus

### **Introduction**

Ce tutoriel explique comment écrire un cas de test en utilisant Cactus. Il y a plusieurs types de cas de tests : des cas de tests pour le test unitaire de servlets, pour le test unitaire de taglibs et pour le test unitaire de filtres. Nous allons couvrir les principes génériques à tous les cas de tests puis nous examinerons en détail les détails spécifiques à chacune de ces catégories.

- Principes généraux
- Détails spécifiques

La distribution de Cactus fournit des exemples d'utilisation qu'il est conseillé de lire. Par ailleurs, toutes les "bonnes pratiques" de JUnit s'appliquent aussi aux cas de test de Cactus, parce que ceux-ci sont aussi, par essence, des cas de test JUnit.

### **Principes généraux**

Pour écrire un cas de test, suivons les étapes décrites ci-dessous.

#### **Etape 1: Imports**

Il faut inclure les imports suivants dans la classe de test (`junit.framework.*` est requis parce que Cactus utilise JUnit comme application cliente pour l'appel des tests):

```
import org.apache.cactus.*;
import junit.framework.*;
```

## ***Etape 2: Etendre un TestCase Cactus ou réutiliser un TestCase JUnit***

### **Option A: Etendre une classe Cactus TestCase**

Il nous faut créer une classe (notre classe de test) qui étend un des cas de test Cactus en fonction de ce que nous désirons tester :

- **ServletTestCase**: hériter de cette classe pour écrire des tests unitaires de code qui utilisent les objets de l'API Servlet (`HttpServletRequest`, `HttpServletResponse`, `HttpSession`, `ServletConfig`, `ServletContext`, ...), tel que Servlets ou toute classe java qui a des méthodes manipulant des objets de l'API Servlet. Par exemple:

```
public class TestSampleServlet extends HttpServlet
{
```

- **JspTestCase**: hériter de cette classe pour écrire des tests unitaires de code qui utilisent les objets de l'API JSP (`PageContext`, `JspWriter`, ...), tel que Taglibs ou toute classe java qui a des méthodes manipulant des objets de l'API JSP. Par exemple:

```
public class TestSampleTag extends JspTestCase
{
```

- **FilterTestCase**: hériter de cette classe pour écrire des tests unitaires de code qui utilisent les objets de l'API Filter (`FilterChain`, `FilterConfig`, `HttpServletRequest`, `HttpServletResponse`, ...), tel que filtres ou toute classe java qui a des méthodes manipulant des objets de l'API Filter. Par exemple:

```
public class TestSampleFilter extends FilterTestCase
{
```

### **Option B: Réutiliser un TestCase JUnit**

Cactus est capable d'exécuter un TestCase "purement JUnit" côté serveur. Ceci se fait en utilisant l'objet Test Suite `ServletTestSuite` qui encapsule nos cas de test existants. Par exemple:

```
public class TestJUnitTestCaseWrapper extends TestCase
{
    public static Test suite()
    {
        ServletTestSuite suite = new ServletTestSuite();
        suite.addTestSuite(TestJUnitTestCaseWrapper.class);
        return suite;
    }

    public void testXXX()
    {
    }
}
```

### **Etape 3: les méthodes standard JUnit**

Comme dans un cas de test JUnit normal, définissons les méthodes standard JUnit comme suit:

- Un constructeur avec un seul paramètre (le nom du test) ;
- (option): une méthode `main()` dans laquelle nous démarrons un test runner JUnit pour que notre test soit exécutable ;
- (option): une méthode `suite()` pour lister les tests qui doivent être exécutés par la classe de test (par défaut, toutes les méthodes dont le nom commence par "test").

Par exemple:

```
public TestSampleServlet(String theName)
{
    super(theName);
}

public static void main(String[] theArgs)
{
    junit.swingui.TestRunner.main(new String[]
{TestSampleServlet.class.getName()});
}

public static Test suite()
{
    return new TestSuite(TestSampleServlet.class);
}
```

### **Etape 4 (optionnelle): méthodes `setUp()` et `tearDown()`**

Comme dans JUnit, nous pouvons définir une méthode `setUp()` et une méthode `tearDown()`.

Ces méthodes sont exécutées respectivement avant et après chaque test. Toutefois, alors que dans JUnit ces méthodes sont exécutées côté client, avec Cactus elles sont exécutées côté serveur. Cela signifie que nous serons capable d'accéder dans ces méthodes à un objet implicite de Cactus (un de ces objets de l'API définis à l'étape 2). En d'autres termes, nous serons capable par exemple de mettre une valeur dans la session HTTP avant d'appeler les cas de test, etc.

Comme dans JUnit, les méthodes `setUp()` et `tearDown()` sont optionnelles.

### **Etape 5 (optionnelle): méthodes `begin()` et `end()`**

Les méthodes `begin(...)` et `end(...)` sont l'équivalent côté client des méthodes `setUp()` et `tearDown()` de l'étape précédente. Elles sont appelées côté client avant et après chaque test.

Les méthodes `begin()` et `end()` sont optionnelles.

## **Etape 6: méthodes testXXX()**

Comme dans JUnit, la méthode principale pour un test est la méthode `testXXX()`. La différence étant que ces méthodes sont exécutées dans le conteneur avec Cactus. Chaque cas de test XXX doit avoir une méthode `testXXX()` définie.

Dans nos méthodes `testXXX()` nous allons :

- instancier la classe à tester (nous pouvons également factoriser une instance en la définissant comme variable d'instance de la classe) ;
- paramétriser tout objet côté serveur dont nous avons besoin (par exemple, mettre une variable dans la session Http). En effet, la classe cas de test de Cactus que nous avons étendue à l'étape 2 a plusieurs variables d'instance (ce sont les différents objets de l'API mentionnés à l'étape 2) qui ont été initialisées à des valeurs valides. Selon la classe cas de test que nous étendons, ces variables sont `request` (de type `HttpServletRequest`), `config` (de type `ServletConfig` pour `ServletTestCase` ou de type `FilterConfig` pour `FilterTestCase`), etc (voir les "Détails spécifiques" ci-dessous) ;
- appeler la méthode à tester ;
- exécuter des méthodes `assertXX` standard de JUnit (`asserts(..)`, `assertEquals(..)`, `fail(..)`,etc) pour vérifier que le test a réussi.

Par exemple:

```
public void testXXX()
{
    // Initialize class to test
    SampleServlet servlet = new SampleServlet();

    // Set a variable in session as the doSomething() method that we are
    testing need
    // this variable to be present in the session (for example)
    session.setAttribute("name", "value");

    // Call the method to test, passing an HttpServletRequest object (for
    example)
    String result = servlet.doSomething(request);

    // Perform verification that test was successful
    assertEquals("something", result);
    assertEquals("otherValue", session.getAttribute("otherName"));
}
```

## **Etape 7 (optionnelle): méthodes beginXXX()**

Pour chaque cas de test XXX, nous pouvons définir une méthode correspondante `beginXXX()` (option). Cette méthode permet par exemple d'initialiser des paramètres relatifs à HTTP (paramètres HTTP, cookies, en-têtes HTTP, URL à simuler, etc). Ces valeurs pourront être récupérées dans notre

`testXXX()` en appelant les API appropriées de `HttpServletRequest` (comme `getQueryString()`, `getCookies()`, `getHeader()`, ...).

La signature de la méthode `begin` est:

```
public void beginXXX(WebRequest theRequest)
{
    [...]
}
```

où `theRequest` est l'objet (fourni par Cactus) à utiliser pour régler tous les paramètres relatifs à HTTP.

La description complète de tous les paramètres relatifs à HTTP qu'il est possible de positionner se trouve dans la javadoc de la classe `WebRequest`. Nous pouvons également jeter un coup d'oeil sur les exemples fournis avec la distribution de Cactus.

Les méthodes `beginXXX()` sont exécutées côté client avant l'exécution de `testXXX()` côté serveur ; elles n'ont donc pas accès aux variables de classe qui représentent des objets de l'API (leur valeur est `null`).

#### **Etape 8 (optionnelle): méthodes `endXXX()`**

Pour chaque cas de test XXX, nous pouvons définir une méthode correspondante `endXXX()`. Cette méthode permet par exemple de vérifier les paramètres HTTP renvoyés par le cas de test (tels que contenu de la réponse HTTP, cookies renvoyés, en-têtes HTTP renvoyés,...).

Pour les versions de Cactus jusqu'à 1.1, la signature de la méthode `end` est:

```
public void endXXX(HttpURLConnection theConnection)
{
    [...]
}
```

... et quelques méthodes utilitaires pour extraire le contenu de la réponse et des cookies étaient fournies dans la classe `AssertUtils` (voir la javadoc).

À partir de Cactus 1.2, cette signature a été dépréciée (`deprecated`). Il y a maintenant 2 signatures possibles pour la méthode `end`, selon que nous souhaitons ou non effectuer des vérifications sophistiquées sur le contenu de ce qui est renvoyé. Pour effectuer des vérifications complexes, il faut utiliser l'intégration de Cactus avec `HttpUnit`.

Les méthodes `endXXX()` sont exécutées côté client après l'exécution de `testXXX()` côté serveur; elles n'ont donc pas accès aux variables de classe qui représentent des objets de l'API (leur valeur est `null`).

#### **d. Liens utiles**

- Tutoriel sur l'implémentation des proxy redirecteurs :

[http://web.archive.org/web/20040326102326/http://www.ressources-java.net/cactus/writing/howto\\_TestCase.html](http://web.archive.org/web/20040326102326/http://www.ressources-java.net/cactus/writing/howto_TestCase.html)

- Tutoriel sur l'intégration de **Cactus** avec Ant :

<http://cactus.ressources-java.net/integration/ant/index.jsp>

- Tutoriel sur les signatures des méthodes end :

[http://web.archive.org/web/20041109120105/http://www.ressources-java.net/cactus/writing/howto\\_Htpunit.jsp](http://web.archive.org/web/20041109120105/http://www.ressources-java.net/cactus/writing/howto_Htpunit.jsp)

## 2.4. ENVIRONNEMENT DE DÉVELOPPEMENT J2EE

### a. Choix de Struts : Justification

Apache Struts est un cadre d'applications open-source pour développer (*framework*) des applications web J2EE. Il utilise et étend l'API Servlet Java afin d'encourager les développeurs à adopter l'architecture MVC.

Nous avons choisi cet environnement car tout d'abord il est basé sur un modèle MVC. Grâce à ce concept de programmation, la réalisation de l'application peut se diviser en plusieurs tâches bien spécifiques. Par exemple, un graphiste pourra s'occuper exclusivement de créer les *jsp*, « vues » de l'application.

Struts apporte également tout une bibliothèque de balises qui simplifie énormément la création de pages *jsp* et qui permet de ne pas mélanger du langage Java avec des balises HTML.

Enfin, il permet d'organiser l'application web en *workflows* c'est à dire en cheminements de pages contrôlées et prévues.

### b. Utilisation

#### ➤ Installation

##### **Librairie JAVA**

Dans un premier temps, il est nécessaire d'installer la librairie JAVA. Celle-ci peut être trouvée sur le site de SUN qui lui est dédié : <http://java.sun.com/>. La version qu'il faut télécharger est la dernière JDSE disponible (la 5.0 au jour de rédaction de ce document).

Dans un système UNIX, il faut décompresser le fichier *.bin* fourni sur le site dans le répertoire **/usr/local/lib/** si possible. Ensuite, de manière à faciliter l'utilisation de cette bibliothèque, il faut rajouter le chemin **/usr/local/lib/[jdk\_installée]/bin/** dans la variable d'environnement **PATH** de l'utilisateur.

Sur Windows, l'installation se fait à l'aide d'un installateur, et les propositions par défaut pour l'installation sont correctes.

## Un serveur d'application

Dans un deuxième temps, nous devons installer un serveur d'application. Le choix de l'équipe eMagine s'est porté sur Tomcat, qui est un très bon serveur d'application java open source. Voir la section qui lui est consacrée.

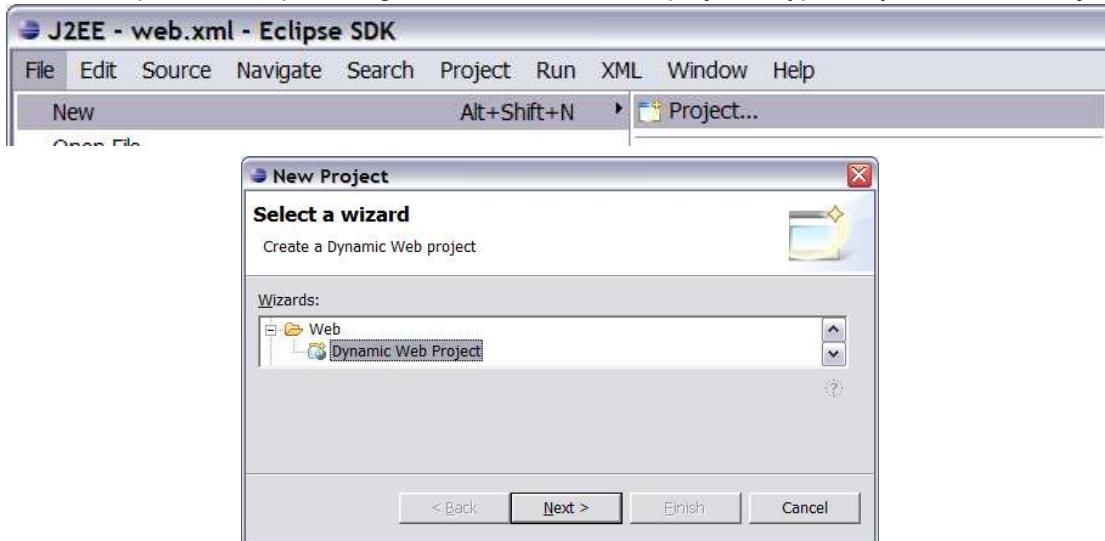
## La librairie Struts

Struts, tout comme Tomcat, est développé par la communauté Apache, et le projet est accessible à l'adresse <http://struts.apache.org/>. Pour l'installer, il faut télécharger la version binaire (menu Download>Binaries). Deux versions peuvent être téléchargées :

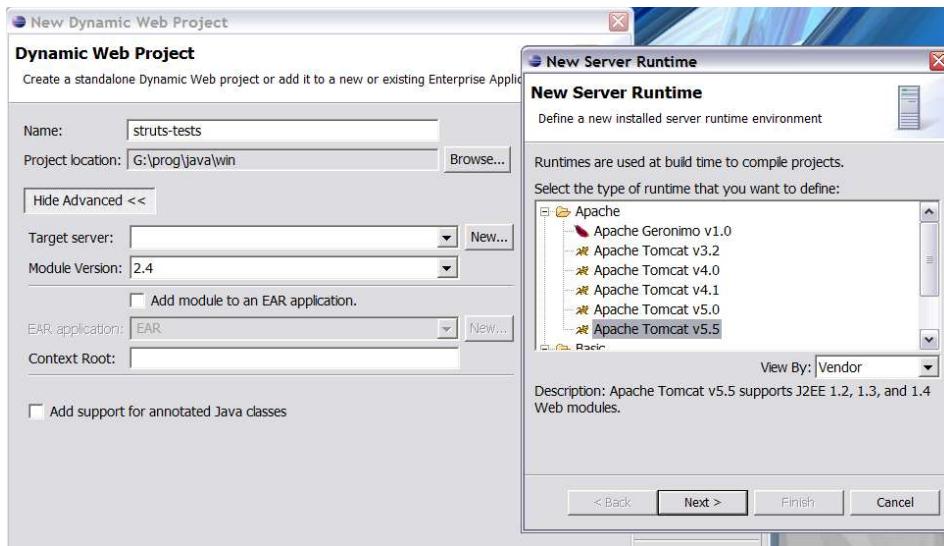
- La version de librairie seule (« Library »). Celle-ci est livrée sous forme d'un ensemble de *.jar* de *.tld* et d'autres fichiers, à placer dans le répertoire **WEB-INF/lib/** de l'application web que l'on veut construire ;
- La version binaire (« binaries »). Cette version contient en plus de la librairie de base, un ensemble d'exemples d'utilisation de la librairie, présentés sous forme de « webapps » (fichiers en *.war* situés dans le répertoire *webapp* de l'archive). Pour les tester, il suffit de placer ces fichiers dans le répertoire **/webapps/** du serveur d'application. Les fichiers seront déployés automatiquement si le serveur est lancé, et pourront être accessibles.

## ➤ Création d'une application web à l'aide d'Eclipse

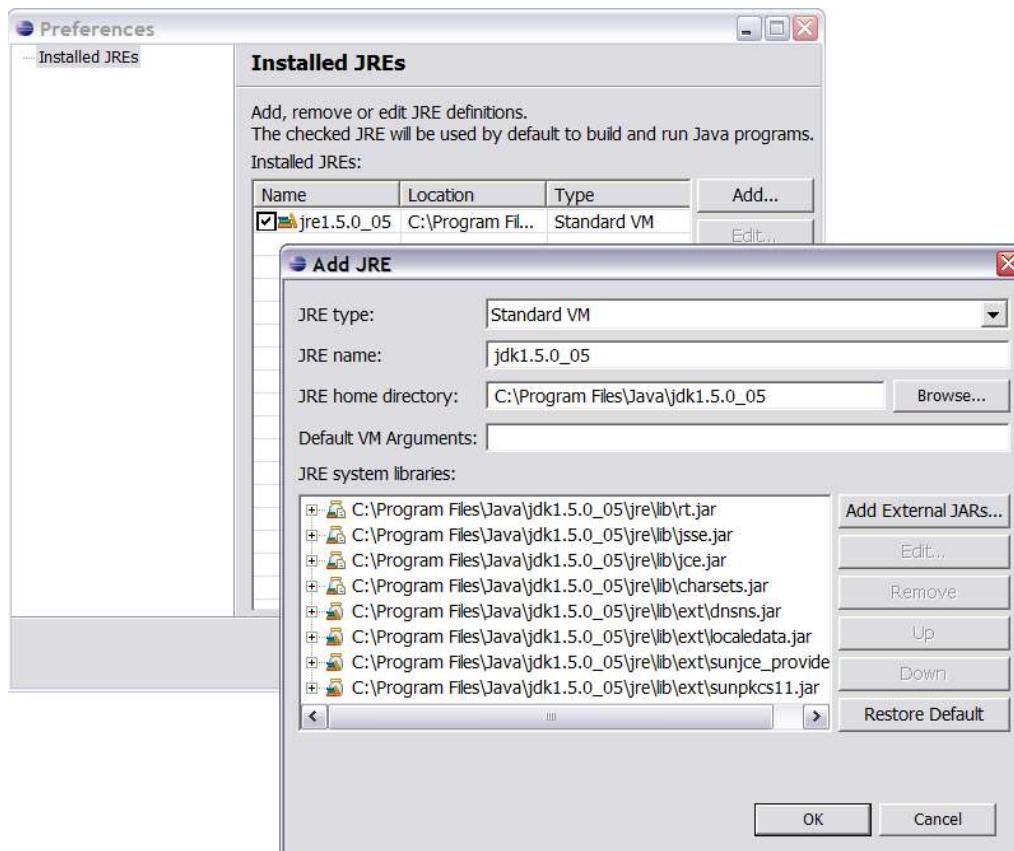
Dans un premier temps, il s'agit de créer un nouveau projet de type « Dynamic Web Project ».



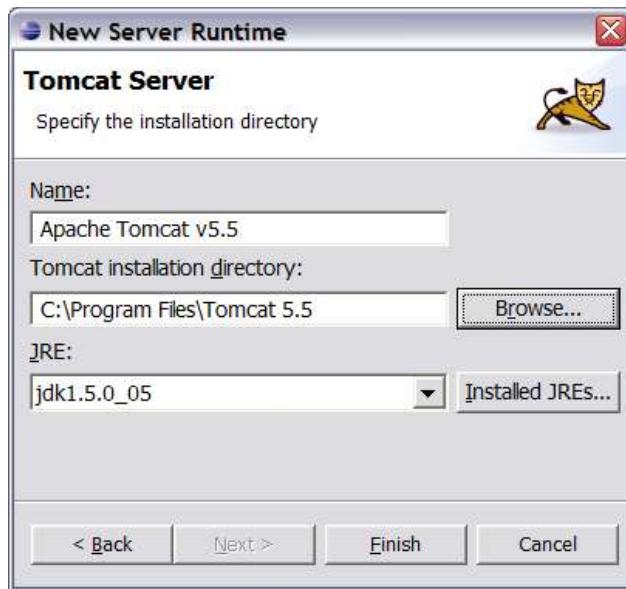
Ensuite, nous renseignons le nom du projet (ici « struts-tests ») ainsi que le « Module Version » à 2.3. Puis, après un clique sur « Show Advanced », nous allons sélectionner Tomcat comme serveur cible (« Target Server »). Comme il n'existe pas encore dans les types de serveurs cibles, nous allons le créer (« New »). Nous sélectionnons alors « Apache Tomcat v5.5 » dans la liste (ou un autre, ceci dépend du serveur installé sur la machine).



Il faut alors saisir le dossier où nous avons installé Tomcat, puis la version de Java qui va l'exécuter. Attention, il faut que cette version soit une JDK et non une JRE, car Tomcat doit compiler les jsp. Comme la jdk n'est pas référencée de base, nous allons la rajouter. Pour cela, cliquer sur « Installed JREs ». Cliquer alors sur « Add », puis renseigner le chemin de la JDK. Appelons la « jdk1.5.0\_05 » (puisque telle est sa version).

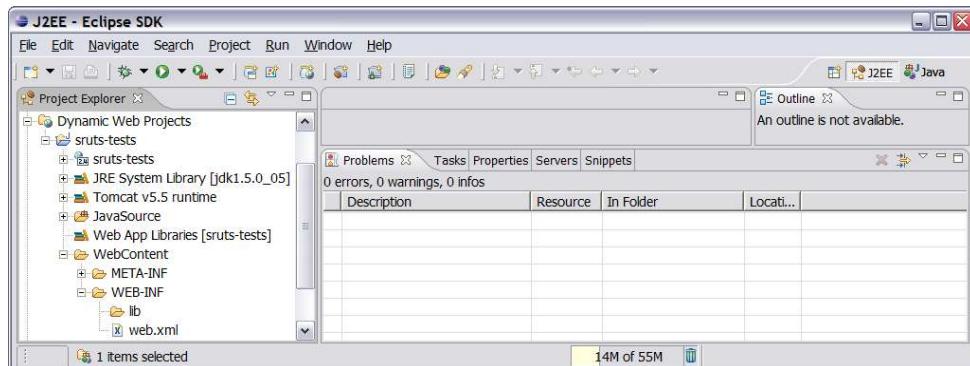


Après validation, voici l'écran que nous obtenons :



Nous pouvons alors terminer la création du projet (en cochant « Add support for annotated Java classes » qui servira pour Hibernate par exemple).

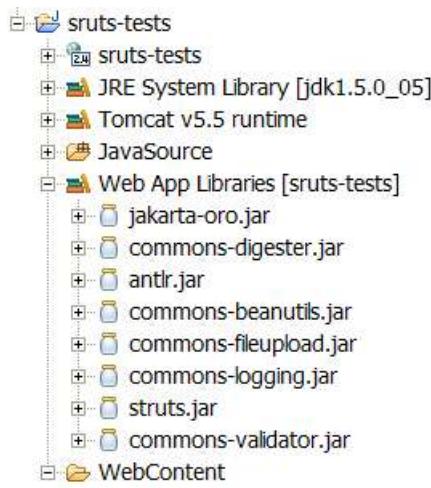
Après la validation d'une licence de sun, nous nous retrouvons avec le projet créé :



### Installation de Struts

Comme défini dans la partie de l'installation de la librairie Struts, il faut copier tous les fichiers *.jar* contenus dans la distribution *struts-[version]-lib* dans le répertoire **WEB-INF/lib/**. Les fichiers *.tld* constituent la définition des balises que nous allons pouvoir utiliser grâce à Struts et doivent être placés (avec tous les fichiers *.xml*) dans le répertoire **WEB-INF/**.

Après la copie, les librairies Struts ont dû être reconnues et incluses dans « Web App Libraries » :



### Fichiers de configuration

Ensuite, il faut définir le fichier de description de notre application. Il s'agit du fichier **WEB-INF/web.xml** dont voici le contenu dans le cadre de notre application de test :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="WebApp_ID">
    <display-name>struts-tests</display-name>
    <!-- Struts Action Servlet -->
    <servlet>
        <servlet-name>action</servlet-name>
        <servlet-class>
            org.apache.struts.action.ActionServlet
        </servlet-class>
        <load-on-startup>2</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>default.html</welcome-file>
        <welcome-file>default.htm</welcome-file>
        <welcome-file>default.jsp</welcome-file>
    </welcome-file-list>

    <!-- Struts Tag Library Descriptors -->
    <taglib>
```

```

<taglib-uri>/tags/struts-bean</taglib-uri>
<taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>
<taglib>
    <taglib-uri>/tags/struts-html</taglib-uri>
    <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>
</web-app>

```

Ce fichier est lui aussi standardisé et suit la norme J2EE d'une application web.

Il faut enfin ajouter la configuration propre à Struts. Ce fichier, qui se nomme

**WEB-INF/struts-config.xml**, définit les articulations de l'application. Struts implémente le modèle MVC (*Modèle-Vue-Contrôleur*) et ce fichier peut être apparenté au **contrôleur**. Voici le contenu de ce fichier pour notre application :

```

<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
    "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd" >
<struts-config>
    <!-- Form Bean Definitions -->
    <form-beans>
        <form-bean name="firstNameBean"
            type="fr.umlvi.r3.strutstests.FirstNameBean" />
    </form-beans>
    <!-- Action Mapping Definitions -->
    <action-mappings>
        <action path="/hello" type="fr.umlvi.r3.strutstests.HelloAction"
            input="/index.jsp" name="firstNameBean" scope="request">
            <forward name="success" path="/hello.jsp" />
            <forward name="failure" path="/error.jsp" />
        </action>
    </action-mappings>
</struts-config>

```

### Fichiers sources et logique Struts

Nous allons maintenant écrire deux fichiers .jsp : le fichier d'accueil du site (*index.jsp*) et le fichier qui traitera la demande « hello » (*hello.jsp*).

Le fichier d'accueil est celui appelé par défaut quand un utilisateur envoie la requête du contexte à vide. Voici le contenu de ce fichier d'exemple :

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/tags/struts-html" prefix="html"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>

```

```

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Test de Struts</title>
</head>
<body>
<h1>Test de Struts</h1>
<html:form action="hello.do" method="GET">
    Votre prénom : <html:text name="firstNameBean" property="firstName" />
    <html:submit value="Say hi!" />
</html:form>
</body>
</html>

```

Dans ce fichier nous pouvons remarquer la présence de balises typiques de struts : `<html:form>`, `<html:text>` ou encore `<html:submit>`. Ces balises servent respectivement à générer un formulaire HTML, à générer un champ de texte dont la valeur va être enregistrée dans un *bean* et enfin, à générer un bouton de soumission de formulaire. Voici le résultat de l'appel :



**Illustration 2 :** Page index.jsp

Lors de la validation du formulaire, d'après le *mapping* défini dans le *struts-config.xml* la requête est accepté (car venant de *index.jsp*) et envoyé au fichier à la **classe d'action java** *HelloAction*. En terme de MVC, il s'agit du **modèle**. Voici le fichier *Webapp/HelloAction.java* :

```

package fr.umlvi.r3.strutstests;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class HelloAction extends Action {

    @Override

```

```

    public ActionForward execute(ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response) throws Exception {
        FirstNameBean firstNameBean = (FirstNameBean) form;
        firstNameBean.setProcessedFirstName(firstNameBean.getFirstName() +
", c'est donc ton prénom !");
        return mapping.findForward("success");
    }
}

```

Nous constatons dans ce fichier Java que les **actions** de Struts permettent de récupérer les formulaires sous forme de *bean*. Celui-ci est classique, avec deux propriétés : *firstName* et *processedFirstName*.

```

package fr.umlv.ir3.strutstests;

import java.io.Serializable;

import org.apache.struts.action.ActionForm;

public class FirstNameBean extends ActionForm implements Serializable {

    private String firstName;
    private String processedFirstName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getProcessedFirstName() {
        return processedFirstName;
    }

    public void setProcessedFirstName(String processedFirstName) {
        this.processedFirstName = processedFirstName;
    }

}

```

Dans l'action, nous demandons de faire suivre (**mapping.findForward()**) la requête à « success » qui est défini dans le *struts-config.xml* comme étant le fichier */hello.jsp* qui est donc dans notre arborescence *Webapp/hello.jsp* :

```

<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<html>
    <head>
        <title>Hello</title>
    </head>
    <body>

```

```

<h1>Hello</h1>

<p>Hello, <bean:write name="firstNameBean"
property="processedFirstName" />!</p>

</body>
</html>

```

Nous pouvons remarquer dans ce dernier fichier la balise **<bean:write/>** qui permet d'afficher la valeur d'une propriété d'un *bean* (la valeur de *processedFirstName* du *bean firstNameBean* ici).

Voici le résultat de l'appel avec pour prénom « Toto » :



### c. Liens utiles

Les articles suivant sont intéressants :

- Le développement J2EE : [http://perso.wanadoo.fr/jm.doudoux/java/dejae/chap016.htm#chap\\_16](http://perso.wanadoo.fr/jm.doudoux/java/dejae/chap016.htm#chap_16)
- Struts et Eclipse : [http://perso.wanadoo.fr/jm.doudoux/java/dejae/chap019.htm#chap\\_19](http://perso.wanadoo.fr/jm.doudoux/java/dejae/chap019.htm#chap_19)

## 2.5. SERVEUR D'APPLICATION

### a. Choix de Tomcat : Justification

Tomcat est un serveur d'application libre et gratuit (*open source*) qui agit comme un conteneur de servlet J2EE. Tomcat implémente les spécifications des servlets et des JSP de Sun Microsystems. Comme Tomcat inclut un serveur HTTP interne, il est aussi considéré comme un serveur HTTP.

Nous avons opté pour ce serveur car il est gratuit, open source et performant.

### b. Utilisation

#### ➤ Installation

Dans un premier temps, il faut installer la librairie Java (cf. Environnement de développement J2EE / Utilisation / Installation / Librairie Java).

Puis il faut télécharger Tomcat. Ce serveur d'application est réalisé par l'équipe d'Apache et le site qui lui est dédié se trouve à l'adresse <http://tomcat.apache.org/>. Actuellement, la version distribuée se décompose en plusieurs parties. La partie qu'il faut télécharger est le « core » du serveur.

Le fichier compressé est à décompresser dans un endroit accessible en lecture **et en écriture**. Sous UNIX le répertoire **/home/[utilisateur\_courant]/** répond à ce besoin. Sous Windows, le « core » est distribué avec un installateur, et les options par défaut conviennent. Il faut simplement renseigner l'identifiant et le mot de passe de l'administrateur du serveur.

#### ➤ Intégration dans éclipse

Un plugin de gestion du serveur Tomcat pour Eclipse existe. Il a été développé par la société Sysdeo. Il est libre et gratuit et peut être téléchargé à l'adresse <http://www.sysdeo.com/eclipse/tomcatpluginfr>.

Il faut ensuite décompresser l'archive téléchargée dans le répertoire *plugins* du répertoire d'Eclipse.

Après avoir lancé Eclipse, un petit panel pour contrôler Tomcat devrait apparaître.



Enfin il faut indiquer le répertoire de Tomcat : Window -> Preferences -> Tomcat.

Ce plugin lance Tomcat en utilisant le JRE par défaut d'Eclipse.

Pour modifier ce JRE utiliser Window -> Preferences -> Java -> Installed JREs.

Ce JRE doit absolument être un JDK (C'est un prérequis pour Tomcat qui utilise javac pour compiler les JSP).

Une fois ce plugin installé, pour lancer Tomcat, il suffit de cliquer sur le bouton représentant le chat de Tomcat (bouton de gauche). Il est aussi possible de l'arrêter (bouton du milieu) ou encore de le redémarrer (bouton de droite).

## ➤ Environnement de Tomcat

Tomcat est conforme à la norme J2EE des serveurs d'application. Son arborescence est donc :

- *bin* : Scripts et exécutables pour différentes tâches : démarrage (startup), arrêt ...etc
- *common* : Classes communes que Catalina et les applications web utilisent
- *conf* : Fichiers de configuration au format XML et les DTD que ces fichiers XML utilisent
- *logs* : Journaux des applications web et de Catalina
- *server* - Classes utilisées seulement par Catalina
- *shared* : Classes partagées par toutes les applications web
- *webapps* : Répertoire contenant les applications web
- *work* : Fichiers et répertoires temporaires

## 2.6. MISE À JOUR DYNAMIQUE D'UNE PAGE INTERNET

### a. Choix d'AJAX : Justification

AJAX est un terme qui désigne deux fonctionnalités de JavaScript qui existent depuis plusieurs années, mais sont restées inaperçues de nombreux développeurs Web jusqu'il y a peu. Depuis, elles ont été redécouvertes, et des applications comme Gmail, Google suggest et Google Maps sont devenues monnaie courante.

Ces deux fonctionnalités de JavaScript sont les possibilités de :

- Faire des requêtes vers le serveur sans avoir à recharger la page,
- Parcourir et travailler avec des documents XML.

Nous avons choisi d'utiliser cette méthode de programmation pour pouvoir faire des saisies de formulaires imposants, tout en ayant un dynamisme interne aux formulaires sans avoir à recharger toute la page. Nous pouvons ainsi nous passer de développer un « client lourd » pour traiter ces saisies, et garder tout l'environnement objet que l'on va créer pour l'utiliser dans ces formulaires. En effet, ces formulaires sont utilisés pour faire des saisies de masse de candidats par exemple, et un temps de réponse immédiat est nécessaire ; un rechargeement de page n'étant donc pas envisageable.

D'autre part, nous utiliserons AJAX pour le développement du module de notification, qui doit lui aussi être dynamique sans recharger la page entière.

## b. Utilisation

### ➤ Exemple basique

#### Étape 1 - Lancement d'une requête HTTP

Afin de faire une requête HTTP vers le serveur à l'aide de JavaScript, il faut disposer d'une instance d'une classe fournissant cette fonctionnalité. Une telle classe a d'abord été introduite dans Internet Explorer sous la forme d'un objet ActiveX appelé XMLHTTP. Par la suite, Mozilla, Safari et d'autres navigateurs ont suivi en implémentant une classe XMLHttpRequest qui fournit les mêmes méthodes et propriétés que l'objet ActiveX original de Microsoft.

Par conséquent, pour créer une instance de la classe (un objet) fonctionnant sur plusieurs navigateurs, nous pouvons utiliser :

```
if (window.XMLHttpRequest) { // Mozilla, Safari, ...
    http_request = new XMLHttpRequest();
} else if (window.ActiveXObject) { // IE
    http_request = new ActiveXObject("Microsoft.XMLHTTP");
}
```

(À des fins d'illustration, le code ci-dessus est une version un peu simplifiée de ce qui est utilisé pour créer une instance XMLHttpRequest. Pour un exemple plus concret, voir la troisième étape)

Certaines versions de certains navigateurs Mozilla ne fonctionneront pas correctement si la réponse du serveur n'a pas un en-tête de type mime XML. Pour les satisfaire, nous pouvons utiliser un appel de fonction supplémentaire pour écraser l'en-tête envoyé par le serveur, juste au cas où il ne s'agit pas de text/xml.

```
http_request = new XMLHttpRequest();
http_request.overrideMimeType('text/xml');
```

La chose suivante à faire est de décider ce que nous voulons faire après avoir reçu la réponse du serveur. À ce stade, il faut juste dire à l'objet de requête HTTP quelle fonction JavaScript devra effectuer le travail d'analyse de la réponse. Cela se réalise en assignant à la propriété onreadystatechange de l'objet le nom de la fonction JavaScript que nous envisageons d'utiliser, comme ceci :

```
http_request.onreadystatechange = nomDeLaFonction;
```

Notons qu'il n'y a pas de parenthèses après le nom de la fonction, ni de paramètres fournis. Par ailleurs, au lieu de donner un nom de fonction, nous pouvons également utiliser la technique JavaScript de définition de fonctions au vol, et spécifier directement les actions à effectuer sur la réponse, comme ceci :

```
http_request.onreadystatechange = function(){
    // instructions de traitement de la réponse
};
```

Ensuite, après avoir déclaré ce qui se passera lorsque la réponse sera reçue, il s'agit de lancer effectivement la requête. Il faut pour cela appeler les méthodes `open()` et `send()` de la classe de requête HTTP, comme ceci :

```
http_request.open('GET', 'http://www.example.org/some.file', true);
http_request.send(null);
```

- Le premier paramètre de l'appel à `open()` est la méthode de requête HTTP – GET, POST, HEAD ou toute autre méthode que nous voulons utiliser est gérée par le serveur. Nous laissons le nom de la méthode en majuscules comme spécifié par la norme HTTP ; autrement certains navigateurs (comme Firefox) peuvent ne pas procéder à la requête. Pour plus d'informations sur les méthodes de requêtes HTTP possibles, il est possible de consulter les spécifications du W3C à l'adresse <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>.
- Le second paramètre est l'URL de la page dont nous faisons la requête. Pour des raisons de sécurité, il n'est pas possible d'appeler des pages se situant sur un autre domaine. Il faut donc veiller à utiliser le nom de domaine exact sur toutes nos pages ou nous obtiendrons une erreur 'permission denied' à l'appel d'`open()`. Une erreur courante est de charger le site via domaine.tld, mais d'essayer d'appeler des pages avec www.domain.tld.
- Le troisième paramètre précise si la requête est asynchrone. Si mis à TRUE, l'exécution de la fonction JavaScript se poursuivra en attendant l'arrivée de la réponse du serveur. C'est le A d'AJAX.

