

Etudes Techniques

Aurélien PIERRON
Baptiste MATHUS
Hakima ZIDOURI
Michel VONGVILAY
Samuel MARMECHE



Historique

Version	Document créé par	Le
1	Samuel Marmeche	19 déc 2004

Version	Document modifié par	Motif	Le

Version	Document validé par	Le



Table des matières

I Études techniques.....	8
I.1 Étude de Java 1.5.....	9
I.1.1 Tiger.....	9
I.1.1.1 Fonctionnalités et améliorations.....	9
I.1.1.2 Des améliorations risquées.....	10
I.1.1.3 Intégration à Eclipse.....	10
I.1.2 Avis sur Tiger.....	11
I.2 Swing / SWT.....	11
I.2.1 Introduction.....	11
I.2.2 SWING.....	11
I.2.3 SWT.....	12
I.2.4 Comparatif.....	13
I.2.4.1 Performances	13
I.2.4.2 Portabilité.....	14
I.2.4.3 Apparence	14
I.2.4.4 Utilisation.....	14
I.2.4.5 Autre.....	15
I.2.5 Conclusion.....	15
I.3 Hibernate.....	15
I.3.1 Exemple simple.....	16
I.3.1.1 Configurer Hibernate.....	16
I.3.1.1.1 Fichier de configuration.....	16
I.3.1.1.2 Le code de mise en place.....	17
I.3.1.2 La classe.....	18
I.3.1.3 Créer la table correspondante.....	19
I.3.1.4 Mapping XML.....	19
I.3.1.5 Conclusion de l'exemple simple	20
I.3.2 Héritage.....	20
I.3.2.1 Les deux classes.....	21
I.3.2.2 Le fichier de mapping.....	21
I.3.2.3 Exemple de code.....	22
I.3.2.4 Récupérer toutes les indisponibilités.....	22
I.3.2.5 Conclusion sur l'héritage.....	23
I.3.3 Cacher Hibernate.....	24
I.3.3.1 Dao.....	24
I.3.3.2 Cacher la Session.....	24
I.3.3.3 Cacher les transactions.....	26
I.3.4 Conclusion sur Hibernate.....	27
I.4 iText/FOP.....	28
I.4.1 Introduction.....	28
I.4.2 iTEXT.....	28
I.4.2.1 Les étapes de création d'un document pdf.....	28
I.4.2.2 Les outils.....	29
I.4.2.2.1 Les objets Chunk, Phrase et Paragraph.....	29
I.4.2.2.2 Chunk :	29
I.4.2.2.3 Phrase :	29
I.4.2.2.4 Paragraph :	30
I.4.2.3 En-tête et Pied de page.....	30
I.4.2.4 Dessiner un tableau avec iText.....	31



I.4.2.4.1 L'objet Table.....	31
I.4.2.4.2 L'objet Cell.....	32
I.4.3 FOP.....	32
I.4.3.1 Architecture	33
I.4.3.2 Présentation de XSL.....	33
I.4.3.3 Structure d'un document XSL-FO.....	34
I.4.3.4 Les balises <fo:root>, <fo:flow>	34
I.4.3.5 Créer des tableaux.....	35
I.4.3.5.1 Les cellules qui prennent plusieurs cases ou colonnes.....	37
I.4.4 Conclusion sur iText et XML FOP.....	37
I.5 L'impression en Java	39
I.5.1 L'impression d'un composant.....	39
I.5.1.1 Processus d'impression.....	39
I.5.1.2 L'interface Printable.....	39
I.5.1.3 La classe PrinterJob.....	40
I.5.1.4 Configurer la page d'impression.....	42
I.5.2 Conclusion sur l'impression en Java.....	42
I.6 JavaMail (SMTP).....	43
I.6.1 Les classes utiles.....	43
I.6.1.1 La classe Session.....	43
I.6.1.2 La classe Message.....	44
I.6.1.3 La classe Transport.....	45
I.6.1.4 La classe Address	45
I.6.2 Exemple complet.....	46
I.7 LDAP.....	50
I.7.1 Introduction.....	50
I.7.2 Présentation.....	50
I.7.3 Description de l'arborescence d'un annuaire.....	50
I.7.3.1 Les différents champs.....	50
I.7.4 Accès à un serveur LDAP.....	51
I.7.4.1 Par un navigateur.....	51
I.7.4.2 Avec LDAPBrowser.....	51
I.7.5 LDAP de l'Université.....	52
I.7.5.1 Son arborescence.....	52
I.7.5.1.1 La branche Groups.....	53
I.7.5.1.2 La branche Users.....	55
I.7.6 Exemples de codes.....	58
I.7.6.1 LDAPIdentification.....	58
I.7.6.2 LDAPParameters.....	60
I.7.6.3 Recherche.....	60
I.8 Maven - Ant.....	62
I.8.1 Introduction.....	62
I.8.2 Maven ou Ant ?.....	62
I.8.2.1 Alors pourquoi avoir développé Maven ?.....	63
I.8.2.2 Alors, qu'est ce qui différencie Maven de Ant ?	64
I.8.2.3 Comment fonctionne Maven ?.....	66
I.8.2.4 Installation de Maven.....	66
I.8.2.5 Repository Maven.....	67
I.8.2.5.1 Structure générale.....	67
I.8.2.5.2 Propriétés du repository local ou distant	69
I.8.3 POM : Project Object Model (project.xml).....	70
I.8.3.1 Composition.....	70



I.8.3.2 Exemple.....	71
I.8.4 Utilisation de Maven.....	72
I.8.4.1 Concept.....	72
I.8.4.2 Générer un projet.....	73
I.8.4.3 Générer un site Web.....	75
I.8.4.4 Compiler un projet.....	76
I.8.5 Créer des GOALS (maven.xml).....	77
I.8.5.1 Structure du fichier 'maven.xml' et création d'un goal.....	77
I.8.5.2 Extension des goals.....	78
I.8.5.3 Utilisation d'une task Ant avec Maven.....	78
I.8.5.4 Écrire un plugin.....	79
I.8.6 Composition.....	79
I.8.6.1.1 Installer le plugin.....	80
I.8.6.1.2 Exemple de structure.....	80
I.8.7 Description de plugins utiles à Chronos.....	80
I.8.7.1 Les Plugins java et clean.....	80
I.8.7.2 Le plug-in Jar.....	80
I.8.7.3 Le plugin Jalopy.....	81
I.8.7.4 Le Plugin site.....	82
I.8.8 Conclusion.....	83
I.9 Continuous Testing.....	84
I.9.1 Introduction.....	84
I.9.2 Rappel sur les tests unitaires.....	84
I.9.2.1 L'intérêt des tests unitaires.....	84
I.9.3 Le plug-in Continuous Testing.....	84
I.9.3.1 Installation.....	85
I.9.3.2 Les objectifs de Continuous Testing.....	85
I.9.3.3 Activer les fonctions de Continuous Testing.....	85
I.9.3.4 Détails sur les apports de Continuous Testing.....	85
I.10 Conclusion sur continuous testing.....	87
I.11 API de Logging.....	87
I.11.1 Log4j.....	87
I.11.1.1 Les catégories.....	88
I.11.1.2 Les priorités.....	88
I.11.1.3 Les appenders.....	89
I.11.1.4 Les layouts.....	90
I.11.1.5 Configuration.....	91
I.11.2 Commons logging.....	92
I.12 Les design patterns dans Chronos.....	95
I.12.1 Introduction.....	95
I.12.2 Historique.....	95
I.12.2.1 Effectivity.....	95
I.12.3 Stockage des données.....	96
I.12.3.1 Hibernate.....	96
I.12.4 Logging.....	96
I.12.4.1 Commons-Logging.....	96
I.12.5 Masquage par interface.....	96
I.12.5.1 Factory.....	97
I.12.5.1.1 Exemple :	97
I.12.6 Simplifier l'utilisation d'une bibliothèque.....	98
I.12.7 Accès aux données : DAO.....	98
I.12.7.1 Définition.....	98



I.12.7.2 À quoi cela sert-il concrètement ?.....	99
I.12.7.3 Exemple d'implémentation.....	99
I.12.7.4 Conclusion.....	100
I.12.8 Conclusion.....	100
II Architecture.....	101
II.1 Présentation de l'architecture.....	101
II.2 RMI.....	102
II.2.1 Introduction.....	102
II.2.2 RMI.....	102
II.2.2.1 Objectif.....	102
II.2.2.2 Principe.....	103
II.2.2.3 Mise en oeuvre.....	103
II.2.2.3.1 Déclaration de l'interface.....	103
II.2.2.3.2 Implantation de l'interface.....	104
II.2.2.3.3 Génération stub/skeleton.....	106
II.2.3 Déclaration de l'interface.....	106
II.2.3.1.1 Réaliser des appels distants.....	109
II.2.4 Le callback.....	112
II.2.4.1 L'objet côté serveur.....	112
II.2.4.2 L'objet partagé côté client.....	114
II.2.4.3 L'objet réalisant le callback.....	115
II.2.5 Le plugin RMI pour Eclipse.....	116
II.2.5.1 Installation.....	116
II.2.5.1.1 Utilisation.....	117
II.2.6 Conclusion.....	117
II.3 PostgreSQL / MySQL.....	118
II.3.1 MySQL.....	118
II.3.2 PostgreSQL.....	118
II.3.3 Conclusion sur PostGreSQL / MySQL.....	119
III Introduction.....	128
IV Les technologies utilisées.....	130
IV.1 Applicatif.....	130
IV.1.1 TomCat / JSP.....	130
IV.1.1.1 Avantages.....	130
IV.1.1.2 Inconvénients.....	130
IV.1.2 Serveur MySQL.....	130
IV.1.2.1 Avantages.....	130
IV.1.2.2 Inconvénients.....	130
IV.1.3 Serveur PostGreSQL.....	130
IV.1.3.1 Avantages.....	130
IV.1.3.2 Inconvénients.....	130
IV.2 Programmation.....	131
IV.2.1 RMI.....	131
IV.2.1.1 Avantages.....	131
IV.2.1.2 Inconvénients.....	131
IV.2.2 JDBC.....	131
IV.2.2.1 Avantages.....	131
IV.2.2.2 Inconvénients.....	131
IV.2.3 Hibernate.....	131
IV.2.3.1 Avantages.....	131
IV.2.3.2 Inconvénients.....	131



IV.3 Note.....	131
V Proposition.....	132
V.1 Architecture TomCat / Hibernate.....	132
V.1.1 Organisation.....	132
V.1.2 Avantages.....	133
V.1.3 Inconvénients.....	134
V.2 Architecture JDBC / MySQL.....	135
V.2.1 Organisation.....	135
V.2.2 Avantages.....	135
V.2.3 Inconvénients.....	136
V.3 Architecture RMI / Hibernate.....	136
V.3.1 Organisation.....	136
V.3.2 Avantages.....	136
V.3.3 Inconvénients.....	137
V.4 Architecture XML.....	137
V.4.1 Organisation.....	137
V.4.2 Avantages.....	138
V.4.3 Inconvénients.....	138
VI Synthèse.....	139
VI.1 Comparatif des solutions.....	139
VI.2 Explication de la notation.....	140
VI.2.1 Installation / Mise en oeuvre / Administration.....	140
VI.2.1.1 Maintenance.....	140
VI.2.1.2 Déploiement.....	140
VI.2.1.3 Installation outils.....	140
VI.2.2 Technique.....	140
VI.2.2.1 Programmation.....	140
VI.2.2.2 Standard.....	140
VI.2.3 Fiabilité / Sécurité.....	141
VI.2.3.1 Fiabilité services.....	141
VI.2.3.2 Fiabilité données.....	141
VI.2.3.3 Stabilité application.....	141
VI.2.3.4 Sécurité données.....	141
VI.2.4 Performances / Ressources.....	142
VI.2.4.1 Rapidité exécution.....	142
VI.2.4.2 Ressources système utilisateur.....	142
VI.2.4.3 Ressources serveur.....	142
VI.2.4.4 Trafic généré.....	142



I Études techniques

Cette partie reprend les études techniques des différentes bibliothèques que nous serons amenés à utiliser. Nous essayons, lorsque plusieurs bibliothèques existent pour répondre à nos besoins, de les comparer puis d'étudier celle que nous avons retenue.

I.1 Étude de Java 1.5

Cette partie est destinée à donner un avis quant à l'utilisation de JAVA 1.5 pour le projet Chronos.

Il liste un certain nombre d'avantages à utiliser Tiger ainsi que les limites que cela pourrait engendrer.

I.1.1 Tiger

Tiger est le nom de la dernière version de J2SE, l'environnement de base de développement en langage Java.

I.1.1.1 Fonctionnalités et améliorations

Voici une liste brève de plusieurs améliorations qu'apporte Tiger sur les versions précédentes de java.

Amélioration et nouveautés	Description rapide
Autoboxing des types primitifs	Le compilateur se chargera de transtyper automatiquement la variable dans son type enrobé : boolean en Boolean, int en Integer, char en Character...
La nouvelle boucle for :	Nul besoin d'instancier explicitement un itérateur sur la collection.
Le mot clé enum	Pour chaque type décrit dans l'énumération, une classe complète (valeur et méthodes utilitaires) est générée.
Les méthodes à arguments variables	Ajout de l'ellipse "..." pour informer le compilateur que le nombre d'arguments est variable.
La nouvelle méthode printf()	Ajout d'une nouvelle méthode permettant de produire des messages formatés sur la sortie standard. Celle-ci vient en complément de l'actuelle méthode <code>System.out.print()</code> . Elle fonctionne comme la fonction <code>printf(...)</code> en C.



Amélioration et nouveautés	Description rapide
La gestion des flux standards : Scanner	L'API <i>Scanner</i> fournit des fonctionnalités de base pour lire tous les flux standards.
Les imports statiques	Cela permet de rendre visibles des méthodes et des variables statiques de la même manière que l'on importe des classes ou des interfaces.

Ne sont citées et décrites brièvement ci-dessus que quelques fonctionnalités de la version 1.5 de Tiger.

Vous pourrez trouver plus de détails sur le lien suivant :

<http://lroux.developpez.com/article/java/tiger/?page=sommaire>

I.1.1.2 Des améliorations risquées

Les améliorations citées ci-dessus visent à améliorer la lisibilité, la simplicité et la rapidité des développements des codeurs JAVA. Cependant, il ne faut pas négliger que certaines peuvent générer des désavantages non-négligeables, comme par exemple celles citées ci-dessous.

Fonctionnalités citées	Limitation
For	Il n'est pas possible d'utiliser cette boucle lorsque l'on souhaite modifier la collection, puisque aucun itérateur n'est accessible.
Imports statiques	Risque de conflit d'espaces de nommage.

I.1.1.3 Intégration à Eclipse

Un détail non négligeable est qu'il n'y a pas de version de compilateur d'Eclipse utilisant JDK 1.5.

Une nouvelle version d'Eclipse appelée « Cheetah », ou Eclipse version 3.1, devrait supporter les fonctionnalités de Tiger. Cependant cette version n'est pas encore disponible et nous n'avons encore que très peu d'information sur sa sortie.

I.1.2 Avis sur Tiger

Tiger offre bien des fonctionnalités qui pourraient faciliter la phase de programmation du projet Chronos. Cependant, le fait qu'elle ne soit pas intégrée à la dernière version stable d'Eclipse (version 3.0.1) risque de nous poser des problèmes et surtout de ralentir notre développement.

En conclusion, nous conseillons cependant d'utiliser JAVA 1.5 pour le projet Chronos sans forcément utiliser les fonctionnalités de Tiger.

I.2 Swing / SWT

I.2.1 Introduction

Il existe à l'heure actuelle deux solutions majeures d'API graphiques en Java: Swing et SWT.

Swing est une bibliothèque graphique multi-plateformes décrivant des composants graphiques complexes, complètement écrits en langage Java, et ne faisant pas appel aux composants de la plateforme, ce qui le rend totalement indépendant du système d'exploitation.

Par opposition, la stratégie de SWT repose sur l'utilisation des composants natifs de l'OS. SWT est malgré tout écrit à 100% en Java : les appels natifs servent uniquement à invoquer les composants de l'OS (mapping "one-to-one"). En conséquence, l'implémentation de l'API SWT n'est pas multi-plateforme mais l'application qui l'utilise l'est.

Le comparatif a pour objectif de donner des éléments de comparaison en faveur ou en défaveur de chacune des solutions pour pouvoir faire un choix. Les différents aspects abordés sont: les performances, la portabilité, l'apparence, l'utilisation

I.2.2 SWING

Swing fait partie de la bibliothèque Java Foundation Classes (JFC). C'est une API dont le but est similaire à celui de l'API AWT mais dont le mode de fonctionnement et d'utilisation est complètement différent. Swing a été intégrée au JDK depuis sa version 1.2. Cette bibliothèque existe séparément pour le JDK 1.1.

La bibliothèque JFC contient :

- l'API Swing : de nouvelles classes et interfaces pour construire des interfaces



graphiques

- Accessibility API :
- 2D API: support du graphisme en 2D
- API pour l'impression et le cliquer/glisser

Swing propose de nombreux composants dont certains possèdent des fonctions étendues, une utilisation des mécanismes de gestion d'événements performants (ceux introduits par le JDK 1.1) et une apparence modifiable à la volée (une interface graphique qui emploie le style du système d'exploitation Windows ou Motif ou un nouveau style spécifique à Java nommé Metal).

Les composants Swing forment une nouvelle hiérarchie parallèle à celle de l'AWT. L'ancêtre de cette hiérarchie est le composant **JComponent**. Presque tous ces composants sont écrits en pur Java : ils ne possèdent aucune partie native sauf ceux qui assurent l'interface avec le système d'exploitation : JApplet, JDialog, JFrame, et JWindow. Cela permet aux composants de toujours avoir la même apparence quelque soit le système sur lequel l'application s'exécute.

Tous les composants Swing possèdent les caractéristiques suivantes :

- ce sont des beans
- ce sont des composants légers (pas de partie native) hormis quelques exceptions.
- leurs bords peuvent être changés

La procédure à suivre pour utiliser un composant Swing est identique à celle des composants de la bibliothèque AWT : créer le composant en appelant son constructeur, appeler les méthodes du composant si nécessaire pour le personnaliser et l'ajouter dans un conteneur.

Les composants Swing utilisent des modèles pour contenir leurs états ou leur données. Ces modèles sont des classes particulières qui possèdent toutes un comportement par défaut.

I.2.3 SWT

SWT propose une approche intermédiaire : utiliser autant que possible les composants du système et implémenter les autres composants en Java. SWT est écrit en Java et utilise la technologie JNI pour appeler les composants natifs. SWT utilise autant que possible les composants natifs du système lorsque ceux sont présentés, sinon ils sont réécrits en pur Java. Les données de chaque composant sont aussi stockées autant que possible dans le composant natif, limitant ainsi les données stockées dans les objets Java correspondant.

Ainsi, une partie de SWT est livrée sous la forme d'une bibliothèque dépendante du système



d'exploitation et d'un fichier .jar lui aussi dépendant du système. Toutes les fonctionnalités de SWT ne sont implémentées que sur les systèmes où elles sont supportées (exemple, l'utilisation des ActiveX n'est possible que sur le portage de SWT sur les systèmes Windows).

Les trois avantages de SWT sont donc la rapidité d'exécution, des ressources machines moins importantes lors de l'exécution et un rendu parfait des composants graphiques selon le système utilisé puisqu'il utilise des composants natifs. Cette dernière remarque est particulièrement vraie pour des environnements graphiques dont l'apparence est modifiable.

Malgré cette dépendance vis à vis du système graphique de l'environnement d'exécution, l'API de SWT reste la même quelque soit la plate-forme utilisée.

En plus de dépendre du système utilisé lors de l'exécution, SWT possède un autre petit inconvénient. N'utilisant pas de purs objets java, il n'est pas possible de compter sur le ramasse miette pour libérer la mémoire des composants créés manuellement. Pour libérer cette mémoire, il est nécessaire d'utiliser la méthode dispose() pour les composants instanciés lorsque ceux ci ne sont plus utiles.

Ainsi SWT repose la problématique concernant la dualité entre la portabilité et les performances.

SWT repose sur trois concepts classiques dans le développement d'une interface graphique :

- Les composants ou contrôles (widgets)
- Un système de mise en page et de présentation des composants
- Un modèle des gestions des événements

I.2.4 Comparatif

I.2.4.1 Performances

SWT semble nettement plus fluide que Swing (et plus rapide), s'intègre mieux avec l'OS et présente l'intérêt de pouvoir éventuellement interagir avec des composants natifs. SWT utilisant des composants natifs ses performances sont très bonnes mais cela au prix de quelques DLL.

Il est vrai que sur une configuration matérielle moyenne, les composants natifs utilisés par SWT sont plus rapides que les composants complexes de Swing. Ceci se traduit par une meilleure rapidité de chargement des composants et surtout une meilleure réactivité aux



sollicitations de l'utilisateur.

Cependant les performances de SWING ont grandement été amélioré dans les dernières versions de la JVM. Pourtant, dans le cas particulier de l'affichage d'un très grand nombre de données, SWT n'est pas la plus performante. En effet, en SWT les données à afficher doivent systématiquement être copiées dans les composants natifs, alors qu'en Swing elles sont utilisées directement, sans recopie. De plus, l'aspect générique des mécanismes implémentés dans les viewers JFace jouent en défaveur de l'optimisation des performances. Pour s'en persuader, il suffit simplement de créer un TableViewer JFace ainsi qu'une JTable Swing affichant tous deux quelques milliers d'éléments, puis de comparer les temps de chargement.

I.2.4.2 Portabilité

Swing est directement inclut dans la JVM donc pas de problème de portabilité.

SWT est porté sur une bonne partie des OS, et pour le distribuer, il suffit d'inclure avec les librairies souhaités. Sa stratégie oblige l'utilisation de bibliothèques natives spécifiques à l'OS alors qu'il n'y a aucune contrainte particulière en Swing. Quelle que soit l'API, l'application est portable, mais dans le cas de SWT, il faudra préalablement installer les bibliothèques natives et les jars nécessaires sur le système du client ou mettre en place un système de chargement dynamique de ceux-ci.

I.2.4.3 Apparence

Swing propose une gestion de « look and feel » qui permet de se rapprocher au mieux des interface native. D'ailleurs il permet la redéfinition de l'apparence grâce au "Look'n'Feel" SWT est codé sur des composants natifs donc plus proche de l'interface habituelle de l'utilisateur. Son look est aussi plus professionnel.

I.2.4.4 Utilisation

Les récentes applications Swing prouvent qu'on peut obtenir un « look and feel » raisonnablement professionnel (JBuilder, TogetherJ), mais c'est plutôt difficile pour y arriver. Alors que ça vient automatiquement avec SWT.

Swing propose une réelle séparation Model - Vue (gestion des données indépendantes de la gestion de l'affichage). Alors que SWT n'as pas une réelle séparation Model – Vue mais la couche Jface permet d'y remédier.

Le GridLayout de SWT est plus facile à utiliser que le gridBagLayout de Swing.

I.2.4.5 Autre

SWT est plus jeunes que SWING, ce qui implique :

- SWING a déjà fait ses preuves sur sa robustesse. SWT semble moins robuste.
- SWT est moins documenté que SWING.

I.2.5 Conclusion

Tout ce que peut faire SWING peut être fait en SWT. Mais il semble que chacun possède ses avantages et ses inconvénients. Le choix repose donc sur les besoins de l'application.

Pour chronos, le plus intéressant serait d'utiliser SWING pour les raisons suivantes :

- SWING est plus portable. Ainsi le client Chronos pourra être utiliser par tous les environnements sans besoin de devoir installer des librairies spécifique à l'OS.
- SWT semble en générale plus performant. Mais c'est pas pour le cas de Chronos car il devra afficher des milliers d'éléments de la manière d'un JTable.
- « TableLayout » remplacera le « GridBagLayout », ce qui nous permettra de contourner la difficulté d'utilisation de ce « Layout ».
- Tous les développeurs de Chronos connaissent déjà SWING. Son utilisation ne nécessite donc pas de formation particulière.
- SWING est très bien documenté et il y a plus de communauté SWING qui pourra plus facilement nous aider ou nous fournir des informations en cas de problème. Un support a ne pas négliger.

SWING semble donc le plus adapté pour Chronos.

I.3 Hibernate

Cette partie présente l'étude de l'utilisabilité du framework Hibernate pour Chronos. Elle présentera rapidement le principe de fonctionnement de Hibernate. Nous irons ensuite plus loin dans les fonctionnalités fournies, nous démontrerons notamment la puissance de l'outil pour mapper des liens d'héritages. Nous finirons par une explication sur la solution adoptée destinée à masquer l'utilisation du framework à la plus grande partie possible de l'application.



I.3.1 Exemple simple

Cette partie présente un exemple d'utilisation simple d'Hibernate. L'objectif est de faire comprendre les principes de base de l'utilisation afin de partir ensuite de ces bases pour expliquer un exemple plus complexe.

Nous mapperons une seule classe pour une seule table pour montrer le bon fonctionnement du framework. Cet exemple utilisera en grande partie la documentation fournie dans la documentation de référence.

Faire gérer une classe par Hibernate se fait basiquement en 4 étapes dont la première n'est à faire que pour la première fois :

1. Configurer Hibernate : lui indiquer la base, le nom d'utilisateur, le mot de passe ;
2. Écrire la classe qu'on souhaite mapper ;
3. Créer la table correspondante ;
4. Effectuer le mapping XML de la classe.

Ces étapes sont détaillées ci-après.

I.3.1.1 Configurer Hibernate

I.3.1.1.1 Fichier de configuration

La configuration du framework est simple et peut être réalisée de plusieurs façons. Nous utiliserons l'une des façons classiques qui consiste à nommer un fichier hibernate.properties que nous placerons dans le classpath pour qu'Hibernate le détecte et l'utilise à l'exécution.

