Développement JEE 5 avec Eclipse Europa

Karim Djaafar

Développement JEE 5 avec Eclipse Europa

CHEZ LE MÊME ÉDITEUR

Ouvrages sur Java et JEE

A. Patricio. – **JPA et Hibernate**.
N°12259, 2008, 390 pages.

J. Dubois, J.-P. Retaillé, T. Templier. – **Spring par la pratique**.
Mieux développer ses applications Java/J2EE avec Spring, Hibernate, Struts, Ajax...
N°11710, 2006, 518 pages.

A. Goncalves. – **Cahier du programmeur Java EE 5**.
EJB 3.0, JPA, JSP, JSF, Web Services, JMS, GlassFish, Ant.
N°12038, 2007, 340 pages.

C. Delannoy. – **Programmer en Java**. *Java 5 et 6.*
N°12232, 5e édition, 2007, 800 pages + CD-Rom.

E. Puybaret. – **Cahier du programmeur Swing**.
N°12019, 2007, 500 pages.

E. Puybaret. – **Cahier du programmeur Java 1.4 et 5.0**.
N°11916, 3e édition, 2006, 380 pages.

J.-P. Retaillé. – **Refactoring des applications Java/J2EE**.
N°11577, 2005, 390 pages.

R. Pawlak, J.-P. Retaillé, L. Seinturier. – **Programmation orientée aspect pour Java/J2EE**.
N°11408, 2004, 462 pages.

R. Fleury. – **Cahier du programmeur Java/XML**. *Méthodes et frameworks : Ant, Junit, Eclipse, Struts-Stxx, Cocoon, Axis, Xerces, Xalan, JDom, XIndice…*
N°11316, 2004, 228 pages.

J. Weaver, K. Mukhar, J. Crume. – **J2EE 1.4**.
N°11484, 2004, 662 pages.

Autres ouvrages sur le développement Web

C. Porteneuve – **Bien développer pour le Web 2.0 – Bonnes pratiques Ajax**.
N°12028, 2007, 580 pages.

R. Goetter. – **CSS 2 : pratique du design web**.
N°11976, 2e édition, 2007, 350 pages.

T. Templier, A. Gougeon. – **JavaScript pour le Web 2.0**.
N°12009, 2007, 492 pages.

M. Plasse. – **Développez en Ajax**.
N°11965, 2006, 314 pages.

D. Thomas *et al*. – **Ruby on Rails**.
N°12079, 2e édition, 2007, 750 pages.

E. Daspet et C. Pierre de Geyer. – **PHP 5 avancé**.
N°12167, 4e édition 2007, 792 pages.

Développement JEE 5 avec Eclipse Europa

Karim Djaafar

Avec la contribution de Olivier Salvatori

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

DANGER
LE PHOTOCOPILLAGE TUE LE LIVRE

Le code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée notamment dans les établissements d'enseignement, provoquant une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'éditeur ou du Centre Français d'Exploitation du Droit de Copie, 20, rue des Grands-Augustins, 75006 Paris.
© Groupe Eyrolles, 2008, ISBN : 978-2-212-12061-5

Mise en page : TyPAO
Dépôt légal : décembre 2007
N° d'éditeur : 7755
Imprimé en France

Avant-Propos

Depuis la sortie de mon premier ouvrage sur Eclipse, il y a quatre ans, couvrant la version 2.0, la plate-forme a énormément changé. Eclipse était à l'époque encore peu utilisé par la communauté Java, qui recourait dans sa grande majorité à des IDE tels que JBuilder ou NetBeans.

Avec la release Europa, sortie début juillet 2007, Eclipse est devenu une plate-forme de référence pour des projets complexes. En ce sens, la fondation Eclipse a atteint son but, qui était de fournir à la communauté Open Source et aux développeurs professionnels une plate-forme fiable bâtie autour d'un ensemble de produits, d'outils et de services.

L'aboutissement de ce long travail a été la mise au point du projet Web Tools, conçu et supporté par les grands éditeurs du moment : IBM, bien sûr, avec sa suite IRAD (IBM Rational Application Developer), mais aussi BEA, avec sa suite WebLogic Workshop, et bien d'autres.

Dans le sillage de la sortie d'Europa sont venus se greffer plus d'une vingtaine de sous-projets aussi ambitieux que ceux des releases précédentes. Citons notamment l'outil de reporting évolué BIRT (Business Intelligence and Reporting Tool), le support du célèbre langage interprété et objet Ruby, avec le projet Eclipse Dynamic Language Tool Kit, les fonctionnalités avancées de collaboration en ligne, avec le sous-projet Mylyn, ou encore le mapping O/R, avec Dali.

Cet ouvrage a pour principal objectif de montrer la maturité de tous ces projets et leur couverture du cycle de développement complet d'une application J2EE/JEE. Au cours de notre parcours, nous nous appuyons sur une démarche centrée sur le modèle, ou MDA (Model Driven Architecture), et l'outil EclipseUML de l'éditeur Omondo.

L'ouvrage couvre également en profondeur la spécification EJB 3.0 et en détaille tous les avantages en comparaison de l'ancienne spécification EJB 2.x, en particulier grâce à l'API JPA (Java Persistence API).

À qui s'adresse l'ouvrage ?

Cet ouvrage est avant tout destiné aux développeurs Java/J2EE souhaitant faire leurs armes avec cette nouvelle spécification et disposant d'un bagage suffisant leur permettant d'intégrer des frameworks tels que JBoss Seam dans leurs projets.

Il sera également utile aux décideurs et aux architectes d'applications J2EE/JEE et d'entreprise désireux de faire le point sur les apports de la spécification EJB3 et des outils et sous-projets offerts par la nouvelle plate-forme Eclipse Europa, qui couvrent l'intégralité du cycle de développement JEE.

Organisation de l'ouvrage

L'ouvrage est structuré en trois grandes parties, plus une annexe.

Partie 1 : La plate-forme de développement Europa

- Le chapitre 1 détaille les nouveautés de la plate-forme Europa.
- Le chapitre 2 introduit le projet Eclipse RCP et présente les nombreux assistants proposés.
- Le chapitre 3 est consacré aux aspects liés à la gestion de configuration et du développement en équipe grâce à CVS et Subversion.
- Le chapitre 4 clôt cette partie par une présentation du profiling de code avec le projet Eclipse Test and Performance Tools Platform (TPTP).

Partie 2 : Développement Web avec le projet Eclipse Web Tools

- Le chapitre 5 est consacré à WTP (Web Tools Platform), un projet phare de la communauté Eclipse pour le développement J2EE/JEE, et à ses deux sous-projets : JST (J2EE Standard Tools) et WST (Web Standard Tools).
- Le chapitre 6 présente l'outillage du projet DTP (Data Tools Platfom), axé sur la manipulation des données.
- Le chapitre 7 introduit au développement Web avec l'outillage Web Tools et les patterns de conception.
- Le chapitre 8 clôt cette partie par une présentation détaillée du puissant framework Seam, la nouvelle plate-forme de développement fondée sur les annotations.

Partie 3 : Développement EJB3 avec Eclipse et Web Tools

- Le chapitre 9 décrit les principaux apports de la norme EJB 3.0 pour les projets JEE et détaille les évolutions de la notion de beans session ainsi que les annotions associées.
- Le chapitre 10 se penche sur l'évolution des beans entité avec l'API JPA et en montre toute la puissance pour la gestion de la persistance et du mapping objet-relationnel.
- Le chapitre 11 complète cette introduction à l'API de persistance avec le projet Dali, destinée à en faciliter la mise en œuvre.
- Le chapitre 12 présente l'atelier EclipseUML pour JEE, dont le but est de simplifier le développement JEE, et met en œuvre la démarche MDA au travers d'une étude de cas.
- Le chapitre 13 conclut l'ouvrage en revenant sur certaines fonctionnalités avancées du framework Seam, notamment la gestion du contexte et la notion d'intercepteurs permettant d'intégrer harmonieusement des EJB 3.0 avec la technologie JSF.

Partie 4 : Annexes

Page Web dédiée

Une page Web dédiée à l'ouvrage est accessible depuis le site des éditions Eyrolles, à l'adresse *http://www.editions-eyrolles.com.*

Lieu d'accueil et de rassemblement des lecteurs, cette page est traitée avec le plus grand soin afin de leur fournir tout le relais nécessaire grâce aux exemples de code complets et aux nombreux articles proposés sur les sujets traités dans l'ouvrage.

Vous pouvez aussi contacter l'auteur directement à son adresse professionnelle *(kdjaafar@jasmineconseil.com)* pour toute demande d'éclaircissement ou avis susceptibles d'aider à l'amélioration de l'ouvrage. Les avis pertinents seront relayés sur la page Web dédiée au livre.

Table des matières

Avant-Propos V

PARTIE I- LA PLATE-FORME DE DÉVELOPPEMENT EUROPA

CHAPITRE 1
Nouveautés d'Europa 3

Le projet Europa 3
Fonctionnalités Europa d'Eclipse 4
Les sous-projets Europa 9
Le projet WTP (Web Tools Platform) 10
TPTP (Test & Performance Tools Platform) 11
BIRT (Business Intelligence and Reporting Tools) 13
DTP (Data Tools Platform) 14
EMF (Eclipse Modeling Framework) 14
GEF (Graphical Editor Framework) 16
GMF (Graphical Modeling Framework) 16
Visual Editor 18
CDT (C/C++ Development Tools) 19
En résumé 19

CHAPITRE 2
Support du client riche avec Eclipse RCP 21

La notion de client riche 21
Architecture d'Eclipse RCP 22
Exemples d'applications RCP 24

Mise en œuvre de RCP 25
Développement d’une application RCP 25
Classes principales d’une application RCP 29
Ajout de fonctionnalités 32
En résumé 53

CHAPITRE 3 55

Support du développement en équipe sous Eclipse avec CVS 55

Développement collaboratif avec CVS 55
L’historique local d’Eclipse 56
Le travail en équipe avec Eclipse 57
L’architecture de gestion de configuration intégrée 57
Principes de gestion des ressources 58
Scénarios de développement en équipe dans Eclipse 59
Mise en œuvre de CVS dans Eclipse 61
Installation et configuration de CVSNT 62
Ajout d’utilisateurs avec pserver 65
Configuration du référentiel CVS dans Eclipse 65
Mise à jour des ressources à partir du menu Mettre à jour 72
Le gestionnaire de versions Subversion 75
Présentation 75
Concepts clés et nouvelles fonctions 76
Mise en œuvre 77
Création du référentiel Subversion 78
Démarrage du démon svn 78
Accéder au serveur svn dans Eclipse 83
Mise en œuvre d’un projet Eclipse avec Subclipse 85
En résumé 86

CHAPITRE 4 87

Profiling de code avec le projet TPTP 87

Architecture de TPTP 88
Composants de TPTP 89
Mise en œuvre de TPTP 90
Vues de profiling disponibles 91

Ouverture de la vue de profiling 94
Analyse des informations de profiling collectées 98
En résumé 100

Partie II - Développement Web avec le projet Eclipse Web Tools

CHAPITRE 5
Le projet WTP (Web Tools Platform) 103
Le sous-projet JST (J2EE Standard Tools) 104
Périmètre de JST 104
Architecture du sous-projet JST 105
L'outillage J2EE standard de JST 106
Outils de support à la création de projets J2EE 107
Outils de support aux serveurs J2EE 109
Le sous-projet WST (Web Standard Tools) 111
Support des facets projet 111
Support à l'outillage HTML, CSS et JavaScript 112
Support des Web Services 113
Mise en œuvre de WTP 114
Configuration de l'environnement d'exécution 115
Configuration des bibliothèques additionnelles spécifiques 117
Configuration d'un projet de développement Web 118
Propriétés du projet 123
Structure et contenu du projet 124
Artefacts du projet et vues JEE 125
En résumé 125

CHAPITRE 6 127
Conception du modèle de données avec le projet DTP (Data Tools Platform) 127
Architecture de DTP 128
Sous-projets de la plate-forme DTP 128
Mise en œuvre de DTP 130
Connexion à la base Apache Derby 130

Création d'un profil de connexion 133
Script SQL de création du schéma de base de données 135
Génération du DDL 137

En résumé 137

CHAPITRE 7

Développement Web avec l'outillage Web Tools et les patterns 139

Principes de base de l'exécution d'une requête JDBC à partir d'un programme Java 139

Design de l'application Web avec Web Tools 141
Création de la page d'accueil 142
Création des pages Login.jsp et LoginSuccess.jsp 145
Création de la servlet LoginServletTask 148

Gestion des commandes avec Web Tools et les patterns 152
Design de la partie métier 153
Création des classes d'accès aux données selon les design patterns Commande et Singleton 154
Création des composants servlets et JSP avec les assistants Web Tools. . . 157
Configuration de la source de données du serveur JBoss 162

Déploiement de l'application sur le serveur JBoss 163

En résumé 163

CHAPITRE 8

Développement Web avec le framework JBoss Seam 165

Les frameworks J2EE 165
Limitations de Struts et JSF 167

JSF (JavaServer Faces) 167
Mise en œuvre de JSF avec Eclipse Web Tools 168
Configuration de la cinématique des pages JSF avec l'éditeur de configuration Web Tools 174
Déploiement de la mini-application helloword sur JBoss 177

JBoss Seam 178
Caractéristiques du framework Seam 178
Mise en œuvre de Seam 181

En résumé 194

Partie III- Développement EJB3 avec Eclipse et Web Tools

CHAPITRE 9
Les fondamentaux EJB3 197
La spécification EJB3 197
D'EJB1 à EJB3. 197
Principales nouveautés d'EJB3 199
Introduction aux beans session 202
Quand utiliser des beans session ? 202
Beans session sans état 203
Beans session avec état. 209
Événements de callback des beans session 211
Beans session EJB2 vs EJB3 212
Les beans message (Message Driven Beans) 213
Mise en œuvre du développement EJB3 avec Web Tools et JBoss . . . 215
Prérequis et configuration. 215
Création et configuration du projet. 216
Test de l'EJB session 222
En résumé 224

CHAPITRE 10
L'API JPA et la gestion des entités 225
JPA (Java Persistence API) 225
Caractéristiques de JPA 227
Les beans entité 228
Annotations de persistance des beans entité. 228
Relations entre beans entité 234
Mapping des relations 235
Propriétés des relations et jointures 235
Utilisation des tables de jointure 241
L'interface Entity Manager 242
Types de gestionnaire d'entités 243
Méthodes de cycle de vie d'une entité EJB3 243
Obtention d'une fabrique EntityManagerFactory 245
Création d'un EntityManager. 246

Principales opérations disponibles sur les entités 246

Persistance d'une entité 246
Recherche d'entités et requêtes JPQL 247
Suppression d'une entité 249
Mise à jour d'une entité 249
Gestion des transactions JTA et non-JTA 250
Méthodes de callback 251

Invocation à partir d'un client Java et d'un client Web 252

Gestionnaire d'entités géré par l'application (client Java) 252
Gestionnaire d'entités géré par le conteneur (client Web) 254
Intégration de la couche de présentation 255

En résumé 258

CHAPITRE 11

Mapping JPA avec Dali 259

Le projet Dali 259

Scénarios de développement et configuration 260

Mise en œuvre de l'API JPA avec Dali 262

Création du projet JPA et connexion à une source de données 263
Création des entités persistantes du modèle et mapping vers la base 267
Création des mappings de base 273
Mapping des relations interentités 279
Mise en œuvre du mapping de version 283
Définition des requêtes nommées de l'interface Query 284

Intégration des entités du modèle logique et mise en œuvre d'un bean client façade 285

Entités Client et Commande 285
Le bean session CommandeManager 288

En résumé 290

CHAPITRE 12

Mise en œuvre de l'application webstock 291

L'architecture MDA 291

Méthodologie de mise en œuvre 292

EclipseUML for JEE 293
L'application webstock 295
Environnement de travail 295
Modélisation avec EclipseUML 295
Export XMI du modèle. 305
Nouveau projet JEE 306
Génération de code 308
En résumé 328

CHAPITRE 13
Configuration EJB3 avancée avec Seam 329
Les contextes Seam 329
Contexte conversationnel 332
Mise en œuvre de l'étude de cas avec Seam 332
Intégration de Seam dans l'architecture MVC 333
Configuration de Seam 334
Cas d'utilisation « Ajout d'un nouveau client » 335
Conversation Seam 340
En résumé 343

Annexe 345
Index 347

Remerciements

Écrire un ouvrage n'est jamais une entreprise solitaire, mais le fruit d'un travail de partage d'idées, d'expériences et de patience.

Je remercie l'équipe d'Omondo, en particulier Vlad Varnica pour ses encouragements et Amira Talbi pour sa contribution à la partie touchant à la modélisation EJB3 et à l'utilisation de l'outil de modélisation EclipseUML.

Je remercie aussi l'équipe d'Eyrolles, en premier lieu Éric Sulpice, qui a accepté l'idée de ce livre, et Olivier Salvatori pour ses précieux conseils et sa patience infinie lors des différentes relectures.

Enfin, un grand merci à mes proches, qui m'ont donné la force de conclure cette entreprise.

Partie I

La plate-forme de développement Europa

Cette première partie est entièrement consacrée à la plate-forme de développement Europa, nom de code de la version 3.3 d'Eclipse, qui englobe une dizaine de sous-projets remarquables, dont une grande partie sont détaillés dans cet ouvrage.

Le chapitre 1 passe en revue les nouveautés d'Europa.

Le chapitre 2 se penche sur le concept de client riche, une solution de rechange à celui de client léger et au Bureau Windows, que le projet Eclipse RCP (Rich Client Platform) met en exergue.

Le chapitre 3 aborde le développement coopératif à la lumière du système de contrôle Subversion, digne successeur de CVS, qui offre un grand nombre d'innovations par rapport à son aîné.

Le chapitre 4 clôt cette partie par une présentation du projet TPTP (Test and Performance Tools Platform), un projet Eclipse visant à fournir à la plate-forme Europa des services de mesures de performance et d'automatisation des tests.

1

Nouveautés d'Europa

La plate-forme Europa, nom de code d'Eclipse 3.3, représente une avancée importante dans la maturité de la plate-forme Eclipse. Elle est constituée d'une suite de sous-projets permettant d'organiser de manière ciblée et cohérente le développement Java.

Après un bref rappel des principales innovations de la plate-forme, nous présenterons chacun des sous-projets qui la composent, dont un certain nombre nous serviront au développement de l'étude de cas faisant office de fil conducteur à l'ouvrage.

Le projet Europa

Le projet Europa a été proposé au public en juillet 2007. Disponible pour les principaux environnements d'exploitation, comme Windows Vista ou Linux, cette nouvelle mouture d'Eclipse est à la fois plus performante (elle fonctionne sous forme d'exécutable et non plus à l'aide d'une JVM) et riche de nouveautés adressant les problématiques de développement suivantes :

- Développement d'applications Java, de plug-in Eclipse et d'applications client riche fondées sur le cœur du projet Eclipse initial, avec le JDT (Java Development Tools) et le PDE (Plug-In Development Environment).
- Développement d'applications Web et J2EE, avec le sous-projet Web Tools (Web and J2EE Development).
- Développement d'applications Web grâce à la création de rapports avec BIRT (Business Intelligence and Reporting Tools).
- Développement d'applications C/C++, avec le projet CDT pour Eclipse C/C++ Development Tooling (que nous ne traitons pas dans cet ouvrage).
- Développement d'applications centrées sur la gestion et l'accès aux données quelles qu'elles soient (bases de données, connecteurs, etc.), avec DTP (Data Tools Platform).
- Développement d'applications centrées sur la modélisation, avec EMF (Eclipse Modelling Framework) et GEF (Graphical Editing Framework).

- Profiling et test d'applications Java, avec TPTP (Test & Performance Tools Platform).
- Génération d'IHM pour Eclipse de type RAD, basées sur une implémentation Swing/ JFC ou SWT avec le projet Eclipse Visual Editor (VE).
- Fourniture d'une infrastructure de génération fondée sur EMF et GEF, permettant de créer un éditeur graphique avec le projet Eclipse Graphical Framework.

Les fonctionnalités associées à ces différents sous-projets sont illustrées à la figure 1.1.

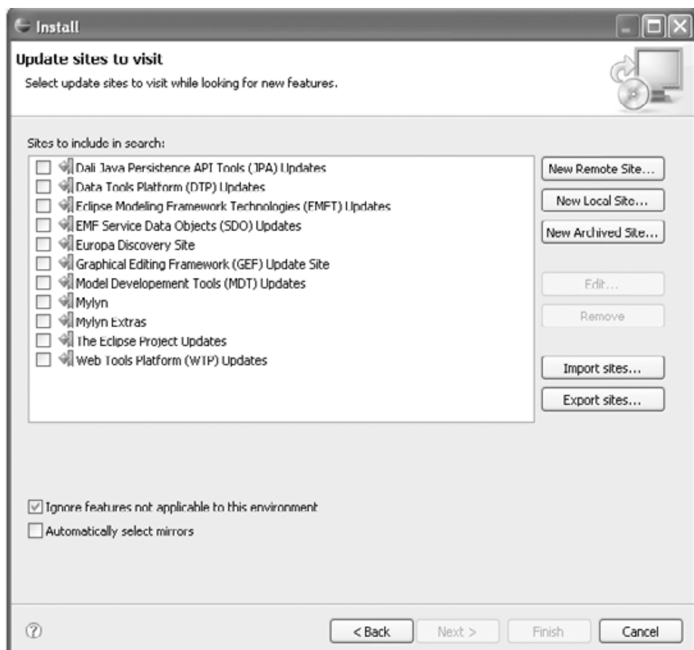


Figure 1.1

Sous-projets d'Europa

Chaque sous-projet nécessite l'installation préalable des fonctionnalités associées. Par exemple, dans le cas du projet Web Tools, il est nécessaire d'installer préalablement les fonctionnalités GEF et EMF (voir en annexe les procédures d'installation).

Fonctionnalités Europa d'Eclipse

La nouvelle release d'Eclipse apporte clairement de nombreuses nouveautés, qui justifient à elles seules la migration des versions 3.1 et 3.2.

Parmi les nombreuses fonctionnalités qui ont fait leur apparition dans la release 3.3, nous retiendrons celles qui nous semblent les plus utiles à la productivité des développeurs.

Nous conseillons cependant aux lecteurs de les explorer toutes au fur et à mesure de leur immersion dans le produit.

Explorateur de projets

L'effort important apporté à l'interface graphique saute aux yeux des habitués d'Eclipse 3.2. Une nouvelle vue, l'explorateur de projets, permet de mieux organiser les projets en fonction de leur particularité.

Par exemple, les projets Java apparaissent sous la forme d'une hiérarchie de répertoires source et de packages plutôt que comme de simples répertoires, tandis dans les projets de type Web apparaissent selon une arborescence respectant la spécification J2EE et maintenant JEE.

La figure 1.2 illustre une vue de l'explorateurs de projets appliquée à la gestion de ressources JEE.

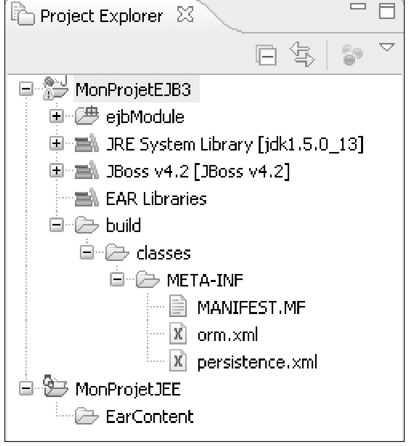


Figure 1.2

La vue explorateur de projets

Vue historique

Eclipse 3.2 apporte de puissantes fonctionnalités d'historique, permettant de combiner les modifications locales avec celles présentes dans le référentiel projet.

La figure 1.3 illustre cette possibilité. Il est possible de revenir à la révision d'un source et de comparer la version courante avec la version historisée.

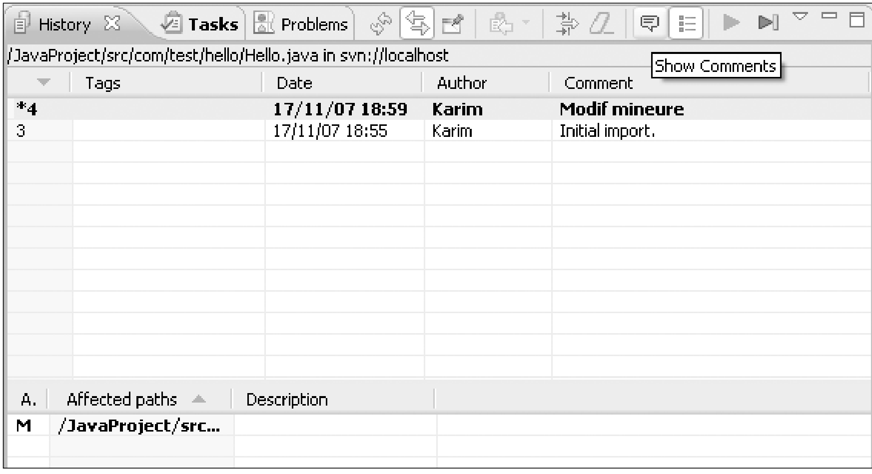


Figure 1.3

Vue historique (History)

Working Sets

Lorsque vous avez de nombreux projets dans un espace de travail, ou workspace, Eclipse, la vue Package Explorer devient vite surchargée. Il est désormais possible de définir ce qu'on appelle des Working Sets, ou jeux de documents, afin de restreindre les ressources affichées et de limiter ainsi le champ d'exécution de certaines opérations, comme la recherche.

Dans un projet multicouche, comportant des centaines de classes, il est de la sorte possible de n'afficher à un instant *t* qu'un sous-ensemble de fichiers. Typiquement, lorsque vous travaillez sur un cas d'utilisation, vous pouvez vouloir n'afficher que sa DAO, ses services et son IHM.

La figure 1.4 illustre un Working Set défini pour l'affichage de ressources de type EJB.

Figure 1.4

Sélection d'un Working Set

Vue Filtres

La vue Filtres permet de regrouper les erreurs par type (incidents Java, incidents relatifs à la configuration du projet, incidents de type PDE, etc.) et par gravité.

La figure 1.5 illustre la configuration du filtre d'affichage des erreurs.

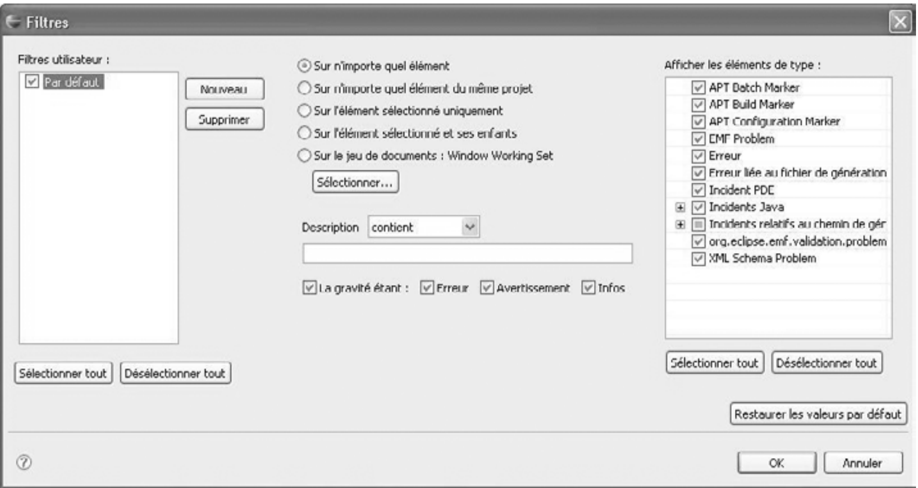


Figure 1.5

Vue Filtres

La vue Cheat Sheet

Puissante et conviviale, la vue Cheat Sheet permet de garder ouverte une fenêtre d'aide en ligne pendant que vous êtes en train de travailler avec un assistant Eclipse.

La figure 1.6 illustre l'aide à la création d'une première application Java sous Eclipse.

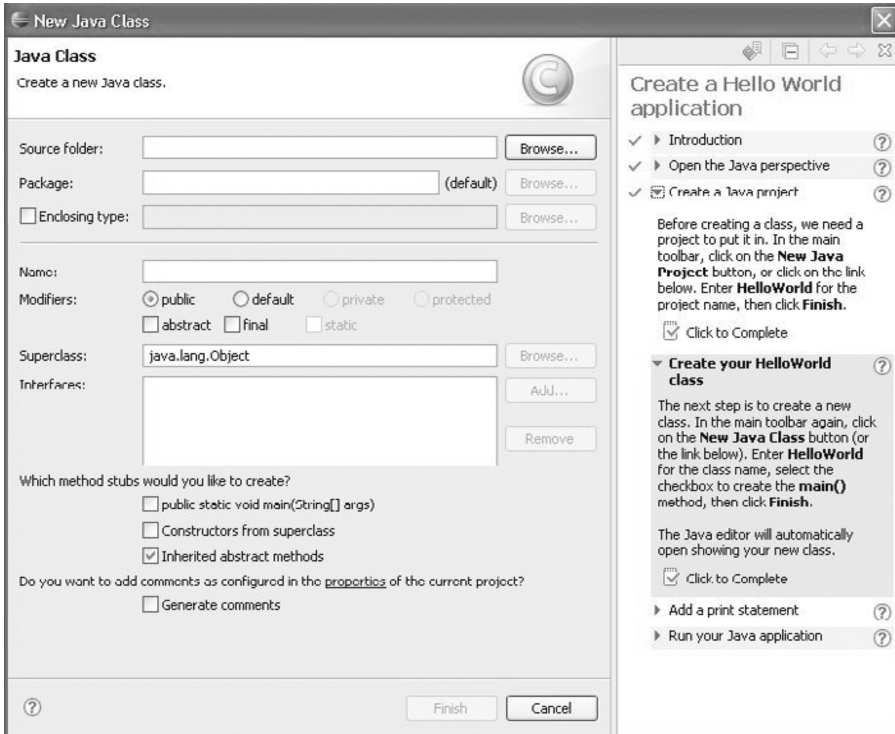


Figure 1.6

Vue Cheat Sheet

Import de projets avec copie

Lors de l'import d'un projet existant, une case à cocher vous permet de demander à Eclipse de copier le répertoire du projet importé dans le répertoire du workspace.

La case n'est pas cochée par défaut, car il n'est pas obligatoire de stocker les projets dans le répertoire du workspace.

Liens vers des ressources fichier

Il est possible d'utiliser un lien vers une ressource projet sans avoir à copier cette ressource dans son workspace, afin de ne pas surcharger ce dernier.

L'intégration des flux RSS

La section Dernières nouvelles de la fenêtre de bienvenue intègre des flux RSS (nécessitant une connexion Internet) permettant de profiter au mieux des informations les plus récentes en provenance de la fondation Eclipse.

La figure 1.7 illustre une vue de la section Latest News.

Figure 1.7

Exemple de support des flux RSS

Nettoyage de code

L'option Nettoyage (menus Source/Nettoyage) est une fonctionnalité puissante de productivité du code, qui permet de demander à Eclipse de nettoyer le code en appliquant des règles configurables *via* la boîte de dialogue illustrée à la figure 1.8.

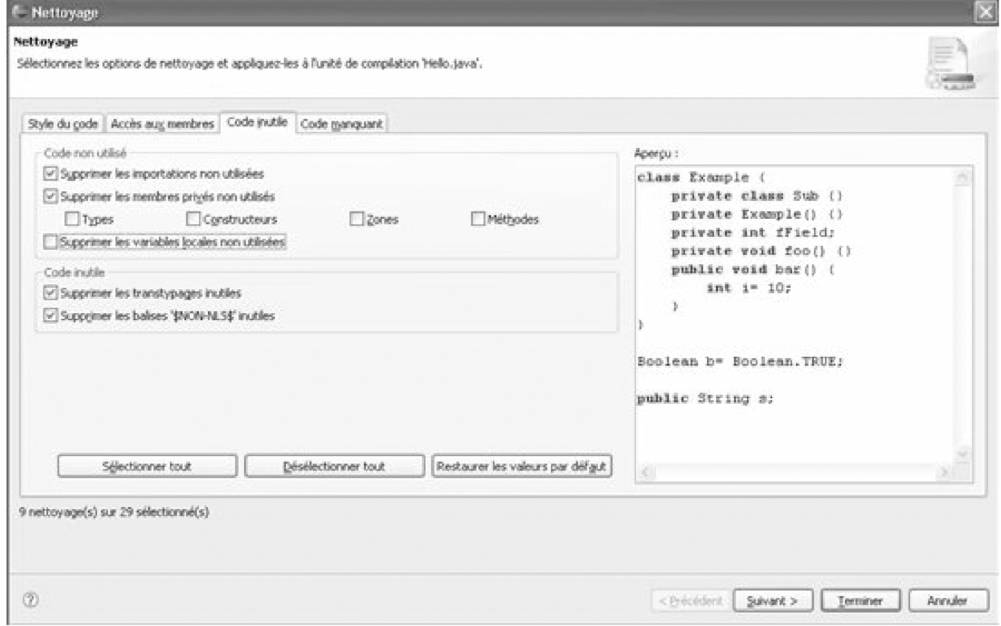


Figure 1.8

Vue Nettoyage

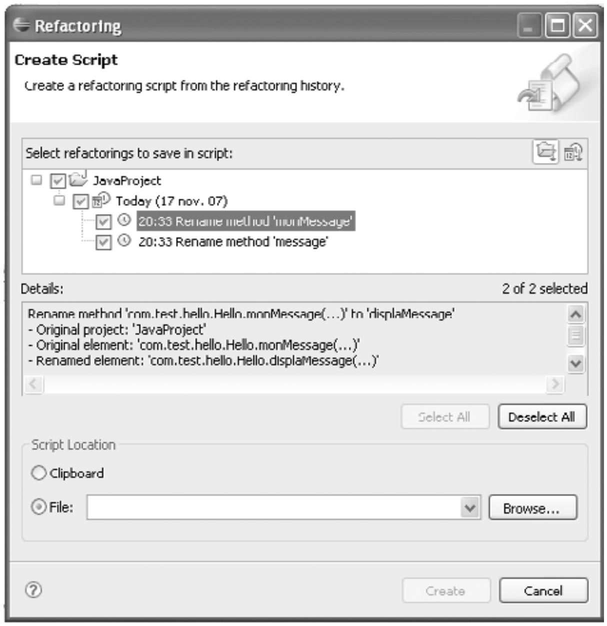
Restructuration de scripts

Europa renforce les fonctionnalités de refactoring de code déjà très avancées de la version 3.1 d'Eclipse. Il est désormais possible de créer des scripts avec un meilleur contrôle et un historique des modifications et de la réorganisation du code.

La figure 1.9 illustre la vue Restructuration disponible dans Europa.

Figure 1.9

Vue Restructuration



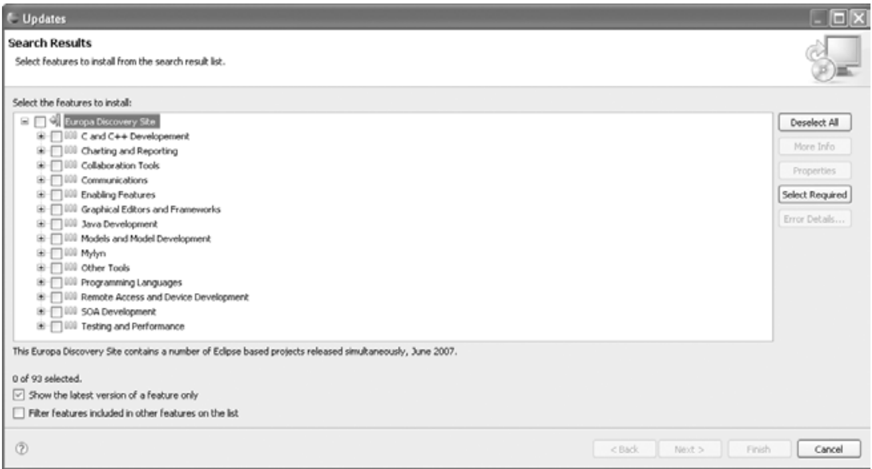
Les sous-projets Europa

Les sous-projets qui constituent Europa sont le fruit de la collaboration de quelque trois cents développeurs à travers plus d'une vingtaine de pays.

Europa ne consiste pas en une intégration totale de ces sous-projets. Chacun d'eux reste un projet Open Source autonome, avec un chef de projet, des développeurs et un planning propres.

Figure 1.10

Sous-projets Eclipse Europa



Parmi ces sous-projets, dont plus d'une vingtaine sont illustrés à la figure 1.10, certains sont historiques à la plate forme Eclipse, comme EMF, JDT et Equinoxe, et couvrent des domaines du développement très variés, en particulier les suivants :

- Business intelligence et reporting avancé avec le projet BIRT.

- Frameworks favorisant l'extensibilité de la plate-forme autour de SOA avec le projet SOA Tool Platform *(http://www.eclipse.org/stp/)* et de nombreux sous-projets associés.
- Fonctionnalités avancées de collaboration en ligne fondées sur les tâches avec le projet Mylyn (anciennement Mylar).
- Nouvel IDE pour le développement avec le langage Ruby dans le projet DLTK (Dynamic Languages Toolkit).
- Projets de support au développement JEE, comme WTP (Web Tools Platform) et Dali.

Le projet WTP (Web Tools Platform)

Apparu le 23 décembre 2005, le projet Web Tools *(http://www.eclipse.org/Web Tools)* visait principalement à répondre à la demande de nombreux développeurs de disposer en standard d'outils avancés pour le support d'applications J2EE avancées.

Ni la plate-forme Eclipse, ni l'outillage Java proposé par défaut (le JDT) ne comportaient d'éditeurs JSP ou XML, d'assistants EJB ou de mécanismes de déploiement sur les principaux serveurs d'applications J2EE compatibles. Cette lacune a été comblée par différents plug-ins commerciaux, tel MyEclipse, ou Open Source, tel Lomboz. Ces solutions imposaient cependant des choix d'environnement particuliers, qui constituaient autant de freins à l'adoption d'un standard commun.

Les principaux objectifs de WTP (Web Tools Platform) sont les suivants :

- étendre la plate-forme Eclipse à l'aide d'outils dédiés afin de couvrir les développements Web et J2EE, et bientôt JEE 5 avec WTP 2.0 ;
- proposer des outils de support à des ressources de type HTML et XML, des services Web, des données, etc. ;
- inclure des outils de support au déploiement d'applications sur les principaux serveurs d'applications du marché (Tomcat, JBoss, Geronimo, WebSphere CE, etc.).

Largement couvert dans cet ouvrage, ce projet, dont une grande part du code est issue de la contribution d'IBM et d'ObjectWeb, est constitué de deux sous projets, WST et JST, et d'un projet « en incubation », JSF :

- WST (Web Standard Tools). Ce sous-projet touche à plusieurs périmètres, notamment le support à l'édition de langages Web (éditeur de code source HTML, XML, XMLSchema, DTD, etc.), le support de serveurs J2EE/JEE et le support des Web Services (éditeur de fichiers WSDL et explorateur de services en particulier). Il fournit des fonctionnalités spécifiques, appelées Project Facets, selon le type de projet supporté. Par exemple une facet Dynamic Web Project est proposée dans l'explorateur de projets de WST.
- JST (J2EE Standard Tools). Ce sous-projet correspond à l'environnement J2EE/JEE. Son objectif est de fournir des outils et des API standards spécifiquement dédiés à la modélisation et au support des spécifications émises par le JCA (Java Community Process) pour les serveurs d'applications et Web utilisant J2SE et J2EE/JEE. La release courante de WTP supporte à cette date les versions J2EE 1.2, 1.3 et 1.4 (le support JEE5, en particulier des assistants de création de ressources EJB3, n'est pas encore natif). Ce projet permet la prise en compte des formats war et ear des projets et un support au développement de servlets, JSP et EJB. Une vue Servers permet notamment de piloter et de configurer les principaux serveurs d'applications du marché.

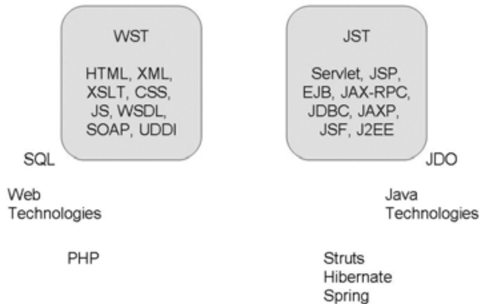
- JSF Tools (JavaServer Faces Tools). Ce projet propose un outillage complémentaire pour le développement d'applications utilisant les JavaServer Faces et une partie des spécifications de la norme JEE5 (JSF 1.2). Lancé fin 2005 par Oracle, ce projet apporte des assistants de configuration de projet pour le support des JSF et des tags JSF dans des pages JSP et l'ajout d'éditeurs spécifiques pour les fichiers descripteurs faces-config.xml permettant une définition graphique de la cinématique de navigation (à l'image des éditeurs dans le framework Struts). Ce sous-projet a été intégré en juin 2006 a la release Callisto et à WTP 1.5, la version WTP 2.0 ayant été quand a elle « bundlée » à la release officielle Europa à partir de juin 2007.

Nous reviendrons en détail sur ce projet important, abondamment mis à contribution dans l'étude de cas de l'ouvrage.

La figure 1.11 illustre la cible et les technologies sous-jacentes de ces différents sous projets.

Figure 1.11

Cibles couvertes par le projet Web Tools



En plus du projet JSF Tools, deux nouveaux sous-projets ont fait leur apparition courant 2006 :

- Dali JPA (Java Persistence API) Tools. Ce projet propose des outils facilitant le mapping objet-relationnel, la persistance et le développement EJB3. Ce projet sera amplement discuté et mis en œuvre au chapitre 10. La version actuelle, Dali JPA Tools 1.0, intégrée à la release Europa propose des fonctionnalités intéressantes pour la génération du fichier descripteur de persistance (persistence.xml), la configuration évoluée d'un projet Java et l'aide aux annotations prévues par la spécification JPA. Dali propose en outre de puissants outils d'ORM, la possibilité de visualiser les champs persistants et les relations ainsi que le mapping de l'élément sélectionné dans l'éditeur de code source.
- ATF (Ajax Toolkit Framework). Créé début 2006, ce projet rattaché à WST a pour objectif la livraison de frameworks et d'outils de support aux différentes versions de runtime Ajax existantes.

Comme nous le verrons tout au long de l'ouvrage, ce projet très dynamique risque fort de devenir l'outil de référence des développements J2EE/JEE 5.

TPTP (Test & Performance Tools Platform)

La mise en œuvre de tests et l'analyse des performances des applications sont des phases essentielles dans le cycle de vie des projets, qui nécessitent des outils appropriés.

C'est dans ce but que le projet TPTP a vu le jour en 2002, d'abord sous le nom de Hyades avant de devenir en 2004 un des projets principaux de Callisto puis d'Europa.

TPTP est organisé en quatre sous-projets :

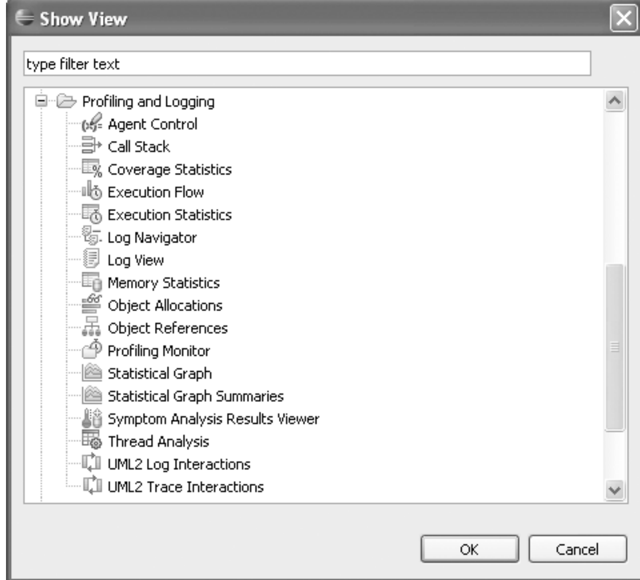
- TPTP Platform Project, qui forme l'ossature de la solution utilisée par les différents outils de test et de mesure de performances. Il fournit les mécanismes permettant de collecter les métriques et de les agréger.
- TPTP Tracing and Profiling Tools Project, qui intègre les outils permettant la collecte des informations sur le fonctionnement d'applications Java s'exécutant ou non sur un serveur J2EE compatible et l'analyse de ces informations (localisation des zones mémoire allouées, détermination des temps d'exécution, etc.). Ces outils s'appuient sur une analyse dynamique de l'application. Cette dernière est exécutée dans un mode particulier, appelé Profiling, qui permet la collecte d'informations sur les temps d'exécution et l'occupation mémoire. Pendant l'exécution ou une fois celle-ci terminée, plusieurs vues permettent d'analyser les résultats.
- TPTP Test Tools Project, qui propose des outils permettant la création de scénarios de tests. Différents types de projets de test sont proposés, tels que les tests automatisés avec JUnit, les tests Web (capture et enregistrement des requêtes HTTP) et les tests nécessitant l'intervention de l'utilisateur.
- TPTP Monitoring Tools Project, qui fournit des outils de collecte et d'affichage des performances de l'application ainsi que du système.

La page officielle du projet TPTP est disponible à l'adresse *http://www.eclipse.org/tptp/*. La version utilisée au chapitre 4 de l'ouvrage est la 4.2.

La figure 1.12 illustre une vue des différentes métriques disponibles dans l'outil TPTP.

Figure 1.12

Métriques disponibles dans TPTP



BIRT (Business Intelligence and Reporting Tools)

L'ambition de ce projet est d'apporter des outils de génération d'états, une solution puissante Open Source pour la création de rapports pour les applications Web en environnement Eclipse.

Les deux composantes de BIRT sont un outil de conception de rapports et un moteur d'exécution installable dans un serveur d'applications J2EE.

BIRT fournit des outils pour la prise en compte des quatre grandes étapes de la création d'un rapport :

1. Récupération des données. Des assistants permettent de définir la façon de récupérer les données à utiliser dans le rapport. Les données peuvent provenir de bases de données relationnelles, de fichiers XML, de services Web ou directement d'objets Java.
2. Transformation des données. Il est courant qu'un rapport n'affiche pas directement les données récupérées mais que des transformations soient nécessaires, telles que tri, regroupement, calcul de sous-totaux, de pourcentages, etc.
3. Ajout de règles métier. La structuration du rapport et les valeurs de certains champs peuvent dépendre de règles plus ou moins complexes. BIRT permet soit d'appeler la logique Java, soit, pour les cas les plus simples, d'embarquer directement dans le rapport du code JavaScript.
4. Mise en forme. Une fois les données récupérées et transformées, reste à gérer la présentation du rapport. BIRT propose un constructeur graphique de rapport, qui permet de présenter les données de diverses façons : tableaux, graphiques, texte formaté, etc.

Comme l'illustre la figure 1.13, BIRT 2.0 permet de générer des rapports aux formats HTML et PDF. L'export des données au format CSV est aussi supporté. Pour en savoir plus sur ce projet très actif, reportez-vous à la page officielle du site *(http://www.eclipse.org/birt/).*

CLASSIC MODELS

Classic Models, Inc.
701 Gateway Boulevard
South San Francisco, CA 9408

Annual Sales By Product Lines

Year		Classic Cars	Motorcycles	Planes	Ships	Trains	Trucks & Buses	Vintage Cars	Total
2003									
	Q1	$152,681	$33,062	$37,136	$24,446	$7,810	$43,593	$106,982	$405,61
	Q2	$194,291	$41,629	$60,387	$60,171	$12,144	$62,804	$81,959	$493,38
	Q3	$243,978	$79,845	$45,047	$24,272	$7,027	$73,842	$110,486	$584,60
	Q4	$778,636	$194,372	$168,261	$102,163	$38,839	$196,416	$311,111	$1,789,678
	Total	$1,369,386	$348,909	$300,822	$201,044	$65,822	$376,657	$610,539	$3,273,181
2004									
	Q1	$317,306	$85,681	$65,159	$66,762	$21,028	$67,942	$136,849	$760,730
	Q2	$252,165	$80,101	$69,780	$30,719	$4,862	$73,696	$119,997	$631,322
	Q3	$419,674	$127,310	$105,973	$66,859	$21,728	$106,833	$200,420	$1,048,801
	Q4	$739,276	$234,150	$204,551	$141,609	$39,278	$200,230	$366,660	$1,925,757
	Total	$1,728,423	$527,243	$445,464	$305,951	$86,897	$448,702	$823,927	$4,366,611
2005									
	Q1	$392,920	$112,384	$103,702	$60,591	$18,235	$101,332	$183,041	$972,208
	Q2	$225,273	$128,874	$35,870	$20,812	$4,076	$41,874	$60,113	$516,894
	Total	$618,193	$241,259	$139,573	$81,404	$22,311	$143,207	$243,156	$1,489,103
Grand Total		$3,716,003	$1,117,412	$885,859	$588,400	$175,030	$968,566	$1,677,622	$9,128,896

Figure 1.13

Vue des possibilités de l'outil de reporting BIRT (total des chiffres de vente par trimestre et par produit)

DTP (Data Tools Platform)

Le projet DTP regroupe un ensemble d'outils permettant de se connecter aux bases de données relationnelles.

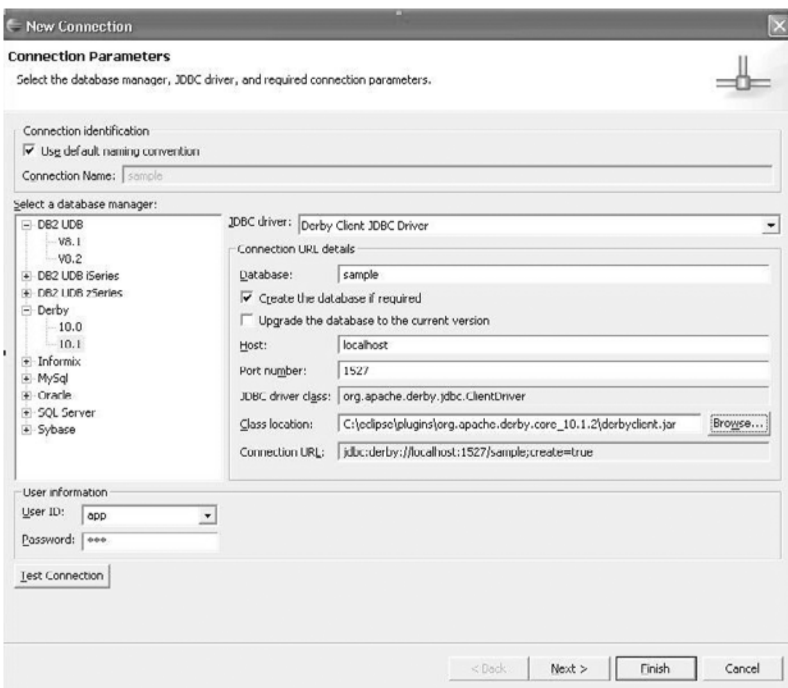
Les principales fonctionnalités qu'il propose sont les suivantes :

- assistants pour définir les informations de connexion JDBC à une base de données relationnelle ;
- explorateur de bases de données (liste des tables, structure des tables, etc.) ;
- éditeurs SQL de deux types : éditeur de code source et éditeur visuel offrant une représentation graphique d'une requête ;
- testeur de requêtes SQL avec visualisation du résultat.

La figure 1.14 illustre la vue associée à la configuration d'une connexion à la base de donnée Derby.

Figure 1.14

Vue de l'assistant de configuration de connexion DTP



EMF (Eclipse Modeling Framework)

Apparu en 2002, le projet EMF *(www.eclipse.org/emf)* permet de décrire un modèle objet puis d'écrire ou de générer le code permettant de le manipuler.

Ce projet très utilisé sert de base à d'autres projets, tels que GEF et GMF (voir plus loin). Relativement riche et complexe, il se présente comme un ensemble d'outils de développement intégrés à l'environnement Eclipse sous forme de plug-in afin d'ouvrir Eclipse au développement « piloté » par les modèles.

Son objectif est de permettre d'appliquer cette approche aussi bien au développement d'applications métier qu'à la construction et à l'intégration de nouveaux plug-ins.

EMF apporte les deux fonctionnalités principales suivantes :

- Définition de modèles objet et de diagrammes UML constitués de packages de classes et de liens entre ces classes. Ces modèle peuvent être élaborés à partir de code Java annoté, de documents XMI issus d'autres modeleurs (par exemple, Omondo est en grande partie nativement intégré à EMF), d'un éditeur arborescent intégré à EMF ou par programmation.
- Transformation d'un modèle EMF en code Java et génération d'un éditeur arborescent spécifique pour construire des entités conformes à ce modèle.

La structure des modèles EMF est définie par le métamodèle Ecore. Ce dernier est conceptuellement proche de MOF (MetaModel Facility), un standard de l'OMG adressant la représentation des métamodèles et leur manipulation.

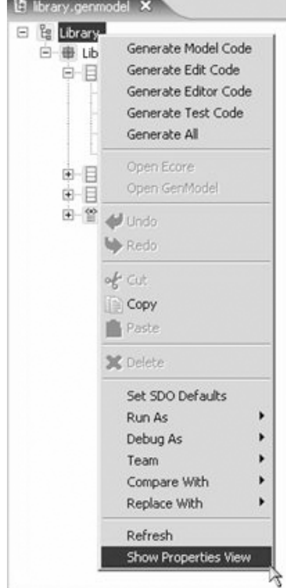
L'une des caractéristiques parmi les plus puissantes de ce projet est d'offrir un cadre et un support à la métamodélisation, qui consiste à spécifier un métamodèle sous forme de modèle EMF et à en produire une implémentation Java afin de représenter les instances de ce métamodèle et de les manipuler.

Eclipse/Europa propose des assistants intégrés permettant la création de modèles EMF ainsi que la génération du code associé au modèle et, bien entendu, des éditeurs adaptés à leur manipulation.

La figure 1.15 illustre un aperçu des outils intégrés de support à la transformation de modèles MOF Eclipse (appelés eCore, qui est l'implémentation Eclipse de la spécification MOF).

Figure 1.15

Outils intégrés de support au projet EMF



EMF est un projet très important Callisto et d'Europa, qui mérite un examen attentif tant ses possibilités sont nombreuses et puissantes.

GEF (Graphical Editor Framework)

Le projet GEF est un environnement pour le développement d'éditeurs graphiques fondés sur l'API Draw2D, une surcouche de la bibliothèque graphique SWT intégrée à Eclipse, qui propose des fonctions avancées de dessin et de calques (Layout).

Pour la petite histoire, GEF (comme Eclipse) a été légué par IBM à la fondation Eclipse. Il était auparavant supporté par les outils de modélisation de Rational.

GEF se fonde sur une architecture de type MVC (modèle, vue, contrôleur). Il propose une boîte à outils permettant de visualiser les graphes mis à la disposition du développeur, ainsi qu'un certain nombre de fonctions de base qu'il est possible d'étendre. Il reste toutefois à la charge de ce dernier de définir une sémantique graphique permettant de faire le lien entre la visualisation et le modèle EMF correspondant et à écrire une partie du code Java permettant d'implémenter l'interface graphique à partir des classes produites par EMF et GEF.

La figure 1.16 illustre un afficheur de diagrammes UML conçu avec le framework GEF.

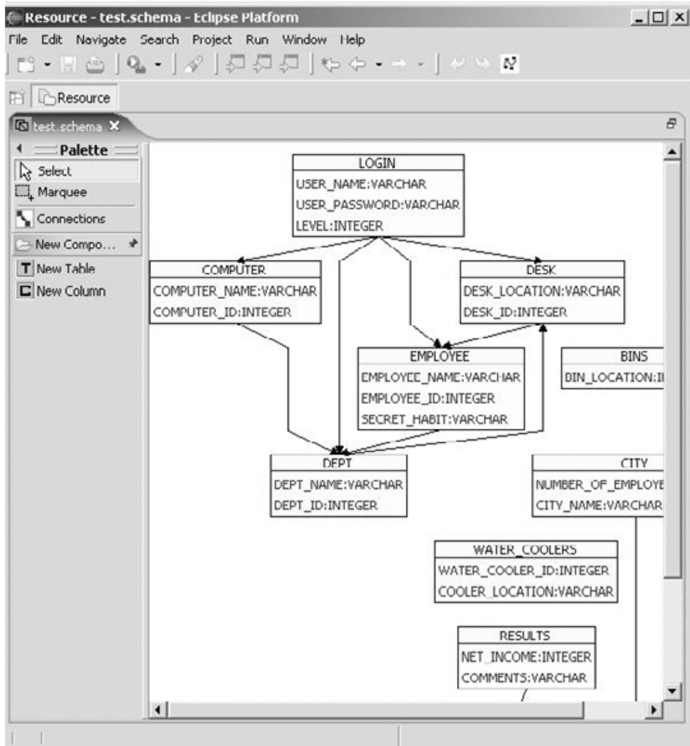


Figure 1.16

Afficheur de diagrammes UML de GEF

Pour plus de détails sur ce projet, reportez-vous à la documentation disponible sur le site du projet, à l'adresse *http://www.eclipse.org/gef/.*

GMF (Graphical Modeling Framework)

Apparu historiquement après GEF et EMF (toujours en legs d'IBM à la fondation Eclipse), GMF permet de relier les deux frameworks, raison pour laquelle il est souvent

assimilé à un environnement RAD pour la construction et le développement d'éditeurs fondés sur EMF/GEF.

L'utilisation du framework GEF est assez ardue. Elle nécessite une phase d'apprentissage et de maîtrise de ses API relativement fastidieuse, en partie parce que GEF n'est pas « couplé » au modèle UML sous-jacent, ce qui peut engendrer un certain nombre de problèmes. GMF est fondé sur un framework piloté par les modèles (MDA).

La cinématique d'utilisation du framework et de ses principaux composants telle qu'illustrée à la figure 1.17 est proposée lors de la création d'un projet GMF *via* l'assistant de création de nouveau projet d'Eclipse.

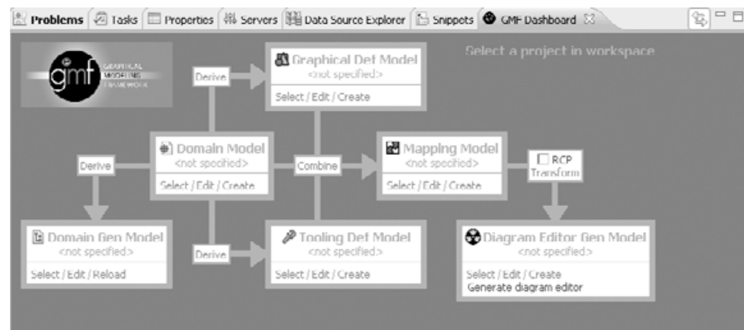


Figure 1.17

Cinématique d'utilisation du framework GMF sous Eclipse

Comme vous pouvez le voir, le cœur du framework GMF s'appuie sur une définition graphique du modèle que l'on souhaite générer. En option, il est possible d'utiliser des outils et des éléments de la palette graphique (menus, barre d'outils, etc.).

Une fois le mapping entre le modèle sélectionné et l'information graphique associée effectué, GMF fournit un générateur de modèles (étape Create Generator Model). Ce dernier permet d'implémenter les détails du modèle permettant d'aboutir à la génération finale sous la forme d'un plug-in directement utilisable dans Eclipse *via* les menus Fichier, Nouveau, Exemples et Model Diagram.

La figure 1.18 illustre un exemple d'éditeur UML conçu avec le framework GMF.

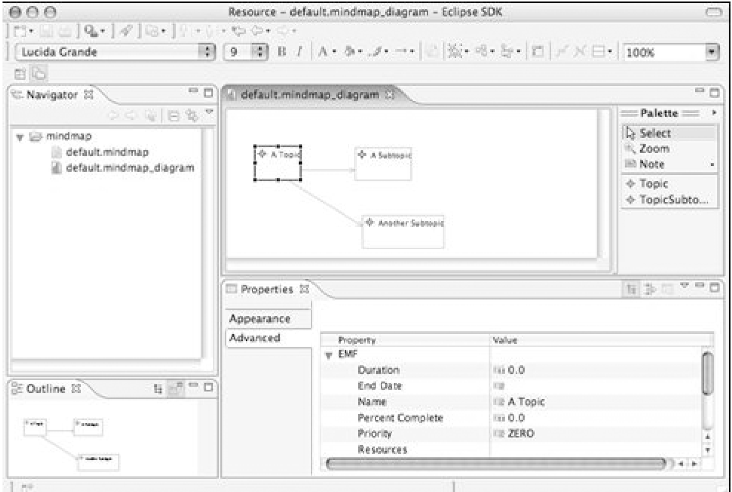


Figure 1.18

Éditeur UML conçu avec le framework GMF

Précisons que ce framework, tout comme les frameworks EMF/GEF, constitue un prérequis à l'installation de Web Tools. Il est d'ailleurs disponible en version bundle sur le site d'IBM alphaworks, à l'adresse *http://www-128.ibm.com/developerworks/eclipse/downloads/.*

Visual Editor

Visual Editor est un plug-in de type RAD pour le développement d'interfaces SWING ou AWT sous Eclipse. Il s'agit d'un environnement complet, comportant une riche palette permettant le développement assisté d'interfaces graphiques évoluées.

La figure 1.19 illustre une vue de l'atelier VE avec un aperçu de la boîte à outils disponible.

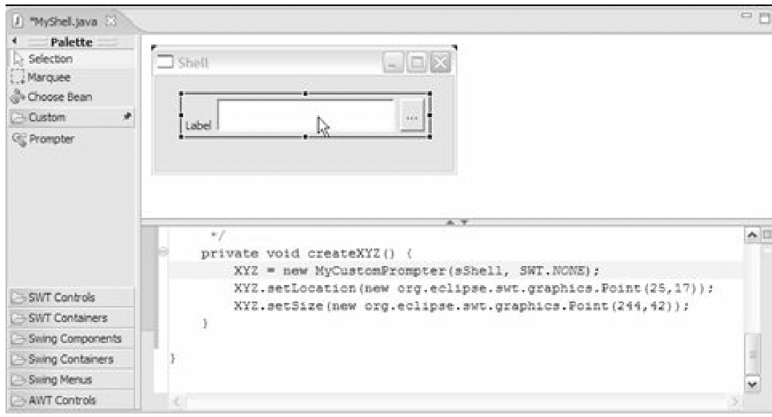


Figure 1.19

Exemple d'éditeur UML conçu avec le plug-in Visual Editor du framework GMF

Le plug-in Visual Editor permet de créer un nouveau type de Visual Class, ainsi que d'utiliser une palette graphique. Visual Editor n'est pas seulement utile pour modéliser rapidement une interface. Il est aussi très performant pour gérer les événements sur divers éléments graphiques.

La figure 1.20 illustre une liste d'événements supportés par le composant de type « bouton » de l'atelierVisual Editor.

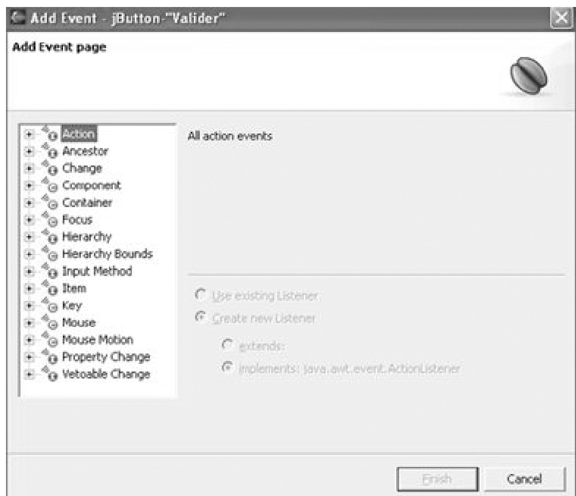


Figure 1.20

Événements supportés par Visual Editor

CDT (C/C++ Development Tools)

Le projet Eclipse CDT (C/C++ Development Tools) s'installe sur la plate-forme Eclipse et l'étend en fournissant des fonctionnalités avancées pour les développeurs C/C++.

CDT inclut les fonctionnalités suivantes :

- éditeur C/C++ fournissant des fonctionnalités élémentaires, comme une syntaxe coloriée, la complétion de code, etc. ;
- debogueur C/C++ ;
- lanceur d'applications C/C++ ;
- compilateur ;
- moteur de recherche ;
- générateur de makefile.

La figure 1.21 illustre une vue de l'atelier de développement CDT sous Eclipse.

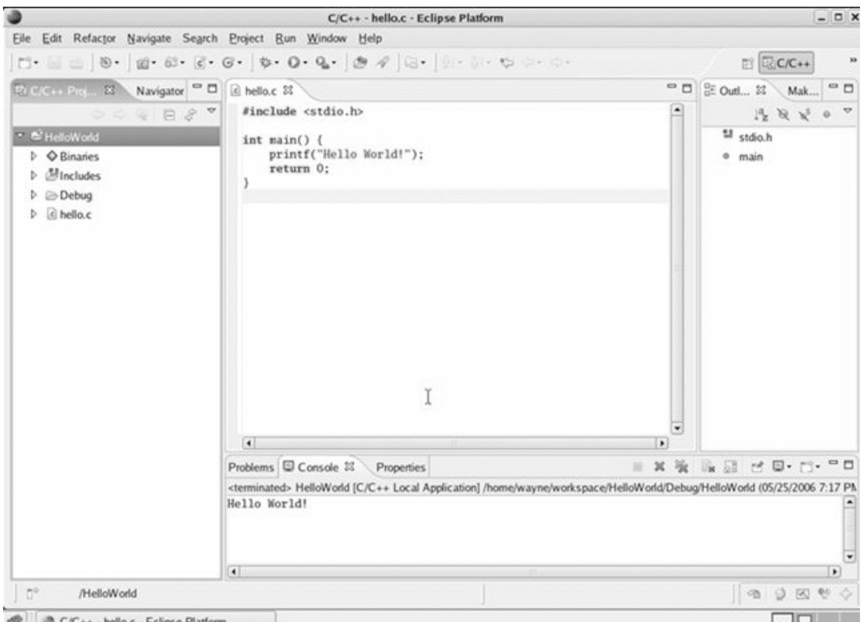


Figure 1.21

Vue de l'atelier de développement CDT sous Eclipse

En résumé

Europa représente une étape importante dans l'histoire d'Eclipse et dans la fourniture d'outils pour le support au développement d'applications de type client riche, Internet riche et serveur.

2

Support du client riche avec Eclipse RCP

Le fort engouement pour les architectures Web 2.0 remet sur le devant de la scène la notion de *client riche,* que l'on retrouve dans une profusion de solutions destinées à donner plus d'interactivité aux applications Web.

Le framework Eclipse RCP (Rich Client Platform) se veut une solution de rechange crédible au bureau Windows. Vous mesurerez dans ce chapitre toute la richesse des outils fournis au développeur pour simplifier son travail.

Après une présentation des bases de la création d'une application RCP et une prise de contact avec les principales classes et interfaces associées, nous détaillerons la création de menus, barres d'outils, vues, éditeurs et boîtes de dialogue, ainsi que l'utilisation des pages de préférences. Nous terminerons le chapitre par le packaging du produit et l'intégration des différents jar.

Le terme workbench, ou Bureau Eclipse, sera utilisé tout au long de ce chapitre.

La notion de client riche

Bien que les applications intranet de type client léger soient largement répandues dans les entreprises, elles ne peuvent couvrir l'ensemble des besoins. Certaines applications présentent des contraintes d'ergonomie, de rapidité, de complexité ou encore d'intégration aux outils bureautiques, que le client léger ne peut satisfaire.

Le concept d'application riche est apparu au cours des années 1990 pour les deux grandes familles d'applications suivantes :

- **Applications Internet riches, ou RIA (Rich Internet Applications)**, qui s'appuient sur l'utilisation d'un navigateur côté client et exploitent soit les fonctionnalités évoluées des navigateurs, soit des extensions. Les applications AJAX telles que Google Maps, Yahoo! Mail, etc., entrent dans cette catégorie. Leur généralisation en entreprise paraît toutefois délicate, du fait de problèmes de productivité et maintenabilité du JavaScript

et du DHTML ainsi que de portabilité entre navigateurs, de limitations imposées par ces derniers en termes d'intégration avec les autres applications et de manque de performances des interfaces graphiques complexes.

- **Applications riches fondées sur un socle applicatif** à installer sur le poste utilisateur. Eclipse RCP appartient à cette catégorie, qui représente une évolution des architectures client/serveur de première et deuxième générations des années 1990. Eclipse RCP apporte des solutions aux deux problèmes principaux de ces architectures, qui sont la distribution de l'application sur les postes utilisateur et leur forte dépendance vis-à-vis des technologies propriétaires. Si ces applications n'atteignent pas la facilité de distribution du client léger, elles atténuent les problématiques d'installation initiale et de gestion des mises à jours et offrent des avantages évidents en matière de réactivé, de qualité des interfaces graphiques, de souplesse (avec le mode déconnecté) et d'intégration poussée aux autres applications installées sur le poste.

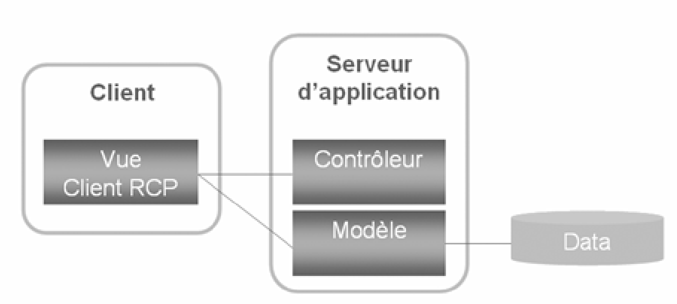
Pour satisfaire les applications d'aujourd'hui, un client riche doit posséder les mécanismes et caractéristiques suivantes :

- interface graphique évoluée ;
- appel à des services métier s'exécutant sur un serveur d'applications ;
- plate-forme indépendante du système d'exploitation hôte ;
- framework de composants standards réutilisables ;
- mécanisme intégré et évolué de mise à jour des composants ;
- extensibilité.

La figure 2.1 illustre un exemple d'architecture « riche ».

Figure 2.1

Exemple d'architecture riche



Architecture d'Eclipse RCP

Eclipse RCP est apparu en 2004 afin d'apporter un support du client riche aux applications fondées sur Eclipse. Une petite portion du noyau d'Eclipse (moins de 5 Mo) est utilisée pour assurer ce support en expurgeant la partie propre au support d'un environnement de développement. Il comprend une technologie de déploiement et d'installation de clients riches sous forme de plug-ins, ainsi qu'une panoplie de plug-ins génériques.

Principal contributeur du projet, IBM l'a placé au cœur de son Workplace Client Technology Rich Edition. Il y a ajouté des composants graphiques (le standard Widgets Toolkit) et une fonction de synchronisation avec une base locale.

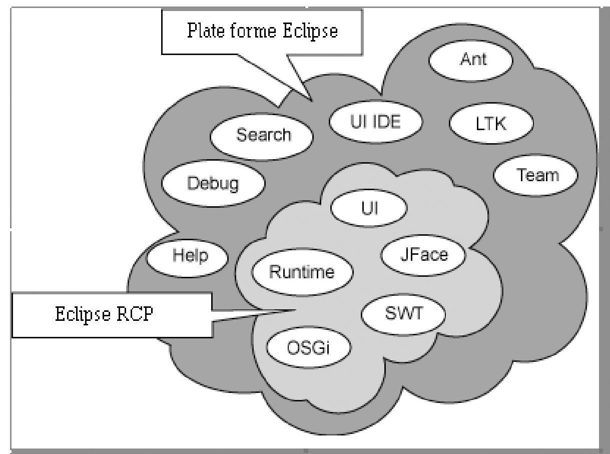


Figure 2.2

Composants d'Eclipse RCP

Comme l'illustre la figure 2.2, Eclipse RCP est constitué des cinq composants principaux suivants :

- Environnement standard d'exécution OSGI (Open Services Gateway Initiative).
- Runtime Eclipse, qui forme le socle de chargement et d'exécution des services Eclipse.
- Workbench générique Eclipse (UI), qui constitue le composant de base de la plate-forme.
- API de bas niveau SWT pour le support des objets graphiques.
- JFace, surcouche graphique fondée sur SWT, permettant de gérer des composants graphiques complexes, tel le workbench générique (UI) constitué de l'ensemble des vues, éditeurs et perspectives formant l'ossature d'Eclipse.

L'utilisation d'Eclipse RCP pour le développement d'applications riches offre les avantages suivants :

- **Support natif du système d'exploitation hôte.** Permet de profiter du « look and feel » de l'environnement graphique utilisé (Windows, MacOS, UNIX), puisque Eclipse RCP s'appuie sur les composants de la bibliothèque SWT, qui utilise de manière native les fonctionnalités du système d'exploitation.
- **Concept de « bureau métier ».** Permet de s'intégrer facilement aux applications bureautiques les plus courantes (Word, Excel, etc.) tout en combinant les développements au sein d'un seul « bureau métier ». Le socle technique mutualise les services d'authentification et SSO ainsi que de communication avec les serveurs d'applications.
- **Mise à jour automatique des plug-ins.** Permet une mise à jour des plug-ins à travers Internet par le biais du gestionnaire de mise à jour.
- **Déploiement multi-OS.** Facilite le packaging et le déploiement des applications par simple glisser-déplacer des répertoires de la distribution RCP dans les répertoires de plug-ins concernés.
- **Meilleures extensibilité et réutilisabilité.** Le partitionnement des applications en plusieurs modules ou plug-ins encourage la création de points d'extension favorisant l'extensibilité de l'ensemble ainsi que sa réutilisation pour d'autres applications.

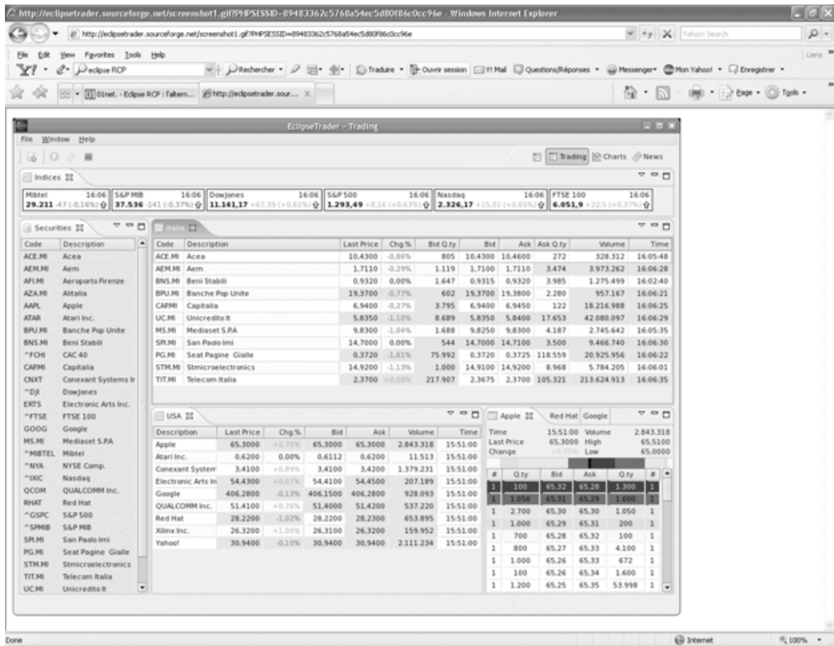
Exemples d'applications RCP

Voici quelques exemples d'applications professionnelles bâties sur le framework RCP.

La figure 2.3 illustre l'interface d'Eclipse Trader *(http://eclipsetrader.sourceforge.net/),* une application RCP Open Source de gestion de portefeuille donnant en temps réel l'évolution des cours de la Bourse.

Figure 2.3

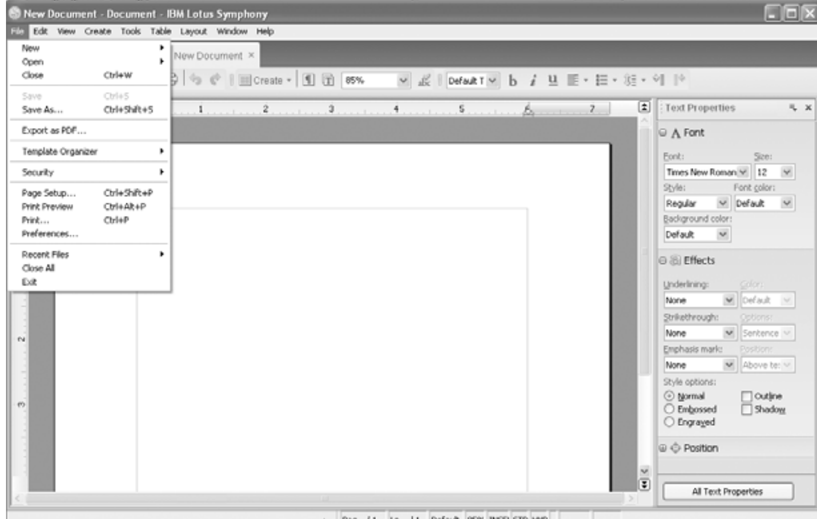
Application de gestion de portefeuilles boursiers fondée sur RCP



La figure 2.4 illustre la célèbre suite Notes de l'éditeur IBM « relookée » et rebaptisée IBM Lotus Symphony pour tirer partie des avantages de la pile RCP.

Figure 2.4

Nouvelle suite IBM Lotus Symphony



Mise en œuvre de RCP

L'objectif de l'environnement Eclipse RCP est de permettre de construire des applications évoluées aussi bien en termes d'IHM que d'intégration aux composants annexes (outils bureautiques, services profitant de l'écosystème Eclipse.

Une application RCP requiert :

- Un programme principal.
- Une perspective.
- Un composant Workbench Advisor, qui permet le contrôle les composants de l'application (menus, barres d'outils, perspectives).

Pour construire une application RCP avec une interface graphique minimale, il est nécessaire d'embarquer les plug-ins org.eclipse.ui et org.eclipse.core.runtime. En comptant l'exécutable Eclipse et le fichier statup.jar nécessaire à son exécution, environ 6,6 Mo d'espace disque sont nécessaires au bon déploiement et fonctionnement d'une application RCP.

Le package org.eclipse.core.resources, qui permet de gérer l'espace de travail Eclipse et les ressources correspondantes, ne fait pas partie de la distribution Eclipse RCP, bien qu'il soit très utile pour l'accès aux ressources Eclipse.

Le plug-in org.eclipse.ui.ide se situe au niveau du workbench, ou espace de travail Eclipse (org.eclipse.ui). Il constitue un ensemble d'API pour l'interface visuelle de l'IDE Eclipse, au sommet de laquelle gravitent d'autres composants, tels le JDT, le PDE, le débogueur et le gestionnaire de versions. L'IDE instancie le workbench ainsi que les menus spécifiques et la barre d'outils Eclipse en ajoutant des vues spécifiques, des pages de préférences et d'autres extensions.

Installation Eclipse et support de la langue française

À l'heure où nous écrivons ces lignes, le package de support de la langue française pour Europa n'est pas encore disponible (seul le package pour Callisto NLpack1-eclipse-SDK-3.2-win32.zip est disponible). Tous les messages relatifs à l'utilisation des assistants se font donc en langue anglaise.

Développement d'une application RCP

Les étapes de développement d'une application RCP sont les suivantes :

1. Création du squelette de l'application RCP.
2. Ajout de fonctionnalités Eclipse à l'application en utilisant les plug-ins suivants :
   – org.eclipse.ui.actionSets
   – org.eclipse.help.toc
   – org.eclipse.ui.preferencePages
   – org.eclipse.ui.views
3. Packaging du produit.
4. Déploiement et exécution du produit en dehors d'Eclipse.

Prérequis

Vous pouvez télécharger la version Eclipse for RCP/Plug-in Developers à partir du site d'Eclipse *(http://www.eclipse.org/downloads)*. C'est la version la plus complète pour le développement d'applications fondées sur le framework RCP. Il ne faut pas utiliser les autres versions, comme Eclipse IDE for JEE, car elles ne comprennent pas l'intégralité des bibliothèques et outils pour le développement dans cet environnement.

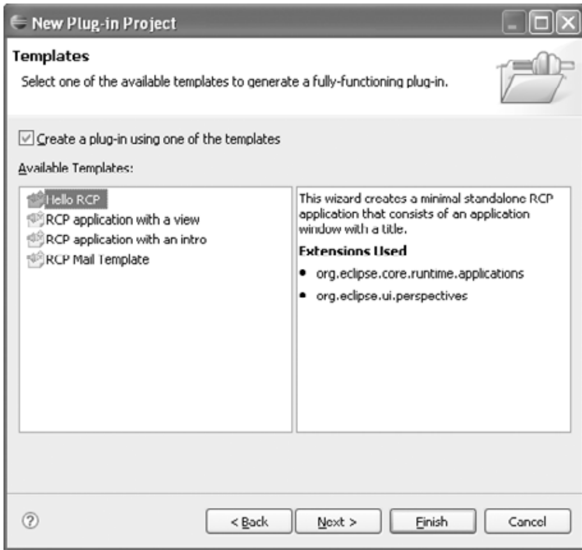
Création du squelette de l'application

Pour créer l'application RCP, procédez de la façon suivante :

1. Lancez la création d'un nouveau projet de plug-in, *via* les menus File, New et Plug-in Development.
2. Dans la liste Plug-in Development de l'assistant de création de projet Eclipse, sélectionnez Plug-in Project, puis cliquez sur Next.
3. Entrez le nom du projet, ici HelloRCP, en laissant les options par défaut (option Create a Java Project cochée avec les options de répertoire source et de sortie src et bin respectifs ainsi que Version Eclipse 3.3). Cliquez sur Next.
4. Dans la page suivante, cochez yes pour l'option « Would you like to create a rich client application ? », et saisissez vos initiales pour le champ associé à Plug-in Provider.
5. Laissez les autres options du plug-in cochées par défaut, et cliquez sur Next.
6. Vous obtenez différents templates sur lesquels vous pouvez vous appuyer pour créer l'application (voir figure 2.5). Pour cet exemple, sélectionnez RCP Hello, puis cliquez sur Next.