Le paramètre de la méthode `send()` peut être n'importe quelle donnée que nous voulons envoyer au serveur en cas d'utilisation de la méthode POST. Les données doivent être sous la forme d'une chaîne de requête, comme :

```
nom=valeur&autrenom=autreValeur&ainsi=desuite
```

## Étape 2 - Gestion de la réponse du serveur

Rappelons que lors de l'envoi de la requête, nous avions fourni le nom d'une fonction JavaScript conçue pour traiter la réponse.

```
http_request.onreadystatechange = nomDeLaFonction;
```

Voyons maintenant ce que cette fonction doit faire.

Tout d'abord, elle doit vérifier l'état de la requête. Si cet état a une valeur de 4, cela signifie que la réponse du serveur a été reçue dans son intégralité et qu'elle peut maintenant être traitée.

```
if (http_request.readyState == 4) {  
    // tout va bien, la réponse a été reçue  
} else {  
    // pas encore prête  
}
```

Voici la liste complète des valeurs de `readyState` :

- 0 (non initialisée)
- 1 (en cours de chargement)
- 2 (chargée)
- 3 (en cours d'interaction)
- 4 (terminée)

(d'après [http://msdn.microsoft.com/workshop/author/dhtml/reference/properties/readystate\\_1.asp](http://msdn.microsoft.com/workshop/author/dhtml/reference/properties/readystate_1.asp))

La seconde chose à vérifier est le code d'état HTTP de la réponse du serveur. Tous les codes possibles sont listés sur le site du W3C (<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>). Dans notre cas, nous sommes seulement intéressés par une réponse 200 OK.

```
if (http_request.status == 200) {  
    // parfait !  
} else {  
    // il y a eu un problème avec la requête,  
    // par exemple la réponse peut être un code 404 (Non trouvée)  
    // ou 500 (Erreur interne du serveur)  
}
```

Maintenant que nous avons vérifié l'état de la requête et le code d'état HTTP de la réponse, nous pouvons faire ce que nous voulons des données envoyées par le serveur. Il existe deux manières d'accéder à ces données :

- `http_request.responseText` – renvoie la réponse du serveur sous la forme d'une chaîne de texte
- `http_request.responseXML` – renvoie la réponse sous la forme d'un objet `XMLDocument` que nous pouvons parcourir à l'aide des fonctions DOM de JavaScript

### Étape 3 - Un exemple simple

Mettons tout cela ensemble et effectuons une requête HTTP simple. Notre JavaScript demandera un document HTML, `test.html`, qui contiendra le texte « Je suis un test. », et nous afficherons le contenu de ce fichier `test.html` dans un message `alert()`.

```

<script type="text/javascript" language="javascript">

    var http_request = false;

    function makeRequest(url) {

        http_request = false;

        if (window.XMLHttpRequest) { // Mozilla, Safari, ...
            http_request = new XMLHttpRequest();
            if (http_request.overrideMimeType) {
                http_request.overrideMimeType('text/xml');
            }
        } else if (window.ActiveXObject) { // IE
            try {
                http_request = new ActiveXObject("Msxml2.XMLHTTP");
            } catch (e) {
                try {
                    http_request = new ActiveXObject("Microsoft.XMLHTTP");
                } catch (e) {}
            }
        }

        if (!http_request) {
            alert('Abandon : Impossible de créer une instance XMLHTTP');
            return false;
        }
        http_request.onreadystatechange = alertContents;
        http_request.open('GET', url, true);
        http_request.send(null);

    }

    function alertContents() {

        if (http_request.readyState == 4) {
            if (http_request.status == 200) {
                alert(http_request.responseText);
            } else {
                alert('Un problème est survenu avec la requête.');
            }
        }

    }
</script>
<span
    style="cursor: pointer; text-decoration: underline"
    onclick="makeRequest('test.html')">
    Effectuer une requête

```

```
</span>
```

Dans cet exemple :

- L'utilisateur clique sur le lien « Effectuer une requête » dans le navigateur ;
- ceci appelle la fonction `makeRequest()` avec un paramètre – le nom `test.html` d'un fichier HTML situé dans le même répertoire ;
- la requête est faite et ensuite (`onreadystatechange`) l'exécution est passée à `alertContents()` ;
- `alertContents()` vérifie si la réponse a été reçue et porte un code OK et affiche ensuite le contenu du fichier `test.html` dans un message `alert()`.

#### Étape 4 - Travailler avec des réponses XML

Dans l'exemple précédent, après que la réponse à la requête HTTP ait été reçue, nous avons utilisé la propriété `reponseText` de l'objet de requête, et celle-ci renvoyait le contenu du fichier `test.html`. Essayons maintenant la propriété `responseXML`.

Tout d'abord, créons un document XML valide qui sera l'objet de la requête. Le document (`test.xml`) contient ce qui suit :

```
<?xml version="1.0" ?>
<root>
    Je suis un test.
</root>
```

Dans le script, il est juste nécessaire de remplacer la ligne de requête par :

```
...
onclick="makeRequest('test.xml')"
..."
```

Ensuite, dans `alertContents()`, il faut remplacer la ligne affichant un message `alert(responseText)` par :

```
var xmldoc = http_request.responseXML;
var root_node = xmldoc.getElementsByTagName('root').item(0);
alert(root_node.firstChild.data);
```

De cette façon, nous avons pris l'objet `XMLDocument` donné par `responseXML` et nous avons utilisé des méthodes DOM pour accéder à certaines données contenues dans le document XML.

#### ➤ Exemple avec les *jsp*

Plusieurs librairies existent pour utiliser des balises *jsp* qui vont générer le code nécessaire au fonctionnement de la technique AJAX. La première que l'on peut citer est : AjaxTags (<http://ajaxtags.sourceforge.net/index.html>).

Cette librairie de *tags* additionnels permet notamment de :

- Créer des champs de saisie qui supportent l'auto complétion,
- Afficher des petites boites d'information (qui permettent par exemple de définir un terme d'un texte),
- Remplir des fenêtres grâce à des données demandées de manière asynchrone par l'intermédiaire d'un simple clique chez le client,
- Fabriquer des petites fenêtres avec les propriétés de miniaturisation, de réapparition et de fermeture complète,
- De mettre à jour un formulaire en interrogeant le serveur de manière asynchrone suivant les actions de l'utilisateur.

(Voir les démos de l'ensemble des fonctionnalités à l'adresse <http://ajaxtags.no-ip.info/>).

Une autre librairie de *tags* spécialement développée pour le *framework* Struts permet de faire l'auto complétion d'un champ : Struts-Layout (<http://struts.application-servers.com/>).

## 2.7. BIBLIOTHÈQUE DE GÉNÉRATION DE GRAPHIQUES

### a. Choix de Cewolf : Justification

Cewolf est une bibliothèque qui utilise le moteur de rendu de JFreeChart pour générer des diagrammes à partir de données.

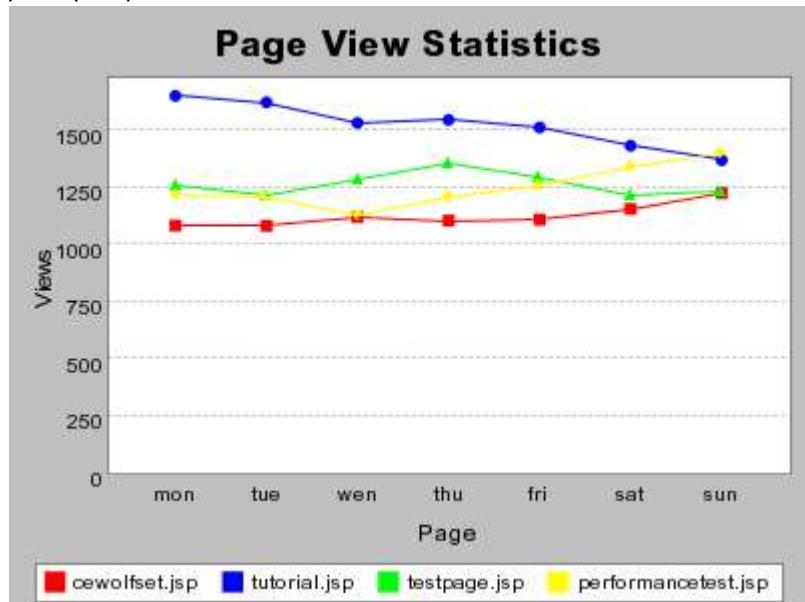
L'intérêt de Cewolf est sa compatibilité et son intégration dans le concept J2EE. Elle propose un environnement de balises *jsp* qui permet de construire tout type de graphiques et de diagrammes et de les afficher dans l'application web.

Nous avons choisi d'utiliser cette bibliothèque pour la génération des diagrammes nécessaires au module de statistiques de notre application. Il fallait absolument une librairie capable de s'intégrer facilement dans notre environnement d'application web.

### b. Utilisation

Pour illustrer l'utilisation de cette librairie, nous allons nous baser sur l'application web d'exemple réalisée dans « Environnement de développement J2EE » (Struts). (Tutorial basé sur <http://cewolf.sourceforge.net/new/index.html>)

Supposons que nous voulions réaliser les statistiques des pages visitées de notre site et l'afficher dans un graphique à peu près comme celui-ci :



## ➤ Préparation de l'application

Dans un premier temps, il faut télécharger les fichiers distribués par Cewolf. Il faut copier les *jar* se trouvant dans le dossier */lib* de la distribution dans le dossier **/WEB-INF/lib/** de notre application web. Voici les fichiers à copier :

- *jfreechart-\*-demo.jar*
- *jfreechart-\*jar*
- *jcommon-\*jar*
- *commons-logging.jar*
- *cewolf.jar*
- *batik-xml.jar*
- *batik-util.jar*
- *batik-svggen.jar*
- *batik-dom.jar*
- *batik.awt-util.jar*

Par ailleurs, le fichier *overlib.js* du dossier */etc* doit être copié dans le dossier racine de l'application web (le dossier **Webapp/** dans notre cas). Ce fichier permet d'afficher des bulles d'aide pour les navigateurs basés sur Mozilla.

Puis dans Eclipse, faire un clique-droit sur le répertoire du projet > Properties > Java Build Path > Libraries > Add JARs... Puis aller sélectionner tous les *jars* ajoutés dans **WEB-INF/lib**.

Notre application web doit être maintenant prête à utiliser Cewolf. Si des problèmes surgissent lors de l'utilisation des *tags* Cewolf (par exemple « No Tags » s'affiche dans la console de Tomcat), il faut copier également le fichier *cewolf.tld* qui se trouve dans le dossier /etc de Cewolf, et le mettre dans le répertoire racine de notre application. Il faut ensuite référencer cette librairie de *tags* dans le fichier *WEB-INF/web.xml*.

Pour tester si notre configuration est bonne, nous pouvons nous rendre à l'adresse <http://localhost:8080/struts-tests/cewolf?state>. Un message apparaît : « **Cewolf servlet up and running.** »

#### ➤ Fabriquer un *DatasetProducer*

Comme Cewolf utilise le modèle MVC, les données affichées dans les graphiques sont séparés de la vue définie dans les pages *jsp*. Nous pouvons donc les modifier séparément.

Pour fabriquer un graphique comportant les données correctes, il est nécessaire de construire un objet qui implante l'interface `de.laures.cewolf.DatasetProducer`. Cet objet est appelé à chaque fois qu'il faut générer un nouveau graphique. Voici un exemple d'implantation d'un *DatasetProducer* qui sera utilisé pour fournir les données nécessaires à notre scénario d'exemple.

```
package fr.uml.v.ir3.strutstests;

import java.io.Serializable;
import java.util.Date;
import java.util.Map;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.jfree.data.category.CategoryDataset;
import org.jfree.data.category.DefaultCategoryDataset;

import de.laures.cewolf.DatasetProduceException;
import de.laures.cewolf.DatasetProducer;
import de.laures.cewolf.links.CategoryItemLinkGenerator;
import de.laures.cewolf.tooltips.CategoryToolTipGenerator;

public class PageViewCountData implements DatasetProducer,
    CategoryToolTipGenerator, CategoryItemLinkGenerator, Serializable {

    private static final Log log = LogFactory.getLog
    (PageViewCountData.class);

    // These values would normally not be hard coded but produced by
    // some kind of data source like a database or a file
    private final String[] categories = {"mon", "tue", "wen", "thu",
    "fri", "sat", "sun"};
    private final String[] seriesNames = {"cewolfset.jsp",
    "tutorial.jsp", "testpage.jsp", "performancetest.jsp"};
```

```

    /**
     * Produces some random data.
     */
    public Object produceDataset(Map params) throws
DatasetProduceException {
    log.debug("producing data.");
    DefaultCategoryDataset dataset = new DefaultCategoryDataset(){
        /**
         * @see java.lang.Object#finalize()
         */
        protected void finalize() throws Throwable {
            super.finalize();
            log.debug(this +" finalized.");
        }
    };
    for (int series = 0; series < seriesNames.length; series++) {
        int lastY = (int)(Math.random() * 1000 + 1000);
        for (int i = 0; i < categories.length; i++) {
            final int y = lastY + (int)(Math.random() * 200 - 100);
            lastY = y;
            dataset.addValue(y, seriesNames[series], categories[i]);
        }
    }
    return dataset;
}

/**
 * This producer's data is invalidated after 5 seconds. By this
method the
 * producer can influence Cewolf's caching behaviour the way it
wants to.
 */
public boolean hasExpired(Map params, Date since) {
    log.debug(getClass().getName() + "hasExpired()");
    return (System.currentTimeMillis() - since.getTime()) > 5000;
}

/**
 * Returns a unique ID for this DatasetProducer
 */
public String getProducerId() {
    return "PageViewCountData DatasetProducer";
}

/**
 * Returns a link target for a special data item.
 */
public String generateLink(Object data, int series, Object category)
{

```

```

        return seriesNames[series];
    }

    /**
     * @see java.lang.Object#finalize()
     */
    protected void finalize() throws Throwable {
        super.finalize();
        log.debug(this + " finalized.");
    }

    /**
     * @see
     */
    org.jfree.chart.tooltips.CategoryToolTipGenerator#generateToolTip
(CategoryDataset, int, int)
    */
    public String generateToolTip(CategoryDataset arg0, int series, int
arg2) {
        return seriesNames[series];
    }

}

```

Nous pouvons constater que ce DatasetProducer n'est pas très utile. Normalement cette classe devrait essayer d'accéder à une source de données (par exemple, une base de données) pour obtenir les informations nécessaires.

Un DatasetProducer doit implanter trois méthodes. La plus importante d'entre elles est la méthode **produceDataset()**. C'est elle qui produit en définitive les données qui devront être rendu sous forme de graphique. Cette méthode prend un paramètre *map* qui est rempli grâce à des *tags* spéciaux des *jsp* (ce que nous verrons par la suite).

La méthode **hasExpired()** est appelée par le *framework* Cewolf s'il existe déjà un objet produit par ce DatasetProducer dans le cache de Cewolf. En retournant la valeur *true*, le DatasetProducer indique que les données précédentes ont expirées.

En fournissant un ID unique via la méthode **getProducerId()**, le *framework* Cewolf identifie un DatasetProducer type. Deux instances de DatasetProducer qui renvoient le même identifiant sont censées produire les mêmes données.

#### ➤ Installer le *servlet* Cewolf dans l'application web

Le rendu final de tous les graphiques est exécuté par une *servlet*. Celle-ci doit être installée dans notre application web. Cewolf est composé d'une classe de *servlet* `de.laures.cewolf.CewolfRenderrer` qui est chargée de faire ce travail.

Il faut donc éditer notre *web.xml* afin d'y ajouter les lignes suivantes :

```
<servlet>
```

```

<servlet-name>CewolfServlet</servlet-name>
<servlet-class>de.laures.cewolf.CewolfRenderer</servlet-class>
<!-- Renseigne l'implantation du storage -->
<init-param>
    <param-name>storage</param-name>
    <param-value>de.laures.cewolf.storage.TransientSessionStorage</param-
value>
</init-param>
<!-- Renseigne la localisation de overlib.js -->
<init-param>
    <param-name>overliburl</param-name>
    <param-value>overlib.js</param-value>
</init-param>
<!-- turn on or off debugging logging -->
<init-param>
    <param-name>debug</param-name>
    <param-value>true</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>

```

Il faut ensuite rajouter le mapping associant le *servlet* de rendu à un type d'URL :

```

<servlet-mapping>
    <servlet-name>CewolfServlet</servlet-name>
    <url-pattern>/cewolf/*</url-pattern>
</servlet-mapping>

```

## ➤ Définition du graphique dans une *jsp*

Voici la page *jsp* que nous réalisons pour afficher le graphique :

```

<%@page contentType="text/html" %>
<%@taglib uri='/WEB-INF/cewolf.tld' prefix='cewolf' %>
<HTML>
<BODY>
<H1>Page View Statistics</H1>
<HR>
<jsp:useBean id="pageViews"
class="fr.uml.vir3.strutstests.PageViewCountData"/>
<cewolf:chart
    id="line"
    title="Page View Statistics"
    type="line"
    xaxislabel="Page"
    yaxislabel="Views">
    <cewolf:data>
        <cewolf:producer id="pageViews"/>
    </cewolf:data>
</cewolf:chart>

```

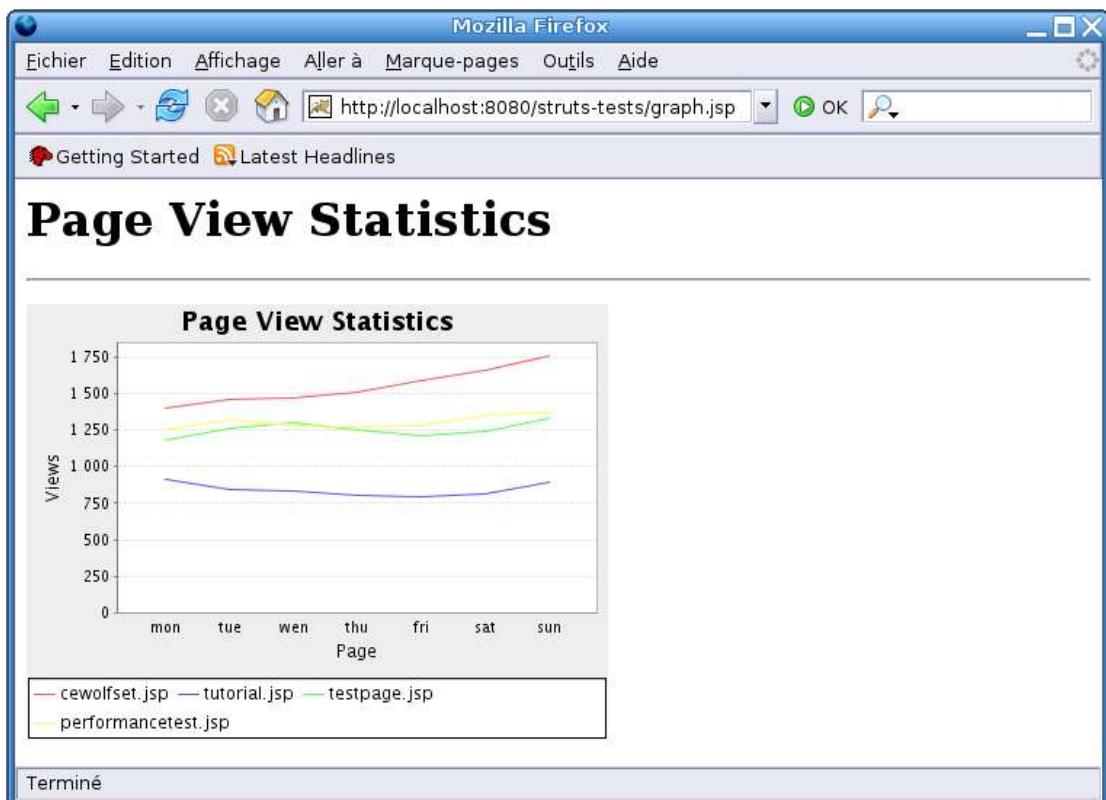
```

<p>
<cewolf:img chartid="line" renderer="cewolf" width="400" height="300" />
</p>
</BODY>
</HTML>

```

Nous pouvons constater l'utilisation de plusieurs balises de cewolf. La première, **<cewolf:chart>** définit le graphique avec tous ses paramètres. Le rendu concret est réalisé par la balise **<cewolf:img>** qui ajoute la balise **<img>** correcte dans la page HTML générée. L'interconnexion entre les différentes balises cewolf dans une *jsp* est faite à l'aide de l'attribut « **id** ».

Lorsque nous appelons cette *jsp* depuis un navigateur, nous voyons ceci :



## 2.8. BIBLIOTHÈQUE DE GÉNÉRATION D'EMAILS

### a. Choix de JavaMail : Justification

JavaMail est une API fournit par Sun, permettant d'envoyer et de recevoir des e-mail. C'est la seule API disponible sous java pour l'envoi de mail. Dans cette présentation, nous étudierons simplement l'envoi d'e-mail. La réception n'étant pas nécessaire pour notre projet.

### b. Concept

(d'après l'étude technique de Chronos sur JavaMail)

L'API est disponible à cette adresse : <http://java.sun.com/products/javamail>. Elle se présente sous la forme d'une archive jar, **mail.jar**. Afin d'accéder et créer un message, il faut utiliser un objet du JavaBean Activation Framework (JAF). Il est également disponible à la même adresse et est nommé **activation.jar**.

## ➤ Les classes utiles

### **La classe Session**

La classe Session établit la connexion avec le serveur de mail. C'est elle qui encapsule les données liées à la connexion (options de configuration et données d'authentification : login, password, nom du serveur).

C'est à partir d'un objet **Session** que toutes les actions concernant les mails sont réalisées. En effet, avant n'importe quelle action, il faut récupérer une Session.

Une session peut être unique ou partagée par plusieurs entités. Pour créer ces sessions nous utilisons les méthodes :

- **getInstance( Properties, Authenticator )** : session unique,
- **getDefaultInstance(Properties, Authenticator )** : session partagée.

Pour obtenir une session, deux paramètres sont attendus :

- Un objet Properties qui contient les paramètres d'initialisation. Un tel objet est obligatoire ;
- Un objet Authenticator optionnel qui permet d'authentifier l'utilisateur auprès du serveur de mail.

```
// création d'une session unique
Session session = Session.getInstance(props,authenticator);
// Récupère la session partagée par défaut
Session defaultSession = session.getDefaultInstance(props,authenticator);
```

La méthode **setDebug()** qui attend en paramètre un booléen est très pratique pour déboguer car avec le paramètre **true**, elle affiche des informations lors de l'utilisation de la session notamment le détail des commandes envoyées au serveur de mail.

### **La classe Message**

La classe Message est une classe abstraite qui encapsule un message.

Lorsque nous avons un objet **Session**, le message peut être créé.

Avant de commencer, il faut savoir qu'un message est composé de deux parties :

- Une en-tête qui contient un ensemble d'attributs (auteur, destinataire, sujet ...)

Ces attributs peuvent être:

**TO** : Adresse Internet du destinataire

**FROM** : Adresse Internet de l'expéditeur

**Date** : date et heure de l'expédition

**Subject** : Le sujet du message

**Message-ID** : numéro d'identification du message

- Un corps qui contient les données à envoyer (contenu)

Pour la plupart de ces données, la classe **Message** implémente l'interface **Part** qui encapsule les attributs nécessaires à la distribution du message (auteur, destinataire, sujet ...) et le corps du message.

Le contenu du message est stocké sous forme d'octets. Pour accéder à son contenu, il faut utiliser un objet du JavaBean Activation Framework (JAF) : **DataHandler**. Ceci permet une séparation des données nécessaires à la transmission et du contenu du message. Ce dernier peut ainsi prendre n'importe quel format.( texte, Html, image, etc.).

La classe **Message** possède deux constructeurs par défaut, et nous utiliserons le constructeur **Message(session)**. La méthode **setText()** permet de facilement mettre une chaîne de caractères dans le corps du message avec un type **MIME** « **text/plain** ». Pour envoyer un message dans un format différent, par exemple HTML, on utilise la méthode **setContent()** qui prend en paramètre un objet et une chaîne qui contient le type **MIME** du message.

Un message peut contenir plusieurs objets il faut alors utiliser un **MultiPart Message**. Cet objet contient un ou plusieurs objets **BodyPart**. La structure d'un objet BodyPart ressemble à celle d'un simple objet Message. Donc chaque objet BodyPart contient des attributs et un contenu.

### **La classe Transport**

La classe **Transport** se charge d'envoyer le message avec le protocole adéquat grâce à sa méthode **send()**. C'est une classe abstraite. Il est possible d'obtenir un objet Transport dédié au protocole particulier utilisé par la session en utilisant la méthode **getTransport()** d'un objet **Session**. Dans ce cas, il faut :

1. Établir la connexion en utilisant la méthode **connect()** avec le nom du serveur, le nom de l'utilisateur et son mot de passe ;
2. Envoyer le message en utilisant la méthode **sendMessage()** avec le message et les destinataires. La méthode **getAllRecipients()** de la classe **message** permet d'obtenir les destinataires du message ;
3. Fermer la connexion en utilisant la méthode **close()**.

Il est préférable d'utiliser une instance de **Transport** comme expliqué ci dessus lorsqu'il y a plusieurs mails à envoyer car il est possible de maintenir la connexion avec le serveur ouverte pendant les envois.

La méthode **static send()** ouvre et ferme la connexion à chacun de ces appels.

## La classe Address

La classe **Address** est une classe abstraite dont héritent toutes les classes qui encapsulent une adresse dans un message. Deux classes filles sont actuellement définies :

- **InternetAddress** ;
- **NewsAddress** (non détaillée ici).

Un objet **InternetAddress** est nécessaire pour chaque émetteur et destinataire de mail. Cependant JavaMail ne vérifie pas l'existence des adresses, c'est le serveur mail qui s'en chargera.

Pour créer une adresse mail, il suffit de passer cette adresse au constructeur :

```
Address address = new InternetAddress("login@etudiant.univ-mlv.fr");
```

Si nous voulons que le nom apparaisse à côté de l'adresse mail, il suffit de passer au constructeur, l'adresse et le nom :

```
Address address = new InternetAddress("login@etudiant.univ-mlv.fr", "Prenom Nom");
```

Il est nécessaire également de préciser l'adresse de l'expéditeur du message. Pour identifier cet expéditeur, nous pouvons utiliser les méthodes **setFrom()** et **setReplyTo()** de la classe **Message**.

```
message.setFrom(address);
```

Si notre message doit montrer plusieurs adresses d'expéditeurs, alors nous utiliserons la méthode **addFrom()** en passant en paramètre un tableau d'adresse :

```
Address address[] = ...; message.addFrom(address);
```

Il existe trois types prédéfinis de la classe **Address** :

- **Message.RecipientType.TO** : destinataire direct ;
- **Message.RecipientType.CC** : copie conforme ;
- **Message.RecipientType.BCC** : copie cachée.

Ainsi la méthode **addRecipient()** permet de préciser, le destinataire et le type d'envoi.

```
Address toAddress = new InternetAddress("login@etudiant.univ-mlv.fr");
Address ccAddress = new InternetAddress("nom.prenom@xxx.com");
message.addRecipient(Message.RecipientType.TO, toAddress);
message.addRecipient(Message.RecipientType.CC, ccAddress);
```

## Exemple complet

Cet exemple permet d'envoyer un mail avec une pièce jointe à plusieurs personnes.

Une vérification de la présence du fichier à joindre est réalisée dans un premier temps. Si aucun fichier n'est fourni, alors la méthode `setText()` de la classe **Message** est utilisée. Sinon un message de type **MimeMultipart** est créé.

```
public static void sendMail (final Collection to, final String subject, final String body, final Collection files) throws MessagingException
{
    final String _From          = getSenderAddress ();
    final String _Subject       = subject;
    final String _Body          = body;
    final ArrayList _To         = new ArrayList (to);
    final ArrayList _Files;
    if (files != null)
        _Files      = new ArrayList (files);
    else
        _Files = null;
    // Set properties
    Properties prop = System.getProperties ();
    prop.put ("mail.smtp.host", getSMTPServerAddress ());
    // Set session
    Session session = Session.getDefaultInstance (prop, null);
    session.setDebug (true);
    Message message = new MimeMessage (session);
    // Set sender
    message.setFrom (new InternetAddress (_From));
    // Get all recipients (destinataires)
    InternetAddress [] internetAddresses = new InternetAddress [_To.size
());
    int cpt = 0;
    Iterator it = _To.iterator();
    while (it.hasNext ())
    {
        internetAddresses [cpt] = new InternetAddress ((String) it.next
());
        cpt++;
    }
    // Set message's informations
    message.setRecipients (Message.RecipientType.TO, internetAddresses);
    message.setSubject (_Subject);
    message.setSentDate (new Date ());
    message.setHeader ("Chronos", ChronosParameters.CHRONOS_VERSION);
    // If there is no file to send
    if (_Files == null)
        message.setText (_Body);
    else
    {
```

```

// It is a multipart message
Multipart multipart = new MimeMultipart ();
// The first part of the message is created
BodyPart messageBodyPart = new MimeBodyPart ();
messageBodyPart.setText (_Body);
multipart.addBodyPart (messageBodyPart);
// The files are added to the message
Iterator itFiles = _Files.iterator();
while (itFiles.hasNext ())
{
    messageBodyPart = new MimeBodyPart ();
    DataSource source = new FileDataSource
((String)
itFiles.next ());
    messageBodyPart.setDataHandler (new
DataHandler
(source));
    messageBodyPart.setFileName (source.getName ());
    multipart.addBodyPart (messageBodyPart);
}
// Add the file to the message
message.setContent (multipart);
}
// Send the mail
Transport.send (message);
}

```

### c. Liens utiles

Voici une liste de liens utiles pour l'API JavaMail :

- Le site officiel :

<http://java.sun.com/products/javamail/>

- Une page de présentation de tutoriel :

<http://java.sun.com/developer/onlineTraining/JavaMail/index.html>

- Comment envoyer des emails en HTML :

<http://java.sun.com/developer/EJTechTips/2004/tt0426.html#1>

- Envoyer des emails en HTML, avancé :

<http://java.sun.com/developer/EJTechTips/2004/tt0625.html#1>

## **2.9. BIBLIOTHÈQUE DE GÉNÉRATION DE FEUILLES EXCEL**

### **a. Choix de Java POI : Justification**

Java POI est une librairie libre et gratuite qui permet principalement de générer des fichiers au format Microsoft Excel.

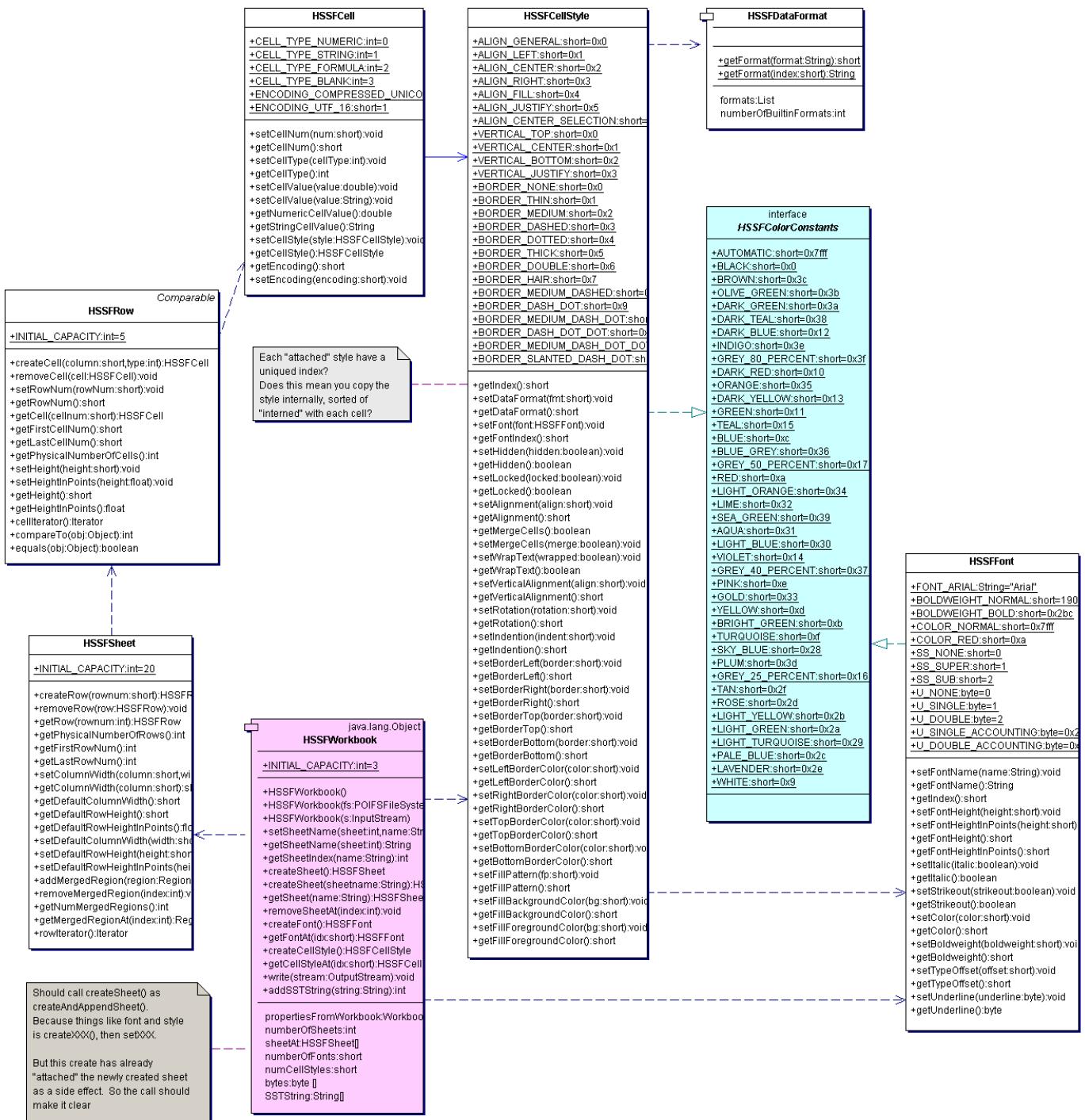
Nous avons décidé d'utiliser cette librairie pour satisfaire le besoin de génération d'export au format Excel.

Java POI regroupe en fait un ensemble de plusieurs librairies. Celle qui va nous intéresser plus particulièrement est l'API HSSF qui est le module permettant de lire et d'écrire des fichiers XLS (format de fichier Excel de Microsoft).

Cette API couvre l'ensemble des fonctions qui nous sont nécessaires pour notre module d'export.

## b. Concept

Voici le schéma complet de l'API HSSF permettant de lire et d'écrire les fichiers XLS.



Si nous regardons ce schéma, nous nous apercevons que tout le concept des feuilles Excel a été décortiqué et regroupé en plusieurs objets. Ainsi, nous retrouvons les classeurs (« HSSFWorkbook ») regroupant un ensemble de feuilles de calcul (« HSSFSheet »), les lignes contenues dans une feuille (« HSSFRow »), ainsi que les cellules de ces lignes (« HSSFCell »).

## ➤ Installation

Pour installer HSSF, il faut aller sur le site de POI <http://jakarta.apache.org/poi/index.html> et télécharger la librairie complète. Ensuite, il faut la rajouter dans le classpath courant de l'application.

## ➤ Utilisation

Nous allons nous baser sur un exemple pour expliquer le fonctionnement de cette API. L'exemple sera, en fait, un cas que nous aborderons dans notre application : exporter le résultat d'une requête SQL dans un fichier XLS.

Dans un premier temps, nous créons le package. C'est un exemple ici.

```
package fr.umlv.ir3.hssftest;
```

Ensuite nous importons les librairies qui vont nous servir.

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.math.BigDecimal;
import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
```

Les trois dernières librairies sont celles de POI. Les autres (java.sql.\*) vont simplement servir à établir une connexion à la base de données.

Voici maintenant notre classe ainsi que sa méthode principale qui prend en paramètre un **ResultSet** qui est le résultat d'une requête SQL **SELECT** quelconque, ainsi que le nom du fichier **.xsl** à générer.