Hibernate a besoin de quelques paramètres évidents comme le driver jdbc à utiliser, la chaîne de connexion jdbc, l'utilisateur et le mot de passe à utiliser. Nous pouvons positionner un grand nombre de paramètres. Dans notre exemple, nous en positionnons également quelques uns liés à l'utilisation du pool jdbc¹ livré avec le framework et le dialecte² à utiliser.

```
hibernate.connection.driver_class = org.postgresql.Driver
```

1 Il est indiqué dans la documentation qu'il est déconseillé de l'utiliser en production

2 Information supplémentaire et optionnelle donnée à Hibernate pour lui permettre de « tuner » les dialogues qu'il effectue avec la base de données en fonction des particularités de celle-ci.


```
hibernate.connection.url = jdbc:postgresql://localhost/test_hibernate
hibernate.connection.username = batmat
hibernate.connection.password = batmat
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statement=50
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
```

I.3.1.1.2 Le code de mise en place

Nous récupérons la configuration avec laquelle nous allons pouvoir récupérer une SessionFactory. Nous indiquons à cette configuration quelle classe elle va gérer.

```
package de.berlios.chronos.technique.hibernate;

import net.sf.hibernate.HibernateException;
import net.sf.hibernate.Session;
import net.sf.hibernate.SessionFactory;
import net.sf.hibernate.Transaction;
import net.sf.hibernate.cfg.Configuration;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class MyTests
{
    private static Log log = LogFactory.getLog(MyTests.class);

    public static void main(String[] args) throws HibernateException
    {
        Configuration config = new Configuration().addClass(User.class);
        SessionFactory factory = config.buildSessionFactory();
        Session s = factory.openSession();

        Transaction tx = s.beginTransaction();

        // Travail avec la Session
        //...

        tx.commit();
    }
}
```



```
s.close();  
}  
}
```

I.3.1.2 La classe

Nous utiliserons une simple classe *User* qui aura pour attribut *id*, *name* et *password*. Il est important que cette classe soit un simple *JavaBean* et n'ai besoin d'hériter d'aucune classe ou d'implémenter aucune interface du framework *Hibernate* pour pouvoir être gérée.

```
package de.berlios.chronos.technique.hibernate;  
  
import org.apache.commons.logging.Log;  
import org.apache.commons.logging.LogFactory;  
  
public class User  
{  
    Log log = LogFactory.getLog(User.class);  
    int id;  
  
    String name;  
  
    String password;  
  
    public User()  
    {  
  
    }  
  
    /**  
     * @param _id the id of the user to be created  
     * @param _name the name of the user to be created  
     * @param _passwd the passwd of the user to be created  
     */  
    public User(int _id, String _name, String _passwd)  
    {  
        log.trace("Construction User("+_id+", "+name+", "+passwd+"");  
        this.id = _id;  
        this.name = _name;  
        this.password = _passwd;  
    }  
}
```

```
//All getters and setters  
...  
}
```

Note : Nous remarquons ici l'utilisation des commons logging, Nous ne détaillerons pas ici leur utilisation parce qu'ils le sont dans une autre partie de l'étude technique.

I.3.1.3 Créer la table correspondante

Nous ne détaillerons pas ici la création. Nous indiquons simplement la correspondance entre attributs de la classe et champs de la table. Il est à noter qu'il n'est nullement obligatoire qu'attributs et champs en correspondance soient nommés de façon identique.

```
CREATE TABLE utilisateur (  
  id integer NOT NULL,  
  name character varying(200) NOT NULL,  
  "password" character varying(200) NOT NULL  
);
```

I.3.1.4 Mapping XML

Ce mapping sera, comme l'exemple complet lui-même, assez simple. Pour qu'Hibernate le détecte, il faut nommer ce fichier du nom de la classe à mapper et le placer au même endroit que la classe qu'il mappe. Ainsi, au moment du `addClass(User.class)`, le framework parviendra à trouver ce fichier.

```
<?xml version="1.0"?>  
<!DOCTYPE hibernate-mapping  
  PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"  
  "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">  
  
<hibernate-mapping>  
  
  <class name="de.berlios.chronos.technique.hibernate.User" table="utilisateur">  
  
    <id name="id" type="int" unsaved-value="null" >  
      <column name="id" not-null="true"/>  
      <generator class="assigned"/>  
    </id>
```

```
<property name="name">
  <column name="NAME" not-null="true"/>
</property>

<property name="password"/>
</class>

</hibernate-mapping>
```

Se référer à la documentation de référence pour plus de détails, indiquons simplement que nous choisissons ici de nous occuper nous-mêmes de générer l'id de la classe (*generator class="assigned"*).

I.3.1.5 Conclusion de l'exemple simple

Nous avons montré ici un exemple simple d'utilisation d'Hibernate. Entrons maintenant un peu plus dans le vif du sujet avec un cas un peu plus concret et intéressant qui soit en relation avec Chronos.

I.3.2 Héritage

Hibernate fournit plusieurs façons de gérer la relation d'héritage qui peut exister entre deux classes. Nous choisissons ici d'utiliser le mapping une table/une classe afin de conserver un schéma relationnel très proche de la hiérarchie des classes.

Nous utiliserons ici un exemple un peu plus lié à Chronos pour illustrer ce principe. Dans Chronos, *Unavailability* est une super-classe de *Course*. Elle factorise par exemple les attributs de début et de fin de l'indisponibilité.

Nous simplifierons ici les objets de la façon suivante pour l'exemple :

- *Unavailability* ne stocke pas de période. Les attributs de date sont ceux destinés à gérer l'historisation par le design pattern Effectivity.
- *Course* n'a pour attribut qu'un nom.

L'objectif de cet exemple est à la fois de montrer à quel point il est simple de mapper une relation d'héritage avec hibernate, mais aussi de montrer concrètement comment l'historisation des cours et des indisponibilités pourra être gérée.

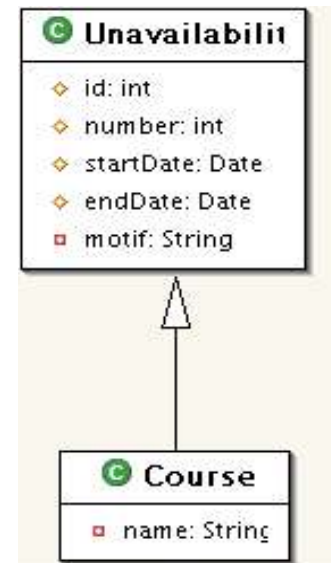
I.3.2.1 Les deux classes

Unavailability et *Course* sont deux simples JavaBeans.

Nous remarquerons ici l'utilisation de l'attribut *id* dans la classe *Unavailability*. C'est la clé primaire de l'objet. Notons que celui-ci jouera donc aussi le rôle de clé pour l'objet dérivé *Course*.

L'attribut *number* correspond à un numéro de cours. C'est cet attribut qui permet de retrouver le même cours à différentes périodes de validité. Par exemple, si nous souhaitons savoir l'historique d'un cours donné, il suffit d'utiliser ce numéro pour retrouver tous ses états successifs, l'utilisation d'une simple clause *ORDER BY* permettrait alors de les trier selon la date.

Ces périodes de validité sont stockées grâce à la date de début et date de fin. Ces périodes **ne se recouvrent pas**.



I.3.2.2 Le fichier de mapping

Nous avons utilisé ici l'attribut *joined-subclass* du mapping d'Hibernate. L'attribut *id* qui constitue la clé primaire d'une indisponibilité (cours ou "véritable" indisponibilité) est généré en demandant à Hibernate d'utiliser un générateur de type séquence. Comme le SGBD postgres sous-jacent le gère, il suffit de créer la séquence hibernate_sequence qu'Hibernate utilise par défaut.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>

    <class name="de.berlios.chronos.technique.hibernate.heritage.Unavailability" table="tb_Unavailability"
        polymorphism="implicit" >

        <id name="id" column="id">
            <generator class="sequence"/>
        </id>
```

```
<property name="number">
  <column name="number" not-null="true"/>
</property>

<property name="startDate">
  <column name="startDate" not-null="true"/>
</property>

<property name="endDate" />

<joined-subclass name="de.berlios.chronos.technique.hibernate.heritage.Course"
table="tb_Course">
  <key column="id"/>
    <property name="name"/>
  </joined-subclass>
</class>

</hibernate-mapping>
```

Un seul fichier suffit pour mapper les deux classes. En effet, c'est celui de la super-classe Unavailability, Course est mappée dans son fichier en tant que joined-class.

I.3.2.3 Exemple de code

Récupérer toute la base de données

La gestion de la relation d'héritage par Hibernate apporte une très grande puissance. Par exemple, la simple requête HQL `"from Object"` renverra dans une *List* tous les objets mappés de la BD puisque tout objet Java dérive de la classe *Object*.

I.3.2.4 Récupérer toutes les indisponibilités

De la même façon qu'il est possible de récupérer toute la base, nous pouvons ne récupérer qu'un type d'objet donné.

Il pourrait nous être intéressant par exemple, lorsque nous souhaiterons afficher l'emploi du temps d'un professeur, de pouvoir récupérer tous les moments où il est occupé. Pour cela, nous devons donc récupérer à la fois ses indisponibilités et les cours qu'il donne. Ceci nous est très simple grâce à la relation d'héritage qui existe entre ces deux objets. La requête "from

Unavailability" répondra exactement à ce besoin. Elle renverra dans une même liste, tous les cours et toutes les indisponibilités. Pour sélectionner seulement ceux d'un professeur donné, il suffira d'ajouter la clause de sélection adéquate et l'opération sera terminée.

Exemple de code destiné à afficher toutes les indisponibilités (pour les tests, les méthodes toString des deux classes ont été redéfinies pour en afficher les données) :

```
List unavailabilities = s.find("from Unavailability");

for (Iterator it = unavailabilities.iterator(); it.hasNext();)
{
    Unavailability un = (Unavailability) it.next();
    System.out.println(un + " | " + un.getClass());
}
```

Dans l'ordre avant d'effectuer cette itération, nous avons :

- créé une indisponibilité ;
- créé un cours ;
- modifié ce cours.

```
Indispo : (48,10,Wed Dec 29 23:08:23 CET 2004,null,Chui absent,) | class
de.berlios.chronos.technique.hibernate.heritage.Unavailability
Cours : (50,Wed Dec 29 23:08:30 CET 2004,null,motif,Voilà le cours 12 après modification) | class
de.berlios.chronos.technique.hibernate.heritage.Course
Cours : (49,Wed Dec 29 23:08:23 CET 2004,Wed Dec 29 23:08:30 CET 2004,motif,Voilà le cours 12) |
class de.berlios.chronos.technique.hibernate.heritage.Course
```

On obtient trois lignes. Une pour l'indisponibilité, une pour le cours avant modification (celui qui est historisé) et le nouveau cours qui représente la version actuelle du cours (après modification).

I.3.2.5 Conclusion sur l'héritage

Hibernate facilite grandement l'accès aux données. L'outil nous évite de devoir effectuer la tâche simple et rébarbative consistant à chaque fois à instancier et peupler nous même les objets auxquels nous souhaitons accéder.

L'inquiétude que nous pouvions avoir des capacités de l'outil provenait en grande partie des



spécificités d'un langage orienté objet, comme l'héritage par exemple. Nous avons montré ici que l'outil offre une grande simplicité de gestion de cette problématique.

I.3.3 Cacher Hibernate

Comme tout outil ou bibliothèque logicielle, il est intéressant de tenter de cacher son utilisation au maximum au reste de l'application. Ainsi, si nous désirons ensuite modifier ou supprimer l'utilisation de l'outil tiers, l'impact sur le code applicatif se trouve fortement réduit.

La problématique ici était donc de parvenir à ce que Chronos utilise des classes du framework Hibernate en le moins d'endroits possibles. Cette partie présente la stratégie adoptée pour ce faire.

I.3.3.1 Dao

La première étape pour une application est de trouver le moyen d'accéder aux données. Comme toujours, il est intéressant de centraliser cet accès aux données. Pour répondre à cela, nous avons utilisé le pattern Data Access Object.

L'objet n'étant pas ici d'expliquer ledit pattern, nous nous attacherons plutôt à expliquer les relations avec Hibernate. Pour plus d'explications sur ce design pattern, le lecteur se reportera à la partie dédiée à cela.

Comme l'accès aux données est centralisé par les DAOs en question, il restait donc à écrire du code capable de gérer l'objet Session d'Hibernate ainsi que les transactions de la façon la plus propre que nous estimions.

Pour la centralisation de l'accès à l'objet Session, nous avons utilisé le code de la classe HibernateUtil fourni dans la documentation de référence de l'outil.

I.3.3.2 Cacher la Session

Pour permettre l'accès à la même session, un objet non Thread-safe contrairement à la SessionFactory, dans tout un thread de l'application, nous avons donc simplement utilisé et très légèrement modifié le code fourni dans la documentation de référence³ :

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

3 http://www.hibernate.org/hib_docs/reference/fr/html/quickstart.html



```
private static final SessionFactory sessionFactory;

static
{
    try
    {
        // Crée la SessionFactory
        sessionFactory = new Configuration().configure()
            .buildSessionFactory();
        log.trace("Création de la SessionFactory");
    } catch (HibernateException ex)
    {
        log.error("Problème à la création de la SessionFactory");
        throw new RuntimeException("Problème de configuration : "
            + ex.getMessage(), ex);
    }
}

private static final ThreadLocal<Session> session = new ThreadLocal<Session>();

public static Session currentSession() throws HibernateException
{
    Session s = (Session) session.get();
    // Ouvre une nouvelle Session, si ce Thread n'en a aucune
    if (s == null)
    {
        s = sessionFactory.openSession();
        log.trace("Ouverture de la session s" + s);
        session.set(s);
    }
    return s;
}

public static void closeSession() throws HibernateException
{
    Session s = (Session) session.get();
    session.set(null);
    if (s != null)
    {
        log.trace("Fermeture de la session " + s);
        s.close();
    }
}
```



```
}  
}  
}
```

Extrait de la documentation de référence :

Non seulement cette classe s'occupe de garder `SessionFactory` dans un de ses attributs statiques, mais en plus elle garde la `Session` du thread courant dans une variable de type `ThreadLocal`. Vous devez bien comprendre le concept Java de variable de type `thread-local` (locale à un thread) avant d'utiliser cette classe utilitaire.

Une `SessionFactory` est `threadsafe` : beaucoup de threads peuvent y accéder de manière concurrente et demander une `Session`. Une `Session` est un objet non `threadsafe` qui représente une unité de travail avec la base de données. Les `Sessions` sont ouvertes par la `SessionFactory` et sont fermées quand le travail est terminé

I.3.3.3 Cacher les transactions

Pour permettre des opérations de ce type :

Essayer

```
Commencer la transaction  
Adao.saveA(paramètres);  
Adao.getAllA();  
Bdao.saveB(paramètres);
```

Si erreur Rollback transaction

Si OK Commit transaction

Il fallait permettre d'utiliser la gestion des transactions sans obliger le code externe aux DAOs à utiliser `HibernateUtil`, auquel cas les efforts destinés à cacher l'outil auraient été vains.

Pour permettre cela, la classe `TransactionManager` a été conçue. Le code résultant est très proche de celui de la classe `HibernateUtil`.

```
/**  
 * This class lets you control the transaction process.  
 */  
public class TransactionManager  
{  
    private static final ThreadLocal<Transaction> transaction = new ThreadLocal<Transaction>();
```



```
public static void beginTransaction() throws TransactionException
{
    try
    {
        Session s = HibernateUtil.currentSession();
        Transaction tx = s.beginTransaction();
        transaction.set(tx);

    } catch (HibernateException e)
    {
        throw new TransactionException("Unable to begin transaction", e);
    }
}

public static void commit() throws TransactionException
{
    try
    {
        transaction.get().commit();
        transaction.set(null);
    } catch (HibernateException e)
    {
        throw new TransactionException("Unable to commit transaction", e);
    }
}

public static void rollback() throws TransactionException
{
    try
    {
        transaction.get().rollback();
        transaction.set(null);
    } catch (HibernateException e)
    {
        throw new TransactionException("Unable to rollback transaction", e);
    }
}
}
```

I.3.4 Conclusion sur Hibernate

Cette étude nous a permis d'étudier Hibernate et de montrer que son utilisation pourra s'avérer



intéressante. Sa faible complexité de mise en place et sa facilité d'utilisation font que nous avons choisi de l'utiliser pour développer Chronos.

De plus, la renommée de l'outil laisse penser que nous ne devrions pas trouver de frein à son utilisation pour notre projet qui, somme toute, ne manipule pas de concepts hors du commun qui ne soit traité par l'outil.

Pour finir, le fait que le framework soit OpenSource et de grande qualité est rassurant quant aux problèmes que nous pourrions rencontrer. La taille de la communauté des développeurs qui l'utilisent est telle que nous ne devrions avoir aucun soucis à trouver de l'aide et de la documentation en de nombreux endroits si le besoin s'en faisait sentir.

I.4 iText/FOP

I.4.1 Introduction

Ce document décrit les possibilités existantes de génération d'un document PDF en java. Nous verrons donc les outils iText ainsi que FOP.

I.4.2 iTEXT

iText est une bibliothèque Java permettant la génération de fichiers au format PDF. Cette bibliothèque est disponible à l'adresse suivante :

<http://www.lowagie.com/iText/>

iText est disponible à partir de la JDK 1.2. Il est placé sous licence GPL et LGPL.

I.4.2.1 Les étapes de création d'un document pdf

Cinq étapes sont nécessaires à la création d'un document pdf :

1. **Création** d'un objet Document :

```
Document document = new Document (...);
```

2. **Création d'un Writer** sur Document

```
PdfWriter.getInstance(document, ...);
```

3. **Ouverture** du Document



```
document.open();
```

4. **Ajout** du contenu au Document

```
document.add(...);
```

5. **Fermeture** du Document

```
document.close();
```

Exemple simple : Création d'un document PDF contenant du texte

```
public class String2Pdf {  
    public static void main(String[] args) {  
        Document document = new Document();  
        String out = "Test.pdf";  
        try {  
            PdfWriter.getInstance(document, new FileOutputStream(out));  
            document.open();  
            document.add(new Paragraph(args[0]));  
        } catch (DocumentException de) {  
            System.err.println(de.getMessage());  
        } catch (IOException ioe) {  
            System.err.println(ioe.getMessage());  
        } finally {  
            document.close();  
            System.out.println(out+ " a été créé correctement.");  
        }  
    }  
}
```

I.4.2.2 Les outils

I.4.2.2.1 Les objets Chunk, Phrase et Paragraph

I.4.2.2.2 Chunk :

C'est une petite partie de texte qu'on l'on peut ajouter. On y définit la couleur, le style etc...

```
Chunk chunk = new Chunk();
```

I.4.2.2.3 Phrase :

C'est une série de Chunk, on peut définir le leading avec une Phrase.



```
Phrase phrase = new Phrase();  
phrase.add(chunk);
```

I.4.2.2.4 Paragraph :

C'est une série de Phrase, on peut aussi modifier l'indentation et l'alignement du texte à partir d'un paragraph.

```
Paragraph paragraph = new Paragraph();  
paragraph.add(phrase);
```

I.4.2.3 En-tête et Pied de page

Itext donne la possibilité d'ajouter au document pdf des en-têtes et pieds de pages.

L'objet `HeaderFooter` est destiné à ce genre d'action. La procédure est simple : on utilise des **Chunk**, des **Phrase** et des **Paragraph**.

1. Création d'un Paragraph ;
2. Création d'un HeaderFooter ;
3. Ajout du Paragraph à L'HeaderFooter.

Exemple de programme : pour un en-tête centré

```
public HeaderFooter createHeader() {  
    // On crée le Paragraph  
    Paragraph paragraph = new Paragraph();  
    paragraph.add("MON ENTETE");  
  
    //Creation du HeaderFooter  
    HeaderFooter header = new HeaderFooter(paragraph,false);  
    header.setAlignment(HeaderFooter.ALIGN_CENTER);  
    return header;  
}
```

Exemple de programme : pour un pied de page

```
public HeaderFooter createFooter() {  
    //On crée un ou deux phrases  
    Phrase phrase = new Phrase();
```



```
Phrase phrase1 = new Phrase();
phrase.add(new Phrase("Mon pied de page"));
phrase1.add(new Phrase("2eme texte"));
// On crée notre paragraph
Paragraph paragraph = new Paragraph();
//On ajoute nos phrases au Paragraph
paragraph.add(phrase);
paragraph.add(phrase1);
Paragraph paraFooter = new Paragraph();
paraFooter.add(paragraph);
//Creation du HeaderFooter
HeaderFooter footer = new HeaderFooter(paraFooter, false);
return footer;
}
```

I.4.2.4 Dessiner un tableau avec iText

I.4.2.4.1 L'objet Table

```
Table table = new Table(...);
```

L'objet Table est destiné à créer un tableau. Il compte un certain nombre de méthodes intéressantes dont :

SetPadding() et SetSpacing() : pour le remplissage et l'espacement des cellules

SetBorderColor() : pour la couleur de la bordure du tableau

SerBorderWidth() : pour définir la taille de la bordure

Exemple :

```
Table table = new Table(11);
table.setBorderWidth(2);
table.setBorderColor(new Color(0, 0, 255));
table.setPadding(5);
table.setSpacing(3);
```



I.4.2.4.2 L'objet Cell

```
Cell cellule = new Cell(...);
```

L'objet Cell vient en complément de l'objet Table, il sert à créer des cellules dans une Table.

Deux méthodes intéressantes chez l'objet Cell :

setRowspan() et setColspan() : définit le nombre de lignes et de colonnes occupées par la cellule en question.

Exemple :

```
Cell cell = new Cell(sCell);  
cell.setColspan(2);  
cell.setHorizontalAlignment(Cell.ALIGN_CENTER);  
cell.setVerticalAlignment(Cell.ALIGN_CENTER);
```

Un exemple est disponible sur le repository du projet Chronos à l'adresse :

<https://opensvn.csie.org/chronos/Dev/ChronosTests/src/de/berlios/chronos/technique/export/tableEnPDF/EmploiDuTemps.java>

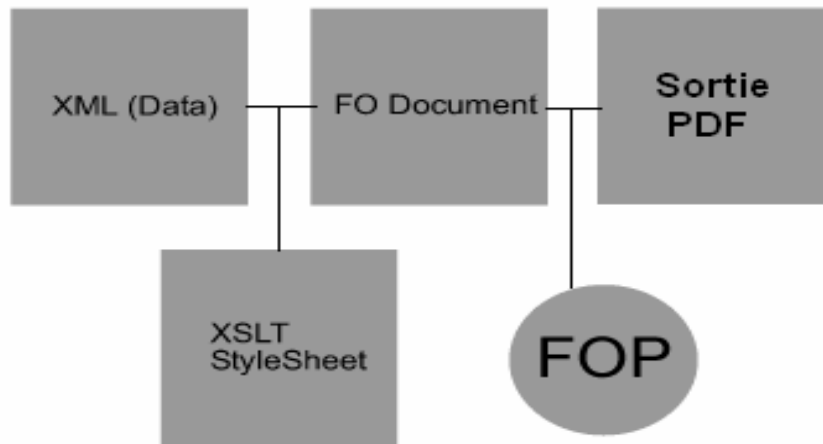
I.4.3 FOP

FOP (Formatting Objects Processor) est une bibliothèque Java permettant l'écriture de différents types formats. Sa cible principale reste les fichiers au format PDF.

Plus précisément, FOP est un processeur XSL acceptant en entrée un fichier XML comportant des Formatting Objects pour produire en sortie un document au format PDF.

Site : <http://xml.apache.org/fop/>

I.4.3.1 Architecture



I.4.3.2 Présentation de XSL

XSL (eXtensible Stylesheet Language) est constitué de deux parties :

- XSL Transformations (XSLT) et
- XSL Formatting Objects (XSL-FO)

Une feuille de style sépare la structure logique et la présentation d'un document de son contenu. Une feuille de style XSLT définit une transformation d'un document XML vers un autre type de document XML.

XSL-FO est un langage XML pour faire de la mise en page de façon beaucoup plus précise que ce qu'il est possible de faire avec HTML et CSS.

Un document XSL-FO définit plusieurs choses primordiales pour créer un document imprimable de haute qualité :

- Des informations sur la taille physique de la page (A3, ...)
- Des informations sur les marges, en-tête et pieds de page...
- Des informations sur les polices, leur taille, couleur...
- Le texte à imprimer, avec les éléments de mise en page.

Convertir un document XML en fichier PDF se fait en deux étapes élémentaires :

- Utiliser une feuille de style XSLT pour transformer le document XML en un fichier XSL-FO. Pour cela on invoque un processeur XSLT avec le document XML et la feuille



de style XSLT.

- On se sert d'un programme tel que FOP pour convertir le document XSL-FO en fichier PDF.

I.4.3.3 Structure d'un document XSL-FO

I.4.3.4 Les balises <fo:root>, <fo:flow> ...

L'élément <fo:root> :

Il s'agit de l'élément racine du document XSL-FO. Voici un exemple qui montre une structure typique de document :

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  --- --- ---
  le reste du document
  --- --- ---
</fo:root>
```

L'élément <fo:flow> :

L'élément **fo-flow** définit le contenu qui sera placé dans les marges, polices, etc. courantes. Jusqu'à présent nous n'avons fait que définir la page dans laquelle nous voulions travailler, il s'agit maintenant d'insérer du contenu.

L'insertion du contenu se fait par le logiciel (ou la bibliothèque) de rendu. FOP va calculer les saut de lignes, sauts de pages, colonnes... automatiquement ou en fonction de paramètres spécifiés.

L'exemple qui suit énonce que ce <fo:flow> contiendra le contenu du region-body du document.

```
<fo:flow flow-name="xsl-region-body">
```

L'élément <fo:block> :

L'élément **fo:block** est l'élément de base pour formater un bloc de texte. Il est similaire au <p> du HTML. Un élément **fo:block** provoque toujours un retour chariot.



L'élément **<fo:inline>** :

L'élément **<fo:inline>** définit des nouvelles propriétés supplémentaires à l'intérieur d'un **<fo:block>**.