Figure 2.5

Templates Eclipse RCP disponibles



7. L'assistant vous rappelle un certain nombre d'informations sur le projet RCP que vous avez conçu, dont le titre de la fenêtre de l'application (Hello RCP), l'ID de l'application (application), le nom du package (hellorcp), la classe de l'application (Application). Cliquez sur Finish.

Vous devez obtenir l'écran illustré à la figure 2.6.

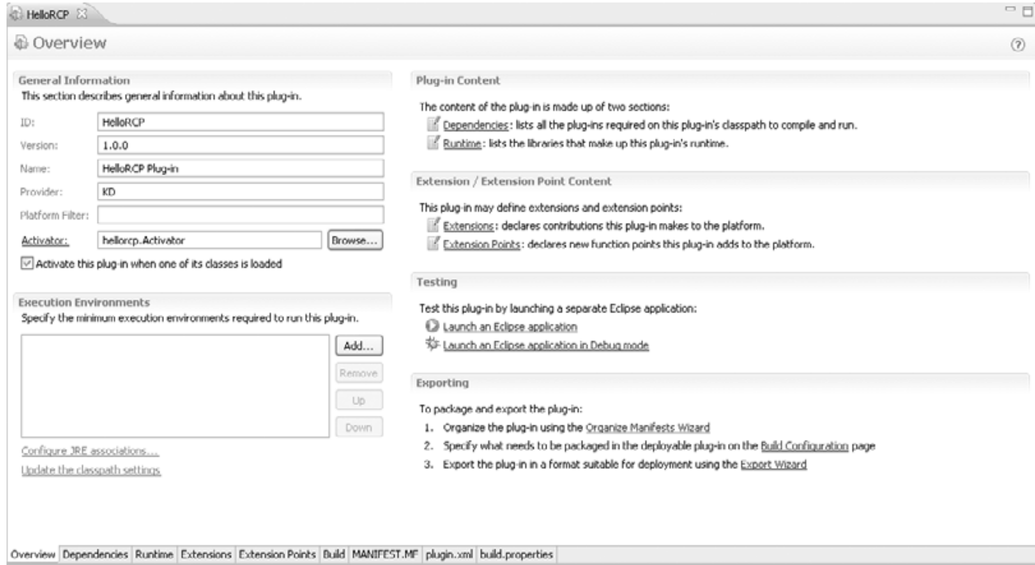


Figure 2.6

Configuration d'une application Eclipse RCP

La structure projet illustrée à la figure 2.7 doit être créée.

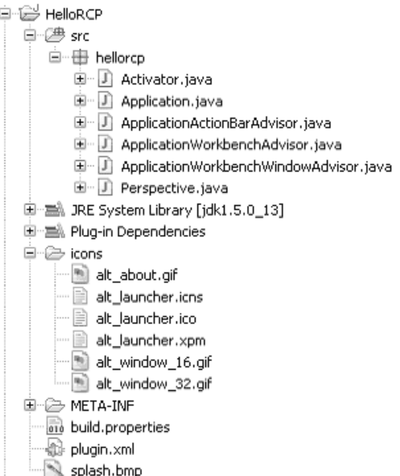


Figure 2.7

Structure du projet RCP après création

Noms de projets

Il est d'usage dans la communauté des développeurs Eclipse de nommer les projets sous la forme d'espaces de noms, à la manière des noms de packages Java. Puisque chaque plug-in est développé dans un projet, cette convention pratique permet de faire correspondre le nom du projet avec les identifiants des plug-ins qu'il contient.

Démarrage de l'application RCP

Si vous exécutez l'application en cliquant sur l'option Launch an Eclipse Application, après avoir basculé dans la perspective Resource et sélectionné le projet HelloRCP, vous obtenez la fenêtre dépouillée illustrée à la figure 2.8.

Figure 2.8

Première application Eclipse RCP

Examinons à présent les éléments nécessaires à la production de cette application.

Le template RCP Hello a ajouté les extensions org.eclipse.core.runtime.applications et org.eclipse.ui.perspectives, qui permettent de lier le plug-in à l'infrastructure Eclipse (que vous pouvez voir en cliquant sur l'onglet Extensions du plug-in illustré à la figure 2.9), ainsi que les six classes Java suivantes :

- Activator
- Application
- ApplicationActionBarAdvisor
- ApplicationWorkbenchAdvisor
- ApplicationWorkbenchWindowAdvisor
- Perspective

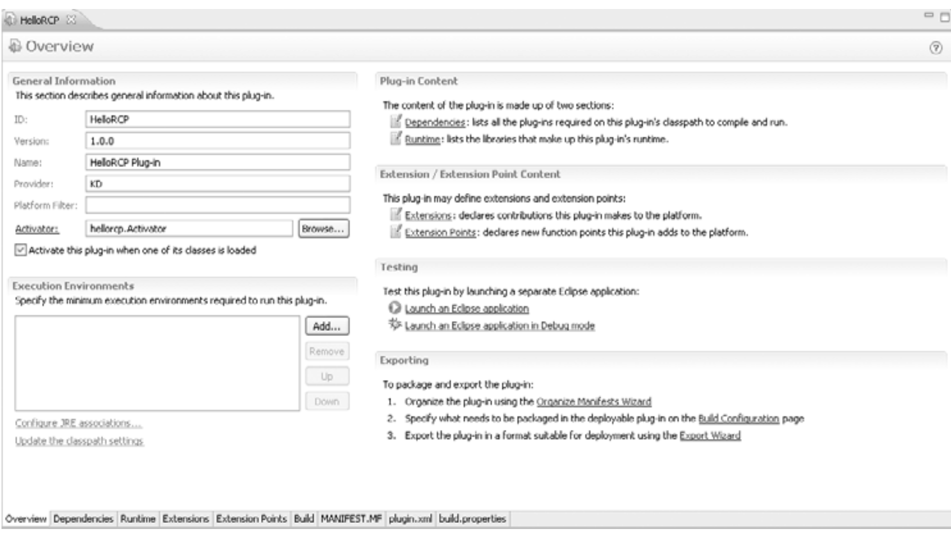


Figure 2.9

Onglet Extension du plug-in HelloRCP décrivant les principaux points d'extension

Classes principales d'une application RCP

La classe Application implémente l'interface IApplication. Elle lance le workbench Eclipse, ouvre une fenêtre, boucle en attente des événements utilisateur (clic, saisie, etc.) et se termine à la fermeture de la fenêtre :

```
/**
 * This class controls all aspects of the application's execution
 */
ublic class Application implements IApplication {

    /* (non-Javadoc)
     * @see org.eclipse.equinox.app.IApplication#start
     ➥(org.eclipse.equinox.app.IApplicationContext)
     */
    public Object start (IApplicationContext context) throws Exception {
        Display display = PlatformUI.createDisplay();
        try {
                int returnCode = PlatformUI.createAndRunWorkbench
                ➥(display, new ApplicationWorkbenchAdvisor());
                if (returnCode == PlatformUI.RETURN_RESTART)
                    return IApplication.EXIT_RESTART;
                else
                        return IApplication.EXIT_OK;
        } finally {
                display.dispose();
        }

    }
```

La classe Activator contrôle le cycle de vie du plug-in :

```
/**
 * The activator class controls the plug-in life cycle
 */
public class Activator extends AbstractUIPlugin {

    // The plug-in ID
    public static final String PLUGIN_ID = "HelloRCP";

    // The shared instance
    private static Activator plugin;

    /**
     * The constructor
     */
    public Activator() {
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.plugin.AbstractUIPlugin#start
     ➥(org.osgi.framework.BundleContext)
     */
    public void start(BundleContext context) throws Exception {
        super.start(context);
        plugin = this;
    }
```

```
    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.plugin.AbstractUIPlugin#stop
     ➥(org.osgi.framework.BundleContext)
     */
    public void stop(BundleContext context) throws Exception {
        plugin = null;
        super.stop(context);
    }

    /**
     * Returns the shared instance
     *
     * @return the shared instance
     */
    public static Activator getDefault() {
        return plugin;
    }

    /**
     * Returns an image descriptor for the image file at the given
     * plug-in relative path
     *
     * @param path the path
     * @return the image descriptor
     */
    public static ImageDescriptor getImageDescriptor(String path) {
        return imageDescriptorFromPlugin(PLUGIN_ID, path);
    }
}
```

La classe ApplicationActionBarAdvisor gère les différentes actions pour les barres de menus et d'état ainsi que les coolbars, variantes particulières de plusieurs contrôles ToolBar :

```
public class ApplicationActionBarAdvisor extends ActionBarAdvisor {

    public ApplicationActionBarAdvisor(IActionBarConfigurer configurer) {
        super(configurer);
    }

    protected void makeActions(IWorkbenchWindow window) {
    }

    protected void fillMenuBar(IMenuManager menuBar) {
    }

}
```

Les différentes méthodes de cette classe sont les suivantes :

- makeActions, qui permet de définir des actions pour chaque menu et barre de menus. Une action est un simple objet qui possède un composant GUI. La liste des actions supportées par le Workbench Eclipse peut être trouvée en lisant la javadoc associée à la classe abstraite ActionFactory, accessible sur le site de l'aide en ligne Eclipse *(http://help.eclipse.org/help32/index.jsp).*

- fillMenuBar, qui permet de remplir la barre de menus avec n'importe quelle action que vous souhaitiez voir apparaître ici.
- fillCoolBar, qui permet de remplir la coolbar. Une coolbar est une collection de toolbars, et une toolbar une collection d'actions.
- fillStatusLine, qui permet de remplir la barre d'état.

La classe ApplicationWorkbenchAdvisor configure le Workbench et définit la perspective par défaut de l'application à afficher :

```
import org.eclipse.ui.application.IWorkbenchWindowConfigurer;

public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {

    private static final String PERSPECTIVE_ID = "HelloRCP.perspective";

    public WorkbenchWindowAdvisor createWorkbenchWindowAdvisor
    ➥(IWorkbenchWindowConfigurer configurer) {
        return new ApplicationWorkbenchWindowAdvisor(configurer);
    }

    public String getInitialWindowPerspectiveId() {
    return PERSPECTIVE_ID;
    }
}
```

Comme vous pouvez le voir dans l'extrait de code précédent, il faut obligatoirement déclarer une chaîne représentant l'id de la perspective principale (celui déclaré dans le fichier de configuration plug-in.xml et auquel il est fait référence par le biais de la variable PERSPECTIVE_ID) et redéfinir la méthode getInitialWindowPerspectiveId.

La classe ApplicationWorkbenchWindowAdvisor permet de contrôler le comportement de la fenêtre (taille, titre) et les composants qui sont présents par défaut (barre d'état, barres d'outils) :

```
public class ApplicationWorkbenchWindowAdvisor extends WorkbenchWindowAdvisor {

    public ApplicationWorkbenchWindowAdvisor(IWorkbenchWindowConfigurer configurer)

        super(configurer);
    }

    public ActionBarAdvisor createActionBarAdvisor(IActionBarConfigurer configurer)

        return new ApplicationActionBarAdvisor(configurer);
    }

    public void preWindowOpen() {
        IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
        configurer.setInitialSize(new Point(400, 300));
        configurer.setShowCoolBar(false);
        configurer.setShowStatusLine(false);
        configurer.setTitle("Hello RCP");
    }
}
```

La classe Perspective représente un ensemble de vues, d'éditeurs et de menus de l'application. Il est obligatoire de définir au moins une perspective (ce sera la perspective par défaut).

Une perspective implémente l'interface IPerspectiveFactory et la méthode `createInitialLayout()`, qui permet de définir la position de chaque vue ou éditeur :

```
public class Perspective implements IPerspectiveFactory {

    public void createInitialLayout(IPageLayout layout) {
    }
}
```

Les différentes méthodes de cette classe sont les suivantes :

- `createActionBarAdvisor()` : crée un nouvel Action Bar Advisor pour configurer la barre d'actions de la fenêtre *via* le configurateur de barre d'action.
- `preWindowOpen()` : effectue différentes actions d'initialisation avant l'ouverture de la fenêtre (taille de la fenêtre, titre, affichage ou non de la barre d'outils, etc.).

Ajout de fonctionnalités

Vous allez étendre au fur et à mesure les fonctionnalités de votre première application RCP, en ajoutant du code spécifique aux méthodes générées par défaut.

Ajout d'une barre d'état

1. Ouvrez la classe ApplicationWorkbenchWindowAdvisor, et modifiez la méthode `preWindowOpen()` de la façon suivante :

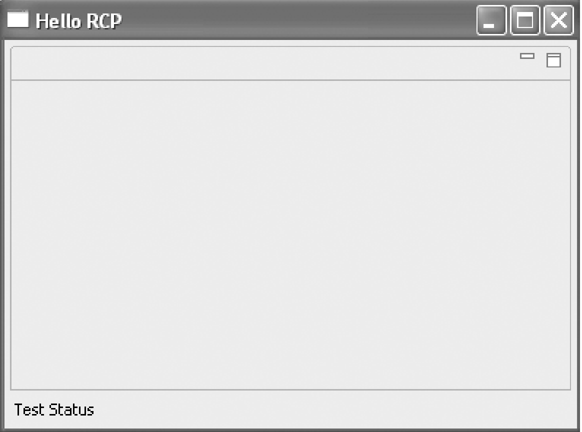
```
package hellorcp;

import org.eclipse.swt.graphics.Point;
import org.eclipse.ui.application.ActionBarAdvisor;
import org.eclipse.ui.application.IActionBarConfigurer;
import org.eclipse.ui.application.IWorkbenchWindowConfigurer;
import org.eclipse.ui.application.WorkbenchWindowAdvisor;
import org.eclipse.jface.action.IStatusLineManager;


public class ApplicationWorkbenchWindowAdvisor extends WorkbenchWindowAdvisor {

    public ApplicationWorkbenchWindowAdvisor(IWorkbenchWindowConfigurer configurer)

        super(configurer);
    }

    public ActionBarAdvisor createActionBarAdvisor(IActionBarConfigurer configurer)

        return new ApplicationActionBarAdvisor(configurer);
    }

    public void preWindowOpen() {
        IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
        configurer.setInitialSize(new Point(400, 300));
           configurer.setShowStatusLine (true);
```

```
        configurer.setShowCoolBar(false);
        configurer.setShowStatusLine(false);
        configurer.setTitle("Hello RCP");
    }
}


public void postWindowOpen() {
        IStatusLineManager statusline = getWindowConfigurer()
        getActionBarConfigurer().getStatusLineManager();

        statusline.setMessage(null, "Test Status");
    }
```

2. Relancez l'application pour vérifier la bonne prise en compte de ces modifications (voir figure 2.10).

Figure 2.10

Ajout d'une barre d'état à l'application HelloRCP



Ajout des actions (menus et barres d'outils) et fonctionnalités

Continuez votre parcours initiatique avec l'ajout de menus à votre application HelloRCP.

Les menus et actions associées à une application RCP peuvent être ajoutés par programmation ou par le biais de points d'extension. Vous allez procéder dans un premier temps par programmation en modifiant la méthode `makeActions()` de la classe ApplicationActionBarAdvisor et en l'ajoutant aux méthodes `fillMenuBar()` et `fillCoolbar()`.

1. Ouvrez la classe ApplicationActionBarAdvisor, et apportez les modifications suivantes :

```
import org.eclipse.jface.action.MenuManager;
import org.eclipse.jface.action.IMenuManager;
import org.eclipse.ui.IWorkbenchActionConstants;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.jface.action.Separator;
import org.eclipse.ui.application.ActionBarAdvisor;
import org.eclipse.ui.application.IActionBarConfigurer;

public class ApplicationActionBarAdvisor extends ActionBarAdvisor {
```

```
private IWorkbenchAction iExitAction;


protected void makeActions(IWorkbenchWindow window) {

        iExitAction = ActionFactory.QUIT.create(window);
        register(iExitAction);
        iAboutAction = ActionFactory.ABOUT.create(window);
        register(iAboutAction);
        iNewWindowAction = ActionFactory.OPEN_NEW_WINDOW.create(window);
        register(iNewWindowAction);
    }

protected void fillMenuBar(IMenuManager menuBar) {

        MenuManager fileMenu = new MenuManager("&Fichier",

                        IWorkbenchActionConstants.M_FILE);

        MenuManager helpMenu = new MenuManager("&Aide",

                        IWorkbenchActionConstants.M_HELP);

        menuBar.add(fileMenu);

        menuBar.add(helpMenu);

        // Ajoute le menu Fichier

        fileMenu.add(iNewWindowAction);

        fileMenu.add(new Separator());

        fileMenu.add(iExitAction);
        iExitAction.setText("Quitter");

        // Ajoute le menu Aide
        helpMenu.add(iAboutAction);

    }
```

Après la définition des vues, vous devez remplir la barre de menus puisque vous l'avez définie comme visible dans la classe ApplicationWorkbenchWindowAdvisor. Pour cela, vous devez éditer la classe ApplicationActionBarAdvisor. Cette classe est composée de différentes méthodes qui permettent de remplir les Action Bars et de définir des actions à exécuter pour chacun des menus (Fichier et Aide en particulier) en utilisant les MenuManager de la bibliothèque JFace qui permet de gérer les menus.

2. Relancez à nouveau l'application, qui doit ressembler à celle illustrée à la figure 2.11.

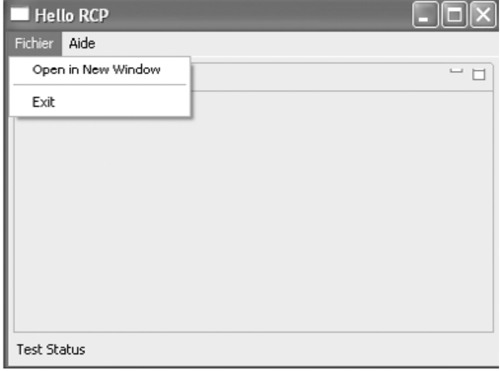


Figure 2.11

Ajout d'une barre de menus à l'application HelloRCP

Ajout d'une barre d'outils (Toolbar)

Vous allez ajouter une barre de boutons par le biais des méthodes `fillCoolbar()` et `ApplicationActionBarAdvisor()`.

1. Pour ce faire, allez dans la classe ApplicationActionBarAdvisor, et ajoutez les actions correspondantes dans la méthode makeActions :

```
import org.eclipse.ui.actions.ActionFactory;

import org.eclipse.swt.SWT;

import org.eclipse.jface.action.ICoolBarManager;
import org.eclipse.jface.action.IToolBarManager;
import org.eclipse.jface.action.ToolBarManager;
import org.eclipse.jface.action.ToolBarContributionItem;

private IWorkbenchAction iSaveAction;



protected void fillCoolBar(ICoolBarManager coolBar) {

         // Ajoute une nouvelle barre d'outils a l'application

        IToolBarManager toolbar = new ToolBarManager(SWT.FLAT | SWT.RIGHT);

        coolBar.add(new ToolBarContributionItem(toolbar, "main"));

        // Ajoute les entrees a la barre d'outils

        toolbar.add(iSaveAction);

        toolbar.add(iExitAction);

}


protected void makeActions(IWorkbenchWindow window) {

        iExitAction = ActionFactory.QUIT.create(window);
        register(iExitAction);
```

```
        iAboutAction = ActionFactory.ABOUT.create(window);
        register(iAboutAction);

        iNewWindowAction = ActionFactory.OPEN_NEW_WINDOW.create(window);
        register(iNewWindowAction);


        iSaveAction = ActionFactory.SAVE.create(window);
        register(iSaveAction);


}
```

Dans cet extrait, Eclipse RCP fournit des actions standards, comme celles utilisées dans la méthode makeActions (par exemple dans la ligne `iNewWindowAction = ActionFactory.OPEN_NEW_WINDOW.create(window)`, qui crée une deuxième instance de l'application en cours dans une nouvelle fenêtre lorsque l'utilisateur clique sur ce menu.

2. Dans la classe ApplicationWorkbenchWindowAdvisor, modifiez la méthode `preWindowOpen()` pour qu'elle affiche une barre de boutons sous la barre de menus avec `configurer.setShowCoolBar(true)`. Vous devez obtenir le résultat illustré à la figure 2.12.

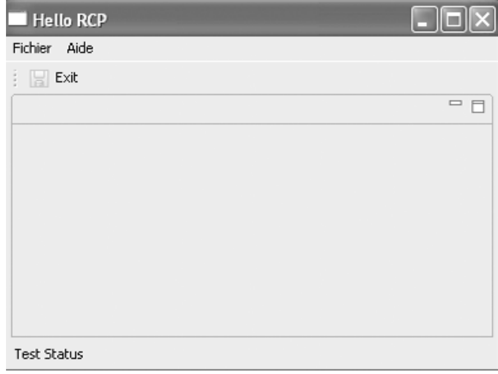


Figure 2.12

Ajout d'une barre de boutons

3. Ajoutez les autres options du menu de manière que votre application ressemble à la vue illustrée à la figure 2.13.

Figure 2.13

Ajout des nouvelles fonctionnalités à l'application Hello RCP

Cinq menus vont être ajoutés :

– Fichier (déjà implémenté lors de l'étape précédente)

– Fenêtre (à implémenter)

– Aide (déjà effectué à l'étape précédente)

– Fenêtre/Ouvrir perspective (à implémenter)

– Fenêtre/Afficher vue (à implémenter)

Ainsi que six actions :

– Fichier/Sortie (déjà implémentée à l'étape précédente)

– Fenêtre/Ouvrir perspective/Autre (à implémenter)

– Fenêtre/Afficher vue/Autre (à implémenter)

– Fenêtre/Préférences (à implémenter)

– Aide/Sommaire Aide (à implémenter)

– Aide/A Propos (déjà implémentée à l'étape précédente)

Il s'agit d'actions prédéfinies incluses par défaut dans Eclipse mais que RCP fournit pour des raisons pratiques.

Vous allez modifier la classe ApplicationActionBar pour personnaliser la méthode fillMenuBar, qui permet de personnaliser la barre de menus (vous pouvez en voir le détail dans le code source téléchargeable sur la page Web dédiée à l'ouvrage).

4. Enrichissez la classe ApplicationActionBarAdvisor visitée précédemment par le code suivant :

```
public class ApplicationActionBarAdvisor extends ActionBarAdvisor {

    private IWorkbenchAction iExitAction;
    private IWorkbenchAction iAboutAction;
    private IWorkbenchAction iNewWindowAction;
    private IWorkbenchAction iSaveAction;
    private IWorkbenchAction helpAction;
    private IWorkbenchAction preferencesAction;


private MenuManager perspectiveMenu = new MenuManager("&Ouvrir Perspective");
    private MenuManager viewMenu = new MenuManager("Afficher Vue");



public ApplicationActionBarAdvisor(IActionBarConfigurer configurer) {
        super(configurer);
    }

protected void makeActions(IWorkbenchWindow window) {
        iExitAction = ActionFactory.QUIT.create(window);
        register(iExitAction);

        iAboutAction = ActionFactory.ABOUT.create(window);
        register(iAboutAction);
```

```
        iNewWindowAction = ActionFactory.OPEN_NEW_WINDOW.create(window);
        register(iNewWindowAction);

        iSaveAction = ActionFactory.SAVE.create(window);
        register(iSaveAction);


        helpAction = ActionFactory.HELP_CONTENTS.create(window);
            register(helpAction);

            preferencesAction = ActionFactory.PREFERENCES.create(window);
            register(preferencesAction);

    perspectiveMenu.add(ContributionItemFactory.PERSPECTIVES_SHORTLIST.create(window));
    viewMenu.add(ContributionItemFactory.VIEWS_SHORTLIST.create(window));


    }


    protected void fillMenuBar(IMenuManager menuBar) {
        MenuManager fileMenu = new MenuManager("&Fichier"
        ➥,IWorkbenchActionConstants.M_FILE);

        MenuManager windowMenu = new MenuManager("&Fenêtre",
        ➥ IWorkbenchActionConstants.M_WINDOW);

        MenuManager helpMenu = new MenuManager("&Aide",
        ➥ IWorkbenchActionConstants.M_HELP);

        menuBar.add(fileMenu);
        menuBar.add(windowMenu);
        // Add a group marker indicating where action set menus will appear.
        menuBar.add(new GroupMarker(IWorkbenchActionConstants.MB_ADDITIONS));
        menuBar.add(helpMenu);


        // Fichier

        fileMenu.add(iNewWindowAction);
        fileMenu.add(new Separator());
        fileMenu.add(iExitAction);


        windowMenu.add(perspectiveMenu);
        windowMenu.add(viewMenu);
        windowMenu.add(new Separator());
        windowMenu.add(preferencesAction);

        // Aide

        helpMenu.add(helpAction);
        helpMenu.add(new Separator());
     helpMenu.add(iAboutAction);

    }
```

5. Ouvrez le fichier plugin.xml si ce n'est déjà fait.
6. Sélectionnez l'onglet Overview, et lancez à nouveau l'application RCP en cliquant sur Launch an Eclipse application. Les menus de votre application doivent ressembler à ceux illustrés à la figure 2.13.

   Il est possible à ce stade d'ajouter quelques fonctionnalités.
7. Cliquez sur Fenêtre, Ouvrir perspective et Autre. La perspective créée par le template Hello RCP s'affiche.
8. Cliquez sur Fenêtre, Afficher vue et Autre. Une vue vide s'affiche, que vous remplirez un peu plus tard.
9. Cliquez sur Fenêtre et Préférences. Vous devez voir s'afficher une boîte de dialogue Préférences vide.
10. Cliquez sur Aide et About. Une boîte de dialogue s'affiche ne comportant que trois boutons (Détails des plug-in, Détails de la configuration et OK). En cliquant sur Plug-in Details, vous voyez s'afficher votre plug-in dans la liste de ceux faisant partie de l'application RCP (voir figure 2.14).

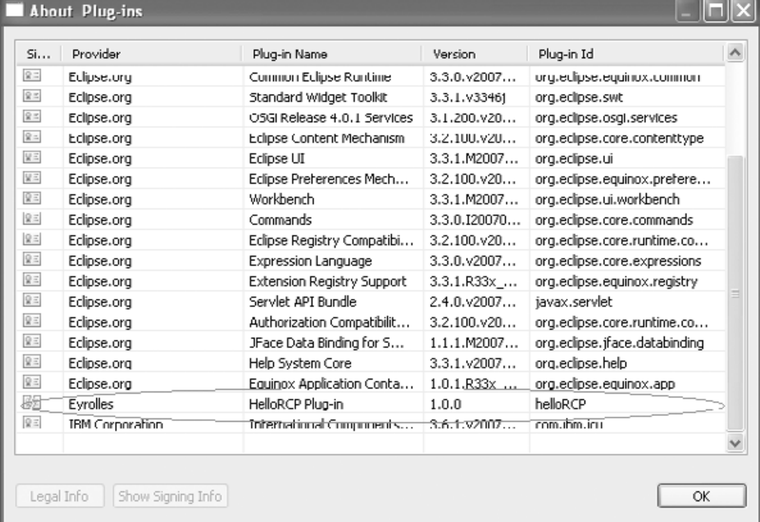


Figure 2.14

Ajout du plug-in Hello RCP dans la liste des plug-ins Eclipse

> **Copyright**
>
> Vous pouvez remarquez que les informations de Copyright sont grisées *(Legal Info)*. Pour activer cette fonctionnalité, arrêtez votre application, et copiez à la racine du répertoire du workspace contenant votre projet RCP les fichiers about.ini et about.html fournis dans l'exemple. Relancez ensuite votre application pour valider cette fonctionnalité.

Ajout de fonctions Eclipse

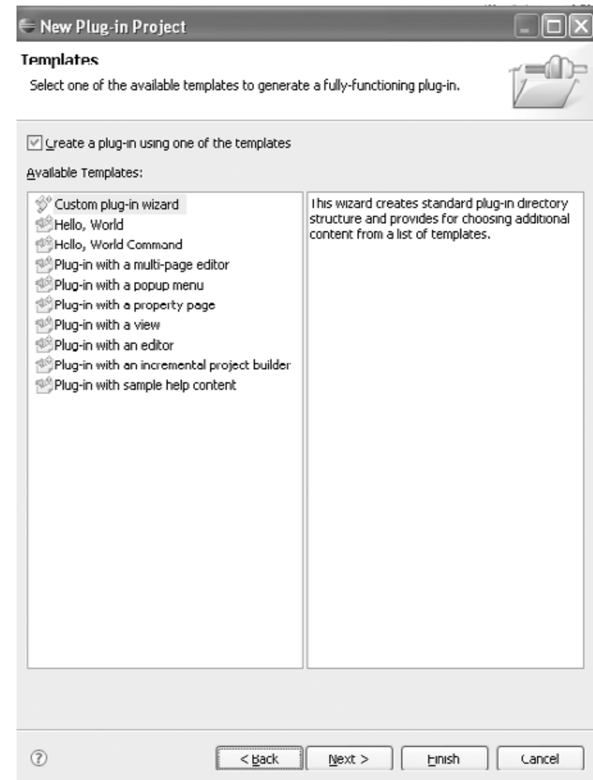
À ce stade, l'application Hello RCP est encore réduite à sa plus simple expression. Vous allez remplir la vue et compléter l'option Préférences ainsi que la table des matières de l'aide :

1. Sélectionnez File, New (ou Ctrl+N), puis Plug-in Project, et entrez comme nom de projet com.eclipsercp.HelloRCP.ui. Cliquez sur Next.

2. Choisissez l'Assistant de plug-in personnalisé afin de créer une structure de répertoire de plug-ins standard permettant de sélectionner des éléments supplémentaires dans une liste de modèles.
3. Sélectionnez no pour l'option « Would you like to create a rich client application », puis cliquez sur Next.
4. Sélectionnez Custom plug-in wizard (voir figure 2.15).

Figure 2.15

Configuration du plug-in (1/2)



5. Cliquez sur Next.
6. Choisissez Deselect All, puis sélectionnez les options Hello World Action Set (jeu d'actions), Help Table of Contents (table des matières d'aide), Preference Page et enfin View (vue associée au plan de travail). Vous devez obtenir la vue illustrée à la figure 2.16.
7. Cliquez sur Finish ou sur Next si vous souhaitez personnaliser les différents templates choisis (texte associé à la table des matières, nom des packages, message associé à la boîte de dialogue, nom de la vue et de la classe associée, type d'affichage de la vue en tableau ou en tree view).

   Pour exécuter l'application, vous allez cette fois utiliser l'onglet Aperçu, et non le lien Lancer une application Eclipse. Vous lancerez ensuite l'application Hello RCP par le biais de l'option Exécuter de la barre d'outils.
8. Sélectionnez l'option Run As, Open Run Dialog, puis, dans l'assistant de création, de gestion et d'exécution des configurations qui s'ouvre, sélectionnez l'onglet Plug-in.

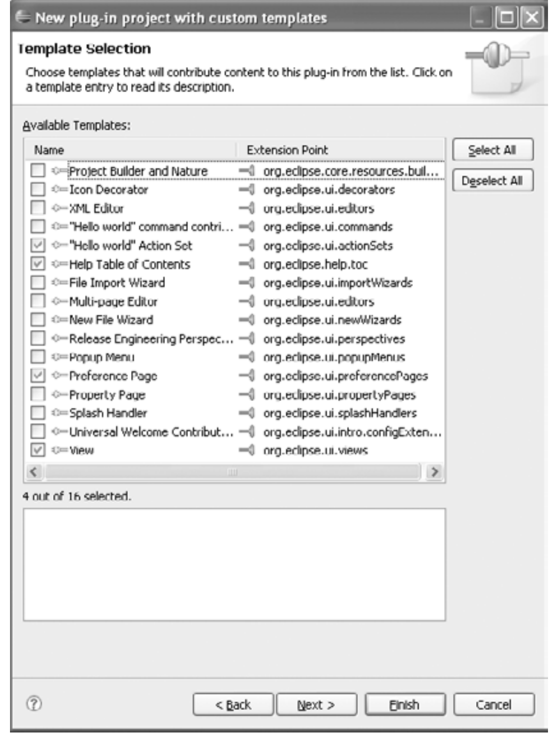


Figure 2.16

Configuration du plug-in (2/2)

9. Cochez dans la liste le plug-in com.eclipsercp.HelloRCP.ui qui vient d'être créé et helloRCP, comme illustré à la figure 2.17.

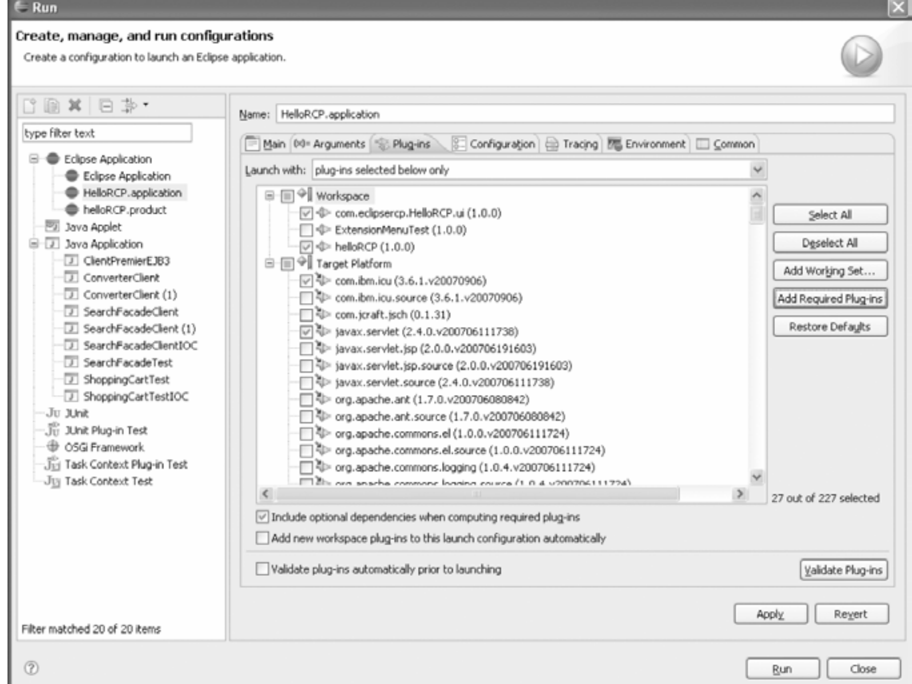


Figure 2.17

Assistant de configuration de lancement

10. Cliquez sur Add required Plug-ins, et sélectionnez les plug-ins suivants :
    – org.apache.ant
    – org.apache.lucene
    – org.eclipse.help.appserver
    – org.eclipse.help.base
    – org.eclipse.help.ui
    – org.eclipse.help.webapp
    – org.eclipse.tomcat
    – org.eclipse.ui.forms
11. Cliquez sur Run puis sur Exécuter.

    Votre application doit à présent afficher le menu Sample Menu illustré à la figure 2.18.

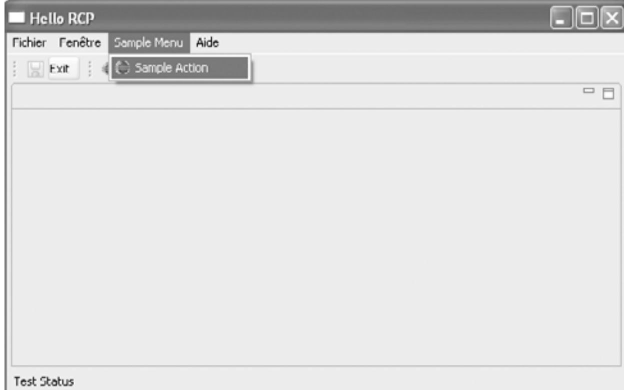


Figure 2.18

Application Eclipse RCP après ajout de fonctionnalités

12. En cliquant sur Fenêtre, Afficher Vue puis Autre, vous devez voir s'afficher la vue plus riche illustrée à la figure 2.19, contenant une aide en ligne dynamique ainsi qu'une vue Exemple de vue.

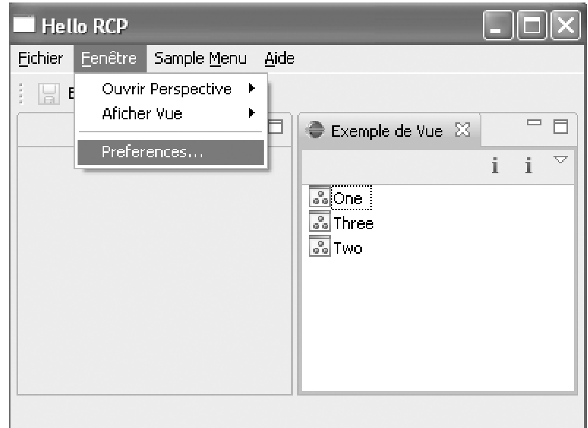


Figure 2.19

Application Eclipse RCP enrichie

13. En cliquant sur Exemple de vue, vous devez voir s'afficher sous forme de liste d'icônes activables les options illustrées à la figure 2.20.

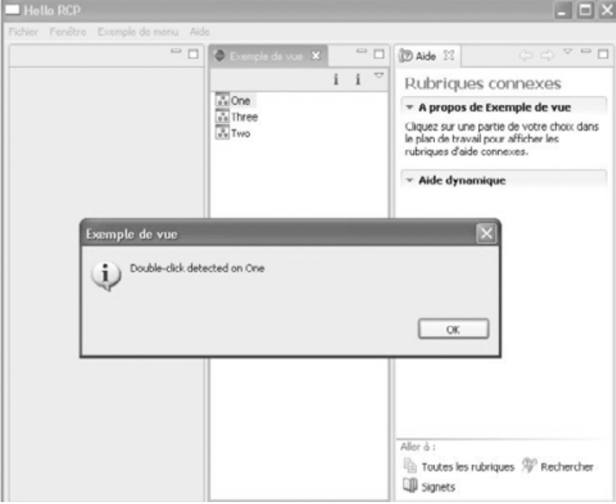


Figure 2.20

Application Eclipse RCP finale

Comme l'illustre la figure 2.21, le menu Préférences s'est enrichi d'un exemple de préférence, ajouté automatiquement. La page d'aide a également été ajoutée au menu Préférences du fait de l'ajout du plug-in d'aide lors de la configuration du plug-in personnalisé.

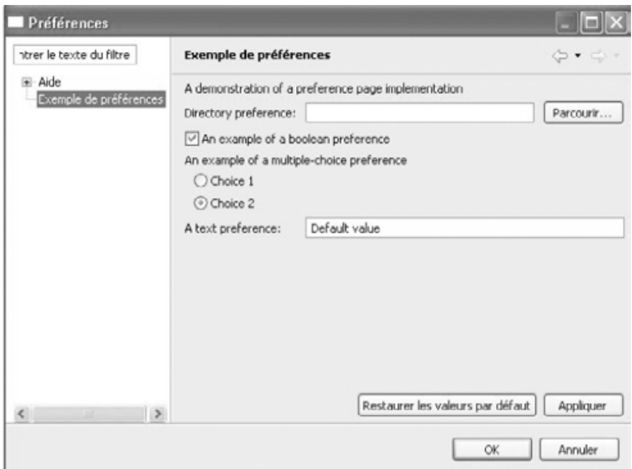


Figure 2.21

Menu Préférences de l'application RCP

Affichage des entrées associées au menu et à la barre d'outils *via* les extensions

Vous allez à présent utiliser l'outillage RCP pour afficher les points d'extension provenant d'éditeurs tiers (vous en avez eu un aperçu lorsque vous avez sélectionné quelques plug-ins à la section précédente).

La déclaration d'une extension se fait en XML. Chaque plug-in peut prévoir des points d'extension sur lesquels d'autres plug-ins viendront se « brancher », comme l'illustre la figure 2.22.

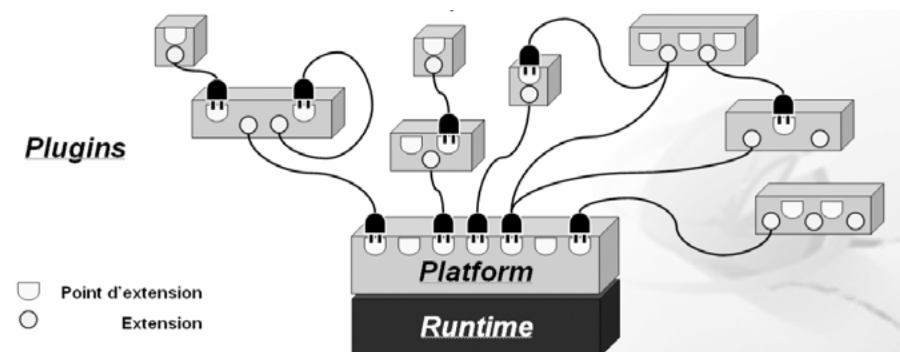


Figure 2.22

Points d'extension

Un plug-in peut étendre ses propres points d'extension (exemple actionSets), un point d'extension étant déclaré par un seul plug-in. La notion d'extension est fondamentale pour la compréhension du framework RCP et plus généralement du développement de plug-ins sur cette plate forme. Vous pouvez approfondir ce sujet sur le portail dédié au projet RCP, à l'adresse *http://www.eclipse.org/rcp.*

Pour avoir la liste des points d'extension, ouvrez le fichier plugin.xml, et sélectionnez l'onglet Extensions. Vous pouvez voir la liste des points d'extension définis pour cette application, comme illustré à la figure 2.23.

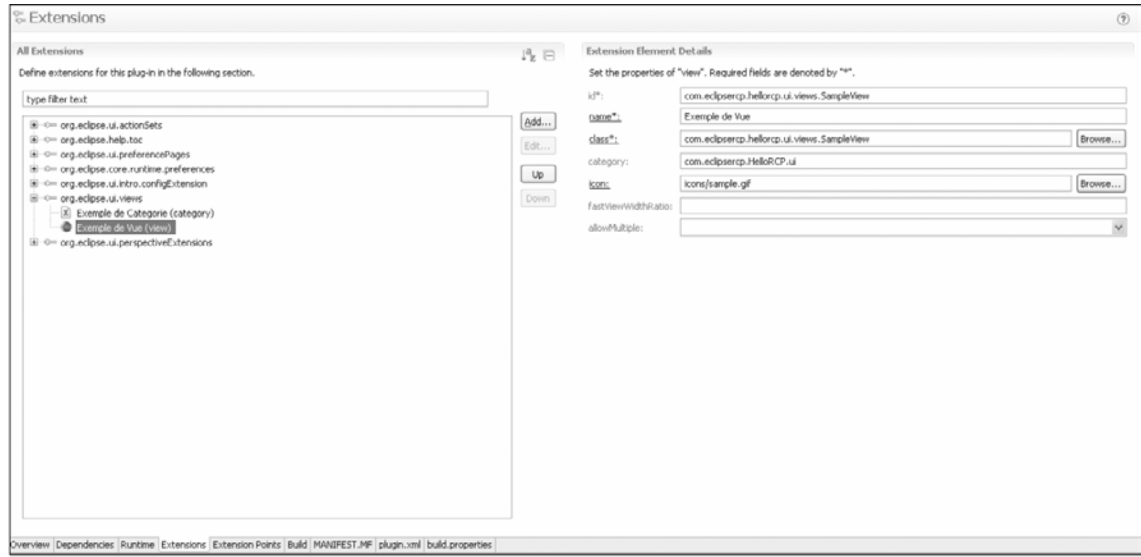


Figure 2.23

Assistant de définition des points d'extension Eclipse/RCP

Vous pouvez ainsi voir le code généré pour les actions associées à la vue qui a été définie, SampleView.java (notez le code associé à la méthode makeActions qui permet la gestion des actions associées à l'IHM) :

```
public class SampleView extends ViewPart {
    private TableViewer viewer;
    private Action action1;
    private Action action2;
    private Action doubleClickAction;

    /*
     * The content provider class is responsible for
     * providing objects to the view. It can wrap
     * existing objects in adapters or simply return
     * objects as-is. These objects may be sensitive
     * to the current input of the view, or ignore
     * it and always show the same content
     * (like Task List, for example).
     */

    class ViewContentProvider implements IStructuredContentProvider {
        public void inputChanged(Viewer v, Object oldInput, Object newInput) {
        }
        public void dispose() {
        }
        public Object[] getElements(Object parent) {
                return new String[] { "One", "Two", "Three" };
    }
    }
    class ViewLabelProvider extends LabelProvider implements ITableLabelProvider {
        public String getColumnText(Object obj, int index) {
                return getText(obj);
        }
        public Image getColumnImage(Object obj, int index) {
                return getImage(obj);
        }
        public Image getImage(Object obj) {
                return PlatformUI.getWorkbench().

    getSharedImages().getImage(ISharedImages.IMG_OBJ_ELEMENT);
        }
    }
    class NameSorter extends ViewerSorter {
    }

    /**
     * The constructor.
     */
    public SampleView() {
    }

    /**
     * This is a callback that will allow us
     * to create the viewer and initialize it.
     */
    public void createPartControl(Composite parent) {
        viewer = new TableViewer(parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
```

```
        viewer.setContentProvider(new ViewContentProvider());
        viewer.setLabelProvider(new ViewLabelProvider());
        viewer.setSorter(new NameSorter());
        viewer.setInput(getViewSite());
        makeActions();
        hookContextMenu();
        hookDoubleClickAction();
        contributeToActionBars();
    }

    private void hookContextMenu() {
        MenuManager menuMgr = new MenuManager("#PopupMenu");
        menuMgr.setRemoveAllWhenShown(true);
        menuMgr.addMenuListener(new IMenuListener() {
                public void menuAboutToShow(IMenuManager manager) {
                        SampleView.this.fillContextMenu(manager);
                }
        });
        Menu menu = menuMgr.createContextMenu(viewer.getControl());
        viewer.getControl().setMenu(menu);
        getSite().registerContextMenu(menuMgr, viewer);
    }

    private void contributeToActionBars() {
        IActionBars bars = getViewSite().getActionBars();
        fillLocalPullDown(bars.getMenuManager());
        fillLocalToolBar(bars.getToolBarManager());
    }

    private void fillLocalPullDown(IMenuManager manager) {
        manager.add(action1);
        manager.add(new Separator());
        manager.add(action2);
    }

    private void fillContextMenu(IMenuManager manager) {
        manager.add(action1);
        manager.add(action2);
        // Other plug-ins can contribute there actions here
        manager.add(new Separator(IWorkbenchActionConstants.MB_ADDITIONS));
    }

    private void fillLocalToolBar(IToolBarManager manager) {
        manager.add(action1);
        manager.add(action2);
    }

    private void makeActions() {
        action1 = new Action() {
                public void run() {
                        showMessage("Action 1 executed");
                }
        };
        action1.setText("Action 1");
        action1.setToolTipText("Action 1 tooltip");
        action1.setImageDescriptor(PlatformUI.getWorkbench().getSharedImages().
                getImageDescriptor(ISharedImages.IMG_OBJS_INFO_TSK));
```

```
        action2 = new Action() {
                public void run() {
                        showMessage("Action 2 executed");
                }
        };
        action2.setText("Action 2");
        action2.setToolTipText("Action 2 tooltip");
        action2.setImageDescriptor(PlatformUI.getWorkbench().getSharedImages().
                        getImageDescriptor(ISharedImages.IMG_OBJS_INFO_TSK));
        doubleClickAction = new Action() {
                public void run() {
                        ISelection selection = viewer.getSelection();
                        Object obj =
((IStructuredSelection)selection).getFirstElement();
                        showMessage("Double-click detected on "+obj.toString());
                }
        };
    }

    private void hookDoubleClickAction() {
        viewer.addDoubleClickListener(new IDoubleClickListener() {
                public void doubleClick(DoubleClickEvent event) {
                        doubleClickAction.run();
                }
        });
    }
    private void showMessage(String message) {
        MessageDialog.openInformation(
                viewer.getControl().getShell(),
                "Exemple de Vue",
                message);
    }

    /**
     * Passing the focus request to the viewer's control.
     */
    public void setFocus() {
        viewer.getControl().setFocus();
    }
}
```

Packaging

Il ne vous reste plus qu’à packager votre plug-in sous la forme d’un module déployable et exécutable en dehors d’Eclipse.

Pour packager le plug-in, procédez de la façon suivante :

1. Faites un clic droit sur le projet helloRCP, puis sélectionnez New et Product Configuration dans le menu contextuel.
2. Nommez l’application par exemple HelloRCP.Product dans l’assistant de configuration de produit Eclipse.
3. Activez la case Use a launch configuration.
4. Cliquez sur Finish.

L'éditeur de définition du produit apparaît (onglet Aperçu) comme illustré à la figure 2.24.

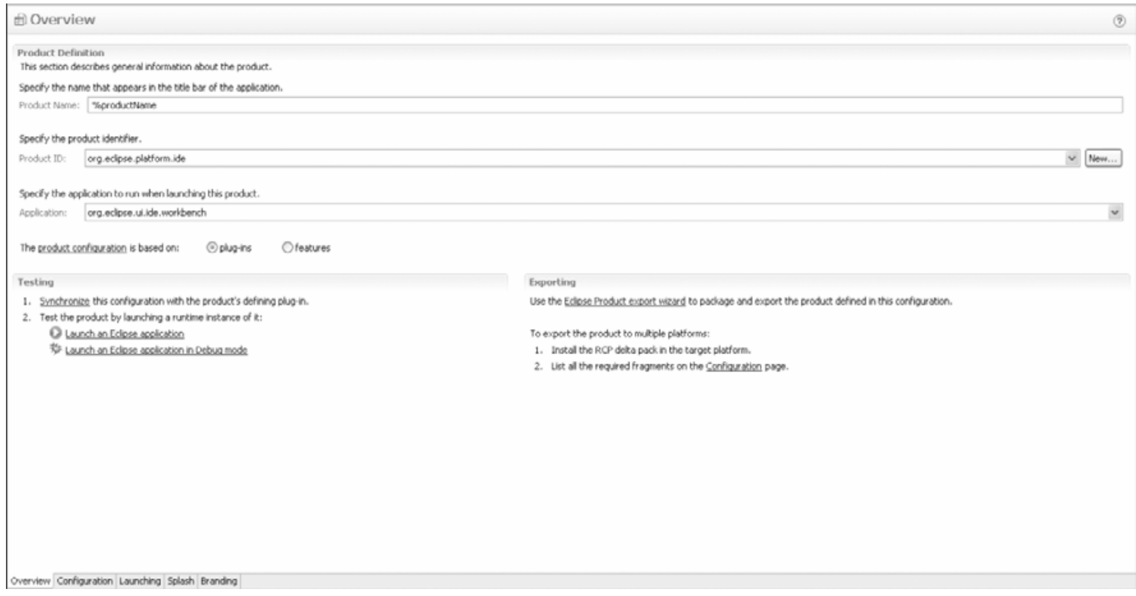


Figure 2.24

Éditeur de définition du produit Eclipse RCP

5. Saisissez HelloRCP devant le champ Product Name.
6. Cliquez sur le bouton New devant le champ Product ID. L'assistant de définition de nouveau produit apparaît comme illustré à la figure 2.25.

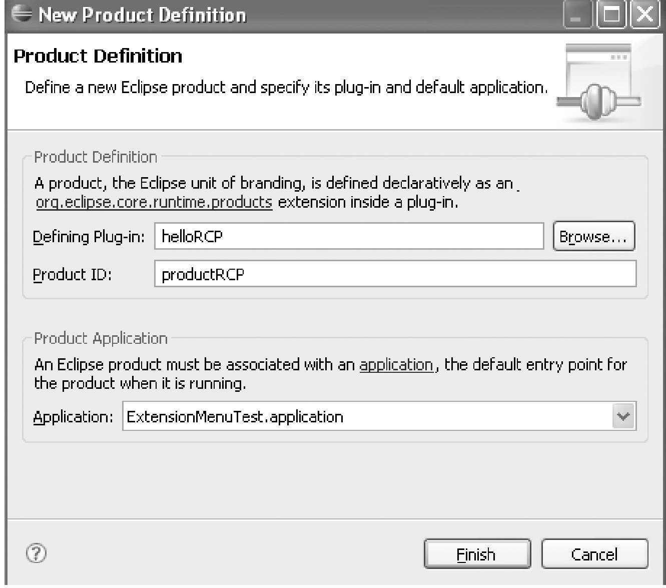


Figure 2.25

Éditeur de définition du produit Eclipse RCP (suite)

7. Sélectionnez helloRCP dans le champ Defining Plug-in et productRCP dans le champs Product ID. Cliquez sur Finish.
8. Activez la case features, et basculez dans l'onglet Configuration (voir figure 2.26).

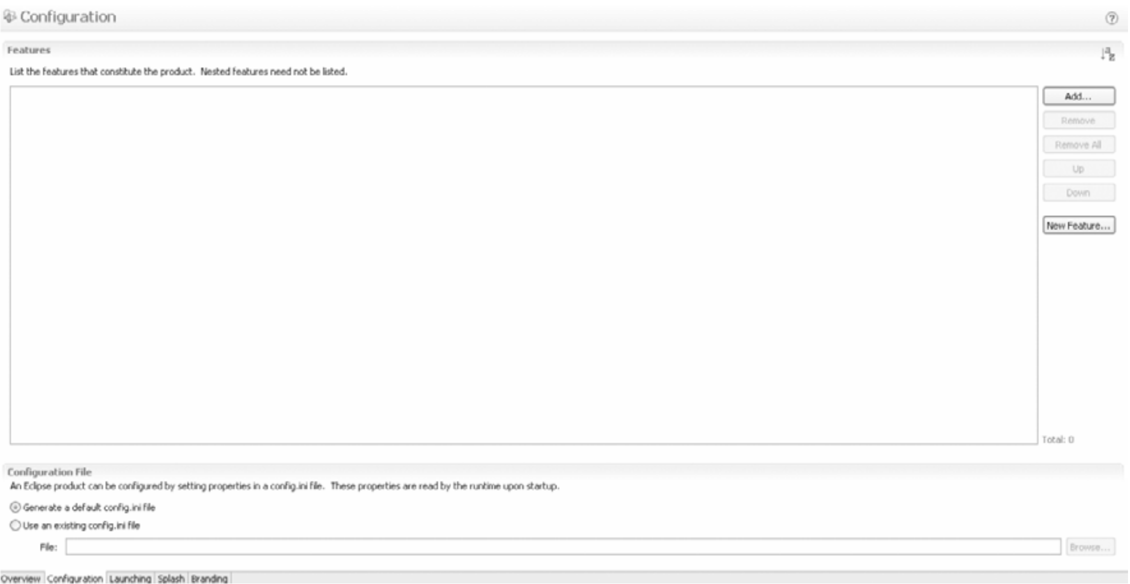


Figure 2.26

Spécification des « features » du produit

9. Cliquez sur New Feature pour intégrer les extensions qui ont permis de bâtir votre application (voir figure 2.27).

Figure 2.27

Définition des « features »

10. Entrez helloRCP-feature devant le champ nom de projet, et cliquez sur Next (voir figure 2.28).

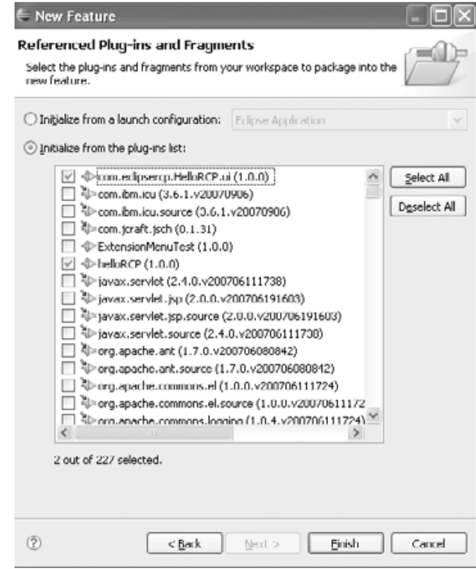


Figure 2.28

Liste des plug-ins à référencer dans le produit

11. Cochez com.eclipsercp.HelloRCP.ui et helloRCP comme liste des plug-ins à initialiser, et cliquez sur Finish.
12. Basculez vers l'onglet Information illustré à la figure 2.29.

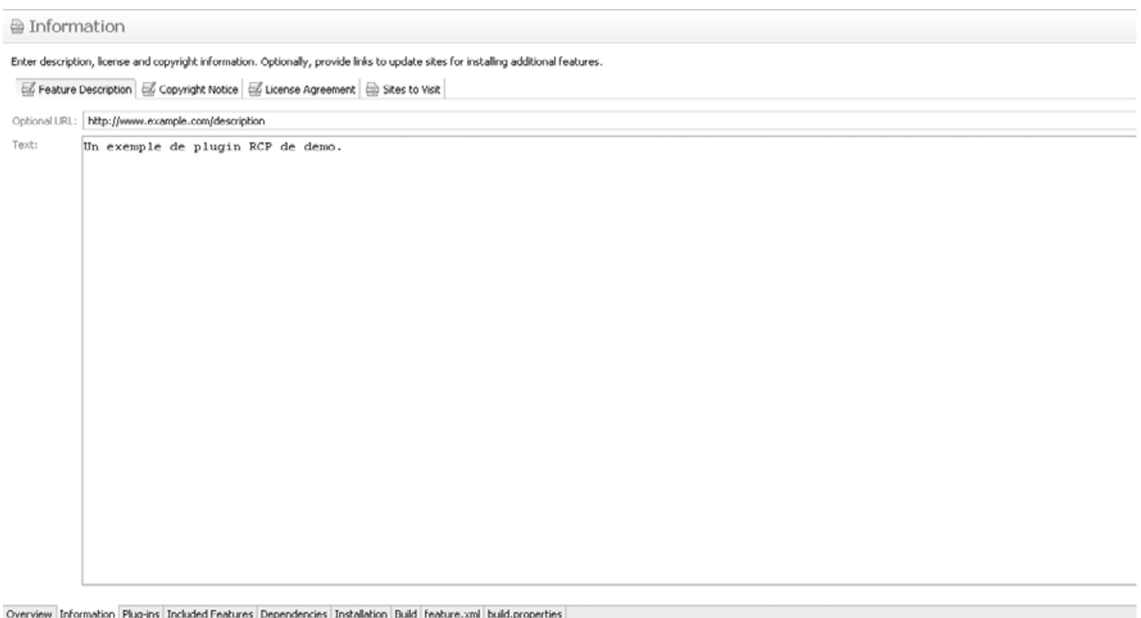


Figure 2.29

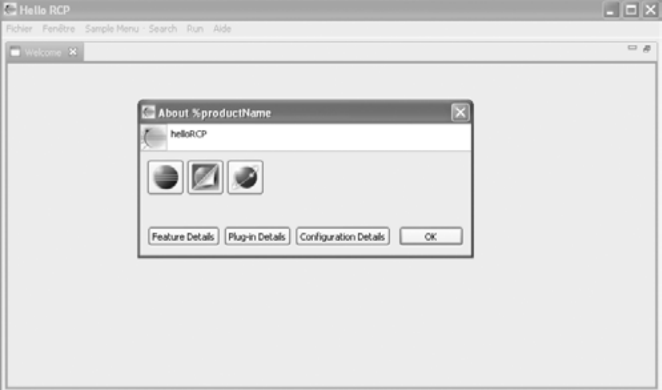
Page d'informations du plug-in

13. Sélectionnez l'onglet Licence Agreement, et entrez license.html devant le champ Optional URL.
14. Sélectionnez Included Features, et cliquez sur Add.
15. Sélectionnez les caractéristiques org.eclipse.rcp et com.ibm.plte.help, et sauvegardez le tout (Ctrl+S).
16. Revenez à la page HelloRCP.product, et sélectionnez l'onglet Branding.
17. Spécifiez en option les images associées au produit packagé devant les champs 16x16 Image ainsi que pour le champ About Dialog.
18. Saisissez devant le champ Test associé helloRCP, et sauvegardez le tout (Ctrl+S).
19. dans l'onglet Splash spécifiez helloRCP devant le champs Plug-in Splash Screen

Le packaging du produit est terminé. Vous pouvez le tester en allant sur la page HelloRCP_Product et en le lançant dans l'onglet Overview *via* l'option Launch Product. Vous devez voir s'afficher la page illustrée à la figure 2.30.

Figure 2.30

Lancement de l'application Hello RCP



Déploiement en dehors d'Eclipse

Une fois l'application RCP développée, l'objectif est de l'exécuter sans que l'utilisateur se soucie du code Eclipse ni du runtime sous-jacent. Pour ce faire, vous pouvez opter pour des produits qui se chargeront de l'installation de votre application (exemple InstallShield), ou plus simplement vous appuyer sur l'arborescence générée qui va contenir le programme lanceur startup.jar, config.ini et tous les plug-ins et fonctions requis par le produit.

Eclipse fournit pour ce faire un assistant d'exportation de produit pour vous aider à construire le répertoire d'installation de votre système de fichiers.

Tous les projets Eclipse RCP contiennent un fichier build.properties utilisé par Eclipse pour déterminer quel fichier et ressources il doit exporter.

1. Dans la page de construction du plug-in Hello RCP (Build Configuration), sélectionnez les fichiers ou répertoires à exporter avec le plug-in (voir figure 2.31) :
   – about.html
   – about.ini
   – about.properties

– répertoire icône

– splash.bmp

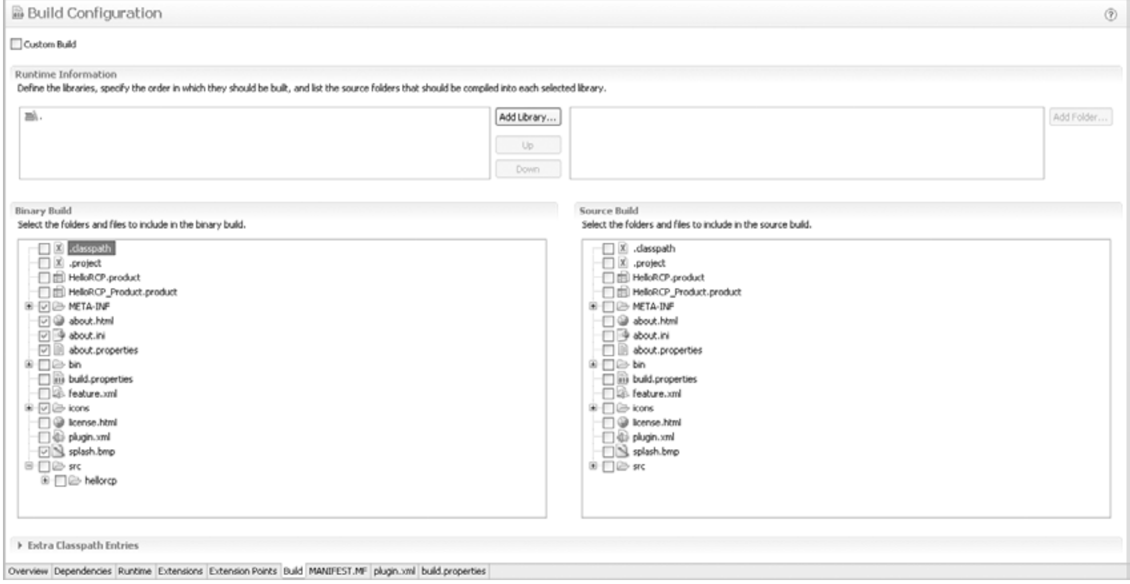


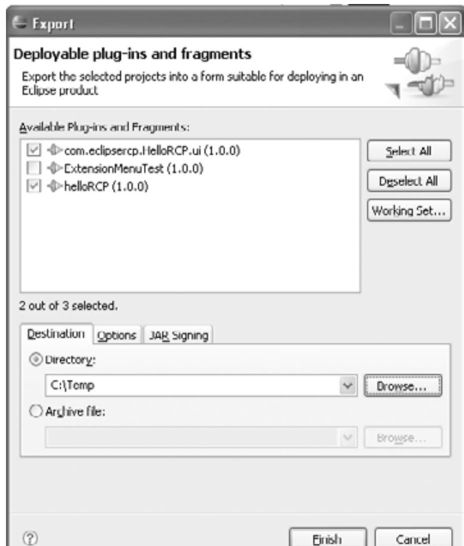
Figure 2.31

Assistant de construction de configuration

2. Dans l'éditeur de page de construction du projet (feature.xml), sélectionnez le fichier license.html a exporter.
3. Ouvrez le fichier plugin.xml, et exportez le plug-in avec l'assistant d'export illustré à la figure 2.32 en spécifiant un répertoire de destination (les plug-ins seront placés dans le sous-répertoire plugin du répertoire choisi).

Figure 2.32

Assistant de construction de configuration



4. Dans l'onglet Options, spécifiez si le code source doit être inclus en même temps que les binaires (option Include source code) ou sous forme de Jar individuel pour chaque plug-in. Cliquez sur Finish.

Vous avez achevé la construction de la distribution de l'application de démo à l'aide des outils de déploiement prévus avec l'outillage RCP. Vous pourrez vous en servir comme base pour des applications riches plus élaborées.

En résumé

Cette prise de connaissance de l'atelier PDE d'Eclipse vous a permis, en partant d'un exemple simple, d'apprécier toute la richesse de l'environnement de développement de plug-in sur lequel est bâti Eclipse.

Vous avez mis en œuvre les nombreux composants offerts par la plate-forme (boîtes de dialogue, barres d'outils et de menus, etc.) pour créer des applications riches avec Eclipse RCP.

Vous verrez au chapitre suivant un autre aspect du développement avec Eclipse : le développement coopératif à l'aide de CVS et Subversion.

3

Support du développement en équipe sous Eclipse avec CVS

Ce chapitre est dédié à la mise en œuvre de la solution logicielle de gestion des versions et de gestion collaborative des sources CVS.

Après un rappel de la problématique du développement en équipe et des concepts associés (notion de référentiel de développement et d'architecture SCMS), nous détaillerons l'outil Open Source CVS (Concurrent Versions System) et son intégration dans Eclipse.

Comme nous le verrons, Eclipse propose une perspective CVS dédiée, appelée Exploration du référentiel CVS, qui permet de gérer les échanges et le contenu des projets stockés sous CVS de manière totalement intégrée à l'IDE.

L'une des fonctionnalités remarquables de l'IDE Eclipse est sa capacité à gérer nativement les ressources des projets en équipe à travers un référentiel partageable et des vues dédiées. Eclipse supporte notamment la gestion de versions, qui permet au développeur de comparer sa version locale avec celle stockée dans le référentiel de développement et de remplacer ses ressources locales par une version plus récente du projet s'il le désire.

Nous introduirons en fin de chapitre Subversion, parfois abrégé SVN, qui apparaît comme le digne successeur de CVS, dont il comble certains manques avérés qui nuisaient à son développement.

Pour des raisons de clarté, nous nous référons à la version Eclipse Callisto en langue française.

Ces concepts peuvent être appliqués sans difficulté particulière à Eclipse Europa en version anglaise, puisque le support de la langue française n'est pas disponible à l'heure ou nous mettons sous presse.

Développement collaboratif avec CVS

Eclipse est un environnement de développement intégré fondé sur la notion de fichier. Il maintient un workspace, ou espace de travail, dans lequel l'ensemble des données du projet sont stockées dans des fichiers. Cet espace contient la structure du projet sur

laquelle travaille le développeur d'une équipe. À la différence de VisualAge for Java, cette structure s'appuie sur le système de fichiers propre à l'OS cible, ainsi que sur les fichiers journaux et les métadonnées des plug-ins installés.

Chaque fois qu'une ressource est modifiée dans le projet, elle est stockée dans le SF (système de fichiers) local. Il est en outre possible de stocker l'espace de travail en partage sur le réseau.

Ce mécanisme de gestion des sources à partir du SF présente les deux avantages suivants :

- Intégration facile avec les autres outils, puisque le travail s'effectue directement avec le SF de la machine hôte.
- Meilleure flexibilité dans la gestion des sources et du processus de développement.

L'historique local d'Eclipse

Eclipse conserve un historique des modifications locales, lesquelles peuvent dès lors être réinitialisées à leur état antérieur.

L'historique local de chaque fichier est mis à jour automatiquement lorsque vous créez ou modifiez un fichier. Chaque fois que vous sauvegardez un fichier, une copie de son état précédent est conservée. Cela vous permet de comparer l'état courant du fichier avec l'état précédent et de remplacer le fichier par son état antérieur.

Dans l'historique local, chaque état est identifié par la date et l'heure de sa sauvegarde, comme l'illustre la figure 3.1.

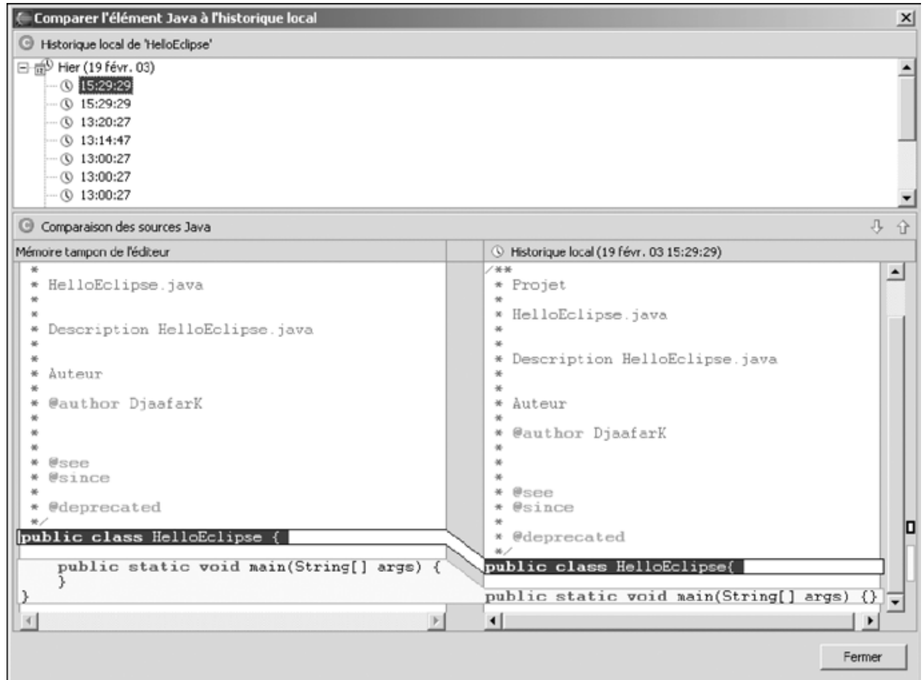


Figure 3.1

Vue de l'historique de l'espace de travail d'Eclipse

> **Historique et fichiers**
> Les fichiers peuvent posséder un historique local, mais pas les dossiers ni les projets.

Eclipse vous permet de comparer et de remplacer votre code courant par une des versions gérées localement. Pour voir l'historique d'un fichier, il vous suffit, à partir du menu contextuel du fichier, de sélectionner Historique local puis Comparer à.

Vous pouvez sélectionner dans la liste différents états du fichier et les comparer à la version courante de ce dernier. Vous pouvez aussi remplacer n'importe quelle portion de cette version courante du fichier par celle figurant dans l'historique local (voir figure 3.1).

> **Historique local**
> L'historique local est indépendant de l'environnement de développement d'équipe et du référentiel partagé.

Il est possible de modifier les paramètres de conservation des fichiers, exprimés en jours, ainsi que la taille allouée à l'historique en sélectionnant Préférences, Plan de travail et Historique local.

Le travail en équipe avec Eclipse

Il est rare qu'un développeur travaille seul à l'élaboration d'une application. Le plus souvent, il fait partie d'une équipe de développement.

Ce travail collectif nécessite un système de contrôle des versions supportant les fonctionnalités suivantes :

- historique des modifications effectuées par les membres de l'équipe ;
- moyen de coordonner et d'intégrer les tâches effectuées.

Un référentiel partageable est indispensable à ce travail en équipe. Ce référentiel est placé à l'endroit où l'ensemble des artefacts du projet sont stockés, maintenus et partagés par l'équipe de développement. Cela permet une gestion fine des accès et des modifications multiutilisateur des versions du projet ainsi que des snapshots, ou instantanés, logiciels, qui offrent une vision de l'état du projet à un instant donné.

La communication entre le référentiel et les développeurs s'effectue à travers un réseau local ou distant.

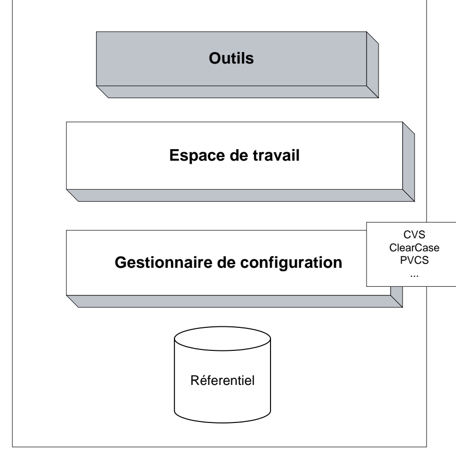
L'architecture de gestion de configuration intégrée

Dans Eclipse, les développeurs effectuent leur travail dans des espaces de travail séparés. Périodiquement, ils effectuent leurs modifications dans l'espace partageable du gestionnaire de configuration SCMS (Software Configuration Management System). Ce dernier doit au préalable avoir été installé et configuré.

La plate-forme Eclipse possède une architecture plugable, ouverte aux systèmes de gestion de configuration tiers, ou SCMS, comme l'illustre la figure 3.2.

Figure 3.2

Architecture d'Eclipse et SCMS



Eclipse est livré avec un adaptateur standard pour le SCMS CVS (Concurrent Versions System). CVS est disponible sur un certain nombre de plates-formes, dont Windows, Linux, AIX, HP-UX et Solaris.

Principes de gestion des ressources

Avant d'aborder dans le détail la mise en œuvre de la gestion de configuration avec CVS, il est nécessaire de rappeler un certain nombre de notions fondamentales sur ce sujet très important pour le développement courant d'un projet.

Notion de branche

Comme nous l'avons vu, l'architecture du modèle de développement d'Eclipse permet l'utilisation d'un référentiel plugable tiers au lieu d'un référentiel propriétaire. Périodiquement, les développeurs effectuent leurs modifications dans un espace partagé, appelé branche, du système de gestion de configuration logicielle qui a été auparavant installé et configuré sur le serveur.

Le processus de mise à disposition de ressources aux autres membres de l'équipe par le biais du référentiel est appelé *release.* Cela équivaut à une copie des fichiers modifiés vers le répertoire partagé. Le terme *team stream* est souvent utilisé dans la littérature consacrée à Eclipse pour désigner l'espace commun de travail de l'équipe.

Toutes les modifications sont marquées *outgoing changes,* ou modifications sortantes, lorsque vous effectuez une synchronisation avec le référentiel. Le processus inverse, appelé *catch up,* de récupération des ressources à partir du référentiel de développement consiste à copier les modifications effectuées par les autres membres de l'équipe dans l'espace de travail local.

Branche et HEAD

Chaque référentiel possède au moins une branche, ou répertoire partagé principal du projet, appelée par défaut Head dans CVS.

Notion de version

Lorsqu'une ressource est versionnée, une copie non modifiable est dite *released* dans le référentiel. Il est alors possible, quoique non recommandé, de versionner un fichier ou un répertoire.

Le versionnement le plus courant d'un projet est la sauvegarde de l'intégralité des versions des ressources du projet. Il est possible de choisir le versionnement à partir d'un espace de travail personnel ou d'une branche. Le plus recommandé est le versionnement à partir de l'espace de travail personnel, car le développeur est censé connaître l'état des versions des ressources projet qui doivent être rendues released dans la version finale.

Il peut arriver qu'un même projet apparaisse dans différentes branches, une branche représentant l'état en cours du projet, et les ressources évoluant indépendamment les unes des autres. Cela n'a rien d'anormal pour un projet qui possède, par exemple, deux états, un état de maintenance corrective et un projet intégrant de nouvelles fonctionnalités.

Les modifications validées dans la branche n'ont aucun effet sur les autres branches jusqu'à l'opération de fusion (merge) des modifications à partir de cette branche dans une autre branche ou dans le head.

Notion de rôles

Contrairement à d'autres référentiels de développement propriétaires, l'espace de travail Eclipse ne possède aucune notion de droit sur les ressources du projet. Les droits en écriture sont simplement contrôlés par les permissions d'écriture sur les répertoires contenant la ressource.

N'importe quel membre de l'équipe de développement ayant les droits d'accès sur une ressource peut la modifier, voire la supprimer. Chaque utilisateur du système de gestion de configuration logicielle possède un compte sur le serveur où le référentiel d'équipe a été créé et configuré par l'administrateur du système de configuration logicielle en charge des développements. Il y accède par le biais d'un client CVS intégré à Eclipse.

Ce système de configuration logicielle installé sur le serveur permet de gérer et d'administrer les comptes CVS client à travers un référentiel d'équipe, soit ici CVSNT, un gestionnaire de version serveur Open Source compatible avec à la fois simple et fiable d'utilisation.

Scénarios de développement en équipe dans Eclipse

Importante pour le maintien de la cohérence des livrables, la coordination des développements au sein d'une équipe est rendue possible par le SCMS (Software Configuration Management System) intégré à Eclipse.

Le modèle de gestion des conflits qui peuvent survenir lorsque deux développeurs effectuent des modifications sur une même ressource est dit « modèle optimiste de versions concourantes ». Le modèle suppose que les conflits liés aux modifications qui peuvent survenir dans l'équipe sont rares.

Ce modèle de collaboration est bien dans l'esprit de la communauté Open Source, où les développeurs restent souvent déconnectés du référentiel d'équipe pendant de longues périodes et synchronisent occasionnellement leur référentiel avec le répertoire partagé, ou branche.

Scénario de développement séquentiel

La figure 3.3 illustre un modèle de développement séquentiel :

1. Le développeur 1 crée un fichier, FIC1, puis valide ses modifications courantes dans la branche de développement.
2. Le développeur 2 récupère la nouvelle version du fichier FIC1 dans son espace de travail à partir de la branche de développement du projet.
3. Il modifie à son tour le fichier et valide ses modifications dans la branche.
4. Le développeur 1 récupère les modifications effectuées par le développeur 2 sur le fichier FIC1 et les valide dans la branche.

Le cycle continue jusqu'à ce que le fichier FIC1 soit finalisé. Ce scénario favorise l'absence de conflits dans la gestion commune des ressources.

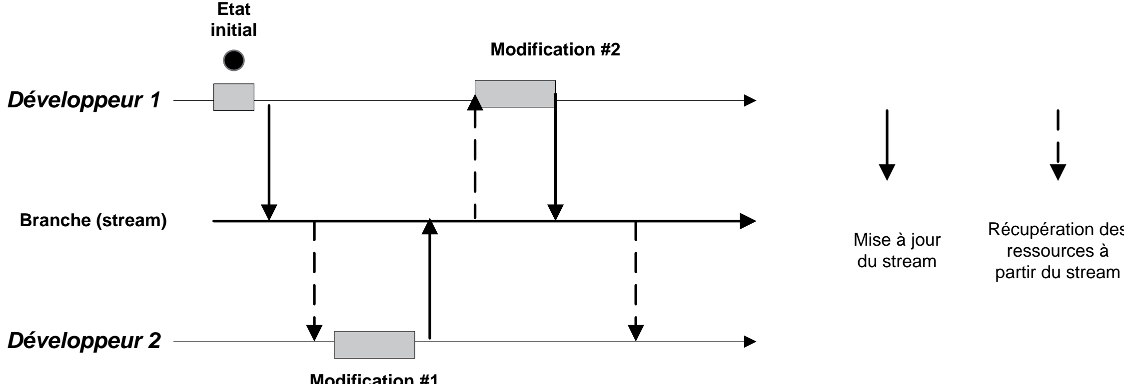


Figure 3.3

Développement séquentiel utilisant le référentiel d'équipe

Scénario de développement parallèle sur une seule branche

Le scénario de développement parallèle sur une seule branche *(single stream)* est plus courant que le précédent. Il repose sur le fait que plus d'un développeur travaille généralement en même temps sur un même fichier dans une même branche du développement.

La figure 3.4 illustre ce scénario, moins optimiste que le précédent.

1. Les développeurs 1 et 2 récupèrent la version 1.3 du fichier à partir de la branche commune de développement.
2. Le développeur 1 modifie le fichier et le met à disposition dans la branche.
3. Le développeur 2, qui travaille également sur le même fichier, est prêt à le mettre à disposition dans l'espace partagé mais doit d'abord fusionner (merge) ses modifications avec celles du développeur 1 avant de valider ses changements.