```
public class HSSFExport {
    public static void exportSQLDatasToXSLWorkbook(ResultSet resultSet,
                                                   String nameOfXSLWorkbookFile) throws IOException, SQLException
{
```

D'abord nous créons le classeur :

```
HSSFWorkbook wb = new HSSFWorkbook();
```

Puis la feuille de calcul « export » :

```
HSSFSheet sheet = wb.createSheet("export");
```

Ensuite nous allons exporter les titres des colonnes du résultat dans la feuille de calcul. Pour ceci, nous créons d'abord une nouvelle ligne tout en haut du fichier (ligne d'ordonnées 0) :

```
HSSFRow row = sheet.createRow((short) 0);
```

Puis nous récupérons les titres du ResultSet, et les exportons un par un dans la ligne, en créant à chaque fois une nouvelle cellule :

```
ResultSetMetaData metaData = resultSet.getMetaData();
int nbOfColumn = metaData.getColumnCount();
for (int i = 0; i < nbOfColumn; i++) {
    row.createCell((short) i).setCellValue(
        metaData.getColumnLabel(i + 1));
}
```

Maintenant nous allons extraire les valeurs de chaque ligne du ResultSet.

Pour cela, nous mettons en place une boucle qui crée à chaque nouveau tuple de donnée, une nouvelle ligne dans la feuille de calcul puis pour chaque colonne du tuple, une cellule et y insère la valeur :

```
for (int j = 0; resultSet.next(); j++) {
    // Création d'une nouvelle ligne dans la feuille
    row = sheet.createRow((short) j + 1);
    for (int i = 0; i < nbOfColumn; i++) {
        Object value = resultSet.getObject(i + 1);
        // création de la case et export de sa valeur (suivant son type)
        if (value != null) {
            if (value instanceof Date) {
                row.createCell((short) i).setCellValue((Date) value);
            } else if (value instanceof Boolean) {
                row.createCell((short) i).setCellValue((Boolean) value);
            } else if (value instanceof BigDecimal) {
                row.createCell((short) i).setCellValue(
                    ((BigDecimal) value).doubleValue());
            } else if (value instanceof Integer) {
                row.createCell((short) i).setCellValue((Integer) value);
            } else if (value instanceof Long) {
                row.createCell((short) i).setCellValue((Long) value);
            } else if (value instanceof Double) {
                row.createCell((short) i).setCellValue((Double) value);
            } else {
                row.createCell((short) i)
                    .setCellValue(value.toString());
            }
        }
    }
}
```

Enfin, dans un dernier temps, nous écrivons le classeur en mémoire, dans un fichier .xls :

```
FileOutputStream fileOut = new FileOutputStream(nameOfXSLWorkbookFile);
    wb.write(fileOut);
    fileOut.close();
}
```

Voici maintenant la méthode main qui a pour but de créer une connexion à une base mysql, de faire un SELECT de manière à récupérer la liste des utilisateurs, ainsi que tous leurs paramètres :

```
public static void main(String[] args) throws ClassNotFoundException,
    SQLException, IOException {
    Class.forName("com.mysql.jdbc.Driver");
    Connection connection = DriverManager.getConnection(
        "jdbc:mysql://127.0.0.1:3306/mysql", "root", "");
    Statement statement = connection.createStatement();
    ResultSet resultSet = statement.executeQuery("SELECT * FROM user");
    exportSQLDatastoXSLWorkbook(resultSet, "toto.xls");
    resultSet.close();
    statement.close();
    connection.close();
}
```

Nous considérons que la base mysql se trouve sur l'ordinateur local, que l'utilisateur « root » a un mot de passe vide et qu'il a les droits de sélections sur le domaine local.

Nous obtenons alors un fichier « toto.xls » comportant toutes les données de la table des utilisateurs de la base mysql :

## 2.10. OUTILS DE DÉPLOIEMENTS

Les outils de déploiements permettent de déployer l'application pour qu'elle soit prête à être utilisée.

### a. Comparatif

	Langages	Plate formes	Apprentissage	Documentation
Make	Tous	Toutes	Moyen	Bonne
Automake/Autoconf	C/C++	Unix/cygwin	Difficile	Bonne
Cmake	C/C++	unix/win32	Facile	Bonne
Tmake	C/C++	unix/win32	Facile	Moyenne
Qmake	C/C++	unix/win32/MacOSX	Facile	Bonne
Ant	Java	Toutes	Facile	Bonne

### b. Justification

Ant est gratuit et librement diffusable. Il est téléchargeable, copiable, intégrable à vos projets. Il est possible de réaliser des produits commerciaux sans avoir à reverser une partie de des gains à la fondation. Ainsi, par exemple, le site E-Commerce O'Reilly ou les principaux IDEs comme Eclipse, JBuilder, Netbeans ou WebSphere Studio Application Developer l'ont parfaitement intégré à leur infrastructure.

Ant est portable, il fonctionne aussi bien sous Linux, Solaris, Windows 9x/NT/XP, sous Novell Netware 6 ou MacOS X.

Ant est populaire puisqu'il est supporté à la fois par de grandes firmes commerciales comme **IBM**, **SUN** ou **BEA** et par la communauté Open Source. Il fait l'objet de forums très fréquentés par la communauté internationale, ainsi que de listes de diffusions. De nombreux articles à son sujet sont d'ailleurs accessibles via le World Wide Web et plusieurs ouvrages ont d'ores et déjà été publiés.

## c. Tests et exemples

Ant est un outil créé par la fondation Apache (qui gère déjà le serveur éponyme du même nom, PHP, Struts, Tomcat...), qui permet de gérer la construction d'une application. A l'origine un sous-projet de Jakarta, il est depuis devenu l'un des projets majeurs de la fondation, de par son utilité et sa popularité.

Pour les habitués, il peut se comparer à la commande make (même rôle, même fonctionnalité), mais offre l'avantage de pouvoir intégrer des tests tout du long du processus de construction (avec JUnit par exemple), d'être multi-plate formes, d'être extensible de multiples manières...

Couplé à un système de type CVS, Ant permet de créer des "*nightly builds*" : chaque nuit, sur une machine de base, il compile l'ensemble des fichiers sources d'un projet, et fait son rapport au(x) développeur(s). Cela permet de générer des versions utilisables, de tester les lignes de code de la journée, et de faire passer une batterie de tests à l'ensemble du projet, le tout sans intervention humaine. Lorsque l'équipe de développement arrive le lendemain, le rapport de Ant peut leur permettre de cerner les erreurs, ou de constater que le programme se compile parfaitement et est fonctionnel.

### ➤ Installation

Après avoir téléchargé une distribution du produit et l'avoir décompressé dans son répertoire définitif, il faut impérativement créer les trois variables d'environnement nécessaires au fonctionnement d'Ant. Par exemple, sous Windows dans un fichier .bat, ou via l'onglet "Avancé" des Propriétés Système :

```
SET ANT_HOME=C:\tools\ant
SET JAVA_HOME=C:\j2sdk1.4.2_02
SET PATH=%PATH%;%ANT_HOME%\bin
```

### ➤ Utilisation

Ant permet de gérer l'ensemble des travaux à l'aide d'un fichier XML, build.xml, spécifique à chaque projet, qui permet de spécifier les cibles, dépendances, conditions, opérations et paramètres de la construction.

```
<project name="monProject" default="compileProject">
    <target name="init">
        <property name="sourceDir value="/src">
        <property name="outputDir value="/classes">
    </target>

    <target name="cleanUpOutputDir" depends="init">
        <deltree dir="${outputDir}" />
    </target>

    <target name="makeOutputDir" depends="cleanUpOutputDir">
        <mkdir dir="${outputDir}" />
    </target>
```

```

<target name="compileProject" depends="makeOutputDir">
    <javac srcdir="${sourceDir}" destdir="${outputDir}" />
</target>
</project>

```

Ce premier fichier build.xml est déjà facile à lire : on comprend vite les quatre tâches à accomplir à chaque lancement : définir les chemins d'accès, effacer répertoire de destination (qui contient les fichiers du dernier lancement), créer à nouveau le dossier de destination, et lancer javac sur nos fichier .java. A la fin du processus, si toutes les étapes se sont bien déroulées, on aura nos .class dans le dossier / classes.

L'attribut default de la balise globale project permet de pointer vers la balise target à exécuter. Etant donné que, via l'attribut depends, chaque balise target dépend de la précédente, on permet ainsi le parcours de toutes les opérations à accomplir. De son coté, la cible init, qui sert à initialiser certaines propriétés (une sorte de constructeur, donc) est toujours appelée en premier.

Bien qu'optionnel, cet attribut depends se rend très utile car il permet donc de conditionner les tâches du fichier : une tâche donnée ne sera exécutée que si celle dont elle dépend est terminée, et ainsi de suite.

Une opération peut aussi dépendre de plusieurs opérations : depends="makeOutputDir, cleanUpOutputDir".

Les commandes deltree, mkdir et javac, bien que connues, font en fait appel à des fonctions de Ant (et non aux commandes système directement). Ant dispose ainsi d'un bon nombre de commandes : copydir, copyfile, jar, javadoc...

Une fois que le fichier XML adéquat pour votre application est placée dans son dossier, il suffit ensuite de taper "ant" dans la console pour construire l'application - du moins, sa version compilable pour le moment.

L'utilisation de Ant permet de mettre en application une méthode de développement efficace, faisant appel à l'intégration continue. Une application est construite autant de fois que nécessaire.

## 2.11. SYSTÈME DE GESTION DE BASE DE DONNÉES

Un système de gestion de base de données est nécessaire pour l'application, c'est lui qui va permettre d'avoir une persistance des données et de centraliser les données.

### a. Comparatif

Deux concurrents Open Source apparaissent dans la gestion de base de données :

- PostgreSQL
- MySql

L'étude porte sur PostgreSQL 8.1 et MySQL 4. MySQL 5 ne sera pas étudié ici, considérant que la version est trop récente pour être stable, malgré qu'elle comble de nombreuses lacunes (procédures stockées et déclencheurs).

	<b>PostgreSQL</b>	<b>MySQL</b>
SQL standard	<i>PostgreSQL</i> comprend un bon sous-ensemble de SQL92 et même un bon sur-ensemble; <i>PostgreSQL</i> dispose de capacités d'héritage objet (outer joins existent)	<i>MySQL</i> comprend un sous-ensemble de SQL92 mais il manque notamment de sous-requêtes et de vues.
VITESSE	<i>PostgreSQL</i> est relativement lent, mais avec beaucoup d'options d'optimisation. De l'option -F pour réécriture de requêtes jusqu'aux différentes stratégies d'optimisation (SET KSQO, SET GEQO). <i>PostgreSQL</i> créé un nouveau processus à chaque connexion (ralenti en Solaris/Windows mais pas en linux)	<i>MySQL</i> est très rapide avec des requêtes simples, mais beaucoup moins avec des complexes. Les connexions sont gérées très rapidement.
STABILITE	<i>PostgreSQL</i> est très stable à partir de la version 7.1	<i>MySQL</i> est très utilisé et très stable.
INTEGRITE	<i>PostgreSQL</i> a des transactions/rollbacks et clés étrangères	<i>MySQL</i> gère les transactions et les clés étrangères.
PROCEDURES STOCKEES	<i>PostgreSQL</i> peut être étendu avec des fonctions écrites en C, pgsql, python, perl et tcl .	<i>MySQL</i> ne gère pas tout cela, juste des bibliothèques partagées pour fonctions C.
VEROUILLAGE	<i>PostgreSQL</i> utilise MVCC (MultiVersion Concurrency Control)	<i>MySQL</i> est censé faire la même chose dans sa version 4.0; jusqu'alors, seulement verrouillage de tables.
BLOBS	En <i>PostgreSQL</i> , les blobs sont spéciaux; ils se gèrent dans une table à part et ils sont disjoints des tables normales	En <i>MySQL</i> c'est un type comme un autre, malgré quelques limitations.
TRIGGERS	En <i>PostgreSQL</i> les triggers sont intégrés	En <i>MySQL</i> il n'existe pas de triggers.

## b. Justification

Suite à ce tableau comparatif il a été décidé de sélectionner PostGreSQL qui intègre beaucoup plus de fonctionnalité que MySQL

## c. Utilisation

### ➤ Concept

Une base de données contient les éléments suivant :

### Casts

Les casts permettent de convertir différentes données.

### Langue

Langage de programmation supporté par la base de données, par défaut plsql.

## **Schéma**

Un schéma est le cœur de la base relationnelle, par défaut un schéma public est créé.

## **Agrégats**

Les agrégats permettent d'effectuer des opérations sur une une colonne.

## **Conversion**

La conversion permet de créer des conversion de codage (LATIN, etc..).

## **Domaines**

Les domaines permettent de définir une seule fois des contraintes pour plusieurs colonnes dans des tables différentes.

## **Fonctions**

Les fonctions permettent de manipuler les données de la base de donnée à l'aide d'un langage de programmation. Elles peuvent être appelée à partir des clients.

## **Fonction Déclencheur**

Les fonctions « déclencheur » permettent d'exécuter du code lors de l'insertion, la suppression, la mise à jour des données de la base de données.

## **Opérateurs**

Les opérateurs permettent d'implémenter des opération ( +, -, /, etc..) sur des types de données qui ne comportant pas d'opérateur.

## **Séquences**

Les séquences permettent de gérer l'auto-incrémentation des colonnes, à champ unique par exemple.

## **Tables**

Il s'agit des tables de la base de données.

## **Contraintes**

Les contraintes permettent de contrôler les données de la table.

## **Index**

Les index permettent d'augmenter les rapidité de traitement.

## **Règle**

Les règles sont des Triggers simplifiés.

## **Type**

Le type permet de définir un type utilisateur (structure de données).

## **Vue**

Les vues sont des tables virtuelles qui sont créés par une expression SQL.

## **Utilisateur**

Il existe des listes d'utilisateurs stockées dans pg\_hba.conf et pg\_ident.conf. Elles réferent les utilisateurs capables de se connecter à la base de données.

## **Groupe**

Cet élément permet de rassembler des utilisateurs par groupe. Chaque groupe dispose de ces propres droits.

### **d. Tests et exemples**

Les bases de données relationnelles étant devenues assez utilisées, les fonctions SQL standards sont très largement expliquées sur internet. Par conséquent, nous n'allons utiliser et tester que les fonctionnalités avancées et spécifiques à PostgreSQL.

## ➤ **Index**

Les index permettent de spécifier la méthode de référencement d'une colonne, de plusieurs colonnes, ou même d'une partie d'une colonne.

Supposons que nous enregistrons un journal d'accès à un serveur web dans une base de données. La plupart des accès proviennent de classes d'adresses IP internes à l'organisation, mais certaines viennent d'ailleurs (disons des employés connectés par modem). Si les recherches sur des adresses IP concernent essentiellement les accès extérieurs, il n'y a pas besoin d'indexer les classes d'adresses IP qui correspondent au sous-réseau de votre organisation.

Supposons que la table soit comme ceci :

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

Pour créer un index partiel qui corresponde à l'exemple, il faut utiliser une commande comme celle-ci :

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
    WHERE NOT (client_ip > inet '192.168.100.0' AND client_ip < inet
    '192.168.100.255');
```

Une requête typique qui peut utiliser cet index est :

```
SELECT * FROM access_log WHERE url = '/index.html' AND client_ip = inet
    '212.78.10.32';
```

Une requête qui ne peut pas l'utiliser est :

```
SELECT * FROM access_log WHERE client_ip = inet '192.168.100.23';
```

Ce type d'index nécessite que les valeurs courantes soient prédéterminées. Si la distribution des valeurs est inhérente (du fait de la nature de l'application) et statique (ne changeant pas dans le temps), ce n'est pas trop difficile. Par ailleurs, si les valeurs courantes sont simplement dues au hasard, cela peut demander beaucoup de travail de maintenance.

Il est possible de spécifier les méthodes de tri : btree, hash, rtree et gist , avec le mot clef USING.

**Remarque :** Il est impossible d'utiliser des sous requêtes et des expressions d'agrégats dans la close where de l'index.

## ➤ Déclencheur

Cet exemple de déclencheur nous assure que toute insertion, modification ou suppression d'une ligne dans la table emp est enregistrée (c'est-à-dire auditée) dans la table emp\_audit. L'heure et le nom de l'utilisateur sont conservées dans la ligne avec le type d'opération réalisé.

```
CREATE TABLE emp (
    empname      text NOT NULL,
    salary        integer
);

CREATE TABLE emp_audit(
    operation     char(1)  NOT NULL,
    stamp         timestamp NOT NULL,
    userid        text      NOT NULL,
    empname       text      NOT NULL,
    salary        integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
BEGIN
    --
    -- Ajoute une ligne dans emp_audit pour refléter l'opération réalisée
    -- sur emp,
    -- utilise la variable spéciale TG_OP pour cette opération.
    --
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
        RETURN NEW;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
        RETURN NEW;
    END IF;
    RETURN NULL; -- result is ignored since this is an AFTER trigger

```

```

END;
$emp_audit$ language plpgsql;

CREATE TRIGGER emp_audit
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();

```

Une utilisation des déclencheurs est le maintien d'une table résumée d'une autre table. Le résumé résultant peut être utilisé à la place de la table originale pour certaines requêtes — souvent avec des temps d'exécution bien réduits. Cette technique est souvent utilisée pour les statistiques de données où les tables de données mesurées ou observées (appelées des tables de faits) peuvent être extrêmement grandes.

## ➤ Agrégats

Dans PostgreSQL, les fonctions d'agrégat sont exprimées comme des « valeurs d'état » et des « fonctions d'état de transition ». Autrement dit, un agrégat peut être défini en termes d'état modifié chaque fois qu'une entrée est traitée. Pour définir une nouvelle fonction d'agrégat, on choisit un type de donnée pour la valeur d'état, une valeur initiale pour l'état et une fonction de transition d'état. La fonction de transition d'état est simplement une fonction ordinaire qui pourrait aussi bien être utilisée hors du contexte de l'agrégat. Une « fonction finale » peut également être spécifiée, au cas où le résultat désiré pour l'agrégat soit différent des données devant être conservées comme valeur courante de l'état.

Ainsi, en plus des types de données de l'argument et du résultat vus par l'utilisateur, il existe un type de donnée pour la valeur d'état interne qui peut être différent de ces deux derniers.

Si nous définissons un agrégat qui n'utilise pas de fonction finale, nous avons un agrégat qui calcule pour chaque ligne une fonction des valeurs de colonnes. `sum` est un exemple de cette sorte d'agrégat. `sum` commence à zéro et ajoute toujours la valeur de la ligne courante à son total en cours. Par exemple, si nous voulons faire un agrégat `sum` pour opérer sur un type de donnée pour des nombres complexes, nous avons seulement besoin de la fonction d'addition pour ce type de donnée. La définition de l'agrégat sera :

```

CREATE AGGREGATE somme_complexe (
    sfunc = ajout_complexe,
    basetype = complexe,
    stype = complexe,
    initcond = '(0,0)'
);

SELECT somme_complexe(a) FROM test_complexe;

somme_complexe
-----
(34,53.9)

```

(Dans la pratique, nous aurions seulement nommé l'agrégat `sum` et laissé PostgreSQL déterminer quel genre de somme appliquer à une colonne de type `complexe`.)

La définition précédente de `sum` renverra zéro (la condition d'état initial) s'il n'y a pas de valeurs d'entrée non `NULL`. Peut-être désirons-nous, que dans ce cas, elle retourne `NULL`. Le standard SQL prévoit que la fonction `sum` se comporte ainsi. Nous pouvons faire ceci simplement en omettant l'instruction `initcond`, de sorte que la condition d'état initial soit `NULL`. Ordinairement, ceci signifierait que `sfunc` aurait à vérifier l'entrée d'une condition d'état `NULL`, mais pour la fonction `sum` et quelques autres agrégats simples comme `max` et `min`, il suffit d'insérer la première valeur d'entrée non `NULL` dans la variable d'état et ensuite de commencer à appliquer la fonction de transition d'état à la seconde valeur non `NULL`. PostgreSQL fera cela automatiquement si la condition initiale est `NULL` et si la fonction de transition est marquée `<< strict >>` (c'est-à-dire qu'elle ne doit pas être appelée pour des entrées `NULL`).

Un autre comportement par défaut d'une fonction de transition `<< strict >>` est que la valeur d'état précédente est gardée inchangée chaque fois qu'une entrée `NULL` est rencontrée. Ainsi, les valeurs `NULL` sont ignorées. Si nous avons besoin d'un autre comportement pour les entrées `NULL`, il ne faut pas définir la fonction de transition comme `<< strict >>`, mais la coder pour vérifier les entrées `NULL` et faire le nécessaire.

`avg` (`average = moyenne`) est un exemple plus compliqué d'agrégat. Il demande deux état courants : la somme des entrées et le compte du nombre d'entrées. Le résultat final est obtenu en divisant ces quantités. La moyenne est typiquement implémentée en utilisant comme valeur d'état un tableau de deux éléments. Par exemple, l'implémentation intégrée de `avg(float8)` ressemble à :

```
CREATE AGGREGATE avg (
    sfunc = float8_accum,
    basetype = float8,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0}'
);
```

Les fonctions d'agrégat peuvent utiliser des fonctions d'état de transition ou des fonctions finales polymorphes, de sorte que les mêmes fonctions peuvent être utilisées pour implémenter de multiples agrégats. Pour aller encore plus loin, la fonction d'agrégat elle-même peut être spécifiée avec un type de base et un type d'état polymorphes, permettant ainsi à une unique définition de servir pour de multiples types de données d'entrée. Voici un exemple d'agrégat polymorphe :

```
CREATE AGGREGATE array_accum (
    sfunc = array_append,
    basetype = anyelement,
    stype = anyarray,
    initcond = '{}'
);
```

Ici, le type d'état effectif pour n'importe quel appel d'agrégat est le type tableau, ayant comme éléments le type effectif d'entrée.

Voici le résultat quand on utilise deux types de donnée effectifs différents comme arguments :

```
SELECT attrelid::regclass, array_accum(attname)
FROM pg_attribute WHERE attnum > 0
AND attrelid = 'pg_user'::regclass GROUP BY attrelid;
attrelid |           array_accum
-----
+
-
 pg_user | {username,usesysid,usecreatedb,usesuper,usecatupd,passwd,valuuntil,useconfig}
(1 row)

SELECT attrelid::regclass, array_accum(atttypid)
FROM pg_attribute WHERE attnum > 0
AND attrelid = 'pg_user'::regclass GROUP BY attrelid;
attrelid |           array_accum
-----
+
 pg_user | {19,23,16,16,16,25,702,1009}
(1 row)
```

## ➤ Domaines

Cet exemple crée le type de données `code_postal_us`, puis l'utilise dans la définition d'une table. Un test d'expression rationnelle est utilisé pour vérifier que la valeur ressemble à un code postal US valide.

```
CREATE DOMAIN code_postal_us AS TEXT
CHECK(
    VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE courrier_us (
    id_adresse SERIAL NOT NULL PRIMARY KEY
, rue1 TEXT NOT NULL
, rue2 TEXT
, rue3 TEXT
, ville TEXT NOT NULL
, code_postal code_postal_us NOT NULL
);
```

## ➤ Transactions

Les *transactions* forment un concept fondamental de tous les systèmes de bases de données. Le point essentiel d'une transaction est qu'il assemble plusieurs étapes en une seule opération tout-ourien.

Les états intermédiaires entre les étapes ne sont pas visibles pour les autres transactions concurrentes, et si un échec survient empêchant la transaction de bien se terminer, alors aucune des étapes n'affecte la base de données.

Par exemple, considérez la base de données d'une banque qui contiendrait la balance pour différents comptes clients, ainsi que les balances du total du dépôt par branches. Supposez que nous voulons enregistrer un virement de 100 euros du compte d'Alice vers celui de Bob. En simplifiant énormément, les commandes SQL pour ceci ressembleraient à ça :

```
UPDATE comptes SET balance = balance - 100.00
    WHERE nom = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE nom = (SELECT nom_branche FROM comptes WHERE nom = 'Alice');
UPDATE comptes SET balance = balance + 100.00
    WHERE nom = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE nom = (SELECT nom_branche FROM comptes WHERE nom = 'Bob');
```

Les détails de ces commandes ne sont pas importants ici ; le point important est que cela nécessite plusieurs mises à jour séparées pour accomplir cette opération assez simple. Les employés de la banque voudront être assurés que soit toutes les commandes sont effectuées soit aucune ne l'est. Il ne serait pas acceptable que, suite à une erreur du système, Bob reçoive 100 euros qui n'ont pas été débités du compte d'Alice. De la même façon, Alice ne restera pas longtemps une cliente si elle est débitée du montant sans que celui-ci ne soit crédité sur le compte de Bob. Nous avons besoin d'une garantie comme quoi si quelque chose se passe mal, aucune des étapes déjà exécutées ne prendra effet. Grouper les mises à jour en une *transaction* nous donne cette garantie. Une transaction est dite *atomique* : du point de vue des autres transactions, cela se passe complètement ou pas du tout.

Nous voulons aussi la garantie qu'une fois une transaction terminée et validée par le système de base de données, les modifications seront enregistrées de façon permanente et ne seront pas perdues même si un arrêt brutal arrive peu après. Par exemple, si nous enregistrons un retrait d'argent par Bob, nous ne voulons surtout pas que le débit de son compte disparaisse lors d'un crash à sa sortie de la banque. Une base de données transactionnelle garantit que toutes les mises à jour faites lors d'une transaction sont enregistrées dans un stockage permanent (c'est-à-dire sur disque) avant que la transaction ne soit validée.

Une autre propriété importante des bases de données transactionnelles est en relation étroite avec la notion de mises à jour atomiques : quand de multiples transactions sont lancées en parallèle, chacune d'entre elles ne doit pas être capable de voir les modifications incomplètes faites par les autres. Par exemple, si une transaction est occupée à calculer le total de toutes les branches, il ne serait pas bon d'inclure le débit de la branche d'Alice sans le crédit de la branche de Bob, ou vice-versa. Donc, les transactions doivent être tout-ou-rien non seulement pour leur effet permanent sur la base de données, mais aussi pour leur visibilité au moment de leur exécution. Les mises à jour faites ainsi par une

transaction ouverte sont invisibles aux autres transactions jusqu'à la fin de celle-ci, moment qui rendra visible toutes les mises à jour simultanément.

Avec PostgreSQL, une transaction est réalisée en entourant les commandes SQL de la transaction avec les commandes `BEGIN` et `COMMIT`. Donc, notre transaction pour la banque ressemblera à ceci :

```
BEGIN;
UPDATE comptes SET balance = balance - 100.00
    WHERE nom = 'Alice';
-- etc etc
COMMIT;
```

Si, au cours de la transaction, nous décidons que nous ne voulons pas valider (peut-être nous sommes-nous aperçu que la balance d'Alice devenait négative), nous pouvons envoyer la commande `ROLLBACK` au lieu de `COMMIT`, et toutes nos mises à jour jusqu'à maintenant seront annulées.

En fait, PostgreSQL traite chaque instruction SQL comme étant exécutée dans une transaction. Si vous ne lancez pas une commande `BEGIN`, alors chaque instruction individuelle se trouve enveloppée avec un `BEGIN` et (en cas de succès) un `COMMIT` implicites. Un groupe d'instructions entouré par un `BEGIN` et un `COMMIT` est quelque fois appelé un *bloc transactionnel*.

**Note :** Quelques bibliothèques clients lancent les commandes `BEGIN` et `COMMIT` automatiquement.

Il est possible de contrôler les instructions dans une transaction d'une façon plus granulaire avec l'utilisation des *points de sauvegarde*. Les points de sauvegarde vous permettent d'annuler des parties de la transaction tout en validant le reste. Après avoir défini un point de sauvegarde avec `SAVEPOINT`, nous pouvons, si nécessaire, annuler jusqu'au point de sauvegarde avec `ROLLBACK TO`. Toutes les modifications de la transaction dans la base de données, entre le moment où le point de sauvegarde est défini et celui où l'annulation est demandée, sont annulées mais les modifications antérieures au point de sauvegarde sont conservées.

Après avoir annulé jusqu'à un point de sauvegarde, il reste défini, donc il est possible, de nouveau d'annuler plusieurs fois, et rester au même point. Par contre, si nous sommes sûrs de ne plus avoir besoin d'annuler jusqu'à un point de sauvegarde particulier, il peut être libéré pour que le système puisse récupérer quelques ressources. Il faut garder à l'esprit que libérer un point de sauvegarde ou annuler les opérations jusqu'à ce point de sauvegarde libérera tous les points de sauvegarde définis après lui.

Tout ceci survient à l'intérieur du bloc de transaction, donc ce n'est pas visible par les autres sessions de la base de données. Quand et si le bloc de transaction est validé, les actions validées deviennent visibles en un seul coup aux autres sessions, alors que les actions annulées ne deviendront jamais visibles.

Reprendons notre exemple de banque. Supposons que nous débitons le compte d'Alice de \$100.00, somme que nous créditions au compte de Bob, pour trouver plus tard que nous aurions dû créditer le compte de Wally. Nous pouvons le faire en utilisant des points de sauvegarde comme ceci :

```

BEGIN;
UPDATE comptes SET balance = balance - 100.00
    WHERE nom = 'Alice';
SAVEPOINT mon_pointdesauvegarde;
UPDATE comptes SET balance = balance + 100.00
    WHERE nom = 'Bob';
-- oups ... oubliions ça et créditons le compte de Wally
ROLLBACK TO mon_pointdesauvegarde;
UPDATE comptes SET balance = balance + 100.00
    WHERE nom = 'Wally';
COMMIT;

```

Cet exemple est bien sûr très simplifié mais il y a beaucoup de contrôle possible dans un bloc de transaction grâce à l'utilisation des points de sauvegarde. De plus, `ROLLBACK TO` est le seul moyen pour regagner le contrôle d'un bloc de transaction qui a été placé dans un état d'annulation par le système à cause d'une erreur, plutôt que de tout annuler et de tout recommencer.

#### ➤ Clef secondaire

Le problème est de s'assurer que personne n'insère de lignes dans la table `temps` qui ne correspondent pas à une entrée dans une table `villes`. Ceci maintient l'*intégrité référentielle* de vos données. Dans les systèmes de bases de données simples, ceci serait implémenté (si possible) en vérifiant en premier lieu que la table `villes` dispose bien d'un enregistrement correspondant, puis en insérant ou en empêchant l'insertion du nouvel enregistrement dans `temps`. Cette approche présente un certain nombre de problèmes et n'est pas très pratique, donc PostgreSQL peut s'en charger automatiquement.

La nouvelle déclaration des tables ressemblerait à ceci :

```

CREATE TABLE villes (
    ville      varchar(80) primary key,
    emplacement point
);

CREATE TABLE temps (
    ville      varchar(80) references villes,
    temp_haute int,
    temp_basse int,
    prcp      real,
    date       date
);

```

Maintenant, essayons d'insérer un enregistrement non valide :

```

INSERT INTO temps VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');
ERROR: insert or update on table "temps" violates foreign key constraint
"temps_ville_fkey"
DETAIL: Key (ville)=(Berkeley) is not present in table "villes".

```

## ➤ Héritages

L'héritage est un concept provenant des bases de données orientées objet. Il ouvre de nouvelles possibilités intéressantes dans la conception de bases de données.

Créons deux tables : une table `villes` et une table `capitales`. Naturellement, les capitales sont aussi des villes, donc vous voulez un moyen pour afficher implicitement les capitales lorsque vous listez les villes.

```
CREATE TABLE capitales (
    nom        text,
    population real,
    altitude   int,      -- (en pied)
    etat       char(2)
);

CREATE TABLE non_capitales (
    nom        text,
    population real,
    altitude   int      -- (en pied)
);

CREATE VIEW villes AS
    SELECT nom, population, altitude FROM capitales
    UNION
    SELECT nom, population, altitude FROM non_capitales;
```

Ceci fonctionne bien pour les requêtes, mais c'est inadéquate lorsque vous avez besoin de mettre à jour plusieurs lignes par exemple.

Voici une meilleure solution :

```
CREATE TABLE villes (
    nom        text,
    population real,
    altitude   int      -- (en pied)
);

CREATE TABLE capitales (
    etat       char(2)
) INHERITS (villes);
```

Dans ce cas, une ligne de `capitales` hérite de toutes les colonnes (`nom`, `population` et `altitude`) de son parent, `villes`. Le type de la colonne `nom` est `text`, un type natif de PostgreSQL pour les chaînes de caractères à longueur variable. Les capitales d'état ont une colonne supplémentaire, `état`, qui affiche leur état. Dans PostgreSQL, une table peut hériter d'aucune ou de plusieurs autres tables.

Par exemple, la requête suivante trouve les noms de toutes les villes, en incluant les capitales des états, situées à une altitude de plus de 500 mètres :

```
SELECT nom, altitude
  FROM villes
 WHERE altitude > 500;
ce qui renvoie :
  nom      | altitude
-----+-----
 Las Vegas |      2174
 Mariposa   |      1953
 Madison    |      845
(3 rows)
```

Autrement, la requête suivante trouve toutes les villes qui ne sont pas des capitales et qui sont situées à une altitude d'au moins 500 mètres :

```
SELECT nom, altitude
  FROM ONLY villes
 WHERE altitude > 500;
  nom      | altitude
-----+-----
 Las Vegas |      2174
 Mariposa   |      1953
(2 rows)
```

Ici, `ONLY` avant `villes` indique que la requête ne doit être lancée que sur la table `villes`, et non pas sur les tables sous `villes` suivant la hiérarchie des héritages. La plupart des commandes dont nous avons déjà discutées (`SELECT`, `UPDATE` et `DELETE`) supportent cette notation (`ONLY`).

**Note :** Bien que l'héritage soit fréquemment utile, il n'a pas été intégré avec les contraintes uniques ou les clés étrangères, ce qui limite leur utilité.

## 2.12. PERSISTANCE

La persistance permet de se passer de l'insertion des éléments dans la base de données, c'est la couche persistante qui s'occupe de charger et décharger les objets suivant les besoins du programme.

### a. Comparatif

Il existe de nombreuses technologies de persistance, dont trois majeures :

**Serialisation de Java :** La serialisation permet de faire persister les objets, mais un graphe sérialisé d'objets ne peut être accédé que comme un tout ; il est impossible de retrouver des données à partir du flux sans dé-serialiser le flux entier.

**CMP :** Les Beans ne sont pas sérialisables et finalement peu portables. Ils ne supportent pas les associations et les requêtes polymorphiques.

**Hibernate** : Il n'a pas tout ces désavantages, c'est d'ailleurs pour cette raison que les EJBs version 3 de Java vont reprendre les concepts d'Hibernate. Jboss intègre d'ailleurs les spécifications EJB 3 avec les outils Hibernate. Les EJB 3.0 n'étant qu'en phase de spécification elles ne seront pas étudiées.

### b. Justification

Plus simple à utiliser que les EJBs 2.0, et plus flexible que la serialisation, hibernate est non intrusif (il n'est pas nécessaire d'hériter d'une classe mère pour que l'objet soit persistant). C'est pour cela que nous l'avons sélectionné.

### c. Tests et exemples

L'utilisation d'hibernate n'est pas des plus simple, mais une fois acquise elle paraît très logique. Cette exemple va se décomposer en deux parties :

- L'intégration d'hibernate dans un projet.
- L'utilisation basique d'hibernate

#### ➤ Intégration dans un projet

Il est à noter que la plupart des configurations par fichier xml, et même les taches Ant sont réalisables à l'exécution en code java.

##### **Configuration**

Tout d'abord, il faut définir le chemin des classes (après avoir créé un nouveau projet) :

- Copier toutes les librairies d'Hibernate 3 ainsi que les librairies tierces. (voir le fichier lib/README.txt dans Hibernate).
- Copier hibernate-annotations.jar et lib/ejb3-persistence.jar à partir de la distribution Hibernate des Annotations dans le chemin des librairies.

Il est recommandé de démarrer Hibernate à partir d'un bloc statique, la plupart du temps on utilise la classe HibernateUtil :

```
package hello;

import org.hibernate.*;
package persistance;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.HibernateException;
import org.hibernate.Session;
```

```

import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

    private static Log log = LogFactory.getLog(HibernateUtil.class);

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory
            sessionFactory = new AnnotationConfiguration()
                .configure()
                //Ajoute le fichier de configuration hibernate
                .addFile("src/hibernate.cfg.xml");

            buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static final ThreadLocal session = new ThreadLocal();

    public static Session currentSession() throws HibernateException {
        Session s = (Session) session.get();
        // Open a new Session, if this Thread has none yet
        if (s == null) {
            s = sessionFactory.openSession();
            session.set(s);
        }
        return s;
    }

    public static void closeSession() throws HibernateException {
        Session s = (Session) session.get();
        session.set(null);
        if (s != null)
            s.close();
    }
}

```

L'autre solution est de mettre les classes dans le fichier hibernate.cfg.xml, voici une équivalence :

```

<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

```

```

<hibernate-configuration>
    <session-factory>
        <mapping package="<paquetage>" />
        <mapping class="<paquetage.class>" />
    </session-factory>
</hibernate-configuration>

```

Ici la classe paquetage.class est ajoutée au système de persistance.

La configuration par le fichier xml est la plus utilisée.

### **Création de la base de données**

Une fois la configuration d'hibernate terminée, il faut créer une tâche ant pour créer la base de données à partir des classes métier java :

```

<project name="eMagine" basedir=".">
    <taskdef
        name="hibernatetool"
        classname="org.hibernate.tool.ant.HibernateToolTask">
        <classpath>
            <fileset dir="lib">
                <include name="**/*.jar"/>
            </fileset>
        </classpath>
    </taskdef>

    <target name="SchemaExport"
        description="Export le schema dans un format compréhensible de
la base de données">

        <hibernatetool destdir="classes" classpath="classes" >
            <annotationconfiguration
configurationfile="src/hibernate.cfg.xml"/>
            <hbm2ddl export="true" console="false" drop="false"
create="true"
                outputfilename="hibernate.ddl" delimiter=";" />
        </hibernatetool>
    </target>
</project>

```

Cette tâche crée le schéma (au format SQL) puis l'insère dans la base de données. Pour que cet outil puisse créer le schéma correspondant aux objets java, il faut spécifier les paramètres nécessaires dans ces objets.

