les Cahiers du Programmeur

J2EE

Jérôme Molière

EYROLLES

les Cahiers du Programmeur

J2EE

Dans la collection
Les Cahiers du programmeur

Les Cahiers du programmeur Java 1.4 et 5.0
Emmanuel PUYBARET – N°11478, 2004. Avec CD-Rom.
À travers plus de cinquante mini-applications et la création d'un forum de discussion, le développeur débutant apprendra à installer le SDK, à utiliser Java et sa bibliothèque standard, à recourir aux applets, servlets et JSP, exploiter une base de données, concevoir des interfaces graphiques, faire de la programmation multi-threading… CD-Rom offert avec Eclipse et JBuilder X.

Les Cahiers du programmeur PHP 5
PHP objet et XML.
Stéphane MARIEL – N°11234, 2004.
L'étude de cas, une application IRC de rencontre sur le Web, tire parti de tout ce qu'offre PHP 5 : design patterns et objets, création de documents XML à la volée, transformation XSL pour des sites accessibles même depuis votre téléphone mobile, utilisation de SQLite…

Les Cahiers du programmeur Zope/Plone
K. AYEVA, O. DECKMYN, P.-J. GRIZEL, M. RÖDER
N°11393, 2004, 216 pages.
Du cahier des charges jusqu'à la mise en production, cette deuxième édition montre comment créer et personnaliser un site intranet d'entreprise avec Plone 2.0. Le développeur découvrira comment gérer différents types de contenu, mettre en œuvre des services de workflow et d'indexation, gérer les droits, les interfaces utilisateur et les formulaires.

Les Cahiers du programmeur UML
Pascal ROQUES – N°11070, 2002.
UML est un outil simple et universel : nullement réservé aux applications Java ou C++, il peut servir à modéliser des sites Web marchands, dont la complexité en fait des candidats naturels à la modélisation. Toutes les étapes de conception sont décrites, abondamment illustrées et expliquées, à travers une démarche située à mi-chemin entre processus lourd et processus léger.

Les Cahiers du programmeur ASP.NET
Thomas PETILLON – N°11210, 2003.
Ce cahier décrit la mise en place d'une base de données publiée et éditable via ASP.NET en VB.NET et C#. Le développeur apprendra à manipuler des données XML, mettre en œuvre des services Web, sécuriser et déployer la base.

Les Cahiers du programmeur PHP/Java Script
Philippe CHALEAT et Daniel CHARNAY – N°11089, 2002.
En une douzaine d'ateliers pratiques, allant de la conception d'aides multi-fenêtrées en JavaScript à la réalisation de services Web, en passant par les templates PHP et les annuaires LDAP, on verra qu'autour de formulaires HTML, on peut sans mal réaliser des applications légères ergonomiques et performantes.

Les Cahiers du programmeur Java/XML
Renaud FLEURY, Caroline de VASSON – N°11316, 2005.
Au fil de la refonte d'un système d'information de site e-commerce, ce cahier illustre les meilleures pratiques J2EE à mettre en oeuvre tant pour la modélisation et la présentation de flux XML que pour la conception de services web.

À paraître dans la même collection

J.-P. LEBOEUF – PHP/MySQL – N°11496, 2e édition 2004

Chez le même éditeur

B. MARCELLY, L. GODARD
Programmation OpenOffice.org
Macros OOoBASIC et API
N°11439, 2004, 700 pages.

R. HERTZOG
Debian GNU/Linux
N°11398, 2004, 300 pages (coll. Cahiers de l'Admin)

E. DREYFUS
BSD, 2e édition
N°11463, 2004, 300 pages (coll. Cahiers de l'Admin)

L. DERUELLE – Développement J2EE avec Jbuilder
N°11346, 2004, 726 pages avec CD-Rom.

H. BERSINI, I. WELLESZ.
L'orienté objet. Cours et exercices en UML 2 avec Java, Python, C# et C++
N°11538, 2004, 520 pages (collection Noire)

A.-L. QUATRAVAUX, D. QUATRAVAUX – Réussir un site web d'association… avec des outils gratuits !
N°11350, 2004, 340 pages.

S. BLONDEEL, D. CARTRON, H. SINGODIWIRJO
Débuter sous Linux avec Knoppix et Mandrake
N°11559, 2005, 440 pages.

S. GAUTIER, C. HARDY, F. LABBE, M. PINQUIER
OpenOffice.org 1,1 efficace
N°11348, 2003, 336 pages avec CD-Rom.

Jérôme **Molière**

les Cahiers du Programmeur J2EE

Avec la contribution
de Stéphane **Bailliez**,
Frédéric **Baudequin** et Gaël **Thomas**

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

© Groupe Eyrolles, 2003, 2005, ISBN : 2-212-11574-1

Avant-propos

Java est né il y a environ huit ans. Depuis lors, le petit monde de l'informatique voit en ce langage un trouble-fête, lent pour certains, révolutionnaire pour d'autres. Loin des mythes et des légendes urbaines, ce langage a fait sa place dans le monde de l'entreprise. Pendant ce temps, le loisir d'un jeune doctorant finlandais se transformait en enjeu économique colossal : Linux. Avec ce nouveau système d'exploitation, le monde découvrait un concept jusque-là réservé à quelques idéalistes : le logiciel libre. Les années 1990 ont fait entrer l'informatique dans une ère industrielle et ce nouvel âge pour l'informatique amène les contraintes inhérentes à toute industrie : procédures et qualité.

Quel est l'objectif de cet ouvrage ?

Ce livre expose comment aborder un développement Java avec une démarche professionnelle et présente des outils (Open Source pour la plupart) apportant de réels plus dans cette optique de qualité professionnelle. Cela signifie que l'application ne doit pas se contenter de faire ce que l'on souhaite à la base, mais qu'elle doit en plus être maintenable donc documentée et codée suivant un formalisme bien établi. La réutilisation logicielle doit être une préoccupation permanente, non pas comme une contrainte, mais comme la conséquence même de la méthodologie de développement adoptée.

Pourquoi une telle démarche ? L'heure n'est plus aux développeurs isolés au fond de leur garage. Le développement de logiciels étant entré dans une ère industrielle, il est indispensable de se doter de méthodes et d'outils à même de satisfaire les besoins inhérents à ce changement de phase. Le travail en équipe implique ainsi des outils (gestion de versions), charte de codage, documentation (javadoc et diagrammes UML).

De plus, le monde de l'entreprise n'étant pas le moins du monde caritatif, il est bon d'envisager un projet informatique sous un jour financier en prenant en compte :

- le coût des bibliothèques utilisées (utiliser des solutions libres ne fait que favoriser la diminution du coût) ;
- la réduction des coûts de maintenance (idéal si elle est faite par d'autres) ;
- le facteur qualité, car dans un environnement où tout est quantifié, il est bon de se doter d'outils nous permettant de nous prémunir contre les mauvaises surprises (absence de commentaires dans le code, variables orthographiées sans logique, etc.).

Il faut en fait rappeler un principe universel (en informatique) étayé par des statistiques : moins on écrit de code, moins il y a de bogues. Une étude sur de nombreux projets (utilisant divers langages dans divers environnements) avait montré que l'on pouvait s'attendre à :

- un bogue majeur toutes les 1 000 lignes de code ;
- un bogue toutes les 100 lignes de code ;
- un bogue mineur toutes les 10 lignes de code.

Il est donc crucial de comprendre qu'il vaut mieux écrire peu de code testé et qualifié longuement, plutôt que livrer une grosse masse de code non testé. De là découle le souci permanent à travers cet ouvrage d'utiliser des solutions standards, des termes standards, voire des logiciels ou protocoles qui le sont aussi.

Pour ce faire, le lecteur va suivre un fil directeur sous forme d'une application simple (gestion des bookmarks d'une société) pour une société fictive : BlueWeb. Cette application nous permettra de mettre en œuvre du code et des outils s'inscrivant dans la ligne de conduite énoncée précédemment.

Évidemment, le format de l'ouvrage ne permet pas de présenter exhaustivement toutes les notions qui y sont évoquées (programmation par contrats, eXtreme Programming, EJB, servlets, etc.). Le lecteur pourra se reporter aux annexes bibliographiques ainsi qu'aux diverses annotations tout le long du livre. En résumé, il s'agit de proposer des solutions et surtout un état d'esprit pragmatiques permettant d'aborder plus facilement le développement avec les technologies Java.

Cet ouvrage essaie avant tout d'apporter des réponses concrètes à des problèmes récurrents, notamment ceux exprimés sur le Web.

▸ nntp://fr.comp.lang.java

À qui s'adresse cet ouvrage ?

Ce livre doit permettre à un jeune développeur (professionnel ou étudiant), possédant les rudiments nécessaires au développement en Java (compilation, bases du langage), de trouver des exemples concrets de mise en œuvre de différents thèmes tels que :

- les design patterns (tout au long de l'ouvrage) ;
- les architectures logicielles modernes (3 couches et extensions) dans le chapitre dédié à ce sujet (**chapitre 2**).

Des introductions à des techniques telles que l'utilisation de la servlet API (**chapitre 4**) ou des EJB (**chapitre 5**), enrichies de pointeurs vers des sites web et références bibliographiques, doivent permettre d'appréhender ces technologies.
Enfin, un lecteur expérimenté tel qu'un chef de projet ou un responsable qualité doit trouver à travers le **chapitre 7** des éléments lui permettant d'accroître sa maîtrise sur la qualité de son projet.
Tout le long de l'ouvrage, Ant, l'outil de gestion du processus de compilation du projet Apache, sera utilisé pour guider le lecteur vers des solutions garantissant la portabilité annoncée par Java. Cet outil propose des solutions simplifiant diverses tâches quotidiennes telles que :

- la compilation et la création de documentation ;
- le déploiement (tant côté client que serveur) ;
- l'obtention de métriques et indicateurs sur un projet.

Donc, quel que soit votre rôle dans un projet Java, vous trouverez matière à gagner du temps et de la qualité (automatisation de tâches répétitives).
De plus, des bibliothèques telles qu'Oro ou Rachel proposent de réelles alternatives Open Source à des solutions propriétaires. Ainsi, à travers la société fictive BlueWeb, on peut voir l'esquisse d'un nouveau mode de travail par lequel les équipes de développement obtiennent des résultats probants tout en contribuant à l'effort d'une communauté. Car tout problème signalé, toute documentation donnée ou toute publicité offerte à ces projets par leur utilisation ne peut que les renforcer et, par un effet boule de neige, rendre encore plus pérenne la solution adoptée.

Cet ouvrage espère pouvoir briser la timidité de certains décideurs vis-à-vis des solutions Open Source qui me semblent atteindre (pour certains projets du moins) une qualité difficile à trouver dans la plupart des outils du commerce.

À LA LOUPE **Les incontournables design patterns**

L'expression « design patterns », que nous ne traduirons pas, provient du titre d'un ouvrage d'architecture de M. Christopher Alexander *et al.*. Cet ouvrage s'efforce d'apporter une sémantique commune lors de la construction de cités pour différents problèmes récurrents en ce domaine. Cette idée a ensuite été reprise au début des années 1990 par Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides dans leur ouvrage *Design Patterns : catalogue de modèles de conception réutilisables*. Cet ouvrage est un élément absolument indispensable qui a sa place au panthéon des ouvrages en informatique. Cette idée d'essayer d'apporter un vocabulaire, une modélisation et une implémentation pour des problèmes courants respecte vraiment la philosophie que l'on va essayer de mettre en pratique au cours de cet ouvrage : la réutilisation. Il importe de bien comprendre qu'en essayant de proposer (et non d'imposer) un vocabulaire universel pour des problèmes qui le sont autant (s'assurer de l'unicité d'un objet par exemple pour le pattern Singleton), ce livre permet de régler les problèmes de communication apparaissant au cours d'un projet (au sein d'une équipe, avec un client, avec d'autres développeurs). UML permet d'aller un cran plus loin, en proposant un formalisme commun de représentation de vos modélisations. En adoptant une telle philosophie, votre travail sera plus simple et plus efficace, car votre code source comme vos modèles seront plus faciles à comprendre et donc plus efficaces au sens de la productivité pour votre entreprise. Il s'agira donc ici, non pas de réécrire ce qui l'a déjà été (avec infiniment plus de talent et d'imagination), mais de proposer quelques mises en pratique de ce qui doit devenir un de vos outils de travail.

Structure de l'ouvrage

L'ouvrage prend pour prétexte un projet virtuel afin de conduire le lecteur de bout en bout à travers toutes les couches de l'architecture proposée.

- L'introduction présente la société fictive, l'équipe et les spécifications du projet.
- Le **chapitre 1** décrit les traits de l'architecture retenue. Toutes les couches seront passées en revue. Ce chapitre présente les points clés nous poussant à adopter des architectures évoluées en lieu et place des architectures du type deux couches.
- Le **chapitre 2** donne une vue d'ensemble de l'environnement de développement (Ant et CVS) et introduit le projet Commons d'Apache (de quoi remplir vos besaces de développeurs Java).
- Le **chapitre 3** s'intéresse à la partie graphique (client) de l'application. Il y sera question entre autres du problème de la cohérence des saisies utilisateur. On détaillera le choix d'une nouvelle bibliothèque graphique, SWT (issue du projet Eclipse, un outil de développement).
- Le **chapitre 4** décrit la couche de présentation des données (utilisation de la servlet API). Il sera question de l'utilisation du XML comme format d'échange entre les couches, ainsi que d'un design pattern particulier : le pattern Commande.
- Le **chapitre 5** s'intéresse aux objets métier (EJB) et surtout à une façon originale de les coder permettant une meilleure productivité et un gain en portabilité. L'outil utilisé, XDoclet, nous donnera l'occasion d'introduire le concept des *doclets* Java.
- Le **chapitre 6** décrit comment mettre en place un outil de déploiement d'applications à travers un réseau via l'utilisation de Java Web Start. Évidemment, on s'intéressera aux problèmes classiques liés à ce sujet (gestion des versions, mise à jour) et, d'un point de vue plus pragmatique, aux coûts induits pour un chef de service.
- Le **chapitre 7** s'intéresse à des outils permettant de contrôler la qualité sur un projet Java. Ainsi, il s'agira de proposer des solutions concrètes permettant d'assurer la conformité de vos sources avec les conventions de nommage adoptées par votre équipe ou encore d'exposer un outil permettant de fournir des indicateurs sur le niveau de conception/réutilisation d'un paquetage Java (regroupement hiérarchique de classes au sein d'unités).
- Le **chapitre 8** proposera l'examen rétrospectif d'une partie du code de l'application BlueWeb.
- Le **chapitre 9** revient sur les technologies utilisées qui permettent de traiter superficiellement des frameworks web ainsi que des micro-conteneurs.

Bien entendu, on utilisera de façon permanente Ant et on éclairera dès que nécessaire les modèles de conception employés.

De l'utilisation de termes anglais

Nous avons souvent gardé les termes anglais, car il nous semble indispensable de donner au lecteur des références facilitant son apprentissage par la suite, dans un contexte où 95 % des documentations disponibles sont en langue anglaise. Par exemple, nous n'avons pas traduit le terme *design pattern* par *motif* ou *modèle de conception* afin de faciliter les recherches du lecteur intéressé par le sujet.

Remerciements

Que les personnes suivantes trouvent ici l'expression de tous mes remerciements.

Je remercie mon équipe de relecteurs (Ahmed Tachouaft, Nicolas Delsaux, Alban Peignier, François Xavier LeBail et Sébastien Dehud) pour sa sympathie et son expertise.

Merci à Laurent Fourmy (Softeam) pour son soutien, ainsi que pour le prêt d'une licence Objecteering m'ayant permis de réaliser les différents diagrammes UML de cet ouvrage. Merci aussi à Nicolas Bulteaux (Softeam) pour son aide en tant qu'auteur du modèle d'intégration de JUnit dans Objecteering.

Enfin merci à ceux qui m'ont côtoyé durant la difficile gestation de ce premier enfant (toute l'équipe Eyrolles et mes proches). Mes pensées vont également à Martine qui aurait été associée à cette nouvelle édition si la vie n'en avait décidé autrement…

Enfin une petite mention spéciale pour Stéphane Bailliez : bon voyage l'ours…

Jérôme Molière

jerome@javaxpert.com

Table des matières

BlueWeb

L'équipe de développement

Bob
chef de projet

Michel
responsable qualité

Yann
chargé de la
partie cliente

Pat
chargé des
bases de données

Steve
chargé du déploiement
et packaging

Introduction à l'étude de cas et à J2EE

En guise d'introduction, nous présenterons le contexte de notre étude de cas : l'entreprise virtuelle BlueWeb, son projet et son équipe de développement. Ce chapitre sera aussi prétexte à une justification des technologies J2EE au regard de technologies plus anciennes.

SOMMAIRE

MOTS-CLÉS

- Projet pilote
- Gestionnaire de signets

BlueWeb : l'entreprise

Ce chapitre décrit une entreprise n'ayant d'existence que le long de cet ouvrage…

BlueWeb est une entreprise spécialisée dans la conception de sites Internet (intranet) clés en main pour des clients importants (banques, industries et administrations). Pionnière du Web en France, elle doit dorénavant opérer un tournant clé de son histoire et fournir des solutions complexes (techniquement) en vue de réaliser des portails en intranet/extranet fédérant diverses applications (solutions proches de l'EAI).

EAI (Enterprise Application Integration)

Nom désignant les techniques qui permettent d'intégrer différentes applications développées sans connaissance les unes des autres. Il s'agit donc d'utiliser des passerelles standards, telles que XML ou le protocole HTTP, pour transformer en un tout cohérent une série d'applications hétérogènes.

L'entreprise se prépare donc à faire un véritable saut technologique en abandonnant les compétences maintenant obsolètes qui avaient fait sa renommée :

- HTML ;
- scripts JavaScript ;
- scripts CGI, peu réutilisables et peu sûrs.

JSP (Java Server Pages)

Technologie permettant de définir des pages web créant du contenu dynamique. Cette API fait partie de celles regroupées au sein des spécifications J2EE.

HTML et JavaScript ne sont pas obsolètes dans le sens où toutes les pages web du monde sont codées en HTML et souvent enrichies par des scripts JavaScript, mais ces seules technologies ne peuvent plus répondre aux besoins des applications d'aujourd'hui. C'est pourquoi, via des technologies comme les JSP/servlets ou les ASP dans le monde Microsoft, les pages HTML sont bien plus souvent dynamiques (produites dynamiquement par des programmes) que statiques (réalisées directement avec un éditeur de texte ou un outil dédié, comme Dreamweaver ou Quanta+).

Servlet

Composant web géré par un container et permettant la génération de contenu dynamique. Le chapitre 4 présente cette notion et signale des sites web ou des ouvrages de référence sur ce sujet. Cette API fait également partie des spécifications J2EE.

L'exploitation de formulaires et l'envoi de courriers électroniques font partie de ces tâches autrefois réservées aux scripts CGI (codés en C ou en Perl par exemple) mais qui, pour des raisons de montée en charge, de réutilisation et aussi de sécurité, font maintenant les beaux jours des JSP/servlets. En fait, J2EE (voir encadré) apporte aux entreprises les avantages liés à toute technologie objet par rapport à des langages procéduraux comme le C, tels que la réutilisation et la facilité de maintenance. De plus, de par sa conception, Java est un langage sûr et portable. Ceci résout un des problèmes liés à l'utilisation des CGI : les failles de sécurité (par exemple les « buffer overflow », résultats d'une mauvaise conception, qui permettent à des programmeurs rusés d'introduire et de lancer du code indésirable sur le serveur).

EJB (Enterprise JavaBeans)

Nom d'une spécification de Sun (actuellement en version 2.1) qui vise à définir une bibliothèque permettant d'obtenir des composants métier côté serveur. Ces composants sont déployés au sein d'un conteneur pouvant leur fournir divers services de haut niveau comme la prise en charge de la persistance ou celle des transactions.

B.A.-BA **Qu'est-ce que J2EE ?**

J2EE (Java 2 Enterprise Edition) regroupe à la fois un ensemble de bibliothèques (servlets/EJB/JSP) et une plate-forme logicielle. C'est une version de la plate-forme Java spécialement étudiée pour les problèmes des applications d'entreprise. Elle fournit de nombreuses solutions au niveau de la gestion des problèmes récurrents dans ce domaine : sécurité (API JAAS), prise en charge des transactions (API JTA et gestion déclarative dans les EJB). JDBC fait aussi partie de ces technologies. Actuellement, la dernière version des spécifications disponible est la 1.4.

Cependant, avant de migrer toutes ses équipes de développement vers J2EE, BlueWeb veut utiliser ces technologies sur un projet test afin d'en retirer une expérience capitalisable pour son futur. Pour cela, quelques développeurs de l'équipe Recherche & Développement ont été choisis.

Projet pilote

Ce type de projet est dénommé projet pilote dans de nombreuses entreprises. Il fait partie de l'arsenal des managers qui doivent faire face à un changement radical de technologie…

L'équipe de développement

L'équipe de développement choisie est composée de 5 personnes :

- Bob, le chef de projet, est un programmeur C émérite et, via son expérience du Perl, il a acquis une expertise dans le maniement des expressions régulières. Son bon sens et son pragmatisme sont les garde-fous de BlueWeb depuis des années. Le monde de J2EE, et en particulier la foule de projets issus de l'Open Source, le fascinent…
- Yann est un ancien stagiaire fraîchement sorti de l'école. Il est tombé dans l'objet quand il était tout petit et « parle » très bien UML et Java. Il va être en charge de la partie cliente et sera associé à Steve, qui devra assurer le déploiement/packaging de l'application.
- Pat est un spécialiste des bases de données. Programmeur C et DBA certifié Oracle et Sybase, il a le profil type du codeur côté serveur.
- Comme de coutume pour les projets au sein de BlueWeb, l'équipe de développement devra soumettre code et documentation à l'approbation du responsable qualité de la société : Michel, homme rigoureux entre tous.

DBA (DataBase Administrator)

Administrateur de bases de données : nom donné aux personnes qui créent et maintiennent de grosses bases de données.

À LA LOUPE **Java et l'Open Source**

Java a connu un succès très rapide, et ce au moment même où Linux introduisait la notion d'Open Source auprès du grand public. Cette synergie a alors donné le jour à de nombreux projets communautaires (au sens où un ensemble de développeurs issus des quatre coins de notre planète participent à un même projet dans le seul but d'améliorer tel ou tel aspect de leur travail/passion). Beaucoup de projets sont nés, beaucoup sont morts, mais la dynamique n'est pas retombée et aujourd'hui, certains d'entre eux constituent de réelles forces de proposition au sein de la communauté d'utilisateurs Java. Ainsi, Exolab accueille différents projets (Castor par exemple), de même qu'Apache accueille Ant, Lucene, Tomcat, Oro…. Il faut préciser que l'Open Source n'est en rien une spécificité de Java, puisque les premiers projets de ce type sont Linux, Apache, Samba, Sendmail…

Technologies et méthodologie

Ce projet doit être l'occasion de mettre en œuvre pour la première fois les technologies J2EE (servlets et EJB) et d'adopter une approche méthodologique fortement teintée par les principes majeurs de l'eXtreme Programming.

MÉTHODE L'eXtreme Programming

L'eXtreme Programming est un mouvement récent (1998) très original, puisqu'il s'agit à la fois d'une méthodologie et d'outils concrets visant à prendre enfin en compte dans un projet un aspect oublié par les autres mentors de l'objet : l'homme. Avec certains préceptes et certaines pratiques courantes, ce mouvement tâche d'améliorer la communication et la qualité du code.
Il faut quand même préciser qu'à la base, les pratiques décrites par les gourous de l'eXtreme Programming sont très bien adaptées à de petites équipes, pour des projets où les développeurs sont en contact avec le client final (pas toujours vrai).
Vous trouverez par la suite un peu plus d'informations sur ce mouvement très intéressant et l'on ne saurait que trop chaudement recommander la lecture de l'ouvrage suivant :

📖 J.-L. Bénard, L. Bossavit, R. Médina, D. Williams.- *Gestion de projet eXtreme Programming*, Eyrolles 2002.

Les notions les plus largement appréciées au sein de l'équipe sont le travail en binôme (*pair programming*) et l'importance accordée aux tests unitaires

Certains diagrammes UML seront utilisés pour fixer le vocabulaire commun, délimiter avec précision le contour fonctionnel de l'application et faciliter le codage.

À ce titre, il va donc être question de réaliser une application facilement distribuable.

Pour cela, le **chapitre 6** de cet ouvrage présente l'optique de BlueWeb, qui a déjà eu ce type de préoccupations et a donc une expérience qu'elle veut utiliser afin de se prémunir contre les problèmes fréquemment rencontrés. Bien évidemment, il s'agira de minimiser les interventions humaines (trop onéreuses) et d'utiliser le Web comme média de communication pour les mises à jour.

L'interface graphique sera développée en utilisant une bibliothèque assez récente mais qui fait déjà beaucoup de bruit : SWT. Il s'agit là en fait de la base du projet Eclipse, projet initié par IBM et dont le développement est assuré par un consortium.

L'idée d'utiliser cette bibliothèque plutôt que la très documentée Swing (devenue la bibliothèque standard pour les interfaces graphiques en Java) provient du fait que BlueWeb a la preuve (avec Eclipse) que l'on peut faire un très bon produit avec cette bibliothèque.

Il s'agit donc de mesurer, sur un projet concret, l'impact sur une planification de projet d'une telle bibliothèque en matière de temps d'adaptation, d'obtention de réponses et d'informations. Il s'agira aussi de mesurer la stabilité de cette bibliothèque. Toutes ces questions ne peuvent trouver réponse que par le biais d'un

réel projet et, étant donné l'importance de la question, il vaut mieux utiliser un projet pilote qu'un réel projet client pour mettre en œuvre cette bibliothèque. Pour BlueWeb, il est évident que l'utilisation de Swing sur un projet aussi simple aurait été une solution de facilité, car l'abondance de la documentation (livres/articles) aurait permis de surmonter les rares difficultés posées par l'interface graphique à réaliser. Cependant, cela n'aurait nullement permis de se forger une opinion sur les qualités/défauts de SWT et donc n'aurait contribué qu'à repousser un problème qui de toute évidence se posera bientôt pour BlueWeb.

Avec une optique similaire, BlueWeb décide de mettre en œuvre les EJB. Il s'agit de voir comment les mettre en œuvre et de pouvoir capitaliser des expériences concrètes d'utilisation de ces composants. D'autres solutions permettraient de gérer la persistance des données :

- l'API JDBC (Java DataBase Connectivity) ou via iBatis ;
- certains produits comme Hibernate, Cayenne JDO, KodoJDO, Speedo ou encore TopLink ;
- la norme JDO (Java Data Objects) de Sun.

ATTENTION JDO et Castor JDO

Il n'y a pas d'erreur dans la liste ci-dessus (quelques omissions) : par un caprice du sort, il se trouve qu'un produit (castor JDO) porte un nom aujourd'hui proche de celui d'une norme de Sun. En fait, le produit et la norme visent tout deux à assurer la persistance d'objets Java, mais le produit Castor JDO n'implémente pas du tout la norme de Sun !

L'utilisation des EJB un jour ou l'autre semble évidente pour BlueWeb et, de nouveau, ce projet pilote semble être l'occasion parfaite. En choisissant cette solution, on pourra compter sur des conclusions étayant le savoir-faire de BlueWeb sur des questions comme :

- la performance des EJB (en mode CMP) ;
- la facilité et le temps de codage ;
- La portabilité réelle.

Enfin, le choix d'adopter une couche de présentation réalisée par des servlets Java est dû à l'importance de HTTP dans l'informatique d'aujourd'hui.

Plus anecdotiquement, le code client se devra de mettre en œuvre le produit JUnit (support des tests unitaires), et ce afin de mesurer concrètement les bénéfices d'une telle approche, ainsi que l'impact sur la gestion d'un projet. Pourquoi se contenter d'une portion seule du code ? Avec ce projet, BlueWeb ne prétend pas chercher à atteindre la réutilisation mais juste à tester grandeur nature différentes solutions ; il en découle qu'il n'est nullement nécessaire d'imposer l'utilisation de ce framework à toute l'équipe si d'aventure l'expérience s'avérait négative avec ce produit.

OUTIL iBatis

iBatis se trouve dans l'incubator Apache.

- http://incubator.apache.org/projects/ibatis.html

OUTILS Hibernate, Cayenne JDO, KodoJDO, Speedo et TopLink

- www.hibernate.org
- www.objectstyle.org/cayenne/
- www.solarmetric.com
- http://speedo.objectweb.org/
- www.oracle.com

B.A.-BA Framework

Ce mot désigne un ensemble de classes destiné à réaliser une fonction ou rendre un service, et ce de manière réutilisable. La distinction avec une API est un peu subtile, mais on peut caricaturer en constatant qu'un framework est plus qu'une API. En citant la définition donnée dans l'ouvrage *Design Patterns* : « Un framework est un ensemble de classes qui coopèrent et permettent des conceptions réutilisables dans des catégories spécifiques de logiciels. ».

L'application

Description des fonctionnalités

L'application doit permettre la saisie, la mise à jour et l'interrogation de signets. Ainsi, il sera possible de factoriser la recherche et l'archivage des adresses fréquemment utilisées dans l'entreprise. En effet, jusqu'à présent, tous les employés de la société utilisaient les fonctionnalités de leur navigateur web (Netscape ou IE), ce qui avait l'inconvénient de multiplier l'espace disque (secondaire) mais aussi de faire perdre du temps et du savoir, car de nombreux sites peuvent avoir de l'intérêt pour de nombreux employés. Ainsi, pour les développeurs, des portails web comme celui d'IBM, sont des sites qu'ils ajoutent tous dans leurs listes de favoris.

Un annuaire d'entreprise du type LDAP permettrait ce type de fonctionnalités, mais le coût d'administration d'un tel applicatif lui ferait perdre tout intérêt et ne résoudrait pas pour BlueWeb la nécessité d'apprendre à manipuler les nouvelles technologies. En effet, LDAP est une norme permettant de situer des informations dans un entrepôt de données. Un exemple typique d'utilisation de LDAP est un annuaire d'entreprises pour lequel on dispose de clés de recherche (des noms) et qui permet d'obtenir diverses informations (téléphone, fonction dans l'entreprise, salaire, etc.). Notre application pourrait naturellement être réduite à peu de choses en utilisant une implémentation de cette norme. Mais il ne faut voir en l'application qu'un prétexte à l'acquisition de compétences et non un simple applicatif.

La première version du logiciel ne prendra pas en charge des notions comme l'internationalisation, pour gagner du temps et de la simplicité. En effet, cette contrainte ne répond pas à un besoin pressant des clients (utilisateurs en interne) et n'apporte rien techniquement, mais elle demande du temps et de l'énergie…

Analyse des données

Étant donné l'extrême simplicité du sujet, on peut se contenter d'une petite étude réduite à quelques lignes et quelques diagrammes pour comprendre la partie métier traitée par cette application. Il s'agit en fait dans notre application de simplement entreposer, saisir et rechercher des fiches désignant un site web.

Pour BlueWeb (comme pour la plupart des autres sociétés), un *signet* est simplement le nom donné à l'ensemble suivant de données :

- libellé ;
- URL (adresse HTTP ou FTP) ;
- description : commentaire sur le site.

Évidemment, étant donné le nombre de signets par utilisateur (disons 400 en moyenne), la base de données globale peut atteindre les 15 000 signets, ce qui

Signets

*Bookmark*s en anglais : correspond aux favoris sous Internet Explorer ou marque-pages sous Mozilla Firefox. Ces petites fiches permettent de se déplacer rapidement vers certains sites web et de les classer par thèmes ou centres d'intérêts.

B.A.-BA LDAP (Lightweight Directory Access Protocol)

LDAP est un standard et non un produit. Différentes implémentations sont disponibles, comme OpenLDAP qui est un produit libre, ou encore le serveur fourni dans Exchange de Microsoft, Microsoft Active Directory.

B.A.-BA Internationalisation

L'internationalisation, ou *i18n* en abrégé pour les américains, désigne le modèle de développement, l'API et les méthodes de conception nécessaires à la prise en charge de plusieurs langues dans une application. Parmi les différents concepts en présence, on peut citer pêle-mêle :

- le modèle de conception (design pattern) Adapter ;
- l'utilisation de clés en lieu et place de tous les labels et autres messages d'erreur ;
- les classes `ResourceBundle`, `Locale` et `SimpleDateFormat`, qui peuvent servir de briques de base lors de l'élaboration de schémas ou frameworks maison prenant en charge de telles contraintes.

implique fatalement de classer par dossiers que nous appellerons *thèmes*, un thème étant alors la collection des données suivantes :

- un libellé ;
- une description (commentaire) ;
- une liste (peut-être vide) d'enfants (thèmes et/ou signets).

Cela nous amène à dégager une modélisation du type :

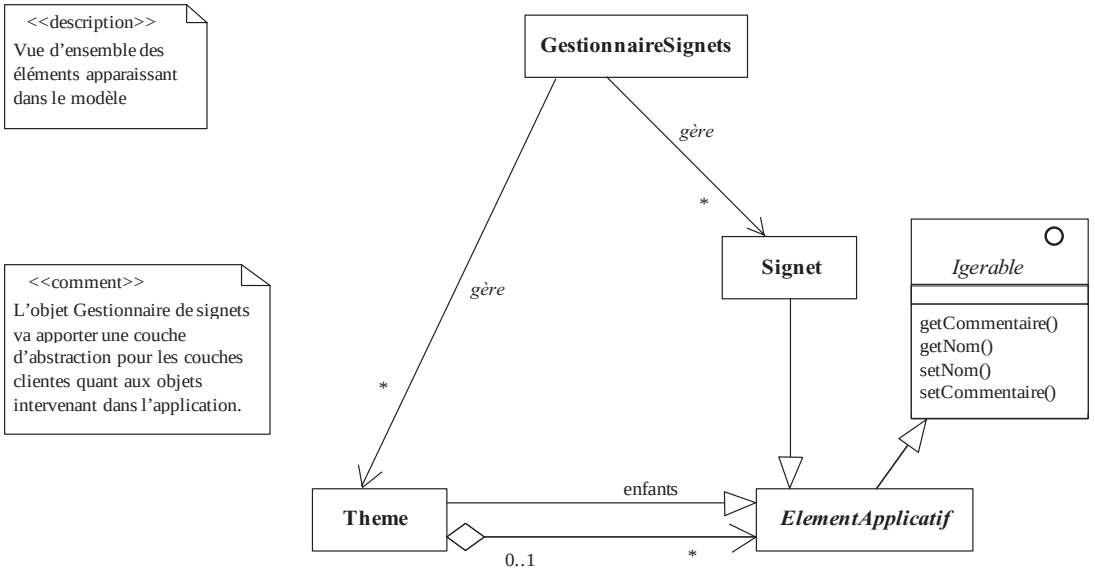


Figure 1 Modélisation de notre application

- `ElementApplicatif` est une classe abstraite implémentant l'interface `IGerable`, qui permet de regrouper les méthodes communes à des objets gérables dans l'applicatif.
- Les classes `Signet` et `Theme` sont des classes filles de `ElementApplicatif`.

Cette modélisation nous conduit directement à la notion de modèle composite (design pattern Composite), dans la mesure où le diagramme fait apparaître qu'un `Theme` peut accepter comme enfant tout type d'`ElementApplicatif`, c'est-à-dire un `Theme` ou un `Signet`. On peut noter à titre de remarque, pour les lecteurs non familiers avec la norme UML, que ce schéma adopte une notation officielle. Ainsi, le petit rectangle de commentaire balisé `<<<comment>>>` utilise ce que l'on appelle une `tagged-value` en UML. Ici, `comment` est donc une `tagged-value` particulière, qui permet de préciser quel type de commentaire on veut utiliser De la même façon, le rectangle situé au-dessus utilise une autre `tagged-value`, qui a la valeur `description`. Pour de plus amples informations sur UML, on ne peut que recommander les excellents ouvrages de Pascal Roques et en particulier, dans la même collection que notre livre, le cahier du programmeur *UML – Modéliser un site e-commerce*.

Spécifications techniques

L'application ne requiert pas de gestion de différents profils utilisateurs. Autrement dit, il n'y aura pas de gestion de privilèges (droits d'accès à certaines données ou fonctionnalités) et donc pas d'écran d'invite permettant à l'utilisateur d'entrer dans l'applicatif après saisie d'un nom et d'un mot de passe.

Une architecture du type 3-couches étendue doit permettre :

- une vaste possibilité de déploiements (sur une machine ou plusieurs) ;
- un déploiement via le Web pour ne pas avoir à gérer des mises à jour via CD-Rom et intervention humaine ;
- le fonctionnement en mode client/serveur sans nécessité d'installer de protocoles propriétaires sur le poste client ;
- une interface cliente riche (donc non réduite à de simples pages HTML) ;
- l'utilisation d'un serveur d'applications JBoss + Tomcat, afin de se familiariser avec les technologies définies dans la norme J2EE ; cet outil a le mérite d'être Open Source, donc simple à obtenir (librement accessible sur Internet sans investir financièrement dans des licences logicielles) et à installer (pas de clé nécessitant un contact avec un service commercial) ;
- l'utilisation d'un outil de `make`, de manière à pouvoir maîtriser la compilation et les étapes suivantes du cycle de vie du produit, et ce indépendamment de tout outil de développement (JBuilder, Eclipse ou tout autre IDE) ;
- l'utilisation d'une couche d'abstraction de la base de données, qui doit permettre l'utilisation d'une vaste gamme de produits du marché (Sybase, Oracle, SQL Server ou PostgresSQL).

OUTILS **JBoss et Tomcat**

JBoss est un serveur d'applications Open Source qui est maintenant à l'âge de la maturité, indiscutablement de qualité professionnelle. C'est un conteneur EJB conforme aux spécifications de la version 2.1 de cette norme (la dernière publiée) qui propose un *bundle* (en marketing signifie deux produits livrés ensemble) avec le *servlet-engine* le plus populaire : Tomcat.
Tomcat est lui aussi un projet très populaire et permet d'obtenir un *servlet-engine* conforme aux dernières spécifications (2.4). Ces deux produits sont de véritables références et leur succès est attesté par le nombre de récompenses qu'ils ont obtenues.

- http://www.jboss.org
- http://jakarta.apache.org/tomcat

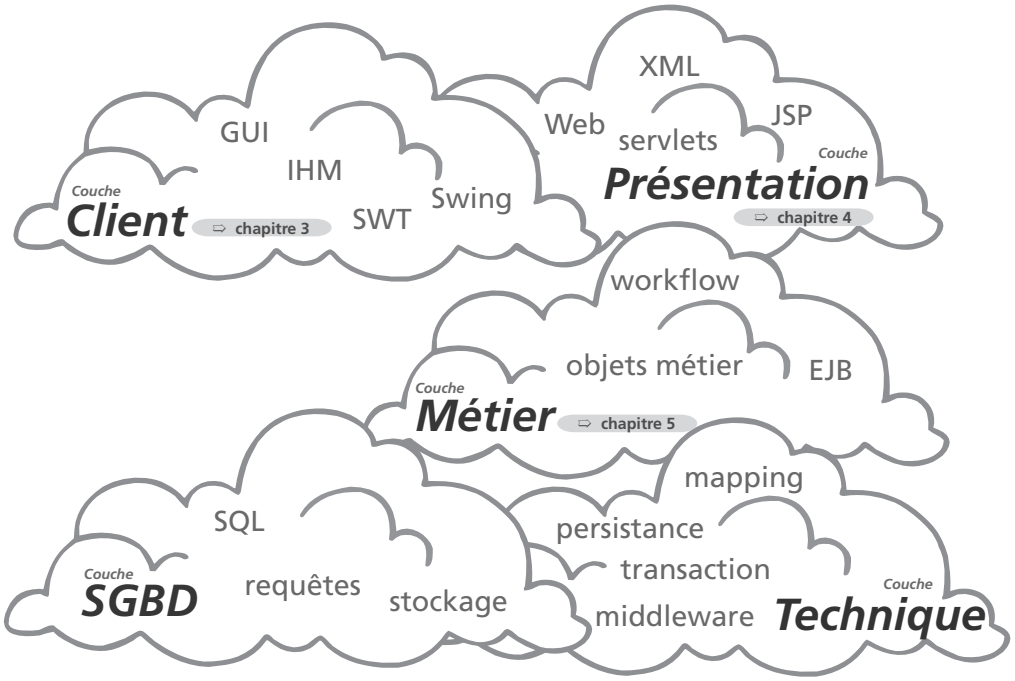
En résumé…

Pour planter un décor, nous avons constitué une équipe virtuelle dans une société qui l'est tout autant, mais en introduisant des concepts et des pratiques qui ne le sont pas. Ainsi la notion de projet pilote est-elle une réalité. Le vocabulaire important a également été expliqué. Nous avons aussi présenté succinctement quelques-uns des produits que nous rencontrerons tout au long de l'ouvrage.

La réussite d'un projet pilote tient dans le dosage subtil de différents ingrédients – un peu comme dans l'art du cocktail :

- un cahier des charges simple ;
- une définition claire des obstacles en termes d'acquisition de connaissances (difficultés techniques et outils) ;
- une bonne définition du planning ;
- une équipe cohérente et son adéquation avec les objectifs visés.

chapitre 1


Couche
Client
⇨ chapitre 3
GUI
IHM
Swing
SWT
XML
Web
servlets
JSP
Couche
Présentation
⇨ chapitre 4
workflow
objets métier
EJB
Couche
Métier
⇨ chapitre 5
mapping
SQL
persistance
transaction
Couche
SGBD
requêtes
stockage
middleware
Couche
Technique

Une architecture à 5 couches pour BlueWeb

Les choix techniques et leur justification doivent non seulement obéir à des contraintes techniques, mais aussi bien souvent à des arguments visant à rassurer les décideurs quant à leur investissement, comme nous le verrons dans le cadre de notre projet fictif BlueWeb.

SOMMAIRE

MOTS-CLÉS

- Open Source
- Choix techniques
- Pérennité
- Qualité
- Déploiement

Un modèle à 5 couches pour le projet de gestion des signets

L'application de gestion des signets étant un projet pilote pour BlueWeb, l'équipe en charge de ce projet se doit d'adopter une façon de travailler, de coder et de documenter le projet qui deviendra un exemple pour les développements futurs. Ceci dit, cette application doit aussi servir à défricher les nouveaux espaces ouverts par ce changement de technologie, l'architecture globale du projet entend bien s'affranchir des sérieuses limitations rencontrées précédemment.

Ce type de modèles est issu d'une extension naturelle des modèles dits à 3 couches. Avant de le présenter, revenons un peu en arrière. L'apparition des technologies objet et l'importance croissante du Web ont vu la fin des environnements articulés autour des modèles de développement dits à 2 couches. C'est-à-dire la fin des interfaces clientes assurant à la fois l'affichage des informations, les interactions avec l'utilisateur mais aussi l'application de la logique métier et les appels à la base de données sous-jacente. Trop chères en coût de maintenance et déploiement, peu adaptées à la montée en charge, ces architectures monolithiques ont cédé la place aux modèles dits à 3 couches. Commençons par revenir sur le modèle à 3 couches en le comparant au modèle à 2 couches.

Le modèle à 5 couches, objet de ce chapitre, est donc une variante du modèle à 3 couches séparant encore plus strictement les responsabilités :

- couche cliente limitée à l'affichage/interactions utilisateur ;
- couche de présentation des données : partie serveur rendant possible un accès à tout type de partie cliente (client léger Web, application C++ ou encore client riche Swing) via un protocole basique (HTTP) aux services métier (ajouter un signet ou créditer un compte) ;

LES BASES **Modèles à 2 et 3 couches**

La figure ci-contre permet de schématiser les différences entre ces deux types d'architectures logicielles.
Elle illustre la manière dont les responsabilités se trouvent éclatées et comment la montée en charge peut être envisagée (une seule machine a besoin d'être puissante).
On peut alors envisager une véritable réutilisation des composants :

- pour la partie cliente, des composants graphiques de haut niveau ;
- pour la partie serveur, des composants « métier ».

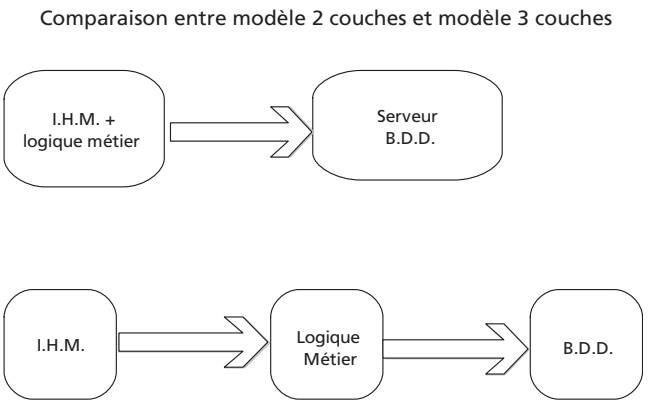


Figure 1–1 Modèles à 2 et 3 couches

- services métier : composants réutilisables rendant des services strictement identifiés.
- services techniques : composants permettant de réaliser différentes tâches du type persistance des données, gestion des transactions etc. ;
- stockage des données : c'est le rôle de la base de données.

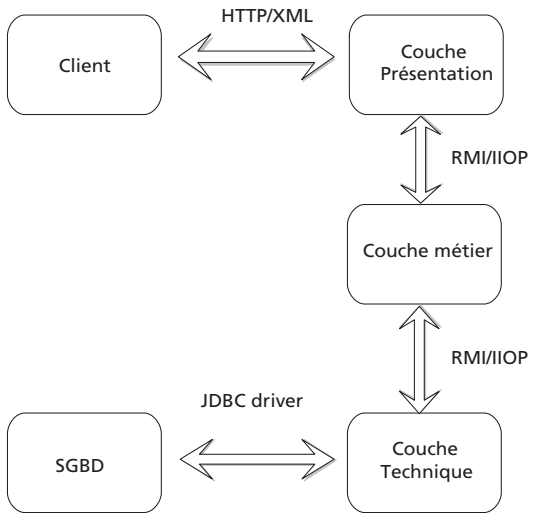


Figure 1–2 Modèle à 5 couches

Ce type d'architecture permet donc d'atteindre différents objectifs obsédant les DSI (Direction du système d'information) :

- Déploiement simple, mise à jour aisée donc les coûts sont diminués (TCO).
- Factorisation de la logique de l'entreprise. En effet, comme on le voit sur la figure 1-1, la logique embarquée dans une application adoptant une architecture à 2 couches est mélangée avec le code de gestion des événements utilisateur et celui d'affichage des éléments graphiques. Bref, dans un tel contexte, il est difficile de pouvoir prétendre réutiliser ce que l'on appelle la logique métier.
- Délégation des parties ardues techniques (mapping objet/relationnel, gestion des transactions) à des composants spécialement développés par des experts dans le domaine.

En conséquence les architectures à 5 couches sont en parfaite adéquation avec les nouveaux environnements d'exécution d'applications, à savoir les serveurs d'applications. Les vendeurs de ce type de produits argumentent sur les thèmes récurrents de la montée en charge, de la facilité de déploiement et de la qualité des composants techniques.

TCO

Acronyme anglais (Total Cost Ownership) signifiant coût total d'acquisition, la maintenance représente souvent un coût nettement supérieur à celui de la phase de développement d'un logiciel. Réduire le coût de la phase de maintenance/déploiement implique alors une baisse de cet indicateur. Ceci est donc une très bonne nouvelle pour les DSI.

ATTENTION **Réutilisation**

Même s'il est difficile de rendre le code métier réutilisable dans un environnement du type Delphi ou Visual Basic, il n'est absolument pas impossible d'y arriver. Il faut beaucoup de rigueur car cela va un peu à l'encontre de la philosophie de ces outils très bien adaptés à ce que l'on appelle le RAD (Rapid Application Development). De même, l'utilisation d'une belle architecture 3 ou 5 couches n'implique en rien le caractère systématique de la réutilisation des composants mais il est plus facile d'y arriver… RAD est un procédé de développement visant à opérer de manière itérative, par prototypage, en précisant à chaque itération. La philosophie de ce type de développement est très axée sur le retour des utilisateurs par rapport aux interfaces graphiques (réalisées dans la phase précédente) et tente de limiter d'une phase à l'autre la réutilisation de code.

Mapping

Ce terme anglais est trop délicat à traduire pour s'y risquer. Il signifie l'étape de transformation d'un objet Java par exemple en une entrée (ou plusieurs) dans la base de données et vice versa. Cette étape est imposée par l'écrasante supériorité des bases relationnelles sur le marché des bases de données.

L'architecture proposée pour notre application exemple, la gestion des signets pour la BlueWeb compagnie, est une architecture reposant sur un modèle à 5 couches déployée au sein d'un serveur d'applications compatible avec les spécifications J2EE. Cette application étant un projet pilote pour la société, on cherchera à respecter au plus haut point les spécifications de Sun et ce, afin de jauger l'adéquation des normes avec les besoins ressentis lors de projets sur le terrain. Les spécifications J2EE sont accessibles sur le site de Sun à l'adresse :

- http://java.sun.com/j2ee/1.4/docs/tutorial/doc/

Les sections suivantes vont examiner l'une après l'autre les différentes couches logicielles en présentant la solution retenue par BlueWeb pour l'implémentation au niveau de l'application.

B.A.-BA **Serveurs d'applications**

Cette expression en vogue depuis quelques années est très générique, trop peut-être, car elle signifie : application publiant des services afin de les rendre utilisables via un réseau. Contrairement à un serveur FTP ou à un serveur de fichiers (Samba sous Unix ou NFS), un serveur d'applications ne se contente donc pas de diffuser de l'information exploitée par le poste client puisqu'il « travaille ». Il n'est pas dédié à un service particulier et exécute des programmes à la demande de l'application cliente.

Un serveur d'applications minimal pourrait être une combinaison Apache + PHP puisqu'il permettrait à des utilisateurs d'accéder à des services (boutique en ligne comme dans l'exemple du cahier du programmeur UML de Pascal Roques). Ces termes vont jusqu'à couvrir des implémentations beaucoup plus riches telles que de véritables implémentations de la norme J2EE, comme Weblogic Server de BEA, WebSphere d'IBM etc.

📖 P.Roques, *Cahier du programmeur UML*, Eyrolles, 2002

Couche de présentation des données : servlets

Cette technologie simple et robuste, nous permettra d'exposer nos services métier via une couche HTTP très simple d'accès (en utilisant un protocole standard, on s'assure de ne pas nécessiter des installations difficiles sur les PC de bureau) sous forme de XML, c'est-à-dire présenter sous un format XML des objets manipulés côté serveur. Les servlets ont donc un rôle de présentation des données (transformation en XML) ainsi qu'une obligation de découplage entre le client et la partie serveur, ce découplage étant assuré par le protocole « standard » utilisé pour atteindre une servlet : HTTP. Cette couche nous permettrait aussi de gérer l'authentification des utilisateurs et de conserver leurs paramètres personnels en mémoire par utilisation de sessions HTTP.

Cette couche a donc pour vocation de rendre les installations des postes clients moins complexes et totalement indépendantes de notre application. Ceci évite l'utilisation de protocoles (IPX/SPX), RMI-IIOP) qui entraîneraient des déplacements de techniciens sur les postes clients et donc des augmentations des coûts d'administration. Dans un contexte d'entreprise, on peut espérer qu'au moins une couche TCP/IP soit installée sur le poste client. Sans elle, le PC ne peut pas communiquer avec un réseau standard, et ne permet donc pas l'accès à Internet, la messagerie standard etc. Requérir simplement ce protocole revient alors à ne rien installer du tout dans la plupart des cas, car bien rares sont les entreprises sans réseau interne, messagerie ni partage de fichiers. L'ouverture d'une application sur le monde extérieur est une absolue nécessité et qui peut se targuer de savoir de quoi demain sera fait ? C'est pour cela qu'adopter une couche de présentation des données de ce type, permet éventuellement de conserver notre logique serveur tout en optant pour un autre type de client (pages HTML, client Delphi ou VB, etc.).

RÉFÉRENCE **Apprentissage des servlets**

De nombreux sites et ouvrages vous permettront de vous familiariser avec les servlets mais le guide ou tutorial suivant est particulièrement digne d'intérêt même s'il date un peu. C'est ce site qui m'a permis de comprendre cette technologie il y a quelques années de cela.

- http://www.novocode.com/doc/servlet-essentials/

Objets métier – couche de persistance : EJB

Utiliser des objets codés en Java (ou dans tout autre langage objet) pour représenter des données stockées dans une base relationnelle fait fatalement apparaître le besoin de transformer ces objets afin de les stocker ou de les lire.

L'application ne nécessite pas tout l'éventail des technologies proposées par les spécifications des EJB mais BlueWeb désire utiliser cette technologie sur un petit projet avant de la mettre en œuvre sur des projets plus ambitieux. On utilisera donc des beans de type `session` (stateless) et des beans de type `entities` (CMP).

L'usage des EJB n'est pas indispensable pour notre application mais permet de se familiariser avec la technologie, de poser des méthodes de développement et aussi de les tester grandeur nature. D'autres solutions seraient envisageables dans ce contexte précis même si elles présentent toutes différents inconvénients :

- problèmes de portabilité du code produit ;
- problèmes lors de la montée en charge ;
- pas de syntaxe déclarative pour la gestion des transactions.

Le projet doit permettre de tester différents conteneurs EJB et ne peut donc reposer sur les spécificités de l'un ou l'autre. Le codage des EJB sur ce projet devra donc tenter de se préserver d'utiliser toute spécificité de tel ou tel produit. Attention ceci n'est pas innocent car, malgré la norme, tous les vendeurs de produits proposent des extensions à celle-ci, rendant du même coup votre application J2EE liée à un seul produit. Il faut donc trouver un moyen de prévenir ce type de travers et ce moyen existe via l'utilisation d'outils de génération de code source et de fichiers de déploiement.

ALTERNATIVE **MVC2 et JDBC**

Une solution souvent utilisée et préservant l'élégance et donc la maintenance du code est l'utilisation d'un framework MVC2 (pour Modèle vue contrôleur). Ce type de framework (Struts, Barracuda) permet de créer des applications destinées au Web (HTML ou XML) tout en séparant strictement les responsabilités entre les différents acteurs (servlets et JSP). JDBC est le nom de l'API de Sun permettant l'accès aux bases de données. JDBC est simple d'emploi, assez puissante et très souple. Cette bibliothèque est souvent le dernier recours vous permettant d'assurer la persistance d'un bean (la persistance n'est plus alors assurée par le conteneur mais par le développeur, elle est donc appelée BMP par opposition aux beans entités du type CMP).

PRÉCISION **Spécifications EJB**

Les EJB proposent d'autres types de beans (clients de files de messages) mais les choix adoptés ici sont tout de même ceux adoptés dans la majorité des projets à base d'EJB. En effet, les sessions beans statefull sont trop coûteux en ressources (avantageusement remplacés par des sessions HTTP) et la persistance des beans est suffisamment complexe pour rebuter la majeure partie des développeurs préférant laisser cette tâche au conteneur.

Les différents types d'EJB

Les EJB proposent différents types de composants, assurant chacun des rôles différents. On peut citer les entités (*entities beans*) dont le rôle est de décrire un enregistrement physique dans une base de données ou encore les composants de session (*session beans*) dont le rôle est de fournir des services de haut niveau. On pourrait par exemple imaginer un bean session `GestionnaireCompte` proposant un service `crediteCompte`(). Cette méthode manipulerait différents objets physiques de la base (donc différentes entités). Les beans sessions peuvent avoir un état ou non. Ils seront alors dits statefull ou stateless. Les beans entités peuvent être codés à la main (en JDBC) pour assurer la persistance, ils sont alors dits BMP (Bean Managed Persistence), ou cette persistence peut-être confiée au conteneur en réalisant ainsi des beans CMP (Container Managed Persistence). Enfin, un autre type de composant est dorénavant disponible (depuis la version 2.0 des spécifications EJB) : les clients de files de messages. Le but de ces composants est alors de pouvoir s'abonner à une file de messages et de recevoir de manière asynchrone les notifications de nouveaux messages disponibles dans la queue. Une présentation un peu moins rapide des EJB est faite au chapitre 5. Il faut s'attarder un peu sur le rôle prépondérant du conteneur, qui est un réceptacle de hauts niveaux pouvant fournir de multiples services à nos objets. Par exemple, c'est lui qui va assurer l'activation/passivation de nos objets. En effet, les objets au sein d'un conteneur EJB ont un cycle de vie assez complexe et ce, afin d'éviter des instanciations (créations) coûteuses en performance. Le conteneur fournit des caches d'objets de diverses natures permettant de ne pas avoir à créer sans cesse des objets à chaque appel de méthode. Cette stratégie permet aussi de ne pas avoir à s'occuper de la gestion des problèmes de concurrence entre requêtes. L'ouvrage de Richard Monson Haefel chez O'Reilly : *Enterprise Java Beans*, devrait vous permettre de creuser ce sujet passionant mais complexe.

Doclet

Les doclets ont été introduits avec le JDK 1.2 et proposent un moyen simple d'enrichir et de personnaliser la façon de générer la documentation accompagnant le code Java.

OUTIL **XDoclet**

D'une licence on ne peut plus ouverte, simple d'emploi, très facile à interfacer avec Ant, cet outil est le candidat idéal pour être notre garde-fou, il nous permet de conserver notre liberté vis-à-vis du conteneur.

- http://xdoclet.sourceforge.net/

Ces outils parmi lesquels on peut citer EJBGen ou XDoclet permettent de générer tout le matériel nécessaire au déploiement des EJB via l'écriture des simples classes d'implémentation. Bien entendu, l'écriture de ces classes suppose l'enrichissement du code source Java traditionnel de labels (tags) Javadoc spécifiques via des doclets.

Client riche – SWT

Avant d'en dire plus sur la nature de ce type d'interfaces, on peut commencer par les opposer aux clients dits légers, à savoir de simples pages HTML affichées dans un butineur web. Ces dernières disposent de nombreux avantages, entre autres :

- aucune installation n'est nécessaire sur le poste client (il y a toujours un navigateur web) ;
- elles sont très légères pendant l'exécution ;
- elles sont très simples à réaliser, puisque de nombreux outils sont disponibles et ce depuis de nombreuses années (DreamWeaver par exemple) ;
- très bien standardisées si l'on se limite à de l'HTML en version 3.2.

L'appellation « client riche » en revanche implique des interfaces complexes et évolutives, basées sur des composants réutilisables. Il s'agit de proposer à l'utilisateur des interfaces ergonomiques disposant de fonctionnalités de haut niveau (impression, aide en ligne et aides contextuelles). De plus, l'utilisateur doit pouvoir compter sur une interface assez intelligente pour s'assurer de la validité de ses saisies.

L'utilisation de composants évolués du type listes et autres arbres ou tables permet d'obtenir des interfaces beaucoup plus riches fonctionnellement que de simples pages HTML.

La simplicité de déploiement (un navigateur est toujours installé sur un PC de bureau) est l'avantage majeur de ces interfaces clientes. La richesse des interfaces possibles avec des clients dits « lourds » vient contrebalancer cet avantage.

POLÉMIQUE **Langages de script et DHTML**

L'apparition du JavaScript, VBScript et du DHTML ont rendu possible des pages HTML proposant des combo, listes et autres composants (widgets), mais ceci en sacrifiant la portabilité du code (le code JavaScript n'est pas interprété de la même façon sur tous les navigateurs) et la réutilisation/maintenance puisque ce code s'avère cauchemardesque à faire évoluer. Ceci explique le choix d'une couche cliente plus lourde mais portable et axée sur l'utilisation de composants réutilisables. Certaines alternatives existent mais leur aspect confidentiel est une entrave réelle à leur utilisation dans des projets de grande envergure. En effet, les clients n'aiment pas les prises de risque trop importantes ce qui doit dissuader quiconque de leur proposition utilisant de telles solutions.

La problématique liée au choix tourne ici au dilemme en raison des besoins et impératifs suivants :

- code portable ;
- code réutilisable ;
- puissance des composants ;
- interface légère (peu coûteuse en ressources mémoire et CPU).

La bibliothèque SWT et le projet Eclipse

Le projet Eclipse (initié par IBM) a engendré la création d'une bibliothèque (en anglais toolkit) graphique portable Java reposant sur des composants natifs au système. Ce projet ambitieux nous permet dorénavant d'avoir à notre disposition une alternative sérieuse à la bibliothèque Swing côté client. SWT est cette bibliothèque et va être utilisée afin d'obéir à une des contraintes liées au poste client : processeur standard (PII 450 MHz) et 128 Mo de RAM. Dans de telles conditions, la majeure partie des applications Swing ne peut être exécutée tout en gardant un rafraîchissement convenable. La page principale du projet Eclipse est disponible à l'adresse suivante : http://www.eclipse.org. Eclipse est un environnement de développement (IDE) très souple et modulaire, ce qui le rend extrêmement populaire. De plus, sa conception unique le rend très bien adapté à tous types d'utilisations, même éloignées du développement. Nous aurons l'occasion d'en dire plus sur ce projet un peu plus tard. Concentrons-nous sur SWT, ce toolkit graphique nous permettant de relever le défi des clients riches.

RAPPEL Swing

Depuis la sortie du JDK 1.2, Java inclut une API robuste et puissante utilisable pour la création d'interfaces graphiques (IHM ou GUI en anglais). Cette bibliothèque est actuellement mûre en matière de fonctionnalités et de stabilité des API mais pose toutefois le problème des performances. La question des performances réelles de Swing est un sujet épineux et sans trop s'engager, on peut constater qu'il est très difficile d'obtenir un comportement correct sur de petites machines avec une bibliothèque comme swing. En revanche, en tant que première arrivée sur le marché, Swing est la référence en la matière, elle bénéficie ainsi de nombreuses bibliothèques complémentaires permettant de simplifier l'impression (qui est une des grosses faiblesses historiques de l'API) ou d'ajouter de nouveaux composants (calendriers, gestion de graphes, etc.). Le choix n'est donc pas entièrement innocent car si on n'adopte pas Swing, on renonce à JavaHelp (aide en ligne) à Quest JClass et autres produits de qualité.

POLÉMIQUE Quelle technologie choisir ? SWT et XUL

De nombreuses possibilités s'offrent à nous, alors pourquoi choisir SWT qui semble être le challenger ? Quelles autres possibilités seraient aussi envisageables ? Si la première question trouve une réponse dans la liste des critères de choix énoncée précédemment, la seconde mérite une légère digression. Si Swing est la référence, SWT le challenger, XUL pourrait être le trouble-fête. En effet, XUL est issu du projet Mozilla et permet de générer des interfaces graphiques dynamiquement moyennant un descriptif XML. Malheureusement, XUL n'est pas destiné à générer du code Java et le projet JXUL visant à le faire semble abandonné. Dommage… À noter toutefois que XUL s'inscrit dans la lignée d'un autre projet Open Source : Glade, développé pour le projet Gnome (environnement graphique destiné à Linux et autres Unix).

OUTILS Quest JClass

- www.quest.com/jclass/

Le morceau de code suivant donné à titre indicatif (sans rapport avec notre application), montre un exemple simple d'utilisation de SWT.

Les imports nécessaires, rien d'original.

```
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Shell;
```

Le display est une notion propre à SWT, c'est le composant commun aux applications, qu'elles soient graphiques ou en ligne de commande, nécessitant l'affichage d'informations.

```
/**
* @author jerome
*/
public class HelloWorldSWT {
   public static void main(String[] args) {
      Display display = new Display();
```

Le shell joue le rôle d'un frame ou un Jframe en AWT/Swing.

```
// associe le shell au display
     Shell shell = new Shell(display);
     shell.setText("Hello world revisite en SWT");
     shell.setToolTipText("un tooltip debile");
```

On crée ici un layout, dont le type est propre à SWT (un des layouts disponibles en SWT 2.0), mais la notion est exactement la même que celle qui vous est sûrement familière avec AWT/Swing.

```
// cree un layout...
     // le positionne en mode auto wrapping..
     RowLayout layout = new RowLayout();
     layout.wrap=true;
```

Définit ce layout comme gestionnaire de placements d'objets pour le shell précédemment créé…

```
// associe le layout au shell
     shell.setLayout(layout);
```

Un widget SWT est toujours construit à partir d'un composite parent. Ici le composite est le shell… On remarquera aussi un paramètre SWT.NULL permettant de modifier le style du composant créé. (voir API SWT pour plus de détails).

```
// cree les widgets
     Label label = new Label(shell, SWT.NULL);
```

Remarquez les similitudes entre les méthodes disponibles sur les différents composants. C'est un aspect assez agréable du codage en SWT engendré par la remarquable hiérarchie de classes.

```
     label.setText("Coucou monde");
     Button button = new Button(shell,SWT.NULL);
     button.setText("Cliquez moi");
     button.setToolTipText("Si vous n'avez rien a faire..");
     Button button2 = new Button(shell,SWT.PUSH);
     button2.setText("Un autre bouton sans but");
     button2.setToolTipText("Si vous n'avez rien a faire..");
   // calcule la taille optimale
     shell.pack();
     // ouvre le shell
     shell.open();
     // attend la fermeture pour libérer les ressources
     // graphiques
     while (! shell.isDisposed()) {
        if (! display.readAndDispatch()) display.sleep();
     }
   }
}
```

Si SWT est issue du projet Eclipse, l'utilisation de cette bibliothèque n'implique en rien l'utilisation d'Eclipse. L'utilisation d'un `jar` contenant le code Java compilé de SWT et d'une bibliothèque C compilée pour la plate-forme (DLL sous Windows) suffit à mettre à disposition la bibliothèque. Afin de clarifier l'apparente symbiose SWT/Eclipse, la gestion des signets ne va pas se faire en tant que plug-in Eclipse (même si on aurait pu l'imaginer) mais bel et bien en tant qu'application à part entière (stand alone) et SWT n'est alors qu'une bibliothèque Java comme les autres (peu de différence avec un parseur XML ou une bibliothèque orientée réseau).

Déploiement

Le déploiement de l'application doit être le plus simple possible (maintenance aisée), reposer ainsi sur une technologie permettant :

- une gestion des mises à jour logicielles ;
- une installation simple sur le poste client ;
- une installation simple sur le serveur ;
- l'économie de la bande passante réseau grâce à la gestion de caches sur le poste client.

OUTIL **Java Web Start**

Sun propose depuis quelques mois une solution très intéressante quant au déploiement d'applications à travers Internet (intranet) : Java Web Start. Ce produit livré en standard avec le JDK 1.4 permet de gérer des déploiements d'applications Java sur un réseau (LAN ou WAN) en offrant une installation sur le poste client réduite au strict minimum (donc simple) et une installation serveur triviale (un MIME type à ajouter à la liste de ceux gérés par votre serveur web).

▸ http://java.sun.com/products/javawebstart/

Jar (Java ARchive)

Format d'archive proche du format zip. Il permet de packager des classes Java de manière à les distribuer efficacement. Le format Jar apporte des fonctionnalités de signature, *versioning* (gestion des versions) et *packaging* via un manifeste. Bien entendu le JDK fournit un utilitaire (jar) permettant de créer ou d'extraire de tels fichiers. On peut noter que sa syntaxe est proche de l'outil Unix : tar.

Plug-in

Il s'agit d'un ensemble de classes permettant d'étendre les fonctionnalités d'un système existant. Cela sous-entend que ce système a été pensé dans cette optique et qu'il a mis en œuvre une architecture assez ouverte pour accueillir à la volée de nouveaux composants. Un bel exemple d'une telle architecture est l'éditeur de texte de Romain Guy : Jext.

Parseur

Ce terme désigne un outil permettant d'extraire de l'information dans un fichier structuré suivant une grammaire donnée. Il est issu du vocabulaire lié à la théorie de la compilation dont le contexte dépasse largement le cadre de cet ouvrage.

Base de données

Comment stocker nos données ? Dans une base de données relationnelle. Pourquoi ce choix ? Car même si des solutions telles que des fichiers texte peuvent être envisagées, elles n'ont rien de professionnel dans le sens où elles n'offrent aucun des avantages d'une base de données comme :

- sécurité dans les accès (les moteurs de base de données peuvent gérer des droits applicatifs) ;
- sécurité dans les traitements (un bon moteur de base de données doit supporter les transactions) ;
- pas d'écriture de code spécifique à la base puisqu'on utilise un langage de requêtes standard (SQL), alors que l'utilisation d'un fichier texte requiert l'écriture de code (parsing) nécessaire à la lecture/écriture des données ;
- vitesse des traitements et puissance des requêtes (les moteurs de base de données sont optimisés pour entraîner des coûts minimes) ;
- puissance et facilité d'administration.

Le seul avantage d'une solution à base de fichiers est alors sa simplicité de mise en œuvre et l'absence d'administration mais cela ne pèse pas lourd dans la balance…

Notre application ne devra pas reposer sur des spécificités de telle ou telle base de données et donc ne réclamera qu'une compatibilité avec la norme ANSI SQL 92 nous assurant la disponibilité d'un driver JDBC.

Ceci permet d'ouvrir le choix sur une très vaste gamme de produits, pour n'en citer que quelques-uns :

- MySQL ;
- Postgres ;
- Oracle ;
- DB2 ;
- Sybase.

Bien évidemment cette compatibilité exclut formellement Microsoft Access. En effet, Access n'est pas un produit conforme aux normes ANSI SQL.

> ATTENTION **MySQL**
>
> Ce produit dispose de la majeure partie des fonctionnalités attendues d'un moteur de base de données (support du SQL ANSI 92, triggers) mais la récente introduction des transactions dans le produit en fait une fonctionnalité encore un peu expérimentale…

En résumé…

En optant pour une architecture à 5 couches, Blueweb applique le principe « d'abstraire pour maintenir » et joue la carte de la stabilité et de l'évolutivité. Le projet étant plus un prétexte qu'une application critique, l'équipe choisit de miser sur des technologies émergentes. De plus, en tant que projet pilote, l'application a pour vocation d'être la plus formatrice possible pour l'équipe en charge de sa réalisation. La part de risque reste mince, puisque les EJB sont une technologie arrivant à phase de maturation et que SWT/Jface trouve en Eclipse la parfaite illustration de sa mise en pratique.

L'architecture 5 couches proposée pour ce projet test est une architecture souple permettant de répondre à tous types de contraintes (déploiement des 5 couches sur une seule machine, un PC portable utilisé pour des démonstrations par exemple) comme un déploiement intégrant répartition de charge (*load balancing*) ou *clustering* avec une machine hôte pour la base de données et un serveur d'applications tournant sur plusieurs machines virtuelles pour de très grosses montées en charge. En bref, essayant d'éclaircir ce jargon d'informaticiens, l'architecture retenue est celle qui nous permet de répondre à une large gamme de besoins allant de la démonstration (donc facilité d'installation et utilisation) jusqu'aux applications de très hautes performance et disponibilité (applications cruciales pour une entreprise en rapport avec des chaînes de production par exemple). De plus, elle est propice à une réelle utilisation des composants en limitant au possible les couplages entre les diverses couches.

Enfin, on peut citer comme significative la position de leader tenue par Apache sur le marché des serveurs web ou la qualité d'un produit comme Samba. Un pragmatisme élémentaire fera se poser la question : pourquoi payer quand on peut avoir un équivalent gratuit ? Pourquoi s'enfermer quand on peut avoir la liberté ?

Load-balancing

Ce terme anglais peut se traduire par répartition de la charge et désigne la préoccupation pour certaines applications critiques (donc très chargées) de répartir le nombre de requêtes à traiter sur plusieurs machines, ce afin de garder des temps de réponse acceptables.

Clustering

Ce terme anglais est traduit classiquement en français par le concept de grappes de machines. Il s'agit en fait d'une façon artificielle de faire d'un groupe de machines (la grappe) une seule aux yeux du reste des machines du réseau. Ceci a l'avantage de créer des machines d'une puissance monstrueuse à partir de nombreuses petites machines tout en rendant très stable l'applicatif lancé sur cette machine artificielle, puisqu'il est peu probable que toutes les machines tombent en panne en même temps. C'est une manière séduisante pour traiter à la fois les problèmes de montée en charge de certaines applications (on multiplie les ressources mémoire et le nombre de processeurs) et la disponibilité des applications (permet théoriquement d'aboutir à une disponibilité de 100% à moindres frais).

chapitre 2

Environnement de développement CVS et Ant

Le travail en équipe, le respect de chartes de développement, les outils de gestion de projet sont autant de points constituant un environnement de développement. La méthodologie utilisée, elle aussi, influe sur la façon de travailler et donc sur les structures et procédures à mettre en place. Les outils exposés par la suite sont la base du travail en équipe et visent à former un ensemble cohérent et facilitant la productivité.