Si l'on souhaite par exemple mettre en italique plusieurs mots dans un paragraphe, il faut utiliser **<fo:inline>**. L'utilisation de **<fo:block>** à la place fera apparaître les mots en italique comme un paragraphe séparé.

Il existe encore beaucoup d'éléments qui ne seront pas détaillés dans cette documentation tel que :

- **<fo:layout-master-set>** : il spécifie des définitions de pages ;
- **<fo:simple-page-master>** : il définit le layout d'une page particulière ;
- **<fo:region-body>** : définit la zone principale au centre de la page (car XSL-FO définit cinq régions dans une page) ;
- **<fo:page-sequence>** : il définit une séquence de layouts à utiliser dans le document.

I.4.3.5 Créer des tableaux

Les tableaux peuvent être assez compliqués à créer, cela dépend de la complexité du tableau, néanmoins le fonctionnement n'est pas sans rappeler celui des tableaux en HTML. Voici une liste des balises utilisées pour les tableaux :

- **<fo:table>**
- **<fo:table-body>**
- **fo:table-column>**
- **<fo:table-row>**
- **<fo:table-cell>**
- **<fo:table-caption>**

La principale difficulté des tableaux XSL-FO est que l'on doit spécifier la taille de toutes les colonnes du tableau. Voici un exemple typique de tableau :

```
<fo:table table-layout="fixed">
  <fo:table-column column-width="140pt"/>
  <fo:table-column column-width="140pt"/>
  <fo:table-body>
    <fo:table-row>
      <fo:table-cell border-style="solid" border-color="black" border-
```

```
width="2pt" padding-before="2pt" padding-after="2pt" padding-start="4pt"
padding-end="4pt">
    <fo:block>Du texte ici</fo:block>
</fo:table-cell>
<fo:table-cell border-style="solid" border-color="black" border-
width="2pt" padding-before="2pt" padding-after="2pt" padding-start="4pt"
padding-end="4pt">
    <fo:block>plus de texte </fo:block>
</fo:table-cell>
</fo:table-row>
<fo:table-row>
    <fo:table-cell border-style="solid" border-color="black" border-
width="2pt" padding-before="2pt" padding-after="2pt" padding-start="4pt"
padding-end="4pt">
    <fo:block>dans une cellule a part</fo:block>
</fo:table-cell>
    <fo:table-cell border-style="solid" border-color="black" border-
width="2pt" padding-before="2pt" padding-after="2pt" padding-start="4pt"
padding-end="4pt">
    <fo:block>Dans une autre cellule</fo:block>
</fo:table-cell>
</fo:table-row>
</fo:table-body>
</fo:table>
```

Cela semble bien compliqué, mais on devra se servir d'une feuille de style XSLT pour générer ce document.

Le véritable travail du tableau se fait dans les balises **table-cell**, où l'on doit à chaque fois définir un ensemble de propriétés qui ne s'héritent pas de balises de plus haut niveau, et doivent donc être spécifiées à chaque fois.

L' exemple spécifie deux types d'attributs : border et padding.

Des propriétés comme **border-style**, **border-width**, **border-color** définissent les propriétés des bordures de chaque cellule, cependant il en existe encore beaucoup d'autres.

Les propriétés padding définissent l'espace qui doit être laissé entre le contenu de la cellule et les bords de la cellule, on peut spécifier un espacement différent à droite, à gauche, en haut et

en bas.

I.4.3.5.1 Les cellules qui prennent plusieurs cases ou colonnes

Il existe encore beaucoup de propriétés pour les tableaux, nous allons donc nous pencher sur la manière de créer des cellules qui prennent plus d'une case ou plus d'une colonne : ce qui s'appelle du **spanning**.

Les propriétés, combinées avec les propriétés de bordure et de padding permettent de réaliser la plupart des tableaux d'usage courant.

Les propriétés **number-columns-spanned** et **number-rows-spanned** définissent le nombre de colonnes ou de rangées qu'occupera une cellule. Une fois que l'on a défini qu'une cellule prendra un certain nombre de colonnes ou de rangées, les éléments **table-cell** qui suivent sont placés au prochain emplacement disponible.

Par exemple si le tableau comporte trois colonnes, et que sur une ligne donnée, la première cellule occupe deux colonnes (cellule 4), la deuxième cellule de cette ligne (cellule 5) sera dans la troisième colonne.

Le principe est le même pour les colonnes.

Exemple :

colonne1	colonne2	colonne3
cellule1	cellule2	cellule3
cellule4		cellule5
cellule6	cellule7	cellule8
	cellule9	cellule10

I.4.4 Conclusion sur iText et XML FOP

Les outils iText et XML FOP offrent tous deux de grandes possibilités pour créer des fichiers PDF en java.

IText semble complet et simple d'utilisation cependant, les formats des fichiers XML doivent répondre rigoureusement au format iText.

Fop est un outil très puissant dans la mesure où les possibilités sont très grandes tout en gardant séparées les données de la mise en forme. La difficulté réside, cependant, dans la diversité du langage XSL-FO.



Nous utiliserons donc l'outil iText pour le projet Chronos dans la mesure où il répond tout à fait à nos attentes et est facile d'utilisation.

I.5 L'impression en Java

Ce document décrit une méthode d'impression dans le langage JAVA.

Dans le cadre du projet Chronos, deux possibilités d'offrent à nous : nous pouvons soit choisir d'imprimer l'emploi du temps depuis son format PDF, puisque ce dernier est converti en PDF à chaque publication, soit imprimer le composant *emploi du temps* lui même.

Nous décrivons ici la méthode d'impression d'un composant.

I.5.1 L'impression d'un composant

On utilise l'API « *Print* » du paquetage **java.awt.print**.

Cette API permet d'imprimer des textes contenus dans un composant texte (*TextArea*, *JTextPane*, *JEditorPane*, etc...) aussi bien qu'une interface utilisateur complète.

I.5.1.1 Processus d'impression

Le processus d'impression est l'ensemble des actions que doit réaliser le programme pour envoyer des données vers une imprimante.

- Indiquer au système d'impression quels objets de l'interface sont à imprimer : dans notre cas il s'agira de notre composant *emploi du temps* ;
- Appeler la fonction d'impression pour l'objet à imprimer ;
- Indiquer au système les données concernant l'impression: nombre de pages, format de la page, etc... ;
- La fonction d'impression envoie les données vers l'imprimante (méthode *paint()* comme pour les sorties à l'écran sur le contexte graphique *java.awt.Graphics*).

I.5.1.2 L'interface Printable

Pour être imprimable, une classe doit implémenter l'interface *Printable*. L'interface *Printable* définit deux constantes et une méthode :

- *PAGE_EXISTS* ;
- *NO_SUCH_PAGE* ;
- *print(Graphics graphics, PageFormat pageFormat, int pageIndex)*.

La méthode *print* est la méthode qui est appelée par une instance de la classe *PrinterJob*.



Les paramètres de *print* sont :

- *graphics* une référence au contexte graphique ;
- *pageFormat* une référence à une instance de *PageFormat* permettant de définir, ou connaître la taille du papier, ses marges, et son orientation ;
- *pageIndex* le numéro de la page à imprimer. La numérotation commence à 0.

Exemple de programme :

```
import java.awt.*;
import java.awt.print.*;
public class Exemple extends JPanel implements Printable
{
    public Exemple() {}
    public int print( Graphics g, PageFormat format, int nb)
    {
        // Imprime au maximum 2 pages
        if(nb>=2)
            return NO_SUCH_PAGE;
        // impression en rouge et au milieu de la page du numéro de page
        g.setColor(Color.red);
        g.setFont(new Font("arial", Font.BOLD, 30));
        FontMetrics font= g.getFontMetrics();
        java.awt.geom.Rectangle2D rect = font.getStringBounds(""+pi,g);
        int x = (int) (format.getWidth()- rect.getWidth())/2;
        int y = (int) (format.getHeight() - rect.getHeight())/2;
        g.drawString(""+ nb, x, y);
        return PAGE_EXISTS;
    }
}
```

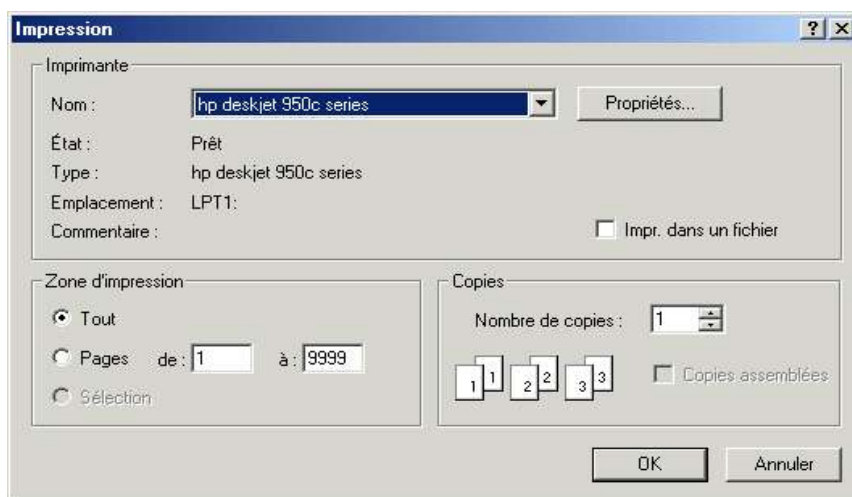
Lorsque *print* retourne `NO_SUCH_PAGE` l'instance de *PrintJob* arrête d'appeler la méthode *print*.

I.5.1.3 La classe PrinterJob

La classe *PrinterJob* permet de créer des jobs d'impression et de les lancer :


```
PrinterJob printJob = PrinterJob.getPrinterJob();
if (pageFormat == null)
{
    pageFormat = printJob.defaultPage();
}
printJob.setPrintable( l'objet Printable, pageFormat);
if (printJob.printDialog())
{
    // le dialogue d'impression
    try
    {
        printJob.print();
    }
    catch (PrinterException exception)
    {
        JOptionPane.showMessageDialog(this, exception);
    }
}
```

Ce programme lance une fenêtre d'impression dans laquelle, on choisit l'imprimante vers laquelle on souhaite envoyer l'impression :



Quelques méthodes intéressantes :

- *PrinterJob()* : Un *PrinterJob* est créé par un appel à la méthode de classe static *getPrinterJob* ;
- *void cancel()* : Supprime un *PrinterJob* ;
- *PageFormat defaultPage()* : Créer un *PageFormat* par défaut ;
- *PageFormat defaultPage(PageFormat page)* : clone le *PageFormat* en paramètre ;
- *int getCopies()* : retourne le nombre de copies à imprimer ;
- *abstract String getJobName()* : retourne le nom du document à imprimer ;
- *static PrinterJob getPrinterJob()* : Créer et retourne un *PrinterJob* ;
- *abstract String getUserName()* : retourne le "user name" ;
- *abstract boolean isCancelled* : retourne true si le *PrinterJob* va être supprimé dès que possible ;
- *abstract PageFormat pageDialog (PageFormat page)* : affiche une boîte de dialogue de configuration de la page ;
- *abstract void print* : lance l'impression ;
- *abstract boolean printDialog* : affiche une boîte de dialogue d'impression ;
- *abstract void setCopies(int copies)* : positionne le nombre de copies à imprimer ;
- *abstract void setJobName(String jobName)* : donne un nom au document à imprimer ;
- ...

I.5.1.4 Configurer la page d'impression

```
pageFormat = printJob.pageDialog(pageFormat);
```

Cette ligne de code permet de configurer les caractéristiques de la page.

I.5.2 Conclusion sur l'impression en Java

Le langage JAVA offre une des moyens d'impression de composant JAVA grâce à l'interface *Printable*. Dans le cadre de notre projet Chronos, nous étudierons donc les méthodes afin d'imprimer notre composant *emploi du temps*.



I.6 JavaMail (SMTP)

JavaMail est une API fournie 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.

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`.

I.6.1 Les classes utiles

I.6.1.1 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 on utilise 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 debugger

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.

I.6.1.2 La classe Message

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

Lorsque vous avez votre 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, mais seul celui ci sera utilisé : **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 un chaîne qui contient le type **MIME** du message.



Un message peut contenir plusieurs objets il faut alors utiliser un **MultiPart Message**. Cet objet à son tour contient des 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.

I.6.1.3 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** tel qu'expliqué ci dessus lorsqu'il y a plusieurs mails à envoyer car on peut maintenir la connexion avec le serveur ouverte pendant les envois.

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

I.6.1.4 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 vous désirez 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");
```

Vous devez aussi préciser l'adresse de l'expéditeur du message. Pour identifier cet expéditeur, on peut utiliser les méthodes **setFrom()** et **setReplyTo()** de la classe **Message**.

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

Si votre message doit montrer plusieurs adresses d'expéditeur, alors on utilise 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);
```

I.6.2 Exemple complet

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

Il y a une vérification afin de vérifier la présence du fichier à joindre. Si aucun fichier n'est fourni, alors on utilise la méthode `setText()` de la classe `Message`. 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);  
}
```



I.7 LDAP

I.7.1 Introduction

Ce document constitue une présentation de l'annuaire LDAP présent à l'Université ainsi qu'une présentation de l'API JNDI. L'annuaire LDAP

I.7.2 Présentation

LDAP (Lightweight Directory Access Protocol) désigne une base de données ainsi que le protocole pour y accéder. Il est optimisé en lecture : on va très souvent y rechercher des informations, mais très rarement les modifier.

LDAP est principalement utilisé comme annuaire simplifiant la gestion des profils de personnes et de ressources, favorise l'interopérabilité des systèmes d'informations à travers le partage de ces profils, et améliore la sécurité d'accès aux applications.

LDAP est un standard destiné à normaliser l'interface d'accès aux annuaires.

I.7.3 Description de l'arborescence d'un annuaire

Une base de données LDAP ressemble à une arborescence de fichiers sous UNIX, avec quelques différences :

- La racine est à droite ;
- Le séparateur n'est pas un '/' mais une ',' ;
- La racine porte un nom particulier.

Exemple :

```
dn: uid=apierr01, ou=Users, ou=Etudiant, dc=univ-mlv, dc=fr
```

signifie qu'il y a une arborescence de répertoires (au sens Unix) : `fr/univ-mlv/Etudiant/Users`, contenant un identifiant `uid=apierr01`.

I.7.3.1 Les différents champs

Les champs différents importants sont :



- **dn** (Distinguished Name) : c'est l'équivalent du nom de fichier ;
- **dc** (domain component) : morceau du nom de domaine ;
- **ou** (Organizational Unit) : unité de l'organisation ;
- **cn** (Common Name) : le nom complet ;
- **sn** (Surname) : nom de famille ;
- **givenName** : prénom ;
- **uid** (*userid*) : il s'agit d'un identifiant unique obligatoire.

I.7.4 Accès à un serveur LDAP

I.7.4.1 Par un navigateur

Les URLs LDAP ([RFC2255](#)) permettent aux clients Web d'avoir un accès direct au protocole LDAP. La syntaxe est de la forme :

```
ldap[s]://<hostname>:<port>/<base_dn>?<attributes>?<scope>?<filter>
<base_dn>      : DN de l'entrée qui est le point de départ de la recherche
<attributes>   : les attributs que l'on veut consulter
<scope>        : la profondeur de recherche dans le DIT à partir du
                  <base_dn> : "base" | "one" | "sub"
<filter>       : filtre de recherche, par défaut (objectClass=*)
```

exemples :

```
ldap://ldapetud/uid=apierr01, ou=Users, ou=Etudiant, dc=univ-mlv, dc=fr?sn?
(uid=apierr01)
```

I.7.4.2 Avec LDAPBrowser

Si vous voulez visualiser le contenu du LDAP de l'Université, Baptiste à installé et configuré LDAPBrowser.

Pour y accéder :

```
cd ~/../bmathus/pub/ldapbrowser/
```

```
java -jar lbe.jar
```

Choisir école parmi ses configurations.

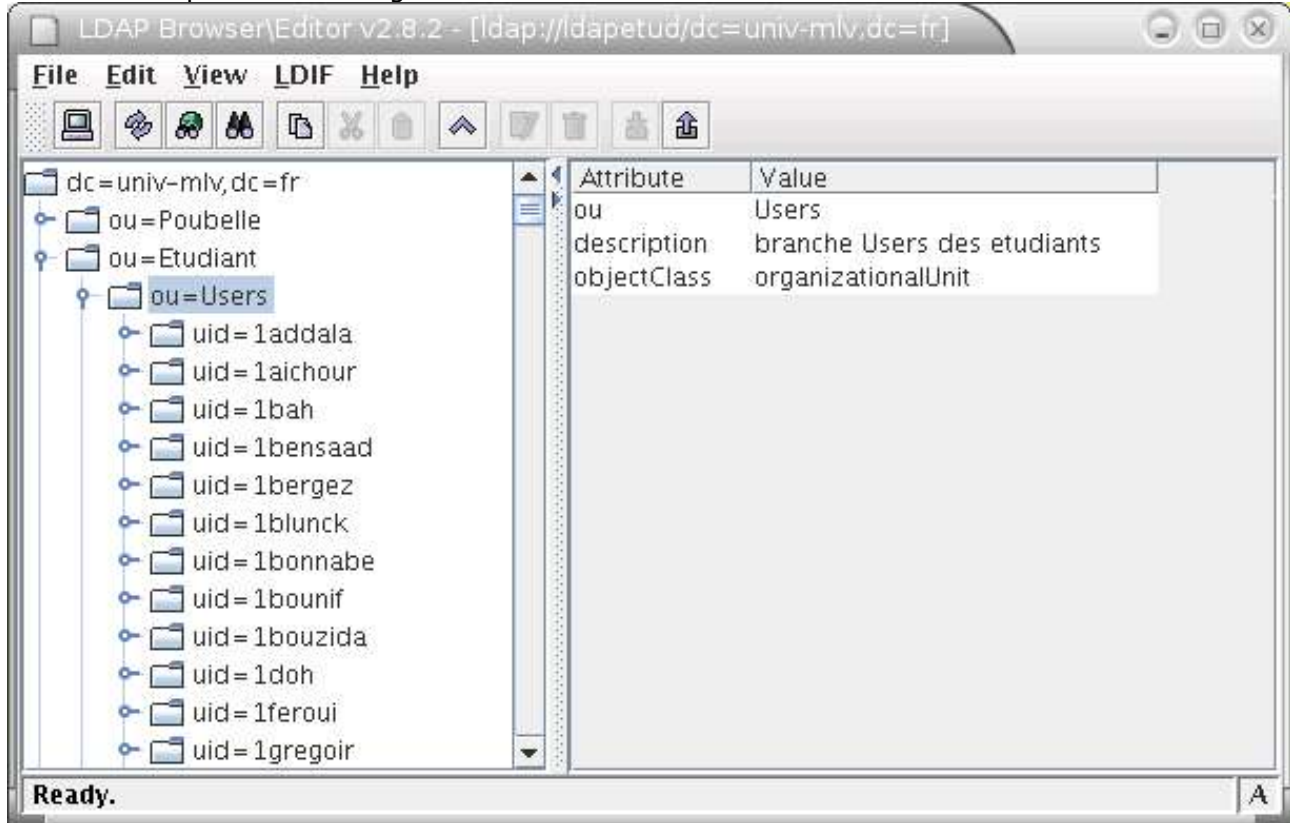


Illustration 1 Branche Users du LDAP de l'Université

I.7.5 LDAP de l'Université

I.7.5.1 Son arborescence

Voici l'arborescence de l'annuaire LDAP de l'Université :