### ➤ **Mapping des objets et Utilisation**

Pour utiliser les objets dans le mécanisme de persistance il faut qu'il implémente **Serializable**.

En outre, ils doivent être ajoutés à la configuration générale (vue ci-avant). Les annotations faisant parties de la nouvelle version d'hibernate qui sera intégrable dans les EJB 3.0, nous allons faire une étude globale.

En plus d'ajouter la classe dans la configuration il faut ajouter l'annotation suivante :

```
//Annotation à ajouter pour qu'une classe soit considérées comme persistance
@Entity
public class Entite {
...
}
```

### Utilisation des identifiants

Les classes hibernate doivent avoir un identifiant quoiqu'il arrive, cette identifiant est de types long, il s'exprime comme suit dans une classe :

```
@Entity
public class Entite {

    //membre privé
    private Long id;
    ...

    //Annotation pour spécifier l'identifiant
    @Id
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    ...
}
```

Les entités EJB3 sont des POJOs (Plain Old Java Object). En fait ils représentent exactement le même concept que les entités persistantes de Hibernate. Leurs mappings sont définis par des annotations de JDK 5.0. Les annotations viennent dans deux catégories, les annotations de mapping logiques (permettant de décrire le modèle d'objet, les associations de classe, etc...) et le mapping des annotations relationnelles (décrivant le schéma, les tables, les colonnes, les index, etc...). Nous mélangerons des annotations des deux catégories dans les exemples de code suivant.

Les annotations sont dans le paquet `javax.persistence.*`.

### Déclaration d'une entité

Chaque classe persistante est une entité et est déclarée en utilisant l'annotation `@Entity` (au niveau de la classe) :

```
@Entity
public class Flight implements Serializable {
    Long id;
```

```

@Id
public Long getId() { return id; }

public setId(Long id) { this.id = id; }
}

```

`@Entity` déclare la classe comme entité (c.-à-d. une classe persistante POJO), `@Id` déclare l'identifiant de cet entité. Les autres déclarations de mappings sont implicites. La classe Flight est mappée sur la table Flight utilisant l'id comme clef primaire.

L'annotation `@Entity` nous permet de définir si une entité devra être consulté par ses méthodes de getters/setters (défaut) ou si le gestionnaire d'entité devra accéder aux champs de l'objet directement :

```

@Entity(access = AccessType.PROPERTY) ou @Entity(access =
AccessType.FIELD)

```

Les spécifications EJB3 exigent de déclarer des annotations sur l'élément qui sera utilisé (méthode gettters/setters ou attribut).

### Définir la table

`@Table` est placé au niveau de la classe. Il permet de définir la table, le catalogue, et les noms de schéma pour le mapping d'entité. Si aucun `@Table` n'est défini les valeurs par défaut sont employées : le nom sans réserve de la classe de l'entité.

```

@Entity(access=AccessType.FIELD)
@Table(name="tbl_sky")
public class Sky implements Serializable {
...

```

Il est également possible de définir des contraintes uniques à la table en utilisant l'annotation `@UniqueConstraint` en même temps que `@Table` (pour une contrainte unique liée à une colonne simple, référez-vous au `@Column`).

### Versionning pour la fermeture optimiste

Il est possible d'ajouter des possibilités de fermeture optimistes à une entité en utilisant l'annotation `@Version` :

```

@Entity()
public class Flight implements Serializable {
...
    @Version
    @Column(name="OPTLOCK")
    public Integer getVersion() { ... }
}

```

La propriété de version sera mappée sur la colonne OPTLOCK, et le gestionnaire d'entité l'emploiera pour détecter les mises à jour conflictuelles.

## ➤ Mapper les propriétés simples

### Déclaration des mapping de base des propriétés

Par défaut, chaque propriété non statique d'une entité est considérée comme persistante, à moins qu'elle soit annotée comme @Transient. L'annotation @Basic permet de déclarer la stratégie de mapping pour une propriété :

```
@Transient  
String getLengthInMeter() { ... }
```

### Déclaration des attributs de colonne

Les colonnes utilisées pour tracer des propriétés peuvent être définies en utilisant l'annotation @Column. A utiliser pour remplacer les valeur par défaut. Cette annotation peut être utilisé dans des champs :

- non annoté du tout
- annoté avec @Basic
- annoté avec le @Version
- annoté avec le @Lob

```
@Entity()  
public class Flight implements Serializable {  
...  
@Column(updatable = false, name = "flight_name", nullable = false, length=50)  
public String getName() { ... }
```

La propriété nommée est mappée à la colonne flight\_name, qui n'est peut pas être NULL, a une longueur de 50 et ne peut pas être mis à jour (rendant la propriété immuable).

Cette annotation peut être appliquée aux propriétés régulières aussi bien que des propriétés de type @Id ou de type @Version.

## HQL

HQL est un langage permettant de manipuler les objets Hibernate.

### HQL et les autres

Il existe plusieurs moyens de récupérer des objets à partir de la base de données :

- Par requête SQL pure.
- Par Critère.

- Par le language HQL.

La librairie Criteria est trop restrictive. Le langage SQL quant à lui, est trop près de la base de données pour être intéressant, bien qu'il permettent de faire quelques actions (rares) de plus que HQL

HQL est un langage qui s'approche de SQL, mais qui est orienté objet :

```
Query query = session.createQuery("from User");
```

Récupère la totalité des éléments de la table User, sous forme d'objet User.

L'équivalent en SQL serait :

```
Query query = session.createQuery(" select * from USERS {u} ", User.class);
```

On voit tout de suite que la HQL est plus claire à manipuler que le SQL

Enfin en criteria cela donnerait :

```
Criteria criteria = session.createCriteria(User.class);
```

Le mode criteria est plus simple, mais ne permet pas de faire autant de chose que les deux langages précédent, par exemple il est impossible de faire des opérations arithmétiques. Il n'y a aucune notion de script et tout se fait par l'appel de fonction.

### **HQL de base**

Maintenant qu'une vue globale des manière de récupérer les données à été présentée, nous n'allons plus utiliser que le HQL.

A présent pour récupérer la liste des objets :

```
List results = session.createQuery(" from User u order by u.name asc ").list()
```

L'exemple suivant permet de ne récupérer qu'une partie des résultats :

```
List results = session.createQuery(" from User u order by u.name asc ")
    .setFirstResult(40)
    .setMaxResults(20)
    .list()
```

### **Utiliser les paramètres nommés :**

```
String queryString = " from Item item where item.description like :searchString";
```

Le signe deux-points suivi par un nom de paramètre indique un paramètre nommée. Ensuite, nous pouvons utiliser l'interface query pour lier une valeur au paramètre searchString :

```
List result = session.createQuery(queryString)
    .setString("searchString", searchString)
    .list();
```

searchString étant une variable chaîne fournie par l'utilisateur, nous utilisons la méthode `setString()`.

## 2.13. LOGGEUR

### a. Comparatif

Il existe deux systèmes de logging actuellement utilisé en java :

- log4J
- JSR47

	log4J	JSR47
Disponibilité	Depuis 1999	Depuis 2001
JDK requis	« 1.1 »	« 1.4 »
Inclus dans la JDK	non	oui
Filtre logique	À la IPChains	booleen

### b. Justification

Les systèmes de filtre proposés par log4j sont plus intéressants que de simples opérations booléennes.

### c. Utilisation

log4j est un framework extensible qui permet de logger et déboguer les applications Java. En plus d'être extrêmement pratique il est gratuit et Open Source. (*Apache Software License 1.1*)

Le framework utilise trois types de composants :

- Logger,
- Appender,
- Layout.

Ces composants communiquent entre eux pour déterminer :

- Quels messages doivent être loggés,
- La façon de logger les messages,
- La présentation des messages.

#### ➤ Logger

Le Logger est le point d'entrée d'un message. C'est pratiquement une des seules classes que nous allons utiliser.

log4j gère les Logger de façon hiérarchique. C'est à dire qu'un Logger peut avoir des enfants et des parents. La hiérarchie est gérée par le nom des Logger. Les niveaux sont définis par des points, un peu comme la structure des packages de classes.

Par exemple :

- com
- com.liguo
- com.liguo.layout

*com* est le parent de *com.liguo* et l'ancêtre de *com.liguo.layout*.

log4j possède par défaut un Logger à la racine de la hiérarchie. Il se nomme **rootLogger**.

Lorsque l'on demande à log4j de créer un Logger, il vérifie dans la configuration s'il y a quelque chose de défini pour celui-ci. S'il ne trouve rien, il remonte dans la hiérarchie jusqu'à ce qu'il trouve quelque chose. C'est pourquoi le rootLogger qui est à la racine est très important.

## ➤ Appender

Appender est une interface dont les deux rôles sont de :

- Filtrer les messages,
- Déterminer le mode de sortie des messages selon l'implémentation.

Un logger peut avoir un nombre X d'appender. Pour ajouter un appender manuellement, il suffit de faire : monLogger.addAppender(monAppender);

**Note** : Un Logger utilise les appender de ses ancêtres. Donc les appender définis pour le *rootLogger* seront utilisés par **tous** les Logger, peu importe leur niveau.

log4j fournit de base les appender utilisés pour les tâches courantes :

- **ConsoleAppender** : envoie les messages dans la console du système (cf. Tomcat dans le cadre d'une application web),
- **FileAppender** : envoie les messages dans un fichier,
- **SocketAppender** : envoie les messages dans un Socket,
- **SMTPAppender** : envoie les messages par mail.

Il en existe d'autres pour les utilisateurs plus avancés et il est évidemment possible de créer son propre appender custom en implémentant l'interface org.apache.log4j.Appender.

## ➤ Layout

Les Layout définissent le format du message. Il est possible, par exemple de récupérer la date, l'heure, le nom de la classe qui envoie le message et le niveau de logging au début de chaque message. C'est le rôle du Layout de faire ça et non celui du développeur qui envoie le message.

## ➤ Niveau

Lorsqu'on envoie un message à un logger, il faut spécifier un niveau. Par défaut log4j offre les niveaux suivants :

- DEBUG,
- INFO,
- WARN,
- ERROR,
- FATAL.

Les niveaux sont gérés par ordre de priorité, DEBUG étant le moins important.

Par exemple, durant la période de développement, on utilise le niveau DEBUG. Une fois que l'application est déployée en production, on change le niveau pour WARN afin d'**ignorer** les messages de débogage, **sans modifier une seule ligne dans notre code**.

**Note** : Il est possible de créer ses propres niveaux.

## ➤ Installation

Il faut tout d'abord aller sur le site de la fondation Apache pour télécharger log4j :

<http://logging.apache.org/log4j/docs/download.html>

Télécharger la dernière version stable. Ensuite extraire l'archive dans n'importe quel répertoire de votre disque dur.

L'archive devrait avoir la structure de répertoires suivante :

- logging-log4j-x.x.x
  - build
  - contribs
  - dist
  - docs
  - exemples
  - src

Le répertoire **dist** devrait contenir un sous-répertoire **lib**. Ce dernier devrait contenir le fichier **log4j-x.x.x.jar** (remplacez x.x.x par le numéro de version)

Pour installer log4j, il suffit d'inclure le fichier **log4j-x.x.x.jar** dans le classpath. Dans le cas d'une application web avec Tomcat, c'est dans le répertoire **/WEB-INF/lib**.

L'installation est terminée.

## ➤ Configuration

Il existe plusieurs façons de configurer log4j. Celle que nous allons utiliser est assez simple et souple. Nous allons mettre ça dans un fichier de propriétés externe. Donc pas besoin de recompiler notre code pour modifier la configuration.

Tout d'abord, il faut créer un fichier nommé **log4j.properties** et le sauvegarder dans le classpath de l'application. Dans le cas d'une application web avec Tomcat, c'est dans le répertoire **/WEB-INF/classes**

Voici le contenu de notre fichier : (les lignes avec un # sont des commentaires)

```
#définition du niveau et des Appender du rootLogger
log4j.rootLogger=DEBUG, monAppender
#configuration de "monAppender"
#nous allons envoyer les messages dans la console de Tomcat
log4j.appender.monAppender=org.apache.log4j.ConsoleAppender

#définition du Layout pour "monAppender"
log4j.appender.monAppender.layout=org.apache.log4j.PatternLayout

#définition du pattern d'affichage pour "monAppender"
#voici un exemple de sortie que l'on va obtenir : 2005-06-18 14:53:37 DEBUG
[Main] Hello World
log4j.appender.monAppender.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss}
%-5p [%c{1}] %m%n
```

Pour modifier le niveau de logging des classes se trouvant dans le package com.liguo.layout, il faut simplement ajouter la ligne suivante dans le fichier de configuration :

```
log4j.logger.com.liguo.layout=WARN
```

## ➤ Utilisation

Le logger va être utilisé dans une classe représentant la gestion d'un service :

```
import org.apache.log4j.Logger;

public class HelloService {

    /**
     *déclaration et création du Logger
     *on le met statique afin d'économiser la mémoire
     */
    private final static Logger logger = Logger.getLogger
(HelloService.class);

    public String helloWorld() {
        logger.debug("Le service helloWorld est invoqué");
        return "Hello, world";
    }
}
```

Si jamais la construction du message est lourde (transformation d'un DOM XML en String, concanénation de String, conversion de valeur numérique en String, etc..), il suffit d'utiliser la méthode `isDebugEnabled()` afin de vérifier si le logger ignorera ou non ce message.

```
if(logger.isDebugEnabled()){
    logger.debug("Affiche de la section #" + sectionId);
}
```

## ➤ Conclusion

Tout ceci est suffisant pour utiliser log4j, mais ce n'est qu'un bref aperçu de tout ce qu'il peut faire puisque c'est un framework extensible.

**Exemple simple :** On pourrait décider d'envoyer un mail aux développeurs pour les messages de niveau ERROR et FATAL qui sont lancés dans une application qui est en production. De cette façon, on est toujours au courant de l'état de notre application sur le serveur sans avoir à parcourir constamment le fichier de logs.

## 2.14. GESTION DE VERSION

### a. Comparatif

Ce tableau comparatif montre les différentes fonctions des différents systèmes de gestion de source :

	Atomique	Renommage, suppression	Copie	RéPLICATION du dépôt	Propagation des changement	Permission sur le dépôt	Patch
CVS	Non	Non	Non	Non	Non	Oui	Non
Aegis	Oui	Oui	Non	Oui	Oui	Oui	Oui
Arch	Oui	Oui	Non	Oui	Oui	Oui	Oui
BitKeeper	Oui	Oui	Oui	Oui	Oui	Oui	Oui
ClearCase	Oui	Oui	Oui	Non	Oui	Oui	Non
CMSynergy	Oui	Oui	Oui	Oui	Oui	Non	Oui
Co-Op	Oui	Oui	Non	Oui	Oui	Oui	Oui
Darcs	Oui	Oui	Non	Oui	Oui	Non	Oui
Monotone	Oui	Oui	Oui	Oui	Oui	Oui	Oui
OpenCM	Oui	Oui	Non	Non	Non	Oui	Oui
Perforce	Oui	Non	Oui	Oui	Oui	Oui	Oui
PureCM	Oui	Oui	Oui	Non	Oui	Oui	Oui
Subversion	Oui	Oui	Oui	Oui	Oui	Oui	Non
Superversion	Oui	Oui	Non	Oui	Non	Non	Non
svk	Oui	Oui	Oui	Oui	Oui	Oui	Non
Vesta	Oui	Oui	Oui	Oui	Oui	Oui	Non
Visual SourceSafe	Non	Non	Oui	Non	Non	Oui	Non

## b. Justification

Il existe un grand nombre de logiciels du même type. Le plus connu d'entre eux et le plus répandu actuellement est sans doute `cvs`. Le tableau ci-dessus montre la liste des utilitaires.

On pourra justifier rapidement le choix de `Subversion` par les arguments suivants :

- Il est multi-plate forme,
- Il s'agit d'un logiciel libre,
- Il fonctionne de manière centralisée,
- Son utilisation et son administration sont plus faciles que `cvs`,
- Il supporte plusieurs modes d'accès distants, dont `SSH et WebDAV via Apache`,
- Il historise l'architecture des répertoires (ce que `cvs` ne fait pas).

L'utilisation de `subversion` sans client graphique est très désagréable, un client graphique nommée SubClipse, qui s'intègre parfaitement à `eclipse` va donc être utilisé. SubEclipse est le seul plugin disponible pour `eclipse`. Il permet de faire toutes les opérations sur un repository `SubVersion`. Il s'intègre parfaitement à l'environnement de développement `eclipse`. L'étude va porter sur `subVersion`, en effet SubClipse n'est qu'une interface graphique de `subversion`. Les clients graphiques ne permettent pas de faire plus, mais proposent des interfaces plus élaborées que la ligne de commande. Ils permettent notamment de naviguer dans le dépôt comme dans un explorateur de fichiers, d'afficher les informations de manière plus structurée, de garder un historique des logs saisis, etc.

Un autre client graphique réservé aux utilisateurs windows est aussi très utilisé : `TortoiseSVN`, il s'ajoute directement au menu contextuel de l'explorateur de fichiers. Les commandes sont les mêmes que celles décrites dans ce document, mais elles sont lancées par un menu et bénéficient d'interfaces graphiques plus conviviales. Il sera donc aussi utilisé pour la gestion des sources en dehors de l'IDE `Eclipse`.

## c. Concept

### ➤ Utilisation

#### **Notions générales**

##### Dépôt (repository)

Un dépôt `Subversion` est l'emplacement central où sont stockées toutes les données relatives aux projets gérés. Le dépôt est accédé via une URL locale ou distante. Il contient l'historique des versions des fichiers stockés, les logs enregistrés lors des modifications, les dates et auteurs de ces modifications, etc.

Un dépôt apparaît de l'extérieur comme un système de fichiers composé de répertoires au sein desquels on peut naviguer, lire et écrire selon les permissions accordées.

### projets

Au sein d'un dépôt se trouvent un ou plusieurs projets. À chaque projet correspond en général un répertoire situé à la racine du dépôt et qui contient lui-même les fichiers et dossiers du projet proprement dit. **Exemple d'arborescence :**

```
(dépôt)---+/batchxsl---+/trunk
          |
          |
          +--/branches
          |
          +--/tags

          +--/css-----+/trunk
          |
          |
          +--/branches
          |
          +--/tags

          +--/test-----+/rep1
          |
          +--/rep2
```

### Copie de travail (working copy)

La copie de travail est un répertoire situé en local sur le poste de l'utilisateur et qui contient une copie d'une révision donnée des fichiers du dépôt. C'est cette copie qui sert de base de travail et qui est modifiée en local avant d'être importée (sauvegardée) vers le dépôt.

### Révisions

Chaque modification faite au dépôt constitue une révision. Le numéro de révision commence à 1 et augmente incrémenté à chaque opération. Sa valeur n'a aucune importance, mais c'est un indicateur qui permet de revenir à une version donnée d'un ou plusieurs fichiers.

## 2.2 Opérations

### Checkout

Le *checkout* est l'opération qui consiste à récupérer pour la première fois les fichiers déjà existants au sein d'un projet du dépôt. Cette opération ne se fait, en général, qu'une fois par projet.

Le résultat est une copie de travail.

### Import

L'*import* est l'opération inverse du *checkout*. Il consiste à placer dans le dépôt des fichiers locaux déjà existants pour y créer un nouveau projet. Cette opération ne se fait en général qu'une fois par projet.

## Update

L'*update* consiste à synchroniser la copie de travail locale avec le dépôt en récupérant la dernière version des fichiers du dépôt.

C'est à cette occasion que des conflits de version peuvent apparaître.

## Commit

Un *commit* est l'opération inverse d'un *update*. Elle consiste à mettre à jour le dépôt à partir de la copie de travail locale. Une nouvelle révision est alors créée. Un log (simple message texte contenant une description des modifications effectuées) doit être saisi à cette occasion.

À noter que pour qu'un *commit* soit possible, il faut que la copie de travail corresponde à la dernière version du dépôt (modifications locales exceptées). Si ce n'est pas le cas, il est nécessaire d'effectuer d'abord un *update* et de résoudre les conflits éventuels avant de réessayer le *commit*.

## **Création d'un nouveau projet**

La première chose à faire lors de la première utilisation est de créer un nouveau projet. Deux cas de figure peuvent se présenter : ou bien le projet existe déjà au sein d'un dépôt et il s'agit de récupérer ce projet en local pour en faire une copie de travail, ou bien ce projet existe en local et doit être importé au sein du dépôt.

### Projet déjà existant au sein d'un dépôt

Si le projet existe déjà au sein du dépôt, une seule commande suffit pour effectuer un *checkout* et récupérer la dernière version des fichiers : il s'agit de la commande `svn co`.

```
$ svn co https://cens-srv-devel.ens-lsh.fr/svnrep/formationsvn .
A credits.htm
A index.htm
Checked out revision 36.
```

La commande précédente a effectué un *checkout* du projet `formationsvn` (situé dans un répertoire racine du même nom) dans le répertoire courant.

Le résultat de la commande indique que deux fichiers ont été récupérés et que la dernière révision est la révision numéro 36.

### Import d'un projet déjà existant en local

Si le projet n'existe pas dans le dépôt et qu'il faut le créer à partir de fichiers locaux, la commande à utiliser est `svn import`. Cette opération n'est en théorie effectuée que par la personne chargée de l'administration du dépôt. Voir la documentation officielle pour plus d'informations.

### **Récupération de la dernière version du projet**

Avant de travailler sur les fichiers du projet, il faut s'assurer que l'on est bien synchronisé avec le dépôt, c'est à dire que la copie de travail correspond bien à la dernière révision en cours. Pour cela, il faut effectuer un *update* à l'aide de la commande `svn update` :

```
$ svn update  
U index.htm  
Updated to revision 37.
```

La commande indique qu'un fichier du répertoire, vraisemblablement modifié par quelqu'un d'autre depuis notre dernier *update*, a été mis à jour dans notre copie de travail.

### **Mise à jour des modifications dans le dépôt**

Une fois qu'on a modifié des fichiers, il faut basculer ces modifications au sein du dépôt pour qu'elles soient accessibles aux autres utilisateurs. Cette opération s'effectue à l'aide de la commande `svn commit` :

```
$ svn commit -m "Ajout d'une personne dans les credits"  
Sending      credits.htm  
Transmitting file data .  
Committed revision 38.
```

Toute opération de *commit* s'effectue en indiquant un message décrivant les modifications effectuées (ici directement dans la ligne de commande). Il est possible d'effectuer cette opération sur un répertoire entier, ou sur seulement un ou plusieurs fichiers.

Si des modifications ont eu lieu par un autre utilisateur du dépôt depuis le dernier *update*, un message d'erreur le signale. Il faut alors effectuer un nouvel *update* et résoudre d'éventuels conflits avant de relancer le *commit*.

### **Récupération d'une version antérieure d'un fichier**

#### **Récupération de la dernière version**

Lorsqu'on travaille sur un fichier, il peut arriver que les modifications effectuées ne soient pas bonnes et qu'on souhaite revenir à la version antérieure du fichier (tel qu'il était lors du dernier *update*).

La commande `svn revert` est faite pour ça :

```
$ svn revert credits.htm  
Reverted 'credits.htm'
```

Cette commande annule les modifications effectuées depuis le dernier *update*. À noter que tout est effectué en local, et qu'un accès au dépôt n'est pas nécessaire.

#### **Récupération d'une version antérieure du dépôt**

On peut aussi souhaiter revenir à une version antérieure d'un fichier situé dans le dépôt. Il faut alors utiliser `svn update` en précisant le numéro de la révision et le ou les fichiers :

```
$ svn update -r 36 credits.htm  
U credits.htm  
Updated to revision 36.
```

### **Gestion des fichiers du dépôt**

Subversion propose un ensemble de commandes pour ajouter, supprimer ou renommer des fichiers du dépôt.

## Ajout d'un fichier

Il faut utiliser `svn add`. A noter que l'ajout n'est effectif qu'au prochain *commit* :

```
$ svn add liens.htm
A          liens.htm

$ svn commit -m "Ajout du fichier de liens"
Adding      liens.htm
Transmitting file data .
Committed revision 39.
```

## Suppression d'un fichier

Il faut utiliser `svn delete` :

```
$ svn delete liens.htm
D          liens.htm

$ svn commit -m "Suppression d'un fichier"
Deleting    liens.htm
Committed revision 40.
```

Là aussi, la suppression n'est effective qu'au *commit* suivant.

## Renommer un fichier

Il faut utiliser `svn move` :

```
$ svn move credits.htm merci.htm
A          merci.htm
D          credits.htm

$ svn commit -m "Renommage d'un fichier"
Deleting    credits.htm
Adding      merci.htm
Committed revision 41.
```

## Résolution des conflits

Les conflits peuvent intervenir au moment d'un *update*, lorsque des modifications ont été faites à la fois dans la copie de travail et dans le dépôt. Par exemple, si un utilisateur a édité un fichier en local pour lui rajouter une ligne, et qu'un autre utilisateur du dépôt a effectué un « *commit* » entre temps, le *commit* effectuée par le premier utilisateur va générer l'erreur suivante :

```
$ svn commit
Sending      merci.htm
svn: Commit failed (details follow):
svn: Your file or directory 'merci.htm' is probably out-of-date
svn:
The version resource does not correspond to the resource within the
transaction. Either the requested version resource is out of date
(needs to be updated), or the requested version resource is newer than
the transaction root (restart the commit).
```

Il vous faut alors effectuer un *update*, ce qui va mettre en concurrence les deux versions du ou des fichiers concernés. Deux cas de figure peuvent alors se présenter :

Dans le premier cas, le conflit peut être résolu automatiquement par Subversion car les modifications ne concernent pas les mêmes parties du fichier. Dans ce cas vous obtiendrez le message suivant :

```
$ svn update  
G merci.htm  
Updated to revision 42.
```

Il est quand même conseillé de vérifier manuellement le résultat de cette résolution « automatique ».

Dans le deuxième cas, les modifications ne peuvent être fusionnées automatiquement car elles concernent les mêmes parties d'un fichier. Dans ce cas un conflit est signalé lors de l'*update* :

```
$ svn update  
C merci.htm  
Updated to revision 43.
```

Dans ce cas, deux nouveaux fichiers font leur apparition dans votre copie de travail. Dans l'exemple précédent, on se retrouve avec :

- *merci.htm.mine* : copie du fichier tel qu'il se trouvait dans votre copie de travail, en local, avant de faire l'*update*. C'est la version que vous souhaitez « commiter » avant de détecter un conflit ;
- *merci.htm.r42* : version du fichier pour la révision 42, c'est à dire lors de votre dernier *update*. C'est la version qui a servi de base pour les deux utilisateurs du dépôt qui ont travaillé en parallèle ;
- *merci.htm.r43* : version du fichier pour la révision 43, c'est à dire la version actuellement dans le dépôt. Il s'agit de la version modifiée par un autre utilisateur, « commitée » avant votre *update*, et dont le contenu est à l'origine du conflit.
- *merci.htm* : il s'agit d'une version qui, en quelque sorte « résume » les trois autres en faisant apparaître les différences entre versions au sein d'un seul fichier.

Dès lors, le travail consiste à éditer le fichier *merci.htm* jusqu'à ce que le conflit soit résolu. Une fois ce travail terminé, on signale que le conflit est résolu à l'aide de la commande *svn resolved* :

```
$ svn resolved merci.htm  
Resolved conflicted state of 'merci.htm'
```

On peut alors effectuer le *commit* final.

## **Tronc, branches, tags...**

Les notions de tronc, de branches et de *tags* sont assez spécifiques aux logiciels de contrôle de versions. C'est ce qui explique que les arborescences des répertoires de projet contiennent souvent comme premier niveau de sous-répertoires les dossiers `trunk`, `branches` et `tags`.

En général, on définit par « tronc » la version centrale du programme, le développement principal « officiel ».

Une « branche » est en général créée lorsqu'un développement « secondaire » est mis en route, que ce soit pour ajouter une nouvelle fonctionnalité ou parce que certains développeurs souhaitent essayer de prendre une autre direction pour certains aspects du développement. Une branche peut, au bout d'un certain temps, soit être à nouveau fusionnée dans le « tronc », soit disparaître, soit donner lieu à un nouveau programme.

La notion de *tags* correspond en partie à celle de *release*, c'est à dire de marquage d'une certaine révision du projet comme composant une version du projet. Une fois que le développement a atteint une certaine stabilité, on pourra par exemple créer un *tag* pour marquer la sortie de la version 1.0. Ceci permettra de revenir facilement à cette version, indépendamment du numéro de révision sous-jacent correspondant.

Nous n'entrerons pas dans le détail de ces concepts et commandes ici, mais on peut juste citer que la création de branches ou de *tags* ne sont en fait que des copies créées par la commande `svn copy`. La commande `svn switch`, elle, permet de faire passer la copie de travail d'une branche à une autre.

## **Aide intégrée**

Une aide est intégrée à l'interface en ligne de commande. Pour obtenir de l'aide générale, et notamment la liste des commandes possibles, il suffit de faire :

```
svn help
```

Pour obtenir de l'aide sur une commande particulière, il faut utiliser :

```
svn help <nom de la commande>
```

## **Informations sur la copie de travail**

Pour obtenir des informations sur la copie de travail en cours, on peut utiliser `svn info` :

```
$ svn info
Path: .
URL: https://cens-srv-dev1.ens-lsh.fr/svnrep/formationsvn
Repository UUID: 090fa6ab-88f7-0310-b83e-cd111ae4905a
Revision: 40
Node Kind: directory
Schedule: normal
Last Changed Author: jbarnier
Last Changed Rev: 40
Last Changed Date: 2005-05-30 14:58:11 +0200 (lun, 30 mai 2005)
```

## Voir l'historique des modifications d'un fichier ou projet

La commande `svn log` permet d'afficher l'historique de toutes les modifications d'un fichier donné en paramètre ou d'un projet entier :

```
$ svn log index.htm
-----
r37 | jbarnier | 2005-05-30 14:42:26 +0200 (lun, 30 mai 2005) | 2 lines
Modification du titre de la page
-----
r36 | jbarnier | 2005-05-30 14:34:05 +0200 (lun, 30 mai 2005) | 2 lines
Ajout de deux fichiers de test.
```

La commande `svn blame` permet d'obtenir des informations sur un fichier ligne par ligne, avec la révision et l'auteur correspondants :

```
$ svn blame index.htm
36    jbarnier <html>
36    jbarnier <head></head>
36    jbarnier <body>
37    jbarnier <h1>Un bien beau titre, vraiment</h1>
36    jbarnier </body>
36    jbarnier </html>
```

## Voir le statut de la copie de travail

La commande `svn status` permet d'avoir des informations sur l'état de la copie de travail depuis le dernier *update* :

```
$ svn status -v
?                               credits.htm
        40      40 jbarnier   .
        41      41 jbarnier   merci.htm
        40      37 jbarnier   index.htm
```

## Parcourir le dépôt

La commande `svn list` permet d'afficher le contenu du dépôt à distance :

```
$ svn list https://cens-srv-dev1.ens-lsh.fr/svnrep/formationsvn
index.htm
merci.htm
```

## Utiliser les propriétés

Les propriétés sont des attributs attachés à un ou plusieurs fichiers du dépôt et qui permettent des comportements particuliers.

Sans entrer dans le détail, on ne citera que deux types de propriétés. Tout d'abord, la propriété

`svn:ignore` permet de retirer explicitement certains fichiers du contrôle de version. Par exemple, si votre éditeur de texte enregistre systématiquement une copie de sauvegarde de vos fichiers avec l'extension `.bak`, il peut être intéressant de systématiquement mettre de côté ces fichiers en positionnant la propriété correspondante :

```
$ svn propset svn:ignore *.bak .
property 'svn:ignore' set on '..'
```

Une autre utilisation intéressante concerne l'utilisation de mots-clés. Ceux-ci sont des identifiants insérés dans les fichiers du projet et qui seront remplacés au moment du *commit* par des informations propres à Subversion, comme le nom du fichier, le numéro de révision, l'auteur et la date de la dernière modification, etc.

Par exemple, si l'utilisateur insère la chaîne `$Id$` dans son fichier, celle-ci sera automatiquement remplacée par un résumé de ces informations. Voici la valeur correspondant au fichier source de ce document :

```
$Id: formation_svn.tex 135 2005-08-02 09:47:51Z julien $
```

Pour que ces mots-clés fonctionnent, il faut positionner la propriété `svn:keywords` de manière adéquate pour les fichiers concernés :

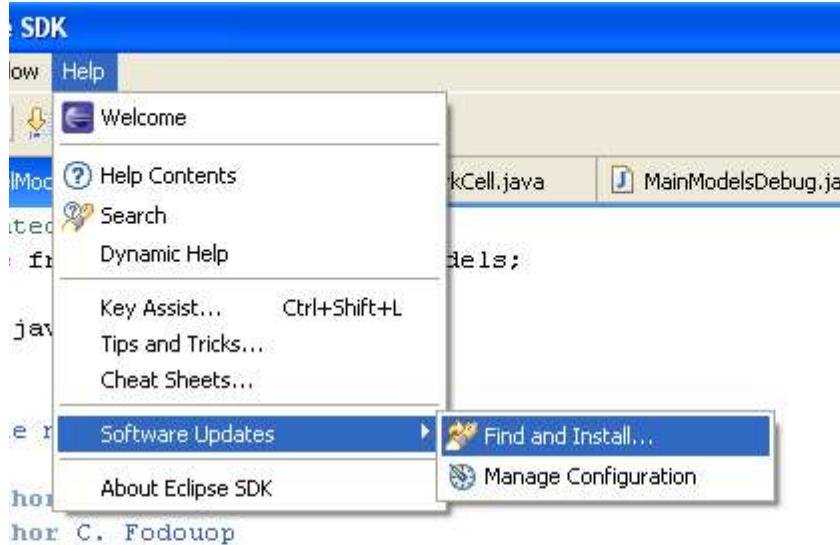
```
$ svn propset svn:keywords "Id" index.htm
property 'svn:keywords' set on 'index.htm'
```

#### d. Tests et exemples

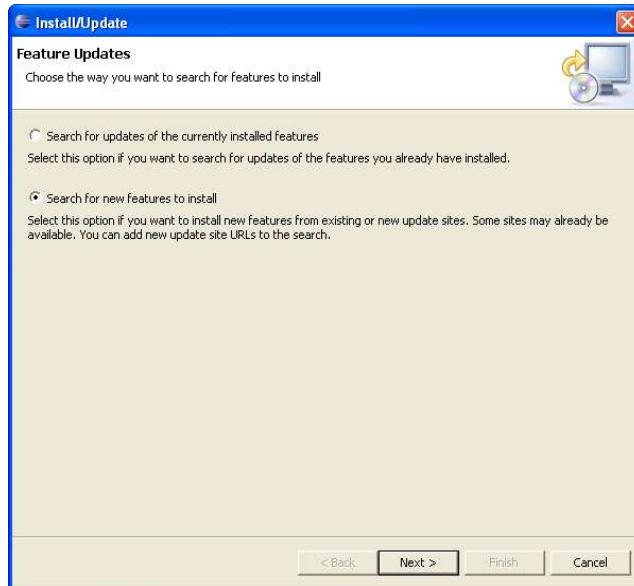
### ➤ Installation du module subversion

Démarrer Eclipse.

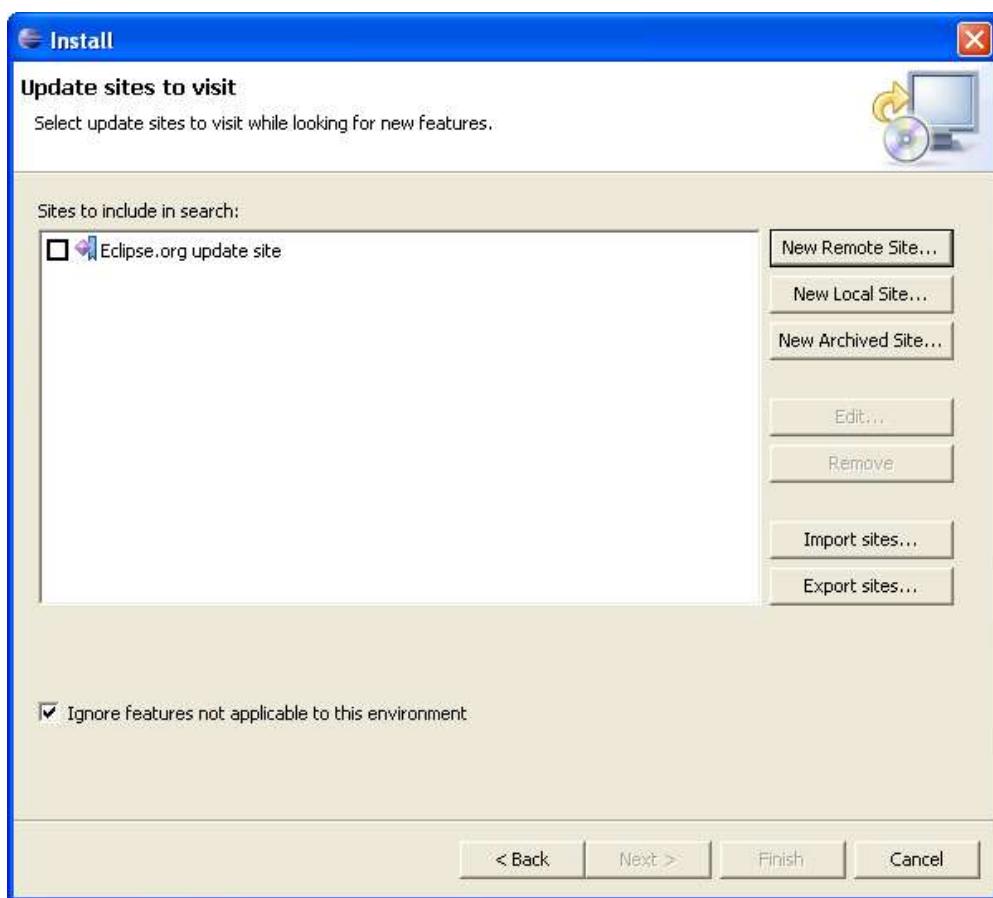
Sélectionner Help=>Software Updates=>Find and Install...