SOMMAIRE

MOTS-CLÉS

- CVS
- Ant
- task
- build-file/makefile
- ligne de commandes
- client HTTP
- collections et types primitifs

OUTIL **CVS**

CVS est lui aussi un produit Open Source, qui possède des antécédents tels que la question de son adéquation avec les besoins de BlueWeb ne se pose pas. C'est la Rolls des outils de gestion de configuration, car il est gratuit, performant, stable et relativement simple à installer. De plus, son adminsitration est réduite au minimum. Utiliser ce produit, c'est choisir en toute sérénité une valeur sûre. On peut rappeler à titre anecdotique que des projets tels que le noyau Linux, le serveur web Apache et bien d'autres sont gérés et développés par des centaines de développeurs sur la planète en utilisant cet outil.

- http://www.cvshome.org/

RÉFÉRENCE **Subversion**

Ce produit est disponible sur la page suivante : http://subversion.tigris.org/. Subversion est un produit déjà très mûr ; il ne lui manque plus qu'un peu de temps de manière à ce qu'il soit intégré dans des outils de développement comme Eclipse ou Ant.

ALTERNATIVES **Continuus et ClearCase**

Ces deux produits sont sûrement les plus aboutis en la matière, seulement leur coût (prix de la licence) et le niveau de formation requis avant utilisation réservent leur utilisation à des entreprises fortunées. Par ailleurs, il faut signaler qu'il est nécessaire de dédier environ 1 personne sur 10 développeurs pour la seule administration de ces produits, ce qui a évidemment un coût non négligeable (quand il est possible de l'envisager).

ALLER PLUS LOIN **Pourquoi une machine de production ?**

Il est indispensable de passer outre les problèmes de `classpath` et de faire en sorte que les livrables (classes Java ou fichiers Java) se comportent correctement dans un environnement « sain ». C'est pour cela qu'une machine doit être dédiée à la production et que celle-ci ne peut se faire sur le poste de l'un des développeurs du projet. La seule contrainte est d'avoir une JVM installée et un Ant proprement configuré, puis de lancer l'exécution du `makefile`…

Le tableau suivant plante le décor commenté dans la suite de ce chapitre. La justification du choix du serveur d'applications ayant été faite précédemment, on se concentrera sur les autres facettes des choix par la suite.

Gestionnaire de code source	Outil de make	Serveur d'applications
CVS	Ant	Ensemble intégré Jboss + Tomcat

Gestionnaire de code source : CVS

Depuis de nombreuses années, BlueWeb utilise CVS comme outil de gestion des sources et de contrôle de versions, donc ce nouveau projet va naturellement se voir offrir son espace réservé sur le serveur CVS.

Les développeurs auront l'esprit libre pour travailler, un outil éprouvé se chargera de pallier d'éventuelles « bêtises » rendant toujours possible le retour en arrière vers une version précédente, tandis que le responsable qualité aura toujours la possibilité de restaurer un environnement « labellisé ». La gratuité de l'outil nous amène à nous demander pourquoi s'en priver… On peut noter que tout autre gestionnaire de code source (Microsoft Visual Source Safe, Rational Clearcase, Continuus, etc.) aurait pu être utilisé. CVS est utilisé avec Ant, notamment au chapitre 7 consacré à l'audit du code. Le grand rival libre de CVS, Subversion, est désormais à l'état de production. Ce projet est conçu de manière à ressembler le plus possible à CVS tout en gommant les quelques défauts connus de ce vénérable ancêtre.

Production des versions : Ant

Le responsable qualité du projet, Michel, alarmé par ses lectures, tient à être le plus détaché possible des outils de production fournis avec les environnements de développement et à pouvoir utiliser la puissance d'une machine neutre pour procéder à des *builds* nocturnes.

Il décide donc d'utiliser sa machine comme machine de référence, pour lancer des tâches répétitives durant la nuit. L'installation est minime (un JRE + l'installation de Ant et quelques bibliothèques), ne coûte pas cher à la société, ne le perturbe pas durant son travail (car Michel dort la nuit…) et doit permettre de déceler très vite différents problèmes (problèmes de compilation, de tests unitaires non conformes, qualité). Bien entendu, cet outil sera utilisé avec le serveur de fichiers sources (CVS), afin de produire à la demande une version labellisée ou encore opérer de la mise en forme sur tout le contenu du référentiel de sources.

En fait, au niveau du choix de cet outil, BlueWeb n'a guère d'alternatives, car si tous les environnements de développement du marché (JBuilder, Forté, Netbeans, Eclipse, etc.) proposent des fonctions de compilation des sources (ainsi qu'une interaction avec les référentiels de code source), ils se révèlent insuffisants car ils ne couvrent qu'une faible partie des besoins en matière de production des versions :

- interaction avec le serveur CVS ;
- compilation du code ;
- *packaging* (création d'archives au format jar, etc.) ;
- exécution des tests unitaires ;
- génération de documentation ;
- instrumentation des fichiers sources (c'est-à-dire transformation de ceux-ci par l'intervention d'un outil) ;
- intégration d'outils tiers (génération automatique de code, d'indicateurs).

Bref, la concurrence sur ce marché est très limitée et Ant s'avère être l'outil standard dans le monde Java, même si l'on doit reconnaître que les utilisateurs de l'outil Unix make doivent pouvoir arriver aux mêmes fonctionnalités.

Continuous integration

Cette expression provient du jargon du mouvement lié à l'Extreme Programming et notamment aux idées « d'intégration perpétuelle » (pour *continuous integration* dans le vocabulaire original anglais). Les buts sont multiples mais ils permettent surtout de :

- tester le code entreposé dans le serveur CVS ;
- générer des versions du produit sans intervention humaine ;
- appliquer divers outils sur l'ensemble des sources du projet de manière à traquer des bogues de non-régression (tests unitaires) et tester la conformité du code avec la charte de codage interne.

De l'utilisation de la variable CLASSPATH

L'aparté dédié aux machines de production permet d'aborder un problème épineux lié au développement en Java : l'utilisation de la variable d'environnement CLASSPATH. Cette variable a pour rôle de délimiter au sein d'une variable toutes les bibliothèques (`.zip` ou `.jar`) et tous les chemins d'accès aux fichiers `.class`. Une mauvaise maîtrise de cette variable entraîne de nombreuses questions sur les forums, mais le risque le plus grand provient d'une maîtrise partielle des concepts liés à cette variable. En effet, vous ne devez jamais supposer quoi que ce soit sur le « CLASSPATH » de la machine sur laquelle s'exécute votre application. Il vous revient de *packager* votre application correctement pour qu'elle puisse être autonome et surtout sans conséquence pour l'environnement de votre client. Pour cela, Ant ou un script (shell sous Unix ou batch sous DOS) seront vos meilleures armes. Un bon programmeur Java n'utilise pas la variable d'environnement CLASSPATH !

Ant – Vocabulaire de base

La compréhension de l'outil nécessite seulement deux mots, même si la maîtrise réelle du produit réclame un peu plus de travail, mais le jeu en vaut la chandelle…

- Task (tâche) : une tâche est au sein d'un makefile l'équivalent d'une instruction dans un langage de programmation, c'est-à-dire un objet servant à accomplir une fonction bien spécifique sur éventuellement un certain nombre de paramètres d'entrée (nom du répertoire contenant les classes) et entraînant un certain résultat (fichier `.jar`). Par exemple, la tâche `copy` permet de copier un fichier (ou un ensemble de fichiers), la tâche `jar` permet de créer un fichier jar, etc.

RAPPEL **Ant**

Ant est un outil de make, c'est-à-dire d'abord destiné à gérer la compilation de code source Java et la production de fichiers `.class` (*bytecode*). Cet outil est entièrement écrit en Java et utilise des makefile XML (fichiers `build.xml`).

- Target (cible) : une *target* est un point d'entrée dans un makefile (donc caractérisé par son nom : `build`, `doc`, `all`) et vise à accomplir une des étapes d'un processus de production.

Par exemple, on pourrait décider de créer un makefile compilant les sources et d'appeler cette étape du processus global de production par *`compile`*. Une autre étape pourrait être la destruction du résultat d'une compilation appelée *`clean`*. Notons qu'un projet contient des cibles (*targets*) faisant appel à différentes tâches éventuellement interdépendantes. Voici ci-dessous un exemple de makefile pouvant répondre à ce besoin.

Exemple de makefile

Création d'un projet dénommé « Petit Projet » dont le répertoire de base (`basedir`) est le répertoire courant (`.`) et la cible par défaut (`default`) est `compile`.

```
<project name="Petit Projet" default="compile" basedir=".">


<!-- ici on positionne des valeurs pour ce build -->
  <!-- d'une maniere tres originale on indique que ${src} va
representer le directory src -->
```

`src` va désigner le répertoire où seront stockés les fichiers sources. De même, `build` désigne le répertoire créé par la compilation.

```
<property name="src" location="src"/>
  <property name="build" location="build"/>
<!-- cette cible realise l'init de notre build-file -->
  <!-- ici reduite a cree le repertoire pointe par la variable
   ${build} (valeur build d'apres l'initialisation precedente -->
<target name="init">
<!- creee le repertoire ${build} -->
<mkdir dir="${build}"/>
</target>
  <!-- cette cible compile le code source -->
  <target name="compile" depends="init"
```

Ici, on peut noter l'utilisation de l'attribut `description` permettant de renseigner un utilisateur de l'utilité d'une cible. Remarquez que ce message n'est affiché que quand l'utilisateur désire en savoir plus sur votre processus de construction, c'est-à-dire en tapant `ant -projecthelp`.

```
        description="compile le code source requiert l'execution
                     de la tache init " >
<!-- Compile le code contenu dans: ${src} en deposant les classes
     dans : ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>
<!-- un peu de menage pouvant etre utile, cette cible detruit le
repertoire ${build} -->
<!-- donc pour etre sur de repartir de zero il faudra l'invoquer
     via un: ant clean -->
  <target name="clean"
        description="detruit le repertoire de build" >
<!-- Detruit le repertoire ${build} -->
    <delete dir="${build}"/>
  </target>
</project>
```

Adapter Ant à ses besoins

L'objet de ce livre n'est pas d'aborder Ant de manière exhaustive (est-ce possible ?). On ne tient pas à évoquer toutes les tâches prédéfinies (*built-in tasks*) et encore moins à évoquer tous les contextes d'utilisation possibles, mais étant donné l'importance de l'outil au sein de l'ouvrage, il est nécessaire de s'attarder un peu sur quelques spécificités de ce produit qui le rendent aussi particulier.

Tâches utilisateurs (Custom Tasks)

Ant est un outil pouvant être étendu à l'infini et adapté à vos besoins particuliers et ce, très simplement.

Pour ce faire, chaque besoin propre doit être exprimé sous forme d'une nouvelle tâche dite *custom task*.

Avant, seul le codage des tâches en Java était disponible ; depuis la version 1.5, Ant vous permet dorénavant de coder vos tâches via deux types de langages :

- Java ;
- un langage de script respectant la norme BSF (pour *Bean Scripting Framework*) d'IBM.

REMARQUE Ce livre et les scripts

Nous ne nous intéresserons qu'à la manière classique de coder une *custom task* en Java, ceci en raison du nombre de langages de script possibles (BeanShell, Rhino…) et afin que le code demeure portable (il ne nécessite pas l'installation de la dernière version de Ant).

Codage en Java d'une tâche utilisateur

Pour un premier exemple de codage d'une tâche Ant, nous n'allons pas nous lancer dans un développement important. Nous implémenterons une tâche de Ant : la tâche `Echo`. Celle-ci redirige vers la console (l'écouteur par défaut) le message donné en entrée.

Principe de codage : convention JavaBean

Très simple à coder, une *custom task* ne réclame qu'une correspondance entre les attributs spécifiés dans le fichier XML pour l'appel de cette tâche avec des attributs d'instance de la classe Java.

Chaque attribut de votre classe personnalisée doit être privé et doit mettre à disposition une méthode `setXXX` permettant de positionner la valeur de l'attribut `XXX`. C'est une façon de coder très proche des conventions régissant les JavaBeans (plus simple).

Une seule contrainte : votre classe doit hériter de la classe `org.apache.tools.ant.Task`.

Enfin, le code correspondant à l'exécution de votre tâche doit être placé dans une méthode publique `execute()`.

Une tâche Ant très simple

On place notre classe dans un paquetage spécifique…

```
package com.blueweb.tasks;
```

Voici les imports requis par cette classe. Ici, on s'attache à la classe `Task` du paquetage `org.apache.tools.ant`. Notez aussi qu'il est de bon goût d'attraper les exceptions du type `BuildException` et d'en lever de nouvelles en cas de problème.

```
import org.apache.tools.ant.Task;
import org.apache.tools.ant.BuildException;
/**
* Une classe singeant la task Echo (built in task de Ant)
* Elle n'a pas d'autre prétention que de donner un exemple
* simple de programmation d'une custom task Ant
* @author J.MOLIERE - 2002
*/
public class SimpleEcho extends Task{
```

Surtout bien remarquer la convention JavaBean utilisée lors du codage des tâches Ant. Un attribut, renseigné lors de l'appel dans le fichier XML (build.xml) se verra associer des méthodes, dites accesseurs.

```
// attribut d'instance
   // c'est le message qui va être affiché
  private String echoMessage = "";
/**
   * Cette méthode est un « setter » pour l'attribut echoMessage. Elle
   * permet de positionner la valeur du message à afficher.
   * Remarquez bien la casse dans le nom de la methode
*/
  public void setEchoMessage(String aMessage){
    echoMessage = aMessage;
  } // setEchoMessage()
  /**
  * c'est ici que l'on place le code devant
  * être execute lors de chaque appel de la task
  * dans notre cas un simple System.out.println
  * @exception BuildException, si quelque chose se passe mal...
*/
  public void execute() throws BuildException{
```

Ici quoi de plus simple qu'un simple appel à `System.out.println()`. Néanmoins, le code placé dans cette méthode peut être aussi complexe que nécessaire. Pour des exemples de tâches utiles, n'hésitez pas à consulter le code source des tâches Ant, il est là pour cela.

```
System.out.println("---> " + echoMessage );
  } // execute()
}
```

Appel de la tâche SimpleEcho

```
<!-- Simple build file -->
<!-- pas de construction de jar ou de classes -->
```

Ce projet a pour nom BlueWeb, très original n'est-ce pas ? La cible par défaut s'appelle `default` ce qui là encore est très original.

```
<!-- juste un appel a notre custom task SimpleEcho -->
<project name="BlueWeb" default="default">
<!-- le tag taskdef permet d'associer un nom a une classe Java de
votre classpath -->
```

Chaque tâche extérieure à Ant doit être déclarée de cette façon. Ici on présuppose que le fichier `.class` correspondant à cette classe se trouve dans votre CLASSPATH. C'est une chose qu'il faut éviter de présupposer, mais qui permet dans notre cas de se concentrer sur l'essentiel.

```
<!-- ici notre tache s'appellera bluewebecho -->

<taskdef  name="bluewebecho"
          classname="com.blueweb.tasks.SimpleEcho"/>
<target name="default">
```

Une fois la tâche définie, on peut maintenant l'invoquer depuis notre cible principale.

```
<bluewebecho echoMessage="Bonjour Monde"/>
</target>
</project>
```

Le build-file Ant et l'approche qualité

Ant : le couteau suisse du développement Java ?

Maintenant que sont posées les briques, commençons à monter les murs…

Pour ses qualités d'outil multi-plates-formes, de facilité d'écriture des règles de notre processus de production (la syntaxe XML évite les pièges rebutant le néophyte face à son premier makefile) et d'extensibilité très simple, Ant est l'outil clé, le fil conducteur et le dénominateur commun via lequel nous appliquerons différents types de traitements sur notre code source :

- instrumentation du code source afin de créer du *bytecode* depuis un fichier source Java (comme le fait XDoclet qui, via un fichier d'implémentation d'un EJB, crée des classes Java et des fichiers de déploiement) ;
- jeu de tests, lancement des différents tests unitaires et rédaction d'un rapport, ce qui permet de prévenir les bogues dits de non-régression (c'est-à-dire des bogues provenant suite à des modifications de code source opérationnel) ;
- contrôle qualité du code source, via lancement d'une tâche nommée `checkstyle` permettant de signaler des anomalies dans le respect des chartes de nommage ;
- application des contraintes de déploiement et d'architecture (voir chapitre 6 sur le déploiement pour un exemple de *custom task* permettant de déployer notre application via Java Web Start).

Notre *build file* va donc contenir des processus tout à fait différents puisqu'ils n'impliquent pas les mêmes acteurs :

- Le chef de projet et le responsable qualité s'attacheront à faire respecter à la fois les conventions de codage adoptées en interne et la méthodologie de développement (*Extreme Programming* ?). Ils seront exigeants quant à la fourniture de jeux de tests et au bon déroulement de ceux-ci au sein d'une suite de tests.
- Le responsable du déploiement cherchera à produire une version conforme aux spécifications et aux contraintes du projet.
- Les codeurs de la partie serveur utiliseront le makefile avant toute production de manière à déployer au sein de leur serveur d'applications leurs composants (servlets/EJB).

B.A.-BA Build-file

Ce terme désigne le nom donné dans le vocabulaire Ant au fichier XML décrivant les étapes constituant le cycle de production d'une version d'une application. Un fichier de ce type aura donc différentes parties couvrant compilation, création d'archives, etc.

PRÉCISION Réutilisation

Il n'est pas indispensable de tout réinventer et d'adopter des normes à l'opposé de ce qui se fait dans le petit monde Java… N'hésitez surtout pas à réutiliser et à vous inspirer des différents grands projets Open Source : JBoss, Jakarta, etc.

Factoriser

Ce terme est important, il est donc préférable de l'utiliser plutôt que de le contourner. Il désigne le fait de regrouper, de conserver en un seul endroit des données relatives à un sujet. Ce terme vient tout droit du jargon mathématique. Il relate une préoccupation constante du monde de l'entreprise.

Choix des bibliothèques

Toute application réclame un certain nombre de services standards (logs, *parsing* XML, sérialisation Java ↔ XML, etc.). L'approche objet, poussant à factoriser tout ce qui peut l'être et à réutiliser tout ce qui l'est, rend inadmissible le fait de

voir certaines équipes se lancer dans la réalisation du énième framework de gestion des logs. Réutilisons !

Critères de choix

Maintenant que le mot réutilisation est acquis, il convient de se fixer des bornes cadrant nos choix. Quels sont les critères ? Quel est leur impact sur votre projet ? S'il est impossible de répondre de manière générique à cette question, étant donné que chaque société a un historique propre, que chaque équipe a des sensibilités et des contraintes budgétaires différentes, il est évident que les critères suivants vont intéresser chacun d'entre nous (sans forcément avoir la même hiérarchie d'importance d'un contexte à un autre) :

- **Type de licence de la bibliothèque** : GPL ou LGPL, Apache ou BSD, Sun ou propriétaire… autant de déclinaisons possibles mais la lecture des documentations associées à la licence d'une bibliothèque vaut toujours la peine requise pour le faire. En effet, imaginez que, pour un navigateur web écrit en Java à but lucratif et à optique propriétaire, BlueWeb choisisse un parseur XML à licence GPL… Les licences du type Apache ou BSD sont réellement très permissives, il est donc peu probable qu'opter pour une API distribuée en ces termes soit un mauvais choix.
- **Adéquation avec les spécificités du projet.** Pour certaines applications (domaine de l'embarqué, applets, etc.), le volume occupé en mémoire ou en espace de stockage (taille du fichier jar) peut être un critère rédhibitoire, éliminant de fait de très bonnes bibliothèques. C'est pour cela notamment que l'on peut trouver des implémentations ultra-légèret de parseurs XML.
- **Prix.** Une bibliothèque peut être très économique (maintenance/coût de développement) tout en étant payante. Il faut toutefois veiller à ce qu'elle n'implique pas des *royalties* faisant chuter la marge sur une vente de notre logiciel.
- **Maintenance.** Quels sont les contrats proposés, quelle est l'équipe de développement, quel impact peut-on avoir sur l'ajout de nouvelles fonctionnalités ?
- **Qualité de la bibliothèque et de la documentation associée**. Il faut aussi tenir compte de la disponibilité de listes de diffusion/forums d'entraide et du nombre de livres/présentations relatifs au produit.

POLÉMIQUE **La licence GPL**

Pour de plus amples informations sur cette licence logicielle, se reporter au site web www.gnu.org. Cette licence est une sorte de choléra juridique obligeant tout produit utilisant une composante à licence GPL à être lui-même sous licence GPL. Ceci ne signifie pas que c'est une mauvaise licence, elle est seulement très exigeante… Cette exigence est d'ailleurs la raison d'être d'une licence plus permissive, la LGPL.

Par exemple :

- http://mitglied.lycos.de/xmltp/intro.html

Le JDK 1.4 comporte désormais sa propre API de *logging* : `java.util.logging`

BlueWeb joue la carte de l'Open Source

BlueWeb sait faire la part des choses entre phénomènes de mode et réalisme économique. Elle décide donc de s'investir dans certains projets Open Source particulièrement valables.

- Log4j pour le mécanisme de gestion des logs (puissance de l'API, portabilité non limitée au dernier JDK 1.4.2) ;
- Oro pour la gestion des expressions régulières ;

- Castor XML, pour traiter la problématique de la transformation d'objets Java en XML et vice versa.

Le chapitre 3 vous en dira plus sur ces trois bibliothèques.

En cas de problèmes majeurs ou de besoins spécifiques, elle ne s'interdit pas de recourir à d'autres produits, mais ces deux choix cadrent déjà en grande partie les développements futurs.

De même qu'au niveau des conteneurs J2EE, BlueWeb n'a aucun intérêt à se lier avec un des ténors du marché et joue aussi sur ce plan la carte de l'Open Source en réduisant d'autant plus ses frais pour la phase de développement, mais en n'excluant pas d'intégrer sa solution sur d'autres conteneurs, suite à des souhaits émanant de clients (d'où le souhait de portabilité et de respect des normes).

Le tandem choisi est le duo de produits (*bundle*) JBoss + Tomcat, proposant une solution J2EE gratuite, bien documentée, soutenue par une très large communauté de développeurs.

Le projet Jakarta Commons

Le projet Jakarta (sous-projet Apache) a donné naissance à de multiples projets dont l'indispensable projet Ant présenté précédemment. Parmi ceux-ci, il est bon de s'intéresser à une boîte à outils regroupant des composants répondant à des besoins fréquents dans divers domaines : le projet Commons.

Ce projet ambitieux met à disposition de tous des bibiothèques simples et puissantes répondant à de nombreuses problématiques :

- simplifier et standardiser le dialogue avec des composants serveurs (Net), commons-net ;
- répondre à des carences du langage Java (Lang), commons-lang ;
- amener une couche d'abstraction au dessus de la couche de gestion des traces utilisée (Logging), commons-logging ;
- simplifier la gestion de fichiers de configuration XML (Digester) ;
- fournir d'autres implémentations aux interfaces de collections définies dans le paquetage `java.util`, commons-collections ;
- fournir un framework à même de simplifier la gestion des options dans un programme en ligne de commandes (CLI), commons-cli ;
- et bien d'autres.

Sans chercher à passer en revue tous ces outils, nous présenterons ici de courtes introductions pour quatre de ces produits.

RÉFÉRENCE Le projet Commons

La page web concernant les divers projets rassemblés au sein de ce projet est :
▸ http://jakarta.apache.org/commons

Introduction : l'univers de la ligne de commandes

Même si les interfaces graphiques sont dorénavant omniprésentes, l'époque des outils fonctionnant en ligne de commandes n'est toutefois pas encore révolue. En effet, dès qu'il s'agit d'automatiser le fonctionnement d'une application

(insertions dans une base en mode batch, manipulation journalière de fichiers de données, etc.), le passage par des IHM complexes est exclu et les bons vieux outils en ligne de commandes reprennent tout leur sens.

Pour tous ceux ayant eu à gérer des programmes pouvant accepter diverses options, il est inutile de rappeller le calvaire du programmeur devant gérer le contrôle des options passées par l'utilisateur en tenant compte des divers contextes :

- informations obligatoires et options facultatives ;
- options nécessitant des valeurs et simples *flags* ;
- options mutuellement exclusives...

La figure 2-1a propose une capture de la page de manuel de la commande Unix `ps`. Cette capture cherche juste à sensibiliser le lecteur sur la complexité de certaines commandes et sur le nombre d'options disponibles pour certaines d'entre elles. Dans le monde du langage C, une bibliothèque s'était imposée : `getopt`. Celle-ci fournissait un moyen de simplifier la gestion des arguments pour tout programme écrit en C ou C++. Commons CLI propose le même type de fonctionnalités, mais adaptées au monde Java.

B.A.-BA **Pages de manuel**

Pour les lecteurs peu coutumiers du monde Unix, il est bon de rappeler que tout système Unix fournit à l'utilisateur un système d'aide en ligne, consultable à tout moment par le biais de la commande `man`.

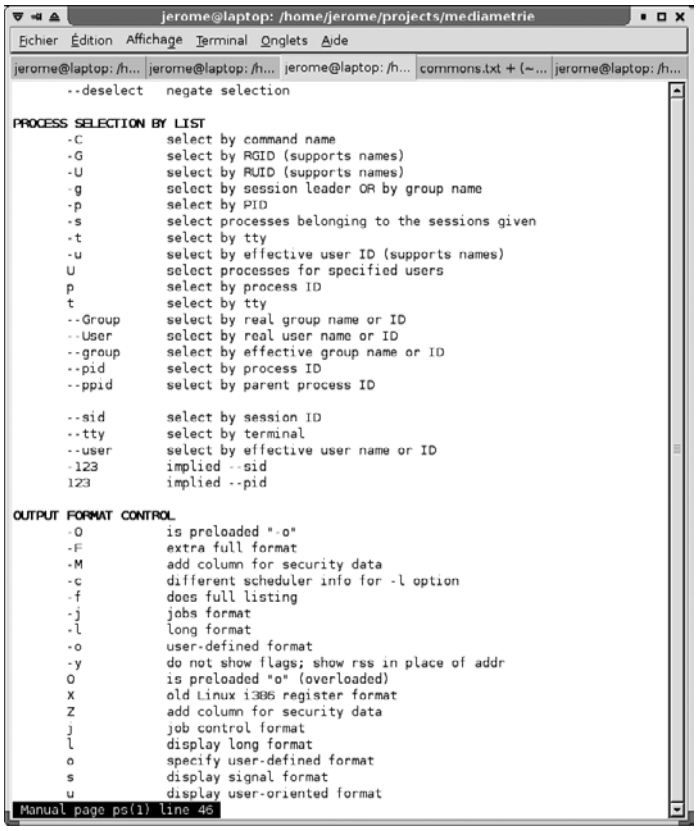


Figure 2–1a
Capture d'écran de la page de manuel de la commande ps

Il faut aussi préciser qu'afin de simplifier la lecture des scripts de gestion des systèmes Unix, on a introduit une notation plus verbeuse pour les diverses options des programmes, connue sous le nom de style GNU et opposée au très austère style POSIX. Il s'agit en fait simplement d'introduire des noms d'options explicites. On réservera traditionnellement l'utilisation des noms longs pour des scripts (augmentation de la clarté de ceux-ci), tout en utilisant les noms d'options abrégés pour l'exécution de commandes en mode interactif.

Ainsi, la figure 2-1b propose un extrait de la page de manuel de la commande `tar` (gestion d'archives de répertoires).

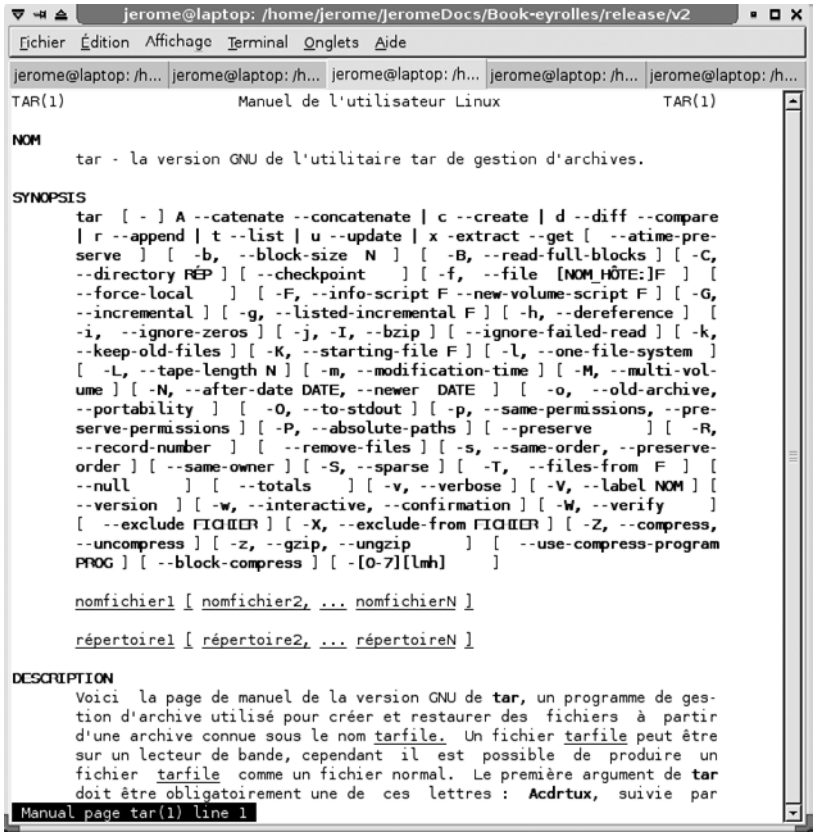


Figure 2–1b
Options de la commande tar

Cette figure permettra au lecteur diligent de remarquer que certaines options seront disponibles sous deux formes :

- une forme abrégée par le biais d'une simple lettre (`c` pour `create` par exemple), destinée à une utilisation au quotidien ;
- une forme développée (`create`) destinée à une utilisation dans un script.

Il s'agit bel et bien d'un progrès facilitant la compréhension de scripts d'administration complexes...

Le projet Commons CLI

Ce projet vise à fournir une bibliothèque de bon niveau permettant de faciliter autant que faire se peut la création de programmes nécessitant des options saisies par l'utilisateur au lancement du programme.

Il va donc fournir des classes permettant de modéliser les intervenants dans ce contexte, à savoir :

- une instance d'une implémentation de l'interface `org.apache.commons.cli.CommandLineParser`, permettant d'obtenir une référence sur un objet du type `org.apache.commons.cli.CommandLine` modélisant la ligne de commandes ;
- une instance de la classe `Options`, représentant l'ensemble des options gérées par un programme, et pouvant être instanciée directement ou en passant par la méthode statique de la classe `OptionBuilder`.

Les principales relations entre intervenants sont détaillées dans le schéma UML de la figure 2-2.

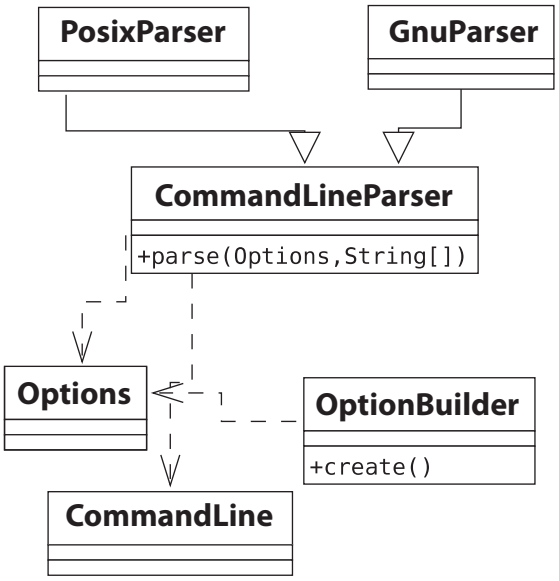


Figure 2–2
Relations entre intervenants

Rien de tel qu'un petit exemple pour appréhender ces concepts...

Le code suivant permet de gérer une application dotée de diverses options :

- `help(h)` pour obtenir de l'aide ;
- `exit (x)` pour sortir ;
- `config (c)` pour utiliser un fichier de configuration transmis avec l'option `file`.

Il n'y a pas de quoi révolutionner l'informatique, mais c'est un programme suffisant pour démontrer la puissance du produit...

```
package commons.cli;

import org.apache.commons.cli.BasicParser;
import org.apache.commons.cli.CommandLine;
import org.apache.commons.cli.CommandLineParser;
import org.apache.commons.cli.GnuParser;
import org.apache.commons.cli.HelpFormatter;
import org.apache.commons.cli.Option;
import org.apache.commons.cli.OptionBuilder;
import org.apache.commons.cli.Options;
import org.apache.commons.cli.ParseException;
import org.apache.commons.cli.PosixParser;
```

Imports nécessaires au programme et déclaration du paquetage hôte

```
public class CliExemple {
```

Une classe démontrant l'utilisation de la bibliothèque Commons Cli.

```
public static void main(String[] args) {
     Options options= new Options();
     Option help = new Option( "h","help",false,
                               "print this message" );
     Option exit= new Option("x","exit",false,
                            "exit the application");
     OptionBuilder.withArgName("file");
     OptionBuilder.hasArg();
     OptionBuilder.withDescription(
                           "specify the config file to be used");
     OptionBuilder.isRequired(false);
     OptionBuilder.withLongOpt("config");
     Option config= OptionBuilder.create("c");
     options.addOption(config);
     options.addOption(help);
```

Étape n°1 : définir les différentes options gérées par le programme.
Ici ce programme test est assez pauvre donc ne gère que trois options :

- `help` pour obtenir de l'aide,
- `exit` pour quitter,
- `config` pour passer un nom de fichier de configuration.

Bien entendu, le programme ne fait rien d'utile de ces options...

```
     CommandLineParser parser = new BasicParser();
     try{
       CommandLine line = parser.parse( options, args );
```

Il s'agit ensuite de définir le type de parseur manipulé pour la gestion des options de notre programme. Ici c'est un parseur minimaliste qui a été choisi. Une fois le parseur instancié, on peut se lancer dans le parsing des options passées par l'utilisateur (d'où l'utilisation de la variable `args`). Notez bien le type retourné : `CommandLine`.

```
      if (line.hasOption("h")){
        HelpFormatter formatter = new HelpFormatter();
        formatter.printHelp( "CliExemple", options );
        }
        // a t'on donne un fichier de config en param ?
        if(line.hasOption("c")){
        // il semble que oui...
        // quel est ce fichier ?
        System.out.println("fais qqch du fichier de config =" +
                           line.getOptionValue("c"));
        }
```

On peut désormais exploiter l'instance de `CommandLine` (ligne de commandes) obtenue en la questionnant notamment par le biais de sa méthode `hasOption()`.
On note la nécessité de gérer une exception du type `ParseException`.

```
        if(line.hasOption("x")){
        System.out.println("See you next time");
        System.exit(1);
        }
    }
    catch( ParseException exp ) {

        // oops, something went wrong
        System.err.println( "Parsing failed. Exception is =" +
                    exp.toString() +"  Reason: " + exp.getMessage() );
    }

  }
}
```

Les figures 2-3a, 2-3b et 2-3c montrent divers lancements de l'application vus dans la console d'Eclipse.

Figure 2–3a

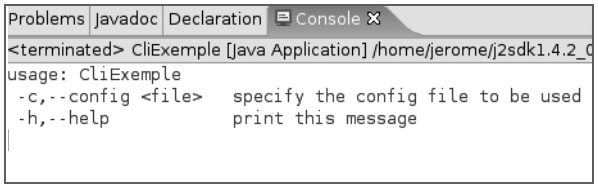


Figure 2–3b

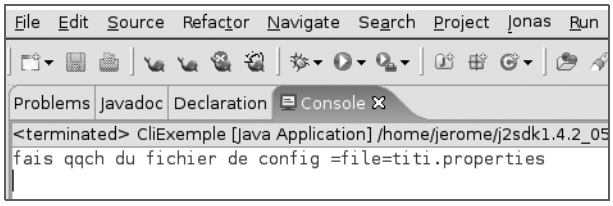
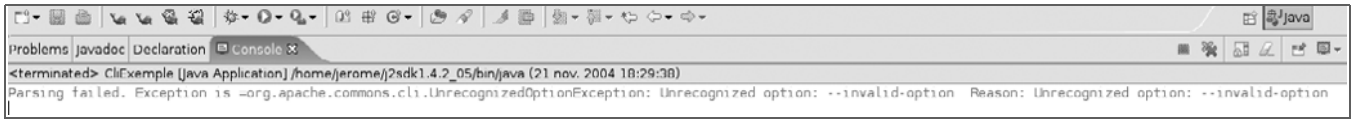


Figure 2–3c



L'exemple suivant illustre comment créer des groupes d'options de manière à créer des arguments mutuellement exclusifs. Ainsi, dans le cas d'un programme d'archivage et de compression (tel le fameux Winzip), l'utilisateur ne pourra choisir lors de la même invocation qu'une option entre compression et décompression.

Déclaration du paquetage et imports nécessaires au programme.

```
package commons.cli;

import org.apache.commons.cli.CommandLine;
import org.apache.commons.cli.CommandLineParser;
import org.apache.commons.cli.GnuParser;
import org.apache.commons.cli.Option;
import org.apache.commons.cli.OptionGroup;
import org.apache.commons.cli.Options;
import org.apache.commons.cli.ParseException;
import org.apache.commons.cli.PosixParser;
```

```
/**
 * <p>
 * Cet exemple va creer des groupes d'options mutuellement exclusives
 * ainsi dans le cas d'utilisation d'un utilitaire d'archivage
 * on va chercher a compresser ou decompresser mais pas les 2 en meme
temps
 * </p>
 * @author jerome@javaxpert.com
 *
 */
public class AdvancedCliExample {
   public static void main(String[] args) throws ParseException {
      Option compress= new Option("c",false,
                                  "compress the given file");
      Option uncompress= new Option("x",false,
                                  "uncompress the given file");
      OptionGroup group=new OptionGroup();
      group.addOption(compress);
      group.addOption(uncompress);
      Options opts= new Options();
      opts.addOptionGroup(group);
      CommandLineParser parser = new GnuParser();
      CommandLine line=parser.parse(opts,args);
      if(line.hasOption("c")){
         System.out.println(" compression demandee");
      }
      if(line.hasOption("x")){
         System.out.println("decompression...");
      }
   } //main()
}
```

Ici, on va bâtir les options à partir d'un groupe d'options, c'est-à-dire une instance de la classe `OptionGroup`. Cette classe modélise les options mutuellement exclusives (ici compression et décompression).

Les figures 2-4a et 2-4b illustrent la console d'Eclipse lors de deux lancements de ce programme avec une option de compression ou avec les deux options positionnées (ce qui est inacceptable à nos yeux).

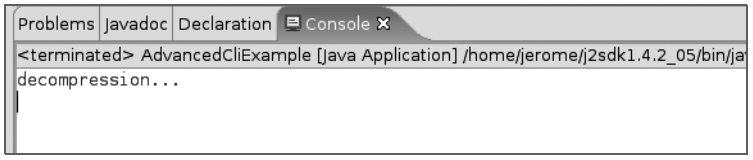


Figure 2–4a

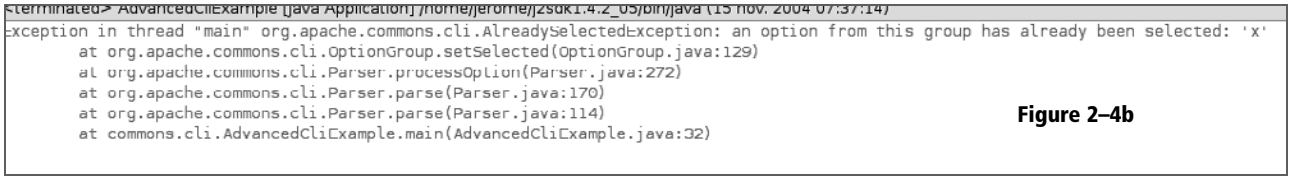


Figure 2–4b

PRÉCISION **Les difficultés du Web**

Ce projet prend tout son sens quand on saisit le nombre de finesses et de pièges propres au développement en utilisant le protocole HTTP : gestion des divers types de requêtes (POST/GET), manipulation de cookies, autant de cas concrets pour lesquels le langage Java n'offre guère de facilité.

PIÈGE **Attention aux versions**

Comme toujours dans le monde du logiciel libre, il s'agit d'être extrêmement attentif aux versions utilisées. La dernière version courante de production est la 2.0.2 tandis qu'on peut déjà se familiariser avec la version 3.0-alpha2. Cette version nécessite une autre bibliothèque du projet Commons : Commons Codec, qui permet de prendre en compte divers formats (codecs).

Commons HTTP client

Lassé par la pauvreté des interfaces web développées en HTML standard ou effondré par le manque de portabilité des scripts JavaScript ? Le développement de clients lourds invoquant des composants serveurs par requêtes HTTP est devenu une pratique fréquente...

Néanmoins, malgré la richesse du langage Java et de sa couche réseau en particulier, l'invocation d'une page web n'est pas chose simple. Inutile de réinventer la roue et de réinvestir du temps dans des parties techniques, mieux vaut utiliser une bibliothèque éprouvée, telle que Commons HTTP client.

Le projet

Étendre le spectre des fonctionnalités offertes par le paquetage `java.net` tout en vous permettant d'étendre le produit et de l'adapter à vos besoins, tel est le défi relevé par ce projet.

L'URL permettant de télécharger le produit est la suivante :

- http://jakarta.apache.org/commons/http-client/.

Le produit n'exige qu'une seule dépendance : Commons Logging, la bibliothèque d'abstraction du paquetage de gestion des traces applicatives.

Méthodologie d'utilisation du produit

L'invocation d'une page sur un serveur web (par le protocole HTTP) et le traitement de la réponse réclament les étapes suivantes :

- instanciation de la classe centrale du produit `HttpClient` ;
- instanciation d'une méthode d'invocation de la page par le biais d'une implémentation concrète de `HttpMethod` (la classe `GetMethod` par exemple) ;
- exécution de l'invocation (et traitement des erreurs éventuelles) ;
- traitement de la réponse ;
- restitution des ressources (connexion HTTP).

Ce scénario type masque quelque peu la puissance du produit, puisque chaque étape peut être adaptée aux besoins précis de votre application.

Avant de rentrer dans des détails d'adaptation du produit à vos besoins, examinons la traduction en code Java de ce scénario.

Applications exemples

Nous allons aborder ici un exemple simpliste d'utilisation de la bibliothèque, permettant d'introduire les principaux acteurs. Il s'agira ici d'afficher dans la console le contenu de la page d'accueil délivrée par Apache en local sur une machine GNU Linux Debian.

Soit une page conforme à la copie d'écran proposée en figure 2-5.

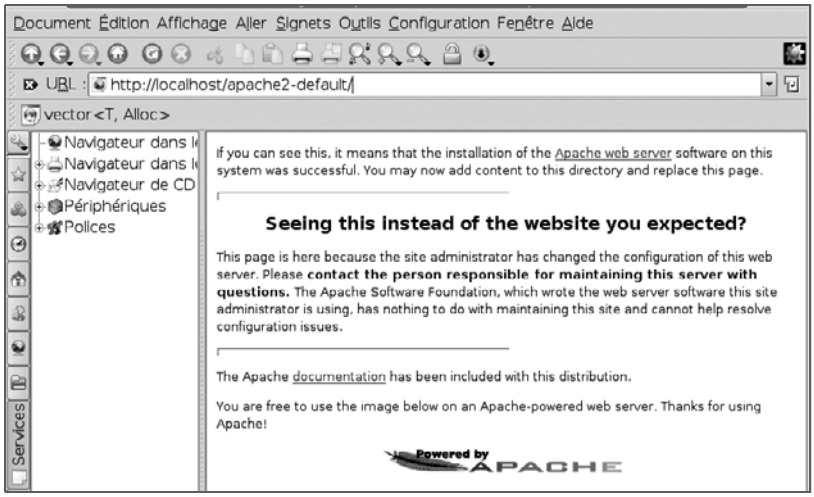


Figure 2–5
La page d'accueil d'Apache 2

Pour cela, on utilisera le code suivant :

```
package commons.http;

import java.io.IOException;

import org.apache.commons.httpclient.HttpClient;
import org.apache.commons.httpclient.HttpException;
import org.apache.commons.httpclient.HttpMethod;
import org.apache.commons.httpclient.HttpStatus;
import org.apache.commons.httpclient.methods.GetMethod;

/**
 * <p>
 * Ouvre une connexion HTTP avec une requete de type GET sur la
 * page d'accueil d'un serveur web (port par défaut 80) en local.
 * Affiche le résultat (page HTML) dans la console...
 * </p>
 * @author jerome
 */

public class SimpleHttpClient {


  public final static String TARGET_URL="http://localhost:80";

  public static void main(String[] args) {
     // client HTTP
     // permet d'obtenir de nombreuses infos
     // et de parametrer le dialogue
     // mais utilise ici sans fioritures
     HttpClient client=new HttpClient();
     HttpMethod method=new GetMethod(TARGET_URL);
```

- Déclaration du paquetage contenant ces exemples.
- Imports nécessaires au programme.
- Définition de la classe.
- Définition d'une constante correspondant à la page demandée.
- Les objets au centre de l'API Commons HttpClient sont :
  - les instances de la classe `HttpClient` ;
  - les instances de la classe `HttpMethodParams`.

  Ces derniers modélisent les diverses façons d'invoquer une page web telles que définies dans la RFC officielle.
  Ici, il s'agit d'un test simple, donc utilisant une requête de type GET.

Lance la récupération de la page.
Attention : le caractere orienté réseau de cette démarche implique la gestion de différente exceptions.
La methode `executeMethod` retourne un status de l'invocation.

L'objet `method` nous permet de récupérer la réponse sous forme d'une chaîne.

Qu'il y ait ou non une exception, on libère la connexion HTTP détenue.

```
      try {
         int status_code=client.executeMethod(method);
         if(status_code!=HttpStatus.SC_OK){
            System.out.println(
                  "probleme lors de la tentative d'invocation");
         }
         // ok cela va bien...
         String result_page=method.getResponseBodyAsString();
         System.out.println(
                "Page recue depuis le serveur web:\n"+ result_page);
      } catch (HttpException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
      finally{
         // libere la connexion
         method.releaseConnection();
      }
   }
}
```

La figure 2-6 représente la sortie capturée dans Eclipse d'une exécution de ce programme.

Figure 2–6
La sortie capturée dans la console Eclipse

Communication avec un serveur web par la méthode POST

Afin de contourner les limitations en terme de volume de données transmises par le *buffer* entre le client (navigateur ou client lourd) et le serveur inhérentes à la méthode `GET`, le protocole HTTP introduit une méthode `POST` utilisant un *buffer* pour le transport des paramètres et de la réponse. La validation de formulaires est l'exemple le plus remarquable d'utilisation de cette facilité du protocole.

Le code suivant démontre comment mettre cette forme de dialogue en œuvre.

```
package commons.http;                                          ◄ Déclaration du paquetage

import java.io.IOException;                                    ◄ Imports nécessaires au programme
import org.apache.commons.httpclient.HttpClient;
import org.apache.commons.httpclient.HttpException;
import org.apache.commons.httpclient.HttpMethod;
import org.apache.commons.httpclient.HttpStatus;
import org.apache.commons.httpclient.methods.PostMethod;

/**
 * <p>
 * invoque une des servlets exemples de Tomcat
 * en utilisant la méthode POST du protocole HTTP
 * </p>
 * @author jerome
 */
public class PostClient {                                      ◄ Définition de la classe

   public final static String TARGET_URL=                      ◄ Constante représentant la page appelée..
                "http://localhost:8080/servlet/RequestInfoExample";
   public static void main(String[] args) {
      HttpClient client=new HttpClient();
      HttpMethod method=new PostMethod(TARGET_URL);            ◄ Le client va ici utiliser une méthode de type
      try {                                                      PostMethod.
                                                                 Tout le reste du code reste identique...
         int status_code=client.executeMethod(method);
         if(status_code!=HttpStatus.SC_OK){
            System.out.println(
                 "probleme lors de la tentative d'invocation");
         }
         String result_page=method.getResponseBodyAsString();
         System.out.println(
                "Page recue depuis le serveur web:\n"+ result_page);
      } catch (HttpException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
      finally{
         method.releaseConnection();
      }
   }
}
```

Pour tester ce code, il suffit de démarrer un serveur Tomcat, puis de lancer le code du client.

À ce moment, on peut obtenir une sortie du type :

```
Page recue depuis le serveur web:
<html><body><head>
<title>Exemple d''information sur la requête</title>
</head>
<body bgcolor="white">
<a href="../reqinfo.html">
<img src="../images/code.gif" height=24 width=24 align=right border=0
alt="view code"></a>
<a href="../index.html">
<img src="../images/return.gif" height=24 width=24 align=right
border=0 alt="return"></a>
<h3>Exemple d''information sur la requête</h3>
<table border=0><tr><td>Méthode:</td>
<td>POST</td></tr>
<tr><td>URI de requête:</td>
<td>/servlet/RequestInfoExample</td></tr>
<tr><td>Protocole:</td>
<td>HTTP/1.1</td></tr>
<tr><td>Info de chemin:</td>
<td>null</td></tr>
<tr><td>Adresse distante:</td>
<td>127.0.0.1</td></tr></table>
```

Simple n'est-ce pas ? Pas de temps perdu à gérer les subtilités du protocole HTTP, simplement du code 100 % utile...

Adaptation du produit à vos besoins

Poussons un peu plus avant nos investigations et examinons quelques fonctionnalités avancées offertes par le produit...

L'exemple précédent ne prenait que peu en compte les problèmes inhérents à l'utilisation d'un réseau et passait sous silence l'éventuel échec des connexions vers le serveur HTTP distant.

Que se passe-t-il maintenant si l'on cherche à atteindre un site distant et non local et que la connexion échoue ? La bibliothèque renverra-t-elle une exception ou un code d'erreur lors de la première tentative infructueuse ?

Avec la version 3.0 de la bibliothèque, cette réponse est définitivement non, car l'aspect communication réseau de la bibliothèque fait apparaître une classe associée à la méthode d'accès à la page (l'instance de la classe `HttpMethod`) : le `MethodHanler`.

Ceci nous permet d'adopter la politique de notre choix, lors de la tentative de connexion vers un site.

Cette configuration se fera en précisant sur l'objet `HttpMethod` un paramètre de type `HttpMethodParams.RETRY_HANDLER`. L'implémentation exacte de cet

objet est à choisir entre une implémentation par défaut (fournie par la bibliothèque) et une implémentation plus personnelle...

Ceci n'est qu'une des nombreuses possibilités d'extension envisageables avec cette bibliothèque. La documentation de l'API permettra au lecteur intéressé de connaître la liste exhaustive de ces points d'extension.

Commons Collections et Commons Primitives

Les structures de données sont les véritables fondations de tout projet. Ces deux projets s'intéressent à compenser des carences laissées par le paquetage standard `java.util` malgré l'apparition depuis le JDK 1.2 des fameuses Collections.

Le paquetage Commons Collection vous fournit diverses collections permettant aux développeurs de trouver des réponses pragmatiques à divers besoins récurrents. Ainsi, vous cherchez un cache pour vos objets, allez-vous devoir le développer vous-même ? Ce cache se doit d'utiliser un algorithme basé sur la fréquence d'utilisation des divers objets, mais le paquetage `java.util` ne fournit rien de tel en standard.

Votre application fait grand usage d'entiers, de flottants ? Vous avez constaté l'impossibilité d'utiliser des collections Java pour stocker vos objets et vous avez décidé de contourner ceci en instanciant les diverses classes *wrappers* du paquetage `java.lang` (`Integer`, `Float` etc..). Quel dommage et quelle perte de ressources (mémoires et CPU) ! Pas de panique, le projet Commons Primitives permet de stocker vos types primitifs dans des listes, fournit la possibilité d'obtenir des itérateurs.

PRÉCISION Caches applicatifs

Le cache ici présenté est le type le plus simple possible. Pour des besoins plus ambitieux, il faudra regarder du côté de divers projets libres parmi lesquels on peut citer ehcache.

Création d'un cache d'objets

Dans cet exemple, nous allons simuler la création d'un cache d'objets, c'est à dire un conteneur d'objets de taille maximale fixée (ici 10) devant décider des objets à éliminer en étudiant la fréquence d'utilisation de ceux-ci. Nous simulons ainsi le fonctionnement d'un cache au sein d'un serveur d'applications.

```
package commons.collections;
```

Déclaration du paquetage.

```
import java.util.Iterator;
import java.util.Map;
import org.apache.commons.collections.LRUMap;
```

Imports nécessaires.

```
/**
 * <p>
 * Mise en evidence de la puissance des implementations proposees
 * par le projet Commons Collections.
 * Programme de test utilisant une implementation
 * de l'interface java.util.Map propre à ce produit.
 * Il s'agit ici d'experimenter la LRUMap permettant de creer des
 * caches tels que ceux utilises dans JBOSS et autres serveurs
 * d'applications.
 * @author jerome
 *
 */
```

Déclaration de la classe.

Cette méthode est un petit utilitaire permettant de connaître le contenu d'une *map*.

```
public class CollectionsExemple {
// affiche le contenu d'une Map
   // methode utititaire
   static void displayMap(final Map aMap){
      for (Iterator iter=aMap.entrySet().iterator();iter.hasNext();){
         Map.Entry entry=(Map.Entry)iter.next();
         System.out.println("cle = "+ entry.getKey() +
                            " valeur =" + entry.getValue());
      }
   }
```

Notre exemple va débuter en remplissant une structure avec dix éléments. Notre *map* est implémentée en utilisant une des classes du projet Commons Collection: la LRUMap.
Notre *map* a pour contrainte (d'après le constructeur) d'être limitée en taille à 10 éléments.
Nous allons expérimenter ceci en provoquant la sortie du cache d'un des 10 objets. Pour cela, nous devrons solliciter certains objets (les 9 premiers), ainsi le cache saura qu'un des objets a moins de chance d'être désiré et l'élira donc à l'éviction.

```
public static void main(String[] args) {
     String[] KEYS={"coucou","hello","tutu","tata","titi",
                    "monde","world","hallo","welt","kiki"};
     // obtenir une reference sur une Map implementee sous forme
     // d'une LRUMap. Une fois pleine (ici 10 elements car precisee
     // dans le constructeur) cette structure va evacuer des objets
     // en scrutant les dates de derniere utilisation tres utile pour
     // batir des caches d'objet de taille fixe comme dans le cas de
     // serveurs d'applications
     Map map = new LRUMap(10);
     // remplit la map
     for(int i=0;i<10;i++){
        map.put(KEYS[i],new Integer(i));
     }
     // accede aux objets de la map
     // tous sauf le dernier element (kiki)
     for(int i=0;i<9;i++){
        map.get(KEYS[i]);
     }
     displayMap(map);
     // on ajoute un element mais la taille maximale est atteinte
     // il faudra supprimer l'element utilise il y a le plus de
     // temps...
     map.put("nouveau","salut Java");
     // on affiche la nouvelle Map
     displayMap(map);
   }//main()
}
```

Utilisation des Commons Primitives

L'exemple de code suivant illustre une utilisation basique de ce projet afin de stocker une large collection d'entiers avant de réaliser la somme de ces nombres de manière à retrouver la formule de Gauss donnant la somme des n premiers entiers naturels : n*n(+1)/2.

Déclaration du paquetage hôte

```
package commons.primitives;
```

Imports nécessaires au programme.

```
import org.apache.commons.collections.primitives.ArrayIntList;
import org.apache.commons.collections.primitives.IntIterator;
import org.apache.commons.collections.primitives.IntList;
```

```
/**
 * <p>
 * Utilisation du paquetage Commons Primitives pour stocker
 * une liste d'entiers. Cette liste d'entiers (1,2,3,....,n)
 * va ensuite être utilisée de manière à réaliser la somme
 * des entiers naturels on va retrouver le fameux résultat
 * du a Gauss : n*(n+1)/2
 * </p>
 * @author jerome@javaxpert.com
 *
 */
public class SommeEntiers {
   //  fix the number of elements
   private final static int MAX_ELEMENTS=500000;

public static void main(String[] args) {
     int sum=0;
     // utilisation d'une structure proche d'une liste
     // a la sauce java.util.List
     IntList int_list= new ArrayIntList(MAX_ELEMENTS);
     for(int i=0;i<MAX_ELEMENTS;i++){
        int_list.add(i);
     }
     // Utilisation d'un iterateur proche du java.util.Iterator
     for(IntIterator iter=int_list.iterator();iter.hasNext();){
        sum+=iter.next();
     }
     System.out.println("La somme des N premiers entiers où N = " +
                        MAX_ELEMENTS + "est =" + sum +
                        "\nLa formule predit =" +
                        (MAX_ELEMENTS*(MAX_ELEMENTS+1)/2));
   } // main()
}
```

Déclaration de la classe.

Définition d'une constante.

Définition de la structure (`ArrayIntList`)

Remplissage de la structure.

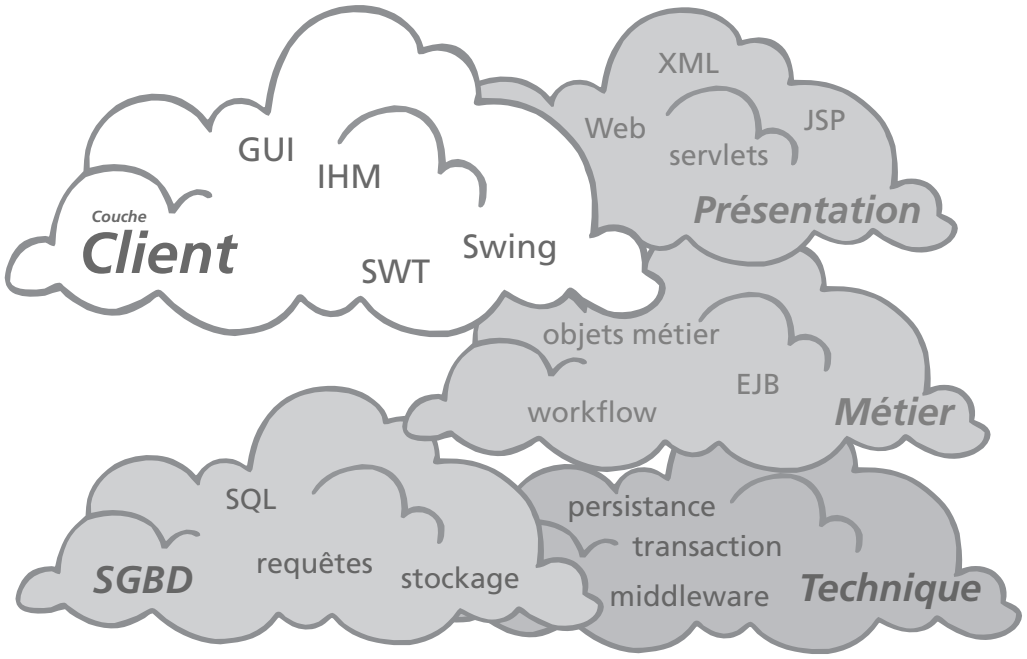
Utilisation de la structure de données (contenant les entiers) par le biais d'un itérateur.

Ces petits exemples n'ont pour seule ambition que vous faire appréhender l'éventail de possibilités offertes par le projet Commons. À travers les projets présentés, il faut bien retenir le spectre couvert par ce projet que l'on pourrait qualifier de méta-projet. Il couvre des besoins liés à des domaines aussi variés que les structures de données, la programmation réseau, la manipulation de Beans et de fichiers XML, la gestion de traces et bien d'autres. Une véritable aubaine pour tout développeur Java.

En résumé…

Les outils centraux de l'équipe de développement sont dorénavant posés. Ant va nous permettre d'orchestrer les procédures de contrôle de la qualité du code source, de génération des versions et le déploiement de l'application. Quant à CVS, il est là pour aider à produire et gérer des versions. Enfin, le problème épineux du choix d'une bibliothèque a été résolu.

chapitre 3


Couche
Client
GUI
IHM
SWT
Swing
XML
Web
servlets
JSP
Présentation
objets métier
EJB
workflow
Métier
SQL
SGBD
requêtes
stockage
persistance
transaction
middleware
Technique

Interfaces graphiques pour la couche cliente

L'interface graphique d'une application est la partie « visible » pour un utilisateur. La négliger annule tout le travail fait en amont. L'ergonomie de notre application va donc conditionner son utilisation. Réactivité, présentation, sobriété, sont autant de facteurs à prendre en compte pour ne pas passer à côté de nos objectifs.

SOMMAIRE

MOTS-CLÉS

- GUI/IHM
- Eclipse/SWT
- Widgets
- Pattern Décorateur
- Validation des saisies/expressions régulières
- JUnit

Choix de la bibliothèque SWT

Tous les critères précédemment évoqués pourraient être utilisés afin d'apporter un élément de plus dans la bataille de l'API graphique opposant, à ma gauche, Swing le poids lourd de Sun, contre, à ma droite, SWT le challenger. Une recherche sur Internet vous permettrait de trouver un grand nombre d'articles tentant d'opposer les deux bibliothèques. Cette approche est un peu stérile car, d'un point de vue pragmatique, il est préférable d'avoir du choix…

Mais ce choix de SWT trouve une bien meilleure justification si, comme l'équipe de BlueWeb, on voit en Eclipse bien plus qu'un simple IDE car, outre cet aspect, Eclipse se révèle être une plate-forme logicielle fabuleuse :

- riche en fonctionnalités ;
- facilement extensible ;
- aisément configurable.

En effet, Eclipse peut être perçu comme un réceptacle universel faisant tourner des applications n'ayant que peu ou pas de rapports avec Java. Ainsi, un plug-in Eclipse dote votre environnement d'un IDE pour C/C++, un autre vous renvoie dans le passé avec un environnement dédié au Cobol, d'autres enfin permettent de gérer Tomcat ou JBoss. Certains éditeurs comme Rational ont déjà franchi le pas et proposent dorénavant leurs nouveaux produits sous forme de plug-ins. XDE est ainsi le successeur de Rational Rose. Le plus étonnant dans le mouvement autour de ce produit est l'organisation sous forme d'un consortium ouvert, ce qui est vraiment déroutant puisque l'on retrouve des sociétés aux intérêts a priori concurrents réunies autour de la table de discussion…

Se pourrait-il que tous ces éditeurs voient en Eclipse le réceptacle tant attendu ? Comment échapperaient-ils aux attraits d'un produit déjà doté de ce qu'il y a de mieux sur le marché, avec par exemple un système d'aide en ligne contextuelle se basant sur Lucene et Tomcat ?

B.A.-BA La notion de placement automatique (layout)

Il est indispensable d'introduire la notion de *layouts* dans ce chapitre dédié à l'interface graphique. C'est une notion clé dans le développement d'interfaces graphiques. Il s'agit d'utiliser un algorithme de placement des composants (layout) permettant de s'adapter dynamiquement à des changements de contexte (redimensionnement de l'écran, ajout de composants) plutôt que de spécifier explicitement la localisation de chaque bouton ou fenêtre (par les coordonnées X/Y). On peut imaginer un grand nombre d'algorithmes ayant leurs contraintes propres (contraintes cartographiques (Est, Ouest, Nord, Sud), pourcentage, etc.). Cette notion de layouts a donc le principal mérite d'éviter les problèmes fâcheux liés aux tailles d'écran (boutons invisibles en passant d'un moniteur 17 pouces à un 15 pouces) et donc d'éviter de lier son interface à une résolution d'affichage. En revanche, les layouts induisent un développement plus complexe (pas de solution satisfaisante parmi les rares outils proposant des aides graphiques à la construction d'IHM), ainsi qu'un coût en matière de performances (puisqu'il y a calcul là où classiquement on amenait des coordonnées figées). Il est unanimement reconnu que dans le monde Java, toute interface graphique doit utiliser ce type de stratégie. Il faut mettre en garde le lecteur contre certains outils proposant des layouts permettant de positionner des composants « en dur ».

IDE
Integrated Development Environment

Un environnement de développement intégré désigne un outil regroupant les fonctions d'édition, de compilation et de débogage. Ainsi, Visual C++, Borland C++, JBuilder ou JCreator peuvent être cités parmi la longue liste des IDE disponibles sur le marché.

B.A.-BA WSAD (WebSphere Studio Application Developer)

WSAD est une surcouche commerciale d'Eclipse développée par IBM et remplaçant l'ancien environnement de développement de cette société : Visual Age.

Fondation Eclipse

La fondation Eclipse regroupe une cinquantaine de sociétés (Toshiba, IBM, Borland, Rational, etc.).

OUTIL Lucene

Lucene est un autre projet de l'Apache Group. Il s'agit d'un moteur de recherche en texte libre extrêmement performant, facilement adaptable à vos besoins puisqu'il est codé en Java, manipulable aisément dans des contextes aussi différents qu'une application autonome ou un contexte J2EE.

Malgré tout, SWT est encore un projet jeune. C'est pourquoi il faut quand même préciser qu'il reste des bogues ou des maladresses de cohérence dans la conception du produit. Par exemple, on peut citer le fantôme dans le cas de l'utilisation d'une table SWT, des incohérences dans les hiérarchies de classes (comme la classe `KeyEvent` qui n'hérite pas de `Event`). Cependant, le potentiel est énorme et l'existant plus que séduisant…

IDÉE REÇUE La création d'interfaces graphiques est une chose simple

Contrairement aux idées reçues, la création d'interfaces graphiques est loin d'être une tâche simple. Pourquoi un tel constat ? L'ergonomie est une science qui n'est pas enseignée dans les formations d'ingénieurs, les architectures modernes impliquent un haut niveau de flexibilité et le contexte de développement (utilisation de layouts, client/serveur) est difficile. De plus, l'exigence des clients se porte toujours plus sur l'aspect d'une application que sur son fonctionnement réel et son adéquation avec le cahier des charges. Il est donc bon de bien se persuader que tous ces faits contribuent à rendre ce domaine complexe et donc coûteux. Ainsi, l'utilisation de layouts dans du code client est indispensable (voir encadré sur ce sujet), mais complique terriblement le développement puisque, à l'heure actuelle, aucun outil de développement Java ne permet de produire un code réellement satisfaisant. On est loin des environnements comme Delphi ou Visual Basic, qui permettent de développer des IHM quasiment sans codage. De plus, il est préjudiciable d'avoir à reprendre éternellement (avec plus ou moins de succès) la problématique de représentation de nos objets métier sous forme de composants graphiques. Certaines initiatives récentes, comme le projet Gatonero, permettent de créer des interfaces graphiques utilisables, configurables et respectant des normes ergonomiques satisfaisantes, et ce sans intrusion dans votre logique métier (pas de modification du code). Ce type d'approche permet de réduire les coûts de développement, d'éviter les bourdes ergonomiques classiques (peu d'entreprises disposent d'ergonomes qualifiés) et de réaliser des applicatifs à l'aspect cohérent (donc non dépendant des fantaisies du développeur). Un projet à surveiller…

OUTIL Eclipse Visual Editor

Eclipse Visual Editor permet de créer des interfaces graphiques AWT/Swing et SWT. Son ambition, au-delà de la génération d'interfaces pour le langage Java, est de proposer la création d'interfaces pour les langages C/C++ ainsi que des ensembles de widgets particuliers. Il est basé sur le Visual Editor de WebSphere Studio Application Developer 5.x.

Retour à l'application de gestion des signets

Représentation graphique envisagée : l'arbre

À la manière d'un explorateur Windows, notre base de données contenant les signets saisis par le personnel de BlueWeb peut être représentée comme un arbre du type :

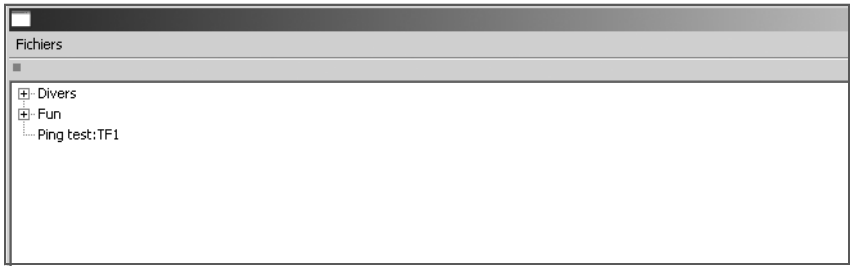


Figure 3–1 Aperçu d'un arbre réalisé avec SWT

B.A.-BA Widget

C'est un terme provenant de la contraction des deux mots *window* et *gadget*. Un widget est le nom d'un contrôle ou composant graphique.

Figure 3–2 Voici un arbre plus fourni (avec des titres de qualité)

Étant données les spécifications consignées précédemment, chaque nœud (équivalent d'un dossier dans l'explorateur de fichiers Microsoft) est appelé un thème, tandis qu'une feuille représente un signet. Chaque thème peut avoir de 0 à n sous-thèmes, et de 0 à n signets.

La capture d'écran de la figure 3–2 représente un cas de figure plus réaliste…

Choix du widget : SWT ou JFace

JFace est une surcouche au-dessus de SWT mettant à notre disposition des composants plus abstraits, plus puissants et nous permettant de réaliser un code à l'aspect plus élégant. La question du choix entre la version brute de fonderie et celle issue de JFace n'en est pas réellement une, car quasiment tous les composants complexes (arbres, tables, etc.) devraient être choisis dans JFace plutôt que directement dans la SWT. En fait, JFace est un intermédiaire entre SWT et vos objets (du modèle vous intéressant). Il permet donc de travailler avec une abstraction plus grande.

Le TreeViewer JFace

JFace nous fournit par le biais de ce composant un moyen de disposer d'une large gamme de fonctionnalités intégrées au sein d'une architecture robuste et élégante. En effet, le TreeViewer nous permet de gérer des **objets métier** (adaptés à notre modèle) qui nous offrent un découplage souhaitable (pour favoriser évolution/maintenance) entre l'affichage (labels, images) et les données (intégrité, relations...).

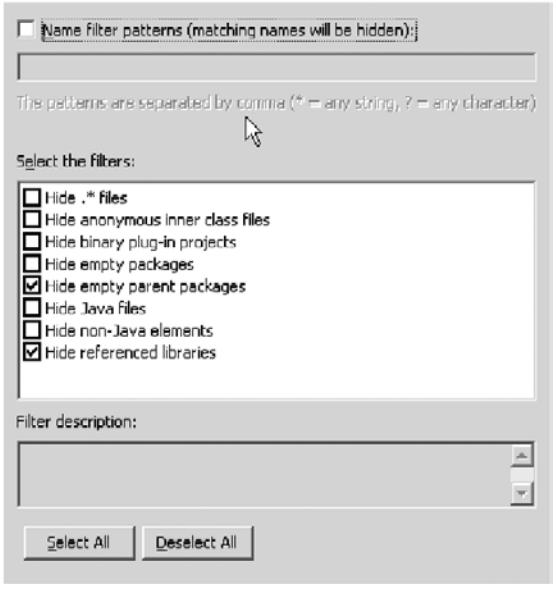


Figure 3–3
Les filtres en action dans Eclipse

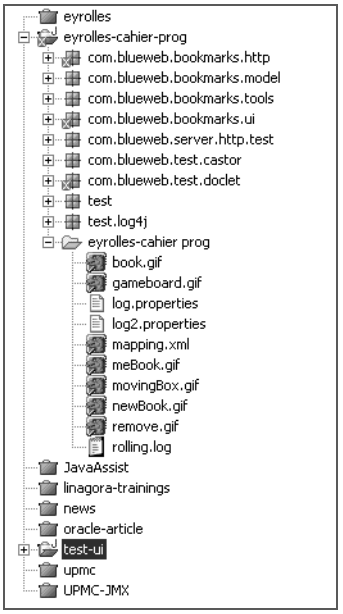


Figure 3–4 Un projet Eclipse sans utilisation de filtres

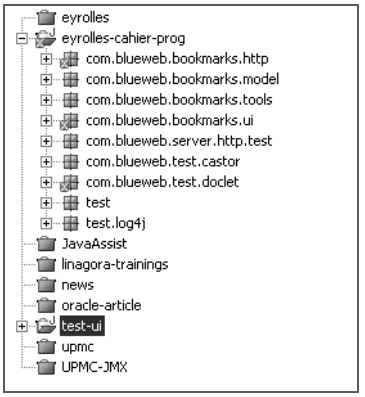


Figure 3–5 Le même projet après application d'un filtre d'affichage

C'est une architecture classique en vérité puisque nous retrouvons là les éléments caractérisant un design pattern répandu depuis les premières interfaces graphiques en SmallTalk : Modèle Vue Contrôleur ou MVC.