```
etudiant
|
+ dc=univ-mlv,dc=fr
|
+ ou=Poubelle
+ ou=Etudiant
|
+ ou=Users
```

- + ou=Samba
- + ou=Groups

Donc les branches qui nous intéressent sont : Users et Groups.

I.7.5.1.1 La branche Groups

C'est dans cette branche que l'on retrouve l'ensemble des informations concernant les différentes promotions. Par exemple pour les IR, elles sont appelées igin0X où X représente un entier de 1 à 6.

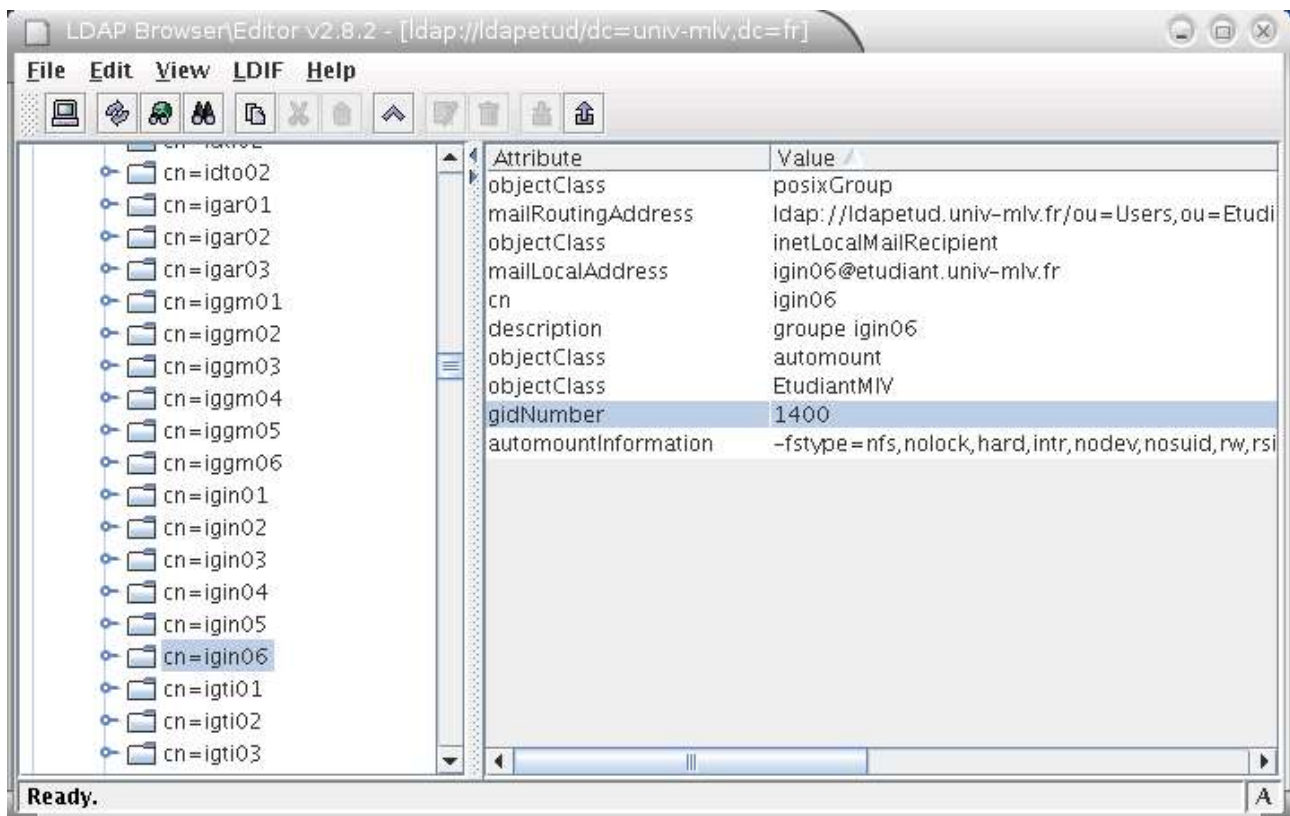


Illustration 2 Ensemble des informations disponibles pour une promotion

Les informations pertinentes :



Les informations pertinentes accessibles sont :

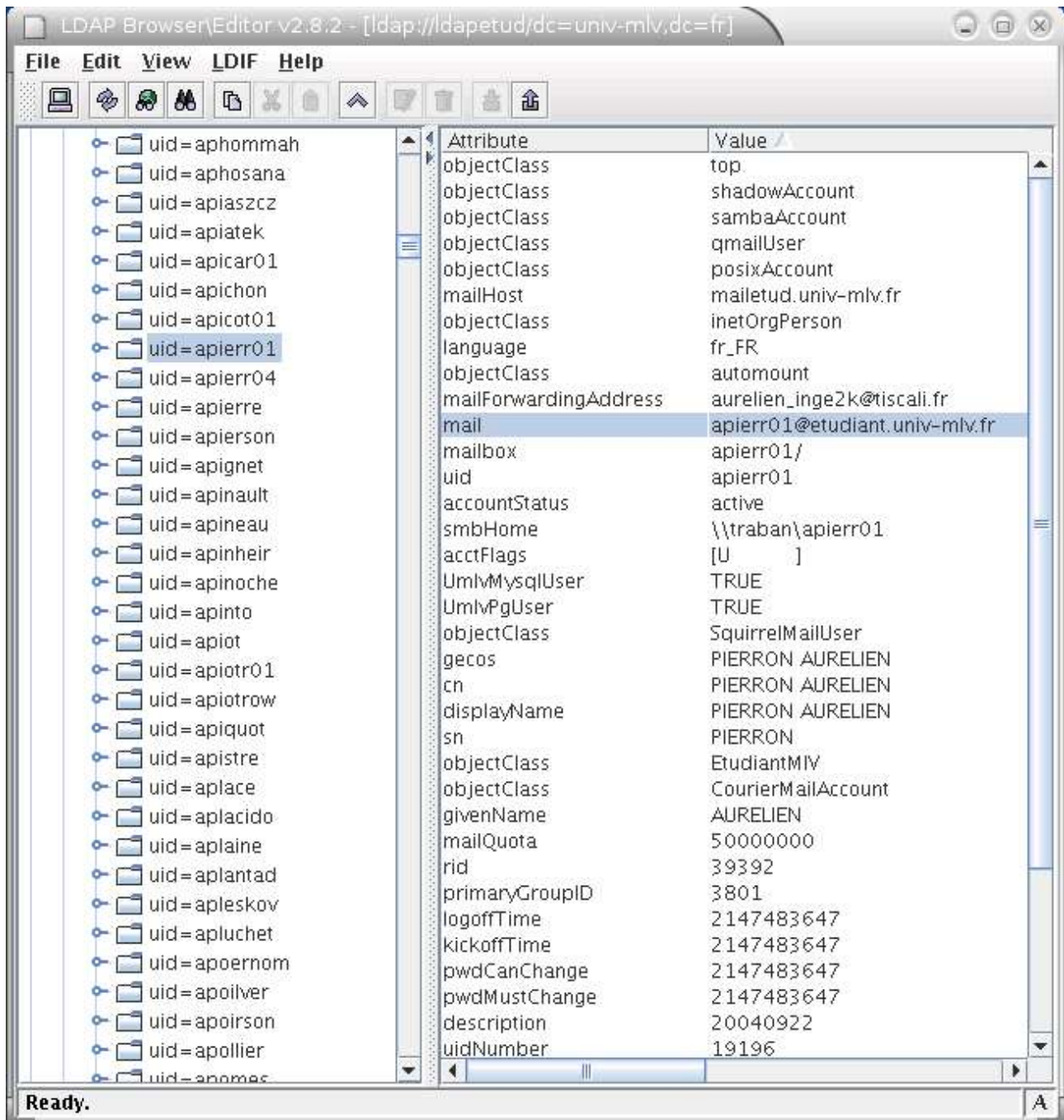
- **cn** : son identifiant unique ;
- **gidNumber** : numéro attribué à la promotion.

Ces informations sont utiles pour le développement de Chronos.



I.7.5.1.2 La branche Users

C'est dans cette branche que l'on retrouve l'ensemble des informations concernant un enseignant ou apprenti. Cependant la plupart des enseignants sont indexés par leur nom de famille alors que les apprentis sont indexés par leur login. On peut noter toute fois des exceptions sur les enseignants tel que J. Cervelle qui est indexé comme un apprenti.

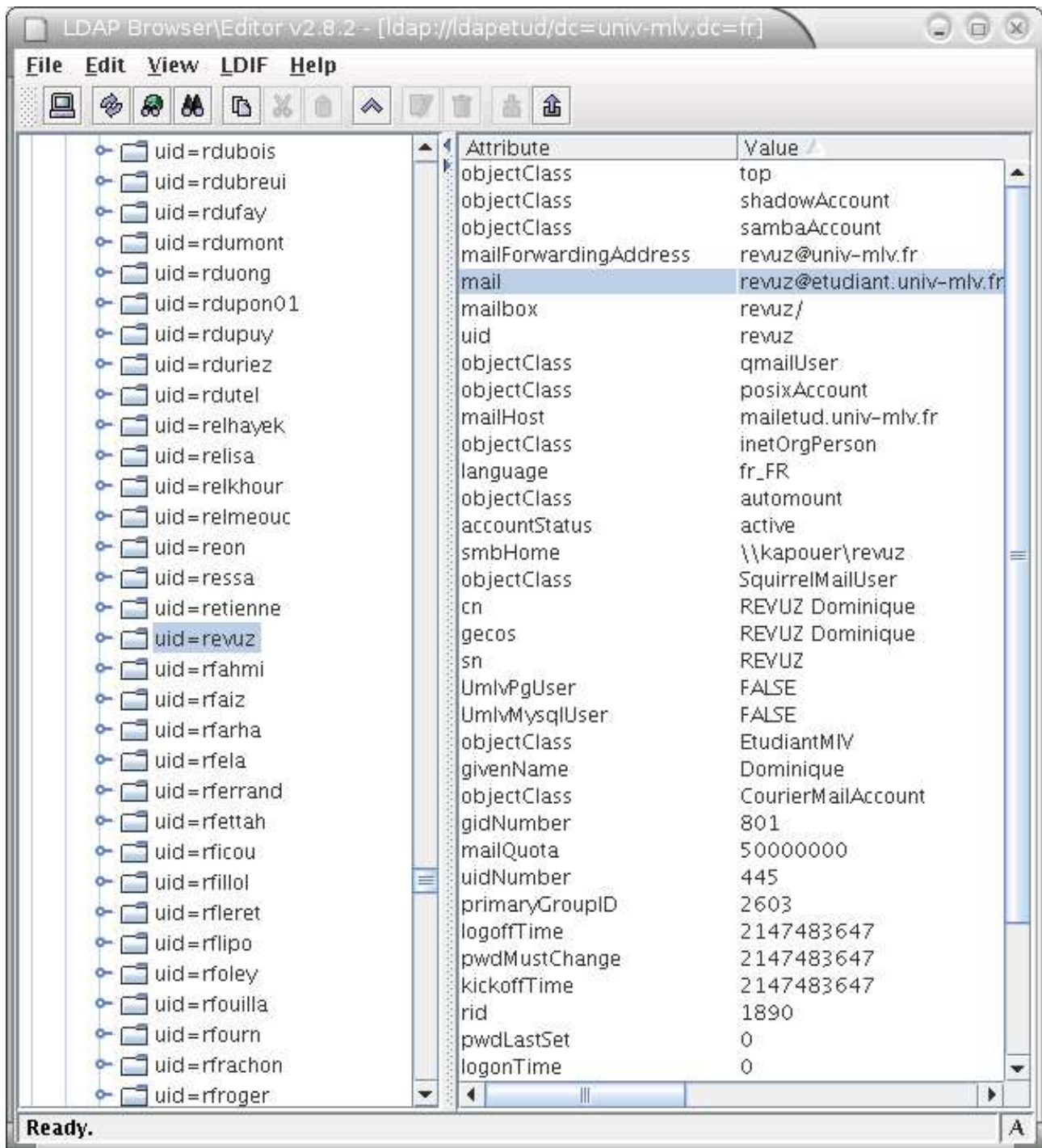


The screenshot shows the LDAP Browser/Editor v2.8.2 window. The left pane displays a tree of LDAP entries, with 'uid=apierr01' selected. The right pane shows the details of this entry in a table format.

Attribute	Value
objectClass	top
objectClass	shadowAccount
objectClass	sambaAccount
objectClass	qmailUser
objectClass	posixAccount
mailHost	mailetud.univ-mlv.fr
objectClass	inetOrgPerson
language	fr_FR
objectClass	automount
mailForwardingAddress	aurelien_inge2k@tiscali.fr
mail	apierr01@etudiant.univ-mlv.fr
mailbox	apierr01/
uid	apierr01
accountStatus	active
smbHome	\\traban\apierr01
acctFlags	[U]
UmlvMySQLUser	TRUE
UmlvPgUser	TRUE
objectClass	SquirrelMailUser
gecos	PIERRON AURELIEN
cn	PIERRON AURELIEN
displayName	PIERRON AURELIEN
sn	PIERRON
objectClass	EtudiantMIV
objectClass	CourierMailAccount
givenName	AURELIEN
mailQuota	50000000
rid	39392
primaryGroupID	3801
logoffTime	2147483647
kickoffTime	2147483647
pwdCanChange	2147483647
pwdMustChange	2147483647
description	20040922
uidNumber	19196

Illustration 3 Ensemble des informations disponibles pour un apprenti

Pour un enseignant les informations sont identiques, seul l'uid n'est pas construit de la même manière (pour la plupart des enseignants) :



Attribute	Value
objectClass	top
objectClass	shadowAccount
objectClass	sambaAccount
mailForwardingAddress	revuz@univ-mlv.fr
mail	revuz@etudiant.univ-mlv.fr
mailbox	revuz/
uid	revuz
objectClass	qmailUser
objectClass	posixAccount
mailHost	mailletud.univ-mlv.fr
objectClass	inetOrgPerson
language	fr_FR
objectClass	automount
accountStatus	active
smbHome	\\kapouer\revuz
objectClass	SquirrelMailUser
cn	REVUZ Dominique
gecos	REVUZ Dominique
sn	REVUZ
UmlvPgUser	FALSE
UmlvMysqlUser	FALSE
objectClass	EtudiantMIV
givenName	Dominique
objectClass	CourierMailAccount
gidNumber	801
mailQuota	50000000
uidNumber	445
primaryGroupID	2603
logoffTime	2147483647
pwdMustChange	2147483647
kickoffTime	2147483647
rid	1890
pwdLastSet	0
logonTime	0

Il n'y pas de façon formelle de différencier un apprenti d'un enseignant.

Les informations pertinentes :



Les informations pertinentes accessibles sont :

- **uid** (*userid*) : un identifiant unique ;
- **sn** : nom de famille de la personne ;
- **givenName** : prénom de la personne ;
- **cn** : Nom prénom de la personne (peut être reconstituée avec sn et givenName) ;
- **gidNumber** : numéro de la promotion de la personne (non visible ici) ;
- **mail** : e-mail de la personne.

Ces informations sont utiles pour le développement de Chronos.

I.7.6 Exemples de codes

Sur le repository dans `Dev\Exemples\LDAP`, vous trouverez plusieurs fichiers :

- `LDAPIdentification.java` : classe permettant de vérifier un mot de passe ;
- `LDAPParameters.java` : paramètres nécessaires aux deux classes ;
- `Recherche.java` : exemple de recherche dans l'annuaire.

Il n'y a pas de bibliothèque à charger. Les requêtes LDAP sont basées sur l'API JNDI qui est fournie en standard avec le [J2SE](#) depuis sa version 1.3.

I.7.6.1 LDAPIdentification

Cette classe ouvre simplement une mire de connexion et affiche si le mot de passe est bon ou non.

Afin de vérifier le login et le mot de passe d'une personne, il faut créer un contexte initial. Pour cela, il faut initialiser une `HashMap` à l'aide différente `Properties`.

Exemple :



```
import javax.naming.*;
import javax.naming.directory.*;

...

public class LDAPIdentification
{
    private Hashtable authEnv;
    ...

    public LDAPIdentification (final String sUserName, final String sPassWord)
    {
        // Set up environment for creating initial context
        authEnv      = new Hashtable(11);

        sBase        = "ou=Users,ou=Etudiant,dc=univ-mlv,dc=fr";
        sDn           = "uid=" + sUserName + "," + sBase;

        authEnv.put (Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi ldap.LdapCtxFactory");
        authEnv.put (Context.PROVIDER_URL, LDAPParameters.LDAP_URL);
        authEnv.put (Context.SECURITY_AUTHENTICATION, "simple");
        authEnv.put (Context.SECURITY_PRINCIPAL, sDn);
        authEnv.put (Context.SECURITY_CREDENTIALS, sPassWord.getBytes());
    }

    public void connectLDAP () {
        try {
            DirContext authContext = new InitialDirContext(authEnv);
        }
        catch (AuthenticationException authEx) {
            // Identification mauvaise
        }
        catch (NamingException namEx) {
```



I.7.6.2 LDAPParameters

C'est une classe statique dans laquelle il y a des informations nécessaires aux deux classes, comme l'adresse du serveur LDAP par exemple.

I.7.6.3 Recherche

Cette classe permet de faire une recherche dans le LDAP. Il y a plusieurs types de recherches et pour le moment seule une recherche basique est présentée. Elle permet de rechercher le `gidNumber` de la promotion et d'afficher tous les apprentis ayant ce `gidNumber`.

Pour effectuer une recherche, il est nécessaire d'obtenir un Context. Une fois obtenu, il faut lancer une recherche dans l'arborescence à l'aide d'un Attributes.

Exemple :



```
public DirContext getDirContext () throws Exception
{
    Hashtable authEnv = new Hashtable (11);
    authEnv.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.ldap.LdapCtxFactory");
    authEnv.put (Context.PROVIDER_URL, "ldap://ldapetud:389");

    // Creation du context initial
    DirContext ctx = new InitialDirContext (authEnv);
    return ctx;
}

public void doBasicSearch () throws Exception
{
    DirContext ctx = getDirContext ();
    // ignore attribute name case
    Attributes matchAttrs = new BasicAttributes (true);

    // -- Recherche du gidNumber du groupe ign06 --
    matchAttrs.put (new BasicAttribute ("cn", "igin06"));
    NamingEnumeration answer = ctx.search ("ou=Groups,ou=Etudiant,dc=univ-
mlv,dc=fr", matchAttrs);

    // Traitement des résultats
}
```



I.8 Maven - Ant

I.8.1 Introduction

Maven est un outil « **open source** » de Apache Jakarta. Il permet de **faciliter et d'automatiser la gestion et la construction** d'un projet java. Le site officiel est : <http://maven.apache.org>.

Le premier but de Maven est de permettre aux développeurs de connaître rapidement l'état global du développement du projet. C'est dans ce but que Maven :

- Facilite le processus de construction
- Fournit un système de construction uniforme
- Fournit des informations utiles sur le projet
- Fournit clairement les grandes lignes directrices de développement
- Fournit les éléments nécessaires pour faire des tests complets
- Fournit une vision cohérente et globale du projet
- Permet d'ajouter de nouvelles fonctionnalités de façon transparente

Il permet notamment :

- d'automatiser la compilation, les tests unitaires et le déploiement des applications du projet (jar, war)
- de gérer les dépendances des bibliothèques nécessaires au projet
- de générer des documentations du projet : rapport de compilation et des tests unitaires, javadoc
- de générer un site web complet du projet

Maven est basé sur le **concept de « Project Object Model »** (POM). Le développement et la gestion du projet sont contrôlés depuis le POM. Maintenant beaucoup de projet de Apache Jakarta utilise Maven.

I.8.2 Maven ou Ant ?

Il n'y a rien que Maven fasse que Ant ne puisse faire. De plus Ant et Maven sont tous deux



développés par l'équipe « Apache Jakarta ».

I.8.2.1 Alors pourquoi avoir développé Maven ?

La structure de développement et le déploiement (EARs, WARs et EJB-JARs) des projets J2EE sont aujourd'hui beaucoup plus standardisés. Ant est plus ancien que Maven. Ainsi Maven propose les fonctionnalités de ANT mais aussi de nombreuses autres qui répondent plus aux standards et aux besoins des développeurs d'aujourd'hui. Maven est aussi plus flexible car il permet de créer plus facilement ses plugings (en jelly) à la différence de Ant qui reste relativement statique. Enfin, les scripts Ant ne sont pas réutilisables entre projets, alors que le but de Maven est justement de fournir des fonctionnalités réutilisables

De plus, le concept POM permet d'avoir une compréhension plus simple et plus complète du projet. Contrairement à Ant, Maven offre des outils de gestion de projet de haut niveau. En effet, en examinant un script Ant, il est difficile de trouver les dépendances du projet ou toutes autres informations telles que les développeurs, la version ou le site web.

Pour ma part, je trouve que Maven est plus simple à écrire. A titre de comparaison, la création d'un projet simple prend 2 à 3 lignes avec Maven contre une douzaine de lignes avec ANT.

On peut grossièrement faire une analogie avec le C pour Ant et Java (ou C++) pour Maven. D'ailleurs, les plugins Maven sont un peu comme les bibliothèques JDK. Maven, comme Java, réduit la complexité par rapport à Ant. Cependant il y a quelques tâches que seul le C peut faire, mais pour la majorité des développements, Java suffit. Ce que je voulais souligner c'est que Maven et Ant ciblent deux différents aspects du problème de construction de projet. Ainsi Maven s'adapte pour la plupart des projets, cependant Ant est plus adapté pour les projets très spécifiques et très complexes.

Pour **résumer**, l'utilisation de Maven par rapport à Ant fournit trois principaux avantages:

1. L'aptitude de créer des goals (fonctions) réutilisables
2. Une large bibliothèques de goals prédéfinis (les plugins)
3. Les scripts sont plus flexibles grâce au langage Jelly



I.8.2.2 Alors, qu'est ce qui différencie Maven de Ant ?

Pour cela, je vais juste expliquer les différences des composants présents dans les deux outils.

	Maven	Ant
Ensemble de tâches	Goal	Target
Fichier standard de construction	project.xml, maven.xml	build.xml
Ordre de lecture des propriétés	<ol style="list-style-type: none">1. <code>\${maven.home}/bin/driver.properties</code>2. <code>\${project.home}/project.properties</code>3. <code>\${project.home}/build.properties</code>4. <code>\${user.home}/build.properties</code>5. Propriétés systèmes définies par l'option ligne de commande -D <p>La dernière définition l'emporte.</p>	<ol style="list-style-type: none">1. Propriétés systèmes définies par l'option ligne de commande -D2. Propriétés chargées par le tag <code><property></code> <p>La première définition l'emporte.</p>
Règle de construction et langage	Les règles de construction sont plus dynamiques (similaires à un langage de programmation). Ce sont des exécutable XML basés sur Jelly script, qui inclue les Ant task et JSLT.	Les règles de constructions sont plus ou moins statiques (à moins que vous utilisiez le tag <code><script></code>)
Extention du langage	Les plugins sont écrits en Jelly (langage XML).	Les plugins sont écrits en langage Java.



Prérequis des Goal/Target	Attribut prerequisite du tag <goal>	Attribut depends du tag <target>
Extensibilité des règles de construction	Les « Build goals » sont extensibles en définissant <preGoal> et <postGoal> .	Elles ne sont pas vraiment extensibles; on peut tout de même simuler les <preGoal> et <postGoal> en utilisant les <script> .

Tableau 1. Comparaison Maven et Ant

I.8.2.3 Comment fonctionne Maven ?

Voici une schématisation des principales composantes de Maven :

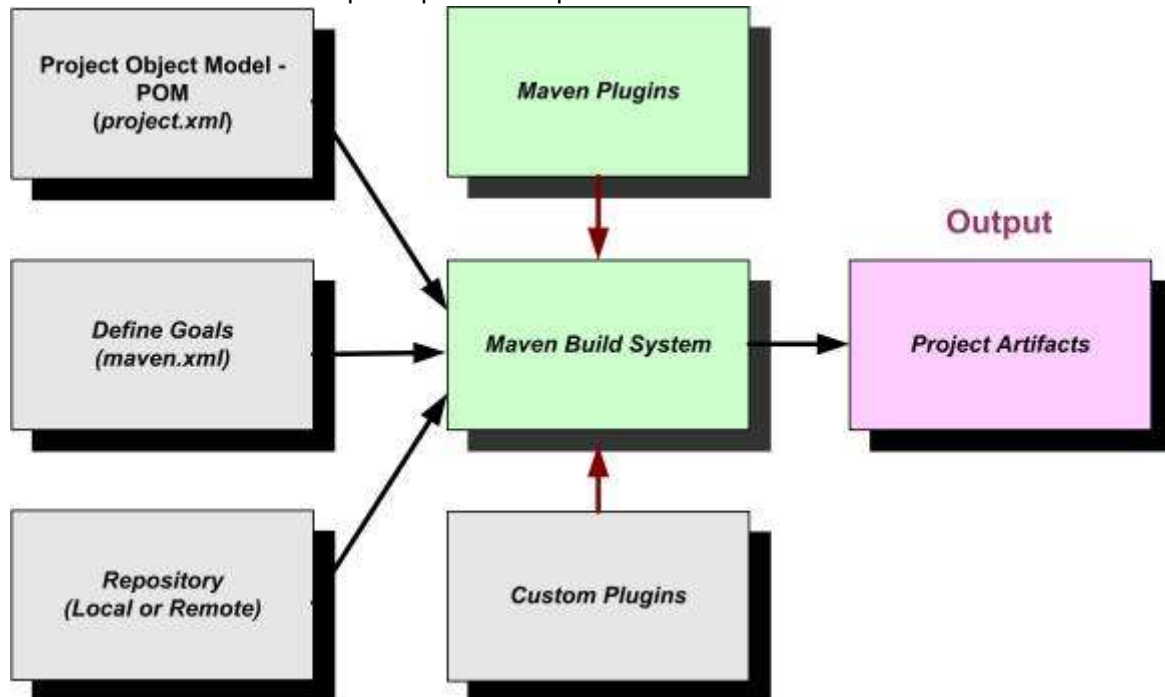


Figure 1 Overview of Maven build elements

I.8.2.4 Installation de Maven

Maven s'installe presque comme Ant :

- Télécharger le fichier compressé de Maven sur :
<http://maven.apache.org/start/download.html>
- Décompresser le fichier

```
> tar xzvf maven-1.0.tar.gz
```
- Définir la variable d'environnement MAVEN_HOME avec le répertoire d'installation de Maven

```
> export MAVEN_HOME=${mavenDirectory}
```
- Vérifier que la variable JAVA_HOME est définie, sinon la positionner avec le répertoire d'installation JDK
- Ajouter à la variable d'environnement PATH le répertoire \$MAVEN_HOME/bin



```
> export PATH=${MAVEN_HOME}/bin:${PATH}
```

- (cette étape est optionnelle)

Créer le repository local pour pouvoir utiliser Maven hors-ligne et surtout pour pouvoir le partager entre plusieurs équipes. Cela permet de regrouper tous les jars du projets et d'éviter de les retélécharger

```
> $MAVEN_HOME/bin/install_repo.sh $HOME/.maven/repository
```

Pour s'assurer de l'installation correcte de Maven, il suffit de saisir la commande suivante:

```
> maven -v
```

```
-- --  
|  \ /  |__ _Apache__ --  
| | \ / | / _ ` \ V / -_) ' \ ~ intelligent projects ~  
|_|  | \ __, | \ / \___|_|_| v. 1.0
```

Si vous obtenez le message ci-dessus, Bravo! Vous êtes maintenant prêt à utiliser Maven.

I.8.2.5 Repository Maven

Le **repository** représente aussi un autre **élément important** de Maven.

Afin de bien gérer les **dépendances**, Maven utilise un système qui s'appuie sur des **repositories** pour **télécharger automatiquement** les composants qu'il a besoin. Mais pour éviter que les fichiers se téléchargent à chaque reconstruction, Maven **stocke automatiquement les dépendances** nécessaires dans le **repository local**.

Par exemple, à la première exécution de maven, maven télécharge plusieurs plugins requis. Il se peut que cela prenne un certain temps.

I.8.2.5.1 Structure générale

Maven a standardisé la structure pour son repository. L'exemple suivant montre sa structure générale.

Pour ajouter vos propres dépendances, créer un répertoire dans repository en respectant la



structure suivante :

```
${user.home}/.maven
|- repository
  |- my-project      <-- project group ID -->
    |- jars          <-- artifact type + 's',e.g. jars, wars, ears -->
      |- my-project-1.0.jar    <-- artifact id + version -->
```



I.8.2.5.2 Propriétés du repository local ou distant

Ces repositories peuvent être locaux à la machine ou distants accessibles via HTTP. Pour certains, il sera utile de définir le proxy pour Maven.

Principales propriétés

Voici le tableau récapitulatif de ces propriétés :

maven.repo.remote	Specifies to the remote repositories a comma-separated list of URLs; <code>http://www.ibiblio.org/maven</code> is used by default.
maven.proxy.host, maven.proxy.port, maven.proxy.username, maven.proxy.password	If you are behind a firewall and require proxy authentication to access the Internet, these settings will come in handy.
maven.repo.local	Specifies where downloaded dependencies are cached, by default in <code>\${MAVEN_HOME}/repository</code> . In UNIX environments, to share the repository directory with multiple teams, you can create a special group for developers and give it read/write access to the repository directory.

Tableau 3. Properties for remote and local repositories

Exemple de proxy

Exemple : Créer le fichier `${user.home}/build.properties` ou ajouter le contenu suivant.

```
## -----  
## ${user.home}/build.properties  
## -----  
maven.proxy.host = proxyweb.univ-mlv.fr  
maven.proxy.port = 3128  
#maven.proxy.username = username  
#maven.proxy.password = password
```



I.8.3 POM : Project Object Model (project.xml)

Maven est orienté projet, donc le projet est l'entité principale gérée par Maven. Il est nécessaire de fournir à Maven une description du projet (Project descriptor) sous la forme d'un document XML nommé project.xml et situé à la racine du répertoire contenant le projet.

I.8.3.1 Composition

En général, trois principales parties composent le fichier project.xml :

- La partie gestion du projet inclue les informations telles que l'organisation du projet, la liste des développeurs, la localisation du code source, et l'URL du système pour déceler les bugs.
- La partie dépendance du projet inclut les informations concernant les dépendances du projet. Actuellement la version 1.0 beta 8 de Maven supporte uniquement les fichiers de dépendance au format JAR.
- La partie de build et de documentation (rapports) contient les informations du build telles que le répertoire du code source, des tests unitaires, et les rapports à générer définis dans le build.

Le fichier project.xml contient un identifiant unique de projet et un group ID. Ce group ID est particulièrement utile quand le projet contient des sous-projets. Tous ces sous-projets devraient partager le même groupe ID mais chacun d'entre eux devrait avoir un <id> distinct.

Il est également possible d'inclure la valeur d'un tag défini dans le document dans un autre tag.

Ex : <shortDescription>\${pom.name} est un test avec Maven</shortDescription>

On peut aussi faire hériter d'un fichier project.xml existant dans lequel des caractéristiques communes à plusieurs projets sont définies. La déclaration dans le fichier du fichier père se fait avec le tag <extend>. Dans le fichier fils, il suffit de redéfinir ou de définir les tags nécessaires.



I.8.3.2 Exemple

Voici un exemple d'un fichier project.xml :

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!-- l'element racine d'un fichier projet -->
<project>
  <!-- version du POM. Ce tag n'est pas encore exploité. -->
  <pomVersion>3</pomVersion>

  <!-- groupe id du projet. -->
  <!-- Si present, l'id est utilisé comme nom de dossier ds le repository -->
  <!-- pour regrouper tous les jars du projet -->
  <groupId>myGroup</groupId>

  <!-- Unique id du projet. Il est aussi utilisé pr générer des noms de fichier
  <!-- i.e, le fichier JAR est nommé : <id>--<version> -->
  <id>chronos</id>

  <!-- Le nom court du projet -->
  <name>Chronos IG2000</name>

  <!-- numero de version du projet. (Pas de norme à suivre) -->
  <currentVersion>0.0.1</currentVersion>

  ...
  <!--
----- -->
  <!-- Partie gestion de projet -->
  <!--
----- -->
  ...
  <!--
----- -->
  <!-- Partie dépendances du projet -->
  <!--
----- -->
  ...
  <!--
----- -->
  <!-- Partie build et documentation du projet -->
  <!--
----- -->
  ...
</project>
```

Listing 1. Squelette d'un project.xml

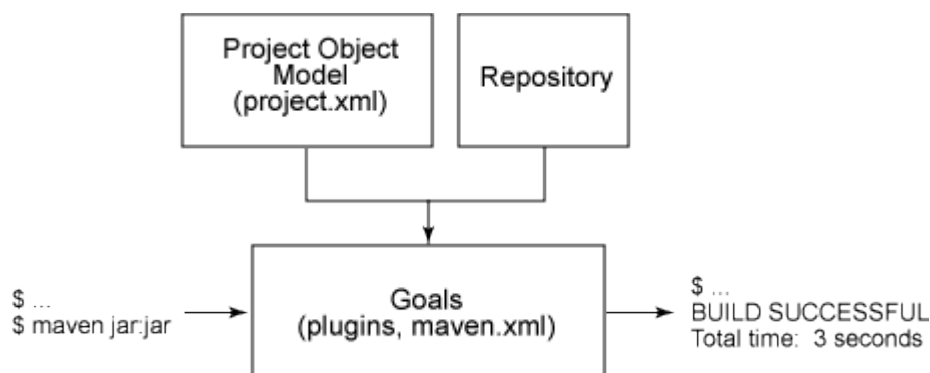
Tous les descripteurs du fichier project.xml sur : <http://maven.apache.org/reference/project-descriptor.html>

I.8.4 Utilisation de Maven

I.8.4.1 Concept

Les tâches exécutées par Maven reposent sur des plugins qui sont des fichiers jar. Chaque plugin permet d'effectuer des tâches particulières prédéfinies appelées « goals ». Les goals sont configurables dans le fichier de configuration de maven : **maven.xml**

La figure suivante récapitule l'architecture utilisée :



Maven s'utilise en ligne de commande sous la forme suivante :

Maven plugin:goal (ex : > maven java:compile)
(si le goal n'est pas spécifié le goal par défaut sera exécuté, s'il est définit.)