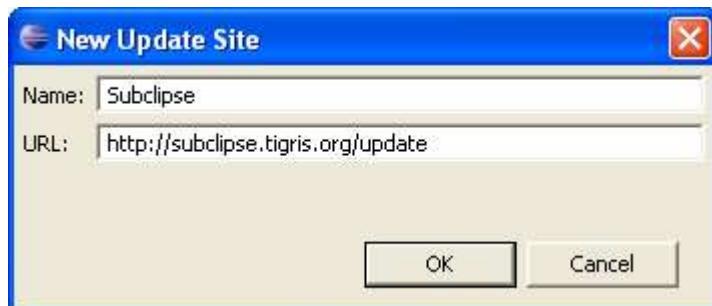
La fenêtre suivante s'ouvre :



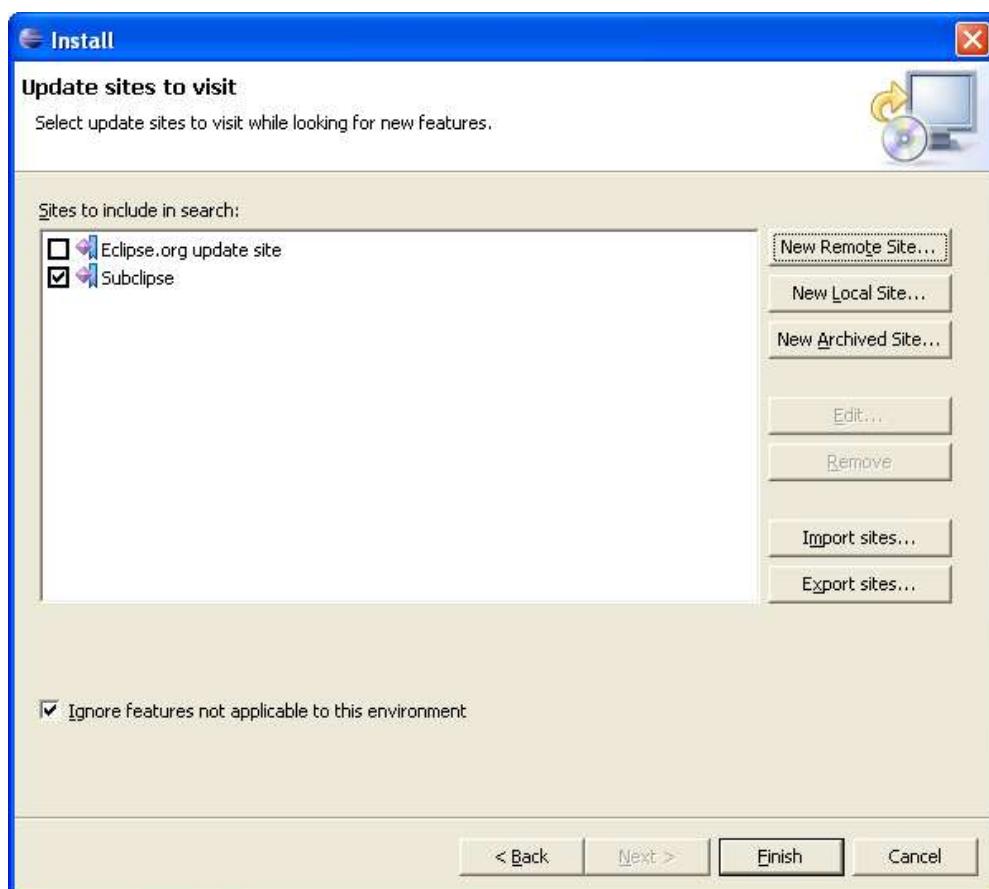
Sélectionner 'Search for new features to install'



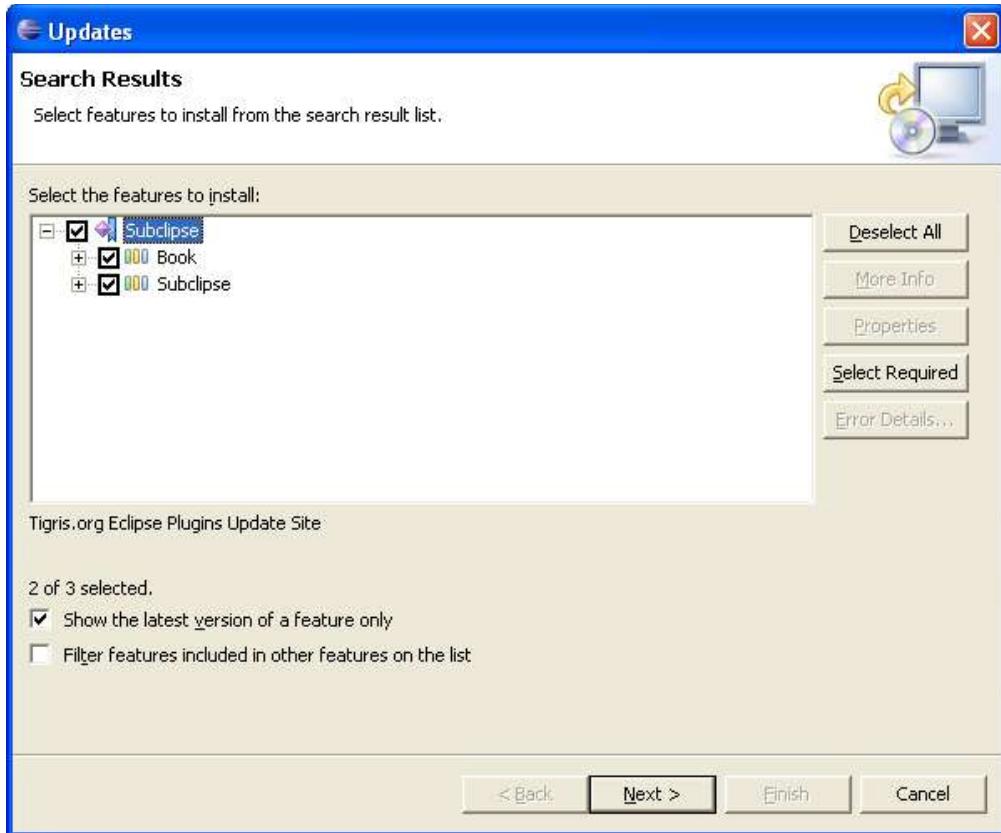
Cliquer sur le bouton 'New Remote Site....'



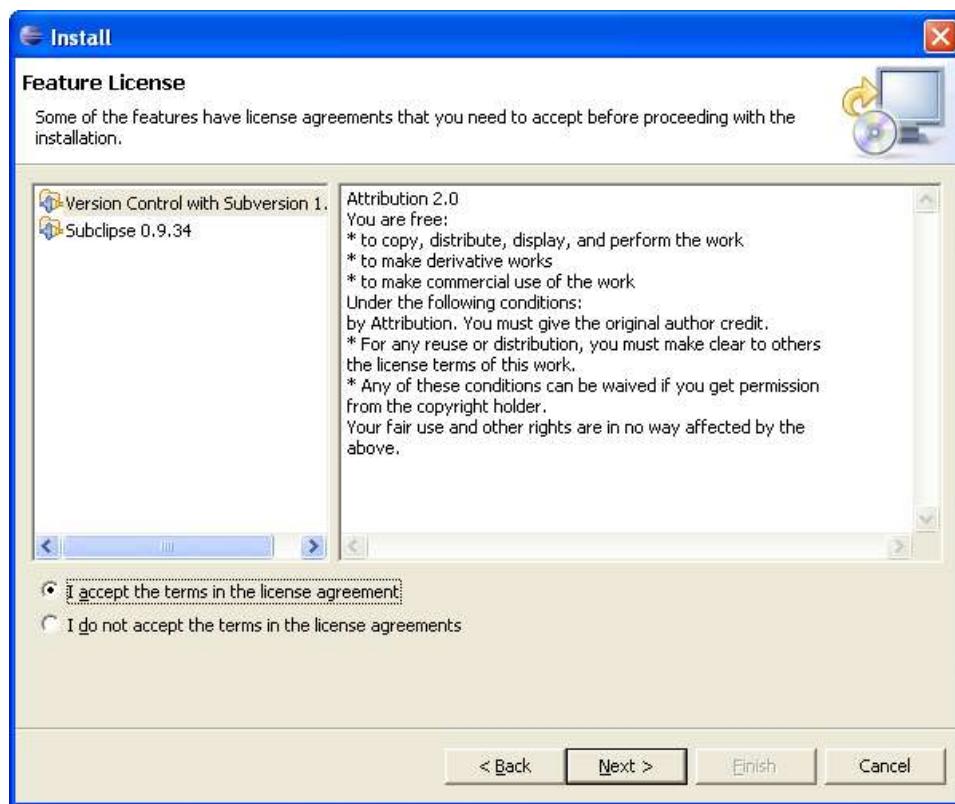
Compléter les champs comme ci-dessus



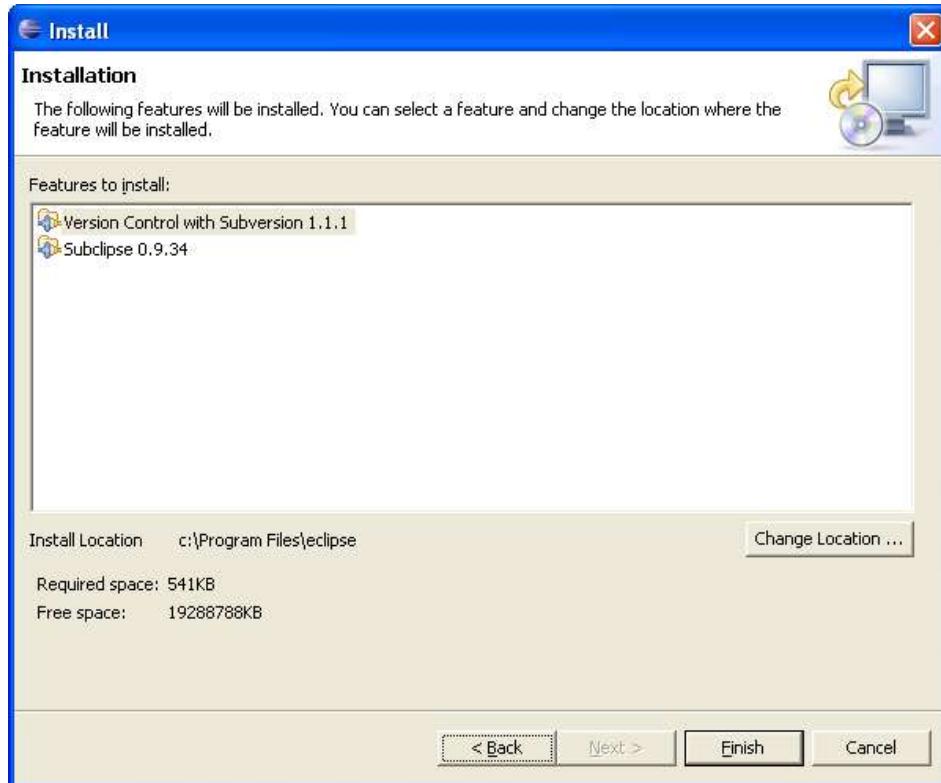
Cocher la case 'Subclipse' puis cliquer sur finish



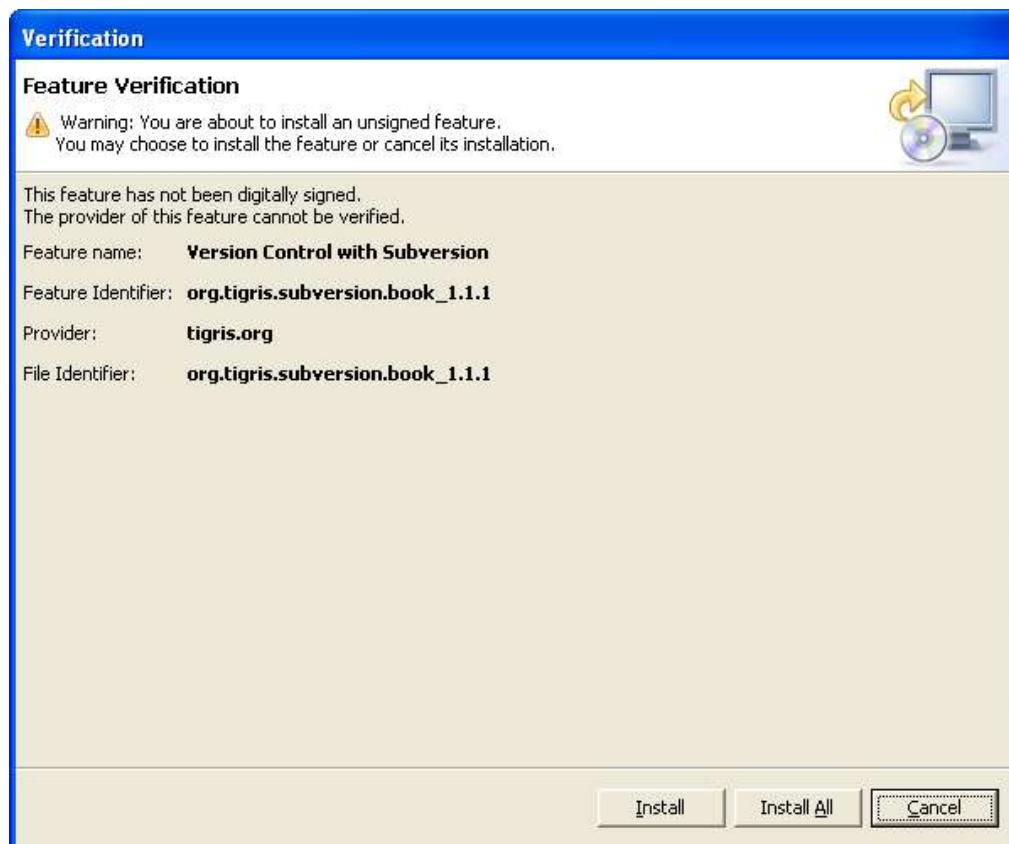
Cocher les trois cases puis cliquer sur next



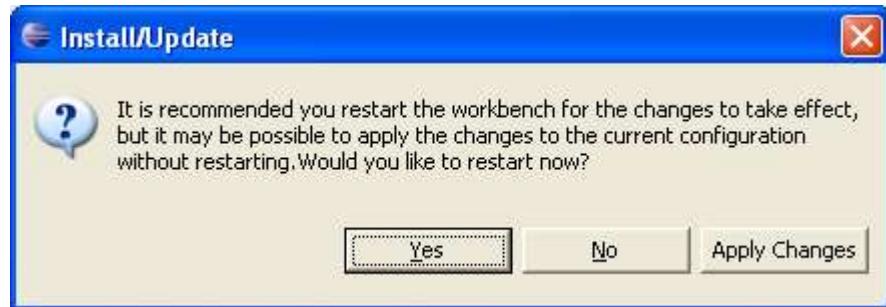
Selectionner 'I accept the terms in the licence agreement' puis cliquer sur 'next'.



Cliquer sur 'Finish'

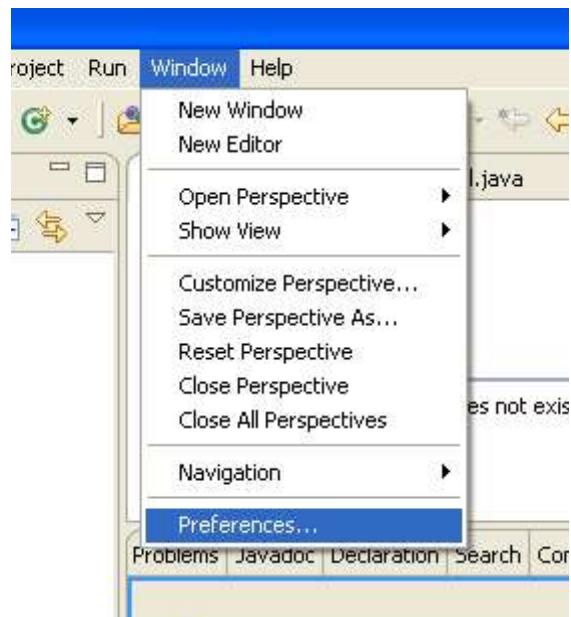


Cliquer sur Install All.

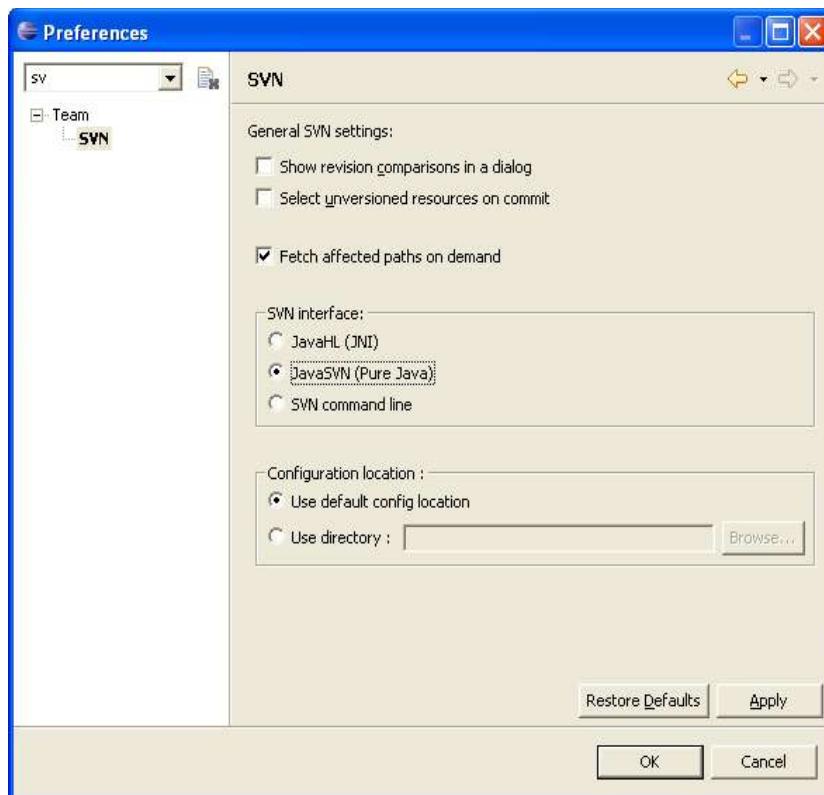


Cliquer sur 'yes'

➤ Configuration de Subversion



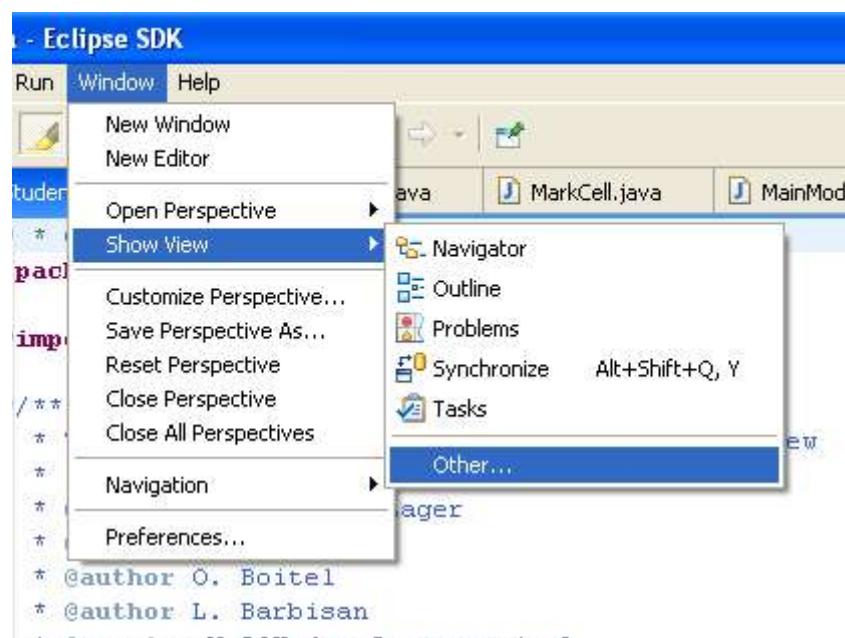
Aller dans le menu Windows=>Préférences puis sélectionner dans l'arborescence Team/SVN :



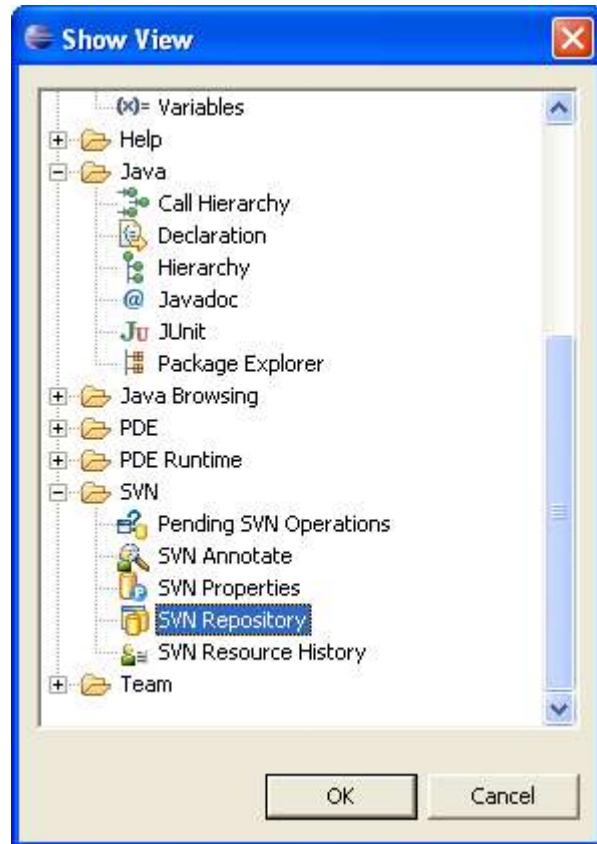
Dans la Frame 'SVN interface : ' Selectionner 'JavaSVN (Pure Java)'.

#### ➤ Récupération du repository

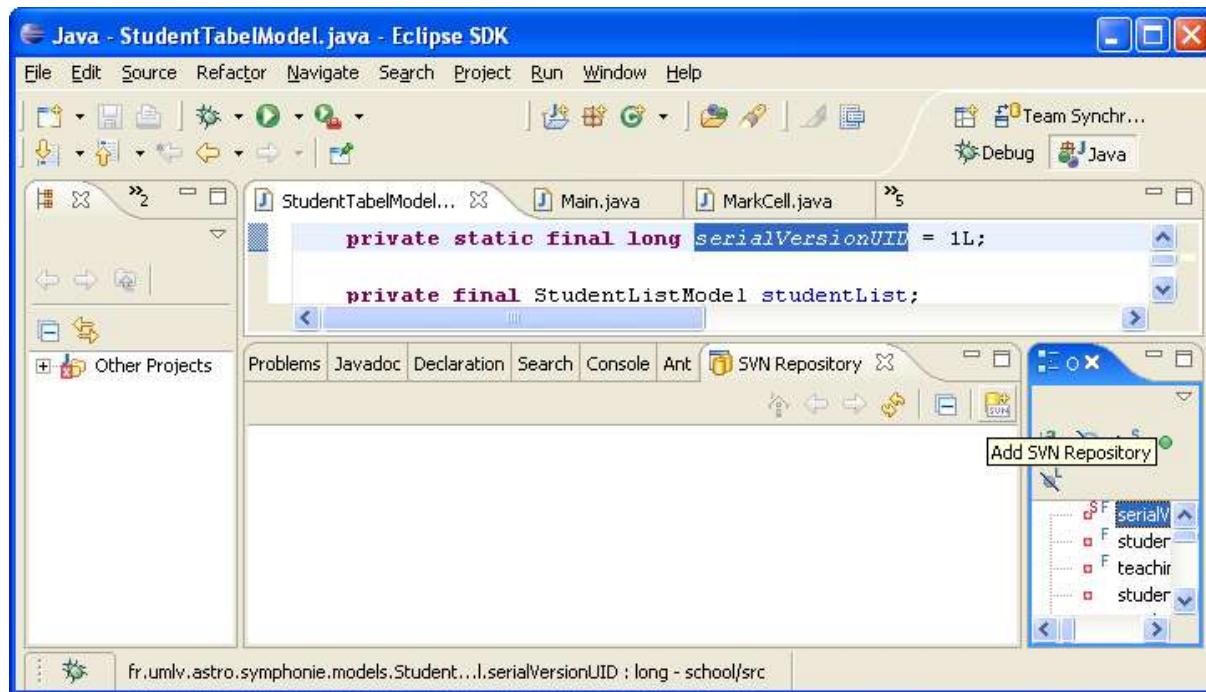
Ouvrir la vue SubVersion à l'aide du menu Window>Show View/Other...



Selectionner SVN Repository :

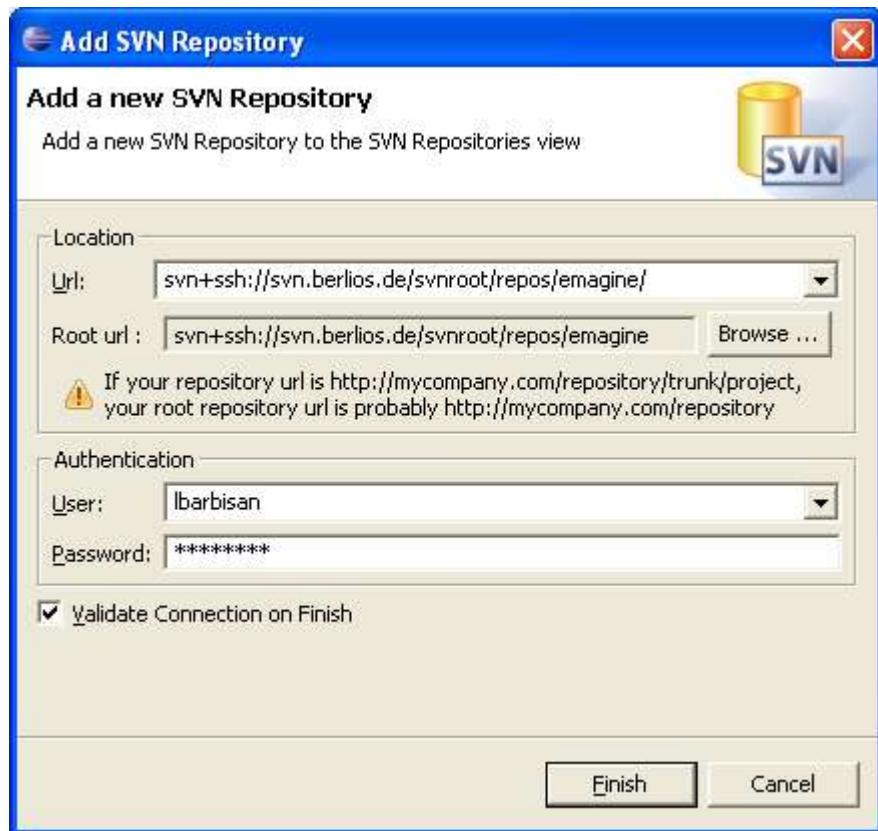


Cliquer sur 'OK'

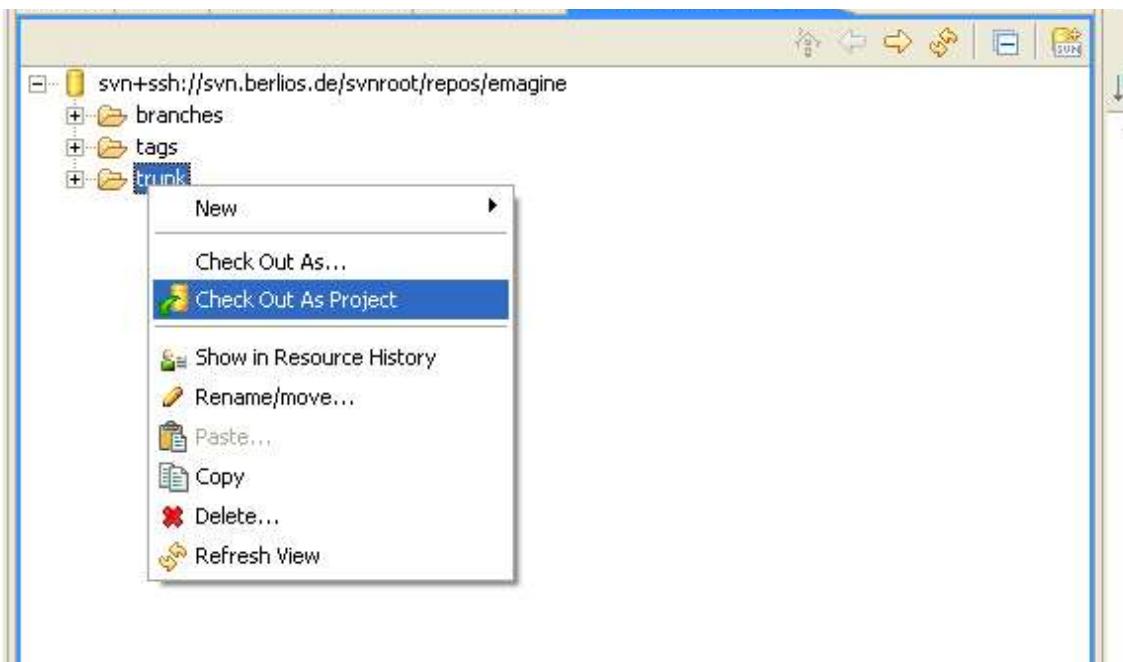


Dans le fenêtre 'SVN Repository'

Selectionner 'Add SVN Repository' (bouton le plus à droite)

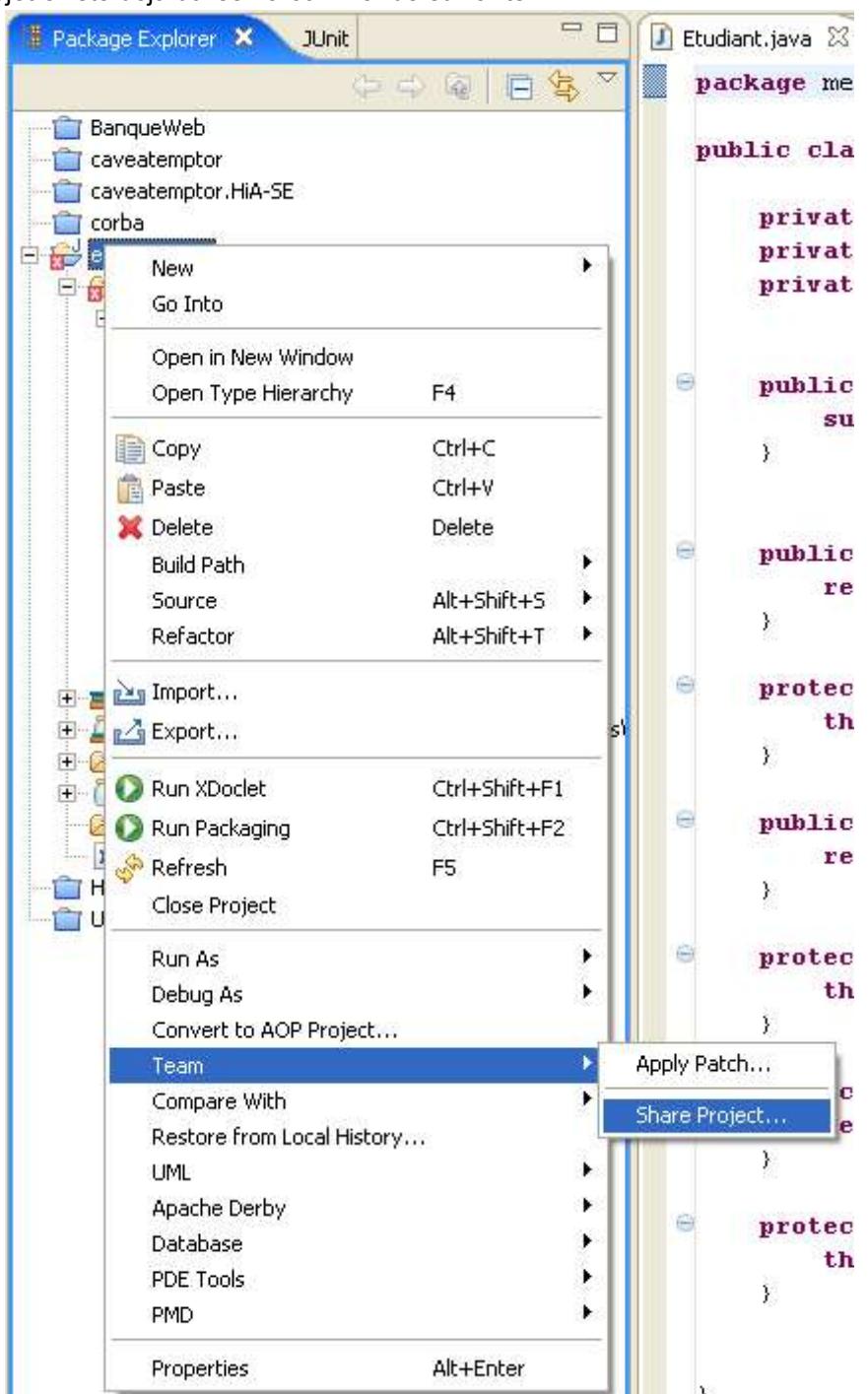


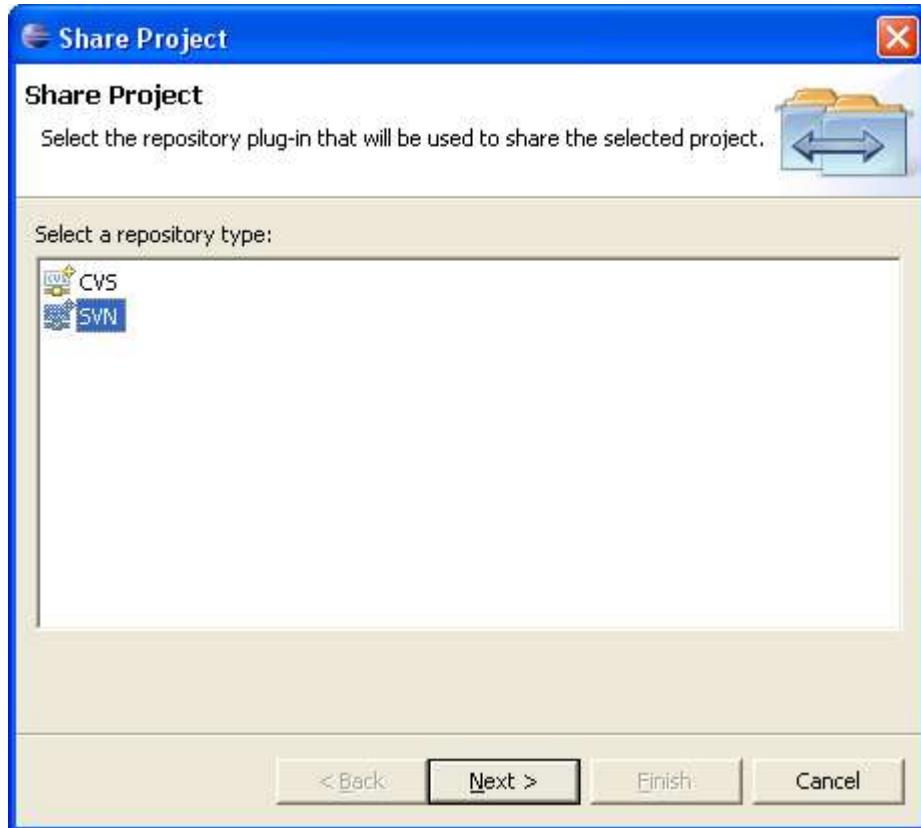
Rentrer les champs comme ci-dessus en utilisant le nom d'utilisateur et le mot de passe correct (ceux du serveur project.srv-net.com)



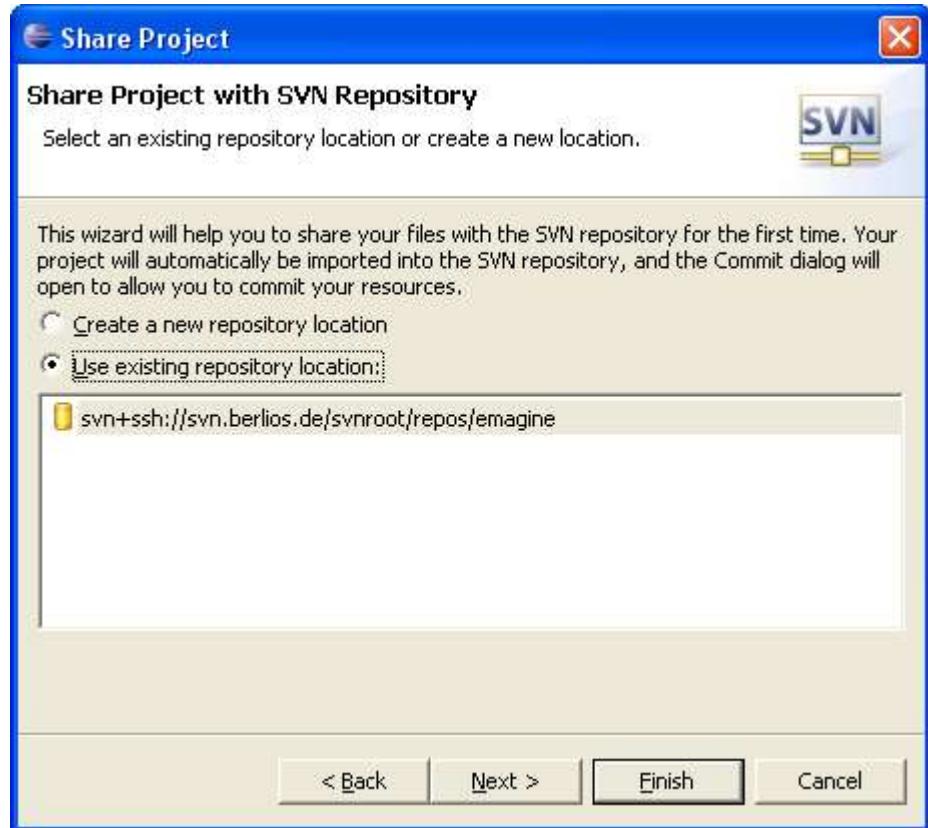
Sélectionner le répertoire pour la création du projet (répertoire trunk). Puis faire Check Out As Project.