Cet aspect classique est à rapprocher du modèle dit UI (User Interface) adopté dans Swing, qui est finalement assez proche par la conception générale.

Ceci a pour effet de permettre des fonctionnalités de très haut niveau telles que les filtres. Dans une interface JFace, on peut cliquer sur un filtre pour ne sélectionner que les objets de tel ou tel type (figure 3–3). Ce composant très puissant est utilisé massivement dans Eclipse lui-même, comme le montrent les figures suivantes.

Ce menu est obtenu par clic sur la flèche visible en haut à droite de l'image (à côté de la croix de fermeture). Ainsi, la figure 3–4 représente un projet dont tous les fichiers sont affichés. La figure 3–5 présente le même projet où l'on n'a sélectionné que les fichiers .java.

Une autre fonctionnalité avancée est la possibilité de trier le contenu de notre arbre suivant divers critères ; c'est le travail de Viewers Sorters.

B.A.-BA MVC et modèle dit UI

Le modèle UI n'est qu'une variante du MVC présentant la particularité de regrouper en un seul et même objet deux des entités définies par le MVC.

Les acteurs

Voici venu le moment de détailler les différents intervenants nécessaires au codage de l'arbre contenant nos objets métier (signets et thèmes) :

- le `ContentProvider`, qui permet de fournir un contenu au widget (peuple l'arbre avec nos objets métier) ;
- le `LabelProvider`, qui permet d'associer images et textes aux différents objets de l'arbre ;
- le `TreeViewer`, c'est-à-dire le widget en lui-même ;
- les objets métier (signets,thèmes).

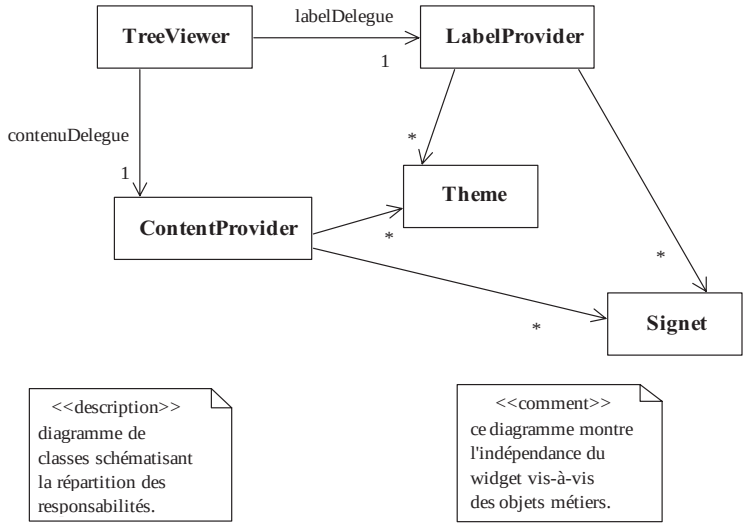


Figure 3–6
Diagramme de classe UML schématisant les intervenants dans notre application

B.A.-BA Délégation

Ce mot désigne, dans notre contexte objet, l'utilisation d'une classe tierce pour implémenter une opération plutôt que la création d'une classe dérivée. C'est un vieux débat entre partisans/adversaires de l'héritage/délégation. Comme souvent, l'approche pragmatique vise à essayer d'utiliser les deux techniques à bon escient sans proscrire l'une ou l'autre.

ASTUCE Analogie avec Swing

Ces notions sont proches de celles utilisées dans Swing avec l'interface `Renderer` et l'interface `Model`.

Ce diagramme UML de la figure 3–6 permet d'avoir une vue d'ensemble représentative du mécanisme de délégation mis en œuvre dans le `TreeViewer`. L'arbre est alimenté en données par le biais du `ContentProvider`, les textes et icônes directement liés aux objets métier sont issus du travail du `LabelProvider`. On pourrait donc, sans rien changer à notre `TreeViewer`, passer à une alimentation différente de l'arbre avec des objets métier différents ; la seule modification à apporter serait dans l'initialisation de l'objet `TreeViewer` (et non dans son code), via l'utilisation de deux implémentations différentes des interfaces `ItreeContentProvider` et `ILabelProvider`.

De l'utilité du découplage vues/données

Le codage de l'arbre abordé précédemment montre clairement les bénéfices d'un découplage entre données et affichage. En effet, les problèmes difficiles tels que le rafraîchissement de vos vues sont gérés automatiquement par JFace (comme le ferait Swing). De plus, votre code se trouve découpé en responsabilités strictes facilitant ainsi des changements ultérieurs.

Les filtres SWT évoqués précédemment sont un bel exemple du design pattern Décorateur (Decorator dans l'ouvrage de référence). Ce pattern permet de « décorer », c'est-à-dire changer l'apparence, mais aussi les fonctionnalités (comme on le verra dans l'exemple suivant) d'un objet sans être contraint de multiplier le nombre de classes dérivées. Ceci a pour but de conserver le contrôle d'une famille d'objets, car la maintenance du code d'une centaine de classes est beaucoup plus difficile que le même travail appliqué sur une dizaine de classes.

Ce Pattern a été conçu dans une optique d'utilisation dans un contexte graphique, dans le cas de widgets pouvant être personnalisés avec l'ajout d'images, de bords arrondis et autres. Cependant, il peut trouver des applications intéressantes dans des contextes bien différents. Ainsi, un exemple fameux d'utilisation de ce pattern se trouve dans le paquetage standard Java `java.io`, avec les classes `Stream/Reader/Writer`. En effet, ces classes sont conçues pour coopérer et peuvent être emboîtées telles des poupées russes, offrant ainsi une gamme de fonctionnalités bien supérieures au simple nombre des classes.

Le diagramme de la figure 3–7 montre quelques classes filles de la classe `Writer` et va nous permettre d'introduire un exemple de code très simple illustrant la puissance du pattern Décorateur.

```
// imports et contexte omis
Writer writer=new BufferedWriter(new FileWriter("/tmp/out.txt"));
// utilisation / fermeture etc.
```

Ici, on construit un `Writer`, permettant d'écrire vers un fichier de sortie. Ce `Writer` est « bufférisé », donc plus rapide (moins d'accès au disque). On voit alors que l'on obtient une « décoration » sans pour autant avoir une classe `BufferedFileWriter`. De même, si l'on voulait écrire vers une `String`, on utiliserait un `StringWriter` en lieu et place du `FileWriter`.

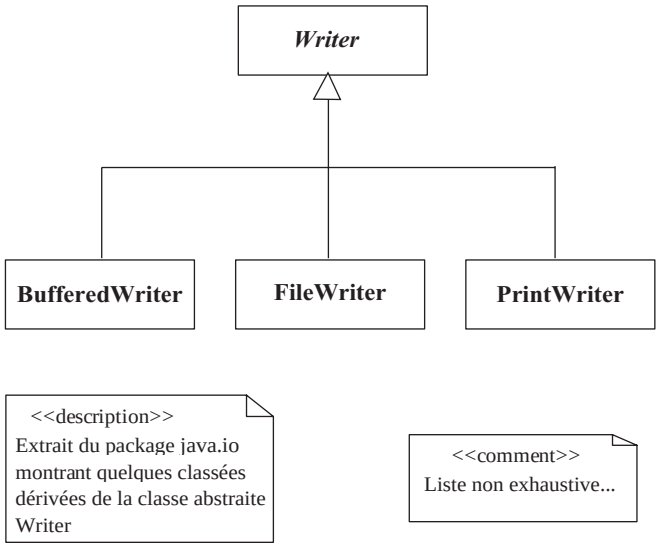


Figure 3–7
Extrait de la hiérarchie de Sun (paquetage java.io)

Ce `Writer` pourrait être « bufférisé » en reprenant la même construction que précédemment sans qu'il y ait non plus de classe `BufferedStringWriter`. Bien entendu, on peut emboîter différents types de `Writer` (sur plus de deux niveaux) pour obtenir ainsi une grande variété de décorations à partir d'un nombre réduit de classes initiales (filles de la classe `Writer`).

Validation des saisies

Pourquoi valider des saisies ?

Dans le monde du client/serveur, encore plus que dans tous les autres types d'applications, les vérifications des saisies utilisateur tiennent un rôle clé, puisqu'il faut imaginer que toute requête cliente fausse induira un appel réseau, une vérification impliquant une exception métier (saisie invalide), puis en retour de la requête une réponse indiquant la saisie invalide. Bref, l'utilisateur peut avoir attendu longtemps (si le réseau est lent dans le cas d'une connexion modem par exemple, si le serveur est saturé, etc.) pour un résultat potentiellement prédictible puisque, par exemple, si l'on demande de saisir un nombre entier, saisir un flottant est une saisie invalide. Donc, doter son application de tests de saisie simples ne peut que faire gagner votre application en stabilité et en performances puisque si l'on sollicite moins souvent le serveur, celui-ci ne peut que réagir plus rapidement. En effet, *on fait diminuer le trafic sur notre réseau et on préserve la CPU du serveur.*

ARCHITECTURE **Où placer les validations ?**

Utiliser des validations des saisies clientes ne doit pas vous empêcher de placer des contrôles de cohérence dans votre logique métier. Ces contrôles ne peuvent être qu'identiques à des tests effectués sur la partie cliente ou beaucoup plus complets suivant les cas. Il s'agit surtout de s'assurer que les vérifications effectuées sur le poste client ne requièrent pas une trop grande connaissance de la partie métier, afin de réduire le couplage et de favoriser le développement par composants. En toute logique, vérifier la bonne saisie d'une valeur entière, d'un numéro de carte bleue ou encore d'une adresse de courrier électronique ne demande pas réellement de connaissance sur l'utilisation de cette valeur côté serveur. Un autre exemple pourrait être la vérification de la saisie d'un numéro de sécurité sociale français : sa composition est publique, quelle que soit son utilisation dans votre logique métier, et ne pas le vérifier lors des saisies est une faille dans la logique applicative (c'est aussi rassurant pour l'utilisateur de se sentir guidé).

Dans le cas de l'applicatif BlueWeb, il s'agira de vérifier que les URL saisies par l'utilisateur ne sont pas erronées, tout en précisant bien le contexte de notre vérification :

- limitée à du *pattern/matching* côté client (c'est-à-dire à la conformité de la saisie par rapport à une expression régulière) ;
- donc pas de requête réseau type requête DNS ou HTTP ;
- dans un premier temps, le pattern utilisé ne sera qu'un garde-fou, puisqu'il ne s'attachera pas à respecter strictement les RFC régissant les différents protocoles (HTTP, FTP, etc.).

En général, une adresse valide (URL) sera du type :

```
(Protocole://)?(unechainevalide.)+(nom_domaine)/(URI)
```

avec :

- `Protocole` : `http`, `https`, `ftp` (`gopher` et `wais` ne sont plus guère utilisés) ;
- `unechainevalide` : tous les caractères `a-z`, `A-Z`, `0-9`, etc. ;
- `nom_domaine` : `.com`, `.fr`, `.uk`, `.org`, `.net`, etc. ;
- `URI` : un chemin du type `titi/toto/img.html`

On utilisera une syntaxe à la mode EBNF.

Comment vérifier une saisie ?

Les structures manipulées dans vos interfaces (entiers, flottants, codes, adresses, etc.) peuvent en majorité être décrites de manière générique par ce que l'on appelle une expression régulière. La simple utilisation d'une bibliothèque d'expressions régulières vous permet de doter votre application de fonctionnalités très avancées concernant la manipulation de chaînes de caractères.

L'utilisation d'ORO se fait selon les étapes suivantes :

- obtention d'un compilateur d'expressions régulières (permet entre autre de vérifier la bonne syntaxe de notre expression régulière) ;
- obtention d'un pattern à partir du compilateur et de notre expression régulière (correspondant à la vérification d'une URL dans l'exemple) ;
- vérification de la chaîne saisie contre le pattern via le `PatternMatcher`.

DÉBAT Quelle bibliothèque choisir ?

Le grand nombre de bibliothèques dédiées à cette tâche nous oblige à nous poser cette question. Alors, entre Jakarta ORO, Jakarta Regex, le portage GNU, la bibliothèque incluse dans le JDK 1.4 (`java.util.regex`), laquelle faut-il choisir ? C'est un choix sûrement arbitraire, mais les qualités d'ORO font qu'il est difficile de migrer vers une autre bibliothèque. Quant à la nouvelle bibliothèque du JDK 1.4, il est regrettable que Sun ait refusé d'intégrer ORO, car ce paquetage est moins puissant et plus restrictif que ORO.

B.A.-BA RFC (Request For Comment)

Ce sigle peut être traduit par « appel à commentaires » et date des premiers temps d'Internet, une époque où tous les protocoles basiques (mail, FTP, TCP/IP) étaient élaborés de manière « artisanale » entre chercheurs passionnés. Malheureusement, cette époque est révolue, même si les RFC demeurent…

B.A.-BA EBNF

Pour « forme de Backus-Naur étendue » : c'est la syntaxe habituellement utilisée pour tout ce qui concerne l'analyse lexicale et la création de grammaires. C'est une notation fortement liée à la théorie de la compilation, ce qui dépasse notablement le cadre de cet ouvrage…

B.A.-BA Expressions régulières

Ce terme introduit en informatique par le monde Unix, désigne la façon de vérifier la cohérence d'une chaîne avec un motif défini à l'avance (pattern). Des langages comme Perl et des outils comme sed, awk ou encore Emacs utilisent ces techniques pour offrir de stupéfiantes fonctionnalités de recherche/substitution.

OUTIL ORO

La bibliothèque ORO est un sous-projet du projet Apache offert par son auteur originel (Daniel Savarese) à la communauté. Cette bibliothèque est rapide, performante et puissante (syntaxe AWK, Perl 5).

▸ http://jakarta.apache.org/oro

Ceci peut-être résumé par le diagramme de séquence UML suivant :

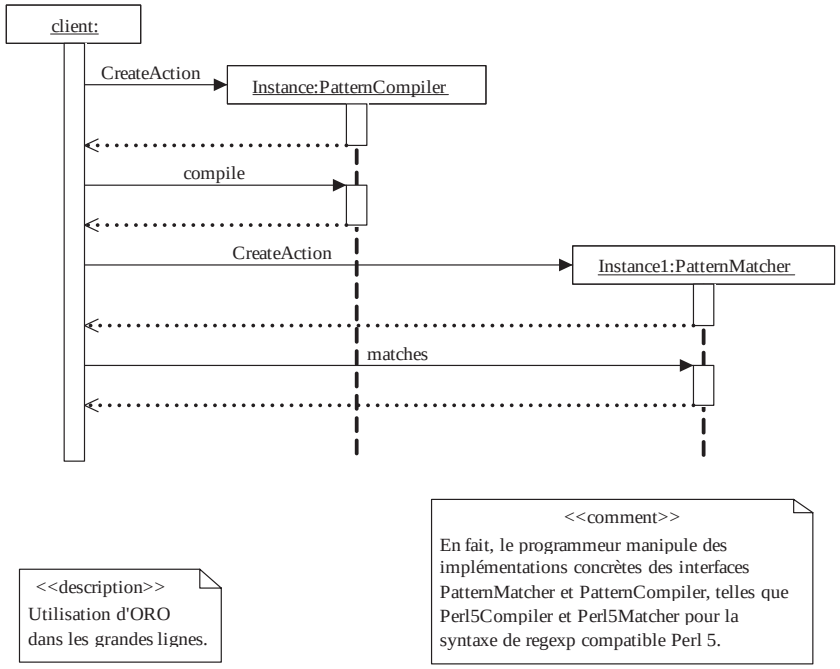


Figure 3–8 Diagramme de séquences UML schématisant l'utilisation d'ORO

Ce diagramme simplifie quelque peu les choses, puisqu'il fait apparaître `PatternMatcher` et `PatternCompiler` en tant que classes, alors qu'elles sont proposées en tant qu'interfaces dans le paquetage Oro.

Le paquetage principal de ce produit est : `org.apache.oro.text.regex`. On peut traduire ceci par le code suivant :

B.A.-BA **Interfaces**

Une interface en Java est faite pour permettre la prise en charge de l'héritage multiple par le langage, mais cet héritage multiple est un héritage de services (contrats, intentions) et non de code (classes). Ainsi, il est possible d'implémenter plusieurs interfaces mais on ne peut étendre qu'une seule classe (par défaut `java.lang.Object`). Cette notion d'interface permet d'imaginer de nombreuses utilisations (remplace élégamment les pointeurs de fonctions chers aux adeptes du C/C++ par exemple) et a le mérite de mettre en évidence la notion de services publiés par un objet tout en se préservant de l'implémentation. C'est une notion clé du langage qu'il faut maîtriser afin de tirer pleinement parti des nombreuses possibilités de Java.

Classe de test utilisant le paquetage ORO pour la validation des saisies

Un petit programme de test des expressions régulières manipulées avec ORO.

```
package test;
import org.apache.oro.text.regex.Pattern;
import org.apache.oro.text.regex.Perl5Compiler;
import org.apache.oro.text.regex.Perl5Matcher;
/**
* @author J.MOLIERE - 2003/01
* <p>
* une micro classe montrant comment utiliser ORO de manière
* à valider des cibles (URLS) sur une expression régulière
* espérée générique... Cette expression régulière sera créée
* suivant une syntaxe du type Perl5.
* En fait, elle est un peu trop simple pour prévenir
* toutes les saisies erronées.
* </p>
*/
```

```
public class UrlTester {
   // les URLs a tester
   private static String[] cibles = {"http://amazon.com","https://
waba.kiki.org","ftp://a.b.com","oura/titi","http://oura/fr"};
   // la regexp magique!!!
```

On peut remarquer que la syntaxe d'une expression régulière est quelque peu rugueuse, du moins au début…

```
private final static String REGEXP = "(^(http|ftp|https|file)://
)?(\\S+\\.)+(net|org|fr|uk|com|de)(/(\\S)+)?";
   /**
    * la méthode main
    */
   public static void main(String[] args) throws Exception{
      // on utilisera une syntaxe compatible Perl5
      // d'où cette implémentation du PatternCompiler
```

ORO acceptant plusieurs dialectes, il faut préciser le type de compilateur à utiliser. Ici, on choisit un style Perl 5.

```
      Perl5Compiler compiler = new Perl5Compiler();
```

ORO nous fournit un objet `Pattern` moyennant la fourniture de notre expression régulière.

```
      // obtient le Pattern grâce au compilateur
      Pattern pattern = compiler.compile(REGEXP);
```

Cet objet permet ensuite de comparer nos données au masque générique accepté (l'expression régulière).

```
      Perl5Matcher matcher = new Perl5Matcher();
      // itère sur les cibles
      for(int i = 0;i < cibles.length;i++){
         boolean result = matcher.matches(cibles[i],pattern);
         System.err.println(cibles[i] + " matchee ? = "+ result);
      }
   }
}
```

Soit une sortie sur la console (après exécution de `java test.UrlTester`) :

```
http://amazon.com matchee ? = true
https://waba.kiki.org matchee ? = true
ftp://a.b.com matchee ? = true
oura/titi matchee ? = false
http://oura/fr matchee ? = false
```

Une seule pseudo-URL n'est pas reconnue comme valide, ce qui est normal étant donné la syntaxe de notre expression régulière et l'URL testée. En packageant un peu notre code, on peut obtenir l'extrait suivant :

Nouvelle version de la classe de test des saisies d'URL

```
package com.blueweb.bookmarks.tools;
import org.apache.oro.text.regex.MalformedPatternException;
import org.apache.oro.text.regex.Pattern;
import org.apache.oro.text.regex.Perl5Compiler;
import org.apache.oro.text.regex.Perl5Matcher;
/**
*<p>
* Ce composant permet de tester la validité d'une URL.
* Bien sûr, la validité des tests qu'il effectue dépend
* essentiellement de l'expression régulière, donc plus celle-ci
* sera complète, meilleures seront ses réponses.
```

```
* </p>
* <p>
* Notre composant est très simple et ne propose qu'un seul service
* isValidURL()
* </p>
* @author BlueWeb - 2003/01
*/
public class UrlTester {
   private static Perl5Compiler compiler;
   private static Perl5Matcher matcher;
   private static Pattern pattern;
   private final static String REGEXP = "(^(http|ftp|https|file)://
➥)?(\\S+\\.)+(net|org|fr|uk|com|de)(/(\\S)+)?";
  static{
     compiler = new Perl5Compiler();
     matcher = new Perl5Matcher();
     try{
        pattern = compiler.compile(REGEXP);
     }
      // oops ! quelqu'un a touche à la string REGEXP
      // inutile de continuer , jette une RuntimeException
     catch(MalformedPatternException e){
        throw new RuntimeException("Bad Pattern for URL testing");
     }
   }

   /**
   * <p>
   * Teste une URL (anURL) contre la regexp générale
   * </p>
   * @param anURL, String contenant l'URL à tester
   * @return true, si l'URL est valide, false autrement
   */
   public static boolean isValidURL(String anURL){
      return (matcher.matches(anURL,pattern));
   } // isValidURL()
}
```

Utilise un bloc statique pour l'initialisation des matcher et compiler. Ceci permet de gagner en rapidité plutôt que de refaire à chaque appel les mêmes initialisations, ainsi que de jeter une exception dès le chargement de la classe si l'on a modifié la chaîne `regexp` en la rendant incorrecte (au sens de la grammaire Perl 5).

Voici une sortie de la console Java mettant en évidence une modification rendant incorrecte l'expression régulière, ce qui se traduit par une `MalformedPatternException` levée au moment de l'appel à la méthode `compile` de `Perl5Compiler` pendant le chargement de la classe. Avec une telle solution, il est impossible de passer à côté d'un tel problème pour le composant (et il ne vaut mieux pas).

```
java.lang.ExceptionInInitializerError
Caused by: java.lang.RuntimeException: Bad Pattern for URL testing
   at com.blueweb.bookmarks.tools.UrlTester.<clinit>(UrlTester.java:37)
Exception in thread "main"
```

Maintenance du code : le framework JUnit

Comme le bout de code précédent le montre, les expressions régulières constituent un outil d'une grande puissance, mais leur syntaxe hermétique en fait des proies faciles pour les petits bogues insidieux. Enlevez le caractère ^ par suite d'une erreur de frappe ou d'un mauvais copier/coller et votre composant accepterait des URL du type : que_fais_je_là_http://w3.org.

C'est relativement problématique et l'équipe BlueWeb ne peut qu'avoir conscience d'un tel problème. Il s'agit donc de doter notre composant en charge des validations de saisie de garde-fous nous protégeant de ce que l'on appelle les bogues de régression. Cela signifie qu'il nous faut faire en sorte qu'une fois notre composé « packagé », nous nous assurions que les fonctionnalités présentes en version 1 continuent de marcher à chaque nouvelle version. Il s'agit de ne pas revenir en arrière. Il s'agit là bien sûr de simples tests unitaires, c'est-à-dire de tests opérés dans un contexte réduit au seul composant.

Pour mettre en œuvre ce framework, il suffit de respecter les étapes suivantes :

1. Créer une classe de test héritant de `junit.framework.TestCase` (nommer cette classe en suffixant par `Test` le nom de votre classe à tester).
2. Créer une suite de tests comprenant au minimum votre classe de test nouvellement créée.
3. Lancer un des *runners* JUnit parmi les trois disponibles (mode console, mode graphique ou intégré à Ant via la task JUnit).

La classe de test doit être codée selon le schéma suivant :

1. Constructeur acceptant une chaîne de caractères.
2. Créer au moins une méthode `testXYZ` pour chaque méthode publique de la classe à tester.
3. Créer éventuellement une méthode `setUp()` initialisant des ressources manipulées dans les méthodes `testXYZ`.
4. Créer éventuellement une méthode `tearDown()` libérant des ressources manipulées dans les méthodes `testXYZ`.

Tout de suite, voici le code permettant de tester notre composant de vérification des URL.

Classe de test permettant de valider unitairement le code précédent…

```
package com.blueweb.bookmarks.tools;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
/**
*<p>
* La classe de test attachée à la classe UrlTester.
* Ici, l'importance des noms de classe prend tout son sens
```

OUTIL **JUnit**

JUnit est un outil issu d'une des obsessions de l'eXtreme Programming : le test. Il fournit un framework élégant et léger, simple de mise en œuvre et permettant d'opérer des tests unitaires sur des composants. Il est disponible à l'URL suivante : http://www.junit.org. À ce jour, la dernière version disponible est la 3.8.1.

POUR ALLER PLUS LOIN **JUnit**

Le JUnit Cook book doit vous permettre de bien démarrer avec JUnit. Il est fourni avec d'autres documents dans le `.zip` de la distribution JUnit.

- http://junit.sourceforge.net/doc/cookbook/cookbook.htm

```
* puisque l'on voit a quel point UrlTester est un nom mal choisi.
* On utilise le framework JUnit.
* </p>
*/
public class UrlTesterTest extends TestCase {
   // rien a faire dans notre cas
   // mais c'est ici que l'on pourrait ouvrir un fichier ou une
   // connexion BDD utilisée dans notre test
   protected void setUp(){
   } // setUp()

   // rien a faire ici
   // mais c'est ici que l'on pourrait fermer un fichier ou une
   // connexion BDD utilisée dans notre test
   protected void tearDown(){
   } // tearDown()
   /**
    * <p>
    * ce constructeur permet d'afficher quel test échoue
   * </p>
    */
   public UrlTesterTest(String aName){
      super(aName);
   }//constructor()

   /**
    * <p>
    * c'est la méthode permettant de tester le seul service
    * de notre classe UrlTester
    * attention au nom!!
    * </p>
    */
   public void testIsValidURL(){
      assertTrue);
   } // testIsValidURL()

   /**
    * <p>
    * construit la suite de tests à derouler...
    * ici on utilise la méthode basée sur l'introspection Java
    * sympa pour les fainéants...
   * </p>
    */
   public static Test suite(){
      return new TestSuite(UrlTesterTest.class);
   } // suite()
}
```

On doit vérifier que `isValidURL` renvoie bien vrai pour une URL valide comme celle du noyau linux. Bien entendu, un test plus significatif prendrait en compte différents protocoles (ftp, https), différents types d'URL, des URI. Bref, il suffit d'ajouter des `assertEquals`. Évidement on peut inclure des tests renvoyant `false`.

Il ne reste plus qu'à lancer le test (voir figure 3-9).

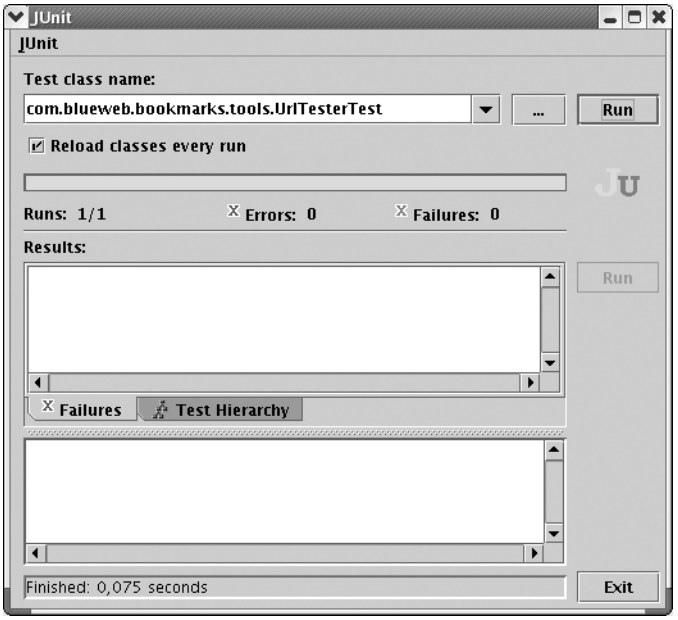


Figure 3–9 Exécution de notre test unitaire avec le programme fourni avec JUnit

Cette capture d'écran présente le lancement du *runner* graphique Swing JUnit. Pour le lancer, il suffit de saisir la ligne suivante dans une console :

```
java junit.swingui.TestRunner
com.blueweb.bookmarks.tools.UrlTesterTest
```

B.A.-BA **Programmation par contrat (DBC, Design By Contract)**

Contraction anglaise pour *Design By Contract*, ou programmation par contrats en français. C'est un concept introduit par un des gourous de l'objet, le français Bertrand Meyer, auteur du langage objet Eiffel. Il s'agit d'insister sur le fait qu'un développeur publiant son API passe un contrat entre lui-même et l'utilisateur de son API. En effet, il propose des services valables dans un certain contexte ; hors de ce contexte (le contrat est rompu), il ne garantit plus le fonctionnement. Bertrand Meyer introduit la notion d'assertions, vérifications de tests induisant des réponses à VRAI ou FAUX. Ces assertions peuvent être de trois types : les pré-conditions, les post-conditions et les invariants. C'est un domaine passionnant, mais qui une fois encore dépasse le cadre de l'ouvrage. Plus d'informations sont disponibles sur le site de la société ISE commercialisant Eiffel :

- http://www.eiffel.com.

Les tests unitaires en pratique

Une pratique prônée par XP semble d'abord étrange, puis se révèle logique : c'est le commandement de Ron Jeffries : commencer par coder les tests. Surprenant en effet, mais c'est une pratique très saine puisque, avant de coder un composant, il est souhaitable de connaître les services qu'il va proposer et donc d'être à même de fixer les différents contrats (au sens de la programmation par contrat (DBC) de Bertrand Meyer).

Si l'on connaît le contrat offert par un service, pourquoi ne pas commencer par coder la classe de tests qualifiant ce composant ? Ce point est réellement un des éléments importants mis en évidence par XP.

De plus, l'importance de l'automatisation des tests implique au sens XP que 100 % des tests passent, et ce pour tous les composants. Ce point entend l'intégration du lancement de nos tests via un outil de `make` type Ant, de façon à ne pas reposer sur l'intervention humaine, mais au contraire sur un caractère systématique. Si l'on écrit du code de test, ce n'est pas pour qu'il soit ignoré…

L'outil le plus simple pour réaliser cette intégration reste l'utilisation de la tâche `JUnit` dans Ant. Pour illustrer l'usage de cette tâche sur le composant de validation d'URL, rien de tel qu'un petit `build file` dédié à cette tâche.

Script ANT lançant l'exécution du jeu de test présenté précédemment

```
<project name="BlueWeb" default="main" basedir=".">



<property name="src" location="."/>
  <property name="build" location="build"/>
  <property name="reports" location="reports"/>
  <property name="reports.html.dir" location="${reports}/html"/>
<!-- initialisation -->
  <target name="init"
          description ="creee les repertoires utilises par la suite">
<!-- utilise la task mkdir pour creer les repertoires -->
    <mkdir dir="${reports}"/>
    <mkdir dir="${reports.html.dir}"/>
    <mkdir dir="${build}"/>
  </target>
  <target name="compile" depends="init" >
    <javac srcdir="${src}" destdir="${build}"
           includes="com/blueweb/bookmarks/tools/*.java"/>
  </target>
  <target name="clean" description="prepare le grand menage...">
    <delete dir="${reports}"/>
    <delete dir="${build}"/>
  </target>
<target name="test" depends="compile">
<junit printsummary="yes" fork="yes" haltonfailure="yes" >
  <formatter type="xml" />
  <classpath>
    <pathelement location="${build}"/>
    <pathelement path="${java.class.path}"/>
  </classpath>
  <test name="com.blueweb.bookmarks.tools.UrlTesterTest"
        todir="${reports}"/>
</junit>
<junitreport todir="${reports.html.dir}">
  <fileset dir="${reports}">
    <include name="TEST-*.xml"/>
  </fileset>
  <report format="noframes" todir="${reports.html.dir}"/>
</junitreport>
</target>
  <!-- target principale (par defaut) -->
<target name="main" depends="test" >
</target>
</project>
```

Crée un projet BlueWeb dont la cible par défaut s'appelle « main » et positionne le répertoire par défaut sur le répertoire courant.

Positionne quelques propriétés telles que le répertoire source, les répertoires utilisés pour le stockage des classes compilées et des rapports.

Ici on supprime les 2 répertoires où sont stockés les résultats de ce build (classes et rapports).

C'est la « target » (cible) qui nous intéresse ; elle va invoquer les deux « tasks » JUnit et JUnitReport de manière à réaliser l'exécution et un rapport sur ce lancement.

Ici, on veut stocker le rapport HTML dans le répertoire précisé. On utilise une présentation du type NOFRAMES pour interdire la production de pages Web avec des frames (car Lynx ne les prend pas en charge).

CONSEIL **Installation**

Attention, reportez-vous à la section du manuel Ant relative aux *optional tasks* pour une référence sur la manière de modifier votre configuration pour utiliser les tâches optionnelles `Junit` et `JunitReport`.

L'exécution de ce code doit produire un rapport (voir figure 3-10).

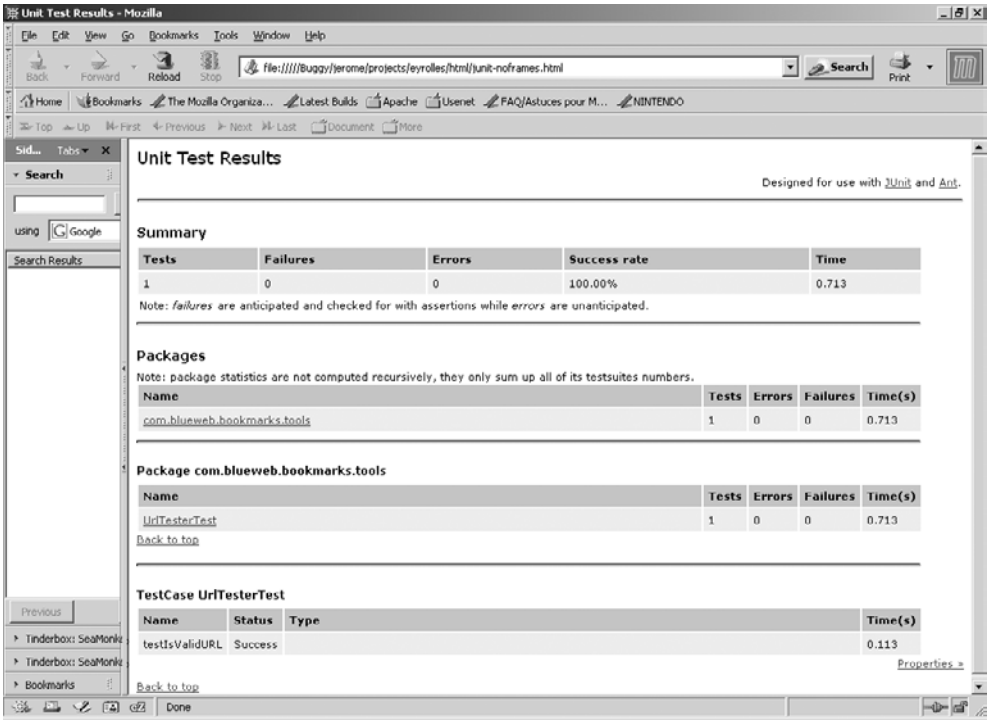


Figure 3–10 Rapport HTML délivré par JUnit et la task style de ANT

Intégration des tests unitaires dans le cycle de vie du logiciel

Il faut rappeler ici quelques-uns des grands principes (*best practices*) permettant de tirer le meilleur parti de l'utilisation d'un framework de tests unitaires :

- Il est bon de coder les classes de test avant même de coder les objets à tester. Pourquoi ? Car cela lie fortement cette pratique avec la phase de conception et permet d'éviter la phrase rituelle : « je le ferai plus tard ». Non, il faut intégrer les tests unitaires dans la phase de codage et le fait de les intégrer de bonne heure (avant le codage même des composants à tester) permet d'être sûr ne pas les « oublier ».
- Il est impératif de rendre l'exécution de ces tests automatique et indissociable de votre processus de génération de livrables (code, paquetages). Pour cela, la task `ANT JUnit` est l'outil rêvé.
- Il est important de coder ces scripts `ANT` de manière à ce que toute erreur soit sanctionnée par l'arrêt du processus de *build*. En effet, si on a l'opportunité de détecter un problème rapidement, il faut faire en sorte de le régler rapidement.

Si les deux derniers points trouvent des solutions pratiques dans les scripts et la documentation de la tâche `JUnit`, comment faire pour intégrer en douceur JUnit dans votre travail quotidien ?

Une solution réside dans une intégration très précoce en faisant en sorte de générer vos suites de tests par votre outil de modélisation (AGL). En effet, des outils comme Objecteering de Softeam (disponible via l'URL http://www.objecteering.com) permettent de modéliser puis générer le code de vos tests. Comment ? Eh bien ! regardons cela…

Plaçons-nous dans le contexte simple d'un paquetage réduit à une seule classe, un convertisseur euro (donc une mini-calculette fort utile pour retrouver nos marques depuis le changement de monnaie).

Ce n'est rien de plus qu'une classe fournissant un constructeur et des méthodes de conversion (franc vers euro et euro vers franc).

Maintenant, grâce au module de tests pour Java, nous pouvons définir une situation de test sous la forme d'un diagramme de séquences UML. Ainsi, pour notre exemple, avec un convertisseur, on peut modéliser cela sous la forme de la figure :

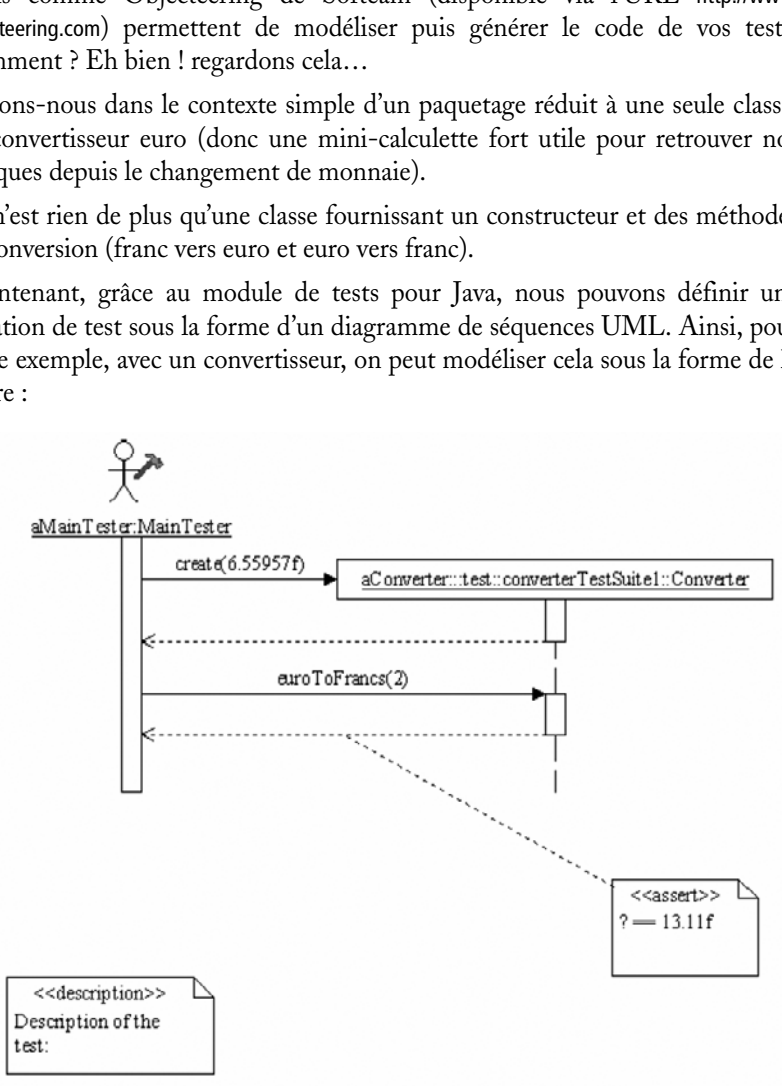


Figure 3–12
Diagramme de séquence synthétisant le test d'une fonctionnalité de conversion (euro vers franc)

Soit, en paraphrasant le diagramme illustré sur la figure 3-12 :

- On crée l'objet convertisseur en passant en paramètre le taux de la conversion (`flottant 6.55957`).

Figure 3–11 Classe de convertisseur euro modélisée avec UML

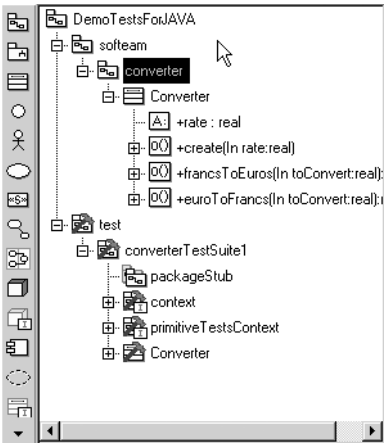


Figure 3–13 Vue d'un test unitaire au sein d'Objecteering, module de test

Bien entendu, cette phase est facilitée par la présence d'assistants…

OUTIL **Artima et TestNG**

Il peut être intéressant de mentionner Artima Suite Runner pouvant compléter avec grand bénéfice JUnit. Ce produit est né de l'expérience concrète de Bill Venners (auteur du meilleur ouvrage disponible sur la machine virtuelle Java) avec JUnit.
Le lecteur plus curieux s'intéressera à TestNG écrit par Cédric Beust qui préfigure peut-être l'avenir des tests unitaires avec son framework à base d'annotation.

- http://www.artima.com
- http://www.beust.com/testng/

À LIRE

- Kernighan & Pike, *The Practice of Programming*, Addison-Wesley, 1999.
- J.-L. Bénard, L. Bossavit, R. Médina, D. Williams, *Gestion de projet eXtreme Programming*, Eyrolles 2002

- On compare le montant obtenu en convertissant 2 euros en francs avec la valeur de référence (ici 13.11).

Le projet vu dans l'outil est du type montré à la figure 3-13 qui illustre comment, à partir d'un composant à tester, on peut obtenir une suite de tests.

À défaut de pouvoir être exhaustive (il ne s'agit pas de se substituer à la documentation de ce produit), le but de cette section était de montrer un moyen d'intégrer élégamment la création de tests unitaires dans votre projet. Pour cela, quoi de mieux dans l'esprit que le produit contenant la modélisation de votre projet. Ainsi, vos cas de tests seront partie intégrante de la documentation générée depuis le produit et ainsi les éventuels problèmes de mauvaise compréhension seront supprimés. En effet, étant donné la simplicité des diagrammes de séquence (simplicité nécessaire), les chances de mauvaise interprétation des fonctionnalités (testées) seront très faibles. L'utilisation d'une telle approche permet de plus de réduire la phase un peu laborieuse et répétitive de codage inhérente au framework JUnit. Nous avons alors adopté une démarche pragmatique permettant de réduire le risque d'erreurs et d'améliorer la productivité. Cette attitude est appelée en japonais « kaizen » comme le rappellent dans leur ouvrage les auteurs de *Pragmatic programmer : from journeyman to master* : essayer de prendre le recul nécessaire à l'analyse permanente de notre travail de manière à trouver des petites astuces ou outils nous permettant de le réaliser mieux et plus vite.

Nous ne détaillerons pas les multiples possibilités offertes par le module dédié aux tests d'Objecteering, qui peut même automatiser la packaging et le déploiement de vos composants au sein d'une instance de Cactus. Rappelons simplement en passant que Cactus est aussi un sous-projet du projet Jakarta et qu'il vise à fournir un framework générique pour les différents types de tests unitaires sur des composants serveurs.

- http://jakarta.apache.org/cactus/index.html

Cette section nous a permis d'évoquer les fondements de l'utilisation du framework JUnit et de proposer une façon de l'intégrer en douceur dans vos projets.

Gestion des traces applicatives

L'approche adoptée par BlueWeb est résolument pragmatique, issue de l'expérience tirée d'années de développement avec différentes technologies. BlueWeb a pleinement conscience que le corollaire d'une démarche utilisant des tests unitaires est l'utilisation de traces applicatives permettant de résoudre les conflits avec les clients, d'accélérer les procédures de correction d'erreurs en enlevant la place au doute. En effet, une trace bien faite doit permettre de suivre étape par étape le chemin de l'utilisateur jusqu'à l'erreur qu'il signale. Même un projet aussi modeste que celui de BlueWeb réclame l'utilisation d'une bibliothèque de gestion des traces (logs), ce qui par ailleurs constituera un plus à ajouter à l'actif de l'équipe.

Kernighan et Pike utilisent des traces plutôt que des outils de débogage pour les raisons suivantes :

- la facilité avec laquelle on peut être perdu dans des structures complexes lors de sessions de débogage ;
- le manque de productivité de ces outils par rapport à une solution manipulant avec justesse des traces de contrôle ;
- l'aspect persistant des traces par rapport à des sessions de debogage.

Pourquoi ? me direz-vous. L'utilisation de `System.out.println()` ou `System.err.println()` permettant d'afficher des messages dans la console de sortie et dans la console d'erreur (généralement la console MS-DOS ou la boîte *shell* utilisée pour lancer l'application) s'avère largement insuffisante, et ce pour différentes raisons :

- manque de rapidité de la solution (solution excessivement coûteuse en ressources) ;
- manque de souplesse (trop figée).

En effet, il est indiscutable qu'une utilisation de traces via des `System.out.println` est très lente, mais le point crucial réside dans le manque de souplesse de la solution lorsqu'on veut :

- supprimer certaines traces (celles correspondant à du debogage doivent être supprimées avant un passage en production) ;
- ajouter des traces (pour traquer un bogue) ;
- créer un nouveau fichier contenant une partie des traces (de manière à séparer en plusieurs fichiers des traces volumineuses) ;
- communiquer avec un serveur de traces (dans le cas d'une application en réseau, en utilisant des services du type `syslog` sous Unix ou le service d'événements de Windows) ;
- modifier le format de la totalité ou d'une partie des traces (suite à une demande émanant de clients).

Tous ces problèmes, BlueWeb les a connus et cherche donc un moyen pour les occulter avant même qu'ils ne surgissent. Il s'agit pour l'équipe d'adopter un framework pouvant répondre à tous ces besoins. Pour cela, il n'y a rien de tel qu'une recherche sur Internet pour connaître l'état de l'art…

ESSENTIEL **Traces**

N'utilisez pas les `System.err.println()` et autres `System.out.println()`, cela pollue le code, ralentit l'application et pose des problèmes une fois déployé chez un client.

Log4J ou java.util.logging ?

Écartons d'emblée, comme l'a fait BlueWeb, la possibilité de créer son propre framework dédié aux traces, et ce pour les sempiternelles raisons :

- tâche ardue et longue ;
- coût élevé pour la société ;
- pourquoi réinventer la roue ?

Figure 3–14 Logo du projet Log4J

Quelles sont alors les solutions envisageables dans ce domaine ? Le nombre de solutions existantes est très important, mais en y regardant de plus près, les projets utilisent en majorité une bibliothèque Log4J.

Log4J est une bibliothèque faisant partie du projet Jakarta (http://jakarta.apache.org/log4j). Elle brille par diverses qualités, notamment sa simplicité d'utilisation, ses bonnes performances, sa conception élégante la rendant à la fois souple et évolutive et le fait qu'elle soit très paramétrable. Il s'agit presque d'un standard de facto.

D'un autre côté, on peut remarquer qu'avec la nouvelle plate-forme (JDK 1.4), Sun a intégré un framework de trace (paquetage `java.util.logging`). Si l'on ne peut que se réjouir d'une telle démarche, il faut quand même constater que celui-ci présente de sévères limitations :

- il a moins de fonctionnalités ;
- il peut nécessiter du codage pour obtenir des fonctionnalités déjà présentes dans Log4J ;
- l'API Sun est maintenue par Sun, tandis que Log4J évolue grâce à la communauté qui l'utilise.

De plus, le passé de Log4J ne fait que jouer en sa faveur, puisque c'est une bibliothèque utilisée par de nombreux développeurs depuis plusieurs années.

Pour ceux qui ne trouvent pas ici les arguments leur permettant de faire ce choix, voir la page web suivante pour des éléments de comparaison supplémentaires.

▸ http://www.qos.ch/ac2001/F11-200.html

Ce choix n'est pas évident étant donnée la qualité des compétiteurs, mais pour BlueWeb la décision est prise, car, par souci de cohérence avec ses autres choix, l'équipe ne peut rester insensible aux arguments du produit utilisé dans :

- Tomcat ;
- JBoss.

Cela sera donc Log4J.

Log4J : concepts

Log4J propose une séparation stricte entre l'insertion d'une trace de log, la destination de cette trace (fichier, *socket*) et le format de l'affichage de cette trace, ce qui nous permet de répondre à la majeure partie des contraintes dictant notre choix. La seule question encore en lice reste celle des performances…

La configuration des logs est faite via un fichier XML ou, plus classiquement, via un fichier `.properties` (associations clé/valeur), mais nous y reviendrons plus tard.

Le vocabulaire nécessaire à l'utilisation de Log4J se résume aux mots suivants :

- **Logger.** Cette classe (et ses classes filles) permet d'autoriser ou non différents niveaux de trace. Le programmeur va utiliser une instance de cette classe pour demander l'insertion d'une trace en précisant le niveau requis (`debug`, `info`, `warn`, `error`, `fatal`, classés suivant leur ordre d'importance croissante).

- **Niveau d'une requête**. Sachant que les demandes d'insertion de trace sont classées par ordre, si l'on choisit d'autoriser les requêtes de type `warn`, toutes les demandes faites avec les niveaux `debug` et `info` seront ignorées (jetées) tandis que les autres seront redirigées vers le bon **Appender**.
- **Appender**. C'est le nom de la classe modélisant dans l'architecture Log4J l'objet chargeant l'écriture des logs vers le ou les flux (socket/fichier, etc.) adéquats.
- **Layout**. Nom de la classe modélisant l'objet en charge de la présentation (format) des messages.
- **Configurator**. Cette classe permet d'initialiser le dispositif de logs pour toute une application (il n'y a donc qu'un seul appel par application).

Ce vocabulaire acquis, on peut tenter d'illustrer par un diagramme de séquences UML l'utilisation typique de ce produit (voir figure 3-15).

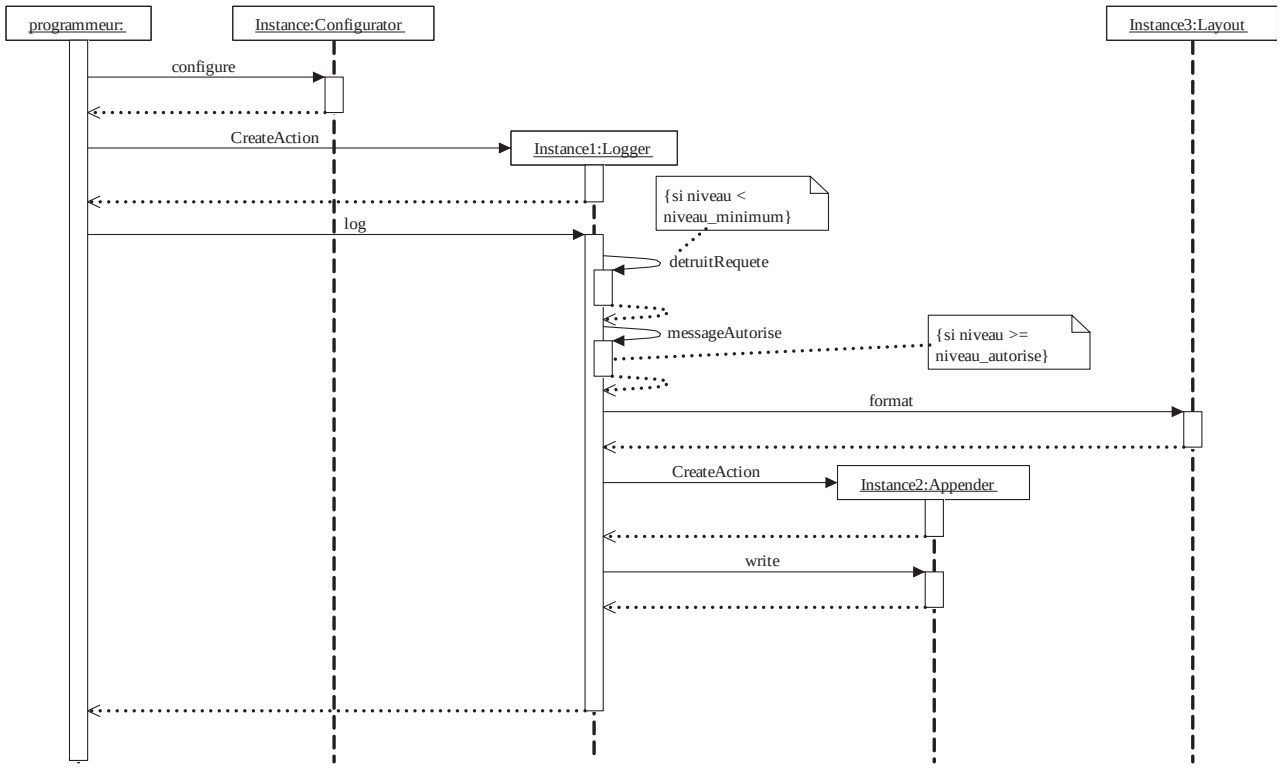


Figure 3–15 Utilisation d'un produit de gestion des traces (diagramme de séquences UML).

Ce diagramme schématise l'utilisation typique de Log4J, en caractérisant la gestion des niveaux de messages et les interactions entre les différents acteurs. Mais en pratique, l'utilisation est beaucoup plus simple, comme le montre l'exemple suivant.

Exemple d'utilisation de Log4J

```
package test.log4j;

import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;

public class Log4jExemple {


  public static void main(String[] args) {

    BasicConfigurator.configure();




    Logger logger = Logger.getLogger(Log4jExemple.class);
    // peut aussi être obtenu par le passage du nom du paquetage
    // Logger logger = Logger.getLogger("test.log4j");

    logger.setLevel(Level.WARN);



    logger.debug("on ne doit pas voir ce message");
    logger.info("celui-ci non plus!!");

    logger.warn("en revanche, celui-la doit etre affiche");

    // fait quelque chose

    logger.fatal("au revoir!!");
  }
}
```

Les imports nécessaires à notre programme, ici réduits à nos dépendances envers Log4J.

Une petite classe démontrant l'utilisation basique de Log4J.

La traditionnelle méthode `main()`.

Configure le système de logs avec les options par défaut : il faut un des configurator une fois par application. Attention : ceci n'a pas besoin d'être fait une fois par classe.

Obtient un `Logger`.

Ajuste le niveau désiré : ici on sélectionne le niveau WARN, donc tous les messages de niveau DEBUG et INFO seront abandonnés.

Positionne 2 traces qui ne seront pas affichées dans l'`appender` par défaut (la console).

Celle -ci doit être affichée.

On s'en va, alors on salue... L'utilisation d'un niveau FATAL pour cela est un peu abusive...

L'exécution de cette classe, via `java test.log4j.Log4jExemple`, donne la sortie, dans la console Eclipse, de la figure 3–16.

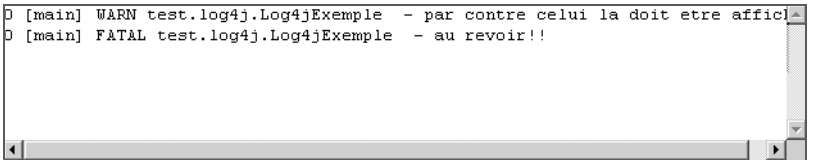


Figure 3–16 Console Java d'Eclipse contenant la sortie du programme de test des traces

Poursuivons sur un autre exemple un peu plus complexe, puisqu'il permet d'introduire un autre Configurator, celui permettant d'utiliser comme source de données un fichier `properties` Java. Il permet de plus de montrer comment utiliser plusieurs `appenders` (ici, l'un envoyant les messages vers la console, l'autre vers un fichier tournant).

Utilisation de plusieurs sorties (appenders)

```
package test.log4j;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
```

```
public class Log4jExempleAvance {
   private static Logger logger =
       Logger.getLogger(Log4jExempleAvance.class);
   private static class Toto{
      public void foo(){
         // faire quelque chose d'utile !!!
      }
   }
   public static void main(String[] args) {
      // teste que l'on ait bien saisi un nom de fichier (un argument)
      if(args.length <1){
         usage();
      }
      PropertyConfigurator.configure( args[0]);
      Toto toto = new Toto();
      logger.debug("toto instancie");
      toto.foo();
      logger.info("foo() invoquee");

      logger.warn("fini!!");
   } // main()

   private static void usage(){
      System.err.println("Mauvais usage!!");
      System.err.println("java test.log4j.Log4jExempleAvance
      ➥     <fichier de configuration");
      System.exit(255);
   } // usage()
}
```

Usage un peu plus avancé de Log4J via un fichier de configuration et donc le configurator associé (`PropertyConfigurator`).

La méthode `main` sera appelée avec un argument : le nom du fichier de configuration utilisé pour les logs.

Voici le fichier de configuration (`log.properties`) correspondant :

```
# ici on utilisera plusieurs appenders
# un de type console, un autre sous la forme d'un fichier tournant
log4j.rootLogger=DEBUG, CON, ROLL
# CON est un appender du type console
log4j.appender.CON=org.apache.log4j.ConsoleAppender
log4j.appender.CON.layout=org.apache.log4j.PatternLayout
# définition du formatage des messages
log4j.appender.CON.layout.ConversionPattern=[%t] %-5p %c - %m%n

# on n'affichera que les messages du niveau WARN ou supérieur
log4j.logger.test.log4j=WARN
# ici ROLL est declare comme un buffer de taille
# maximale 10kb , on conserve 2 copies.
# le fichier est sauve sous le nom rolling.log
log4j.appender.ROLL=org.apache.log4j.RollingFileAppender
log4j.appender.ROLL.File=rolling.log
log4j.appender.ROLL.MaxFileSize=10KB
log4j.appender.ROLL.MaxBackupIndex=2
log4j.appender.ROLL.layout=org.apache.log4j.PatternLayout
log4j.appender.ROLL.layout.ConversionPattern=%d %-5p %c - %m%n
```

Avec ce fichier de configuration, on obtient une sortie console du type de celle de la figure 3–17.

```
[main] WARN  test.log4j.Log4jExempleAvance - fini!!
```

Figure 3–17 Console Java d'Eclipse, montrant l'effet de notre configuration de Log4J

Voici un autre exemple, dans lequel on a modifié un seul paramètre dans ce fichier de configuration :

Fichier de configuration de Log4j (format .properties ici)

```
# ici on utilisera plusieurs appenders
# un de type console, un autre sous la forme d'un fichier tournant
log4j.rootLogger=DEBUG, CON, ROLL
log4j.appender.CON=org.apache.log4j.ConsoleAppender
log4j.appender.CON.layout=org.apache.log4j.PatternLayout
# définition du formatage des messages
log4j.appender.CON.layout.ConversionPattern=[%t] %-5p %c - %m%n
# on n'affichera que les messages du niveau WARN ou superieur
log4j.logger.test.log4j=DEBUG
log4j.appender.ROLL=org.apache.log4j.RollingFileAppender
log4j.appender.ROLL.File=rolling.log
log4j.appender.ROLL.MaxFileSize=10KB
log4j.appender.ROLL.MaxBackupIndex=2
log4j.appender.ROLL.layout=org.apache.log4j.PatternLayout
log4j.appender.ROLL.layout.ConversionPattern=%d %-5p %c - %m%n
```

On obtiendrait une sortie de la forme suivante (figure 3-18) :

```
[main] DEBUG test.log4j.Log4jExempleAvance - toto instancie
[main] INFO  test.log4j.Log4jExempleAvance - foo() invoquee
[main] WARN  test.log4j.Log4jExempleAvance - fini!!
```

Figure 3–18 Observez bien l'effet du changement de configuration sur la sortie dans la console Eclipse

Voilà de quoi terminer notre parcours rapide des possibilités de cette excellente bibliothèque qu'est Log4J.

RÉFÉRENCE **Log4J Manual**

Pour tous les programmeurs curieux d'en savoir plus sur ce produit, on ne peut que conseiller chaudement la lecture du livre de l'auteur de ce produit Ceki Gulcu : *Log4J Manual*, disponible sur le site de l'auteur de Log4J (version *ebook* payante).

- http://www.qos.ch

En résumé…

Ce chapitre a permis d'introduire SWT et JFace, les API du projet Eclipse dédiées au développement d'applications graphiques. De plus, il nous a donné l'occasion d'aborder le problème du contrôle des saisies utilisateur tout en introduisant la notion de tests unitaires et la notion corollaire de bibliothèque de gestion des traces applicatives. On peut remarquer qu'il s'agit ici d'introduire des pratiques saines et des outils de qualité plus que de réaliser une application… En effet, l'accent est mis sur la réutilisation de composants courants, ce qui doit être la préoccupation permanente au sein d'une équipe.

Pour aller plus loin avec ce morceau d'applicatif, il faudrait :

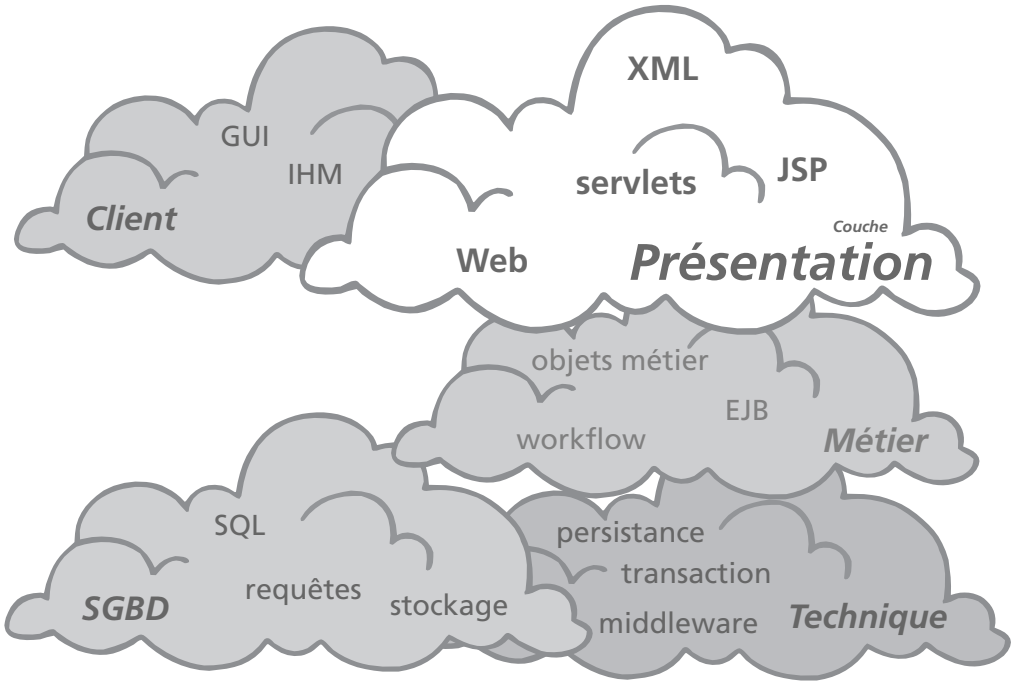
- compléter l'expression régulière utilisée (une recherche sur Internet devrait vous mener vers des pistes sérieuses) ;
- améliorer le nombre de vérifications effectuées dans la méthode `testIsValidURL()`, de manière à vérifier différents protocoles…

Il faut aussi retenir que l'utilisation de tests unitaires, même si elle a été exposée dans un chapitre dédié aux interfaces graphiques, n'est absolument pas réservée aux seuls composants clients.

OUTIL **Eclipse : une plate-forme universelle**

Ce chapitre, en plus de démontrer l'intérêt de l'utilisation de SWT, vise à ouvrir une perspective fort séduisante annoncée par Eclipse : la fin du développement d'applications graphiques en repartant à zéro ou presque. En effet, la sortie imminente de la version 2.1 d'Eclipse va rendre possible (voire aisée) la possibilité d'utiliser Eclipse comme socle universel pour des applications complètement différentes, c'est-à-dire permettre d'enlever tous les modules non nécessaires à votre applicatif tout en gardant tous les composants dignes d'intérêt à vos yeux : aide en ligne, impression, composants de haut niveau permettant la sélection des fichiers, des couleurs, des fontes, etc. Vous n'aurez plus alors qu'à ajouter vos modules pour obtenir une application parfaitement bien adaptée aux contraintes du client en termes de fonctionnalités, et ce en vous concentrant sur votre logique métier et non sur les problématiques récurrentes liées aux impressions… Tout cela constitue une réelle avancée en matière de réutilisation logicielle. Avec JFace, nous disposons d'une brique de plus haut niveau, nous permettant réellement d'imaginer faire d'Eclipse un socle d'applicatifs (un serveur d'applications clientes). En effet, en offrant des services d'aide, d'impression, des éditeurs de code et autres outils de haut niveau, Eclipse nous promet de devenir notre vecteur de distribution d'applications clientes favori.

chapitre 4


XML
GUI
IHM
servlets
JSP
Client
Couche
Web
Présentation
objets métier
EJB
workflow
Métier
SQL
persistance
transaction
requêtes
stockage
SGBD
middleware
Technique

Couche de présentation des données – servlets HTTP

Continuons la navigation dans les couches de notre architecture et abordons la couche de présentation des données. Celle-ci nous rendra indépendants par rapport à la couche cliente via l'utilisation d'un format de données neutre (XML). Cette couche va être accessible par le plus standard des protocoles : HTTP.

SOMMAIRE

MOTS-CLÉS

- HTTP
- Présentation
- Design pattern Commande
- Découplage
- Introspection Java
- Sérialisation

Ouverture et maîtrise du couplage

Trop souvent par le passé, BlueWeb, comme de nombreuses autres entreprises, s'est trouvée prise au piège, bloquée par le manque d'ouverture des solutions logicielles qu'elle avait pu adopter. Plutôt que de faire table rase du passé, elle a décidé d'en tirer les leçons et, par l'adoption d'une architecture à cinq couches, décide de tout mettre en œuvre pour ne pas lier interface graphique et logique métier. Cependant, il faut faire communiquer les deux couches (extrêmes au sens architecture logicielle) et cette communication peut être la source de dépendances.

Pour couper court à tout risque de ce type, l'utilisation d'un format neutre de données tel que XML permet de s'affranchir de toute dépendance, si ce n'est celle du respect d'un format de données sous la forme d'une DTD ou d'un schéma XML.

Bien entendu, cette décision a un coût en termes de performances (car cela sous-entend transformation des objets en XML et vice-versa), mais le jeu en vaut la chandelle s'il y a bien indépendance par rapport à un langage ou un environnement matériel. Cependant, cette indépendance n'a de sens que si les communications entre client et serveur se font, elles aussi via un protocole standard. Il y a donc là une réelle incompatibilité entre ce désir et des solutions fermant de nombreuses portes, comme l'invocation d'objets distants en Java via RMI.

La solution la plus ouverte disponible est sans aucun doute l'utilisation de HTTP comme protocole d'échange, puisque ce dernier est disponible sur tous les postes clients (il suffit d'inclure les couches TCP/IP dans la configuration du poste).

Il reste maintenant à savoir interfacer notre partie serveur avec des requêtes HTTP.

B.A.-BA **DTD (Data Type Definition)**

Une DTD est l'exposé en XML d'une structure de données. Un fichier XML peut référencer une DTD ; dans ce document, toute donnée non conforme à la définition énoncée dans la DTD lèvera une erreur au *parsing*.

APARTÉ **Servlet API**

La dernière version stable disponible est la version 2.4.

POUR ALLER PLUS LOIN

Travailler avec la servlet API

Il est indispensable de posséder « à portée de mains » un exemplaire des spécifications de la servlet API. Vous pourrez les trouver, ainsi que d'autres documents très utiles, sur la page de documentation des servlets accessible à l'adresse suivante :

- http://java.sun.com/products/servlet/docs.html.

Les servlets Java et HTTP

L'édition professionnelle de la plate-forme Java J2EE comporte parmi ses nombreuses composantes une couche appelée « servlet API », qui permet d'exécuter du code Java à partir de requêtes respectant divers protocoles. De par leur conception, les servlets ne sont pas spécifiquement dédiées à HTTP (d'où la classe `HttpServlet` héritant de `Servlet`), mais en pratique HTTP est pour l'instant le protocole implémenté (en considérant HTTPS comme une simple surcouche d'HTTP).

Avec la servlet API et un produit l'implémentant, nous disposons donc d'une réponse standard, robuste et simple d'utilisation.

Rappels sur les servlets

Cette section rappelle brièvement les concepts clés nécessaires à l'implémentation d'une servlet et à son déploiement dans un moteur de servlets.

Codage

Le développement d'une servlet est tout ce qu'il y a de plus simple et répétitif, puisqu'il suit le prototype suivant (à adapter pour refléter votre organisation en paquetages et à compléter pour ajouter vos traitements).

Modèle de servlet

```
package com.blueweb.server.http.test;

import java.io.IOException;
import java.io.Writer;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
* @author BlueWeb - dev Team
* <p>
* Une servlet prototype.
* </p>
*/
public class PrototypeServlet extends HttpServlet {
   /**
    * <p>
    * La méthode init sert à déclarer et réaliser toutes
    * les initialisations nécessaires au fonctionnement
    * d'une servlet (obtention d'une connexion JDBC, ouverture
    * d'un fichier etc.
    * </p>
    * @param aConfig, configuration de la servlet, permet d'obtenir
    * de nombreuses infos
   */
   public void init(ServletConfig aConfig) throws ServletException{
      super.init(aConfig);
      // placer ici toutes les initialisations
      // nécessaires à votre servlet
   } // init()
   /**
    * <p>
    * Cette méthode est la plus fréquemment utilisée ; il s'agit
    * de la méthode permettant de réagir à une requête HTTP de type
    * GET. GET est la méthode HTTP « par défaut », c'est-à-dire celle
    * réalisée lors de la saisie d'une URL dans un navigateur
    * </p>
    * @param aRequest, correspond à un mapping objet de la requête
    * utilisateur
    * @param aResponse, permet d'écrire dans le flux retourné au
    * client
   */
```

Liste des imports requis pour cette classe…

La méthode `init()` est appelée au chargement de la servlet. On y place le code nécessaire aux initialisations du type : obtention d'une source de données, création de structures…

Cette méthode va être appelée par la méthode `service()`, non redéfinie ici, dans le cas d'une requête HTTP de type GET. C'est le type de requêtes le plus fréquemment rencontré.

```
public void doGet(HttpServletRequest aRequest,
                  HttpServletResponse aResponse){
   // Placer ici tout le code nécessaire au traitement...
   // suivi du code nécessaire à l'écriture de la réponse.
   // Le code suivant n'est donné qu'à titre indicatif.
   // Il récupère la valeur associée au paramètre
   // MON_PARAMETRE
   // cela sous-entend une URL du type
   // http://monserveur:sonport/uncontexte/monmapping?
      ➥ MON_PARAMETRE=unevaleur
```

Ici, on va extraire de la requête provenant du client un paramètre nommé MON_PARAMETRE.

```
   String mon_parametre=aRequest.getParameter("MON_PARAMETRE");

   try {
      Writer writer = aResponse.getWriter();
      writer.write("une jolie reponse");
      writer.close();
   } catch (IOException e) {
   }
} // doGet()
```

Descripteur de déploiement

RÉFÉRENCE Descripteur de déploiement

Pour bien commencer avec le descripteur de déploiement, l'URL suivante permet de comprendre les concepts clés de cette étape importante du cycle de vie d'une application J2EE :

- http://jakarta.apache.org/tomcat/tomcat-5.0-doc/appdev/deployment.html

Ce code placé dans un moteur de servlets ne fait rien, car on s'est jusqu'à présent contenté de créer une casse **pouvant réagir** à des requêtes HTTP (GET ou POST). Il s'agit maintenant de préciser au moteur de servlets le contexte nécessaire à l'exécution de notre code.