Il faut exécuter Maven dans le répertoire qui contient le fichier project.xml. Si les paramètres fournis ne sont pas correct, une exception est levée.

Pour obtenir une liste complète des plug-ins à disposition de Maven, il suffit d'utiliser la commande :

```
> maven -g
```

Voici une liste non exhaustive de divers plugins et de leurs goals :



Package, déploiement	Construction	Documentation
jar : deploy, install war : clean, deploy, init, install ear : ear, deploy	java : compile, jar test : match jalopy : format clean	site : generate, deploy javadoc : deploy, install pdf checkstyle : report
Intégration IDE	Autres	
eclipse : add-maven-repo, generate-classpath, generate-project jbulder : generate-library, generate-project	jhibernate Jdiff Junit ChangeLog (Produit un rapport des changements sur un CVS ou SVN)	

La commande *maven -g* permet d'avoir la liste de tous les plugins disponibles avec leurs goals.

Pour plus d'information, aller voir sur : <http://maven.apache.org/reference/plugins/index.html>

I.8.4.2 Générer un projet

Pour débiter un projet, Maven propose un plugin qui permet de générer la structure du projet :



```
> maven genapp
```

```
| V |__Apache__
```

```
| |V| / _ ` \ V / -_) ' \ ~ intelligent projects ~
```

```
| | | \ _ , \ \ ^ _ | | | v. 1.0
```

```
Enter a project template to use: [default]
```

```
default
```

```
Please specify an id for your application: [app]
```

```
myApp
```

```
Please specify a name for your application: [Example Application]
```

```
My Appplication Test
```

```
Please specify the package for your application: [example.app]
```

```
fr.example
```



Il y a plusieurs types de template prédéfinie :

- **default** – Generate un simple projet Jar.
- **Ejb** – Génère un simple EJB.
- **Struts** - Génère une simple application web Struts.
- **Struts-jstl** - Génère une simple application web Struts et JSTL.
- **Web** - Génère une simple application web Struts.
- **Web-jstl** - Génère une simple application web Struts avec JSTL.
- **Complex** - Génère un projet très complexe avec ear, wars, ejbs, struts, xdoclet.

Structure de la template default

```
myApp
|-- project.properties
|-- project.xml
|-- src/
|   |-- java/
|   |   |-- fr/example/
|   |   |   |-- App.java
|   |-- test/
|   |   |-- fr/example/
|   |   |   |-- AbstractTestCase.java
|   |   |   |-- AppTest.java
|   |   |   |-- NaughtyTest.java
|   |-- conf/
|   |   |-- app.properties
```

Ce plugins génère donc la structure de l'application selon les paramètres saisis et génère aussi le squelette des fichiers « project.properties » et « project.xml ». Ce plugin permet de démarrer rapidement sur le développement d'une application.

I.8.4.3 Générer un site Web

Maven propose une fonctionnalité qui permet de générer automatiquement un site web pour le projet regroupant un certain nombre d'informations utiles le concernant.



Pour demander la génération du site, il suffit de saisir la commande

> maven site:generate

Lors de l'exécution de cette commande, un répertoire target/docs est créé contenant les différents éléments du site.

Le site généré est composé de deux grandes parties :

- La partie « Project Info » qui regroupe trois pages : la mailing list, la liste des développeurs et les dépendances du projet.
- La partie « Project report » qui permet d'avoir accès à des comptes rendus d'exécution de certaines tâches : javadoc, tests unitaires, ... Certaines de ces pages ne sont générées qu'en fonction des différents éléments générés par Maven.

Le contenu du site pourra donc être réactualisé facilement en fonction des différents traitements réalisés par Maven sur le projet.

I.8.4.4 Compiler un projet

Dans le fichier project.xml, il faut rajouter un tag <build> qui va contenir les informations pour la compilation des éléments du projet.

Les sources doivent être contenues dans un répertoire dédié, par exemple src. Dans ce cas, le fichier project.xml doit contenir les lignes ci-dessous entre les tags <build> et </build>.

```
<build>
...
<sourceDirectory>
  ${basedir}/src
</sourceDirectory>
...
</build>
```

Pour demander la compilation à Maven, il faut utiliser la commande :

> maven java:compile

Le répertoire « \${project_home}/target/classes » est créé. Les fichiers .class issus de la compilation sont stockés dans ce répertoire.



La commande maven jar permet de demander la génération du packaging de l'application.

> maven jar

Par défaut, l'appel à cette commande effectue une compilation des sources, un passage des tests unitaires s'il y en a et un appel à l'outil jar pour réaliser le packaging.

Le nom du fichier jar créé est composé de l'id du projet et du numéro de version. Il est stocké dans le répertoire racine du projet.

I.8.5 Créer des GOALs (maven.xml)

Les goals de Maven sont extensibles et réutilisables. Mais avant de se lancer corps et âmes sur l'écriture de ses propres goals, le mieux serait de vérifier les plugins existants sur le site de Maven ou `${MAVEN_HOME}/plugins` ou <http://maven-plugins.sourceforge.net/>.

Si aucun de ces plugins ne correspond à vos besoins, Maven vous donne deux possibilités :

- D'étendre les goals standards avec les tag `<preGoal>` ou `<postGoal>`
- D'écrire vos propres goals

Dans les deux cas, le fichier spécial « maven.xml » est utilisé. Il se situe dans le répertoire racine du projet.

Le langage utilisé est le Jelly et le Ant task :

<http://maven.apache.org/reference/maven-jelly-tags/tags.html>

<http://ant.apache.org/manual/tasksoverview.html>

I.8.5.1 Structure du fichier 'maven.xml' et création d'un goal

Voici la structure que doit avoir le fichier maven.xml :



```
<project
  default="nightly-build"
  xmlns:j="jelly:core"
  xmlns:u="jelly:util">

  <goal name="nightly-build">
    <!-- Any ant task, or jelly tags can go here thanks to jeez -->
    <j:set var="goals" value="compile,test" />
    <mkdir dir="${maven.build.dir}" />
    <u:tokenize var="goals" delim=",">${goals}</u:tokenize>
    <j:forEach items="${goals}" var="goal" indexVar="goalNumber">
      Now attaining goal number ${goalNumber}, which is ${goal}
      <attainGoal name="${goal}" />
    </j:forEach>
  </goal>

</project>
```

default="nightly-build" : Cet attribut définit le goal par défaut à exécuter si maven n'a pas d'argument

xmlns:j="jelly:core" : Un namespace. Tous les tags préfixés par **j** seront rapportés à **jelly:core**

<mkdir dir="\${maven.build.dir}"/> : exécute le Ant task **mkdir**

La définition d'un goal peut surcharger d'autre goals de meme nom. Si un projet contient des sous-projets, alors ils hériteront de ces goals.

I.8.5.2 Extension des goals

Maven met à disposition deux tags spéciaux :

- **<preGoal>** définit les règles à exécuter **avant** le goal spécifié
- **<postGoal>** définit les règles à exécuter **après** le goal spécifié

Exemple de <preGoal>

```
<preGoal name="java:compile">
  <attainGoal name="xdoclet:ejbdoclet"/>
</preGoal>
```

I.8.5.3 Utilisation d'une task Ant avec Maven

Il suffit d'utiliser **goal** au lieu de **target** et **attainGoal** au lieu de **antcall** dans le fichier **maven.xml**. Ici **attainGoal** exécute directement le goal spécifié.



Ensuite on peut appeler une tâche directement avec `maven xxx` (xxx étant le nom de la tâche).

```
<goal name="copy:ressources">
  <copy todir="./target/classes">
    <fileset dir="./src">
      <exclude name="**/*.java"/>
    </fileset>
  </copy>
</goal>
<goal name="generate:all">
  <attainGoal name="copy:ressources"/>
  <attainGoal name="site:generate"/>
</goal>
```

```
maven copy:ressources
maven generate:all
```

I.8.5.4 Écrire un plugin

Afin de partager les goals aux autres projets, on les met dans un package pour en faire un plugin. Les plugins se trouvent dans le répertoire `${MAVEN_HOME}/plugins`.

I.8.6 Composition

Un plugin Maven contient au minimum deux fichiers :

- `project.xml` : Le fichier qui décrit le POM du plugin.
- `plugin.jelly` : Le fichier contenant les goals du plugin. Ce fichier ressemble au fichier `maven.xml`.

Les autres fichiers qui pourront être utiles sont :

- `project.properties` : C'est pour personnaliser le processus de construction du plugin
- `plugin.properties` : C'est les valeurs par défaut des propriétés du plugin

Les plugins peuvent aussi avoir leurs propres ressources et dépendances.



I.8.6.1.1 Installer le plugin

Pour rendre ce plugin accessible a toutes les personnes utilisant le meme Maven, utilisez la commande:

```
> maven plugin:install
```

Sinon copier le plugin construit dans votre répertoire local « plugins ».

I.8.6.1.2 Exemple de structure

Exemple d'une structure de plugin

```
hello-plugin-1.0
|-- src
|   |-- java
|   |-- plugin-resources
|-- xdocs
|-- plugin.jelly
`-- project.xml
```

I.8.7 Description de plugins utiles à Chronos

I.8.7.1 Les Plugins java et clean

Ce sont les principaux plugins qui permettent la compilation et le nettoyage des fichiers générés par le compilation.

Goal	Description
java:prepare-filesystem	Créer les répertoires nécessaires à la compilation
java:compile	Compile les fichiers codes sources java du projet Le répertoire source est définit entres les tags <code><build></code> dans <code>project.xml</code> .
clean:clean	Supprime <code>\${maven.build.dir}</code> et <code>\${basedir}/velocity.log</code> .

I.8.7.2 Le plug-in Jar

Les fichiers d'archives Java sont des fichiers « .jar » qui regroupent un ensemble de ressources. Ces archives java comportent essentiellement des classes java compilées « .class », mais elles peuvent également contenir toutes sortes de ressources comme des images et des fichiers de configuration.

Ces fichiers permettent de déployer et de distribuer très facilement des bibliothèques de classes java. L'adjonction à l'archive d'un fichier de configuration particulier, appelé manifeste, permet de rendre l'archive exécutable. Le manifeste doit alors contenir les informations concernant la classe à exécuter lors du lancement de l'archive.

Goal	Description
jar:jar	Créer un fichier jar dans le repertoire « build » du projet sous la forme <code>\${project.id}-\${project.currentVersion}.jar</code> où <code>id</code> and <code>currentVersion</code> proviennent de <code>project.xml</code> .

I.8.7.3 Le plugin Jalopy

Jalopy est un utilitaire open source très pratique qui permet de formater du code source Java et même de vérifier l'application de normes de codage.

Il permet notamment :

- d'indenter le code
- de générer des modèles de commentaires javadoc dynamique en fonction des l'éléments du code à documenter (par exemple générer un tag `@param` pour chaque paramètre d'une méthode)
- d'organiser l'ordre des clauses import
- d'organiser l'ordre des membres d'une classe selon leur modificateur
- de vérifier l'application de normes de codage,
- ...

Goal	Description
jalopy:format	Reformate tous les fichiers sources selon un code de convention



I.8.7.4 Le Plugin site

Goal	Description
Site:generate	<p>Génère le site d'information du projet. Il exécute les goals suivants pour créer la documentation du site :</p> <ul style="list-style-type: none">• jdepend : analyse la qualité du code (calcul des métriques)• checkstyle : analyse si les conventions de codage ont été respectées• changelog : Résume tous les changelog d'un CVS• activity : rapport sur l'activité des développeurs et des fichiers• javadoc : génération de la javadoc du projet• jxr• junit-report : rapport des tests unitaires• tasklist : résume une liste à faire• xdoc

I.8.8 Conclusion

Les grands avantages de Maven sont :

- Les nombreux plugins déjà existants qui répondent à nos besoins
- La diffusion de l'état actuelle du projet via la génération de site est grandement facilitée

Maven est un outil récent mais il est déjà utilisé par de nombreux projet Jarkarta. Il a donc déjà fait ses preuves et on peut avoir confiance sur la maturité de cet outil.

Chronos est un projet qui ne nécessite pas de structure spécifique. Il va donc suivre une structure de développement standard. Maven est donc tout à fait adapter à notre projet.

Le fait d'opter pour Maven au lieu de Ant nous donnerai les avantages suivants :

- De gagner beaucoup de temps sur le développement de script de construction : Utiliser les plugins existants de Maven alors qu'il faudrait écrire de long script pour avoir le même résultat sans compter l'installation des différents éléments..
- D'avoir une diffusion rapide de l'état globale du projet : Le site générer par maven est très complet et permet d'être vu par tous nous serait très utile. C'est agréable de savoir où l'on va et comment on avance.
- De mieux structurer et gérer le projet : Le concept POM nous force à donner les grandes lignes de développement et des étapes de tests.
- Faire des images rapides du projet (SnapShot) pour pouvoir faire un retour en arrière au cas où.

Maven est plus simple et plus facile à prendre en main que Ant.

De plus mevenide permet de l'intégrer dans la plupart des outils de développement.

Alors pourquoi s'en priver !



I.9 Continuous Testing

I.9.1 Introduction

Ce document décrit les possibilités d'un outil de tests basé sur JUnit, appelé Continuous Testing.

Il compte toutes les fonctionnalités de JUnit tout en en intégrant de nouvelles. Cependant son principal but est d'automatiser les tests unitaires.

I.9.2 Rappel sur les tests unitaires

Un test est un bout de code qui provoque l'exécution d'un autre bout de code et qui en analyse le résultat. Le premier bout de code est donc le test qui a été écrit, le second est le programme ou la fonction à tester. Le résultat d'un test est *valide* ou *invalide*.

I.9.2.1 L'intérêt des tests unitaires

L'intérêt des tests unitaires est multiple :

1. Programmer efficacement ;

En effet, écrire de bons tests signifie répondre aux attentes du programme.

2. Programmer rapidement ;

Plus vite une erreur est détectée, plus vite elle sera corrigée.

3. Programmer sans régresser ;

Lorsqu'un programme est correctement testé, le mieux est de le valider une bonne fois pour toute. Une amélioration pourrait « endommager » un programme qui avait déjà été testé et validé. Avec les tests unitaires, il est possible de garantir le bon fonctionnement du programme en question.

4. Programmer en équipe.

Les tests unitaires sont essentiels pour un bon avancement dans une équipe de programmeurs. Plutôt que de valider le code de ses partenaires, il suffit de lancer les tests afin de vérifier le travail de chacun.

I.9.3 Le plug-in Continuous Testing

Le plug-in Continuous Testing est disponible à l'adresse suivante :



<http://www.pag.csail.mit.edu/continuoustesting/>

I.9.3.1 Installation

Son installation est simple :

1. Démarrer Eclipse ;
2. Depuis la barre de menu, sélectionner *Help > Software Updates > Find and Install* ;
3. Choisir *Search for new features to install*, et sélectionner *Next* ;
4. Cliquer sur le bouton *Add Update Site...* ;
5. En nom, le type est *Continuous Testing Site*, et en URL <http://pag.csail.mit.edu/continuoustesting/> ;
6. Un bouton à cocher *Sites to include in Search* doit être sélectionné, puis il faut cliquer sur *Next* ;
7. Sélectionner la case *Continuous Testing* puis *Next* ;
8. Accepter les informations d'utilisation pour la licence et la signature du plug-in pour finir l'installation
9. Continuous Testing sera utilisable lors du prochain lancement d'Eclipse.

I.9.3.2 Les objectifs de Continuous Testing

Continuous Testing est un outil de tests basé sur JUnit, il donne les mêmes informations sous le même format. Cependant, il permet en plus d'automatiser, de filtrer mais aussi de prioriser les tests unitaires.

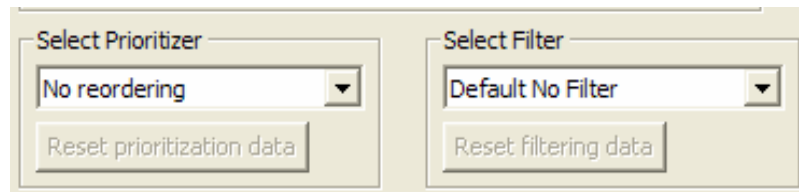
I.9.3.3 Activer les fonctions de Continuous Testing

Continuous Testing s'active sur un projet :

1. Propriétés du projet ;
2. Choisir *Continuous Testing Properties* ;
3. Sélectionner *Enable Informed Testing* et *Enable Continuous Testing* ;
4. Continuous Testing est à présent en cours de fonctionnement.

I.9.3.4 Détails sur les apports de Continuous Testing

Continuous Testing permet de prioriser et de filtrer les tests unitaires dans le but est de leur donner un ordre d'exécution. Cela se fait depuis les propriétés du projet, dans la section *Continuous Testing Properties* :

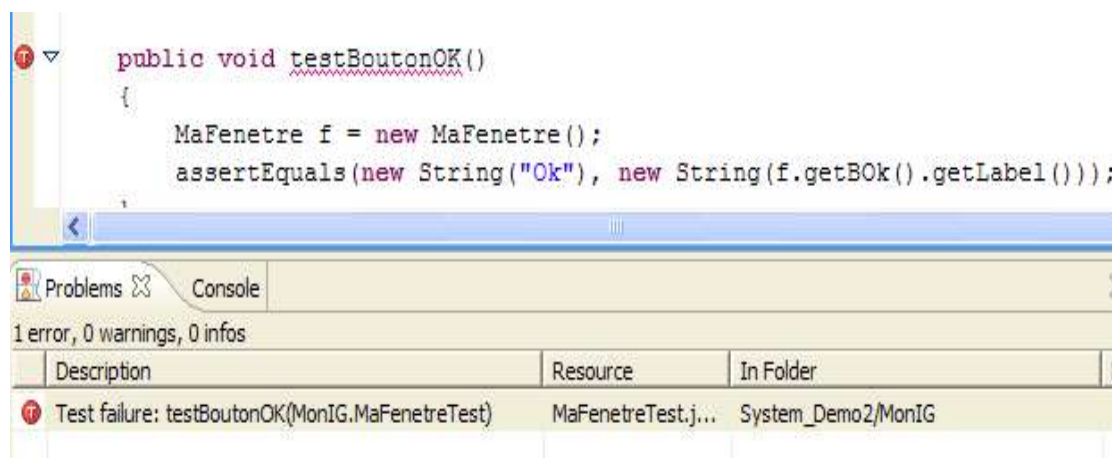
**Priorités :**

- Most Recent Failures First : prioriser les derniers tests qui ont échoués ;
- Most Frequent Failures First : prioriser les tests qui échouent souvent ;
- Quickest Test First : prioriser les tests les plus rapides ;
- Round Robin : donner un ordre de lancement équitable ;
- Random : lancer les tests dans un ordre aléatoire ;
- No reordering : aucune priorité.

Filtres :

- Omit Previous Successes : ne lance que les tests qui ont eu un succès lors de la dernière exécution (pour éviter les régressions) ;
- Most Recent Failures Deleted : ne lance que les tests qui ont échoué ;
- Informed Filter(Ct Only) : ne lance que les tests de la forme *testClassName* où *ClassName* est le nom de la classe.
- Default No Filter : aucun filtre.

En plus des priorités et des filtres, Continuous Testing offre une notification des erreurs. Cela permet de mettre en évidence les méthodes qui échouent lors des tests.



De plus les fonctions en question sont soulignées en rouge, près de celle ci apparaît un *T* en rouge qui signifie « Test Failures » afin de permettre aux programmeurs de détecter plus rapidement les méthodes qui posent problème.

I.10 Conclusion sur continuous testing

En conclusion, nous avons vu que Continuous Testing est un outil performant de tests, qui permet d'améliorer la phase de tests dans le cadre du projet Chronos.

Nous retenons donc que les intérêts des tests unitaires sont : la non régression, la rapidité, la clarté du code, ainsi qu'une documentation efficace.

Les options qu'offre Continuous Testing sont : automatisation, priorité, filtrage des tests, ainsi que notification des erreurs.

I.11 API de Logging

Le logging est important comme outil de journalisation des applications pour permettre de faciliter le débogage lors du développement et de conserver une trace de son exécution lors de l'exploitation en production.

I.11.1 Log4j

Une API très répandue est celle développée par le projet open source log4j du groupe Jakarta. Sun a aussi développé et intégré au JDK 1.4 une API dédiée. Cette API est plus simple à utiliser et elle offre moins de fonctionnalités que log4j mais elle a l'avantage d'être fournie directement avec le JDK. C'est pour cette raison que nous avons décidé d'utiliser Log4j. Nous allons donné dans cette partie un aperçu de cette bibliothèque, sans rentrer dans les détails car nous avons décider d'utiliser l'API Commons Logging afin masquer l'utilisation de la bibliothèque log4j.

La dernière version de log4j est téléchargeable à l'adresse suivante : <http://jakarta.apache.org/log4j/>

Cette API permet aux développeurs d'utiliser et de paramétrer les traitements de logs. Il est possible de fournir les paramètres de l'outil dans un fichier de configuration. Log4j est thread-safe.

Log4j utilise trois composants principaux :



- Catégories : ils permettent de gérer les logs ;
- Appenders : ils représentent les flux qui vont recevoir les messages de log ;
- Layouts : ils permettent de formater le contenu des messages du fichier de log.

I.11.1.1 Les catégories

Les catégories déterminent si un message doit être envoyé dans le ou les logs. Elles sont représentées par la classe **org.apache.log4j.Category**. Le nom de la catégorie permet de la placer dans une hiérarchie dont la racine est une catégorie spéciale nommée root qui est créée par défaut sans nom. La classe **Category** possède une classe statique **getRoot()** pour obtenir la catégorie racine. La hiérarchie des noms est établie grâce à la notation par point comme pour les paquets. Cette relation hiérarchique est importante car la configuration établie pour une catégorie est automatiquement propagée par défaut aux catégories enfants.

L'ordre de la création des catégories de la hiérarchie ne doit pas obligatoirement respecter l'ordre de la hiérarchie. Celle-ci est constituée au fur et à mesure de la création des catégories.

I.11.1.2 Les priorités

Log4j gère des priorités pour permettre à la catégorie de déterminer si le message sera envoyé dans le fichier de log. Il existe cinq priorités qui possèdent un ordre hiérarchique croissant :

- DEBUG ;
- INFO ;
- WARN ;
- ERROR ;
- FATAL.

La classe **org.apache.log4j.Priority** encapsule ces priorités. Chaque catégorie est associée à une priorité qui peut être changée dynamiquement. La catégorie détermine si un message doit être envoyé dans le fichier de log en comparant sa priorité avec la priorité du message. Si celle-ci est supérieure ou égale à la priorité de la catégorie, alors le message est envoyé dans le fichier log.

La méthode **setPriority()** de la classe **Category** permet de préciser la priorité de la catégorie.

Si aucune priorité n'est donnée à une catégorie, elle "hérite" de la priorité de la première catégorie en remontant dans la hiérarchie dont la priorité est renseignée. Au niveau applicatif,



il est possible d'interdire le traitement d'une priorité et de celle inférieure en utilisant le code suivant : **Category.getDefaultHierarchy().disable(Priority p)**.

I.11.1.3 Les appenders

L'interface **org.apache.log4j.Appender** désigne un flux qui représente le fichier de log et se charge de l'envoi de message formaté ce flux. Le formatage proprement dit est réalisé par un objet de type Layout. Ce layout peut être fourni dans le constructeur adapté ou par la méthode **setLayout()**.

Une catégorie peut posséder plusieurs *Appenders*. Si la catégorie décide de traiter la demande de message, le message est envoyé à chaque *Appenders*. Pour ajouter un *Appender* à une catégorie, il suffit d'utiliser la méthode **addAppender()** qui prend en paramètre un objet de type **Appender**.

L'interface Appender est directement implémentée par la classe abstraite **AppenderSkeleton**. Cette classe est la classe mère de toutes les classes fournies avec log4j pour représenter un type de log :

- AsyncAppender ;
- JMSAppender ;
- NTEventLogAppender : log envoyé dans le fichier de log des événements sur un système Windows NT ;
- NullAppender ;
- SMTPAppender ;
- SocketAppender : log envoyés dans une socket ;
- SyslogAppender : log envoyé dans le demon syslog d'un système Unix ;
- WriterAppender : cette classe possède deux classes filles : ConsoleAppender et FileAppender. La classe FileAppender possède deux classes filles : DailyRollingAppender, RollingFileAppender.