Si le projet existe déjà utiliser la commande suivante :

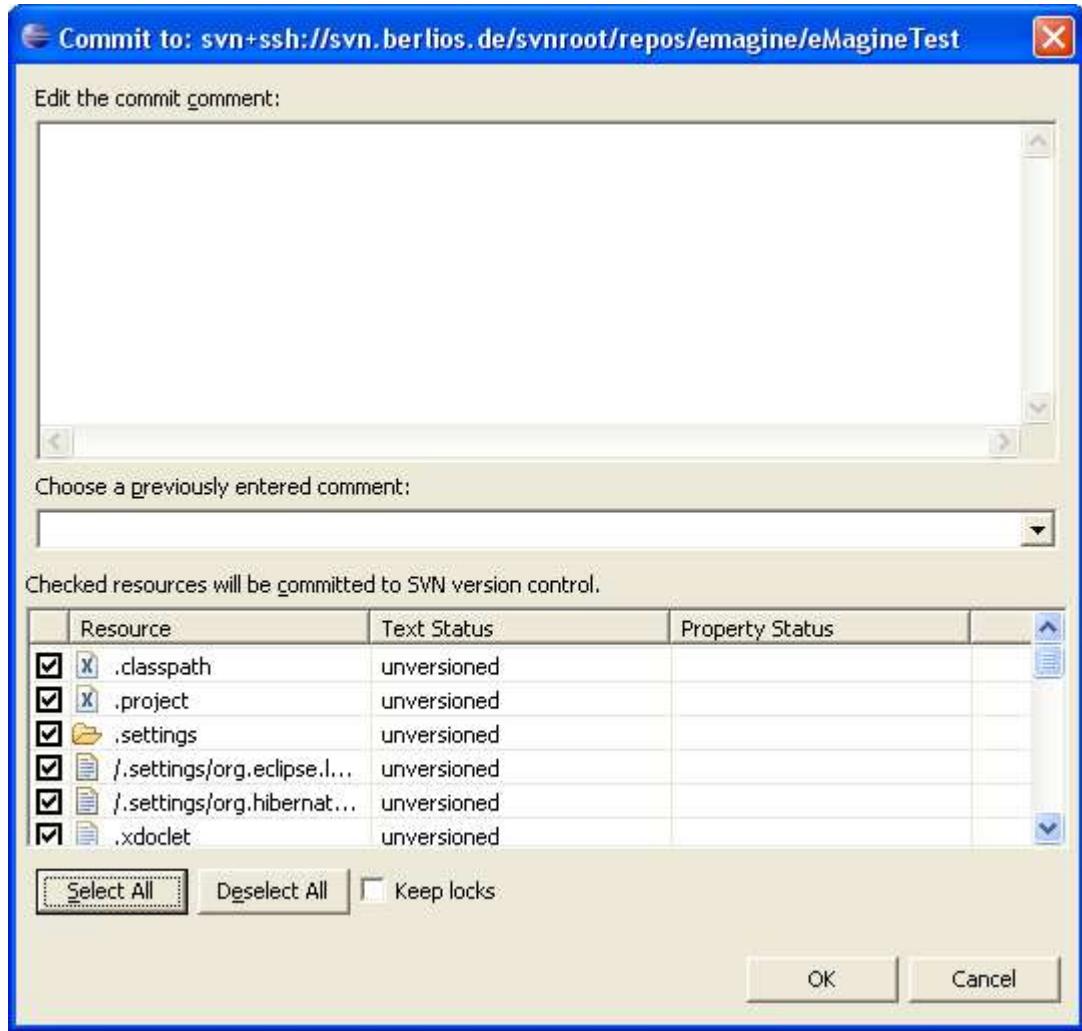




Sélectionner SVN. Puis cliquer sur 'Next'



Cliquer sur Finish



Sélectionner tout les fichier et cliquer sur ok.

Le projet est ajouté. A présent l'arborescence à droite affiche les informations de version de différents fichiers. Lorsqu'un '>' apparaît devant le fichier c'est qu'il est modifié localement. Les autres commandes sont identiques à SubVersion Normal.

## 2.15. TEST DE PERFORMANCE

### a. Justification

JMeter est le seul outil permettant de tester la disponibilité des serveurs en java, il propose les fonctions suivantes :

- Peut charger et tester les performances pour les serveurs HTTP et FTP ainsi que les requêtes SQL via JDBC.
- Complètement portable, fait en java.
- Interface en composants légers (Swing).
- Supporte le multi-threading pour lancer plusieurs tests simultanément.
- Interface graphique optimisée.
- Analyse des tests réalisés.
- Extensible à l'aide de plugin.

JMeter va permettre d'évaluer la capacité de réponse du serveur d'application. Puisque c'est le cœur de l'application eMagine.

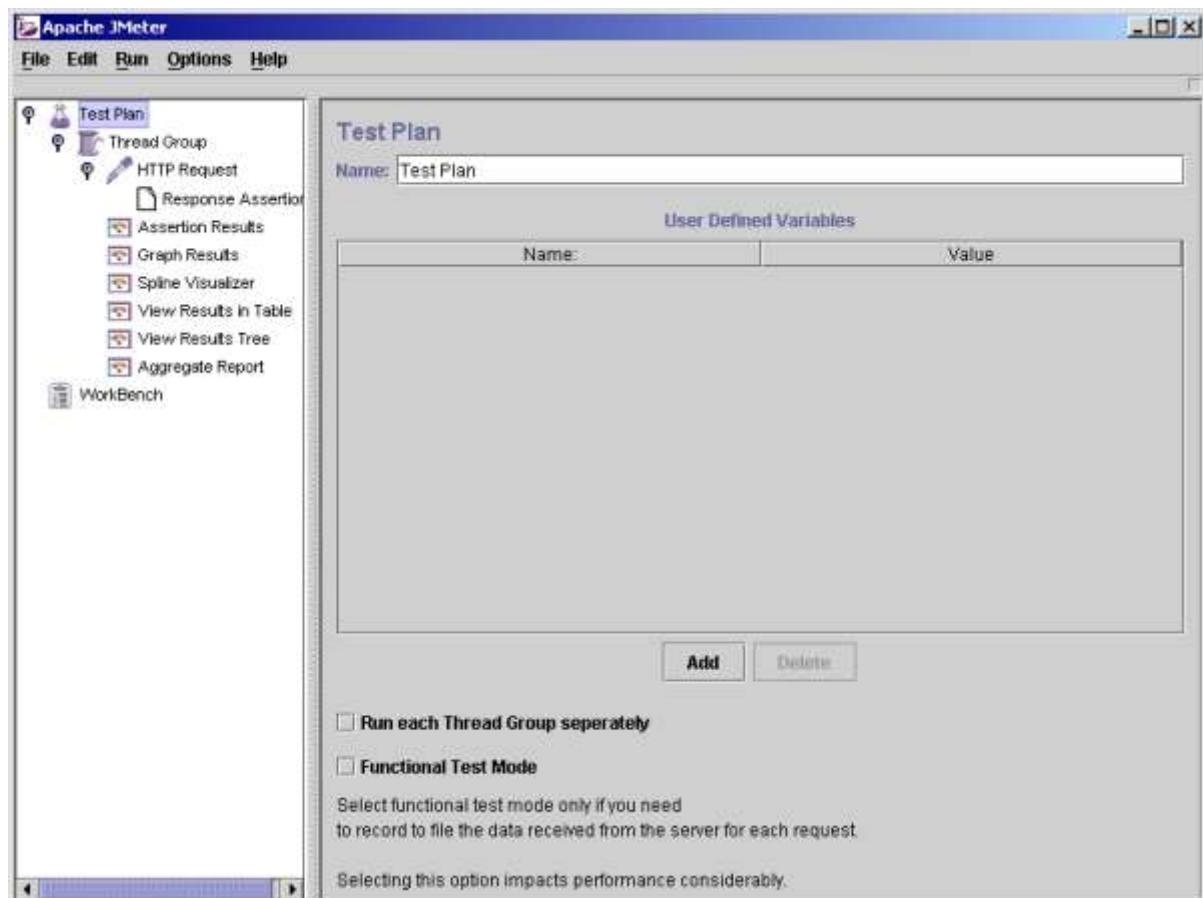
### b. Utilisation

#### ➤ Utilisation

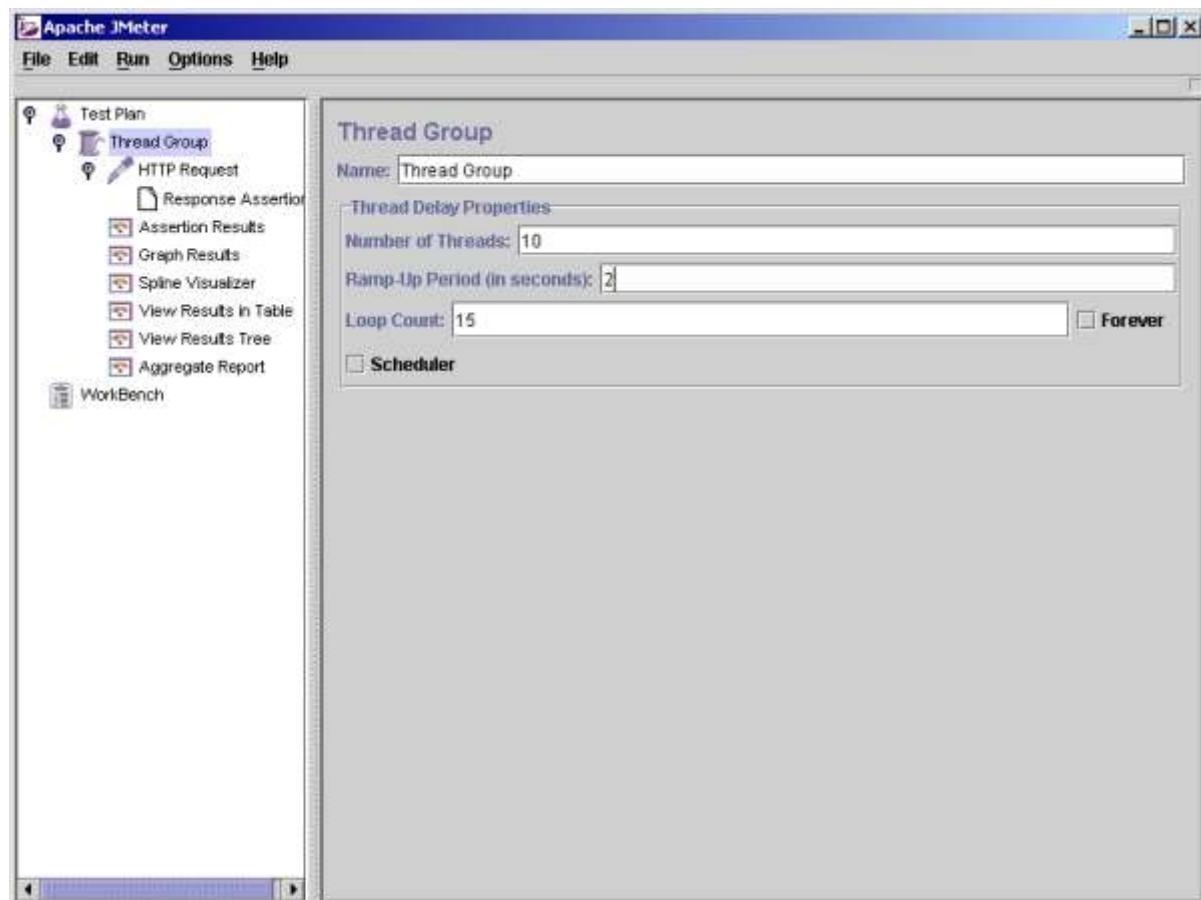
##### ***Les composants de base***

Les composants de base de JMeter sont les éléments indispensables pour réaliser le test d'un Site Internet. Ces éléments sont obligatoirement présents :

- le **Test Plan** : c'est à partir de cet emplacement que l'on peut définir ce que l'on veut tester et comment. On peut également y créer des variables (par exemple le nom d'un serveur à tester) qui pourront être réutilisée dans l'ensemble du test.



- le **Thread Group** : c'est ici que sont définis les paramètres de la simulation : le nombre d'utilisateurs qui se connectent (**number of threads**), l'intervalle de temps où ces utilisateurs se connectent (**Ramp-up Period**) et le nombre de fois où ces utilisateurs se connectent (**Loop Count**). On peut également spécifier que les utilisateurs se connectent indéfiniment en cochant la case **Forever**. Dans l'exemple suivant, 10 utilisateurs se connectent dans un délai de 2 secondes, et cela se répète 15 fois :



- le **WorkBench** : c'est l'emplacement qui permet de garder les composants de test non utilisés. Cela évite de les supprimer si on sait que l'on va les réutiliser par la suite.
- le **SSL Manager** : c'est un composant permettant d'utiliser les connexions sécurisées.

### Les samplers

Les samplers sont les différentes requêtes que l'on peut envoyer à un serveur ou à un Site Internet pour tester ses ressources. Les samplers disponibles dans JMeter sont :

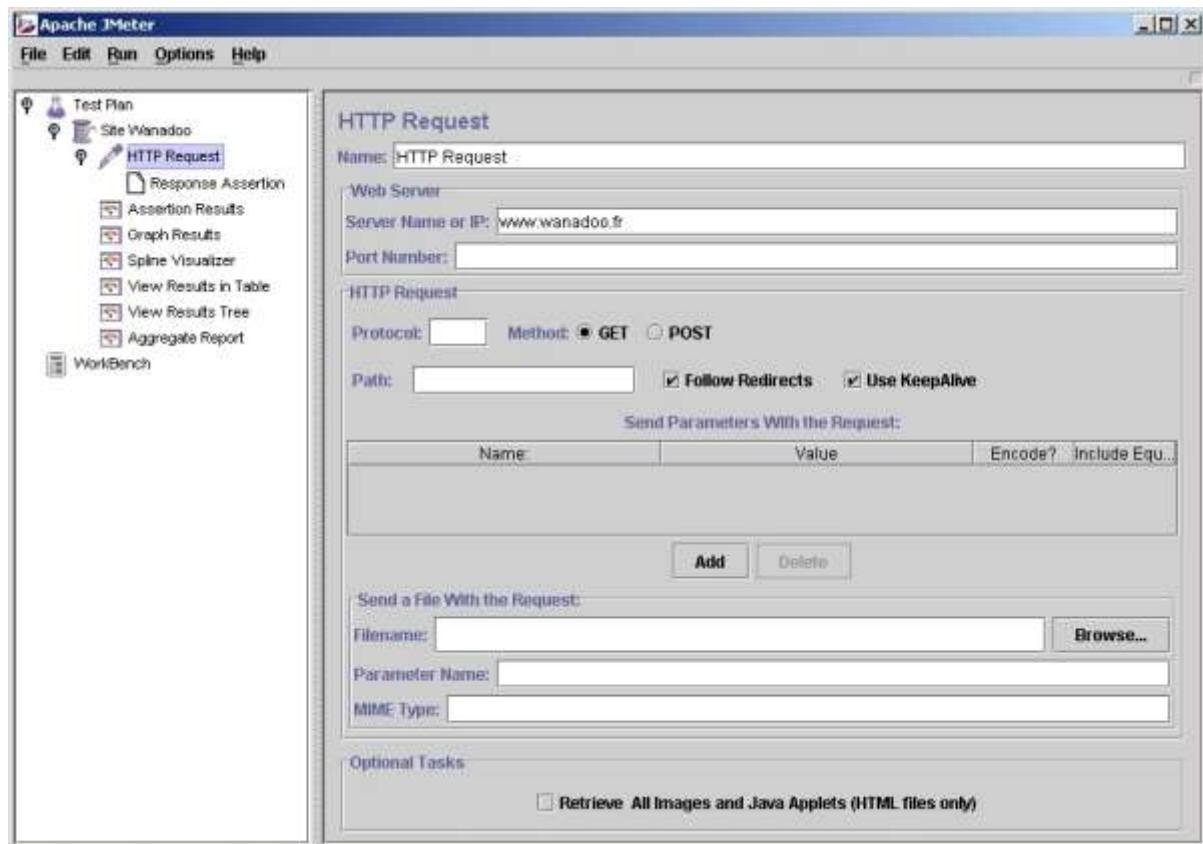
- **HTTP Request** : c'est un composant permettant d'envoyer une requête à un Site Internet. L'HTTP Request récupère la page que nous renvoie normalement le navigateur Internet. Il est également possible d'envoyer un formulaire avec la méthode GET ou POST, en saisissant les

paramètres de ce formulaire dans la section **Send Parameters With the Request**.

Il est aussi possible d'envoyer un fichier avec la requête.

Enfin, il est possible de spécifier si l'on veut recevoir les images qui pourraient être présentes sur la page HTTP qui nous est retournée.

L'exemple suivant demande simplement la page d'accueil du site de Wanadoo :



- **FTP Request** : De la même manière que le HTTP Request, le FTP Request permet de récupérer une page HTML d'un serveur HTTP. En plus du nom du serveur, il faut également saisir l'emplacement du fichier à récupérer ainsi que les informations d'identification sur le serveur, si elles sont nécessaires.

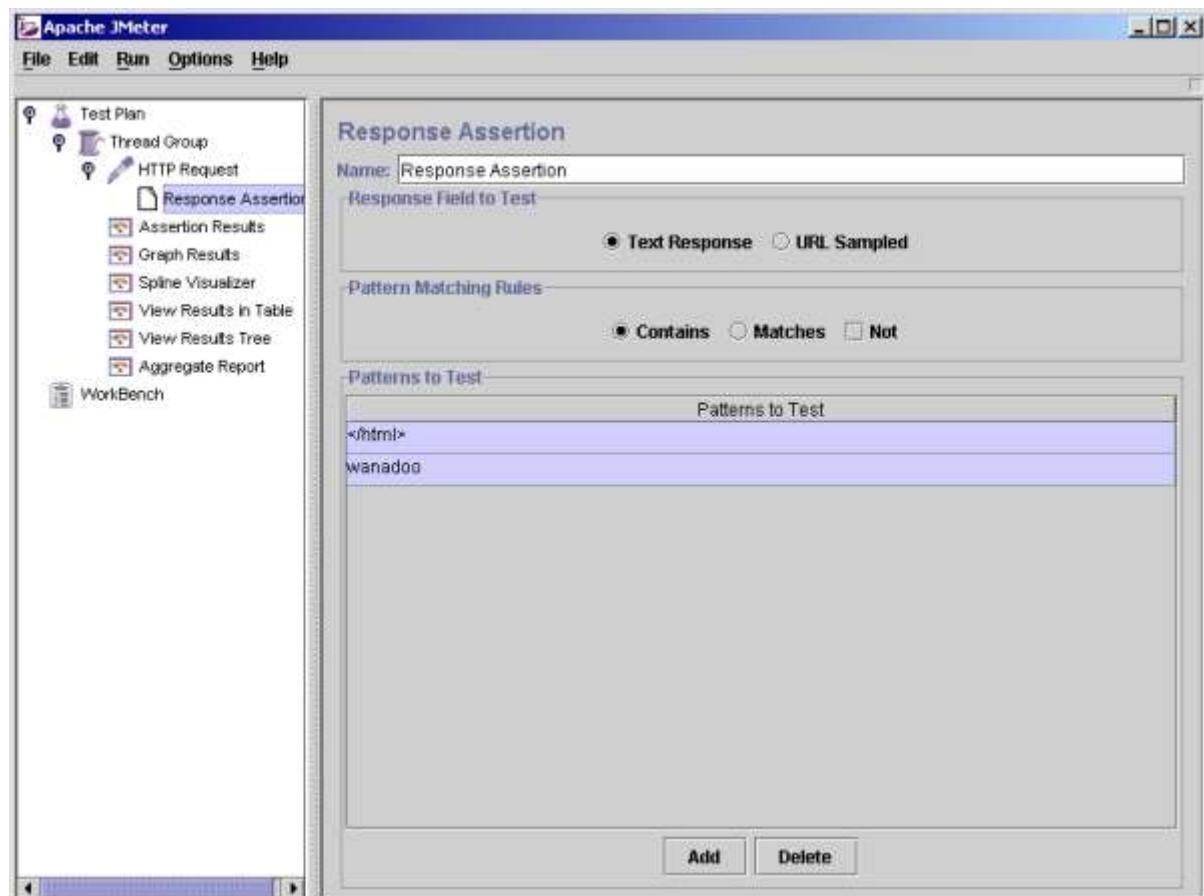


- **JDBC Request** : De la même manière que le HTTP Request, le JDBC Request permet de récupérer des données d'une base de données via les drivers JDBC.
- **JAVA Request** : De la même manière que le HTTP Request, le JAVA Request permet de récupérer des données obtenues à partir d'une applet JAVA.
- **XML Request** : De la même manière que le HTTP Request, le FTP Request permet de récupérer des données à partir d'une page XML.
- **LDAP Request** : De la même manière que le HTTP Request, le FTP Request permet de récupérer des données à partir d'un annuaire LDAP.

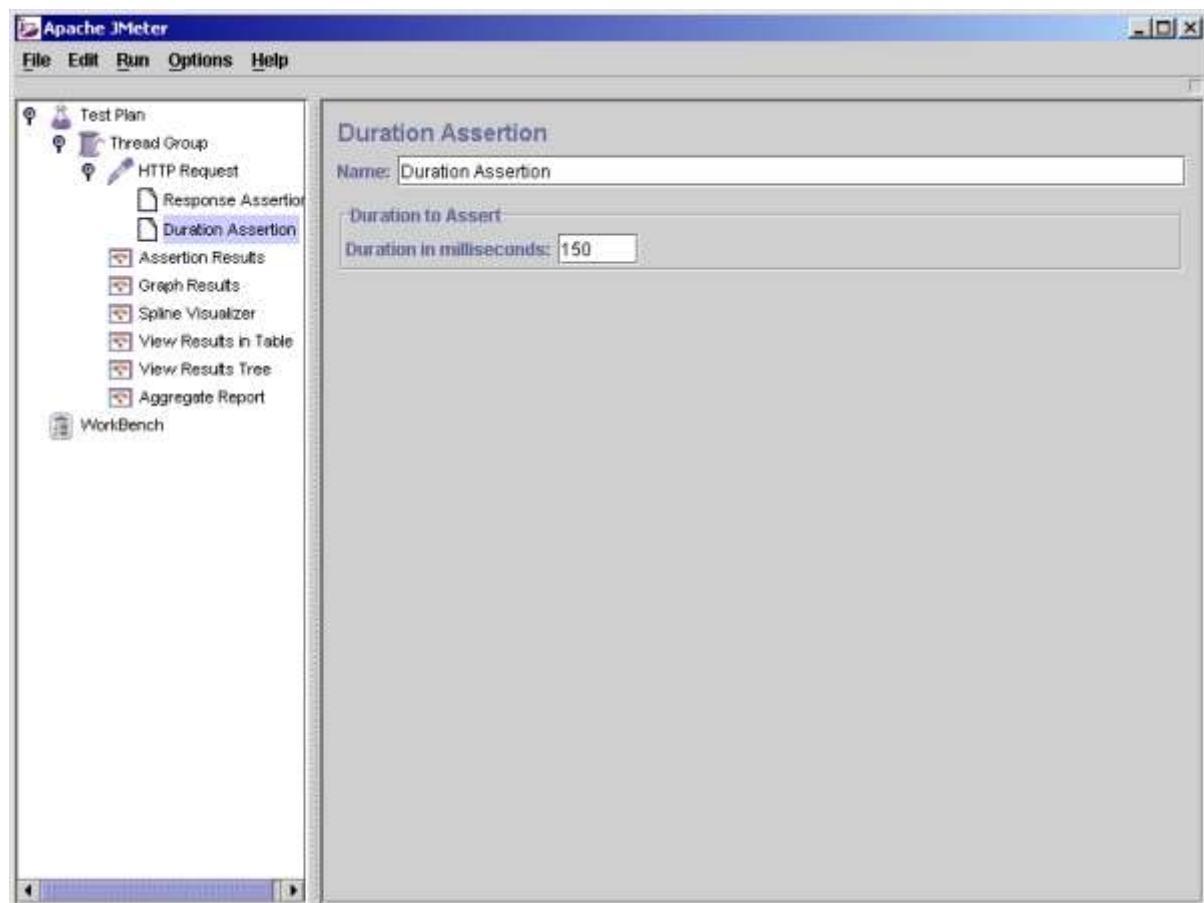
## Les Assertions

Les assertions permettent de tester le contenu des données renvoyées par le serveur pour vérifier, soit qu'elles sont bien conforme à ce que l'on veux, soient qu'elles constituent un document valide.

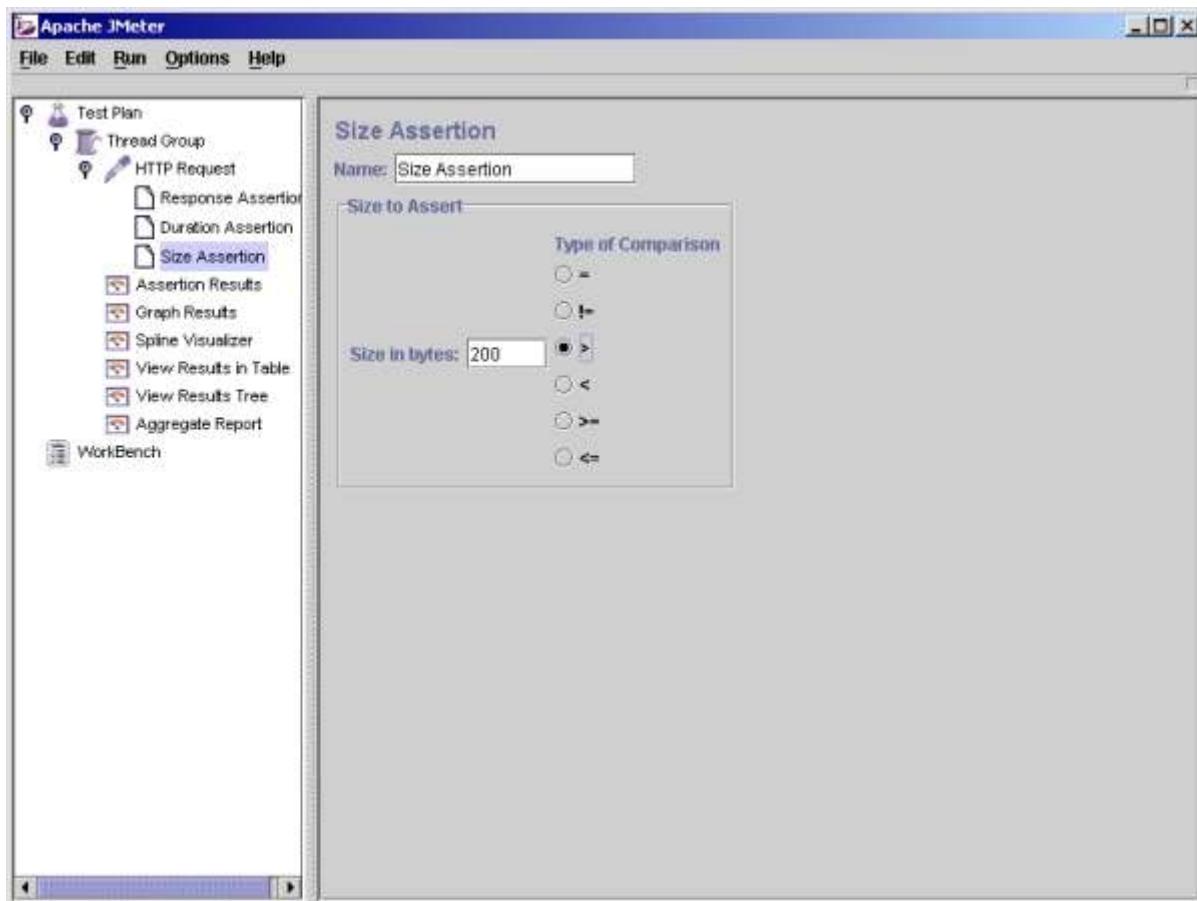
- **Response Assertion** : vérifie si une chaîne de caractère se trouve dans la réponse. L'exemple suivant vérifie que la chaîne **wanadoo** est présente sur la page que l'on demande. De plus, nous vérifions aussi que nous recevons bien une page complète en testant que la balise `</html>` est bien présente (c'est la balise qui doit terminer toutes les pages sur Internet) :



- **Duration Assertion** : c'est un composant permettant de vérifier la performance du serveur en testant le temps que met la réponse à notre requête pour arriver. Si la réponse parvient après le temps qui est défini dans le champ **Duration**, la réponse n'est pas considérée comme valide. Ici, on récupère les réponses qui nous parviennent en moins de 150ms :



- **Size Assertion** : c'est un composant qui permet de récupérer uniquement les pages qui respectent la condition sur la taille, spécifiée par le champ **Size to Assert**. Ici, nous récupérons uniquement les pages de plus de 200bytes :

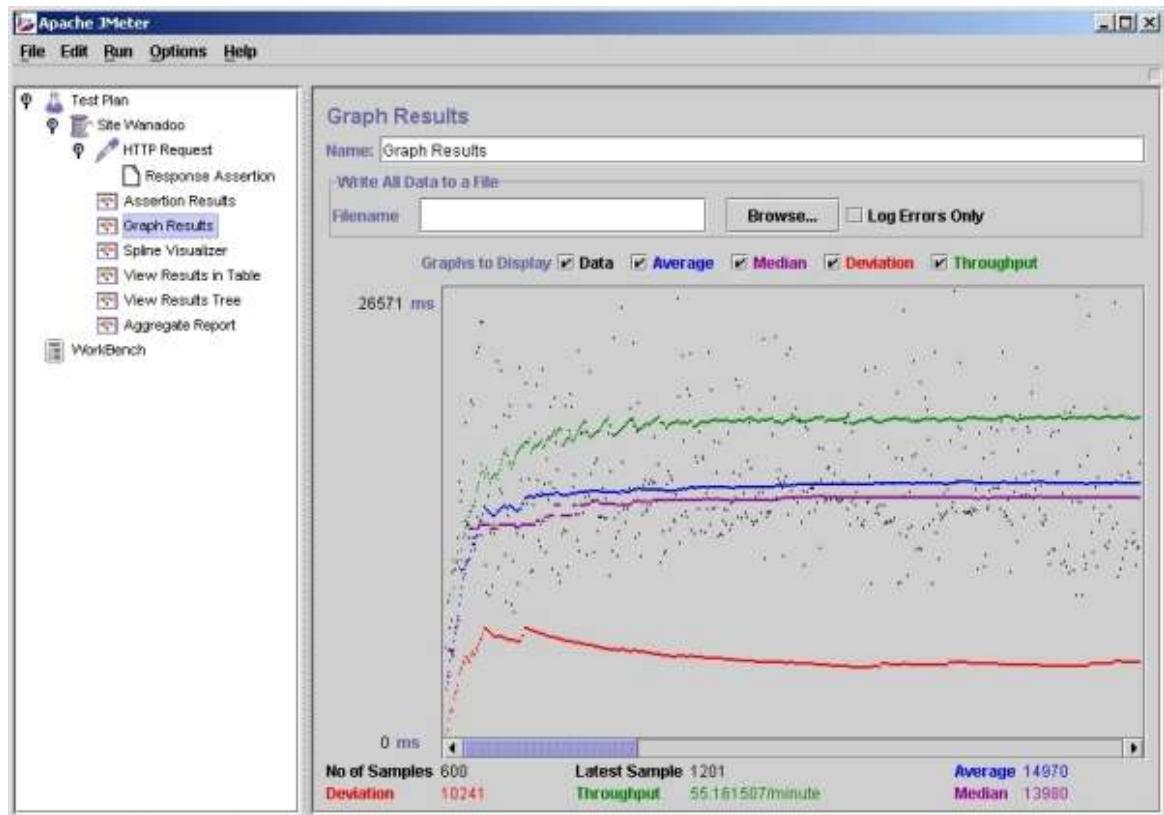


- **XML Assertion** : ce composant ne prend aucun paramètre. S'il est présent dans le test, il vérifie simplement que la page récupérée est bien un document XML valide.