CVS permet de détecter automatiquement les conflits éventuels en comparant les fichiers et en effectuant une fusion des modifications.

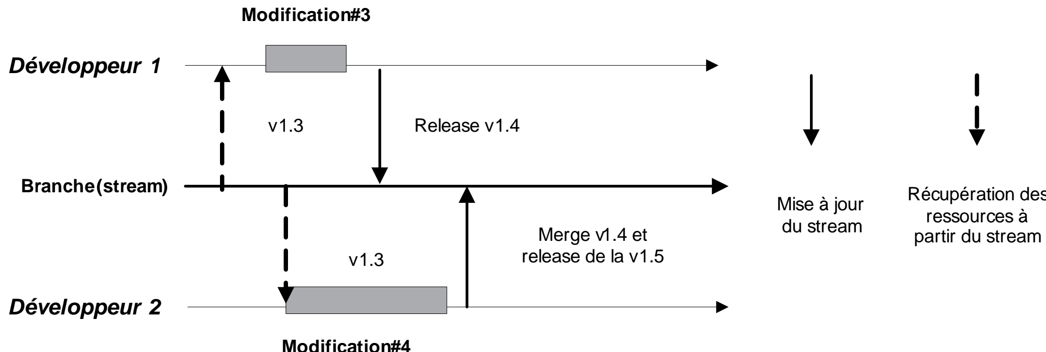


Figure 3.4

Développement séquentiel en utilisant le référentiel d'équipe

Scénario de développement parallèle sur plusieurs branches

Le scénario de développement parallèle sur plusieurs branches *(multiple stream)* est un cas particulier. Il survient dans un type de scénario où, par exemple, un développement consiste en la maintenance d'une application simultanément au développement de nouvelles fonctionnalités.

Dans ce cas, il faut créer une nouvelle branche dans le processus de développement courant, branche qui va permettre la fusion des développements issus du projet de maintenance effectué par un des développeurs de l'équipe et du projet d'évolution des fonctions à la charge d'un autre membre de l'équipe.

Mise en œuvre de CVS dans Eclipse

Maintenant que vous avez à l'esprit les concepts liés à la mise en œuvre d'un système de gestion de configuration des ressources en équipe, vous pouvez entrer dans le détail du SCMS CVS (Concurrent Versions System) livré avec Eclipse.

CVS, dont les premières implémentations sous système UNIX remontent à 1985, est un outil Open Source que l'on ne présente plus au sein de la communauté du libre.

Disponible sur *www.cvshome.org,* il permet à plusieurs personnes de partager simultanément des ressources de type fichier et de gérer l'historique des modifications, ou révisions, effectuées.

CVS offre les fonctionnalités suivantes :

- accès client-serveur, permettant au développeur d'accéder à la dernière version de son code à partir de n'importe quel lieu connecté à Internet ;
- mécanisme de réservation des ressources non exclusif permettant d'éviter les conflits ;
- outils de gestion des ressources disponibles sur la plupart des plates-formes (outils de synchronisation des ressources avec le référentiel, gestion des versions et des conflits, etc.).

CVS maintient un historique complet des codes source sous une forme arborescente montrant l'intégralité des modifications effectuées. À chaque commit envoyé au repository, un nouvel état de l'arborescence est créé ; cet état est appelé révision.

Chaque révision est représentée par un nombre qui s'incrémente. La révision initiale d'un repository créé est zéro. Il est ainsi possible de comparer les fichiers et de les remplacer par une des révisions disponibles à chaque moment. La révision la plus récente dans le référentiel est désignée par head.

Installation et configuration de CVSNT

Avant de partager des projets dans une équipe de développement, il est nécessaire de spécifier un emplacement valide sur un serveur partagé. Ce serveur doit être au préalable correctement configuré avec un repository valide.

Pour disposer d'un serveur CVS sous NT, vous pouvez utiliser la version CVSNT *(http://www.cvsnt.org)*, implémentée sous forme de service NT et supportant les différents protocoles d'authentification NT, dont Kerberos.

Rappelons qu'un référentiel, au sens CVS, est un magasin de données persistant qui coordonne les accès multiutilisateur à des projets et à leur contenu.

Dans un référentiel, les projets peuvent être non modifiables (une seule version de projet) ou modifiables (projet d'une branche). La communication entre le référentiel et les clients de l'espace de travail est assurée *via* des réseaux locaux ou longue distance. Des protocoles d'authentification fournis par des outils externes peuvent être utilisés par CVS.

Le produit Open Source CVSNT présente l'avantage de s'installer facilement sur un serveur Windows NT (NT 4 Serveur ou client avec le service pack 6) ou 2000 (Serveur ou Professionnel) et d'offrir une compatibilité complète avec la version de référence de CVS. La version utilisée ici est la 1.11, disponible à l'adresse *http://www.cvsnt.org/*.

L'installation du serveur CVS ne présente pas de difficultés particulières :

1. Après téléchargement de la version CVSNT sur le site *http://www.cvsnt.org*, choisissez la version précompilée.
2. Cliquez sur le fichier cvsnt-1.11.1.3-72.exe, qui est la version la plus stable et la plus récente, et suivez la procédure d'installation standard sur votre serveur NT.

   La figure 3.5 illustre les composants disponibles pour une installation sur un serveur Windows 2000 Professionnel, Serveur ou XP, avec le service pack 6.

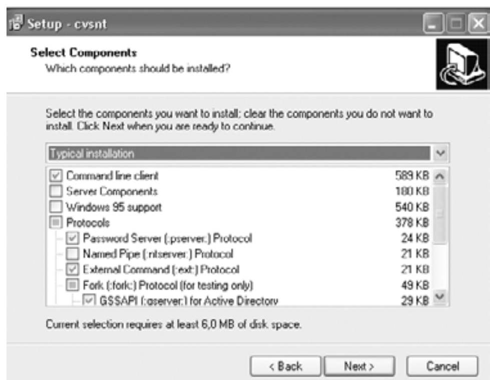


Figure 3.5

Liste des composants CVSNT installés

3. Veillez à posséder les privilèges administrateur adéquats et à ce que votre système de fichiers soit de type NTFS. C'est très important pour la suite de l'installation.

4. Le chemin de l'installation CVSNT et de la commande cvs.exe est normalement ajouté à la variable path de votre serveur. Si ce n'est pas le cas, ajoutez-les à la variable path à partir du Panneau de configuration Windows (c:\programs\cvsnt).

> **Clients Windows NT-2000-95/98**
>
> Si vous souhaitez utiliser un client NT ou Windows 2000, il est préférable d'utiliser le fichier cvs.exe fourni avec la distribution de CVSNT, cette version supportant l'ensemble des fonctionnalités de la version serveur. Les clients Windows 95/98 doivent utiliser le fichier cvs95.exe fourni après installation de CVSNT.

5. Après installation de CVSNT sur votre serveur, sélectionnez Démarrer, Programmes, CVS for NT et Service Control Panel. Vous devez obtenir la boîte de dialogue illustrée à la figure 3.6, permettant de configurer le référentiel de développement de l'équipe.

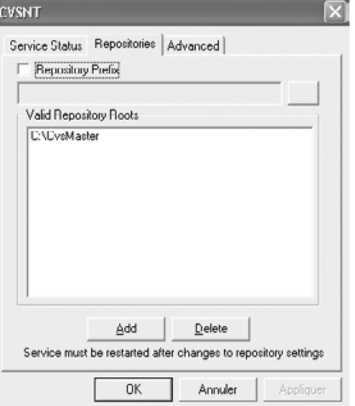
Figure 3.6

Panneau de contrôle de CVSNT

6. Créez un répertoire CVS sur votre disque, par exemple c:\CvsMaster, et attribuez des droits offrant un contrôle total aux comptes utilisateur, y compris le compte SYSTEM.
7. Dans l'onglet Repositories, spécifiez le chemin d'accès complet à votre référentiel de développement.
8. Utilisez le bouton Add pour ajouter un nouveau référentiel, ici Test, en prenant soin au préalable de configurer le préfixe du référentiel, comme illustré à la figure 3.7.

Figure 3.7

Paramétrage du référentiel CVS



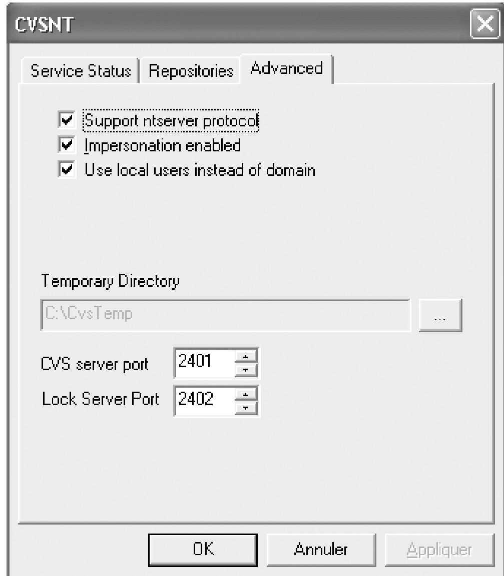
Emplacements des référentiels

Lorsque vous utilisez des préfixes de référentiels — sans être obligatoire, cela s'avère bien pratique si vous disposez de plusieurs référentiels —, ces derniers doivent être localisés sous le même répertoire racine (préfixe). Sur CVSNT, il n'existe qu'un seul préfixe de référentiel. Nous avons noté sur la version de CVSNT en notre possession des difficultés d'accès au référentiel CVS à partir d'Eclipse avec cette option. Il est donc conseillé de ne pas y recourir et d'utiliser uniquement l'option Valid Repository Root ou de vous rendre sur le site de l'auteur du produit *(bo.berglund@telia.com)* pour plus d'information.

9. Dans l'onglet Advanced, précisez les protocoles supportés :
   - « Server side support for ntserver protocol ». Support du protocole ntserver initialement utilisé par CVS et remplacé à présent par le protocole sspi, beaucoup plus sécurisé, supportant le cryptage et fonctionnant au-dessus de TCP/IP.
   - « Impersonation enabled ». Cette option permet à CVSNT d'agir au nom de l'utilisateur qui effectue la commande, notamment dans les permissions associées aux fichiers du système.
   - « Use local users instead of domain ». Utilisez cette option si vous ne possédez pas de domaine NT réservé pour l'authentification des utilisateurs. Ces derniers doivent posséder un compte, ou alias (voir plus loin), valide sur l'hôte hébergeant CVSNT.
10. Précisez dans cette étape le répertoire temporaire CVS, ici C:\CvsTemp, et attribuez tous les privilèges d'accès, y compris pour le compte SYSTEM, comme illustré à la figure 3.8.

Figure 3.8

Paramétrage du référentiel CVS



11. Cliquez sur OK pour que les modifications prennent effet.
12. Redémarrez CVSNT (onglet Service Status) pour prendre en compte les modifications.

Ajout d'utilisateurs avec pserver

À ce stade, vous pouvez tester la liaison avec le référentiel que vous venez d'initialiser en utilisant la méthode de connexion pserver. À partir d'une session DOS, il vous suffit d'entrer les commandes suivantes, où djafaka représente le nom de l'hôte et karim un compte possédant les privilèges d'administrateur NT de la machine :

```
C:\ set cvsroot=:pserver:karim@djafaka:/Test
C:\ cvs login karim
Logging in to :pserver:karim@djafaka:2401:/Test
CVS password: ***
```

Une fois le test de connexion réussi, vous pouvez ajouter ou supprimer des comptes CVS à partir du compte karim ou de tout compte possédant des privilèges d'administrateur équivalents.

CVS n'accepte pas d'utilisateur non connecté à un compte réel existant. Vous pouvez contourner cette contrainte en utilisant un alias vers un compte réel NT existant en utilisant la commande passwd. En voici la syntaxe complète :

```
cvs passwd [-a] [-x] [-X] [-r compte NT existant] [-R] [-D domaine] <compte cvs>
```

Les options de cette commande sont les suivantes :

- -a : ajoute un utilisateur au système.
- -x : désactive l'utilisateur du système.
- -X : supprime l'utilisateur du système.
- -r : nom d'alias d'un utilisateur réel.
- -R : supprime l'alias d'un utilisateur réel.
- -D : utilise un mot de passe du domaine.

La commande suivante ajoute l'identifiant CVS christophe, qui est en fait un alias système obtenu à partir du compte NT réel philippe :

```
cvs passwd -r philippe -a christophe
```

L'ajout d'utilisateurs CVS est requis seulement si vous envisagez d'utiliser le protocole pserver, qui est le protocole par défaut utilisé par CVS et par Eclipse (voir plus loin). Pour des raisons évidentes de sécurité, nous recommandons d'utiliser le protocole SSPI — surtout si vous accédez à votre référentiel *via* Internet —, qui peut facilement traverser les pare-feu installés, contrairement au protocole de connexion pserver.

Configuration du référentiel CVS dans Eclipse

Votre serveur CVS est maintenant configuré sur la machine hôte pour créer un emplacement de référentiel correct dans l'espace de travail Eclipse. Vous pouvez donc configurer la connexion au référentiel à partir de l'outil Eclipse, référentiel que vous avez précédemment initialisé.

Un emplacement de référentiel n'est pas une connexion réelle mais une description de l'endroit où se trouve le référentiel. Les connexions sont ouvertes et fermées, selon le cas, lors de l'exécution des opérations courantes de mise à jour du référentiel ou de récupération d'informations dans le référentiel d'équipe CVS. Ces connexions s'appuient sur les informations que vous avez indiquées dans l'emplacement du référentiel.

Pour configurer le référentiel sous Eclipse, il vous suffit d'utiliser l'explorateur de référentiel (Perspective CVS Repositories) et d'ajouter un référentiel CVS à partir de la boîte de dialogue illustrée à la figure 3.9.

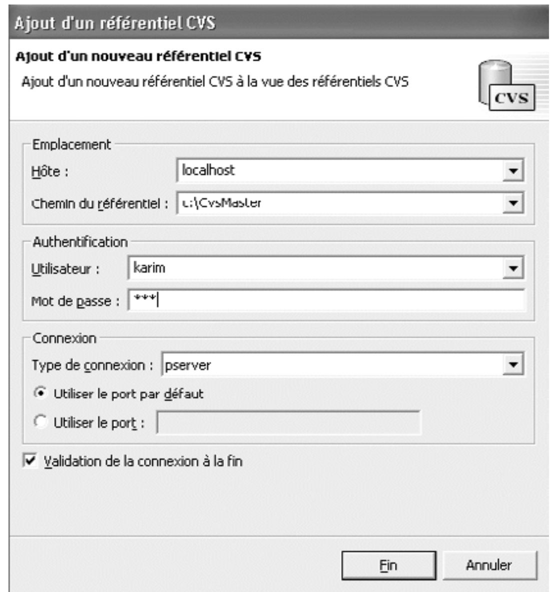


Figure 3.9

Configuration du référentiel CVS

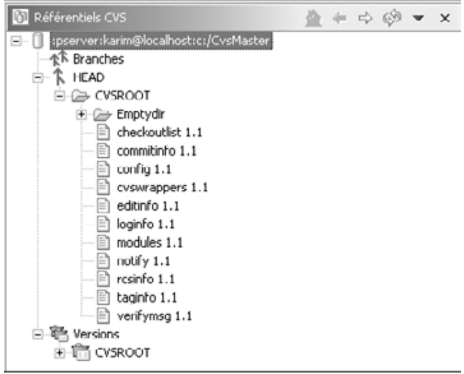
Les paramètres suivants de configuration de la connexion au référentiel CVS sont obligatoires :

- **Hôte.** Nom du serveur hôte.
- **Chemin du référentiel.** Chemin d'accès au référentiel partagé.
- **Utilisateur.** Nom de l'utilisateur accédant à la ressource précédemment configurée sur le serveur.
- **Mot de passe.** Mot de passe associé à l'utilisateur.
- **Type de connexion.** Ce paramètre permet de sélectionner le protocole d'authentification du serveur. Trois protocoles sont supportés par le client CVS livré dans Eclipse :
  - **pserver.** Connexion spécifique de CVS, recommandée pour l'utilisation courante de CVS avec peu de problématique de sécurité.
  - **extssh.** Client SSH 1.0 inclus dans Eclipse.
  - **ext.** Méthode de connexion utilisant un outil externe, tel que SSH, pour se connecter au référentiel. L'outil utilisé par ext est configuré dans la page de préférences, *via* Equipe, CVS et Méthode de connexion externe.
- **Port.** Port par défaut ou spécifique si l'hôte par défaut en utilise un.

Si votre configuration a été correctement validée, la perspective Référentiels CVS doit afficher la liste des référentiels précédemment configurés et installés sur le serveur, comme illustré à la figure 3.10.

Figure 3.10

La perspective Référentiels CVS après connexion



Remarquez que la branche head figure par défaut dans l'explorateur de référentiels. Votre référentiel est maintenant accessible à partir d'Eclipse, et vous pouvez récupérer les ressources d'un projet Eclipse dans votre espace de travail ou enregistrer une version de votre travail dans le référentiel.

Synchronisation du projet

Vous allez créer un projet CvsTest de type Java contenant une seule ressource, une classe Java HelloEclipse, contenue dans un package com.eclipse.test, par exemple. Vous mettrez ce projet à la disposition des autres membres de l'équipe.

1. Dans la vue Navigateur associée à la perspective Java, sélectionnez le projet CvsTest.
2. Dans le menu contextuel du projet, sélectionnez Equipe puis Partager le projet, comme illustré à la figure 3.11.

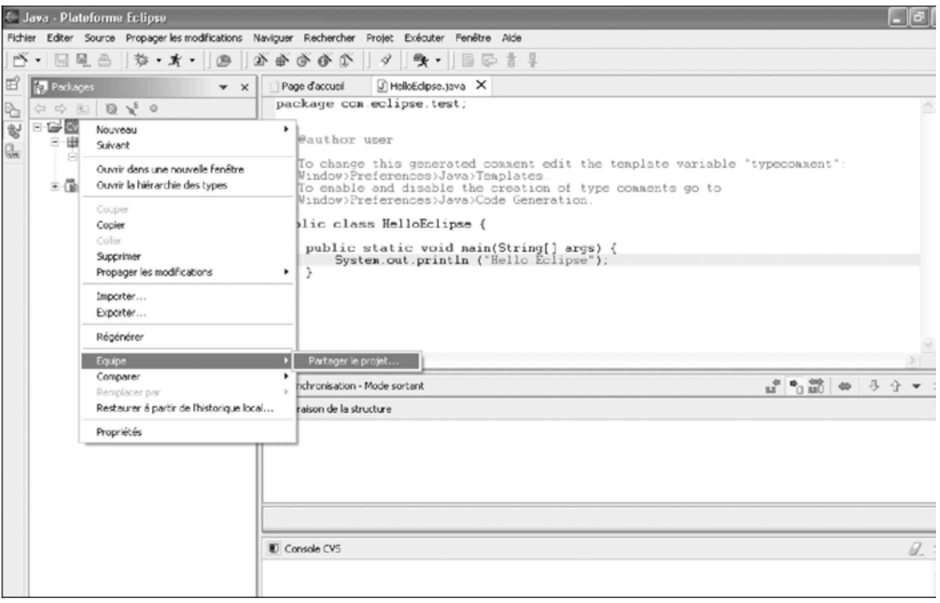


Figure 3.11

Sélection de la fonction Partage de projet

3. Dans la page de l'assistant de partage de projet, sélectionnez l'emplacement créé précédemment, comme illustré à la figure 3.12.

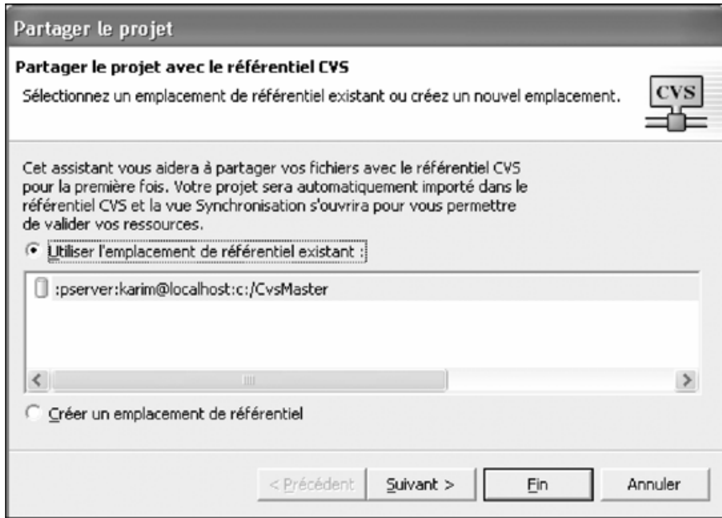


Figure 3.12

Assistant de partage du projet

4. Choisissez le nom du projet local comme nom de projet distant, ou entrez un autre nom.
5. Cliquez sur Fin pour partager le projet avec le référentiel. Le projet s'affiche dans la vue Synchronisation avec l'ensemble des commandes CVS usuelles (voir figure 3.13).

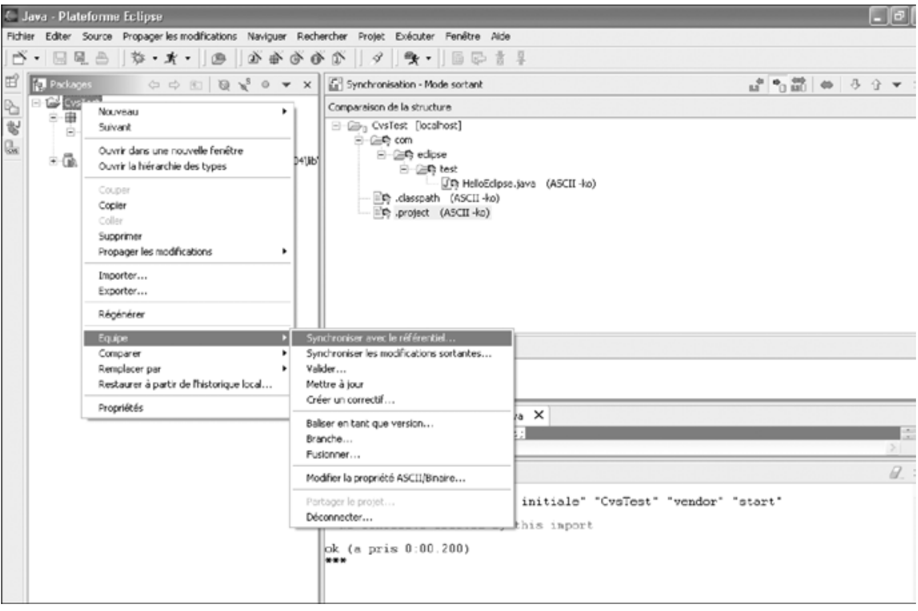


Figure 3.13

Vue Synchronisation après partage du projet

> **La vue Synchronisation**
>
> Si le projet existe déjà à distance, la vue Synchronisation affiche les conflits de fichiers en local et à distance.

Le message qui apparaît dans la barre de messages, au bas de l'écran, signale l'absence de conflit, puisque le projet vient d'être partagé.

Dans le volet Comparaison de la structure, des icônes vous permettent d'afficher toutes les ressources à valider dans le référentiel. Vous pouvez modifier ces icônes au moyen d'objets décorateurs, qui permettent aux membres de l'équipe de distinguer les ressources configurées pour leur référentiel.

> **Décorateurs**
>
> Le client CVS utilise des décorateurs pour afficher indicateurs de modification, balises, mots-clés et révisions. Les décorateurs sont utiles pour ajouter des fonctionnalités à des types de ressources existants. De nombreuses vues standards de l'espace de travail affichent les décorateurs.

Les décorateurs peuvent être contrôlés par l'utilisateur par le biais de la boîte de dialogue Label Decorations, accessible *via* Préférences, Team puis CVS, comme illustré à la figure 3.14.

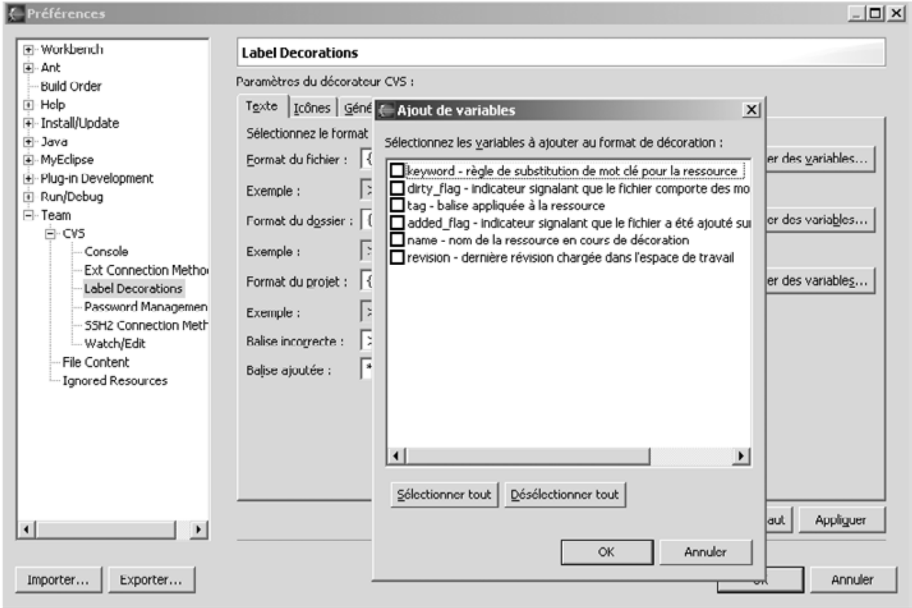


Figure 3.14

Les décorateurs d'intitulés dans Eclipse

Synchronisation du référentiel

Le développeur Eclipse ayant son projet affiché dans le plan de travail Eclipse doit pouvoir se synchroniser avec son référentiel de développement (« Synchroniser avec le référentiel »), mettre à jour ses fichiers (« Mettre à jour ») et les rendre accessibles aux

autres membres de l'équipe (voir figure 3.14) ou versionner son projet (« Baliser an tant que version »).

> **Mise à jour**
>
> Il est toujours préférable de mettre à jour les ressources avant de les valider, afin de résoudre les éventuels conflits entre les ressources de l'espace de travail et celles de la branche.

Vous pouvez définir des filtres pour :

- Visualiser uniquement les modifications entrantes. Les modifications entrantes proviennent de la branche. Si elles sont acceptées, elles remplacent la version de la ressource de l'espace de travail par la dernière version validée dans la branche.
- Visualiser uniquement les modifications sortantes. Les modifications sortantes proviennent de l'espace de travail. Si elles sont validées, elles modifient les ressources de la branche de sorte que celles-ci correspondent à celles de l'espace de travail.
- Visualiser les modifications entrantes et sortantes.

Dans cet exemple, s'agissant d'un nouveau projet, choisissez les modifications sortantes :

1. Faites un clic droit pour ouvrir le menu en incrustation, puis sélectionnez Equipe et Synchroniser avec le référentiel.
2. À partir du menu contextuel illustré à la figure 3.15, sélectionnez Ajouter au contrôle de version. Cette option indique à CVS de créer une entrée dans le référentiel de manière à commencer à gérer l'historique de ce fichier avant sa validation dans le référentiel.

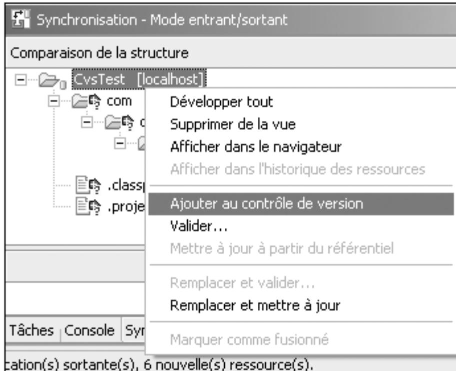


Figure 3.15

Assistant de contrôle de version Eclipse

Vous pouvez suivre le résultat des commandes CVS dans la console CVS dédiée (voir figure 3.16).

Figure 3.16

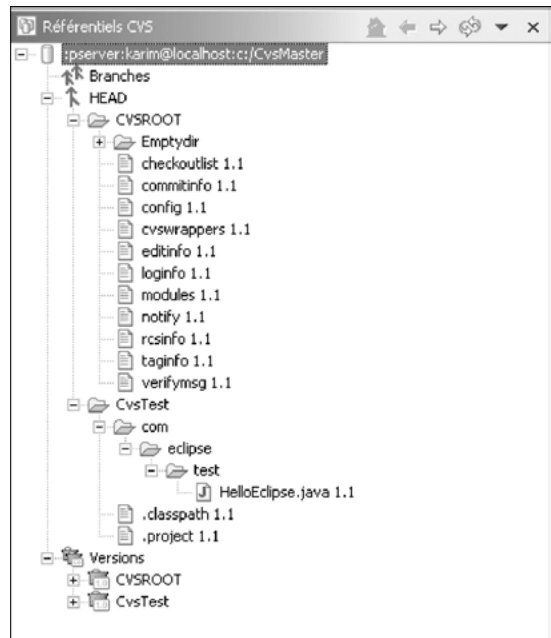
Console CVS

```
Console CVS
***
cvs add -ko ".project" "com/eclipse/test/HelloEclipse.java" ".classpath"
  cvs server: scheduling file `.project' for addition
  cvs server: scheduling file `com/eclipse/test/HelloEclipse.java' for addition
  cvs server: scheduling file `.classpath' for addition
  cvs server: use 'cvs commit' to add these files permanently
ok (a pris 0:00.521)
***
```

3. Sélectionnez l'option Valider pour valider dans le référentiel les modifications sortantes et les rendre visibles aux autres membres de l'équipe. Vous pouvez saisir un commentaire si vous le souhaitez.
4. Revenez dans la perspective Référentiels CVS, et faites un clic droit pour ouvrir le menu contextuel Régénérer la vue. Votre projet ainsi que toutes ses ressources figurent maintenant dans le référentiel C:\CvsMaster, comme illustré à la figure 3.17. Notez le numéro de révision initiale, 1.1.

Figure 3.17

Perspective Référentiels CVS après intégration du projet



> **Validation**
>
> Valider les modifications d'un projet ou des ressources associées équivaut à les copier depuis votre espace de travail vers la branche du référentiel. À chaque validation d'une ressource, un nouveau tag de révision est associé (1.1 pour le tag de révision initial, 1.2 pour la prochaine validation, et ainsi de suite).

Mise à jour et gestion des conflits avec CVS

Une fois votre projet intégré dans le référentiel et ce dernier disponible pour les autres membres de votre équipe, des conflits sur les ressources peuvent survenir. Ces conflits peuvent être résolus de manière automatisée ou manuelle.

Un conflit se produit lorsque vous modifiez en local une ressource pour laquelle une révision plus récente est disponible dans la branche du référentiel.

Dans un tel cas, vous pouvez effectuer l'une des opérations suivantes :

- Récupérer la modification de la branche et supprimer le travail effectué en local. Vous choisirez cette solution si vous avez effectué en local des modifications involontaires ou si vous découvrez que la révision du référentiel est meilleure que la vôtre. Soyez prudent lorsque vous écrasez vos modifications locales, puisque cela détruit le travail accompli.

- Valider votre modification en restituant la révision dans le référentiel. N'effectuez cette opération qu'avec une extrême prudence car elle se traduit par l'écrasement du travail d'autrui. De plus, il est possible que le travail écrasé ait des dépendances dans la branche.
- Fusionner votre travail avec la ressource du référentiel, en sauvegardant en local la ressource fusionnée. Vous pouvez valider le résultat de la fusion ultérieurement.

La plupart du temps, vous effectuerez une fusion, puisque les deux autres choix entraînent des pertes de travail.

Mode de résolution des conflits avec CVS

Lorsque vous travaillez sur un projet dans l'espace de travail, il se peut que d'autres membres de votre équipe valident des modifications dans la copie du projet située dans le référentiel. Pour obtenir ces modifications, vous pouvez « mettre à jour » l'espace de travail de sorte qu'il corresponde à l'état de la branche.

Il convient dans tous les cas de procéder à une mise à jour systématique avant toute validation, de sorte à vous assurer que l'état le plus récent du référentiel est disponible. La commande de mise à jour peut alors être émise de l'une des deux façons suivantes :

- en choisissant Equipe puis Mettre à jour à partir du menu contextuel ;
- en passant par la vue Synchronisation.

L'action de ces deux commandes CVS étant différente, il est important de connaître les trois types de modifications entrantes qui peuvent survenir sur un projet CVS :

- Modification non conflictuelle. Se produit lorsqu'un fichier est modifié à distance mais pas en local.
- Modification conflictuelle pouvant être fusionnée automatiquement. Se produit lorsqu'un fichier ASCII est modifié à distance et en local. Le fichier comporte en ce cas des modifications locales non validées.
- Modification conflictuelle ne pouvant être fusionnée automatiquement. Se produit lorsqu'une ou plusieurs lignes d'un fichier binaire est modifiée à distance et en local (les fichiers binaires ne peuvent jamais être fusionnés automatiquement).

Mise à jour des ressources à partir du menu Mettre à jour

Lorsque vous sélectionnez Equipe puis Mettre à jour, le contenu des ressources locales est mis à jour avec les trois types de modifications entrantes présentés précédemment :

- Pour les modifications non conflictuelles pouvant être fusionnées automatiquement, aucune autre intervention n'est requise.
- Pour les conflits pouvant être fusionnés automatiquement, la ressource locale modifiée est placée dans un fichier au préfixe .# dans le cas où la fusion automatique ne correspondrait pas à ce que souhaite l'utilisateur.
- Pour les conflits ne pouvant être fusionnés automatiquement, soit les fichiers sont fusionnés avec la ressource locale à l'aide d'un texte de marquage spécifique (pour les fichiers ASCII), soit la ressource locale modifiée est placée dans un fichier au préfixe .# (pour les fichiers binaires).

L'ensemble de ces réglages est accessible à partir du menu Préférences d'Eclipse, option CVS, comme illustré à la figure 3.18.

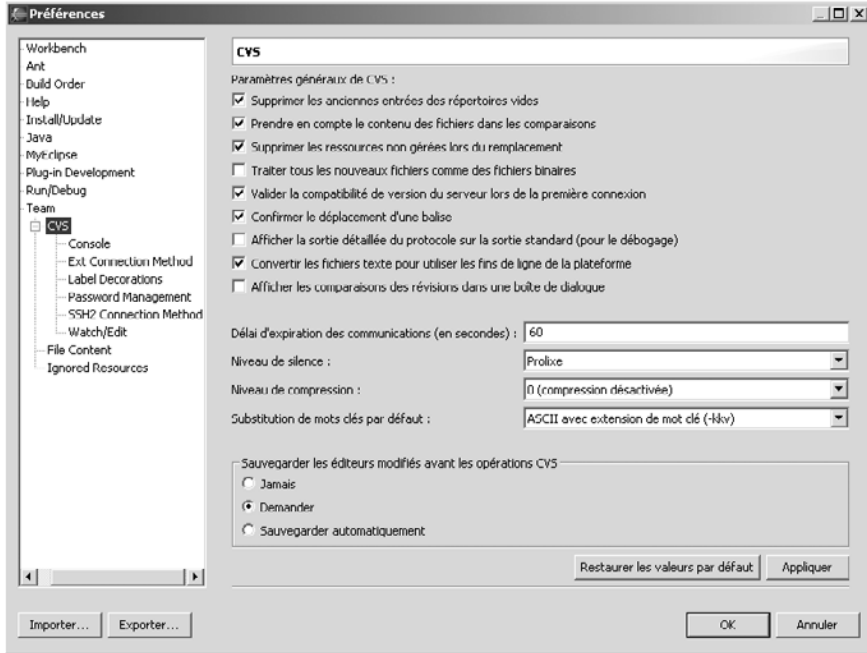


Figure 3.18

Panneau des préférences de CVS

Mise à jour des ressources à partir de la vue Synchronisation

La vue Synchronisation fournit un emplacement unique dans lequel vous pouvez visualiser les modifications que vous avez apportées et celles apportées et validées par d'autres personnes dans le référentiel.

Pour ouvrir la vue Synchronisation en mode entrant, procédez de la façon suivante :

1. Dans la vue Navigator, sélectionnez les ressources à mettre à jour.
2. Dans le menu des ressources sélectionnées, choisissez Equipe puis Synchroniser avec le référentiel. La vue Synchronisation s'affiche.
3. Dans la barre d'outils de la vue Synchronisation, cliquez sur le bouton du mode entrant pour écarter toutes les ressources modifiées (modifications sortantes) de l'espace de travail dont vous disposez.

   En mode entrant, les modifications qui ont été validées dans la branche depuis la dernière mise à jour s'affichent. Cette vue indique le type de chaque modification entrante (non conflictuelle, conflictuelle pouvant être fusionnée automatiquement ou conflictuelle ne pouvant être fusionnée automatiquement).
4. Pour valider les fichiers qui sont en conflit, sélectionnez Remplacer et valider. La copie de l'espace de travail de la ressource est fusionnée avec le référentiel, et toutes les modifications entrantes sont supprimées.

> **Validation**
>
> Le contenu du référentiel n'est pas modifié lors de la mise à jour. Lorsque vous acceptez les modifications entrantes, ces dernières sont intégrées à votre espace de travail. Le référentiel n'est modifié que si vous validez des modifications sortantes.

Versionnement des ressources d'un projet Eclipse

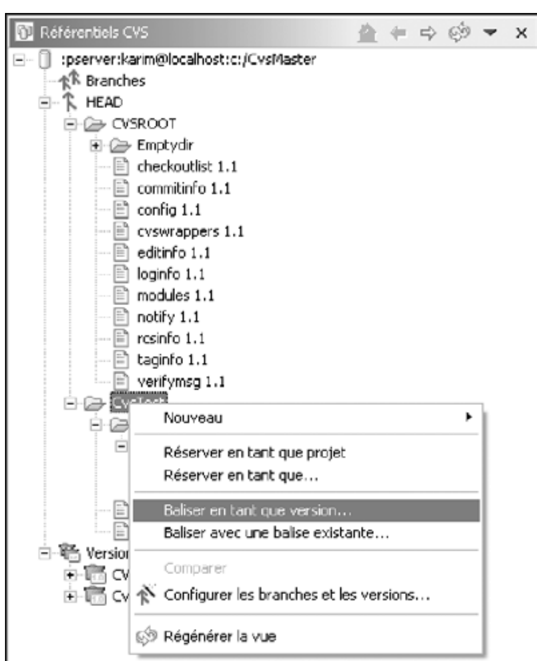
Les ressources sont versionnées afin de capturer un instantané de leur état à un moment donné. Vous pouvez versionner les ressources de CVS en les balisant à l'aide d'un libellé de version.

Lorsqu'une ressource est versionnée, une copie non modifiable de cette ressource peut être extraite du référentiel. Pour mettre en œuvre ce concept dans Eclipse, reprenez l'exemple utilisé précédemment :

1. Dans la perspective Référentiels CVS, sélectionnez le projet que vous souhaitez versionner, CvsTest par exemple.
2. Sélectionnez « Baliser en tant que version » à partir du menu contextuel, comme illustré à la figure 3.19, et entrez une balise de version, Beta1, par exemple.

Figure 3.19

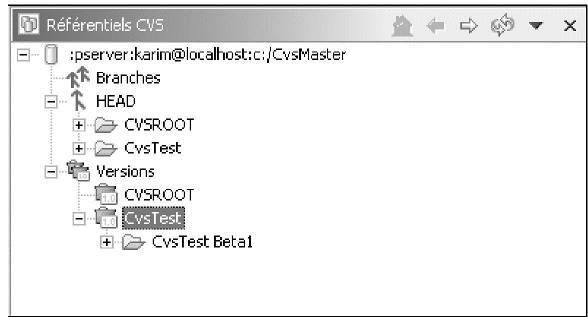
Versionnement de projet



3. Cliquez sur OK. La nouvelle version s'affiche sous le menu Versions, comme illustré à la figure 3.20.

Figure 3.20

Versionnement du projet après balisage



Ajout d'une version à l'espace de travail

Pour réserver un projet à partir d'un référentiel CVS dans le plan de travail et ajouter une version à votre espace de travail, procédez de la façon suivante :

1. Sélectionnez la version que vous souhaitez charger dans la perspective Référentiels CVS.
2. Sélectionnez CvsTest Beta1, et cliquez sur l'option du menu contextuel « Réserver en tant que projet », comme illustré à la figure 3.21.

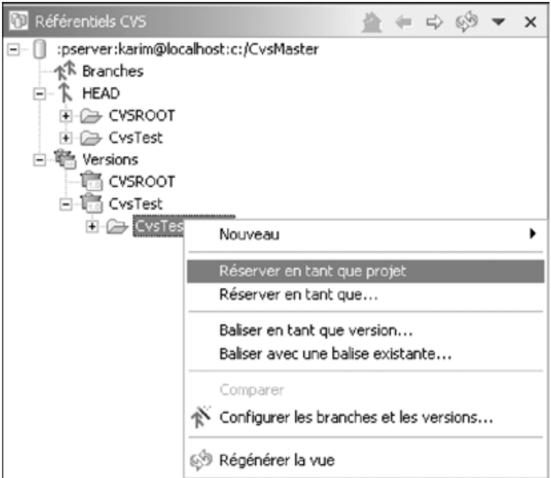


Figure 3.21

Réservation d'une version dans l'espace de travail Eclipse

Sans surprise, un message vous indique que la ressource CvsTest existe déjà dans votre espace de travail et vous propose de la remplacer.

Le gestionnaire de versions Subversion

Conçu pour remplacer CVS, le gestionnaire de versions Subversion (svn) propose davantage de fonctionnalités que son prédécesseur, notamment la gestion du renommage ou les copies de fichiers et de répertoires.

Subversion peut s'utiliser en ligne de commandes ou par le biais d'extensions disponibles pour certains IDE, comme `subclipse` pour `Eclipse`.

Les sections qui suivent détaillent les fonctionnalités les plus intéressantes de ce gestionnaire de versions et souligne les bénéfices qu'il procure en comparaison de CVS.

Présentation

Subversion a été lancé en 2004 par des développeurs de CVS désireux de le remplacer. Déçus par les limitations de CVS, ces développeurs ont doté le serveur de fichiers Subversion d'atouts justifiant une migration.

Parmi les atouts de Subversion, citons notamment les suivants :

- Versionnement des répertoires en plus des fichiers, ainsi que des liens symboliques.

- Renommage des fichiers. Sous CVS renommer un fichier entraîne sa suppression et sa recréation. Ce n'est plus le cas avec Subversion, qui garde l'historique du changement de nom.
- Modification atomique. Il est possible d'envoyer plusieurs modifications d'un ensemble de fichiers. Si la modification échoue sur un des fichiers, l'ensemble de la mise à jour est annulé.
- Support des fichiers binaires, avec différenciation des versions.
- Choix du protocole de réseau. Subversion permet de gérer les accès au référentiel en tenant compte de nouveaux protocoles réseau, dont SSH, favorisant d'autant l'interopérabilité avec les solutions existantes fournies par le serveur (authentification, autorisation, compression, etc.). Par exemple, Subversion peut se pluguer au serveur Apache HTTP en utilisant un module d'extension spécifique et utiliser WebDAV comme protocole.
- Convivialité et simplicité. Les numéros de version sont désormais globaux pour l'ensemble du dépôt et non plus par fichier. Chaque patch a un numéro de révision unique, quels que soient les fichiers touchés. Il devient simple de se souvenir d'une version particulière d'un projet, en ne retenant qu'un seul numéro.

Tous ces avantages justifient à eux seuls la migration, et nous ne saurions trop conseiller de passer le cap. Subversion est distribué comme son prédécesseur sous licence compatible Apache/BSD et est disponible sur le site principal du projet, à l'adresse *http://subversion.tigris.org/*. La version 1.4.2 est disponible en téléchargement sur ce site pour pratiquement toutes les plates-formes disponibles (Linux Red Hat, Debian ou Ubuntu, Windows, etc.).

La distribution cliente pour plate-forme Windows conseillée est TortoiseSVN. Son principal avantage est de s'intégrer parfaitement à l'Explorateur de fichiers de Windows.

Concepts clés et nouvelles fonctions

Subversion ressemble tant à CVS que ceux qui ont l'habitude d'utiliser ce dernier ne devraient pas rencontrer trop de difficultés d'adaptation.

Il est possible de convertir un référentiel CVS grâce au projet cvs2svn *(http://cvs2svn.tigris.org/)*, et les commandes CVS sont pratiquement toutes conservées (co pour récupérer des modifications, commit pour en envoyer de nouvelles, diff pour mesurer la différence entre deux versions et update pour mettre à jour les données du référentiel).

Parmi les nouvelles commandes qui ont fait leur apparition pour compenser les limitations de CVS, signalons move, qui remplace avantageusement la suite de commandes cvs, remote/add/commit), revert, qui retourne à la version précédente d'un fichier, et status, qui donne un résumé des modifications effectuées sur le dépôt depuis la dernière action de l'utilisateur.

Les quelques concepts clés spécifiques à Subversion concernent en particulier les trois dossiers principaux suivants d'un projet svn « typique » :

- /trunk : répertoire principal contenant les modifications au jour le jour du projet ;
- /branches : répertoire servant à la gestion des extensions du projet initial (notion de fork) ou au stockage d'une copie complète du dossier /trunk à un instant donné, par exemple lors de la sortie d'une release majeure du projet ;
- /tags : dossier servant normalement à stocker une version spécifique du projet.

Dans les faits, /branches et /tags ne sont que des copies de /trunk et se gèrent de la même manière. On peut ne utiliser ces deux dossiers ou n'en utiliser qu'un seul et y stocker les fichiers selon les besoins du projet.

Mise en œuvre

Monter un serveur Subversion pour une utilisation avec l'IDE Eclipse ne présente pas de difficulté particulière.

Pour la partie serveur, vous utiliserez la distribution standard pour plate-forme Windows. Pour une utilisation plus professionnelle et plus robuste, il est recommandé d'utiliser la version pour environnement Linux, la version sous Windows étant surtout utilisée ici pour démontrer les fonctionnalités de la solution. Les commandes détaillées dans cette section sont transposables sans modification particulière dans cet environnement.

Après avoir téléchargé la distribution de svn pour Windows (ici la version 1.4.2 disponible sur le site officiel *http://subversion.tigris.org/project_packages.html*), il suffit de lancer le programme d'installation svn-1.4.2-setup.exe et de se laisser guider par l'installateur.

Prenez soin d'ajouter la variable d'environnement %SVN_HOME% avec pour valeur le répertoire d'installation de Subversion, et vérifiez que l'installation s'est bien passée en entrant la commande svn help. Si vous souhaitez connaître la syntaxe d'une commande, entrez svn help <macommandesvn>, où macommandesvn désigne une commande SVN.

Un groupe de programmes appelé Subversion par défaut est installé sous %SVN_HOME%/bin, contenant l'ensemble des outils nécessaires à son fonctionnement, dont les principaux modules suivants :

- svn : programme de commande en ligne client ;
- svnversion : programme de reporting de l'état d'une copie de travail ;
- svnlook : outil d'inspection du référentiel Subversion ;
- svnadmin : outil de gestion du référentiel Subversion ;
- svndumpfilter : outil de filtrage pour le référentiel Subversion ;
- mod_dav_svn : plug-in pour le serveur Apache http 2.0, permettant de rendre votre référentiel accessible aux autres utilisateurs à travers le réseau ;
- svnserve : programme serveur standalone, exécutable comme un programme démon ou invocable par SSH.

Pour achever l'installation, vous devez configurer la variable d'environnement SVN_EDITOR *via* le Panneau de configuration en la positionnant à la valeur du chemin contenant votre éditeur favori (par exemple, Editplus.exe, en prenant soin d'éviter les espaces dans le chemin).

La variable PATH doit être ajoutée, toujours par le biais du Panneau de configuration Windows, pour donner le chemin d'accès aux binaires de svn :

```
PATH=%PATH% ; C:\Program Files\Subversion\bin
```

Le service Windows permettant de démarrer svn doit aussi être installé. Appelé SvnService.zip, ce service est disponible en téléchargement. Il suffit de décompresser l'archive

téléchargée dans un espace temporaire puis de l'installer de manière définitive en entrant sous DOS la commande suivante :

```
SVNService -install
```

En supposant l'installation de Subversion correctement effectuée, vous pouvez commencer à travailler avec Subversion.

Création du référentiel Subversion

Pour initialiser le référentiel Subversion et y accéder ensuite *via* Eclipse, vous devez commencer par créer un répertoire svn_myrepo sous d:\svn_myrepo.

1. Ouvrez une fenêtre de commande DOS, et entrez :

```
svnadmin create "d:\svn_myrepo"
```

   L'exécution de cette commande à pour effet de construire l'arborescence illustrée à la figure 3.22 dans le répertoire servant de référentiel svn_myrepo.

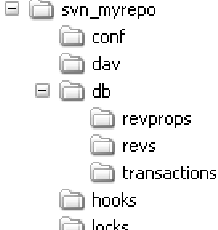


Figure 3.22

Arborescence du référentiel après création

2. Pour configurer les fichiers de configuration svn, ouvrez le répertoire d:\svn_myrepo\conf.
3. Ouvrez le fichier svnserve.conf dans un éditeur de texte, et décommentez les lignes `[general]`, `anon-access = read`, `auth-access = write` et `password-db = passwd` (pour décommenter, il suffit de supprimer le # et l'espace, mais surtout pas les ###).
4. Sauvegardez et fermez l'éditeur.
5. Ouvrez le fichier passwd dans un éditeur de texte, puis décommentez la ligne `[users]`, et ajoutez le username et le password que vous voulez utiliser pour vous connecter au serveur Subversion.
6. Sauvegardez et fermez l'éditeur.

Démarrage du démon svn

1. Dans la fenêtre de commande DOS, lancez le démon svn en entrant le référentiel que vous avez créé précédemment en argument de la commande :

```
svnserve --daemon --root "d:\svn_myrepo"
```

2. Ouvrez une deuxième fenêtre de commande DOS, et entrez la commande :

```
svn mkdir svn://localhost/myproject
```

L'éditeur de texte que vous avez paramétré en variable d'environnement s'ouvre alors sur un fichier comportant le texte :

```
--Cette ligne, et les suivantes ci-dessous, seront ignorées--
A    svn://localhost/myproject
```

3. Entrez un commentaire, par exemple "Création de mon projet SVN" au début du fichier (avant la ligne commençant par "--").
4. Sauvegardez le fichier et fermez l'éditeur.
5. Dans la fenêtre de commande DOS de création, si votre login Subversion est le même que celui de votre session Windows, saisissez votre mot de passe (celui que vous avez entré dans le fichier passwd), et pressez la touche Entrée.
6. Si votre login Subversion est différent de votre login Windows, pressez la touche Entrée à l'invite du password dans la fenêtre de commande. Subversion vous demande le login et le mot de passe que vous avez saisis dans le fichier passwd.
7. Comme illustré à la figure 3.23, Subversion indique "Révision 1 propagée". Ce message vous confirme que la révision a bien été propagée et que vous disposez d'une première version du fichier qui vient d'être chargé.

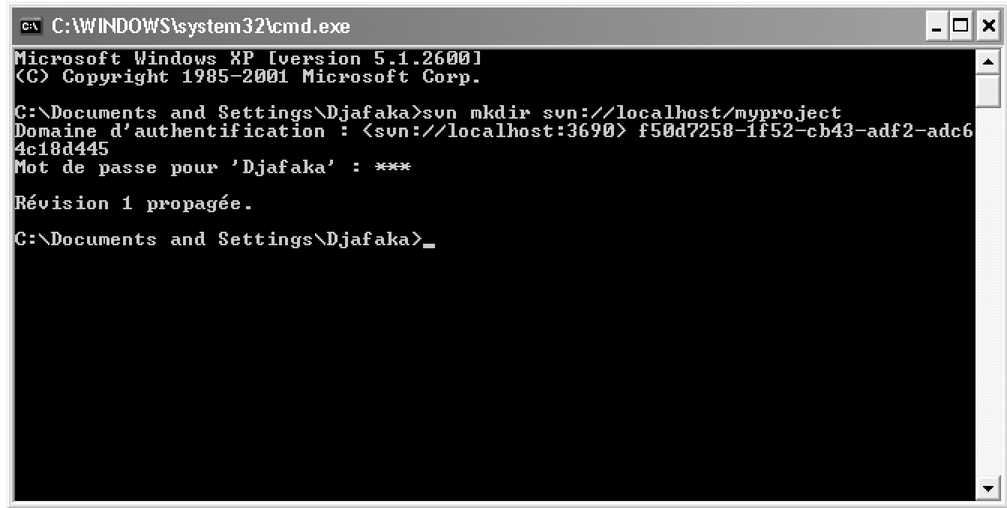


Figure 3.23

Propagation d'une révision avec svn

8. Créez un sous-répertoire trunk en entrant la commande :

```
svn mkdir svn://localhost/MyProjet/trunk
```

9. Comme précédemment, saisissez un commentaire dans le fichier ouvert, puis sauvegardez et fermez l'éditeur. Vous devez constater qu'une deuxième révision a été propagée dans la fenêtre DOS.

Vous pouvez à présent accéder à votre serveur svn en utilisant le client TortoiseSVN symbolisé par une petite tortue (voir la procédure d'installation de TortoiseSVN en annexe) et en cliquant sur l'option Navigateur de référentiel illustrée à la figure 3.24.

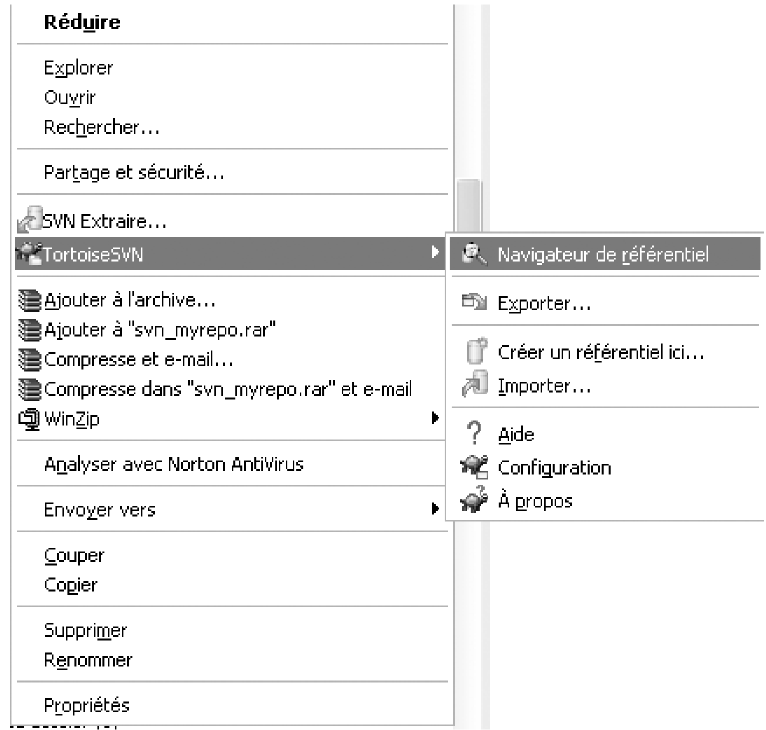


Figure 3.24

Mise en œuvre du client TortoiseSVN

Vous pouvez dès lors travailler directement avec le référentiel précédemment créé en utilisant le navigateur de référentiel et en examinant dans le détail la structure et l'état du référentiel (voir figure 3.25).

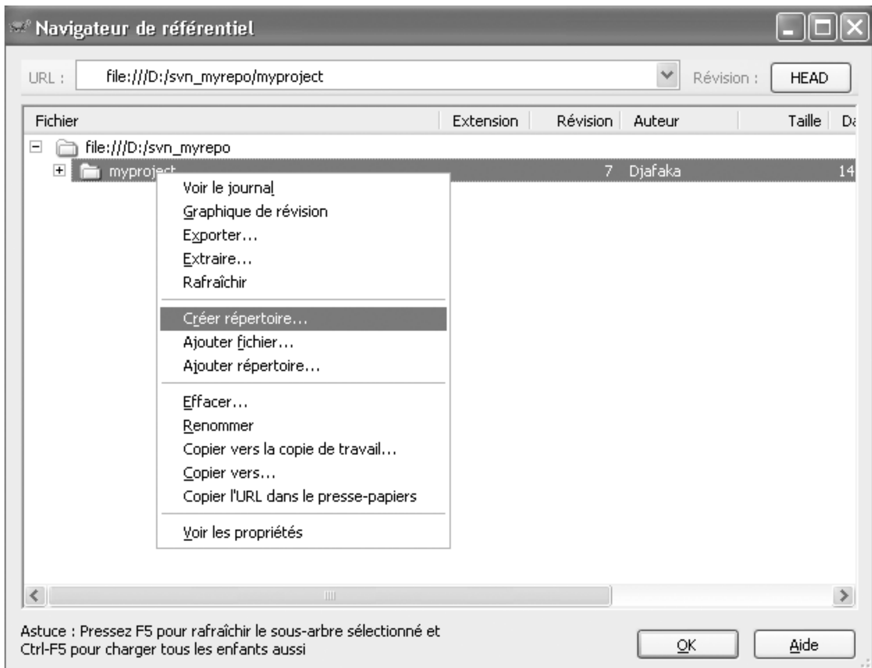


Figure 3.25

Options disponibles du navigateur de référentiel

Vous pouvez exécuter des actions telles que la copie, le renommage ou le déplacement de fichiers ou de répertoires directement dans le référentiel svn_myrepo. Dans la zone URL du navigateur, vous pouvez entrer l'URL du référentiel et de la révision dans lesquels vous souhaitez naviguer. Naviguer dans une ancienne révision peut se révéler utile si vous souhaitez récupérer un fichier malencontreusement supprimé (voir figure 3.26).

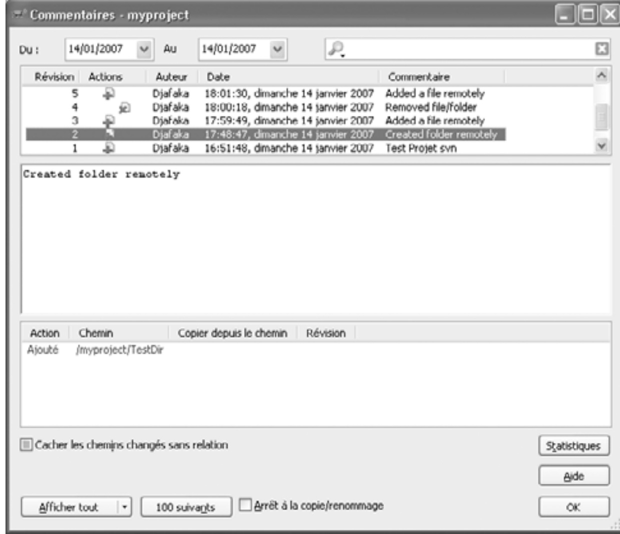


Figure 3.26

Historique des révisions dans le navigateur de référentiel

À l'aide du client TortoiseSVN, vous pouvez créer un fichier dans un répertoire, en procédant de la manière suivante :

10. Créez un répertoire sur votre disque, par exemple d:\MyProject.
11. Faites un clic droit sur le répertoire, et sélectionnez SVN Extraire.
12. Entrez svn://localhost/monprojet/trunk/ comme URL du référentiel, et cliquez sur OK puis à nouveau sur OK dans la fenêtre suivante (voir figure 3.27).

Figure 3.27

Extraction d'un répertoire

Une boîte de dialogue vous demande de confirmer l'extraction, ce qui a pour effet d'ajouter une icône dans l'Explorateur de fichiers Windows.

13. Créez un fichier dans ce répertoire, par exemple tata.txt.

14. Faites un clic droit sur ce fichier, et sélectionnez TortoiseSVN puis Ajouter (voir figure 3.28).

15. Validez en cliquant sur OK.

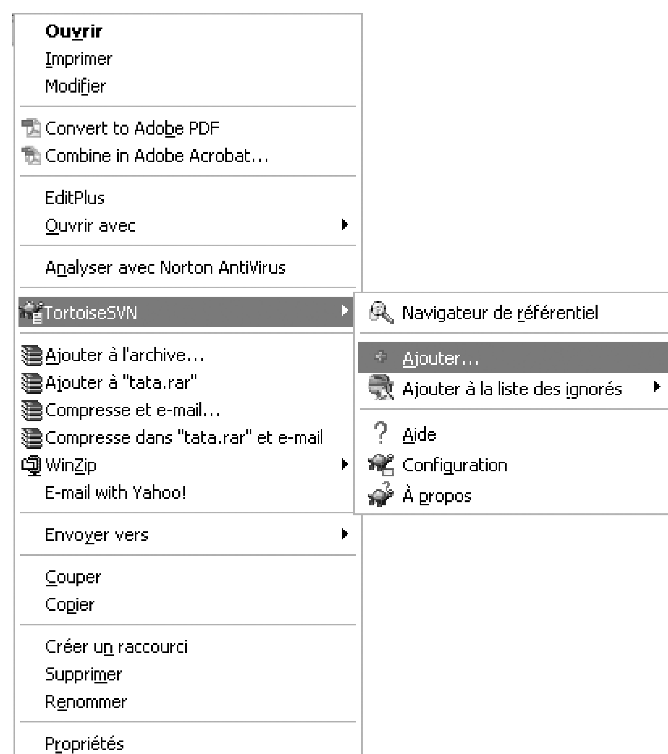


Figure 3.28

Ajout d'un fichier dans Subversion via *le client TortoiseSVN*

16. Une boîte de dialogue confirme l'opération. Cliquez sur OK.

17. Même si la boîte de dialogue confirme l'ajout du fichier, celui-ci n'est pas encore chargé dans le référentiel Subversion. Pour ce faire, faites un clic droit sur le fichier, et cliquez sur SVN Livrer.

18. Dans la fenêtre qui s'affiche (voir figure 3.29), saisissez un commentaire plus cliquez sur OK. Si tout se passe bien, l'envoi du fichier est pris en compte et confirmé par une boîte de dialogue.

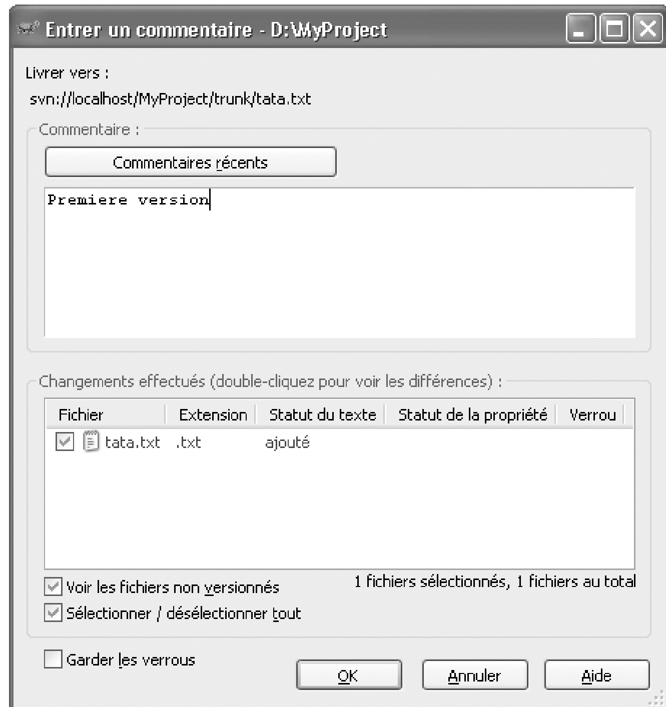
Un clic sur l'Explorateur de référentiel doit confirmer l'opération d'ajout.

https

Il est plus que conseillé d'utiliser https pour tout accès en écriture depuis les clients SVN, afin de protéger les fichiers par mot de passe.

Figure 3.29

Commit du fichier



Installation de svn comme service

Plutôt que de lancer le démon directement en ouvrant chaque fois une fenêtre DOS, il est possible de le faire *via* le service Windows associé.

1. Ouvrez l'archive SVNService.zip téléchargée précédemment.
2. Extrayez SVNService.exe (et les autres fichiers de l'archive) dans le répertoire bin de Subversion. Il est important que le contenu de l'archive soit au même endroit que l'exécutable svnserve.exe.
3. Ouvrez une fenêtre de commande DOS, et entrez :

```
svnservice -install --daemon --root "d:\svn_myrepo".
```

Si tout se passe bien, votre service svn est installé durablement comme service Windows, et vous pouvez le démarrer ou l'arrêter en mode manuel ou automatique. Si vous arrêtez le service, l'accès au référentiel devient impossible, et le message « Can't connect to host "localhost" » s'affiche.

Accéder au serveur svn dans Eclipse

Cette section montre comment configurer Eclipse 3.3 pour utiliser svn.

Eclipse ne supportant pas par défaut Subversion, il est nécessaire d'installer des plug-ins tiers, en particulier celui du projet Subclipse. Subclipse est un projet Open Source *(http://subclipse.tigris.org/)* disponible sous licence EPL (Eclipse Public License).

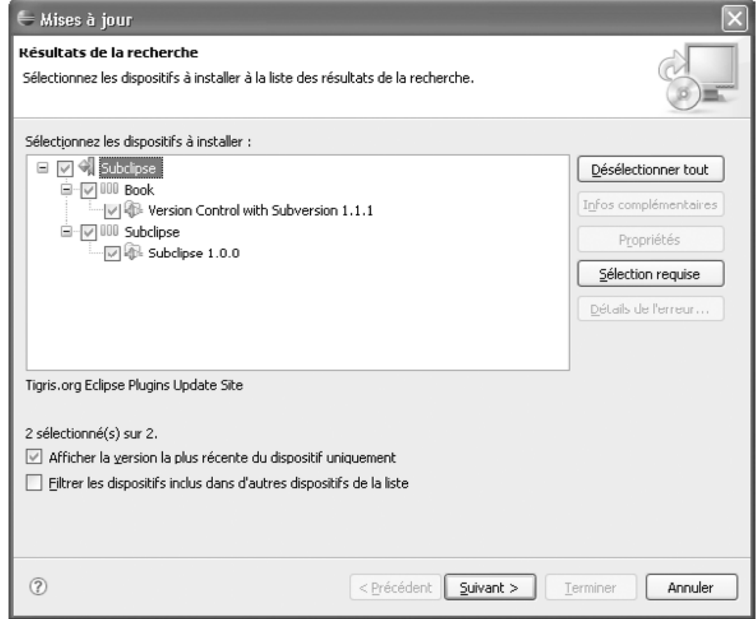
Pour configurer Eclipse pour le support de svn, procédez de la façon suivante :

1. Sélectionnez Aide, Mises à jour de logiciels et Rechercher et installer.

2. Cliquez sur « Rechercher les nouveaux dispositifs à installer » puis sur Suivant.
3. Ajoutez un nouveau site distant nommé Subclipse (URL : *http://subclipse.tigris.org/update),* puis cliquez sur OK et Terminer.
4. Sélectionnez l'ensemble des dispositifs illustrés à la figure 3.30.

Figure 3.30

Dispositifs d'installation de Subclipse



Compatibilité

Veillez à toujours sélectionner une version cliente de Subversion compatible avec votre serveur svn.

5. Cliquez sur Suivant, et acceptez les termes du contrat de licence.
6. Cliquez à nouveau sur Suivant puis sur Terminer pour démarrer le processus d'installation du plug-in dans Eclipse.
7. Relancez votre plan de travail Eclipse afin de prendre en compte l'installation du plug-in.
8. Si l'installation s'est bien déroulée, vous devez découvrir une nouvelle vue Eclipse, en cliquant sur Fenêtre, Ouvrir la perspective, Autre et SVN Repository Exploring.

   Cette vue laisse découvrir la palette svn illustrée à la figure 3.31.

Figure 3.31

Palette svn dans Eclipse

9. Faites un clic droit dans l'espace de travail de svn, puis sélectionnez New et Repository Location, comme illustré à la figure 3.32

Figure 3.32

Configuration du chemin d'accès au référentiel

10. Configurez l'accès au référentiel svn précédemment créé en entrant svn://localhost/, puis cliquez sur Terminer.

Vous pouvez à présent accéder au contenu du référentiel et disposer de toutes les commandes svn d'accès. À la figure 3.33, notez le nom de la révision en face du nom du fichier, tata dans notre exemple.

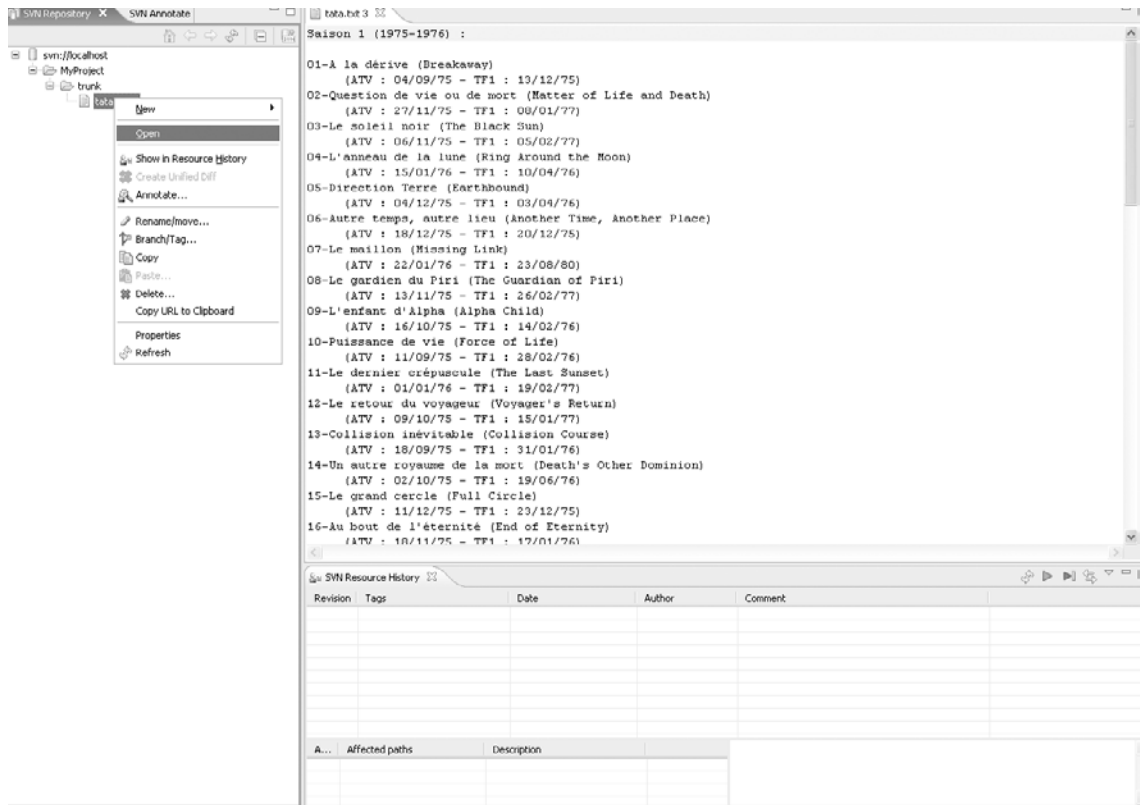


Figure 3.33

Ouverture du client svn dans Eclipse

Mise en œuvre d'un projet Eclipse avec Subclipse

Vous avez à ce stade configuré le plug-in Subclipse et êtes prêt à gérer l'ensemble de vos projets Eclipse avec svn.

Dans cette section, vous allez partager vos projets Eclipse en changeant d'espace de travail.

1. Sélectionnez Fichier puis Accéder à l'espace de travail.

2. Désignez l'emplacement du référentiel (dans cet exemple d:\MyProject).
3. Créez un nouveau projet Eclipse (un projet Java, par exemple).
4. Subclipse vous demande de prendre en charge votre projet sous le contrôle du répertoire de Subversion et de l'ajouter automatiquement. Répondez Oui, et cliquez sur Terminer. Votre projet est désormais intégré au référentiel svn.
5. Pour partager votre projet en équipe, sélectionnez-le par clic droit, et choisissez Equipe puis Partager le projet dans le menu contextuel.
6. Sélectionnez le type de référentiel, ici svn, puis cliquez sur Suivant.
7. Cliquez sur l'emplacement du référentiel existant (svn://localhost), et cliquez sur Suivant.
8. Conservez l'option par défaut « Use project name as folder name », et cliquez sur Terminer.

La configuration permettant le partage de votre projet Eclipse avec le référentiel svn précédemment défini est terminée. Vous pouvez à présent profiter pleinement des nouvelles fonctionnalités de Subversion.

En résumé

Ce chapitre vous a permis de mesurer la facilité de prise en main de l'outil CVS dans Eclipse grâce à la perspective CVS intégrée.

Vous avez fait connaissance avec Subversion et avez pu apprécier sa facilité d'utilisation et ses nombreux avantages par rapport à son « ancêtre » CVS.

Le chapitre suivant vous donnera l'occasion de mettre en œuvre les fonctionnalités de profiling évolué de l'IDE Eclipse grâce au plug-in TPTP.

4

Profiling de code avec le projet TPTP

La mise en place de tests et l'analyse des performances des applications sont des phases essentielles dans le cycle de développement d'un projet. C'est dans cette optique que la plate-forme TPTP (Test & Performance Tools Platform), aussi appelée Hyades, a vu le jour comme sous-projet Eclipse en 2002.

Après une description des fonctionnalités de TPTP et des différents modules qui le constituent, nous détaillerons sa mise en œuvre pratique sur un projet Java avant de passer au profiling d'une application Web.

Le projet TPTP a pour objectif de fournir un environnement complet pour la couverture des tests et la mesure des performances.

Il est constitué des quatre modules suivants organisés en sous-projets :

- TPTP Platform Project, la plate-forme qui fournit le socle utilisé par les différents outils de performance ainsi que les mécanismes de collecte des métriques.
- TPTP Monitoring Tools Project, les outils de monitoring qui permettent de collecter et d'analyser les ressources concernant l'application testée.
- TPTP Test Tools Project, les outils qui facilitent la création des tests (capture et exécution d'une suite de requêtes HTTP).
- TPTP Tracing and Profiling Tools Project, les outils de trace et de profiling des applications Java et J2EE qui permettent de collecter des informations sur le fonctionnement des applications Java et de les analyser (localisation des allocations mémoire, détermination des temps d'exécution).

Eclipse TPTP est une solution extensible qui couvre l'ensemble du cycle de vie d'un projet, depuis les tests unitaires jusqu'à la mise en production et au monitoring de l'application, en passant par l'enregistrement des tests, leur exécution ainsi que la collecte des métriques et le profiling.

Outre IBM, les principaux contributeurs au projet sont Computer Associates, Compuware, Intel, SAP et Scapa Technologies.

Architecture de TPTP

Avant d'entrer dans le cœur de l'architecture du projet TPTP, un des nombreux sous-projets Eclipse *(http://www.eclipse.org/tptp)*, nous allons esquisser une définition de cette activité de plus en plus importante au sein d'une équipe de développement J2EE, voire au sein des équipes en charge des tests de charge et d'optimisation Java, qu'est le profiling de code Java.

Du fait de la complexité croissante des architectures J2EE/JEE, le besoin de connaître dans le détail la santé de l'application, en particulier des composants qu'elle embarque, nécessite la mise en œuvre d'outils dédiés au profiling de code Java.

Le profiling de code consiste à générer un rapport détaillé de l'application analysée, comme son temps d'exécution et le pourcentage d'exécution de chaque procédure, ou méthode dans le cas d'applications Java, contenues au sein de l'application.

Avec un outil de profiling, on recherche essentiellement dans l'application les points suivants :

- présence de goulets d'étranglement à l'origine de dégradations des performances de l'application (temps de réponse, consommation de ressources CPU, etc.).
- consommation mémoire excessive provenant de certains composants ou méthodes de l'application ;
- fuites mémoire occasionnant une consommation anormale de ressources système (présence d'objets volumineux créés par l'application).

Le profiling du code consiste à observer le comportement de l'application en situation de charge, ce qui implique la plupart du temps l'utilisation d'outils de *stress test* ou d'injecteurs spécialisés, qui enregistrent le fonctionnement de l'application, comme loadRuner de Mercury ou QaLoad de Compuware. Il est ensuite possible d'étudier en mode déconnecté l'application et d'investiguer plus avant les problèmes de performances détectés, cette phase intervenant en général en fin de développement, avant son passage en préproduction.

Le projet TPTP offre un certain nombre de fonctionnalités de profiling, notamment les suivantes :

- ensemble de vues et de perspectives permettant d'historiser les résultats des tests d'exécution et de les examiner hors exécution (analyse statique) ;
- outils de test de non-régression et frameworks (plug-in JUnit, assistants graphiques d'automatisation des tests) ;
- possibilité d'interagir et de profiler ses applications, ainsi que, avec les ressources analysées, d'examiner les éventuels problèmes de performances et de gestion mémoire ;
- possibilité de comprendre et de visualiser l'exécution de l'application en mettant en relief les composants qui consomment le plus de ressources.

Composants de TPTP

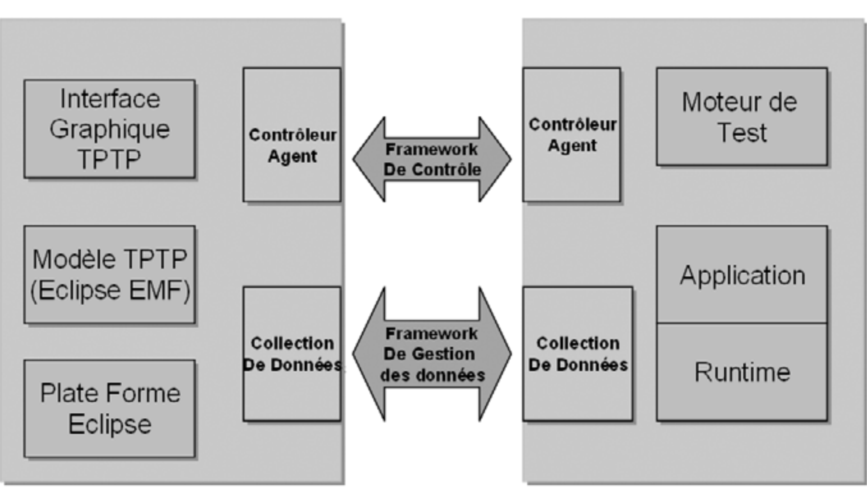
L'architecture de TPTP fournit un cadre à partir duquel une application cliente peut interagir avec une ou plusieurs applications fournissant les métriques (agents), et ce indépendamment de leur localisation.

Cette structure se compose d'une interface client, d'un processus de gestion d'agent, d'une interface d'agent et de quelques agents à usage général. Le processus de gestion d'agent est appelé contrôleur d'agent. C'est ce composant qui permet à une application cliente de se séparer des détails liés à la recherche et au démarrage des différents fournisseurs de données situés sur des machines locales ou distantes, dont le type de plate-forme (système d'exploitation et architecture d'unité centrale) et le langage (C/C++, Java) peuvent être différents.

La figure 4.1 illustre les principaux composants de TPTP.

Figure 4.1

Composants de TPTP



L'architecture de TPTP est constituée de deux parties, un système contrôleur et un système cible.

Le système contrôleur comporte les composants suivants :

- Interface utilisateur à partir de laquelle les tests sont pilotés.
- Modèles de données fondés sur le framework EMF (Eclipse Modeling Framework) incluant des modèles de tests d'exécution, des fichiers de logs et des statistiques.
- Plate-forme Eclipse (éditeurs).
- Interfaces vers un agent de contrôle ainsi qu'un moteur collectant les données statistiques.

Le système cible comporte les composants suivants :

- Application à tester.
- Environnement d'exécution de l'application (JVM et API d'accès associées).

- Moteur de test pour démarrer l'application et son environnement.
- Données collectées.

Comme vous pouvez le voir à la figure 4.2, un contrôleur d'agent, ou processus démon, situé de part et d'autre de l'architecture permet au client de lancer des processus distants et d'interagir avec les agents distants. Ce contrôleur peut ainsi, localement ou à distance, profiler des applications Java/J2EE et importer des fichiers de log en utilisant l'interface JVMPI (Java Virtual Machine Profiler Interface).

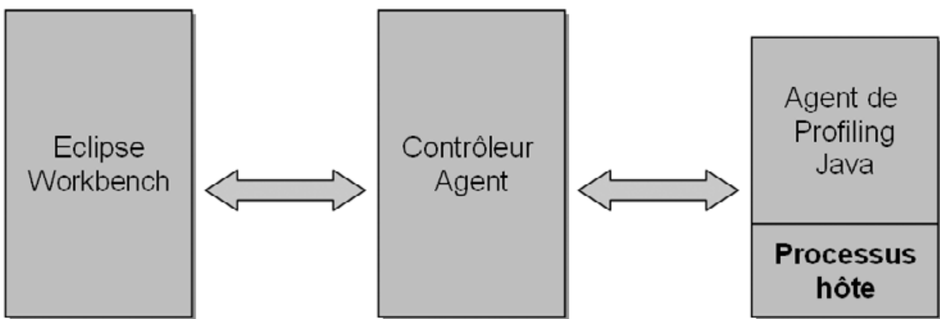


Figure 4.2

L'agent contrôleur de TPTP

Le processus hôte de la figure 4.2 définit le processus contenant l'application à tester.