C'est précisément le rôle du fichier `web.xml`. Celui-ci nous permet d'associer un contexte HTTP à nos servlets, nous autorisant à déclarer comment lier une servlet à une URL.

Fichier XML de description du déploiement, qui utilise un encodage ISO-8859-1, standard pour les pays d'Europe de l'ouest…

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

Dans cette section, on va trouver un certain nombre d'informations très générales, pas toujours importantes, telles que la description ou le nom devant être affiché au sein d'outils…

```
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<!--se référer aux spécifications pour plus d'informations sur cette
DTD, cet exemple de fichier omet la section décrivant les filtres HTTP
-->
<web-app>
<!-- description générale de l'application web -->
```

Ce bloc est d'une importance vitale, il permet d'associer à un nom (celui que vous voulez), une classe Java contenant le code de votre servlet.

```
   <display-name>BlueWeb - Application Gestion Signets</display-name>
   <description>
      Bla-bla qualifiant l'application… N'a qu'un rôle anecdotique…
   </description>
    <!--
         Paramètres d'initialisation du contexte...
         Peuvent être retrouvés par code (depuis une JSP OU SERVLET)
             String value =
               getServletContext().getInitParameter("name");
         Section facultative
    -->
```

```
    <context-param>
      <param-name>webmaster</param-name>
      <param-value>webmaster@blueweb.com</param-value>
      <description>
        adresse électronique du webmestre...
      </description>
    </context-param>
    <servlet>
      <servlet-name>controller</servlet-name>
      <description>
         C'est la servlet controleur (cf D.P MVC). Le chef d'orchestre
         de notre application cote serveur.
      </description>
      <!-- entrer ici une véritable classe, il ne s'agit ici que
           d'un nom fictif -->
      <servlet-class>com.mycompany.mypackage.ControllerServlet
      </servlet-class>
      <!-- montre comment utiliser des parametres dans des servlets -->
      <init-param>
        <param-name>listOrders</param-name>
        <param-value>com.mycompany.myactions.ListOrdersAction
        </param-value>
      </init-param>
      <init-param>
        <param-name>saveCustomer</param-name>
        <param-value>com.mycompany.myactions.SaveCustomerAction
        </param-value>
      </init-param>
      <!-- charge cette servlet au démarrage -->
      <!-- cette valeur >0 veut dire « oui, le conteneur -->
      <!-- doit respecter un ordre croissant dans le -->
      <!-- chargement des servlets » -->
      <load-on-startup>5</load-on-startup>
    </servlet>
    <!-- ajouter autant de balises servlets que nécessaire -->
    <!-- définition des mappings entre URI et servlet à exécuter -->
    <!-- ici, associer la servlet controller à toute URI -->
    <!-- se finissant par .do -->
    <!-- ce mapping adopte la même convention que struts... -->
    <servlet-mapping>
      <servlet-name>controller</servlet-name>
      <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    <!-- s'il y a d'autres servlets, ajouter ici les clauses -->
    <!-- servlet-mapping les concernant -->
    <!-- definition du temps de timeout des sessions HTTP -->
    <session-config>
      <session-timeout>30</session-timeout>    <!-- 30 minutes -->
    </session-config>
</web-app>
```

Ici, on passe des paramètres permettant d'initialiser les actions reconnues comme valides (au sein de notre petit framework MVC).
On utilise un système très proche de celui utilisé au sein de Struts.

Cette section permet de définir une servlet en définissant (au minimum), le nom de celle-ci et la classe `Java` contenant l'implémentation du code à effectuer.

Cette section permet d'effectuer l'association entre une URL et la servlet à exécuter.
Par l'expression régulière `*.do`, on associera toute URL au sein de notre contexte Web finissant par `.do` à la servlet dite `controller`.

Au sein même de ce fichier, vous pourrez trouver des commentaires insérés selon les règles XML, c'est-à-dire placés entre les balises `<!--` et `-->` .

Il s'agit ici, en plus d'informations générales (nom de l'application web), de lister toutes les servlets (en associant à un nom, un nom de classe Java), puis d'associer à une forme d'URL un nom de servlet.

Le diagramme UML suivant synthétise les grandes lignes du mécanisme d'invocation d'une servlet.

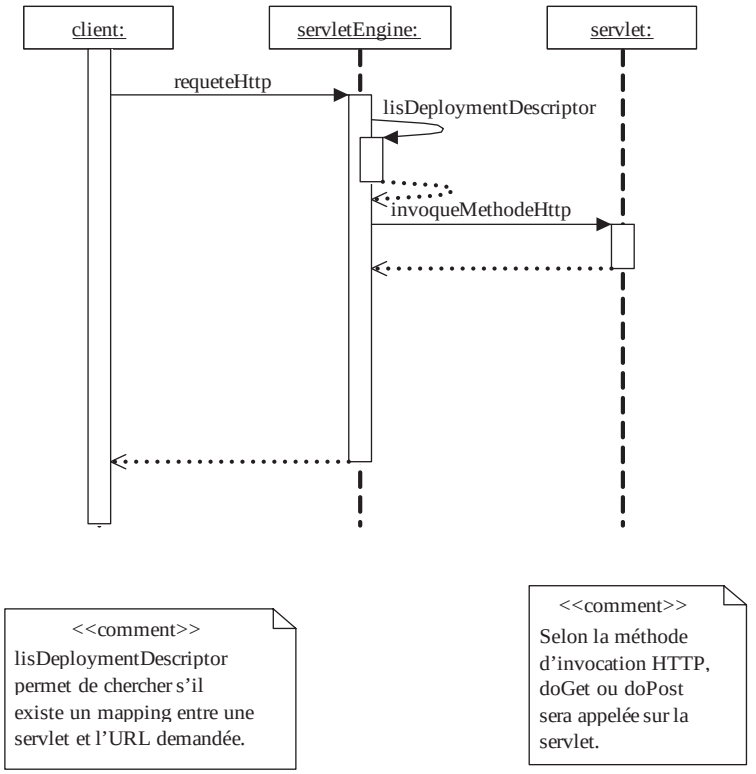


Figure 4–1 Diagramme de séquences UML schématisant le dialogue client/serveur

POUR ALLER PLUS LOIN **Référence sur Tomcat**

Ce diagramme est loin d'être exact, mais il permet de comprendre le principe. Pour en savoir plus sur le fonctionnement d'un moteur de servlets, on ne peut que chaudement recommander l'excellent ouvrage de James Goodwill sur Tomcat.

J. Goodwill, *Apache Jakarta-Tomcat*, APress, 2001.

Packaging

Sans le savoir et tel Monsieur Jourdain, vous êtes en train de réaliser une **WebApp** (application web). Pour la distribuer, et donc l'utiliser, vous devez vous plier aux contraintes fixant le packaging. Il ne peut s'agir ici de rappeler ces contraintes, mais nous mettrons tout cela en pratique par la suite. Suivant le moteur de servlets utilisé, vous pourrez ou non disposer de certains outils facilitant le déploiement. Rappelons simplement que vous aurez le choix entre différentes politiques :

- ajouter une hiérarchie de fichiers à un endroit donné de votre moteur de servlets ;
- déployer un `war file` (archive au format `.jar` contenant une WebApp) ;

- déployer un `ear file` (archive au format `.jar` contenant une *entreprise application*).

À ce stade, vous devez être capables d'aborder le développement de servlets et leur déploiement.

Maintenant examinons comment mettre en place « proprement » cette technologie, de manière à l'insérer dans notre architecture.

Le pattern Commande et les requêtes HTTP

À la manière des frameworks MVC ou MVC2 du type Struts ou Barracuda, l'équipe de BlueWeb décide pour le projet prototype d'opter pour une implémentation « maison » de ce design pattern.

Pourquoi réécrire une implémentation de ce design pattern et ne pas utiliser un des frameworks existant ? Plusieurs raisons ont contribué à opter pour cette voie :

- Écrire une implémentation de ce pattern pour un projet pilote permet de bien l'appréhender et donc de bien le maîtriser.
- Le temps imparti au projet n'étant pas infini, il est plus réaliste d'implémenter quelque chose de fonctionnellement limité plutôt que de se perdre dans les méandres d'un projet ambitieux.
- Cette expérience permettra dans l'avenir d'acquérir des connaissances permettant d'être plus critique (au sens positif du terme) au sujet de produits MVC comme Struts. Ce framework séduisant attire de nombreuses personnes chez BlueWeb et il n'est pas exclu de lancer prochainement un projet pilote utilisant ce produit.

Schéma général du design de cette implémentation

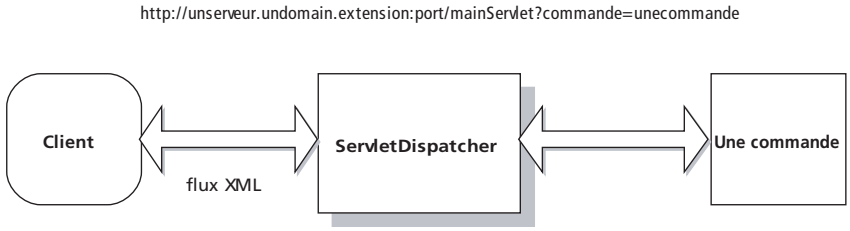


Figure 4–2 Schéma de principe du pattern MVC appliqué au Web

L'idée est simple mais efficace : associer une action utilisateur (ajouter un signet) à une commande (côté serveur) et acheminer toutes les actions via HTTP vers la servlet principale assurant un rôle de répartition et de gestion des erreurs (ainsi qu'un rôle de conversion Java <-> XML).

Avant de regarder l'implémentation, examinons un peu le design pattern Commande…

B.A.-BA MVC et MVC2

Dans le Modèle Vue Contrôleur 2, il n'y a plus qu'un seul contrôleur au lieu de plusieurs. Ceci permet d'éviter une prolifération gênante d'endroits où trouver la logique métier.

ANTI-PATTERN **La servlet magique**

Dans son ouvrage « Bitter Java » Bruce Tate décrit très bien ces servlets « magiques », conçues pour ne faire qu'une seule chose mais dont le code grossit au fur et à mesure des évolutions jusqu'à en devenir impossible à maintenir (les besoins peuvent avoir changé, les cas de figure peuvent s'être multipliés et le code avoir enflé…). Les problèmes de bogues d'exécution n'apparaissent que quand le code franchit le cap fatidique des 64 Ko pour une méthode. Ce design permet de mieux nous affranchir des problèmes que l'on ne peut apprécier à l'avance : en effet, comment prévoir la façon dont va évoluer l'application ?

Le pattern Commande (Command) en quelques mots

Utilisable côté client comme côté serveur, ce pattern permet d'écarter le problème fréquemment rencontré des servlets « magiques ».

L'objectif est donc de modéliser (encapsuler) une action (lister tous les signets, effacer un signet…) sous forme d'une classe, et ce de manière à ce que l'on puisse maintenir l'application simplement en ajoutant de nouvelles commandes pour les nouvelles actions, en modifiant le code d'une commande précise si l'action attachée semble erronée.

Ceci peut être traduit par le petit diagramme de séquences UML de la figure 4-4.

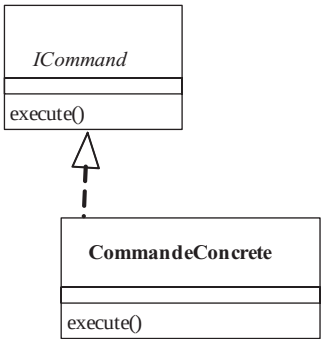


Figure 4–3 Diagramme de classe d'une commande

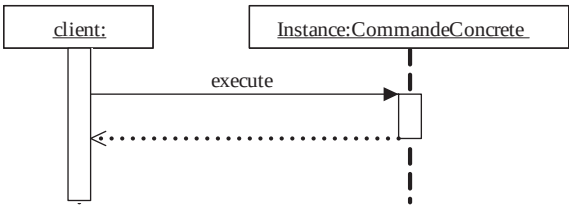


Figure 4–4 Diagramme de séquence de principe

Le design pattern Commande est un des plus célèbres, car il est utilisé dans de nombreux contextes. Ainsi, dans le monde Java/Swing, il est utilisé classiquement pour gérer l'`Undo/Redo`. La classe `AbstractAction` Swing est très proche de l'interface `Command` définie dans le diagramme de classes précédent.

Quels sont les bénéfices de cette conception ? En groupant (isolant) le code d'une action dans une seule classe, on obtient de nombreuses petites classes de taille réduite, donc simples à maintenir et à faire évoluer, car moins sujettes aux effets de bord. Par ailleurs, cette conception a le mérite de permettre une parallélisation des développements, car deux codeurs peuvent travailler en même temps sur deux actions utilisateur différentes.

POUR ALLER PLUS LOIN **Filtres HTTP**

La servlet API 2.3 a introduit une notion de filtrage des requêtes HTTP. Cette nouveauté par rapport à la version précédente (la 2.2) nous permet d'écrire un code portable d'un moteur de servlets à l'autre et fournissant les mêmes fonctionnalités que des API propriétaires telles que les Valve de Tomcat. Avec l'interface `javax.servlet.Filter`, nous disposons d'un moyen simple et puissant de réaliser différentes tâches répétitives, sans pour cela polluer le code de nos servlets. Un filtre HTTP est donc une classe `Java`, qui sera exécutée avant réception de la requête HTTP par notre servlet et après l'envoi de la réponse. L'intérêt devient évident quand il s'agit de réaliser des tâches telles que filtrage, logging ou vérification de droits utilisateur (authentification). Bien sûr, cela sous-entend un moyen de réaliser l'association entre les filtres et les URL. Ceci est standardisé et ce *mapping* est fait au niveau du fichier de déploiement : `web.xml`. La version présentée précédemment ne tenait pas compte de cette notion par souci de simplicité.

Un design pattern en pratique : les Interceptors JBoss

On peut noter un usage très intéressant de ce pattern dans l'architecture du projet JBoss, avec la notion dans ce projet d'**Interceptor**. JBoss 3 avec ce design est un produit extrêmement flexible et configurable selon vos besoins en ajustant quelques fichiers de configuration. Mais avant d'en arriver là, examinons cette notion, fondamentale dans ce produit.

Un **Interceptor** a pour vocation d'encapsuler une action (service technique tel que transaction, authentification, persistance gérée par le conteneur…) au sein d'une invocation d'un EJB par un client. Il s'agit d'une simple classe `Java` qui sera invoquée par le conteneur, lors d'un appel à un service d'un EJB. JBoss est conçu pour permettre l'appel de un à plusieurs de ces objets, lors de toute invocation. Ainsi donc et sans le savoir, avant et après traitement, notre appel est passé à travers une série d'objets. Cette notion est très proche de la notion de filtres HTTP introduite dans l'encadré.

La figure 4-5 présente le diagramme général d'un appel à un service d'un EJB au sein de JBoss.

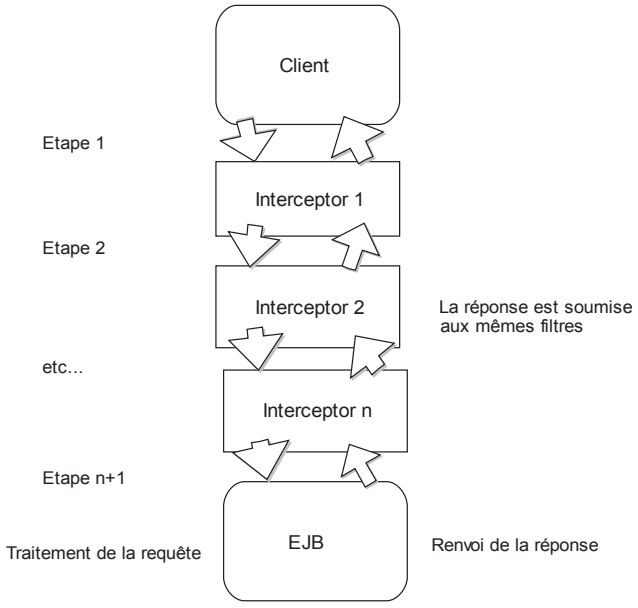


Figure 4–5 La pile des intercepteurs dans JBoss

Les fichiers de configuration de JBoss 3 vous permettent de modifier l'ordre, le nombre et le type des intercepteurs devant être utilisés pour toutes les applications (dans le fichier `standardjboss.xml`) ou pour une seule application (par le fichier `jboss.xml`). Les balises `Interceptor` sont insérées au sein des balises `Container-configuration`.

B.A.-BA **Intercepteur**

Ce terme provient du monde Corba où il désigne un composant dont la fonctionnalité est très proche des intercepteurs dans JBoss. Ces composants permettent de réaliser un certain nombre de tâches de manière transparente pour le codeur, comme l'ajout de traces, l'authentification... Tâches que l'on retrouve dans le fichier de description des intercepteurs dans JBoss. La finalité est d'alléger le code, de diminuer le nombre de paramètres dans les méthodes (et donc d'améliorer la lisibilité et faciliter la maintenance), et aussi d'automatiser l'exécution de tâches répétitives en les exécutant à un niveau très bas dans le protocole en interceptant les appels de méthodes.

Voici un extrait du fichier `standardjboss.xml` livré dans JBoss 3.20RC4 (dernière version disponible à ce jour).

Fichier de configuration de la pile des intercepteurs dans JBoss

Dans cette section, on va trouver la liste des intercepteurs associés aux entités. On voit que ceux-ci sont en rapport avec différents domaines tels que la persistance, la sécurité ou les transactions…

```
<container-configurations>
      <container-configuration>
  <container-name>Standard CMP 2.x EntityBean</container-name>
   <call-logging>false</call-logging>
   <sync-on-commit-only>false</sync-on-commit-only>
   <container-interceptors>
     <interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor
     </interceptor>
     <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
     <interceptor>org.jboss.ejb.plugins.SecurityInterceptor
     </interceptor>
     <interceptor>org.jboss.ejb.plugins.TxInterceptorCMT
     </interceptor>
     <interceptor metricsEnabled="true">
        org.jboss.ejb.plugins.MetricsInterceptor</interceptor>
     <interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor
     </interceptor>
     <interceptor>org.jboss.ejb.plugins.EntityLockInterceptor
     </interceptor>
     <interceptor>org.jboss.ejb.plugins.EntityInstanceInterceptor
     </interceptor>
     <interceptor>org.jboss.ejb.plugins.EntityReentranceInterceptor
     </interceptor>
     <interceptor>
       org.jboss.resource.connectionmanager.CachedConnectionInterceptor
     </interceptor>
     <interceptor>
       org.jboss.ejb.plugins.EntitySynchronizationInterceptor
     </interceptor>
     <interceptor>
       org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor
     </interceptor>
   </container-interceptors>
   <instance-pool>org.jboss.ejb.plugins.EntityInstancePool
   </instance-pool>
   <instance-cache>
     org.jboss.ejb.plugins.InvalidableEntityInstanceCache
   </instance-cache>
   <persistence-manager>
     org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager
   </persistence-manager>
   <transaction-manager>org.jboss.tm.TxManager</transaction-manager>
```

Cet extrait montre la configuration standard d'un composant entité en mode CMP 2.0. Il est impressionnant de constater que tout appel à ce composant induira une invocation en cascade de treize intercepteurs !

Maintenant que le principe est acquis, examinons la structure globale d'un **Interceptor** dans JBoss. Tout d'abord, voici l'interface définissant les services basiques offerts par tout intercepteur.

```
/*
* JBoss, the OpenSource J2EE webOS
* Distributable under LGPL license. See terms of license at gnu.org.
*/
package org.jboss.ejb;
import org.jboss.invocation.Invocation;

/**
* Provides the interface for all container interceptors.
* @author <a href="mailto:rickard.oberg@telkel.com">Rickard Öberg</a>
* @author <a href="mailto:marc.fleury@jboss.org">Marc Fleury</a>
* @version $Revision: 1.10 $
*<p>20011219 marc fleury:</p>
*/
public interface Interceptor
   extends ContainerPlugin
{
   /**
    * Set the next interceptor in the chain.
    * @param interceptor   The next interceptor in the chain.
    */
   void setNext(Interceptor interceptor);

   /**
    * Get the next interceptor in the chain.
    * @return   The next interceptor in the chain.
    */
   Interceptor getNext();
   Object invokeHome(Invocation mi) throws Exception;
   Object invoke(Invocation mi) throws Exception;
}
```

Cette interface est plutôt réduite (4 méthodes seulement). Les méthodes fondamentales sont celles en rapport avec l'invocation d'objet (`invoke` et `invokeHome`). C'est d'ailleurs celles-ci que nous redéfinirons…

Voici ensuite un extrait du code source d'une des implémentations de cette interface, un objet assurant l'écriture de traces.

Exemple d'un intercepteur de JBoss tiré du code source de JBoss

```
/*
* JBoss, the OpenSource J2EE webOS
* Distributable under LGPL license. See terms of license at gnu.org.
*/
package org.jboss.ejb.plugins;

import java.io.PrintWriter;
import java.io.StringWriter;
import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;
import java.rmi.ServerError;
import java.rmi.ServerException;
import java.util.Map;
import javax.ejb.EJBException;
import javax.ejb.NoSuchEntityException;
```

Imports nécessaires à l'exécution et à la compilation de la classe.

Exemple d'un intercepteur de JBoss tiré du code source de JBoss (suite)

```
import javax.ejb.NoSuchObjectLocalException;
import javax.ejb.TransactionRolledbackLocalException;
import javax.transaction.TransactionRolledbackException;
import org.apache.log4j.NDC;
import org.jboss.ejb.Container;
import org.jboss.invocation.Invocation;
import org.jboss.invocation.InvocationType;
import org.jboss.metadata.BeanMetaData;
import org.jboss.tm.JBossTransactionRolledbackException;
import org.jboss.tm.JBossTransactionRolledbackLocalException;
```

Cet intercepteur assure un rôle lié aux traces. Suivant l'état d'un booléen, il tracera l'invocation de la méthode (en ajoutant un message `start method` avant le début de la méthode appelée et un message `Stop method` après la fin de celle-ci.

```
/**
* An interceptor used to log all invocations. It also handles any
* unexpected exceptions.
*
* @author <a href="mailto:rickard.oberg@telkel.com">Rickard Öberg</a>
* @author <a href="mailto:Scott.Stark@jboss.org">Scott Stark</a>
* @author <a href="mailto:dain@daingroup.com">Dain Sundstrom</a>
* @author <a href="mailto:osh@sparre.dk">Ole Husgaard</a>
* @version $Revision: 1.24.2.6 $
*/
public class LogInterceptor extends AbstractInterceptor
{
   // Static
   // Attributes
   protected String ejbName;
   protected boolean callLogging;
   // Constructors
   // Public
   // Container implementation
   public void create()
      throws Exception
   {
      super.start();
      BeanMetaData md = getContainer().getBeanMetaData();
```

Le nom du bean est renseigné dans les métadonnées (fichier de configuration XML, lu et chargé en mémoire).

```
      ejbName = md.getEjbName();
      // Should we log call details
      callLogging = md.getContainerConfiguration().getCallLogging();
   }
   /**
    * This method logs the method, calls the next invoker, and handles
    * any exception.
    *
    * @param invocation contain all infomation necessary to carry out
    * the invocation
    * @return the return value of the invocation
    * @exception Exception if an exception during the invocation
    */
   public Object invokeHome(Invocation invocation)
      throws Exception
   {
```

Stocke le nom de l'EJB dans le thread contexte, manipulé ici comme une pile.

```
      NDC.push(ejbName);
```

Exemple d'un intercepteur de JBoss tiré du code source de JBoss (suite)

```
      String methodName;
      if (invocation.getMethod() != null)
      {
         methodName = invocation.getMethod().getName();
      }
      else
      {
         methodName = "<no method>";
      }
      boolean trace = log.isTraceEnabled();
      if (trace)
      {
         log.trace("Start method=" + methodName);
      }
      // Log call details
      if (callLogging)
      {
         StringBuffer str = new StringBuffer("InvokeHome: ");
         str.append(methodName);
         str.append("(");
         Object[] args = invocation.getArguments();
         if (args != null)
         {
            for (int i = 0; i < args.length; i++)
            {
               if (i > 0)
               {
                  str.append(",");
               }
               str.append(args[i]);
            }
         }
         str.append(")");
         log.debug(str.toString());
      }
      try
      {
        return getNext().invokeHome(invocation);
      }
      catch(Throwable e)
      {
         throw handleException(e, invocation);
      }
      finally
      {
         if (trace)
         {
            log.trace("End method=" + methodName);
         }
         NDC.pop();
         NDC.remove();
      }
   }
```

Est-ce que le mode de `trace` est activé ?

On bâtit la sortie (trace) en ajoutant le nom de la méthode et les arguments... On remarque l'utilisation judicieuse d'un `StringBuffer` en lieu et place de l'opérateur et de la classe `String`. Et ce, pour des raisons de performance.

Exemple d'un intercepteur de JBoss tiré du code source de JBoss (suite)

```
/**
 * This method logs the method, calls the next invoker, and handles
 * any exception.
 *
 * @param invocation contain all infomation necessary to carry out
 * the invocation
 * @return the return value of the invocation
 * @exception Exception if an exception during the invocation
 */
public Object invoke(Invocation invocation)
   throws Exception
{
   NDC.push(ejbName);
   String methodName;
   if (invocation.getMethod() != null)
   {
      methodName = invocation.getMethod().getName();
   }
   else
   {
      methodName = "<no method>";
   }
   boolean trace = log.isTraceEnabled();
   if (trace)
   {
      log.trace("Start method=" + methodName);
   }
   // Log call details
   if (callLogging)
   {
      StringBuffer str = new StringBuffer("Invoke: ");
      if (invocation.getId() != null)
      {
         str.append("[" + invocation.getId().toString() + "] ");
      }
      str.append(methodName);
      str.append("(");
      Object[] args = invocation.getArguments();
      if (args != null)
      {
         for (int i = 0; i < args.length; i++)
         {
            if (i > 0)
            {
               str.append(",");
            }
            str.append(args[i]);
         }
      }
      str.append(")");
      log.debug(str.toString());
   }
```

On récupère le nom de la méthode invoquée s'il y a lieu (non nul).

Exemple d'un intercepteur de JBoss tiré du code source de JBoss (suite)

```
      try
      {
         return getNext().invoke(invocation);
      }
      catch(Throwable e)
      {
         throw handleException(e, invocation);
      }
      finally
      {
         if (trace)
         {
            log.trace("End method=" + methodName);
         }
         NDC.pop();
         NDC.remove();
      }
   }
```

Ce qu'il faut retenir de ce code est que le schéma d'un **Interceptor** est globalement toujours le même, puisqu'il s'agit de procéder suivant quatre étapes :

1. traitement d'entrée ;
2. appel de l'intercepteur suivant ;
3. traitement de sortie ;
4. sortie retournant la valeur renvoyée par la file d'appels.

Cette longue parenthèse doit vous permettre d'appréhender l'importance pratique d'un tel motif de conception et comment, lorsqu'il est utilisé avec justesse, celui-ci permet d'isoler de manière élégante des traitements annexes à votre application. Il est très intéressant de préciser, ici, que dans le cas de JBoss, la logique est poussée à son paroxysme, puisque tout le travail est réalisé par ces intercepteurs et que le code du serveur EJB central est quasiment celui d'un serveur JMX (Java Management Extensions), c'est-à-dire un serveur d'administration d'applications locales ou distantes.

Implémentation dans le projet BlueWeb

Nous n'allons pas détailler tout le code utilisé pour cette couche, mais simplement nous attarder sur quelques passages.

Le point clé de notre implémentation réside dans l'interface `IHttpCommand` présentée ci-après.

Code source complet

Pour obtenir un listing complet du code du projet, se reporter au site web compagnon de l'ouvrage.

- www.editions-eyrolles.com

```
package com.blueweb.bookmarks.http;
import java.util.Map;
/**
```

```
* @author BlueWeb - 2003
* <p>
* Cette interface définit les fonctionnalités d'une commande
* HTTP au sens de notre architecture (Pattern Command).
* Le seul service requis est en fait l'implémentation de la
* méthode execute().
* Une commande sera lancée depuis la servlet controleur,
* puis le résultat post-traite de retour dans la servlet controleur.
* </p>
*/
```

Interface minimaliste mais suffisante pour notre propos. C'est la plus fidèle représentation du design pattern Commande.

```
public interface IHttpCommand {

   /**
    * <p>
    * exécute une requête cliente transmise par la servlet controleur.
    * Celle-ci se doit d'alimenter correctement la map d'entrée
    * et doit se charger de sérialiser (s'il y a lieu) les objets
    * transmis en sortie (dans outputMap)
    * </p>
    * @param inputMap, tableau associatif contenant les objets
    * nécessaires à l'exécution de la commande
    * @param outputMap, tableau associatif contenant les objets
    * retournés au client
    * @exception CommandAbortedException, si l'exécution échoue
   */
   public void handleRequest(Map inputMap,Map outputMap)
           throws CommandAbortedException;

}
```

Cette interface utilise une exception dont le code (très simple) est :

```
package com.blueweb.bookmarks.http;
/**
* @author BlueWeb
* <p>
* Une exception levée dès l'échec de l'exécution d'une commande
* </p>
*/
public class CommandAbortedException extends Exception {

   /**
    * Constructor for CommandAbortedException.
    * @param aMessage
    */
   public CommandAbortedException(String aMessage) {
      super(aMessage);
   }
}
```

On peut remarquer aussi un passage de la servlet contrôleur (classe `MainServlet`) permettant d'initialiser au chargement de la servlet (c'est-à-dire au chargement de la webapp d'après les directives du `web.xml`) la liste des commandes valides et

les classes d'implémentation les réalisant (sous la forme d'un objet `java.util.Properties`).

```
public void init(ServletConfig aConfig) throws ServletException{
   //invocation de la méthode parente indispensable
   super.init(aConfig);
   //obtient le contexte d'exécution
   servletContext = aConfig.getServletContext();
  try{
String mapping_file = aConfig.getInitParameter( "commands.file" );
//obtient le fichier de commandes via
//les paramètres d'initialisation de la servlet
servletContext.log("mapping file = " + mapping_file );// log
commandsMap = new Properties();// initialise la map
try{
    // charge le fichier
   commandsMap.load(servletContext.getResourceAsStream(mapping_file));
   logger.debug("fetched a number of commands = " + commandsMap.size());// log
}
catch(Exception e){
   logger.info("unable to access the properties file");
}
```

Présentation des données

Le découplage souhaité entre partie cliente et logique serveur nécessite l'utilisation d'un format neutre de données, si l'on désire portabilité et indépendance du langage (remplacer la partie cliente par une interface C++ par exemple).

XML est ce format neutre. Nos servlets sont accessibles par HTTP ; maintenant, nous allons faire en sorte qu'elles soient capables d'interpréter des requêtes comportant des paramètres XML et bien entendu de renvoyer du XML en sortie.

On voit apparaître clairement deux phases au cours desquelles il y a transformation sous forme d'objets des paramètres transmis (« désérialisation ») et transformation en XML d'objets Java (« sérialisation »). Notre architecture est donc dans l'esprit comparable aux architectures d'objets distribués type Corba (Common Object Request Broker Architecture).

Maintenant se pose la question : comment sérialiser des objets Java en XML ?

APARTÉ **Sérialisation**

Dans le monde Corba, cette étape s'appelle le *marshalling*, le pendant de la désérialisation étant appelé *unmarshalling*. Ces mots sont repris dans d'autres vocables comme dans celui de SOAP. C'est le nom d'une technologie qui facilite la publication d'objets sur le Web : les Web Services. Cette étape permet de transformer un objet en flux ; elle a bien sûr son pendant (la désérialisation), qui transforme un flux (fichier texte) en un objet (Java ou C++).

Sérialisation d'objets Java

C'est un sujet brûlant puisqu'il est à la base de bien des préoccupations de l'industrie. En effet, la sérialisation d'objets est utilisée intensivement dans les serveurs d'applications (EJB), les Services web (SOAP), etc.

L'équipe BlueWeb a opté pour une bibliothèque nommée Castor, permettant de créer un pont entre le monde objet et XML, et ce à un faible coût en termes de performances. Cette bibliothèque manipulée dans la servlet et dans la couche cliente se doit en effet d'être économe en ressources (mémoire et CPU).

OUTIL **Castor**

Castor est un produit sous licence BSD qui s'avère agréable, stable, bien maintenu et rapide. Si l'on ajoute une documentation de qualité, alors pourquoi s'en priver ?

- http://castor.exolab.org

Castor propose deux types de sérialisation/désérialisation :

- une première, totalement dynamique, basée sur l'introspection Java ;
- une seconde méthode utilisant un fichier de mapping XML guidant la conversion.

B.A.-BA Introspection

L'introspection est un des atouts de Java qui permet dynamiquement de découvrir puis d'utiliser des objets. Les principales classes intervenant dans ce mécanisme sont : `java.lang.Class`, `java.lang.ClassLoader` et le paquetage `java.lang.reflect`. Vous trouverez ci-après un exemple de code mettant en jeu ces mécanismes. Attention, l'introspection est un outil qui fait très bien son travail, mais qui doit être limité à une utilisation ponctuelle pour des morceaux de code nécessitant ce degré de liberté, car cette technique permet d'outrepasser toutes les règles objets de visibilité, etc.

La première méthode a l'avantage d'être très simple à mettre en œuvre (puisqu'elle ne nécessite rien d'autre que le *bytecode* de la classe à traiter).

Cet exemple montre comment charger dynamiquement une classe par le biais de la classe `Class` et de la méthode `forName()`.Puis comment invoquer une méthode en connaissant son nom et sa signature exacte (arguments). Il faut toutefois bien spécifier aux programmeurs débutants que cette méthode extrêmement puissante n'a de sens que dans certains contextes et qu'elle n'a quasiment plus rien d'objet puisqu'elle permet d'outrepasser les principes de visibilité de méthodes ou d'attributs.

```
package com.blueweb.test.castor;
import java.lang.reflect.Method;
/**
* @author J.MOLIERE - 2003/01
* <p>
* Une classe mettant en oeuvre l'introspection en Java.
* Donnée à titre informatif, elle ne fait rien de
* spectaculaire et ne montre pas toutes les possibilités de ce
* paquetage
* </p>
*/
public class TestIntrospection {
   static class Base{
      public void foo(){
         System.err.println("Base::foo");
      }
   }

   static class Heritiere extends Base{
      public void foo(){
         System.err.println("Heritee:foo");
      }
   }

   /**
    * <p>
    * méthode main.
    * ne s'occupe pas des exceptions..
    * </p>
    */
```

```
   public static void main(String[] args) throws Exception{
      Class dyn_class = Heritiere.class;
      // on demande à cette instance de nous donner sa classe mère
      Class super_class = dyn_class.getSuperclass();
      // affiche dans la console la super-classe de la classe
      // héritière
      System.err.println("Super classe = " + super_class.getName());
      // instancie la classe héritière puis appelle la méthode foo()
      // c'est une invocation classique régie par le compilateur
      Heritiere obj_instance = (Heritiere)dyn_class.newInstance();
      obj_instance.foo();
      // invoque foo() dynamiquement
      // la méthode foo n'a pas de paramètres d'où le null
      // voir javadoc...

      // attention ici aucun garde-fou
      // le compilateur ne peut vous protéger
      Method method_foo = dyn_class.getDeclaredMethod("foo",null);
      method_foo.invoke( dyn_class.newInstance(),null);


   }
}
```

On va utiliser la méta-classe `Class` pour :
- instancier dynamiquement un objet ;
- appeler un service sur cet objet ;
- récupérer et afficher sa super-classe.

Utilisation typique de Castor/XML

La figure 4-6 permet de visualiser les grandes étapes d'un processus de désérialisation (transformation XML vers Java). Elle se base sur l'utilisation d'un fichier de mapping (deuxième méthode).

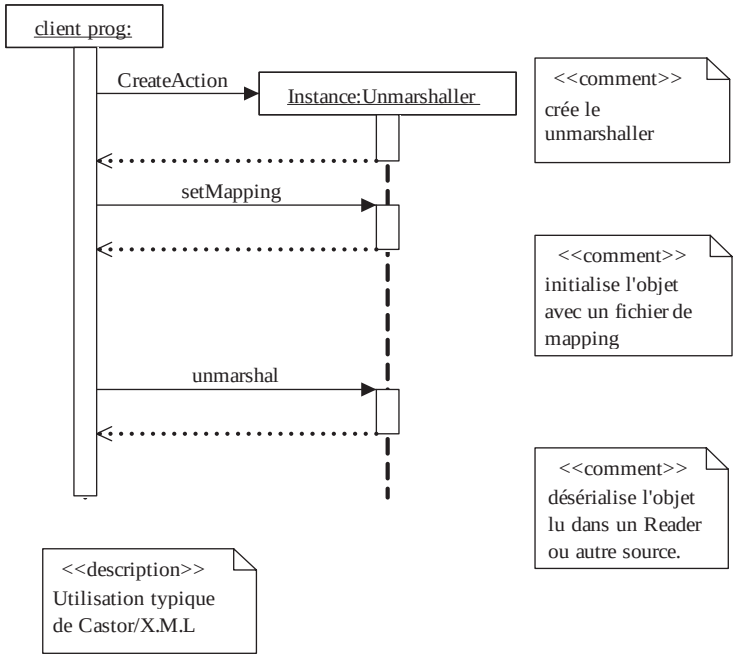


Figure 4–6
Utilisation typique de Castor XML

Soit un code du type :

Utilisation basique de Castor XML

```
package com.blueweb.test.castor;

import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Reader;
import java.io.Writer;
import org.exolab.castor.xml.MarshalException;
import org.exolab.castor.xml.Marshaller;
import org.exolab.castor.xml.Unmarshaller;
import org.exolab.castor.xml.ValidationException;

/**
* @author BlueWeb - 2003
* <p>
* Une classe testant Castor.
* Ce test reste très simple intentionnellement.
* Utilise la première forme de sérialisation/désérialisation possible:
* celle basée sur l'introspection Java (donc pas de fichier de
* mapping). On notera le fait que Castor dépende de xerces (xmlAPI.jar
* et xercesImpl.jar)
* On peut aussi noter que l'on peut utiliser les méthodes statiques
* des classes Marshaller et Unmarshaller plutôt que de les instancier…
* </p>
*/

public class MarshalUnmarshalClass {
   /**
    * <p>
    * ignore les exceptions, dans ce contexte d'un petit
    * bout de code de test.
    * </p>
    */
   public static void main(String[] args) throws IOException,
  IOException, MarshalException, MarshalException, ValidationException {
      // crée Toto, qui vient d'avoir 18 ans...
      DummyClass test_object = new DummyClass();
      test_object.setAge(18);

      test_object.setName("toto");
      // pour fêter cela, on va sauvegarder ces informations
      // dans un fichier XML (c\temp\castor-out.xml)
      Writer writer = new FileWriter("c:\\temp\\castor-out.xml");
      Marshaller marshaller = new Marshaller(writer);
      marshaller.marshal(test_object);
      writer.close();
```

Import des classes utilisées.

REMARQUE Adaptation du code

Il est laissé à la charge du lecteur le soin de modifier le chemin (`c:\temp\castor-out.xml`) afin de l'adapter à son système, ainsi que la tâche de modifier son environnement pour faire tourner l'exemple…

On instancie un objet nous servant de référence pour nos tests ultérieurs. Cet objet est une instance de la classe `DummyClass`, qui est réduite à quelques champs et aux accesseurs associés (nom et âge) .

Le résultat de la sérialisation de cet objet par Castor sera stocké dans un fichier, ici sous Windows avec un chemin `c:\temp\castor-out.xml`. Bien entendu, les « Unixiens » devront adapter ce chemin et les amateurs de Windows devront veiller à créer un répertoire `temp` sous la racine du lecteur C.

```
      // on va quand même s'assurer que l'objet a été bien sauvegardé
      Reader reader = new FileReader("c:\\temp\\castor-out.xml");
      // on peut remarquer ici le raccourci d'écriture
      // DummyClass.class
      Unmarshaller unmarshaller = new Unmarshaller(DummyClass.class);
      DummyClass read_object = (DummyClass)unmarshaller.unmarshal(reader);
      System.err.println("Objet lu a un nom = " +
          read_object.getName() + " et un age =" +
          read_object.getAge());

   }
}
```

Rien de tel qu'une vérification du résultat… Pour cela, on va lire le fichier créé précédemment et utiliser un `Unmarshaller` pour créer une instance de notre classe originale (`DummyClass`). On vérifiera la cohérence des informations stockées dans le fichier et celles de l'objet de test en examinant la console…

La classe `DummyClass` est une simple classe sérialisable (donc transportable sur un réseau ou sur un système de fichiers) dont le code est fourni ci-après :

```
package com.blueweb.test.castor;
import java.io.Serializable;

public class DummyClass implements Serializable {


   public DummyClass() {
      super();
   }

   private String name;

   private int age;
   public int getAge() {
      return age;
   }

   public String getName() {
      return name;
   }

   public void setAge(int age) {
      this.age = age;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

Une petite classe de test contenant quelques champs : elle est sérialisable (voir `java.io.Serializable`).

Constructeur de `DummyClass`.

Renvoie l'âge (`int`).

Renvoie le nom (`string`).

Instancie l'âge : le paramètre `age` contient l'âge à instancier.

Instancie le nom : le paramètre `name` contient le nom à instancier.

Maintenant, regardons comment utiliser l'autre forme de transformation proposée par Castor. Ici, il va s'agir d'utiliser un fichier de *mapping* guidant la transformation plutôt que d'utiliser la découverte dynamique des attributs proposée par Java.

```
package com.blueweb.test.castor;

import java.io.FileReader;
import java.io.FileWriter;
import java.io.Reader;
import java.io.Writer;
import org.exolab.castor.mapping.Mapping;
import org.exolab.castor.xml.Marshaller;
import org.exolab.castor.xml.Unmarshaller;
/**
* @author BlueWeb
*/
public class UtilisationMapping {
   private final static String OUT_FILE= "c:\\temp\\out-castor-2.xml";
   private final static String MAPPING_FILE ="c:\\temp\\mapping.xml";
   public static void main(String[] args) throws Exception {

      DummyClass toto = new DummyClass();
      toto.setName("toto");
      toto.setAge( 18);

      Writer writer = new FileWriter(OUT_FILE);
      Marshaller marshaller = new Marshaller(writer);

      Mapping mapping_file = new Mapping();
      mapping_file.loadMapping(MAPPING_FILE);
      marshaller.setMapping(mapping_file);

      marshaller.marshal(toto);

      Reader reader = new FileReader(OUT_FILE);
      Unmarshaller unmarshaller = new Unmarshaller(DummyClass.class);
      unmarshaller.setMapping(mapping_file);
      try{
         Object obj_from_stream = unmarshaller.unmarshal(reader);
         System.err.println("Classe de l'objet lu = " +
             obj_from_stream.getClass().getName());
         DummyClass read_object =(DummyClass)obj_from_stream;
         System.err.println("Nom de l'objet lu = " +
             read_object.getName()+ " Age = " + read_object.getAge() );
      }
      catch(Exception e){
         e.printStackTrace();
      }
   }
}
```

Utilise un fichier de mapping pour guider la transformation Castor. Ici il est impératif de manipuler les méthodes d'instance et non les simples méthodes statiques.

Toto a toujours 18 ans.

On sauvegarde...

Ici on charge un fichier de mapping.

Puis on sérialise.

On teste si cela a fonctionné.

On peut utiliser un fichier de mapping comme celui-ci :

```
<?xml version="1.0"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Object Mapping DTD
    Version 1.0//EN"   "http://castor.exolab.org/mapping.dtd">
<mapping>
    <class name="com.blueweb.test.castor.DummyClass" access="shared">
            <map-to xml="dummy-class"/>
            <field name="Age" type="integer">
                    <bind-xml name="age" node="attribute"/>
            </field>
            <field name="Name" type="java.lang.String">
                    <bind-xml name="name" node="attribute"/>
            </field>
    </class>
</mapping>
```

Bien entendu, ce fichier est un des plus basiques possible, Castor permettant de faire des choses très puissantes grâce à ce fichier (et à ses API).

Choisir un mode de sérialisation

Une fois maîtrisées, il faut choisir entre les deux techniques et ce n'est pas une chose simple, car vient se poser l'éternelle question (dilemme) entre généricité et rapidité. En effet, comme toute opération basée sur l'introspection Java, la première option se veut indépendante de tout fichier et peut donc s'adapter à n'importe quel contexte (donc rapide à mettre en oeuvre et simple) mais s'avère plus lente que la seconde option utilisant un fichier de mapping.

Pour BlueWeb, un autre argument prend encore plus d'importance et devient déterminant dans le choix de la seconde solution. En effet, jusque-là nous avons passé sous silence que l'argument lié à l'introspection est le fait de s'adapter à toute situation (n'importe quel objet Java ou document XML) mais que se passe-t-il dans notre applicatif si cette étape n'est pas maîtrisée ? Cet argument n'a aucun intérêt et se transforme peut-être en inconvénient majeur car l'équipe de BlueWeb tient à garder une parfaite maîtrise sur le fonctionnement de son application… En plus, la différence de performances permet de ne pas payer trop chèrement ces étapes ajoutées artificiellement de sérialisation/désérialisation.

Voilà, avec tous ces éléments, nous serons en mesure de mettre en œuvre au sein d'une servlet et au niveau du client la « tuyauterie » nécessaire à une sérialisation Java ↔ XML.

Il convient de veiller à divers aspects lors de tels choix techniques :

- volume des données transportées (devient un handicap dans le cas d'applications distantes et de médias à faibles bandes passantes) ;
- temps processeur requis par ces transformations (qui est un obstacle à la montée en charge du côte serveur) ;
- facilité d'utilisation et impact sur votre code – il est en effet dangereux de modifier vos objets pour les adapter à la façon dont ils vont être sérialisés, car que se passera-t-il si vous changez de produit ?

PERFORMANCES Castor

On recommande au lecteur de se livrer à un petit test avec un chronomètre (2 `System.currentTimeMillis()`) et de sérialiser/désérialiser une masse importante de données (disons 5 000 objets), et ce en utilisant les deux techniques.

Une autre approche avec SOAP

Le monde du Web ne parle plus que de SOAP (Simple Object Access Protocol), alors pourquoi ne pas l'utiliser ? L'équipe BlueWeb ne vient-elle pas de réinventer la roue avec ce framework utilisé sur le protocole HTTP puisqu'il s'agit d'une forme de distribution d'objets sur HTTP via XML, ce qui est précisément le but de SOAP. En effet, la finalité est la même, mais écarter SOAP de la discussion ne veut pas dire ne pas s'en préoccuper.

SOAP est une solution qui n'est que l'avant-coureur des *web services* de demain. Séduisante dans l'esprit, cette solution n'est pas exempte de problèmes :

- lente car trop verbeuse pour une réelle utilisation sur Internet ;
- trop générique donc très limitative en termes de programmation (pas de support des collections) ;
- prise en charge de la sérialisation XML côté client vite problématique (écriture des *serializers* quasi obligatoire) ;
- promesse de découverte dynamique des services loin d'être tenue.

Il reste que SOAP tient ses promesses au moins dans le domaine du développement côté serveur, puisque le code serveur n'est en rien lié à SOAP (juste déployé en tant que service web).

La bonne solution actuellement au niveau des implémentations SOAP se situe peut-être au niveau de Glue, qui dispose de nombreux atouts par rapport à ses concurrents directs, dont une véritable aide au développement, des performances bien plus acceptables (largement supérieures à Axis) et une bonne documentation. Mais ce produit qui, malheureusement, est amené à disparaître, est doté d'une licence très spéciale pouvant être une entrave à son utilisation.

RÉFÉRENCE

L. Maesano, C. Bernard, X. Le Galles, *Services Web en J2EE et .NET – Conception et implémentation*, Eyrolles 2003.

OUTIL **Axis**

L'ancien projet Apache-SOAP est maintenant (depuis la sortie de la version 1.0) devenu Axis. Ce projet permet de tester SOAP en s'intégrant dans un moteur de servlet du type Tomcat.

- http://xml.apache.org/axis/

OUTIL **Glue**

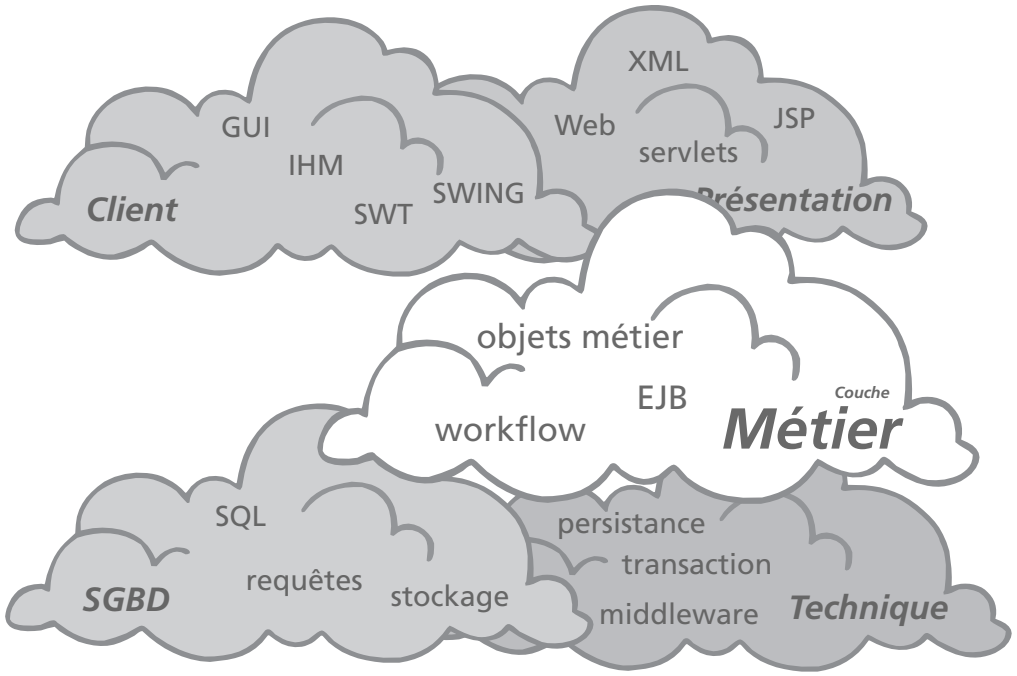
- http://www.webmethods.com

En résumé…

L'ouverture et la portabilité impliquent fatalement des répercussions sur la vitesse de réponse d'une application, mais le jeu en valant la chandelle, il est important de tirer parti de bibliothèques performantes pour amener à notre application cette ouverture sur l'extérieur si précieuse, et ce sans détériorer trop les performances. L'utilisation de servlets pour assurer cette fonction de présentation est assez intéressante car c'est une technologie assez simple, robuste et très efficace. En la conjuguant avec le design pattern Commande on vient de mettre en place à peu de frais un système évolutif et assez élégant. Voilà de quoi relever le challenge des *web services*. Au hasard d'un regard un peu attentif sur le pattern Commande, nous avons pu nous intéresser à une des forces de JBoss : les Intercepteurs.

L'examen du problème de la sérialisation d'objets nous a donné l'occasion de faire un petit détour par la problématique des *web Services* et de constater à quel point il est regrettable de n'en être qu'aux tâtonnements de ce qui est la technologie du futur (sûrement proche) : SOAP. Les lecteurs intéressés par le sujet peuvent voir en XML-RPC un compromis plus réaliste, puisque cette technologie offre des performances supérieures à celles de SOAP mais au détriment des ambitions affichées…

chapitre 5


XML
GUI
Web
JSP
servlets
IHM
SWING
Client
SWT
Présentation
objets métier
EJB
Couche
workflow
Métier
SQL
persistance
transaction
requêtes
stockage
SGBD
middleware
Technique

Couche métier avec les EJB

Le développement d'EJB pour coder l'implémentation de la couche métier peut se révéler très laborieux si l'on n'utilise pas d'outils nous permettant de nous concentrer sur « l'essentiel » sans se perdre dans le fatras de détails. Ce chapitre présente de tels outils tout en donnant les bases du développement d'EJB.

SOMMAIRE

MOTS-CLÉS

- EJB
- Logique métier
- Génération de code automatique
- Doclet
- XMI
- JBoss
- XDoclet

Rappels sur les EJB

Cette section est consacrée aux rappels des principes fondamentaux régissant le codage des composants métier via la norme EJB. Il ne s'agit que d'un guide opératoire et non d'un tutoriel dont le volume et la complexité dépasseraient largement le cadre de cet ouvrage.

Voici les différents types de composants mis à disposition par la spécification EJB (version 2.1) :

- **Les beans de type session.** Ce sont des composants dans lesquels vous allez coder la logique métier relative à tel ou tel sujet. Ils ont un rôle de chef d'orchestre et vont souvent gérer les interactions entre plusieurs composants de type entité. Ils peuvent être du type `stateful` (avec état), c'est-à-dire permettant de simuler une notion de contexte utilisateur, ou alors `stateless` (sans état), c'est-à-dire que d'une invocation à l'autre notre composant est recyclé par le serveur d'applications (et donc réinitialisé).
- **Les beans de type entité**. Ils correspondent à une ligne d'une table d'une base de données et représentent des entités de notre modèle physique de données. Ces données persistantes peuvent, elles aussi, être de deux types. Avec le type CMP (Container Managed Persistency), le conteneur se charge seul de la lecture/stockage de notre entité en base de données. Avec le type BMP (Bean Managed Persistency), c'est au développeur que revient la charge de la persistance des données. Ce type de composants est développé via une utilisation massive de JDBC.
- **Les clients JMS**. Ce nouveau type de composants permet de créer des clients de files de messages professionnelles telles que MQ-Series (IBM) ou MS-MQ (Microsoft). Ces clients seront abonnés à des sujets proposés par moteur de messages.

Nous allons établir ici une liste succincte des différents acteurs que nous devrons coder, tout en rappelant le rôle de chacun d'entre eux au sein de l'ensemble. Pour faciliter ce rappel, nous utiliserons l'exemple classique du compte bancaire avec une classe dénommée `Compte`.

- Classe implémentation : `CompteBean`, c'est la classe dans laquelle nous devons placer le code correspondant à la logique métier (comment se traduit une ouverture de compte, une fermeture de compte ou encore un débit d'une somme sur ce compte).
- `Home interface` : `CompteHome` permet de localiser et de manipuler (`findAll` ou `findByPrimaryKey`) les instances de notre classe `Compte`.
- `RemoteInterface` : c'est cette interface qui va être manipulée par le client. Elle contient donc la liste de tous les services disponibles (implémentés dans la classe `CompteBean`).
- Le fichier `ejb-jar.xml`, ou fichier de description de déploiement (*deployment-descriptor*), permet d'empaqueter les composants en une unité (fichier `jar`) qui va être déployée au sein du serveur d'applications. C'est dans ce

RÉFÉRENCE

L'excellent ouvrage *Mastering EJB* (Roman *et al.*) traduit sous le titre *EJB fondamental* (Eyrolles 2002) peut être téléchargé en anglais au format PDF sur le site suivant :

- http://www.theserverside.com

ATTENTION **Portabilité du code**

Il faut bien comprendre que dans le lot de fichiers nécessaires au déploiement d'un EJB, une partie seulement de ces fichiers est assurée d'être réutilisable d'un conteneur à un autre. Suivant le serveur d'applications utilisé, vous devrez adapter une série de fichiers de configuration afin de pouvoir déployer vos composants. Vous devez bien comprendre cette notion si vous devez assumer une phase de migration de serveur d'applications et vous devrez à ce moment-là être à même de quantifier les délais nécessaires à cette migration.

fichier que l'on va énoncer les contraintes de sécurité vis-à-vis de nos composants, ainsi que les contraintes liées à la gestion des transactions (voir section *assembly-descriptor* de ce fichier).

Ceci représente la partie portable et indépendante de la version de la spécification utilisée lors du développement/déploiement, c'est-à-dire la partie minimale du travail à effectuer, et ce pour chaque composant (même si en fait on n'a en vérité qu'un seul fichier `ejb-jar.xml` par application, il faut quand même le mettre à jour).

Maintenant, il faut constater que chaque serveur d'applications (JBoss, Weblogic, Websphere…) requiert d'autres informations avant de déployer votre composant. C'est la part non portable, d'un serveur à l'autre, du travail à effectuer.

Ceci fait, nous pouvons passer au déploiement en lui-même, qui bien entendu n'est pas normalisé (donc chaque serveur d'applications proposera sa façon de faire…). Ceci peut aller de l'application graphique pour des serveurs comme Websphere à la simple copie de fichiers dans un endroit donné (pour JBoss, voir répertoire `deploy`).

Ceci peut se traduire en résumé par un schéma directeur du développement cadencé en trois étapes :

1. codage des composants ;
2. modification/création du fichier de description du déploiement ;
3. *Packaging*/déploiement.

Maintenant que ces grandes étapes sont rappelées, revenons sur la partie codage…

Les trois classes énoncées ci-dessus nécessaires au codage d'un composant, sont en fait souvent plutôt juste la partie basique nécessaire, mais elles peuvent être complétées par d'autres classes, à savoir :

- `CompteData`, une classe dite *holder* ou *value object*, permettant de rassembler en un seul élément transportable par le réseau toute l'information nécessaire à la caractérisation de l'instance spécifique du composant auquel elle est rattachée. Cela signifie que l'on va rassembler, sous forme d'une instance d'une classe `CompteData`, toute l'information caractérisant une instance de la classe `Compte`. Pour plus d'informations sur ce procédé, se reporter à l'encadré dédié à cet effet.
- Pour les composants déployés dans des conteneurs EJB appliquant la spécification EJB 2.1, la norme CMP 2.1 (*Container Managed Persistence*) permet de créer des vues non plus seulement des nos objets destinés à une utilisation via le réseau (sérialisation par le protocole RMI/IIOP), mais aussi des objets dits « locaux ». Ces objets pourront être manipulés par référence (pointeur direct comme dans tout programme Java). Ceci implique un gain de performance mais nous oblige à coder deux autres classes : `CompteLocal` et `CompteLocalHome`, qui seront donc la `LocalInterface` (pendant de la `RemoteInterface`) et la `LocalHome interface` (pendant de la `Home interface`).

LÉGENDE URBAINE Les outils graphiques

Il s'agit d'un piège trop souvent utilisé par les vendeurs d'outils, qui sont prompts à proposer de nombreux outils graphiques avec lesquels en quelques clics de souris vous pouvez faire ceci ou cela. Attention, ce type d'outils est extrêmement dangereux, car ils sont de réelles entraves à la bonne compréhension des mécanismes sous-jacents et rendent impossible l'automatisation de certaines tâches d'administration. L'avantage que peut y trouver un débutant, s'avère vite être une entrave pour le professionnel. En revenant au cas précis des outils graphiques permettant le déploiement des EJB dans un serveur d'applications, il faut bien comprendre que ce type de solutions s'avère problématique lorsqu'il s'agit de redéployer à distance une version de vos composants. Ce cas de figure est exactement celui que vous rencontrerez en clientèle et il faut anticiper ce type de situations. L'utilisation d'un script est une solution beaucoup plus simple à mettre en œuvre pratiquement, car vous pourrez aisément demander un accès par `ssh`, vous permettant de vous connecter à distance et de lancer votre script. La mécanique de déploiement d'EJB offerte par JBoss est d'une efficacité et d'une simplicité exemplaires, même si elle semble manquer de belles interfaces graphiques.

ATTENTION Compatibilité de votre serveur d'applications avec une version des spécifications

Quasiment tout est dit dans le titre…Il faut faire bien attention aux déboires rencontrés lors d'un changement de serveur d'applications, car certains produits, comme Websphere, ne prennent en charge à l'heure actuelle que la version 1.1 des spécifications des EJB. La migration d'un produit acceptant les EJB 2.0 vers un produit n'appliquant que les EJB 1.1 ne se fait pas sans douleurs.

DESIGN PATTERN ValueObject

Le design pattern *ValueObject* voit ses origines dans les premiers frameworks de distribution d'objets sur un bus de données comme Corba. La problématique sous-jacente à la création de ce design pattern n'est donc pas récente et ce principe gouverne toutes les architectures utilisant des objets distribués. Nous sommes là dans un contexte où le client et le serveur sont dans l'absolu situés sur des machines différentes et les communications entre ces deux composants soumises à une contrainte non négligeable : le média de transport (connexion téléphonique, NUMERIS, fibre optique ou réseau 10 Mb sont autant de conditions différentes d'exécution en termes de bande passante et donc de saturation). Admettons que dans le contexte de l'objet `Compte`, chaque compte ait un attribut avec le nom du titulaire de ce compte (type `string`), un identifiant (entier sur 10 chiffres), un découvert maximum autorisé (type `float`) et un champ remarque (`string`) permettant au chargé de compte de stocker divers commentaires. Une application de gestion pourrait avoir besoin pour un affichage, du nom du titulaire et de l'identifiant du compte, ainsi que du découvert maximal autorisé… Naïvement, on pourrait penser depuis le client faire `getNom()`, puis `getId()`, puis enfin `getMaxAut()` sur la référence de notre objet `Compte`. Hélas, c'est une approche trop naïve !!! En effet cette façon de faire aurait l'effet désastreux de provoquer trois requêtes sur notre réseau (une pour chaque appel du type `getAttribut()`). L'idée est donc simple, il nous suffit de concentrer cette information en une seule et même classe puis de remplacer notre série d'appels par : `getValueObject()`.

DESIGN Pourquoi choisir l'utilisation d'entités en mode CMP ?

Cette question vaut un éclaircissement. Premièrement, il est logique de commencer par essayer de tirer parti pleinement de la puissance de son conteneur et d'essayer de minimiser son travail. L'écriture d'un code ayant trait au domaine de la persistance et des bases de données est une tâche ardue, voire un challenge si l'on cherche à ne pas se coupler à un moteur de base de données. L'écriture de code performant est difficile dans le sens où ce domaine technique est issu directement des mathématiques ensemblistes. Le désir d'indépendance par rapport à la base devient vite une contrainte si l'on considère la pauvreté de la norme ANSI SQL 92. Tout cela montre qu'une telle tâche ne doit pas être commencée à la légère. De plus, le conteneur peut bénéficier d'avantages considérables en termes de performances avec ce qu'il est convenu d'appeler le problème des « (n+1) requêtes » (de la documentation est disponible sur de nombreux sites ou forums pour expliquer plus avant ce problème). Deuxièmement, le fait de commencer par utiliser un tel composant n'implique pas le déploiement au final de ce composant, car si vous vous heurtez à un des cas de figure pour lesquels l'approche par CMP s'avère problématique, il ne tient qu'à vous de remplacer cette classe par une classe assurant elle-même sa persistance (BMP). Il s'agit là encore d'une approche pragmatique, peut-être décriable aux yeux de certains intégristes, mais qui doit vous permettre de trouver le juste équilibre entre performance et coût de développement.

Un petit exemple de bean de session : un convertisseur d'euros

Tout cela étant bien théorique jusque-là, rapprochons-nous du code par le biais d'un exemple très simple, mais qui nous permettra de mettre en évidence le travail nécessaire au déploiement d'un EJB dans un conteneur. Reprenons l'exemple utilisé lors du chapitre 3 : le convertisseur monétaire, euros vers francs et vice versa. Il peut être parfaitement justifié, dans le cadre d'une application amenée à manipuler des monnaies, d'avoir à utiliser un composant de ce type.

Quel est le code nécessaire pour réaliser ce composant ? Quels sont les fichiers guidant le déploiement ?

Implémentation des services

Ce bean de session très simple propose un seul service, le calcul d'une conversion franc vers euro ou euro vers franc. Il n'a d'autre utilité que de montrer l'énorme travail que requiert le déploiement d'un EJB. Bien entendu, étant donné la nature simpliste de ce composant, il n'a pas d'interactions avec l'extérieur, et ne nécessite donc pas de dialogue avec d'autres EJB...

Classe d'implémentation de notre EJB convertisseur d'euros

Dans un cas aussi simple, l'implémentation des services l'est forcément aussi… Ici on se contente d'une multiplication par le taux de base de conversion…
Cette classe est la classe d'implémentation des services de notre composant (`conversions`). On peut se demander si ce n'est pas la seule utile...

```
package test.ejb;
import javax.ejb.*;
public class ExempleSessionBean implements SessionBean {
       private float tauxDeBase = 6.55957f;
    /**
     * Calcul euro vers franc
     * @param aConvertir, montant à convertir (euros) vers des francs
     * @return somme convertie en francs
     */
```

```
public float euroToFranc(float aConvertir) {
  System.err.println("conversion de " + aConvertir + " euros vers des
francs");
            return aConvertir*tauxDeBase;
    }
    /**
     * Calcul de la conversion francs vers euros
     * @param aConvertir, montant en francs a convertir en euros
     * @return somme convertie en euros
     */
public float francToEuro(float aConvertir){
             return aConvertir/tauxDeBase;
    }
    public void ejbActivate() {
    }
    public void ejbPassivate() {
    }
    public void setSessionContext(SessionContext ctx) {
    }
    public void ejbRemove() {
    }
    public void ejbCreate(){
    }
}
```

Les méthodes suivantes sont en rapport avec le cycle de vie des objets au sein du conteneur EJB : activation, passivation, création ou destruction. Il n'y a rien à faire dans les cas d'activation, passivation, création ou destruction, de même que le contexte est ici indifférent.

Cette classe a pour vocation de fournir l'implémentation des services offerts par notre composant. Les méthodes de conversion ont été mises en évidence, le restant du code est la « plomberie » classique dans ce contexte des EJB. Les méthodes `ejbRemove()`, `ejbActivate()` et affiliées seront implémentées pour nous par le conteneur EJB. Le seul contexte induisant une implémentation explicite de ces méthodes est celui d'entités en mode BMP.

RAPPEL Bean Managed Persistence

En mode BMP, il incombe au développeur d'assurer lui-même la persistance de ses objets.

Les services étant très simples, les commentaires sont quasiment superflus. Il suffit d'opérer une multiplication (division) par notre taux de conversion puis de retourner cette valeur. Rien de plus simple…

Home Interface

Cette classe permettra d'appeler des services basiques (créer un objet par exemple). Elle est essentielle mais n'amène aucune nouveauté (surtout dans le cadre d'un EJB session). Dans le cadre d'un EJB entité, on y trouverait des méthodes permettant de trouver un objet en particulier, ces méthodes sont dites `finder` dans le jargon… Le but de ce code est de montrer l'aspect rébarbatif du codage d'EJB à la main : le code ci-après ne contient pas la moindre ligne de code Java utile et des déclarations redondantes par rapport au code précédemment exposé. Tout cela pour si peu !

Home interface rattachée à notre EJB

```
package test.ejb;

public interface ExempleSessionHome
extends javax.ejb.EJBHome
{
   public static final String COMP_NAME="java:comp/env/ejb/
                                        ➥ ExempleSession";
   public static final String JNDI_NAME="ejb/exemple/session";

public test.ejb.ExempleSessionRemote create()
      throws javax.ejb.CreateException,java.rmi.RemoteException;
}
```

Home interface du bean `ExempleSession`. Peu de services dans le cadre d'un EJB session, si ce n'est la seule méthode `create()`.

Cette méthode crée et renvoie un objet du type `ExempleSessionRemote`. Il faudra nécessairement appeler cette méthode sur un objet du type `ExempleSessionHome` avant de pouvoir appeler nos services de conversion, et ce afin de manipuler un proxy correct.

Le commentaire général est édifiant : pas la moindre trace de code utile, uniquement des déclarations pouvant être faites automatiquement…

Remote Interface (interface distante)

Cette classe permettra d'invoquer les services de notre objet à distance, puisqu'il faut bien garder à l'esprit qu'un client EJB peut être situé à n'importe quel endroit du réseau (donc sur une toute autre machine que celle hébergeant le conteneur EJB). Notre proxy expose les mêmes services que ceux définis dans le composant d'implémentation (calcul franc vers euro et euro vers franc).

```
package test.ejb;
public interface ExempleSessionRemote
extends javax.ejb.EJBObject{



   public float euroToFranc( float aConvertir )
      throws java.rmi.RemoteException;
   public float francToEuro( float aConvertir )
      throws java.rmi.RemoteException;
}
```

Il s'agit du proxy nous permettant d'invoquer les méthodes offertes par notre objet distribué sur le réseau. On ne s'étonne donc pas de retrouver les déclarations de nos deux services `euroToFranc()` et `francToEuro()`.

Les méthodes sont bien celles exposées par notre Bean. Notez la levée possible d'une exception de type `RemoteException`.

DESIGN PATTERN Proxy

Ce design pattern est très célèbre en raison de son utilisation systématique dans toute architecture où il y a distribution d'objets sur un réseau. En caricaturant, le principe de ce design pattern est d'utiliser un effet « Canada-dry » (« cela a la couleur de l'alcool, mais ce n'est pas de l'alcool »), c'est-à-dire de faire croire au client qu'il invoque un service sur un objet qu'il peut manipuler alors qu'en fait celui-ci ne fait que relayer la requête à un objet situé peut-être sur une autre machine. Ce principe permet de camoufler au programmeur les difficultés liées à la programmation en réseau : protocoles, séquencement des caractères (LITTLE-ENDIAN ou BIG-ENDIAN), etc.

La raison d'être de cette classe est de fournir un « proxy » de l'objet distribué sur le réseau (notre classe `ExempleSessionBean.java`).

Le design pattern Proxy

Le design pattern Proxy permet de mettre en œuvre un des principes majeurs utilisés en programmation objet : la délégation. Sans chercher à entrer dans l'éternel débat héritage/délégation, l'utilisation de la délégation a le mérite très clair de délester l'utilisateur de problèmes qui ne sont pas les siens. Ce pattern est utilisé systématiquement dans les architectures utilisant des objets distribués (Corba ou EJB par exemple) mais ne se cantonne pas à cette seule utilisation. Ainsi, le récent avènement de l'AOP (programmation orientée aspects) permet de comprendre tout l'intérêt de cette approche. On peut signaler en passant que JBoss est codé avec une telle approche et fait une consommation massive de proxies dynamiques.

On peut synthétiser le principe général de ce motif de conception par le diagramme de séquences UML de la figure 5-1.

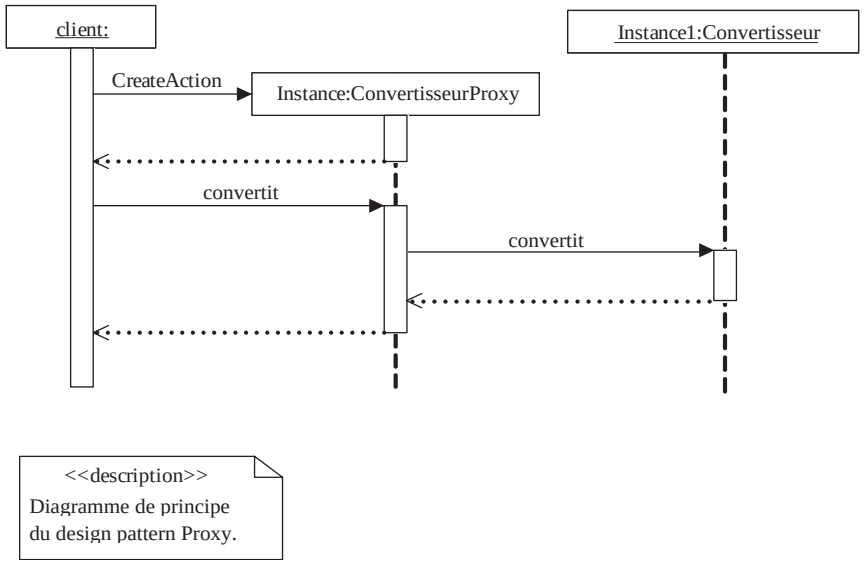


Figure 5–1
Le design pattern Proxy

L'intérêt majeur est de manipuler un objet (le proxy) qui encapsule la complexité du contexte et qui nous détache de l'implémentation utilisée finalement. Ainsi, si l'on change l'implémentation de la méthode `convertis()` dans notre classe `Convertisseur`, l'objet proxy sera identique et donc le code client pourra rester le même. Depuis le JDK 1.3, Java prend en charge ce concept de proxies avec notamment la classe `java.lang.reflect.Proxy`. Il est intéressant de montrer la puissance de ce mécanisme en action sur un exemple simple. Dans l'exemple suivant, nous allons ajouter des fonctionnalités élémentaires en matière de trace, et ce sans modifier notre classe centrale. Soit l'interface suivante :

Une interface très simple

```
package test.proxies;
public interface ISimple {
   public String getNom();
   public String getCommentaire();
}
```

Cette interface n'a d'intérêt que pour l'exemple et propose deux services élémentaires.

Il s'agit d'une interface quelconque, dont une implémentation type (très simple) est fournie ici :

Implémentation de notre interface ISimple

```
package test.proxies;
public class SimpleImpl implements ISimple {
   public String getNom() {
      System.err.println("SimpleImpl::getNom() je vais retourner le nom");
      return "Simple";
   }
```

Classe implémentant les services définis dans l'interface `ISimple`. On peut noter le nom `SimpleImpl`, où `Impl` signifie implémentation. Cette convention est souvent utilisée.