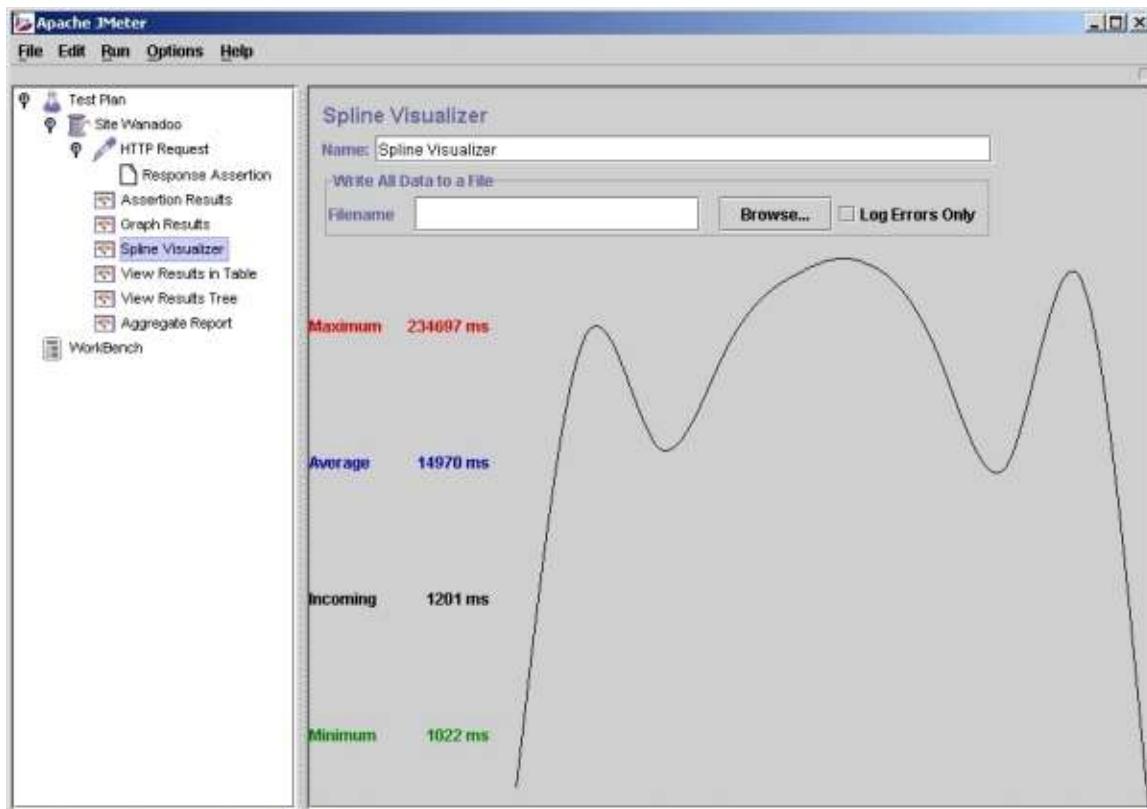
### Les Listeners

Les listeners sont les composants qui fournissent les résultats des tests que nous venons d'effectuer. Voici les différents listeners disponibles dans JMeter :

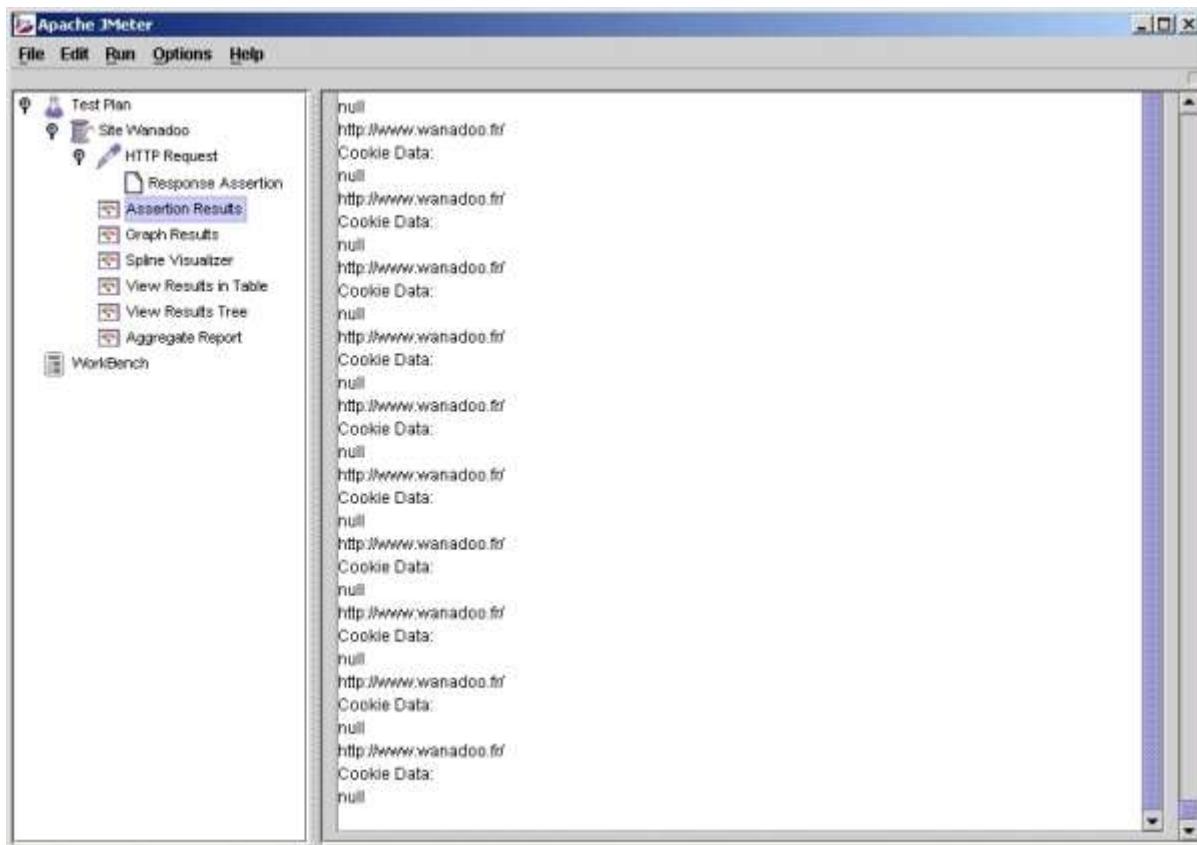
- **Graph results** : ce composant fournit un graphique présentant la moyenne (average), le temps médian (median), la déviation (écart-type) des temps de réponse des différentes requêtes effectuées, ainsi que le taux du nombre de requêtes effectuées par minute. On y trouve aussi les temps et le taux globaux pour le test (sous le graphique). Voici l'exemple d'un graphique créé par ce composant :



- **Spline visualizer** : ce composant fournit un graphique présentant les temps de réponse des toutes les requêtes effectuées. Voici un exemple de ce type de graphique :



- **Assertion results :** ce composant affiche les résultats de toutes les requêtes effectuées. Dans l'exemple suivant, toutes le requêtes effectuées ont été un succès. On voit également les données stockées dans un cookie, s'il y en a. Dans cet exemple, toutes les requêtes ont été un succès et il n'y a pas de données dans le cookie renvoyé :



- **Mailer visualizer** : c'est un composant qui permet d'envoyer un mail à un administrateur en cas de trop nombreux échecs aux requêtes effectuées. Les paramètres de ce composant sont les paramètres du mail à envoyer et les limites d'échec ou de succès à partir duquel un mail doit être envoyé. Voici un exemple pris sur le site de JMeter :

**Mailer Visualizer**

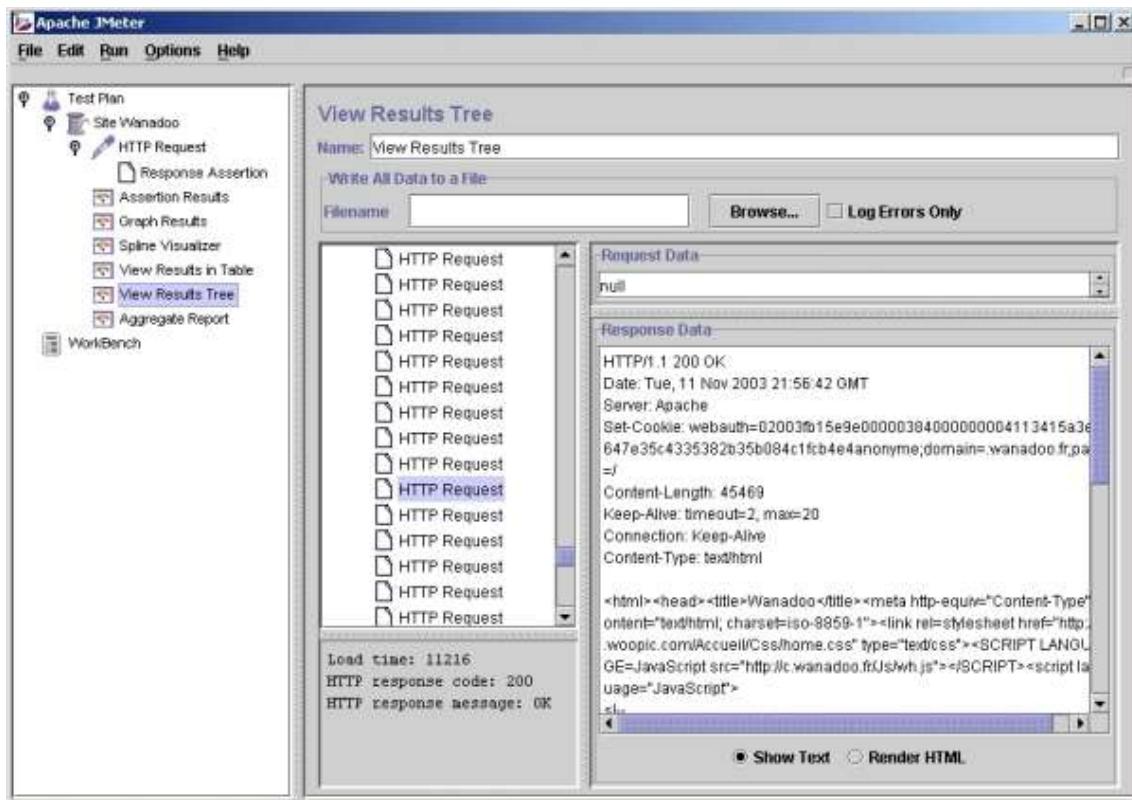
**Name:** Mailer Visualizer

**Mailing attributes**

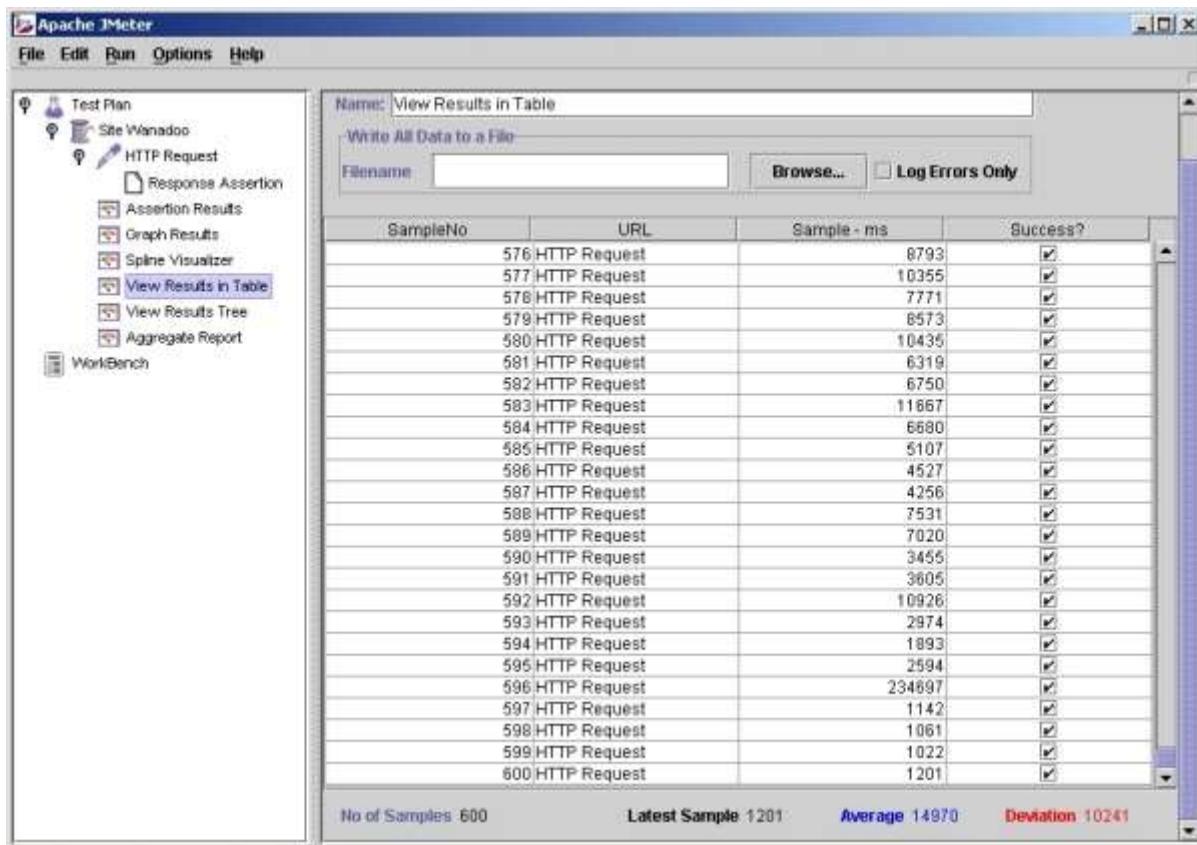
<b>From:</b>	from@server.co0
<b>Addressie(s):</b>	to@server.co0
<b>SMTP Host:</b>	smtp.server.co0
<b>Failure Subject:</b>	server failure
<b>Success Subject:</b>	server recovered
<b>Failure Limit:</b>	2
<b>Success Limit:</b>	2

**Test Mail**   **Failures:**

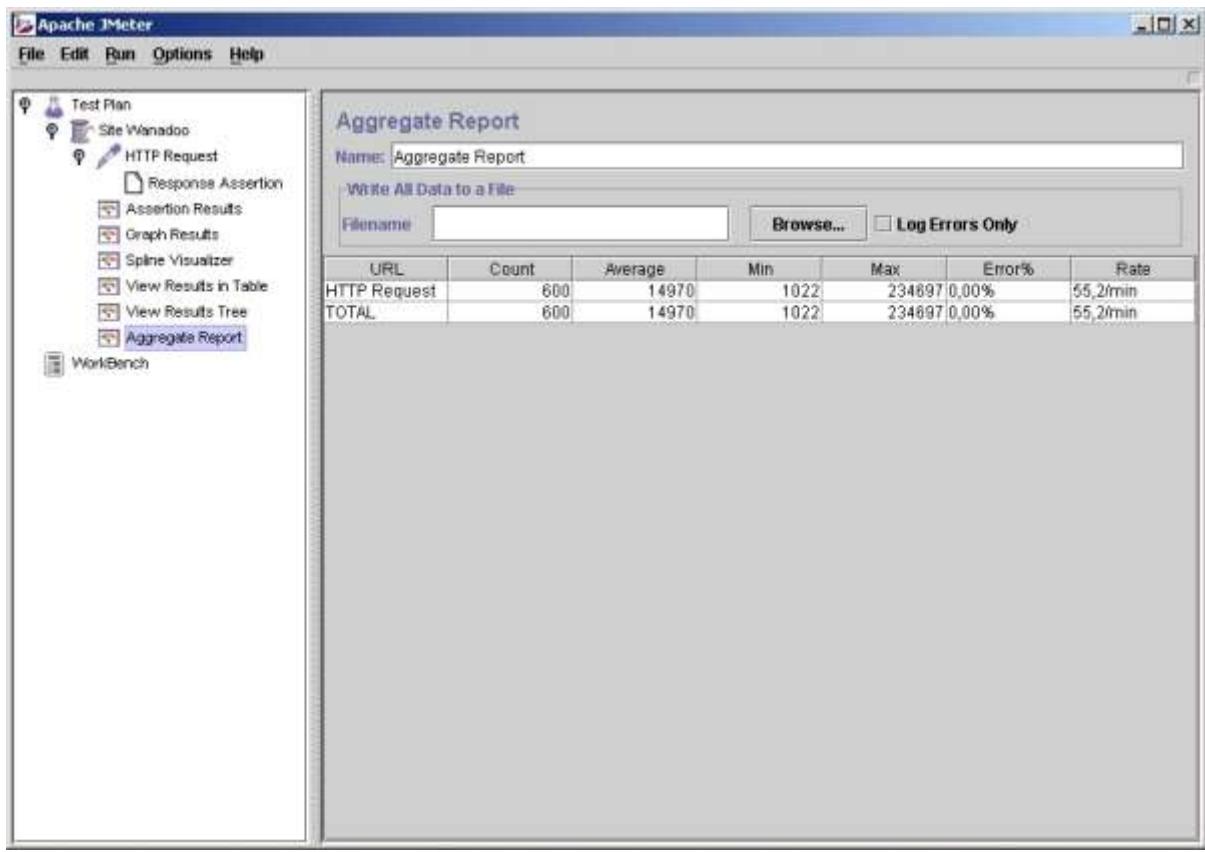
- **Tree results** : c'est un composant qui affiche l'ensemble des réponses aux requêtes effectuées. On y trouve pour chaque requête effectuée, le type de la requête et le contenu de la page récupérée (**Response Data**). Voici par exemple le contenu de la page d'accueil du site de Wanadoo :



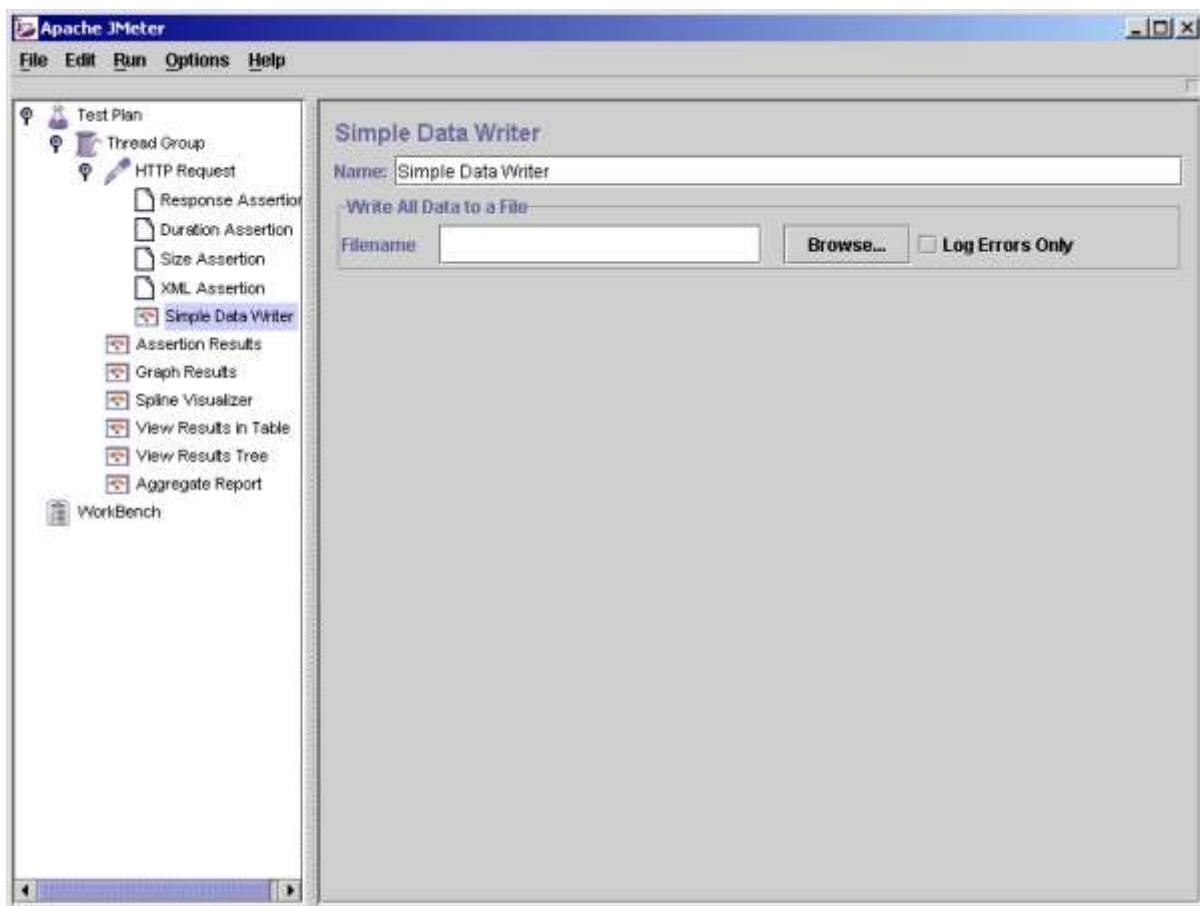
- **Table results** : c'est un composant qui crée une table qui reprend l'ensemble des requêtes effectuées. On y trouve pour chaque requête effectuée, le type de la requête (**URL**) le temps de réponse (**Sample -ms**) et si la réponse de la requête est correcte (**success?**).



- **Aggregate report** : c'est un composant qui crée une table des statistiques qui reprend l'ensemble des tests. On y trouve le nombre de requêtes effectuées, les temps de réponse (moyen -> **average**, minimum -> **min** et maximum -> **max**) pour chaque requête, le pourcentage des requêtes ayant échouées (Error%) et le taux de requêtes par minute (rate). Voici un exemple avec les résultats pour une simple requête :



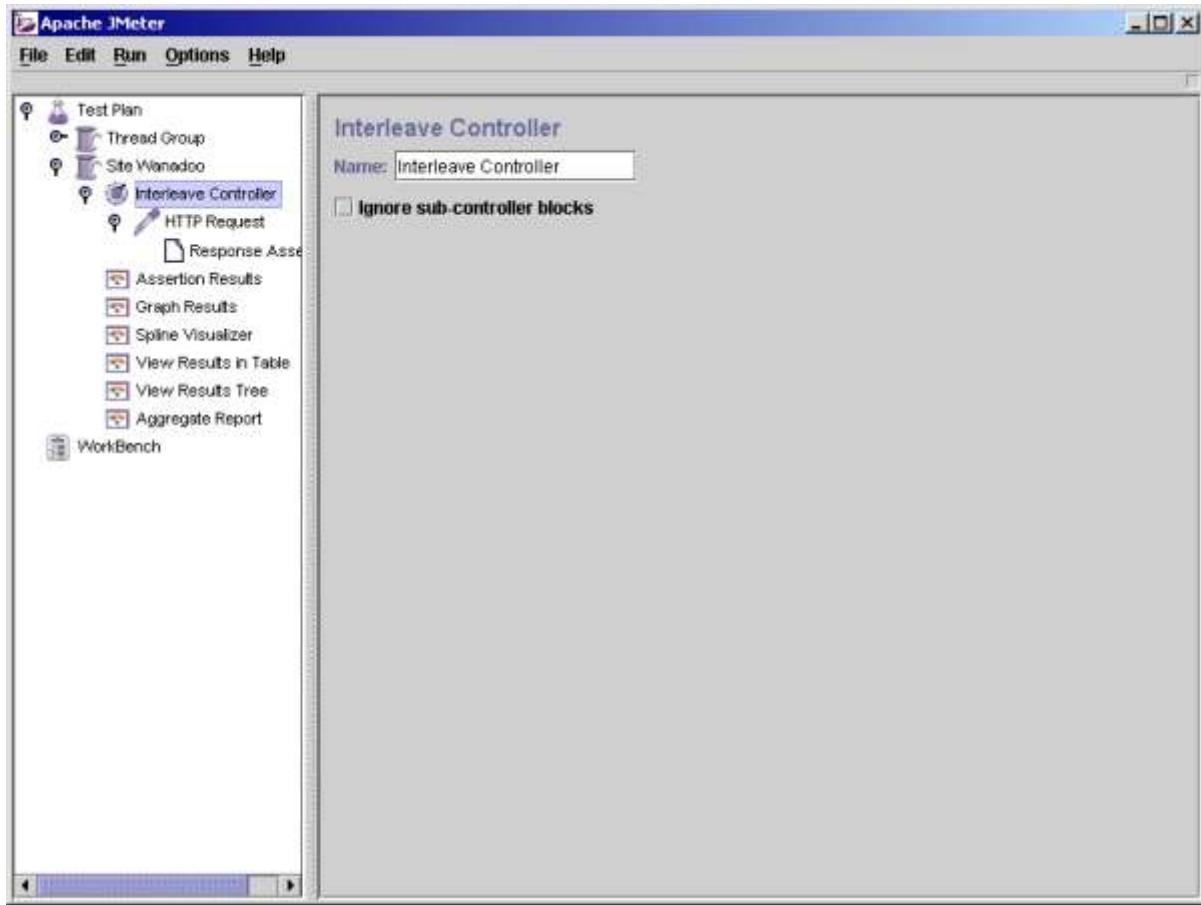
- **Simple data writer** : c'est un composant qui écrit les résultats des tests dans un fichier. Il est aussi possible de spécifier s'il on veut uniquement les résultats lors des erreurs (**Log Errors Only**). Le seul paramètre de ce composant est le nom du fichier de sortie (**Filename**) :



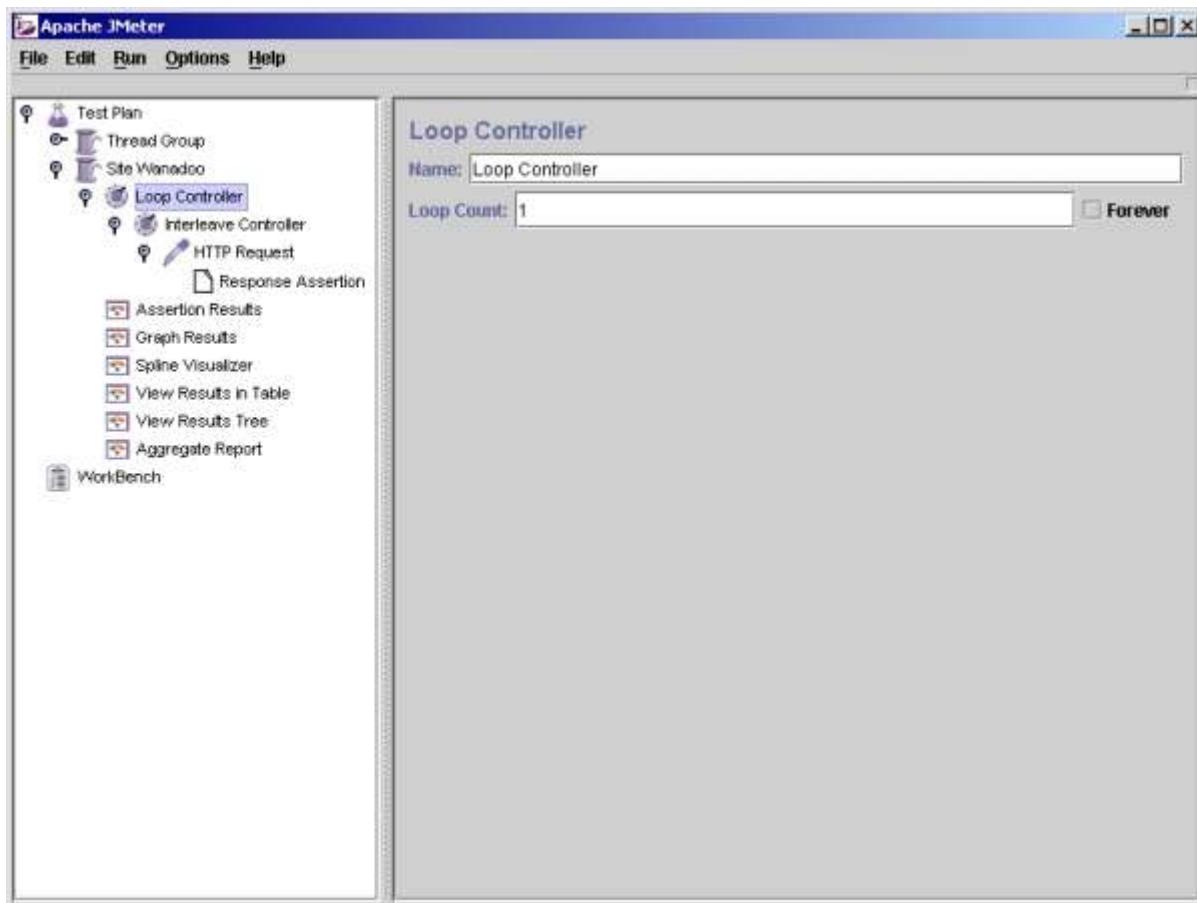
## Les Logic Controllers

Les Logic Controllers permettent de spécifier les paramètres des tests à effectuer :

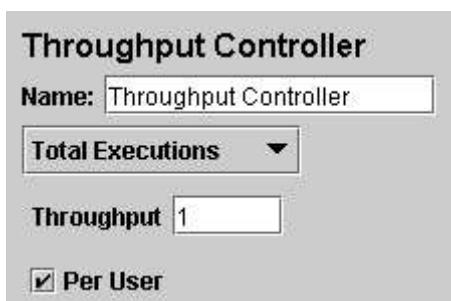
- **Interleaves Controller et Random Controller** : ces composants permettent d'envoyer les requêtes dans un ordre aléatoire afin que le test reproduise plus fidèlement la réalité d'un site Web auquel on ne demande pas toujours les pages dans le même sens. Dans un test, ces composants se placent avant les requêtes. Voici un exemple pour un Interleaves Controller :



- **Loop Controller** : ce composant permet de lancer plusieurs fois la ou les requêtes spécifiées après ce composant, même s'il est spécifié dans le **Thread Group** qu'il doit y avoir 5 boucles pour les requêtes. Dans cet exemple, la HTTP Request est réalisée une seule fois dans l'ensemble du test.



- **Once Only Controller** : ce composant permet de lancer une seule fois la ou les requêtes spécifiées après ce composant, même s'il est spécifié dans le **Thread Group** qu'il doit y avoir 5 boucles pour les requêtes. Ce composant ne prend aucun paramètre.
- **Throughput Controller** : ce composant permet d'exécuter le test un nombre limité de fois. Voici un exemple de ce composant pris sur le site de JMeter :



## Les Configuration Elements

Ces composants permettent de gérer les paramètres de certains sites ou serveurs, comme les cookies, les sites sécurisés,... Voici les principaux composants disponibles dans JMeter :

- **HTTP, FTP, LDAP, JDBC et JAVA Request Defaults** : ces composants permettent de définir des paramètres par défaut pour les serveurs où les pages où sont effectués les tests, où alors les paramètres de connexion à une base de données via les drivers JDBC : par exemple, l'adresse du serveur, le protocol utilisé, le numéro du port, le chemin d'accès, le fichier à récupérer.... Toutes les informations fournies dans ces composants sont valable pour toutes les requêtes effectuées après. Voici deux exemple pris sur le site de JMeter : le **HTTP Request Defaults** et le **FTP Request Defaults** :

**HTTP Request Defaults**

Name:	HTTP Request Defaults		
Protocol:			
Server Name or IP:			
Path:			
Port Number:			
<b>Send Parameters With the Request:</b>			
Name:	Value	Encode?	Include Equ...
<input type="button" value="Add"/> <input type="button" value="Delete"/>			

**FTP Request Defaults**

Name:	FTP Request Defaults
Server:	
File to retrieve from server:	

- **HTTP Authorization Manager** : ce composant permet de fournir un login et un mot de passe pour réaliser des tests sur des sites ou des pages où l'accès est restreint. Voici un exemple de composant, pris sur le site de JMeter :

### HTTP Authorization Manager

Name: HTTP Authorization Manager

Authorizations Stored in the Authorization Manager

Base URL	Username	Password

**Add** **Delete** **Load** **Save As...**

- **HTTP Cookie Manager** : ce composant permet de stocker et d'envoyer des cookies au serveur, en spécifiant le contenu des cookies à envoyer. Voici un exemple de ce composant :

### HTTP Cookie Manager

Name: HTTP Cookie Manager

Cookies Stored in the Cookie Manager

Name	Value	Domain	Path	Secure	Expiration

**Add** **Delete** **Load** **Save As...**

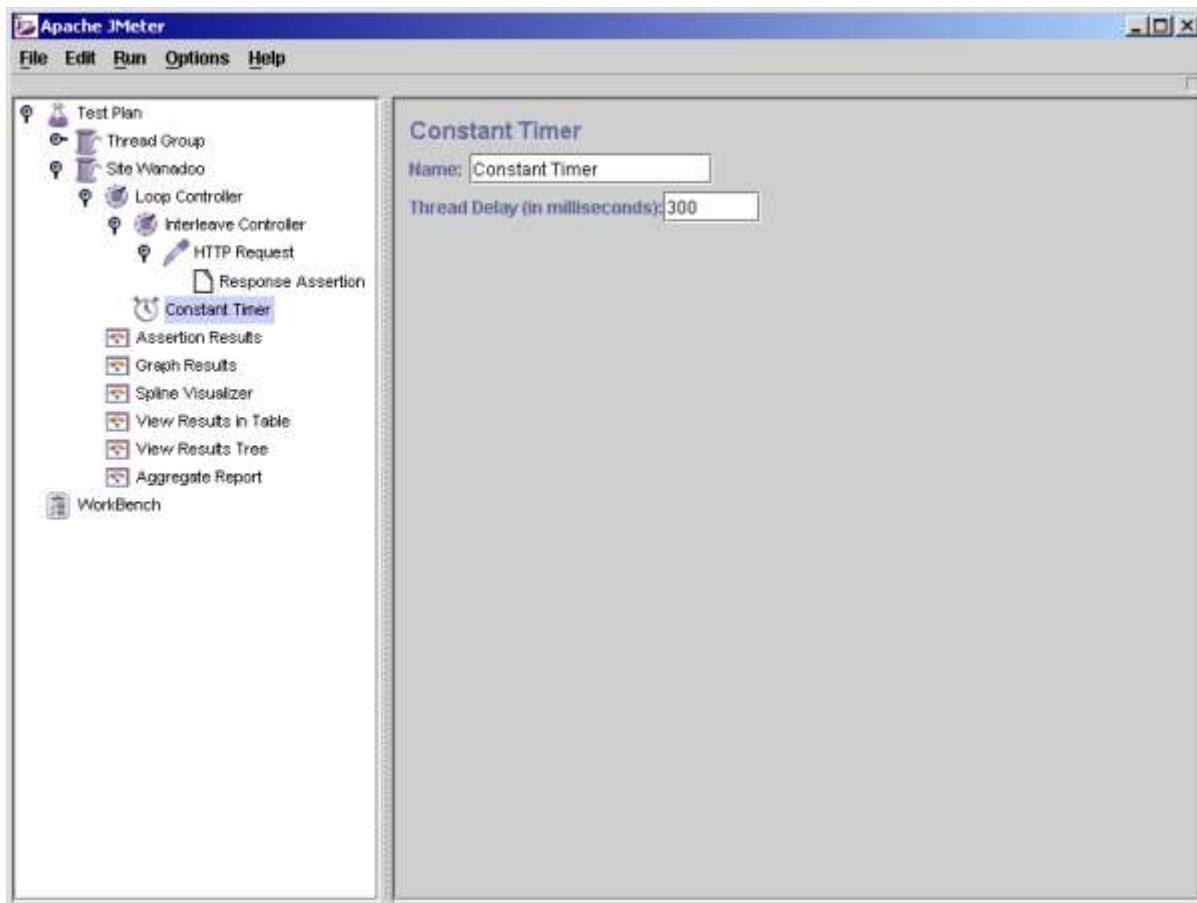
- **HTTP Header Manager** : ce composant permet de définir les en-têtes HTTP des requêtes envoyées. Voici un exemple de ce composant pris sur le site de JMeter :



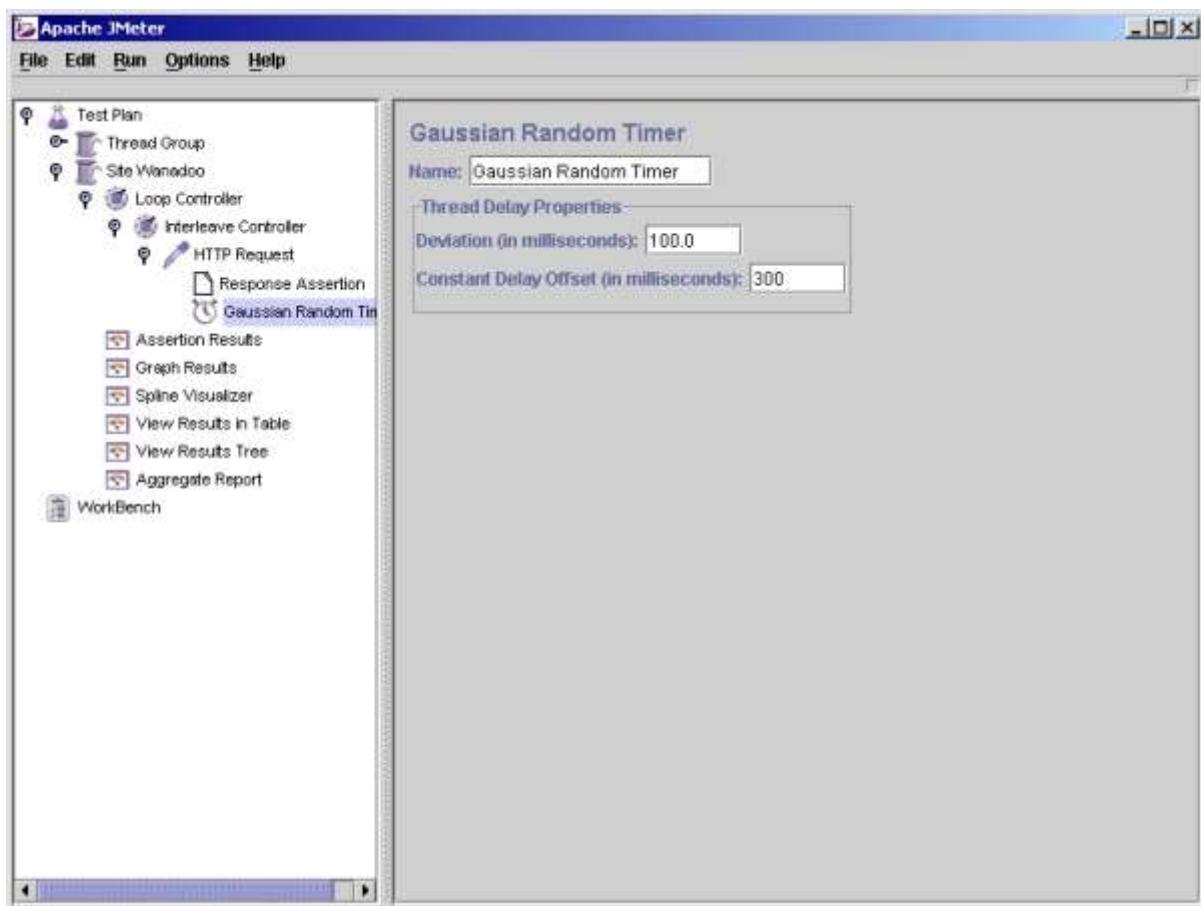
### **Les Timers**

Les timers sont des composants permettant d'insérer une pause entre chaque requête envoyées, afin que la simulation soit plus réaliste. Voici les timers disponibles avec JMeter :

- **Constant timer** : ce composant permet d'insérer un temps constant entre chaque requête envoyée : dans cet exemple, 300ms :



- **Gaussian Random Timer** et **Uniform Random Timer** : ce composant insère une pause aléatoire entre chaque requête envoyée. Voici un exemple d'un Gaussian Random Timer qui insère une pause de 300ms par requête (**Constant Delay**), à plus ou moins 100ms par requête (**Deviation**):