Mise en œuvre de TPTP

Comme pour l'ensemble des projets Eclipse, l'installation peut s'effectuer sous trois formes : en ligne, *via* le gestionnaire de mise à jour Eclipse et en utilisant un site Europa miroir, ou par téléchargement manuel des fichiers d'installation correspondant sur le site du projet. Nous avons opté pour la distribution tptp.runtime.allInOne.win32.win32.x86-TPTP-4.4.0.3.zip (TPTP 4.4.0.3), contenant une version d'Eclipse 3.3 (Europa) ainsi que l'ensemble des prérequis (EMF 2.3, GEF 3.3 et Web Tools 2.0).

La version téléchargée doit être décompressée dans un répertoire de destination, par exemple C:\Europa\TPTP.

Cette version intègre déjà un agent, contrairement aux précédentes, qui nécessitaient l'installation et la configuration d'un contrôleur d'agent sur la machine à profiler. Par contre, le profiling sur une machine distante ou l'exécution d'un test TPTP nécessite toujours que le contrôleur d'agent soir installé sur cette machine.

Dans la suite du chapitre, vous mettrez en œuvre TPTP à partir d'une application exemple, hcframe.jar, dont les sources sont disponibles lors de l'installation de l'outil sous TPTP_install_dir\eclipse\plugins\org.eclipse.tptp.platform.collection.framework_4.0.0.

Vues de profiling disponibles

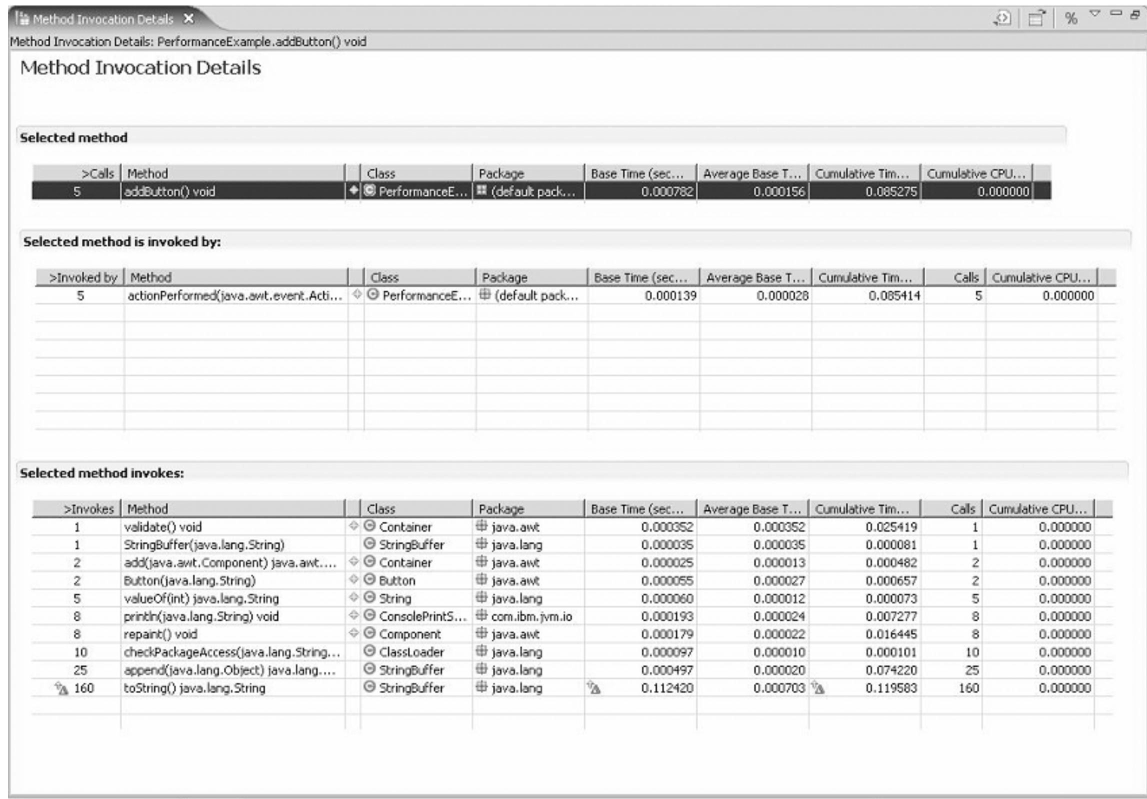
La plate-forme TPTP propose un outil de profiling pour identifier et isoler les problèmes de performance courants d'une application Java, comme les goulets d'étranglement, les fuites mémoire et les éventuelles contentions de ressources.

L'outil cible ce type de problème, quel que soit le niveau de complexité de l'application, depuis une application Java « simple » jusqu'aux plug-ins Eclipse et à des applications d'entreprise complexes s'exécutant sur des machines multiples et sur des plates-formes différentes.

Étant nativement intégré à la plate-forme Eclipse, TPTP est facilement extensible et facile à utiliser, permettant par exemple l'ajout de vues personnalisées par le développeur pour analyser des données.

TPTP dispose de nombreuses vues spécialisées dans un domaine particulier du profiling d'application, en particulier les suivantes :

- Possibilité d'identifier les objets et méthodes qui consomment le plus de ressources, en utilisant les vues Memory Statistic et Method Invocation Detail (voir figure 4.3).



>Calls	Method	Class	Package	Base Time (sec...	Average Base T...	Cumulative Tim...	Cumulative CPU...
5	addButton() void	PerformanceE...	(default pack...	0.000782	0.000156	0.085275	0.000000

>Invoked by	Method	Class	Package	Base Time (sec...	Average Base T...	Cumulative Tim...	Calls	Cumulative CPU...
5	actionPerformed(java.awt.event.Acti...	PerformanceE...	(default pack...	0.000139	0.000028	0.085414	5	0.000000

>Invokes	Method	Class	Package	Base Time (sec...	Average Base T...	Cumulative Tim...	Calls	Cumulative CPU...
1	validate() void	Container	java.awt	0.000352	0.000352	0.025419	1	0.000000
1	StringBuffer(java.lang.String)	StringBuffer	java.lang	0.000035	0.000035	0.000081	1	0.000000
2	add(java.awt.Component) java.awt....	Container	java.awt	0.000025	0.000013	0.000482	2	0.000000
2	Button(java.lang.String)	Button	java.awt	0.000055	0.000027	0.000657	2	0.000000
5	valueOf(int) java.lang.String	String	java.lang	0.000060	0.000012	0.000073	5	0.000000
8	println(java.lang.String) void	ConsolePrintS...	com.ibm.jvm.io	0.000193	0.000024	0.007277	8	0.000000
8	repaint() void	Component	java.awt	0.000179	0.000022	0.016445	8	0.000000
10	checkPackageAccess(java.lang.String...	ClassLoader	java.lang	0.000097	0.000010	0.000101	10	0.000000
25	append(java.lang.Object) java.lang....	StringBuffer	java.lang	0.000497	0.000020	0.074220	25	0.000000
160	toString() java.lang.String	StringBuffer	java.lang	0.112420	0.000703	0.119583	160	0.000000

Figure 4.3

Vue Method Invocation Detail

- Analyse de la gestion de la concurrence des threads et de l'exécution de l'application au niveau thread (vue Thread Analysis). La figure 4.4 illustre une telle vue, chaque thread étant représenté par une barre verticale.

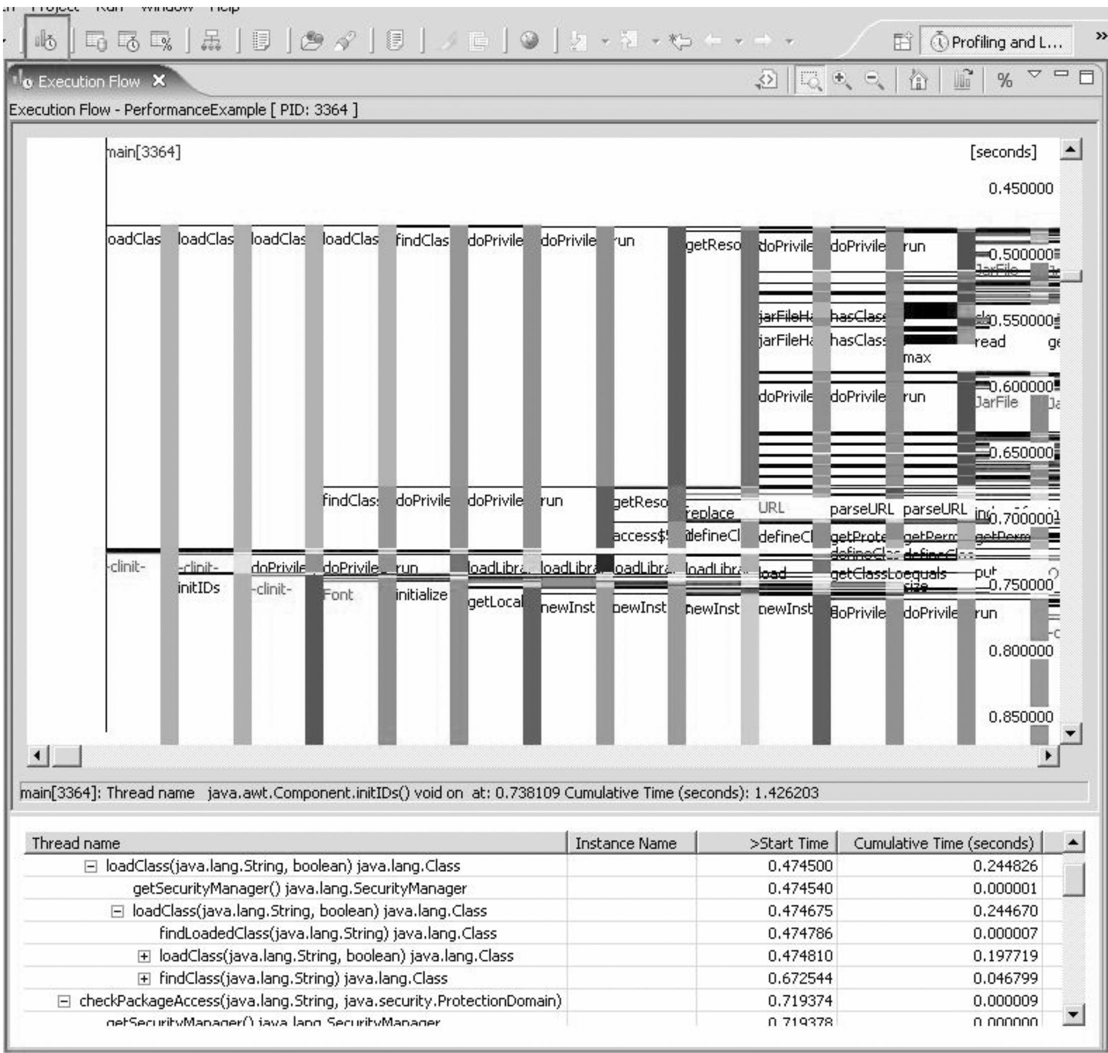


Figure 4.4

Vue Execution Flow

- Localisation des fuites mémoires grâce à la vue Object Reference, qui affiche les références par type d'objet (voir figure 4.5).
- Identification des méthodes les plus fréquemment appelées grâce à la vue Method Invocation, qui représente de manière graphique les invocations des méthodes appelées et appelantes (voir figure 4.6).
- Analyse du flux d'exécution du programme et identification des différentes phases de son exécution (voir figure 4.7).

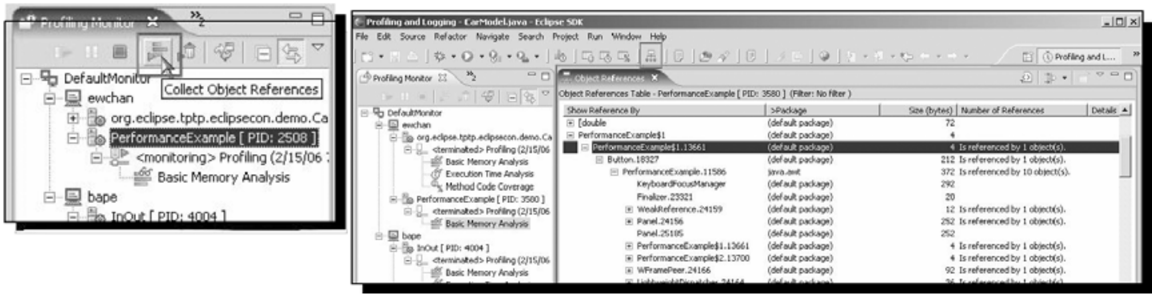


Figure 4.5

Vue Object Reference

Method Invocation: java.lang.ClassLoader.loadClass(java.lang.String) java.lang.Class

main(java.lang.String[]) void : 1/1

(default package).PerformanceExample.main(java.lang.String[]) void in at: 1.028433 Cumulative Time (seconds): 0.219346

>Invocations	Instance Name	Start Time	Cumulative Tim...	Average Base Time (se
loadClass(java.lang.String) java.lang.Class		1.107982	0.000022	
loadClass(java.lang.String) java.lang.Class		1.108102	0.000009	
loadClass(java.lang.String) java.lang.Class		1.108637	0.001358	
loadClass(java.lang.String) java.lang.Class		1.112441	0.001177	
loadClass(java.lang.String) java.lang.Class		1.113935	0.000977	
loadClass(java.lang.String) java.lang.Class		1.115114	0.000011	

Figure 4.6

Vue Method Invocation

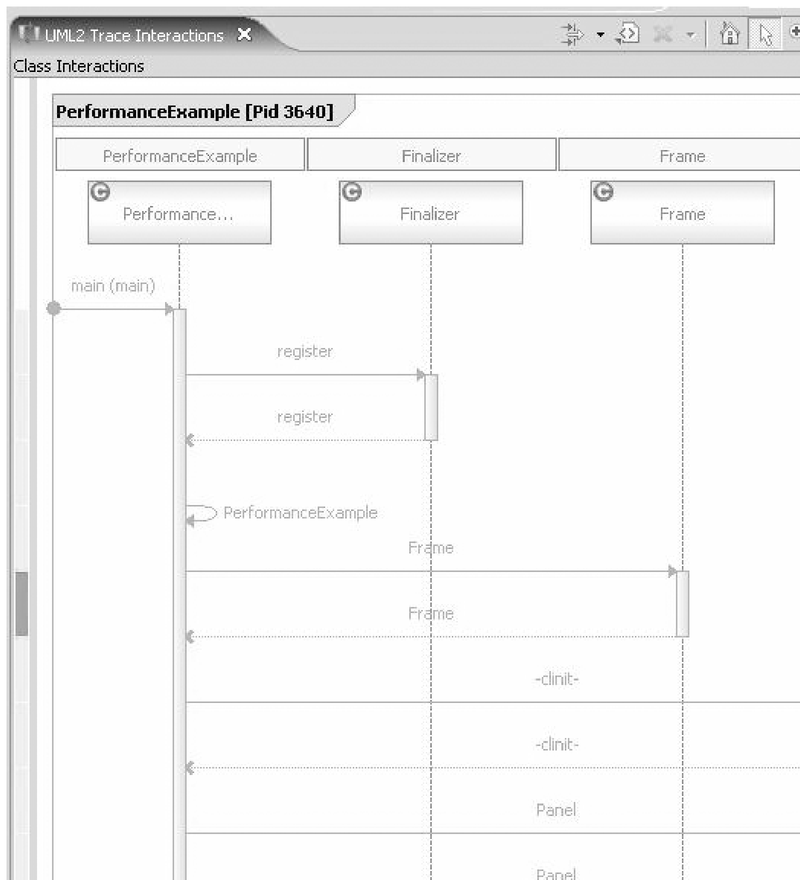


Figure 4.7

Vue Interaction View

Ouverture de la vue de profiling

Pour analyser les données collectées à partir d'une JVM, la vue et la perspective associées Profiling and Logging offre un certain nombre d'options avancées, que vous allez mettre en œuvre un des exemples déjà fournis.

1. Créez un projet Java par le biais de l'assistant de création de projet Eclipse (en entrant, par exemple, TPTPProjectJava dans le champ Nom du projet).
2. Dans le menu Window, cliquez sur Open Perspective, Other, Profiling and Logging, puis cliquez sur OK.
3. Cliquez sur le menu déroulant Profil, puis sélectionnez Profiler et Open Profil Dialog. La boîte de dialogue du profil de lancement s'ouvre.
4. Sélectionnez External Java Application.
5. Cliquez sur l'icône New launch configuration.
6. Saisissez Ma Configuration TPTP devant le champ Name.
7. Cliquez sur Test Connection pour valider la connexion avec l'agent (local dans votre cas) comme illustré à la figure 4.8.
8. Dans l'onglet Main, entrez org.eclipse.hyades.test.collection.framework.PerformanceExample devant le champ Nom de classe

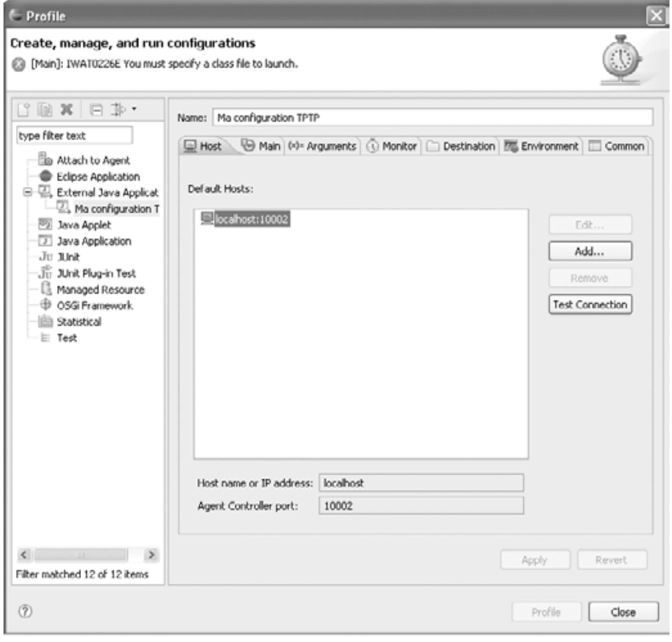


Figure 4.8

Configuration de l'agent local

9. Cliquez sur Add JAR pour ajouter un fichier JAR (<TPTP_INSTALL_DIR>\eclipse\plugins\ org.eclipse.tptp.platform.collection.framework_4.2.101.v200709261752 \hcframe.jar) puis cliquez sur Apply.
10. Cliquez sur l'onglet Monitor pour spécifier les critères de profiling et cochez l'option Memory Analysis pour une analyse de la mémoire lors de l'exécution de l'application (voir figure 4.9).

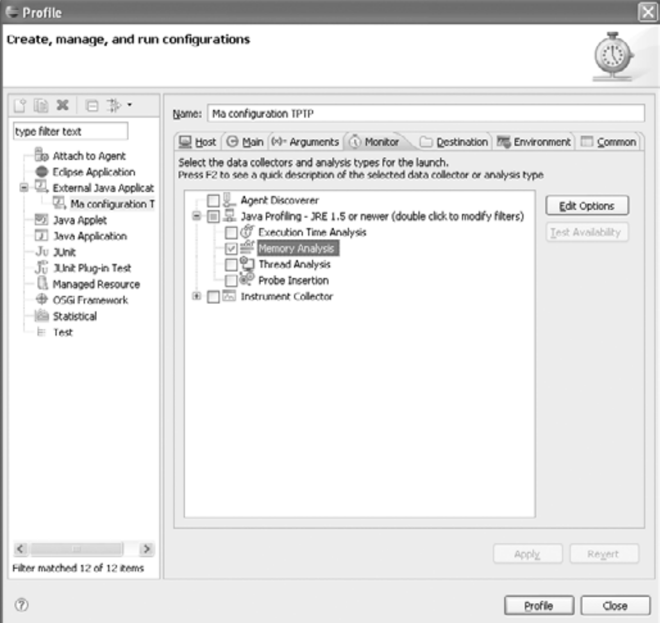


Figure 4.9

Configuration des vues de profiling associées

11. Cochez l'option de profiling Execution Time Analysis.
12. Cliquez sur le bouton Edit Options en regard de cette option pour spécifier des analyses statistiques (voir figure 4.10).

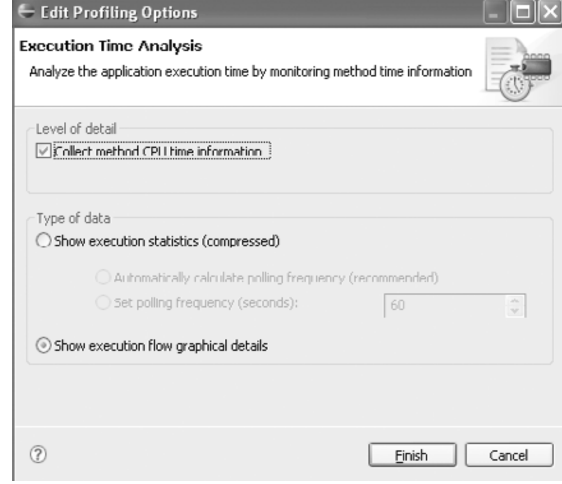


Figure 4.10

Configuration de la vue d'analyse mémoire en fonction du temps

13. Cochez l'option Show execution flow graphical details pour afficher les détails du flux d'exécution sous forme de graphique, et réglez la profondeur des classes (Boundary class depth) à 1, puis cliquez sur Finish.
14. Double-cliquez sur l'option Java Profiling pour modifier les filtres (voir figure 4.11). Cette option permet d'agir sur le niveau des informations que vous souhaitez analyser. C'est surtout utile pour ignorer les classes ou les packages que vous ne voulez pas analyser et réduire le volume des informations collectées lors du profiling.

Figure 4.11

Configuration des filtres de profiling

15. Sous la zone Contents of selected filter set, cliquez sur Add pour ajouter les packages dont vous voulez collecter les données.
16. Saisissez org.eclipse.hyades.test.collection.framework.* dans le champ associé au nom de la classe (l'astérisque permet de n'inclure que les méthodes relatives à cet exemple), et cliquez sur OK.
17. Vérifiez que les autres classes figurant dans la liste des classes à filtrer sont positionnés à EXLUDE (exclues de la liste des classes à profiler).
18. Cliquez sur Suivant pour accéder à la page Limites, qui permet de contrôler la quantité de données collectées en fonction du temps écoulé ou du nombre d'appel de méthodes spécifié. Acceptez les valeurs par défaut.
19. Cliquez sur Finish pour appliquer les modifications effectuées.
20. Cliquez sur l'onglet Destination, et laissez les valeurs par défaut déjà positionnées (champs Profiling project et Monitor). Ces champs représentent les ressources de profiling chargées de stocker les données à partir de la session de profiling.
21. Cliquez sur Apply pour valider l'ensemble des modifications et Profile. Les ressources de profiling requises sont créées, et l'agent et le processus apparaissent dans la vue Moniteur de profiling (voir figure 4.12).

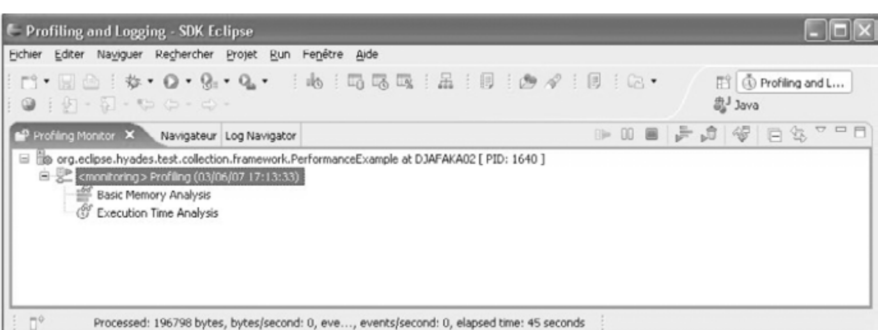


Figure 4.12

Vue Monitoring du profiling après configuration

22. Activez la visualisation des moniteurs en sélectionnant Présentation distribuée. Remarquez que les informations collectées sont enregistrées dans les ressources de profiling configurées précédemment, en particulier dans DefaultMonitor, comme l'illustre la figure 4.13.

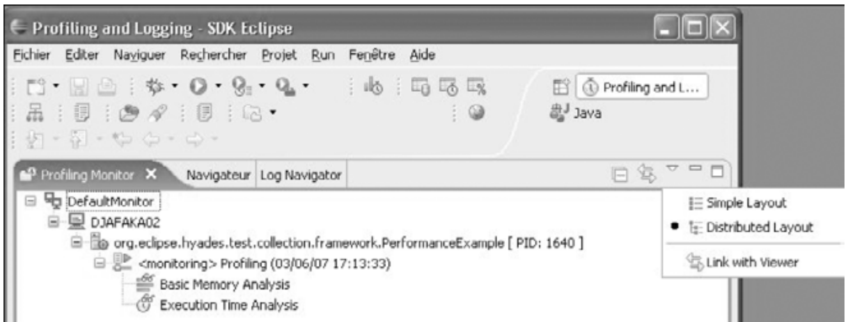


Figure 4.13

Configuration de la vue distribuée (vue moniteur)

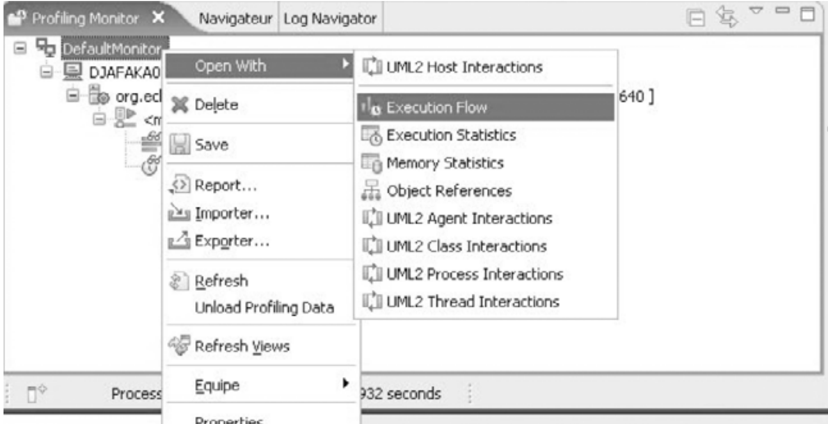
Analyse des informations de profiling collectées

La vue du flux d'exécution permet de représenter l'exécution du programme, dont les threads sont représentés horizontalement, et le temps par l'axe vertical.

1. Pour accéder à cette vue, sélectionnez votre moniteur dans la vue Moniteur de profiling, et sélectionnez Open With puis Execution flow, comme illustré à la figure 4.14.

Figure 4.14

Vue flux d'exécution



2. Pour examiner une méthode donnée, cliquez sur son nom pour la sélectionner. Toutes les méthodes appelées sont ensuite mises en évidence en jaune.

   La séquence des appels de méthode se déroule de gauche à droite ; les parties actives de l'unité d'exécution sont identifiées par cette progression de gauche à droite. La longueur verticale de la zone mise en surbrillance indique le temps de base de la méthode. La valeur exacte du temps de base s'affiche sur la ligne d'état.

   Comme vous pouvez le voir sur l'exemple de la figure 4.15, la vue flux d'exécution affiche le déroulement d'appel de chaque méthode, organisé sous forme de « bandes » de couleur, chaque bande représentant le nom du thread (sur la figure 4.15 les threads gc, main et AWT-Event_Queue). Vous pouvez bien sûr sélectionner les threads que vous souhaitez visualiser, ainsi que leur couleur, dans le graphe d'exécution par simple clic droit.

   Le temps est affiché (en seconde) sur l'axe vertical, en commençant par le haut du diagramme.

3. Pour examiner une méthode donnée, cliquez sur son nom pour la sélectionner. Toutes les méthodes appelées sont ensuite mises en évidence en jaune. La longueur verticale de la zone mise en surbrillance indique le temps de base de la méthode. La valeur exacte du temps de base s'affiche sur la ligne d'état (voir l'exemple de la méthode main invoquée à la figure 4.16).

4. Si la zone analysée est trop saturée et difficile à lire, faites un zoom avant dans la zone saturée.

5. Pour réinitialiser la vue, cliquez sur l'icône représentant une maison.

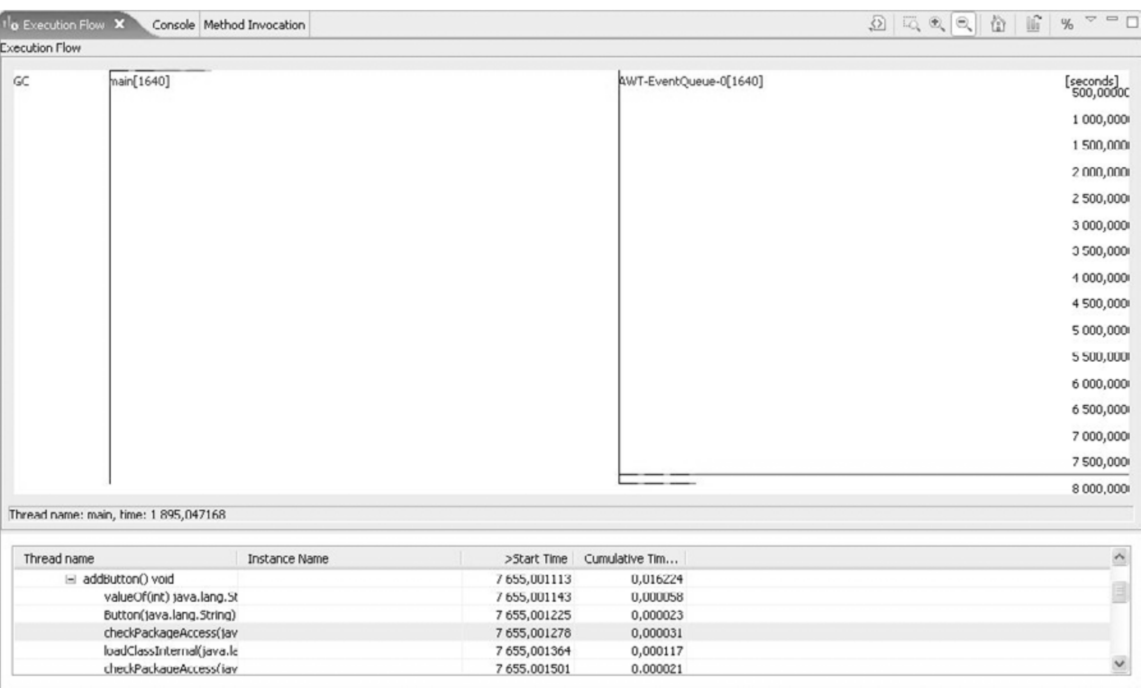


Figure 4.15

Vue flux d'exécution en action

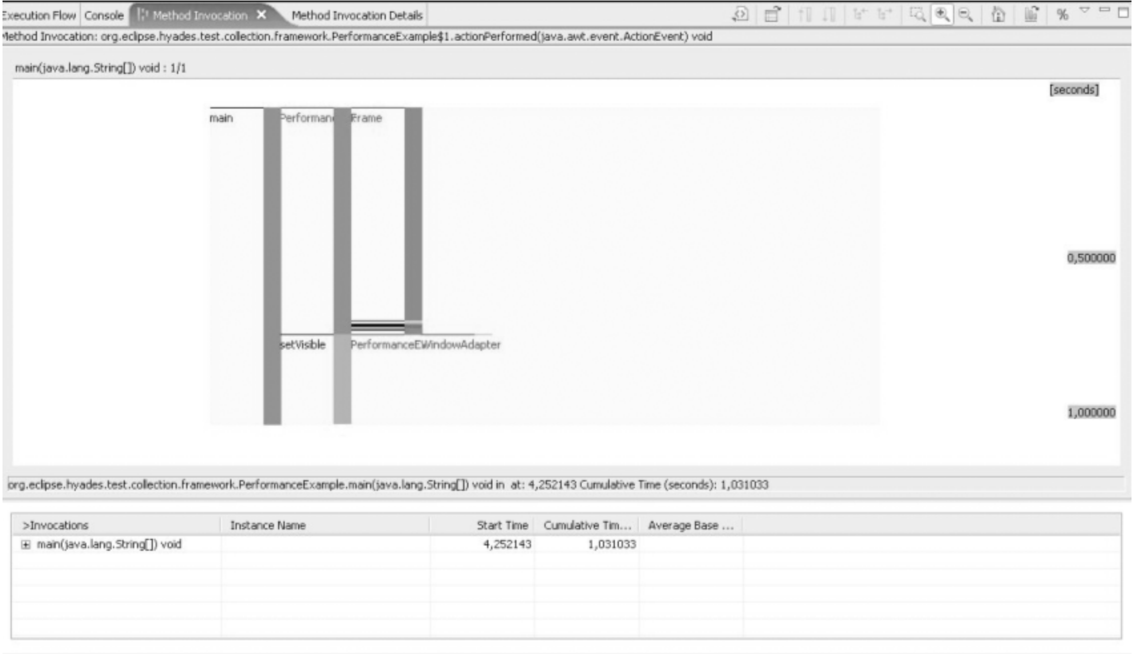


Figure 4.16

Répartition dans le temps des invocations des méthodes

6. Vous pouvez analyser le détail de l'invocation de la méthode en cliquant sur l'option Show Method Invocation Details du menu contextuel de la méthode (voir figure 4.17).

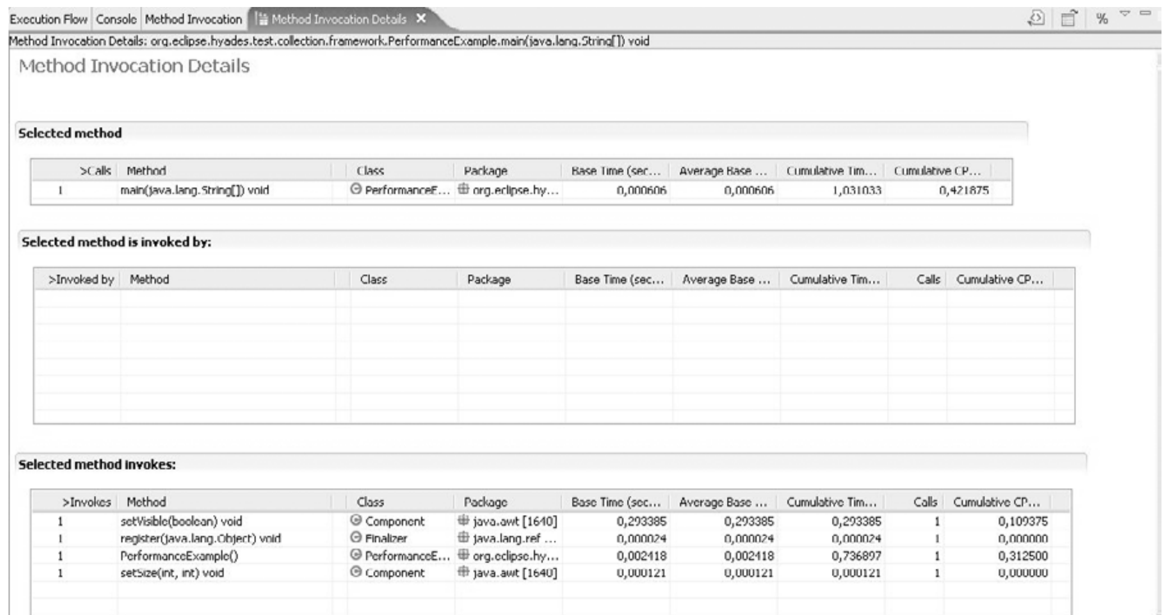


Figure 4.17

Détail des méthodes invoquées

Cette brève présentation du profiling de code avec la vue flux d'exécution vous a permis de mesurer toute la richesse de l'outil dans le domaine du profiling Java. Pour aller plus loin avec la solution TPTP, examinez avec attention les autres vues adaptées au profiling Java.

En résumé

Ce chapitre vous a permis de faire connaissance avec le projet Eclipse Test & Performance Tools Platform et d'apprécier toute la puissance de ses fonctions de visualisation.

Ces dernières permettent de localiser avec précision les performances et l'utilisation mémoire de vos applications, qu'elles soient de simples applications Java ou des applications Web complexes.

Partie II

Développement Web avec le projet Eclipse Web Tools

Cette partie est consacrée au développement Web avec Eclipse et le projet Web Tools.

Après un rappel des objectifs et la genèse de ce sous-projet Eclipse devenu partie intégrante de l'IDE Eclipse/Europa, nous passerons en revue le sous-projet Eclipse DTP (Data Tools Platform), qui propose une suite d'outils pour manipuler votre modèle physique de données et construire votre base de données.

Nous passerons ensuite à la mise en œuvre du développement Web de l'étude de cas webstock à l'aide de patterns standards et de la panoplie Web Tools.

Enfin, nous conclurons cette partie consacrée au développement Web avec le framework Seam, un framework très innovant pour le développement Web 2.0 et qui on le verra facilite grandement le développement JSF/EJB3.

5

Le projet WTP (Web Tools Platform)

Le projet Web Tools est constitué de sous-projets, en particulier JST (J2EE Standard Tools), destiné à fournir à la communauté l'outillage nécessaire au support de la spécification J2EE/JEE, et WST (Web Standard Tools), qui a pour but d'enrichir la plate-forme Eclipse d'outils de développement d'applications Web en utilisant J2EE.

Le projet JST forme l'ossature de la plate-forme Web Tools en fournissant un cadre de développement respectant la spécification J2EE, ainsi qu'un ensemble d'assistants pour le support du serveur d'applications JBoss, qui sert de cible et de support au déploiement de l'application développée.

Le projet WTP inclut les éléments suivants :

- éditeurs pour les langages de pages Web (HTML, CSS, JavaScript) ;
- éditeurs pour les documents XML et associés (XML, DTD, XSD et WSDL) ;
- support de projets J2EE *via* des assistants ;
- support des services Web *via* des assistants ;
- support des bases de données *via* SQL.

Le projet WTP est constitué de sous-projets, notamment les suivants :

- JST (J2EE Standard Tools). Accessible à l'adresse *http://www.eclipse.org/Web Tools/jst/main.php,* ce sous-projet propose des plug-ins pour faciliter le développement d'applications respectant la norme J2EE 1.4 (en particulier JSP, servlets, EJB, JCA, JMS, JNDI, JDBC, Java Web Services, JAX et JSR-associées). Le support pour les spécifications JCP (Java Community Process), accessible sur *http://www.jcp.org,* utilisées dans les applications Web mais non incluses dans la spécification J2EE 1.4, est adapté au cas par cas. Par exemple, JSF fait l'objet d'une implémentation particulière (voir la page du projet JSF *http://www.eclipse.org/jsf*). Rappelons que le rôle premier du JCP est de coordonner l'évolution du langage Java et d'en maintenir la cohésion et la compatibilité, notamment par le biais de certifications.

- WST (Web Standard Tools). Propose un socle pour le développement d'applications Web sous Eclipse.
- JSF (JavaServer Faces). Propose des plug-ins pour faciliter le développement d'applications Web utilisant les JSF.
- Dali. Propose des plug-ins pour faciliter le mapping O/R avec l'API JPA.
- ATF (Ajax Toolkit Framework). Supporte plusieurs conteneurs et serveurs d'applications, en particulier Tomcat (versions 3.2 à 5.5) et JBoss 4.*x* et 5.0 Bêta. Le site officiel du projet est à l'URL *http://www.eclipse.org/Web Tools/.*

JEE5

Bien que WTP supporte la notation J2EE 1.4, nous utiliserons par raccourci la nouvelle définition de la norme Java Platform Entreprise Edition, appelée JEE5, pour le support des JDK 1.5 et 1.6 en lieu et place de J2EE, qui supporte plus spécifiquement les versions des JDK 1.4/1.3/1.2.

Le sous-projet JST (J2EE Standard Tools)

JST est né d'un effort du team Eclipse pour fournir aux utilisateurs de la plate-forme Eclipse un framework standardisé pour la création d'outils de développement adaptés aux applications fondées sur la spécification J2EE.

Le sous-projet JST fonctionne en synergie avec le sous-projet WST (Web Standard Tools), que nous présentons à la section suivante, pour constituer le noyau du projet WTP (Web Tools Platform) en offrant toute une panoplie d'outils de développement, de test, de déploiement et d'administration des applications Web fondées sur le langage Java.

JST est surtout spécialisé dans le support des technologies J2EE, même si certaines technologies hors de ce scope sont également supportées, *via* le standard JCP, à l'image de XDoclet, la populaire technologie d'annotation des sources. D'autres technologies, comme Hibernate ou Velocity, bien que très connues dans le monde du développement Java, ne sont pas supportées par ce projet.

En résumé, JST permet « l'incubation » de projets potentiellement porteurs grâce à certaines fonctionnalités qu'il embarque et qui peuvent être utilisées par le biais de plug-ins tiers supportant ces projets.

Périmètre de JST

JST fournit à Eclipse un ensemble d'outils et de technologies standards pour le développement d'applications Java conformes à la spécification J2EE 1.4.

Le tableau 5.1 récapitule les standards JCP supportés par JST.

Tableau 5.1 Standards JCP supportés par JST

JSR-3 Java Management Extensions (JMX) 1.2
JSR-5 Java API for XML Parsing (JAXP) 1.2
JSR-45 Debugging Support
JSR-54 JDBC API 3.0
JSR-67 SOAP with Attachments API for Java (SAAJ) 1.2

Tableau 5.1 Standards JCP supportés par JST *(suite)*

JSR
JSR-77 J2EE Management API 1.0
JSR-88 Deployment API 1.1
JSR-93 Java API for XML Registries (JAXR) 1.0
JSR-101 Java API for XML-based RPC (JAX-RPC) 1.1
JSR-109 Web Services
JSR-112 J2EE Connector Architecture (JCA) 1.5
JSR-115 Java Authorization Contract for Containers (JACC)
JSR-152 JavaServer Pages (JSP) 2.0
JSR-153 Enterprise JavaBeans (EJB) 2.1
JSR-181-Metadata for Web Services,
JSR-154 Servlets 2.4
JSR-175: Metadata Facility for the Java^{TM} Programming Language
JSR-907 Java Transaction API (JTA) 1.0
JSR-914 Java Message Service (JMS) 1.1
JSR-919 JavaMail 1.3

En complément, les standards et technologies non-JCP suivants sont inclus dans le périmètre de JST :

- JAAS (Java Authentication and Authorization Service)
- JNDI (Java Naming and Directory Interface)
- XDoclet

Architecture du sous-projet JST

Comme indiqué précédemment, JST fournit un socle pour le développement d'outils J2EE dédiés. Appelé JCM (J2EE Core Model), ce socle contient un ensemble de frameworks et d'objets du modèle permettant d'abstraire les principales fonctionnalités J2EE ainsi que les composants associés et de fournir un ensemble d'API d'accès pour manipuler ces fonctionnalités.

Le modèle JCM est accessible à d'autres éditeurs ou développeurs pour étendre les outils de développement J2EE déjà disponibles dans la plate-forme Eclipse. Comme nous l'avons vu, il fournit en outre un support pour d'autres technologies prometteuse non encore directement supportées par Eclipse. Précisons que les outils fournis par le sous-projet JST sont eux-mêmes des extensions du modèle JCM.

Les extensions du modèle JCM sont les suivantes :

- Modèle de projet J2EE. Fournit un framework pour la gestion de projets J2EE et supporte une structure flexible pour la gestion des composants J2EE et le support des activités associées (compilation, déploiement, etc.).
- Modèle d'éditeur J2EE. Étend l'éditeur standard Eclipse pour fournir un support pour la création et l'édition de ressources J2EE, comme les JSP, les servlets, les EJB, etc. Il fournit en outre la base pour la création d'éditeurs spécifiques pour les textes et les images, en particulier pour le support de fonctionnalités d'édition de syntaxe colorée, d'assistant de code, de refactoring de code, etc.

- Modèle d'artefacts JEE. Représente les sources JEE ainsi que les composants JSP, EJB, descripteurs de déploiement, etc., qui peuvent être créés et gérés au sein d'un projet en même temps que d'autres artefacts, comme les ressources de type image, texte et autres fichiers qui peuvent être packagés dans un module J2EE déployable (war/ejb, jar/rar).
- Modèle de serveur JEE. Fournit les abstractions requises pour supporter le déploiement de modules pour différents types de serveurs d'applications. Il offre de manière additionnelle un mécanisme unifié pour démarrer, administrer et stopper les serveurs d'applications JEE. Ce modèle fournit le moyen de gérer les détails de configuration des serveurs, comme les paramètres JVM, les variables d'accès aux chemins (classpath), etc. Enfin, les modules J2EE tels que le packaging, le déploiement, le débogage ou l'import/export de modules J2EE sont gérés par ce modèle.

L'outillage J2EE standard de JST

Examinons à présent l'outillage JST permettant la création, le support et la gestion d'artefacts JEE. Ces outils sont créés en étendant les modèles et les API rendus disponibles par JCM. Le résultat est un ensemble de vues, de perspectives, d'éditeurs et d'assistants dédiés aux différentes activités de développement.

Chaque outil est dédié à une activité particulière du développement d'applications d'entreprise J2EE. Ainsi, les outils dédiés au support des servlets/JSP sont utiles au développement d'applications Web et à leur déploiement dans le conteneur, tandis que les outils prévus pour le support des EJB permettent de se concentrer sur le développement et le déploiement sur le conteneur d'EJB.

Notion de projets J2EE et de modules

Un projet Eclipse est utilisé pour développer des modules applicatifs J2EE Web, ou EJB. Chaque module J2EE est en fait développé dans un projet séparé et possède son propre accès aux classes définies dans le JDT (outillage Java d'Eclipse). Il ne faut toutefois pas confondre module et projet. Il est ainsi possible que différents projets partagent un même module (par exemple, un module jar contenant les bibliothèques partagées par le projet J2EE).

Un projet Eclipse « basique » fournit une structure pour organiser les ressources dans le projet (ressources de types fichiers et répertoires) :

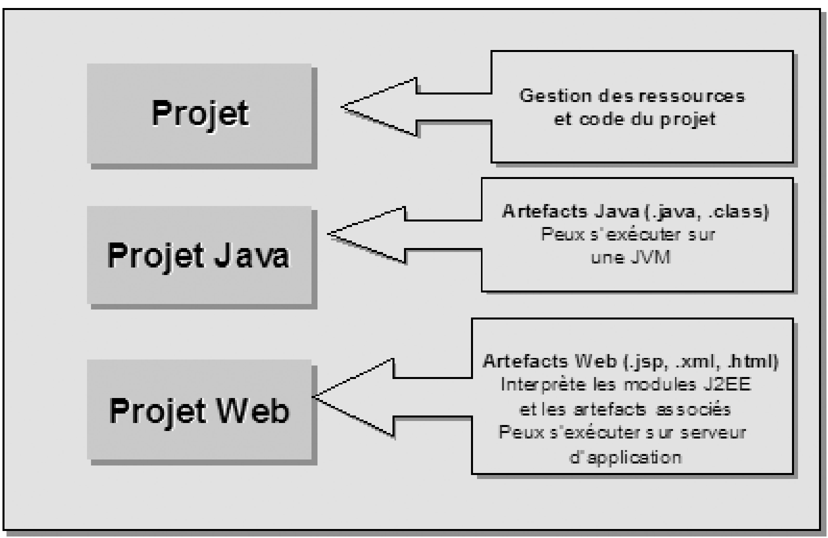
- Projet Java, qui contient les éléments et artefacts (packages, méthodes, classes et attributs) nécessaires au développement Java.
- Projet Web, qui contient en plus du code Java des ressources pour le support du développement Web (bibliothèques Struts, fichiers html, servlets, JSP, bibliothèques de balises, services Web et descripteurs de déploiement).

La figure 5.1 illustre l'organisation en composants d'un projet Eclipse.

La figure 5.2 décrit les constituants du packaging final d'une application JEE, répartis selon les modules projets précédents, l'ensemble de ces modules étant packagés dans un fichier EAR avec leurs descripteurs respectifs.

Figure 5.1

Modules projet d'un projet Eclipse



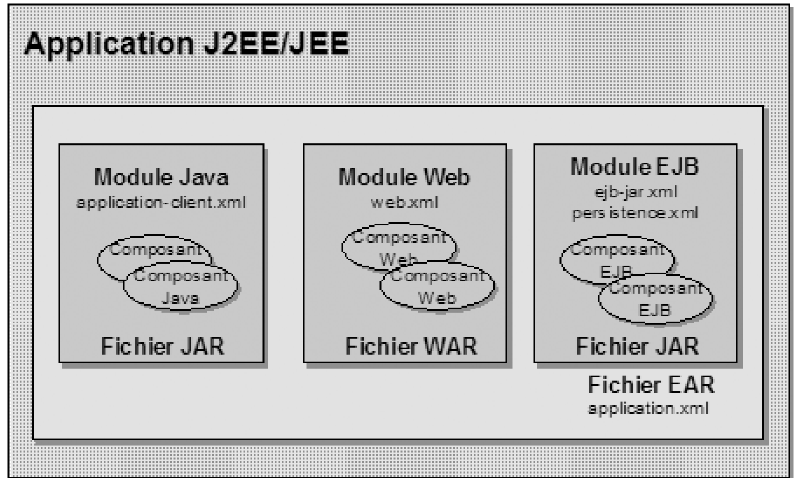


Figure 5.2

Packaging final d'une application J2EE/JEE

Outils de support à la création de projets J2EE

L'outillage de support à la création de projets J2EE permet la création d'un ensemble d'artefacts qui constituent le module de déploiement spécifique de l'application J2EE.

Cet outillage supporte la création de cinq types de projets (voir figure 5.3) :

- Projet EJB (EJB Project) : ensemble de composants beans d'entreprise supportant les artefacts qui seront compilés et déployés au sein d'un conteneur d'EJB sur le serveur d'applications cible.
- Projet de type Web dynamique : ensemble de composants servlets, JSP et Web tiers (bibliothèque de balises, pages HTML, images et documents) qui seront compilés et déployés au sein d'un conteneur Web sur le serveur d'applications cible.

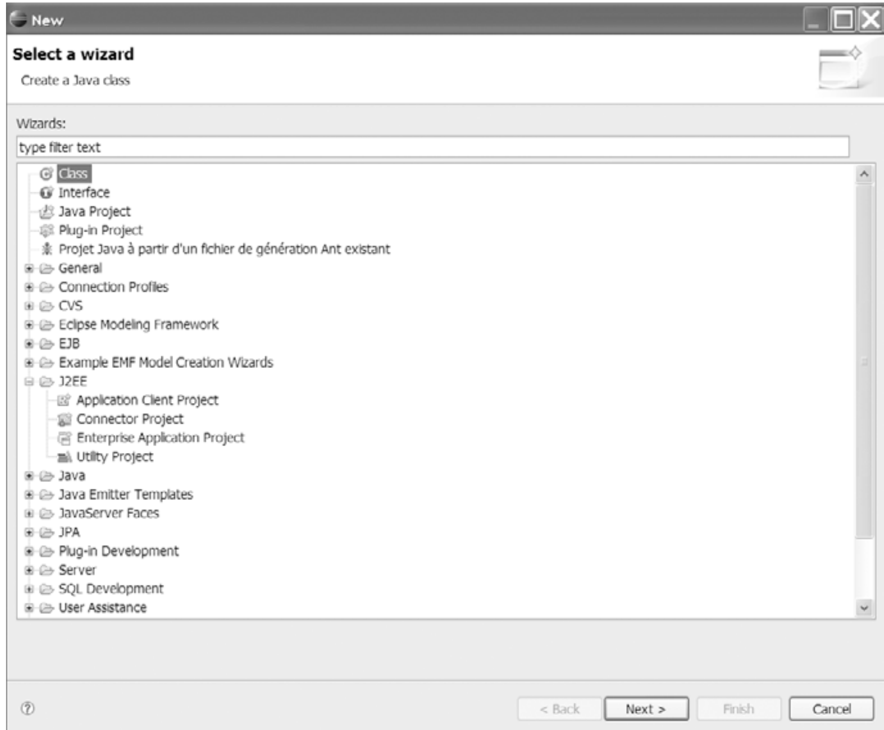
- Projet de client d'application (Application Client Project) : ensemble de composants client conçus pour la consommation de services fourni par une application d'entreprise.
- Projet connecteurs (Connector Project) : ensemble de fichiers source destinés à la création de connecteurs applicatifs pour l'intégration aux systèmes « legacy » spécifiés par la JSR-112 (J2EE Connector Architecture 1.5).
- Projet d'application d'entreprise (Enterprise Application Project) : ensemble de modules représentant une application d'entreprise complète. Ce type de projet fournit la possibilité de référencer différents composants de type EJB, Web, client d'application et connecteur qui seront déployés ensemble.

Les modules J2EE additionnels fournissent de nombreuses fonctionnalités pour la création de projet, notamment la possibilité de créer et de gérer des artefacts selon les types de projets ci-dessus et, surtout, de créer une structure de déploiement type pour le support au projet J2EE (type WAR pour un projet Web, ou EAR pour une application d'entreprise, par exemple).

La figure 5.3 illustre l'assistant de création de projet JST proposant un certain nombre d'options pour un projet d'entreprise J2EE.

Figure 5.3

Assistant de création de projet JST



Europa

À l'heure de la rédaction de cet ouvrage, la version Eclipse 3.3 « Europa » ne dispose pas d'un support de la langue française. Nous nous référons donc à la version anglaise en notre possession.

Outils de support aux serveurs J2EE

Les applications J2EE doivent être déployées dans un conteneur Web ou EJB d'un serveur d'applications J2EE/JEE compatible. Avec JST, la gestion des serveurs sur lesquels les applications J2EE seront déployées est gérées par l'outillage du serveur J2EE.

L'outillage de support aux serveur fournit un mécanisme pour la définition de l'environnement d'exécution du serveur J2EE/JEE ainsi que pour la création d'instances du serveur. Cet outillage assure que, lors de la création d'un projet, ce dernier puisse être déployé.

Comme l'illustre la figure 5.4, lors de la création d'un projet de type Enterprise Application Project, l'assistant de support au projet intégré à JST propose un certain nombre de serveurs d'applications compatibles.

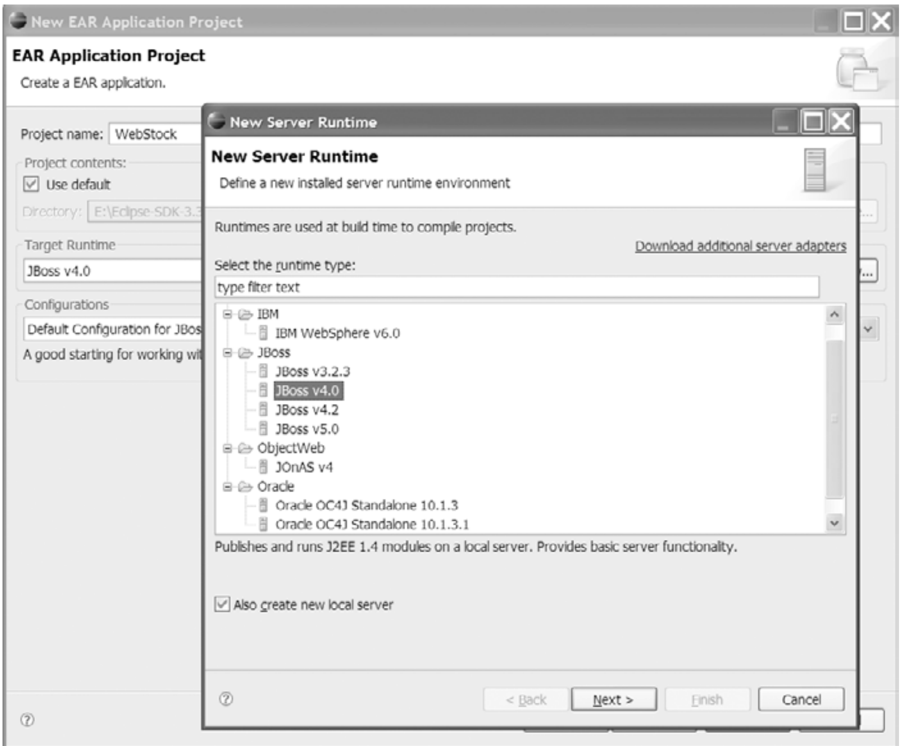


Figure 5.4

Support aux principaux serveurs J2EE/JEE compatibles

Avec JST, un projet J2EE/JEE nécessite l'existence d'une JRE (Java Runtime Environment) et d'un environnement d'exécution J2EE/JEE compatible. L'environnement d'exécution J2EE/JEE fournit un support pour les fonctionnalités d'entreprise utilisées dans le projet. JST supporte la définition de plusieurs versions d'environnements d'exécution J2EE/JEE permettant la création de projets déployables sur différentes versions de JVM et sur différentes versions de serveur d'applications cible.

Le tableau 5.2 décrit les serveurs d'applications supportés par JST.

Tableau 5.2 Serveurs d'applications et spécifications J2EE supportés par JST

Version Web	Version EJB	Serveur d'applications J2EE/JEE
2.2, 2.3 et 2.4	1.1, 2.0 et 2.1	Apache Geronimo 1.0
2.2		Apache Tomcat version 3.2
2.2 et 2.3		Apache Tomcat version 4.1
2.2, 2.3 et 2.4		Apache Tomcat version 5.0
2.2, 2.3 et 2.4		Apache Tomcat version 5.5
2.2, 2.3,2.4 et 2.5		Apache Tomcat 6.0.13
2.2, 2.3 et 2.4	1.1, 2.0 et 2.1	IBM WebSphere 6.0.x
2.2, 2.3, 2.4	2.0 et 2.1	BEA WebLogic Server version 8.1
2.2, 2.3 et 2.4	1.1, 2.0 et 2.1	BEA WebLogici Server version 9.0
2.2, 2.3 et 2.4	1.1 et 2.1	JBoss 3.2.3
2.2, 2.3 et 2.4	2.0 et 2.1	JBoss 4.0
2.2, 2.3 et 2.4	2.0 et 2.1	JBoss 4.2
	3.0	JBoss 5.0 (Beta)
2.2, 2.3 et 2.4	1.1, 2.0 et 2.1	Jonas V4

La figure 5.5 illustre la définition d'une cible serveur d'applications JBoss 4.0.5 (accessible *via* le menu Preferences d'Eclipse).

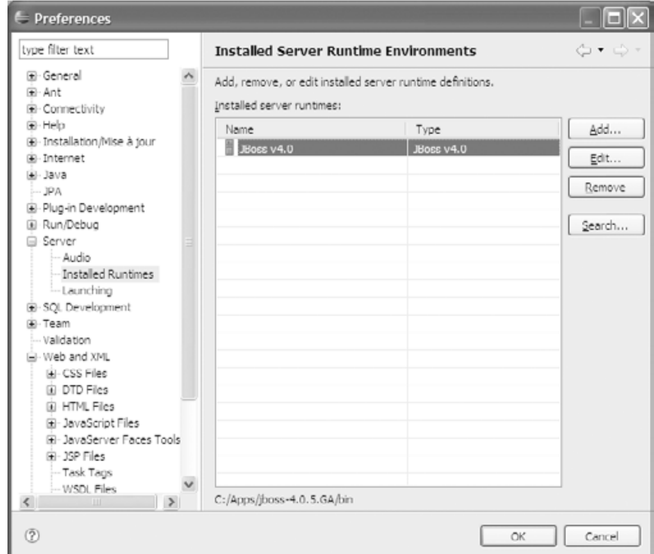


Figure 5.5

Définition d'un environnement serveur JBoss 4.0.5

Des configurations supplémentaires sont disponibles pour chaque environnement cible défini sur le serveur permettant d'agir sur les variables d'accès aux classes, ainsi que sur les arguments d'exécution de la JVM et de démarrage de l'instance.

La figure 5.6 illustre une configuration par défaut définie pour le serveur d'applications JBoss 4.0 (vous pouvez aussi agir sur le mode de publication et de déploiement des composants serveur *via* l'option Automatic Publishing).

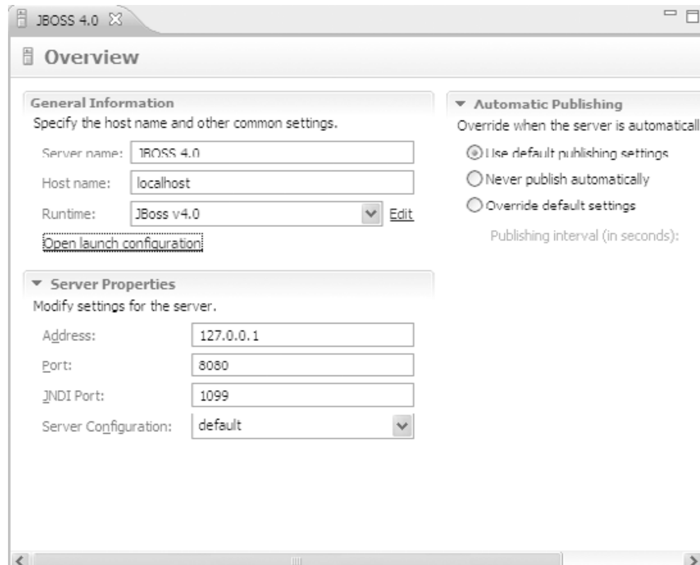


Figure 5.6

Configuration d'un environnement d'exécution serveur (JBoss)

Le sous-projet WST (Web Standard Tools)

Cette section donne une vue d'ensemble des principaux composants du sous-projet WST. Pour en savoir plus sur ce projet, voir *http://www.eclipse.org/Web Tools/wst/components.html.*

Ce sous-projet inclut des outils et des API offrant le support d'applications Web standards indépendantes de la technologie sous-jacente.

WST fournit des fonctionnalités de base qui peuvent être étendues par d'autres projets. Les sections qui suivent donnent des exemples de facilités offertes par ce sous-projet, partie intégrante du projet Web Tools.

Support des facets projet

Parmi ces fonctionnalités, la notion de Project Facets est la plus importante.

Les facets permettent d'ajouter des fonctionnalités à un projet Eclipse et de les supprimer ensuite au besoin si ceux-ci ne sont plus requis et que votre projet supporte d'autres technologies.

En résumé, une facet est une caractéristique que le projet peut supporter (par exemple JSF). Web Tools définit quelques types de projets relativement génériques (par exemple Dynamic Web Project) et offre la possibilité d'ajouter et de supprimer dynamiquement des fonctionnalités à ces projets.

Nommées Project Facets, ces fonctionnalités sont configurables lors de la création du projet et par la suite dans la page des propriétés du projet (voir figure 5.7).

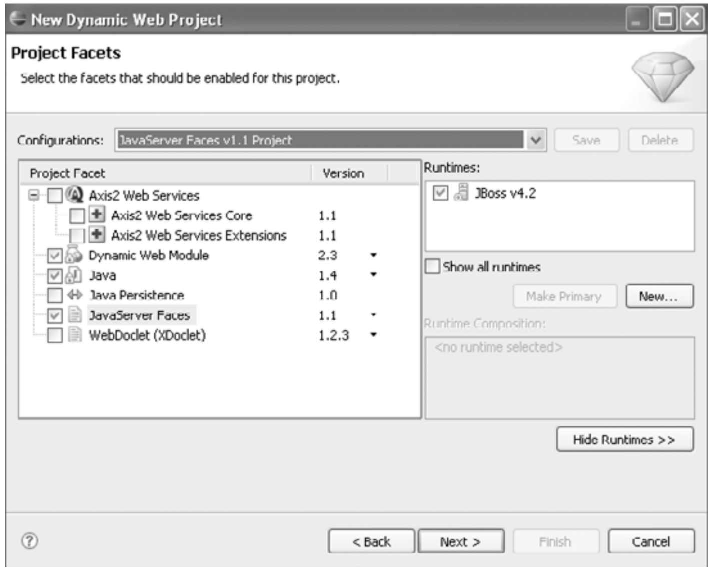


Figure 5.7

Facets proposées par WST

Support à l'outillage HTML, CSS et JavaScript

Comme autre apport remarquable apporté par WST, citons le support aux ressources HTML, CSS et JavaScript.

WST propose des éditeurs de code source pour chacun de ces langages (syntaxe colorée, aide à la complétion, etc.), comme l'illustre la figure 5.8.

Figure 5.8

Aide au support des ressources HTML avec le projet WST

Support des Web Services

WST supporte les moteurs SOAP, en particulier Apache Axis 1.2.1. Pour le développement de Web Services, les trois fonctionnalités les plus visibles sont un éditeur de fichier WSDL (ressemblant à l'éditeur de fichier XMLSchema), des assistants pour la création de Web Services (génération à partir d'une classe Java) et un outil permettant d'invoquer un service Web depuis l'explorateur de services Web (voir figure 5.9).

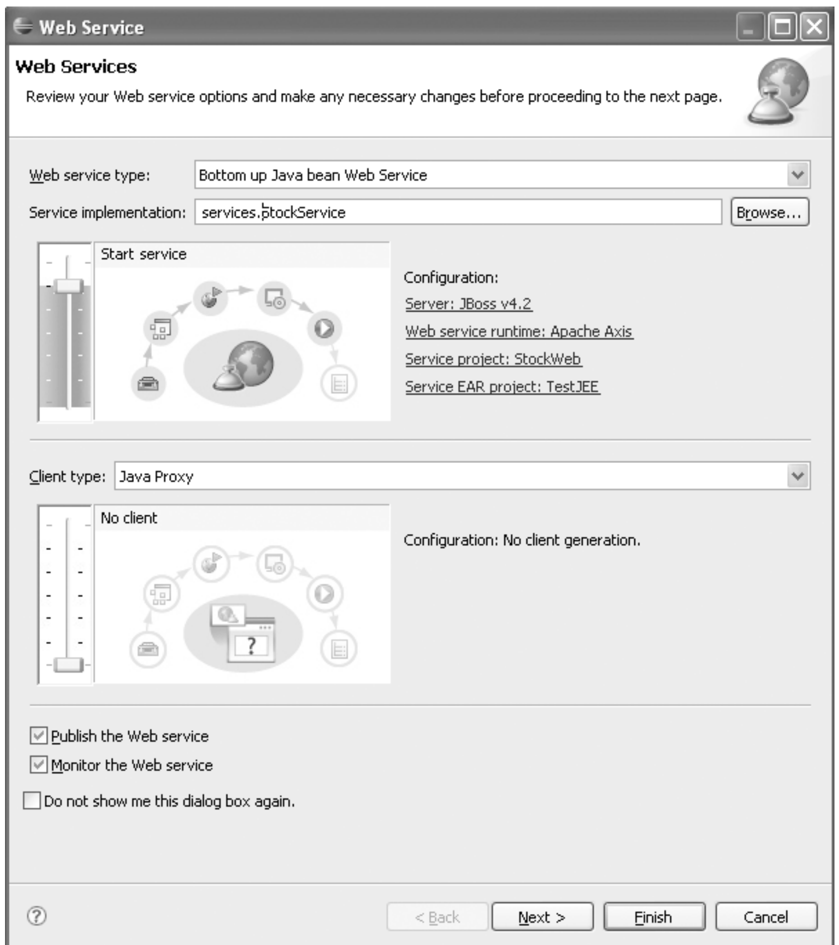


Figure 5.9

Assistant WST de création et de publication de service Web

L'assistant WST de création et de publication de service Web offre les avantages suivants :

- Développement de services Web de type bottom-up/top-down (à partir d'un JavaBean existant ou non).
- Démarrage de services Web dans un projet Web.
- Génération de proxy Java.
- Test de services Web.
- Monitoring de services Web. Web Tools inclut un moniteur embarqué qui permet de surveiller les enveloppes SOAP et les flux entre le client et le service invoqué.

Monitoring TCP/IP

WST inclut un navigateur Web permettant d'accéder aux pages Internet à l'intérieur de l'IDE mais aussi de mesurer le trafic HTTP échangé, permettant ainsi la capture et l'analyse des messages envoyés et reçus à partir d'un port et d'un hôte spécifique.

L'outil de monitoring TCP/IP permet sauvegarde les messages dans un fichier de log, lequel peut ensuite être analysé avec la suite d'outils de test intégré (voir figure 5.10).

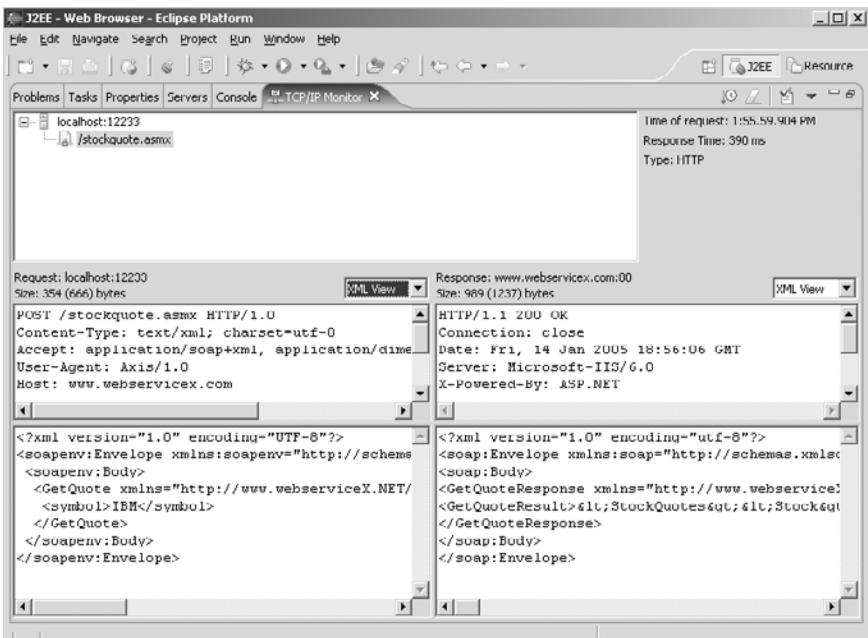


Figure 5.10

Outil de monitoring TCP/IP de WST

Mise en œuvre de WTP

Avant de commencer à utiliser les assistants de développement Web, vous allez commencer par installer et configurer WTP.

La version utilisée ici est celle supportée par Eclipse 3.3 (Europa), à savoir la 2.0 RC4.

La version de WTP 2.0 disponible en téléchargement sur le site du projet *(http://download.eclipse.org/Web Tools/downloads/)* suppose l'installation préalable sur la plate-forme Europa des composants récapitulés au tableau 5.3.

Tableau 5.3 Composants à installer sur Europa

Composant	Version
Eclipse Platform (JDT, PDE)	Eclipse SDK 3.3 (Europa)
Eclipse Modeling Framework (EMF, XSD InfoSet, SDO)	emf-sdo-xsd-SDK-2.3.0RC4
GEF (Graphical Editing Framework)	GEF-SDK-3.3RC4
DTP (Data Tools Platform)	dtp-sdk_1.5RC4

L'installation de ces composants ne présente aucune difficulté particulière et s'effectue, comme pour la distribution d'Eclipse, par une décompression des binaires dans le répertoire d'installation d'Eclipse (par exemple, sous Windows c:\MonEclipse\3.3\eclipse). Une fois ces composants téléchargés, prenez soin de lancer l'IDE Eclipse par le biais de l'option –clean (sous c:\MonEclipse\3.3\eclipse\bin).

Configuration de l'environnement d'exécution

Vous allez commencer par créer un environnement d'exécution Java. Cette étape initiale est indispensable, car la création d'un environnement d'exécution J2EE dépend de l'existence d'une JRE installée et fonctionnelle, accessible *via* Preferences, Java, JRE installés (voir figure 5.11).

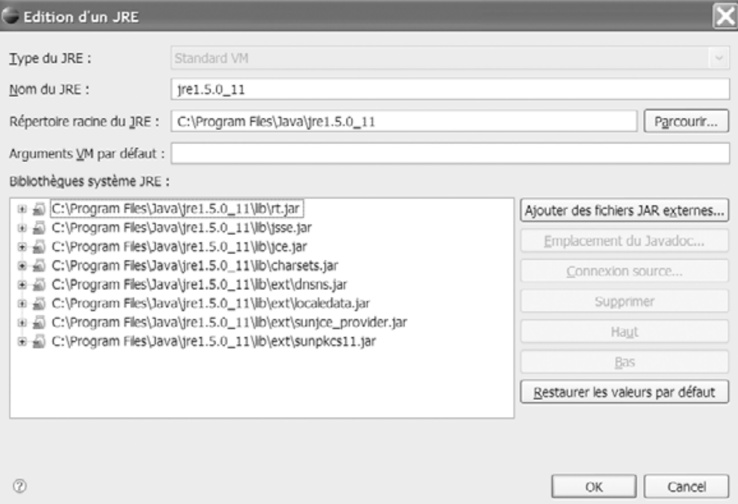


Figure 5.11

Configuration de la JRE

> **Version de JRE**
>
> Si vous ne connaissez pas l'emplacement de votre JRE, vous pouvez repérer celle-ci en consultant la variable JAVA_HOME définie dans votre système.

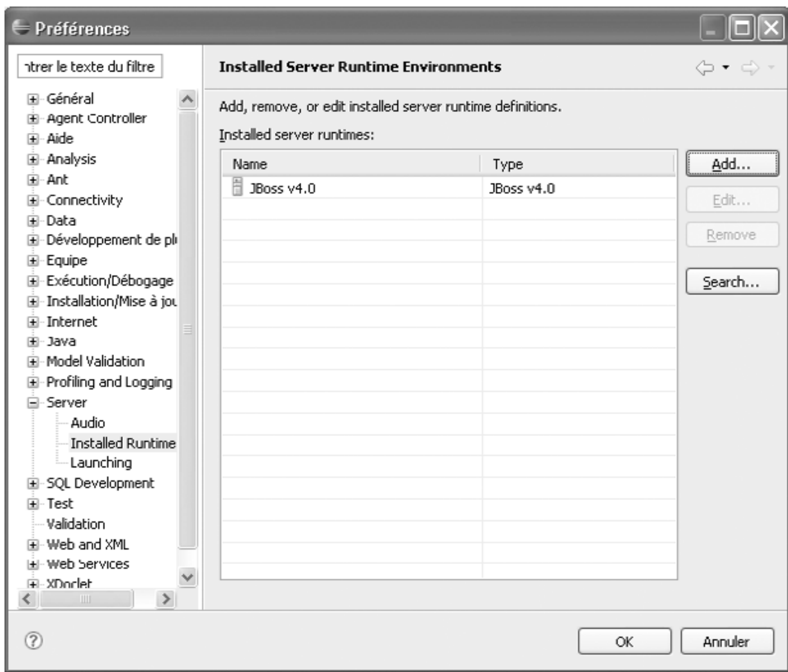
Dans cette configuration, optez pour une JRE 1.5.0 Update 11. La sélection du répertoire d'installation de la JRE installe toute les bibliothèques associées à cette dernière, sachant qu'il est aussi possible d'ajouter des bibliothèques jar additionnelles dans l'environnement d'exécution de la JVM.

L'étape suivante consiste à configurer l'environnement d'exécution du serveur J2EE offrant le support des caractéristiques spécifiques qui seront utilisées au sein de l'application J2EE déployée. À l'inverse de la configuration de la JRE vue précédemment, qui est généralement fournie par Sun Microsystems, le runtime J2EE est disponible à partir d'une variété d'éditeurs proposant leur propre implémentation de serveurs J2EE compatibles (généralement inclus dans l'environnement du SA).

La configuration du serveur est accessible par le biais de Window, Preferences, Installed Runtimes et Add. L'assistant de définition et de configuration s'affiche alors comme illustré à la figure 5.12 (ici avec JBoss).

Figure 5.12

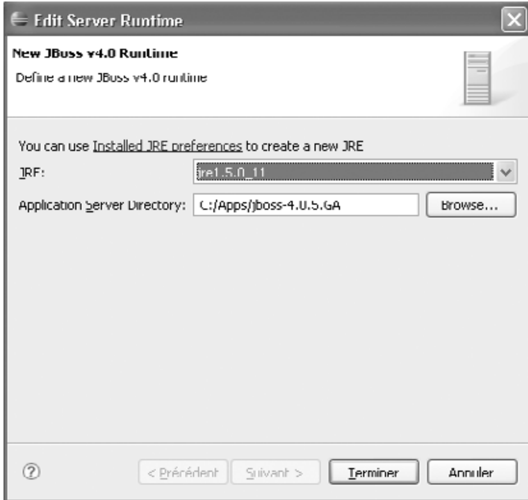
Configuration du serveur JBoss (1/2)



L'assistant de configuration du serveur permet, en fonction du serveur cible présélectionné, de définir la localisation de la JRE, ainsi que du serveur d'applications, avec intégration immédiate des bibliothèques jar disponibles pour ce serveur (voir figure 5.13).

Figure 5.13

Configuration du serveur JBoss (2/2)



Eclipse supporte la création de plusieurs environnements J2EE, chaque environnement étant dédié à un serveur d'applications particulier installé sur le système.

Il est possible de créer plusieurs configurations d'exécution fondées sur des installations particulières du serveur d'applications cible. Cela peut se révéler utile notamment pour tester l'application en utilisant différentes versions de JRE.

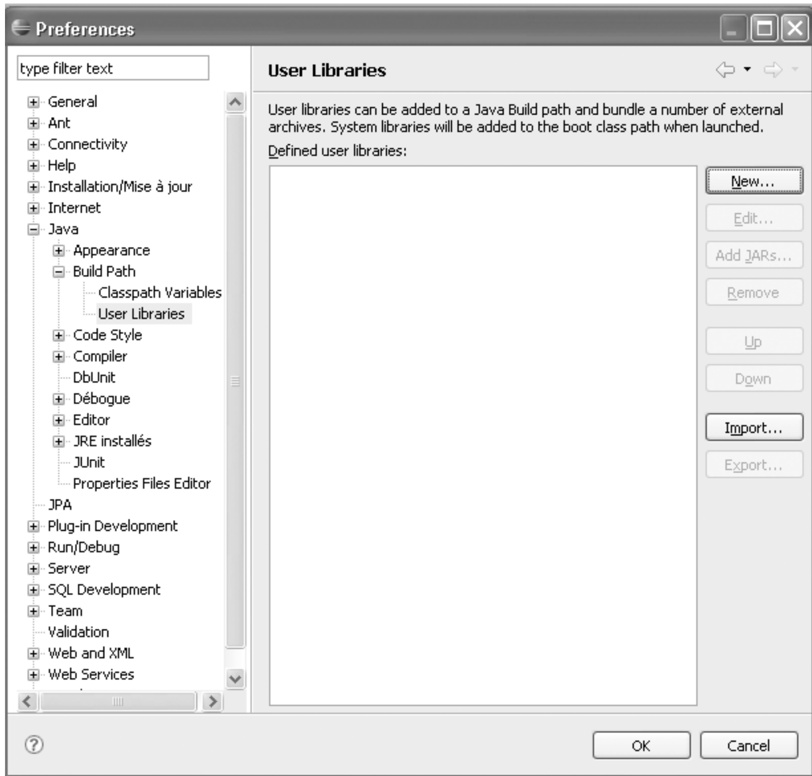
Configuration des bibliothèques additionnelles spécifiques

Dans certaines situations, il peut être nécessaire au support de certaines fonctionnalités du serveur d'applications cible d'ajouter certaines bibliothèques, appelées User Librairies, ou bibliothèques utilisateur, accessibles par le biais des options Java, Build Path et User Libraries du menu Préférences d'Eclipse (voir figure 5.14).

C'est en particulier le cas si vous utilisez la version du serveur JBoss 4.0.5 illustré dans cette section, pour supporter par exemple la technologie EJB3 du serveur JBoss (voir figure 5.14).

Figure 5.14

Ajout de bibliothèques additionnelles



Dans le contexte du développement EJB3, vous devez utiliser certaines bibliothèques de support à cette technologie pour le serveur JBoss 4.0.5.

Procédez pour cela de la façon suivante :

1. Cliquez sur New, et nommez votre bibliothèque JBOSS_EJB3.
2. Ajoutez les jar suivants (*via* le bouton Add JARS) :

```
$JBOSS_HOME/server/default/lib/ejb3-persistence.jar
$JBOSS_HOME/server/default/deploy/ejb3.deployer/jboss-ejb3.jar
$JBOSS_HOME/server/default/deploy/ejb3.deployer/jboss-ejb3x.jar
$JBOSS_HOME/server/default/lib/jboss-j2ee.jar
```

La configuration de votre bibliothèque doit ressembler à celle illustrée à la figure 5.15 (en fonction du répertoire d'installation de votre serveur d'applications JBoss).