```
  public String getCommentaire() {
    // difficile de faire plus simple...
  System.err.println("SimpleImpl::getCommentaire() je vais retourner
                ➥ le commentaire");
    return "sans commentaire";
  }
}
```

Pour ajouter des fonctionnalités de trace basiques, il suffit de créer un proxy (ici statique pour rester simple) dont le code est le suivant :

Un proxy de trace

Classe de test montrant comment ajouter un niveau de trace basique à un objet par ce design pattern.

```
package test.proxies;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
```

Ce proxy va ajouter des fonctions de trace de très bas niveau. Chaque appel de méthode sur l'objet délégué par le biais de ce proxy se verra précédé d'un message ; de même qu'à la sortie de chaque méthode.

```
public class TraceurProxy implements InvocationHandler {
  // objet sur le lequel on agit
  // tous les appels à la méthode invoke() (voir plus bas)
  // induiront l'appel d'une des méthodes de cet objet.
  private Object delegue;

  // constructeur privé pour éviter des
  // appels directs
  private TraceurProxy(Object unObjetDelegue){
    delegue=unObjetDelegue;
  }
```

Il s'agit là d'une Factory : cette méthode sert à créer des références. Elle est indispensable vu le caractère privé du constructeur.

```
  /**
   * @param unObjetDelegue, objet vers lequel on va rediriger
   * les requêtes * @return Object, on va créer des objets modifiés
   * @see Proxy#newProxyInstance()
   */
public static Object createInstance(Object unObjetDelegue){
```

Ligne de code un peu brutale au premier abord… Il s'agit de créer un proxy ajoutant la fonctionnalité de trace basique. Pour cela, on utilise la classe standard Java (depuis le JDK 1.3) `Proxy`, en lui précisant le chargeur de classes à utiliser, l'interface implémentée par l'objet ainsi que la classe fournissant le type de proxy que nous attendons (ajout de traces).

```
  return Proxy.newProxyInstance(
    unObjetDelegue.getClass().getClassLoader(),
    unObjetDelegue.getClass().getInterfaces(),
    new TraceurProxy(unObjetDelegue));
  } // createInstance()

  /* @param unProxy, le proxy utilisé
   * @param uneMethode, la méthode invoquée par le client
   * @param args, arguments transmis
   * @see
   *   java.lang.reflect.InvocationHandler#invoke(java.lang.Object,
*   java.lang.reflect.Method, java.lang.Object[])*/
  public Object invoke(Object unProxy, Method uneMethode, Object[] args)
    throws Throwable {
    Object return_object = null;
    try{
```

```
      System.out.println("Je vais rentrer dans la méthode = " +
              uneMethode.getName());
         return_object=uneMethode.invoke(delegue,args);
         System.out.println("Je suis sorti de la méthode "+
              uneMethode.getName());
      }
      catch(Exception e){
         System.err.println("Une erreur inattendue s'est produite = " +
              e.getMessage());
      }
      return return_object;
   } // invoke()
}
```

Ici, grâce à la délégation, on peut se contenter d'encadrer l'appel (délégation) de la méthode par notre code de trace.

Pour illustrer tout cela, codons une classe de test mettant tout en œuvre.

Classe d'illustration de la mise en œuvre de proxies

```
package test.proxies;
/**
* Une classe principale montrant nos proxies en action...
* @author j.moliere
*/
public class Main {
   public static void main(String[] args) {

      ISimple simple= (ISimple) TraceurProxy.createInstance(
              new SimpleImpl());  // on appelle la méthode getNom()
      simple.getNom();
      // on appelle la méthode getCommentaire()
      simple.getCommentaire();
   } // main()
}
```

Ici l'objet simple est en fait un proxy sur une instance de la classe `SimpleImpl` (qui implémente l'interface `ISimple`). Ce proxy ajoute les fonctionnalités de trace espérées.

On crée un proxy en passant par cette méthode statique. On ajoute des fonctionnalités sans rien changer à l'utilisation de notre objet !
On est obligé d'utiliser l'opérateur de cast car le type statique de retour est `java.lang.Object`.
L'objet délégué est ici une instance de notre implémentation de l'interface `Isimple`.

En la lançant (par `java test.proxies.Main`), on obtient une sortie du type :

```
Je vais rentrer dans la méthode = getNom
SimpleImpl::getNom() je vais retourner le nom
Je suis sorti de la méthode getNom
Je vais rentrer dans la méthode =
getCommentaireSimpleImpl::getCommentaire() je vais retourner le
commentaire
Je suis sorti de la méthode getCommentaire
```

En bref et pour conclure sur cet aparté à propos du motif de conception Proxy, il faut retenir que par cette architecture, vous pouvez obtenir un exemple de conception vous camouflant une complexité parfois énorme (comme c'est le cas dans le contexte d'une utilisation d'un middleware comme Corba) et protégeant votre code client d'éventuelles modifications dans les objets délégués (puisque vous ne savez rien d'eux).

B.A.-BA AOP (Aspect Oriented Programming)

La programmation orientée aspects a pour but de simplifier la maintenance de code en séparant strictement logique métier et décorations techniques (code de traces, logging, authentification, etc.). C'est un domaine passionnant et relativement méconnu encore. Un outil Java proposant cette approche est librement disponible sur le site :

- http://www.aspectj.org.

R. Pawlak et al., *Programmation orientée aspect pour Java/J2EE*, Eyrolles, 2004.

Description du déploiement

Attention, ce fichier est validé avec une DTD officielle de Sun. Il permet de standardiser le déploiement et l'assemblage d'EJB au sein d'un conteneur EJB.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
```

Une première partie générale avec des informations de description…

```
<ejb-jar >
```

Descripteur de déploiement.

```
<description><![CDATA[No Description.]]></description>
  <display-name>Descripteur de déploiement pour notre bean de session
exemple</display-name>
   <enterprise-beans>
```

Cette section rassemble tous vos EJB (MDB, entités ou session).

```
<!-- Session Beans -->
      <!-- session réduite à un seul bean -->
      <session >
         <description><![CDATA[exemple bean session]]></description>
```

B.A.-BA JNDI

JNDI est une API de Sun dont la vocation est de résoudre les problèmes de recherche d'objets. Il s'agit de proposer une réponse à la problématique des services de nommage. On peut utiliser une analogie avec une bibliothèque où l'on associe à un nom d'ouvrage (disons : l'art de la guerre de Sun Tzu) un emplacement dans des rayons (par exemple B3). Une application utilisant des objets distribués va nécessiter le même type de services puisqu'une partie cliente va utiliser un nom d'objet de manière à le localiser, puis va invoquer une méthode (ou plusieurs) sur l'objet obtenu après recherche. La terminologie anglaise veut que l'on utilise le terme de « binding » pour désigner l'association faite entre un nom et un objet. Pour en revenir à notre exemple, on a associé au nom `ejb/test/ExempleSession` la classe Java `test.ejb.ExempleSessionBean`. Ce mécanisme vous est forcément familier (sans le savoir) car l'exemple le plus célèbre de service de nommage est sans aucun doute DNS, l'outil rendant possible l'utilisation simple d'Internet, puisqu'il permet d'associer à un nom (www.javasoft.com) une adresse IP (celle du serveur web honorant les requêtes pour ce nom de domaine). Dans un domaine plus proche des EJB, Corba dispose avec le COS d'un service de nommage très puissant mais assez difficile d'approche. On peut citer un autre type de service de nommage avec le protocole SMB de Microsoft, qui permet d'accéder à des machines au sein d'un réseau Microsoft (avec la convention de nom UNC du type \\nom_machine).

Pour les habitués de Java-RMI (objets distribués en environnement Java), il est bon de préciser que JNDI est une API très simple à mettre en œuvre puisqu'ils retrouveront la plupart des méthodes de l'objet `Naming` à travers l'objet `javax.naming.Context`.

Il est utile de préciser d'emblée qu'il faut voir JNDI plus comme une interface que comme une implémentation de service de nommage, dans le sens où cette API vous permet, en paramétrant votre contexte (`javax.naming.InitialContext`), de questionner des services de nommage tels que LDAP, NIS, COS. Ceci explique le fait que la première ligne d'un bloc de code dédié à la recherche d'objets via JNDI soit du type :

Figure 5–2 Un exemple de service de nommage en action : le voisinage réseau Microsoft

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.fscontext.RefFSContextFactory");
Context ctx = new InitialContext(env);
```

Ici env va désigner une table associative (hashtable) au sein de laquelle on définit une implémentation particulière permettant de créer des objets du type `InitialContext`. Cette section est à comparer avec le début du programme client permettant d'invoquer un EJB session déployé au sein de JBoss.

```
         <ejb-name>ExempleSession</ejb-name>
         <home>test.ejb.ExempleSessionHome</home>
         <remote>test.ejb.ExempleSessionRemote</remote>
         <ejb-class>test.ejb.ExempleSessionBean</ejb-class>
         <session-type>Stateless</session-type>
         <transaction-type>Container</transaction-type>
      </session>
      <!-- Entity Beans -->
      <!-- Message Driven Beans -->
   </enterprise-beans>
   <!-- Relationships -->
   <!-- Assembly Descriptor -->

   <assembly-descriptor >
     <!--To add additional assembly descriptor info here, add a file
to yourXDoclet merge directory called assembly-descriptor.xml that
contains the <assembly-descriptor></assembly-descriptor> markup.-->
   <!-- finder permissions -->
   <!-- transactions -->
   <!-- finder transactions -->
   </assembly-descriptor>
</ejb-jar>
```

Section vide dans notre cas exemple.

Pas d'EJB de ce type.

Cette section guide l'assemblage des EJB. Elle est ici réduite à néant car il n'y a qu'un seul EJB…

Ce fichier XML est standardisé par la norme J2EE. Il est nommé `ejb-jar.xml`). Il sera donc utilisé par n'importe quel conteneur EJB. Malheureusement, il ne suffit pas et doit être complété par un (ou plusieurs) fichier(s) spécifique(s) au conteneur que vous utilisez.

Dans le cas de JBoss, le fichier nécessaire est nommé jboss.xml) :

Fichier de description propre à JBoss jboss.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS 3.0//EN"
 "http://www.jboss.org/j2ee/dtd/jboss_3_0.dtd">
<jboss>
   <enterprise-beans>
    <!--
     To add beans that you have deployment descriptor info for, add
    a file to your XDoclet merge directory called jboss-beans.xml that
     contains the <session></session>, <entity></entity> and
     <message-driven></message-driven> markup for those beans.
    -->
      <session>
         <ejb-name>ExempleSession</ejb-name>
         <jndi-name>ejb/exemple/session</jndi-name>
         <configuration-name>Standard Stateless SessionBean
         </configuration-name>
      </session>
   </enterprise-beans>
   <resource-managers>
   </resource-managers>
</jboss>
```

À ce stade-là, il ne reste plus qu'à déployer l'application après l'avoir empaquetée.

Mise en œuvre de JBoss

Nous allons ici donner les grandes lignes permettant de préparer le déploiement de vos applications au sein de JBoss. Pour cela, il suffit de :

1. Télécharger la dernière version de JBoss (format `.zip` ou `.tar.gz`). Pour cela, le plus simple consiste à faire pointer votre navigateur web vers l'URL :
   - http://sourceforge.net/project/showfiles.php?group_id=22866.
2. Décompresser la version choisie sur votre machine. On ne peut que fortement conseiller d'utiliser un répertoire du type `c:\jboss` sous Windows (attention aux problèmes liés aux espaces dans les noms de répertoires, donc évitez `c:\Program Files\Jboss`).
3. Positionner une variable d'environnement `JBOSS_HOME` pointant vers le répertoire choisi précédemment. Par exemple sous Unix (shell `bash`), modifiez ou créez un `.bashrc` dans votre répertoire personnel (`HOME_DIR` ou `~`) avec une ligne du type : `export JBOSS_HOME=/usr/local/jboss`. Sous Windows, vous pouvez utiliser le panneau de configuration pour ajouter cette variable.
4. Utiliser le script de démarrage adapté à votre système (`run.sh` ou `run.bat`) dans le répertoire `bin` de la distribution JBoss (par exemple `c:\jboss\bin`).

Ça y est, JBoss est prêt à fonctionner. Comment s'assurer qu'il fonctionne correctement ? C'est assez simple, il suffit de tester que certains ports standards répondent correctement :

- Sous Windows ou Unix, vous pouvez utiliser le programme `telnet`. En lançant un `telnet buggy 1099` depuis ma machine mini (Windows), je peux m'assurer que le serveur JNDI est en place sur la machine serveur (Linux).

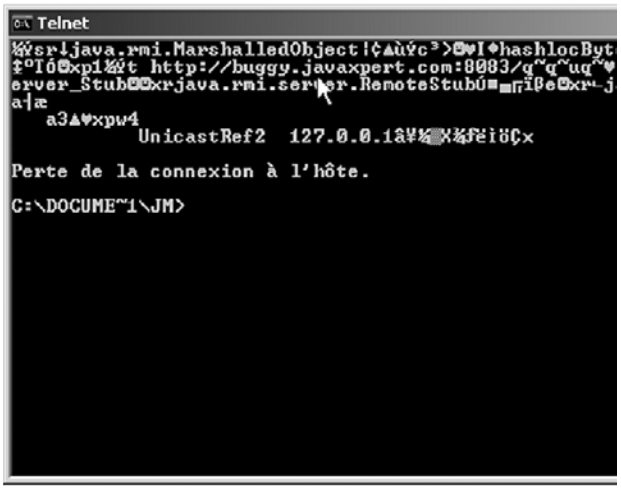


Figure 5–3 Utilisation de Telnet sous Windows ou Unix pour s'assurer du bon fonctionnement de JBoss

On obtient un écran un peu bizarre (figure 5–3), une sortie peu exploitable mais c'est bon signe !

- Sous Unix, on peut utiliser sur la machine serveur (celle hébergeant JBoss) le programme `ps`, permettant de lister les processus. Voici une capture d'une console Unix utilisant ce programme (figure 5–4).
- Enfin, la meilleure solution reste de tester grandeur nature avec une application.

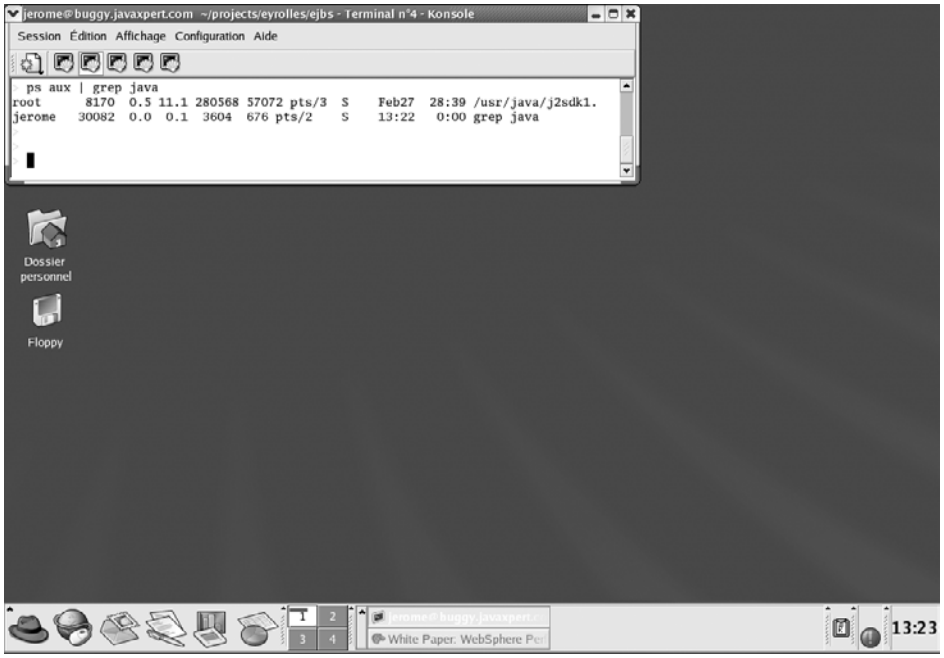


Figure 5–4 Le programme ps dans une console Unix

Il ne reste plus qu'à créer une archive (`.ear` ou `.jar`) de manière à déployer notre composant. Pour cela, rien de tel qu'utiliser les atouts offerts par Ant.

CROSS REFERENCE **Build-file prêt à l'emploi**

Vous trouverez dans la suite de ce chapitre un fichier Ant prêt à l'emploi accomplissant de nombreuses tâches, dont notamment l'empaquetage d'une application. On ne peut que vous inviter à reprendre les sections de ce fichier dédiées à cet usage.

Programme client de test

Une fois tout cela empaqueté (se reporter au *build-file* commenté plus loin dans ce chapitre) et déployé dans JBoss, il nous faut un programme de test invoquant les méthodes de conversion proposées par notre EJB.

Programme client de notre EJB

Cette classe montrant comment utiliser un EJB depuis un contexte client.

```
package main.client;
import javax.naming.InitialContext;
import java.io.FileInputStream;
import java.util.Properties;
import test.ejb.*;

class SessionClient {
        /**
         * méthode main() démontrant les étapes nécessaires à
         *  l'invocation d'un service sur un EJB:
         * - Initialise un contexte JNDI
         * - Opère la recherche de notre EJB session
         * - Obtient la home interface
         * - crée un bean de session
         * - puis invoque le service de conversion
         */
  public static void main(String[] args){
  try {
       // crée un objet Properties
        Properties props = new Properties();
        // charge le fichier jndi.properties
        props.load( new FileInputStream("jndi.properties"));
        // utilise cet objet pour initialiser le contexte JNDI
        // de recherche
        InitialContext jndiContext = new InitialContext(props);
        System.err.println("Initial context initialized");
        // utilise ce contexte pour chercher notre bean session
        // attention à utiliser le bon nom
        ExempleSessionHome home = (ExempleSessionHome)
           jndiContext.lookup("ejb/exemple/session");
        // utilise la home interface pour obtenir un proxy sur notre EJB
        ExempleSessionRemote session = home.create();
        float montant=(float)1000f;
        // procède à la conversion
        float francs=session.euroToFranc(montant);
        System.out.println("Conversion = " + francs);
     }
     catch(Exception x) {
        x.printStackTrace();
     }
  }
}
```

On ne peut que préciser le caractère très typique de ce programme de test. En fait, tous vos programmes clients auront la même structure, à savoir : initialisation d'un contexte JNDI pour permettre les recherches d'objets dans votre contexte particulier (serveur EJB utilisé, port et nom de machine), puis obtention d'une Home interface de manière à pouvoir créer ou rechercher vos objets distants, puis appel aux services de vos EJB.

En compilant (avec un `CLASSPATH` correct) et en lançant ce programme (par `java main.client.SessionClient`), vous devez être à même d'observer une sortie du type :

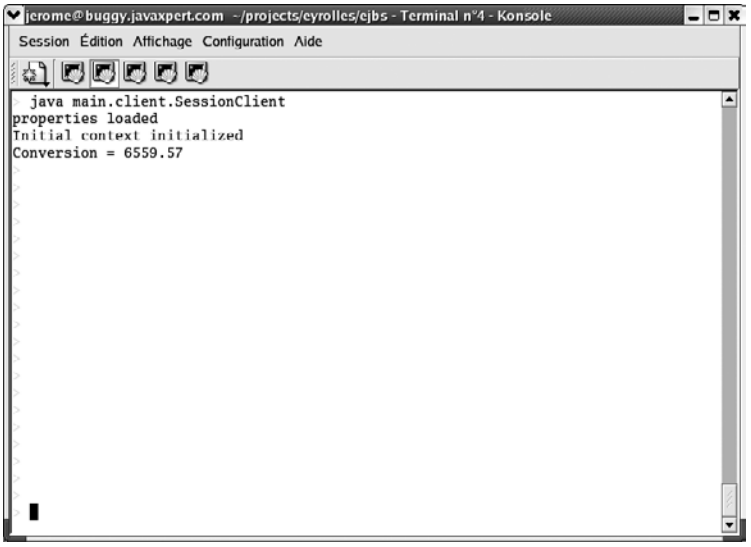


Figure 5–5 Lancement du client

L'exécution de ce programme présuppose que vous fournissiez un fichier texte (`jndi.properties`) initialisant le contexte des recherches JNDI. Ce fichier doit être dans le répertoire à partir duquel vous lancez le programme.

Voici le contenu de ce fichier :

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

Tel quel, ce fichier suppose que le programme client sera lancé depuis la même machine que celle faisant tourner le serveur JBoss. Si vous possédez deux machines, n'hésitez pas à modifier ce fichier en changeant la ligne `java.naming.provider.url`, de manière à remplacer la valeur `localhost` par l'adresse IP ou le nom DNS de la machine serveur.

Conclusion partielle – travail nécessaire au codage d'un EJB à la main…

Le travail nécessaire au codage à la main d'un EJB est donc très important. Imaginons un moyen de nous concentrer sur la partie intéressante et d'oublier toute la « tuyauterie » nécessaire.

Étant donné la redondance d'informations entre, d'une part, les interfaces et classes abstraites et, d'autre part, les classes d'implémentation créées à partir des

consignes de déploiement par le conteneur, on pourrait imaginer faire générer pour nous par un outil tiers ce code peu intéressant et propice aux erreurs. Une telle approche permet résolument de raccourcir le cycle de développement, d'améliorer la qualité du code (une fois un outil validé, on peut être sûr de ce qu'il produit) et de nous concentrer sur la logique applicative, qui est le plus intéressant pour une entreprise.

Comment ? Quelle technique utiliser ? La réponse est simple : les doclets Java.

Introduction aux doclets

Les doclets introduites avec le JDK 1.2 permettent de modifier l'apparence de la documentation créée via javadoc, ou du moins s'agit-il là de leur utilisation première. Cependant, avec un peu d'imagination, on voit que l'on dispose d'un outil assurant le parsing d'un source Java (et le faisant vite et bien), ainsi que d'une API robuste. C'est plus que suffisant pour détourner l'outil de sa destination originelle et imaginer faire de notre fichier Java une source de données moins réduite que la simple somme des codes des différents services. En ajoutant quelques balises (`@monsignet blah blah`) qui n'ont de sens que pour nous, on peut imaginer beaucoup d'autres utilisations à cette technique, par exemple :

- production de documentation ;
- écriture de code ;
- modification du code initial (instrumentation).

Citons quelques produits utilisant cette mécanique :

- iDoclet : outil d'instrumentation de code permettant d'ajouter à Java la gestion des assertions au sens de la programmation par contrats ;
- XDoclet : nous reviendrons bientôt sur cet outil… ;
- DocBookDoclet : pour imprimer au format DocBook vos API ;
- Bouvard& Pécuchet : un ensemble de deux outils visant à améliorer la qualité du code.

Manipuler les doclets

Avant d'utiliser un produit, comprendre les tenants et aboutissants semble naturel. Pour cela, rien de tel qu'une petite mise en pratique du sujet. BlueWeb décide donc de réaliser un petit exemple très simple mettant en œuvre un doclet « maison » avec une balise personnalisée (`blueweb`, de manière très originale). Ce programme peu ambitieux n'est destiné qu'à mieux percevoir comment fonctionnent des outils complexes ; il ne s'agit nullement d'espérer un résultat utilisable par la suite.

B.A.-BA Doclets

Avec le JDK 1.2, Sun a introduit la possibilité de personnaliser très simplement la documentation créée via javadoc : juste en codant une classe Java particulière appelée doclet. L'idée centrale est de ne pas à avoir à redévelopper du code dédié au parsing du code Java et donc de laisser javadoc le faire pour vous ! Ensuite, vous n'avez qu'à utiliser les méthodes de très haut niveau mises à disposition par l'API javadoc (paquetage `com.sun.javadoc`) pour mettre en forme selon vos désirs. L'outil javadoc crée en standard une documentation HTML rattachée à chaque source Java via exploration de celui-ci et en réagissant face à certaines balises particulières (`@author`, `@param`, etc.), mais rien ne nous empêche de générer du code Java… Une très bonne introduction sur ce domaine est disponible à l'URL suivante :

- http://www.javaworld.com/javaworld/jw-08-2000/jw-0818-javadoc_p.html

ALLER PLUS LOIN Pharos, le moteur de recherche dédié à Java

- http://pharos.inria.fr/Java/query.jsp?cids=c_2210

est l'URL permettant de lancer une recherche adaptée à ce domaine sur la base de données du site Pharos, superbe outil issu du travail de l'INRIA et de volontaires (Olivier Dedieu, Nicolas Delsaux, entre autres).

POLÉMIQUE Sur l'art de faire du jetable

Ce sujet revient souvent sur la table au cours de projets. Trop souvent, des projets débutent sur des bases issues de produits jetables… Il semble bon de limiter l'utilisation de code jetable aux simples sujets nécessitant un retour pour validation esthétique, approfondissement d'un point technique ou apprentissage sur un thème précis. Sinon, les risques de voir en production un code initialement destiné à la poubelle sont plus que probables.

Voici donc le code de ce doclet de test :

Classe d'illustration du concept de doclet avec la création d'une balise Javadoc « maison »

Attention, ces classes étant des classes internes Sun, la portabilité de ce code n'est pas assurée sur toutes les machines virtuelles possibles. Pour compiler/exécuter ce code, il est nécessaire d'avoir le `tools.jar` dans le classpath du projet.

```
package com.blueweb.test.doclet;
import com.sun.javadoc.ClassDoc;
import com.sun.javadoc.MethodDoc;
import com.sun.javadoc.RootDoc;
import com.sun.javadoc.Tag;

/**
* @author BlueWeb
* <p>
* Petite classe de test mettant en oeuvre les doclets Java.
* Ici, nous allons manipuler un tag bien à nous (nous nous
* contenterons d'afficher les commentaires insérés dans ce tag
* mais on peut imaginer d'autres utilisations...
* </p>
*/
public class CustomTag {
   /**
    * <p>
    * la méthode start() est appelée automatiquement par l'outil
    * javadoc. Pour exécuter notre classe, il faudra utiliser
    * une syntaxe de ce type :
    * javadoc -doclet com.blueweb.test.doclet.CustomTag
   * </p>
    */
```

Une fois la balise de début repérée, on demande l'invocation de la méthode `dump()`.

```
    public static boolean start(RootDoc root){
        dump(root.classes(), "blueweb");
        return true;
    }
   /**
    * <p>
    * affiche les tags associés à des commentaires de méthode
    * pour une liste de classes passées en argument, le tag recherché
    * étant blueweb (passé en paramètre tagName)
    * </p>
    * @param classes, liste des classes
    * @param tagName, nom du tag cible (blueweb)
   */
    private static void dump(ClassDoc[] classes, String tagName) {
```

Cette méthode permet d'illustrer les différentes possibilités offertes par l'API des doclets. On voit comment l'on peut obtenir des informations concernant une méthode ou un attribut, les tags (balises) associés aux commentaires d'une méthode…

```
   // itère sur la collection...
        for (int i=0; i < classes.length; i++) {
        // un drapeau utilisé pour éviter de dupliquer
        // les affichages du nom de la classe
            boolean classNamePrinted = false;
            // extrait les infos sur les commentaires de méthodes
            // pour la classe en cours
            MethodDoc[] methods = classes[i].methods();
            // itère sur ce tableau
            for (int j=0; j < methods.length; j++) {
```

```
            // pour une méthode donnée, n'extrait que les
            // tags dont le nom est celui de notre cible
               Tag[] tags = methods[j].tags(tagName);
               if (tags.length > 0) {
                   if (!classNamePrinted) {
                      System.out.println("\n" + classes[i].name() + "\n");
                       classNamePrinted = true;
                   }
                   System.out.println(methods[j].name());
                   for (int k=0; k < tags.length; k++) {
                       System.out.println("   " + tags[k].name() + ": "
                           + tags[k].text());
                   }
               }
           }
       }
   }
}
```

Étant donné que nous utilisons un tag particulier, il est bon de tester avec un fichier source possédant des occurrences de ce tag… C'est précisément l'objet de la classe de test qui suit.

Classe exemple utilisant notre balise Javadoc (@blueweb)

Une classe bien inutile mais permettant de tester notre doclet…

```
package com.blueweb.test.doclet;
public class TesClass {

   /**
   * @blueweb oh un tag
   * @blueweb deuxieme tag dans foo
    * @autretag pas interessant celui la
   */
   public void foo(){
   }

   /**
    * @autretag ne doit pas apparaître
   */
   private void bar(){
   }

   /**
    * @blueweb oho un autre tag sympa
   */
   protected int foobar(){
      return -1;
   }
}
```

Et voici le résultat de l'exécution de ce code. On utilise comme paquetage cible `com.blueweb.test.doclet`, étant donné qu'il y a peu de chance de trouver ailleurs des traces de cette balise inutile, `javadoc -help` pour les options de javadoc.

```
> javadoc -doclet com.blueweb.test.doclet.CustomTag
com.blueweb.test.doclet
Loading source files for package com.blueweb.test.doclet...
Constructing Javadoc information...
TesClass
foo
   @blueweb: oh un tag
   @blueweb: deuxieme tag dans foo
foobar
   @blueweb: oho un autre tag sympa
```

Pourquoi le tag documentant la méthode `bar()` de la classe de test n'apparaît-il pas lors de l'exécution de ce doclet ? La réponse tient simplement dans la conception de javadoc, qui par défaut ne regarde pas les méthodes privées (c'est tout à fait normal puisqu'un programmeur client n'a pas à s'embarrasser avec les détails de notre implémentation).

Ce petit bout de code très simple a permis à l'équipe BlueWeb de comprendre les rudiments de la technologie des doclets et d'acquérir le recul nécessaire pour apprécier un tel outil…

Outil rattaché : XDoclet

Figure 5–6 Logo du projet XDoclet

Comme nous venons de le voir, le développement d'un composant EJB peut réclamer jusqu'à sept classes et un fichier XML. De plus, la norme EJB s'appuyant sur de grosses conventions de nommage, ce travail s'avère vite fastidieux et donc propice à de nombreuses petites erreurs dues à un manque d'attention ou à une utilisation excessive du copier/coller. Il faut donc chercher une solution logicielle à même de nous mâcher le travail. C'est là qu'intervient XDoclet, un outil Open Source qui utilise le concept des doclets Java pour fournir à partir du seul code source réellement utile de la liste précédente : le bean d'implémentation (`CompteBean` dans notre exemple) et les composants rattachés, à savoir `Home interface`, `Remote Interface`, `LocalHome` et `Local Interface`, fichier de description du déploiement…

XDoclet est disponible à l'URL suivante : http://xdoclet.sourceforge.net. La dernière version disponible est la 1.2.2, qui semble corriger de nombreux bogues de la précédente version de développement.

ALTERNATIVE EJBGen

Pour les utilisateurs de Weblogic, il faut signaler l'excellent outil de Cedric Beust, maintenant intégré dans Weblogic Server (depuis la version 7) : EJBGen. Il utilise aussi l'idée des doclets Java. Malheureusement, pour des raisons historiques, il n'est pas placé sous licence Open Source.

- http://beust.com/ejbgen

Introduction à XDoclet

XDoclet se présente sous la forme d'une tâche Ant et, comme tout doclet, parse un ensemble de fichiers source `.java`.

L'idée centrale régissant l'utilisation de cet outil est que, lors du développement d'un EJB, la seule classe d'implémentation (`MonEJBBean.java`) enrichie de quelques informations doit permettre de guider le reste du déploiement de ce composant. Il vous suffit donc d'enrichir le fichier `MonEJBBean.java` des différents

tags introduits par XDoclet, puis de lancer XDoclet via Ant pour obtenir tous les fichiers nécessaires au déploiement de votre composant au sein de votre serveur d'applications, c'est-à-dire :

- la `home interface` associée (en local comme en *remote* suivant vos directives) ;
- la `remote interface` associée (en local comme en *remote* suivant vos directives) ;
- la description de déploiement (`ejb-jar.xml`) ;
- les `Value Object` (ou `Data Object` suivant la version de XDoclet utilisée) ;
- les fichiers complémentaires propres à votre serveur d'applications.

Eh oui ! tout cela est obtenu à partir d'un seul et même fichier et en plus avec l'indépendance par rapport au serveur d'applications puisque XDoclet gère spécifiquement les serveurs :

- JBoss ;
- Weblogic (Bea) ;
- Jonas (objectweb) ;
- Websphere (IBM) ;
- Jrun (Macromedia) ;
- Resin (caucho) ;
- Pramati ;
- Easerver (Sybase) ;
- Hpas (HP) ;
- Orion (Orion).

De plus, cet outil propose aussi de guider la gestion du mapping objet/relationnel si vous le confiez à une des technologies suivantes :

- JDO (Java Data Objects, spécification de Sun implémentée par différents produits) ;
- Hibernate ;
- MVCSoft.

Pour en finir avec la liste des technologies prises en compte par XDoclet, il faut préciser que ce produit s'interface très bien avec Struts.

De plus, de par son architecture et son *design*, XDoclet est un produit ouvert et évolutif, vous permettant via les *merge points* d'adapter la génération à votre contexte et, en plus, d'être étendu (vous pouvez baser votre doclet sur XDoclet). Et tout cela est proposé sous une licence légale proche de la licence Apache…

Même si l'utilisation telle qu'illustrée par le projet BlueWeb est typiquement tournée vers les EJB, ce produit ne se cantonne pas à ce domaine. Ainsi, on peut citer l'utilisation de XDoclet pour les classes manipulant JMX (création de MBeans), le déploiement de servlets, etc.

La solution avant de lancer une équipe dans un tel produit est peut-être de qualifier l'adéquation du produit avec vos besoins, de faire un test et dans le pire des cas de figer son choix sur une version, quitte à ne pas bénéficier des nouvelles fonctionnalités introduites lors de versions ultérieures.

POLÉMIQUE **XDoclet, la « silver bullet »**

Avec une présentation si élogieuse, vous pouvez voir en XDoclet la balle magique, solution à tous vos problèmes. Il s'agit réellement d'un très bon outil, mais il est bon de préciser que la documentation n'est pas toujours à la hauteur du produit, que la principale source d'aide (la liste de diffusion) est une des plus désagréables qui soit (gare aux *newbies* J) et que les codeurs de XDoclet n'ont aucun scrupule à changer tous les tags lors du passage de la version 1.1 à la version 1.2 et rendent ainsi caduques vos codes source utilisant les anciens tags, ce qui est rédhibitoire dans la plupart des cas ! Attention, la liste est quand même de très bonne qualité puisqu'on y croise fréquemment Erik Hatcher, auteur du meilleur ouvrage dédié à Ant, ainsi que David Jencks, auteur de la couche de gestion des connecteurs de JBoss.

Ajoutez des doclets dans vos build-files

Et maintenant, place à l'action ! Examinons le *build-file* adapté d'exemples trouvés sur Internet pour adapter XDoclet (principalement celui réalisé par David Jencks de l'équipe JBoss pour la compilation/déploiement des exemples accompagnant JBoss). Ce fichier est livré tel quel, mais les annotations tiendront lieu de commentaires…

Script Ant modèle utilisé dans les exemples JBoss

```
<?xml version="1.0"?>
<!-- Template build file -->
<project name="template" default="main" basedir=".">
<!--
      Give user a chance to override without editing this file
      (and without typing -D each time they run it)
   -->
<property file=".ant.properties" />
   <property file="${user.home}/.ant.properties" />
   <property name="Name" value="Template"/>
   <property name="version" value="1.1"/>


   <target name="check-jboss" unless="jboss.home">
      <fail>
         Property "jboss.home" is not set. Please use the file
         ".ant.properties" in this directory ${basedir} to
         set this property. It must point to the directory which
         contains the following directory: "deploy", "conf", "tmp"
         etc.
      </fail>
   </target>

   <target name="check-jboss" unless="jboss.home">
      <fail>
         Property "jboss.home" is not set. Please use the file
         ".ant.properties" in this directory ${basedir} to
         set this property. It must point to the directory which
         contains the following directory: "deploy", "conf", "tmp"
         etc.
      </fail>
   </target>

   <target name="wrong-jboss" unless="jboss.present">
      <fail>
         Property "jboss.home" is set but it does not seem
         to point to the right directory. The file "run.jar"
         must be available at ${jboss.home}/bin.
      </fail>
   </target>

<target name="check-xdoclet" unless="xdoclet.home">
      <fail>
         Property "xdoclet.home" is not set. Please use the file
         ".ant.properties" in this directory ${basedir} to
         set this property. It must point to the root directory of
```

Utilise un fichier properties (`.ant.properties`) situé dans le répertoire courant ou dans le répertoire personnel de l'utilisateur. Ce fichier évite de modifier le build-file à tout instant.

Cette cible permet de s'assurer que JBoss est bien présent en contrôlant la présence de la propriété `jboss.home`. Si cette dernière est absente, affiche le message défini dans `fail`. Remarquez l'utilisation de `unless` et `fail`…

Cette cible complète la précédente en contrôlant une éventuelle mauvaise valeur pour la propriété `jboss.home` et ceci grâce à une variable `jboss.present`. Si `jboss.present` est à `false`, le message d'erreur s'affiche.

On contrôle la présence de XDoclet.

Script Ant modèle utilisé dans les exemples JBoss (suite)

```
        XDoclet distribution.
      </fail>
   </target>
<target name="wrong-xdoclet" unless="xdoclet.present">
      <fail>
         Property "xdoclet.home" is set but it does not seem
         to point to the right directory. The file "xdoclet.jar"
         must be available at ${xdoclet.home}/lib.
      </fail>
   </target
   <target name="check-environment">
      <antcall target="check-jboss"/>
      <available property="jboss.present"
                 file="${jboss.home}/bin/run.jar"/>
      <antcall target="wrong-jboss"/>
      <antcall target="check-xdoclet"/>
      <available property="xdoclet.present"
                 file="${xdoclet.home}/lib/xdoclet.jar"/>
      <antcall target="wrong-xdoclet"/>
   </target>
   <target name="init" depends="check-environment">
      <echo message="build.compiler = ${build.compiler}"/>
      <echo message="user.home = ${user.home}"/>
      <echo message="java.home = ${java.home}"/>
      <echo message="ant.home = ${ant.home}"/>
      <echo message="jboss.home = ${jboss.home}"/>
      <echo message="xdoclet.home = ${xdoclet.home}"/>
      <echo message="java.class.path = ${java.class.path}"/>
      <echo message=""/>
      <available property="jdk1.3+" classname="java.lang.StrictMath" />
   </target>
   <property name="jboss.lib" value="${jboss.home}/lib" />
   <property name="jboss.client" value="${jboss.home}/client" />
   <!-- Configuration used on JBoss 3 to run your server. There must
        be a directory with the same name under "${jboss.home}/server"
     -->
   <property name="jboss.configuration" value="default" />
   <property name="jboss.deploy"
       value="${jboss.home}/server/${jboss.configuration}/deploy" />
   <property name="src.dir" value="${basedir}/src"/>
   <property name="src.main.dir" value="${src.dir}/main"/>
   <property name="src.client.dir" value="${src.main.dir}/client"/>
   <property name="src.ejb.dir" value="${src.main.dir}/ejb"/>
   <property name="src.servlet.dir" value="${src.main.dir}/servlet"/>
   <property name="src.resources.dir" value="${src.dir}/resources"/>
   <property name="src.web.dir" value="${src.dir}/web"/>
   <property name="src.etc.dir" value="${src.dir}/etc"/>
   <property name="lib.dir" value="${basedir}/lib"/>
   <property name="build.dir" value="${basedir}/build"/>
   <property name="build.tmp.dir" value="${build.dir}/tmp"/>
   <property name="build.deploy.dir" value="${build.dir}/deploy"/>
```

XDoclet est-il correctement défini ?

Cette cible regroupe tous les tests nécessaires au contrôle de l'environnement de travail. On peut remarquer que cette cible utilise des `antcall` pour exécuter des cibles (`check-jboss`, `check-sxdoclet`).
Cette cible vérifie les propriétés `xdoclet.present` et `jboss.present` en s'assurant de la disponibilité de certains fichiers grâce à la tâche `available`.

Réalise les initialisations. Utilise aussi la `task available` pour positionner un drapeau permettant de savoir si la version de JDK est supérieure à 1.3 ou non via la présence (absence) d'une classe (`java.lang.StrictMath`). Initialise des variables à partir des propriétés générales définies dans le fichier `.ant.properties` de manière à simplifier l'écriture du build-file (la rendre plus légère). Dépendance envers la cible `check-environment`, ce qui veut dire que si la configuration n'est pas bonne, cette cible ne sera jamais atteinte…

Script Ant modèle utilisé dans les exemples JBoss (suite)

```
   <property name="build.generate.dir" value="${build.dir}/generate"/>
   <property name="build.classes.dir" value="${build.dir}/classes"/>
   <property name="build.war.dir" value="${build.dir}/war"/>
   <property name="build.client.dir" value="${build.dir}/client"/>
   <property name="build.bin.dir" value="${build.dir}/bin"/>
   <property name="build.javadocs.dir" value="${build.dir}/docs/api"/>
```

Construit le `classpath` nécessaire au lancement de XDoclet via la directive `pathelement`.

```
   <path id="xdoclet.path">
      <pathelement location="${ant.home}/lib/ant.jar" />
<!-- AS Maybe necessary for Ant 1.5 and XDoclet 1.3
      <pathelement location="${ant.home}/lib/xmlParserAPIs.jar" />
      <pathelement location="${ant.home}/lib/xercesImpl.jar" />
      <pathelement location="${ant.home}/lib/bcel.jar" />
      <pathelement location="${xdoclet.home}/lib/xjavadoc.jar" />
-->
      <pathelement location="${xdoclet.home}/lib/xdoclet.jar" />
      <pathelement location="${jboss.client}/log4j.jar" />
   </path>
```

Construit le `classpath` dit de base…

```
   <path id="base.path">
      <path refid="xdoclet.path"/>
      <pathelement location="${jboss.client}/jboss-j2ee.jar" />
      <pathelement location="${jboss.client}/jnp-client.jar" />
      <pathelement location="${jboss.client}/jbossmq-client.jar" />
      <pathelement location="${jboss.client}/jbosssx-client.jar" />
      <pathelement location="${jboss.client}/concurrent.jar" />
      <pathelement location="${jboss.client}/jaas.jar" />
      <pathelement location="${jboss.lib}/jboss-jmx.jar" />
      <pathelement location="${jboss.home}/server/
                  ➥ ${jboss.configuration}/lib/jbosssx.jar" />
      <pathelement location="${jboss.home}/server/
                  ➥ ${jboss.configuration}/lib/mail.jar" />
      <pathelement location="${build.classes.dir}" />
   </path>

<!-- ========================== -->
<!-- Generates the necessary EJB classes and deployment descriptors -->
<!-- ========================== -->
```

Lance la création des EJB via XDoclet. Utilise pour cela une des deux `tasks` de XDoclet : `ejbdoclet`. On voit ici une illustration de l'utilisation de `taskdef`, où comment ajouter des `tasks` optionnelles ou inconnues lors de l'exécution d'un `build Ant`. Dépendance envers la cible `init`.

C'est ici que se fait l'appel à `ejbdoclet`, avec tous les paramètres nécessaires (version des spécifications EJB, répertoire de destination). Les fichiers cibles seront tous les fichiers présents dans le répertoire pointé par la variable `src.ejb.dir` et finissant par `Bean.java`, donc toutes les classes abstraites d'implémentation.

```
   <target name="xdoclet-generate" depends="init">
      <taskdef
         name="ejbdoclet"
         classname="xdoclet.ejb.EjbDocletTask"
      >
         <classpath refid="xdoclet.path"/>
      </taskdef>

      <ejbdoclet
         sourcepath="${src.ejb.dir}"
         destdir="${build.generate.dir}"
         classpathref="base.path"
         excludedtags="@version,@author"
         ejbspec="${ejb.version}"
         mergedir="${src.resources.dir}/xdoclet"
         force="${xdoclet.force}"
      >
```

Script Ant modèle utilisé dans les exemples JBoss (suite)

```
        <fileset dir="${src.ejb.dir}">
           <include name="**/*Bean.java"/>
        </fileset>
        <packageSubstitution packages="session,entity"
            substituteWith="interfaces"/>
        <dataobject/>
        <remoteinterface/>
        <homeinterface/>
        <entitypk/>
        <entitybmp/>
        <entitycmp/>
        <deploymentdescriptor destdir="${build.dir}/META-INF"/>
        <!-- AS 4/29/02 Do not validate XML files because JBoss 3.0
            message driven will report an wrong error because
            it uses the wrong jboss.dtd -->
       <jboss version="${jboss.version}"
           xmlencoding="UTF-8"
           typemapping="${type.mapping}"
           datasource="${datasource.name}"
           destdir="${build.dir}/META-INF"
           validateXml="false"
        />
      </ejbdoclet>
   </target>

<!-- ===================================-->
<!-- Compiles the source code           -->
<!-- ===================================-->

   <target name="compile" depends="xdoclet-generate">
      <mkdir dir="${build.classes.dir}"/>
      <javac
         destdir="${build.classes.dir}"
         debug="on"
         deprecation="off"
         optimize="on"
         classpathref="base.path"
      >
         <src path="${src.ejb.dir}"/>
         <src path="${build.generate.dir}"/>
      </javac>
      <javac
         srcdir="${src.client.dir}"
         destdir="${build.classes.dir}"
         debug="on"
         deprecation="off"
         optimize="on"
         includes="**/*.java"
         classpathref="base.path"
      >
      </javac>
   </target>
```

Ce sont les options de génération, telles que demander la génération des `remote interfaces`, des `home interfaces`, des entités, du `ejb-jar.xml`, etc.

Ici sont regroupées les informations propres à JBoss, telles que le nom de la source de données utilisée, la version, etc.

Compile le code source (celui fourni + celui produit par XDoclet) en précisant la volonté d'obtenir un code optimisé.

Script Ant modèle utilisé dans les exemples JBoss (suite)

```
<!-- ================================-->
<!-- Creates the jar archives        -->
<!-- ================================-->
<target name="jar" depends="compile">
   <mkdir dir="${build.deploy.dir}"/>
   <mkdir dir="${build.client.dir}"/>
   <mkdir dir="${build.bin.dir}"/>
   <jar
      jarfile="${build.deploy.dir}/ejb-test.jar"
   >
      <fileset
         dir="${build.classes.dir}"
         includes="test/entity/**,test/session/**,
               ➥   test/message/**,test/interfaces/**"
      >
      </fileset>
      <fileset
         dir="${build.dir}"
         includes="META-INF/**"
      >
      </fileset>
   </jar>
   <jar
      jarfile="${build.client.dir}/client-test.jar"
   >
      <fileset
         dir="${build.classes.dir}"
         includes="test/interfaces/**,test/client/**"
      >
      </fileset>
   </jar>
</target>
<!-- ========================================-->
<!-- Compiles the WEB source code           -->
<!-- ========================================-->
<target name="compile-web" depends="compile"
        if="servlet-lib.path">
   <mkdir dir="${build.war.dir}"/>
   <path id="web.path">
      <path refid="base.path"/>
      <pathelement location="${servlet-lib.path}"/>
   </path>
   <javac
      destdir="${build.war.dir}"
      debug="on"
      deprecation="off"
      optimize="on"
      classpathref="web.path"
   >
      <src path="${src.servlet.dir}"/>
   </javac>
</target>
```

Commence l'empaquetage en archivant en un `jar` tous les EJB. Évidemment, cette tâche requiert l'exécution au préalable de la compilation des sources (d'où la dépendance envers `compile`).

Compile le code des servlets.

Script Ant modèle utilisé dans les exemples JBoss (suite)

```
<!-- ================================================-->
<!-- Creates the war archives                        -->
<!-- ================================================-->
   <target name="war" depends="compile-web" if="servlet-lib.path">
      <mkdir dir="${build.deploy.dir}"/>
      <copy todir="${build.war.dir}/WEB-INF">
         <fileset dir="${src.etc.dir}/WEB-INF"
                  includes="jboss-web.xml"/>
      </copy>
      <war
         warfile="${build.deploy.dir}/web-client.war"
         webxml="${src.etc.dir}/WEB-INF/web-client.xml"
      >
         <fileset dir="${src.web.dir}"/>
         <fileset dir="${build.war.dir}"/>
         <classes dir="${build.classes.dir}"
                  includes="test/interfaces/**"/>
      </war>
   </target>

   <target name="deploy-server" depends="jar,war">
      <copy todir="${jboss.deploy}">
         <fileset dir="${build.deploy.dir}"
                  includes="*.jar,*.war,*.ear">
         </fileset>
      </copy>
   </target>

   <!-- ================================================== -->
   <!-- Creates the client binary                          -->
   <!-- ================================================== -->

   <target name="create-client" depends="jar">
      <!-- Convert the given paths to Windows -->
      <pathconvert targetos="windows" property="jboss.home.on.windows" >
         <path>
            <pathelement location="${jboss.home}" />
         </path>
      </pathconvert>
      <pathconvert targetos="windows" property="java.home.on.windows" >
         <path>
            <pathelement location="${java.home}" />
         </path>
      </pathconvert>
      <!-- Convert the given paths to Unix -->
      <pathconvert targetos="unix" property="jboss.home.on.unix" >
         <path>
            <pathelement location="${jboss.home}" />
         </path>
      </pathconvert>
      <pathconvert targetos="unix" property="java.home.on.unix" >
         <path>
            <pathelement location="${java.home}" />
         </path>
      </pathconvert>
```

Code Web (servlets) en un fichier .war.

C'est ici que l'on crée vraiment la distribution cliente.

Script Ant modèle utilisé dans les exemples JBoss (suite)

```
      <echo message="JBoss Home on Unix: ${jboss.home.on.unix}"/>
      <echo message="Java Home on Unix: ${java.home.on.unix}"/>
      <filter token="jboss.home" value="${jboss.home.on.windows}"/>
      <filter token="java.home" value="${java.home.on.windows}"/>
      <copy todir="${build.bin.dir}" filtering="yes">
        <fileset dir="${src.etc.dir}/bin" includes="run-*client.bat">
        </fileset>
      </copy>
      <copy file="${src.etc.dir}/bin/lcp.bat"
            todir="${build.bin.dir}"/>
      <filter token="jboss.home" value="${jboss.home.on.unix}"/>
      <filter token="java.home" value="${java.home.on.unix}"/>
      <copy todir="${build.bin.dir}" filtering="yes">
         <fileset dir="${src.etc.dir}/bin" includes="run-*client.sh">
         </fileset>
      </copy>
      <copy file="${src.etc.dir}/jndi.properties"
            todir="${build.bin.dir}"/>
   </target>
   <target name="main" depends="deploy-server,create-client">
   </target>


   <target name="clean" depends="init">
      <delete dir="${build.dir}"/>
   </target>
</project>
```

La cible `main`, appelée par défaut, elle se contente par ses dépendances d'invoquer les cibles `deploy-server` et `create-client`.

Ici se déroule le grand ménage : `clean` fait en sorte de remettre les choses à leur état initial en supprimant tout ce qui a été construit.

Voilà le code intégral d'un `make file` réaliste intégrant XDoclet mais aussi assurant la création des archives après compilation et le déploiement sur le serveur d'applications. Ce déploiement est réduit dans ce cas à peu de choses puisque JBoss dispose d'un composant dédié à cette tâche se basant sur la date de dernière modification : en recopiant votre archive (`.ear`, `.jar` ou `.war`) dans le répertoire de déploiement, vous allez mettre à disposition du code plus récent qui sera immédiatement chargé et déployé. Attention donc, sous d'autres serveurs d'applications cette tâche peut être plus ardue.

En passant…

Ce long fichier n'a pas pour seule vocation de remplir des pages : il permet d'appréhender la puissance de Ant et des outils compagnons (tels que XDoclet). Il illustre parfaitement l'utilité de cet outil, qui permet de faire assez simplement des foules de choses utiles.

Familiarisons-nous avec XDoclet

Les sources de l'étude de cas sont livrées sur le site d'accompagnement de l'ouvrage :

- www.editions-eyrolles.com

Avant de se plonger dans le code de l'application de gestion des signets, regardons un peu le code d'un exemple simple d'EJB (bean session sans état) tel que codé avec XDoclet. Cet exemple est issu de la documentation XDoclet, il adopte les tags de la version 1.1.2 (l'ancienne version) mais permet de comprendre la mécanique.

Exemple d'EJB codé avec XDoclet (enrichi de tags XDoclet)

```
package examples;
import java.rmi.*;
import javax.ejb.*;
import org.w3c.dom.Document;
```

Exemple d'EJB codé avec XDoclet (enrichi de tags XDoclet) (suite)

```
import org.xbeans.DOMEvent;
import org.xbeans.DOMSource;
import org.xbeans.DOMListener;
import org.xbeans.XbeansException;
/**
*   This is the EJB Receiver Xbean
*   @ejb:bean type="Stateless"
*         name="org.xbeans.communication.ejb.receiver.Receiver"
*         jndi-name="org.xbeans.communication.ejb.receiver.Receiver"
*         display-name="EJB Receiver Xbean" *

*   @ejb:env-entry name="channelBean" type="java.lang.String"
value="your.channel.bean"
*/


public class ReceiverBean implements SessionBean, DOMSource {
  // EJB Methods
  private SessionContext sessionContext;
  public void ejbCreate() throws CreateException { }
  public void ejbRemove() { }
  public void ejbActivate() { }
  public void ejbPassivate() { }
  public void setSessionContext(SessionContext sessionContext) {
    this.sessionContext = sessionContext;
  }
  // DomListener method
  /**
   * The method that the sender uses to pass the Document over
   * @param Document incomingDocument
   * @throws RemoteException
   * @throws XbeansException
   *
   * @ejb:interface-method view-type="remote"
   */
  public void documentReady(Document incomingDocument)
         throws RemoteException, XbeansException {
    if (DOMListener == null) {
       try {
          if (channelBean == null) {
             String cBean = (String) new javax.naming.InitialContext().
                          ➥ lookup("java:comp/env/channelBean");
           if (cBean == null) {
              throw new Exception();
           } else {
              this.setChannelBean( cBean );
           }
         }
      } catch (Exception e) {
         throw new XbeansException(incomingDocument.getNodeName(),
             "ejb receiver", "next component not established",
             "The component needs to be configured.");
     }
```

Ce tag est le plus intéressant de tous ; il précise que l'on a affaire à un bean de type stateless (sans état), donne son nom, son nom JNDI.

Notons cet autre tag XDoclet pour définir des valeurs d'environnement. Il permet de ne pas coder en dur le canal écouté par notre « listener DOM ».

Cette méthode comme d'habitude doit implémenter de manière triviale les méthodes en liaison avec le cycle de vie de l'objet dans le conteneur.

Il s'agit ici du code d'implémentation, les services fournis par le bean. Ici, c'est un composant écoutant des événements DOM (une des méthodes de parsing de documents XML).

Exemple d'EJB codé avec XDoclet (enrichi de tags XDoclet) (suite)

```
      try {
        this.setDOMListener( (DOMListener)
              Class.forName( channelBean ).newInstance() );
      } catch (Exception e) {
         throw new XbeansException(incomingDocument.getNodeName(),
           "ejb receiver", "could not create the channelBean: " +
           channelBean + " " + e,
           "The component needs to be configured correctly
           ➥ (channelBean).");
      }
    }
    DOMListener.documentReady(new DOMEvent(this, incomingDocument));
  }
  // DomSource methods
  private DOMListener DOMListener;
  /**
   * Set the DOMListener (usually worked out from the channelBean
   * property)
   *
   * @param newDomListener
   */
  public void setDOMListener(DOMListener newDomListener) {
    DOMListener = newDomListener;
  }
  /**
   * Retrieve the DOMListener
   *
   * @return DOMListener
   */
  public DOMListener getDOMListener() {
    return DOMListener;
  }
  // Channel Bean: This is configured in the &lt;env-entry&gt;
  //                in the ejb-jar.xml deployment descriptor
  private String channelBean;
  /**
   * Set the channel bean to keep the chain going
   * @param newChannelBean
   */
  public void setChannelBean(String newChannelBean) {
    channelBean = newChannelBean;
  }
  /**
   * Retrieve the channel bean name
   * @return String
   */
  public String getChannelBean() {
    return channelBean;
  }
}
```

Voilà, ce qu'il faut retenir est comment, avec quelques balises, on économise une masse de travail assez importante. Ceci doit intéresser tout développeur, qui doit être enclin à la paresse.

Tirez la quintessence de XDoclet

Cette section n'a pas la prétention d'être un manuel de référence sur les EJB, elle vise juste à introduire une notion fondamentale (les doclets), ainsi qu'un outil de choix : XDoclet. Il faut bien retenir que ce produit vous permet de vous concentrer sur l'essentiel (implémentation de la logique métier (pour des beans session) et définition de vos attributs persistants (pour des entités). Avec un seul fichier source Java, enrichi de marqueurs conformes à la version de XDoclet que vous utilisez, vous laissez la charge au produit de produire l'arsenal de code et fichiers de déploiement rattachés à ce fichier. Cette démarche a le mérite de vous libérer d'erreurs liées à une utilisation massive du copier/coller, qui est la seule alternative raisonnable. Avec un tel outil dans votre besace, vous êtes sûrs de raccourcir les temps de développement et de mieux maîtriser les phases de migration d'un serveur d'application à un autre. Là encore, c'est adopter une attitude pragmatique que de confier ces tâches ingrates à un outil.

Intégrer XDoclet dans votre cycle de développement

Si XDoclet est un outil dont le caractère indispensable vient d'être prouvé, il nous reste encore à trouver une manière élégante de l'intégrer dans notre conception. Pour cela, reposons le contexte d'un environnement classique :

- AGL supportant UML ;
- IDE ou éditeur de texte ;
- fichiers `.java`.

L'utilisation de XDoclet telle que présentée précédemment pose problème, dans le sens où les tags nécessaires à son exécution sont consignés uniquement dans le code des fichiers d'implémentation de nos beans. Cela constitue une faiblesse indéniable et dénote un manque de continuité dans le cycle si l'on adopte une approche type MDA. En substance, ce type d'approche vise à voir dans le modèle UML la seule source d'informations importante, puisque le modèle va être utilisé comme référentiel servant à l'élaboration d'une grande partie du code.

B.A.-BA **AGL**

L'atelier génie logiciel désigne des outils capables de prendre en charge une large partie (si ce n'est l'intégralité) du cycle de développement (analyse, conception, codage, test, production de documentation). On peut citer ArgoUML, Objecteering, Rational XDE et bien d'autres.

B.A.-BA **MDA (Model Driven Architecture)**

L'approche MDA tente de regrouper toute l'intelligence au sein du modèle, de manière à y trouver un référentiel unique de documentation et à pouvoir utiliser des outils de production de code depuis le modèle.

Vers une amorce de solution

Imaginons l'approche idéale. Elle nécessiterait les étapes suivantes :

1. création du modèle UML ;
2. enrichissement du modèle avec des informations guidant le déploiement ;
3. export de ce modèle ;
4. lancement de Ant et production des fichiers.

Voilà une approche séduisante. Cependant, parmi ces quatre étapes, la troisième semble problématique : comment exporter un modèle UML indépendamment de l'outil utilisé ? La réponse est simple car l'OMG gérant UML a anticipé ce type d'utilisations : la solution se nomme XMI. Avec cette arme entre nos mains, on peut imaginer un outil permettant de lire du XMI et de créer les fichiers nécessaires au déploiement en s'appuyant sur XDoclet.

B.A.-BA OMG (Object Management Group)

Organisme en charge de la standardisation et de l'évolution d'UML.

B.A.-BA XMI

Format neutre de données s'appuyant sur du XML pour véhiculer le contenu de modèles UML afin de faciliter les communications entre outils.

Intégration de Xdoclet dans le cycle de développement

Figure 5–7
Intégration de XDoclet dans le cycle de développement

AndroMDA : le candidat idéal

Il faut constater qu'étant donné les choix restrictifs imposés par notre recherche, il n'y a pas beaucoup de candidats possibles. À vrai dire, il n'y en aurait même qu'un seul : AndroMDA (anciennement UML2EJB). Ce projet Open Source possède donc toutes les vertus désirées. Vous le trouverez à l'adresse ci-contre.

▸ http://uml2ejb.sourceforge.net/

Intégration de l'outil

Examinons comment mettre en œuvre pratiquement cet outil par l'intermédiaire d'un extrait de *build-file* Ant.

Extrait de script Ant invoquant AndroMDA (UML2EJB)

```
<unzip src="Modele.zargo" dest="${build.dir}/unzipped" />

<style basedir="${build.dir}/unzipped"
       destdir="${build.dir}/model"
       extension=".xml"
       includes="*.xmi"
       style="${env.UML2EJBHOME}/xmi-to-simpleOO.xsl"
/>
<uml2ejb basedir="${build.dir}/model"
         ejbDestdir="${generated.src.dir}"
         implDestdir="${handcrafted.src.dir}"
         includes="*.xml"
         lastModifiedCheck="true"
         templatePath="${env.UML2EJBHOME}"
         useDefaultTemplateConfig="true"
/>
```

Décompression du fichier modèle UML.

Transformation en un document XMI.

Appel de la task `uml2ejb`.

Dans cet extrait de code (portion de fichier `build.xml`), on suppose récupérer un modèle UML provenant de l'outil ArgoUML (outil Open Source). Le fichier `modele.zargo` est un fichier compressé ; on le décompresse donc avec la tâche `unzip`. Puis, à l'aide de la tâche `style` (voir chapitre 7 pour plus d'informations et exemples d'utilisation de cette tâche), on transforme le fichier XMI par XSLT. Enfin, on utilise la tâche `uml2ejb` pour lancer la création de nos EJB.

B.A.-BA XSLT (Extensible Stylesheet Language Transformations)

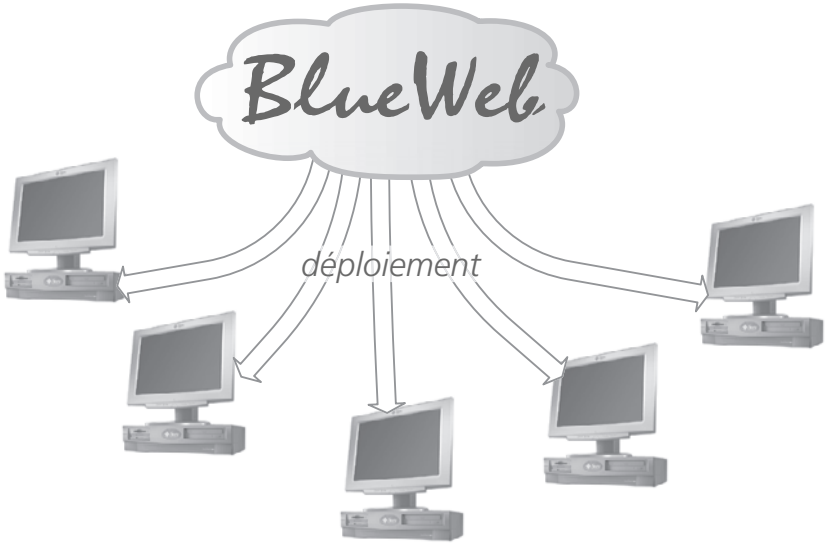
Transformation d'un format de fichier en un autre format, guidée par XML.

Même si cet outil est encore jeune, il nous propose une solution élégante et puissante permettant de gagner en productivité tout en ayant un référentiel d'informations complet (modèle UML). Il est donc intéressant de jeter un œil à cet outil en devenir.

En résumé…

Ce chapitre ne peut être une présentation exhaustive des EJB, mais il en a rappelé les principes fondamentaux avant de s'attarder sur comment développer intelligemment des EJB en faisant faire à un outil le gros du travail rébarbatif. De plus, nous avons saisi l'occasion de présenter un « vrai » *buid-file*, prêt à l'emploi, pouvant vous mâcher le travail lors de futurs développements J2EE. Bref, ce chapitre prétend présenter des solutions pragmatiques à même de faciliter et d'accélérer les développements en environnement J2EE. En proposant par le biais de AndroMDA (UML2EJB) une solution vous permettant d'intégrer les informations de déploiement dans votre modèle UML, on peut penser que vous disposez de toutes les briques permettant d'effectuer un travail efficace.

chapitre 6



Déploiement et gestion des versions avec Ant et Java Web Start

Une fois l'application cliente prête, il faut la diffuser aux utilisateurs. Dans notre cas, la diffusion se limite aux employés de BlueWeb via le réseau interne de la société. Quelle politique adopter et avec quels outils ? Nous verrons comment utiliser Java Web Start pour déployer notre application sur le réseau intranet de BlueWeb.

SOMMAIRE

MOTS-CLÉS

- Déploiement
- Versions logicielles
- Coût d'administration
- Java Web Start
- Factory
- Singleton

Pourquoi développer une application si son déploiement est si complexe/coûteux que personne ne peut l'utiliser ? Comment réduire ces coûts ?

Gérer les versions et maintenir l'application

Pour BlueWeb, les questions de gestion et de maintenance sont cruciales. L'expérience a en effet montré qu'il n'est pas envisageable de faire face au problème des appels récurrents à l'assistance technique pour des bogues connus, répertoriés et corrigés. La politique de déploiement, simple, efficace et réaliste qui réponde à ces exigences est la suivante :

1. Automatiser la mise à jour du produit dès qu'une nouvelle version est publiée (sans intervention manuelle).
2. Supprimer toute intervention sur le poste client qui se limiterait à l'insertion d'un CD-Rom contenant la nouvelle version.
3. N'utiliser que des technologies s'appuyant sur des standards pour éviter des configurations trop exotiques sur le poste client (éviter d'installer des protocoles réseau trop propriétaires en se contentant des classiques TCP/IP, FTP et HTTP par exemple).

L'équipe BlueWeb a remarqué l'émergence d'une technologie chez Sun : Java Web Start, dont le fonctionnement pour le déploiement d'une application est simplifié (voir figure 6-1).

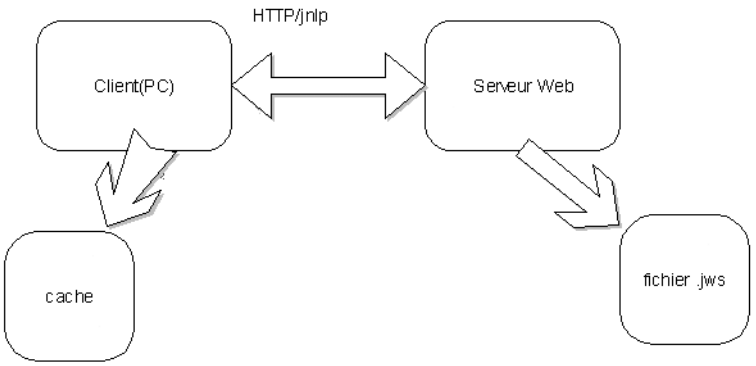


Figure 6–1 Schéma de principe de la technologie Java Web Start

Un utilisateur télécharge depuis son PC l'application qui vient remplir le cache applicatif, il utilise l'application (via les icônes de lancement JWS) jusqu'à ce qu'une nouvelle version soit disponible (il y a contact entre le PC client et le serveur pour savoir si une nouvelle version est disponible ou non), celle-ci est téléchargée puis remplace la précédente version dans le cache, etc.

PRATIQUE Coût d'administration d'une application

Tous les managers et directeurs de service doivent être vigilants sur cet indicateur. Il s'agit pour eux de limiter les interventions de techniciens sur les postes clients pour de simples mises à jour où ils se contentent d'insérer un CD-Rom et de cliquer sur des boutons. L'expérience acquise au cours des années 1990, où le nombre d'applications de type deux couches (Delphi ou VB par exemple) a explosé, leur a permis de comprendre qu'ils ne pouvaient plus se permettre de mobiliser des ressources humaines pour des opérations aussi triviales qu'insérer un CD-Rom et cliquer sur des boutons. De plus, lorsque l'entreprise est répartie sur plusieurs sites, les coûts et délais de mise à jour de tels applicatifs croît exponentiellement.

POLÉMIQUE L'expérience des technologies propriétaires

En plus des problèmes de pérennité de telles solutions, se pose pour les décideurs le problème des coûts de maintenance des postes clients. En effet, la plupart des machines de grandes sociétés sont configurées par installation d'une image, d'un clone d'une installation validée par l'entreprise. L'utilisation de technologies standards permet donc de s'assurer de l'adéquation de ces postes clients avec l'applicatif. Car il faut bien comprendre que redéployer 20 000 postes ne se fait pas en un jour… Pour un poste bureautique typique, on peut imaginer une image (type `ghost` de Symantec) comprenant : un système (disons Windows 2000), une suite bureautique (Works, Office ou OpenOffice.org), un navigateur (Mozilla Firefox ou IE).

Le diagramme de séquence UML suivant schématise les interactions entre les acteurs intervenant lors du lancement d'une application déployée via Java Web Start.

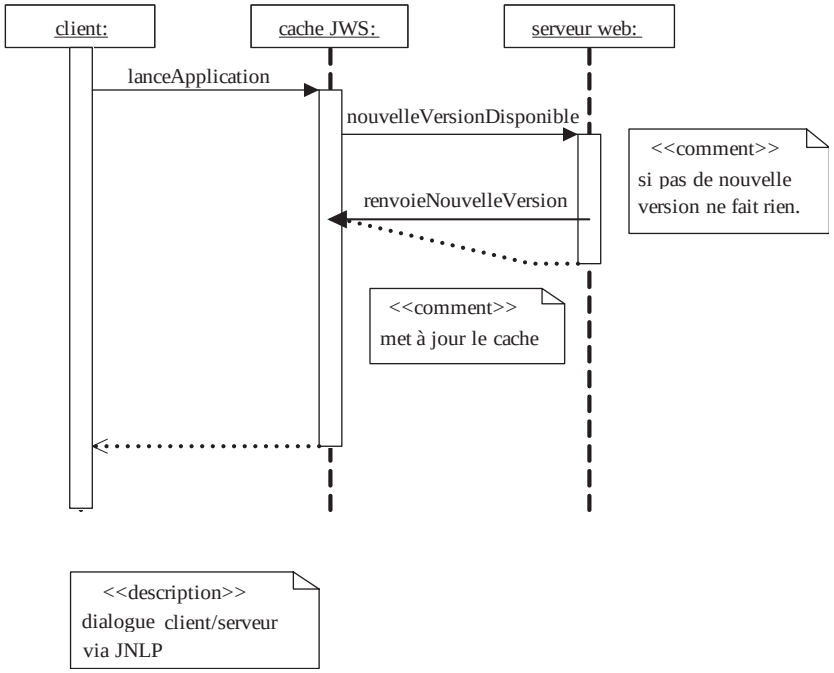


Figure 6–2
Diagramme de séquences UML pour le dialogue client/serveur

Les JRE récents (1.4.x) incluant Java Web Start, l'installation de ce produit dans de telles conditions est réduite à néant. Voici une copie d'écran de la fenêtre principale de Java Web Start (voir figure 6-3), obtenue après avoir cliqué sur l'icône Java Web Start.

Figure 6–3
Lancement d'applications avec Java Web Start

JNLP

JNLP signifie *Java Network Launching Protocol* et désigne le protocole développé par Sun pour Java Web Start.

RÉFÉRENCE Java Web Start

L'URL http://java.sun.com/products/javawebstart/ est la page d'accueil Sun de Java Web Start, alors que la page de Gerald Bauer (http://vamphq.com/) est sûrement sur Internet la meilleure collection de ressources dédiée à ce sujet.

Un écran propose d'installer des raccourcis sur le bureau (figure 6-4).

Figure 6–4 Création des raccourcis

Utilisation de Java Web Start sur le réseau BlueWeb

Que faut il mettre en place au sein de BlueWeb pour pouvoir déployer l'application via Java Web Start ?