Pour créer un *Appender*, il suffit d'instancier un objet d'une de ces classes.

Tout comme les priorités, les *Appenders* d'une catégorie mère sont héritées par les catégories filles. Pour éviter cette héritage par défaut, il faut mettre le champ *additivity* à *false* en utilisant la méthode **setAdditivity()** de la classe **Category**.



I.11.1.4 Les layouts

Ces composants représentés par la classe **org.apache.log4j.Layout** permettent de définir le format du fichier de log. Un layout est associé à un appender lors de son instantiation.

Il existe plusieurs layouts définis par log4j :

- DateLayout ;
- HTMLLayout ;
- PatternLayout ;
- SimpleLayout ;
- XMLLayout.

Le PatternLayout permet de préciser le format du fichier de log grâce à des motifs qui sont dynamiquement remplacés par leur valeur à l'exécution. Les motifs commencent par un caractère % suivi d'une lettre :

Motif	Role
%c	le nom de la catégorie qui a émis le message
%C	le nom de la classe qui a émis le message
%d	le timestamp de l'émission du message
%m	le message
%n	un retour chariot
%p	la priorité du message
%r	le nombre de millisecondes écoulées entre le lancement de l'application et l'émission du message
%t	le nom du thread
%x	NDC du thread
%%	le caractère %

Il est possible de préciser le formatage de chaque motif grâce à un alignement et/ou une troncature. Dans le tableau ci dessous, la caractère # représente une des lettres du tableau ci dessus, n représente un nombre de caractères.



Motif	Rôle
%#	aucun formatage (par défaut)
%n#	alignement à droite, des blancs sont ajoutés si la taille du motif est inférieure à n caractères
%-n#	alignement à gauche, des blanc sont ajoutés si la taille du motif est inférieure à n caractères
%.n	tronque le motif si il est supérieur à n caractères
%-n.n#	alignement à gauche, taille du motif obligatoirement de n caractères (troncature ou complément avec des blancs)

I.11.1.5 Configuration

Pour faciliter la configuration de log4j, l'API fournit plusieurs classes qui implémentent l'interface **Configurator**. La classe **BasicConfigurator** est la classe mère des classes **PropertyConfigurator** (pour la configuration via un fichier de propriétés) et **DOMConfigurator** (pour la configuration via un fichier XML).

La classe **PropertyConfigurator** permet de configurer log4j à partir d'un fichier de propriétés ce qui évite la recompilation de classes pour modifier la configuration. La méthode `configure()` qui attend un nom de fichier permet de charger la configuration.

Fichier logging.properties

```
# Affecte à la catégorie root la priorité DEBUG et un appender nommé
# CONSOLE_APP
log4j.rootCategory=DEBUG, CONSOLE_APP
# l'appender CONSOLE_APP est associé à la console
log4j.appender.CONSOLE_APP=org.apache.log4j.ConsoleAppender
# CONSOLE_APP utilise un PatternLayout qui affiche : le nom du thread, la
# priorité,
# le nom de la catégorie et le message
log4j.appender.CONSOLE_APP.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE_APP.layout.ConversionPattern= [%t] %p %c - %m%n
```

La classe **DOMConfigurator** permet de configurer log4j à partir d'un fichier XML ce qui évite aussi la recompilation de classes pour modifier la configuration. La méthode `configure()` qui attend un nom de fichier permet de charger la configuration. Cette méthode nécessite un



parser XML de type DOM compatible avec l'API JAXP.

Fichier logging.properties

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="CONSOLE_APP" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="[%t] %p %c - %m%n"/>
    </layout>
  </appender>
  <root>
    <priority value="DEBUG" />
    <appender-ref ref="CONSOLE_APP" />
  </root>
</log4j:configuration>
```

I.11.2 Commons logging

Dans le but de faciliter l'utilisation de la bibliothèque de logging, nous avons décidé d'utiliser la bibliothèque Commons Logging, de la Jakarta Fondation, qui encapsule l'usage de plusieurs systèmes de logging et facilite ainsi leur utilisation dans les applications. Ce n'est pas un autre système de log mais il propose un niveau d'abstraction qui permet sans changer le code d'utiliser indifféremment n'importe lequel des systèmes de logging supportés. Son utilisation est d'autant plus pratique qu'il existe plusieurs systèmes de log dont aucun des plus répandus, Log4j et l'API logging du JDK 1.4, ne sont dominants.

Le grand intérêt de cette bibliothèque est donc de rendre l'utilisation d'un système de log dans le code indépendant de l'implémentation de ce système. JCL encapsule l'utilisation de Log4j, l'API logging du JDK 1.4 et LogKit.

Le package, contenu dans le fichier commons-logging-1.0.3.zip peut être téléchargé sur le site <http://jakarta.apache.org/commons/logging.html>. Il doit ensuite être décompressé dans le répertoire contenant les bibliothèques de Java.

L'inconvénient d'utiliser cette bibliothèque est qu'elle n'utilise que le dénominateur commun des système de log qu'elle supporte : ainsi certaines caractéristiques d'un système de log particulier ne pourront être utilisées via cette API .

La bibliothèque propose une fabrique qui renvoie, en fonction du paramètre précisé, un objet qui implémente l'interface **Log**. La méthode statique **getLog()** de la classe **LogFactory** permet d'obtenir cet objet : elle attend en paramètre soit un nom sous la forme d'une chaîne de caractères soit un objet de type **Class** dont le nom sera utilisé. Si un objet de type log possédant ce nom existe déjà alors c'est cette instance qui est renvoyée par la méthode sinon c'est une nouvelle instance qui est retournée. Ce nom représente la catégorie pour le système log utilisé, si celui ci supporte une telle fonctionnalité.

Exemple pour une classe Math déclarant quelques fonctions (applicable pour n'importe quelle classe) :

```
// ...
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
// ...

public class Math {
    // Cree le logger de la classe
    static Log logger = LogFactory.getLog(MathImpl.class.getName());
    int value;
    // ...

    public void add(int a) {
        logger.info("Addition invoquée avec comme paramètre a=" +
            String.valueOf(a));
        if (a==0)
            logger.warn("Ajouter zéro ne modifie pas la valeur!");
        value = value + a;
    }

    public void subtract(int a) {
        logger.info("Soustraction invoquée avec comme paramètre a=" +
            String.valueOf(a));
        if (a==0)
            logger.warn("Soustraire zéro ne modifie pas le valeur!");
        value = value - a;
    }

    public int getValue() {
        logger.info("getValue() invoquée");
        return value;
    }
}
```



Par défaut, la méthode `getLog()` utilise les règles suivantes pour déterminer le système de log à utiliser :

- Si la bibliothèque Log4j est incluse dans le classpath de la JVM alors celle ci sera utilisée par défaut par la bibliothèque Commons Logging.
- Si le JDK 1.4 est utilisé et que Log4j n'est pas trouvé alors le système utilisé par défaut est celui fourni en standard avec le JDK (**`java.util.logging`**)
- Si aucun de ces systèmes de log n'est trouvé, alors JCL utilise un système de log basic fourni dans la bibliothèque : **SimpleLog**. La configuration de ce système se fait dans un fichier nommé `simplelog.properties`

Il est possible de forcer l'usage d'un système de log particulier en précisant la propriété **`org.apache.commons.logging.Log`** à la machine virtuelle.

Pour complètement désactiver le système de log, il suffit de fournir la valeur **`org.apache.commons.logging.impl.NoOpLog`** pour la propriété **`org.apache.commons.logging.Log`** à la JVM. Attention dans ce cas, plus aucun log ne sera émis.

Comme on peut le voir dans l'exemple précédent, on génère un message d'information à chaque fois que les méthodes sont invoquées, et un message d'alerte (warning) qui les deux méthodes de soustraction et d'addition sont appelées avec un paramètre égal à zéro.

On peut aussi générer des messages pour d'autres niveaux de gravité avec :

- `logger.debug("Message") ;`
- `logger.trace("Message") ;`
- `logger.error("Message") ;`
- `logger.fatal("Message").`

La bibliothèque tentera de faire correspondre au mieux les niveaux de gravité avec le système de log utilisé.

I.12 Les design patterns dans Chronos

I.12.1 Introduction

Ce document présente les design patterns qui sont utilisés dans le logiciel Chronos. Il donne des explications quant aux modèles de conceptions utilisés, mais a aussi un rapport intime avec les normes de développement. Tout développeur du projet se doit de comprendre et d'adhérer à l'utilité et l'importance de chacun des points présentés dans ce document pour être capable de participer de façon correcte et efficace au développement de Chronos.

On présentera chaque problème ou chaque point que le logiciel doit traiter. Pour certains de ceux-ci, on présentera la réponse au problème ainsi qu'un exemple type de son implémentation.

I.12.2 Historique

I.12.2.1 Effectivity

Le design pattern utilisé pour gérer cette notion omniprésente d'historique et d'historisation est la première méthode de résolution qui vienne à l'esprit et que Martin Fowler⁴ a appelé Effectivity⁵.

Une donnée ou un ensemble de données permettant de connaître la période pendant laquelle l'objet est effectif est stockée dans celui-ci. Typiquement, on aura tendance à stocker une date de début de validité, ou d'effectivité et une date de fin.

L'intérêt est de pouvoir retrouver à chaque moment passé quel était l'état de l'objet. Pour que ce modèle fonctionne, avant chaque modification il faut « clôturer » l'objet en cours (un cours par exemple) en lui positionnant une date de fin d'effectivité à la date en cours. Créer une copie de l'objet avec pour date de début d'effectivité la même date en cours (ou après celle-ci, mais pas avant, attention). On finira par vérifier que la date de fin d'effectivité est bien absente du nouvel objet créé.

Ce design pattern a l'avantage d'être très simple à comprendre et de permettre de gérer complètement le cycle de vie et de modifications d'un objet. Il est notamment possible, ce qui sera fait dans Chronos, de gérer un attribut d'état permettant d'indiquer ce qui a conduit à la création d'un objet (modification, suppression, annulation...).

4 <http://www.martinfowler.com>

5 <http://www.martinfowler.com/ap2/effectivity.html>



I.12.3 Stockage des données

I.12.3.1 Hibernate

Se référer à l'étude technique éponyme.

I.12.4 Logging

Les différentes bibliothèques de logging existantes sont nombreuses. C'est pourquoi nous souhaitons trouver un moyen d'en utiliser une sans affecter tout notre code.

Les commons-logging d'Apache répondent exactement à ce besoin.

I.12.4.1 Commons-Logging

Les commons-logging sont aux apis de logging ce que JDBC est aux bases de données, JTA aux apis de transaction...

Ils offrent une interface unifiée et simplissime à plusieurs systèmes de logging sous-jacent. Par défaut, par exemple, c'est l'API du JDK qui sera utilisée. Si elle est présente, c'est la bibliothèque log4j d'Apache qui sera utilisée.

On peut ainsi dans tout le code utiliser une interface d'API simple qui peut difficilement être remise en question au vu de sa simplicité pour utiliser selon le désir l'api souhaitée en dessous.

I.12.5 Masquage par interface

Dans toute l'application, ce principe sera très fortement préconisé.

Dans la mesure du possible, toutes les méthodes devront travailler avec des interfaces de niveau le plus élevé possible.

L'exemple le plus simple est l'interface Collection ou l'interface List. On préférera toujours écrire ceci :

```
List list = new ArrayList();
```

ou

```
List getList()
```

à cela



```
ArrayList list = new ArrayList
```

ou

```
ArrayList getList()
```

I.12.5.1 Factory

De même, l'un des intérêts des factory est aussi de permettre de renvoyer une implémentation d'un objet sans que tout l'application doive savoir laquelle. Ceci est un peu en rapport avec le design pattern **Strategy** qui consiste à pouvoir renvoyer l'une ou l'autre implémentation en fonction de paramètres divers par exemple.

I.12.5.1.1 Exemple :

Avec cette méthode, tout le code de l'application sera lié à la fois à l'interface Pouet et à son implémentation PouetImpl :

```
Pouet pouet = new PouetImpl();
```

On préférera l'utilisation d'une factory pour que application ignore que pour l'interface Pouet, elle utilise l'implémentation PouetImpl.

```
//classe final pour interdire la dérivation
public final class PouetFactory
{
    public Pouet createPouet()
    {
        return new PouetImpl();
    }
    /**
     * Singleton pour cette factory.
     */
    public static PouetFactory getFactory()
    {
        if(pouetFactory==null)
        {
            pouetFactory = new PouetFactory();
        }
        return pouetFactory;
    }
    private static final PouetFactory;
```

```
//constructeur privé pour interdire l'instanciation
private PouetFactory()
{
}
}
```

I.12.6 Simplifier l'utilisation d'une bibliothèque

Le design pattern **Façade** a déjà été mis à profit dans l'application pour simplifier l'API JavaMail. Ainsi, grâce à une unique classe MailSender :

Cette classe simplifie pour Chronos l'envoi d'un mail. Les besoins de notre application en terme d'envoi de mail sont limités, il apparaissait donc nécessaire de masquer la hiérarchie des classes de JavaMail pour pouvoir envoyer un mail le plus simplement possible sans contraindre chacun à appréhender BodyPart et autres.

I.12.7 Accès aux données : DAO

L'une des problématiques les plus importantes du code d'un logiciel réside souvent dans la capacité à masquer et à centraliser la façon dont on accède aux données. Depuis toujours, les informaticiens ont cherché à rendre leur code le plus modulaire et réutilisable possible. L'avènement des langages objet ont permis de formidables avancées dans ce domaine.

Toutefois, on aimerait bien pousser cela encore plus loin : et si demain je veux changer de SGBD ? Et si demain je veux changer mon système de gestion de logs ?

On veut briser le plus de dépendances possibles. L'accès aux données est clairement l'un des plus difficiles, tout simplement parce que l'accès aux données est omniprésent dans une application qui a pour souvent pour unique but de les gérer.

De cette constatation est née et a été formalisée une solution : le Data Access Object, objet d'accès aux données.

I.12.7.1 Définition

On va dédier un objet à l'interfaçage application/données. Du côté de l'application, on utilisera uniquement des objets métier, construits, peuplés. C'est à l'intérieur du DAO, du côté des



données que l'on gérera l'instanciation, le peuplement et la gestion bas niveau des objets. Ces Dao fourniront des méthodes de haut niveau, métier et s'occuperont de tout ce qui est purement accès aux données.

I.12.7.2 À quoi cela sert-il concrètement ?

En procédant ainsi, on va pouvoir, si on le désire, changer totalement le système avec lequel on stockera ou accédera aux données sans impacter autre chose que le DAO.

Exemple simple : vous voudriez migrer rapidement une application de C vers Java, mais toutes vos bibliothèques d'accès aux données qui sont fonctionnellement complexes et impossibles à réécrire rapidement sont justement écrites en C. Comment faire ?

Définissez simplement les DAOs possédant les fonctions métier dont vous avez besoin au fur et à mesure et utilisez JNI à l'intérieur de ceux-ci. Vous serez ultérieurement capable de passer à une solution pur java pour votre accès aux données sans jamais impacter tout le code développé pour la nouvelle application Java?

I.12.7.3 Exemple d'implémentation

On définit une Factory pour obtenir une instance d'un DAO. Disons ADao et BDao. Ces deux derniers sont des interfaces métier. La chose la plus importante à faire est d'écrire le contrat des méthodes des Daos. L'implémentation de ceux-ci sera faite ultérieurement et relève du détail.

```
public class DaoFactory
{
    private DaoFactory()
    {
        //non instanciable par l'extérieur
    }
    private static DaoFactory instance = new DaoFactory();
    public DaoFactory getInstance()
    {
        return instance;
    }
    public ADao getADao()
    {
        return new ADaoImpl();
    }
    public BDao getBDao()
```

```
{  
    return new BDaoImpl();  
}  
}
```

Il ne reste ensuite plus qu'à écrire `ADaoImpl` et `BDaoImpl` afin que les méthodes qu'elles implémentent respectent le contrat défini dans les interfaces et le travail est terminé.

I.12.7.4 Conclusion

Les Daos sont une façon simple et puissante de gérer l'accès aux données. Toute la partie accès aux données de Chronos est complètement basée dessus et respecte exactement les règles énoncées ci-dessus.

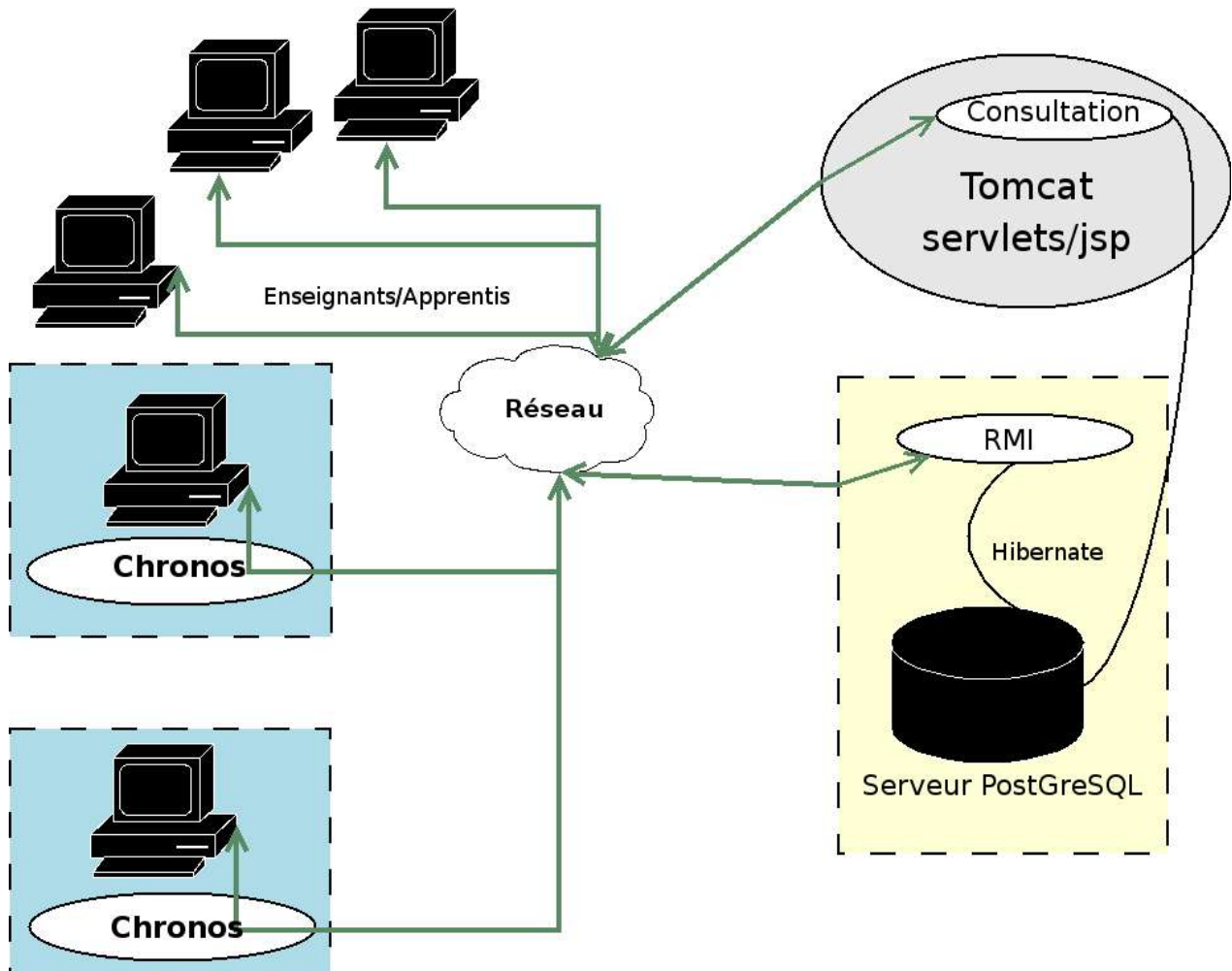
I.12.8 Conclusion

Les design patterns présentés ici font partie de ceux les plus utilisés. Ils devraient permettre de poser de bonnes bases au développement d'un code clair et normé. Toutefois, il est évident que seul le bon vouloir du développeur peut permettre la réussite du projet. Un design pattern théoriquement efficace peut très bien être appliqué de manière inefficace si on n'y prend pas garde. Il sera donc primordial pour chacun de faire vérifier régulièrement le code qu'il développe par d'autres. La méthode *Expert Programming* consistant à travailler à deux en même temps sur une même portion de code pour les modules complexes et importants pour le reste du code de l'application pourra aussi être utilisée à profit si le besoin s'en fait sentir.

II Architecture

Dans cette partie, nous présenterons l'architecture technique que nous avons retenue, puis l'étude des différents logiciels ou bibliothèques permettant de mettre en place cette architecture.

II.1 Présentation de l'architecture



Les raisons de notre choix sont regroupés dans le fichier Proposition_Architecture inclut en annexe. Ce fichier compare plusieurs architectures possibles et s'efforce de dresser les avantages et inconvénients de chaque architecture.



II.2 RMI

II.2.1 Introduction

Ce document a pour but la présentation de la bibliothèque java RMI, afin de pouvoir la mettre en oeuvre lors du développement de notre application. Nous avons en effet choisi d'utiliser pour notre application, une architecture reposant sur des objets partagés. Ce document s'attachera donc à présenter pour quelles raisons nous avons choisi RMI pour la gestion des objets partagés. À travers des exemples, nous indiquerons des solutions pour le *callback* avec RMI. Nous présenterons aussi l'utilisation du plugin 'RMI for Eclipse' qui permet de faciliter le développement d'application partagées utilisant RMI.

Nous avons donc choisi d'utiliser une architecture reposant sur des objets partagés. Pour cela, nous disposons des outils Corba et RMI pour mettre en place ce type d'architecture dans le langage Java. Corba est le plus standardisé de ces deux outils car il permet de créer des objets partagés avec beaucoup de langages de programmation et repose sur un protocole standardisé pour la communication entre objets. RMI ne s'adresse quant à lui qu'au langage java et peut aussi utiliser le protocole IIOP de Corba.

Ce qui nous a poussé à choisir RMI, c'est justement le fait qu'il s'adresse à un environnement java, et qu'il est inclus en standard avec les distributions java. Corba est plus compliqué à mettre en place sur un tel environnement et l'intérêt de pouvoir utiliser d'autres langages pour l'application ne nous concerne pas.

II.2.2 RMI

II.2.2.1 Objectif

L'objectif est de permettre à des applications clientes d'invoquer des méthodes sur des objets distants, c'est-à-dire localisés dans une autre application (dans une autre JVM de la même machine physique ou sur une autre machine accessible via Internet) communément appelée serveur.

Le *package* **java.rmi** et ses sous-*packages* (**JAVA.RMI.SERVER ET JAVA.RMI.DGC**) contiennent la définition de classes et d'outils permettant l'implantation et l'utilisation d'objets distants.



II.2.2.2 Principe

Le mécanisme proposé permet à une application s'exécutant sur une machine M1 de créer un objet et de le rendre accessible à d'autres applications: cette application (et la machine M1) joue donc le rôle de **serveur**.

Les autres applications manipulant un tel objet sont des **clients**. Pour manipuler un objet distant, un client récupère sur sa machine une représentation de l'objet appelé **stub** : ce **stub** implante l'interface de l'objet distant. C'est via ce **stub** que le client pourra invoquer des méthodes sur l'objet distant. Une telle invocation sera transmise au serveur (le protocole TCP est utilisé) afin d'en réaliser l'exécution.

Du côté du serveur, un **skeleton** a en charge la réception des invocations distantes, de leur réalisation et de l'envoi des résultats. Chaque fois que le serveur reçoit une invocation distante, il lance un nouveau thread pour répondre à cette invocation. Corba repose sur le même principe.

Nous essayerons, dans la présentation de la mise en oeuvre de RMI, de faire le rapprochement avec une mise en oeuvre reposant sur Corba afin de comparer ces deux solutions et de faciliter la compréhension des membres de l'équipe (Corba ayant été vu en cours).

II.2.2.3 Mise en oeuvre

II.2.2.3.1 Déclaration de l'interface

Tout d'abord il faut spécifier l'interface et donc les méthodes qui pourront être invoquées sur l'objet. Cette interface sera connue du côté du client et du côté du serveur. Afin de permettre l'appel des méthodes de l'interface, celle-ci doit hériter de l'interface **java.rmi.Remote** qui ne contient aucune méthode propre, mais a comme objectif de permettre la désignation des objets distants. Ainsi un objet distant va se présenter comme une instance d'une classe implantant l'interface **java.rmi.Remote** (ou de toute interface basée sur cette interface):

```
import java.rmi.*;
public interface MonInterface extends Remote {
    /* spécification des méthodes de l'interface */
}
```

Par ailleurs, chacune des méthodes de l'interface doit être en mesure de prendre en compte les différentes exceptions distantes susceptibles d'être rencontrées lors de la détection d'une anomalie (**java.rmi.RemoteException**). Cette exception est de type **Runtime**, il faut donc

impérativement la catcher.

Ainsi, dans la séquence suivante, une méthode renvoyant un entier et prenant en paramètres deux entiers est déclarée dans l'interface:

public int methode(int x, int y) throws RemoteException;

Il est important de ne pas perdre de vue que seules les méthodes de la classe implantant l'interface **Remote** seront accessibles aux clients.

Exemple : cet exemple très simple plante un compteur accessible à distance. Il s'agit essentiellement d'une variable entière modifiable et consultable, pour laquelle le nombre de de fois qu'elle a été référencée est de plus consultable. L'interface accessible à distance est la suivante:

```
import java.rmi.*;
public interface Compteur extends java.rmi.Remote {
    public void incr() throws RemoteException;
    public int getValue() throws RemoteException;
    public int getAccess() throws RemoteException;
}
```

En Corba, la mise en oeuvre de l'interface repose sur la déclaration en langage IDL de cette interface. Cette interface IDL sera ensuite 'convertie' en langage JAVA à l'aide de pré-compilateurs IDL.

II.2.2.3.2 Implantation de l'interface

Du côté du serveur, il faut implanter l'interface définie précédemment. Il sera ainsi possible sur le serveur de créer une instance de l'objet qui sera accessible par les clients.

Il s'agit ici de donner une implantation de l'interface définie lors de la première étape : c'est cette implantation qui permettra de rendre un objet accessible à distance par des clients.

Par ailleurs, cette implantation doit permettre l'exposition de l'objet distant. Différentes solutions sont possibles. La plus simple consiste à étendre la classe **java.rmi.server.UniCastRemoteObject** .

Cette classe du package RMI fournit des implantations de nombreuses méthodes de `java.lang.Object` adaptés aux objets distants. Par ailleurs, elle définit différents constructeurs et méthodes statiques qui réalisent l'export des objets distants. Les objets d'une classe ainsi



définie seront par construction automatiquement exportés et les sockets standards seront utilisées (de manière transparente pour les utilisateurs) pour la communication.

Toute autre solution devra d'une part donner une implantation appropriée aux objets distants des méthodes de `java.lang.Object` et d'autre part réaliser explicitement l'exposition des objets distants en appelant la méthode statique **exportObject** de la classe **java.rmi.server.UnicastRemoteObject**.

La manière la plus simple de définir une classe implantant une interface pour un objet distant est la suivante:

```
import java.rmi.*;
public class MonObjetDistant extends java.rmi.server.UnicastRemoteObject
    implements MonInterface {
    /* définition de la classe (objet, méthodes) */
}
```

Exemple : la classe **CompteurImpl** définie ci-après implante l'interface **Compteur**. On y note:

- la classe **CompteurImpl** d'une part implante **Compteur** et d'autre part étend **UnicastRemoteObject**;
- une définition de chacune des méthodes de l'interface. Chacune de ces fonctions prend en compte la possibilité d'une occurrence d'exception;
- la définition d'un constructeur **CompteurImpl** qui est indispensable car spécifiant l'occurrence possible d'une **RemoteException**. Ce constructeur ne sera utilisable que par un serveur;
- les différentes méthodes **incr**, **getValue** et **getAccess** qui modifient des variables sont spécifiées **synchronized** afin d'assurer leur atomicité.