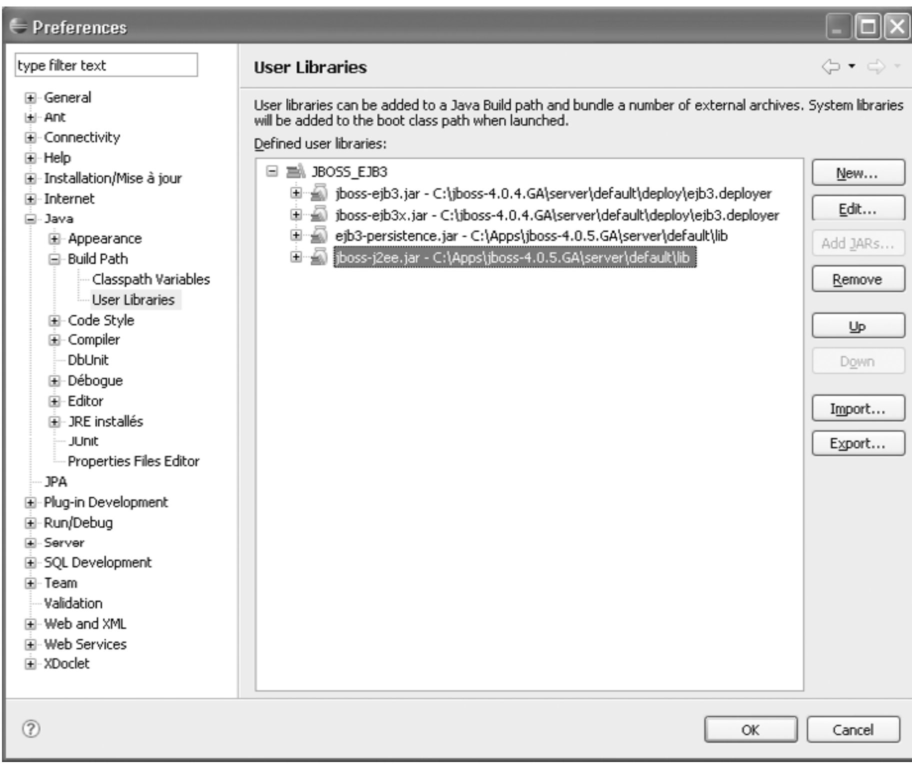


Figure 5.15

Configuration des bibliothèques JBoss de support aux technologies EJB3

3. Cliquez sur OK pour terminer l'opération. Vous intégrerez ensuite celle-ci à votre projet.

Sans anticiper sur les fonctionnalités du serveur JBoss et les configurations spécifiques aux projets EJB3, que nous présentons un peu plus loin dans cet ouvrage, cette étape de définition des bibliothèques spécifiques au projet est cruciale dans la configuration du poste de développement afin de permettre une standardisation des développements.

Passez à présent aux fonctionnalités spécifiques de support au développement Web apportées par le projet JST pour la configuration d'un projet Web.

> **Support des fichiers jar externes**
>
> L'ajout et la définition de bibliothèques externes au chemin de compilation du projet pour l'utilisation de l'API EJB3 n'est plus nécessaire avec les versions supérieures à JBoss 4.2.

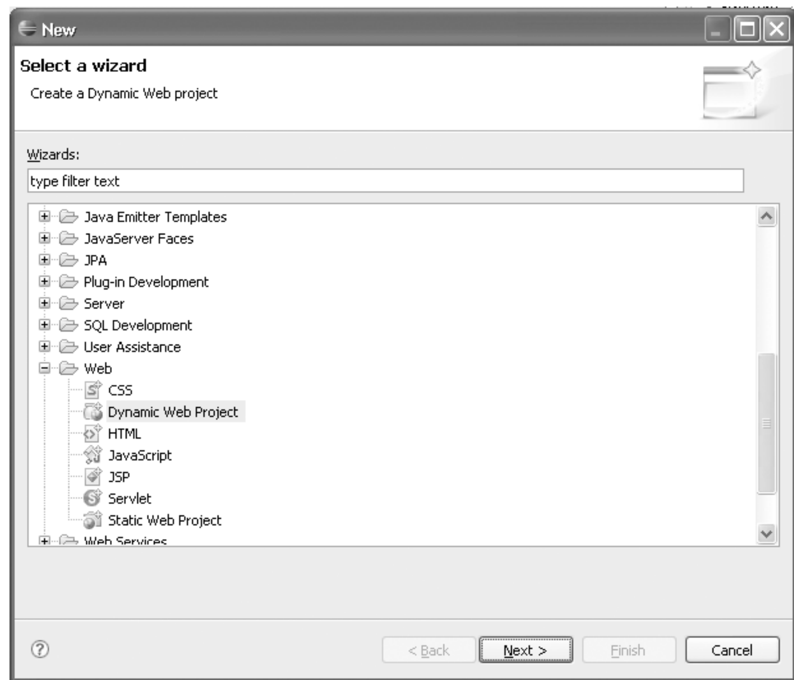
Configuration d'un projet de développement Web

Vous allez définir votre projet comme étant de type Dynamic Web, c'est-à-dire de support aux technologies JSP et servlets.

1. Sélectionnez l'assistant de création de projet Eclipse *via* Fichier, Nouveau, Autre et Dynamic Web Project (voir figure 5.16).

Figure 5.16

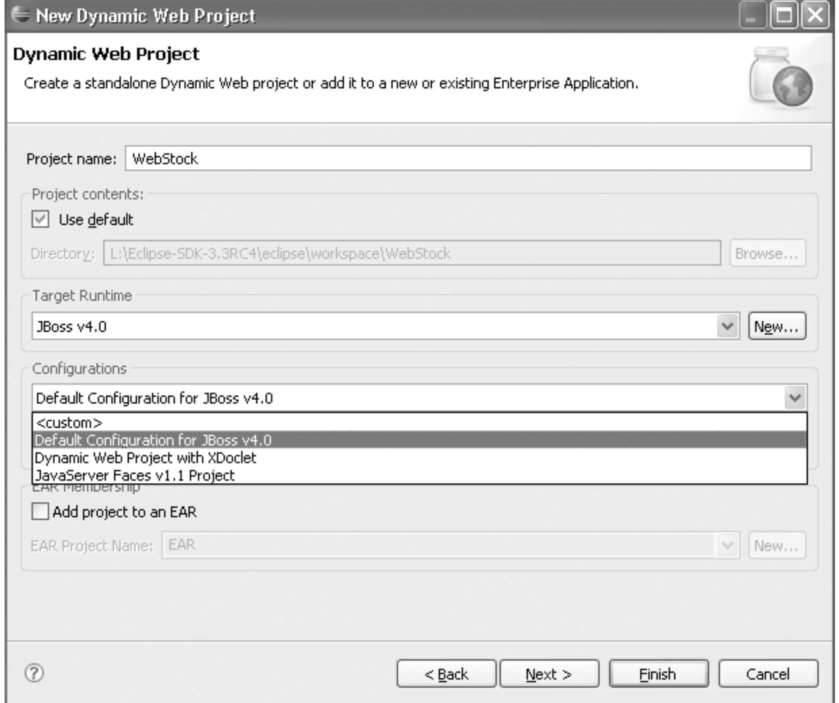
Assistant de création de projet Web dynamique



2. Cliquez sur Next, puis entrez WebStock devant le nom du projet (voir figure 5.17). Notez que l'environnement d'exécution cible du projet (champ Target Runtime) pointe sur la cible SA définie précédemment (JBoss 4.0).

Figure 5.17

Configuration du projet Web dynamique



Notez que, depuis Web Tools 2.0, trois types de configurations spécifiques sont proposés :

– Défault Configuration for JBoss v4.0, par défaut pour le développement d'applications sur Jboss 4.0.

– Dynamic Web Project with XDoclet, pour le support des annotations avec le standard XDoclet.

– JavaServer Faces v1.1 Project, pour le support de l'implémentation de référence de la technologie JSF.

3. Choisissez la première option, puis cliquez sur Next (voir figure 5.18).

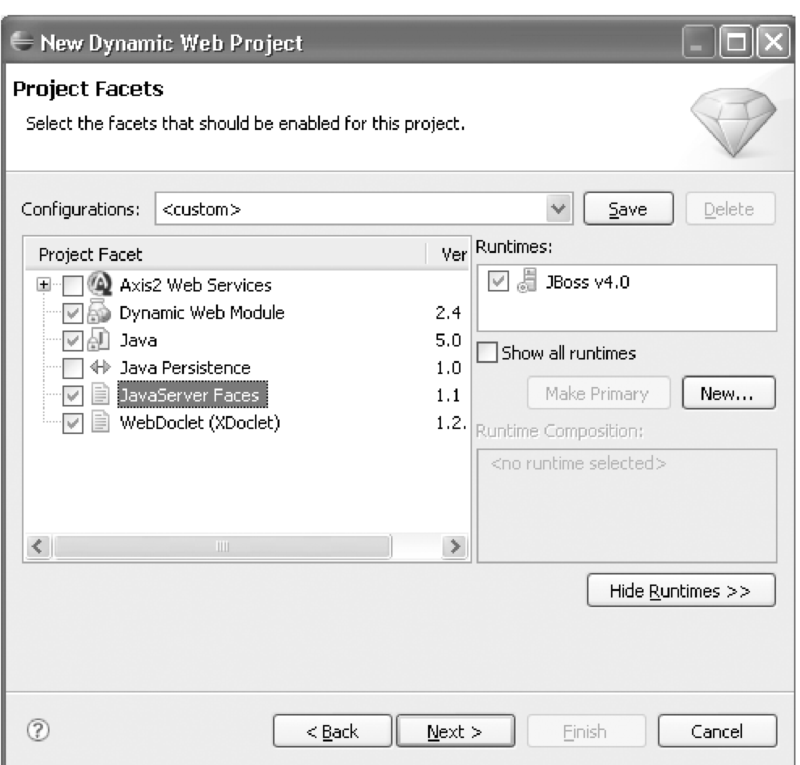


Figure 5.18

Configuration du projet Web dynamique avec l'option Project Facet

Project Facet

Comme évoqué plus haut, les fonctionnalités Project Facet permettent de gérer toute la variété de projets et de supports des différentes variétés de frameworks. Web Tools offre la possibilité d'ajouter et de supprimer dynamiquement ces fonctionnalités aux projets. Les Project Facets sont configurables lors de la création du projet et par la suite dans la page des propriétés du projet.

4. Sélectionnez les facets JavaServer Faces 1.1 et WebDoclet (XDoclet) 1.2, puis cliquez sur Next.

5. Dans la boîte de dialogue qui s'affiche, configurez le module Web :

– Context Root : permet de définir le chemin virtuel au sein duquel le contenu de l'application Web sera accédé sur le SA.

– Content Directory : répertoire contenant les artefacts Web (fichiers HTML, JSP, fichiers graphiques, etc.).

– Java Source Directory : répertoire contenant les sources des classes, des servlets et des beans. Lorsque ces ressources sont ajoutées au projet Web, celles-ci sont automatiquement compilées, et le résultat compilé est ajouté au répertoire WEB-INF/classes.

6. Cliquez sur Next. La page illustrée à la figure 5.19 s'affiche.

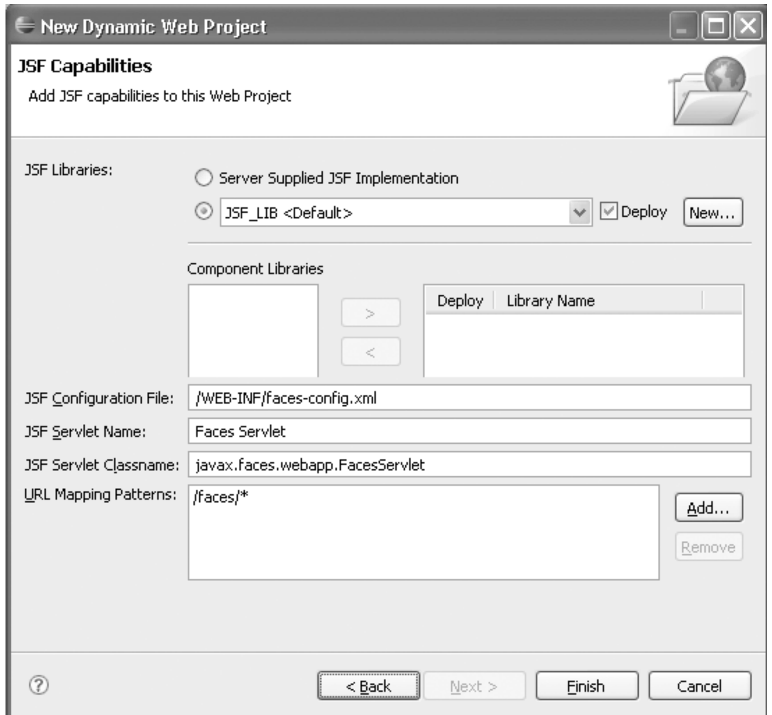


Figure 5.19

Options des Project Facets

Cet assistant configure les principaux fichiers de configuration JSF (faces-config.xml, répertoire de mapping des URL, etc.). Pour utiliser la fonctionnalité de référence de la technologie JavaServer Faces version 1.1, il faut télécharger l'implémentation de référence de cette technologie, à l'adresse *http://java.sun.com/javaee/javaserverfaces/download.html).*

7. Cliquez sur New pour créer la bibliothèque et l'associer au jar téléchargé contenant les bibliothèques JSF 1.1.

8. Cliquez sur Finish pour terminer la configuration du projet Web dynamique, puis cliquez sur la nouvelle perspective fournie par JST, *via* Fenêtre, Ouvrir la perspective, Autre et Java JEE. Vous devez voir s'afficher l'arborescence de votre projet JEE comme illustré à la figure 5.20.

   La vue d'affichage des erreurs laisse apparaître un problème de configuration de l'outil de génération de code XDoclet, qui permet le développement orienté attribut (Attribut-Oriented Programming). Il a surtout conquis ses lettres de noblesse pour fournir, grâce aux très populaires outils de build Ant, une solution de génération multi-AS et multi-technologies J2EE (Struts, JSF, EJB, etc.).

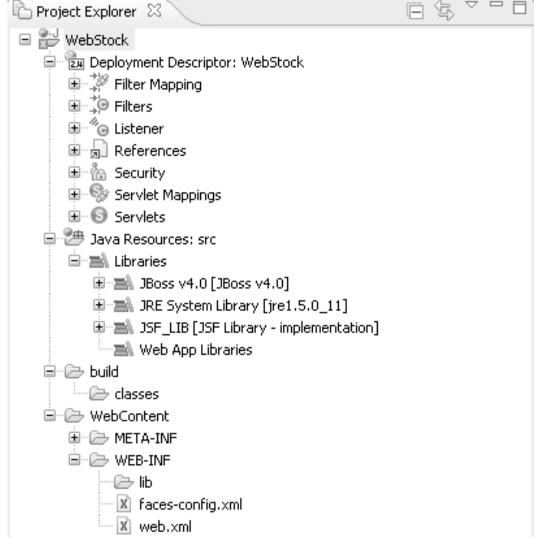


Figure 5.20

Arborescence du projet Web

9. Pour l'utiliser et l'intégrer dans Eclipse, téléchargez la dernière version de l'outil à partir du site de référence *http://xdoclet.sourceforge.net/*. À ce jour, la version la plus à jour est la 1.2.3.
10. Décompressez la distribution dans un répertoire dédiée (par exemple, L:\XDoclet1.2.3).
11. Faites pointer la version de XDoclet comme illustré à la figure 5.21.

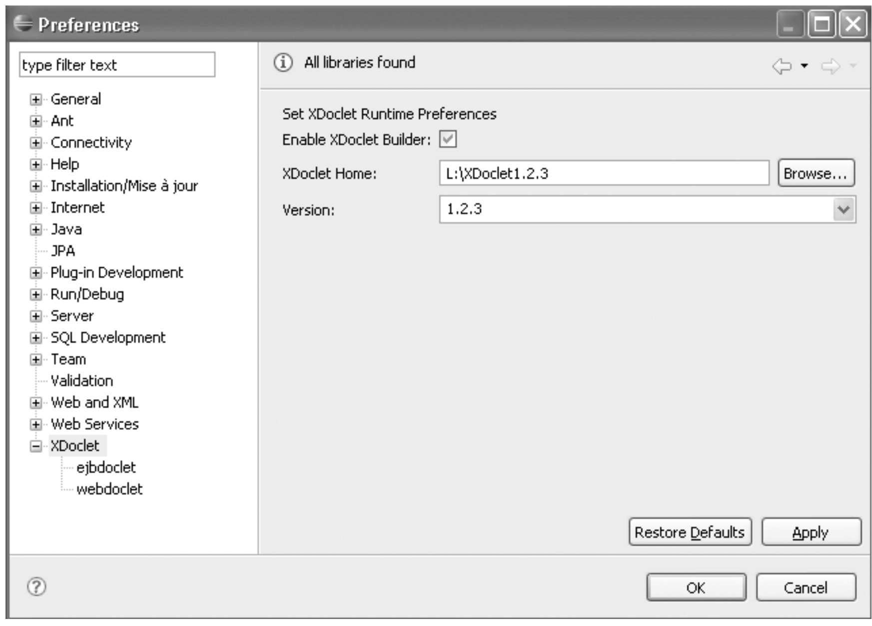


Figure 5.21

Configuration de XDoclet

12. Cliquez sur OK, puis régénérez votre projet en sélectionnant Projet et Nettoyer.

Vous devez retrouver votre espace de travail exempt de toute erreur de compilation.

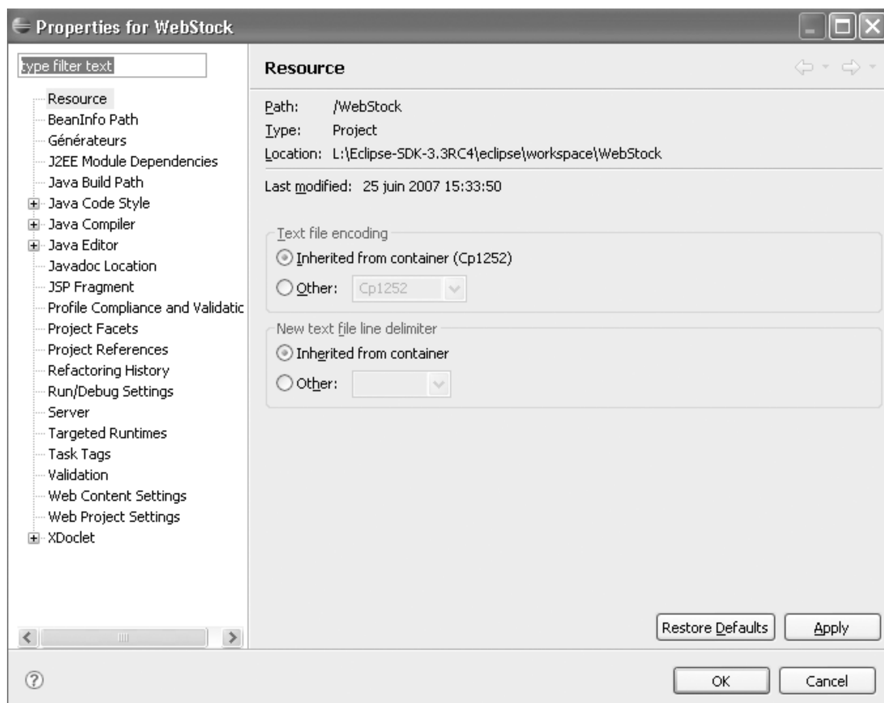
Propriétés du projet

Les projets J2EE sont créés avec des propriétés par défaut par l'assistant de création de projet fourni par JST. Ces propriétés sont généralement suffisantes pour la majorité des projets développés.

Il arrive toutefois que ces propriétés ne répondent pas au besoin des applications développées. JST fournit la possibilité de modifier les réglages du projet par le biais de l'option Properties du menu contextuel du projet en cours de développement (voir figure 5.22).

Figure 5.22

Propriétés du projet J2EE



Le tableau 5.4 récapitule les propriétés de configuration du projet J2EE.

Tableau 5.4 Propriétés de configuration du projet J2EE

Propriété	Description
Resource	Fournit un certain nombre d'informations sur le projet en cours, comme sa localisation sur le système de fichiers, la date de dernière modification, le type d'encodage, etc.
BeanInfo Path	Permet l'activation de l'introspection BeanInfo sur le projet.
Générateurs	Configure les générateurs du projet et l'ordre de leur invocation.
J2EE Module Dependencies	Module de gestion des dépendances (bibliothèque jar externe ou projet utilitaire en particulier)
Java Build Path	Indique la location de fichiers additionnels de projet ou de bibliothèques à inclure lors de la construction du projet. Gère également l'ordre de référencement de ces artefacts.

Tableau 5.4 Propriétés de configuration du projet J2EE *(suite)*

Propriété	Description
Java Code Style	Spécifie le style du code utilisé pour la génération de code pour appliquer des conventions de nommage, des règles de style et de commentaire.
Java Compiler	Spécifie les réglages pour le compilateur associé au projet Java.
Java Editor	Spécifie les actions à effectuer lors de la sauvegarde du code source.
Javadoc Location	Spécifie la localisation de la documentation au format javadoc du projet.
JSP Fragment	Définit l'encodage par défaut, la directive page et la valeur du type de contenu pour le fragment JSP (projet dynamique uniquement).
Profile Compliance and Validation	Spécifie le niveau d'interopérabilité WS-I et WS-I SSBP (Attachments Profile and Simple SOAP Binding Profile).
Project Facets	Permet de modifier les caractéristiques du projet.
Project References	Spécifie les projets à référencer.
Refactoring History	Historique de refactoring
Run/Debug Settings	Gère la configuration de lancement associée (Java Applet/Java application).
Server	Utilise le serveur spécifié lors du démarrage du projet.
Targeted Runtimes	Configuration serveur cible
Task Tags	Spécifie les balises qui seront utilisées pour indiquer les tâches.
Validation	Affiche les validateurs qui vont s'exécuter durant la validation du projet.
Web Content Settings	Propriétés par défaut du projet appliquées en particulier au type de document (xHTML 1.0 Strict/Frameset/MP 1.0/Basic 1.0)
Web Project Settings	Spécifie le contexte root du projet Web.
Xdoclet	Configure l'outil Xdoclet.

Structure et contenu du projet

Comme indiqué précédemment, un projet Eclipse est représenté dans le système de fichiers sous une forme arborescente, cette structure contenant l'ensemble des artefacts du projet.

Eclipse crée un certain nombre de fichiers au format XML au sein de cette arborescence afin de gérer l'information du projet.

Deux fichiers et un répertoire importants sont créés systématiquement lors de la création du projet J2EE. Ils peuvent être trouvés dans le système de fichiers à la racine du répertoire du projet J2EE :

- project : contient des informations concernant la nature, le générateur (builder) et des propriétés additionnelles définies dans le projet.
- classpath : contient le chemin d'accès aux différents jar et répertoires qui seront utilisés lors de la compilation du projet.
- setting : répertoire contenant les préférences du projet et sa configuration (facets utilisées, infos de localisation, etc.).

> **Édition**
> Si vous souhaitez éditer ces fichiers, ce qui n'est pas recommandé, faites en une copie.

Artefacts du projet et vues JEE

Une fois le projet créé avec Web Tools, de nouveaux artefacts peuvent être ajoutés.

Ces artefacts comprennent les composants servlets, les pages JSP et les EJB. JST fournit une perspective JEE spécialisée pour le support des projets JEE et des différents artefacts contenus au sein du projet. Par exemple, une application Web JEE contient généralement plusieurs fichiers qui nécessitent d'être modifiés, un serveur sur lequel les artefacts vont être déployés et une vue console pour afficher la sortie du serveur.

La perspective Java EE fournit un accès direct à un certain nombre de vues utiles, dont le tableau 5.4 récapitule les principales :

Tableau 5.4 Vues associées à la perspective JEE

Vue	Description
Project Explorer (Explorateur de projets)	Fournit une vue de type navigation pour parcourir les différents artefacts du projet JEE, chaque composant type étant représenté par une icône particulière (JSP, bibliothèque, etc.).
Structure	Affiche la structure des éléments contenus dans le document en cours d'édition en utilisant une vue arborescente.
Problems (Problèmes)	Liste les problèmes existants dans le projet en cours de modification (problèmes de compilation, etc.).
Tasks (Tâches)	Fournit une liste des éléments assignés sous forme de tâches ou de « pense bête » pour le développeur (ajout de commentaire, etc.).
Propriétés	Permet d'inspecter et modifier les propriétés de l'élément courant sélectionné.
Servers	Permet de contrôler les opérations de l'instance serveur configurée (arrêt/relance, etc.).

Des vues supplémentaires peuvent être ouvertes en sélectionnant les options Window, Show View et Other d'Eclipse puis en sélectionnant la vue requise. La nouvelle vue s'ouvre à côté des vues existantes de la perspective JEE et les complémente de manière personnalisée.

Il est ensuite possible de sauvegarder ces vues personnalisées par le biais Window et Save Perspective As pour les réutiliser ensuite.

La synchronisation s'opère immédiatement entre chaque vue JEE et la source du document dans l'éditeur.

En résumé

Ce chapitre vous a permis de vous familiariser avec l'outillage de support au développement JEE inclus dans WTP et les différentes vues et propriétés permettant de gérer ce type de projet.

Le chapitre suivant vous fournira l'occasion de faire connaissance avec un autre projet utile au développement Web et au support des données, le projet Data Tools.

6

Conception du modèle de données avec le projet DTP (Data Tools Platform)

Ce chapitre présente l'outillage du projet Eclipse Data Tools Platform, centré sur la manipulation des structures de données, qui facilite la définition du modèle physique de la base de données.

DTP est un nouveau projet principal de la communauté Eclipse. Originellement proposé par Sybase en février 2005, il a entraîné dans son sillage une large et solide communauté et est actuellement géré par un comité comprenant Sybase, IBM et Actuate.

DTP est un framework orienté utilisateur centré sur les problématiques d'accès aux données (data centric), devenues incontournables dans le développement d'applications.

Le projet offre les avantages suivants :

- centralisation de la définition des connecteurs aux principales sources de données en un seul endroit, au niveau de l'instance de la plate-forme Eclipse ;
- centralisation de la définition des connexions et de leur création ;
- possibilité pour les outils d'accéder à un ensemble abstrait de connexions en masquant les détails de leur implémentation ;
- masquage de l'hétérogénéité des connexions et des outils pour chaque source de données ;
- possibilité de partager les définitions de connexions, facilitant l'intégration de la solution au niveau de la plate-forme Eclipse et l'utilisation.

Architecture de DTP

Fondé sur le framework EMF, DTP consiste en un modèle de base, des connecteurs et des outils SQL pour l'interrogation et la manipulation des données. Il propose une architecture de conception orientée modèle (Model Driven Design) qui supporte les principales bases de données relationnelles du marché.

Les sections qui suivent détaillent les trois sous-projets clés qui constituent l'ossature du projet DTP. La version que vous utiliserez est la celle livrée en bundle avec Europa (DTP 1.5).

Sous-projets de la plate-forme DTP

DTP *(http://www.eclipse.org/datatools/)* est le projet central, autour duquel gravite les sous-projets Model Base, CMF (Connection Management Framework) et SQL Development Tools.

Model Base

Ce projet fournit les fondations pour le développement de la couche d'accès aux données. Il utilise une approche orientée modèle avec UML, fondée sur le framework EMF (Eclipse Modeling Framework).

Il inclut les composants suivants :

- modèle de définition de donnée ;
- éditeur de modèle ;
- modèle SQL ;
- modèle d'interrogation SQL (spécifications SQL 99/03) ;
- modèle de requête SQL/XML.

Les principales caractéristiques de ce sous-projet sont les suivantes :

- gestionnaire de version, les modèles pouvant être aisément versionnés à travers le composant de gestion de versions intégré à Eclipse ;
- support d'un éditeur de modèle intégré visuel avec EMF ;
- support de javadoc, les modèles pouvant être publiés et documentés ;
- extensibilité, les modèles étant extensibles et n'étant pas dépendants d'un système de gestion de données propriétaire ;
- standardisation, les modèles étant compatibles avec les standards existants, comme SQL et JDBC.

CMF (Connectivity Management Framework)

Le projet CMF fournit les ressources pour la gestion des connexions aux diverses sources de données JDBC ainsi que l'outillage de support approprié pour l'utilisation des connexions.

Le framework pour la définition ou la création de drivers est fondé sur un certain nombre de modèle proposés par défaut (d'autres modèles peuvent être créés par le biais de points d'extension DMF).

Une fonctionnalité clé du sous-projet connecteur est la possibilité offerte aux utilisateurs de saisir une fois leur information et de la réutiliser à travers l'ensemble des outils d'Eclipse. Conçu sur le modèle de DMF, le framework CMF définit des connexions spécifiques à travers la configuration des données et fournit des instances de connexions.

La figure 6.1 illustre les définitions de drivers accessibles par le biais du menu Préférences d'Eclipse.

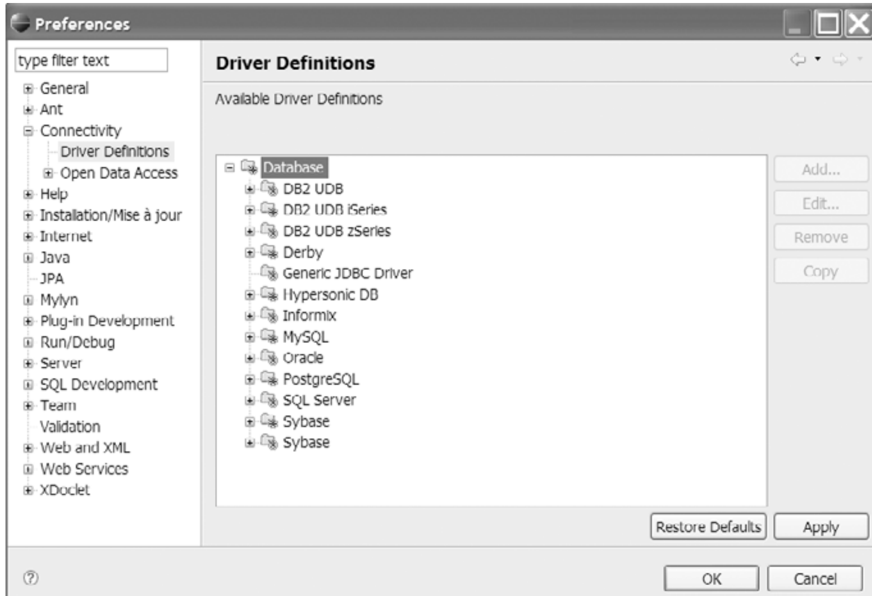


Figure 6.1

Définitions de drivers sous Eclipse

Le projet CMF propose également dans la vue Eclipse un explorateur de sources de données sous la forme d'instances de profils de connexion, ou CP (Connection Profile), associé à un framework d'accès aux données, ou ODA (Open Data Access). Cela permet aux applications d'accéder aux données à partir de sources de données standards et personnalisées.

ODA permet la connectivité des données entre les consommateurs et fournisseurs de données à travers des interfaces d'exécution en fournissant une abstraction à travers le concept de donnée, de la même manière que CMF fournit une abstraction à travers la notion de connexion à ces mêmes données.

SQL Development Tools

Ce projet fournit les frameworks et outils permettant de travailler avec SQL et d'étendre les fonctions des outils d'interrogation. L'éditeur de routines et le débogueur de routines (Routine Editor et Routine Debugger) fournissent un moyen d'interagir avec les procédures stockées et les fonctions.

Le parseur de requêtes SQL permet aux outils SQL d'utiliser un arbre d'analyse associé à la requête SQL. De son côté, le framework SQL Execution Plan permet une analyse fine et le tuning des requêtes SQL. Il est également possible d'exécuter des procédures stockées.

Mise en œuvre de DTP

Après cette rapide présentation du projet DTP, vous allez mettre en œuvre l'outillage du projet DTP à travers l'étude de cas webstock.

Le script de création du modèle webstock figure sur la page Web dédiée à l'ouvrage pour les cibles Apache Derby et Hypersonic, SGBD que nous supposons installé et configuré (voir en annexe pour les détails d'installation et de configuration). L'outillage DTP (version DTP 1.5), qui n'est pas livré en standard dans la plate-forme Europa, doit être correctement installé et mis à jour par le biais du gestionnaire de mises à jour Eclipse.

La figure 6.2 illustre l'état de la configuration de l'installation après mise à jour de l'outillage DTP.

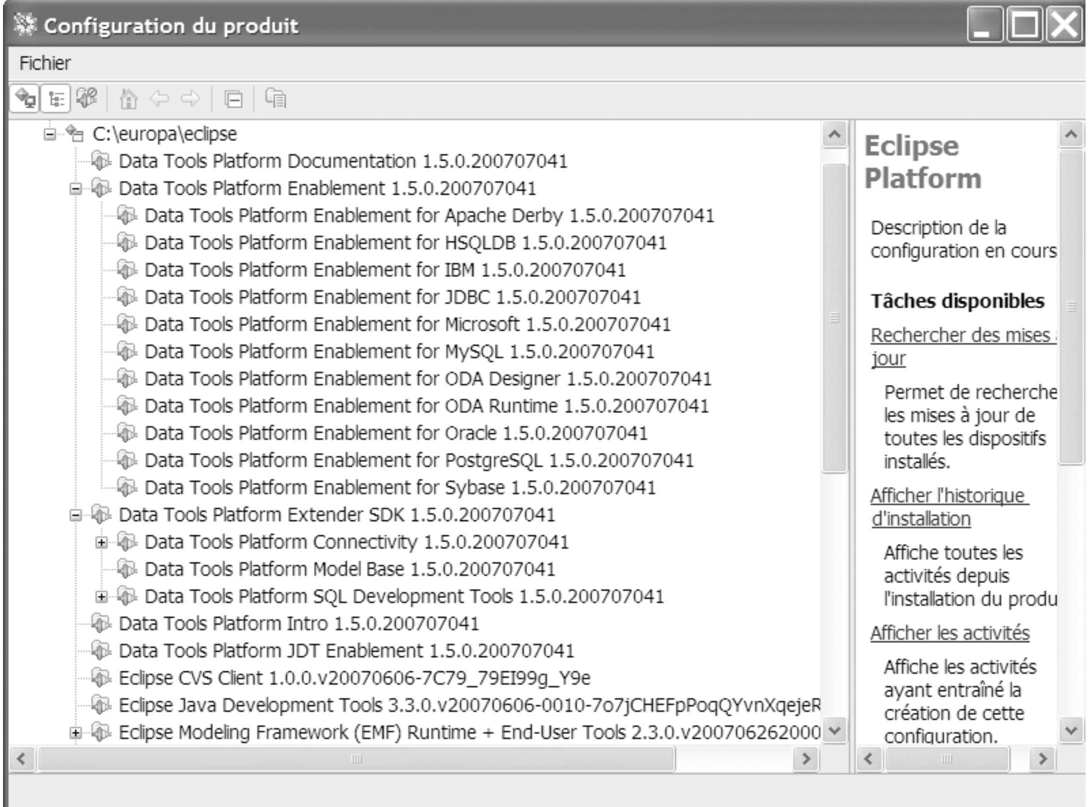


Figure 6.2

Configuration de l'installation après mise à jour de l'outillage DTP

Connexion à la base Apache Derby

La première chose à effectuer pour la mise en route de DTP est la configuration de la connexion à la base cible. Nous nous fondons ici sur la base Derby, une base de données relationnelle Java qui offre la particularité de pouvoir être embarquée dans des applications Java.

Derby est le nom de l'implémentation Open Source du SGBD dérivée d'IBM Cloudscape, la version commerciale de la base d'IBM.

Vous utiliserez essentiellement cette base pour vos tests et votre développement, étant entendu que, pour des déploiements finaux en production, il est recommandé d'utiliser des bases telles que MySQL 5 ou équivalentes.

L'installation et la prise en main de Derby ne présentent pas de réelles difficultés non plus que sa configuration. Nous supposons que le répertoire d'installation de la base Derby (désigné par la variable d'environnement DERBY_HOME) se trouve sous C:\Tools\db-derby-10.2.2.0-bin.

1. Positionnez dans la variable classpath de votre installation les bibliothèques derby.jar et derbytools.jar sous le répertoire lib de DERBY_HOME.
2. Lancez la commande ij pour créer la base de test webstockdb (voir figure 6.3).

```
C:\Tools\db-derby-10.2.2.0-bin>java -Dij.protocol=jdbc:derby: org.apache.derby.t
ools.ij
version ij 10.2
ij> connect 'webstockDB;create=true';
ij>
```

Figure 6.3

Commande interactive ij sous DOS

3. La commande ij étant une application Java, vous devez lancer une JVM et indiquer le nom complet de la classe principale que vous souhaitez exécuter (ici l'outil ij). En cas d'exception, vérifiez que le fichier jar derbytools.jar est présent. La commande connect permet la connexion à la base de données, en l'occurrence ici la chaine, ou URL, de connexion *jdbc:derby:webstockDB;create=true.* La propriété create=true, passée au driver JDBC Derby, spécifie la création de la base si celle-ci n'existe pas.
4. Quittez la session de connexion à la base Derby en entrant exit.
5. Lancez la commande dir sous Windows ou ls sous UNIX. Vous devez constater qu'un nouveau répertoire webstockdb a été créé, ainsi que les sous-répertoires webstockdb/log et webstockdb/seg0, qui contiennent les données de la base webstockdb.

Copie de la base de données

Pour créer une copie de votre base de données Derby, vous devez simplement stopper l'instance Derby, effectuer une copie du répertoire et des sous-répertoires (ici webstockdb) et relancer le serveur. La restauration de la base est aussi simple : stoppez le serveur, puis remplacez la structure de la base avec la copie restaurée, et relancez le serveur.

6. Revenez sur l'IDE Eclipse, et cliquez sur le bouton Add après avoir sélectionné dans le menu Preferences d'Eclipse la définition du driver Derby 10.2.
7. Sélectionnez l'option Derby Embedded Driver (voir figure 6.4).

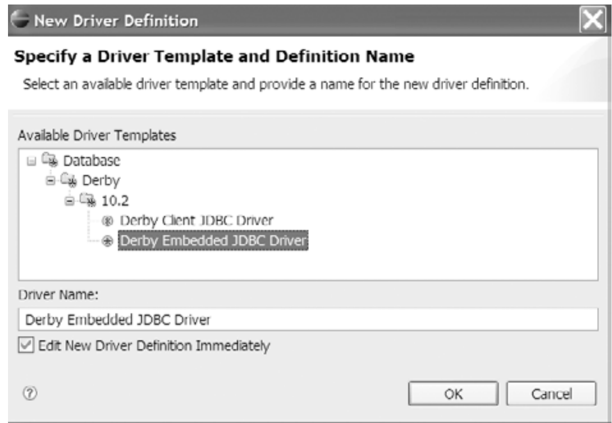


Figure 6.4

Définition du driver Derby avec DTP

8. Supprimez le fichier driver derby.jar, et ajoutez la définition des drivers suivants à partir de l'emplacement DERBY_HOME\lib\ (voir figure 6.5) :
   – Connection URL
   – Database Name : webstockdb
   – Password : admin
   – User Id : derby

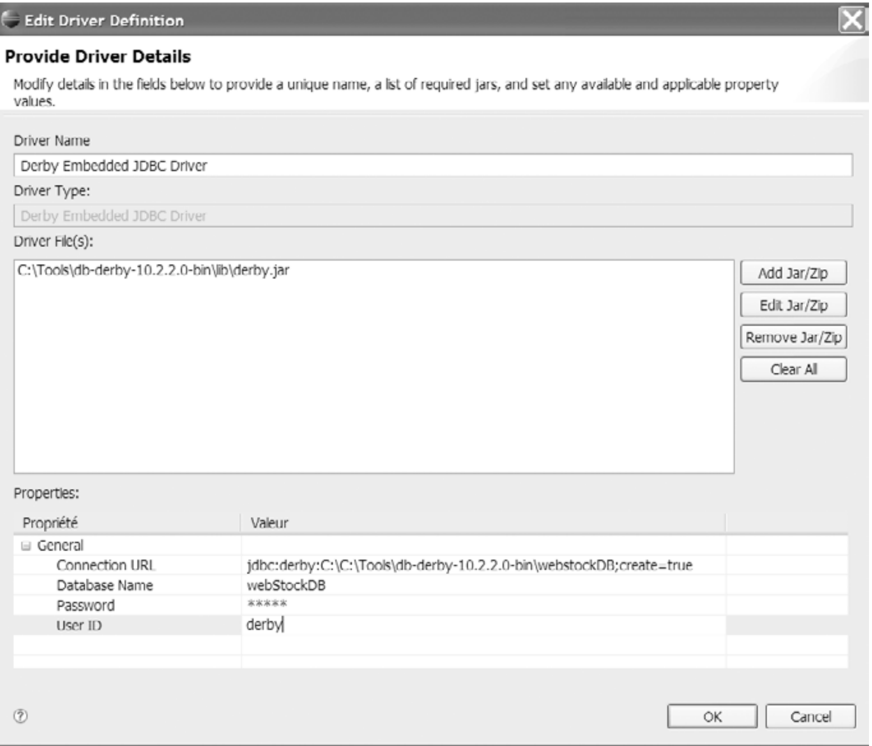


Figure 6.5

Configuration du driver Derby avec DTP

9. Cliquez sur OK et fermez la boîte de dialogue Preferences d'Eclipse.

Création d'un profil de connexion

Une fois effectuée la définition du driver Apache Derby, vous pouvez utiliser cette définition comme base pour la création d'un profil de connexion.

1. Ouvrez la perspective Database Development (*via* Window, Open Perspective et Other). Le bureau Eclipse doit ressembler à celui illustré à la figure 6.6.

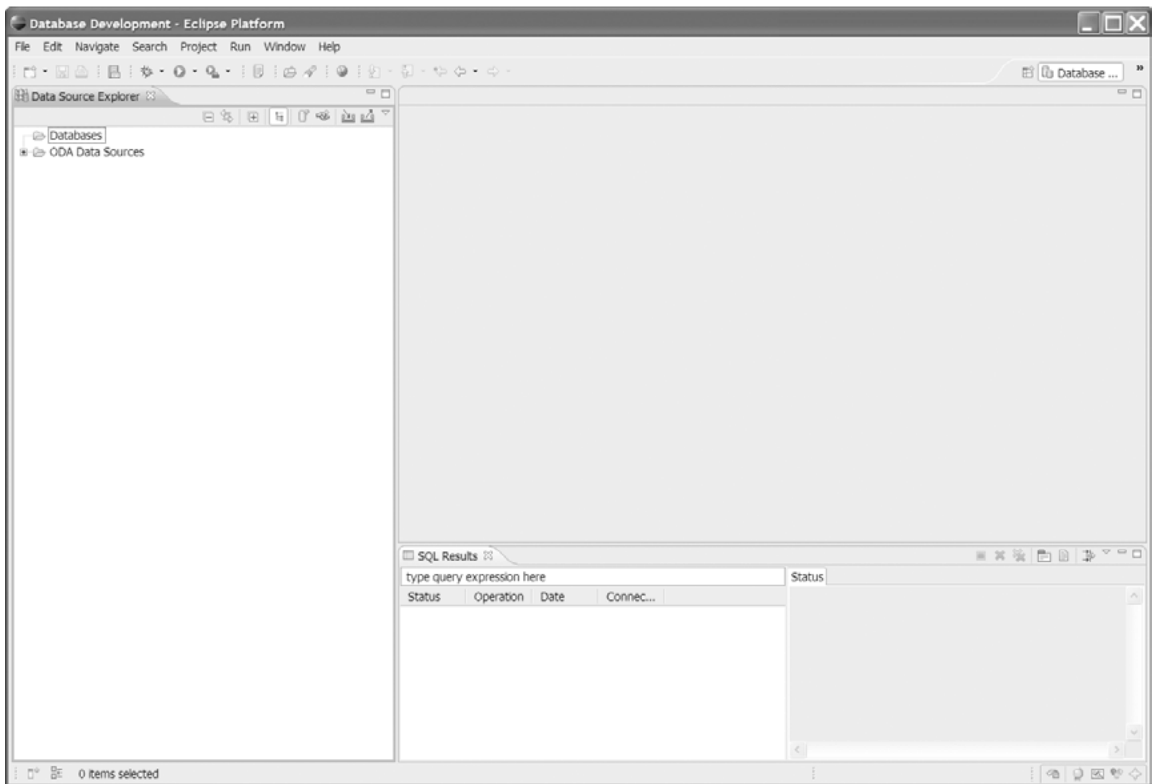


Figure 6.6

Création d'un profil de connexion avec DTP (1/3)

2. L'explorateur de sources de données Eclipse est accessible en sélectionnant Databases puis New.

Figure 6.7

Création d'un profil de connexion avec DTP (2/3)

3. Sélectionnez Derby Embedded Database, puis cliquez sur Next.
4. Donnez un nom au profil de connexion (par exemple MyDerby) et optionnellement une description, puis cliquez sur Next (voir figure 6.8).

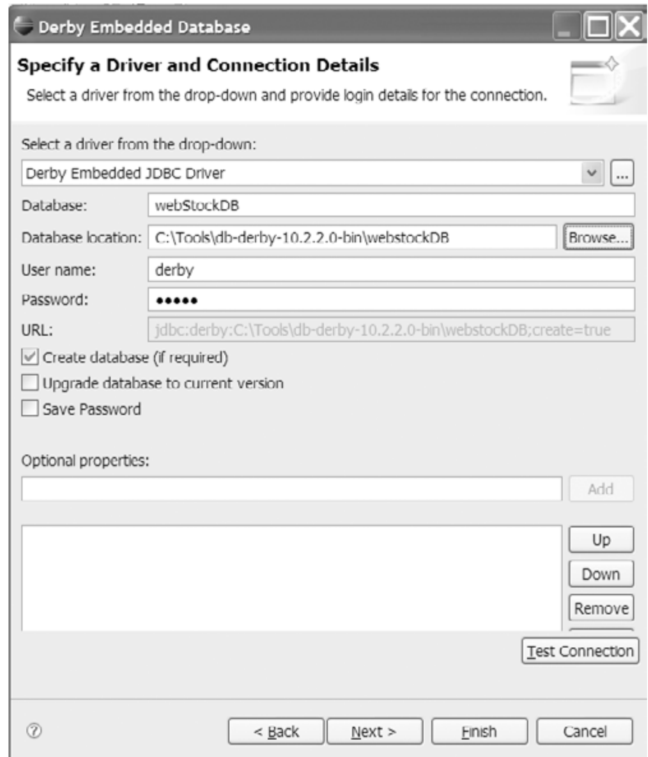


Figure 6.8

Création d'un profil de connexion avec DTP (3/3)

5. Cliquez sur Finish pour terminer la création du profil de connexion.

   Vous êtes prêt à vous connecter à l'instance Apache Derby.

6. Développez le nœud Databases dans la vue Data Source Explorer, puis sélectionnez le profil de connexion que vous venez de créer par clic droit, et sélectionnez Connect.

Après quelques secondes, vous devriez voir apparaître le profil de connexion développé avec le contenu de la base Derby webstockdb vide (voir figure 6.9).

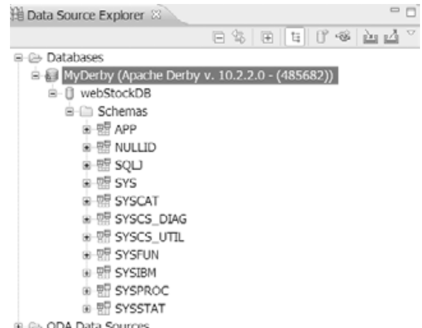


Figure 6.9

La vue explorateur de source de données DTP

Script SQL de création du schéma de base de données

À ce stade, vous allez créer un script SQL et l'exécuter sur l'instance webstockdb que vous venez de créer. Pour ce faire, vous allez créer un projet pour stocker le fichier SQL et l'associer au profil SQL.

1. Sélectionnez File, New, Project et Java Project à partir de la liste des projets disponibles, puis cliquez sur Next.
2. Saisissez webstockdata comme nom de projet, et cliquez sur Finish. Votre projet doit apparaître dans la vue Navigateur de l'IDE Eclipse.
3. Créez un dossier (*via* les options du menu contextuel Eclipse New et Folder) SQL pour contenir le fichier de script SQL webstockddl.sql.
4. Faites un clic droit sur le projet créé, et sélectionnez New, Other, SQL Development et SQL File, puis cliquez sur Next.
5. Remplissez les champs suivants (voir figure 6.10) :
   – Nom du fichier SQL : webStockDDL
   – database server type : Derby_10.x
   – Connection profile name : WebStockDB
   – Database name : webStockDB

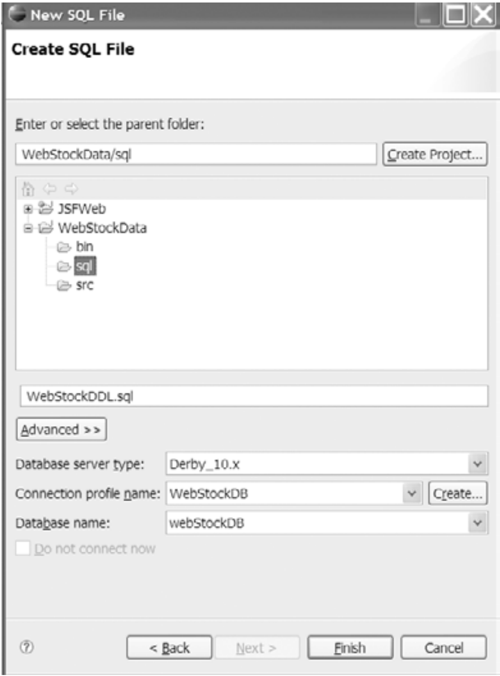


Figure 6.10

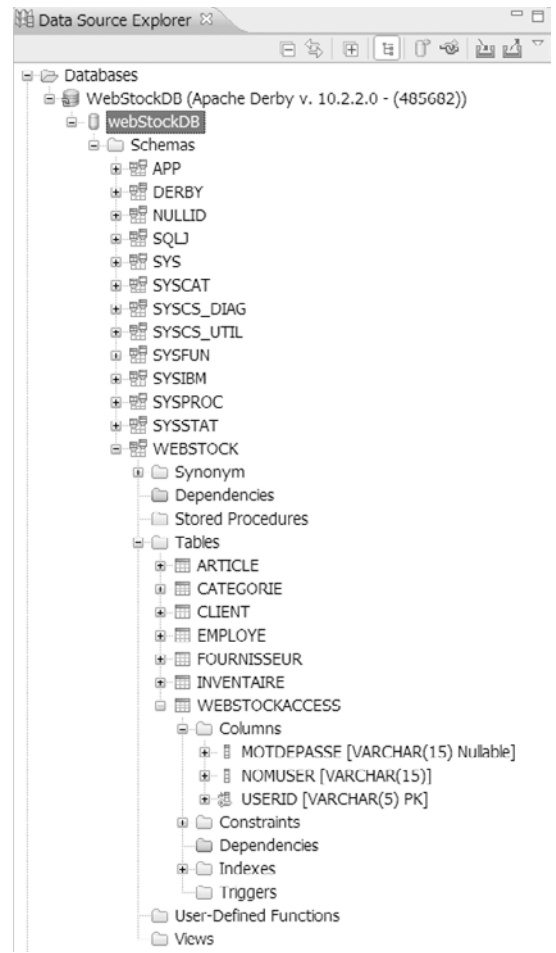
Création du fichier SQL avec l'outillage DTP

6. Cliquez sur Finish.
7. Faites un copier-coller du contenu du fichier script de création du schéma de la base webStock dans le fichier webStockDDL.sql *via* l'éditeur de script associé.

8. Cliquez sur Execute All pour lancer l'exécution du script. Si tout se passe bien, vous devez voir s'afficher dans l'explorateur de source de données les tables illustrées à la figure 6.11.

Figure 6.11

Vue explorateur après création du schéma de la base webstock



ARTICLE

ARTICLEID [VARCHAR(5)]	NOMARTICLE [VARCHAR(30)]	ARTICLECATEGORIEID [VARCHAR(5)]	FOURNISSEURID [VARCHAR(5)]	DESCRIPTION [VARCHAR(50)]	POIDS [DECIMAL(10 , 2)]	IMAGE_URL [VARCHAR(50)]
AR001	ABIT AG8 (Intel i915P)	CAT03	FOU01	CARTE MERE ABIT SOCKET ...	125,00	abitag8.gif
AR002	ASUSTEK A7N8X	CAT03	FOU01	CARTE MERE ASUSTEK DE T...	110,00	asusteka7n8x.gif
AR003	HITACHI 120 GO 7200 RPM	CAT04	FOU02	DISQUE DUR HITACHI 120 G...	150,00	hitachi120RPM2004.gif
<new row>						

Figure 6.12

Vue SQL Results après exécution d'une requête SQL

> **Vue SQL Results**
>
> Utilisez la vue SQL Results de l'outillage DTP pour afficher l'état des requêtes lors de leur exécution. Cette vue (qui s'obtient *via* Window, Show View, Other puis SQL Development SQL Results du menu Eclipse) affiche le résultat de l'instruction SQL courante sous forme tabulaire (statut de la requête, opération, date, profil de connexion).

9. Refaite les même étapes avec le script de peuplement de la base webstockdata disponible sur la page Web associée à l'ouvrage.
10. Vous pouvez afficher le contenu et extraire les données provenant des tables de la base par le biais de la vue Data Source Explorer.
11. Faites un clic droit, et sélectionnez les options contextuelles Data et Sample Contents pour l'affichage ou Data et Extract pour une extraction des données Vous pouvez également éditer les données affichés par le biais des options Data et Edit du même menu contextuel.

La figure 6.13 illustre l'affichage du contenu de la table `Article`.

ARTICLE

ARTICLEID [VARCHAR(5)]	NOMARTICLE [VARCHAR(30)]	ARTICLECATEGORIEID [VARCHAR(5)]	FOURNISSEURID [VARCHAR(5)]	DESCRIPTION [VARCHAR(50)]	POIDS [DECIMAL(10 , 2)]	IMAGE_URL [VARCHAR(50)]
AR001	ABIT AG8 (Intel i915P)	CAT03	FOU01	CARTE MERE ABIT SOCKET ...	125,00	abitag8.gif
AR002	ASUSTEK A7N8X	CAT03	FOU01	CARTE MERE ASUSTEK DE T...	110,00	asusteka7n8x.gif
AR003	HITACHI 120 GO 7200 RPM	CAT04	FOU02	DISQUE DUR HITACHI 120 G...	150,00	hitachi120RPM2004.gif
<new row>						

Figure 6.13

Affichage du contenu de la table Article

Pour insérer une ligne dans la table, il suffit de sauvegarder l'éditeur en sélectionnant File puis Save (ou Ctrl+S). La vue SQL Results affiche le résultat de l'insertion.

Génération du DDL

Vous pouvez générer le DDL (Data Definition Language) à partir du modèle déjà généré en sélectionnant le dossier `Tables` du schéma webstock dans la vue explorateur de données et en cliquant sur Generate DDL dans le menu contextuel (voir figure 6.14).

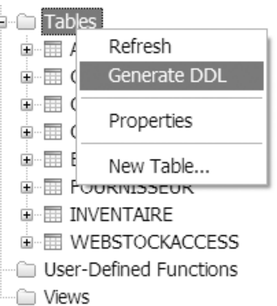


Figure 6.14

Génération du script DDL à partir du schéma de la base

En résumé

Vous en avez terminé avec les étapes de création de la base webstock qui sert de cadre à l'ouvrage.

Vous pouvez à présent vous concentrer sur le développement de la partie Web de l'étude de cas en utilisant les assistants et outils du projet Web Tools.

7

Développement Web avec l'outillage Web Tools et les patterns

Ce chapitre présente l'outillage Web Tools pour le développement Web sans l'aide d'un framework particulier, en utilisant les bonnes pratiques de conception.

Nous commencerons par décrire l'outillage du projet Web Tools (WTP) et rappellerons les principes de base de l'architecture d'une application Web et de ses composants (IHM et couche d'accès aux données) ainsi que les étapes qui vont de son développement jusqu'à son déploiement.

Nous terminerons par le design et la mise en œuvre d'une portion d'application Web selon une approche fondée sur les modèles de conception ou pattern. Ce design et cette mise en œuvre s'effectueront en logique 3-tiers traditionnelle, c'est-à-dire sans l'utilisation d'un framework particulier et utilisant des composants de type servlet et JSP, ainsi que les bibliothèques de balises JSTL (JavaServer Pages Standard Tag Library).

Enfin, nous ferons un tour d'horizon des fonctionnalités offertes par Web Tools pour simplifier le développement Web. Nous supposons acquises les bases du développement par servlets/JSP et JSTL, ainsi que les notions propres aux API JDBC, que nous nous contenterons de rappeler.

Principes de base de l'exécution d'une requête JDBC à partir d'un programme Java

Avant d'entrer dans le design et la mise en oeuvre du développement Web avec Web Tools, nous allons résumer les bases de l'interrogation SQL d'une base de données à partir d'un programme Java.

L'exécution d'une requête JDBC, implique trois concepts essentiels :

- une connexion base de données (objet Connection) ;

- une requête SQL (objet Statement) ;
- le résultat de la requête (objet ResultSet).

Pour utiliser ces objets, vous devez les importer dans votre application, comme le montre l'extrait de code suivant, dont le code complet est disponible sur la page Web dédiée à l'ouvrage :

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.SQLWarning;
import java.sql.Statement;
import java.sql.ResultSet;

public class FirstQuery {
    private static final String driver = "org.apache.derby.jdbc.EmbeddedDriver" ;
    private static final String url = "jdbc:derby:WebstockDB" ;
    private static final String qry =
        "SELECT articleId, nomArticle, articleCategorieId, fournisseurId,
        ➥description, poids FROM webstock.article" ;
```

Les classes Java et constantes nécessaires à l'interrogation de la base de données (URL d'accès à la base webstockdb et requêtes d'interrogation à la table webstock.article) sont importées explicitement. Chaque colonne est invoquée pour éviter les effets de bord en cas de modification du schéma de base de données sous-jacent.

Les opérations requises pour exécuter une requête à la base à partir d'un programme Java sont relativement simples, comme l'illustre l'extrait de code suivant, qui présente la méthode doQuery avec inclusion de la clause SQLException, qui permet de remonter l'exception vers le code appelant :

```
static void doQuery(Connection con) throws SQLException {

    SQLWarning swarn = null ;

    Statement stmt = con.createStatement() ;
    ResultSet rs = stmt.executeQuery(qry) ;

    while (rs.next()) {

        System.out.println("Numéro Article: " + rs.getString("articleId")) ;
        System.out.println("Nom Article:  " + rs.getString("nomArticle")) ;
        System.out.println("Catégorie Article:  "
        ➥+ rs.getString("articleCategorieId")) ;
System.out.println("Fournisseur Article:  " + rs.getString("fournisseurId")) ;
        System.out.println("Description: " + rs.getString("description") + '\n') ;
        System.out.println("Poids Article: " + rs.getString("poidsArticle") + '\n')


swarn = rs.getWarnings() ;

    if(swarn != null){
            printSQLWarning(swarn) ;
        }
    }
    rs.close() ;
    stmt.close() ;
}
```

Dans la méthode doQuery, vous créez d’abord un nouvel objet JDBC Statement en utilisant la méthode createStatement sur l’objet Connection. Vous utilisez ensuite la méthode executeQuery sur l’objet Statement créé pour envoyer la chaîne de requête à la base Apache Derby où celle-ci est exécutée. Vous accédez ensuite aux résultats de la requête dans le programme Java en utilisant l’implémentation resultset fournie par le package du driver JDBC embarqué dans Derby et en itérant ensuite sur cet objet (rs.next()).

Au sein de cette boucle, vous accédez aux six colonnes de l’enregistrement en utilisant la méthode getString appliquée à l’objet resultset. La méthode getString peut accéder aux colonnes de la table de deux façons : en utilisant le numéro de colonne original dans la requête, par exemple `getString(1)`, ou par `getString("articleId")`. Nous préconisons l’usage explicite des noms de colonnes pour éviter tout risque d’ambiguïté.

La méthode doQuery vérifie explicitement d’éventuels avertissements sur l’objet resultset après accès à chaque nouvel enregistrement effaçant le précédent éventuellement généré. Pour invoquer la méthode doQuery, il est nécessaire d’établir au préalable une connexion à la base de données et d’appeler celle-ci au sein d’un bloc `try…catch`, comme dans l’extrait de code suivant :

```
public static void main(String[] args) {
    Connection con = null ;

    try {
        Class.forName(driver) ;
        con = DriverManager.getConnection(url);

        SQLWarning swarn = con.getWarnings() ;

        if(swarn != null){
            printSQLWarning(swarn) ;
        }

        doQuery(con) ;

    } catch (SQLException se) {
        printSQLException(se) ;
    }
```

Design de l’application Web avec Web Tools

Reprenez le projet webstock ainsi que la base webstock sous Derby, que vous avez commencé à construire au cours des chapitres précédents.

Vous allez développer le cas d’utilisation relatif à la connexion et à l’identification de l’utilisateur en utilisant l’outillage Web Tools. Pour ce faire, vous développerez les ressources suivantes :

- construction d’une page de login d’accès à l’application Web Login.jsp ;
- construction d’une servlet de traitement LoginServlet chargée de vérifier les informations de connexion à la base webstock ;
- connexion à la base webstock *via* le driver JDBC Derby et accès à la table WebStock-Access ;

- en cas de succès lors de l'authentification, transfert du contrôle à une autre servlet de traitement, LoginSuccess, *via* l'objet RequestDispatcher ;
- déploiement/test et débogage sous Web Tools de l'application Web déployée.

Ces ressources seront ensuite déployées sur un serveur JBoss 4.0 ou JBoss 4.2, dont la configuration avec WTP a été abordée au chapitre 5.

Création de la page d'accueil

Vous allez construire la page d'accueil standard index.html de l'application Web. Cela vous permettra de faire connaissance avec l'éditeur HTML intégré de Web Tools et d'avoir un aperçu de ses possibilités :

1. Sélectionnez le projet webstock créé au cours des chapitres précédents puis le dossier WebContent.
2. Dans l'assistant de création de page HTML, cliquez sur New puis HTML.
3. Dans le champ File Name, saisissez index, puis cliquez sur Next.
4. Sélectionnez le template proposé par défaut (4.0.1 Transitional), puis cliquez sur Finish.

   Vous devez obtenir le résultat illustré à la figure 7.1 (notez la présence de l'éditeur à syntaxe coloré pour le contenu HTML et du volet Structure pour la navigation au sein des balises HTML).

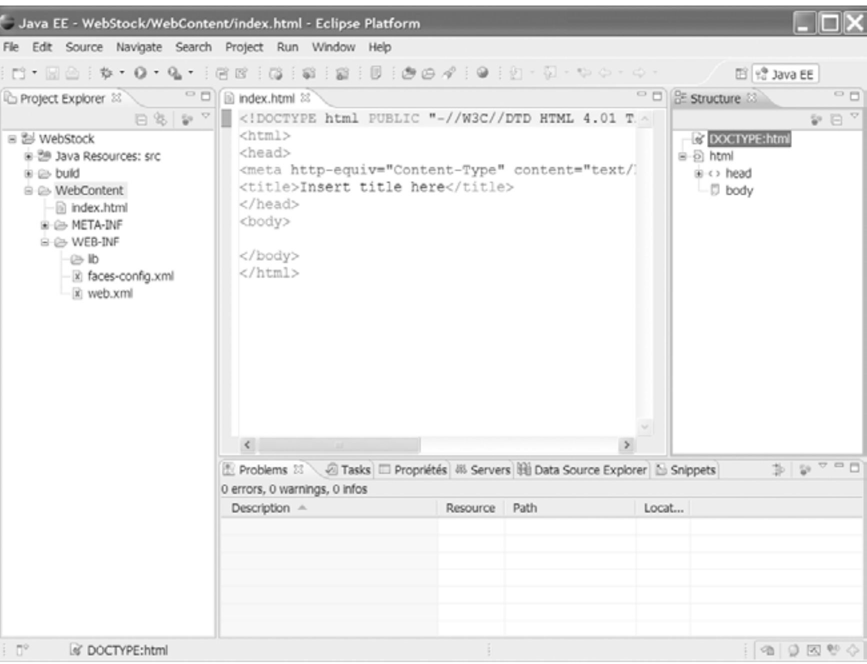


Figure 7.1

Éditeur HTML de Web Tools

5. Remplacez le contenu situé entre les balises <body> et </body> par le contenu suivant :

```
<body>
<p align="center"> </p>
```

```
<p align="center"><strong><font size="6"
    style="BACKGROUND-COLOR: #999999"><IMG alt="" src="WebStore.gif"
    border="0">  WebStock</font></strong></p>

<p align="center"> </p>

<p align="center"> </p>

<p align="center"><a title="Login"
    href="http://localhost:8080/LoginServlet/Login.jsp"><font
    style="BACKGROUND-COLOR: #ffffff"><strong><font size="4">
Bienvenue dans le</font></strong> <strong><font size="4">Système</font>
➥</strong><strong><font
    size="4">WebStore</font></strong></font></a></p>

<p align="center"> </p>

<p align="center"> </p>
<hr>
<br>
<br>

<p align="center"><STRONG>(c) Eyrolles <font
    style="BACKGROUND-COLOR: #ffffff">2005</font></STRONG></p>

<p><br>
<br>
</p>
</body>
```

6. Copiez la figure WebStore.gif dans le répertoire WebContent.
7. Faites une publication du contenu des ressources vers le serveur JBoss préalablement configuré en actionnant l'onglet Servers, puis sélectionnez le serveur JBoss dans le menu contextuel (voir figure 7.2).

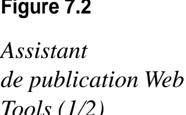
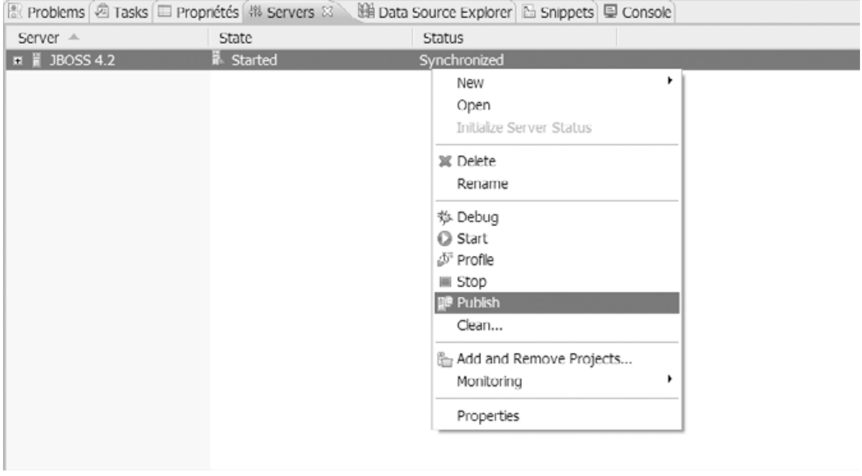


Figure 7.2

Assistant de publication Web Tools (1/2)



8. Après publication avec succès des ressources figurant dans l'onglet Console, cliquez sur l'option du menu contextuel Start pour lancer le serveur JBoss.
9. Sélectionnez la page index.html dans la vue explorateur de projet, et exécutez votre projet *via* l'option Run As, comme illustré à la figure 7.3.

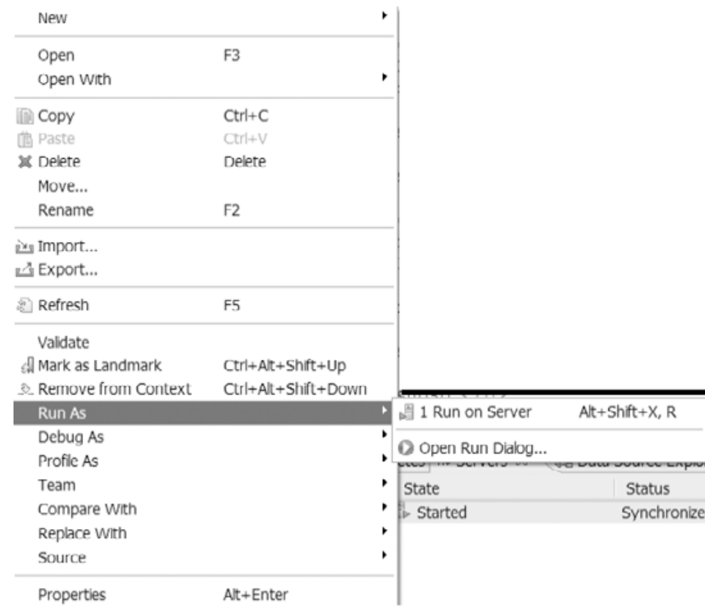


Figure 7.3

Assistant de publication Web Tools (2/2)

10. L'assistant de sélection de définition d'un serveur s'ouvre. Sélectionnez le serveur que vous avez préalablement défini (ici JBoss 4.2), puis cliquez sur Next (voir figure 7.4).

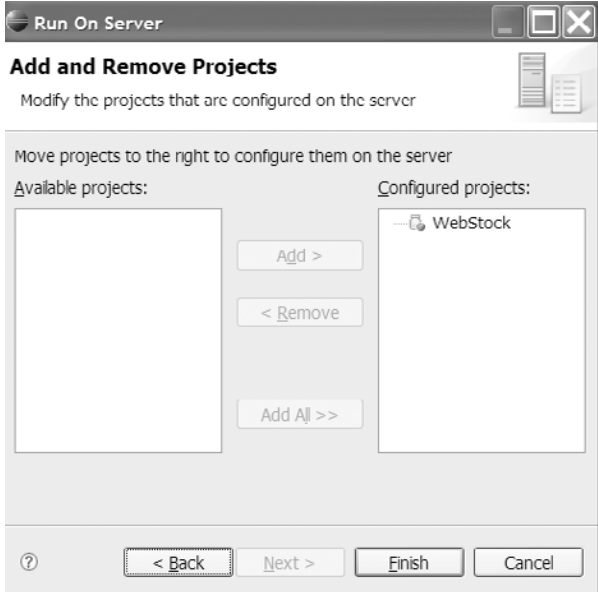


Figure 7.4

Assistant de définition de serveur

11. Votre projet en cours de définition s'affiche dans la colonne Configured projects. Cliquez sur Finish. Votre page d'accueil s'affiche comme illustré à la figure 7.5.

Figure 7.5

Exécution après déploiement de la page d'accueil de l'application webstock

Création des pages Login.jsp et LoginSuccess.jsp

Vous allez créer vos premières pages JSP en vous aidant de l'assistant de création de pages JSP de Web Tools :

1. Sélectionnez le projet webstock créé aux chapitres précédents puis le dossier WebContent destiné à contenir vos pages JSP, et sélectionnez New puis JSP.
2. Cliquez sur Next.
3. Dans le champ File name, saisissez comme nom de la page JSP Login. Cliquez sur Next.
4. Choisissez le premier modèle de template (JSP with html Markup), puis cliquez sur Finish. La page JSP est automatiquement générée dans l'éditeur Web Tools.
5. Remplacez le contenu de la page générée par le contenu suivant :

```
<%@ page import="java.util.Calendar"%>

<%
String path = request.getContextPath();
String basePath = request.getScheme()+"://"+request.getServerName()+":
➥"+request.getServerPort()+path+"/";
%>
<HTML>
<HEAD>
<TITLE>Page d'Identification</TITLE>
```

```
</HEAD>
<BODY>
<CENTER><BR>
<BR>
<H2>Login Page</H2>
<BR>
<BR>
Saisissez votre nom et votre mot de passe<BR>
<BR>
<FORM METHOD=POST ACTION="/WebStock/LoginServletTask">
<TABLE>
    <TR>
        <TD>Nom Utilisateur :</TD>
        <TD><INPUT TYPE=TEXT NAME=userName></TD>
    </TR>
    <TR>
        <TD>Mot De Passe :</TD>
        <TD><INPUT TYPE=PASSWORD NAME=password></TD>
    </TR>
    <TR>
        <TD ALIGN=RIGHT COLSPAN=2><INPUT TYPE=SUBMIT VALUE=Login></TD>
    </TR>
</TABLE>
</FORM>
</CENTER>

<% out.println("Date courante: " + Calendar.getInstance().getTime());
%>

</BODY>
</HTML>
```

6. Pour apprécier les fonctionnalités de complétion de code de Web Tools, positionnez votre curseur sur le début de la balise recherchée, puis pressez Ctrl+Espace (voir figure 7.6).

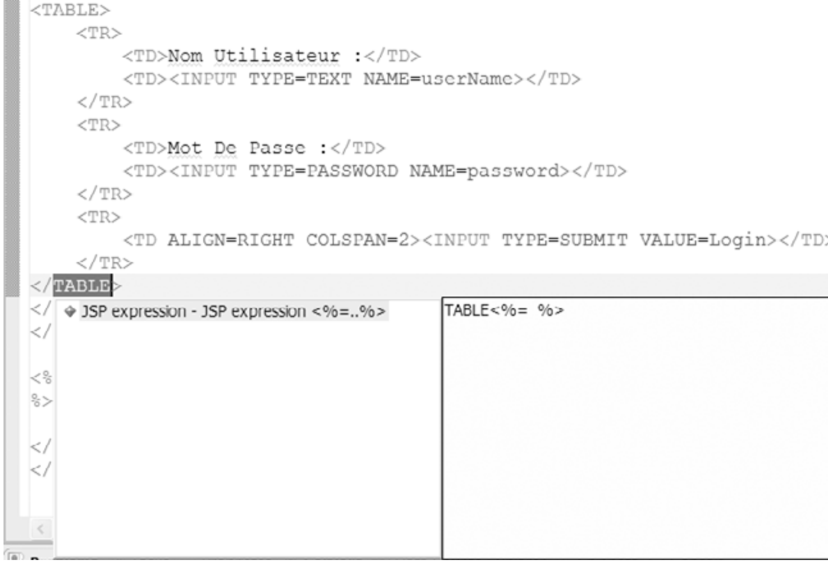


Figure 7.6

Outil de complétion de code de Web Tools

Éditeur de JSP

Vous avez la possibilité de personnaliser l'éditeur de JSP Web Tools (coloration des commentaires, des éléments de scripting, etc.) *via* le menu Preferences d'Eclipse en sélectionnant Web And XML puis le sous-menu JSP Files.

7. Sauvegardez votre page (Ctrl+S), et déployez-la sur le serveur JBoss comme lors des étapes précédentes. Votre page doit ressembler à celle illustrée à la figure 7.7.

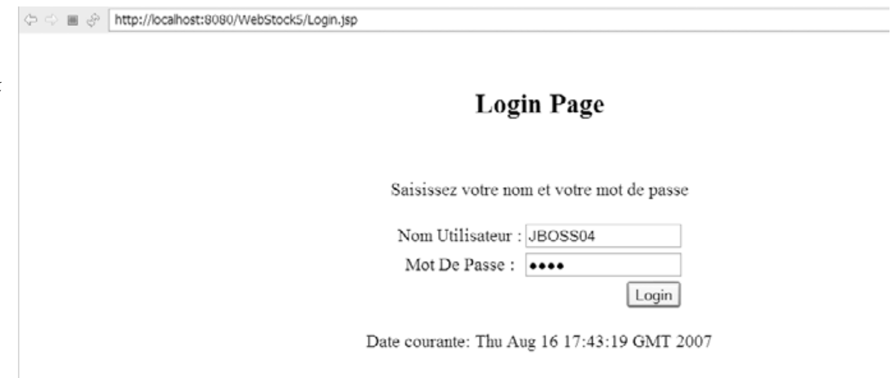


Figure 7.7

Page d'authentification de webstock

8. Sous le même dossier, créez votre seconde page JSP LoginSuccess.jsp, qui, en cas de succès de l'identification, routera l'utilisateur vers cette page :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
➥"http://www.w3.org/TR/html4/loose.dtd">
<%@ page import="java.util.Calendar"%>

<%
String path = request.getContextPath();
String basePath = request.getScheme()+"://"+request.getServerName()+":
➥"+request.getServerPort()+path+"/";
%>
<HTML>
<HEAD>
<TITLE>Page d'Identification</TITLE>
</HEAD>
<BODY>
<p>
<CENTER>

<H2>Connexion réussie ! </H2>

</CENTER>
<p align="center"><STRONG><%= request.getParameter("userName") %><font size="5"
    style="BACKGROUND-COLOR: #ffffff"></font></STRONG></p>
```

```
<p><br>
<br>

</BODY>
</HTML>
```

Le résultat de la page générée lors d'une connexion doit s'afficher comme illustré à la figure 7.8.

Figure 7.8

Page Login-Success.jsp après réussite d'une connexion

http://localhost:8080/WebStock5/LoginServletTask

Connexion réussie !

JBOSS04

Création de la servlet LoginServletTask

La ressource servlet LoginServletTask est chargée de la validation du formulaire.

Pour la créer, procédez comme suit :

1. Créez un package com.webstock.chap07 (*via* les options New et Package après sélection du dossier Java Resources).
2. Sélectionnez le package dans la vue explorateur de projets, puis cliquez à partir du menu contextuel sur New et Other, et sélectionnez l'assistant Web et Servlet. Cliquez sur Next.
3. Dans le champ Class name, entrez LoginServletTask, puis cliquez sur Finish. Le code de la servlet avec les méthodes doGet/doPost génériques est automatiquement généré.
4. Saisissez le code suivant dans cette servlet :

```
import java.sql.*;
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * @author djafaka
 *
 * TODO Pour changer le modèle de ce commentaire de type généré, allez à :
 * Fenêtre - Préférences - Java - Style de code - Modèles de code
```

```
 */
public class LoginServletTask extends HttpServlet {

/**
 * Constructor of the object.
 */
public LoginServletTask() {
    super();
}

/**
 * Destruction of the servlet. <br>
 */
public void destroy() {
    super.destroy(); // Just puts "destroy" string in log
    // Put your code here
}

/**
 * The doGet method of the servlet. <br>
 *
 * This method is called when a form has its tag value method equals to get.
 *
 * @param request the request send by the client to the server
 * @param response the response send by the server to the client
 * @throws ServletException if an error occurred
 * @throws IOException if an error occurred
 */
public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        sendLoginForm(response, false);
      }
private void sendLoginForm(HttpServletResponse response,
        boolean withErrorMessage)
        throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Login</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY>");
        out.println("<CENTER>");

        if (withErrorMessage)
          out.println("Login failed. Please try again.<BR>");

        out.println("<BR>");
        out.println("<BR><H2>Login Page</H2>");
        out.println("<BR>");
        out.println("<BR>Please enter your user name and password.");
        out.println("<BR>");
```

```
        out.println("<BR><FORM METHOD=POST>");
        out.println("<TABLE>");
        out.println("<TR>");
        out.println("<TD>User Name:</TD>");
        out.println("<TD><INPUT TYPE=TEXT NAME=userName></TD>");
        out.println("</TR>");
        out.println("<TR>");
        out.println("<TD>Password:</TD>");
        out.println("<TD><INPUT TYPE=PASSWORD NAME=password></TD>");
        out.println("</TR>");
        out.println("<TR>");
        out.println("<TD ALIGN=RIGHT COLSPAN=2>");
        out.println("<INPUT TYPE=SUBMIT VALUE=Login></TD>");
        out.println("</TR>");
        out.println("</TABLE>");
        out.println("</FORM>");
        out.println("</CENTER>");
        out.println("</BODY>");
        out.println("</HTML>");
      }


/**
 * The doPost method of the servlet. <br>
 *
 * This method is called when a form has its tag value method equals to post.
 *
 * @param request the request send by the client to the server
 * @param response the response send by the server to the client
 * @throws ServletException if an error occurred
 * @throws IOException if an error occurred
 */
 public void doPost(HttpServletRequest request,
         HttpServletResponse response)
         throws ServletException, IOException {

         String userName = request.getParameter("userName");
         String password = request.getParameter("password");
         System.out.println("doPost"+userName);
         if (login(userName, password)) {
           RequestDispatcher rd =
             request.getRequestDispatcher("LoginSuccess.jsp");
           rd.forward(request, response);
         }
         else {
           sendLoginForm(response, true);
         }
       }
 boolean login(String userName, String password) {
    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
      Connection con =
        DriverManager.getConnection("jdbc:derby ","derby","");
      System.out.println("got connection");

      Statement s = con.createStatement();
      String sql = "SELECT USERID FROM webstock.webstockaccess" +
```

```
            " WHERE NomUser='" + userName + "'" +
            " AND MotDePasse='" + password + "'";

          ResultSet rs = s.executeQuery(sql);
          if (rs.next()) {
            rs.close();
            s.close();
            con.close();
            return true;
          }
          rs.close();
          s.close();
          con.close();
        }
        catch (ClassNotFoundException e) {
          System.out.println(e.toString());
        }
        catch (SQLException e) {
          System.out.println(e.toString());
        }
        catch (Exception e) {
          System.out.println(e.toString());
        }
        return false;
      }


    /**
     * Initialization of the servlet. <br>
     *
     * @throws ServletException if an error occure
     */
    public void init() throws ServletException {
        // Put your code here
    }

    }
```

La servlet LoginServletTask essaie de faire correspondre les valeurs saisies dans le formulaire précédant le login et le mot de passe avec celles des colonnes NomUser et MotDePasse stockées dans la table WebStockAccess.

Lorsque l’utilisateur saisit dans son navigateur l’URL http://localhost:8080/WebStock/login.jsp et qu’il clique sur le bouton Login, la méthode doGet est invoquée. Celle-ci appelle la méthode privée sendLoginForm, qui envoie la page HTML à l’utilisateur pour se connecter.