## Les PreProcessors

Les PreProcessors sont des outils permettant :

- de récupérer les liens et les formulaires d'une page (HTML Link Parser)
- de récupérer et renvoyer aux autres requêtes les paramètres passés dans l'URL de la page (HTML URL Re-writing Modifier) :

**HTTP URL Re-writing Modifier**

Name: URL Rewriting Modifier

Session Argument Name

Path Extension (use ";" as separator)

Do not use equals in path extension (Intershop Enfinity compatibility)

- de définir des variables spécifiques à chaque utilisateur (User Parameters) et qui peuvent être utilisées n'importe où dans le test avec la syntaxe  **`${nom_de_la_variable}`** :

**User Parameters**

Name: User Parameters

Update Once Per Iteration

**Parameters**

Name:	User_1	User_2	User_3
username	user1	user2	user3
password	pass1	pass2	pass3
category	cat1	cat2	cat3
color	red	green	

Add Variable      Delete Variable

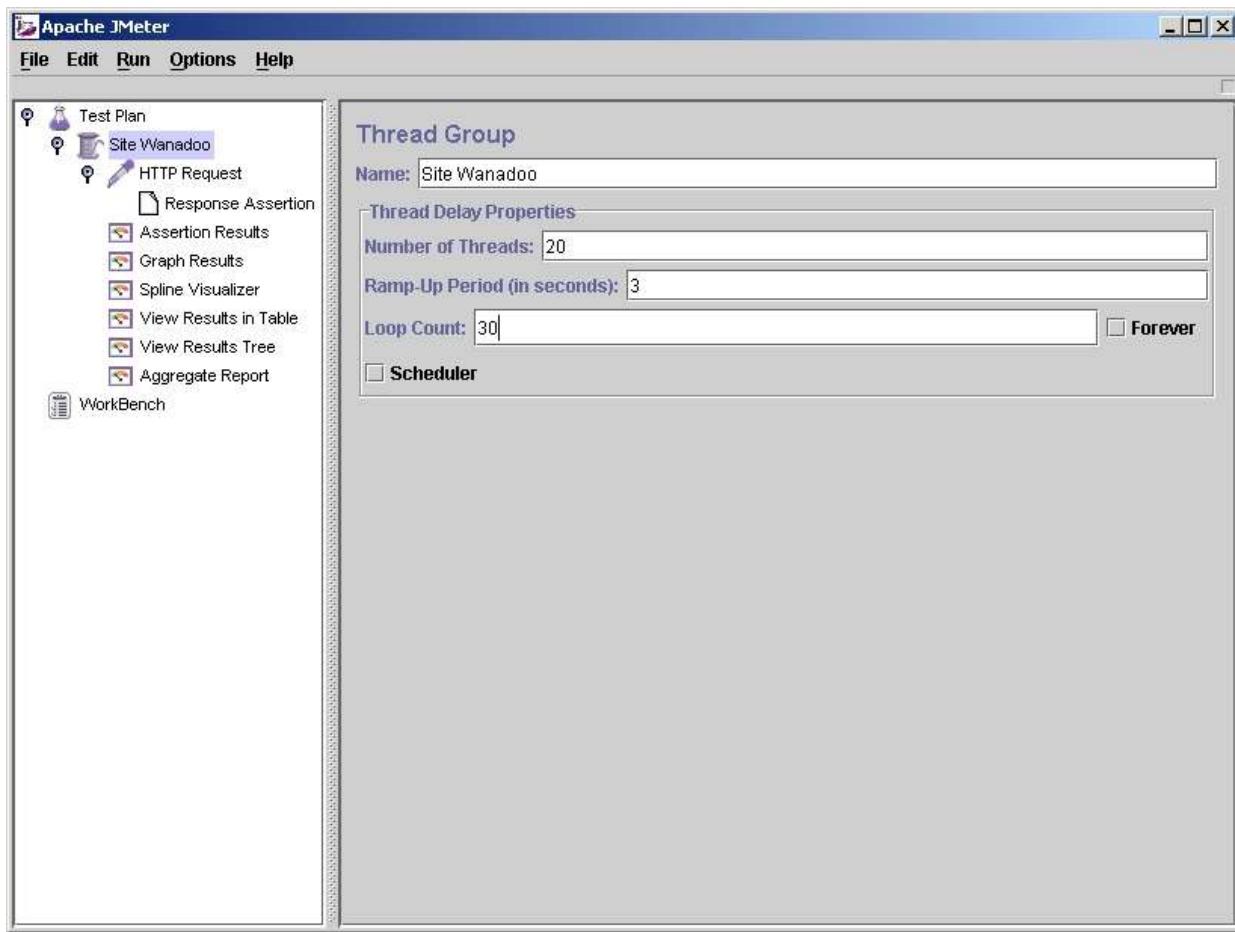
Add User      Delete User

### c. Tests et exemples

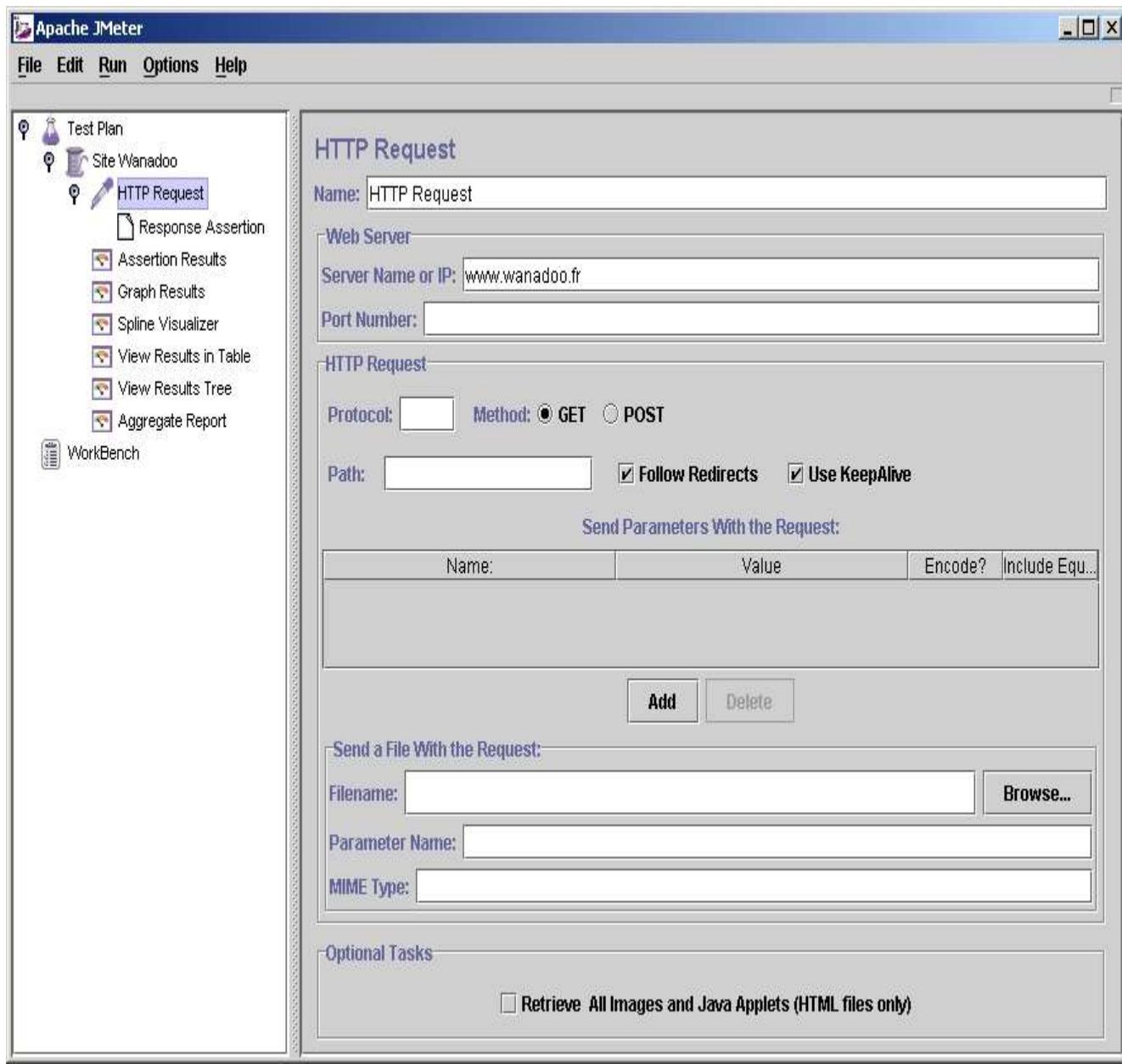
Un exemple très basique a été réalisé pour bien comprendre le fonctionnement de JMeter. Il a pour but de voir dans quel ordre créer les différents composants du test.

Il permet de tester la page d'accueil du site de Wanadoo en simulant 20 utilisateurs se connectant en 3 secondes, le tout répété 30 fois. Il faut selectionner le **Test Plan** dans l'arborescence de gauche, puis insérer un

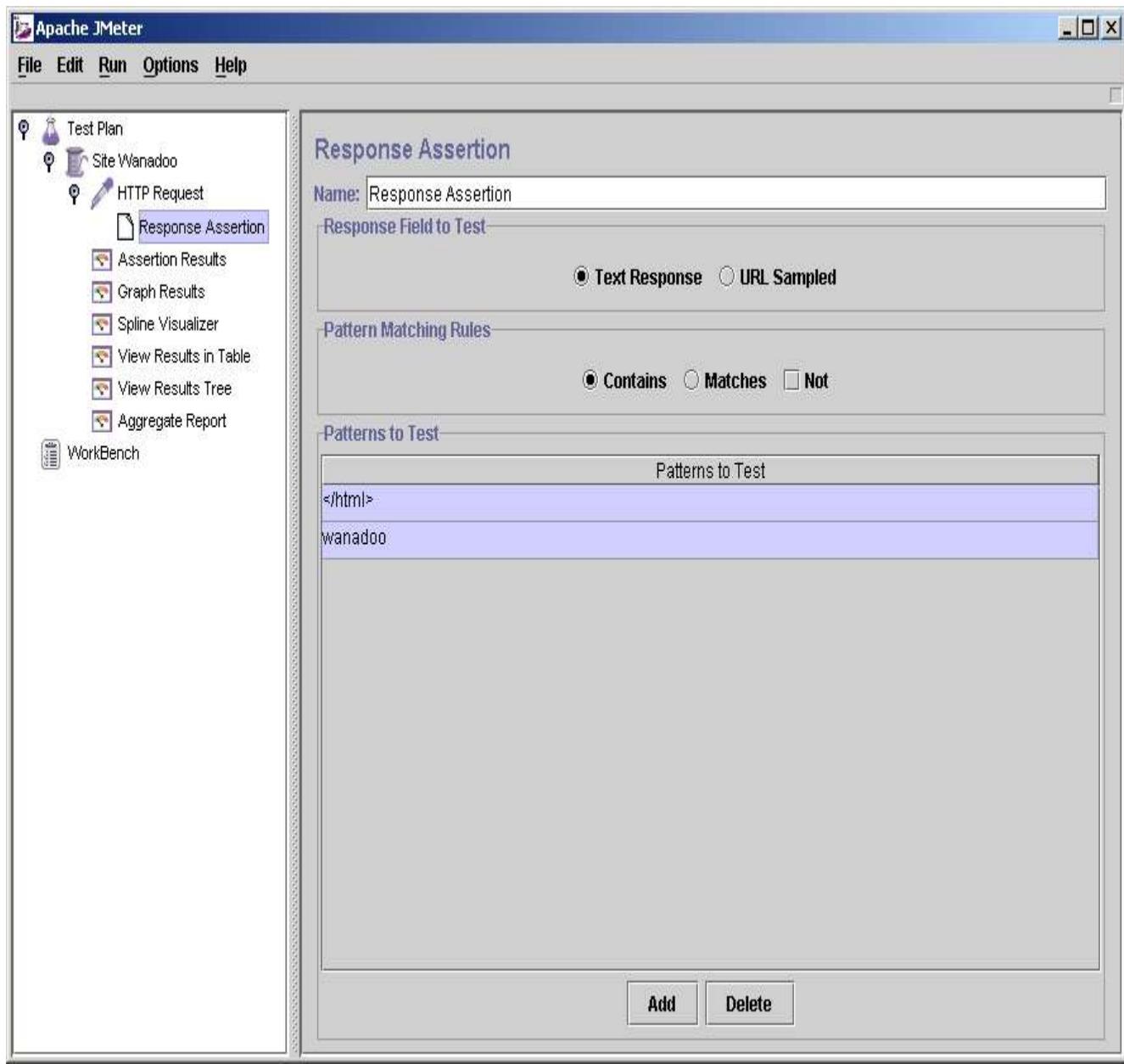
**Thread Group** par un click droit de la souris :



Sur ce **Thread Group**, il faut ensuite de la même manière créer une **HTTP Request**, qui permettra de récupérer la page d'accueil du site de Wanadoo par une méthode GET:



Sur cette HTTP Request, il faut ensuite créer une **Response Assertion** afin de vérifier que la page récupérée est complète (balise </html>) et qu'il y a bien la chaîne **Wanadoo** dans cette page :



Après cela, il faut ajouter les Listeners afin de voir les résultats des requêtes.

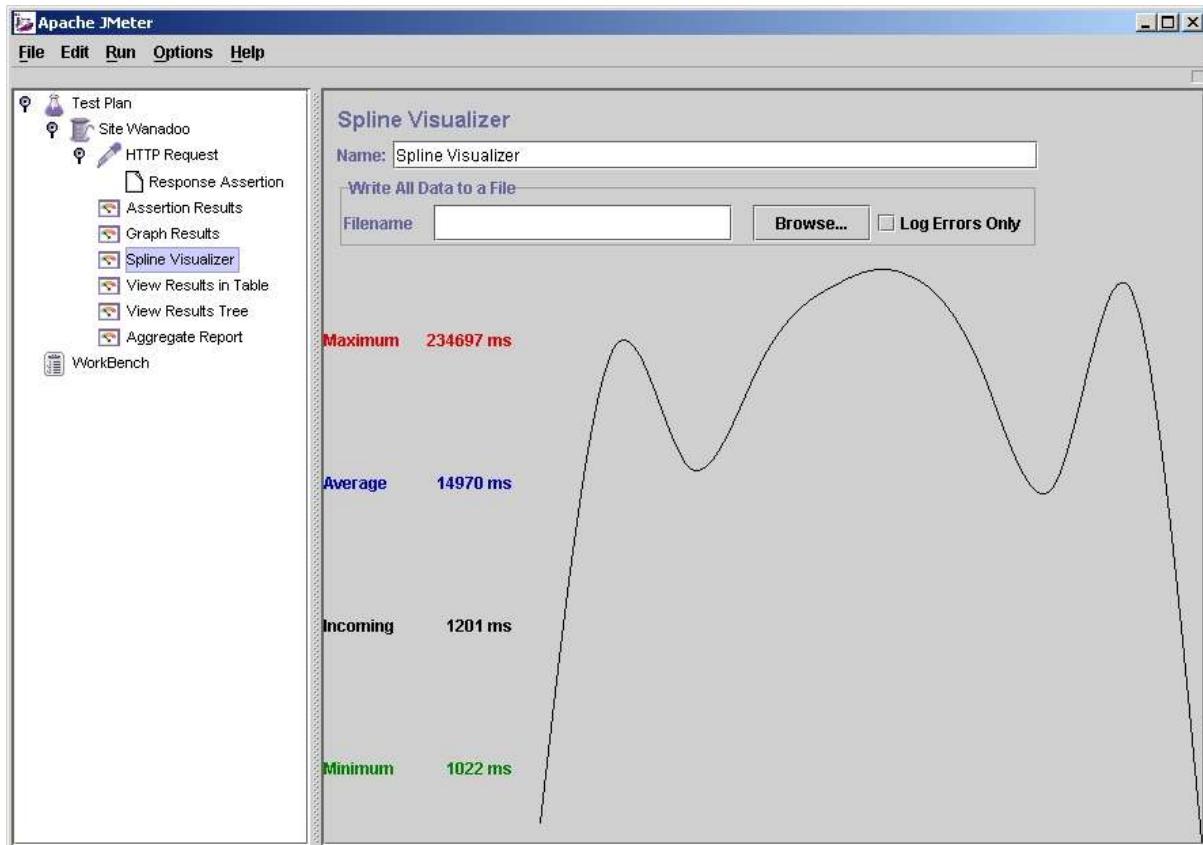
Pour ce test, ont été créés sur le **Thread Group** une **Assertion Results**, un **Graph Results**, un **Spline Results**, un **View Results in Table**, un **View Results Tree** et un **Aggregate Report**. Comme cela, les Listeners s'appliquent à toutes les requêtes effectuées dans le **Thread Group**.

Il faut ensuite lancer le test par le menu **Run -> Start**.

Une fois le test terminé, on peut voir les résultats obtenus.

Pour le **Assertion Results**, on voit que toutes les requêtes ont été effectuées avec succès (page complète + chaîne Wanadoo trouvée) :

The screenshot shows the Apache JMeter interface. The menu bar includes File, Edit, Run, Options, and Help. The left sidebar contains a tree view with nodes: Test Plan, Site Wanadoo, HTTP Request, Response Assertion, Assertion Results (which is selected and highlighted in blue), Graph Results, Spline Visualizer, View Results in Table, View Results Tree, and Aggregate Report. A separate node labeled WorkBench is also present. The main panel displays the results of the Assertion Results node. It lists multiple rows of data, each consisting of three columns: the URL (http://www.wanadoo.fr/), the Cookie Data (which is consistently "null"), and the status (Cookie Data:). There are approximately 15 such rows.



On voit bien les graphiques obtenus par le Graph Results et le Spline Visualizer.

Le View Results in Table montre bien les temps et les résultats de chaque requête :

SampleNo	URL	Sample - ms	Success?
576	HTTP Request	8793	✓
577	HTTP Request	10355	✓
578	HTTP Request	7771	✓
579	HTTP Request	8573	✓
580	HTTP Request	10435	✓
581	HTTP Request	6319	✓
582	HTTP Request	6750	✓
583	HTTP Request	11667	✓
584	HTTP Request	6680	✓
585	HTTP Request	5107	✓
586	HTTP Request	4527	✓
587	HTTP Request	4256	✓
588	HTTP Request	7531	✓
589	HTTP Request	7020	✓
590	HTTP Request	3455	✓
591	HTTP Request	3605	✓
592	HTTP Request	10926	✓
593	HTTP Request	2974	✓
594	HTTP Request	1893	✓
595	HTTP Request	2594	✓
596	HTTP Request	234697	✓
597	HTTP Request	1142	✓
598	HTTP Request	1061	✓
599	HTTP Request	1022	✓
600	HTTP Request	1201	✓

No of Samples 600      Latest Sample 1201      Average 14970      Deviation 10241

Le View Results Tree montre bien le contenu des pages récupérées par chaque requête :

Request Data  
null

Response Data

```

HTTP/1.1 200 OK
Date: Tue, 11 Nov 2003 21:56:42 GMT
Server: Apache
Set-Cookie: webauth=02003fb15e9e0000038400000004113415a3e647e35c4335382b35b084c1fc4e4anonyme;domain=.wanadoo.fr;path=/
Content-Length: 45469
Keep-Alive: timeout=2, max=20
Connection: Keep-Alive
Content-Type: text/html

<html><head><title>Wanadoo</title><meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"><link rel=stylesheet href="http://woopic.com/Accueil/Css/home.css" type="text/css"><SCRIPT LANGUAGE="JavaScript" src="http://c.wanadoo.fr/Js/wb.js"></SCRIPT><script language="JavaScript">
<!--
-->
```

Show Text  Render HTML

Et le **Aggregate Report** nous affiche bien les statistiques du test réalisé :

The screenshot shows the Apache JMeter interface with the 'Aggregate Report' selected in the left sidebar. The main panel displays the following data:

URL	Count	Average	Min	Max	Error%	Rate
HTTP Request	600	14970	1022	234697	0,00%	55,2/min
TOTAL	600	14970	1022	234697	0,00%	55,2/min

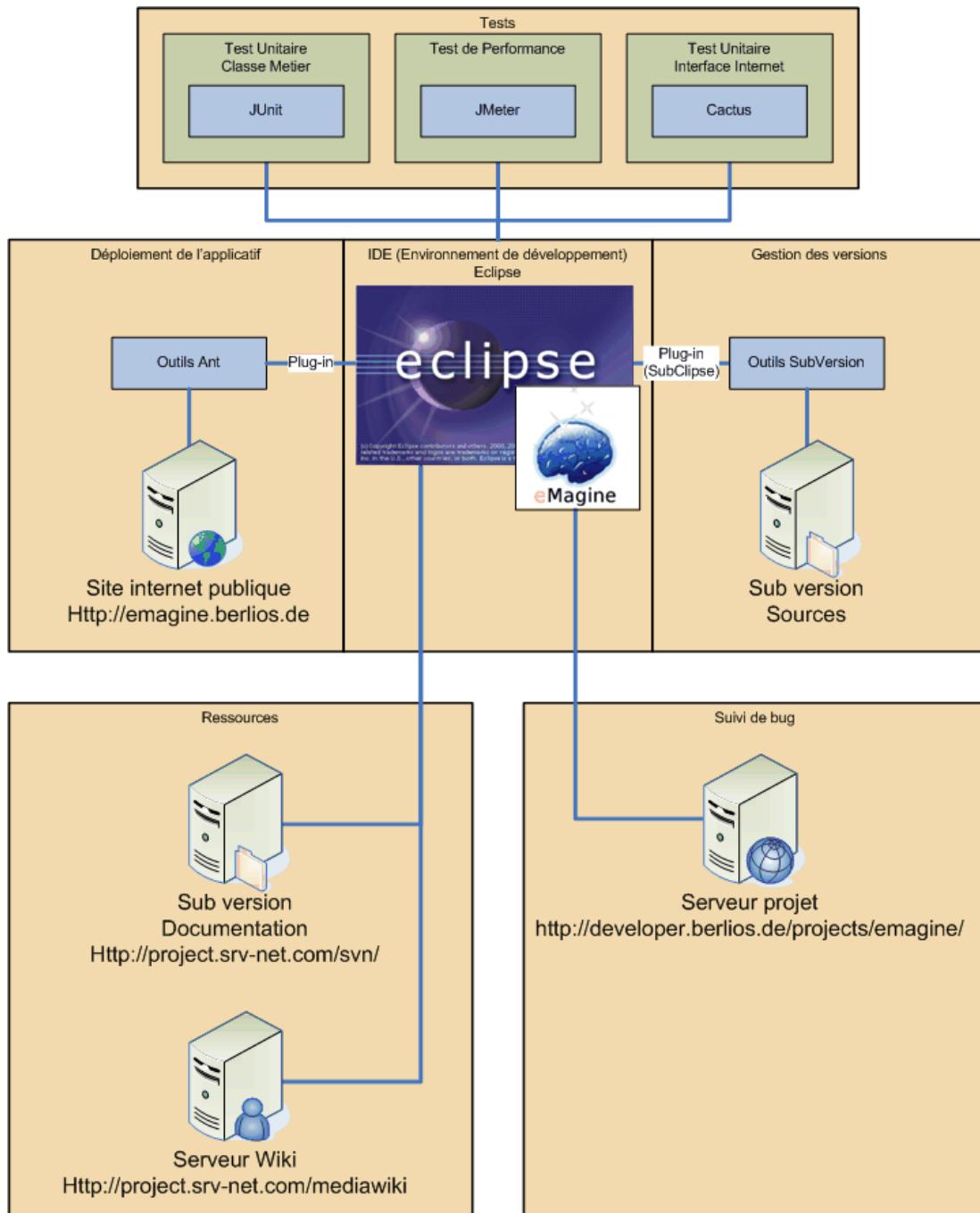
Il faut noter que les résultats de ce test ne sont pas très réalistes, surtout concernant les temps de réponses, mais cela n'était pas le but de ce test.

### 3. ARCHITECTURE FINALE

Maintenant que les solutions techniques ont été validés, il est nécessaire de présenter la nouvelle architecture ainsi que les applications dans le projet eMagine :

#### 3.1. IMPLICATION DANS LE DÉVELOPPEMENT eMAGINE

L'architecture de développement se présente actuellement ainsi :



Le schéma précédent décrit l'architecture utilisé pour le développement :

### a. IDE : Eclipse

#### ➤ Avantages

Eclipse est un outil de développement gratuit (open source), tous les membres de l'équipe peuvent donc l'avoir sans problème. Par ailleurs, c'est un outil qui possède de nombreux plugins qui le rendent adaptatif et évolutif. Il existe par exemple, tous les plugins dont nous avons besoins pour le développement de nos différentes parties, notamment des plugins permettant de gérer Tomcat, Hibernate ou encore Subversion.

#### ➤ Inconvénients

Cet outil n'est pas basiquement orienté vers le développement d'applications J2EE. Il faut donc compter sur les différents plugins qui afin de parer à ce manque.

### b. Tests unitaires d'applications J2EE : Cactus

#### ➤ Avantages

L'environnement de tests unitaires d'applications J2EE Cactus est tout d'abord un outil gratuit (open source). Il est conceptuellement bien construit, et permet de tester l'ensemble des fonctionnalités d'une application web J2EE sur le principe des Junit. Il est nécessaire pour contrôler la qualité et l'avancée de la programmation de notre application.

#### ➤ Inconvénients

Cet outil est conséquent au niveau de sa mise en place. Il faut y penser dès le début du développement.

### c. Outils de déploiement : Ant

#### ➤ Avantages

une fois la configuration Ant terminée, les composants deployable d'eMagine seront automatiquement créés ( jar executable, schéma de base de données, etc ). Ce qui permet d'oublier la phase de déploiement.

#### ➤ Inconvénients

Demande un surcout de travail au début du développement.

### d. Outils de gestion de version : SubVersion

#### ➤ Avantages

Subversion va permettre d'éviter les conflits lorsqu'il y a plusieurs developpeur.

### **e. JUnit :**

#### ➤ **Avantages**

JUnit est complètement intégré à Eclipse. Il offre donc un moyen de test extrêmement pratique. De plus les test unitaires pour de gros projets sont absolument nécessaires, pour éviter les régressions. Le cas idéal serait que tout les tests JUnit soient écrits avant le développement. Cela permettrait de vérifier l'algorithme des classes implémentant les interfaces créés lors de l'élaboration cahier des charges technique.

#### ➤ **Inconvénients**

Il faut que tout les test JUnit soit rédigés avant que le développement proprement dit soit lancé. Demande du travail en plus, ainsi qu'une recherche pour obtenir des tests pertinants.

### **f. JMeter :**

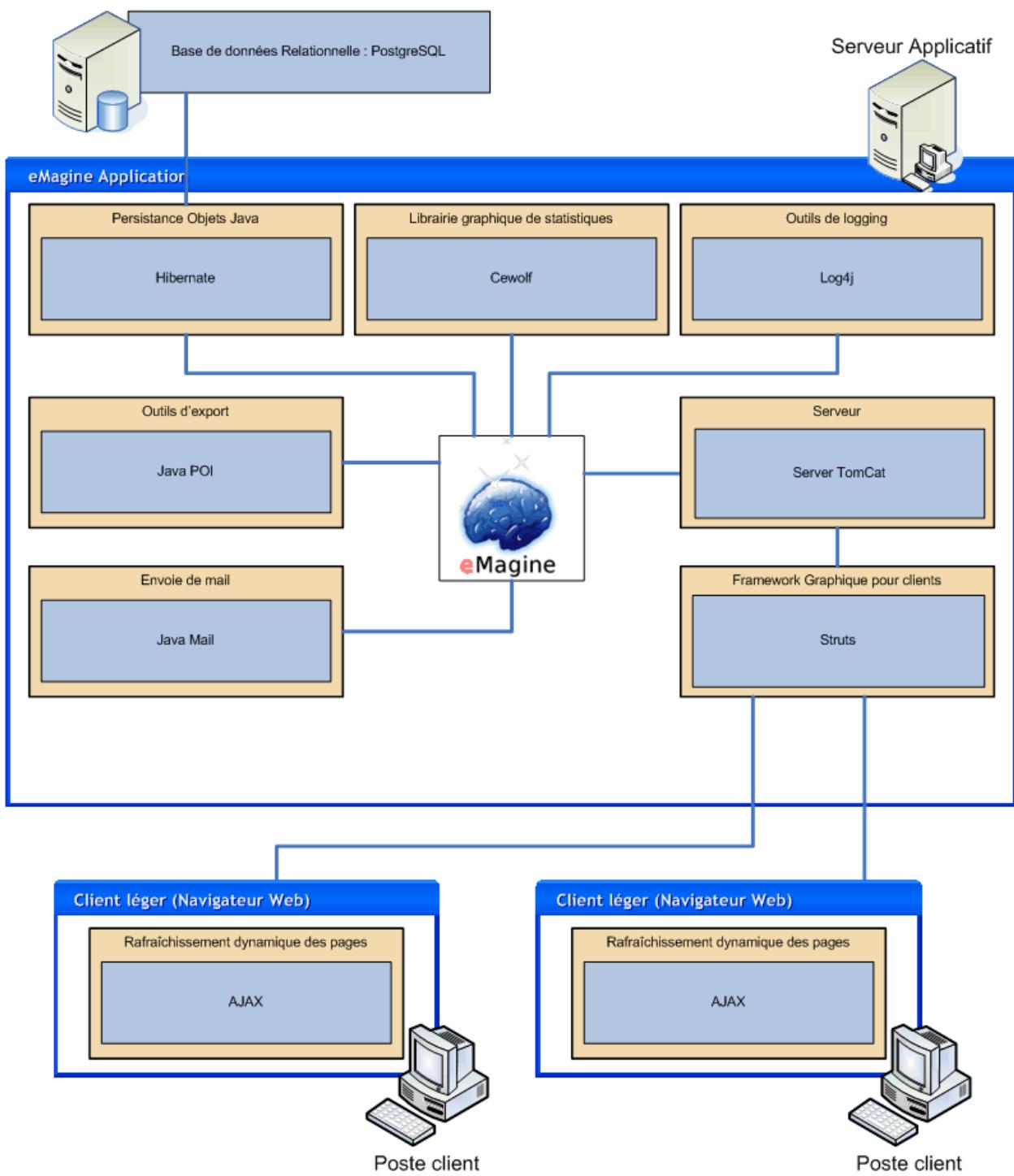
#### ➤ **Avantages**

JMeter va permettre de jauger les performance de l'application et de déterminer les goulots d'étranglement dans eMagine, eMagine étant une application travaillant à travers le réseau il est nécessaire d'optimiser les transactions à travers le réseaux.

#### ➤ **Inconvénients**

Cela demande une surcharge de travail supplémentaire pour la rédaction et le déroulement des tests.

### 3.2. IMPLICATION DANS L'ARCHITECTURE eMAGINE



## **a. Environnement de développement J2EE : Struts**

### ➤ Avantages

Struts permet de bénéficier d'un framework MVC pour développer notre application web J2EE. L'ensemble des balises proposées par Struts permet de développer la partie « Vue » (c'est à dire les pages web) plus proprement et plus rapidement. D'autre part, sa logique conceptuelle MVC permet de séparer le travail à réaliser entre plusieurs « métiers » spécifiques. Chacun de ces métiers n'ont pas à connaître la manière de fonctionner des autres métiers.

### ➤ Inconvénients

Nous n'avons pas trouvé de plugin Eclipse gratuit permettant d'avoir une logique assistée pour le développement Struts. Néanmoins, JST permet une première approche simplifiant le développement.

## **b. Serveur d'applications : Tomcat**

### ➤ Avantages

Tomcat est un serveur d'application J2EE gratuit (open source) et performant. Il s'intègre parfaitement avec le plugin JST d'Eclipse car il est standard aux normes J2EE.

### ➤ Inconvénients

Il ne gère pas les EJBs. Nous n'en avons pas besoin dans le cadre de notre application pûre web, mais pour la persistance des données offerte par les EJB, nous devons utiliser Hibernate.

## **c. Mise à jour dynamique d'une page : AJAX**

### ➤ Avantages

Le regroupement de technologies AJAX va nous permettre de mettre à jour nos pages internet de saisie de manière dynamique, sans recharger l'ensemble de la page. Ceci permet donc de s'affranchir du développement d'un client lourd pour les saisies de masse par exemple, où une liste doit être rafraîchie en fonction d'une autre.

Par ailleurs, il existe déjà des plugins Struts capables de générer des pages basées sur AJAX.

### ➤ Inconvénients

Au niveau du trafic réseau, cette solution reste tout de même plus gourmande qu'une solution de client lourd. Mais dans le cadre de notre application, notamment en prenant en compte le nombre d'utilisateurs limité, ceci est envisageable.

## **d. Génération de graphiques : Cewolf**

### ➤ Avantages

Cewolf permet de générer des graphiques en s'intégrant parfaitement dans une application web J2EE. Sa génération est robuste car basée sur l'API éprouvée de JFreeChart.

## e. API mails : JavaMail

### ➤ Avantages

JavaMail permet l'envoi d'emails de manière relativement simple au niveau de la programmation. L'API est bien structurée. Elle gère par ailleurs l'envoie de pièces jointes, ce qui est le but de notre utilisation des emails pour notre application.

## f. Génération de fichiers Excel : Java POI et HSSF

### ➤ Avantages

L'API HSSF du projet de Jakarta Java POI nous permet d'exporter les listes visibles dans notre application vers un format de fichier Microsoft Excel 97. C'est la seule API gratuite (open source) qui existe dans le monde Java.

### ➤ Inconvénients

Java POI permet de gérer les fichiers Microsoft Word, mais ne permet pas à priori de faire du publipostage, fonction dont nous avons besoin dans le cadre de notre application.

## g. Logueur : Log4j

### ➤ Avantages

Log4j est une bibliothèque de logging universelle permettant de tracer d'une seule et unique façon toutes les informations d'eMagine. C'est pour cette raison qu'un logueur a été mis en place.

### ➤ Inconvénients

La totalité de l'équipe eMagine doit se familiariser avec log4j.

## h. Hibernate

### ➤ Avantages

L'avantage majeur d'Hibernate est qu'il n'est pas nécessaire de développer et d'écrire le code de génération de la base de données. En effet la base de données est créée à partir du code source Java. De plus Les classes métiers n'ont pas besoin d'hériter d'une super classe, ce qui leur permet d'hériter d'une autre classe; en effet, en Java il est impossible de faire de l'héritage multiple.

### ➤ Inconvénients

La syntaxe des annotations est à connaître. Chaque classe persistante doit avoir un constructeur sans arguments. Néanmoins il peut être privé. Tout les types de retour pour les ensembles doivent être des interfaces (par exemple lorsqu'une fonction renvoyait une ArrayList, elle devra renvoyer son interface de base, c'est à dire Collection) pour pouvoir être persistante.

## CONCLUSION

L'étude technique a permis d'éclaircir les technologies existantes. Cela permet d'avoir une vue technique sur l'ensemble du projet, ainsi que sur l'architecture. Il prépare la réalisation du cahier des charges techniques en apportant les réponses techniques que n'avait pas le cahier des charges fonctionnel.

## RESSOURCES

Ce document s'est inspiré des sources suivantes :

Eclipse :

<http://perso.wanadoo.fr/jm.doudoux/java/dejae/indexavecframes.htm>

Cactus :

[http://66.249.93.104/search?q=cache:\\_Yq89ShdmEsJ:cactus.ressources-java.net/getting\\_started.jsp+cactus+tutorial&hl=fr&lr=lang\\_fr&client=firefox](http://66.249.93.104/search?q=cache:_Yq89ShdmEsJ:cactus.ressources-java.net/getting_started.jsp+cactus+tutorial&hl=fr&lr=lang_fr&client=firefox)

[http://web.archive.org/web/20041109120105/www.ressources-java.net/cactus/writing/howto\\_testcase.jsp](http://web.archive.org/web/20041109120105/www.ressources-java.net/cactus/writing/howto_testcase.jsp)

Ajax:

[http://developer.mozilla.org/fr/docs/AJAX:Premiers\\_pas.](http://developer.mozilla.org/fr/docs/AJAX:Premiers_pas.)

Outils Ant :

<http://www-sop.inria.fr/dream/seminaires/buildtools-25-05-05.pdf>

<http://ludovic.developpez.com/Ant/>

Junit :

<http://www.design-up.com/methodes/testsunitaires/junit/presentation.html>

[http://igm.univ-mlv.fr/~dr/XPOSE2003/JUnit\\_tour/](http://igm.univ-mlv.fr/~dr/XPOSE2003/JUnit_tour/)

Système de gestion de base de données :

<http://traduc.postgresqlfr.org/pgsql-8.0.4-fr>

<http://www.postgresql.org/>

<http://medias.obs-mip.fr/www/Activite/Formation/Informatique/Bdweb2/bdweb2005.html>

Gestion des sources :

[http://blog.nozav.org/docs/formation\\_svn.html](http://blog.nozav.org/docs/formation_svn.html)

Log4j :

<http://www.qos.ch/ac2001/F11-200.html>

<http://wiki.media-box.net/tutoriaux/java/log4j>

Jmeter :

<http://www-igm.univ-mly.fr/~dr/XPOSE2003/vbougard/index.php>