En réalité, il faut peu de choses :

1. configurer le serveur web interne de manière à ce qu'il gère l'extension `.jnlp` ;
2. installer un JRE 1.4 sur les postes clients (ou 1.3.x + les binaires Java Web Start) ;
3. créer le fichier `.jnlp` et la page HTML devant être installés sur le serveur web ;
4. et bien sûr empaqueter l'application cliente de manière à la distribuer via le réseau.

Nous ne détaillerons pas ici comment installer un JRE sur une machine cliente, mais on peut noter au passage que cette procédure peut être intégrée dans l'image d'une machine réalisée avec un outil du type Symantec `Ghost`. Ce type de procédé permet de proposer des machines identiques et de réduire les coûts d'installation. Mais il faut gager que les DSI le savent déjà… La machine virtuelle Java (JRE) s'installe par lancement de l'assistant d'installation sous Windows ou d'un paquetage (`.deb` ou `.rpm`) sous Linux.

ATTENTION **Configuration du serveur**

La manipulation suivante, réalisée par l'administrateur système pour configurer le serveur Apache utilisé pour l'intranet de BlueWeb, n'est pas forcément adaptée à votre serveur (en particulier s'il s'agit d'un serveur IIS de Microsoft).

Configuration du serveur web

Dans le cas d'Apache (serveur web utilisé chez BlueWeb), il suffit d'ajouter, dans le fichier `.htaccess` global de l'installation, les lignes suivantes :

```
AddType application/x-java-jnlp-file    .jnlp
AddType application/x-java-archive-diff .jardiff
```

ATTENTION **Éditer httpd.conf sous root**

Il faut des privilèges du type `root` pour configurer le fichier Apache `httpd.conf` sous Unix.

Vous pouvez aussi insérer ces instructions dans le fichier de configuration générale d'Apache `httpd.conf`.

Création du fichier .jnlp

N'importe quel éditeur de textes convient pour cette tâche. Il suffit juste de comprendre les balises utilisées par Java Web Start pour commencer l'édition de ce fichier.

Fichier de déploiement .jnlp

```
<jnlp href="bookmarks.jnlp" codebase="http://intranet/bookmarks/jnlp">
  <information>(1)
    <title>Gestion des Signets</title>
    <vendor>BlueWeb dev team</vendor>
    <homepage href="http://intranet/bookmarks"/>
    <description>Une petite application de gestion de signets.</description>
    <icon href="img/signet.gif"/>
    <offline-allowed />
  </information>
  <security>(2)
    <all-permissions />
  </security>
  <resources>(3)
    <j2se version="1.4 1.2+" />
    <jar href="lib/bookmarks.jar" />
    <jar href="lib/oro.jar" />
    <jar href="lib/httpclient.jar" />
    <jar href="lib/castor.jar" />
    <jar href="lib/log4j.jar" />
    <jar href="lib/swt.jar" />
    <jar href="lib/workbench.jar" />
  </resources>
  <application-desc main-class="com.blueweb.bookmarks.gui.MainGui" />(4)
</jnlp>
```

Un fichier `.jnlp` est divisé en quatre sections :

1. section `information` : pour des informations très générales telles que le nom de l'application, l'auteur, etc. ;
2. section `security` : pour la gestion des droits de l'applicatif par rapport au poste client (jusqu'à maintenant, seule la valeur indiquée était possible, mais ceci devrait changer dans le futur) ;
3. section `resources` : pour la description des ressources requises par cette application, qui doivent être empaquetées en fichiers `jar` signés numériquement ;
4. puis la description de l'application comprenant la classe principale à exécuter, ainsi que les éventuels paramètres passés à celle-ci (comme en ligne de commandes).

Ce fichier utilise des URL comportant le nom de la machine serveur sur laquelle l'application est déployée (pour le téléchargement). Dans le cas de BlueWeb, cette machine est la machine référencée sous le nom d'`Intranet` par les serveurs DNS internes.

RAPPEL

SWT utilise une bibliothèque native.

PRATIQUE Gérer l'hétérogénéité d'un parc

Le parc de machines clientes a été précisé comme étant uniquement constitué de machines Windows, ce qui est un cas extrêmement simple, voire simpliste. Souvent l'hétérogénéité prime et l'on peut trouver pêle-mêle des Mac, des PC, des stations de travail… SWT, quoique dépendant de bibliothèques natives, reste portable sur différentes plates-formes ; la distribution de notre application dans un tel contexte nécessiterait juste un raffinement de la page HTML permettant d'accéder au fichier `.jnlp` déployé sur notre serveur, en y ajoutant par exemple un script JavaScript permettant d'aiguiller vers différents fichiers `.jnlp` correspondant aux différentes architectures matérielles.

B.A.-BA Signature de documents électroniques

La signature électronique de documents utilise un algorithme asymétrique. Ce type d'algorithmes repose sur deux clés. La première, dite clé privée, doit rester secrète et permet de signer le document ; la seconde, dite clé publique, permettra de le déchiffrer ou de vérifier la signature. Cette dernière est publiée (donc tout sauf confidentielle, d'où son nom…).

La section `resources` contient la liste des `jar` utilisés par l'application (bibliothèques telles que ORO, SWT, etc.), ainsi que, bien entendu, le jar contenant l'application elle-même. Cependant, elle peut comporter aussi une section intitulée `nativelib` précisant les ressources natives requises pour le fonctionnement d'une application. Or, dans le cas de l'application développée par BlueWeb, nous devons justement utiliser cette balise car, ne l'oublions pas, SWT utilise une bibliothèque native (`.dll` sous Windows ou `.so` sous Unix) pour le lien avec la plate-forme matérielle.

Il faut donc modifier notre descripteur de déploiement `.jnlp` pour prendre en compte cette modification.

La section précédente mentionne l'utilisation de `jar` signés, mais de quoi s'agit-il ?

En fait, les techniques mathématiques dites de cryptographie permettent d'assigner des valeurs identifiant l'émetteur d'un message ou le créateur d'un document. Il s'agit donc de signer numériquement des documents permettant d'instaurer une confiance pendant des échanges – à la façon de la carte d'identité attestant de votre identité. Cette confiance peut-être renforcée via certaines sociétés comme Thawte, qui s'élèvent au rang d'autorité de certification (CA).

Dans le cas du déploiement via Java Web Start, il s'agit de donner à l'utilisateur qui se voit proposer un message lui demandant s'il veut bien installer l'application sur son poste, un gage de confiance pour qu'il accepte de le faire. Il s'agit en fait de le rassurer, puisque la signature du document (ici une archive au format `jar`) ne modifie en rien le contenu de ce document.

La manipulation de clés publiques et privées et la signature de `jar` est une tâche assez ingrate et lourde, que nous devons automatiser. Pas de problème, Ant vient à notre rescousse…

Empaquetage de l'application : intégrer la signature de jar dans un build-file Ant

Comme toute équipe de développeurs, l'équipe BlueWeb chargée de l'application de gestion des signets aime ménager ses efforts. Utilisant déjà Ant pour diverses tâches, elle décide naturellement de lui confier cette nouvelle charge. Cette étape d'empaquetage de l'application est une des illustrations de la puissance d'Ant et permettrait de justifier à elle seule l'utilisation de cet outil. En effet, la tâche est répétitive, peu enthousiasmante et donc toute désignée comme source d'erreurs difficiles à détecter.

Cependant, il faut préalablement se doter des outils nécessaires à la signature de documents, le minimum vital en la matière étant la possession d'un certificat.

Créer un trousseau de clés avec les outils du JDK

Un certificat est une clé numérique entreposée dans un trousseau (comme toute clé) : le *keystore*. Le JDK intègre tout ce qu'il faut pour cela, en particulier l'outil `keytool` conçu pour la gestion des clés numériques. Son utilisation dans le cas de BlueWeb est la suivante, pour créer un trousseau (le keystore) :

Utilisation de l'outil keytool pour créer un certificat de test

```
keytool -genkey -keypass bdncomp1 -storepass blueweb
Quels sont vos prÃ©nom et nom ?
  [Unknown] :
Quel est le nom de votre unitÃ© organisationnelle ?
  [Unknown] : blueweb
Quel est le nom de votre organisation ?
  [Unknown] : rd
Quel est le nom de votre ville de rÃ©sidence ?
  [Unknown] : space
Quel est le nom de votre Ã©tat ou province ?
  [Unknown] :
Quel est le code de pays Ã  deux lettres pour cette unitÃ© ?
  [Unknown] : FR
Est-ce CN=Unknown, OU=blueweb, O=rd, L=space, ST=Unknown, C=FR ?
  [non] : oui
```

Capture d'écran et accents mal gérés

Les symboles copyright et autres caractères étranges sont simplement le résultat d'une mauvaise gestion de l'encodage de caractères français dans une console. La capture omet la saisie du mot de passe permettant d'accéder au trousseau de clés.

Le trousseau de clés est maintenant créé (dans le répertoire `home`) sous le nom `.keystore`. Nous pouvons passer à la signature des documents…

ALLER PLUS LOIN Autorité de certification

Bien entendu, pour une utilisation en interne et pour un exemple, il est inutile de se compliquer la vie. En revanche, le choix des mots de passe (ou *pass-phrase*) est réellement important pour une utilisation professionnelle face à des clients. Ici, on a utilisé des certificats auto-signés, sans la moindre certification par une autorité tierce. C'est amplement suffisant techniquement, mais complètement inutile pour un réel déploiement, où il est indispensable d'acheter un certificat en contactant Verisign ou une autre autorité de certification reconnue en standard par les navigateurs.

Signer ses jars avec signjar

Là encore, le travail est minime puisque Ant intègre une tâche dédiée à cet usage : `signjar`. Voici un petit *build-file* de test utilisé par BlueWeb.

Script Ant utilisé pour signer un fichier jar avec notre certificat

```
<project name="test signature jar" default="main">
  <target name="main">
    <echo>Signature du jar</echo>
    <signjar jar="test.jar" alias="mykey" storepass="123456"
             keystore="/home/jerome/.keystore" keypass="bdncomp1"/>
    <echo> signature finie..</echo>
  </target>
</project>
```

Ce fichier très simple montre comment utiliser la tâche `signjar`. Par défaut, `keytool` crée le certificat dans le répertoire home de l'utilisateur. On peut remarquer l'utilisation de deux mots de passe distincts dans ce *build-file*, l'un permettant d'accéder au trousseau de clés (`123456` dans cet exemple), l'autre étant associé à la clé (alias `mykey` par défaut).

En sauvant ce *build-file* sous le nom `chap6-build-sign.xml`, vous obtenez (au chemin près) une sortie du type :

Lancement du script

```
ant -buildfile chap6-build-sign.xml
Buildfile: chap6-build-sign.xml
main:
     [echo] Signature du jar
  [signjar] Signing Jar : /shared/public/Mes documents/test.jar
     [echo] signature finie..
BUILD SUCCESSFUL
Total time: 5 seconds
```

À LA LOUPE Certificat unique pour les jars utilisés par Java Web Start

Java Web Start requiert que tous les `jar` nécessaires à l'exécution d'une application soient signés avec le même certificat. Par conséquent, si vous utilisez un `jar` déjà signé par son auteur, il faut prévoir de refaire l'empaquetage de ce code pour pouvoir le distribuer.

On peut remarquer la relative lenteur de cette tâche (5 secondes). C'est tout à fait normal étant donné les calculs effectués et la taille du fichier de test utilisé (le fichier `jar` original pesant près d'un Mo).

Les éléments nécessaires à l'intégration dans un processus de *build* sont à présent tous réunis, il ne reste plus qu'à les intégrer dans vos projets.

Déploiement sur le serveur web

Une fois votre application prête à être mise en production, il ne reste plus qu'à la placer dans le répertoire convenable sur votre machine faisant office de serveur web. Pour cela, dans la majeure partie des configurations, il ne s'agit que d'une simple copie de fichiers. Là encore, Ant peut vous être utile. Nous laisserons le soin au lecteur diligent de réaliser un petit *build-file* accomplissant cette fonction ou de l'intégrer dans un des fichiers déjà présentés. Il est inutile de rappeler que la documentation Ant vous sera utile, en particulier la section détaillant la tâche `copy`.

Répercussions sur le code client pour l'équipe BlueWeb

ALLER PLUS LOIN

Services de l'API Java Web Start

La vision donnée par cette application est un peu réductrice puisqu'elle s'affranchit, en raison de son architecture, des problèmes d'écriture/lecture sur le disque du poste client ou de problèmes d'impression. Dans le cas d'applications nécessitant de telles fonctionnalités (comme c'est le cas pour le Notepad distribué par Sun), il faut alors s'orienter vers les API JNLP offrant la notion de Services. Différents services sont disponibles (lecture, écriture, impression, etc.), tous s'utilisant assez simplement et s'obtenant par *lookup* (recherche) sur l'objet `ServiceManager`. Là encore, c'est une autre histoire…

En proposant un mode de déploiement par le Web, Java Web Start nous permet de diffuser élégamment notre application via un réseau (Internet ou intranet) en utilisant le plus commun des protocoles : HTTP. En revanche, on peut se demander quelles sont les répercussions, sur notre code client, de l'empaquetage sous forme de `jar` signés requis par cette technique.

Il faut bien comprendre que la seule différence se situe au niveau de l'empaquetage et concerne l'accès aux ressources (images, textes, sons s'il y a lieu).

La manipulation de ressources contenues dans l'archive d'une application distribuée via Java Web Start est assez lourde et un peu délicate. Le programmeur (de la partie cliente) est ainsi obligé d'utiliser la méthode `getResourceAsStream()` de la classe `java.lang.ClassLoader` pour charger une icône ou un fichier de configuration. Cette difficulté n'est pas insurmontable, mais elle alourdit le code et peut désarçonner un jeune développeur.

Voici un exemple de code issu de la documentation officielle Sun montrant comment charger des icônes stockées dans le `jar` contenant l'application :

Extrait de code montrant le travail nécessaire pour instancier une icône contenue dans un jar

```
// Get current classloader
ClassLoader cl = this.getClass().getClassLoader(); (1)
// Create icons
Icon saveIcon = new ImageIcon( cl.getResource( "images/save.gif" )); (2)
Icon cutIcon = new ImageIcon( cl.getResource( "images/cut.gif" ));
```

La première étape (1) consiste donc à obtenir une instance de la classe `ClassLoader` à partir de l'instance courante (et de la méthode `getClass()` renvoyant une instance de la méta-classe `Class` associée) tandis que la seconde étape (2) utilise la méthode `getResource()` pour instancier l'icône.

Il faut rappeler brièvement que Java utilise des objets, dont le rôle est d'assurer le chargement des classes, tout en veillant à ne pas enfreindre les règles de sécurité (implémentées dans l'objet `SecurityManager`). Java est doté d'une faculté de chargement des classes dynamiques, s'appuyant en partie sur les objets de la

classe `Class`, qui permet de manipuler des classes, attributs ou méthodes dynamiquement. Les méthodes `getResource()` et `getResourceAsStream` sont utilisées pour piocher des ressources (fichiers textes, sons, icônes).

Cette explication est loin d'être simple. Et pourtant, le code omet par souci de simplicité la gestion des exceptions.

En fait, pour des tâches aussi triviales qu'afficher des icônes, images ou lire un fichier de configuration, il existe une alternative beaucoup plus simple et conviviale, qui permet de prendre un peu de recul par rapport à la technologie sous-jacente.

Rachel, un auxiliaire de choix

La bibliothèque Rachel a justement pour but d'exécuter ces tâches triviales. Comme énoncé lors d'un chapitre précédent, l'idée de cette bibliothèque est simple mais très ingénieuse : elle consiste à proposer un nouveau type d'URL et à associer dynamiquement (lors de l'initialisation de l'application par exemple) une classe `StreamHandler` sachant comment traiter les URL utilisant ce protocole.

▸ http://rachel.sourceforge.net

Par la suite, on bénéficie directement du fait que nos ressources en tant qu'URL seront traitées comme s'il s'agissait de fichiers locaux ou de pages web.

On pourrait donc avoir la ligne suivante dans notre application de gestion des signets :

```
URL signetIconUrl = new URL( "class://BlueWebAnchor/images/signet.gif" );
```

Si l'on a pris la peine de déclarer comme suit notre `StreamHandler` sachant gérer le protocole dont les URL commencent par `class://` :

```
URL.setURLStreamHandlerFactory( new VampUrlFactory() );
```

ce petit effort permet incontestablement de simplifier le code et de s'éviter des tracas par la suite car la gestion des `ClassLoaders` dans Java Web Start est loin d'être simple…

Ainsi parée, l'équipe BlueWeb se trouve dans une situation plus confortable et peut appréhender le développement plus sereinement. Lors du codage malheureusement, l'équipe s'aperçoit que les temps de développement sont considérablement rallongés par la lourdeur inhérente au mode de déploiement via Java Web Start. En effet, une fois le code compilé, l'équipe cliente est obligée d'utiliser le *build-file* permettant de signer les `jar`, de les déposer sur le serveur web, puis de relancer l'application. Elle constate que la version 1.1 de Java Web Start n'est pas exempte de bogues dans la gestion des versions, bogues qui obligent à effacer à la main certains fichiers sur les machines de développement… Bref, cette approche n'est pas concluante pour la productivité de l'équipe (plusieurs minutes perdues pour une modification de quelques secondes).

B.A.-BA Encapsuler

Ce terme est un des mots-clés en programmation objet et une des clés du succès si vous voulez obtenir du code réutilisable et robuste. Il traduit l'idée de cacher les détails de votre implémentation en ne proposant qu'une interface (face visible) la plus stable possible. Pourquoi faire cela ? Moins le programmeur est lié aux détails de votre implémentation, plus vous avez de maîtrise sur celle-ci. Or, il faut bien se convaincre qu'un projet vit, et que les choix faits aujourd'hui peuvent être remis en cause demain. Comment mettre ce principe en pratique ? Il suffit de cacher vos attributs (visibilité privée) et, comme suggéré dans cet exemple, d'utiliser des couches d'abstraction permettant de vous prémunir contre d'éventuels changements de stratégie.

ATTENTION Le juste équilibre

Vous devrez toujours veiller à conserver un équilibre raisonnable entre performance et niveau d'abstraction dans votre code. Car si les couches d'abstraction font gagner en stabilité, elles alourdissent vos applications en terme de performances.

Il y a là un réel désir, pour l'équipe, de s'abstraire du mode de déploiement, qui dans ce cas a de trop grandes répercussions sur le travail. En réfléchissant au problème, l'équipe BlueWeb, finit par tirer profit de l'utilisation de Rachel. Rachel permet en effet de créer un dénominateur commun lors de l'accès aux ressources puisqu'il est évident que pour un déploiement classique, l'accès à un fichier peut se faire par des URL du type : file://images/monicone.gif, alors qu'avec Rachel l'accès à une image lors du déploiement via Java Web Start se fera par une URL du type : class://MonAncre/images/monicone.gif.

Pour illustrer le problème rencontré par l'équipe de développement, rappelons que pendant la phase de développement d'une version, les développeurs utilisent des fichiers (donc accessibles via des file://), alors qu'une phase de tests de version se fait sur une version empaquetée utilisant des fichiers `jar`. L'accès aux ressources étant différent dans ces deux contextes, il est important de tenter de l'uniformiser. C'est ce que permet Rachel.

Les design pattern en action : Adaptateur, Singleton et Factory

L'équipe se propose d'utiliser un composant qui permette de faire abstraction du mode de déploiement. Ce dernier sera responsable de créer, dans le contexte adapté, l'URL permettant enfin d'accéder à nos fichiers.

L'idée maîtresse peut être résumée par les diagrammes de classe et de séquence UML ci-après. Il s'agit simplement d'ajouter un niveau d'abstraction et de cacher l'utilisation de Rachel de manière à encapsuler les appels à cette bibliothèque.

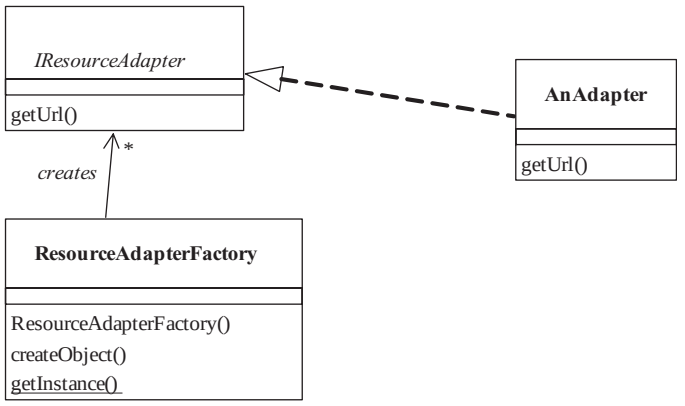


Figure 6–5 Le design pattern Adapter en action

Cette conception mêle différents design patterns et mérite un peu d'attention.

Le Singleton (même nom en français) vise à contrôler le nombre d'objets instanciés d'une classe. Dans son implémentation la plus classique, ce nombre est réduit à 1, d'où le nom… Dans le schéma, notre objet `Fabrique` (Factory) est codé suivant ce principe assurant l'unicité des instances d'une classe.

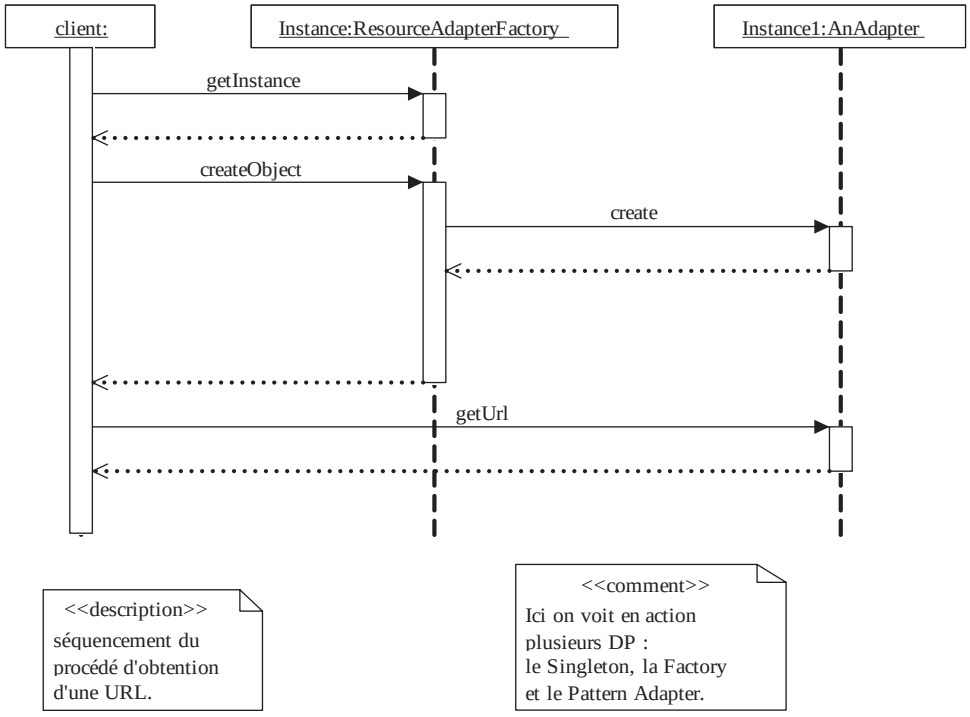


Figure 6–6 Obtention d'une URL

Le codage d'un singleton implique sur la classe `Singleton` les conditions suivantes :

- qu'elle ait un constructeur privé (qu'on ne peut donc pas appeler de l'extérieur) ;
- qu'elle mette à disposition l'instance par une méthode statique (qui ne peut pas être instanciée par un constructeur public) classiquement appelée `getInstance()`.

DESIGN PATTERN **L'adaptateur**

Le design pattern Adapter (adaptateur en français) a pour but d'adapter une interface exposée par une classe à une autre interface ou de simplifier l'implémentation d'une interface. Une utilisation remarquable de ce motif de conception est faite dans la bibliothèque graphique AWT, dans le domaine de la gestion des événements. En effet, les écouteurs d'un type d'événements (*Listener*) sont des implémentations d'interfaces parfois lourdes. Ainsi l'interface *WindowListener* permet de réagir dans le cas d'événements de fermeture, ouverture, mise en icône d'une fenêtre. Si un développeur veut juste réagir à un de ces cas, disons la fermeture d'une fenêtre (`windowClosed()`), il peut au choix implémenter toutes les méthodes requises par l'interface (solution lourde) ou hériter de la classe `WindowAdapter`, qui fournit des implémentations vides à toutes les méthodes de cette interface. Cette dernière solution est évidemment beaucoup plus rapide.

DESIGN PATTERN **Singleton**

Ce motif de conception a pour but de maîtriser le nombre d'instances d'une classe au sein d'un contexte (machine virtuelle en Java). Il tire son nom du cas particulier fréquent où ce nombre vaut un. Il est généralement de coutume de conjuguer ce design pattern avec le design pattern Factory, de manière à s'assurer qu'une seule instance de notre usine à objets est disponible.

DESIGN PATTERN **Factory**

Ce motif de conception a pour but de proposer un objet réalisant la construction d'autres objets : une usine à objets. Pourquoi une telle conception ? Cela sert pour camoufler les véritables classes d'implémentation ou encore pour permettre de modifier la classe utilisée suivant un paramétrage (fichier de configuration par exemple).

Prenons l'exemple du code Java suivant :

Exemple de classe illustrant le codage d'un singleton

```
public class SingletonDemo{
       // la variable singleton
       // va être utilisée par getInstance() puisque l'on désire
       // que cela soit la seule référence disponible
private static singletonInstance = new SingletonDemo();

private SingletonDemo(){
              // du code d'init...
       } // constructeur()

public static SingletonDemo getInstance(){
              return singletonInstance();
       } // getInstance()

// plus tout le code des services proposés par cette classe...
}
```

Hé oui, le constructeur est privé !

Cette méthode est le passage obligé depuis l'extérieur pour obtenir une référence sur cette classe (voir return SingletonDemo, instance tant désirée...).

Le design pattern Factory (Fabrique, Usine en français) vise à découpler le client des choix d'implémentations faits par le codeur d'une API. Cette façon de penser son code permet de s'adapter à différentes situations (contextes) sans que le code client soit touché. D'une utilisation extrêmement répandue, on le retrouve à de nombreux endroits dans le JDK comme dans des produits à large audience et s'avère très souvent couplé au Singleton. Ainsi, le paquetage `java.util` met à disposition la classe `Calendar` (abstraite) permettant de manipuler dates et calendriers. La documentation de l'API montre clairement que cette classe est en fait une Factory ! Le paquetage `java.net` lui aussi utilise ce design pattern pour s'abstraire du protocole ; c'est d'ailleurs grâce à cela que Rachel (vue précédemment) nous propose une solution aussi élégante et simple pour gérer l'accès aux ressources.

Pour cela, il faut s'assurer que le programmeur client manipule des classes abstraites (ou interfaces) sans jamais savoir quelle classe d'implémentation est effectivement utilisée pour rendre le service demandé.

POUR ALLER PLUS LOIN

Paquetage de sécurité JCE

On ne peut qu'inviter le lecteur à se pencher sur la conception du JCE (Java Cryptographic Environment), qui propose via sa structuration en SPI (Service Provider Interface, ou interface de fournisseur de services) une très belle illustration de ce que l'on peut faire avec le design pattern Factory.

En résumé…

Ce chapitre montre comment utiliser de manière pragmatique Java Web Start avec :

- Ant, pour faciliter la création des `jar` signés et le déploiement sur le serveur web ;
- certains design patterns, de manière à s'adapter à différents contextes en limitant les répercussions.

Bien entendu, tout n'a pas été dit sur Java Web Start ni sur les design patterns mentionnés, mais le lecteur motivé saura tirer profit de cette introduction pour aller glaner plus d'information sur Internet ou chez son libraire…

On peut retenir que Java Web Start abordé sous un tel angle s'avère être une solution très intéressante, puissante et naturellement intégrée dans votre poste client (livrée avec les JRE récents).

chapitre 7


Maintenance
Métrique
Non-régression
Portabilité
Extensibilité
Qualité

Audit du code et qualité logicielle

La qualité logicielle n'est pas une utopie mais sous-entend organisation, discipline et procédures. Voyons quelques *best practices* permettant de gagner en productivité dans le cas du travail en équipe. Pour cela, nous nous appuierons sur des indicateurs, les métriques de code, ainsi que sur des documents précisant les conventions de nommage au sein d'un projet/entreprise.

SOMMAIRE

MOTS-CLÉS

- Audit
- Métrique
- Qualité

Chartes et conventions de codage

Chaque société avait pour habitude de re-développer sa charte de codage, document officiel, fixant avec précision comment nommer une méthode ou un attribut ou encore comment indenter son code source.

Ceci va à l'encontre de la logique et de l'optique générale propice aux développements objets. Commençons par nous interroger sur la disponibilité via Internet de tels documents. Une simple requête avec Google et quelques mots-clés permet de trouver de nombreux pointeurs. Parmi ceux-ci, citons la convention de nommage d'Ambysoft, qui est réellement un document indispensable. C'est d'ailleurs cette convention que BlueWeb a choisie. Pourquoi utiliser un tel document, a priori trop important ou complexe pour nos besoins ? Simplement parce que le propre des besoins est d'évoluer – l'eXtreme Programming en tient parfaitement compte – et une équipe pouvant fluctuer, il est vital de faciliter l'incorporation de nouveaux arrivants en faisant en sorte qu'ils ne soient pas perturbés.

Une convention comme celle de Scott Ambler a le mérite de rassembler différentes pratiques couramment employées par la majorité des développeurs Java. Donc, adopter une telle convention c'est aussi faciliter le dialogue avec toutes les communautés de développeurs (assistance technique, forums, stagiaires ou nouveaux arrivants dans l'équipe).

L'effet Stroop (voir encadré) est un effet scientifiquement démontré, via les expériences du psychologue de même nom, nous forçant à prendre en compte dans tout projet les limites du cerveau humain. Celles-ci ont un effet immédiat sur la gestion à long terme d'une application informatique et notamment sur la phase de maintenance. En effet, il est évident que l'on est habitué à réagir à certains stimuli ; ainsi, un nom de classe du type `IUnNom.java` fait penser à un nom d'interface d'après les conventions en vigueur dans de nombreux projets Java. Une mauvaise utilisation de ces notations peut donc induire des pertes de temps indéniables lors d'une phase de recherche de bogues.

REMARQUE L'hypothèse judicieuse des besoins en constante évolution

La prise en compte de l'évolution des besoins dans les méthodes mêmes de conception est une des forces de l'eXtreme Programming. En effet, les besoins logiciels évoluent très vite. C'est un fossé qui sépare l'ingénierie logicielle de l'ingénierie civile.

📖 J.-L. Bénard, L. Bossavit , R. Médina , D. Williams. – *Gestion de projet eXtreme Programming, avec deux études de cas.* Eyrolles, 2002.

OUTIL Convention de nommage d'Ambysoft

▸ http://www.ambysoft.com/javaCodingStandards.html

B.A.-BA Conventions de nommage et digression sur l'effet Stroop

Les conventions de nommage sont consignées dans des documents souvent indigestes mais qui sont d'une nécessité absolue lors d'un développement collaboratif. Ils permettent en effet de donner une cohésion à l'ensemble du code source produit par une équipe. Il en découle des repères visuels permettant de faire gagner beaucoup de temps. Ces repères sont le pendant positif de ce qu'il est convenu d'appeler l'effet Stroop. Le psychologue J.Ridley Stroop a réalisé en 1935 une expérience d'une simplicité confondante pour mettre en évidence les limites du cerveau humain, via l'écriture de noms de couleur (bleu écrit en rouge, puis rouge écrit en bleu et ainsi de suite) suivie de la lecture à haute voix. Les résultats de cette expérience sont édifiants et permettent de comprendre les désastres sur le plan professionnel d'une mauvaise convention de codage ou, encore pire, de l'absence de convention. Le protocole de cette expérience et ses résultats sont publiés sur le Web. En un mot, l'étude prouve l'existence d'interférences entre différents types de stimuli sur notre cerveau, en particulier l'influence des couleurs sur l'interprétation de mots. Elle permet de comprendre l'importance capitale de la présentation du code en matière de maintenance du code.

▸ http://psychclassics.yorku.ca/Stroop/

On comprend donc bien l'intérêt de ce type de document pour prévenir les notations abusives et bâtir des fondations communes à tous les intervenants d'un projet. On pourrait, à titre d'exemple, regrouper toutes les constantes d'un paquetage dans une classe `Constants` en faisant ainsi une *Booch Utility class*, préfixer les interfaces par la lettre `I` pour distinguer visuellement dans un code source les classes abstraites des interfaces, etc.

B.A.-BA **Booch utility class**

Nom donné (par référence aux travaux de Grady Booch, un des inventeurs d'UML et méthodologiste de renom) à des classes ne pouvant être instanciées et offrant donc des services de portée de classe.

L'outil Checkstyle

Checkstyle est un outil visant à assurer le respect de certaines conventions de nommage, afin de garder un style de codage propre et simple. Bien entendu, cet outil peut être intégré dans un *build-file* Ant via la tâche du même nom.

▸ http://checkstyle.sourceforge.net/

Il est extrêmement configurable (supporte par défaut la convention de nommage de Sun issue des JSR). Il permet de choisir outre la convention de nommage à respecter, le type de sortie (fichier texte ou XML), ce qui permet d'automatiser la génération de rapports HTML envoyés par courrier électronique si l'on ajoute à cette tâche l'usage des tâches `style` et `mail`.

Obtention d'un rapport avec Checkstyle

Ainsi, le responsable Qualité chez BlueWeb désire vérifier chaque jour la conformité des sources produites par l'équipe avec les conventions adoptées (Sun) et souhaite que le contenu de cet audit de code lui soit envoyé par courrier électronique sous la forme d'une page HTML. Le serveur de courrier de Blueweb étant la machine nommée `mail.blueweb.com`, le build-file suivant permet d'accomplir ces objectifs.

Script Ant déclenchant la production d'un rapport d'audit du code source

```
<project name="BlueWeb" default="main" basedir=".">
  <target name="init" description ="crée les répertoires utilisés
     par la suite">
     <!-- utilise la task mkdir pour créer les répertoires -->
     <mkdir dir="reports"/>
     <mkdir dir="reports/html"/>
  </target>
  <!-- déclare la task puis l'invoque -->
  <target name="declare" depends="init">
     <!-- declare la task checkstyle -->
     <taskdef name="check"
        classname="com.puppycrawl.tools.checkstyle.CheckStyleTask"
     />
  </target>
  <!-- supprime les fichiers créés -->
  <target name="clean" description="prépare le grand ménage...">
     <delete dir="reports"/>
  </target>
```

◂ Le build-file est organisé en quatre cibles (targets) : `main` (cible principale), `clean` (destruction des fichiers produits), `declare` (qui déclare la `taskdef`) et `init`.

L'exécution de checkstyle se fera sur tous les fichiers Java (`**/*.java`) contenus dans le répertoire `src`. Ici, la sortie choisie est de type XML (`formatter type="XML"`).
Le fichier produit est traité par la tâche `style` pour opérer une série de transformations via XSLT. Attention, cette tâche requiert Xalan, donc se référer au manuel Ant pour l'installation de cette bibliothèque.
Cette tâche attend un paramètre `style="checkstyle.xsl"`, qui va définir le fichier XSL utilisé pour la transformation. 1

La tâche `mail`, elle aussi, requiert un certain nombre de bibliothèques (`action.jar`, `mail.jar`). 3
Le paramètre `mailhost` contient l'adresse IP ou le nom (entrée dans le DNS) du serveur SMTP utilisé pour l'envoi. Pour les utilisateurs de distributions Linux ayant installé Sendmail ou Postfix, cette valeur peut-être `localhost`.

```
  <!-- target principale (par défaut) -->
  <target name="main" depends="declare" description=
         "Target principale, dépend de la target declare">
    <check failureProperty="checkstyle.failure"
           failOnViolation="false">
        <fileset dir="src" includes="**/*.java"/>
        <formatter type="xml" toFile="reports/checkstyle_report.xml"/>
    </check>
    <!-- transforme le fichier XML de sortie en HTML via XSLT -->
    <style in="reports/checkstyle_report.xml" out="reports/html/
       ➥ checkstyle_report.html" style="checkstyle.xsl"/> 2


    <!-- envoi d'un courrier électronique s'il y a des erreurs -->
    <mail from="michel@blueweb.com"
          tolist="michel@blueweb.com,dev@blueweb.com"
          mailhost="mail.blueweb.com"
          subject="Checkstyle violation(s) in project
                     ➥     ${ant.project.name}"
          files="reports/html/checkstyle_report.html"/>
  </target>
</project>
```

1 Des exemples de fichiers tel `checkstyle.xsl` sont livrés dans le répertoire `etc` d'Ant (`$ANT_HOME/etc` sous Unix et `%ANT_HOME%\etc` sous Windows). La version `checkstyle-frames.xsl` pouvant poser problème, on conseille d'utiliser le fichier `checkstyle-noframes.xsl` en lieu et place.

À ce stade du build 2, nous disposons d'un fichier HTML (dans le répertoire `reports/html`) qu'il ne reste plus qu'à envoyer par courrier électronique.

3 Le manuel Ant, une fois de plus, vous aidera à installer ces bibliothèques .jar. Dans cet exemple, le responsable qualité de BlueWeb a demandé à l'administrateur système de créer sur son serveur de courrier un alias `dev@blueweb.com` contenant la liste des développeurs du projet.

Utilisation de métriques

Les métriques de code sont des indicateurs (rien de plus) permettant d'évaluer différents paramètres dont :

- le degré de dépendance d'un paquetage par rapport aux autres paquetages ;
- le degré d'abstraction d'un paquetage (plus il y a d'interfaces et de classes abstraites, plus ce degré est fort).

Leur intérêt pratique est de pouvoir visuellement jauger le degré de « réutilisabilité » d'un paquetage. En effet, plus un paquetage est dépendant des autres, moins il sera réutilisable et stable. Le pragmatisme veut que l'on ait un certain nombre de paquetages très stables (définissant des interfaces et classes abstraites), puis d'autres paquetages d'implémentation proposant du code pour les interfaces définies.

Exemple : le paquetage java.net

Dans ce paquetage de base du JRE, Sun a décidé d'opter pour une très grande flexibilité qui vous permet en tant que développeur d'utiliser d'une manière identique une URL pointant vers un site web via HTTP ou HTTPS (*SSL over HTTP*), vers un site FTP ou encore sur un serveur de news (NNTP). Tout ceci se fait de manière transparente pour le développeur client et réclame donc une mécanique d'abstraction du protocole basée sur une interface Java. C'est justement le propos de l'interface `java.net.URLStreamHandlerFactory`, qui permet d'associer à un nom de protocole (`http` par exemple) une classe héritant de la classe abstraite `java.net.URLStreamHandler`.

Ainsi, d'une manière très simple, vous pouvez ajouter un protocole lors de l'exécution de votre programme et associer à son nom la classe permettant de le gérer.

C'est d'ailleurs ainsi que dans la documentation de Rachel, son auteur Geral Bauer propose d'utiliser son paquetage d'accès à des ressources contenues dans des `jar` délivrés sur le poste client via JNLP, le protocole développé pour Java Web Start.

```
URL.setURLStreamHandlerFactory( new VampUrlFactory() );
```

Cette ligne permet d'ajouter un nouveau protocole… Les URL seront du type `class://unchemin/vers/une/ressource`. On enregistre la factory (`VampUrlFactory`) auprès de la classe URL via la méthode statique `setURLStreamhandlerFactory`.

Puis vous créez vos URL à l'aide des lignes de code qui suivent :

```
URL appIconUrl = new URL( "class://ThemeAnchor/images/inform.gif" );
ImageIcon appIcon = new ImageIcon( appIconUrl );
```

Ceci permet de créer une icône à partir d'une image contenue dans un `jar` distribué via Java Web Start. La classe `URL` va jouer le rôle de proxy sous-traitant à la bonne *factory* (celle en charge du protocole désiré) la charge d'instanciation de l'URL et de sa gestion.

OUTIL **Rachel**

Comme nous l'avons vu au chapitre précédent, Rachel est une bibliothèque permettant de manipuler le contenu d'archives au format `jar` d'une application distribuée via Java Web Start. On ne saurait que trop chaudement recommander cette bibliothèque, qui permet de diminuer considérablement le travail du développeur dans le cas d'un déploiement via Java Web Start.

- http://rachel.sourceforge.net

Outil rattaché aux métriques : JDepend

JDepend est un outil développé par Jim Clarke qui fournit des métriques qualifiant la qualité de conception du code pour un ensemble de paquetages Java.

- http://www.clarkware.com/software/JDepend.html

ALTERNATIVE **JMetra**

JMetra est un outil récent proposant aussi des métriques de code et qui s'intègre très bien à Ant. La visite de l'adresse suivante peut vous intéresser :

- http://www.hypercisioninc.com/jmetra.

Bien d'autres outils peuvent être envisagés. Une recherche sur Internet avec les termes `java metrics code` devrait vous fournir de nombreuses autres pistes.

ALTERNATIVE **Javancss**

L'outil javancss fournit un certain nombre de métriques sur votre code. Il peut être un complément intéressant à JDepend. Il ne propose néanmoins aucune tâche Ant, ce qui fait que son utilisation se borne pour l'instant à une invocation en ligne de commandes.

- http://www.kclee.com/clemens/java/javancss/

Il réalise un audit partiel (notre responsable qualité ne doit pas licencier ses développeurs à la simple vue d'un rapport…), mais qui permet quand même de fournir des indicateurs déjà significatifs sur la possibilité de maintenir et de réutiliser du code produit.

JDepend fournit plusieurs interfaces, dont une en mode console. C'est cette dernière que nous adopterons pour les exemples suivants.

Comprendre les dépendances entre paquetages

Admettons que JDepend produise un résultat du type :

```
com.blueweb.ejb
|  com.blueweb.servlets
|→   com.blueweb.ejb
```

Ceci indique :

- une dépendance entre `com.blueweb.ejb` et `com.blueweb.servlets` ;
- puis une dépendance entre `com.blueweb.servlets` et `com.blueweb.ejb`.

Nous avons donc affaire à une dépendance cyclique signifiant que les deux paquetages sont profondément liés. Cette dépendance est tellement forte qu'il est inimaginable de modifier l'un sans recompiler l'autre, leur réutilisation ne peut se faire qu'en bloc. Ceci peut bien sûr arriver, mais dénote généralement d'une faiblesse dans la conception, puisque ceci peut signifier :

- que les deux paquetages n'en font qu'un (ou devraient n'en faire qu'un) ;
- qu'un des deux paquetages possède une référence non désirée vers l'autre.

En se replaçant sur le plan architectural, un tel résultat montre un gros problème de conception, puisque dans le modèle à cinq couches présenté dans le chapitre consacré à l'architecture, la couche contenant la logique métier (donc vraisemblablement `com.blueweb.ejb`) ne dépend en aucun cas de la couche de présentation des données (`com.blueweb.servlets`). Cet outil est donc pertinent !

Voici un autre exemple :

```
com.blueweb.ui
|→   com.blueweb.ejb
|  com.blueweb.servlets
|→   com.blueweb.ejb
```

Quel dommage de ne pas avoir changé notre conception, car l'interface graphique se trouve à présent prisonnière des évolutions des paquetages `ejb` et `servlets`. On constate là encore un cycle entre `com.blueweb.servlets` et `com.blueweb.ejb`, mais en plus `com.blueweb.ui` (notre interface utilisateur), qui dépend d'un paquetage pris dans une relation cyclique, en devient totalement tributaire. De nouveau, l'examen de notre modèle à 5 couches devrait nous montrer qu'on ne souhaite pas qu'une telle chose se produise.

RAPPEL **Paquetages**

Cette notion est fondamentale en Java. Elle permet de réaliser des groupements (hiérarchiques) de classes par thèmes. Ceci a pour vocation première de prévenir tout risque de conflit (collision) entre classes. En effet, en prenant un exemple simple, comment faire pour distinguer votre classe `List` (structure de données contenant une collection d'objets utiles dans votre projet et dotée de certaines fonctionnalités) de celle fournie par Sun permettant d'afficher un composant graphique ? Ce problème est récurrent en informatique et Sun, avec la solution des paquetages, adopte une position proche (dans l'esprit) de celle du comité de normalisation du C++. Les paquetages Java permettent de plus de fournir des repères visuels à votre équipe.

TAO OBJET **Principes fondamentaux de l'objet**

La programmation objet, via une série de principes fondamentaux, a érigé une ligne de défense très forte contre les déviances engendrées par une mauvaise utilisation d'un langage objet et une mauvaise analyse. Ces principes (OCP, DIP, ADP, principe de substitution de Liskov, etc.) doivent figurer dans les bagages de tout utilisateur averti des technologies objet. Si l'on reprend l'exemple montrant des dépendances cycliques, le principe ADP (Acyclic Dependancy Principle) interdit formellement ce type de dépendances. Parmi tant d'autres pointeurs possibles, voici l'URL d'une excellente présentation (nécessite OpenOffice.org Impress ou PowerPoint) :

- http://www.design-up.com/data/principesoo.pdf
- http://www.cijug.org/presentation/OOPrinciples_JUG.ppt (en anglais)

Utilisation des indicateurs de paquetages

JDepend nous fournit toute une série d'indicateurs chiffrés caractérisant chaque paquetage (voir tableau 7-1).

Tableau 7–1 Vue synthétique des indicateurs proposés par JDepend

CC	Le nombre de classes concrètes du paquetage (donc toutes les classes non abstraites et sans les interfaces)
CA	Le nombre de classes abstraites et d'interfaces d'un paquetage.
Ca	Couplage afférent, nombre de classes en dehors de ce paquetage dépendantes de classes(/interfaces) définies dans ce paquetage.
Ce	Couplage sortant (*efferent*), nombre de classes de ce paquetage dépendantes de classes(/interfaces) définies dans un paquetage extérieur.
A	Abstraction du paquetage (nombre flottant variant de 0 à 1)
I	Instabilité (nombre flottant variant de 0 à 1)
D	Distance à la séquence principale
C	Si le paquetage contient une dépendance cyclique

L'instabilité totale d'un paquetage n'est pas problématique en elle-même, puisque si une application ne disposait que de paquetages totalement stables, il y a fort à parier qu'elle ne puisse pas réaliser grand-chose d'utile (car bien évidemment, sans implémentation, pas de code exécuté). Donc cet indicateur ne doit pas être considéré avec plus d'autorité que de raison.

En fait, d'après Robert Martin dans son célèbre article : *OOD Design Quality Metrics* en 1994, les paquetages stables (`I=0`) devraient être constitués de classes abstraites (en Java, par extension d'interfaces), tandis que les paquetages instables (`I=1`) devraient donc contenir des classes concrètes (non abstraites).

La ligne « idéale » ou *main sequence*, d'après Robert Martin, permet de situer nos paquetages relativement à un segment idéal et, par ce biais, de considérer si le degré d'abstraction d'un paquetage (indicateur `A`) est cohérent avec son degré de stabilité (`I`), sans tomber dans les extrêmes rarissimes au sein d'un projet :

- (`I=0,A=1`) ;
- (`I=1,A=0`).

Donc, via l'indicateur D (*distance from main sequence* ou distance à la séquence principale), Robert Martin nous propose un moyen de focaliser notre attention sur tous les paquetages dont la distance (`D`) est loin de 0 (donc proche de 1). À partir de quelle valeur de `D` faut-il commencer à réexaminer un paquetage ? Un chiffre de 0.2 ou 0.3 semble raisonnable, car il ne faut pas tomber dans l'intégrisme… De plus, il faut aussi tenir compte des classes efférentes dont sont dépendantes vos classes. En effet, certaines classes comme `String` et certaines autres des paquetages de base du JDK ont une stabilité assurée…

B.A.-BA **Interface homme machine (IHM)**

IHM, UI (User Interface) et GUI (Graphical User Interface) sont des mots quasiment interchangeables dans cet ouvrage comme dans bien d'autres documents d'informatique.

Cet article contient toutes les formules de calcul des différents indicateurs, ainsi que de nombreux commentaires et exemples. De plus, il propose le schéma original représentant la distance à la séquence principale.

- http://www.objectmentor.com/resources/articles/oodmetrc.pdf

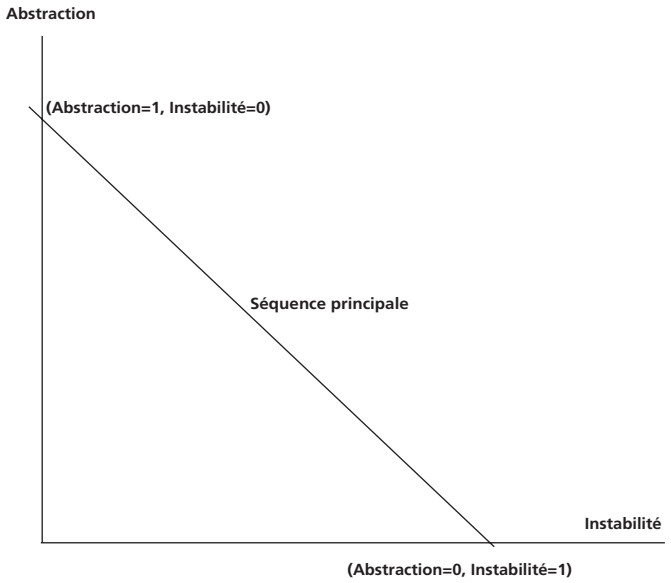


Figure 7–1
Distance à la séquence principale

La figure 7-1 (issue de l'article de Robert Martin) montre clairement ce qu'est la séquence principale. En ajoutant quelques points sur le diagramme, on est à même d'imaginer ce qu'est la distance à la séquence principale.

Utilisation de JDepend dans un build-file Ant

En plus de nous donner une vision assez claire de la qualité de conception dans notre projet, JDepend s'intègre très aisément dans un build-file, via la tâche optionnelle `JDepend` fournie avec Ant.

Pour cela, n'oubliez pas d'ajouter le fichier `optional.jar` dans votre `classpath`. Vous trouverez ci-après un build-file réalisant l'invocation de JDepend, la transformation du rapport produit du format initial XML en HTML puis l'envoi d'un courrier électronique. Mise à part l'invocation de la tâche `jdepend`, ce build-file ressemble à s'y méprendre au précédent. On voit en argument de la tâche jdepend, un exemple d'utilisation de `path-element` dans l'attribut `sourcespath`.

Script Ant démontrant l'invocation de JDepend

Initialisation

```
<project name="BlueWeb" default="main" basedir=".">

  <target name="init" description ="crée les répertoires utilisés par
la suite">
    <!-- utilise la tâche mkdir pour créer les répertoires -->
    <mkdir dir="reports"/>
    <mkdir dir="reports/html"/>
  </target>
```

```
  <target name="clean" description="prépare le grand ménage...">
    <delete dir="reports"/>
  </target>
  <target name="main" depends="init" description="Target principale,
     dépend de la target init">
    <jdepend outputfile="reports/jdepend.xml" format="xml">
      <sourcespath>
        <pathelement location="src"/>
      </sourcespath>
    </jdepend>
    <style in="reports/jdepend.xml" out="reports/html/jdepend.html"
           style="jdepend.xsl"/>
    <mail from="michel@blueweb.com"
          tolist="michel@blueweb.com"
          mailhost="mail.blueweb.com"
          subject="Métriques du projet ${ant.project.name}"
          files="reports/html/jdepend.html"/>
  </target>
</project>
```

Supprime les fichiers produits.

Target principale (par défaut)

Invoque la tâche jdepend.

Transforme le fichier XML de sortie en HTML via XSLT.

Envoi d'un courrier s'il y a des erreurs.

Tests unitaires : le framework JUnit

Ce framework, directement issu des pratiques prônées par l'eXtreme Programming et mis à disposition par Kent Beck et associés sur le site http://www.junit.org/index.htm, vise à fournir un outil assurant l'exécution de tests unitaires et de tests de non-régression. Son utilisation présente quelques limitations, mais il permet quand même d'avoir un outil minimaliste assurant un périmètre de fonctionnalités données. Son usage est très répandu, ce qui implique une grande diversité d'outils et d'articles gravitant autour de ce produit.

Nous ne rentrerons pas dans les détails d'utilisation de l'outil, ni dans ceux de son intégration avec Ant, mais ils sont largement commentés et donc toute recherche sur le Web devrait vous permettre de trouver matière à son apprentissage. Nous nous contenterons de citer les grandes étapes permettant d'utiliser ce framework :

1. Associer à chaque classe qui doit être testée, une classe de test héritant de `junit.framework.TestCase`.
2. Coder les méthodes de tests, une par fonctionnalité de la classe à tester.
3. Créer vos suites de tests (ensemble d'instances de la classe `TestCase`), en ajoutant vos objets `TestCase` à une instance la classe `junit.framework.TestSuite`, via la méthode `addTest()`.
4. Puis lancer la suite de tests via la méthode `run()`.
5. Éventuellement, regrouper les initialisations nécessaires à vos tests dans une méthode `setup()`, et procéder à la libération dans une méthode `tearDown()`.

À lire

- J.-L. Bénard, L. Bossavit, R. Médina, D. Williams, *Gestion de projet eXtreme Programming*, Eyrolles 2002
- J. Newkirk, R. Martin, *eXtreme Programming in Practice*, Addison Wesley 2001
- R. Hightower, N. Lesiecki, *Java Tools for eXtreme Programming*, Wiley 2002

Le chapitre 3 fournit une présentation synthétique de JUnit.

ALTERNATIVE **Jtest de Parasoft**

D'autres outils sont disponibles en ce domaine dont certains, comme celui de la société Parasoft, permettent d'aller beaucoup plus loin que JUnit. Malheureusement, de tels produits sont payants, difficilement accessibles et d'usage peu répandu.

- http://www.parasoft.com/jsp/products/home.jsp?product=Jtest

L'intégration de l'exécution de vos tests dans votre build-file se fera naturellement via une tâche optionnelle (`Junit`) dont le résultat peut lui aussi être transformé en HTML, via la tâche `style`, puis envoyé par courrier électronique via la tâche `mail`. Bref, pour vous maintenant rien de neuf à l'horizon…

Mise en forme du code source

L'aspect purement visuel du code source ayant une importance non négligeable, il est bon de se doter d'outils nous permettant d'améliorer automatiquement la mise en forme voire le contenu de notre code source.

Jalopy, disponible à l'adresse http://jalopy.sourceforge.net, permet de mettre en forme votre code source automatiquement en respectant certaines conventions de codage telles que :

- indentation du code ;
- positionnement des accolades ;
- ajustement des espaces.

Bien entendu, Jalopy s'intègre en douceur dans votre procédure de build, via une tâche Ant disponible dans la rubrique plug-ins de ce site.

Essayons prudemment cet outil en le testant sur notre code source, mais en fournissant un répertoire de destination (`destdir`), en utilisant la convention de nommage par défaut (Sun). C'est justement la fonction du build-file suivant.

ALTERNATIVE ImportScrubber

D'autres outils peuvent être insérés dans votre processus de build, et ce de manière à automatiser encore plus l'amélioration du code source partagé dans votre référentiel CVS. Par exemple, le projet `ImportScrubber` disponible à l'adresse :

- http://importscrubber.sourceforge.net/

permet d'automatiser l'analyse des imports de classes présents dans votre code source et de créer des classes de sortie conformes aux règles communément admises de bonne programmation en Java telles qu'éviter des imports de paquetages entiers en remplaçant cette ligne par autant de lignes que de classes de ce paquetage réellement utilisées dans votre classe.

Script Ant mettant en forme du code source avec Jalopy

La tâche Jalopy ne présente pas de difficulté particulière d'utilisation. Ce build-file montre ici un exemple de déclaration d'une tâche via `taskdef` et la définition du `classpath` (charge tous les `jar` présents dans le répertoire `lib` de la distribution Jalopy). Attention, vous devrez modifier la propriété `dir.jalopy` de manière à refléter votre installation (`c:\java\libs\jalopy` par exemple).

Définition de la tâche Jalopy.

```
<project name="BlueWeb" default="main" basedir=".">
  <property name="dir.jalopy" value="/dvpt/Java/jalopy"/>
<!-- initialisation -->
  <target name="init" description ="crée les répertoires utilisés par
    la suite">
  </target>
  <!-- supprime les fichiers produits -->
  <target name="clean" description="prépare le grand ménage...">
    <delete dir="dest"/>
  </target>
<!-- target principale (par défaut) -->
<target name="main" depends="init" description="invoque jalopy">
<taskdef name="jalopy"
           classname="de.hunsicker.jalopy.plugin.ant.AntPlugin">
    <classpath>
      <fileset dir="${dir.jalopy}/lib">
        <include name="*.jar" />
      </fileset>
    </classpath>
  </taskdef>
```

```
    <jalopy fileformat="unix"
            history="file"
            historymethod="adler32"
            loglevel="info"
            threads="2"
            destdir="dest"
            >
      <fileset dir="src">
        <include name="**/*.java" />
      </fileset>
    </jalopy>
</target>
</project>
```

Configurer votre propre convention

Jalopy vous permet d'utiliser la convention de votre choix et ce via un fichier XML devant être appelé depuis l'attribut `convention= « path/vers/convention.xml »`. Ceci vous permettra de refléter des paramétrages généraux tels que :

- le nombre d'espaces insérés à la place d'une tabulation (indentation) ;
- le style Kernighan & Richie ou C Ansi (définition du placement des parenthèses pour chaque bloc).

Attention, ce travail est relativement coûteux en temps de rédaction (du fichier) et en temps de test. Nous vous conseillons vivement d'utiliser un fichier existant tel que le fichier suivant (fichier Jalopy correspondant à la norme de codage du produit Maven, laissé tel quel dans le CVS du projet Maven pour la version 1.0b7).

Fichier de configuration de Jalopy

```
<jalopy>
    <printer>
        <alignment>
            <ParamsMethodDef>false</ParamsMethodDef>
            <varAssigns>false</varAssigns>
            <varIdents>false</varIdents>
            <ternaryExpresssion>false</ternaryExpresssion>
            <ternaryValue>false</ternaryValue>
        </alignment>

        <sorting>
            <variable>false</variable>
            <class>false</class>
            <modifiers>
                <use>false</use>
            </modifiers>
            <method>false</method>
            <order>Static Variables/Initializers,Instance Variables,
               Instance Initializers,Constructors,Methods,
               Interfaces,Classes
            </order>
```

Début du fichier de configuration avec un premier bloc permettant de gérer les alignements au niveau des définitions de méthodes, assignements de valeurs, etc.

Ce deuxième bloc contrôle le tri des éléments. Ainsi, avec cette configuration, on demande à Jalopy de ne pas trier les variables ou méthodes mais de trier les imports.

Fichier de configuration de Jalopy (suite)

```
            <use>true</use>
            <constructor>false</constructor>
            <orderModifiers>public,protected,private,abstract,
                static,final,synchronized,transient,volatile,
                native,strictfp
            </orderModifiers>
            <interface>false</interface>
        </sorting>

        <blankLines>
            <beforeHeader>0</beforeHeader>
            <beforeCommentSingleLine>1</beforeCommentSingleLine>
            <afterLastImport>1</afterLastImport>
            <afterPackage>1</afterPackage>
            <afterClass>1</afterClass>
            <beforeControl>1</beforeControl>
            <afterBraceLeft>0</afterBraceLeft>
            <keepUpTo>1</keepUpTo>
            <beforeJavadoc>1</beforeJavadoc>
            <beforeCommentMultiLine>1</beforeCommentMultiLine>
            <afterDeclaration>1</afterDeclaration>
            <afterInterface>1</afterInterface>
            <afterMethod>1</afterMethod>
            <afterBlock>1</afterBlock>
            <beforeBlock>1</beforeBlock>
            <beforeDeclaration>1</beforeDeclaration>
            <afterFooter>0</afterFooter>
            <afterHeader>1</afterHeader>
            <beforeBraceRight>0</beforeBraceRight>
            <beforeCaseBlock>1</beforeCaseBlock>
            <beforeFooter>0</beforeFooter>
        </blankLines>
        <header>
        <text>/*
            * The Apache Software License, Version 1.1
            (…)
            */</text>
            <smartModeLines>5</smartModeLines>
            <keys>The Apache Software License</keys>
            <use>false</use>
        </header>

        <whitespace>
            <paddingLogicalOperators>true</paddingLogicalOperators>
            <beforeMethodDeclarationParenthesis>false
            </beforeMethodDeclarationParenthesis>
            <padddingBrackets>false</padddingBrackets>
            <afterComma>true</afterComma>
            <paddingBitwiseOperators>true</paddingBitwiseOperators>
            <paddingAssignmentOperators>true
            </paddingAssignmentOperators>
            <beforeLogicalNot>false</beforeLogicalNot>
            <padddingTypeCast>false</padddingTypeCast>
```

Gestion des lignes vides. Ajoute (si non présente) une ligne vide après la ligne package, le dernier import, etc.

Fichier de configuration de Jalopy (suite)

```
            <paddingRelationalOperators>true
            </paddingRelationalOperators>
            <beforeStatementParenthesis>true
            </beforeStatementParenthesis>
            <afterCastingParenthesis>true</afterCastingParenthesis>
            <beforeBraces>true</beforeBraces>
            <beforeCaseColon>false</beforeCaseColon>
            <paddingMathematicalOperators>true
            </paddingMathematicalOperators>
            <padddingBraces>true</padddingBraces>
            <beforeMethodCallParenthesis>false
             </beforeMethodCallParenthesis>
            <afterSemiColon>true</afterSemiColon>
            <beforeBracketsTypes>false</beforeBracketsTypes>
            <padddingParenthesis>false</padddingParenthesis>
            <paddingShiftOperators>true</paddingShiftOperators>
            <beforeBrackets>false</beforeBrackets>
        </whitespace>
        <indentation>
            <label>false</label>
            <braceRightAfter>0</braceRightAfter>
            <extends>-1</extends>
            <implements>-1</implements>
            <leading>0</leading>
            <continationIf>false</continationIf>
            <parameter>-1</parameter>
            <general>4</general>
            <commentEndline>1</commentEndline>
            <firstColumnComments>true</firstColumnComments>
            <continationIfTernary>false</continationIfTernary>
            <tabs>
                <size>4</size>
                <use>false</use>
            </tabs>
            <caseFromSwitch>false</caseFromSwitch>
            <throws>-1</throws>
            <braceLeft>0</braceLeft>
            <deep>30</deep>
            <braceRight>0</braceRight>
            <useMethodCallParams>false</useMethodCallParams>
            <continuation>4</continuation>
            <braceCuddled>1</braceCuddled>
        </indentation>
        <comments>
            <javadoc>
                <addField>0</addField>
                <remove>false</remove>
                <templates>
                    <methods>
                        <return> * @return DOCUMENT ME!</return>
                     <exception> * @throws $exceptionType$ DOCUMENT ME!
                        </exception>
```

Ce bloc agit sur l'indentation (effet d'escalier) du code source. Ici, on définit une indentation à 4 caractères sans utiliser le caractère tabulation (Tab). Ce type de paramétrage permet d'éviter des désagréments dans le cas de l'utilisation d'un outil comme CVS…

Définition de l'allure des commentaires ; se fait en utilisant des modèles (`templates`). Bien entendu, rien de tel qu'un modèle pour changer de présentation en cas de besoin…

Fichier de configuration de Jalopy (suite)

```
                    <top>/**| * DOCUMENT ME!</top>
                    <bottom> */</bottom>
                    <param> * @param $paramType$ DOCUMENT ME!
                    </param>
                </methods>
            </templates>
            <tags>
                <in-line />
                <standard />
            </tags>
            <addClass>0</addClass>
            <checkInnerClass>false</checkInnerClass>
            <addCtor>0</addCtor>
            <addMethod>0</addMethod>
            <parseComments>false</parseComments>
            <checkTags>false</checkTags>
        </javadoc>
```

Ce bloc définit comment sont gérées les séparations entre les divers blocs d'un programme (classes).

```
        <separator>
            <method>Methods</method>
            <staticVariableInit>Static variables/initializers
            </staticVariableInit>
            <ctor>Constructors</ctor>
            <fillCharacter>·</fillCharacter>
            <interface>Interfaces</interface>
            <instanceInit>Instance initializers</instanceInit>
          <instanceVariable>Instance variables</instanceVariable>
            <class>Classes</class>
        </separator>
        <formatMultiLine>false</formatMultiLine>
       <insertSeparatorRecursive>false</insertSeparatorRecursive>
        <insertSeparator>false</insertSeparator>
        <removeMultiLine>false</removeMultiLine>
        <removeSingleLine>false</removeSingleLine>
    </comments>
    <footer>
        <keys />
        <text />
        <use>false</use>
        <smartModeLines>0</smartModeLines>
    </footer>
```

Ce bloc définit la troncature (wrapping) des lignes. La valeur utilisée (80 dans ce cas) est la valeur généralement admise, car elle permet d'éviter au lecteur d'avoir à défiler sur l'écran pour pouvoir lire une ligne de code.

```
    <wrapping>
        <arrayElements>0</arrayElements>
        <afterThrowsTypes>false</afterThrowsTypes>
        <afterImplementsTypes>false</afterImplementsTypes>
        <beforeThrows>false</beforeThrows>
        <afterChainedMethodCall>false</afterChainedMethodCall>
        <afterExtendsTypes>false</afterExtendsTypes>
        <paramsMethodCall>false</paramsMethodCall>
        <lineLength>80</lineLength>
        <beforeExtends>false</beforeExtends>
        <beforeOperator>true</beforeOperator>
        <paramsMethodDef>false</paramsMethodDef>
```

Fichier de configuration de Jalopy (suite)

```
            <beforeImplements>false</beforeImplements>
            <paramsMethodCallIfCall>false</paramsMethodCallIfCall>
            <afterLabel>true</afterLabel>
            <use>true</use>
        </wrapping>
        <braces>
            <removeBracesBlock>true</removeBracesBlock>
            <emptyInsertStatement>false</emptyInsertStatement>
            <treatMethodClassDifferent>false
            </treatMethodClassDifferent>
            <insertBracesIfElse>true</insertBracesIfElse>
            <removeBracesDoWhile>false</removeBracesDoWhile>
            <insertBracesWhile>true</insertBracesWhile>
            <removeBracesFor>false</removeBracesFor>
            <insertBracesFor>true</insertBracesFor>
            <removeBracesIfElse>false</removeBracesIfElse>
            <removeBracesWhile>false</removeBracesWhile>
            <rightBraceNewLine>true</rightBraceNewLine>
            <leftBraceNewLine>true</leftBraceNewLine>
            <emptyCuddle>false</emptyCuddle>
            <insertBracesDoWhile>true</insertBracesDoWhile>
        </braces>
        <history>
            <policy>History.Policy [disabled]</policy>
        </history>
        <environment />
        <chunks>
            <byComments>true</byComments>
            <byBlankLines>true</byBlankLines>
        </chunks>
    </printer>
    <transform>
        <import>
            <policy>ImportPolicy [expand]</policy>
            <groupingDepth>1</groupingDepth>
            <grouping>javax:2|java:2</grouping>
            <sort>true</sort>
        </import>
        <misc>
            <insertUID>false</insertUID>
           <insertLoggingConditional>false</insertLoggingConditional>
            <insertExpressionParenthesis>true
            </insertExpressionParenthesis>
        </misc>
    </transform>
    <general>
        <styleName>Sun</styleName>
        <styleDescription>Sun Java Coding Convention
        </styleDescription>
        <backupLevel>0</backupLevel>
        <backupDirectory>bak</backupDirectory>
```

Ici, on définit la gestion des accolades (*braces* en anglais).

Ce bloc définit la politique de transformation de code source.

Ici, il s'agit de la transformation des lignes d'imports.

Section générale, définissant principalement le type de convention utilisé (ici, il s'agit de la convention Sun).

Fichier de configuration de Jalopy (suite)

```
        <threadCount>1</threadCount>
        <sourceVersion>13</sourceVersion>
        <forceFormatting>false</forceFormatting>
    </general>
    <messages>
        <ioMsgPrio>30000</ioMsgPrio>
        <parserMsgPrio>30000</parserMsgPrio>
        <parserJavadocMsgPrio>30000</parserJavadocMsgPrio>
        <printerMsgPrio>30000</printerMsgPrio>
        <printerJavadocMsgPrio>30000</printerJavadocMsgPrio>
        <showErrorStackTrace>true</showErrorStackTrace>
        <transformMsgPrio>30000</transformMsgPrio>
    </messages>
    <internal>
        <version>4</version>
    </internal>
</jalopy>
```

Ce très long fichier de configuration montre le degré de paramétrage sur votre code source tel que vous pouvez l'obtenir avec Jalopy. Il faut donc retenir que ce produit permet de vérifier la conformité de votre code source avec la convention choisie et qu'il offre aussi la possibilité d'appliquer automatiquement une mise en forme spécifique.

Autres mises en forme

Comme toujours dans le monde du logiciel libre, la diversité et la multitude des projets permet de couvrir un large spectre de besoins différents avec, il est vrai, un risque de recoupements de fonctionnalités entre produits. Ant étant devenu un projet majeur, un grand nombre de projets gravitent autour de lui et, bien entendu, plusieurs d'entre eux s'intéressent à la mise en forme du code source.

Si Jalopy est peut-être le projet le plus connu, il est bon de citer d'autres initiatives permettant d'obtenir des fonctionnalités similaires ou complémentaires. Cette section va proposer différents outils, méritant eux aussi d'entrer dans votre boîte à outils de responsable qualité.

Gestion des directives d'import de classes Java

L'utilisation de paquetages en Java, structurant l'information, soulève diverses polémiques qu'il est bon d'aborder dans cette partie. En effet, par souci de confort pendant leur travail, certains développeurs prennent l'habitude de réaliser des imports en utilisant l'expression régulière `.*` signifiant au compilateur le désir de pouvoir accéder à toutes les classes accessibles du paquetage précisé. Cette pratique est évidemment tout à fait compréhensible et même économique d'un point de vue temps de codage, mais elle pose beaucoup de problèmes pour la maintenance d'un projet, puisqu'elle a le fâcheux défaut de masquer les dépendances réelles de la classe. C'est pourquoi il est considéré comme une bonne pratique de n'utiliser qu'un import exact des classes utilisées dans un projet.

C'est à ce niveau-là que les outils doivent entrer en jeu, car on ne peut obliger le développeur à allonger ses cycles de développement ni renoncer à obtenir une maintenance efficace de son code. Bien entendu, divers produits de développement (comme Eclipse) intègrent déjà des fonctionnalités de *refactoring* de code, mais leur utilisation repose sur la bonne volonté des programmeurs et n'ont rien de systématique.