Cette dernière méthode possède deux arguments : un objet HttpServletResponse, que la méthode peut utiliser pour envoyer le résultat au navigateur, et un type booléen, withErrorMessage. Ce booléen est un drapeau qui indique si le message d’erreur doit être envoyé avec le formulaire. Ce message d’erreur informe l’utilisateur que le précédent login a échoué.

Lorsque la servlet est appelée pour la première fois, aucun message n’est envoyé. Cela explique que la valeur false soit transmise en argument à la méthode sendLoginForm (voir l’extrait de code précédant la méthode doGet).

Après l'envoi du formulaire à la servlet *via* l'attribut ACTION="/webstock/LoginServletTask" (voir code de la JSP login.jsp précédente) et la transmission des paramètres à la servlet, la méthode login(userName, password) est appelée *via* la méthode doPost de la servlet. Cette méthode retourne true ou false en fonction du résultat de la requête dans la base webstock par le biais de la table WebStockAccess.

En cas de succès, la requête est transmise à une JSP LoginSuccess pour un traitement associé à une connexion réussie au système webstock. En cas d'échec, la méthode doPost appelle à nouveau la méthode sendLoginFom avec un message d'erreur sendLoginForm (response, true).

Gestion des commandes avec Web Tools et les patterns

Vous allez maintenant construire les composantes de la partie gestion des commandes de l'application webstock à l'aide des assistants du projet Web Tools.

Au chapitre suivant, vous utiliserez un framework adapté au design d'applications 3-tiers, qui facilitera l'interaction entre les couches de présentation, la logique métier et la persistance des données avec l'aide des patterns Commande et Singleton.

Cette partie simplifiée de l'implémentation de l'application webstock concerne deux tables du modèle de données, les tables `Commande` et `Client`, et est constituée des cas d'utilisation suivants :

- Les clients doivent être enregistrés sur le site pour placer leurs commandes.
- Les clients peuvent commander des articles.
- Les clients peuvent voir leur commande.
- Les administrateurs peuvent afficher la liste des clients enregistrés.

Le système sera implémenté en utilisant un modèle de programmation fondé sur les servlets et le design pattern Commande.

La figure 7.9 décrit le design de cette partie de l'application.

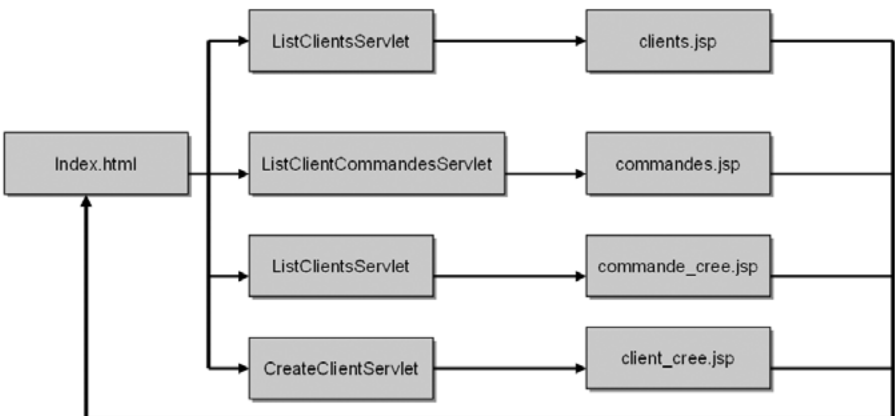


Figure 7.9

Cinématique de la gestion des commandes

Design de la partie métier

Avant toute chose, vous devez concevoir les classes métier qui supporteront le développement de l'application Web. La figure 7.10 rappelle la portion du domaine métier de l'étude webstock concernée et les relations de dépendance associées.

À la partie III de l'ouvrage, vous reviendrez plus amplement sur ces éléments du modèle lors de l'étude du mapping objet-relationnel avec l'API JPA ainsi que sur les beans entité POJO.

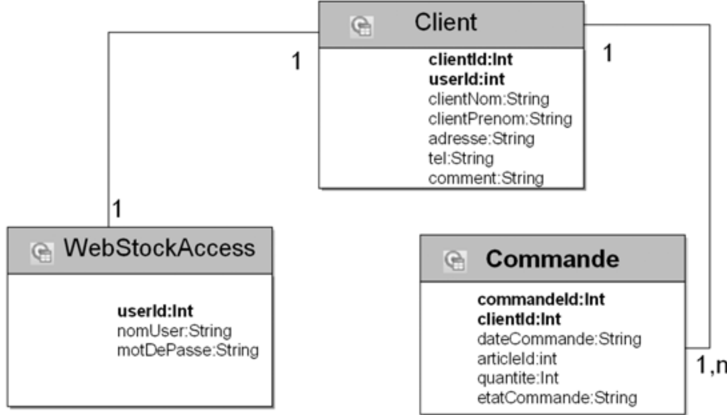


Figure 7.10

Design du modèle de données de la gestion des commandes

Vous utiliserez l'assistant de génération de code fourni par Eclipse pour la génération de code automatique des getters et setters à partir des champs de la classe concernée : par exemple la classe `Commande` (options de menu contextuel Source et Generate Getters and Setters) illustrée à la figure 7.11.

Figure 7.11

Assistant de génération de getters/-setters Eclipse

Generate Getters and Setters
Select getters and setters to create:
address
clientId
clientNom
clientPrenom
comment
tel
userId
Select All
Deselect All
Select Getters
Select Setters
Allow setters for final fields (remove 'final' modifier from fields if necessary)
Insertion point:
Cursor position
Sort by:
Fields in getter/setter pairs
Access modifier
public protected default private
final synchronized
Generate method comments
The format of the getters/setters may be configured on the Code Templates preference page.
14 of 14 selected.
OK Cancel

Faites de même avec les deux autres classes du modèle considéré, que vous regrouperez pour des raisons de lisibilité et de bonne pratique logicielle dans un package com.webstock.chap07.domain que vous aurez préalablement créé à l'aide de l'assistant de création de package sous Java Resources.

Création des classes d'accès aux données selon les design patterns Commande et Singleton

Vous allez créer des classes d'accès aux données en utilisant le design pattern Commande, qui permet aux programmes et aux objets GUI d'être complètement séparés des actions qu'ils initient.

Lorsque l'interface utilisateur reçoit une commande, elle demande à l'objet Commande de s'occuper de toutes les actions associées, sachant que la règle fondamentale de ce pattern est de ne pas se soucier des tâches qui seront exécutées.

L'interface du pattern Commande pour la gestion des données est la suivante :

```
package com.webstock.chap07.command;

import java.sql.Connection;
import java.sql.SQLException;

public interface DatabaseCommand {
public Object executeDatabaseOperation(Connection conn) throws SQLException ;
}
```

Les classes CRUD (Create/Read/Update/Delete) suivantes implémentent le pattern Commande représenté par l'interface DatabaseCommande pour effectuer les opérations dans la base et insérer les enregistrements dans les tables `Commande` et `Client` :

```
public class CreateClient implements DatabaseCommand
public class CreateCommande implements DatabaseCommand
public class ListClients implements DatabaseCommand
public class ListCommandesClients implements DatabaseCommand
```

Pour permettre l'exécution de vos classes `Commande`, vous devez créer une classe permettant d'accéder à la source de données, obtenir une connexion SQL et exécuter une commande d'accès aux données particulières.

Pour ce faire, vous vous aiderez d'un second pattern incontournable, le pattern Singleton, qui appellera la classe CommandExecution, garantissant l'instanciation à un seul objet :

```
private static CommandExecutor myOnlyInstance = null;

public static CommandExecution getInstance() throws NamingException {
    if (myOnlyInstance == null) {
        myOnlyInstance = new CommandExecution();
    }
    return myOnlyInstance;
}
```

L'exécution proprement dite d'une classe particulière `Commande` s'effectue par le biais de l'invocation suivante :

```
Object o = CommandExecution.getInstance().executeDatabaseCommand
("Instance particulière d'un objet commande d'accès aux données")
```

Pour invoquer l’objet `Commande` d’affichage de la liste des clients, vous aurez, par exemple :

```
…
try {

        ArrayList<Client> list = (ArrayList<Client>)CommandExecution.getInstance()
        ➥.executeDatabaseCommand(new command.ListClients());
        request.setAttribute("clients", list);
        RequestDispatcher rd = getServletContext().getRequestDispatcher("/clients.jsp");
        rd.forward(request, response);
    } catch (Exception e) {
        throw new ServletException(e);
    }
}
```

L’accès à la source de données s’effectue par un lookup du contexte JNDI, comme le montre l’implémentation de la méthode `getDataSource()` suivante, la référence à cette ressource étant stockée dans le fichier de source de données associé (voir configuration de la source de données du serveur JBoss) :

```
public DataSource getDataSource() throws NamingException {

    if (ds == null) {

        InitialContext ctx = new InitialContext();
        Context envCtx = (Context) ctx.lookup("java:comp/env");
        ds = (DataSource) envCtx.lookup("jdbc/WebStockDB");

    }

    return ds;
}
```

L’exécution d’une commande d’accès aux données particulières utilise la méthode `executeDatabaseCommand()` :

```
public Object executeDatabaseCommand(DatabaseCommand c) throws Exception {

    Connection conn = null;
    try {
        conn = getConnection();
        Object o = c.executeDatabaseOperation(conn);
        return o;
    } catch (SQLException e) {
        throw e;
    } catch (NamingException ne) {
        throw ne;
    } finally {
        if (conn != null) conn.close();
    }
}
```

À présent que les bases de votre conception Java ont étés implémentés, vous pouvez passer à la mise en œuvre en utilisant l’outillage Web Tools.

Développement des classes CRUD pour les objets métier *Client* et *Commande*

À présent que l'ossature pour l'invocation des méthodes métier a été développée pour les beans métier `Commande` et `Client`, passez aux opérations CRUD proprement dites sur ces classes.

Vous verrez les opérations de création/affichage des enregistrements associés à la classe `Client`, qui vaut aussi pour l'objet métier `Commande` (le code source complet est disponible sur la page Web dédiée à l'ouvrage).

Voici le code de création et d'affichage associé à la classe métier `Client` :

```
package com.webstock.chap07.command;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.webstock.chap07.domain.Client;

public class CreateClient implements DatabaseCommand {

private Client clt;

public CreateClient(Client c) {
    this.clt = c;
}

public Object executeDatabaseOperation(Connection conn) throws SQLException {

    PreparedStatement stmt = conn.prepareStatement("INSERT INTO CLIENT (CLIENTID, USERID,
    ➥CLIENTNOM, CLIENTPRENOM, ADDRESSE,TEL,COMMENT) VALUES ( ?, ?, ?, ?, ?, ?, ?)");
    stmt.setInt(1, clt.getClientId());
    stmt.setString (2, clt.getUserId());
    stmt.setString (3, clt.getClientNom());
    stmt.setString (4, clt.getClientPrenom());
    stmt.setString (5, clt.getAdresse());
    stmt.setString (6, clt.getTel());
    stmt.setString (7, clt.getComment());
    int lignes_maj = sta.executeUpdate();
    stmt.close();
    return new Integer(lignes_maj);
}

}


package com.webstock.chap07.command;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;
import java.util.ArrayList;
import com.webstock.chap07.domain.Client;

/**
 * Affiche les clients existants dans la base Webstock
 */
```

```
public class ListClients implements DatabaseCommand {

public Object executeDatabaseOperation(Connection conn) throws SQLException {


    ArrayList<Client> list = new ArrayList<Client>();
    Statement sta = conn.createStatement();
    ResultSet rs = sta.executeQuery("SELECT CLIENTID, CLIENTNOM, CLIENTPRENOM,
    ➥ADRESSE, TEL, COMMENT FROM CLIENT");
    while(rs.next()) {
        Client clt = new Client();
        clt.setClientId(rs.getInt(1));
        clt.setClientNom(rs.getString(2));
        clt.setClientPrenom(rs.getString(3));
        clt.setAdresse(rs.getString(4));
    clt.setTel(rs.getString(5));
        clt.setComment(rs.getString(6));

        list.add(clt);
    }

    rs.close();
    sta.close();

    return list;


}

}
```

Création des composants servlets et JSP avec les assistants Web Tools

Selon le modèle classique MVC (modèle, vue, contrôleur), vous avez implémenté à la section précédente la couche modèle constituée d'objets du domaine et des classes `Commande`. La couche contrôleur sera implémentée sous la forme d'un composant servlet et la vue sous la forme de pages JSP.

Web Tools fournit un ensemble complet d'assistants et d'éditeurs pour faciliter la gestion des servlets et des JSP. Nous supposons créé un projet Web de type Web dynamique, appellé WebStock.

1. Créez un package sous le répertoire src créé par défaut, et appelez-le com.webstock.chap07.servlet
2. Lancez l'assistant de création de servlet en sélectionnant File, New, Other et Web-Servlet, puis saisissez ListClientsServlet comme nom de servlet.
3. Cliquez sur Next, et spécifiez une description optionnelle pour la servlet, ainsi que des paramètres d'initialisation éventuels lors du chargement de la servlet (méthode `init()` de l'objet ServletConfig) et le mapping de l'URL (/ListClientsServlet par défaut).
4. Cliquez sur Next.

5. À cette étape, l’assistant propose un certain nombre d’options concernant les interfaces que la servlet peut implémenter (par défaut, l’interface javax.servlet.Servlet) ainsi que les méthodes qui sont automatiquement générées lors de la création de la servlet. Cochez la méthode doGet comme illustré à la figure 7.12.

Figure 7.12

Assistant de création de servlet et de définition des méthodes

6. Cliquez sur Finish pour générer la servlet. Le code suivant de la servlet est généré (notez que le fichier descripteur web.xml de l’application Web est automatiquement mis à jour) :

```
<display-name>WebStock</display-name>
 <servlet>
   <description>Affichage des clients</description>
   <display-name>ListClientsServlet</display-name>
   <servlet-name>ListClientsServlet</servlet-name>
  <servlet-class>
  com.webstock.chap06.servlet.ListClientsServlet
  </servlet-class>
 </servlet>
 <servlet-mapping>
   <servlet-name>ListClientsServlet</servlet-name>
   <url-pattern>/ListClientsServlet</url-pattern>
 </servlet-mapping>
</web-app>
```

Code des servlets de support à la gestion des objets *Commande* et *Client*

Terminez par le code associé aux servlets de gestion des clients, respectivement ListClientsServlet pour l’affichage des clients et CreateClientServlet pour la création de

l'instance client (le code complet pour la gestion des commandes, ListCommandes-Servlet et CreateCommandeServlet est disponible sur la page Web dédiée à l'ouvrage) :

```
package com.webstock.chap07.servlet ;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.RequestDispatcher;
import java.util.ArrayList;

import com.webstock.chap07.domain.*;
import com.webstock.chap07.command.*;

/**
 * Classe d'implementation pour la servlet ListClients
 *
 */
 public class ListClientsServlet extends javax.servlet.http.HttpServlet {
    /* (non-Java-doc)
 * @see javax.servlet.http.HttpServlet#HttpServlet()
 */

/* (non-Java-doc)
 * @see javax.servlet.http.HttpServlet#doGet(HttpServletRequest request,
 ➥HttpServletResponse response)
 */
protected void doGet(HttpServletRequest request, HttpServletResponse response)
➥throws ServletException, IOException {

    try {

        ArrayList<Client> list = (ArrayList<Client>)CommandExecution.getInstance()
        ➥.executeDatabaseCommand(new com.webstock.chap07.command.ListClients());
        request.setAttribute("clients", list);
        RequestDispatcher rd = getServletContext().getRequestDispatcher("/clients.jsp");
        rd.forward(request, response);

    } catch (Exception e) {
        throw new ServletException(e);
    }
}

/* (non-Javadoc)
 * @see javax.servlet.GenericServlet#init()
 */
public void init() throws ServletException {
    // TODO Auto-generated method stub
    super.init();

    try {
        CommandExecution.getInstance();
    } catch (Exception e) {
        throw new ServletException(e);
    }
}
}
```

Comme vous pouvez le remarquer sur ce dernier code, l'attribut requête "clients" est positionné à un objet générique ArrayList et l'objet RequestDispatcher est utilisé pour renvoyer la réponse à la page JSP clients.jsp. Enfin, dans la méthode `init()` de la servlet ListClientsServlet, une instance de la commande CommandExecution est ajoutée afin de récupérer la source de données (DerbyDS) et la « cacher » pour les besoins futurs de gestion des connexions à la base WebStockDB.

```
package com.webstock.chap07.servlet;

import java.io.IOException;
import java.sql.Timestamp;

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

import com.webstock.chap07.command.CommandExecution;
import com.webstock.chap07.command.DatabaseCommand;
import com.webstock.chap07.command.CreateClient;
import com.webstock.chap07.domain.Client;

/**
 * Servlet implementation class for Servlet: CreateCustomerServlet
 *
 */
 public class CreateClientServlet extends javax.servlet.http.HttpServlet implements
 ➥javax.servlet.Servlet {

/* (non-Java-doc)
 * @see javax.servlet.http.HttpServlet#doGet(HttpServletRequest request,
 ➥HttpServletResponse response)
 */
protected void doGet(HttpServletRequest request, HttpServletResponse response)
➥throws ServletException, IOException {

    try {

        // creation du client
        String prenom = request.getParameter("prenom");
        String nom = request.getParameter("nom");
        int userid = request.getParameter("userid");
        String tel = request.getParameter ("tel");
        String adresse = request.getParameter ("adresse");
        String comment = request.getParameter("comment");
        int clientId = Math.abs((int)System.currentTimeMillis());
        Client c = new Client();
        c.setClientId(clientId);
        c.setPrenom (prenom);
        c.setNom(nom);
        c.setUserId(userid);
        c.setAdresse(adresse);
        c.setTel(tel);
        c.setComment(comment);

        DatabaseCommand command = new CreateClient(c);
```

```
        int lignes = (Integer)CommandExecution.getInstance()
        ➥.executeDatabaseCommand(command);
        RequestDispatcher rd = getServletContext().getRequestDispatcher
        ➥("/client_cree.jsp");
        rd.forward(request, response);

    } catch (Exception e) {
        throw new ServletException(e);
    }

}

/* (non-Javadoc)
 * @see javax.servlet.GenericServlet#init()
 */
public void init() throws ServletException {
    // initialize servlet
    super.init();

    try {
        CommandExecution.getInstance();
    } catch (Exception e) {
        throw new ServletException(e);
    }

}
}
```

Le code de la page clients.jsp créé par l'assistant de création de page JSP de Web Tools conclut la gestion de l'affichage des clients :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
➥pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Liste des clients<title>
</head><body>

<jsp:useBean id="clients" type="java.util.ArrayList<domain.Client>" scope="request"/>

<b>Liste des clients enregistrés dans la base WebStock:</b><br>
<table border="1">
<tr>
<th>Id</th>
<th>Nom</th>
<th>Prenom</th>
<th>Telephone</th>
<th>Commentaire</th>
</tr>
<% for(com.webstock.chap07.domain.Client c : clients) { %>
<tr>
<td><%= c.getClientId() %></td>
<td><%= c.getClientNom() %></td>
<td><%= c.getClientPrenom() %></td>
<td><%= c.getTel() %></td>
```

```
<td><%= c.getAdresse %></td>
<td><%= c.getComment() %></td>

<td><a href="/WebStock/ListCommandes?clientId=<%= c.getClientId() %>">Commandes</a>
➥</td>
<% } %>
</body>
</html>
```

Remarquez dans ce dernier extrait l'utilisation de certaines fonctions de J2SE 5.0, comme l'utilisation des génériques et de la boucle for.

Configuration de la source de données du serveur JBoss

Avant de démarrer le serveur JBoss, vous devez ajuster sa configuration pour qu'il fonctionne avec le SGBD Derby, l'ajustement de la configuration devant être effectué dans le répertoire de configuration du serveur (JBOSS_HOME\conf, soit, dans notre configuration particulière, C:\jboss-4.2.0.GA\server\default\conf).

Vous devez également mettre à jour la référence à la source de données pour qu'elle fonctionne avec la base Derby, JBoss étant par défaut configuré avec la base de données HypersonicDB.

Bonne pratique

Une bonne pratique pour ajuster les fichiers de configuration et gérer ses configurations de manière pratique consiste à copier ces derniers dans un répertoire et à les ajuster en fonction de vos besoins. JBoss peut être démarré avec le nom du répertoire de configuration (sans le chemin complet) comme premier paramètre et l'option –c (run –c maconfig, par exemple).

1. Copiez le driver JDBC du SGBD Derby (derby.jar), dans le répertoire lib de la configuration.
2. Créez un fichier derby-ds.xml dans le répertoire deploy avec la configuration suivante :

```
<datasources>
   <local-tx-datasource>
      <jndi-name>DerbyDS</jndi-name>
      <connection-url>
jdbc:derby:${jboss.server.data.dir}${/}derby${/}WebStockDB;create=true
      </connection-url>
      <!-- The driver class -->
      <driver-class>org.apache.derby.jdbc.EmbeddedDriver</driver-class>
      <user-name>derby</user-name>
      <password></password>
      <min-pool-size>5</min-pool-size>

      <max-pool-size>20</max-pool-size>
      <idle-timeout-minutes>5</idle-timeout-minutes>
      <track-statements/>
      <depends>jboss:service=Derby</depends>
   </local-tx-datasource>
</datasources>
```

Déploiement de l'application sur le serveur JBoss

À ce stade, votre application peut être publiée et déployée sur votre serveur JBoss par l'assistant de publication de Web Tools.

Pointez votre navigateur sur l'URL http://localhost:8080/Webstock/index.html. Si tout se passe bien, une page d'accueil vous propose un certain nombre d'options, comme l'affichage des clients enregistrés dans la base webstock illustré à la figure 7.13.

Figure 7.13

Page d'affichage de la liste des clients

Liste des clients enregistrés dans la base WebStock:

Id	Nom	Prenom	Adresse	Telephone	Commentaire	Commande
CL001	ALPHONSO	PIERRE	2 Rue des Amandiers NANTERRE	0134567889	Client régulier	CLOO1
CL002	DUPONT	François	2 Rue des PCs PARIS	0145234455	Client occasionnel	CLO02

accueil

En résumé

Vous avez vu dans ce chapitre que la mise en œuvre de l'outillage Web Tools pour un développement sans framework particulier exige une certaine rigueur dans la construction de l'ossature sous-tendant le développement Web.

Le chapitre suivant se penche sur Seam, un puissant framework destiné à compléter et faciliter l'outillage à la disposition du développeur JBoss.

8

Développement Web avec le framework JBoss Seam

Ce chapitre complète l'outillage Web Tools avec Seam, un framework « Web 2.0 Ready », qui facilite le développement JEE, en particulier pour les applications fondées sur les technologies JSF et EJB3.

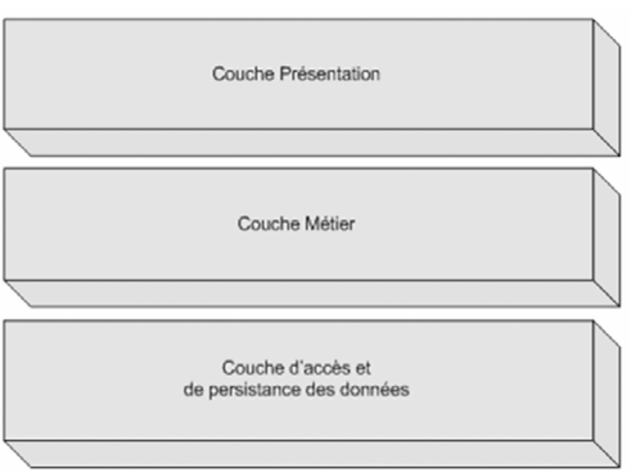
Après un bref rappel de la genèse des frameworks JEE et des fondamentaux du framework Seam, vous mettrez en œuvre ce dernier sur l'étude de cas.

Les frameworks J2EE

Depuis la fin des années 1990, les best practices de développement Java/J2EE, ont imposé, bien avant l'apparition des frameworks tels que Struts ou Spring, une approche de développement en couches : couche présentation, couche métier et couche d'accès aux données et de persistance, comme l'illustre la figure 8.1.

Figure 8.1

Architecture J2EE/ JEE standard



Cette approche en couches s'appuyait sur les API de base des servlets. Une servlet traite les requêtes Web en entrée et affiche le résultat sous forme de balises HTML, qui permettent l'affichage de pages au sein d'un navigateur. L'inconvénient de cette approche était que les balises HTML chargées de la présentation étaient mêlées à du code Java, ce qui impactait fortement la maintenabilité de l'ensemble et ne permettait pas une répartition efficace des rôles au sein de l'équipe projet (designer de pages Web et développeurs Java).

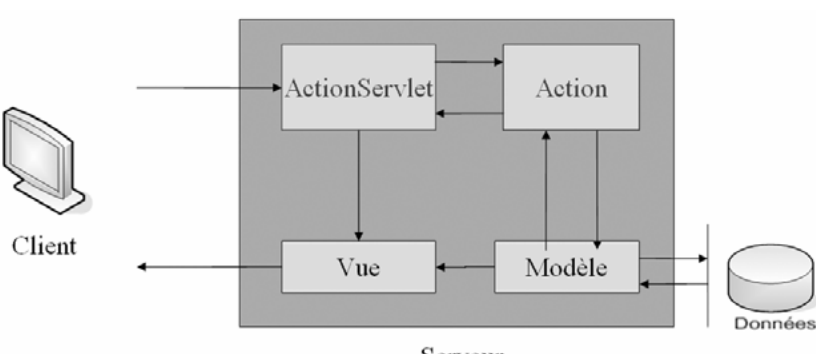
Est ensuite venue la technologie JSP (JavaServer Pages), qui permet aux développeurs d'embarquer des éléments dynamiques au sein de pages HTML. Ces éléments dynamiques utilisent des balises spécifiques (directive d'import, de code Java à exécuter, de résultats à afficher, etc.), ainsi que des variables implicites (request, response, session, etc.).

Bien que mieux structurée que les servlets, cette approche de développement présentait plusieurs inconvénients, le principal d'entre eux étant le manque de lisibilité du code du fait du mélange du HTML et du Java, d'où une confusion entre la présentation et la logique applicative.

Le framework Open Source Struts de la fondation Apache est ensuite apparu au début des années 2001. Dédié à la conception d'applications Web à base de servlets, de JSP et de JavaBeans, ce framework fonctionne schématiquement comme illustré à la figure 8.2.

Figure 8.2

Architecture de Struts



Conformément au modèle appelé MVC2, Struts s'appuie sur une classe principale, la classe org.apache.struts.action.ActionServlet, qui assure le rôle de contrôleur. Celle-ci est associée à toutes les requêtes HTTP se terminant par .do *via* la configuration du fichier de mapping associé (balise <servlet-mapping>). En plus de cette classe, un fichier de configuration struts-config.xml contient une liste d'URL dédiées chacune à une action. Pour transférer les données du formulaire à l'action, Struts utilise une instance dérivée d'ActionForm. Le lien entre cette instance, l'URL demandée et l'action est aussi défini dans le fichier de configuration Struts.

Côté modèle, Struts n'impose rien, et aucune classe n'est fournie, laissant une totale liberté de choix des technologies à utiliser pour l'implémentation du modèle (ce qui est aussi un inconvénient dans la mesure ou aucun standard n'est proposé pour organiser cette partie du développement).

Peu de temps après sont apparues les premières implémentations Sun des spécifications JSF (JavaServer Faces) — JSR-000127. Conçues pour le développement rapide d'applications Web, ces spécifications proposent une interface graphique riche associée à une logique de programmation événementielle. Pour cela, la norme définit un cycle normalisé de

traitement des requêtes et fournit une abstraction complète du protocole HTTP. En découle une gestion transparente de l'état des composants.

En comparaison de JSP, de l'API servlet et de Struts, qui oblige les développeurs à produire beaucoup de code pour gérer les aspects présentation, interaction et ergonomie, les composants JSF prennent en charge automatiquement ces opérations.

Limitations de Struts et JSF

Le framework Struts est aujourd'hui déprécié du fait des nouveaux besoins des applications Web 2.0 complexes. Les raisons à cela sont nombreuses. Citons en vrac l'absence de formulaires de type stateful permettant la conservation des états et une gestion du contexte évolué, le fait qu'une seule JSP soit associée à un unique form bean (bean associé à un formulaire), qu'une seule classe Action puisse mapper un form bean pour chaque configuration, sauf à passer par des solutions de contournement difficilement maintenables, etc.

Ces limitations ont favorisé l'émergence de JSF, qui propose une approche plus orientée composant, se rapprochant d'une approche davantage RAD et à la .Net. Par contre, toute la « plomberie » nécessaire à l'invocation des méthodes et des services métier avec EJB3 ainsi que la gestion des sessions reste à la charge du développeur, ce qui complexifie grandement le développement d'applications JEE.

> **Web 2.0**
>
> Le concept de Web 2.0 est vu comme la promesse d'une nouvelle version de l'Internet. Selon la définition de l'éditeur Tom O'Reilly, le Web 2.0 repose sur un ensemble de modèles de conception. Ces systèmes architecturaux plus intelligents permettent d'utiliser des modèles légers qui rendent possible la syndication des données et des services. « Le Web 2.0, dit-il, c'est le moment ou les gens réalisent que ce n'est pas le logiciel qui fait le Web mais les services. »

JSF (JavaServer Faces)

Un bref rappel des fondamentaux de la technologie JSF n'est pas superflu ici. Le lecteur débutant dans cette technologie est invité à examiner les nombreux ouvrages sur le sujet, notamment ceux cités en annexe.

JSF est devenu un standard du JCP pour le développement d'applications Web, en particulier pour le développement de la couche présentation. Cet état de fait conforte le constat que Struts est déprécié pour le développement d'applications Web 2.0 performantes.

JSF fournit les caractéristiques remarquables suivantes :

- Gestion simplifiée de la navigation interpage.
- Modèle de composants graphiques standards (widgets).
- Gestion des erreurs et de la validation des formulaires.
- Gestion événementielle évoluée, qui se rapproche de Visual Basic.
- Gestion évoluée des composants JavaBeans (avec la notion importante de backing beans, spécialement utilisés dans JSF pour encapsuler certaines parties des composants qui constituent une page et faire le lien avec la couche métier). Les backing beans

sont utiles pour afficher les données provenant de la couche métier ou pour la saisie des données de l'utilisateur.

- Support de l'internationalisation.

Toutes ces caractéristiques permettent au développeur de se concentrer sur son application et de se décharger de la gestion rébarbative des pages et des composants graphiques qui la composent.

Mise en œuvre de JSF avec Eclipse Web Tools

Dans cette section, vous allez créer et exécuter une application JSF simple de type HelloWord destinée à illustrer les composants d'une application Web fondée sur cette technologie et à évaluer la richesse des outils de support au développement JSF offerts par le projet Web Tools.

Le serveur JBoss 4.2 dispose d'un support natif de l'implémentation de référence de Sun (JSF 1.2). Cela vous épargnera les étapes d'intégration et de configuration des bibliothèques JSF dans le projet.

Nous supposons la cible définie dans la configuration de l'IDE (menu Preferences).

1. Créez un nouveau projet Web de type dynamique (options d'Eclipse File, New, Other, Web et Dynamic Web Project).
2. Appelez-le HelloJSF, puis cliquez sur Next (voir figure 8.3).

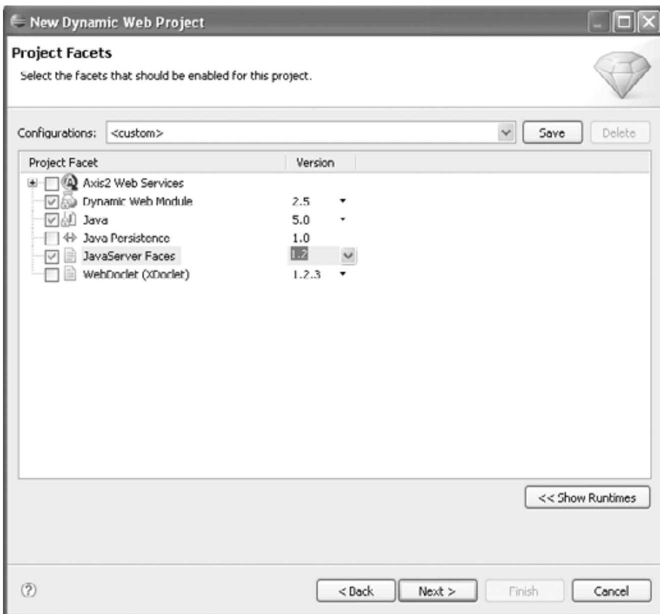


Figure 8.3

Configuration du projet avec les fonctionnalités Facet de Web Tools

Vous pouvez constatez que Web Tools propose les implémentations de référence de JSF.

3. Cochez cette option, et sélectionnez la version 1.2, puis cliquez sur Next.
4. Les champs indiquant le contexte root de l'application et les répertoires Web (WebContent) et source (src) sont proposés. Cliquez sur Next.

La boîte de dialogue suivante de l'assistant de création de projet Web dynamique propose de configurer les bibliothèques JSF ainsi que l'emplacement des différents fichiers de configuration JSF du projet.

5. Sélectionnez lServer Supplied JSF Implementation, et cliquez sur Finish (voir figure 8.4).
6. Cliquez sur I Agree dans la boîte de dialogue de copyright concernant la version des bibliothèques JSF utilisées.

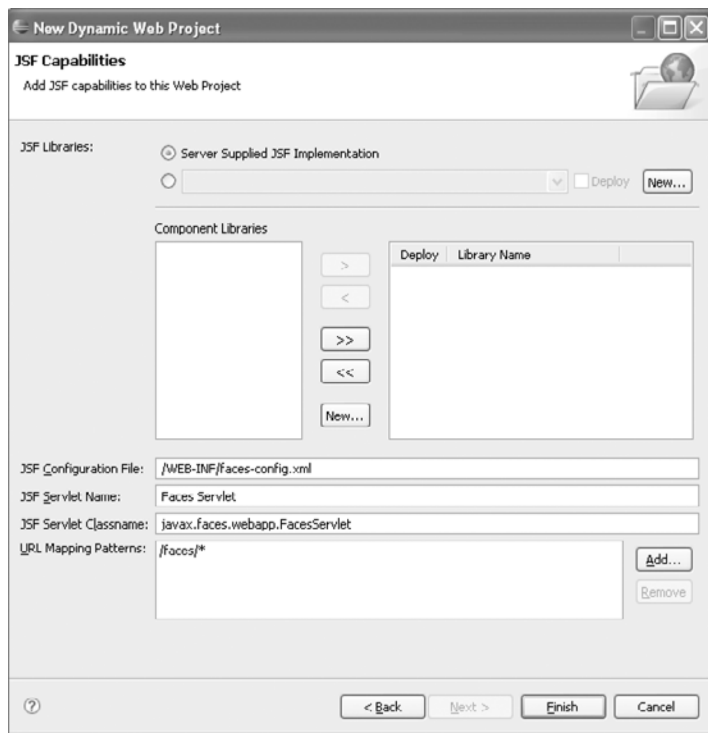


Figure 8.4

Configuration du projet dynamique avec JSF

L'arborescence « typique » d'un projet Web/JSF est générée comme illustré à la figure 8.5.

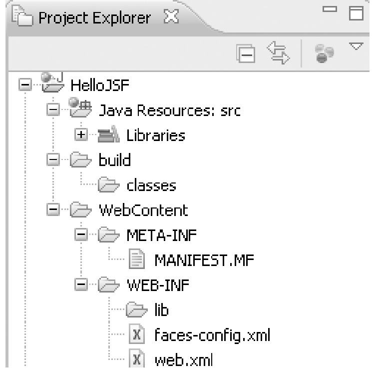


Figure 8.5

Arborescence du projet Web/JSF

7. Créez votre première page JSF en sélectionnant l'assistant de création de page JSP puis en donnant à la page le nom hello.jsp. Cliquez sur Finish.
8. Ajoutez les taglibs JSF dans la page créée en profitant des fonctionnalités de complétion automatique de code offertes par Web Tools (voir figure 8.6).

```
*hello.jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>


<html>
<f:view>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Hello JSF</title>
</head>
<body>
<h:form>

</h:form>

</body>
</f:view>
</html>
```

Figure 8.6

Construction de la page JSF

Les deux lignes (requises) contenant la balise @taglib sont des directives qui font respectivement référence à l'emplacement des balises JSF qui définissent les éléments de base de JSF ainsi qu'aux éléments HTML de la page.

9. Ajoutez juste au-dessus de la balise <f:view> les lignes suivantes :

```
<h:outputLabel> rendered="true" value="Nom" </h:outputLabel>
  <h:inputText tabindex="#{loginBean.nom}"></h:inputText>
</f:view>
```

Ces lignes vont permettre la construction d'un formulaire destiné à transmettre la valeur saisie à un backing bean (loginBean) d'affichage.

10. Créez la classe com.chap08.jsf.hello suivante à l'aide de l'assistant de création de classe Eclipse :

```
package com.chap08.jsf.hello;

public class LoginBean {
String nom;

public LoginBean () {
    super();
}

public String getNom() {
    return nom;
}
```

```
public void setNom(String nom) {
    this.nom = nom;
}
}
```

11. Dans l'explorateur de projet, double-cliquez sur le fichier de configuration JSF. L'éditeur JSF intégré s'affiche comme illustré à la figure 8.7.

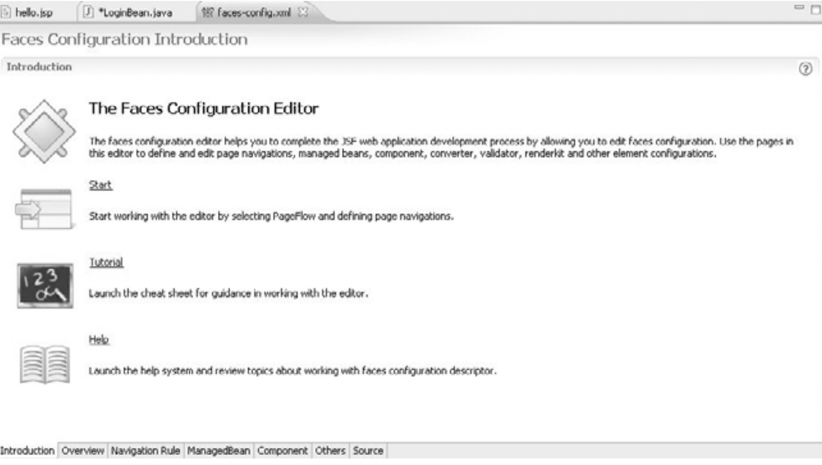


Figure 8.7

Assistant Faces Configuration Editor

12. Sélectionnez l'onglet ManagedBean, puis cliquez sur Add pour lancer l'assistant de création ManagedBean illustré à la figure 8.8.

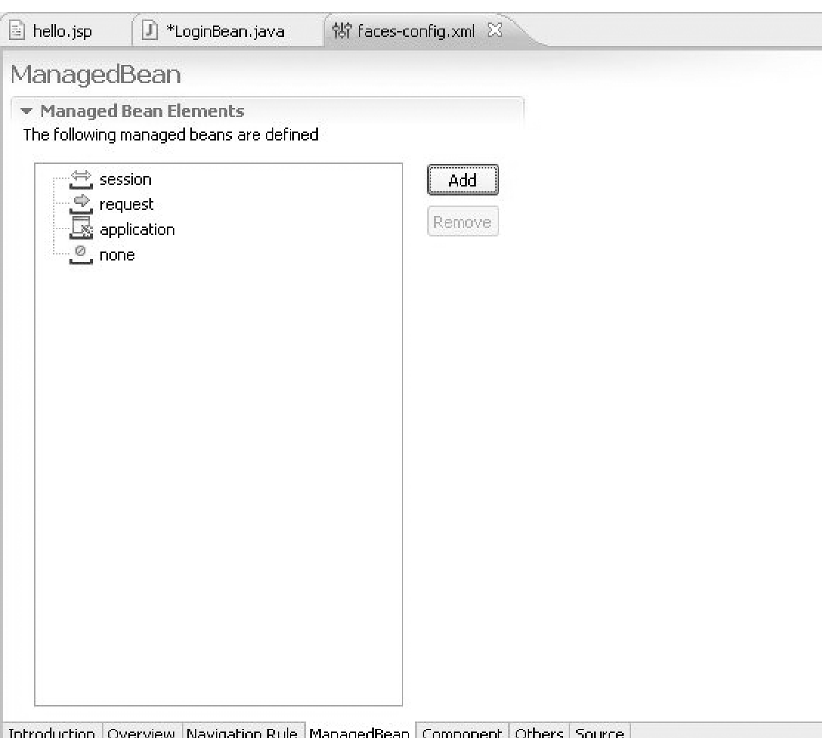


Figure 8.8

Configuration du ManagedBean (1/2)

13. Sélectionnez la classe com.chap08.jsf.hello.loginBean et cliquez sur Next.
14. La boîte de dialogue de dialogue de configuration propose de définir le scope du bean (les scopes possibles sont application, session, request ou none). Sélectionnez session, et cliquez sur Finish.
15. L'écran suivant propose l'état de la configuration générale du bean ainsi enregistré. Cliquez sur Ctrl+S pour sauvegarder la configuration (voir figure 8.9).

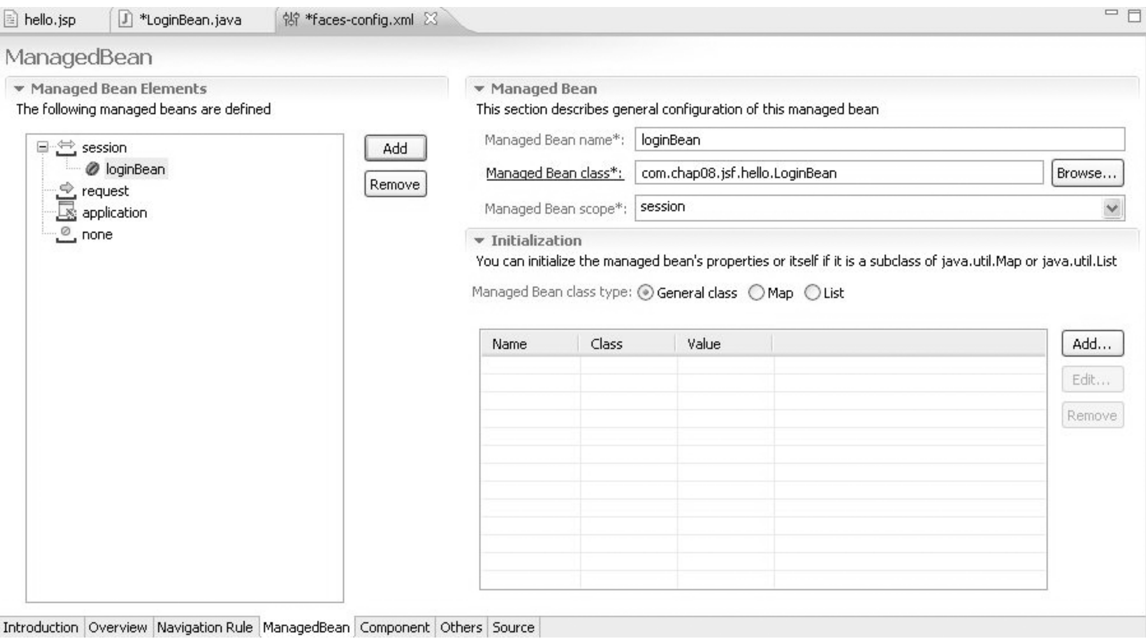


Figure 8.9

Configuration du ManagedBean (2/2)

La référence de ce bean est définie dans la configuration du fichier faces-config.xml avec l'attribut session correctement positionné :

```
<managed-bean>
    <managed-bean-name>
    loginBean</managed-bean-name>
    <managed-bean-class>
    com.chap08.jsf.hello.LoginBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

16. Basculez sur la page hello.jsp.
17. Faites un clic droit sur la page, et sélectionnez Validate dans le menu contextuel.
18. Ajoutez la balise JSF `<h :inputMessage value= »#{} »></h :inputMessage>`, comme illustré à la figure 8.10.
19. Sélectionnez loginBean dans la liste déroulante.
20. Entrez le nom d'une propriété, par exemple message, et sauvegardez la configuration.

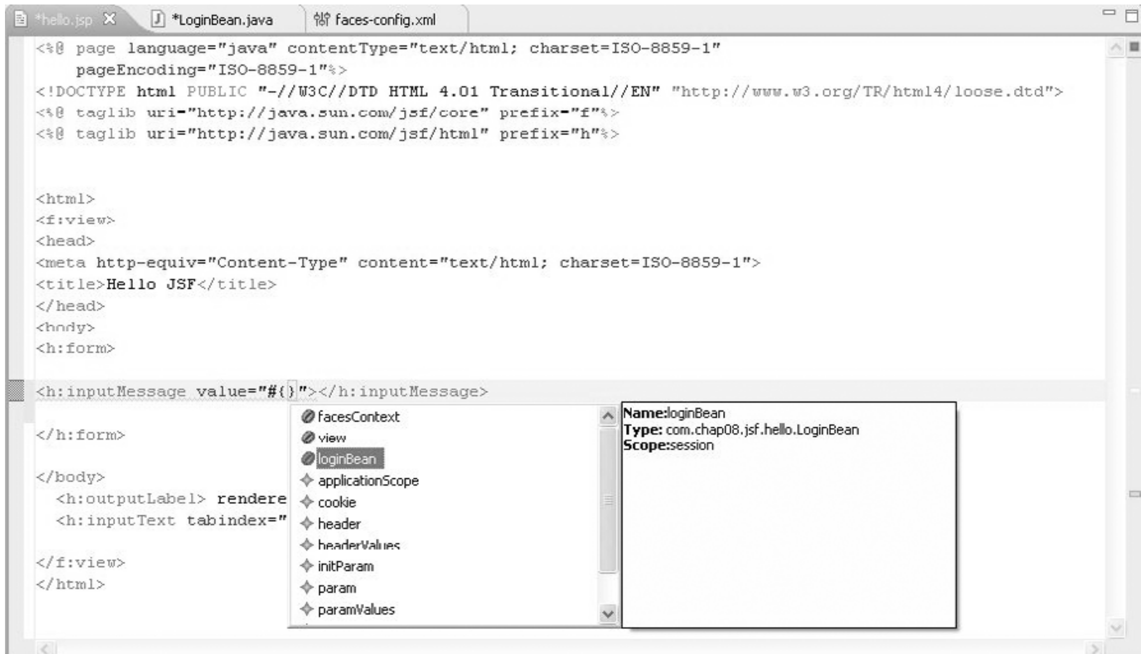


Figure 8.10

Configuration du bean dans la page JSF

21. Ajoutez la référence à une ressource de type message *via* le code suivant, juste après la balise de fin </title> :

```
<f :loadBundle basemane= "com.chap08.jsf.hello.message" var="msg"/>
```

22. Ajoutez la référence au fichier properties correspondant en créant, dans le package com.chap08.jsf.hello sous le dossier Java Resources:src, un fichier messages.properties contenant les lignes suivantes (destinées à externaliser les libellés utilisés par les pages JSP, ce qui est une bonne habitude à prendre) :

```
nom=Nom
login=Login
welcome=Bienvenue sur JSF
fin= !
```

Lors de la compilation, ce fichier sera copié dans le répertoire classes du projet et découvert par le runtime Eclipse lors de l'exécution.

23. Complétez votre page hello.jsp pour qu'elle intègre au final les lignes d'affichage de la page avec les champs Login ainsi que la référence aux libellés saisis :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
➥"http://www.w3.org/TR/html4/loose.dtd">
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<html>
```

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Hello JSF</title>
<f:loadBundle basename="com.chap08.jsf.hello.messages.messages" var="msg"/>
</head>
<body>

<f:view>
  <h:form>

   <h:messages layout="table"></h:messages>
   <h:panelGrid columns="2">
      <h:outputLabel rendered="true" value="#{msg.nom}"></h:outputLabel>
      <h:inputText value="#{loginBean.nom}" tabindex="0"></h:inputText>
  </h:panelGrid>

  <h:commandButton action="hello" value="#{msg.login}"></h:commandButton>
</h:form>

</f:view>
</body>
</html>
```

Comme vous pouvez le voir dans ce dernier extrait, la balise outputLabel affiche un message « Nom » à partir du fichier properties créé lors des étapes précédentes.

La ligne contenant la balise commandButton permet de soumettre le formulaire HTML en lui passant l'attribut action positionné à « hello », qui va « matcher » ce mot-clé dans les sous-balises de la balise principale navigation-outcome.

Configuration de la cinématique des pages JSF avec l'éditeur de configuration Web Tools

Vous allez commencer par créer une deuxième page JSP destinée à afficher un message de bienvenue ainsi qu'à récupérer dans la session le login de la personne. Vous utiliserez pour cela l'éditeur JSF Faces Configuration Resource Editor, qui vous aidera grandement dans la gestion de la cinématique des pages.

1. À l'aide de l'assistant de création de page JSF, créez la page welcome.jsp suivante :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
➥"http://www.w3.org/TR/html4/loose.dtd">
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>

<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Bienvenue</title>
<f:loadBundle basename="com.chap08.jsf.hello.messages" var="msg"/>
```

```
</head>
<body>
<f:view>

 < h3>
    <h:outputLabel value="#{msg.welcome}#{loginBean.nom}"></h:outputLabel>
     <h:outputLabel value="#msg.fin}" />
  </h3>

</f:view>

</body>
</html>
```

2. Créez une troisième page JSP, la page index.jsp suivante associée à la balise welcome-file du fichier de configuration web.xml afin de permettre de router l'application vers la page hello.jsp :

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
  <jsp:forward page="/hello.jsf" />
</body>
</html>
```

Vous allez maintenant configurer la cinématique des pages JSF avec l'éditeur de configuration Web Tools.

3. Double-cliquez sur le fichier faces-config.xml pour lancer l'éditeur de configuration des pages JSF et sélectionnez l'onglet Navigation Rule.
4. Cliquez sur le menu Palette pour faire apparaître l'éditeur graphique JSF Web Tools illustré à la figure 8.11.

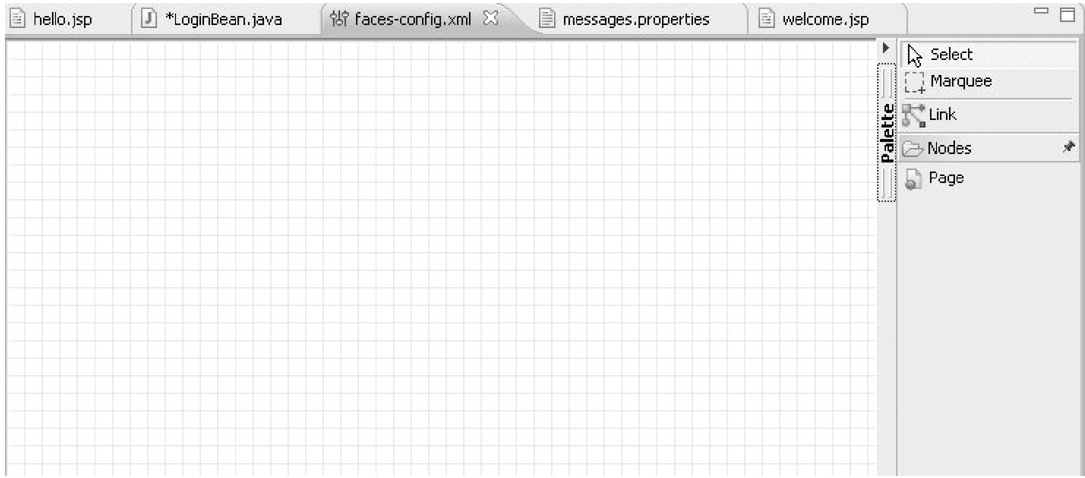


Figure 8.11

Configuration de la cinématique JSF avec l'assistant Web Tools (1/2)

5. Faites un glisser-dépose des pages hello.jsp et welcome.jsp dans l'éditeur JSF, et reliez-les à l'aide de l'option Link de la palette.
6. Sélectionnez la ligne et configurez-la *via* l'onglet Properties en renseignant les propriétés suivantes (voir figure 8.12) :
   – From Outcome : hello.
   – Redirect : false.

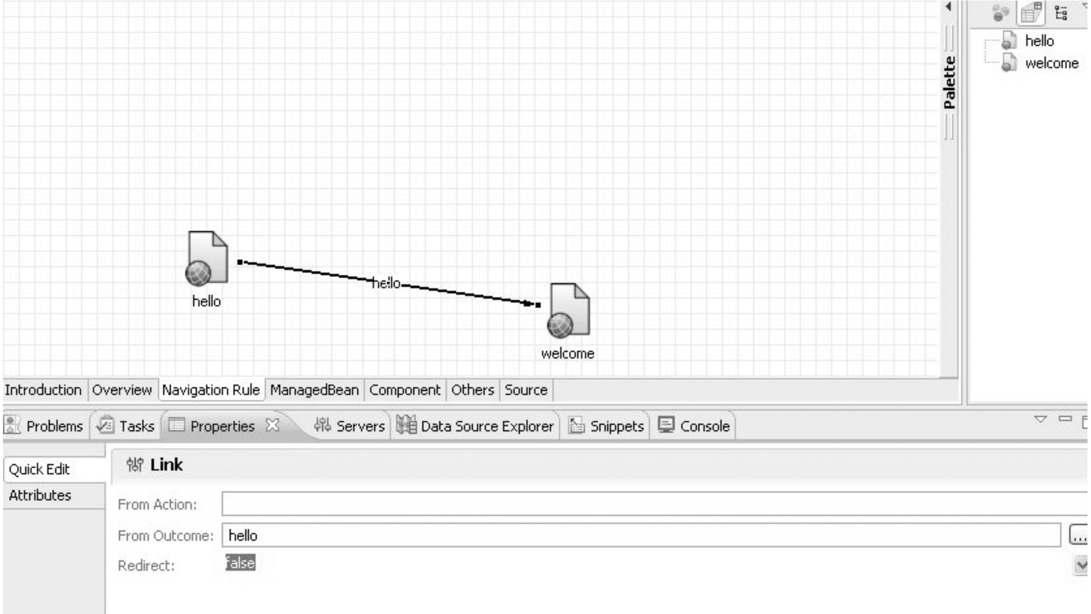


Figure 8.12

Configuration de la cinématique JSF avec l'assistant Web Tools (2/2)

Vous pouvez voir l'impact de cette définition de la cinématique du projet helloJSF dans le fichier de configuration JSFfaces-config.xml :

```
<navigation-rule>
    <display-name>hello</display-name>
    <from-view-id>/hello.jsp</from-view-id>
    <navigation-case>
        <from-outcome>hello</from-outcome>
        <to-view-id>/welcome.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

Ce bloc XML définit les règles de navigation au sein des pages du projet. La balise <from-view-id> définit la page initiale avant l'invocation de la requête vers le bean managé ; dans votre cas, il s'agit de la page hello.jsp.

Le listener associé au bean ne dispose que d'une option par l'intermédiaire de la chaîne hello retournée : l'invocation de la page welcome.jsp définie dans le fichier de configuration JSF ci-dessus.

Déploiement de la mini-application helloword sur JBoss

Il vous reste un dernier ajustement à apporter à votre fichier descripteur de déploiement web.xml généré :

1. Remplacez le contenu de la ligne contenant la balise <url-pattern> :

```
<url-pattern>/faces/*</url-pattern>
```

Par le contenu suivant :

```
<url-pattern>*.jsf </url-pattern>
```

`*.jsf` est le modèle d'URL utilisé au lieu de `faces/*` pour signaler que la page redirigée dans la page index.jsp doit être prise en charge par la servlet JSF au sein du conteneur Web de JBoss.

2. Faites une publication complète de votre projet sur le serveur *via* l'option du menu contextuel du serveur Publish, et lancez votre serveur. Cela a pour effet de vous placer automatiquement sur la page hello.jsp, comme l'illustre la figure 8.13.

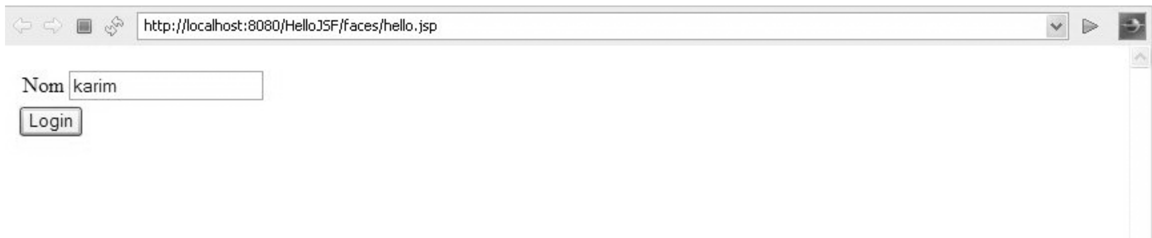


Figure 8.13

Déploiement de helloword avec l'assistant de déploiement Web Tools (1/2)

3. Cliquez sur le bouton Login. Vous devez voir invoquée la page de bienvenue illustrée à la figure 8.14.

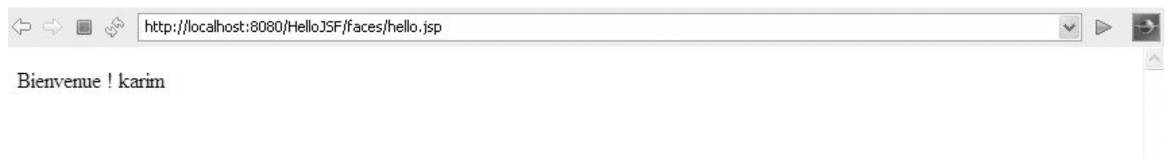


Figure 8.14

Déploiement de helloword avec l'assistant de déploiement Web Tools (2/2)

Cette section construite autour d'un exemple simple vous a permis d'apprécier la richesse de la panoplie Web Tools.

JBoss Seam

Après un rappel de la genèse du projet JBoss Seam et de ses caractéristiques fondamentales, vous verrez comment configurer Seam au sein de vos projets JEE fondés sur l'IDE Eclipse Europa et Web Tools. Les bibliothèques associées à la version du framework utilisée ici figurent sur la page Web dédiée à l'ouvrage (répertoire lib).

Conçu par les équipes R&D des laboratoires JBoss et par l'un des concepteurs du framework ORM Hibernate, Gavin King, Seam est un framework pour les applications Web 2.0 qui permet d'unifier et d'intégrer les technologies suivantes :

- SOA (Service Oriented Architecture), telles qu'AJAX (Asynchronous JavaScript and XML) ;
- JSF 1.2 (avec Seam 2.0) ;
- EJB 3.0 ;
- portlets Java ;
- gestion des processus métier (jBPM) et des workflows.

Conçu avant tout pour éliminer la complexité de l'architecture et de l'interface de programmation (API), Seam permet aux développeurs d'associer des applications Web complexes à des objets Java simples (POJO) annotés, des composants graphiques Web et une petite quantité de code XML(voir plus loin la définition des objets POJO).

Pour ce faire, Seam applique le modèle de programmation par annotations avec configuration par exception d'EJB3 à toute la pile d'applications Web. Cela a pour effet de combler l'espace entre EJB 3.0 et JSF sur la plate-forme JEE5 (Java Enterprise Edition 5.0). Cette nouvelle version est un modèle d'unification étroitement intégrée, qui active les applications avec état, sans état, transactionnelles ou fondées sur des processus, telles que les workflows et les flux de pages.

La simplicité de Seam facilite l'intégration avec les ESB (Enterprise Service Bus) et bientôt JBI (Java Business Integration), une norme édictée dans la JSR-208 définissant une approche orientée composant permettant le routage des composants *via* des messages.

En résumé, JBoss Seam est un framework clé en main permettant d'affronter les nouveaux standards du Web 2.0 d'aujourd'hui et de demain.

Caractéristiques du framework Seam

Seam présente de nombreux avantages et un certain nombre de fonctionnalités innovantes, parmi lesquelles :

- **Développement fondé sur les POJO et le pattern DI (Dependency Injection).** Seam est un conteneur léger (lightweight framework) qui promeut l'usage des objets POJO (Plain Old Java Objects) comme composants service de l'infrastructure sous-jacente. Il n'est pas nécessaire d'utiliser des interfaces ou des classes abstraites à hériter pour masquer les composants de votre application. L'approche Seam se fonde sur le design pattern DI, ou injection de dépendances, qui permet de déléguer au conteneur toutes les tâches fastidieuses de gestion du cycle de vie des composants et d'injecter des services ou d'autres objets au sein d'objets POJO grâce aux annotations : vous créez simplement vos objets POJO pour modéliser votre modèle métier ou implémenter le processus métier de votre application en utilisant les données.

- **Couche de traitement à distance fondée sur AJAX.** Seam Remoting permet aux beans session EJB3 d'être invoqués directement par le client de navigation Web *via* la technologie AJAX. Les beans session apparaissent au développeur comme de simples objets JavaScript et masquent la complexité de la sérialisation en XML et de l'API XMLHttpRequest.
- **Gestion déclarative de l'état des applications.** Jusqu'à présent, les applications Web J2EE implémentaient la gestion des états manuellement, une approche générant bogues et dysfonctionnements de la mémoire lorsque les applications ne parvenaient pas à nettoyer les attributs de session. Seam supprime ces désagréments. La gestion déclarative de l'état s'appuie sur le modèle de contexte enrichi défini par Seam.
- **Modèle de programmation contextuel.** Auparavant, une session HTTP était le seul moyen de gérer les états successifs d'une application Web. Seam fournit désormais plusieurs contextes, avec états de granularité différents, depuis le niveau conversation jusqu'au niveau processus métier, libérant ainsi les développeurs des contraintes des sessions HTTP. Seam ajoute trois contextes aux nombreux déjà existants : Conversation, Process et Application. Le premier permet de définir et de gérer simplement les conversations entre requêtes ; le deuxième gère les processus afin de définir des tâches et conversations entre eux de manière similaire à JBoss jBPM ; le troisième est un contexte disponible durant toute l'application.
- **Prise en charge des applications à base de processus.** Seam intègre en toute transparence la gestion des processus métier *via* JBoss jBPM, simplifiant ainsi radicalement l'implémentation des workflows complexes et des applications de flux de pages.
- **Intégration de portails.** Seam prend en charge les portails compatibles JSR-168, tels que JBoss Portal.

Pour toutes ces raisons, Seam représente un excellent choix pour le développement d'applications Web utilisant les technologies JSF et EJB 3.0, auxquelles il apporte une parfaite synergie. Les EJB stateless et stateful deviennent ainsi des composants de gestion événementielle complémentaires de JSF. De plus, les composants JSF peuvent lier leur état aux beans entité. La généralisation de techniques telles que l'inversion de contrôle (ou IOC) et les contextes Seam libère le développeur de la gestion manuelle des relations et du cycle de vie au sein des différents composants de l'application.

La figure 8.15 illustre l'architecture JEE finale en logique 3-tiers avec le framework JBoss Seam.

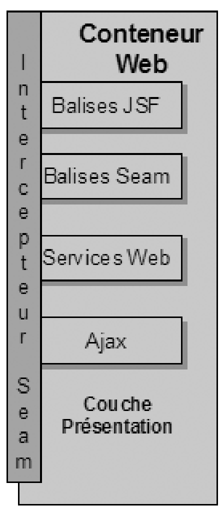
Seam permet « d'envelopper » les objets POJO, ou classes Java « simples », d'annotations Java liées à la couche présentation grâce à des intercepteurs Seam. Cela permet au développeur de se concentrer sur la logique métier, la présentation et la couche de données sans se soucier de la façon de les lier ensemble et de les faire interagir.

Précisons que le framework Seam fonctionne avec n'importe quel serveur d'applications supportant les EJB 3.0 (à l'heure ou nous écrivons ces lignes JBoss 4.2.0 GA avec le support EJB3 ou le SA compatible JEE5 Glassfish de Sun). Vous pouvez même utiliser Seam dans un conteneur de servlets comme Tomcat.

La figure 8.16 illustre les composants fondamentaux de l'architecture Seam : JSF pour la partie présentation, Hibernate (ou JDBC) pour la persistance et les composants JavaBeans pour la logique applicative.

Figure 8.15

Mise en œuvre de l'architecture 3-tiers pour les application JEE avec Seam



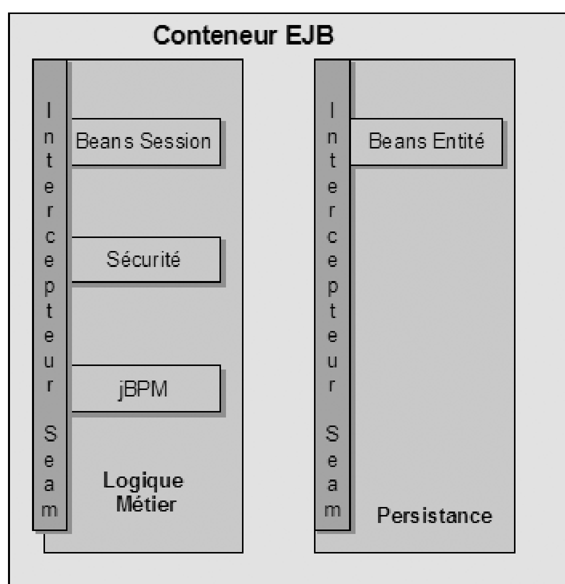
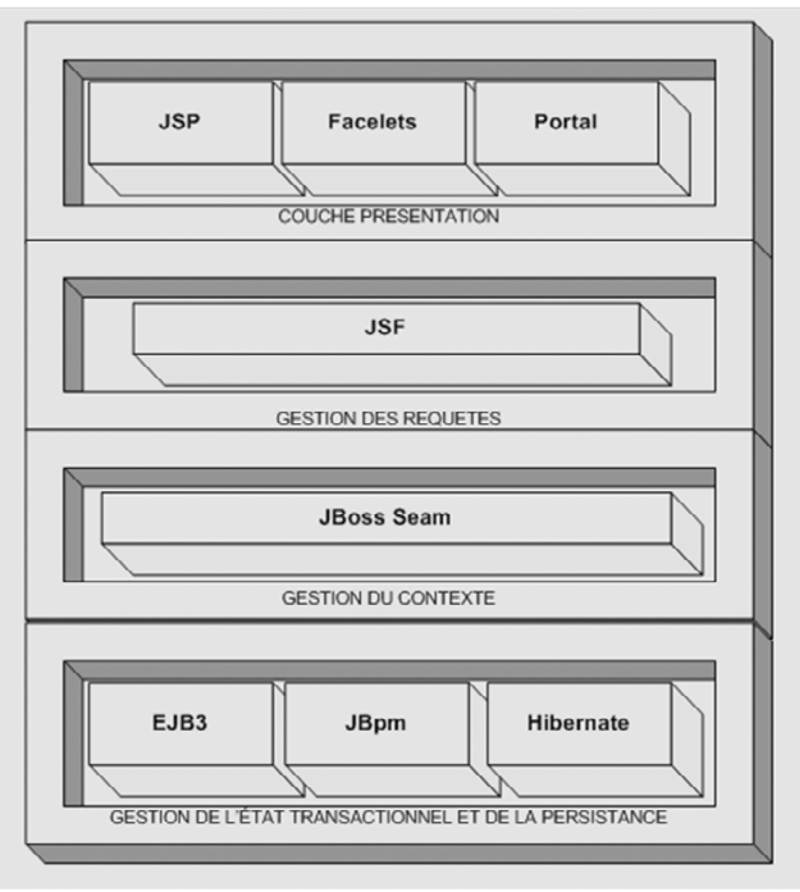


Figure 8.16

Composants de l'architecture Seam



En résumé, construire une application avec Seam est un processus relativement simple, qui ne nécessite que le codage des composants suivants :

- Composants métier, qui représentent le modèle métier. Ces objets peuvent être des beans entité utilisant l'API de persistance JPA (Java Persistence API) ou des objets POJO avec Hibernate, qui sont automatiquement mappés à des tables de la base de données.
- Pages Web JSF pour la présentation. Les champs de la page sont mappés au modèle de données *via* le langage JSF EL (Expression Lanuguage).
- Beans session EJB3 ou objets POJO annotés agissant à la façon d'un gestionnaire d'événements pour les pages Web JSF et mettant à jour le modèle en se fondant sur les données saisies par l'utilisateur.

> **POJO**
>
> POJO est l'acronyme de Plain Old Java Objects et fait référence à la simplicité d'utilisation des objets Java en comparaison de la lourdeur des composants EJB. Par définition, un objet POJO (que l'on peut traduire par objet Java), n'implémente pas d'interface spécifique à un framework particulier.

Nous aurons l'occasion de revenir sur ces concepts et de les mettre en œuvre au cours des chapitres suivants, qui traitent spécifiquement du développement EJB3 avec Seam et JBoss.

Mise en œuvre de Seam

Cette section s'efforce de préciser les étapes nécessaires à la mise en œuvre du framework Seam au sein de vos projets.

Vous devez disposer d'une version Java 5 ou ultérieure afin de pouvoir utiliser les annotations, ainsi que d'une implémentation de JSF pour la partie présentation. Cela suppose l'utilisation d'un serveur d'applications qui les supporte, comme c'est le cas des serveurs JBoss AS 4.2.x et 5.x *(bêta),* qui disposent d'une version embarquant l'implémentation de référence des spécifications JSF 1.2.

À la place d'EJB3, il est aussi possible d'utiliser le framework objet-relationnel Hibernate comme technologie de persistance.

La distribution de Seam utilisée dans cet ouvrage supportant la version du serveur JBoss 4.2 est la release 2.0, disponible sur la page Web dédiée à l'ouvrage. Nous supposons que vous l'avez téléchargée et décompressée dans un répertoire (par défaut jboss-seam-2.0.0.BETA1). À l'heure ou nous mettons sous presse, la release Seam 2.0.0 CR1 dite « stable » est disponible. Elle offre, entre autres nouvelles fonctions, une intégration poussé avec Hibernate et le support de JBoss EL (Expression Language) Enhancement, une extension du language d'expression unifié de JEE5 (voir le site officiel du projet pour plus de précisions http://labs.jboss.com/jbosseam/download/index.html).

Ant 1.6+ est également requis, tous les scripts générés par Seam s'appuyant sur cet outil de génération.

Rappels et précisions sur l'utilisation des facelets Seam

Le framework Seam offre une technologie de substitution à JSP avec le support du moteur de template Facelets. Les facelets offrent des avantages indéniables par rapport à JSP, notamment le fait de ne pas dépendre d'une balise JSP. Vous pouvez ainsi commencer à employer les nouveaux dispositifs de JSF 1.2 sans attendre la disponibilité d'une balise de JSP.

Fonctionnant avec JSF 1.1 et 1.2, les facelets facilitent la conception des pages en même temps que leur lisibilité, tout en réduisant la quantité de code nécessaire pour intégrer des composants dans la vue et en n'exigeant pas de balises Web.

Très légère, la technologie Facelet améliore les performances de JSF de 30 à 50 % en « bypassant » le moteur de JSP et en utilisant les balises xHTML directement comme composants « vue ».

Enfin, et c'est important pour le support de la technologie JSF, Facelets fournit un framework de templating pour JSF utilisant le modèle Seam d'injection de dépendances pour assembler les pages, au lieu de le faire manuellement, portion de page par portion de page à afficher (haut de page ou header, bas de page ou footer et barre de menus).

Seam pour des applications Web POJO

Seam a été conçu à l'origine pour se placer au-dessus de JEE5 afin de servir de pont entre les technologies JSF et EJB3. Toutefois, Seam n'oblige pas le développeur à fonder son architecture sur EJB3. Tout objet POJO avec une annotation @Name peut être managé par les services du conteneur. Rappelons qu'un objet POJO n'implémente pas d'interface spécifique d'un framework, à la différence d'un composant EJB.

Il est dès lors possible d'utiliser les transactions ou la persistance objet et de sécuriser votre application par le biais des frameworks Spring, Hibernate, etc., tout en ne faisant appel qu'à des objets Java classiques. L'astuce réside dans l'injection d'une dépendance au sein des paramètres du constructeur d'un objet (en particulier par l'utilisation des annotations @in et @out, par exemple, qui forment le cœur du modèle de programmation Seam).

Par exemple, vous pouvez développer l'objet POJO manager suivant au lieu d'un bean session EJB3 :

```
@Name("manager")
public class ManagerPojo {

  @In (required=false) @Out (required=false)
  private Employe employe;

  // @PersistenceContext (type=EXTENDED)
  @In (create=true)
private EntityManager em;
```

L'utilisation d'objets POJO pour remplacer les composants EJB3 ne présente toutefois pas que des avantages. Certes, les objets POJO sont beaucoup plus faciles à développer puisqu'ils ne requièrent pas d'annotations EJB3 ni d'interfaces spécifiques. De plus, si tous vos composants métier sont des objets POJO Seam, vous pouvez exécuter votre

application en dehors du conteneur du serveur d'applications EJB3 (tout conteneur Web compatible, comme Tomcat, est suffisant).

Toutefois, un objet POJO dispose de moins de fonctionnalités qu'un composant EJB3, car il ne peut pas compter sur les services du conteneur EJB3.

Citons notamment les limitations suivantes :

- L'injection @PersistenceContext ne fonctionne pas avec les objets POJO alors qu'elle est utilisée de manière transparente par le conteneur EJB3. Pour obtenir un objet EntityManager avec Seam POJO, il est nécessaire d'initialiser EntityManager dans le fichier de configuration Seam et d'utiliser l'annotation @in pour l'injecter dans l'objet POJO. Nous verrons ce point en détail dans les chapitres suivants.
- Il n'existe pas de support pour le niveau déclaratif des méthodes transactionnelles en POJO, qui reste du ressort du développeur, contrairement aux beans EJB3 (utilisation de l'annotation @TransactionAttribute). Il est donc nécessaire de configurer Seam pour démarrer une transaction base de données lorsque la requête est reçue jusqu'à l'affichage de la réponse.
- Pas de composants POJO de type MDB (message driven bean).
- Pas de support pour les méthodes asynchrones (@Asynchronous).
- Pas de support de la sécurité gérée par le conteneur.

En résumé, faire le choix de POJO complexifie la configuration à mettre en œuvre et réduit l'accès aux services offerts nativement par le conteneur.

Configuration de Seam avec les composants POJO

Vous allez construire une application POJO en utilisant les API de la spécification JPA (Java Persistence API), fondement de la norme EJB3, qui permet de gérer le cycle de vie d'un objet persistant. Précisons que cette API peut être utilisée en dehors d'un serveur d'applications, par exemple avec Tomcat en utilisant simplement Java SE.

JPA est un composant important de la spécification JEE5, et EJB3 en particulier, et constitue d'ailleurs un des apports majeurs de cette spécification qui va dans le sens d'une plus grande simplicité de mise en œuvre des EJB.

Votre application utilisera Hibernate JPA comme fournisseur JPA.

L'application permet de gérer une liste d'employés selon l'approche POJO en s'appuyant sur les API JPA. Comme vous le verrez, lorsque l'utilisateur clique sur le bouton du formulaire d'ajout d'employé, Seam crée un composant managé `employe`. Il invoque alors la méthode `updateEmploye()` sur le composant Seam managé.

Configuration du projet POJO

Pour déployer une application en dehors d'un conteneur EJB3, vous devez configurer Seam de manière qu'il puisse prendre en charge les services normalement gérés par le conteneur. Vous pouvez également utiliser le générateur Seam (seam-gen), qui permet, en sélectionnant un type de projet war, de construire la configuration d'un projet de type POJO au sein du serveur JBoss, ou simplement réutiliser un modèle existant en utilisant un modèle de fichier de construction build.xml comme celui que nous présentons ci-dessous.

Une structure de projet Seam/POJO avec JPA se présente selon la structure monApp-SeamPOJO suivante :

```
| + src
      | + Sources Java
|+  view
    |+  pages web (.xhtml), css et images
|+ resources
   |+ WEB-INF
        |+ web.xml
        | + components.xml
        | + faces-config.xml
        | + navigation.xml
        | + pages.xml
        |+ jboss-web.xml
    |+ META-INF
        |+ persistence.xml
    |+  seam.properties
|+ lib (libraires jar SEAM)
    |+ App bibliothèques spécifiques
|+ test
|+ build.xml
|+ build.properties
```

Cette structure projet Seam/POJO nécessite la personnalisation des composants suivants :

- Ajout des composants Seam et des classes nécessaires à l'application dans le répertoire src.
- Ajout des pages Web, des images et des autres ressources dans le répertoire view.
- Ajout des bibliothèques requises sous le répertoire lib (support des Facelets avec la bibliothèque jsf-facelets.jar et du framework Seam comme jboss-seam-ui.jar et jboss-seam-debug.jar).
- Ajout de bibliothèques spécifiques sous le répertoire lib pour le support, par exemple, de composants AJAX. Cela implique de modifier en conséquence le fichier build.xml si vous souhaitez packager vos jar en dehors du fichier app.jar (voir la tâche Ant pojar correspondante).
- Modification du fichier navigation.xml sous resources/WEB-INF pour définir les règles de navigation entre les pages.
- Modification du fichier resources/WEB-INF/pages.xml pour inclure les paramètres des pages de type restful.
- Modification du fichier resources/META-INF/persistence.xml pour spécifier les options personnalisées de persistance (type de framework JPA ou hibernate par exemple).
- Modification des paramètres de configuration de l'application (changement du nom du projet dans le fichier build.xml et modification du fichier resources/WEB-INF/jboss-web.xml pour le positionnement du contexte root de l'application).

Le modèle de fichier build.xml de votre application de gestion des employés avec support JPA est fourni pour examen ci-dessous (il s'appuie sur le fichier de propriété build.properties contenant la référence au serveur d'applications JBoss) :

```
<project name="MonPOJO" default="deploy" basedir=".">

 <description>Mon projet POJO</description>
 <property name="projname" value="jpa" />

 <property file="build.properties"/>
 <property name="jboss.deploy"
   location="${jboss.home}/server/default/deploy"/>

 <property name="lib" location="../lib" />
 <property name="testlib" location="../lib/embeddedejb3" />
 <property name="applib" location="lib" />
 <path id="lib.classpath">
   <fileset dir="${lib}" includes="*.jar"/>
   <fileset dir="${testlib}" includes="*.jar"/>
   <fileset dir="${applib}" includes="*.jar"/>
 </path>

 <property name="resources" location="resources" />

 <property name="src" location="src" />
 <property name="test" location="test" />
 <property name="view" location="view" />

 <property name="build.classes" location="build/classes" />
 <property name="build.jars" location="build/jars" />
 <property name="build.test" location="build/test" />
 <property name="build.testout" location="build/testout" />

 <target name="clean">
   <delete dir="build"/>
 </target>

 <target name="main" depends="compile,pojojar,war"/>

 <target name="compile">
   <mkdir dir="${build.classes}"/>
   <javac destdir="${build.classes}"
          classpathref="lib.classpath"
          debug="true">
     <src path="${src}"/>
   </javac>
 </target>

 <target name="test" depends="compile">

   <taskdef resource="testngtasks"
            classpathref="lib.classpath"/>

   <mkdir dir="${build.test}"/>

   <javac destdir="${build.test}"
          debug="true">
     <classpath>
       <path refid="lib.classpath"/>
       <pathelement location="${build.classes}"/>
     </classpath>
```

```
      <src path="${test}"/>
    </javac>

    <copy todir="${build.test}">
      <fileset dir="${build.classes}" includes="**/*.*"/>
      <fileset dir="${resources}" includes="**/*.*"/>
      <fileset dir="${testlib}/conf" includes="*.*"/>
    </copy>
    <!-- Overwrite the WEB-INF/components.xml -->
    <copy todir="${build.test}/WEB-INF" overwrite="true">
      <fileset dir="${test}" includes="components.xml"/>
    </copy>

    <testng outputdir="${build.testout}">
      <classpath refid="lib.classpath"/>
      <classpath path="${build.test}"/>
      <xmlfileset dir="${test}" includes="testng.xml"/>
    </testng>

  </target>

  <target name="pojojar" depends="compile">
    <mkdir dir="${build.jars}"/>

    <jar destfile="${build.jars}/app.jar">
      <fileset dir="${build.classes}">
        <include name="**/*.class"/>
      </fileset>
      <fileset dir="${resources}">
        <include name="seam.properties" />
      </fileset>
      <fileset dir="${applib}">
        <include name="*.jar" />
      </fileset>
      <metainf dir="${resources}/META-INF">
        <include name="persistence.xml" />
      </metainf>
    </jar>
  </target>

  <target name="war" depends="pojojar">
    <mkdir dir="${build.jars}"/>

    <war destfile="${build.jars}/${projname}.war"
         webxml="${resources}/WEB-INF/web.xml">
      <webinf dir="${resources}/WEB-INF">
        <include name="faces-config.xml" />
        <include name="components.xml" />
        <include name="navigation.xml" />
        <include name="pages.xml" />
        <include name="jboss-web.xml" />
      </webinf>
      <lib dir="${lib}">
        <include name="jboss-seam.jar" />
        <include name="jboss-seam-ui.jar" />
        <include name="jboss-seam-debug.jar" />
        <include name="jsf-facelets.jar" />
```

```
        <include name="hibernate3.jar" />
        <include name="hibernate-entitymanager.jar" />
        <include name="hibernate-annotations.jar" />
        <include name="ejb3-persistence.jar" />
      </lib>
      <lib dir="${build.jars}"
           includes="app.jar"/>
      <fileset dir="${view}"/>
    </war>
  </target>