```
import java.rmi.*;
import java.rmi.server.*; // pour UnicastRemoteObject

public class CompteurImpl extends UnicastRemoteObject implements Compteur {
    private int cpt; // champ valeur du compteur
    private int ref; // champ comptant de le nombre de références à l'objet
    public CompteurImpl(int n) throws RemoteException {
        cpt = n;
        ref = 0; }
    public void incr() throws RemoteException {
        synchronised {
            ref++;
            cpt++; }
    }
    public int getValue() throws RemoteException {
        ref++;
        return cpt; }
    public int getAccess() throws RemoteException {
        return ref; }
}
```

II.2.2.3.3 Génération stub/skeleton

Il s'agit maintenant de générer le **stub** et le **skeleton** de l'objet distant. Ceci se fait en utilisant le compilateur RMI fournit dans le j2sdk.

```
--> javac Compteur.java
--> javac CompteurImpl.java
--> rmic CompteurImpl
```

Deux fichiers Compteur_skel et Compteur_stub sont alors créés dans le package courant.

En Corba, on utilise le compilateur idlj pour générer ces fichiers du langage IDL vers Java ainsi que de nombreux autres fichiers. Par exemple, pour une interface Compteur déclaré en idl, seront générés les fichiers suivants : Compteur.java, _CompteurStub, CompteurHelper.java, CompteurHolder.java, CompteurOperations.java et CompteurPOA.

II.2.3 Déclaration de l'interface

Il faut ensuite définir l'objet (l'exposer) afin de le rendre visible aux clients. C'est le premier rôle de l'application serveur. Un second est de faire connaître l'existence de l'objet par l'intermédiaire d'un service de nommage.

Voici les différentes opérations qui doivent être codées dans le serveur:



- définir un gestionnaire de sécurité (objet de la classe **RMISecurityManager**) qui autorisera le chargement depuis une autre application de classes sérialisables. Cette opération est réalisée par la séquence

```
if(System.getSecurityManager() == null)
    System.setSecurityManager (new RMISecurityManager());
```

Pour pouvoir utiliser les objets partagés ayant déclaré un SecurityManager, il faut, au niveau du serveur, déclarer dans le fichier `java.policy` une règle permettant aux clients d'utiliser ces objets.

L'ajout de règles se fait dans le fichier ***java.policy***, situé dans le sous-répertoire ***lib/security*** du répertoire home de JAVA à l'aide de l'utilitaire ***policytool***.

Lancer l'utilitaire ***policytool*** et ouvrir le fichier `java.policy`. Sélectionner 'Codebase <ALL>' afin d'ajouter une règle s'appliquant pour toutes les applications. Pour cela, cliquer sur 'modifier une règle'. Ajouter ensuite la nouvelle règle avec pour paramètres :

- SocketPermission : `java.net.SocketPermission`
- nom de cible : *
- actions : `accept, connect, resolve`.

Enregistrer le fichier `java.policy`.

- créer les objets distants par instanciation de classes Remote:

```
MonObjetDistant unObjet = new MonObjetDistant;
```

- exposer les objets, c'est-à-dire les rendre accessibles dans le serveur : cette opération a comme effet de créer et de lancer le serveur de l'objet exposé sous la forme d'une thread. Si la classe de l'objet a été définie comme une extension de ***UnicastRemoteObject***, cette exposition est automatique. Par contre, si ce n'est pas le cas, l'objet doit être exposé explicitement par appel de la méthode statique ***exportObject*** de la classe ***UnicastRemoteObject***:

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

UnicastRemoteObject.exportObject(unObjet);,

- Faire connaître l'existence des différents objets au serveur de noms (**Object Registry**) de la machine sur laquelle ce serveur s'exécutera. Ce serveur de noms accepte comme opérations l'enregistrement, l'effacement et la consultation : ces opérations sont réalisées par invocations de méthodes statiques de la classe **java.rmi.Naming** :

- **void Naming.bind(String nom, Remote obj)** : associe l'objet au nom spécifié;
- **void Naming.rebind(String nom, Remote obj)** : réassocie le nom au nouvel objet;
- **void Naming.unbind(String nom)** : supprime l'enregistrement correspondant;
- **Remote Naming.lookup(String nom)** : renvoie la référence (**stub**) de l'objet enregistré sous le nom donné;
- **String[] Naming.list(String nom)** : renvoie la liste des noms enregistrés.

Un serveur contiendra donc pour chaque objet dont il souhaite permettre l'accès à distance une invocation de la forme

Naming.rebind("nomUnObjet", unObjet);

Exemple : nous donnons la définition de la classe **CompteurServeur** correspondant à un serveur exposant un compteur de la classe **CompteurImpl**.

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

public class CompteurServeur{
    public static void main(String[] args) throws Exception{
        if(System.getSecurityManager() == null)
            System.setSecurityManager(new RMISecurityManager());
        CompteurImpl compteur = new CompteurImpl(0);
        Naming.rebind("cpt", compteur);
    }
}
```



L'implantation en Corba, nécessite plus d'étapes pour définir un serveur:

- Initialiser l'ORB.
 - Créer un *servant*.
 - Récupérer le rootPOA
 - Activer le POAManager
 - Référencer le *servant* par le POA.
 - Récupérer le service de nommage.
 - Référencer l'objet dans le service de nommage.
-
- **Lancement du serveur de noms** : le serveur de noms (Object Registry) doit évidemment être activé s'il ne l'a pas encore été avant de pouvoir lancer le serveur d'objets. Ce lancement est réalisé par la commande **rmiregistry**. Le serveur correspondant est associé par défaut au port TCP **1099**. Un numéro de port différent peut être spécifié en paramètre au lancement de la commande. Sur une machine UNIX, le serveur de noms serait lancé en arrière plan par la commande :

```
--> rmiregistry &
```

- **Lancement d'un serveur d'objets**

Cette opération consiste simplement en le lancement d'une application Java. On lance un serveur de compteurs de la classe **CompteurServeur**, ici sous un système unix :

```
--> java CompteurServeur&
```

II.2.3.1.1 Réaliser des appels distants

Enfin nous allons réaliser des appels distants au travers d'applications clientes sur les objets partagés.

- Nous décrivons brièvement les différentes opérations que devra réaliser un client afin de pouvoir accéder à un objet distant (pour lequel évidemment un serveur du type



précédemment décrit est supposé avoir été lancé avec succès).

- Le client doit tout d'abord s'adresser au serveur de noms auprès duquel l'objet auquel il souhaite accéder a été enregistré afin de récupérer un **stub** de l'objet (objet de la classe **Remote**). Cette opération est réalisée en invoquant la méthode statique **lookup** de la classe **Naming**. Pour cela, il doit nommer complètement l'objet concerné au travers d'une URL de la forme générale suivante :

rmi://machine:port/nom

Dans une telle URL,

- si le nom de machine est omis, la machine locale est considérée;
- si le numéro de port est omis, le numéro par défaut (1099) est considéré.

La méthode lookup renvoyant un objet de la classe générique Remote, un **cast** de ce résultat sur la classe de l'interface doit être réalisé par le client.

Remarque importante : la classe implantant l'objet n'est pas accessible au client. Il doit utiliser celle définissant l'interface.

- comme lors de l'enregistrement d'un objet par un serveur, la méthode lookup est susceptible de provoquer une exception (**RemoteException**). Il est donc nécessaire d'en prévoir l'occurrence.
- La récupération d'un stub par un client a ainsi la forme générale suivante:

```
MonInterface stub;  
try {  
    stub = (MonInterface)Naming.lookup(adresse_de_l'objet);  
} catch (Exception e){----- }
```

- Une fois le stub récupéré, un client peut invoquer sur ce talon des méthodes de l'interface comme il ferait sur un objet local:

```
objet=stub.méthode(paramètres);
```

Exemple



La classe **CompteurClient** accède au compteur publié sous le nom **cpt** par le serveur lancé précédemment. L'application correspondante doit être appelée avec un premier paramètre donnant le nom DNS du site hébergeant l'objet et :

- sans autre paramètre, la méthode **getAccess** sur le compteur correspondant (nombre d'accès);
- avec un seul autre paramètre, la méthode **getValue** sur le compteur correspondant (valeur du compteur);
- dans les autres cas, la méthode **incr** sur le compteur.

```
import java.rmi.*;
import java.rmi.server.*;
public class CompteurClient{
    public static void main(String[] args) throws Exception{
        if(args.length == 0){
            System.err.println("bad usage");
            System.exit(2);
        }
        Compteur co = (Compteur) Naming.lookup("rmi://" + args[0] + "/cpt");
        if(args.length == 1)
            System.err.println(co.getref());
        else if (args.length == 2)
            System.err.println(co.getvalue());
        else
            co.incr();
    }
}
```

L'implantation en Corba, nécessite les étapes suivantes pour définir un client:

- Initialiser l'ORB.
- Récupérer le service de nommage.
- Récupérer la référence de l'objet partagé
- Invoquer la (ou les) méthode(s) distante(s) sur cet objet.

Un exemple d'exécutions successives de client est donné ci-après (**localhost** est le nom de la machine où le serveur est lancé):

```
--> CompteurClient localhost xxx
0
--> CompteurClient localhost xxx
0
--> CompteurClient localhost xxx
0
--> java CompteurClient localhost
3
--> java CompteurClient localhost aaa bbb
--> java CompteurClient localhost aaa
1
--> java CompteurClient localhost
5
--> java CompteurClient localhost aaa
1
```

II.2.4 Le callback

Nous allons ici essayer de répondre à la problématique du callback avec RMI car cette fonctionnalité pourrait s'avérer intéressante pour le logiciel. Une des méthodes qui s'offre à nous est de déclarer des clients étant eux mêmes des serveurs RMI afin que le serveur puisse entrer en contact avec ces clients et leurs envoyer des informations. C'est ce qui est mis en oeuvre dans l'exemple du package **de.berlios.chronos.technique.rmi.horloge**, dans lequel, un serveur horloge possède un objet partagé, et les clients se connectant à cet objet vont s'enregistrer et lui donner une référence vers eux mêmes, afin que ce dernier puisse leur communiquer l'évolution de l'heure toutes les 2 secondes.

II.2.4.1 L'objet côté serveur

L'interface **TimeServer**, qui représente un objet serveur de temps, possède une méthode `registerTimeMonitor` (Monitor pour client) avec laquelle le client appelant va pouvoir enregistrer sa référence afin que ce dernier puisse le recontacter grâce à la méthode **tellMeTheTime()** de l'interface **TimeMonitor** acceptant une `Date` en paramètre.



```
import java.rmi.*;

public interface TimeServer extends java.rmi.Remote
{
    public void registerTimeMonitor( TimeMonitor tm) throws RemoteException;
}
```

Cette interface est implémentée par l'objet **RMIServer** suivant :

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
public class RMIServer implements Remote, TimeServer {
    ...
    public static void main ( String[] args ) {
        ...
        rmi = new RMIServer();
        LocateRegistry.createRegistry( PORT );
        System.out.println( "Registry created" );
        UnicastRemoteObject.exportObject( ((TimeServer)rmi) );
        Naming.rebind( "//" + HOST_NAME + ":" + Integer.toString( PORT ) + "/" +
            "TimeServer", rmi );
        System.out.println( "Bindings Finished" );
        System.out.println( "Waiting for Client requests" );
        ...
    }
    public void registerTimeMonitor( TimeMonitor tm ) {
        System.out.println( "Client requesting a connection" );
        TimeTicker tt;
        tt = new TimeTicker( tm );
        tt.start();
        System.out.println( "Timer Started" );
    }
}
```

L'objet RMI serveur implémente donc la méthode **registerTimeMonitor()** de l'interface **TimeServer**. Cette méthode crée un nouvel objet **TimeTicker** qui sera un thread chargé de communiquer avec les clients se connectant sur l'objet partagé. Nous verrons plus loin la déclaration de cet objet.

Note: Les ... indique les instructions try... catch sur les différentes exceptions pouvant avoir



lieu (cf \$RMI).

II.2.4.2 L'objet partagé côté client

Du côté du serveur, on déclare aussi un objet partagé **TimeMonitor** contenant une seule méthode **tellMeTheTime()** permettant au serveur de donner la nouvelle date toutes les 2 secondes aux clients.

```
import java.rmi.*;
import java.util.Date;

public interface TimeMonitor extends java.rmi.Remote
{
    public void tellMeTheTime( Date d ) throws RemoteException;
}
```

Voici maintenant une applet cliente qui va implanter l'interface **TimeMonitor** et va se charger de contacter le serveur d'horloge, de s'enregistrer auprès de ce dernier et d'afficher l'heure.



```
import java.rmi.*;

class Applet1 extends Applet implements TimeMonitor {
    public void init() {
        ...
        try {
            System.out.println( "Exporting the Applet" );
            UnicastRemoteObject.exportObject( this );
            URL base = getDocumentBase();
            String hostName = base.getHost();
            if ( 0 == hostName.length() ) {
                hostName = HOST_NAME;
            }
            String serverName = "rmi://" + hostName + ":" + getParameter
            ( "registryPort" ) + "/TimeServer" ;
            System.out.println( "Looking up TimeService at: " + serverName );
            try {
                ts = (TimeServer)Naming.lookup( serverName );
            }
            catch ( Exception e ) {
                System.out.println( "" + e );
            }
            ts.registerTimeMonitor( this );
            System.out.println( "We have been registered!" );
        }
        catch ( RemoteException re ) {
            System.out.println( "" + re );
        }
    }
    public void tellMeTheTime( Date d ) {
        textArea1.appendText( d.toString() + "\n" );
    }
}
```

Cette classe implante donc l'interface **TimeMonitor** et définit donc la méthode **tellMeTheTime()** qui se charge de rafraîchir l'affichage de la date. C'est cette méthode qui sera utilisée lors du *callback*. La méthode **init()** de l'applet se charge de contacter l'objet partagé du serveur d'horloge et exécute la méthode distante **registerTimeMonitor()** avec sa propre référence, ce qui permettra au serveur de pouvoir contacter ce client.

II.2.4.3 L'objet réalisant le *callback*

Enfin, du côté du serveur, nous allons maintenant voir comment se déroule la communication callback qui s'effectue grâce à l'objet **TimeTicker**. Cet objet est ici déclaré dans le même fichier que l'objet **RMIServer**.