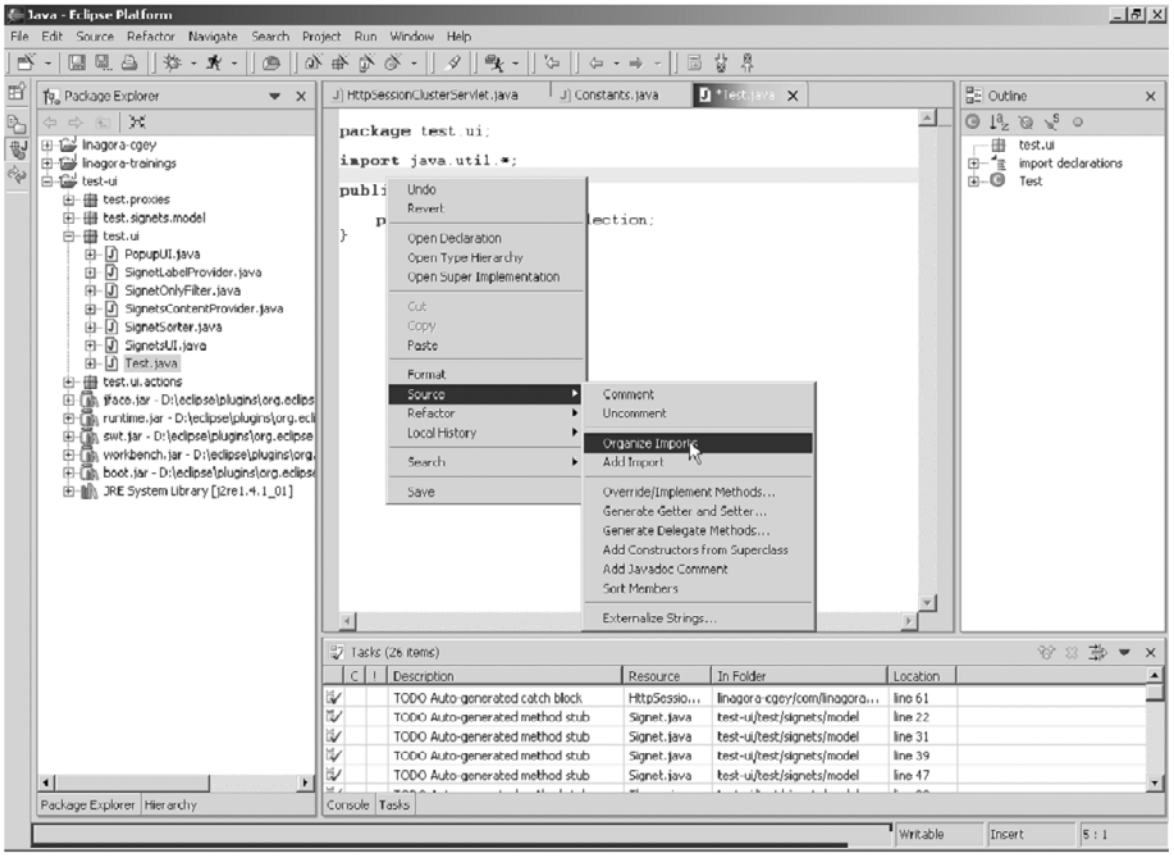


Figure 7–2 Capture d'écran montrant comment lancer l'organisation des imports depuis Eclipse

Comme d'habitude, pas de panique, Ant est la solution vous permettant d'automatiser une telle tâche. Il ne reste plus qu'à trouver un produit à même de faire ce travail.

Diverses solutions existent, telles que :

- ImportScrubber ;
- CleanImports.

Opter pour de tels outils revient à laisser libres ses développeurs de choisir le style de développement qui leur convient le mieux tout en conservant un code source de qualité. Autant garder tous les avantages…

OUTILS

- http://importscrubber.sourceforge.net/
- http://www.euronet.nl/users/tomb/cleanImports/index.html

Voici un exemple commenté d'utilisation d'ImportScrubber, qui est la solution apparue la première.

Extrait de script Ant démontrant l'utilisation de la tâche ImportScrubber

Ici, on définit une cible nommée `scrub` permettant de lancer la procédure de nettoyage de code source.

Ici l'on procède à la déclaration de la tâche `ImportScrubber` (sous le nom `scrub`) en fournissant le nom de la classe Java contenant la tâche utilisée (`net.sourceforge.importscrubber.ant.ImportScrubberTask`)

Il s'agit là d'un *hack* permettant de s'assurer que le compilateur ne supprime pas de références de classes (avec l'utilisation de l'option de debug)

Voici l'appel à la tâche `ImportScrubber`. Plus réduit et simple semble difficile…

```
<target name="scrub">

     <taskdef name="scrub" classname="net.sourceforge.importscrubber.
                                    ➥ ant.ImportScrubberTask"/>

    <javac
       deprecation="false"
       debug="true"
       optimize="false"
       srcdir="${sourceDir}"
       destdir="${sourceDir}"
       classpath="${libDir}bcel.jar;${libDir}junit.jar"/>
    <scrub root="${sourceDir}" classRoot="${classesDir}"
         format="top" recurse="true"/>
    <delete>
       <fileset dir="${sourceDir}" includes="**/*.class"/>
    </delete>
  </target>
```

Comme le montre cet exemple, obtenir un code source propre se fait donc simplement et automatiquement. Voici un exemple de résultat de l'exécution de ce produit sur un code simple :

Avant	Après
`Package com.foo.bar;`	`package com.foo.bar;`
`import java.io.*;`	`import java.io.File;`
`import javax.swing.*;`	`import java.net.SocketOptions;`
`import com.us.Something;`	`import java.net.URL;`
`import javax.swing.JLabel;`	`import java.net.URLConnection;`
`import java.io.IOException;`	
`import java.net.*;`	`import javax.swing.JLabel;`
`import com.us.SomethingElse;`	`import javax.swing.JOptionPane;`
`import java.lang.Integer;`	
`import java.awt.*;`	`import com.us.Something;`
`import java.awt.Frame;`	`import com.us.SomethingElse;`

Quel est l'effet de ce produit ? Organiser logiquement (alphabétiquement et avec une précédence aux paquetages Sun) les lignes d'import, tout en uniformisant ceux-ci en n'utilisant que des noms explicites (plus d'utilisation du `.*`).

ASTUCE **Outils reposant sur l'analyse des fichiers class**

ImportScrubber, comme de nombreux autres outils du même type, utilise une bibliothèque d'analyse du code binaire (*byte code* Java), en l'occurrence la bibliothèque BCEL du projet jakarta. Étant données les spécifications de la machine virtuelle Java et le format des classes Java, une bonne utilisation du produit tiendra compte du fait qu'il vaut mieux éviter de déclarer plusieurs classes au sein du même fichier Java. C'est une des limitations de ce produit.

- http://importscrubber.sourceforge.net/limitations.html

Analyse du code source

Même bien formaté et passant le stade de la compilation, un fichier source (qu'il soit en Java, en C ou dans tout autre langage) peut renfermer différents pièges rendant ardue sa maintenance, surtout si elle est assurée par un développeur n'ayant pas développé ce composant.

Quels sont ces pièges ? Voici une liste non exhaustive de différents travers complexifiant la tâche de maintenance d'une application :

- variables non utilisées (le développeur chargé de la maintenance pourra perdre du temps à essayer de comprendre à quoi elle sert, et ce en pure perte) ;
- paramètres de méthodes non utilisés (même problème que précédemment, mais en affectant aussi les développeurs utilisant cette méthode) ;
- méthodes inutilisées (pourquoi aller chercher d'éventuelles bogues dans une méthode non utilisée ?) ;
- exceptions non traitées (par des blocs `try/catch` laissés vides).

Comme nous en prenons l'habitude, on peut de nouveau proposer une solution du monde du logiciel libre permettant de détecter ces différents problèmes. Cette solution se nomme PMD.

OUTIL **PMD**

- http://pmd.sourceforge.net/

PMD dispose d'une ouverture vers différents IDE du marché (Jedit, JBuilder, Emacs, Eclipse et bien d'autres) mais aussi d'une tâche Ant prête à l'emploi. Le listing suivant illustre un extrait de build-file Ant typique.

Extrait de script Ant démontrant comment appeler PMD depuis Ant

```
<target name="pmd">


   <taskdef name="pmd" classname="net.sourceforge.pmd.ant.PMDTask"/>




   <pmd rulesetfiles="rulesets/imports.xml" shortFilenames="true">
            <formatter type="html" toFile="pmd_report.html"/>
            <fileset dir="${src.dir}">
                <include name="**/*.java"/>
            </fileset>
   </pmd>
</target>
```

Création d'une cible intitulée `pmd`, contenant tous les appels nécessaires à l'obtention d'un joli rapport HTML.

Comme avec toutes les tâches dites externes, nous devons définir la tâche que nous allons manipuler en associant à un nom (ici `pmd`) le nom d'une classe Java héritant de la classe `Task` définie dans le paquetage `Ant`.

Ici se trouve l'appel à la tâche `pmd` en elle-même. On précise le fichier contenant les règles guidant l'analyse. Le rapport produit est au format HTML (`pmd_report.html`) et se basera sur tous les fichiers Java contenus dans le répertoire pointé par la variable `${src.dir}`.

Il est à noter que PMD vous permet de créer vos propres règles et de produire un fichier en XML, qui pourra être ensuite transformé en HTML par la désormais classique tâche `Style`.

La page web indiquée à l'adresse ci-contre permet de recenser le résultat de l'exécution de PMD avec une règle de détection de code inutilisé sur différents projets Java (Ant, ORO, log4j, etc.).

EXEMPLE **PMD sur Ant, ORO, ...**

- http://cvs.apache.org/~tcopeland/pmdweb/

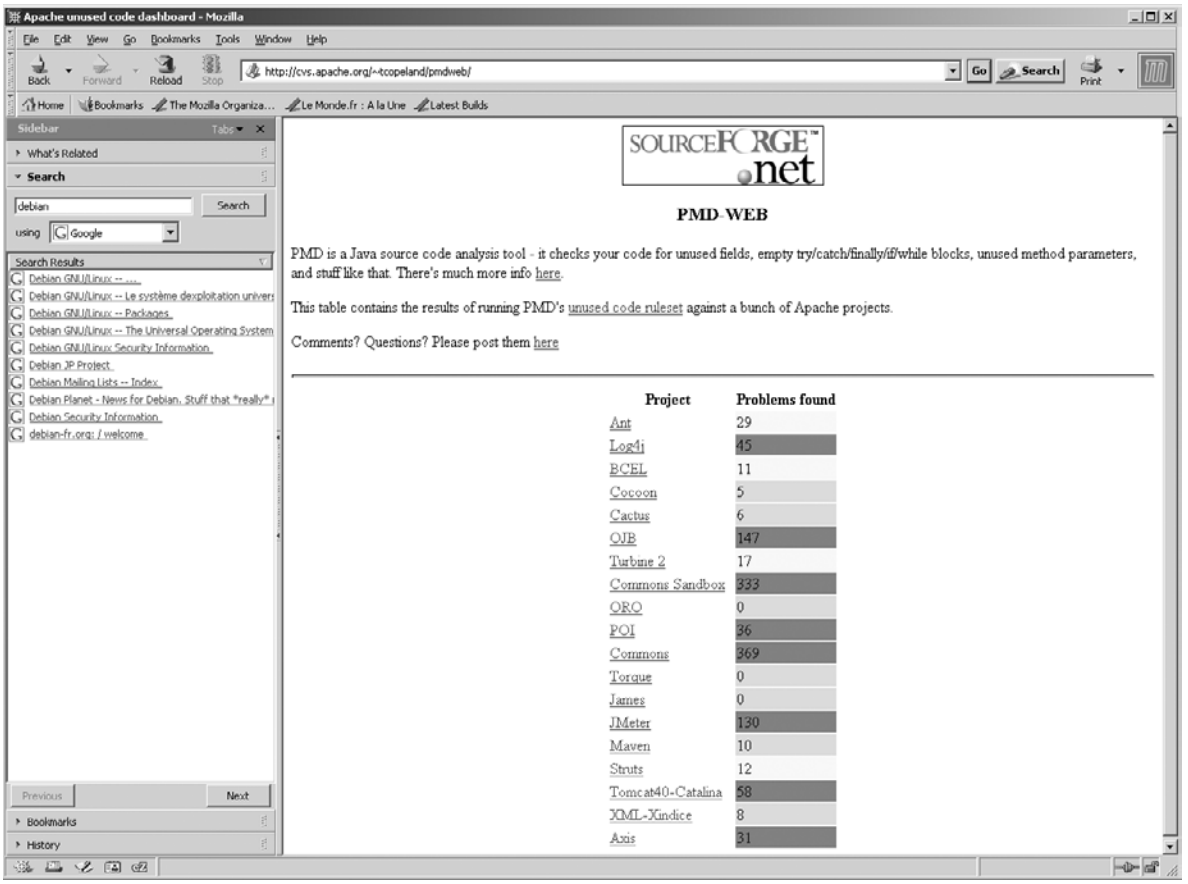


Figure 7–3 Différents projets Apache passés à l'examen par PMD

Il s'agit là d'un outil réellement indispensable à tout responsable qualité, car il permet de prévenir des problèmes sournois. L'exemple le plus frappant est sans aucun doute l'utilisation d'une règle réfutant dans le code la présence d'exceptions non traitées, car cela implique, sans réaction de la part de l'équipe menant à terme le projet incriminé, le fait de laisser partir en clientèle un code pouvant ne pas fonctionner et ce sans que l'on en ait la moindre trace.

Renforcement de l'architecture d'un projet

Une équipe peut mettre en place de belles architectures logicielles (convenablement structurées, délimitant strictement les responsabilités de chaque couche, essayant de demeurer génériques et donc non dépendantes d'implémentations particulières en utilisant des motifs de conception comme la fabrication d'objets par des Factory), elle reste toutefois menacée par un développeur ne jouant pas le jeu et utilisant des implémentations particulières sans passer par les Factory mises en place.

Il est extrêmement difficile de détecter ce cas par revue du code source, puisqu'il faut analyser la (ou les) classe(s) concernée(s) avec précision pour découvrir le problème.

Exemple concret chez BlueWeb

Admettons que, pour réaliser une tâche utilitaire, nos amis de la société BlueWeb aient créé une hiérarchie de classes Java telle que présentée dans la figure 7-4.

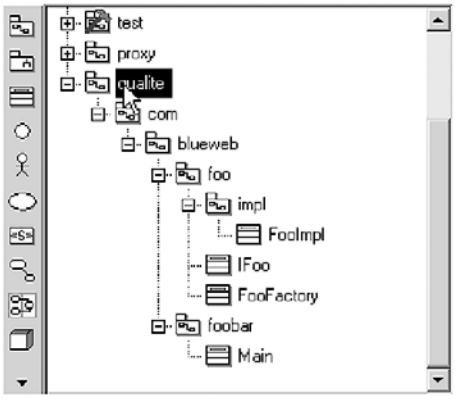


Figure 7–4
Hiérarchie de paquetages

Cette figure nous propose donc :

- un paquetage `com.blueweb.foo`, contenant une interface `Ifoo` et une classe concrère `FooFactory` ;
- un paquetage `com.blueweb.foo.impl`, contenant les classes concrètes implémentant l'interface `IFoo` (ici, une seule classe `FooImpl`) ;
- et enfin, un paquetage `com.blueweb.foobar`, contenant une classe `Main` utilisant les services définis dans l'interface `Ifoo`.

Cette conception est assez classique, le nom des classes en lui-même laisse pressentir que les concepteurs cherchent à séparer clairement la définition des services et leurs implémentations, de manière à pouvoir changer ces dernières de manière transparente pour le client. Cette conception laisse donc suggérer un séquencement d'appels du type :

1. Obtenir une référence sur la Factory (`FooFactory`).
2. Appeler la méthode `createObject()` sur la Factory de manière à récupérer une implémentation de l'interface `IFoo`.
3. Puis invoquer le service désiré sur l'instance créée par notre usine à objets.

Mais que se passe-t-il si un programmeur BlueWeb distrait ou inconscient (peut-être fatigué…) ne respecte pas ce schéma et instancie directement la classe `FooImpl` ? Ceci peut-être fait par un extrait de code du type suivant.

Cas d'une non-utilisation du motif de conception Factory (usine)

```
package com.blueweb.apackage;
import com.blueweb.util.foo.impl.FooImpl;
/**
* @author jm
*/
public class Main {
   public static void main(String[] args) {
      new FooImpl();
   }
}
```

Dans un cas comme celui-ci, toute la belle conception tombe à l'eau et l'indépendance par rapport à l'implémentation n'est pas respectée, ne laissant augurer que des problèmes.

Solution logicielle : Macker

Là encore, Ant va devenir votre allié en vous proposant un moyen simple et élégant de vous prémunir contre ce type de manquements aux conceptions mises en place. Pour cela, il suffit d'intégrer dans vos scripts Ant un appel au logiciel Macker. Ce produit permet de créer des règles (au format XML) vous permettant de vérifier que vos motifs de conception sont bien respectés. Une fois intégré dans Ant, vous pourrez dormir sur vos deux oreilles…

▸ http://innig.net/macker/

Pour cela, Macker exige un fichier de règles listant les accès autorisés et ceux interdits pour votre application (ainsi, dans l'exemple précédent, on interdirait l'accès au paquetage contenant les implémentations depuis tout paquetage différent de `com.blueweb.foo`).

La syntaxe utilisée par Macker est du XML pouvant ressembler à ceci (il s'agit d'un des exemples simples fourni dans la distribution du produit) :

Fichier de configuration des règles de Macker

Macker utilise un fichier de configuration XML. Rien d'original en fait…

```
<?xml version="1.0"?>
<macker>
```

Définition d'une règle nommée `Modularity rules`

```
    <ruleset name="Modularity rules">
```

Définition d'une variable pointant vers le paquetage de base

```
        <var name="module-base"
             value="net.innig.macker.example.modularity" />
```

Itération pour chacun des sous-paquetages contenus. On s'intéresse aux noms de paquetages contenant la chaîne `impl`.

```
        <foreach var="module" regex="${module-base}.(**).impl.**">
```

Définit un motif appelé `inside`, qui permettra de repérer le paquetage contenant les classes d'implémentation. Celles-ci ne devront pas être accessible depuis n'importe quel paquetage.

```
            <pattern name="inside"
                     regex="${module-base}.${module}.impl.**" />
```

```
            <pattern name="factory"
                    regex="${module-base}.${module}.*Factory" />


            <access-rule>
                <message>${from} must access the ${module}
                    module through its API</message>
                <deny> <to pattern="inside" /> </deny>
                <allow><from pattern="inside" /></allow>
                <allow><from pattern="factory" /></allow>
            </access-rule>

        </foreach>
    </ruleset>
</macker>
```

Définit un autre motif, permettant de désigner les paquetages contenant les usines à objets (points d'entrées obligatoires dans notre conception).

Après les définitions, on définit la politique d'accès aux paquetages. Ici, on n'autorisera l'accès aux classes des paquetages contenant les implémentations que depuis d'autres classes des mêmes paquetages (pour permettre de l'héritage par exemple) ou depuis les paquetages contenant des usines. Il est à noter que l'on définit un message à afficher en cas de manquement à cette règle.

Il ne reste plus qu'à observer le résultat du passage de Macker sur notre exemple précédent. Pour cela, adaptons la définition de la règle de manière à scruter nos paquetages (`com.blueweb.*`) :

Fichier de règles adapté au contexte de BlueWeb

```
<?xml version="1.0"?>
<macker>
    <ruleset name="Modularity rules">
        <var name="module-base" value="com.blueweb" />

        <foreach var="module" regex="${module-base}.(**).impl.**">
            <pattern name="inside"
                     regex="${module-base}.${module}.impl.**" />
            <pattern name="factory"
                     regex="${module-base}.${module}.*Factory" />
            <access-rule>
                <message>${from} must access the ${module}
                       module through its API</message>
                <deny> <to pattern="inside" /> </deny>
                <allow><from pattern="inside" /></allow>
                <allow><from pattern="factory" /></allow>
            </access-rule>
        </foreach>
    </ruleset>
</macker>
```

Puis utilisons un petit script Ant de test pour voir l'effet produit sur notre code.

Script Ant démontrant comment invoquer Macker

```
<project name="macker-test" default="main">


<!-- Declare a catch-all classpath for the project, its libs, and
Macker -->
```

On crée un petit projet de test dénommé `macker-test`, dont la cible par défaut est la cible `main`.

Script Ant démontrant comment invoquer Macker (suite)

On définit un `classpath` permettant d'accéder aux bibliothèques utilisées par Macker.

```
<path id="build.classpath">
    <pathelement location="${build.classes.dir}" />
</path>
<path id="macker.classpath">
    <pathelement location="${build.classes.dir}" />
    <fileset dir="${lib.dir}" />
    <fileset dir="${macker.lib.dir}" />
</path>
<property name="macker.classpath" refid="macker.classpath" />
```

Cette cible contient tout le code nécessaire à l'invocation de Macker.

```
<!-- Run the darn thing! -->
<target name="macker">
```

Comme d'habitude, on déclare notre tâche, qui n'est pas connue de Ant, par un `taskdef`.

```
<!-- Declare the Macker task -->
        <taskdef name="macker"
    classname="net.innig.macker.ant.MackerAntTask"
    classpath="${macker.classpath}" />
```

Ici, on se charge d'invoquer Macker en plaçant dans le répertoire courant le fichier XML précédent (celui contenant notre règle), en prenant soin de le sauver sous un nom du type `macker-rules.xml`

```
<macker>
    <rules dir="." includes="**/*macker*.xml" />
```

Ici, on précise les fichiers `.class` devant être analysés.

```
<classes dir="${build.classes.dir}">
        <include name="**/*.class" />
    </classes>
```

Ici, on définit le paquetage de base (dans notre cas `com.blueweb` est une valeur raisonnable).

```
<var name="basepkg" value="com.blueweb" />
    <classpath refid="build.classpath" />
    </macker>
</target>
<target name="main" depends="macker"/>
</project>
```

Ce script, pour pouvoir être lancé, demande la création d'un fichier *properties* contenant la valeur de certaines variables utilisées (par exemple build.classes.dir ou lib.dir). Voici un exemple de tel fichier pouvant vous servir à l'adaptation sur votre machine :

Fichier de propriétés nécessaire à l'exécution du script Ant

```
lib.dir=/dvpt/Java/lib
macker.lib.dir=/dvpt/Java/macker/lib
src.dir=macker-test-src
build.classes.dir=macker-build
```

En lançant Ant sur notre script et en utilisant le fichier de propriétés proposé on obtient une sortie du type de la figure 7–5.

Cela nous permet bien d'obtenir le résultat escompté : protéger notre conception contre les raccourcis d'un programmeur inconscient.

POUR ALLER PLUS LOIN **Macker**

Dans l'immédiat, Macker ne dispose pas d'une sortie au format XML des informations nées de l'examen des fichiers `.class` de votre projet, mais cette tâche devrait voir le jour sous peu. C'est un peu dommage, car cela limite l'utilisation de Macker et nous prive de jolis rapports au format HTML envoyés automatiquement chaque nuit.

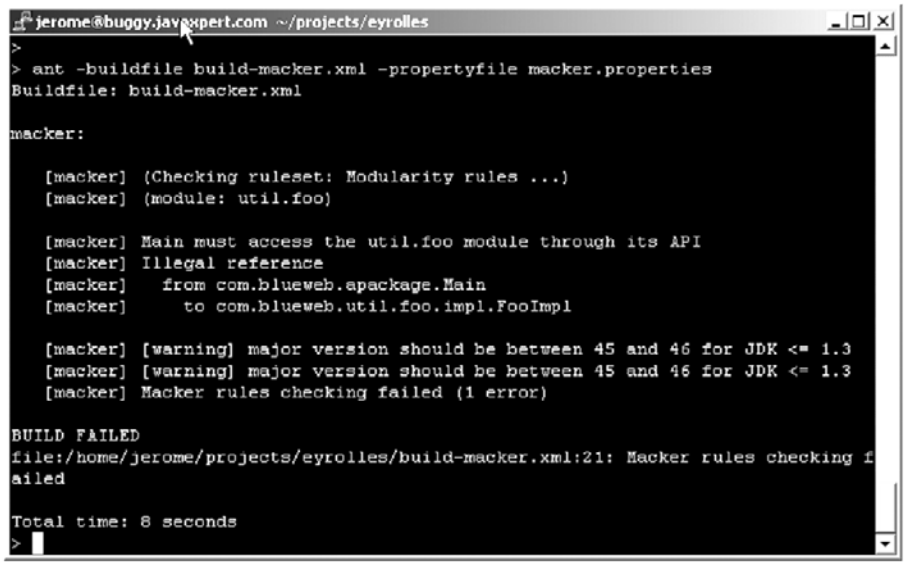


Figure 7–5 Sortie écran représentant l'exécution de notre script Ant dans le cas d'une violation de nos règles

Avec un outil comme Macker, on peut donc disposer d'un garde-fou efficace, simple à mettre en œuvre et s'intégrant très bien dans Ant et donc dans votre processus de déploiement. Là encore, pourquoi s'en priver ?

Interaction avec CVS

Michel, responsable qualité chez BlueWeb, n'a pas pu rester insensible à l'une des fameuses pratiques issues d'XP, les *nightly builds*. Il voit en Checkstyle et JDepend de très bons outils lui permettant d'obtenir chaque matin un état de santé de la qualité de ce projet et par conséquent n'imagine pas une seconde se priver d'une telle manne d'informations à bon marché. Il reste cependant confronté à une difficulté technique : l'obtention des sources correspondant à la version désirée : il peut s'agir de la dernière version, de celle en cours de développement ou de toute version ultérieurement définie.

Ne parlons plus de difficulté technique car la gestion des versions reste le problème de CVS et cet outil la gère bien. Ant, en bon outil de *make*, ne pouvait pas manquer de nous fournir des tâches permettant de dialoguer avec notre serveur de sources.

Il n'y a donc plus de problèmes, il reste seulement à domestiquer la tâche CVS et c'est justement le propos du build-file suivant, qui ne fait qu'extraire les sources du serveur CVS et les déposer dans le répertoire `CVS-out`.

Ceci est en fait un *check-out* des sources (copie en local), qui est l'une des opérations basiques autorisées par les gestionnaires de sources.

ALTERNATIVE Clearcase ou Visual Source Safe

Bien entendu, Ant ne se limite pas au seul dialogue avec CVS, alors si vous utilisez ClearCase (de Rational) ou Visual Source Safe (Microsoft), ou encore Continuus, vous trouverez des tâches optionnelles vous permettant de réaliser le couplage avec votre serveur de sources.

B.A.-B.A Check-out et check-in CVS

Un *check-out* consiste à récupérer le contenu (tout ou partiel) d'un projet géré en configuration afin de le ramener sur le poste client. C'est une étape nécessaire avant d'envisager des modifications sur les sources. Le *check-in* est le pendant de cette action, permettant de remonter vos modifications vers le serveur de sources.

Utilisation des tâches CVS depuis un script Ant

L'exécution de ce build-file présuppose la définition d'une variable d'environnement CVSROOT. Sur une machine Unix utilisant un `shell bash`, ceci peut être fait via un export CVSROOT=... La tâche Ant permet aussi de définir cette variable via le positionnement de l'attribut `cvsRoot`. Notre exemple réalisant un check-out, qui est la commande par défaut pour cette tâche, il n'y a pas besoin de spécifier une commande via l'attribut `command`. Ce build-file suppose que le code source est stocké dans un module CVS dénommé `Signets` : ceci est l'information spécifiée par l'attribut `package`.

```
<project name="BlueWeb" default="main" basedir=".">
  <!-- initialisation -->
  <target name="init" description ="crée les répertoires utilisés par
   la suite">
     <mkdir dir="CVS-out"/>
  </target>
<!-- supprime les fichiers créés -->
  <target name="clean" description="prépare le grand ménage...">
    <delete dir="CVS-out"/>
  </target>
<!-- target principale (par défaut) -->
<target name="main" depends="init" description="réalise un check-out
 CVS des sources d'un projet">
<cvs dest="CVS-out" package="Signets" />
</target>
</project>
```

OUTIL **L'auxiliaire idéal, CruiseControl**

Discipliné, répétitif jusqu'à l'épuisement, rigoureux, réfléchi (mais pas trop), obstiné, communiquant, adaptatif... (si vous pensez à la description de l'employé modèle, vous vous trompez d'ouvrage, ce chapitre n'est pas extrait des aventures de Dilbert) ; il s'agit bien sûr de CruiseControl.

Notre responsable qualité a repéré cet outil sur Internet (http://cruisecontrol.sourceforge.net). En application du concept de *continuous build*, CruiseControl prend sur lui de vérifier périodiquement si des modifications ont été apportées dans le gestionnaire de configuration. Lorsque c'est le cas, il déclenche un processus de build dont il surveillera la réussite... ou l'échec.

Dans la pratique, CruiseControl est une application Java qui prend en charge la surveillance d'un projet. Elle se décompose ainsi :

- `ModificationSet` : ces objets ont pour objectif de repérer si des modifications ont été apportées aux sources du projet. Plusieurs implémentations sont proposées par défaut pour vérifier l'état des gestionnaires de configuration les plus répandus (en particulier CVS). Michel choisit évidemment d'utiliser un `CVSModificationSet`, qu'il configure pour qu'il scrute le répertoire CVS du projet. Il paramètre une « période de silence » de 5 minutes. Cette période permet de ne pas déclencher une procédure de build entre deux *commits* d'un développeur.
- `Builder` : lorsque CruiseControl détecte au moins une modification et qu'elle est suivie des quelques minutes d'inactivité salvatrices, il déclenche une procédure de build. C'est-à-dire que, pendant une phase de *commit* des sources (au cours de laquelle de nombreux fichiers seront déposés après modification), cette procédure ne sera pas débutée. CruiseControl gère le déclenchement d'une cible Ant. Michel configure un AntBuilder pour qu'il invoque une cible dédiée incluse dans le `build.xml` du projet. Cette cible met à jour le répertoire local utilisé par CruiseControl, supprime les fichiers temporaires et reconstruit le projet entièrement (et plus encore, voir plus bas).
- `Scheduler` : le `Scheduler` de CruiseControl permet de configurer le temps d'attente entre deux builds consécutifs. Michel choisit de laisser 5 minutes de répit à CruiseControl.
- `Publishers` : pour que le travail de CruiseControl ne reste pas sans voix, un projet se voit pourvu de `publishers`. Ceux-ci ont en charge de rapporter au reste du monde le succès ou l'échec du dernier build. Michel choisit de configurer un `HTMLEmailPublisher`. Celui-ci commence par créer un rapport HTML du log d'activité du build. Puis (tel que l'a configuré Michel), il le transmet a tous les développeurs dont les modifications ont été prises en compte par ce build. En cas d'échec, ce rapport est envoyé à la liste de diffusion regroupant toute l'équipe de développement. Michel fait en sorte d'obtenir à chaque fois ce rapport pour suivre l'activité de CruiseControl.
- `Labels` : afin de clarifier le travail de CruiseControl, celui-ci associe à chaque tentative fructueuse un label. Il s'agit par défaut d'un numéro de build : `<prefix>-<nnn>`. CruiseControl fournit cette information sous la forme d'une propriété au script Ant. Cela permet au concepteur du script d'identifier la version en cours de construction. Si le build est réussi, le numéro est incrémenté.

Voilà un scénario, peut-être futuriste de prime abord, mais qui est la meilleure façon d'intégrer le concept d'intégration perpétuelle cher à Martin Fowler. Il ne faut pas compter sur l'intervention de l'homme, il faut des outils, fournissant des rapports, permettant de signifier si oui ou non les modifications ont été fructueuses.

Vers un build-file réaliste : rassemblons les morceaux…

Si Michel spécifie clairement ses besoins, nous sommes à ce moment prêts à coder le build-file dédié à la qualité. On pourrait imaginer le scénario suivant :

1. Extraire les sources de CVS (check-out).
2. Appliquer Jalopy avec la convention de nommage préparée par Michel.
3. Appliquer Checkstyle pour détecter les failles par rapport à la convention de nommage (plus importantes qu'un mauvais nombre d'espaces insérés pour l'indentation et autres peccadilles). Produire un rapport puis l'envoyer par courrier électronique.
4. Appliquer JDepend de manière à déceler d'éventuelles failles dans la conception. Il est bon que le chef de projet soit dans la liste des destinataires du courrier contenant le rapport établi.
5. Éventuellement réaliser un check-in dans CVS permettant d'avoir un code « propre ».

Suivant la formule consacrée, on laissera au lecteur le soin de travailler sur un `makefile` allant dans ce sens.

En résumé…

Ce chapitre souhaite montrer comment avec un petit peu de temps, quelques outils bien choisis, un chef de projet ou un responsable qualité peut gagner en productivité et en maîtrise sur son projet. Bien entendu, les solutions proposées ici ne sont pas parfaites et ne couvrent pas tous les besoins, mais après une analyse de vos besoins spécifiques et une phase de recherche vous devriez être à même d'intégrer de nouveaux outils dans votre processus de build.

chapitre 8

Classe de test cliente accédant aux objets distribués (EJB)

Cette classe est placée dans le paquetage `client`.

```
package client;
import java.util.Properties;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import ejb.GestionnaireSignets;
import ejb.GestionnaireSignetsHome;
```

La classe de test cliente se connecte à un serveur JBoss configuré avec les valeurs par défaut (1099 pour le serveur JNDI) et on suppose que ce serveur est sur la même machine.

```
/**
* classe de test cliente.
*/
public class TestClient
{
    /**
    * La classique méthode main(). Fait tout le travail.
    */
    public static void main(String[] args)
    {
```

La méthode `main ()` obtient la référence sur l'EJB session, appelle les méthodes métier puis affiche les résultats.

On crée en mémoire des propriétés permettant d'assurer la connexion au serveur JBoss. Ici ce serveur est supposé être local.
Le travail requis pour lire un fichier de propriétés est quasiment nul…C'est une solution que l'on ne saurait trop conseiller.

```
       Properties env = new Properties();
       env.setProperty("java.naming.factory.initial",
                "org.jnp.interfaces.NamingContextFactory");
       env.setProperty("java.naming.provider.url", "localhost:1099");
       env.setProperty("java.naming.factory.url.pkgs",
           "org.jboss.naming");
       try
       {
          InitialContext jndiContext = new InitialContext(env);
          Object ref = jndiContext.lookup("GestionnaireSignets");
```

Implémentation de la logique métier BlueWeb avec XDoclet

Après avoir présenté les différentes couches de notre application maquette, il est temps pour le lecteur de rassembler les morceaux. Nous allons aborder l'utilisation de JBoss d'une manière plus détaillée et conclure sur le tour d'horizon des possibilités offertes par le logiciel libre dans le cadre d'une architecture à 5 couches.

SOMMAIRE

MOTS-CLÉS

- EJB
- CMR
- Clé primaire
- Performance
- Intégration des outils Open Source
- JMX
- JBoss
- PostgreSQL

Logique métier de l'application BlueWeb

L'application maquette choisie par BlueWeb s'avère être très simple d'un point de vue logique métier : quelques règles seulement permettent l'ajout d'un thème ou d'un signet, il n'y a pas de calculs et le processus de validation des données est uniquement basé sur des expressions régulières. En fait, la logique est celle de la gestion d'un arbre de données (qui est d'ailleurs la structure choisie pour l'affichage de la partie cliente).

Rappelons les principales règles :

- Un signet est attaché à un et un seul thème.
- Un thème peut contenir des sous-thèmes.
- Un thème peut contenir de 0 à n signets.

Cette section s'intéresse au code source requis pour les EJB assurant cette fonction de gestion des données (signets et thèmes). Le code source présenté ci-après adopte une position classique, dans laquelle la gestion des relations père-fils induites par notre logique applicative est assurée par l'application elle-même. C'était la seule solution jusqu'à l'apparition des spécifications EJB 2.0. Elle assure donc une portabilité maximale par rapport au marché des serveurs d'applications. Néanmoins, elle présente le défaut de laisser à notre charge un certain nombre de tâches qui pourraient revenir au conteneur EJB ; parmi elles, on ne peut manquer de citer le cas de la destruction en cascade de données (la destruction d'un thème doit induire la destruction de toutes les données qui lui sont rattachées). Une alternative à cette position serait d'utiliser ce que la spécification 2.0 des EJB appelle CMR (voir ci-contre).

B.A.-BA CMR (Container Managed Relationship)

La gestion des relations par le conteneur (*Container Managed Relationship*) est une nouveauté de la norme EJB 2.1 permettant de laisser le soin au conteneur de gérer les relations entre objets (entités de la base), et ce quelle que soit la cardinalité de la relation (1-1, 1-n, etc.). Cette fonctionnalité n'a d'existence standard que par l'ajout par rapport aux versions précédentes des spécifications d'une nouvelle section dans le fichier `ejb-jar.xml` permettant de définir les relations entre EJB entités.

Code de l'EJB session (sans état)

Examinons de ce pas le composant `Session` (EJB session sans état) qui sert de chef d'orchestre pour nos deux objets entités, à savoir `Thème` et `Signet`. C'est par cet objet que vont transiter toutes les requêtes clientes, qu'il s'agisse d'une création d'un thème ou d'un signet, de la mise à jour ou bien encore d'une recherche.

Le code proposé ci-dessous utilise des tags XDoclet (dans sa version 1.2.2). Ainsi, nous nous contenterons de présenter le code de la classe d'implémentation de notre EJB, laissant à XDoclet le soin de créer les parties redondantes de code imposées par la norme EJB (`Home` et `Remote interface`…).

VOCABULAIRE Manager

L'examen de code source, lié au monde EJB ou non, vous donnera souvent l'occasion de rencontrer des classes intitulées : `<XXXX>Manager`. Ces classes, qui traduites en français deviendraient des `Gestionnaires <XXXX>`, assurent un rôle de chef d'orchestre et pilotent le comportement de plusieurs classes. Elles permettent de vous abstraire des détails d'implémentation de bibliothèques dont vous n'avez pas à comprendre le fonctionnement. Elles s'inscrivent dans la même logique que le motif de conception `Façade` présenté dans un chapitre précédent. Notre composant session pilotant nos composants entités va ainsi devenir un `Manager`.

RAPPEL EJB

Le chapitre 5 comporte des éléments d'introduction à la terminologie employée dans le monde des EJB, donc n'hésitez pas à y retourner pour vous rafraîchir la mémoire.

Code source de notre composant de session enrichi de tags XDoclet

```
package ejb;
import java.rmi.RemoteException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Collection;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.CreateException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import ejb.ThemeHome;
import ejb.Theme;
/**
* @ejb.bean name="GestionnaireSignets" type="Stateless"
* jndi-name="GestionnaireSignets"
* @ejb.ejb-ref ejb-name="SignetBean" view-type="local"
* @ejb.ejb-ref ejb-name="ThemeBean" view-type="local"
*
*/
public class GestionnaireSignetsBean implements SessionBean{

   public void ejbActivate() throws EJBException, RemoteException {
   }
   public void ejbPassivate() throws EJBException, RemoteException {
   }
   public void ejbRemove() throws EJBException, RemoteException {
   }
   public void setSessionContext(SessionContext arg0)
      throws EJBException, RemoteException {
   }
   public void ejbCreate() {
   }
   /**
    * Cette méthode fait partie de l'interface déclarée de notre EJB
    * @ejb.interface-method view-type="remote"
    * @ejb.transaction type="Required"
    */
   public String[] getThemes() {
      System.err.println("getThemes");
      ArrayList themes_list = new ArrayList(50);
      try{
         Context ctx = new InitialContext();
         Object ref = ctx.lookup("java:comp/env/ejb/ThemeBean");
         ThemeHome home = (ThemeHome)

                  PortableRemoteObject.narrow(ref,ThemeHome.class);
         Collection all_themes = home.findAll();
         for(Iterator iter = all_themes.iterator();iter.hasNext();){
            Theme curr_theme = (Theme) iter.next();
            themes_list.add(curr_theme.getName());
         }
      }
```

Dans ce paquetage `ejb`, nous définissons un EJB session sans état. Classe gérant la collection de signets...
Bean session assurant la façade de notre logique métier avec l'extérieur.

Balises utilisées par XDoclet pour guider la génération du code.

Avec le tag XDoclet `ejb.ejb-ref`, on déclare que cet EJB va en manipuler deux autres (les entités `Theme` et `Signet`), via des références locales (sans passer par la sérialisation Java, mais par des pointeurs).

Cette première partie comporte les méthodes qui doivent être implémentées pour respecter l'interface `Session`, à savoir :
`ejbActivate()`
`ejbPassivate()`
`ejbRemove()`
etc.
Ces méthodes sont principalement liées au cycle de vie d'un objet au sein du conteneur.

Ici sont rassemblées les méthodes constituant l'interface exposée par l'EJB. Les méthodes métier en quelque sorte.
L'ajout d'un thème implique d'obtenir la `Home interface` de l'EJB `Theme` puis créer un nouvel objet `Theme` en lui passant les paramètres convenables.

Pour cela, il faut donc utiliser un objet de la classe `Context` (paquetage `javax.naming`) pour procéder à la recherche (`lookup()`).

Puis, il faut utiliser la méthode `narrow()` de l'objet `PortableRemoteObject` afin d'obtenir une référence valide.

Code source de notre composant de session enrichi de tags XDoclet (suite)

```
    catch(Exception e){
      e.printStackTrace();
    }
    return (String[]) themes_list.toArray(
              new String[themes_list.size()]);
  }
  /**
   * @ejb.interface-method view-type="remote"
   * @ejb.transaction type="Required"
   */
```

Ajoute un thème en le rattachant au parent spécifié...

```
  public void addTheme(int id,String name,String remark,
                       int parentId){
```

Ce bloc montre comment obtenir une référence sur l'interface Theme, puis comment utiliser un service (ici le finder `findAll()`) (1).

```
    Context ctx =null;
    try{
     ctx = new InitialContext();
      Object ref = ctx.lookup("java:comp/env/ejb/ThemeBean");
      ThemeHome home = (ThemeHome) PortableRemoteObject.narrow(
                ref,ThemeHome.class);
      Theme theme = home.create(new Integer(id),name,remark,
                                new Integer(parentId));
    }
    catch(Exception e){
      e.printStackTrace();
    }
  } // addTheme()
  /**
   * @ejb.interface-method view-type="remote"
   * @ejb.transaction type="Required"
   */
```

L'ajout d'un signet sous un thème implique d'obtenir la Home interface de l'EJB `Signet`, puis de créer un nouvel objet `Signet` en lui passant les paramètres convenables.

```
  public void addSignetUnderTheme(int signetId,int parentId,
              String name,String remark, String address){
    System.out.println("J'ajoute le signet = " + name +
              " sous le parent = " + parentId);
```

Pour cela il faut donc utiliser un objet de la classe `Context` (paquetage `javax.naming`) pour procéder à la recherche (`lookup()`).
Puis, il faut utiliser la méthode `narrow()` de l'objet `PortableRemoteObject` afin d'obtenir une référence valide.
On pourrait très facilement imaginer différents contrôles tels que la vérification de l'existence du thème ou encore la conformité de l'URL du signet, et même éventuellement ajouter un test permettant d'éviter les doublons…

```
    Context ctx =null;
    try{
     ctx = new InitialContext();
      Object ref = ctx.lookup("java:comp/env/ejb/SignetBean");
      SignetHome home = (SignetHome) PortableRemoteObject.narrow(
                          ref,SignetHome.class);
      Signet theme = home.create(new Integer(signetId),name,
                          new Integer(parentId),address,remark);
    }
    catch(Exception e){
      e.printStackTrace();
    }
  } //addSignetUnderTheme()
  /**
   * enlève tous les signets et thèmes...
   * Cette méthode fait partie de l'interface déclarée de notre EJB
   * @ejb.interface-method view-type="remote"
   * @ejb.transaction type="Required"
   */
```

Code source de notre composant de session enrichi de tags XDoclet (suite)

```
  public void removeAll(){
    System.out.println("Commence le grand ménage en base... ");
    Context ctx =null;
    try{
     ctx = new InitialContext();
      Object ref = ctx.lookup("java:comp/env/ejb/SignetBean");
      SignetHome home = (SignetHome) PortableRemoteObject.narrow(
                                    ref,SignetHome.class);
      Collection all_signets = home.findAll();
      for(Iterator iter=all_signets.iterator();
          iter.hasNext();){
          Signet current_obj = (Signet) iter.next();
          System.out.println("suppression du signet = " +
                    current_obj.getName() );
          current_obj.remove();
      }
      ref = ctx.lookup("java:comp/env/ejb/ThemeBean");
     ThemeHome theme_home = (ThemeHome) PortableRemoteObject.narrow(
                                      ref,ThemeHome.class);
      Collection all_themes = theme_home.findAll();
      for(Iterator iter=all_themes.iterator();iter.hasNext();){
          Theme current_obj = (Theme) iter.next();
          System.out.println("suppression du theme = " +
                    current_obj.getName() );
          current_obj.remove();
      }
      System.out.println("Boulot fini !!");
    }
    catch(Exception e){
       e.printStackTrace();
    }
  } // removeAll()
  /**
   * Liste les thèmes fils du thème specifie...
   * @param parentId, id du thème parent...
   * @return Integer[], tableau contenant les ID des thèmes trouvés
   * @ejb.interface-method view-type="remote"
   * @ejb.transaction type="Required"
   */
  public Integer[] listThemesUnder(int parentId){
     System.out.println("liste des themes attaches au parent = " +
               parentId);
     Context ctx =null;
     try{
        ctx = new InitialContext();
        Object ref = ctx.lookup("java:comp/env/ejb/ThemeBean");
        ThemeHome theme_home = (ThemeHome) PortableRemoteObject.narrow(
               ref,ThemeHome.class);
       Collection all_themes = theme_home.findByThemeParent(
               new Integer(parentId));
       ArrayList list_fetched = new ArrayList();
```

La suppression d'un thème implique d'obtenir la Home `interface` de l'EJB Theme, puis d'appeler la méthode `findAll` renvoyant tous les thèmes disponibles. En itérant sur la collection d'objets retournée, on applique sur chaque objet la méthode `remove()`. On procède de la même façon pour la suppression des signets.

La liste des thèmes rattachés à un thème parent se fait très simplement en appelant le finder déclaré dans le bean ThemeBean : `findByThemeParent()`. En itérant sur la collection d'objets ramenés, on peut ensuite bâtir le tableau d'objets devant être retourné.

Code source de notre composant de session enrichi de tags XDoclet (suite)

```
        System.out.println("J'ia trouve = " + all_themes.size() +
            " themes fils...");
        for(Iterator iter=all_themes.iterator();iter.hasNext();){
            Theme current_obj = (Theme) iter.next();
            list_fetched.add(current_obj.getId());
        }
        return (Integer[])(list_fetched.toArray(
                        new Integer[list_fetched.size()]));
      }
      catch(Exception e){
        e.printStackTrace();
      }
      return null;
    } // listThemesUnder()

    /**
     * @param parentId, id du thème parent...
     * @return Integer[], tableau contenant les ID des signets trouvés
     * @ejb.interface-method view-type="remote"
     * @ejb.transaction type="Required"
     */
    public Integer[] listSignetsUnder(int parentId){
      System.out.println("liste des signets rattachés au theme id = " +
          parentId);
      Context ctx =null;
      try{
      ctx = new InitialContext();
      Object ref = ctx.lookup("java:comp/env/ejb/SignetBean");
      SignetHome signet_home = (SignetHome)
                PortableRemoteObject.narrow( ref,SignetHome.class);
      Collection all_signets = signet_home.findByThemeParent(
                new Integer(parentId));
      ArrayList list_fetched = new ArrayList();
      System.out.println("Nombre de signets trouvés = " +
                all_signets.size() );
     for(Iterator iter=all_themes.iterator();iter.hasNext();){
           Theme current_obj = (Theme) iter.next();
           list_fetched.add(current_obj.getId());
      }
      return (Integer[])(list_fetched.toArray(
                          new Integer[list_fetched.size()]));
       }
       catch(Exception e){
          e.printStackTrace();
       }
       return null;
    } // listSignetsUnder()
}
```

Liste les signets attachés au thème spécifié...

Pour obtenir la liste des signets rattachés à un parent donné, il suffit de chercher grâce au finder dédié (`findBy ThemeParent()`), puis de parcourir la liste trouvée de manière à construire la valeur de retour.

Ce code n'a rien d'optimisé (volontairement) dans le sens où, à chaque appel (de chacune des méthodes utilisées par les clients du type `addTheme()` ou `listSignetsUnder()`, on est condamné à refaire les mêmes recherches (`lookup`) par JNDI sur les objets (interfaces `Home` des objets `Theme` et `Signet`).

Ceci est réellement pénalisant en termes de performances et donc indigne d'une version de production. Néanmoins, l'intérêt pédagogique existe, puisque cela montre combien la mécanique générale est répétitive et donc à quel point le code métier est réduit. Il ne s'agit en fait que d'appels sur la bonne méthode de recherche et d'une série de traitements (mise en forme, création ou suppression). On peut préciser que l'on indiquera ultérieurement un moyen permettant de faciliter la recherche des objets de type `Home interfaces` (voir renvoi 1 dans le code, page 176).

Enfin, comme précisé dans les commentaires du listing précédent, le code proposé ici est réellement minimaliste (peut-être trop) dans le sens où un certain nombre de contrôles (typiquement pour l'insertion d'un nouveau signet) ne sont pas faits. Il ne tiendra qu'au lecteur d'ajouter autant de contrôles qu'il le désire.

De même, la gestion des exceptions présentée ici ne correspond pas aux attentes d'une application professionnelle, mais à ce stade vous avez tous les éléments en votre possession…

PERFORMANCE

Recherche des Home Interfaces

Très logiquement, il est généralement trivial d'éviter de refaire plusieurs fois le même traitement et d'essayer de factoriser le code correspondant. Mais dans ce contexte des EJB, la recherche (`lookup`) JNDI étant une opération coûteuse, il est extrêmement important de ne pas multiplier les requêtes inutiles. L'ordre de grandeur du ratio de la durée d'une telle opération relativement à la durée des différentes opérations (simples dans notre cas) du type ajout ou suppression de thèmes ou signets est de 1. Donc, en supprimant ces recherches coûteuses de vos méthodes métier, vous pouvez diviser par deux le temps de réponse de chacune de ces opérations. Attention, ce n'est qu'un ordre de grandeur.

Code des entités (EJB entity type CMP)

Les deux classes suivantes utilisent elles aussi XDoclet pour guider la création et le déploiement de vos composants. Les listings seront donc limités au code des classes d'implémentation. La balise `ejb.bean` avec `type="CMP"` définira que les entités seront contrôlées par le conteneur.

Code source de la classe Signet enrichi de tags XDoclet

```
package ejb;
import java.rmi.RemoteException;
import javax.ejb.EJBException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.RemoveException;
import javax.ejb.CreateException;
/**
* Classe reflétant un signet dans la BDD.
* Entity bean de type CMP (géré par le conteneur)
* @author jm
* @ejb.bean
*    type="CMP"
*    cmp-version="2.x"
*    name="SignetBean"
*    schema="Signet"
*    local-jndi-name="SignetBean"
*    view-type="local"
```

Tags génériques (non spécifiques à un serveur d'applications). Ici on définit une entité de type CMP, en utilisant le générateur conforme à la version 2.0 des spécifications. On définit le type de vue (ici locale) et les noms (JNDI) ainsi que le type de clé primaire utilisé dans la table correspondante.

B.A.-BA **Clé primaire**

Une clé primaire est un champ (ou une concaténation/agrégat/ensemble de champs) d'une table dont la valeur doit être unique au sein de cette table. Elle permet donc une identification à coup sûr d'une entité. Par exemple, le numéro de sécurité social à 13 chiffres permet d'identifier de manière unique un individu en France. Ce champ revêt donc une importance particulière dans la déclaration d'un EJB entité.

Code source de la classe Signet enrichi de tags XDoclet (suite)

```
*    primkey-field="id"
*
*
* @ejb.pk                  class="java.lang.Integer"
*
* @ejb.home generate="local" local-class="ejb.SignetHome"
* @ejb.interface generate="local" local-class="ejb.Signet"
*
* @ejb.finder
*    signature="Signet findByName(java.lang.String name)"
*    unchecked="true"
*   query="SELECT OBJECT(signet) FROM Signet signet where signet.name
= ?1"
*    result-type-mapping="Local"
*
* @ejb.finder
*    signature="Collection findByThemeParent(java.lang.Integer
*           ➥ parentId)"
*    unchecked="true"
*    query="SELECT OBJECT(signet) FROM Signet signet
*       ➥  where signet.parent = ?1"
*    result-type-mapping="Local"
* @ejb.finder
*    signature="Collection findAll()"
*    unchecked="true"
*    query="SELECT OBJECT(signet) FROM Signet signet"
*    result-type-mapping="Local"
*
* @ejb.persistence
*      table-name="signet_tbl"
*
* @jboss.persistence
*      datasource="PostgresDS"
*      datasource-mapping="PostgreSQL"
*
*/
public abstract class SignetBean implements EntityBean {


  /**
    *
    * @ejb.create-method
    */
    public Integer ejbCreate(Integer id, String name,Integer parentId,
           String address, String remark)
      throws CreateException {
        setId(id);
        setName(name);
        setParent(parentId);
        setUrl(address);
        setRemark(remark);
        return null;
    }
```

Ici commence la déclaration des finders, les méthodes permettant de trouver une ou plusieurs entités au sein de la table. Dans ce cas, on définit un finder permettant la recherche par nom de signet, par identifiant de parent et le traditionnel `findAll()` ramenant toutes les entités de la table. Il est à noter que la syntaxe utilisée dans les finder est celle définie par l'EJB-QL (voir spécifications EJB).

On définit le nom de la table en base représentant notre entité.

Une série de tags spécifiques à JBoss permet de préciser le nom de la `DataSource` à utiliser ainsi que le type de mapping objet/relationnel. Dans ce cas, on utilise une source de données dont le nom JNDI est `PostgresDS` et dont le type est (au sens de la couche de mapping objet/relationnel de JBOSS) `PostgreSQL`.

Cette méthode est appelée par le conteneur lorsqu'un client fait : `monEJB.create(…)`. Le conteneur va chercher automatiquement une méthode `ejbPostCreate` attendant la même liste de paramètres.

Code source de la classe Signet enrichi de tags XDoclet (suite)

```
public void ejbPostCreate(Integer id,String name,
    Integer parentId,String address,String remark)
  throws CreateException{ }
/**
  * @ejb.pk-field
  * @ejb.persistence
  *      column-name="id"
  * @ejb.interface-method
  * @ejb.transaction
  *      type="Supports"
  */
 public abstract Integer getId( );
 public abstract void setId( Integer userId );
 /**
 * @ejb.interface-method view-type="local"
  * @ejb.persistence
  *      column-name="name"
  *      jdbc-type="VARCHAR"
  *      sql-type="varchar(50)"
  */
 public abstract String getName( );
 public abstract void setName( String name );
 /**
 * @ejb.interface-method view-type="local"
  * @ejb.persistence
  *      column-name="remark"
  *      jdbc-type="VARCHAR"
  *      sql-type="varchar(250)"
  */
 public abstract String getRemark( );
 public abstract void setRemark( String remark );
 /**
 * @ejb.interface-method view-type="local"
  * @ejb.persistence
  *      column-name="url"
  *      jdbc-type="VARCHAR"
  *      sql-type="varchar(255)"
  */
 public abstract String getUrl( );
 public abstract void setUrl( String url );
 /**
 * @ejb.interface-method view-type="local"
  * @ejb.persistence
  *      column-name="parent"
  *      jdbc-type="Integer"
  *      sql-type="NUMBER(10)"
  */
public abstract Integer getParent();
public abstract void setParent(Integer id);
public void ejbActivate() throws EJBException, RemoteException {
}
```

On définit le champ jouant le rôle de clé primaire. Comme pour les autres champs, la déclaration est très simple : nom de la colonne, type JDBC et type SQL.

Code source de la classe Signet enrichi de tags XDoclet (suite)

```
   public void ejbLoad() throws EJBException, RemoteException {
   }
   public void ejbPassivate() throws EJBException, RemoteException {
   }
   public void ejbRemove()
      throws RemoveException, EJBException, RemoteException {
   }
   public void ejbStore() throws EJBException, RemoteException {
   }
   public void setEntityContext(EntityContext arg0)
      throws EJBException, RemoteException {
   }
   public void unsetEntityContext() throws EJBException,
RemoteException {
   }
}
```

Voici le code décrivant comment est vu notre objet `Theme` au sein de notre architecture :

Code source de la classe Theme enrichi de tags XDoclet

```
package ejb;
import java.rmi.RemoteException;
import javax.ejb.EJBException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.RemoveException;
import javax.ejb.CreateException;
   public abstract String getName( );
   public abstract void setName( String name );
   /**
   * @ejb.interface-method view-type="local"
   * @ejb.persistence
   *      column-name="remark"
   *      jdbc-type="VARCHAR"
   *      sql-type="varchar(255)"
   */
   public abstract String getRemark( );
   public abstract void setRemark( String remark );
   /**
   * @ejb.interface-method view-type="local"
   * @ejb.persistence
   *      column-name="parent"
   *      jdbc-type="Integer"
   *      sql-type="NUMBER(10)"
   */
   public abstract Integer getParent();
   public abstract void setParent(Integer id);
   /* (non-Javadoc)
   * @see javax.ejb.EntityBean#ejbActivate()
   */
```

Code source de la classe Theme enrichi de tags XDoclet (suite)

```
   public void ejbActivate() throws EJBException, RemoteException {
   }
   /* (non-Javadoc)
    * @see javax.ejb.EntityBean#ejbLoad()
    */
   public void ejbLoad() throws EJBException, RemoteException {
   }
   /* (non-Javadoc)
    * @see javax.ejb.EntityBean#ejbPassivate()
   */
   public void ejbPassivate() throws EJBException, RemoteException {
   }
   /* (non-Javadoc)
    * @see javax.ejb.EntityBean#ejbRemove()
    */
   public void ejbRemove()
      throws RemoveException, EJBException, RemoteException {
   }
   /* (non-Javadoc)
    * @see javax.ejb.EntityBean#ejbStore()
    */
   public void ejbStore() throws EJBException, RemoteException {
   }
   /* (non-Javadoc)
   * @see
   *  javax.ejb.EntityBean#setEntityContext(javax.ejb.EntityContext)
   */
   public void setEntityContext(EntityContext arg0)
      throws EJBException, RemoteException {
   }
   /* (non-Javadoc)
    * @see javax.ejb.EntityBean#unsetEntityContext()
   */
   public void unsetEntityContext() throws EJBException,
      RemoteException {
      // TODO Auto-generated method stub
   }
}
/**
*
* @ejb.bean
*    type="CMP"
*    cmp-version="2.x"
*    name="ThemeBean"
*    schema="Theme"
*    local-jndi-name="ThemeBean"
*    view-type="local"
*    primkey-field="id"
*
* @ejb.pk                  class="java.lang.Integer"
*
* @ejb.home generate="local" local-class="ejb.ThemeHome"
* @ejb.interface generate="local" local-class="ejb.Theme"
```

Ces tags XDoclet couvrent des aspects généraux et des spécificités nécessaires au déploiement dans JBoss.

Code source de la classe Theme enrichi de tags XDoclet (suite)

```
* @ejb.finder
*   signature="Theme findByThemeName(java.lang.String name)"
*   unchecked="true"
*   query="SELECT OBJECT(theme) FROM Theme theme where theme.name = ?1"
*   result-type-mapping="Local"
*
* @ejb.finder
*   signature="Collection findByThemeParent(java.lang.Integer parentId)"
*   unchecked="true"
*   query="SELECT OBJECT(theme) FROM Theme theme where theme.parent = ?1"
*   result-type-mapping="Local"
* @ejb.finder
*   signature="Collection findAll()"
*   unchecked="true"
*   query="SELECT OBJECT(theme) FROM Theme theme"
*   result-type-mapping="Local"
*
* @ejb.persistence
*   table-name="theme_tbl"
* @jboss.persistence
*   create-table="true"
*   remove-table="true"
*   datasource="PostgresDS"
*   datasource-mapping="PostgresSQL"
*
*/
```

Classe reflétant un thème dans la BDD.
Bean entité de type CMP (géré par le conteneur).

```
public abstract class ThemeBean implements EntityBean {
  /**
    * @ejb.create-method
    */
    public Integer ejbCreate(Integer id, String name,String remark,
           Integer parentId)
      throws CreateException {
        setId(id);
        setName(name);
        setParent(parentId);
        setRemark(remark);
        return null;
    }
    public void ejbPostCreate(Integer id,String name,String remark,
                              Integer parentId)
       throws CreateException{ }
```

Clé primaire dans la base.

```
   /**
     * @ejb.pk-field
     * @ejb.persistence
     *      column-name="id"
     * @ejb.interface-method
     * @ejb.transaction
     *      type="Supports"
     */
```

Code source de la classe Theme enrichi de tags XDoclet (suite)

```
    public abstract Integer getId( );
    public abstract void setId( Integer userId );
   /**
    * @ejb.interface-method view-type="local"
    * @ejb.persistence
    *       column-name="name"
    *       jdbc-type="VARCHAR"
    *       sql-type="varchar(50)"
    */
```

Autre champ persistant

La déclaration d'entités par XDoclet est relativement simple : il suffit, pour chaque champ persistant (colonne de la base), de préciser le nom de cette colonne, le type des données manipulées (en Java puis en SQL). Évidemment, pour le champ représentant la clé primaire, il est indispensable de le préciser par le biais du tag `@ejb.pk-field`.

On peut aussi préciser qu'il est possible de créer une clé primaire composite, c'est-à-dire composée de différents champs. Dans ce cas, il sera indispensable de créer une classe `MonEntitePK` servant de clé primaire et d'avoir plusieurs tags `@ejb.pk-field`, un pour chaque champ intervenant dans la construction de la clé primaire. Là encore, XDoclet est votre ami et vous permet de créer automatiquement la classe permettant d'instancier vos clés primaires composites.

PERFORMANCE Clés primaires composites

Il est généralement souhaitable d'éviter l'utilisation de telles clés, qui induisent des temps de construction beaucoup plus importants que des clés simples. Bien entendu, cela ne signifie pas que vous ne devez jamais en utiliser.

Génération du code et déploiement de nos EJB dans JBoss

Maintenant que le code source pour cette partie métier est en place, il faut :

1. lancer l'appel de XDoclet (pour qu'il produise le code et les fichiers nécessaires au déploiement à notre place) ;
2. automatiser la création d'une archive (`.jar`) ;
3. déployer notre composant dans JBoss.

C'est ce que propose le script XML suivant (build-file Ant).

Script Ant réalisant l'invocation de XDoclet et le déploiement de nos EJB

```
<?xml version="1.0" ?>
<project name="ejb-blueweb" default="default" basedir=".">


    <property file="${user.home}/.${ant.project.name}-
              ➥ build.properties"/>
    <property file="${user.home}/.build.properties"/>
    <property file="build.properties"/>
```

Un projet Ant assurant de multiples fonctions : traitement des sources et utilisation de XDoclet, compilation des EJB, empaquetage, déploiement.

Cherche un fichier `build.properties` dans différents répertoires.

Script Ant réalisant l'invocation de XDoclet et le déploiement de nos EJB (suite)

Charge les variables système.

```
    <property environment="env"/>
    <property name="ant.home" location="${env.ANT_HOME}"/>
    <property name="jboss.home" location="${env.JBOSS_HOME}"/>
    <property name="env.COMPUTERNAME" value="${env.HOSTNAME}"/>
    <property file="common.properties"/>
    <property name="lib.dir" location="${env.LIB_HOME}"/>
    <property file="${lib.dir}/lib.properties"/>
    <property name="env.J2EE_HOME" location="${j2ee.lib.dir}"/>
    <property name="j2ee.home" location="${env.J2EE_HOME}"/>
    <property name="j2ee.jar" location="${j2ee.home}/lib/j2ee.jar"/>
    <path id="xdoclet.classpath">
        <pathelement location="${lib.dir}/log4j.jar}"/>
        <pathelement location="${lib.dir}/commons-logging.jar}"/>
        <pathelement location="${j2ee.jar}"/>
        <fileset dir="${xdoclet.dir}/lib/" includes="*.jar"/>
    </path>
<!-- EJB -->
    <path id="ejb.compile.classpath">
        <pathelement location="${j2ee.jar}"/>
    </path>
    <path id="ejb.test.classpath">
      <path refid="ejb.compile.classpath"/>
    </path>
<!-- Taskdef -->
```

Définit la tâche XDoclet en lui associant un nom de classe Java et un classpath permettant de charger cette classe.

```
    <taskdef name="xdoclet"
             classname="xdoclet.DocletTask"
             classpathref="xdoclet.classpath"
    />
    <patternset id="java.files.pattern" includes="**/*.java"/>
```

Initialisation de la structure de répertoire.

```
    <target name="init">
        <available property="j2ee.jar.present" file="${j2ee.jar}"/>
        <fail unless="j2ee.jar.present">
           j2ee.jar (${j2ee.jar}) is not present.
        </fail>
        <tstamp/>
        <mkdir dir="${build.dir}"/>
        <mkdir dir="${dist.dir}"/>
        <mkdir dir="${test.dir}"/>
        <echoproperties/>
    </target>
```

Pour être sûr de partir sur de bonnes bases, tous les fichiers générés sont supprimés.

```
    <target name="clean" description="Removes build artifacts">
        <delete dir="${build.dir}"/>
        <delete dir="${dist.dir}"/>
    </target>
```

Lance la génération des EJB par XDoclet.

```
    <target name="ejbdoclet" depends="init"
            description="Generate EJB code and descriptors">
        <taskdef name="ejbdoclet"
                 classname="xdoclet.modules.ejb.EjbDocletTask"
                 classpathref="xdoclet.classpath"
        />
        <mkdir dir="${build.dir}/ejb/gen"/>
```

Script Ant réalisant l'invocation de XDoclet et le déploiement de nos EJB (suite)

```
        <ejbdoclet destdir="${build.dir}/ejb/gen"
                   addedtags="@xdoclet-generated at ${TODAY}"
                   ejbspec="2.0"
                   force="true"
                   mergedir="metadata/ejb">
            <fileset dir="${src.dir}">
               <include name="ejb/*Bean.java" />
            </fileset>
            <remoteinterface/>
            <homeinterface/>
            <localinterface/>
            <localhomeinterface />
            <utilobject/>
            <jboss validatexml="false" destdir="${build.dir}"
                   datasource="java:/PostgresDS"
                   datasourcemapping="PostgreSQL"/>
            <deploymentdescriptor validatexml="true"/>
        </ejbdoclet>
    </target>

    <target name="compile-ejb" depends="ejbdoclet">
        <mkdir dir="${build.dir}/ejb/classes"/>
        <javac destdir="${build.dir}/ejb/classes/" debug="true"
               deprecation="true">
            <src location="${build.dir}/ejb/gen" />
            <src location="${src.dir}" />
        </javac>
    </target>

    <target name="package-ejb" depends="compile-ejb"
            description="Package EJB JAR">
        <jar destfile="${dist.dir}/antbook-ejb.jar">
            <fileset dir="${build.dir}/ejb/classes"/>
            <metainf dir="${build.dir}/ejb/gen" includes="*.xml"/>
            <metainf dir="${build.dir}" includes="jboss.xml"/>
        </jar>
    </target>
<!-- JBoss -->
   <target name="undeploy-all-jboss">
        <delete>
          <fileset dir="${jboss.home}/server/default/deploy"
                   includes="antbook-*.ear"
          />
        </delete>
    </target>

   <target name="deploy-jboss" depends="package-ejb"
           description="Deploy to local JBoss">
        <copy file="${dist.dir}/antbook-ejb.jar"
              todir="${jboss.home}/server/default/deploy"
        />
    </target>

    <target name="default" depends="deploy-jboss"/>
</project>
```

Compilation des EJB générés précédemment.

Packaging : création du fichier archive contenant nos EJB.

Le « dé-déploiement » dans JBoss est aisé. Il suffit de supprimer notre fichier `jar` du répertoire de déploiement (`server/default/deploy`) pour annuler le déploiement des EJB dans le conteneur.

Le déploiement est tout aussi aisé. Il suffit de copier notre fichier `jar` dans le répertoire de déploiement (`server/default/deploy`).

Une cible ne faisant qu'appeler la cible `deploy-jboss` puisque dépendante de celle-ci.

L'exécution de ce script implique de fournir un fichier de propriétés `build.properties`. Nous donnons ici en guise d'exemple celui présent sur une Linux Red Hat 9 et il vous appartient d'adapter son contenu en fonction de votre installation et de vos conventions de nommage des répertoires. Ici, elle est proche de celle utilisée pour le projet Jakarta.

Fichier de propriétés build.properties à fournir

```
xdoclet.dir=/dvpt/Java/lib/xdoclet-new
build.dir= build
j2ee.lib.dir=/dvpt/Java
dist.dir=dist
test.dir=testing
jboss.home=/home/jerome/jboss
src.dir=src
```

Avant de pouvoir tester nos EJB, il ne reste plus qu'à paramétrer la base de données à utiliser comme source de données par notre serveur d'applications.

ASTUCE Choix d'une base de données

Pour appréhender la technologie des EJB, il est fortement recommandé de prendre le temps d'installer un gestionnaire de bases de données, car faire l'impasse sur les composants entités revient à se priver d'une des facettes les plus intéressantes des EJB.
Le choix de la base de données est réellement très ouvert en sachant que Hypersonic SQL livrée avec JBoss est une base très limitée et que l'utilisation d'Access de Microsoft (base très populaire sous Windows) est problématique et fortement déconseillée en Java. Suivant votre système d'exploitation, vos compétences, la puissance de votre machine, l'utilisation de MySQL, PostgreSQL ou Interbase sont des choix raisonnables. Pour ceux disposant d'une machine professionnelle et de licences, ne manquez pas l'occasion de tester Oracle ou DB2 d'IBM.

Configuration de PostgreSQL dans JBoss

Cette section va détailler la procédure d'installation de PostgreSQL dans JBoss. Néanmoins, l'utilisation d'une autre base de données (la très populaire MySQL ou bien Interbase de Borland, sans parler de l'indétrônable Oracle) ne pose aucune difficulté particulière : il vous suffira d'adopter la même démarche et de modifier certains paramètres.

La configuration `default` comporte différents sous-dossiers dont deux importants à nos yeux :

- `lib` : dossier regroupant les différentes bibliothèques manipulées par cette configuration. C'est ici que vous devrez placer le `jar` (ou le `.zip` dans le cas d'Oracle) contenant le *driver* de votre gestionnaire de bases de données.
- `deploy` : dossier contenant à la fois vos applications déployées dans cette configuration et les services offerts par cette configuration.

B.A.-BA Driver JDBC

En Java, on accède à toute base de données par l'API JDBC en utilisant TCP/IP comme protocole de dialogue. L'utilisation d'une base de données implique donc nécessairement l'utilisation d'un pilote. Ceux-ci sont classifiés en diverses catégories ; on se contentera de conseiller l'utilisation de pilotes JDBC de type 4. Internet regorge de ressources utiles vous permettant de comprendre cette classification et ce conseil est donné de manière péremptoire. Il faut aussi préciser que la connexion à une base de données se fait par une URL du type :

```
jdbc:<dependant de la base>://
<nom_machine>:<port>:
<autres informations>
```

JBOSS **Configuration**

JBoss permet de choisir entre différentes configurations dès le démarrage. Configuration est ici utilisé dans le sens de l'association entre un nom (`minimal`, `default` ou `all` pour les configurations standards) et des services (intégration d'un *servlet engine*, qu'il s'agisse de Jetty ou Tomcat, d'un gestionnaire de files de messages comme JBOSSMQ, prise en charge du *clustering*, etc.).
Pour une utilisation basique, on choisira la configuration par défaut (`default`). Le choix de la configuration utilisée est fait dès le lancement du serveur, par le biais du script `run.sh` (ou `run.bat`) avec le paramètre `-c <nom_configuration>`. Sans préciser ce paramètre `-c`, vous lancez la configuration permettant toutes les utilisations courantes (mais sans prise en charge du *farming* et du *clustering*). Les configurations disponibles sont regroupées en dessous du répertoire serveur situé sous la racine d'installation de JBoss.
La figure 8-1 montre la hiérarchie des dossiers sous JBoss et détaille les configurations disponibles.

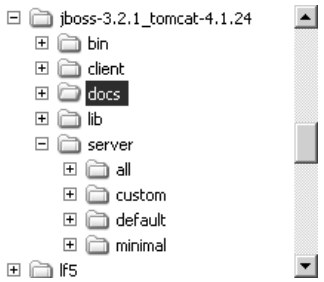


Figure 8–1 Configurations disponibles sous JBoss (en standard)

Au sein de chaque configuration, vous pouvez configurer ou non des bases de données, connecteurs et autres services. Donc l'ajout de votre base de données doit se faire dans au moins une de ces configurations. On admettra désormais que la configuration utilisée est `default` (donc `<chemin_vers_jboss>/server/default`).

L'ajout d'une base de données passe donc par plusieurs phases :

- ajout du pilote JDBC dans le répertoire ;
- ajout d'un service décrivant votre `DataSource` dans le répertoire `deploy` ;
- modification du fichier de mapping objet/relationnel par défaut ;
- suppression de la base de données Hypersonic SQL.

Donc, dans le cas de PostgreSQL, un petit détour par le site http://jdbc.postgresql.org permet d'obtenir la dernière version du driver permettant de piloter votre base. Ce fichier doit être simplement copié vers le répertoire : `<jboss_home>/serveur/default/lib`.

ATTENTION **Version de JBoss**

Cette description n'a de sens que pour la version courante de JBoss, soit la version 3.x.

Puis insérez un fichier nommé `postgres-service.xml` contenant ce qui suit.