  <target name="war421" depends="pojojar">
    <mkdir dir="${build.jars}"/>

    <war destfile="${build.jars}/${projname}.war"
         webxml="${resources}/WEB-INF/web.xml">
      <webinf dir="${resources}/WEB-INF">
        <include name="faces-config.xml" />
        <include name="components.xml" />
        <include name="navigation.xml" />
        <include name="pages.xml" />
        <include name="jboss-web.xml" />
      </webinf>
      <lib dir="${lib}">
        <include name="jboss-seam.jar" />
        <include name="jboss-seam-ui.jar" />
        <include name="jboss-seam-debug.jar" />
        <include name="jsf-facelets.jar" />
        <include name="jsf-api.jar" />
        <include name="jsf-impl.jar" />
        <include name="el-api.jar" />
        <include name="el-ri.jar" />
        <include name="hibernate3.jar" />
        <include name="hibernate-entitymanager.jar" />
        <include name="hibernate-annotations.jar" />
        <include name="ejb3-persistence.jar" />
      </lib>
      <lib dir="${build.jars}"
           includes="app.jar"/>
      <fileset dir="${view}"/>
    </war>
  </target>

  <target name="deploy">
    <copy file="${build.jars}/${projname}.war"
      todir="${jboss.deploy}"/>
  </target>

  <target name="undeploy">
    <delete
        file="${jboss.deploy}/${projname}.war"/>
  </target>
</project>
```

La structure projet complète est disponible sur la page Web dédiée à l'ouvrage (projet MonPOJO du chapitre 08). Il vous suffit de l'importer dans Eclipse et de la lancer *via* l'outil Ant intégré dans l'IDE.

Mise en œuvre du projet Seam POJO

Comme indiqué précédemment, la différence principale entre un projet à base d'EJB3 et un projet POJO réside dans la manière dont l'objet manager est utilisé.

Conception du modèle métier

La mise en œuvre d'un projet Seam nécessite la conception des objets métier et de la couche de persistance associée. Selon l'approche POJO, il s'agit d'identifier les objets métier propres à l'application, soit ici l'entité `Employe` décrite dans l'extrait ci-dessous :

```
@Entity
@name ("employe")
@Table(name="employe")
public class Employe implements Serializable {

  private long id;
  private String name;
  private long salaire;
  private String adresse;
  private String comment;
@Id @GenerateValue
  public Employe () {
    nomprenom ="";
    salaire = 0;
    adresse ="";
    comment ="";
}
```

Examinez également le détail du code de la classe ManagerPojo.java, qui permet de faire le lien entre le bean métier `Employe` et la modélisation des processus métier (gestion des employés).

Cette classe présente un certain nombre de concepts Seam importants en relation avec les annotations @in, @DataModel, @DataModelSelection et @Factory, comme le montre l'extrait suivant :

```
@Name("manager")
public class ManagerPojo {

  @In (required=false) @Out (required=false)
  private Employe employe;

  // @PersistenceContext (type=EXTENDED)
  @In (create=true)
  private EntityManager em;

  // @RequestParameter
  Int pid;

  public String updateEmploye () {
    em.persist (employe);
    return "employes";
  }

```

```
@DataModel
private List <Employe> employes;

@DataModelSelection
private Employe selectedEmploye;

@Factory("employes")
public void findEmployes () {
  employes = em.createQuery("select emp from Employe emp")
                              .getResultList();
}

public void setPid (Long pid) {
  this.pid = pid;

  if (pid != null) {
    employe = (Employe) em.find(Employe.class, pid);
  } else {
    employe = new Employe ();
  }
}

public Int getPid () {
  return pid;
}

public String delete () {
  Employe toDelete = em.merge (selectedEmploye);
  em.remove( toDelete );
  findEmployes ();
  return null;
}

public String update () {
  return "employes";
}
```

Vous devez injecter un objet EntityManager en utilisant l'annotation Seam @In. @DataModel vous permet d'utiliser une liste d'éléments sélectionnables qui transforment le composant `employes` en objet DataModel JSF.

Notez dans l'extrait de la classe ManagerPojo l'utilisation de la méthode `persist()` pour effectuer la persistance de l'entité `employe`. Lors du retour de la méthode `persist()`, l'objet `employe` devient un objet entité « managé » dans le contexte de l'objet EntityManager.

L'annotation @DataModelSelection demande à Seam quel objet doit être injecté à partir de la liste et qui sera sélectionné dans la couche de présentation. Cet objet `selectedEmploye` peut être ensuite utilisé dans la méthode de suppression `delete()` et être appelé à partir de la page de suppression.

L'annotation @Factory entre ensuite en action pour spécifier le type d'initialisation du bean `listEmployes` et répondre aux besoins de la page JSF pour l'affichage des employés (page employes.xhtml).

Le processus Seam d'édition et de mise à jour des employés de la base se déroule de la manière suivante :

1. L'utilisateur déclenche une édition de l'employé en cours *via* l'invocation de l'URL `employe.seam ?pid=1 par exemple`.
2. La page employe.xhtml est traitée, et Seam instancie le composant `listEmployes` pour afficher les données dans la page.
3. Seam injecte la valeur du pid dans l'objet ManagerPojo et appelle la méthode fabrique ManagerPojo.findEmployes pour construire et « éjecter » le composant `listEmployes`.
4. La page est affichée avec le composant `employes`.

Le tiers présentation JSF

Vous pouvez passer à la gestion de la partie présentation, qui va être chargée de l'affichage de la liste des employés (page listEmployes.xhmtl) illustrée à la figure 8.17.

Gestion des Employes WebStock

Liste des employes :

Id	Nom et Prenom	Salaire	Adresse	Comment	Action	
2	karim Seam	1500	18, Rue des Frameworks		Suppression Employe	Edition Employe
3	Pierre JBoss	2500	38, Rue des serveurs		Suppression Employe	Edition Employe

Retour Accueil

Figure 8.17

Page d'affichage de la liste des employés avec Seam et l'approche POJO

Comme vous pouvez le voir, chaque ligne du tableau contient les données de la base ainsi qu'un bouton et un hyperlien (colonne Action). Ce type de tableau est assez difficile à construire en pur JSF, car il n'existe pas de moyen naturel d'associer un identifiant d'enregistrement avec une gestion d'événements liée à un bouton d'action.

Le framework Seam permet d'implémenter ce genre de tableau activable sans grande difficulté, comme le montre l'extrait suivant de la page employes.xhtml :

```
<h:dataTable value="#{employes}" var="employe">
  <h:column>
    <f:facet name="header">Nom et Prenom</f:facet>
    #{employe.nomprenom}
  </h:column>
  <h:column>
    <f:facet name="header">Salaire</f:facet>
    #{fan.salaire}
  </h:column>
  <h:column>
    <f:facet name="header">Adresse</f:facet>
```

```
    #{ employe.adresse}
  </h:column>
  <h:column>
    <f:facet name="header">Comment</f:facet>
    #{ employe.commentaire}
  </h:column>
  <h:column>
    <f:facet name="header">Action</f:facet>
    <h:column>
    <h:commandButton value="Delete"
                     action="#{manager.delete}"/>
    </h:column>
    <h:column>
    <a href="employe.seam?pid=#{ employe.id}">Edition</a>
  </h:column>
  </h:column>


</h:dataTable>
```

Cette page affiche la liste des données de type `employe` en utilisant une balise d'affichage <h:dataTable> qui itère à partir d'un objet liste préparé par l'annotation @DataModel (voir classe ManagerPojo précédente). Chaque propriété de l'entité métier `Employe` est présentée sous forme de colonne dans un tableau. Le bouton Delete fait également partie du tableau et possède sa méthode de gestion d'événement associée (`#{manager.delete}`).

- Le tableau 8.1 récapitule les annotations associées à JSF pour la couche de présentation.

Tableau 8.1 Annotations associées à JSF

Annotation	Description
@DataModel ("nomVariable")	Représente la liste des éléments. Cette annotation permet l'utilisation d'objets `Collection` à partir de l'EJB dans la balise JSF <h:datatable> pour la partie présentation.
@DataModelSelection	Représente l'élément de la liste DataModel sélectionné.
@DataModelSelectionIndex	Injecte l'index de la ligne sélectionné de ListdataModel, MapDataModel ou ArrayDataModel. À utiliser avec l'annotation @DataModel.
@Factory	Permet l'instanciation de l'objet et le déclenchement de la méthode associée lorsque la requête initiée par la couche présentation requiert ce même objet.

Validation des données

La validation des données d'un formulaire est une fonction que tout framework se doit de supporter, et Seam ne déroge pas à la règle.

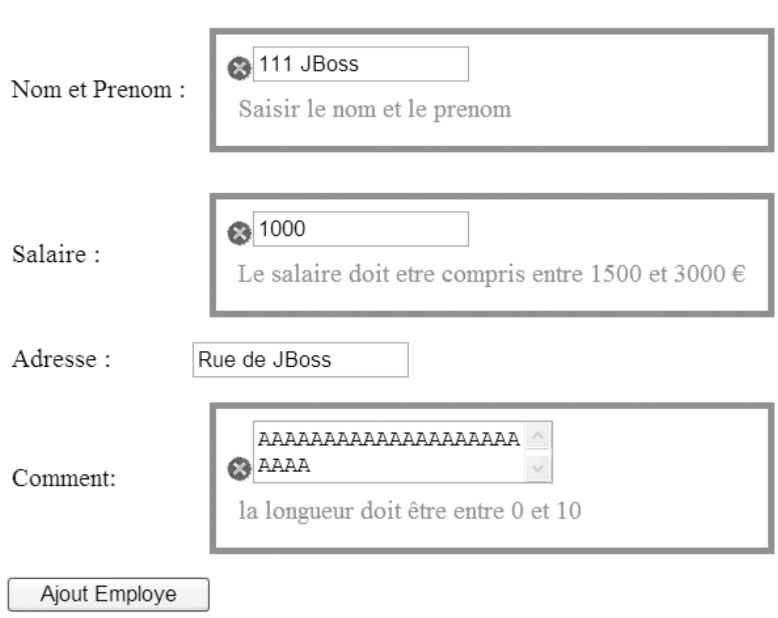
Dans votre application de gestion des employés, vous devez pouvoir valider les champs de formulaire suivants :

- Nom et Prénom : saisie uniquement en caractères alphanumériques.
- Salaire : le montant saisi doit être non nul et compris entre 1 500 et 3 500 €.
- Comment : champ libre limité à 10 caractères.

En cas d'échec de la validation, la page s'affiche à nouveau avec les champs non valides accompagnés d'un message d'erreur, comme l'illustre la figure 8.18.

Figure 8.18

Validation des champs du formulaire avec le framework Seam



Seam valide les entrées utilisateur côté serveur. Dans une approche totalement JSF, la gestion des différentes conditions de validation au sein du formulaire nécessite plusieurs lignes de code, alors qu'avec Seam, seules quelques annotations et balises JSF suffisent, comme le montre le code de la classe Person.java suivante :

```
@Id @GeneratedValue
  public long getId() { return id;}
  public void setId(long id) { this.id = id; }

  @NotNull
  @Pattern(regex="^[a-zA-Z.-]+ [a-zA-Z.-]+",
      message="Saisir le nom et le prenom")
  public String getNomPrenom() { return name; }
  public void setNomPrenom(String name) {this.name = name;}

  // @Min(value=1500) @Max(value=3500)
  @NotNull
  @Range(min=1500, max=3500,
      message="Le salaire doit etre compris entre 1500 et 3000 _")
  public long getSalaire() { return salaire; }
  public void setSalaire(long salaire) { this.salaire = salaire; }

  @NotNull
  public String getAdresse () { return adresse; }
```

```
    public void setAdresse (String adresse) {
      this.adresse = adresse;
    }

    @Length(max=10)
    public String getCommentaire() { return comment; }
    public void setCommentaire(String comment) {
      this.comment = comment;
    }

  }
```

Avec Seam, tous les formulaires sont supportés par les POJO (ou les beans entité), et les contraintes de validation sont directement annotées dans les champs des POJO, comme ici dans les champs `NomPrenom`, `Salaire`, `Adresse` et `Commentaire`.

Chaque annotation prend en argument un attribut message, qui contient le message à afficher dans le formulaire Web si les conditions de validation ne sont pas remplies. Si l'attribut message est omis, un message d'erreur par défaut est utilisé pour le champ annoté. L'attribut @Pattern peut « matcher » le champ saisi à n'importe quelle expression régulière.

Le tableau 8.2 récapitule les annotations disponibles pour la validation des champs de saisie.

Tableau 8.2 Annotations de validation des champs de saisie

Annotation	Description
@Length(max=,min=)	S'applique à un type de donnée chaîne pour vérifier que la taille de la chaîne de caractères est bien dans l'intervalle de longueur défini.
@Max(value=)	S'applique à une propriété numérique ou une représentation chaîne de caractères d'une valeur numérique afin de vérifier que la valeur de la propriété est plus basse que la valeur Max spécifée en paramètre.
@Min(value=)	S'applique à une propriété numérique ou une représentation chaîne de caractères d'une valeur numérique afin de vérifier que la valeur de la propriété est plus grande que la valeur Min spécifée en paramètre.
@NotNull	S'applique à n'importe quel type de propriété pour vérifier que cette propriété est non nulle.
@Past	S'applique à une propriété de type Date ou Calendar pour vérifier si la date est périmée.
@Future	S'applique à une propriété de type Date ou Calendar pour vérifier si la date s'applique à une date future.
@Pattern(regex="regexp", flag=)	S'applique à une propriété de type chaîne pour vérifier si la chaîne respecte l'expression régulière en paramètre. Le paramètre flag permet de valider si l'expression respecte ou non la casse.
@Range(max=,min=)	S'applique à une propriété numérique ou une représentation chaîne de caractères d'une valeur numérique afin de vérifier que la valeur de la propriété est comprise dans l'intervalle fixé.
@Size(max=,min=)	S'applique à une propriété de type collection ou tableau pour vérifier que le nombre d'éléments de la propriété est compris dans l'intervalle [min,max].
@Email	S'applique à une propriété de type chaîne pour vérifier que la valeur de la propriété est sous un format email.
@Valid	S'applique à n'importe quel type de propriété (collection, array ou map). Cette annotation effectue une validation récursive sur l'objet associé.

Même si ces annotations couvrent la majorité des besoins de validation des applications Web, Seam offre la possibilité de construire vos propres annotations de validation. Référez-vous pour cela à la documentation officielle du framework.

En résumé

Vous avez achevé l'examen de l'outillage Web avec Web Tools et découvert les fonctionnalités remarquables du framework Seam pour le développement d'applications Web complexes et riches.

Vous aborderez à la partie III de l'ouvrage l'étude et la mise en œuvre de l'outillage EJB3 avec Eclipe et JBoss.

Partie III

Développement EJB3 avec Eclipse et Web Tools

Cette partie est consacrée au développement EJB3 avec Web Tools et l'outil de modélisation et de génération de code EclipseUML for JEE de l'éditeur Omondo.

Le chapitre 9 présente les concepts fondamentaux de la spécification EJB3 dans sa version définitive et de l'API de persistance JPA, qui forme le socle de la plate-forme JEE5.

Le chapitre 10 introduit la nouvelle API de persistance JPA (Java Persistence API), un des fondements de la spécification EJB3.

Le chapitre 11 détaille la mise en œuvre de cette API dans le projet Dali, devenu partie intégrante du projet WebTools.

Le chapitre 12 est consacré à la modélisation de l'étude de cas webstock avec la solution EclipseUML for EJB3 selon une démarche pilotée par les modèles.

Le chapitre 13 conclut cette partie en présentant les aspects avancés du framework Seam appliqués au développement EJB3.

9

Les fondamentaux EJB3

Ce chapitre rappelle la genèse de la nouvelle spécification EJB3 et présente les nombreux avantages de cette norme, voulue comme une simplification radicale de l'ancienne norme EJB2, jugée trop complexe par la communauté des développeurs Java pour les développements courants.

La spécification EJB3

Apparue sous sa version finale en 2006, la spécification EJB3 dispose de nombreux atouts pour redorer un tant soit peu le blason des EJB, terni en grande partie par la complexité de l'ancienne norme EJB2.

Comme nous allons le voir, cette nouvelle mouture a bénéficié des apports de frameworks tels que Spring et Hibernate et des annotations emmenés par le JDK5 pour permettre à la communauté des développeurs J2EE/JEE de s'en emparer et de l'adapter à leurs projets.

D'EJB1 à EJB3

Attendue avec impatience par la communauté Java, EJB3, dont les premières « épreuves » ont paru en 2003, est une étape importante pour la spécification EJB, qui compte déjà plus de huit ans d'évolutions. Répondant aux souhaits des développeurs et aux best practices de développement, cette nouvelle spécification a pour premier avantage décomplexifier fortement la précédente.

Rappelons les dates clés qui ont jalonné l'évolution de la spécification :

- Mars 1998 : EJB 1.0
- Novembre 1999 : EJB 1.1
- Août 2001 : EJB 2.0
- Novembre 2003 : EJB 2.1

- 2005 : EJB 3.0
- 2006 : phase finale de la spécification EJB 3.0

Définie par la JSR-220, la spécification EJB3 comporte trois documents distincts :

- EJB3 simplified API, qui fournit une description globale du nouveau modèle de développement EJB3.
- EJB3 Core Contracts and Requirements, qui se concentre sur les beans de types session et orientés message (MDB).
- Java Persistence API, qui s'applique aux beans entité et aux frameworks de persistance.

Le nouveau modèle de composants EJB3 définit toujours les mêmes types d'objets :

- Les beans session, qui exécutent les services et opérations métier et orchestrent les transactions. Il existe deux types de bean session :
  - Les beans session stateless (sans état), qui ont une durée de vie limité et qui se limitent à l'invocation de la méthode et à la récupération de son résultat (exemple : une classe MailSender chargée de l'envoi de messages).
  - Les beans session stateful (avec état), qui permettent la conservation de l'état transactionnel (exemple : caddy virtuel).
- Les beans MDB, qui sont invoqués de manière asynchrone et répondent aux événements extérieurs par le biais d'une file de messages (exemple : un bean chargé de la commande d'articles à un centre de gestion des commandes).
- Les beans entité, qui disposent d'un unique identifiant et représentent les données persistantes du modèle métier. Ils peuvent être utilisés pour mapper une entrée dans la table de la base avec une classe Java (mapping objet-relationnel), le serveur d'applications fournissant les fonctionnalités pour charger, mettre à jour et supprimer les valeurs de l'instance de classe dans la base.

La différence fondamentale entre EJB2 et EJB3 tient à ce que les beans entité sont désormais gérés par un fournisseur de persistance particulier (TopLink, Hibernate, etc.), et non plus par le conteneur. Ils ne sont donc plus considérés comme de « vrais » beans d'entreprise. Cette volonté de séparer les services de persistance du conteneur est la réponse au reproche fait à EJB2 de ne rendre utilisables les couches implémentées avec des EJB qu'au sein de conteneurs EJB, rendant problématiques les tests unitaires. Concernant enfin la persistance, EJB3 propose un nouveau modèle de persistance, Java Persistence API, totalement inspiré d'Hibernate.

La spécification JEE5, qui intègre EJB3, prend la suite de J2EE 1.4 (notez l'abandon du « 2 » de J2EE, qui faisait référence au JDK 1.2). La version qui succède à J2SE 5.0 se nomme Java SE 6, ou JSE6. L'objectif de JEE5 est avant tout de simplifier le développement des applications Java d'entreprise. Elle intègre bon nombre de fonctionnalités EJB3, telles que les annotations, la programmation de POJO (Plain Old Java Objects), l'injection de dépendances, de nouvelles API, de nouveaux frameworks, etc.

Les points forts de cette « révolution » du modèle EJB sont détaillés à la section suivante, ainsi que dans les différents chapitres suivants.

Principales nouveautés d'EJB3

Nous ne présentons ici que les fonctionnalités marquantes de la spécification EJB3, lesquelles seront mises en œuvre tout au long des chapitres de cette partie.

Modèle simplifié grâce aux annotations

EJB3 apporte une simplification des développements grâce aux annotations, une nouvelle fonctionnalité du langage Java 5.0 en lieu et place des descripteurs de déploiement. Les annotations sont des métadonnées qui permettent à certains outils de générer des constructions additionnelles à la compilation ou à l'exécution ou encore de renforcer un comportement voulu au moment de l'exécution (comme la nature « sans état » d'un composant EJB).

Les annotations simplifient considérablement l'écriture des programmes, comme nous l'avons vu au chapitre précédent avec Seam. Les annotations permettent « d'attacher » des informations additionnelles (couramment appelées attributs) aux classes Java et aux interfaces, ainsi qu'aux méthodes et aux variables qui les composent.

Ces informations additionnelles apportées par les annotations peuvent être assimilées à des interfaces. Elles peuvent être utilisées dans un environnement de développement tel qu'Eclipse ou WebTools, voire Dali, comme nous le verrons au chapitre 11, et par les différents assistants de déploiement du conteneur JEE. Elles rendent ainsi caduques les descripteurs de déploiement de la norme EJB2 tels que ejb-jar.xml.

La figure 9.1 illustre la complémentarité entre POJO et annotations.

Figure 9.1

Transformation d'un objet POJO avec les annotations en EJB3

Descripteur de la persistance des données simplifié

Le fichier persistence.xml fournit un mécanisme puissant et simple pour la persistance des beans entité, appelée unité de persistance.

En voici un exemple simple :

```
<persistence>
<persistence-unit name="WebStockDB">
<provider>org.hibernate.ejb.HibernatePersistence</provider>
<jta-data-source>java:/WebStockDBDS</jta-data-source>
<properties>
<property name="hibernate.hbm2ddl.auto" value="update"/>
</properties>
</persistence-unit>
</persistence>
```

Ce fichier définit le fournisseur de la source de données (ici le fournisseur Hibernate, adapté au conteneur JBoss). Nous décrivons également dans ce fichier le nom de la

source de données définie par JBoss, ou nom JNDI, qui va être utilisée pour l'invoquer à travers le client à l'aide de la propriété jta-data-source.

La balise <properties> comporte des propriétés additionnelles, comme le fait de générer automatiquement le schéma de la base en fonction du bean entité fourni. La valeur "update" de la propriété "hibernate.hbm2ddl.auto" indique que nous souhaitons qu'Hibernate crée la structure de données automatiquement et la mette à jour si nécessaire (voir la documentation en ligne *http://www.hibernate.org/hib_docs/reference/en/html/tutorial.html).*

Puissant mécanisme d'injection des dépendances (DI)

Tout composant d'une application utilise un tant soit peu les services d'un autre composant pour répondre à ses besoins. Le principal objectif du concept d'injection de dépendances, ou DI (Dependency Injection), est de rendre les dépendances entre composants les plus ténus possibles.

Cela signifie que si un composant a besoin des fonctions d'un autre composant ou d'un gestionnaire de ressources, il doit pouvoir le faire à travers une interface et en utilisant un fichier de configuration adapté, au lieu d'écrire du code. Cela favorise évidemment la flexibilité de l'ensemble de l'application.

Ce concept, popularisé par le framework Spring et incarné par le principe d'Hollywood « ne nous appelez pas, nous vous appellerons », permet de décharger le développeur Java de l'instanciation des objets de l'application et de la résolution des dépendances entre eux.

La figure 9.2 illustre le schéma de principe de l'injection de dépendances. La responsabilité du conteneur est d'injecter l'objet en se fondant sur sa déclaration, à l'inverse, par exemple, d'un lookup JNDI classique, qui nécessite que le bean fasse le travail de recherche des ressources et des beans nécessaires.

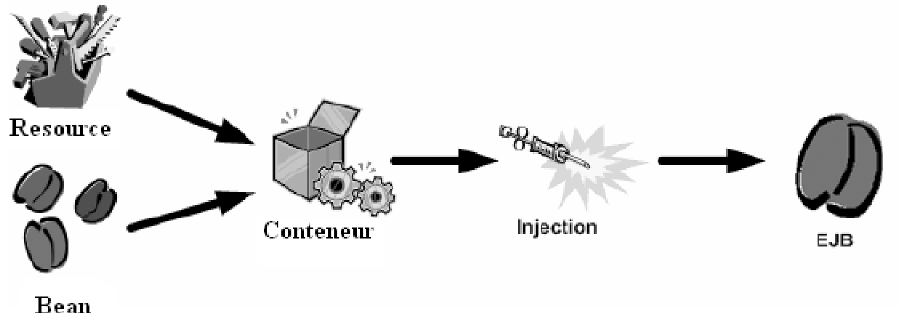


Figure 9.2

Principe du mécanisme d'injection de dépendances

L'exemple suivant illustre l'utilisation de ce principe par le biais de l'annotation javax.annotation.Resource qui s'applique aux ressources de type source de donnée, objets JMS, etc. Le bean délègue au conteneur l'instanciation du contexte monContexte

ainsi que de la datasource maBD, évitant les fastidieux lookups JNDI bien connus des développeurs EJB2 :

```
@Resource(name="maBD", type=javax.sql.DataSource)
@Stateful public class ShoppingCartBean implements ShoppingCart
{
    @Resource SessionContext monContexte;

    public Collection startToShop (String prodNom) {
        ...
        DataSource prodNom = (DataSource)monContexte.lookup("maBD");
      Connection conn = maBD.getConnection():
      ...
  }
  ...
}
```

Nouvelle API de persistance

Désormais, dans JEE5, les entités persistantes sont de simples JavaBeans, les fameux POJO, déclarés à l'aide d'une annotation @Entity. Développée sous les auspices du JSR-220, la nouvelle spécification EJB 3.0 se veut moins académique que l'ancienne et s'inspire des meilleures idées qui ont émergé dans différents contextes : le framework Open Source Hibernate 3.0, les spécifications JDO 2.0 de la JSR-243 ou des solutions commerciales comme TopLink.

En contraste avec les EJB 2.x, la nouvelle API de persistance est utilisable aussi bien dans un contexte dit « managé », c'est-à-dire au sein d'un serveur d'applications, que dans une application J2SE autonome.

Support des intercepteurs

EJB 3.0 intègre une approche POA (programmation orientée aspect) en introduisant la notion d'intercepteur : c'est une méthode qui intercepte l'invocation d'une méthode métier sur un EJB session ou MDB. EJB 3.0 propose même d'associer à un EJB session une liste de classes Interceptors dont les méthodes intercepteront les invocations aux méthodes métier de cet EJB.

Synthèse des frameworks à succès

Toutes ces nouveautés, qui vont vers une plus grande simplicité du modèle EJB, ont déjà été anticipées par des frameworks tels que Spring et Hibernate. C'est en ce sens que nous pouvons affirmer qu'un projet démarré avec la technologie EJB3 bénéficie d'assises et de ressources solides.

À l'heure d'écrire ces lignes, quelques serveurs Open Source et commerciaux supportent déjà la norme JEE5 :

- JBoss 5 (version bêta ; la version stable 4.2 supporte déjà EJB3), qui sert de cadre à cet ouvrage *(www.jboss.com)*.
- Geronimo 2.0.1, sorti en août 2007, de la fondation Apache *(http://geronimo.apache.org)*.

- Glassfish, implémentation de référence de Sun des spécifications JEE5, qui supporte les derniers standards technologiques : JSP 2.1, JSF 1 .2, Servlet 2.5EJB 3.0, JAX-WS 2.0, JAXB 2.0, Web Services Metadata for the Java Platform 1.0, … *(https://glassfish.dev.java.net/).*
- BEA WebLogic 10, premier serveur commercial supportant JEE5.

Curieusement, IBM ne semble pas pressé de sortir une version concurrente supportant EJB3. Attendons les prochaines annonces pour en savoir plus, notamment avec la version 7 du serveur WebSphere, attendue avec impatience à l'heure ou nous mettons sous presse.

Introduction aux beans session

Quel que soit le niveau de la norme EJB utilisée, les beans session sont des composants qui s'exécutent au sein d'un conteneur EJB standalone ou d'un serveur d'applications.

Ces composants sont utilisés pour représenter des cas d'utilisation ou des traitements spécifiques du client. Ils gèrent les opérations sur les données persistantes, mais non les données persistantes elles-mêmes.

Il existe deux types de beans session : les beans non persistants (stateless) et les beans persistants (stateful).

Quand utiliser des beans session ?

Cette question se pose de manière identique avec les EJB3 et les EJB2. Un bean session est utilisé pour modéliser un processus métier nécessitant le maintien d'un état conversationnel pour le client. Il permet en outre de modéliser les traitements associés, souvent avec l'aide d'un bean entité pour les opérations transactionnelles associées et la persistance.

Exemples de situation

Typiquement, un bean session est utilisé pour :

- Modéliser un traitement dont l'état ne sera pas rendu persistant et pour une durée déterminée.
- Modéliser un traitement de gestion des employés d'une société en leur affectant un service et un profil d'identification dans le système cible en liaison avec un annuaire.
- Créer un bordereau de commande pour un client du système.
- Gérer et orchestrer les méthodes dites CRUD (création, lecture, mise à jour, suppression) sur une entité du modèle métier.

Avantages

Voici quelques-uns des avantages offerts par les beans session :

- Amélioration des services du serveur d'applications en apportant des fonctionnalités évoluées de gestion transactionnelle.
- Aide aux besoins de déploiement du client lorsque que l'application cliente n'est pas localisée dans le même serveur.

- Amélioration de la sécurité fournie par le conteneur au niveau du composant et de la méthode.

> **Performances**
>
> Pour des raisons de performances, il est important de distinguer les accès locaux (composants session EJB ou Web situés dans une même application) et les accès distants (composants session entre plusieurs applications).

Beans session sans état

Les beans session non persistants sont les types d'EJB les plus simples à implémenter. Ils ne conservent aucun état de leur conversation avec les clients entre les invocations de méthodes et sont donc facilement réutilisables dans la partie serveur. Comme ils peuvent être mis en cache, ils supportent bien les variations de la demande.

Lors de l'utilisation de beans session non persistants, tous les états doivent être stockés à l'extérieur de l'EJB.

Création de beans session sans état

Pour créer un bean session sans état en utilisant la spécification EJB3, il suffit d'utiliser l'annotation @Stateless appliquée à la classe, comme dans l'exemple suivant :

```
package com.eyrolles.chap09.exempleSession ;

public interface CalculSalaire {
  public final static double tauxHoraire = 7 ;
  public double getSalaire(int nbreHeures) ;
}

package com.eyrolles.chap09.exempleSession ;
import javax.ejb.Local ;

@Local
public interface CalculSalaireLocal extends CalculSalaire {}

import javax.ejb.Stateless;

@Stateless

public class CalculSalaireBean implements CalculSalaireLocal {

  public CalculSalaireBean() {}

  public double getSalaire(int nbreHeures) {
    return nbreHeures*tauxHoraire ;
  }

}
```

La classe CalculSalaireBean (remarquez l'extension Bean après le nom de la classe, qui est une convention de nommage bien pratique) n'utilise pas d'interface particulière, comme javax.ejb.SessionBean, ni d'interface EJBHome ou EJBObject, si familières de générations de développeurs EJB2.

Nous utilisons ici une interface locale spécifiée par l'annotation @Local, CalculSalaire-Local, et le bean session sans état CalculSalaireBean qui l'implémente.

Rappelons que l'utilisation d'une interface locale précisée avec l'annotation @Local (optionnelle, mais utile) spécifie que les méthodes qui sont définies ici sont disponibles uniquement pour les clients qui ont invoqué cet EJB au sein du même conteneur JEE, cette interface locale étant équivalente à l'interface locale utilisée avec EJB2.

À l'inverse, il existe, comme en EJB2, une interface distante spécifiée par l'annotation @Remote (annotation obligatoire). Cette interface est toutefois à utiliser avec modération pour des raisons de performance, du fait du passage des paramètres par valeur et non par référence *via* RMI/IIOP entre les différentes machines virtuelles.

Voici le code correspondant à la définition d'une interface distante (Remote) qu'une simple annotation @Remote suffit à utiliser :

```
package com.eyrolles.chap09.exempleSession ;

import javax.ejb.Remote ;


@Remote
public interface CalculSalaireRemote extends CalculSalaire {}

package com.eyrolles.chap09exempleSession ;
import javax.ejb.Stateless ;

@Stateless

public class CalculSalaireBean implements CalculSalaireRemote {
  public CalculSalaireBean() {}
  public double getSalaire(int nbreHeures) {
    return nbreHeures*tauxHoraire ;
  }
}
```

Il est possible de rendre le bean CalculeSalaireBean accessible à distance et localement avec les mêmes fonctions et *via* la même interface en modifiant la signature du bean de la manière suivante :

```
@Stateless

public class CalculSalaireBean implements
CalculSalaireRemote,CalculSalaireLocal {
  public CalculSalaireBean() {}
  public double getSalaire(int nbreHeures) {
    return nbreHeures*tauxHoraire ;
  }
}
```

Si le bean n'implémente aucune interface (déconseillé), une interface métier locale est générée automatiquement par le conteneur. Elle prend le nom de la classe moins le suffixe Bean ou Impl en fonction de la convention de nommage utilisée.

L'exception levée par le conteneur RemoteException n'est plus nécessaire ici, l'accès distant étant complètement encapsulé. C'est encore un des aspects de la simplification de la norme EJB3.

Interfaces locales et distantes

Une application cliente qui accède aux beans session hébergés dans un conteneur d'applications peut être de trois types :

- Distante : les clients de l'application s'exécutent dans une JVM séparée du bean session auquel ils accèdent (voir figure 9.3). Un client distant accède au bean session à travers une interface métier distante. Un client distant peut être un autre bean, un programme Java client ou simplement une servlet. Le principal avantage de l'utilisation d'une interface distante est l'indépendance de la localisation des traitements. Cela se paye en contrepartie de moins bonnes performances en termes d'invocation de méthodes distantes et de la nécessité de sérialiser les objets et de les passer par valeur.
- Locale : les clients locaux s'exécutent dans la même JVM (voir figure 9.4). Un client local peut être un autre bean ou une application Web utilisant des servlets et des JSP/JSF. Ce type d'interface est dite *location dependant.*
- Web Service : vous pouvez publier un bean session sous forme de service Web. Ce dernier peut être invoqué à partir d'un client de service Web.

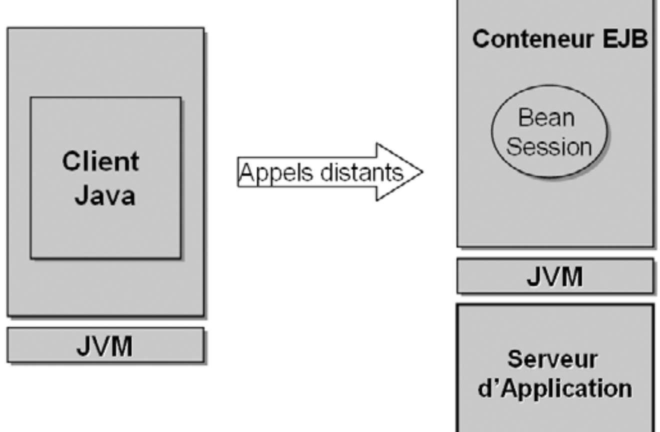


Figure 9.3

Client riche utilisant les interfaces distantes d'un bean session

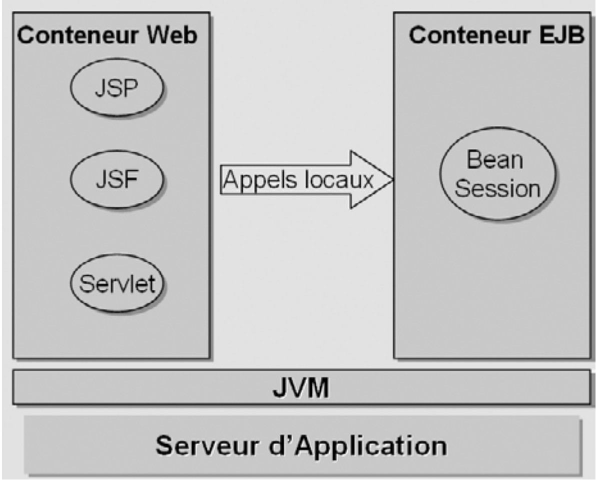


Figure 9.4

Client Web accédant aux méthodes d'un bean session via *une interface locale*

Le choix d'un type d'interface est déterminé par les contraintes de l'application. En règle générale, une application de type client Web qui s'exécute au sein de la même JVM et qui invoque les services d'un bean utilise une interface locale.

À l'inverse, un client Java riche, de type Eclipse RCP par exemple, utilise une interface distante pour invoquer ses méthodes distantes.

Selon les cas, il est conseillé d'utiliser des interfaces métier à la fois locale et distante afin que celles-ci puissent supporter les différents types de clients d'application.

Un client peut obtenir une interface métier du bean session en utilisant l'injection de dépendances ou par un lookup JNDI classique. En utilisant l'injection de dépendances avec l'annotation @EJB, vous pouvez, par exemple, obtenir une interface métier sur le bean session SearchFacade en utilisant le code suivant :

```
public class SearchFacadeClientIOC {

@EJB
SearchFacade searchFacade;

public static void main(String[] args) {

    SearchFacadeClientIOC SearchFacadeClientIOC
        ➥= new SearchFacadeClientIOC(args);
    SearchFacadeClientIOC.doTest();

}

public SearchFacadeClientIOC(String[] args) {

}

  public void doTest() {
    InitialContext ic;

    try {

      ic = new InitialContext();
      List articlesList = searchFacade.articleSearch("Disque Dur");

      for (String article: (List<String>)articlesList) {
        System.out.println(article);
      }
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }


}
```

Cycle de vie des beans et méthodes de callback

Le serveur d'applications contrôle entièrement le cycle de vie des composants EJB déployés en gardant au besoin un pool de beans sans état disponible, l'application n'ayant aucun contrôle sur l'initialisation du bean.

La méthode `create()`, bien connue des développeurs EJB2 — qui est, rappelons-le, une méthode de fabrique utilisée par les clients pour obtenir une référence à un objet EJB —, n'existe plus en EJB3. Elle est remplacée par les annotations, comme le montre l'extrait suivant :

```
import javax.ejb.Stateless;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

@Stateless
public class  CalculSalaireBean implements CalculSalaireLocal {

@PostConstruct
public void init() {
// invoque toutes methodes ou composants necessaires lors de la creation du bean
}

@PreDestroy
public void recycle() {
// invoque toutes methodes ou composants necessaires lors de la destruction du bean
}

}
```

La nouveauté dans ce mécanisme est son caractère optionnel. En EJB2, le développeur était forcé d'écrire ces méthodes, même si elles n'étaient pas implémentées par la suite. La spécification EJB3 se sert des annotations pour définir des méthodes de callback.

Une méthode de callback peut être n'importe quelle méthode définie dans un bean session qui possède une annotation de callback. Le conteneur EJB appelle cette méthode à des moments déterminés du cycle de vie du bean. Par exemple, la méthode de callback `init()` associée à l'annotation @PostConstruct intervient après toutes les injections de dépendances effectuées par le conteneur et avant le premier appel de la méthode métier. La méthode de callback associée à l'annotation @PreDestroy est appelée au moment où l'instance du bean est détruite.

Intercepteurs

La spécification EJB3 fournit des annotations particulières, appelées intercepteurs (à opposer aux méthodes de callback, même si le principe est assez proche), qui permettent d'intercepter l'invocation de méthodes métier. Une méthode intercepteur peut être définie pour un bean session ou un bean orienté message (MDB). Son utilisation peut se justifier dans des situations très particulières, dans lesquelles il est nécessaire d'effectuer certains contrôles, comme par exemple des contrôles de sécurité préliminaires avant une opération de transfert sur un compte.

La mise en œuvre d'un intercepteur s'effectue à l'aide de l'annotation @AroundInvoke appliquée à une méthode particulière. Vous pouvez également définir une classe intercepteur dont les méthodes sont appelées avant qu'une méthode métier soit invoquée sur la classe du bean.

Une classe intercepteur est définie par l'annotation @Interceptor sur la classe du bean à laquelle elle est associée. Cette annotation s'utilise lorsque plusieurs classes intercepteur sont définies.

Les méthodes intercepteur associées à l'annotation @AroundInvoke doivent posséder la signature suivante :

```
public Object <NOM_METHODE>( javax.ejb.InvocationContext ctx) throws
➥java.lang.Exception
```

Ajoutons que vous pouvez définir une méthode intercepteur au sein du bean lui-même, ou dans une classe séparée, sachant qu'il ne peut exister qu'une méthode intercepteur par classe. L'exemple qui suit montre une méthode intercepteur définie dans la classe du bean EmailSystemBean et au niveau de la classe grâce à l'annotation @Interceptors :

```
@Stateless
@Interceptors ({TracingIntercepteur.class})
@Remote(EmailSystem.class)
public class EmailSystemBean implements EmailSystem
{

 @Interceptors({AccountsConfirmIntercepteur.class})
 public void sendBookingConfirmationMessage(long orderId)
   {
      System.out.println("<Dans EmailSystemBean.sendBookingConfirmationMessage
      ➥methode metier");
      String email = "xyz@blabla.com";
      sendMessage(email, "Enregistrement Confirme", "Votre commande " + orderId
      ➥+ "est confirmee !");
      System.out.println("Sortie methode metier EmailSystemBean
      ➥.sendBookingConfirmationMessage >");
   }

@AroundInvoke
   public Object monBeanIntercepteur(InvocationContext ctx) throws Exception
   {
      if (ctx.getMethod().getName().equals("emailLostPassword"))
      {
         System.out.println("*** EmailSystemBean.monBeanIntercepteur - Nom :
         ➥" + ctx.getParameters()[0]);
      }

      return ctx.proceed();
   }

// . . .

}


public class AccountsConfirmInterceptor extends AccountsInterceptor
{
   @Resource(mappedName="java:ConnectionFactory")
   QueueConnectionFactory cf;

   @Resource(mappedName="queue/tutorial/accounts")
   Queue queue;

   @PersistenceContext
   EntityManager em;

   QueueConnection conn;
```

```
    public Object intercept(InvocationContext ctx) throws Exception
    {
       return null;
    }

 // . . .

 }
```

Support des transactions

Il peut exister dans une application un scénario dans lequel un bean session avec état cache ses modifications avant de les stocker dans la base de données. Par exemple, le bean ShoppingCartBean ne peut décider de sauvegarder dans la base les articles accumulés dans le cache de données qu'après notification du conteneur EJB, en particulier lors d'une notification afterCompletion.

Cela s'effectue par le biais de l'interface javax.ejb.SessionSynchronization, qui permet de mettre en place les trois types de notifications suivants par le conteneur :

- afterBegin : indique qu'une nouvelle transaction à été initialisée.
- beforeCompletion : indique que la transaction est sur le point d'être commitée en base.
- afterCompletion : indique que la transaction a été commitée.

L'interface SessionSynchronization est définie comme suit :

```
package javax.ejb;

public interface javax.ejb.SessionSynchronization {
    public abstract void afterBegin( ) throws RemoteException;
    public abstract void beforeCompletion( ) throws RemoteException;
    public abstract void afterCompletion(boolean committed)
        throws RemoteException;
}
```

La méthode `afterCompletion()` est toujours invoquée, que la transaction se soit bien déroulée lors d'un commit ou non lors d'un rollback. Si la transaction a bien été achevée, ce qui signifie que `beforeCompletion()` a été invoquée, le paramètre committed de la méthode `afterCompletion()` est égal à true ; si elle a échoué, il est égal à false.

Beans session avec état

Les beans session persistants, ou avec état, conservent par définition leur état entre l'invocation de leurs méthodes. Ils ont une association 1-1 avec un client et sont capables de conserver leurs états eux-mêmes.

Le conteneur d'EJB a en charge le partage et la mise en cache des beans session persistants, ceux-ci passant par la passivation et l'activation.

Création de beans session avec état

Pour créer un bean session avec état, il suffit de l'annoter avec @Stateful.

L'extrait de code suivant crée le bean avec état ShoppingCartBean, un exemple classique de caddy virtuel (à noter la propriété name pour le désigner) :

```
package com.webstock.chap09.stateful;
import javax.ejb.Remove;
```

```
import javax.ejb.Stateful;
import com.webstock.chap09.service.ShoppingCart;

@Stateful (name="ShoppingCart")
public class ShoppingCartBean implements ShoppingCart, ShoppingCartLocal {
// ...

@PostConstruct
  public void initialize() {
    cartItems = new ArrayList();
}

@PreDestroy
  public void exit() {
    System.out.println("Saved items list into database");
  }

@Remove
public void purchase() {
// persiste l'article dans la base
}
}
```

Ce code ne présente pas de difficulté particulière. L'annotation @Remove associée à la méthode de callback `purchase()` a été ajoutée afin de spécifier au conteneur quand libérer le bean pour sa réutilisation éventuelle. Lorsque cette méthode est exécutée, le bean envoie un signal au conteneur, lequel pourra le recycler ultérieurement. Il est possible d'avoir plusieurs méthodes associées à Remove, mais vous devez au moins en avoir une précédée de l'annotation @Remove.

Les beans session stateful supportent les événements de callback pour la construction, la destruction, l'activation et la passivation.

La figure 9.5 illustre les états associés à un bean session avec état.

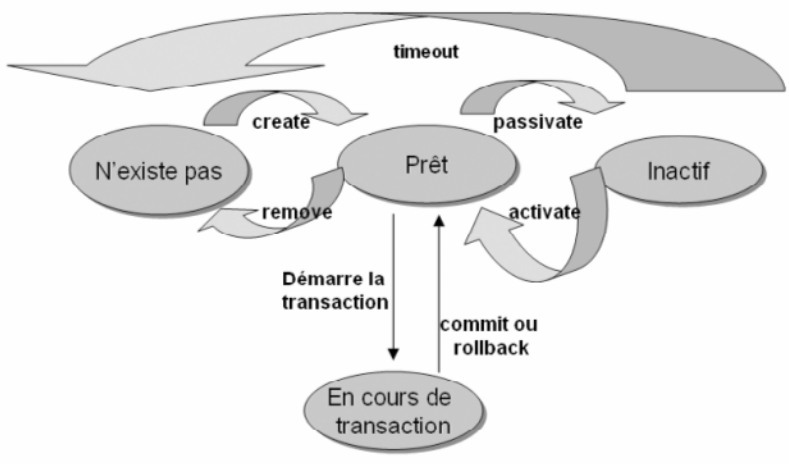


Figure 9.5

Transition des états d'un bean stateful

Événements de callback des beans session

Le tableau 9.1 récapitule les événements de callback et les annotations correspondantes des beans de type session avec ou sans état.

Tableau 9.1 Événements de callback des beans session

Événement de callback	Type de bean	Annotation	Description
Init	Stateful	@Init	Désigne les méthodes d'initialisation pour le bean session avec état. La méthode @PostConstruct est appelée après la méthode @Init.
PreConstruct	Stateless et stateful	@PreConstruct	Invoque la méthode de callback associée juste après qu'une nouvelle instance d'un bean a été créée par le conteneur.

PostConstruct	Stateless et stateful	@PostConstruct	La méthode de callback associée s'exécute lors de l'instanciation par le conteneur du bean. Exemple :
			`@PostConstruct`
			`public void initialize()`
			`{`
			`  cartItems = new ArrayList();`
			`}`
			Si le bean utilise un mécanisme d'injection pour acquérir des ressources, l'événement PostConstruct se déclenche après que l'injection s'est effectuée et avant que la première méthode du bean soit appelée.

PreDestroy	Stateless et stateful	@PreDestroy	Annote une méthode comme méthode de callback invoquée juste avant que l'instance du bean soit détruite. S'utilise pour libérer certaines ressources. Exemple :
			`@Stateful public class ShoppingCartBean`
			`implements ShoppingCart {`
			`private float total;`
			`private Vector productCodes;`
			`public int someShoppingMethod(){...};`
			`...`
			`@PreDestroy endShoppingCart() {...};`
			`}`

PreActivate	Stateful	@PreActivate	Signale que l'instance vient juste d'être réactivée par le conteneur. Son objectif est de permettre à des beans session avec état de maintenir les ressources nécessitant d'être réallouées durant l'activation de l'instance. Exemple :
			`@Stateful`
			`public class MystatefulBean {`
			`...`
			`@PreActivate`
			`   public void passivate() {`
			`   // open socket connections, ...`
			`  }`
			`}`

Tableau 9.1 Événements de callback des beans session *(suite)*

Événement de callback	Type de bean	Annotation	Description

PrePassivate	Stateful	@PrePassivate	Invoquée juste avant que le bean passe à l'état inactif et stocke son état dans le cache. Exemple :
			`@Stateful`
			`public class MystatefulBean {`
			`...`
			` @PrePassivate`
			` public void passivate() {`
			` // close socket connections, ...`
			` }`
			`}`

Remove	Stateful	@Remove	Lorsque la méthode associée à l'annotation @Remove est appelée, le conteneur supprime l'instance du bean du pool après que la méthode s'est exécutée. Exemple :
			`@Remove`
			`public void stopSession()`
			`{`
			`//Corps de la methode pouvant etre vide`
			`System.out.println("Appel de la methode stopSession avec l'annotation @Remove…");`
			`}`

AroundInvoke	Stateless et stateful	@AroundInvoke	Mise en œuvre des méthodes intercepteur.

Beans session EJB2 vs EJB3

Le tableau 9.2 récapitule les principales différences entre EJB2 et EJB3.

Tableau 9.2 Principales différences entre EJB2 et EJB3

Bean session EJB2	Bean session EJB3
Interface distante (Remote)	Interface métier (Business)
Étend l'interface EJBObject et EJBLocalObject.	Simple POJO
Interface Home (interface d'accueil)	Pas d'interface Home
Étend l'interface EJBHome et EJBLocalHome	Non opérant
Doit disposer d'au moins une méthode create.	Non opérant
Les méthodes create doivent générer l'exception CreateException et RemoteException.	Non opérant
Classe d'implémentation du bean	
Implémente l'interface SessionBean.	Implémente l'interface metier.
Doit définir l'ensemble des méthodes de l'interface SessionBean.	Simple POJO
Définit les méthodes métier pour l'interface distante.	Définit les méthodes métier pour l'interface métier.
Type de bean défini dans le fichier descripteur	Type de bean défini par les annotations @Stateless ou @Stateful
Les callbacks sont supportées tout au long du cycle de vie de l'interface SessionBean.	Les callbacks sont supportées à travers les annotations @PreDestroy, @PrePassivate, @PostActivate et @Init.

Tableau 9.2 Principales différences entre EJB2 et EJB3 *(suite)*

Bean session EJB2	Bean session EJB3
Descripteur de déploiement	
Fichier descripteur ejb-jar.xml requis	Les descripteurs de déploiempent sont optionnels. Utilisation du fichier de persistance persistence.xml requis. Fichier orm.xml optionnel.
Bean session client	
Les clients effectuent un lookup pour l'interface d'accueil en utilisant le contexte JNDI.	Utilisation de l'injection de dépendances du conteneur avec l'annotation @EJB
Pour localiser une référence à un bean, la propriété ejb-ref est utilisée.	Non nécessaire. Utilisation de l'annotation @Resource pour l'accès aux ressources

Les beans message (Message Driven Beans)

Les beans orientés message sont utilisés pour gérer des types de traitement asynchrones. Leur objectif est d'écouter des messages sur une file de messages (file d'attente, ou Topic) *via* une interface de programmation API comme JMS (Java Messaging Service).

Rappelons que les beans MDB ne font pas partie de la spécification EJB et qu'ils ont été ajoutés par la suite pour résoudre les problématiques de traitement asynchrone. Avec la spécification EJB3, la classe de bean MDB est annotée avec @MessageDriven, qui spécifie quelle file de message le MDB va surveiller (dans notre exemple queue/mdb). Si la file n'est pas définie, le conteneur JBoss la crée automatiquement au déploiement, sans besoin d'un fichier de configuration XML particulier. La classe du bean doit toutefois implémenter l'interface MessageListener, qui définit une seule méthode `onMessage()` bien connue des développeurs EJB2.

Lorsqu'un message arrive dans la file surveillée par le bean MDB, le conteneur JBoss appel la méthode `onMessage()` de la classe du bean et passe le message entrant comme paramètre de sortie. Dans l'extrait de code suivant, un exemple classique de traitement de message, la méthode `CalculatorBean.onMessage()` récupère le corps du message, le traite en récupérant les paramètres de calcul, effectue l'opération de calcul (méthode calculate) et sauvegarde le résultat dans une classe statique gestionnaire de données RecordManager afin que la page JSP puisse ensuite la traiter.

Le Timestamp `sent` dans le gestionnaire de message se sert de l'identifiant unique pour l'enregistrement de l'opération. La page JSP d'affichage correspondante récupère l'enregistrement de l'opération en se fondant sur l'identification unique du message.

```
@MessageDriven(activationConfig =
{
     @ActivationConfigProperty(propertyName="destinationType",
        propertyValue="javax.jms.Queue"),
     @ActivationConfigProperty(propertyName="destination",
        propertyValue="queue/mdb")
})
public class CalculatorBean implements MessageListener {

  public void onMessage (Message msg) {
    try {
      TextMessage tmsg = (TextMessage) msg;
```

```
        Timestamp sent =
            new Timestamp(tmsg.getLongProperty("sent"));
        StringTokenizer st =
            new StringTokenizer(tmsg.getText(), ",");

        int start = Integer.parseInt(st.nextToken());
        int end = Integer.parseInt(st.nextToken());
        double growthrate = Double.parseDouble(st.nextToken());
        double saving = Double.parseDouble(st.nextToken());
        double result =
            calculate (start, end, growthrate, saving);
            RecordManager.addRecord (sent, result);

      } catch (Exception e) {
        e.printStackTrace ();
      }
    }

    // ... ...
  }
```

Voici le détail des nouvelles annotations :

@MessageDriven : déclare un bean message à partir de sa classe.

@ActivationConfigProperty :

- propertyName : nom de la propriété.
- propertyValue : valeur de la propriété à affecter.
- EJB : injection automatique liée à un EJB (instanciation automatique).

Dans l'extrait de code ci-dessus, vous définissez la configuration du bean MDB *via* l'annotation @ActivationConfigProperty (définition du type et du nom de la file d'attente de destination).

Pour être complet, voici le code du client. La JSP calculator.jsp utilise l'API standard JMS pour obtenir la file de message cible du bean MDB en utilisant le nom de la file (queue/mdb) et envoyer ensuite le message à la file :

```
<%@ page import="com.test.mdb.*, javax.naming.*, java.text.*, javax.jms.*,
➥java.sql.Timestamp"%>

<%

  try {
     InitialContext ctx = new InitialContext();
     queue = (Queue) ctx.lookup("queue/mdb");
     QueueConnectionFactory factory =
        (QueueConnectionFactory) ctx.lookup("ConnectionFactory");
     cnn = factory.createQueueConnection();
     sess = cnn.createQueueSession(false,QueueSession.AUTO_ACKNOWLEDGE);

   } catch (Exception e) {
    e.printStackTrace ();
  }
```

```
  TextMessage msg = sess.createTextMessage(
    request.getParameter ("start") + "," +
    request.getParameter ("end") + "," +
    request.getParameter ("growthrate") + "," +
    request.getParameter ("saving")
  );

  sender = sess.createSender(queue);
  sender.send(msg);

%>
```

Mise en œuvre du développement EJB3 avec Web Tools et JBoss

Afin de mettre en pratique avec JBoss et l'outillage Web Tools les concepts et beans session abordés précédemment, vous allez créer un bean session façade générique destiné à offrir un certain nombre de services de recherche d'articles à l'utilisateur.

Ce bean permettra d'accéder aux enregistrements de la couche back-end, enregistrements qui, pour des raisons didactiques, sont codés en dur à l'intérieur du bean. Cette couche back-end pourra ensuite être transformée en beans entité, que nous détaillons au chapitre suivant.

Prérequis et configuration

Nous supposons déjà configurés une version d'Eclipse 3.3 avec le serveur JBoss 4.2 et le JDK5.

Pour l'installation de l'outillage Eclipse et des sous-projets associés Web Tools et Dali, nous vous conseillons d'utiliser une version bundlée, directement installable et préconfigurée pour un support JEE. Plusieurs éditeurs proposent ce type de version, notamment Yoxos *(http://www.yoxos.com)*.

Vous allez commencer par déclarer le serveur JBoss à Web Tools :

1. Ouvrez la page Preferences d'Eclipse, puis choisissez Server et Installed Server Runtime Environment.
2. Cliquez sur Add.
3. Dans la liste proposée, sélectionnez JBoss 4.2, puis cliquez sur Next.
4. Spécifiez les valeurs correspondant à votre environnement (JRE compatible JDK5 et répertoire d'installation du serveur JBoss 4.2, par exemple C:/jboss-4.2.1.GA).
5. Cliquez sur Finish puis sur OK pour terminer la configuration du serveur.
6. Pour configurer le serveur JBoss ainsi défini, ouvrez la vue Servers.
7. Faites un clic droit dans la vue Servers, et choisissez New Server dans le menu contextuel.
8. La boîte de dialogue de configuration du serveur JBoss s'ouvre alors. Laissez les champs Server's Host Name et server runtime à leur valeur par défaut (respectivement localhost et JBoss 4.2). Cliquez sur Next.

9. L'écran de l'assistant de configuration du serveur JBoss vous propose l'adresse IP du serveur et les ports d'écoute http, JNDI et de la configuration du serveur (respectivement 127.0.0.1, 8080,1099 et default). Cliquez sur Finish, terminant ainsi la configuration du serveur JBoss.

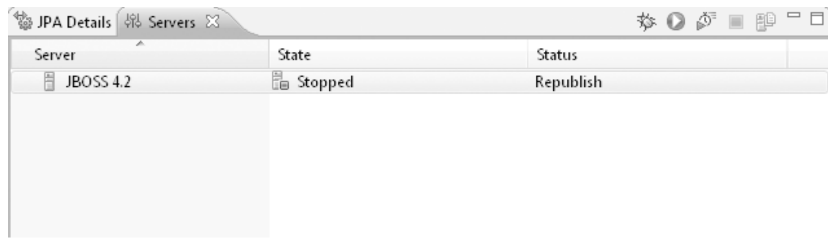


Figure 9.6

Configuration du serveur JBoss

10. Démarrez votre serveur en le sélectionnant par clic droit et en choisissant Start dans le menu contextuel.

Création et configuration du projet

Curieusement, l'outillage et les assistants Web Tools 2.0 ciblent encore les projets J2EE 1.4, comme vous pouvez le constater en créant un projet d'entreprise ou un projet EJB. C'est pour cette raison que vous devez ajouter à la main les bibliothèques de support EJB3, comme vous allez le voir.

Création du projet JEE

1. Ouvrez l'assistant de création de projet Eclipse *via* File, New et Other.
2. Sélectionnez le type de projet EJB, puis cliquez sur Next.
3. Saisissez le nom de votre projet, par exemple MonProjetEJB, et laissez les autres champs inchangés (voir figure 9.7).

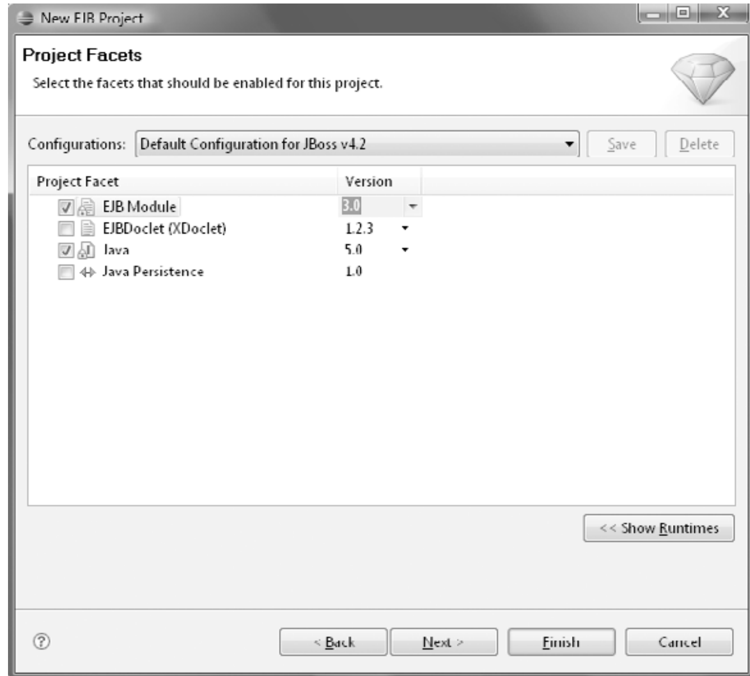
Figure 9.7

Assistant de configuration de projet EJB de Web Tools (1/2)

4. Cliquez sur Next. La boîte de dialogue illustrée à la figure 9.8 affiche les facets associées. Remarquez le support d'EJB 3.0, Java 5.0 et JPA, contrairement à la version JBoss 4.0, qui ne supporte pas nativement le conteneur EJB3. Le support de XDoclet est toujours proposé pour des raisons de compatibilité avec J2EE 1.4, mais n'est plus conseillé pour un développement JEE5.

Figure 9.8

Assistant de configuration de projet EJB de Web Tools (2/2)



5. Cliquez sur Finish. La structure de votre projet J2EE est créée.

Contrairement aux versions antérieures de JBoss, le chemin de compilation du projet fait désormais référence aux bibliothèques de support EJB3, comme ejb3-persistence.jar, jboss-ejb3.jar, jboss-ejb3x.jar.

Développement d'un bean session sans état

Le développement d'un bean session EJB3 sans état comporte les étapes clés suivantes :

1. Définition de l'interface du composant. Créez une interface sous le package com.eyrolles.chapitre09 (sous le module ejbModule déjà généré) et contenant le code suivant :

```
package com.eyrolles.chapter09;

import java.util.List;
import javax.ejb.Remote;

@Remote
public interface SearchFacade
{
  List articlesSearch(String articleType);
}
```

2. Implémentez le composant en créant la classe SearchFacadeBean. Les méthodes implémentées dans le bean doivent correspondre aux méthodes métier déclarées dans l'interface distante ou locale. Les autres méthodes qui ne possèdent pas d'équivalent au niveau de l'interface métier restent privées aux méthodes du bean.

Voici le code du bean SearchFacadeBean contenant une méthode `articleSearch()` qui a été déclarée dans l'interface métier locale et distante, celle-ci retournant une liste statique d'articles basés sur la catégorie de l'article :

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Stateless;
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

@Stateless(name = "SearchFacade")
public class SearchFacadeBean
  implements SearchFacade, SearchFacadeLocal
{
  public SearchFacadeBean() {
  }

  HashMap stockMap = new HashMap();

  public List articleSearch(String articleCategorie) {
    List articleList = new ArrayList();
    if (articleCategorie.equals("Disque Dur")) {
      articleList.add("Hitachi Deskstar 7K1000");
      wineList.add("Maxtor 200 Go Serial ATA II");
      wineList.add("Samsung SpinPoint P - SP2014N - 200Mo");
    }
    else if (articleType.equals("Cartes graphique")) {
      wineList.add("ATI Radeon 7000 - 64 Mo - AGP (ATI Radeon 7000)");
    }

    return articleList;
  }

@PostConstruct
  public void initializeStockList() {
    //countryMap is HashMap

    System.out.println("Invocation de la methode de callback initializeStockList()
    ➥sur evenement @PostConstruct");


    stockMap.put("Disque Dur", "210");
    stockMap.put("Ecrans", "510");
    stockMap.put("Claviers", "670");
    stockMap.put("Cartes graphique", "289");
  }
```

```
@PreDestroy
  public void destroyArticleList() {
    System.out.println("Invocation de la methode de callback destroyWineList()
    ➥sur evenement @PredDestroy");
    stockMap.clear();
  }

  public void afterBegin() {}

  public void beforeCompletion() {}

  public void afterCompletion (boolean b){
        // if (b==false)
          //... traitement echec transaction;
  }

// ...
}
```

Nous avons ajouté une méthode de callback PostConstruct `initializeStockList()` afin d'initialiser la liste des articles en stock lors de l'instanciation du bean. Idéalement, cette liste sera initialisée à partir d'un back-office, alimentée au besoin *via* un bean MDB connecté à une file de messages (typiquement MQ Series). Ici, nous avons simplement codé en dur pour des raisons de simplification ces valeurs dans une structure de type HashMap.

La méthode de callback PreDestroy `destroyArticleList()` survient avant que le conteneur détruise une instance du bean SearchFacade inutilisée ou expirée du pool d'objets. Ici, nous effaçons la HashMap contenant la liste des articles et des quantités associées.

Pour illustrer le mécanisme des intercepteurs, il suffit d'ajouter l'annotation @Around-Invoke au code d'implémentation du bean session SearchFacadeBean :

```
@AroundInvoke
public Object LogMethods (InvocationContext ctx) throws Exception
  {
    String beanClassName = ctx.getClass().getName();
    String businessMethodeNom = ctx.getMethod().getName();
    String target = beanClassName + "." + businessMethodName ;
    long startTime = System.currentTimeMillis();
    System.out.println ("Appel Méthode : " + target);
    try {
      return ctx.proceed();
    }
    finally {
    System.out.println("Sortie méthode " + target);
     long totalTime = System.currentTimeMillis() - startTime;
     System.out.println("Methode metier: " + businessMethodeNom +
     " dans  " + beanClassName + "prends " + totalTime + "ms pour
     s'exécuter...");
  }
}
```

La méthode d'interception LogMethods permet de tracer le temps mis par chaque méthode métier lorsque celle-ci est invoquée par le client Java. Voici sa sortie dans le fichier de log JBoss server.log :

```
2007-10-19 17:13:11,875 INFO  [STDOUT] Invocation de la méthode de callback initialize
➥StockList()  sur evenement @PostConstruct

2007-10-19 17:13:11,875 INFO  [STDOUT] Appel méthode : org.jboss.ejb3.interceptor.Invo-
➥cationContextImpl.articleSearch

2007-10-19 17:13:11,875 INFO  [STDOUT] Sortie méthode org.jboss.ejb3.interceptor.Invo-
➥cationContextImpl.articleSearch

2007-10-19 17:13:11,875 INFO  [STDOUT] Méthode métier  articleSearch dans
➥org.jboss.ejb3.interceptor.InvocationContextImpl prend  0ms pour s'exécuter...
```

Développement d'un bean session avec état

De manière similaire, vous allez développer la classe d'un bean session avec état et son interface métier destinée à conserver les articles ajoutés par un utilisateur dans son caddy virtuel.

Voici le code du bean ShoppingCartBean intégrant les méthodes métier définies (voir plus loin) dans les interfaces locales et distantes :

```
package com.eyrolles.chapitre09;

import java.util.ArrayList;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Remove;
import javax.ejb.Stateful;

@Stateful(name = "ShoppingCart")
public class

ShoppingCartBean
  implements ShoppingCartRemote, ShoppingCartLocal
{
  public ArrayList cartArticles;

  public ShoppingCartBean() {
  }

  public void addArticle(String article) {
    cartArticles.add(article);
  }

  public void removeArticle(String article) {
    cartArticles.remove(article);
  }

  public void setCartArticles(ArrayList cartArticles) {
    this. cartArticles = cartArticles;
  }
```

```
  public ArrayList getCartArticles() {
    return cartArticles;
  }

  @PostConstruct
  public void initialize() {
    cartArticles = new ArrayList();
  }

  @PreDestroy
  public void exit() {
    System.out.println("Sauvegarde de la liste des articles dans la base….");
  }

  @Remove
  public void stopSession() {
    System.out.println("Méthode stopSession dans l'annotation @Remove");
  }
}
```

Le code précédent peut aussi s'écrire de manière équivalente en utilisant les annotation @Remote et @Local :

```
import javax.ejb.Remote;
import javax.ejb.Local;
import javax.ejb.Stateful;


@Local({ShoppingCartLocal.class})
@Remote({ShoppingCart.class})

@Stateful(name = "ShoppingCart")
public class

ShoppingCartBean
  //implements ShoppingCart, ShoppingCartLocal
{


// ….

}
```

Les interfaces locale et distante sont données ci-dessous à titre d'illustration, étant entendu que l'interface locale sera utilisée pour une utilisation au sein du conteneur Web :

```
package com.eyrolles.chapitre09;

import javax.ejb.Local;
@Local
public interface ShoppingCartLocal {

  void addArticle(String wine);
  void removeArticle(String item);
  ArrayList getCartArticles();
}
```

```
package com.eyrolles.chapitre09;

import javax.ejb.Remote;
@Remote
public interface ShoppingCartRemote {

  void addArticle(String wine);
  void removeArticle(String item);
  ArrayList getCartArticles();
}
```

Le bean ShoppingCartBean supporte des méthodes de callback pour la construction, avec l'annotation @PostConstruct, d'une liste d'articles avec la méthode `initialize()`. La méthode de callback `exit()` (laissée ici à titre d'exemple, mais sans implémentation réelle) se charge de la sauvegarder après que toutes les méthodes associées à l'annotation @Remove ont été exécutées.

Test de l'EJB session

Le code de vos beans session stateless et stateful étant écrit (SearchFacade et ShoppingCart), il ne vous reste qu'à le tester par l'écriture d'une application cliente Java de type Remote.

Eclipse fournit un assistant de packaging et de déploiement des EJB relativement intuitif. Il se présente sous la forme d'une archive EAR qui sera déployée sur la configuration de déploiement du serveur JBoss (par défaut <Répertoire d'installation de JBoss>\server\default\deploy). Nous supposons que, durant toutes les étapes du packaging et du déploiement du projet, votre serveur JBoss a bien été démarré.

Vous devez déployer votre projet sur le serveur JBoss configuré *via* l'option Publish du menu contextuel Eclipse disponible dans la vue serveur, en ayant pris soin d'associer le projet à la configuration (voir figure 9.9).

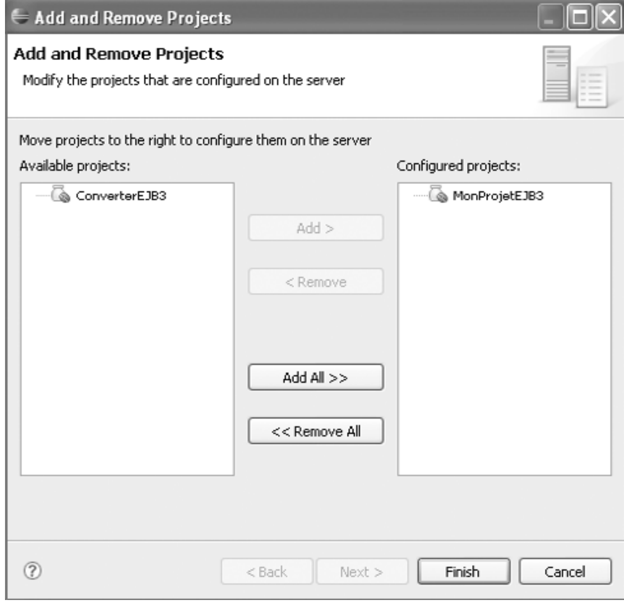


Figure 9.9

Assistant de configuration du déploiement de projet Web Tools

Votre projet est désormais entièrement recompilé et redéployé automatiquement à chaque sauvegarde de chaque ressource fichier du projet (voir figure 9.10).

```
Problems  Javadoc  Declaration  Console  Servers
<terminated> C:\Tools\Java\jdk1.5.0_13\bin\javaw.exe (21 oct. 07 18:18:14)
Buildfile: C:\Europa\eclipse\plugins\org.eclipse.jst.server.generic.jboss_1.5.105.v200709061325\buildfiles\jboss323.xml
deploy.j2ee.ejb:
      [jar] Building jar: C:\MyProjects\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\MonProjetEJB3.jar
     [move] Moving 1 file to C:\jboss-4.2.1.GA\server\default\deploy
BUILD SUCCESSFUL
Total time: 10 seconds
```

Figure 9.10

Déploiement du projet

Écriture du client Java

1. Créez un projet de type Java *via* l'assistant de création de projet Eclipse.
2. Ajoutez les bibliothèques clientes illustrées à la figure 9.11 dans le chemin de compilation des classes (y compris le jar du projet EJB3 déployé sous JBoss).

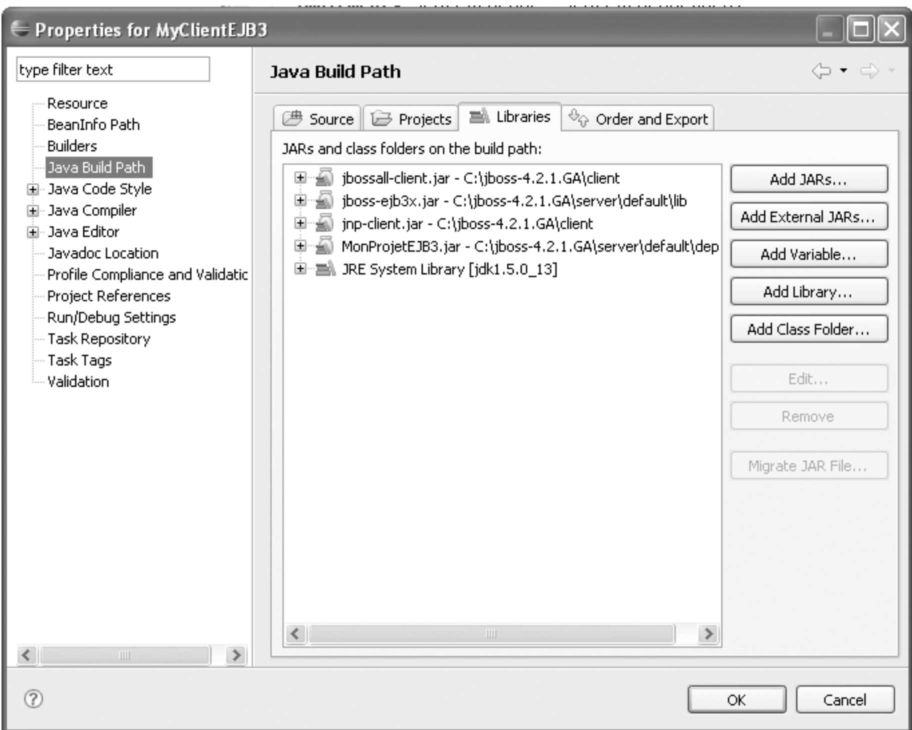


Figure 9.11

Ajout des jar nécessaires à l'application cliente

3. Dans l'option Project References, ajoutez la référence au projet EJB MonProjetEJB3.
4. Ajoutez les classes Java de type client distant suivantes (remarquez dans l'extrait l'invocation de l'interface distante SearchFacade/remote dans le lookup JNDI) :

Voici le code du client correspondant à l'invocation distante des méthodes métier du bean session sans état, l'invocation pour le bean avec état ShoppingCart se faisant de manière similaire :

```
package com.eyroles.chapitre09.client;

import com.eyrolles.chapitre09.*;

import java.util.List;

import java.util.Properties;
import javax.ejb.EJB;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class SearchFacadeClient {


public SearchFacadeClient() {

}

public static void main(String[] args) {

  try {
         Context context = new InitialContext();
         SearchFacade beanRemote = (SearchFacade)context.lookup("SearchFacade/remote");
         List winesList = beanRemote.articleSearch("Disque Dur");
         System.out.println("Affichage liste articles");
         for (String article: (List<String>)articlesList) {
             System.out.println(article);
           }

      } catch (NamingException e) {
         e.printStackTrace();
      }
   }

}
```

> **Configuration Eclipse/Web Tools/JBoss**
>
> La configuration utilisant l'injection de dépendances avec JNDI (en particulier l'utilisation de la balise @EJB) n'est apparemment pas fonctionnelle avec la configuration Eclipse/Web Tools/JBoss. Pour tester vos beans session, vous devez utiliser la configuration plus traditionnelle *via* le look-up JNDI, mais bien sûr « façon EJB3 ». Nous donnons toutefois le code utilisant l'injection de dépendances, que vous pourrez utiliser avec d'autres environnements, comme Ant.

En résumé

Ce chapitre vous a permis de découvrir les aspects innovants du développement EJB3 et les fondamentaux de ce nouveau modèle, en particulier pour le développement des beans session.

Vous verrez au chapitre suivant les aspects liés aux beans entité et à l'API Java Persistence, autre apport important de la spécification EJB3.

10

L’API JPA et la gestion des entités

Nous avons exploré au chapitre précédent les fondamentaux de l’architecture EJB3 ainsi que le modèle de développement des beans session supporté par la nouvelle spécification.

Nous présentons dans ce chapitre le modèle entité introduit par la spécification EJB 3.0, qui fait l’objet d’un document spécifique, preuve du soin apporté à ce sujet sensible entre tous qu’est la persistance des données.

La nouvelle API JPA (Java Persistence API) offre de nombreuses améliorations, qui tendent vers une plus grande simplicité que les précédentes versions du modèle EJB. Comme les beans session, les beans entité deviennent de simples POJO, découplés de leur framework de persistance. Cela permet de les utiliser à l’intérieur comme à l’extérieur de leur conteneur.

JPA (Java Persistence API)

L’API de persistance Java JPA est une spécification de Sun. Fondée sur le concept POJO pour la persistance Java, elle est relativement récente puisque sortie en même temps que JEE5, en mai 2006.

Disponible depuis les premières versions du JDK 1.5, JPA est une norme, et non une implémentation, chaque fournisseur étant libre d’implémenter son propre framework de persistance tant qu’il respecte les spécifications JPA.

JPA permet de mapper les objets POJO avec les tables de la base. Il devient dès lors possible d’utiliser JPA pour stocker les objets Java codés sans avoir à les sous-classer ou à implémenter une interface spécifique, à l’inverse de la lourdeur imposée par EJB 2.x.

La persistance traite des aspects de stockage et de récupération des données applicatives. Elle peut maintenant être programmée avec l’API de persistance Java, devenue un standard dans la spécification EJB 3.0 (JSR-220). Elle est apparue en réponse au manque de

flexibilité et à la complexité de la spécification J2EE 1.4, en particulier en ce qui concerne la persistance des beans entités.

Rappelons que c'est poussé et inspiré par les frameworks Open Source Spring, avec son mécanisme d'injection de dépendances, et Hibernate, et sa gestion de la persistance, qu'une grande partie des mécanismes de persistance EJB3 amenés par JPA ont été construits.

Un des grands atouts de l'API JPA est qu'elle est indépendante de tout fournisseur. Elle peut ainsi s'intégrer facilement à des serveurs d'applications JEE ou JSE (Tomcat).

JPA est implémentée par deux produits de référence : TopLink, un produit commercial (Oracle) devenu libre, et Hibernate.

L'architecture de JPA et son intégration dans l'architecture *n*-tiers sont illustrées à la figure 10.1. JDBC reste toujours la couche standard utilisée en Java pour l'accès aux données.

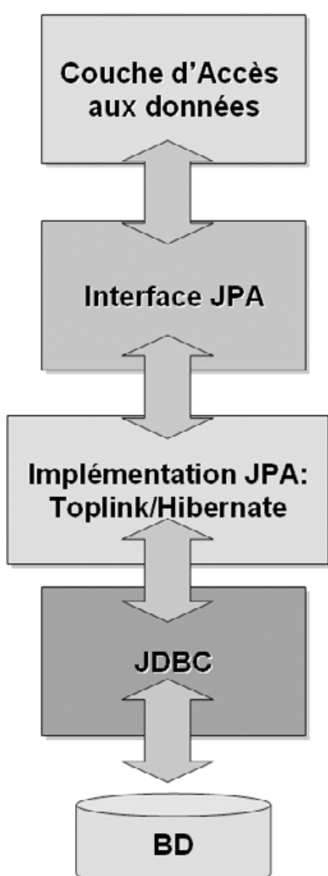


Figure 10.1

Architecture de JPA

Les aspects importants de cette nouvelle architecture sont ses relatives stabilité et standardisation. La couche d'accès aux données dialoguant avec les interfaces JPA, les développements gagnent en souplesse, puisqu'il n'est plus nécessaire de changer de modèle O/R ni de couche DAO (pour l'accès aux données) en fonction de l'outil de mapping utilisé. Quel que soit le produit qui implémente l'API, l'interface de la couche JPA reste inchangée.

Caractéristiques de JPA

Avec JPA, toute la complexité qui faisait frémir les développeurs d’applications Java appelés à développer des projets à base d’EJB est évacuée.

Ses principaux avantages sont les suivants :

- Disparition de la multitude d’interfaces (Home, Remote, Local, etc.).
- Possibilité d’utiliser JPA à l’intérieur comme à l’extérieur d’un conteneur JEE.
- Transformation des beans entité en simples POJO.
- Mapping O/R (objet-relationnel) avec les tables de la base facilitée par les annotations.

La figure 10.2 illustre le mécanisme permettant la transformation de toute classe régulière (exemple `Article`) en table correspondante, en utilisant les annotations.