```
class TimeTicker extends Thread {
    private TimeMonitor tm;
    TimeTicker( TimeMonitor tm ) {
        this.tm = tm;
    }
    public void run() {
        while ( true ) {
            try {
                sleep( 2000 );
                tm.tellMeTheTime( new Date() );
            }
            catch ( Exception e ) {
                stop();
            }
        }
    }
}
```

L'objet **TimeTicker** va donc avoir pour rôle d'exécuter toutes les 2 secondes, la méthode distante **tellMeTheTime()** de l'objet partagé résidant sur le client dont la référence a été passé en argument.

II.2.5 Le plugin RMI pour Eclipse

Il s'agit ici de faire une brève présentation du plugin RMI pour Eclipse afin de voir ce qu'il peut nous apporter.

II.2.5.1 Installation

La dernière version plugin « RMI for Eclipse », se télécharge à l'adresse <http://www.genady.net/rmi/>. Après avoir installé le plugin (il suffit de décompresser le zip à la racine d'Eclipse), il faut rentrer son numéro de licence accessible par le menu 'Windows' -> 'Preferences' -> 'Java' -> 'RMI'. Une licence globale pour l'équipe a été demandée. Après avoir saisi et enregistré ce numéro de licence, sélectionner le projet RMI en cours, et à l'aide du menu contextuel de la souris, sélectionner l'option 'RMI' -> 'Enabled Stub Generation'. Il faut enfin, dans les propriétés de ce projet (Project -> Properties), éditer les options du 'RMI compiler properties' afin de changer l'option de la zone 'protocol selection' à '-v compat' et



d'ajouter l'option '-keep'.

II.2.5.1.1 Utilisation

L'option 'Enabled Stub Generation' sélectionnée précédemment permet au plugin de générer automatiquement les fichiers **stub** et **skeleton** d'un code java. On peut demander manuellement la régénération de ces fichiers par le même menu. Le plugin permet aussi de lancer le rmiregistry avec le menu icône RMI apparaissant dans la fenêtre principale. Il permet aussi bien sûr de lancer les différents serveurs et clients RMI avec la possibilité de spécifier différentes options, comme l'utilisation du rmiregistry.

II.2.6 Conclusion

Nous avons donc vu les différentes étapes menant à la construction d'une application fonctionnant avec des objets partagés avec RMI. Cette bibliothèque apparaît comme plus simple et moins lourde à mettre en place qu'une implantation suivant le standard Corba. Ce gain s'explique notamment par le fait que RMI se destine spécifiquement au langage JAVA. Étant de plus intégré en standard dans les distributions JAVA, il n'y a pas d'installation spécifique à effectuer et de nombreux outils, analogues à ceux utilisés pour Corba, sont disponibles, permettant de mettre rapidement en place des applications de ce type.

Au niveau du callback, l'exemple proposé ici repose sur une solution comportant plusieurs serveurs d'objets partagés, ce qui a l'inconvénient de s'avérer plus complexe à mettre en place. Cette complexité tient du fait que chaque poste client est donc aussi serveur, qu'il faut donc configurer au même titre que le serveur central. Les ressources des postes clients sont donc plus utilisées et on peut opposer cette solution à une implémentation dans laquelle chaque client fait lui-même une attente active (à l'aide d'un thread). L'avantage de la solution avec RMI vient du fait que tout ce qui est nécessaire pour communiquer (protocole, format...) de manière efficace entre les différents éléments de l'application (serveur-clients) est déjà disponible.

II.3 PostgreSQL / MySQL

Cette étude permet de choisir une base de données adaptée au logiciel Chronos. Elle portera sur deux base de données openSource.

II.3.1 MySQL

MySQL est conçue à l'origine pour assurer le support de sites Web dynamiques. Depuis, son socle fonctionnel n'a cessé d'évoluer. Ce qui lui permet dès lors de conquérir de nouveaux domaines d'applications, le champ des environnements de données critiques notamment.

MySQL présente une communauté d'utilisateurs très développée et est plus rapide en terme d'accès que PostgreSQL.

II.3.2 PostgreSQL

PostgreSQL est un système de gestion de bases de données (SGBD) reposant sur un modèle orienté objet. Une technologie qui ouvre des horizons beaucoup plus larges qu'un dispositif relationnel classique, tel que celui de MySQL. Elle permet par exemple de mieux et plus directement prendre en compte les notions de classes, d'héritage et de surcharge, facilitant ainsi la programmation de logiques applicatives plus complexes. Autre différence à noter : aux côtés d'une bibliothèque d'interfaces de programmation particulièrement riche, PostgreSQL dispose d'un langage de procédures qui lui est propre.

Ses points forts :

- PostgreSQL dispose d'un dispositif de contrôle de l'intégrité de la structure et de la cohérence des données ;
- PostgreSQL gère les accès concurrents ;
- Sous forte charge l'accès reste rapide en lecture et écriture ;
- PostgreSQL utilise les langages procéduraux sur le serveur ;
- PostgreSQL utilise les transactions ;
- PostgreSQL utilise les procédures stockées ;
- PostgreSQL gère les clés étrangères, les triggers et les vues (ce qui permet de masquer la complexité de la base de données depuis l'application, évitant ainsi la création de commandes SQL compliquées) ;
- PostgreSQL intègre un langage de procédures (baptisé PL/pgSQL) supportant notamment les boucles et la déclaration de variables.



II.3.3 Conclusion sur PostGreSQL / MySQL

Le fait que PostGreSQL gère les accès concurrents ainsi que les clés étrangères (choses non gérées par MySQL) fait que nous optons pour celle-ci pour le développement de Chronos.



Annexes



Proposition Architecture Technique

Aurélien PIERRON
Baptiste MATHUS
Hakima ZIDOURI
Michel VONGVILAY
Samuel MARMECHE



Historique

Version	Document créé par	Le
1	Samuel MARMECHE	19 déc 2004

Version	Document modifié par	Motif	Le

Version	Document validé par	Le
1	Aurélien PIERRON	27 oct 2004



Table des matières

I Introduction.....	5
II Les technologies utilisées.....	6
II.1 Applicatif.....	6
II.1.1 TomCat / JSP.....	6
II.1.1.1 Avantages.....	6
II.1.1.2 Inconvénients.....	6
II.1.2 Serveur MySQL.....	6
II.1.2.1 Avantages.....	6
II.1.2.2 Inconvénients.....	6
II.1.3 Serveur PostGreSQL.....	6
II.1.3.1 Avantages.....	6
II.1.3.2 Inconvénients.....	6
II.2 Programmation.....	7
II.2.1 RMI.....	7
II.2.1.1 Avantages.....	7
II.2.1.2 Inconvénients.....	7
II.2.2 JDBC.....	7
II.2.2.1 Avantages.....	7
II.2.2.2 Inconvénients.....	7
II.2.3 Hibernate.....	7
II.2.3.1 Avantages.....	7
II.2.3.2 Inconvénients.....	7
II.3 Note.....	7
III Proposition.....	8
III.1 Architecture TomCat / Hibernate.....	8
III.1.1 Organisation.....	8
III.1.2 Avantages.....	10
III.1.3 Inconvénients.....	10
III.2 Architecture JDBC / MySQL.....	11
III.2.1 Organisation.....	11
III.2.2 Avantages.....	12
III.2.3 Inconvénients.....	12
III.3 Architecture RMI / Hibernate.....	13
III.3.1 Organisation.....	13
III.3.2 Avantages.....	13
III.3.3 Inconvénients.....	13
III.4 Architecture XML.....	14
III.4.1 Organisation.....	14
III.4.2 Avantages.....	15
III.4.3 Inconvénients.....	15
IV Synthèse.....	16
IV.1 Comparatif des solutions.....	16
IV.2 Explication de la notation.....	17
IV.2.1 Installation / Mise en oeuvre / Administration.....	17
IV.2.1.1 Maintenance.....	17
IV.2.1.2 Déploiement.....	17



IV.2.1.3 Installation outils.....	17
IV.2.2 Technique.....	17
IV.2.2.1 Programmation.....	17
IV.2.2.2 Standard.....	17
IV.2.3 Fiabilité / Sécurité.....	18
IV.2.3.1 Fiabilité services.....	18
IV.2.3.2 Fiabilité données.....	18
IV.2.3.3 Stabilité application.....	18
IV.2.3.4 Sécurité données.....	18
IV.2.4 Performances / Ressources.....	18
IV.2.4.1 Rapidité exécution.....	18
IV.2.4.2 Ressources système utilisateur.....	19
IV.2.4.3 Ressources serveur.....	19
IV.2.4.4 Trafic généré.....	19



III Introduction

Ce document est une synthèse des propositions d'architectures techniques envisagées pour la réalisation et le déploiement du projet 'Chronos'. Le but de ce document est de détailler chaque proposition d'architecture envisagée en faisant apparaître pour chacune les avantages et les inconvénients au niveau technique, installation, fiabilité, sécurité ou encore performances, puis de fournir une synthèse de toutes les architectures étudiées, afin de déterminer plus facilement l'architecture à mettre en oeuvre pour le projet.





IV Les technologies utilisées

Il s'agit ici de lister l'ensemble des technologies pouvant être utilisées pour la réalisation de l'architecture technique du projet 'Chronos'. Chaque technologie est présentée en terme d'avantages et d'inconvénients.

IV.1 Applicatif

IV.1.1 TomCat / JSP

IV.1.1.1 Avantages

Permet de centraliser l'accès à une application. Ce serveur applicatif est très utilisé sur internet => bonne robustesse.

IV.1.1.2 Inconvénients

Ne permet pas de gérer les EJB.

IV.1.2 Serveur MySQL

IV.1.2.1 Avantages

La base de données MySQL est aussi l'une des plus simple à installer, à configurer et à administrer du fait de nombreux outils existants. Elles sont aussi réputées pour leur système de gestion des fichiers corrompus ainsi que leurs performances. Enfin, le langage de requête s'appuie sur le standard SQL. Fonctionnalité de réplication. Forte communauté d'utilisateurs.

IV.1.2.2 Inconvénients

Risque de problème de fiabilité et de sécurité au niveau du serveur MySQL. On peut aussi noter l'absence de certaines commandes (Rollback ...).

IV.1.3 Serveur PostGreSQL

IV.1.3.1 Avantages

La base respecte de nombreux standards : SQL92, SQL99, SQL ansi, XML. Gestion des subselects, clés étrangères, vues, procédures, fonctions, accès concurrents. Système relationnel objet.

IV.1.3.2 Inconvénients

Installation pas très simple. Pas très rapide.



IV.2 Programmation

IV.2.1 RMI

IV.2.1.1 Avantages

L'API RMI de Java permet d'utiliser le protocole de communication IIOP, standard de communication pour les objets partagés, et donc de communiquer avec des objets développés dans un langage différent.

IV.2.1.2 Inconvénients

Pas de mécanisme pour la description des objets. Le client doit connaître l'adresse exacte du serveur d'objets partagés.

IV.2.2 JDBC

IV.2.2.1 Avantages

Le JDBC permet de se connecter à de nombreuses base de données. S'appuie sur SQL

-> standard.

IV.2.2.2 Inconvénients

Toutes les limitations des langages relationnels.

IV.2.3 Hibernate

IV.2.3.1 Avantages

L'installation du framework est facile à mettre en oeuvre. Ce dernier permet la manipulation par objets pour la sauvegarde des données dans la base (langage de requête HQL Orienté Objet).

IV.2.3.2 Inconvénients

Le framework ne repose pas sur un standard et entraîne une augmentation de l'utilisation de l'espace mémoire lors de la sauvegarde.

IV.3 Note

Lors de ce projet, nous allons être amené à utiliser un annuaire LDAP pour la gestion de l'authentification. Cette annuaire n'apparaît pas dans la description des différentes architectures car c'est un point imposé par le client.

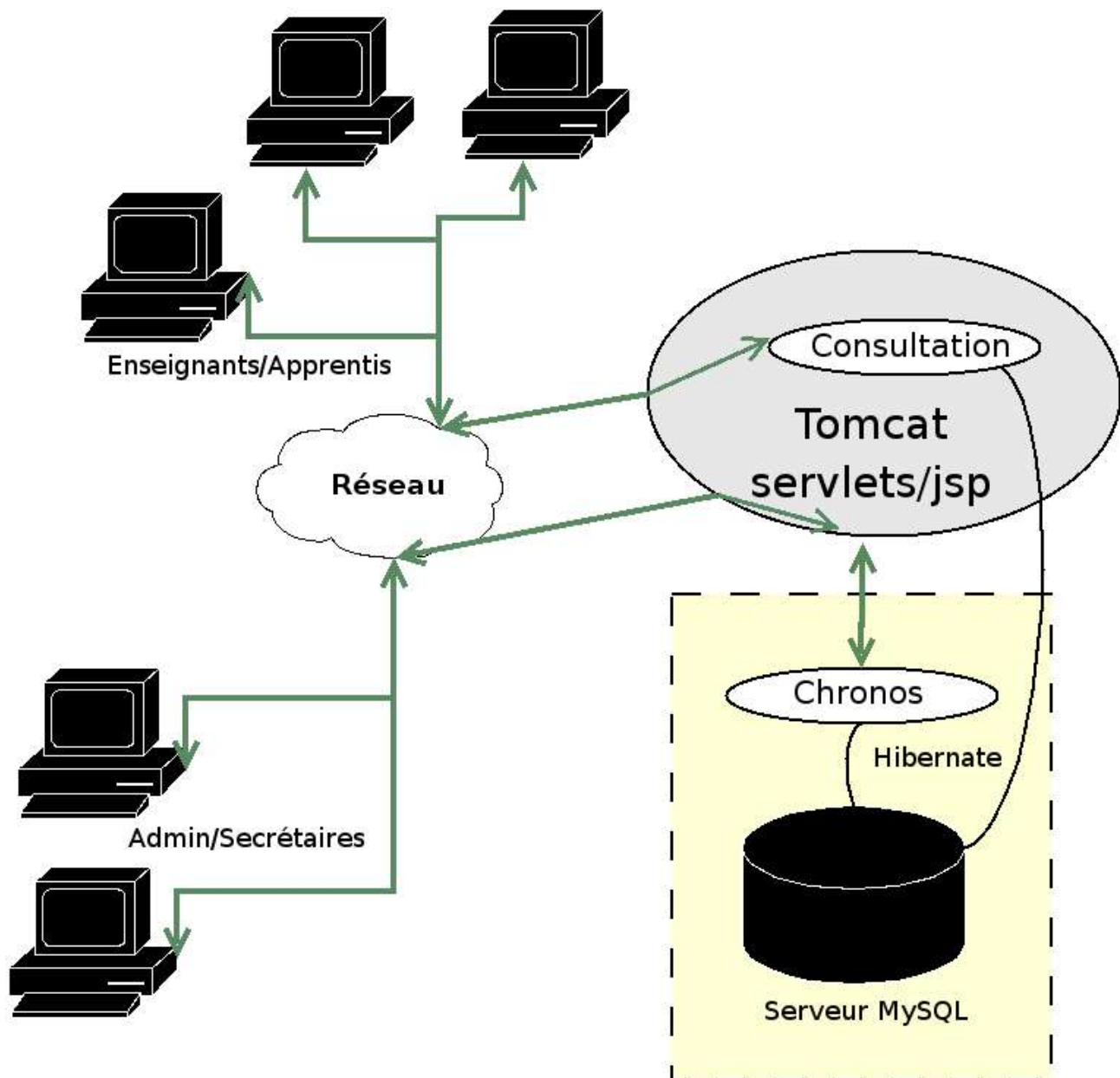


V Proposition

Ce chapitre regroupe les différentes propositions envisagées pour l'architecture technique en essayant de faire apparaître les avantages et les inconvénients de chaque architecture.

V.1 Architecture TomCat / Hibernate

V.1.1 Organisation



V.1.2 Avantages

Facilité de mise en oeuvre et de déploiement de l'application au niveau de l'utilisateur, car il suffit d'un navigateur internet et du plugin java. L'application n'est à installer qu'une seule fois sur le serveur TomCat, il n'y a donc qu'une seule configuration un peu plus lourde à réaliser. Il y a aussi le fait que les futures modifications ou corrections de l'application ne sont à faire qu'au niveau de ce serveur. Du point de vue de la sécurité, les utilisateurs n'auront pas directement accès à la base de données et cette gestion centralisée au niveau du serveur permettra le traitement de la concurrence dans les accès à la base de données. Enfin, ce type d'application nécessite peu l'utilisation des ressources de l'utilisateur. Une seule installation



pour le serveur TomCat (serveur applicatif **et** de consultation).

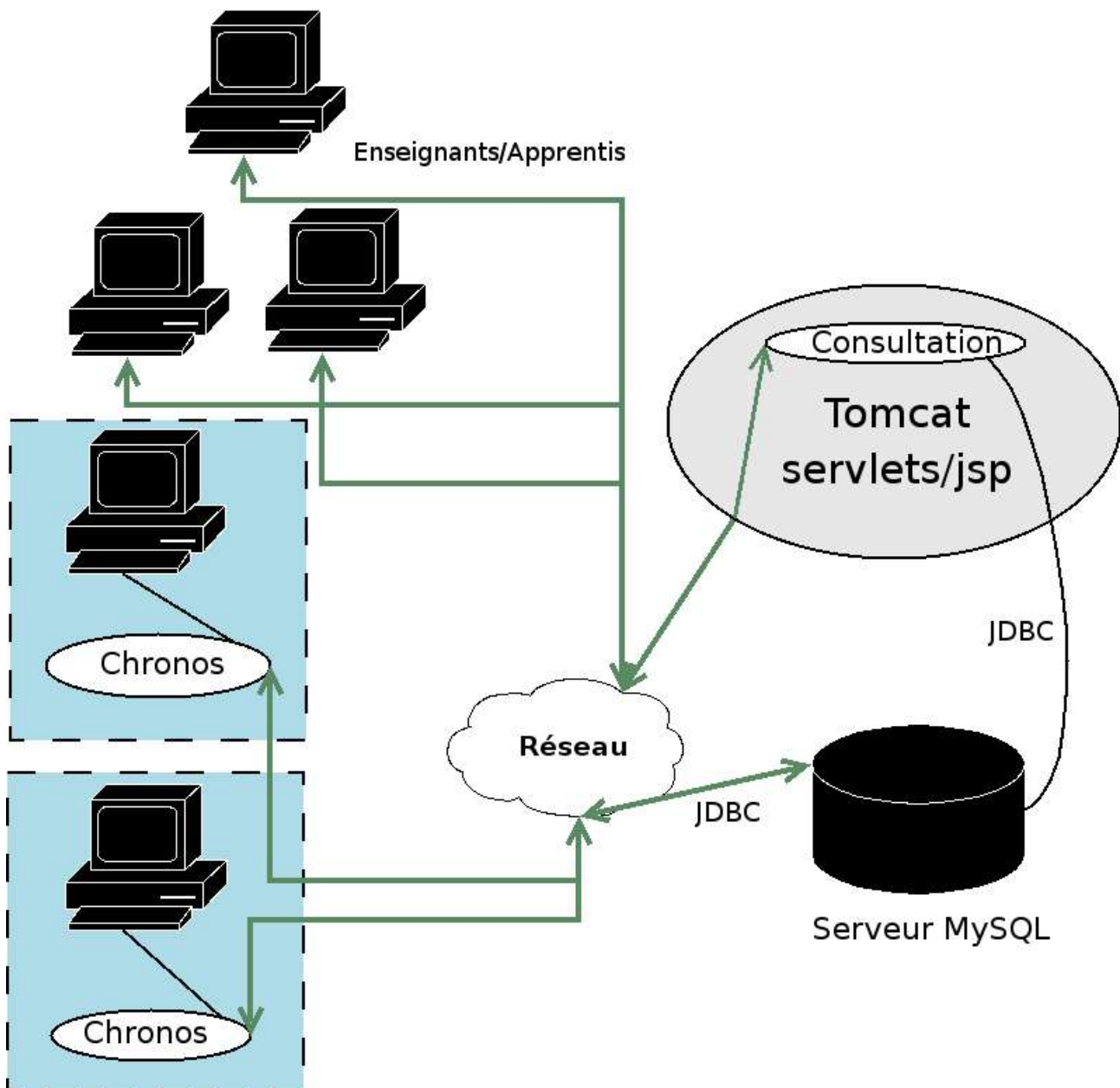
V.1.3 Inconvénients

Configuration et installation du serveur TomCat.

La rapidité de l'exécution de l'application au niveau des utilisateurs dépend en majorité des qualités en terme de performances de la connexion avec le serveur distant. De même, l'application présente doit aussi réaliser des connexions avec le serveur MySQL qui peut ne pas être sur la même machine.

V.2 Architecture JDBC / MySQL

V.2.1 Organisation



V.2.2 Avantages

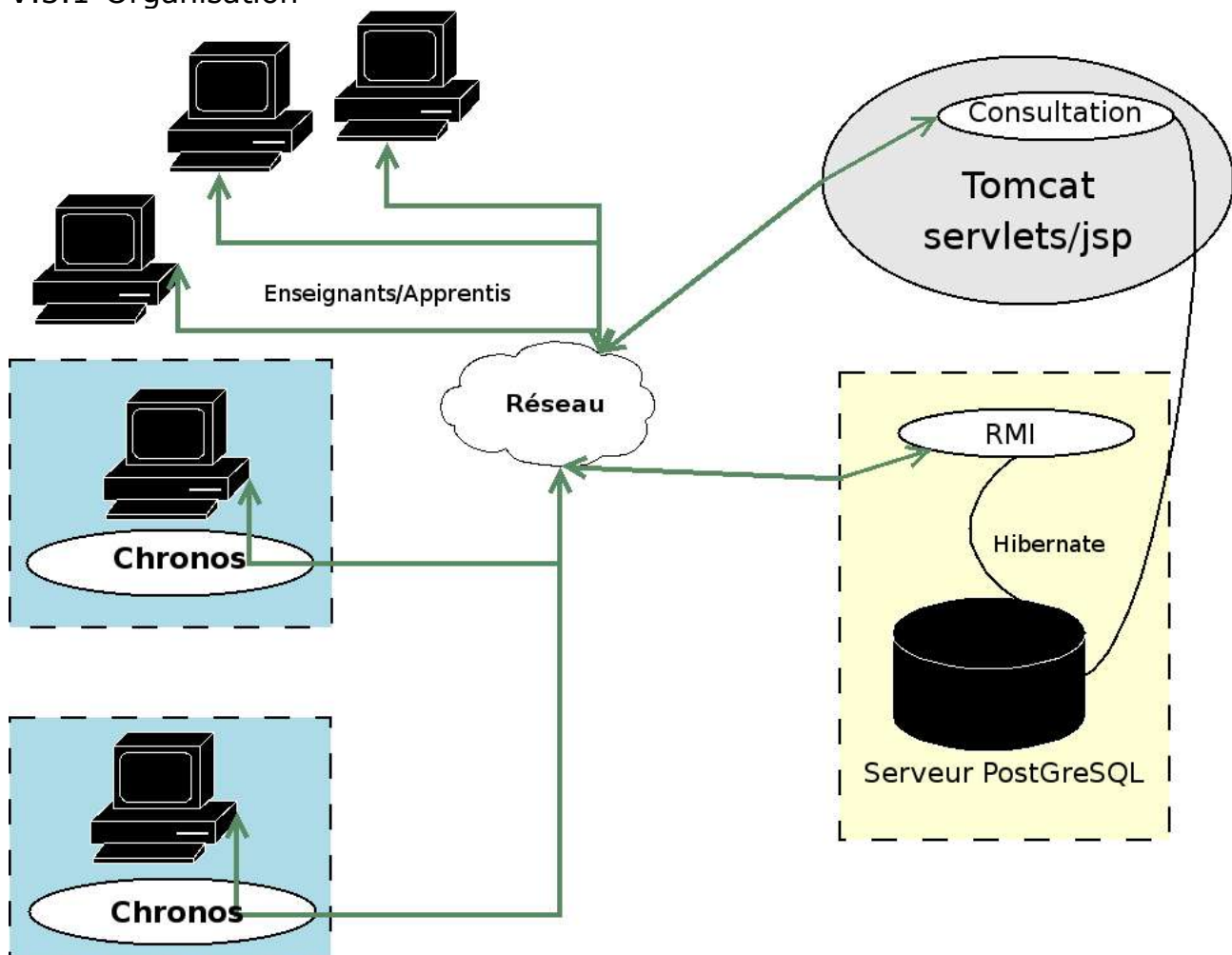
Rapidité d'exécution de l'application car celle-ci est disponible localement.

V.2.3 Inconvénients

Le déploiement et l'installation de l'application finale sont à effectuer sur chaque poste utilisateur. Même problème pour les évolutions ou les corrections de bugs. Il n'y a pas de gestion de la concurrence au niveau des écritures dans la base de données. L'exécution de l'application impacte les performances du système de l'utilisateur.

V.3 Architecture RMI / Hibernate

V.3.1 Organisation



V.3.2 Avantages

Le serveur doit juste posséder un environnement java car RMI est intégré dans le j2sdk. Il y a aussi la possibilité de gérer la concurrence des accès aux données grâce aux objets distants.

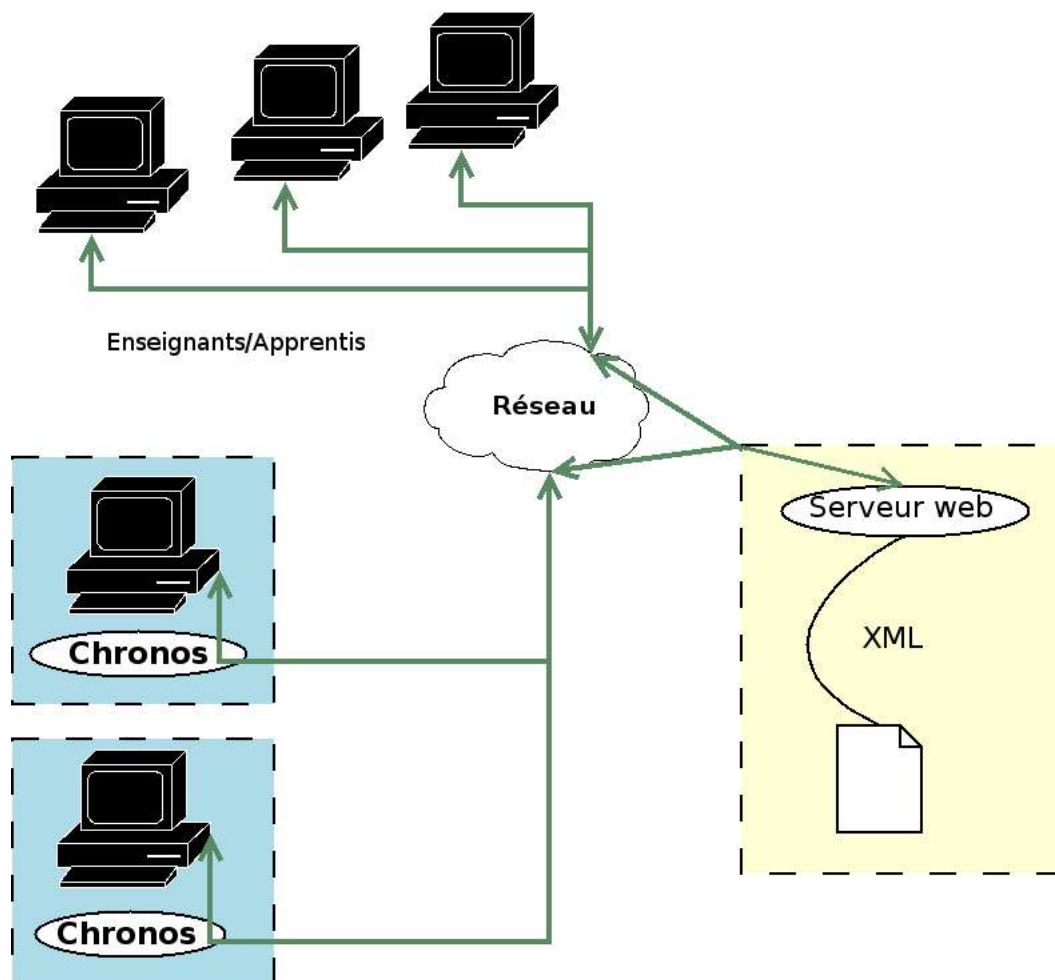
Disponibilité de l'interface utilisateur en local, donc plus de rapidité.

V.3.3 Inconvénients

Le déploiement et l'installation de l'application finale est à effectuer sur chaque poste utilisateur.

V.4 Architecture XML

V.4.1 Organisation





V.4.2 Avantages

Installation rapide du serveur distant qui servira à héberger les fichiers xml de sauvegarde. On peut aussi noter la rapidité d'exécution de l'application.

V.4.3 Inconvénients

Déploiement et installation de l'application finale à effectuer sur chaque poste utilisateur. Mise à jour obligatoire sur chaque machine. En outre la gestion des sauvegardes en XML entraîne une forte limitation au niveau des outils de gestion des données en terme de temps d'accès ou de recherche. Il n'y a aucune gestion de la concurrence des écritures pour la sauvegarde, aucune sécurité. La stabilité de l'application dépend de la stabilité du réseau. Enfin, le fait d'utiliser un client lourd entraîne une utilisation plus intense des ressources du système de l'utilisateur.



VI Synthèse

VI.1 Comparatif des solutions

	Coeff	Architecture TomCat / Hibernate	Architecture JDBC / MySQL	Architecture RMI / Hibernate	Architecture XML
Installation / Mise en oeuvre / Administration (coeff 4)					
Maintenance	0,5	17	15	13	15
Déploiement	2	17	14	14	14
Installation outils	1,5	16	14	14	15
Technique (coeff 9)					
Programmation	7	15	13	16	11
Standard	2	13	16	16	14
Fiabilité / Sécurité (coeff 15)					
Fiabilité services	3	14	15	14	15
Fiabilité données	4	16	16	17	5
Stabilité application	3	13	14	12	14
Sécurité données	5	15	9	16	0
Performances / Ressources (coeff 12)					
Rapidité exécution	5	13	15	12	16
Ressources système utilisateur	4	15	12	16	12
Ressources serveur	2	13	14	13	16
Trafic généré	1	13	15	12	16
TOTAL	40	14,54	13,54	14,69	11,15



VI.2 Explication de la notation

VI.2.1 Installation / Mise en oeuvre / Administration

VI.2.1.1 Maintenance

La maintenance de l'application est le fait de pouvoir mettre à jour cette dernière plus ou moins facilement en cas de correction de bug ou d'évolution. C'est pour cette raison que la meilleure note va à l'architecture TomCat, car l'application est centralisée, il ne faut donc mettre à jour qu'un seul système. Pour cette même raison, l'architecture XML et JDBC/MySQL sont à peu près équivalentes en terme d'effort pour la mise à jour, qui concerne ici les postes des utilisateurs donc un effort plus important. Enfin l'architecture RMI, possède un code côté utilisateur qu'il faut faire évoluer mais aussi un code du côté du serveur.

VI.2.1.2 Déploiement

Le déploiement repose sur les mêmes critères que la mise à jour, mais en ne prenant en compte que le niveau de la première installation de l'application en terme d'effort d'installation au niveau de l'utilisateur seulement. Pour l'architecture TomCat, il n'y a pratiquement aucun travail à fournir côté utilisateur et pour les autres, l'effort est à peu près équivalent.

VI.2.1.3 Installation outils

L'installation des outils concerne la configuration des différents services au niveau du serveur. Donc de la facilité d'installation des différentes technologies utilisées. Le classement de l'architecture la plus facile à la plus difficile à installer est : XML, JDBC/MySQL, RMI/Hibernate, Tomcat/Hibernate.

VI.2.2 Technique

VI.2.2.1 Programmation

On tient compte de la difficulté de programmation avec les différentes techniques choisies. Par exemple XML (Architecture XML) est plus difficile à utiliser pour faire de la sauvegarde qu'une base de données (Architecture JDBC). Hibernate facilite encore plus ce procédé.

VI.2.2.2 Standard

On prend en compte la conformité à un standard des différentes technologies utilisées et on préférera une solution reposant sur des standards éprouvés.



VI.2.3 Fiabilité / Sécurité

VI.2.3.1 Fiabilité services

Donne une indication sur les différentes architectures en terme de fiabilité ou de continuité dans les différents services proposés par l'application. On considère qu'une application est plus fiable qu'une autre dans la gestion de ses services par le degré d'accessibilité des différents services externes utilisés par l'application (Il ne s'agit pas de la stabilité au niveau du utilisateur mais il est sur que cela impact). Typiquement, une architecture reposant sur des objets partagés (Architecture RMI) peut s'avérer moins fiable au niveau des ses services qu'une application fonctionnant en mode entièrement local (Architecture JDBC).

VI.2.3.2 Fiabilité données

La fiabilité des données comprend la gestion de la corruption des données réalisée par les différents systèmes de sauvegarde et la gestion de la concurrence des écritures. Typiquement, une base de données (Architecture JDBC) sera plus fiable qu'un simple fichier xml (Architecture XML) mais moins fiable qu'un système centralisé permettant la gestion de la concurrence (Architecture TomCat et RMI).

VI.2.3.3 Stabilité application

Concerne la stabilité de l'application au niveau de l'utilisateur exclusivement. Le degré de stabilité de cette application est en grande partie déterminé par le fait qu'il s'agisse d'un client local (Architecture JDBC et XML) ou distant (Architecture TomCat et RMI).

VI.2.3.4 Sécurité données

La sécurité des données est un critère important de l'application car le bon fonctionnement des emplois du temps en dépend. Ainsi on distingue :

- l'architecture XML qui utilise un fichier pour sauvegarder ses données et ne fournit donc aucun niveau de sécurité
- l'utilisation directe de la base de données (Architecture JDBC) qui dans le cas de MySQL fournit un niveau de sécurité faible
- et les autres architectures qui centralisent et masquent les accès à la base de données et fournissent donc un niveau de sécurité plus élevé.



VI.2.4 Performances / Ressources

VI.2.4.1 Rapidité exécution

La rapidité d'exécution concerne l'application finale. Cette rapidité peut être impactée par la connexion réseau que ce soit en raison de programmation par objets partagés ou pour réaliser une connexion avec la base de données. Ainsi, on considère qu'une application locale (client lourd) réalisant des connexions avec une base de données distante (Architecture JDBC et XML) est plus rapide qu'une application web (Architecture TomCat) ou qu'une application contenant des objets partagés (Architecture RMI).

VI.2.4.2 Ressources système utilisateur

On note ici l'utilisation faite par l'application du système informatique (ordinateur) de l'utilisateur. Il est clair qu'une application web (Architecture TomCat) utilise moins de ressource qu'une application partagée (Architecture RMI) qui elle même utilise moins de ressource qu'un client lourd (Architecture JDBC et XML).

VI.2.4.3 Ressources serveur

Les ressources du serveur concerne l'utilisation de l'espace mémoire et des ressources du serveur distant. Ainsi, un client léger (Architecture TomCat) utilisera plus de ressources qu'un client lourd avec une base de données et utilisant JDBC pour le stockage (Architecture JDBC). On considère que Hibernate utilise plus d'espace disque que JDBC lors du stockage.

VI.2.4.4 Trafic généré

On regarde ici le trafic induit par l'application au niveau du réseau. On considère l'utilisation des protocoles de communication et que l'utilisation des objets partagés (Architecture RMI) induit un trafic plus important que la simple connexion avec une base de données (Architecture JDBC).