Fichier de configuration d'une source de données Postgres dans JBoss 3

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
  <mbean code="org.jboss.resource.connectionmanager.
              ➥ LocalTxConnectionManager"
         name="jboss.jca:service=LocalTxCM,name=PostgresDS">
    <depends optional-attribute-name="ManagedConnectionFactoryName">
      <mbean code="org.jboss.resource.connectionmanager.RARDeployment"
             name="jboss.jca:service=LocalTxDS,name=PostgresDS">
        <attribute name="JndiName">DefaultDS</attribute>
        <attribute name="ManagedConnectionFactoryProperties">
          <properties>
            <config-property name="ConnectionURL" type="java.lang.String">
                  jdbc:postgresql://localhost:5432/testdb
            </config-property>
            <config-property name="DriverClass" type="java.lang.String">
             org.postgresql.Driver</config-property>
    <!--set these only if you want only default logins,
        not through JAAS-->
```

Fichier de configuration d'une source de données dans JBoss 3.x.

Il s'agit là du seul passage nécessitant un paramétrage. Vous devez ici faire refléter votre installation ; dans le cas présent, la même machine héberge le serveur d'applications et le serveur de bases de données. Attention, ceci n'est pas forcément le cas dans un contexte professionnel. Ici la base de données utilisée s'appelle `testdb`.

Ici on utilise une authentification directe (sans passer par JAAS), le mot de passe n'est pas renseigné, seul un nom d'utilisateur valide est fourni.

Fichier de configuration d'une source de données Postgres dans JBoss 3 (suite)

```
          <config-property name="UserName" type="java.lang.String">
               jerome</config-property>
          <config-property name="Password" type="java.lang.String">
          </config-property>
         </properties>
       </attribute>
       <depends optional-attribute-name="OldRarDeployment">
           jboss.jca:service=RARDeployment,
           name=JBoss LocalTransaction JDBC Wrapper</depends>
     </mbean>
   </depends>
   <depends optional-attribute-name="ManagedConnectionPool">
     <!--embedded mbean-->
     <mbean code="org.jboss.resource.connectionmanager.
                  ➥ JBossManagedConnectionPool"
             name="jboss.jca:service=LocalTxPool,name=PostgresDS">
       <attribute name="MinSize">0</attribute>
       <attribute name="MaxSize">50</attribute>
       <attribute name="BlockingTimeoutMillis">5000</attribute>
       <attribute name="IdleTimeoutMinutes">15</attribute>
       <attribute name="Criteria">ByContainer</attribute>
     </mbean>
   </depends>
   <depends optional-attribute-name="CachedConnectionManager">
       jboss.jca:service=CachedConnectionManager</depends>
   <depends optional-attribute-name="JaasSecurityManagerService">
       jboss.security:service=JaasSecurityManager</depends>
   <attribute name="TransactionManager">java:/TransactionManager
    </attribute>
   <!--make the rar deploy! hack till better deployment-->
   <depends>jboss.jca:service=RARDeployer</depends>
  </mbean>
</server>
```

ASTUCE Nommage d'un service dans JBoss 3.x

JBoss présume que tout service est contenu dans un fichier dénommé :
`<nomduservice>-service.xml`

Attention ! Pour que le déploiement soit un succès, vous devez supprimer le fichier de déploiement de HSQLDB (fournie avec JBoss). Pour cela, rien de plus simple : supprimez le fichier `hsqldb-ds.xml` du répertoire `deploy` (et supprimez éventuellement le pilote dans le répertoire `lib`).

La dernière étape est simple. Il suffit de modifier le fichier `standjbosscmp-jdbc.xml` (disponible dans le répertoire `conf`) de manière à changer le *mapping* standard. Vous trouverez ci-après un extrait (seule partie à modifier en réalité) de ce fichier. Et c'est tout…

Extrait du fichier de configuration par défaut de la persistance dans JBoss

```
<jbosscmp-jdbc>
   <defaults>
      <datasource>java:/DefaultDS</datasource>
      <datasource-mapping>PostgreSQL</datasource-mapping>

      <create-table>true</create-table>
      <remove-table>false</remove-table>
```

Contrôler la configuration de JBoss

Le lancement de JBoss et l'examen des traces est évidemment la première solution (la plus simple et la plus naturelle).

Pour mémoire, les traces du serveur sont disponibles dans le fichier `<jboss_home>/server/default/logs/server.log`.

Une autre solution consiste à utiliser la console JMX de JBoss, de manière à visualiser les informations exactes sur les composants déployés au sein de notre serveur d'applications. Avant d'en dire plus sur JMX, regardons un peu cette drôle de bête dénommée console JMX… La figure 8-2 montre une capture de cette console.

Figure 8–2
Console JMX de JBoss vue depuis Mozilla

> NORME **JMX, l'administration facile**
>
> JMX en question… Cette norme a pour but de donner un moyen standard d'administrer les applications. Elle s'appuie sur des composants dits MBeans (des sortes de JavaBeans) déployés au sein d'un serveur particulier appelé *MBean server*. Ces composants seront administrables à distance par le biais de protocoles (RMI, Corba) et d'adaptateurs (HTML). L'intérêt saute aux yeux après ces quelques illustrations graphiques. Il faut savoir que JMX n'a rien de propre à JBoss, puisque de nombreux produits adoptent dorénavant une console JMX (Weblogic depuis la version 6 ou Tomcat dans sa version 4.1). JBoss se distingue de nombreux autres produits car il ne se contente pas d'exhiber une interface conforme à cette norme : il n'est en réalité qu'un « gros MBean server ». En effet, tout service de JBoss est codé sur la base de Mbeans.

On dispose de beaucoup d'informations dans cet écran, mais allons droit au but et examinons ce qui est une des parties les plus intéressantes de cet outil : l'examen de l'arbre JNDI. Étant donné que l'on utilise un arbre JNDI pour localiser nos objets au sein du serveur et que la base de données configurée a été déclarée sous le nom `defaultDS`, l'examen de cet arbre doit nous renseigner…

La figure 8-3 montre un extrait de l'arbre JNDI (JNDI View).

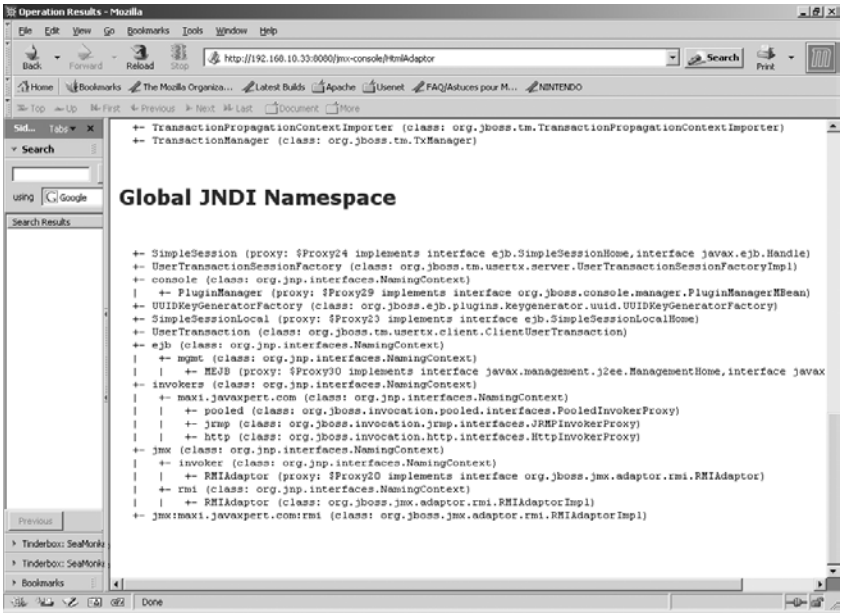


Figure 8–3
Arbre JNDI ; on retrouve notre DataSource

Ce premier indice montre déjà que l'on a bien un objet enregistré dans notre annuaire JNDI avec le bon nom… Maintenant est-ce le bon ? Revenons à la page principale de la console et sélectionnons ce qui est indiqué sur la figure 8-4.

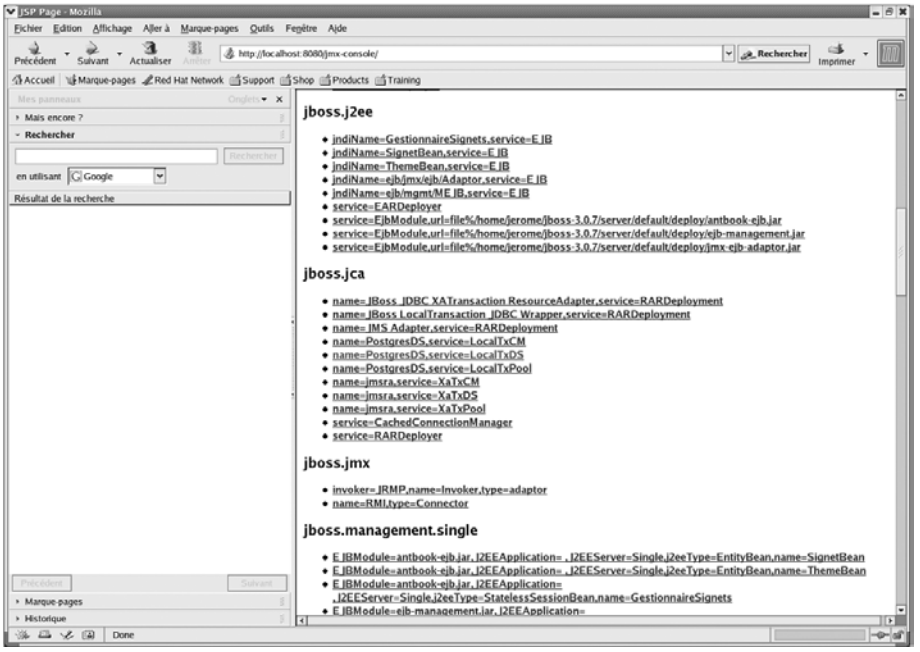


Figure 8–4
L'arbre JNDI avec la liste des connecteurs

La figure 8-5 nous montre le résultat affiché par cette page :

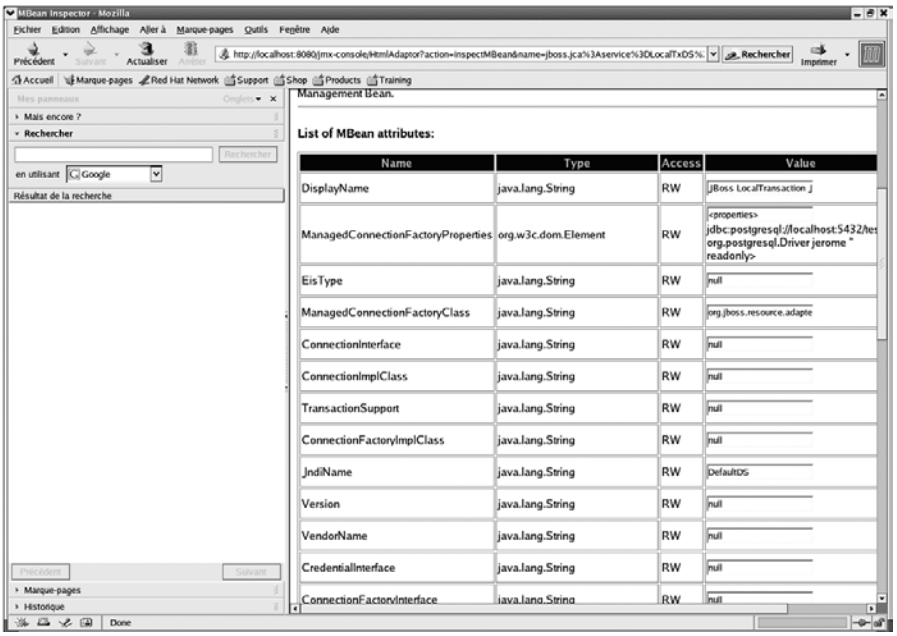


Figure 8–5
Notre DataSource à la loupe

Bien entendu, la réussite de la connexion à la base sous-entend le fait de disposer des autorisations adéquates. La déclaration des droits dépend fortement du type de moteur de base de données utilisé. Dans le cadre de l'utilisation de PostgreSQL, voici un fichier type d'autorisation (`pg_hba.conf`) permettant de se connecter à Postgres en fournissant seulement un nom d'utilisateur valide.

```
# TYPE  DATABASE  USER  IP-ADDRESS      IP-MASK           METHOD
local   all       all                                     trust
host    all       all   127.0.0.1       255.255.255.255   trust
host    all       all   192.168.1.0     255.255.255.0     trust
```

Cette authentification requiert néanmoins d'avoir ajouté explicitement un utilisateur `jerome` dans la liste des utilisateurs Postgres (voir la commande `createuser`). Pour d'autres bases de données, reportez-vous à la documentation des produits.

Tester nos composants

Il serait bien imprudent de commencer à travailler sur l'interconnexion entre les différentes couches sans même avoir pris le soin de tester de manière unitaire nos EJB. Pour cela, un petit programme de test fera très bien l'affaire. Des outils affiliés à JUnit sont disponibles pour l'environnement J2EE mais nous ne les aborderons pas ici.

Classe de test cliente accédant aux objets distribués (EJB)

Cette classe est placée dans le paquetage `client`.

```
package client;
import java.util.Properties;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import ejb.GestionnaireSignets;
import ejb.GestionnaireSignetsHome;
```

La classe de test cliente se connecte à un serveur JBoss configuré avec les valeurs par défaut (1099 pour le serveur JNDI) et on suppose que ce serveur est sur la même machine.

```
/**
* classe de test cliente.
*/
public class TestClient
{
```

La méthode `main ()` obtient la référence sur l'EJB session, appelle les méthodes métier puis affiche les résultats.

```
    /**
    * La classique méthode main(). Fait tout le travail.
    */
    public static void main(String[] args)
    {
```

On crée en mémoire des propriétés permettant d'assurer la connexion au serveur JBoss.Ici ce serveur est supposé être local.
Le travail requis pour lire un fichier de propriétés est quasiment nul… C'est une solution que l'on ne saurait trop conseiller.

```
       Properties env = new Properties();
       env.setProperty("java.naming.factory.initial",
               "org.jnp.interfaces.NamingContextFactory");
       env.setProperty("java.naming.provider.url", "localhost:1099");
       env.setProperty("java.naming.factory.url.pkgs",
           "org.jboss.naming");
       try
       {
```

On obtient un contexte de nommage convenablement configuré. Puis on utilise cet objet pour chercher notre objet (`GestionnaireSignets`).

```
          InitialContext jndiContext = new InitialContext(env);
          Object ref = jndiContext.lookup("GestionnaireSignets");
```

La méthode `narrow` permet d'obtenir une référence valide. Le nom de cette méthode provient en ligne droite du monde Corba.

```
          GestionnaireSignetsHome manager_home =
                (GestionnaireSignetsHome)
          PortableRemoteObject.narrow (ref,
                GestionnaireSignetsHome.class);
```

Ceci fait, on peut en demander une instance (obtenue depuis le pool).

```
          GestionnaireSignets manager = manager_home.create();
```

On fait du ménage en effaçant tous les signets et thèmes…

```
          manager.removeAll();
```

Puis l'on crée quelques objets…

```
      manager.addTheme(1,"titi","un fils rattache a la racine",0);
       manager.addTheme(2," fils de titi","un fils non rattache a la
                    ➥ racine",1);
      String[] themes_list = manager.getThemes();
      for(int i=0;i<themes_list.length;i++){
         System.out.println("theme found = " + themes_list[i]);
      }
      System.out.println("Adding signet " );
      manager.addSignetUnderTheme(1,2,"Java",
          "http://www.javasoft.com/","main java site");
     System.out.println("-----------------------------\nListe des
                    ➥ enfants du theme titi \n-----------");
```

```
        Integer[] children_id =manager.listThemesUnder(1);
        for(int i=0;i<children_id.length;i++){
            System.out.println("J'ai trouve un enfant avec l' id = "
               + children_id[i]);
        }
        System.out.println("-----------------------------\nSignets
                        ➥ rattachés au thème ID = 2");
        Integer[] signets = manager.listSignetsUnder(2);
        for(int i=0;i<signets.length;i++){
            System.out.println("J'ai trouve un enfant avec l' id = "
               + signets[i]);
        }
      }
      catch(Exception e)
      {
         System.out.println(e.toString());
      }
   }
}
```

Ensuite on demande quelques affichages…

Pour lancer ce programme, il suffit d'avoir un `CLASSPATH` contenant :

- le chemin des classes nécessaires au test (`test.ClientTest` et le bean `GestionnaireSignets`) ;
- le fichier `jbossall-j2ee.jar` (fourni dans le répertoire de JBoss) ;
- la bibliothèque de traces `log4j` (fournie avec JBoss dans le même répertoire client).

La figure 8-6 est une capture de l'écran obtenu par l'exécution de `java client.TestClient`.

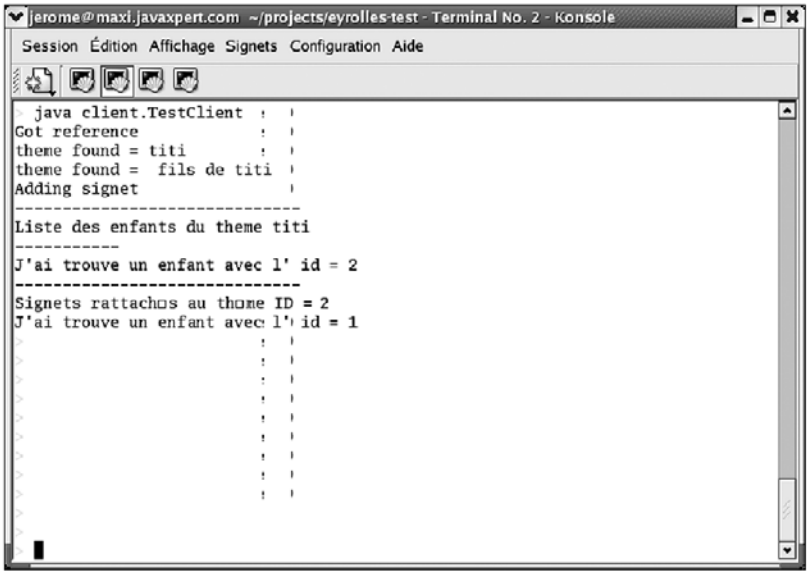


Figure 8–6 Test de notre EJB

La figure 8-7 présente la trace de l'exécution côté serveur.

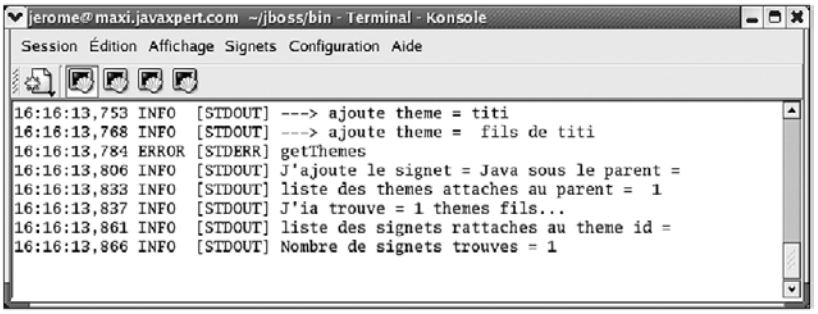


Figure 8–7 Les traces côté serveur

Votre premier EJB est opérationnel.

Ce qu'il reste à faire pour la maquette BlueWeb

Il est l'heure de passer aux bilans. Commençons par celui de notre maquette. Que reste-t-il à implémenter pour arriver aux objectifs initiaux ?

Le cadre de l'interface cliente est prêt. Il reste à coder une logique permettant d'accéder à un modèle de données obtenu par des requêtes HTTP (appels aux servlets) et à transformer les données transportées en XML en objets Java. Cette partie peut être prise en charge par quelques lignes de code utilisant Castor, ces lignes étant bien entendu placées dans la couche de présentation des données (servlet).

Le framework général utilisé côté serveur (servlets) est prêt, il faut lui ajouter quelques commandes à même d'aiguiller les requêtes clientes vers les bons services de notre objet de façade : l'EJB session. Bien entendu, il faudra aussi dans cette couche utiliser Castor de manière à sérialiser les objets en XML. Pour les puristes, l'utilisation d'un filtre (*Filter*) permettrait de créer des traces applicatives très satisfaisantes.

Du côté de la logique métier, l'effort à faire est minime, puisqu'il faut factoriser le code (voir méthode `ejbCreate()` de l'EJB session), appliquer quelques tests permettant de minimiser les risques dus à des incohérences et améliorer la gestion des cas exceptionnels et erreurs applicatives. Bref, tout est en place de ce côté-là.

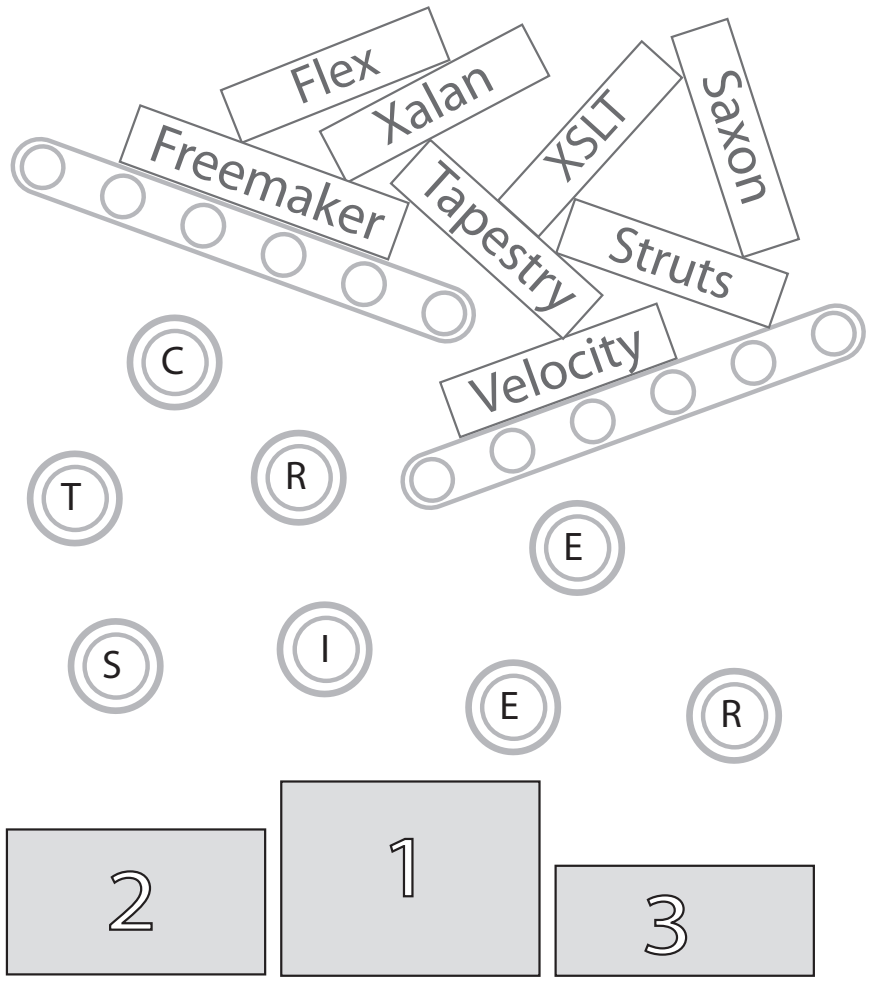
Il est donc laissé à la charge du lecteur désireux d'en savoir plus de finir et d'améliorer ces pistes, de manière à avoir une belle application de gestion des signets.

En résumé… développer avec des outils libres

Nous souhaitons qu'à travers ces huit chapitres et cette application, le lecteur soit en mesure d'appréhender l'état d'esprit nécessaire au développement avec les logiciels libres. De même, ce livre espère être un guide vous permettant de mettre en place votre premier EJB ou votre première servlet.

Qu'il s'agisse de systèmes d'exploitation (Linux ou FreeBSD), de serveurs d'applications (JONAS, Tomcat, Jetty ou JBoss), d'environnements de développement (Eclipse) ou de bibliothèques (JUnit ou castor), le logiciel libre est désormais une réalité économique et un challenge à relever. Quelle satisfaction que de participer, même modestement, à un projet concernant une large communauté (la diffusion des logiciels est mondiale avec le Web) ! Mais attention, le choix des outils demande une attention extrême.

chapitre 9


Flex
Xalan
XSLT
Saxon
Freemaker
Tapestry
Struts
Velocity
C
T
R
E
S
I
E
R
2
1
3

Premières leçons du projet Blueweb

Il est hors de question de se quitter sans prendre le temps de réaliser une synthèse de l'appréhension de ces nouvelles technologies par la sympathique équipe Blueweb. Cette synthèse sera aussi prétexte à l'étude des futures versions des normes proposées dans cet ouvrage et à leurs implications en terme de méthodologie de développement.

SOMMAIRE

MOTS-CLÉS

- Composants
- Frameworks
- Micro-noyaux
- Injection des dépendances
- Inversion des contrôles
- XDOCLET
- framework MVC

Les promesses des EJB

Macroscopiquement, les EJB doivent nous permettre d'aboutir à une grande réutilisation de nos composants métier. Cette promesse n'est pas sans implication lorsqu'elle est formulée à un parterre de concepteurs maniaques des technologies objet. Avant d'aller plus en avant en conjectures philosophiques, revenons à du concret par l'étude d'un code fort simple.

Un petit exemple

Reprenons ensemble l'étude de notre composant chargé d'effectuer une conversion franc vers euro (et vice-versa). On examinera par la suite la seule classe d'implémentation (`ExempleSessionbean.java`).

Implémentation des services

Dans un cas aussi simple, l'implémentation des services l'est forcément aussi. Ici, on se contente d'une multiplication par le taux de base de conversion…

```
package test.ejb;

import javax.ejb.*;

/**
 * <p>
 * Un session Bean tres simple. Propose un seul service, le calcul
d'une conversion franc vers euro ou euro vers franc.
 * N'a d'utilité que pour montrer la masse de travail requise pour le
deploiement d'un EJB.
 * Bien entendu etant donnee la nature simpliste de ce composant il
n'a pas d'interactions avec l'exterieur,
 * donc ne necessite pas de dialogue avec d'autres EJB...
 * Cette classe est la classe d'implementation des services de notre
composant (conversions)
 * En fait on peut se demander si ce n'est pas la seule utile...
 *</p>
 * @author  <a href="mailto:jmoliere@nerim.net">jmoliere@nerim.net</
a>
 */
public class ExempleSessionBean implements SessionBean {
       private float tauxDeBase = 6.55957f;
<B>
    /**
     * Calcul euro vers franc
     * @param aConvertir, montant a convertir (euros) vers des francs
     * @return somme convertie en francs
     */
    public float euroToFranc(float aConvertir) {
  System.err.println("conversion de " + aConvertir + " euros vers des
francs");
            return aConvertir*tauxDeBase;
    }
```

```
    /**
     * Calcul de la conversion francs vers euros
     * @param aConvertir, montant en francs a convertir en euros
     * @return somme convertie en euros
     */
    public float francToEuro(float aConvertir){
            return aConvertir/tauxDeBase;
    }
    /**
     * rien a faire dans le cas de l'activation
     */
    public void ejbActivate() {
    }

    /**
     * rien a faire dans le cas de la passivation
     */
    public void ejbPassivate() {
    }

    /**
     * le contexte est indifferent dans ce cas...
     */
    public void setSessionContext(SessionContext ctx) {
    }

    public void ejbRemove() {
    }

    /**
     * rien a faire dans cette méthode
     */
    public void ejbCreate(){
    }
}
```

Les méthodes suivantes sont en rapport avec le cycle de vie des objets au sein du conteneur EJB : activation, passivation, création ou destruction.

Que peut-on conclure de cet extrait de code ? Notre logique métier soi-disant portable et réutilisable se trouve noyée au sein d'une infrastructure complexe et donc soumise à un fort couplage. De plus, le déploiement au sein d'un conteneur du type EJB complexifie grandement les tâches les plus élémentaires telles que le test. Comme toujours, le monde du logiciel libre regorge de solutions et l'on peut facilement trouver des extensions du framework JUnit permettant de réaliser des tests unitaires sur nos EJB. Néanmoins, même de telles extensions ne faciliteront pas la réutilisation de notre convertisseur en dehors du contexte EJB. Il semble que l'équipe Blueweb tienne là un argument massue à l'encontre de cette technologie.

POUR ALLER PLUS LOIN **Frameworks de tests de composants J2EE**

On peut signaler divers projets permettant d'outrepasser la difficulté de tests de composants tels que les EJB avec des frameworks comme JUnit. Parmi ceux-ci, on peut citer l'excellent projet Cactus de la fondation Apache, disponible à l'URL suivante : http://jakarta.apache.org/cactus/ ou encore le produit JxUnit permettant d'utiliser des fichiers XML en lieu et place de bases de données. Ce dernier est disponible à l'adresse http://jxunit.sourceforge.net/.

IMPORTANT **Portabilité**

Développer un composant attaché à un framework du type EJB revient à renoncer à l'utiliser au sein d'une application autonome ou d'une application web sans conteneur EJB. Ceci est extrêmement dommageable puisque c'est ainsi renoncer aux fondements de l'approche objet.

Composants et dépendances

L'exemple précédent pointe le problème fondamental lié à l'utilisation des EJB : les seules lignes d'imports requises par notre classe pour compiler démontrent le couplage évident entre notre composant et la technologie EJB.

Ceci peut-être résumé par les diagrammes UML des figures 9-1 et 9-2.

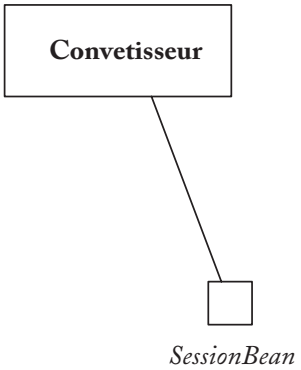


Figure 9–1
Le convertisseur est un bean de session

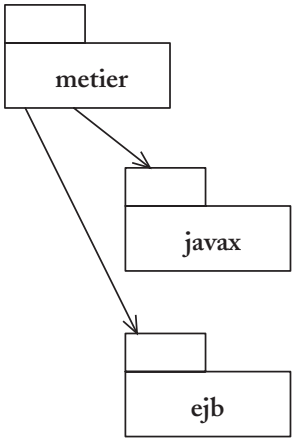


Figure 9–2 Dépendances entre paquetages

DESIGN PATTERN **Injection de dépendances**

L'article de Martin Fowler est disponible sur son site web, à l'adresse suivante :

- http://www.martinfowler.com/articles/injection.html

Il s'agit là d'un véritable tournant dans la conception logicielle et l'on ne peut que chaudement recommander la lecture de cet article qui, sans être la brique fondatrice de cette approche, synthétise très bien la problématique sous-jacente. Cet article a aussi le mérite de dépeindre les différentes approches adoptées pour injecter au déploiement d'un composant ses dépendances. On peut citer le premier article évoquant cette notion et la mettant à portée du plus grand nombre, puisqu'il s'agit d'un article de C++ Report disponible à l'adresse :

- http://www.research.ibm.com/designpatterns/pubs/ph-feb96.txt

Pour être complet dans la généalogie de ce pattern, il faut citer que son origine remonte comme bien d'autres au célèbre centre de recherches de Xerox à Palo Alto au début des années 1980.

De telles dépendances physiques entre nos composants et des classes d'infrastructure telles que celles proposées par le paquetage `javax.ejb` réduisent fortement la portabilité de notre code ; il nous faut donc trouver un moyen de remplacer ces dépendances.

Injection de dépendances

Dans son article « Inversion of Control and the Dependency Injection Pattern », Martin Fowler décrit avec précision un remède (design pattern) permettant d'éviter de tels couplages pour nos composants.

B.A.-BA **Inversion du contrôle**

On désigne ce terme de manière abrégée par « IoC ». Mais de quoi s'agit-il ? Il s'agit simplement du fait de déplacer au sein d'un framework le code nécessaire à la gestion de l'assemblage de vos composants, plutôt que de gérer cet aspect explicitement. Donc, tous les frameworks ou presque proposent cette fameuse inversion du contrôle. Ainsi, J2EE et les EJB proposent un tel aspect.

ALLER PLUS LOIN **Quelle forme d'IoC ?**

Il s'agit bien là du cœur du problème, car avec ce nouveau terme nous n'avons guère fait avancer la science en découvrant qu'un oiseau savait voler… La question à se poser est donc « sous quelle forme va pouvoir se produire cette inversion du contrôle de manière à éviter les couplages et permettre une meilleure approche par composants ? ».

Il s'agit en fait de découpler notre composant du contexte dans lequel on va l'utiliser, en se reposant sur un conteneur de poids mouche (*lightweight container*) procédant à ce que l'on appelle l'injection de dépendances.

En illustrant avec un autre exemple, plus concret pour beaucoup, on pourra appréhender tous les bénéfices que l'on peut retirer d'une telle conception. En utilisant notre application de gestion de signets, on pourrait être tenté par une modélisation du type de celle fournie en figure 9-3.

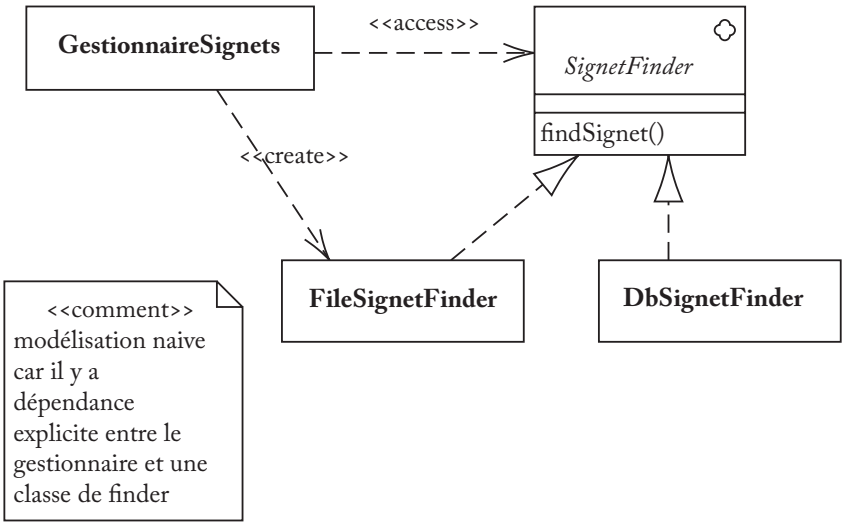


Figure 9–3
Le gestionnaire de signets modélisé

Cette modélisation pose évidemment le problème de dépendances entre notre logique métier de gestion des signets et la classe utilitaire concrète manipulée pour le stockage des données (en base de données ou sous forme de fichiers texte ou XML).

L'IoC propose donc de régler ce problème de dépendances en introduisant un objet tiers dit « assembleur » réalisant l'instanciation des objets `Finder`.

Ceci nous conduirait donc à une modélisation du type de celle proposée en figure 9-3.

L'utilité de cet objet assembleur est donc de ramener le type de persistance utilisé pour le stockage et la lecture de nos signets en base à de la configuration sur notre conteneur… Notre logique métier doit demeurer la même.

L'objet assembleur, en utilisant sa configuration, va donc se trouver en position de déterminer la classe concrète d'implémentation devant être manipulée à l'instant `t` par notre objet gestionnaire de signets. Il va donc se livrer à de l'injection de dépendances…

Pour être complet sur ce sujet, on peut constater qu'il y a diverses manières de se livrer à cette injection de dépendances :

- injection par interfaces (IoC type 1) ;

- injection par accesseurs (IoC type 2) ;
- injection par constructeurs (IoC type 3).

On parlera de conteneurs légers pour une gamme de produits permettant de se livrer à ces injections de code à la volée, sans pour autant être liés à un mode de déploiement (type EJB) autrement que par configuration (donc simplicité de changement).

On peut remarquer que ces divers produits proposent chacun une ou plusieurs façon(s) de voir l'injection de dépendances. Ceci entraînera des optiques différentes de codage de vos composants suivant le framework utilisé. Des trois produits, Avalon est le seul réellement plus ambitieux, plus complexe, ce qui est sûrement cause de la mort annoncée de ce produit ; les débutants pourront donc être rebutés par ce produit. Spring est le projet le plus dans l'air du temps et propose l'intégration de diverses couches (Web avec Webwork et Struts, bases de données avec Hibernate). PicoContainer est selon toute vraisemblance le produit le plus facile à appréhender, quitte à opter pour un autre framework passé le cap de la phase d'apprentissage de cette technologie.

Ce que l'on peut espérer de ces produits

Cette section espère synthétiser les éléments essentiels justifiant l'utilisation de ces produits dénommés micro-noyaux. On peut choisir en toute quiétude un tel produit si l'on souhaite :

- se démarquer d'un fort couplage avec telle ou telle technologie, de manière à envisager une réutilisation réelle de ses composants dans divers contextes ;
- faciliter les étapes de tests et de qualification de ses composants en ne subissant pas le surcoût de complexité induit par des plates-formes aussi lourdes que J2EE ;
- déplacer le couplage envers une technologie vers de la simple configuration (XML typiquement) en ayant la possibilité d'offrir à un composant un aspect de «Session bean» à la demande.

Injection et instrumentation du code

La programmation orientée aspects nous offre la possibilité de sortir hors de notre code métier divers aspects techniques transverses, tels que l'insertion de traces, l'audit du code, la sécurité ou encore la persistance comme dans le cas de l'utilisation conjuguée du produit JBoss AOP et de Hibernate. L'AOP, depuis l'éclosion du projet JBoss, a trouvé un bijou ornant sa couronne et démontrant à la face du monde la puissance de cette nouvelle façon de développer. Comme souvent en informatique, il faut préciser que l'AOP provient des laboratoires de Xerox à Palo Alto (PARC), qui ont introduit avec AspectJ (devenu un sous-projet Eclipse) le premier environnement Java dédié à cette technologie.

B.A.-BA **Conteneurs légers**

On parle aussi de *micro-kernel* (micro-noyaux) pour ces produits permettant de se concentrer sur le développement de composants Java tout à fait standards (ce que l'on appelle parfois des POJO) et reléguant les choix de déploiement à une étape de configuration.

VOCABULAIRE **POJO**

Voici encore un anglicisme sous forme d'acronyme pour qualifier des objets Java normaux, ceux que les anglais appellent des « Bons vieux objets Java». Ceci signifie qu'il ne s'agit pas d'EJB ou de toute incarnation d'une technologie spécifique, mais plutôt d'objets Java répondant à une convention du type Java Beans.

POUR ALLER PLUS LOIN

Choisir un micro-conteneur

On peut citer divers produits libres qualifiables de micro-conteneurs. Parmi ceux-ci, les trois noms les plus fréquemment cités sont :

- Spring, disponible sur le site http://www.springframework.org, qui est un compromis intéressant entre la complexité relative de Avalon et la simplicité de PicoContainer.
- PicoContainer et son homologue NanoContainer, qui sont les produits les plus légers et les plus simples : http://picocontainer.org
- Avalon, le projet Apache, qui est un framework générique permettant de bâtir des conteneurs serveurs et servant d'ossature à divers produits dont James, le serveur SMTP très puissant de la fondation Apache : http://avalon.apache.org/

À NOTER **JBoss AOP**

Jboss AOP est un des nombreux produits disponibles dans le monde Java dédiés à la programmation orientée aspects. On peut citer certains de ses rivaux comme JAC ou AspectJ.

On peut introduire l'instrumentation du *bytecode*, telle que réalisée dans un framework comme Spring avec l'exemple «tarte à la crème» du *logging* réalisé à la volée.

Imaginons donc une classe très simple offrant un service métier quelconque (nous adopterons le traditionnel « HelloWorld » pour ce faire) ; cette classe implémente une interface contenant la définition des services proposés.

Cette classe se concentre sur son coeur de métier sans être dotée de la moindre fonctionnalité technique.

Le diagramme UML de la figure 9-4 décrit le cas de figure envisagé.

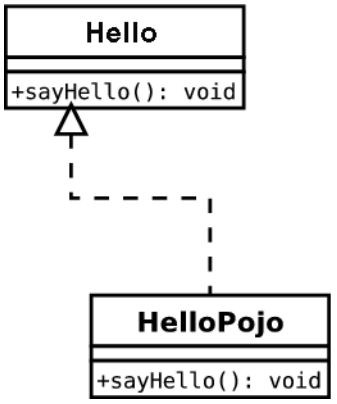


Figure 9–4
Un POJO en action

À NOTER Hibernate

Hibernate est un autre produit hébergé par le JBoss Group. C'est un excellent outil de mapping objet/relationnel, peut-être la solution alliant la plus grande puissance avec les meilleures performances et ce sans payer le prix fort d'un apprentissage rugueux.

En admettant que dans une application utilisant cet objet métier nous voulions obtenir des traces nous donnant le temps nécessaire à l'exécution de chaque service, voici quelles solutions s'offrent à nous :

- ajouter ces traces dans chacune des méthodes visées, ce qui constitue une méthode très intrusive (conséquences sur le code existant), d'autant plus lourde que le nombre de classes est important, et nous amenant à « polluer » notre code avec des aspects techniques transversaux ;
- utiliser un framework d'AOP de manière à contrôler l'instrumentation de nos services métier par fichier de configuration.

Bien évidemment, nous nous intéresserons à cette seconde façon de procéder, en utilisant les facultés de Spring en matière d'AOP.

Nous aurons donc besoin de différents éléments pour mener à bien notre projet dont le code va être contenu dans un paquetage Java `test.pojo` :

- l'interface définissant le service, appelée dans notre exemple `Ihello` ;
- la classe de notre POJO appelée `HelloPojo` ;
- la configuration nécessaire au déploiement de notre composant ainsi qu'à son instrumentation ;
- une classe permettant de mettre en évidence l'instrumentation du code de notre POJO.

Interface implémentée par le POJO

Voici une interface permettant de définir notre service métier.

```
package test.pojo;
/**
 * @author jerome@javaxpert.com
 */
public interface IHello {
   public void sayHello();
}
```

Le fameux POJO

Implémentation minimaliste de notre interface `IHello`. On ne peut que constater l'absence de code de trace dans cette classe. En effet, rien dans le code n'indique l'entrée ou la sortie dans une méthode de notre classe, le `System.out.println()` présent dans la méthode `sayHello` n'étant qu'une implémentation simple d'un service simpliste.

```
package test.pojo;
/**
 * @author jerome
 */
public class HelloPojo implements IHello {

   public HelloPojo(){
      super();
   }

   public void sayHello(){
      System.out.println("Hello from sayPojo::sayHello()");
   }
}
```

La définition des composants dans Spring se fait par ce que l'on appelle le contexte applicatif (*ApplicationContext*) configuré en XML par le biais du fichier `applicationContext.xml` dont le contenu est commenté ci-après.

Configuration du POJO dans Spring

Référence à la DTD permettant de valider ce document XML.

Un contexte applicatif contient une série de beans.
Notre bon vieux POJO se trouve donc cité ici.

On introduit ici un composant technique chargé d'ajouter le code de trace. On nomme de tels composants des intercepteurs.
Ensuite, on introduit un composant virtuel identifié par la chaîne `CallPojo`, de manière à indiquer que l'on souhaite que chaque appel à une méthode de notre Bean Pojo entraîne l'appel à un intercepteur.
On constate que tous les appels à ce composant virtuel induiront le passage par une *`factory`* produisant des *`proxies`*; ces derniers seront instrumentés de manière à déclencher l'appel de nos intercepteurs.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://
www.springframework.org/dtd/spring-beans.dtd">

<beans>
   <bean id="Pojo" class="test.pojo.HelloPojo"/>

   <bean id="logginginterceptor"
         class="test.pojo.interceptors.LoggingInterceptor"/>
   <bean id="CallPojo"
         class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="proxyInterfaces">
        <value>test.pojo.IHello</value>
      </property>
      <property name="interceptorNames">
         <list>
            <value>logginginterceptor</value>
            <value>Pojo</value>
         </list>
       </property>
   </bean>
</beans>
```

Nous devons donc introduire maintenant l'intercepteur chargé d'ajouter les traces chronométrant les invocations de nos services métier. Cette classe est entreposée dans le paquetage `test.pojo.interceptors`.

Implémentation d'un intercepteur de traces

```
package test.pojo.interceptors;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;


/**
 * @author jerome
 *
 */
public class LoggingInterceptor implements MethodInterceptor {


    /* @see org.aopalliance.intercept.MethodInterceptor#invoke(
        org.aopalliance.intercept.MethodInvocation)
    */
   public Object invoke(MethodInvocation arg0) throws Throwable {
      System.out.println("Avant invocation de la méthode ="
                        + arg0.getMethod() + " sur la classe = "
                        + arg0.getMethod().getDeclaringClass());
      long start=System.currentTimeMillis();
      try{
         Object returned=arg0.proceed();
         return returned;
      }
      finally{
         long stop=System.currentTimeMillis();
         System.out.println("invocation terminee en :" + (stop-start)
                        + " ms pour la methode :"+ arg0.getMethod());
      }
   }
}
```

Le paquetage contenant les intercepteurs associés à nos composants métier.
Les imports nécessaires à la compilation de notre intercepteur.

Un intercepteur au sens du framework AOP de Spring se doit d'implémenter l'interface **`MethodInterceptor`**, qui se contente de définir une seule méthode, la méthode **`invoke()`**.

Le code d'une telle méthode se déroule en trois temps :
- ce qui doit être fait avant d'invoquer la méthode désirée ;
- l'appel de la méthode ;
- les traitements postérieurs à l'invocation de la méthode.

REMARQUE **Ressemblance**

Notez bien le degré de ressemblance existant entre ce composant et celui proposé dans le chapitre 5. Il ne faut pas y voir une simple coïncidence.

Il ne nous reste plus qu'à réaliser une petite classe de test invoquant le service **`sayHello()`** à travers le framework Spring.

Classe de test de mise en oeuvre de notre POJO enrichi

```
package test.pojo;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;

import org.springframework.beans.factory.xml.XmlBeanFactory;
```

Paquetage où l'on place notre code de test.
Les imports nécessaires à la compilation du projet.

On doit charger notre contexte applicatif (définition des Beans manipulés dans notre application).
Ce contexte nous permet d'initialiser une `Factory` nous permettant de rechercher les objets manipulés par l'application. Par ce biais, on délègue au framework le soin de résoudre les dépendances et d'initialiser correctement nos objets.

On recherche dans la *factory* une référence sur un composant identifié par `CallPojo`. Cette référence est celle correspondant à notre POJO instrumenté par l'intercepteur de traces.

Ici, on manipule la référence directe à notre POJO, donc cet appel ne sera pas instrumenté.
Nous pourrons nous en convaincre par l'examen de la console...

```
/**
 * @author jerome
 *
 */
public class UI {

   public static void main(String[] args) throws FileNotFoundException
{
     InputStream is = new FileInputStream("applicationContext.xml");
     XmlBeanFactory factory=new XmlBeanFactory(is);


   IHello hello= (IHello)factory.getBean("CallPojo");
     hello.sayHello();



     // ici on utilise la bean definition Pojo!!
     //  la sortie montrera une invocation directe de la methode
sayHello()
     IHello hello2= (IHello)factory.getBean("Pojo");
     hello2.sayHello();

   }
}
```

Maintenant que les éléments sont en place, examinons la sortie écran correspondant à l'exécution de notre application de test.

POUR ALLER PLUS LOIN **Spring**

Ce chapitre ne prétend pas être un manuel consacré à Spring ni à aucun autre des outils présentés. Le lecteur intéressé par ce produit trouvera avec le manuel de référence une très bonne source de documentation.

Struts ou un autre framework web ?

Ce sujet épineux n'ayant pas été évoqué jusque-là, il est temps de donner le ressenti de l'équipe Blueweb quant à cette question centrale. Cependant, avant de prononcer le nom du vainqueur, commençons par introduire le problème.

La problématique

La mise à disposition des systèmes d'information à des clients légers est un impératif pour toutes les entreprises. En effet, les coûts de déploiement et de mise à jour de clients riches sont autant d'obstacles à la prolifération de telles solutions, même si certaines alternatives existent (notamment dans le monde Java, l'utilisation d'un produit comme Java Web Start). L'utilisation de clients légers offre un moyen simple de reléguer les problèmes de mises à jour au placard, tout en s'intégrant parfaitement dans le cadre des politiques de sécurité mises en place dans la plupart des structures, puisque le protocole HTTP est un protocole bien maîtrisé et autorisé dans la plupart des configurations de pare-feu. Le besoin de création de pages web dynamiques est donc entendu, puisque les applicatifs réclament de la personnalisation pour chaque client (suivi d'un compte bancaire, boutique en ligne, etc.). Il s'agit donc ici d'envisager la solu-

tion permettant aux équipes de développement de produire des pages dynamiques le plus efficacement possible. Comment juger de la pertinence d'un outil, sur quels critères ? Examinons tout d'abord les objectifs des produits à même d'être choisis :

- respect des normes, chartes graphiques pour faire des sites à l'ergonomie soignée et indépendante des frasques des développeurs ;
- solidification du travail en équipe, de manière à fournir une infra-structure permettant d'exploiter au mieux les forces de chaque membre d'une équipe ;
- amélioration de la productivité de l'équipe en mettant à sa disposition divers outils amenant des réponses concrètes à des problèmes récurrents dans le monde du Web ;
- intégration simple dans le cycle de production d'une application.

Maintenant que le cadre est planté, nous pouvons examiner les candidats en lice...

Panel de produits envisageables

On examinera ici divers produits regroupés en familles, permettant de recouper le spectre des candidats suivant leur architecture technique, les familles étudiées étant :

- les frameworks MVC d'où est issu le fameux produit Struts ;
- les technologies à base de moteurs de templates (modèles de documents) ;
- la catégorie des inclassables, répertoriant des produits originaux, voire exotiques.

Ces familles de produits ne sont pas forcément mutuellement exclusives et peuvent donc être utilisées en conjonction, en imaginant dans des cas non triviaux d'utiliser un framework MVC comme celui offert par le projet Spring et un produit de création de vues à partir de pages modèles. Essayons de comprendre les différentes philosophies de produits examinées ici.

Les frameworks MVC

Ces produits proposent dans l'esprit un moyen d'obtenir une séparation stricte des responsabilités entre le modèle de données manipulé, les vues graphiques obtenues à partir de celui-ci (interfaces graphiques) et le composant en charge de l'interaction avec l'utilisateur(contrôleur).

Parmi les produits s'inscrivant dans cette logique, on peut citer :

- Struts, un des projets phares de la fondation Apache, massivement adopté dans l'industrie et très en vogue dans les média spécialisés ;
- Barracuda, un projet très proche dans l'esprit, mais dont le développement est stoppé depuis plusieurs mois ;
- JSF ou Java Server Faces, qui est conçu pour supplanter Struts, puisqu'il est mené par le concepteur de Struts en coopération avec un groupe d'experts internationaux ; c'est un projet encore récent et en pleine évolution.

Après avoir présenté l'approche philosophique adoptée par ces produits, il est bon de s'intéresser au ressenti des utilisateurs de ce type de technologie, arguments principalement tirés de la communauté d'utilisateurs Struts :

- L'impact sur le cadre apporté au travail est certain et universellement reconnu.
- L'influence sur le respect des chartes graphiques est indéniable.
- L'intégration au sein d'un projet en équipe demande de l'organisation et des outils (du type easystruts) car le partage du fichier `struts-config.xml` est une véritable entrave dans le cas d'un développement en équipe.
- Les outils d'amélioration de la productivité (validation de formulaires ou couches d'internationalisation) n'apparaissent pas réellement indispensables car souvent trop limités.

En approfondissant un peu, on va écorner un peu plus cette technologie en constatant quelques limites de ce produit :

- Si le Design Pattern MVC ne s'intéresse en rien aux types de vues renvoyées vers les postes clients, il est évident de constater le couplage fort entre Struts et le Web. C'est assez décevant, mais seulement conceptuellement, car il faut reconnaître qu'il s'agit là d'un besoin plus que fréquent actuellement. À titre d'exemple, même si la délivrance d'une interface graphique SWING est possible avec Struts, ceci réclame beaucoup de travail et de méthodologie.
- Il est dommageable pour ces technologies de constater le faible degré de réutilisation des composants graphiques qu'elles induisent. Ceci va totalement à l'encontre de l'adoption de technologies objet, propices à la création de composants réutilisables. Il paraît ainsi très surprenant d'être obligé de redévelopper de projet en projet des composants tels que des formulaires de transfert de fichiers (*upload* de documents).
- Il est extrêmement facile (c'est un effet pervers et non une obligation inhérente à la technologie) de voir des pages réalisées sous la forme de concaténation de diverses technologies (HTML standard, Javascript, VBScript et scriptlets Java). Ce point est crucial, car ce type de dérive rend vain l'espoir de voir les responsabilités séparées entre les membres d'une équipe. Des pages constituées de quatre langages différents doivent être maintenues par un bon développeur web connaissant Java : un comble pour une technologie censée favoriser la séparation des responsabilités.
- L'utilisation massive de JSP suivant les recommandations Struts pose un grave souci pour le travail des développeurs, qui n'est pas directement imputable à Struts. En effet, une JSP n'étant qu'une page transformée à la volée par le conteneur en une servlet Java, le développeur obtiendra en cas d'incident à l'exécution un rapport d'erreurs contenant des références à des classes inconnues. Les phases de mise au point d'applications web utilisant Struts doivent être évaluées avec finesse par le chef de projet en tenant compte de ce facteur.
- La multiplication des bibliothèques intitulées *tags libraries* a aussi l'effet pernicieux de voir de plus en plus de pages JSP briser le confinement architec-

tural qui est celui souhaité par tous. Certaines bibliothèques de ce type permettent en effet de réaliser des appels à la base de données directement depuis une JSP.Où est donc passé le MVC dans un tel cas de figure ?

Enfin, on pourra préciser qu'il ne faut pas négliger les difficultés que peuvent poser de tels produits, qui dans tous les cas demandent de l'apprentissage.

POLÉMIQUE **Struts**

Cette section peut choquer certains lecteurs, qui peuvent argumenter que sur tel ou tel projet, ils ont pu obtenir une application très propre conceptuellement. Les critiques prononcées ici ne se veulent que l'expression de retours d'expérience concrets et ne prétendent en rien décrire toutes les applications Struts.

Technologies à base de modèles de documents

Les moteurs de templates sont des composants (Java dans notre contexte) destinés à rendre le processus de délivrance d'un document aussi simple et rapide que possible, dans la mesure où ce document peut-être envisagé comme un assemblage de sections types paramétrées par un contexte d'utilisation. Ces produits doivent être perçus comme des moyens de délivrer des vues (au sens du MVC) et peuvent donc être intégrés dans d'autres produits examinés ici (comme dans le cas de l'extension de Spring dédiée au Web).

Ainsi, imaginons le cas d'une boutique en ligne souhaitant notifier par courrier électronique tout acheteur suite au passage d'une commande par celui-ci. Ce courrier pourrait être présenté sous la forme suivante :

```
Cher Monsieur Martin,
l'équipe Maboutique.com vous remercie de votre confiance suite à
l'achat fait ce jour: 20/11/2004 à 8h et composé des articles suivant:
Produit                          QuantitéPrix Unitaire euros
Cahiers du programmeur Java tome 2 - J.MOLIERE1225
A kind of Blue - Miles Davis114,80

Soit une commande d'un montant de 39,80 euros réglé par carte bleue.

A bientôt sur MaBoutique.com
```

Un tel courrier peut être vu comme le fruit d'une transformation d'un document original en tenant compte d'informations contextuelles, dans notre cas :

- nom du client ;
- date du jour ;
- heure de la commande ;
- détail des composantes de la commande ;
- montant de la commande ;
- type de règlement effectué.

Ce document original pourrait être écrit de la façon suivante :

```
Cher/e Monsieur/Madame ${client.name},
l'équipe Maboutique.com vous remercie de votre confiance suite à
l'achat fait ce jour: ${date} à ${time} et composé des articles
suivant:
Produit               Quantité            Prix Unitaire euros
${commande.item}      ${commande.item.nb} ${commande.item.prix}
```

```
Soit une commande d'un montant de ${commande.montant} euros réglée par
${commande.type.reglement}.

   A bientôt sur MaBoutique.com
```

À partir d'un tel modèle et des informations du contexte (accès aux composants Java contenant les informations), il est facile d'imaginer le travail effectué par un moteur de templates...

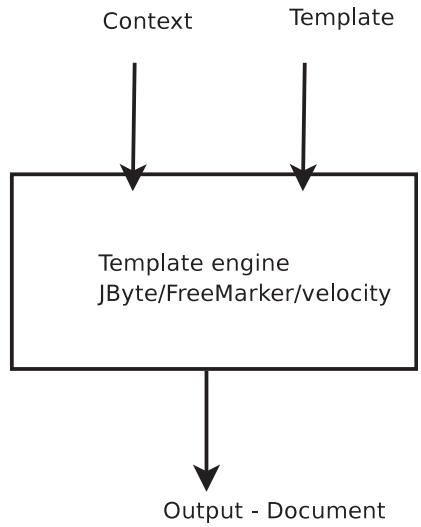


Figure 9–5 Utilisation d'un moteur de templates

Les produits les plus remarquables dans le monde Java sont :

- Freemarker, produit stable et performant, peut-être trop à en juger par la complexité permise par son langage de script ;
- Velocity de la fondation Apache, stable, robuste et puissant, tout cela avec un langage de script (VTL) réduit au strict minimum : un produit à posséder dans sa boîte à outils ;
- StringTemplate, un produit issu des travaux de Terrence Parr, l'auteur du générateur de parseur de renom : ANTLR. Cet outil est disponible à l'adresse suivante : http://www.stringtemplate.org/.

Une autre façon d'envisager ce type de transformations guidées par un modèle consiste en l'utilisation d'un composant proposant la technologie XSLT. Cette façon d'appréhender le problème permet de découpler complètement le modèle de sortie du modèle de données. On peut citer divers composants Java dans le monde du logiciel libre proposant la norme XSLT :

- Xalan de la fondation Apache,
- Saxon.

PRÉCISION JByte

La figure 9-5 mentionne un autre produit (JByte) qui, à défaut d'être célèbre, brille par son extrême légèreté, puisqu'en 8 kilo-octets, il vous ouvre la perspective d'utilisation d'une telle technologie.

B.A.-BA XSLT

La transformation de documents XML guidée par des feuilles de style (elles aussi exprimées en XML) permet d'injecter, dans un composant, une source d'informations (fichier XML) et une feuille de style (extension en `.xsl`), de manière à produire diverses sorties en fonction de la feuille de style choisie. Des produits comme Cocoon utilisent cette technique dans un environnement web, tandis que DocBook est un produit destiné à la création de livres ou articles en XML.

POUR ALLER PLUS LOIN XSLT

Le lecteur désireux d'en savoir plus sur XSLT et sa manipulation en Java est invité à se reporter à l'excellent ouvrage sur Java & XML par Renaud Fleury publié par les Éditions Eyrolles...

XSLT est une technologie attrayante mais pose encore bien des soucis en matière de performance (transformations complexes et lourdes). De plus, il est bon de noter que XSL, la technologie normalisée permettant de développer des feuilles de style, est assez complexe à maîtriser.

Les principaux bénéfices que l'on peut retirer de l'utilisation d'un moteur de templates sont :

- respect d'une charte graphique, argument évident puisque le modèle de document serait dans le cas du Web une page HTML, utilisant une feuille de style (CSS) issue de la charte graphique de l'entreprise ;
- simplicité de l'API propre à ses produits ;
- performances excellentes puisque des produits comme Velocity disposent de caches réduisant au minimum les lectures de données sur le disque.

Dans le cas d'un environnement web, ce type de technique peut paraître délicat à mettre en œuvre, car il réclame une bonne connaissance de la servlet API (servlets, filtres HTTP, listeners de session...).

Les inclassables

Ces produits originaux ne peuvent être classés qu'en un groupe ésotérique. Nous nous intéresserons à deux de ces produits particulièrement originaux :

- Tapestry, un projet issu de la fondation Apache (sous-projet Jakarta),
- Flex de Macromedia, une solution propriétaire fort originale.

Tapestry, le développement 100 % web

Ce projet est en fait une libre adaptation dans l'environnement Java du framework conçu par Apple pour le langage Objective-C. Il affiche clairement ses objectifs :

- permettre une réutilisation des composants graphiques, par le concept d'une palette enrichie de projets en projets ;
- rendre possible la stricte séparation des responsabilités en cantonnant les développeurs Java à l'élaboration de composants (développés en Java), tandis que les développeurs web travaillent dans l'environnement qui leur est familier : le Web, l'inclusion de composants dans les pages HTML se faisant par le biais d'une balise HTML standard, reconnue par les outils de développement type Dreamweaver ou Quanta sous Linux ;
- éviter la complexité de la phase de mise au point d'une application inhérente à l'utilisation des technologies comme les JSP en fournissant toutes les informations utiles au développeur.

APARTÉ Le problème du développement avec des JSP

Toute personne ayant manipulé cette technologie s'est trouvée confrontée à une erreur dans une page induisant une exception traduite en une pile d'appels obscurs puisque inconnus du développeur de la dite page. Il s'agit là du problème le plus perturbant lors d'un développement avec cette technologie.

Flex, la solution joker de Macromedia

Décu par la pauvreté des interfaces proposées à un utilisateur de site web par le langage HTML (même complété par un attirail de scripts JavaScript) ? Alors peut-être que la solution Flex est faite pour vous.

Macromedia, éditeur du célèbre langage Flash conférant un dynamisme supérieur aux pages web (jeux, animations graphiques), propose ce produit permettant de créer des pages incluant des appels à des composants Flash produits dynamiquement. Ces scripts Flash sont décrits en XML (fichiers d'extension `.mxml`) et compilés en Flash par une application web déployée sur un serveur J2EE (Tomcat, Jetty ou JBoss par exemple) et communiquant avec une couche de services métier par le biais de services web en SOAP (XML sur HTTP). Ce type d'architecture se traduit par une vue du type de celle présentée en figure 9-6.

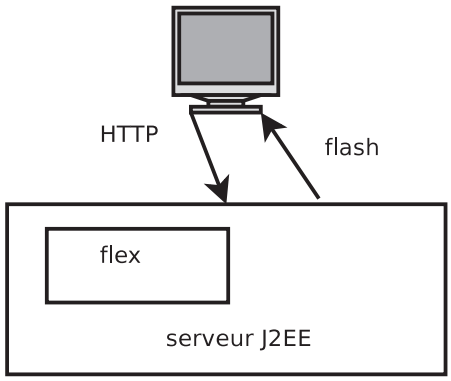


Figure 9–6 Architecture de Flex

Elle peut aussi se traduire sous la forme d'un diagramme de séquences UML fourni en figure 9-7.

Basiquement, Flex se présente donc sous la forme d'une application web (`.war`) déployée au sein de votre serveur d'applications. Cette application web se compose de différents éléments :

- un cache d'applications stockant les fichiers `.SWF` envoyés aux clients,
- un module d'analyse des descripteurs d'applications Flex (fichiers `.mxml`) permettant de résoudre les dépendances d'une application,
- un compilateur générant les fichiers à destination du module Flash installé sur le poste client.

En mode de développement, une requête émise depuis le poste client peut se traduire par le séquencement suivant (figure 9-8).

ALTERNATIVE **Lazlo**

Ce projet commercial à la base, vient d'être offert à la communauté. Il se pose en concurrent direct de Flex et est disponible à l'adresse suivante :

- http://www.laszlosystems.com/

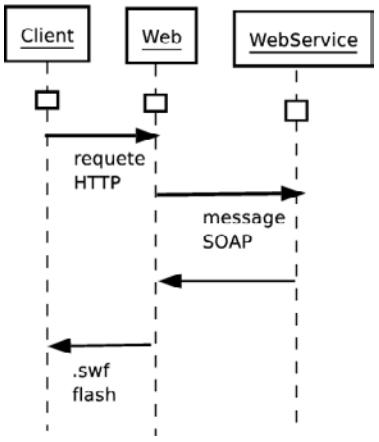


Figure 9–7 Architecture de Flex (UML)

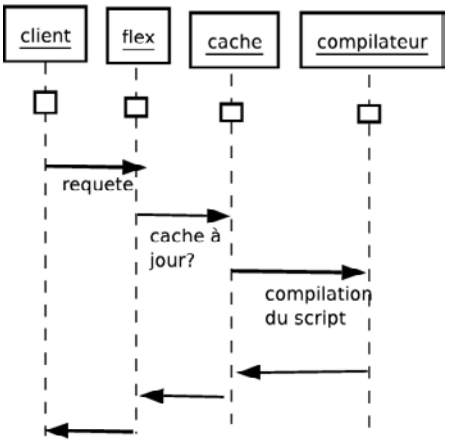


Figure 9–8 Compilation d'un script à la volée

Synthèse des solutions évoquées

Le tableau suivant présente une vue synthétique des divers produits présentés.

Produit	Licence	Doc	Avantages	Inconvénients
Struts	libre (ASF)	livres, articles	structurant, populaire	complexité, loudeur de l'intégration, mise au point de l'application.
Moteurs de modèles	diverses	suivant le produit	API simple, performance	travail de conception nécessaire.
Flex	propriétaire	fournie avec le produit	Puissance de l'interface graphique délivrée au client.	Flex se comporte comme une boîte noire. Disponibilité d'un player Flash compatible avec Flex sur d'autres plates-formes que Linux.
Tapestry	libre	livres, articles	séparation du travail, réutilisation de composants	limitation de l'interface graphique délivrée à du simple HTML.

JMS

L'équipe Blueweb ne peut que déplorer le fait que ce projet pilote n'ait pu entraîner l'utilisation de cette API standard dans le monde Java. L'introduction d'un gestionnaire de messages aurait été purement artificielle dans le cadre du cahier des charges de l'application de gestion des signets.

POUR ALLER PLUS LOIN **JMS**

Le lecteur intéressé par cette norme trouvera un chapitre complet et du code exemple dans l'ouvrage à paraître aux éditions Eyrolles : *Comprendre et exploiter un serveur d'applications J2EE libre*.

DERNIÈRE MINUTE **Avalon, un futur plus qu'incertain**

Il semblerait que ce projet n'ait pu renaître des cendres du projet compagnon de ce dernier dans la fondation Apache : le projet Phenix. Avalon semble donc définitivement mort et enterré et n'est donc pas un projet à choisir à la légère.
Il n'en demeure pas moins que ce produit a été utilisé avec succès dans divers projets dont James, même si les puristes peuvent regretter le type d'injection de dépendances qu'il propose.

Conclusion

Voilà qui achève le voyage entrepris par Blueweb dans le monde vaste et complexe des technologies J2EE. Nous espérons que ce livre vous fournira les pistes vous permettant d'appréhender globalement cette plate-forme vaste et complexe, d'opérer des choix efficients et réalistes et de trouver des références vers des ouvrages à même de satisfaire votre curiosité.

Javamment...

Index

Dépôt légal : décembre 2004 – N° d'éditeur : 7169 – Imprimé en France

Dans la collection
Les Cahiers de l'Admin
dirigée par Nat Makarévitch

Sécuriser un réseau Linux 2e édition

Bernard BOUTHERIN, Benoit DELAUNAY - N°11445, 2004.
À travers une étude de cas générique mettant en scène un réseau d'entreprise, l'administrateur apprendra à améliorer l'architecture et la protection de ses systèmes connectés, notamment contre les intrusions, dénis de service et autres attaques : filtrage des flux, sécurisation par chiffrement avec SSL et (Open) SSH, surveillance quotidienne... On utilisera des outils Linux libres, réputés pour leur efficacité.

Debian

Raphaël HERTZOG – N° 11398, 2004.
Debian GNU/Linux, distribution Linux non commerciale extrêmement populaire, est réputée pour sa fiabilité et sa richesse. Soutenue par un impressionnant réseau de développeurs dans le monde, elle a pour mots d'ordre l'engagement vis-à-vis de ses utilisateurs et la qualité.

BSD 2e édition

Emmanuel DREYFUS – N°11244, 2003.
Ce cahier révèle les dessous d'UNIX et détaille toutes les opérations d'administration UNIX/BSD : gestion des comptes, initialisation de la machine, configuration des serveurs web, DNS et de messagerie, filtrage de paquets... Autant de connaissances réutilisables sous d'autres systèmes UNIX, et en particulier Linux.

Chez le même éditeur

J.-L. BÉNARD, L. BOSSAVIT, R. MÉDINA, D. WILLIAM
L'Extreme Programming
N. 11051, 2002, 350 pages

V. STANFIELD & R..W. SMITH
Guide de l'administrateur Linux
N°11263, 2003, 654 pages.

C. AULDS.
Apache 2.0 Guide de l'administrateur Linux.
N°11264, 2003, 582 pages.

G. MACQUET – **Sendmail**
N°11262, 2003, 293 pages.

C. HUNT. – **Serveurs réseau Linux.**
N°11229, 2003, 650 pages.

R. RUSSELL et al. - **Stratégies anti-hackers**
N°11138, 2e édition 2002, 754 pages.

Dans la collection
Les Cahiers du programmeur

Mac OS X
Gestionnaire de photos avec Cocoa, REALbasic et WebObjects.
Alexandre Carlhian, Jacques Foucry, Jean-Philippe Lecaille, Jayce Piel – avec la collaboration d'Olivier Gutknecht – N°11192, 2003.
Réalisez un gestionnaire de photos consultable via le Web avec Cocoa et Objective-C, REALbasic et WebObjects.

PHP 5
Application de chat avec PHP 5 et XML
Stéphane Mariel – N°11234, 2004.
De la conception à l'exploitation, on créera une application de discussion en ligne en PHP 5 respectant les méthodes éprouvées du développement web : architecture MVC, conception modulaire avec les interfaces, sessions, gestion d'erreurs et exceptions, échanges XML et transformations avec DOM, XPath et SimpleXML.

PHP (2)
Ateliers Web professionnels avec PHP/MySQL et JavaScript.
Philippe CHALEAT et Daniel CHARNAY – N°11089, 2002.
En une douzaine d'ateliers pratiques, allant de la conception d'aides multi-fenêtrées en JavaScript à la réalisation de services Web, en passant par les templates PHP et les annuaires LDAP, on verra qu'autour de formulaires HTML, on peut sans mal réaliser des applications légères ergonomiques et performantes.

PostgreSQL
Services Web professionnels avec PostgreSQL et PHP/XML.
Stéphane MARIEL – N°11166, 2002.
Ce cahier montre comment réaliser simplement des services Web avec PostgreSQL, PHP et XML. Le développeur apprendra à modéliser sa base, tirer parti de la richesse de PostgreSQL (transactions, procédures stockées, types de données évolués...), optimiser ses performances et en automatiser l'administration, sans oublier la réalisation d'un affichage dynamique avec XSLT.

Dans la collection
Accès libre

Débuter sous Linux.
S. Blondeel, H. Singodiwirjo. – N°11349, 2004, 328 pages.
Cet ouvrage guidera des utilisateurs motivés, qu'ils aient ou non utilisé un système MS-Windows, vers la connaissance, l'utilisation et l'administration du système libre et gratuit Linux, qui offre, outre sa puissance, les indispensables de tout poste de travail : traitements de texte, tableurs, mail, navigation Web, messagerie instantanée, jeux, multimédia.

OpenOffice.org efficace.
S. Gautier, C. Hardy, F. Labbe, M. Pinquier.
N°11348, 2004, 350 pages.
OpenOffice.org, suite bureautique gratuite tournant sous Windows, Linux et Mac OS X, inclut tous les modules habituels : traitement de texte, tableur de calcul, présentation, dessin, formules... Écrit par les chefs de file du projet français OpenOffice.org, cet ouvrage montre comment optimiser son environnement de travail et l'utilisation de chaque module d'OOo, comment s'interfacer avec des bases de données telle MySQL, et offre enfin un précis de migration.

Réussir un site Web d'association...
avec des outils gratuits.
A.-L. Quatravaux, D. Quatravaux. – N°11350, 2004, 280 pages.
Souvent peu dotée en moyens, une association doit gérer ses adhérents et membres, faciliter l'organisation du travail entre eux, être visible et soigner son image. Depuis le choix de l'hébergement jusqu'au référencement, en passant par la personnalisation graphique sous SPIP, la configuration d'un serveur d'e-mailing et la création de listes de diffusion, ce livre explique