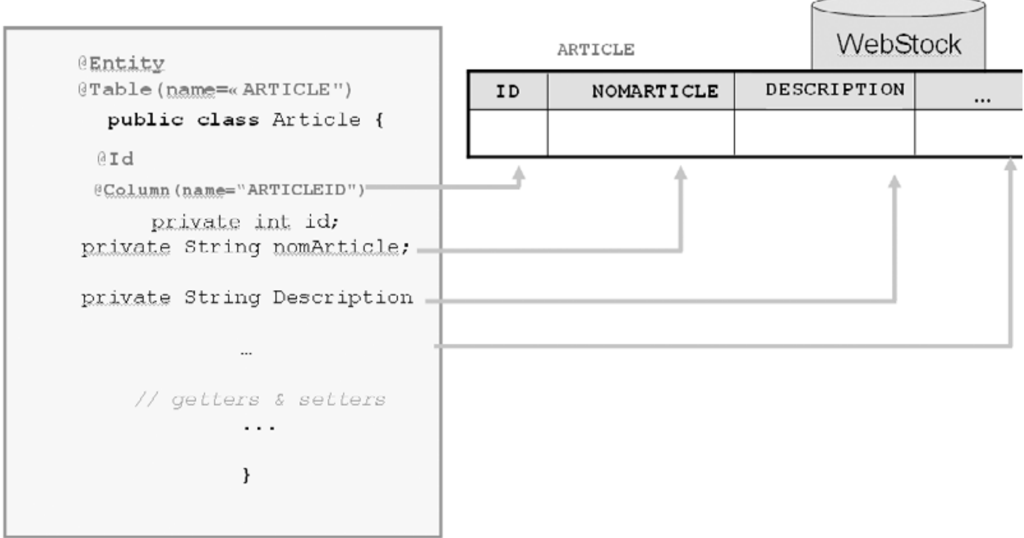


Figure 10.2

Mapping O/R avec JPA

En résumé, JPA fournit les services suivants :

- Mécanisme à la Hibernate permettant de définir déclarativement le mapping O/R et de mapper un objet à une ou plusieurs tables de la base de données grâce aux annotations de J2SE 5.0. Les annotations peuvent être utilisées pour définir des objets, des relations, du mapping O/R, de l’injection et de la persistance du contexte. JPA fournit en outre une option pour utiliser les descripteurs XML au lieu des annotations.
- API permettant de manipuler les beans entité pour effectuer les opérations CRUD de persistance, récupération et suppression des objets. Le développeur s’affranchit ainsi de toutes les tâches rébarbatives d’écriture du code de persistance des objets métier *via* JDBC et les requêtes SQL associées.

- Langage de requête standard pour la récupération des objets (JP QL, une extension de l'EJB QL d'EJB 2.x. C'est sans doute là un des aspects les plus important de la persistance des données, tant les requêtes SQL mal construites ralentissent la base de données. Cette approche affranchit les applications du langage de requête SQL propriétaire.

Les beans entité

Comme indiqué précédemment, avec EJB3 les entités deviennent des objets POJO ordinaires, expurgés de la tuyauterie si complexe d'EJB2. Ils représentent exactement le même concept que les entités de persistance Hibernate.

Dans la terminologie JPA, l'entité se réfère à un ensemble de données qui peuvent être stockées sur un support (base de données, fichier) et récupérées sous la forme d'un tout indissociable.

Une entité possède des caractéristiques spécifiques, comme la persistance ou l'identité (une entité est une instance forcément unique, identifiée grâce à sa clé) et gère les aspects liés à l'atomicité de la transaction.

Une classe entité doit respecter les prérequis suivants :

- Être annotée avec les annotations du package javax.persistance.Entity.
- Posséder un ou plusieurs constructeurs sans argument de type public ou protected.
- Ne pas être déclarée comme final. Aucune méthode ou variable d'instance de persistance ne doit être déclarée final.
- Hériter d'une autre entité ou d'une classe non-entité. Une classe non-entité peut également hériter d'une classe entité.

Les variables d'instance de persistance doivent être déclarées private ou protected et ne pas être accédées directement par les méthodes de la classe entité. Les clients doivent accéder à l'état de l'entité à travers l'invocation de méthodes accesseur.

En résumé, une entité EJB3 est un POJO ordinaire dont le mapping est défini par les annotations du JDK5 ainsi que par les annotations EJB3 et/ou tout framework JEE comme JBoss Seam par exemple.

Le mapping peut être de deux types :

- logique (associations de la classe, etc.) ;
- physique (décrivant le schéma physique, avec les tables, colonnes, index, etc.).

Annotations de persistance des beans entité

Comme nous l'avons vu, toute classe Java peut facilement être transformée en objet entité grâce aux annotations. Les spécifications EJB3 et JPA apportent un certain nombre d'annotations liées à la persistance des entités, comme le mapping à une table, les colonnes associées à des types de données simple, le mapping de clés primaires, le support de la génération automatique de l'identifiant de la clé ou la gestion des relations entre entités.

La figure 10.3 illustre les quatre entités principales du domaine webstock de notre étude de cas : `Article`, `Inventaire`, `Commande` et `Client`.

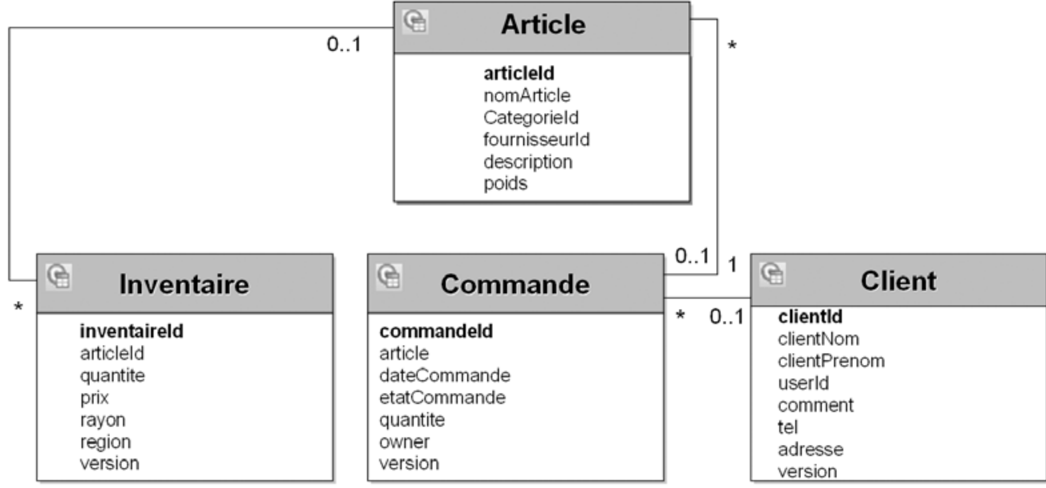


Figure 10.3

Sous-modèle objet des entités webstock

Les cardinalités des relations du modèle sont les suivantes :

- One-to-One (relation entre `Article` et `Inventaire`) : un article est relatif à une ligne d'un inventaire en magasin.
- One-to-One (relation entre `Article` et `Commande`) : chaque enregistrement d'inventaire contient un et un seul article.
- One-to-Many (relation entre `Client` et `Commande`) : spécifie qu'une entité est associée avec une collection d'autres entités. Dans notre exemple, un client peut passer plusieurs commandes (on représente d'ailleurs parfois l'extrémité de la relation, ici `Commande`, par le caractère « * », qui précise la multiplicité).
- Many-to-One (relation entre `Commande` et `Client`) : relation bidirectionnelle relativement utilisée pour spécifier que plusieurs commandes sont associées à un client.

Les cardinalités et les notions de directionnalité et de propriété de la relation (notion liée à la jointure du type de relation) sont essentielles dans la modélisation et le mapping O/. Nous y reviendrons lors du design de l'étude de cas autour de l'extrait du modèle métier webstock.

La marche à suivre pour utiliser une entité au sens JPA du terme est la suivante :

1. Création d'une classe représentant les données d'une ou plusieurs tables, soit un simple JavaBean possédant des setters et des getters, comme ici pour `Article` :

```
public classe Article {

int id ;
private String nomArticle ;
```

```
private String articleCategorieID;
private String fournisseurID;
private String description;
private long poids;


//constructeurs, getter et setter, etc …

public Article() {}
    public Article(int id) {
        this.id = id;
    }
public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
…

}
```

2. Ajout des métadonnées pour indiquer que la classe est une entité :

```
import javax.persistence.* ;

@Entity ❶

public class Article {

int id ;
private String nomArticle ;
private String articleCategorieID;
private String fournisseurID;
private String description;
private long poids;

//constructeurs, getter et setter, etc …

public Employe() {}
    public Employe(int id) {
        this.id = id;
    }
public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
…

}
```

3. Marquage de la classe avec l'annotation @Entity ❶ spécifiant au moteur de persistance que les objets créés avec cette classe peuvent utiliser le support de JPA pour les rendre persistants.

4. Ajout des métadonnées de mapping O/R sous forme d’annotation (ou de fichier XML au besoin) :

```
import javax.persistence.* ;

@Entity(name=?Article?) ❶

@Table (name= “Article”, schema= “webstock”) ❷

public class Article {

  @Id @GeneratedValue ❸

  @Column (name="ARTICLEID") ❹
  private int articleId;

  private String nomArticle ;
  private String articleCategorieID;
  private String fournisseurID;
  private String description;
  private long poids;

  public Article() {}
      public Article(int id) {
          this.id = id;
      }
  public int getId() {
          return id;
      }

      public void setId(int id) {
          this.id = id;
      }

// . . .

}
```

L’ajout de simples annotations à la classe `Article` permet de la rendre persistante.

Le tableau 10.1 récapitule l’ensemble des annotations standards appliquées aux classes entités.

Tableau 10.1 Annotations standards appliquées aux classes entités

Annotation	Description
@Entity ❶	Permet de qualifier la classe comme entité, en précisant au moteur de persistance que les instances d’objets créés à partir de cette classe peuvent utiliser le support de JPA pour être rendues persistantes. Toute entité est identifié par un nom (propriété name) et si non spécifié avec l’aide de la propriété name, prend le nom de la classe. Ce nom est utilisé par le langage JPQL (Java Persistence Query Language) pour référencer l’entité dans les requêtes. Remarque : si l’entité est passée par valeur lors de son transfert par le biais d’un bean session distant à partir de clients Java externes au conteneur EJB, le bean entité doit implémenter l’interface java.io.Serializable et utiliser le protocole RMI/IIOP à travers HTTP.
@Table ❷	Offre la possibilité de désigner la table sous-jacente à laquelle est associée cette classe, mais également de préciser le schéma de données associé.

Tableau 10.1 Annotations standards appliquées aux classes entités *(suite)*

Annotation	Description

@Id ❸	Indique au moteur de persistance quel champ de la classe (il peut en exister plusieurs) est la clé primaire ou l'identifiant de la classe. La valeur de l'identifiant du champ doit être unique par rapport à l'ensemble des instances d'entités. En complément de cette annotation, @GeneratedValue s'applique à la propriété clé primaire du champ en conjonction avec l'annotation @Id afin de permet une génération automatique de la clé selon le choix de la stratégie de génération adoptée : AUTO (par défaut), TABLE, SEQUENCE ou IDENTITY. Exemple :
	`@Entity`
	`public class Article {`
	`    @Id @GeneratedValue (strategy=GenerationType.AUTO)`
	`    private int id;`
	`}`
	Ce mode de création de l'identifiant génère automatiquement une valeur par le biais du fournisseur de persistance JPA sous-jacent (Hibernate ou TopLink) qui sera inséré dans le champs id de chaque entité Employé persistante. Précisons que le mode AUTO est à privilégier dans une phase de prototype ou de développement, les autres modes sont à privilégier en phase d'industrialisation et de production puisque s'appuyant sur le moteur de base de données sous-jacent.

@Column ❹	Permet d'indiquer que le nom de la colonne dans la base de données sera différent de celui de l'attribut défini dans la classe. Il est également possible de spécifier la taille de la colonne en nombre de caractères et de préciser si la colonne peut accepter les valeurs nulles. Exemple : `@Column(name = "PHONENUMERO", nullable = true , length=10)` Le framework de persistance fournit automatiquement la conversion des types de données des colonnes vers certains types de données Java correspondants. Le fournisseur de persistance génère un type de donnée associé à la colonne compatible lors de la génération des tables pour la classe entité lors du déploiement (les types numérique, chaîne et date sont automatiquement convertis).

@Version	Annote un attribut dont la valeur représentera un numéro de version (incrémenté à chaque modification) pour l'entité et sera utilisé pour savoir si l'état d'une entité a été modifiée entre deux moments différents (stratégie optimiste). Exemple :
	`@Entity`
	`public class Flight implements Serializable {`
	`...`
	`    @Version`
	`    @Column(name="OPTIONLOCK")`
	`    public Integer getVersion() { ... }`
	`}`

@Basic	Permet de déclarer la stratégie de récupération pour une propriété. Cette annotation influe sur les performances et la manière dont les données de l'entité sont accédées et fonctionne en combinaison avec le type fetch. Le mode FetchType permet de préciser au fournisseur de différer le chargement de l'état de l'attribut jusqu'à ce que celui-ci soit référencé. Exemple :
	`@Entity`
	`public class Article {`
	`   @Basic(fetch=FetchType.LAZY)`
	`   @Column(name= ?description?)`
	`   private String comments ;`
	`}`
	Ne pas avoir d'annotation pour une propriété est équivalent à l'annotation @Basic.

@Transient	Chaque propriété (champ ou méthode) non statique et non transiente d'un bean entité est considérée persistante, à moins que vous ne l'annotiez comme @Transient. Exemple :
	`@Transient`
	`String getLengthInMeter() { ... } // propriété transient`

Clés primaires composées

Si la clé primaire d'une entité mappe plusieurs colonnes de la table, cette clé primaire est dite composée. EJB3 propose deux annotations pour supporter cette fonctionnalité : @IdClass et @EmbeddedId.

L'annotation @IdClass

L'entité déclare chaque champ qui constitue la clé composée directement dans la classe de l'entité, annotant chacun de ceux-ci avec @Id et spécifiant la classe de la clé composée qui compose ces champs avec @IdClass, comme dans l'exemple suivant :

```
@Entity
@IdClass (ArticlePK.class)
public class Article {

@Id @GeneratedValue
@Column (name="ARTICLEID")
private int articleId;
@Id
private String nomArticle ;

public Integer getArticleId() { return articleId; }
public void setArticleId (Integer articleId) { this.articleId = articleId; }

public void setNomArticle (String nomArticle) { this.nomArticle = nomArticle; }
public String getNomArticle() { return nomArticle; }

// …
}
```

L'annotation @IdClass identifie un POJO ordinaire, c'est-à-dire sans annotations. Tous les mappings requis par les champs constituant la clé primaire sont spécifiés dans les champs de l'entité (les champs annotés par @Id dans l'entité doivent correspondre aux champs de la classe clé primaire composite) :

```
public class ArticlePK implements Serializable {
private integer articleId;
private String nomArticle

public void setArticleId (integer articleId) { this.articleid = articleId; }
public Integer getArticleId() { return articleId; }


public void setNomArticle (String nomArticle) { this.nomArticle=nomArticle; }
}
```

L'annotation @EmbeddedId

L'entité peut designer un ou plusieurs champs qui la constituent comme constituant de sa clé primaire en utilisant l'annotation @EmbeddedId en combinaison avec @Embeddable.

Toute annotation @EmbeddedId doit référencer une classe marquée @Embeddable, comme le montre l'extrait suivant d'une variante de l'entité `Article` :

```
@Entity
public class Article {
@EmbeddedId
private ArticlePK2 articleId ;
```

```
public Integer getArticleId() { return articleId ; }
public void setArticleid (Integer articleId)  { this.articleId = articleId; }

}

@Embeddable
public class ArticlePK2 {
  @Id
  @Column (name="ARTICLEID")
  private int articleId;
  String nomArticle.

  public void setArticleId (int articleId) { this.articleId =
  articleId; }
  public int getArticleId() { return articleId; }

  public void setNomArticle (String nomArticle) { this.nomArticle = nomArticle;}
 public String getNomArticle() { return nomArticle; }

}
```

Choisir @IdClass ou @EmbeddedId ?

@IdClass assure une compatibilité avec la spécification EJB2 en cas d'association de type M:N et si l'on souhaite que la clé primaire de la table association soit composée des clés étrangères vers les tables associées. Dans les autres cas, il est conseillé d'utiliser @EmbeddedId.

Relations entre beans entité

La plupart des beans entité disposent de relations entre eux ce qui produit les graphes de modèles de données si communs aux applications métier.

Dans les sections suivantes, vous allez explorer les différents types de relations qui peuvent exister entre beans entité et découvrir comment les définir et les faire correspondre avec le modèle de données sous-jacent en utilisant les annotations EJB3.

Avant de les aborder dans le détail, un bref rappel s'impose sur la notion de relation dans le mapping O/R.

Directionnalité de la relation

La relation d'une entité à une autre entité est définie par son attribut porté par la classe. Si l'autre entité pointée dispose d'un attribut qui pointe à son tour vers l'entité source, on parle de relation bidirectionnelle (entités `Client/Commande`). Si seulement une des entités de la relation possède une référence vers l'autre, la relation est dite unidirectionnelle.

Cardinalité

Un concept important dans la notion de relation entre les beans entité d'un modèle métier est la cardinalité, notion qui s'applique de manière identique au modèle relationnel et qui permet d'exprimer le nombre d'entités qui existent de part et d'autre de la relation liant les deux instances. Dans cette relation, chaque rôle possède sa propre cardinalité indiquant s'il existe une seule instance d'entité ou plusieurs.

Mapping des relations

Chacun des mappings avec la base de données est déterminé par la cardinalité de la source et le rôle de la cible.

Ce mapping s'exprime de la façon suivante :

- Many-To-One, ou plusieurs à un ;
- One-To-One, ou un à un ;
- One-to-Many, ou un à plusieurs ;
- Many-to-Many, ou plusieurs à plusieurs.

Ces noms de mappings sont aussi les noms des annotations utilisées pour indiquer le type de relation sur les attributs qui seront mappés, sachant que les mappings de relation peuvent être appliqués aux champs et aux propriétés des beans entité.

Une association entre une instance d'entité vers une autre instance d'entité dont la cardinalité de la cible est « un » est dite *monovaluée*. Les mappings de relation de type Many-to-One ou One-to-One entrent dans cette catégorie car l'entité source se réfère au plus à une entité cible.

Si l'entité source référence une ou plusieurs instances d'entité cible, l'association est dite *multivaluée*. Cela concerne les mappings One-to-Many et Many-to-Many.

Propriétés des relations et jointures

Dans une base de données, un mapping de relation entre deux tables signifie qu'une table possède une référence vers une autre table par le biais d'une colonne qui se réfère à une clé, habituellement la clé primaire de l'autre table. Cette colonne est appelée `clé étrangère`.

Transposé dans le modèle JPA, cette colonne est appelée colonne de jointure et est indiquée par l'annotation @JoinColumn.

Dans l'exemple de relation entre `Inventaire` et `Article`, la table `Inventaire` possède une clé étrangère articleId qui référence la table `Article`. Cette clé étrangère est la colonne de jointure qui associe les entités correspondantes `Article` et `Inventaire`.

Un autre concept important à prendre en compte dans la mise en œuvre d'une conception utilisant le mapping JPA est la celle « relation propriétaire », ou *owning side.* Dans une relation entre deux tables associées à deux entités, un des côtés de la relation dispose de la colonne de jointure dans la table. Ce côté est propriétaire de la relation. Le côté qui ne dispose pas de la colonne de jointure est appelé *non owning*, ou non propriétaire, ou encore côté inverse.

Les annotations qui définissent le mapping inter-entités avec les colonnes des tables de la base (en particulier @JoinColumn) sont toujours définies du côté de la relation *owning side*. Dans notre exemple de relation entre les entités `Article` et `Inventaire`, l'annotation @JoinColumn (name="articleId") signifie que la colonne articleId de l'entité source est la clé étrangère de l'entité cible, en l'occurrence `Article`. L'entité `Inventaire` est propriétaire de la relation (puisqu'elle détient la colonne de jointure).

Nom par défaut

Si aucune annotation @JoinColumn n'accompagne un mapping Many-to-One, un nom de colonne par défaut est supposé. Ce nom par défaut est formé de la combinaison des noms d'entités sources et du nom de la clé primaire de l'entité cible séparés par le caractère _.

Relations monovaluées

Mapping One-to-One

Ce type d'association implique que chaque entité est liée à une et une seule entité, et réciproquement.

Typiquement, une relation de ce type lie l'entité `Inventaire` à l'entité `Article` correspondante, comme dans les tables illustrées à la figure 10.4.

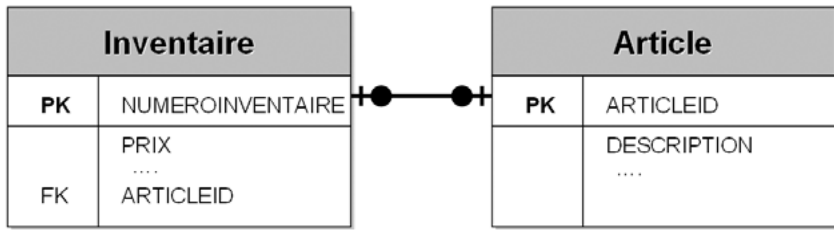


Figure 10.4

Relations One-to-One avec les tables Inventaire et Article

Ce type de mapping se décrit à l'aide de l'annotation @OneToOne. Le mapping One-to-One possédant une colonne de jointure dans la base nécessite de surcharger le nom de la colonne à l'aide de @JoinColumn lorsque le nom généré par défaut ne coïncide pas avec celui de la colonne de la table.

Dans l'extrait de code suivant, @JoinColumn est utilisée pour remplacer le nom de la colonne de jointure par défaut (soit ici `Article_userId` par `ARTICLEID`, clé étrangère de l'entité source `Inventaire`) :

```
Entity
@Table(schema="WEBSTOCK")
public class Inventaire {

@Id
@Column(table="Inventaire", name="inventaireId", insertable=false, updatable=false)
private long numInventaire;

@OneToOne
@JoinColumn(name="ARTICLEID", referencedColumnName="ARTICLEID")
protected Article article;

protected double prix;
protected int quantite;
protected String rayon;
protected String region;
protected int version;

// ….

}
```

La colonne de jointure `ARTICLEID` est déclarée avec @JoinColumn, qui possède le paramètre supplémentaire referencedColumnName. Ce paramètre sert à déclarer la colonne dans l'entité cible qui sera utilisée pour la jointure, soit `ARTICLEID`.

Mapping One-to-One bidirectionnel

Il arrive souvent qu'une entité cible d'une relation One-to-One possède une relation dite « reverse » vers l'entité source. Lorsque c'est le cas, nous appelons ce type de relation bidirectionnelle One-to-One. Elle est représentée en notation UML par une double flèche entre les deux entités.

Dans l'exemple de relation entre `Inventaire` et `Article`, le code suivant met en place ce type de relation bidirectionnelle entre les deux entités :

```
@Entity
@Table(schema="WEBSTOCK")
public class Article {

@Id
@Column(name="ARTICLEID")
@GeneratedValue
protected long articleId;

protected String nomArticle;

@oneToOne (mappedBy="article")
protected Inventaire inventaire;

//…

}
```

Nous devons dans ce cas ajouter la propriété mappedBy pour indiquer que le côté qui possède la relation est l'entité `Inventaire` et non `Article`. Nous devons préciser dans la valeur de l'attribut mappedBy le nom de l'attribut dans l'entité dite propriétaire de la relation et qui pointe à son tour sur l'entité `Article`, soit `article`. Comme l'entité `Article` n'est pas « propriétaire » de la relation, celle-ci n'a pas à fournir l'information de jointure et la colonne associée (pas de balise @JoinColumn).

Mapping Many-to-One

La figure 10.5 illustre le modèle objet correspondant à la relation de type Many-to-One unidirectionnelle entre les entités `Commande` et `Client`, relation illustrant le fait que plusieurs commandes peuvent être liées à un client. L'entité `Commande` est le côté « many » et la source de la relation, et l'entité `Client` le côté « one » et la cible.

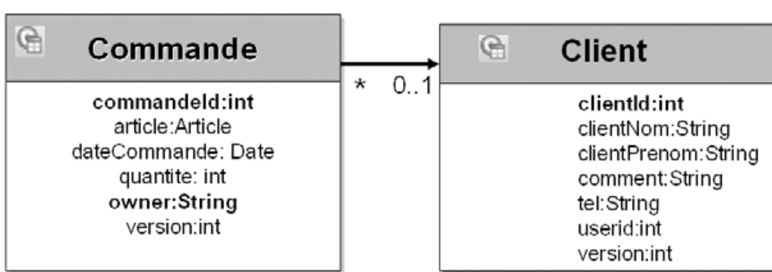


Figure 10.5

Relation Many-to-One entre Commande et Client

La classe `Commande` possède un attribut appelé owner qui va contenir la référence à une instance de `Client`. Ce type de mapping Many-to-One est défini en annotant l'entité source (l'attribut qui se réfère à l'entité cible) avec l'annotation @ManyToOne.

Dans l'extrait de code suivant, l'annotation @ManyToOne est utilisée pour mapper cette relation. Le champ owner de l'entité `Commande` est l'attribut source qui sera annoté :

```
@Entity
@Table(schema="WEBSTOCK")
public class Commande {

@Id
@Column(name="commandeId")
@GeneratedValue
protected long numCommande;
@Temporal(DATE)
protected Date dateCommande;
protected Article article;
protected int quantite;
@Column(name="ETATCOMMANDE")
protected String etat;

@ManyToOne(optional=false)
@JoinColumn(name="CLIENTID", referencedColumnName = "clientId")
protected Client owner;

 // …
}
```

La colonne de jointure est déclarée avec @JoinColumn, qui ressemble à l'annotation @Column mais possède un paramètre de plus, nommé referencedColumnName. Ce paramètre déclare la colonne dans l'entité cible qui sera utilisée pour la jointure. La clé étrangère est nommée `CLIENTID` en lieu et place de la valeur par défaut qui serait utilisée si elle n'était pas spécifiée par le paramètre name de l'annotation @JoinColumn, soit ici `CLIENT_CLIENTID` (formé de la concaténation de l'entité cible et de sa clé primaire).

Relations multivaluées

Lorsque l'entité source référence une collection d'instances de l'entité cible, une association multivaluée est utilisée. Ce type d'association concerne les mappings One-to-Many et Many-to-Many.

Mapping One-to-Many

Lorsqu'une entité est associée à une collection d'entités, ce type de mapping est de type One-to-Many. (type de cardinalité 1 à plusieurs ou 1..*).

La figure 10.6 illustre ce type de relation entre `Commande` et `Client`.

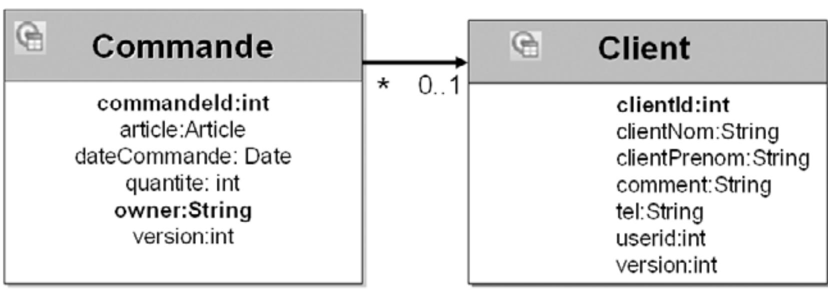


Figure 10.6

Relation One-to-Many entre Commande et Client

Dans cette portion du modèle, la clé étrangère CLIENTID de la table COMMANDE qui référence la table CLIENT correspond dans le jargon JPA à une entité Commande et à une colonne de jointure caractérisée par l'emploi de l'annotation @JoinColumn.

Dans ce type de relation, l'entité Commande « possède » la colonne, et l'entité Client dispose d'une relation One-to-Many utilisant l'annotation correspondante @OneToMany, comme dans l'extrait suivant :

```
public class Client {

@Id
@Column(table="Client")
@GeneratedValue
protected int clientId;

protected int userid;
protected String clientNom;
protected String clientPrenom;
protected String adresse;
@Column(name="TEL")
protected String telNumero;
protected String comment;
@Version
protected int version ;

@OneToMany
protected Collection <Commande> commandes;

// ….
}
```

Notez l'usage du type générique paramétré Collection pour stocker les entités de type Commande, garantissant l'existence de ce seul type dans la liste et éliminant l'usage des opérations de cast.

Mapping One-to-Many bidirectionnel

Une relation One-to-Many est souvent bidirectionnelle par nature et implique un mapping Many-to-One inverse vers l'entité source. Dans l'exemple précédent, il existe un mapping One-to-Many de l'entité Client vers Commande et un mapping retour Many-to-One de l'entité Commande vers Client.

Le code Many-to-One examiné plus haut reste inchangé dans l'exemple de relation bidirectionnelle puisque l'entité Commande « possède » la colonne de jointure CLIENTID et est propriétaire de la relation. Par contre, du côté opposé de la relation et de l'entité Client, il faut mapper la collection de commande de l'entité Commande comme relation de type One-to-Many en utilisant l'annotation @OneToMany, mais enrichie du paramètre name.

Le code correspondant à ce type de relation One-to-Many bidirectionnelle est le suivant :

```
public class Client {

@Id
@Column(table="Client")
@GeneratedValue
protected int clientId;
```

```
protected int userid;
protected String clientNom;
protected String clientPrenom;
protected String adresse;
@Column(name="TEL")
protected String telNumero;
protected String comment;
@Version
protected int version ;

@OneToMany (mappedBy="owner")
protected Collection <Commande> commandes;

// ….
}
```

En résumé, dans une relation One-to-Many bidirectionnelle :

- Le côté Many-to-One est propriétaire de la relation *(owning side)*. Par conséquent, la relation de jointure est définie sur ce côté.
- Le mapping One-to-Many est le côté inverse de la relation. L'élément mappedBy doit donc être utilisé en combinaison avec l'annotation @OneToMany. Si ce paramètre n'est pas mentionné, le fournisseur JPA la traite comme une relation unidirectionnelle de type One-to-Many utilisant une table de jointure.

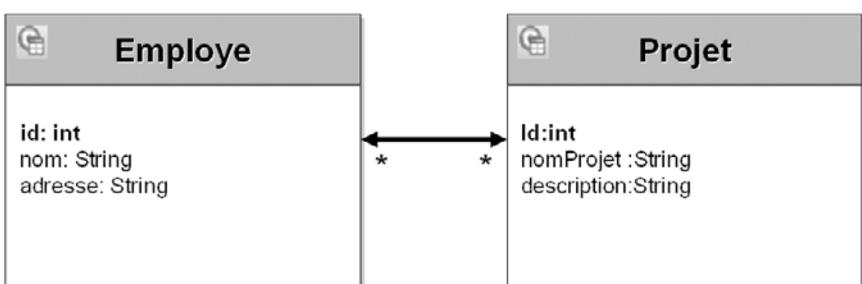
Mapping Many-to-Many

Le mapping Many-to-Many représente un type de relation entre une collection d'objets sources et une collection d'objets cibles. Ce type de mapping, moins usité que les précédentes relations, nécessite la création d'une table intermédiaire pour la gestion des associations entre les enregistrements sources et cibles.

La figure 10.7 illustre un exemple d'application de ce type de relation entre les entités `Employe` et `Projet`. Chaque employé peut travailler sur un ou plusieurs projets, et chaque projet peut à son tour être mis en œuvre par un ou plusieurs employés.

Figure 10.7

Mapping Many-to-Many bidirectionnel



Les mappings Many-to-Many utilisent une table contenant les colonnes des clés primaires pour les tables sources et destination.

Les clés primaires composites nécessitent une colonne pour chaque champ qui constitue la clé composite. Cette table doit exister avant toute utilisation de ce type de relation. La relation est aussi exprimée par l'annotation @ManyToMany pour chaque

attribut défini de part et d'autre de la relation, attribut de type collection, comme le montre l'extrait suivant :

```
@Entity
public class Employe {
    @Id private int id ;
     private String nom;
    private String adresse;
@ManyToMany
    Private Collection <Projet> projets;
//….
}

@Entity
public class Projet {
    @Id private int id ;
    private String nom;
    @ManyToMany (mappedBy="projets")
    private Collection <Employe> employes;
    //…
}
```

La principale différence avec les autres types de mappings, en particulier One-to-Many, est le fait qu'il n'existe pas de colonne de jointure de part et d'autre de la relation. De ce fait, la seule manière de mettre en œuvre ce type de mapping est la création d'une table de jointure séparée. La notion de propriétaire de la relation, notion fondamentale dans ce type de relation bidirectionnelle, doit donc être adaptée à ce contexte.

Il est nécessaire de choisir une des deux entités faisant partie de la relation comme propriétaire de la relation bidirectionnelle, l'autre devant être marquée comme « inverse ».

Dans notre exemple, nous avons choisi l'entité `Employe` mais aurions tout aussi bien pu choisir `Projet`. Comme toute relation bidirectionnelle, la relation inverse doit utiliser l'élément mappedBy pour identifier l'attribut propriétaire.

Utilisation des tables de jointure

Une table de jointure contient simplement deux clés étrangères, ou colonnes de jointure, qui permettent de référencer chacune des entités dans la relation, comme le montre la figure 10.8, qui permet de mettre en œuvre le mapping Many-to-Many.

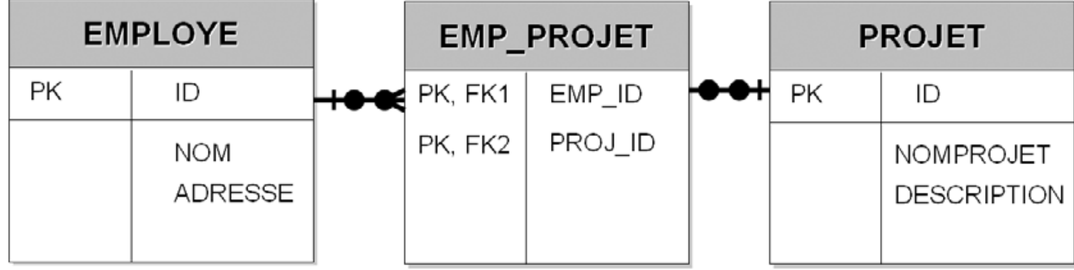


Figure 10.8

Table de jointure pour une relation Many-to-Many

Les tables EMPLOYE et PROJET ainsi que la table de jointure EMP_PROJET qui associe les entités Employe et Projet contiennent seulement les colonnes clés étrangères qui constituent la clé primaire composite. La colonne EMP_ID se réfère à la clé primaire de la table EMPLOYE, PROJ_ID, laquelle se réfère à la clé primaire de la table PROJET.

Afin de pouvoir mapper les tables ainsi décrites, il est nécessaire d'ajouter des informations de types métadonnées pour la classe Employe qui a été désignée dans notre exemple comme propriétaire de la relation.

L'extrait suivant décrit la relation Many-to-Many définie à l'aide des annotations de jointure :

```
@Entity
public class Employe {
     @Id private int id ;
      private String nom
      private String adresse;
      @ManyToMany
      @JoinTable (name="EMP_PROJET",
                      joinColumns=@JoinColumn(name="EMP_ID"),
      inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
      private Collection <Projet> projets;
     // …..
}
```

L'annotation @JoinTable est utilisée pour configurer la table de jointure pour les relations entre les entités Employe et Projet. Cette annotation définit un name, un tableau de colonnes de jointure et un tableau de colonnes de jointure inverse. Ces dernières sont les colonnes de la table d'association qui référencent la clé primaire d'Employe (l'autre extrémité de la relation).

L'interface Entity Manager

Pour assurer la persistance de l'entité dans la base, il est nécessaire d'invoquer une interface JPA spécifique, appelée EntityManager, que nous appelons dans la suite gestionnaire d'entités. Cette interface correspond à l'état d'une connexion avec la base de données et permet de gérer l'ensemble des opérations touchant aux objets @Entity du contexte de persistance.

L'ensemble des instances d'entités qui sont cachées et gérées par le gestionnaire d'entités est appelé contexte de persistance, notion fondamentale pour la compréhension du gestionnaire d'entités JPA.

Les instances d'entités requêtées à travers le gestionnaire d'entités EntityManager peuvent être invoquées à partir du client au sein (JEE) et en dehors (JSE) du conteneur EJB3, les instances pouvant être ensuite mises à jour comme des objets Java ordinaires. Pour appliquer les modifications vers la base, le client invoque la méthode merge() sur l'EntityManager au sein du contexte transactionnel. Les données de l'entité sont ensuite sauvegardées dans la base.

La figure 10.9 illustre l'environnement d'exécution d'un client JPA mettant à jours ses entités en base à l'aide d'un gestionnaire d'entités spécifique.

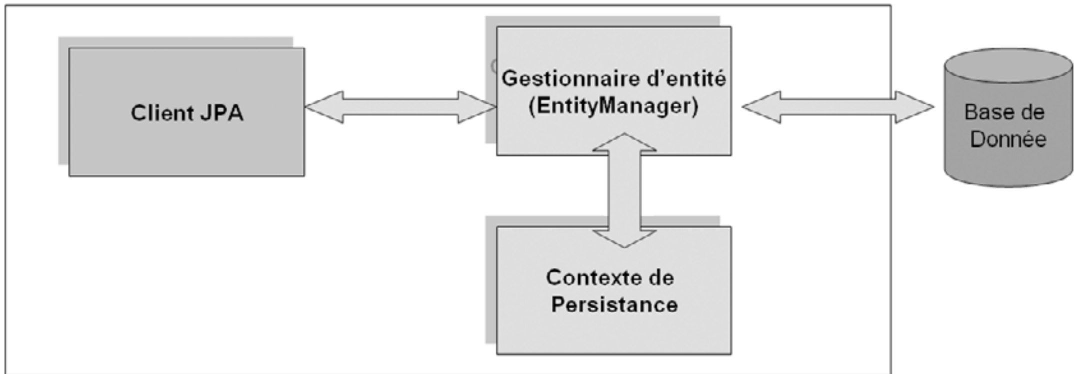


Figure 10.9

Architecture et contexte de persistance d’une application avec JPA

Types de gestionnaire d’entités

Il existe deux types de gestionnaires d’entités :

- Un gestionnaire d’entités géré par le conteneur du serveur d’applications, le contexte de persistance étant limité à une seule transaction.
- Un gestionnaire d’entités géré par l’application, c’est-à-dire en dehors du conteneur du serveur d’applications. Le contexte de persistance reste attaché au gestionnaire d’entités durant toute son existence. C’est l’application qui décide de la durée de vie du gestionnaire d’entités.

Nous aurons l’occasion en fin de chapitre de voir l’incidence sur le code de l’EntityManager, en particulier en termes d’invocation et d’utilisation à partir d’un client Java ou d’une application Web.

EntityManager et beans BMP/CMP

Les concepts qui recouvrent la notion de gestionnaire d’entités ressemblent à ceux concepts spécifiés par EJB2, à savoir les beans entité dont la persistance est gérée par le bean (BMP) et les beans dont la persistance est gérée par le conteneur (CMP). Meur mise en œuvre est cependant simplifiée, puisque cette persistance est simplement gérée par l’API JPA, sans nécessité d’interface particulière.

Méthodes de cycle de vie d’une entité EJB3

Une instance d’entité passe par plusieurs états durant sa durée d’exécution en tant qu’objet Java. Les classes d’entité EJB3 sont créées par des constructeurs simples, en lieu et place des « lourdes » interfaces fabriques familières des développeurs EJB2 Home et LocalHome. Elles peuvent ainsi être facilement transférées d’un conteneur EJB vers le client et mises à jour par ce dernier sans crainte d’une surcharge due à l’invocation d’une méthode de callback imposée par l’ancienne norme EJB.

Le tableau 10.2 récapitule les opérations prises en charge par le gestionnaire d'entités sur ses entités gérées.

Tableau 10.2 Opérations prises en charge par le gestionnaire d'entités

Opération	Description
persist()	Insère l'état d'une entité dans la base de données. Cette nouvelle entité devient alors une entité gérée.
remove()	Supprime l'état de l'entité gérée et ses données correspondantes de la base.
refresh()	Synchronise l'état de l'entité à partir de la base, les données de la BD étant copiées dans l'entité.
merge()	Synchronise les états des entités « détachées » avec le PC. La méthode retourne une entité gérée qui a la même identité dans la base que l'entité passée en paramètre, bien que ce ne soit pas le même objet.
find()	Exécute une requête simple de recherche de clé.
CreateQuery()	Crée une instance de requête en utilisant le langage JPQL.
createNamedQuery()	Crée une instance de requête spécifique.
createNativeQuery()	Crée une instance de requête SQL.
contains()	Spécifie si l'entité est managée par le PC.
flush()	Toutes les modifications effectuées sur les entités du contexte de persistance gérées par le gestionnaire d'entités sont enregistrées dans la BD lors d'un flush du gestionnaire.

Les états associés au cycle de vie d'une instance d'entité sont illustrées à la figure 10.10.

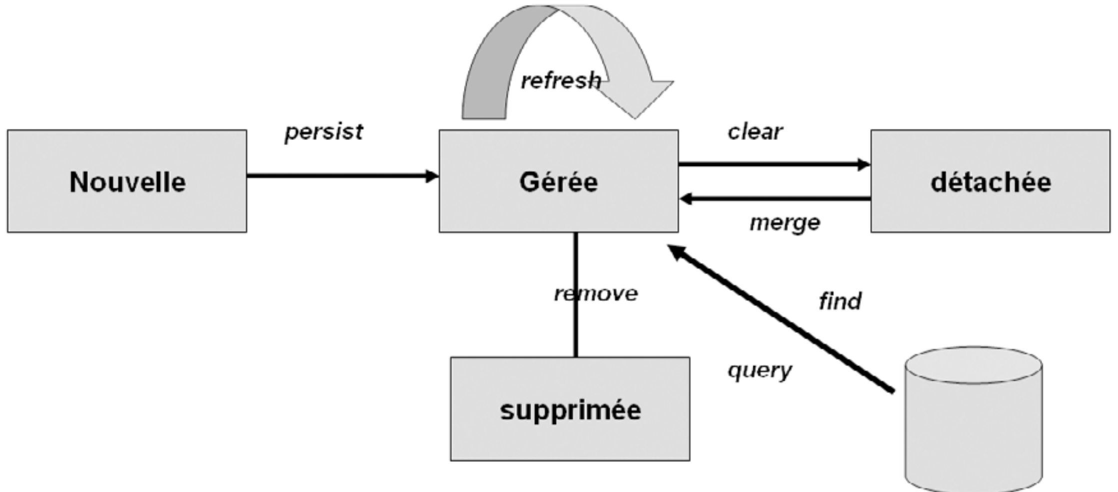


Figure 10.10

États d'une instance d'entité avec JPA

Pour illustrer l'emploi de ces méthodes, vous utiliserez par la suite l'entité `Employe` suivante (en supposant que l'ensemble des opérations s'exécute en dehors du conteneur JEE, c'est-à-dire sous le contrôle d'une JVM 5 et d'un conteneur JEE) :

```
@Entity
@Table (schema="WEBSTOCK", name="EMPLOYE")
public class Employe {
```

```
    @Id
    @Column(name="EMPLOYEID")
    private employeid;

    private String nom;
    private String adresse;
    private long salaire;
    public Employe() {}
    public Employe(int id) { this.id = id; }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getNom() { return name; }
    public void setNom(String name) { this.name = name; }
    public String getAdresse() { return adresse; }
    public void setAdresse (String adrese) {this.adresse= adresse; }
    public long getSalaire() { return salaire; }
    public void setSalaire (long salaire) {this.salaire = salaire;}

    public String toString() {
        return "Identifiant Employe: " + getEmployeId() + " nom: " + getNom()
        ➥+ " Adresse: "+ getAdresse() +  " Salaire: " + getSalaire();
    }
}
```

Les étapes de l'utilisation de cette interface *via* un client JPA sont décrites succinctement ci-après.

Obtention d'une fabrique EntityManagerFactory

Une instance de l'objet EntityManager est toujours obtenue à partir d'une interface EntityManagerFactory. Celle-ci permet la création d'un contexte de persistance :

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("EmployeService");
```

`EmployeService` est le nom de la persistance-unit, ou unité de persistance, définie dans le fichier persistence.xml.

Cette fabrique est liée à une unité de persistance précise. On se rappelle que le fichier de configuration METAINF/persistence.xml permet de définir des unités de persistance et la source de données associée.

Ce fichier imposé par la spécification JPA est créé automatiquement par la facet JPA suivante :

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
    <persistence-unit name="EmployeeService" transaction-type="RESOURCE_LOCAL">
        <class>com.webstock.chap10.employe</class>
        <properties>
            <property name="toplink.jdbc.driver"
             value="org.apache.derby.jdbc.ClientDriver"/>
             <property name="toplink.jdbc.url"
    value="jdbc:derby://localhost:1527/EmpServDB;create=true"/>
            <property name="toplink.jdbc.user" value="APP"/>
            <property name="toplink.jdbc.password" value="APP"/>
        </properties>
    </persistence-unit>
</persistence>
```

La propriété transaction-type indique que vous fonctionnerez dans un mode non-JTA. Les transactions longues et le commit à deux phases ne seront donc pas pris en charge, ce qui n'est pas un souci dans cet exemple didactique. L'élément `class` spécifie la liste des entités qui font partie de l'unité de persistante, ici la simple classe `employe`. Si vous en possédiez plusieurs, chaque entité serait référencée entre chaque balise <class>.

> **TopLink**
>
> TopLink, implémentation de référence de JPA, comporte des extensions de la norme JPA telles que la génération automatique de table (propriété toplink.ddl-generation), associée à trois valeurs possibles : none, create-tables et drop-and-create-tables. Nous ne saurions trop vous conseiller de vous référer à la documentation du fournisseur pour en connaître les caractéristiques précises.

> **Contextes de persistance**
>
> Un contexte de persistance ne peut appartenir qu'à une et une seule unité de persistance associée à un ensemble de classes entités. À l'inverse, une unité de persistance peut contenir plusieurs contextes de persistance. Il faut donc veiller à ce qu'une entité n'appartienne qu'à un seul contexte de persistance pour éviter toute incohérence dans la base. Le gestionnaire d'entités est supprimé avec la méthode `close()` de la classe EntityManager.

Création d'un EntityManager

La ligne suivante permet de créer un gestionnaire d'entités à partir d'une fabrique créée précédemment :

```
EntityManager em = emf.createEntityManager() ;
```

Avec ce gestionnaire d'entités, il est ensuite possible de travailler sur les entités persistantes du modèle logique associé.

Principales opérations disponibles sur les entités

Cette section décrit les principales opérations disponibles pour gérer les entités, en supposant l'initialisation des actions d'obtention d'une fabrique et du gestionnaire d'entités associé.

Persistance d'une entité

Cette opération consiste à prendre une entité *transiente*, c'est-à-dire dont l'état ne peut être sérialisée ou qui ne possède pas de représentation persistante dans la base de données, et à stocker son état afin qu'elle puisse être récupérée ensuite.

Voici le code correspondant à cette opération :

```
Employe emp = new Employe (10) ;
em.persist(emp);
```

Si le gestionnaire d'entités rencontre un problème en appelant la méthode `persist()`, il génère une exception PersistenceException. Autrement, l'employé est stocké dans la base de données.

Lorsque l’appel de la méthode persist() est effectuée, l’instance emp renvoyée est un bean entité managé, c’est-à-dire pris en charge au sein du contexte de persistance du gestionnaire de déploiement.

L’extrait suivant illustre l’intégration de la méthode persist() dans la méthode createEmploye() de création d’un nouvel employé :

```
public Employe createEmploye(int employeid, String employenom,
String adresse, long salaire) {
   Employe article = new Article(employeid);
   emp.setEmployenom (employenom);
   emp.setAdresse (adresse);
   emp.salaire (salaire) ;
   em.persist(emp) ;
   return emp ;
}
```

Le code suivant illustre l’utilisation de la méthode persist() appliquée au contexte JEE avec l’exemple d’un bean session acquérant une instance d’EntityManager à travers l’injection d’un contexte automatisé (@PersistenceContext) attaché à l’unité de persistance définie précédemment (EmployeService) :

```
@Stateless
public class EmployeManager {
@PersistenceContext("EmployeeService ")
private EntityManager em;
public void createEmploye() {
 final Employe emp = new Employe();
 emp.setNom("John Doe");
 emp.setAdresse("Main street");
 em.persist(emp);
}
}
```

> **Méthode *persit()***
>
> L’appel à la méthode persist() ne garantit pas qu’une instruction SQL insert soit effectuée immédiatement. Cette décision relève du gestionnaire d’entités, lequel peut décider de le faire dans la foulée ou plus tard, avant de commiter définitivement la transaction. Si les données de la base ont été modifiées (et validées) en parallèle dans la base, les données récupérées ne tiennent pas compte de ces modifications, ce qui peut provoquer des incohérences dans la base.

Recherche d’entités et requêtes JPQL

Lorsqu’une entité persiste dans la base, il est souvent nécessaire, pour les besoins de l’application, d’en récupérer les informations.

Le code correspondant à la recherche d’un employé avec un numéro donné est déconcertant de simplicité :

```
public Employe findEmploye (int employeId) {
    return em.find (Employe.class, 10);
]
```

Pour récupérer une instance d’Employe, les seules informations nécessaires à faire passer à la méthode find() sont le nom de la classe de l’entité ainsi que l’identifiant de clé primaire associée.

La méthode `find()` ne présente pas la souplesse ni la richesse d'un requête SQL, loin s'en faut. Heureusement, JPA fournit une API Query supportant l'interrogation de la base de données. Les interrogations s'effectuent à travers les entités du modèle métier et utilisent la syntaxe JPQL (Java Persistence Query Language).

L'API Query est essentielle à la jonction entre l'application et l'exécution des requêtes EJB-QL. Les méthodes de celles-ci sont regroupées dans l'interface javax.persistence.Query. L'intérêt principal de cette API est qu'elle permet de créer des requêtes dynamiques sous la forme de simples chaînes de caractères, et non de manière statique, comme c'est le cas au sein d'un descripteur de déploiement ou avec les annotations.

Une requête dynamique est une requête dont les clauses sont fournies à l'exécution, à l'inverse des requêtes nommées, abordées au chapitre suivant, où l'utilisateur doit fournir les critères de la requête ainsi que son nom avant toute utilisation.

Les principales méthodes de cette API sont regroupées dans l'interface EntityManager :

```
public interface EntityManager {
   public Query createQuery(String requeteEJB-QL);
   public Query createNamedQuery(String requeteNommee);
   public Query createNativeQuery(String requeteSQL);
   public Query createNativeQuery(String requeteSQL, Class résultat);
   public Query createNativeQuery(String requeteSQL, String résultat);

}
```

Vous vous intéresserez ici à la première méthode, `CreateQuery()`, qui permet de créer de requêtes dynamiques. Ces requêtes sont moins performantes que les requêtes nommées, car elles ne sont pas précompilées avant leur exécution par le moteur JPA.

L'exemple simple suivant montre comment l'objet Query est créé avec l'interface EntityManager et comment la chaîne JPQL est transmise avec les arguments de la requête exécutée pour avoir la liste des employés de la base :

```
Query query = em.createQuery ("select emp from Employe emp");
Collection emps = query.getResultList();
```

La chaîne de la requête se réfère à l'entité `Employe` et non à la table `Employe` de la base de données. Elle affiche la liste de tous les objets `Employe` associés. L'exécution proprement dite de la requête s'effectue grâce à l'invocation de la méthode `getResultList()`, qui retourne un objet `List` contenant la liste des employés satisfaisant au critère de la requête.

Le code de la méthode `findAllEmployes()` associée retourne un objet `List`, auquel nous appliquons un cast pour le transformer en objet `Collection`, beaucoup plus facile à utiliser :

```
public Collection<Employe> findAllEmployes() {
     Query query = em.createQuery ("select emp from Employe emp") ;
     return (Collection<Employe>) query.getResulList():
}
```

L'exemple de requête suivant est un peu plus paramétré grâce à l'utilisation de l'argument setParameter appliqué à l'objet query pour le passage des arguments de la requête :

```
String queryString =
"SELECT e FROM Employe e "
+ " WHERE e.salaire >= :salaire";
Query query = em.createQuery(queryString);
query.setParameter("salaire", "1000");
List<Employe> liste =
```

```
query.getResultList();
for (Employe e : liste) {
System.out.println(e.getNom());
}
em.close();
emf.close()
```

Blocs try-finally et catch

Pour des raisons évidente de robustesse du code, l'extrait de code précédent devrait être inclus dans un bloc try-finally et pourrait contenir éventuellement des blocs catch pour gérer les exceptions (Runtime-Exception) générées par les méthodes d'EntityManager.

Suppression d'une entité

La suppression d'une instance de bean entité `Employe` suppose que celle-ci soit « managée », c'est-à-dire qu'elle soit présente dans le contexte de persistance. Cela signifie que l'application à l'origine de la suppression soit déjà chargée ou qu'elle accède préalablement à l'entité par le biais de la méthode `find()`, comme dans l'extrait suivant :

```
public void suppressionEmploye(int employeId) {
Employe emp = em.find(Employe.class, employeId);
       if (emp != null) {
           em.remove(emp);
       }
}
```

La méthode `suppressionEmploye()` permet de vérifier, avant toute suppression de l'instance proprement dite avec la méthode `remove()`, son existence dans le contexte du gestionnaire d'entités.

Mise à jour d'une entité

La mise à jour d'une entité suppose que vous disposiez d'une référence à l'entité managée, référence obtenue *via* l'appel de la méthode `find()`, comme dans l'exemple suivant de mise à jour du salaire de l'instance de l'entité `employé` :

```
public Employe augmenteEmployeSalaire(int employeId, long augmentation) {
Employe emp = em.find(Employe.class, id);
if (emp != null) {
  emp.setSalaire(emp.getSalaire() + augmentation);
}
return emp;
}
```

Voici, mis bout à bout, le code complet de création, lecture, modification et suppression d'une entité `employe`, regroupées dans une seule classe des différentes méthodes CRUD décrites précédemment :

```
package com.webstock.chap10.model;
import java.util.Collection;

import javax.persistence.EntityManager;
import javax.persistence.Query;
```

```
public class EmployeService {

String texteRequete= "SELECT e FROM Employe e";

protected EntityManager em;

    public EmployeService(EntityManager em) {
        this.em = em;
    }

    public Employe createEmploye(int employeid, String nom, String adresse,
    ➥long salaire) {
        Employe emp = new Employe(employeid);
        emp.setNom(nom);
        emp.setAdresse(adresse) ;
        emp.setSalaire(salaire);
        em.persist(emp);
        return emp;
    }

    public void removeEmploye(int employeid) {
        Employe emp = findEmploye(employeid);
        if (emp != null) {
            em.remove(emp);
        }
    }

    public Employe augmenteEmployeSalaire (int employeid, long augmentation) {
        Employe emp = em.find(Employe.class, employeid);
        if (emp != null) {
            emp.setSalaire(emp.getSalaire() + augmentation);
        }
        return emp;
    }

    public Employe findEmploye(int employeid) {
        return em.find(Employe.class, employeid);
    }

    public Collection<Employe> findAllEmployes() {
        Query query = em.createQuery(texteRequete);
        return (Collection<Employe>) query.getResultList();
    }

}
```

Gestion des transactions JTA et non-JTA

Le gestionnaire d'entités offre des méthodes pour commencer (begin), commiter (commit), et défaire (rollback) des transactions pour une utilisation des transactions locale (non-JTA) et non locale (JTA, pour le support des transactions distribuées à travers plusieurs gestionnaires de BD).

Le conteneur EJB fournit des services de support à la gestion transactionnelle disponible par défaut pour les beans session et MDB. En dehors d'un serveur d'applications, une application JTA doit utiliser l'interface javax.persistence.EntityTransaction pour pouvoir

travailler avec des transactions locales à une ressource. Une instance de ce type peut s'obtenir à l'aide de la méthode `getTransaction()` de l'EntityManager.

Les six méthodes de support à la gestion transactionnelle offerte par cette interface sont les suivantes :

```
public interface EntityTransaction {
   public void begin() ;
   public void commit() ;
   public void rollback() ;
   public void setRollbackOnly();
   public void getRollbackOnly();
   public void isActive();
}
```

Exemple :

```
EntityManager em;
...
 try {
 em.getTransaction().begin()
 ...
 em.getTransaction().commit();
 }
 finally {
  em.close();
 }
```

Nous enjoignons le lecteur à approfondir ces aspects important liés à la gestion transactionnelle en examinant les détails de l'interface EntityTransaction dans la documentation du fournisseur JPA.

Méthodes de callback

Nous avons évoqué au chapitre précédent le concept d'invocation des méthodes selon des étapes liées au cycle de vie des beans.

Ce concept est applicable aux beans entité. Les méthodes peuvent être annotées pour indiquer qu'elles seront appelées par le fournisseur de persistance quand une entité passera dans une étape donnée de son cycle de vie. Ces méthodes peuvent appartenir à une classe entité ou une classe « à l'écoute » de ces événements (entité listener). Elles peuvent avoir n'importe quel nom, mais doivent posséder une signature ne prenant aucun paramètre et possédant un type de donnée void.

Les méthodes de type final ou static ne sont pas des types de callback valides. Les exceptions générées par ces méthodes ne peuvent donc pas être interceptées par un gestionnaire d'exceptions. La transaction est en ce cas simplement abandonnée.

Les différents types de méthodes de callback sont les suivantes :

- @PrePersist : quand persist (ou merge) s'est terminée avec succès.
- @PostPersist : après insertion dans la BD.
- @PreRemove : quand remove est appelée.
- @PostRemove : après suppression dans la BD.
- @PreUpdate : avant modification dans la BD.

- @PostUpdate : après modification dans la BD.
- @PostLoad : après lecture des données de la BD pour construire l'entité.

L'exemple qui suit illustre une utilisation possible de ces méthodes pour la mesure du temps de la dernière synchronisation du bean entité `Employe` avec la base de données (un bean étant considéré comme synchronisé avec la base chaque fois que ce dernier est lu ou sauvegardé dans celle-ci) :

```
@Entity
public class Employe {
    @Id
    @Column(name="EMPLOYEID")
    private employeid;
    private String nom;
    private String adresse;
    @Transient private long syncTime;
    // ...
    @PostPersist
    @PostUpdate
    @PostLoad
    private void resetSyncTime() {
       syncTime = System.currentTimeMillis();
    }
    public long getCachedAge() {
       return System.currentTimeMillis() - syncTime;
    }
  // ...
}
```

Invocation à partir d'un client Java et d'un client Web

Le code de gestion de votre entité `Employe` ayant été conçu à l'aide de JPA, vous pouvez développer la couche client d'invocation correspondante selon les deux modes permis par cette API :

- Mode client Java, qui offre un contrôle manuel du gestionnaire d'entités en mode J2SE simple. Dans ce mode, c'est l'application qui gère le cycle de vie du gestionnaire d'entités au lieu du conteneur.
- Mode full Web, avec prise en charge automatisé par le conteneur EJB des services de gestion du contexte et de l'interface EntityManager en utilisant les mécanismes embarqués de gestion des dépendances.

Gestionnaire d'entités géré par l'application (client Java)

Reprenez la classe `EmployeService` contenant le code fonctionnel de la gestion de l'entité `Employe`.

Vous avez juste besoin d'invoquer les méthodes d'obtention et d'initialisation d'une fabrique du gestionnaire d'entités, comme indiqué dans le code complet d'un client Java suivant (notez l'utilisation des méthodes de synchronisation du gestionnaire d'entités à l'aide de `begin()` et `commit()` dans Java SE) :

```
package com.webstock.chap10.client;
```

```
import java.util.Collection;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import com.webstock.chap10.model.Employe;
import com.webstock.chap10.model.EmployeService;

public class EmployeCRUDJava {

    public static void main(String[] args) {
        EntityManagerFactory emf =
                Persistence.createEntityManagerFactory("EmployeService");
        EntityManager em = emf.createEntityManager();
        EmployeService service = new EmployeService(em);

        // creation et persistance de l'employe
        em.getTransaction().begin();
        Employe emp = service.createEmploye(10, "John Doe","Main Street", 12000);
        em.getTransaction().commit();
        System.out.println("Employe sauvegarde : " + emp);

        // Recherche employe
        emp = service.findEmploye(10);
        System.out.println("Trouve " + emp);

        // Recherche tous les employes
        Collection<Employe> emps = service.findAllEmployes();
        for (Employe e : emps)
            System.out.println("Employe trouve : " + e);

        // Mise a jour employe
        em.getTransaction().begin();
        emp = service. augmenteEmployeSalaire (10, 1000);
        em.getTransaction().commit();
        System.out.println("Employe mis a jour:  " + emp);

        // supprime un employe
        em.getTransaction().begin();
        service.removeEmploye(10);
        em.getTransaction().commit();
        System.out.println("Employe 10 supprime");

        // Fermeture du gestionnaire d'entite EM et de l'EntityManagerFactory EMF
        em.close();
        emf.close();
    }
}
```

Gestionnaire d'entités géré par le conteneur (client Web)

La création d'applications dont le gestionnaire d'entités est géré par le conteneur est simple. Le seul prérequis est que l'EntityManagerFactory crée l'instance.

Ce qui différencie les deux approches tient à la manière dont la fabrique est obtenue. Une fois la fabrique obtenue, cette dernière peut être utilisée pour créer le gestionnaire d'entités, lequel peut ensuite être utilisé de la même façon que dans l'approche J2SE.

Pour illustrer ce mode, le code suivant crée une interface `EmployeService` censée décrire les opérations CRUD sur l'entité `Employe` :

```
public interface EmployeService {
   public Employe createEmploye(int id, String name, long salary);
   public void deleteEmploye(int id);
   public Employe updateEmployeSalaire (int id, long newSalaire);
   public Employe findEmploye(int id);
   public Collection<Employe> findAllEmployes();
}
```

Ensuite, vous pouvez utiliser l'annotation @PersistenceContexte pour déclarer une dépendance sur le contexte de persistance et permettre une acquisition automatisée.

L'extrait de code suivant illustre l'utilisation de l'annotation @PersistenceContext pour l'acquisition d'un gestionnaire d'entités à travers le mécanisme d'injection de dépendances offert par le conteneur. L'élément unitName spécifie le nom de l'unité de persistance sur laquelle s'appuie le contexte :

```
@Stateless
public class EmployeeServiceBean implements EmployeService {
@PersistenceContext(unitName="EmployeeService")
EntityManager em;
// ...
}
```

Le code suivant présente l'implémentation du bean session `EmployeService` décrivant les opérations CRUD sur l'entité `Employe` dans un contexte JEE (et non plus J2SE). Mesurez au passage toutes les facilités offertes par les services du conteneur, comme @Persistence-Context, qui permet l'injection automatique du contexte pour acquérir le gestionnaire d'entités, et l'élément unitName spécifiant l'unité de persistance :

```
@Stateless
public class EmployeServiceBean implements EmployeService {
  @PersistenceContext(unitName="EmployeService")
  protected EntityManager em;

  public EntityManager getEntityManager() {
     return em;
  }
  public Employe createEmploye(int employeId, String nom, String adresse, long salaire)
 {
     Employe emp = new Employe(employeid);
     emp.setNom(name);
     emp.setAdresse (adresse);
     emp.setSalaire(salaire);
     getEntityManager().persist(emp);
     return emp;
 }
public void deleteEmploye(int employeId) {
```

```
      Employe emp = findEmploye(employeId);
      if (emp != null) {
         getEntityManager().remove(emp);
      }
}
public Employe updateEmployeSalaire(int employeid, long nouvSalaire) {
      Employe emp = findEmployee(employeid);
      if (emp != null) {
         emp.setSalaire(nouvSalaire);
      }
      return emp;
}

public Employe findEmploye(int employeId) {
       return getEntityManager().find(Employe.class, id);
}

public Collection<Employe> findAllEmployes() {
       Query query = getEntityManager().createQuery("SELECT emp
           FROM Employe emp ");
       return (Collection<Employe>) query.getResultList();
}
}
```

Intégration de la couche de présentation

Il ne vous reste plus qu'à invoquer votre bean session stateless à partir d'une servlet. Les paramètres de la requête déterminent l'action correspondante, qui invoque ensuite la méthode dans le bean « injecté » `EmployeService` :

```
import com.webstock.service.EmployeService;

public class EmployeServlet extends HttpServlet {

    private final String TITRE =
        "Chapitre 10: Employee Service Exemple";

    private final String DESCRIPTION =
        "Gestion des opretaions CRUD sur l'entite employe " ;


    // injecte une reference vers EmployeService
    @EJB EmployeService service;

    public void doPost(HttpServletRequest request, HttpServletResponse response)
    ➥throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        printHtmlHeader(out);

        // process request
        String action = request.getParameter("action");
        if (action == null) {
            // Ne rien faire si aucune action demandee
        } else if (action.equals("Creation")) {
            Employe emp = service.createEmploye(
```

```
                parseInt(request.getParameter("createId")),
                request.getParameter("nom"),
                request.getParameter("adresse");
                parseLong(request.getParameter("salaire")));
        out.println("Employe cree:  " + emp);
    } else if (action.equals("Supression")) {
        String id = request.getParameter("removeId");
        service.removeEmploye(parseInt(employeId));
        out.println("Suppression Employe avec identifiant: " + id);
    } else if (action.equals("Update")) {
        String id = request.getParameter("raiseId");
        Employee emp = service.changeEmployeeSalaire(
                parseInt(employeid),
                parseLong(request.getParameter("raise")));
        out.println("employe mise a jour " + emp);
    } else if (action.equals("Find")) {
        Employe emp = service.findEmploye(
                parseInt(request.getParameter("findId")));
        out.println("Trouve " + emp);
    } else if (action.equals("FindAll")) {
        Collection<Employe> emps = service.findAllEmployes();
        if (emps.isEmpty()) {
            out.println("Pas d'employes trouves ");
        } else {
            out.println("Employes trouves: </br>");
            for (Employe emp : emps) {
                out.print(emp + "<br/>");
            }
        }
    }

    printHtmlFooter(out);
}

public void doGet(HttpServletRequest request, HttpServletResponse response)
➥throws ServletException, IOException {
    doPost(request, response);
}

private int parseInt(String intString) {
    try {
        return Integer.parseInt(intString);
    } catch (NumberFormatException e) {
        return 0;
    }
}

private long parseLong(String longString) {
    try {
        return Long.parseLong(longString);
    } catch (NumberFormatException e) {
        return 0;
    }
}

private void printHtmlHeader(PrintWriter out) throws IOException {
    out.println("<body>");
```

```
        out.println("<html>");
        out.println("<head><title>" + TITRE + "</title></head>");
        out.println("<center><h1>" + TITRE + "</h1></center>");
        out.println("<p>" + DESCRIPTION + "</p>");
        out.println("<hr/>");
        out.println("<form action=\"EmployeServlet\" method=\"POST\">");
        // form to create
        out.println("<h3>Creation Employe</h3>");
        out.println("<table><tbody>");
        out.println("<tr><td>employeId:</td><td><input type=\"text\" name=\"createId\"/>
        ➥(int)</td></tr>");
        out.println("<tr><td>Nom:</td><td><input type=\"text\" name=\"nom\"/>(String)
        ➥</td></tr>");
        out.println("<tr><td>Adresse:</td><td><input type=\"text\" name=\"adresse\"/>
        ➥(String)</td></tr>");
        out.println("<tr><td>Salaire:</td><td><input type=\"text\" name=\"salary\"/>
        ➥(long)</td>" +
                    "<td><input name=\"action\" type=\"submit\" value=\"Creation\"/>
                    ➥</td></tr>");
        out.println("</tbody></table>");
        out.println("<hr/>");
        out.println("<h3>Suppresion Employe</h3>");
        out.println("<table><tbody>");
        out.println("<tr><td>Id:</td><td><input type=\"text\" name=\"removeId\"/>
        ➥(int)</td>" +
                    "<td><input name=\"action\" type=\"submit\" value=\"Remove\"/></td>
                    ➥</tr>");
        out.println("</tbody></table>");
        out.println("<hr/>");
        // form to update
        out.println("<h3>Mise a jour employe</h3>");
        out.println("<table><tbody>");
        out.println("<tr><td>Id:</td><td><input type=\"text\" name=\"raiseId\"/>
        ➥(int)</td></tr>");
        out.println("<tr><td>Augmentation:</td><td><input type=\"text\" name=\
        ➥"raise\"/>(long)</td>" +
                    "<td><input name=\"action\" type=\"submit\" value=\"Update\"/>
                    ➥</td></tr>");
        out.println("</tbody></table>");
        out.println("<hr/>");
        // form to find
        out.println("<h3>recherche employe</h3>");
        out.println("<table><tbody>");
        out.println("<tr><td>Id:</td><td><input type=\"text\" name=\"findId\"/>
        ➥(int)</td>" +
                    "<td><input name=\"action\" type=\"submit\" value=\"Find\"/>
                    ➥</td></tr>");
        out.println("</tbody></table>");
        out.println("<hr/>");
        out.println("<h3>recherche des employee</h3>");
        out.println("<input name=\"action\" type=\"submit\" value=\"FindAll\"/>");
        out.println("<hr/>");
    }


    private void printHtmlFooter(PrintWriter out) throws IOException {
        out.println("</html>");
```

```
        out.println("</body>");
        out.close();
    }
}
```

Pour gérer les dépendances externes de l'application qui peuvent concerner des ressources de type JDBC, file de messages ou bean session, le conteneur EJB3 fournit l'annotation @EJB pour une gestion des dépendances simplifiée.

À l'aide de cette annotation, les références aux composants JEE référencés sont résolus dynamiquement par le conteneur ou au sein du code de l'application lorsque l'instance du composant est créée.

Le code complet de ces exemples est disponible sur la page Web dédiée à l'ouvrage (chapitre 10).

En résumé

Ce chapitre a mis en lumière la puissance et la richesse de l'API JPA, fondement de la spécification EJB3. Vous avez vu notamment comment mettre en œuvre les principales annotations JPA appliquées au mapping des relations inter-entités, des méthodes de call-back appliquées aux entités ainsi que des principales opérations permises sur une entité par le biais de l'interface du gestionnaire d'entités.

Le chapitre suivant présente le projet Dali, qui vise à simplifier le mapping O/R.

11

Mapping JPA avec Dali

Au chapitre précédent, vous avez découvert les concepts fondamentaux de l'API JPA. Extraits de code à l'appui, vous avez vu sa relative facilitée de mise en œuvre en même temps que sa puissance.

Dans ce chapitre, vous allez la mettre en œuvre dans Dali, un sous-projet de Web Tools dont le but est de faciliter la mise en œuvre de cette API.

Les outils de support à l'API JPA sur plate-forme Eclipse ont trouvé leur maturité avec le projet Dali. Issu de la mouvance Eclipse, il complète harmonieusement l'outillage Web Tools en offrant des outils de mapping capables de transformer un type de donnée objet en son pendant relationnel dans la base de données.

Eclipse Web Tools 2.0 intègre complètement le projet Dali, rendant plus facile sa mise en œuvre dans les développements autour de la plate-forme Europa.

Le projet Dali

L'objectif de Dali *(http://www.eclipse.org/dali/)* est de faciliter la mise en œuvre de JPA (Java Persistence API). Lancé en 2005 par Oracle, Dali a rejoint JBoss fin 2005.

La cible du projet est le support de la persistance des EJB 3.0 *via* l'API JPA. Depuis son lancement (la version courante est la version Dali 1.5), Dali supporte l'implémentation de référence JPA TopLink Essentials, contribution d'Oracle.

Ce projet offre en particulier les fonctionnalités suivantes :

- Configuration d'un projet Java pour le support de JPA et des annotations pour l'environnement Java SE ou JEE.
- Support des approches de développement top-down (descendante), bottom-up (ascendante) et meet-in-the-middle (mixte).
- Support simplifié des différentes bases de données *via* les nombreux connecteurs disponibles et les différentes vues et perspectives de manipulation des données.

- Intégration d'outils de mapping en liaison avec des vues dédiées, comme JPA Details et JPA Structure, qui permettent de naviguer dans les champs persistants des entités et de la classe, de changer les types de relation, etc.
- Génération des fichiers de persistance persistence.xml et orm.xml pour le support des métadonnées.

Scénarios de développement et configuration

Dali propose différents scénarios de développement autour de JPA et des technologies EJB, Web et POJO. La différence essentielle entre ces scénarios réside dans la configuration de la variable d'environnement classpath.

Si votre application s'exécute au sein d'un conteneur Web ou JEE, elle hérite des bibliothèques JAR du conteneur. Si vous développez une application simple avec JSE5 (sans le support du conteneur), la configuration est entièrement de votre responsabilité. Il suffit pour cela d'ajouter au projet Java sous Eclipse les références aux bibliothèques de référence de TopLink Essentials, l'implémentation de référence de JPA.

La figure 11.1 illustre ce dernier scénario, avec le menu Preferences d'Eclipse configuré pour le support JPA *via* l'option User Libraries.

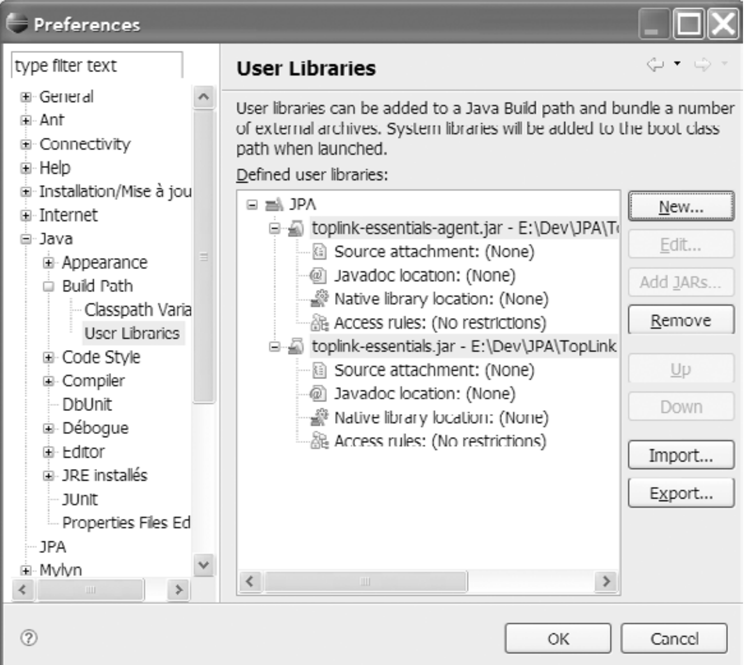


Figure 11.1

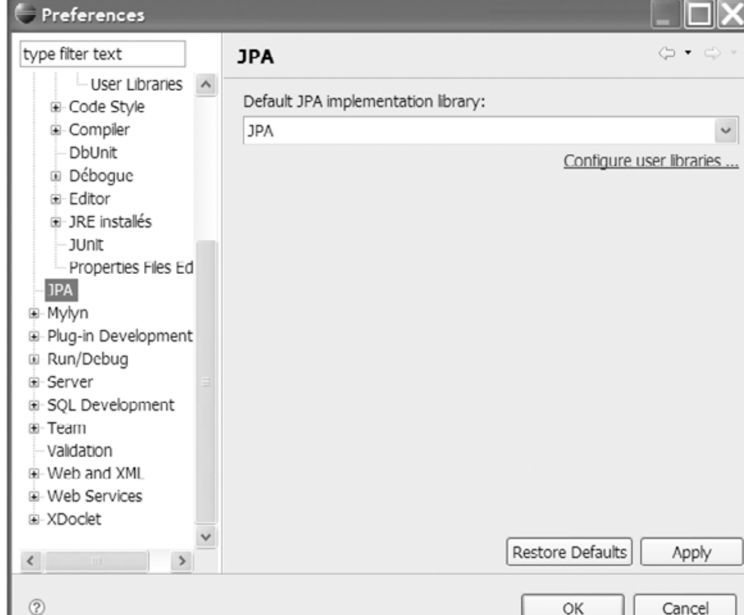
Configuration du support JPA via *le menu Preferences d'Eclipse (1/2)*

Une fois définie la bibliothèque personnalisée, il suffit de l'associer à l'option JPA en prenant bien soin de cliquer sur Apply pour valider l'environnement, comme l'illustre la figure 11.2.

Vous pouvez à ce stade ajouter à votre projet Java le support JPA en choisissant Properties, Java Build Path, Add Library et User Library dans le menu contextuel du projet puis en sélectionnant la bibliothèque JPA précédemment créée.

Figure 11.2

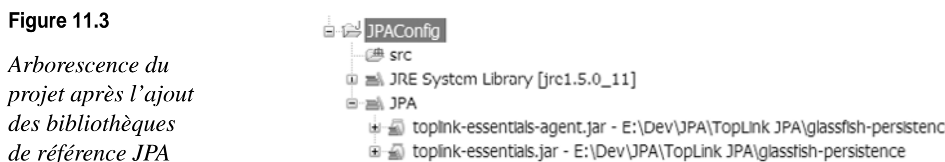
Configuration du support JPA via *le menu Preferences d'Eclipse (2/2)*



Le chemin d'accès aux bibliothèques JPA se présente comme illustré à la figure 11.3.

Figure 11.3

Arborescence du projet après l'ajout des bibliothèques de référence JPA



Avec votre chemin de compilation correctement positionné, vous pouvez créer et compiler vos entités (ici `Employe`) après import des annotations nécessaires. Le code annoté est compilé dans l'éditeur Eclipse, comme illustré à la figure 11.4.

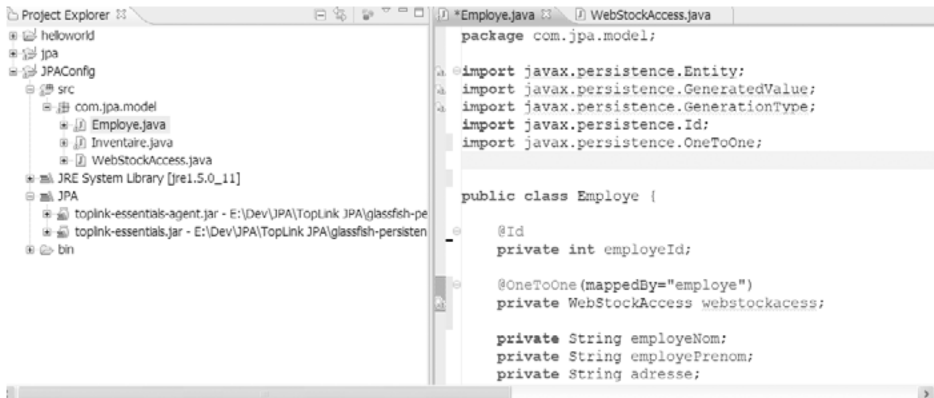


Figure 11.4

Compilation de l'entité dans Eclipse

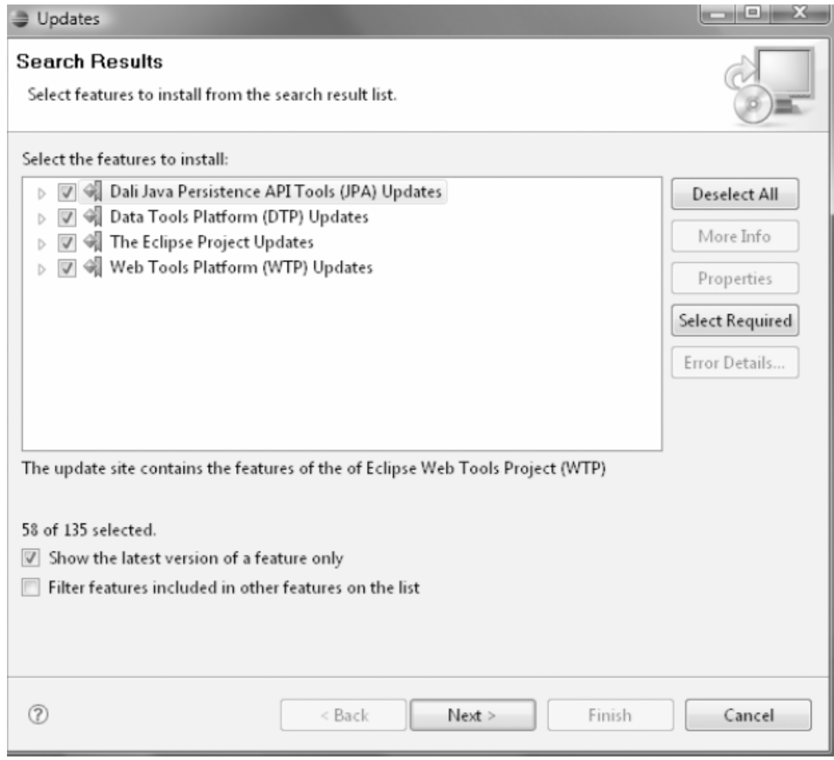
Mise en œuvre de l'API JPA avec Dali

Dali ne requiert qu'une installation d'Europa 3.3 avec le support de l'outillage Web Tools 2.0.

Il importe de mettre à jour la configuration avec les fonctionnalités illustrées à la figure 11.5 *via* le gestionnaire de mise à jour d'Eclipse.

Figure 11.5

Mise à jour de la configuration de l'outillage de mapping Dali



Nous supposons installée et configurée votre base WEBSTOCKDB (voir en annexe pour la configuration de la base HSQLDB livrée en standard avec JBoss 4.2). Notez que Dali permet de régénérer le modèle physique à partir de DDL en entrée.

Nous supposons également configuré (*via* le menu Preferences d'Eclipse) le driver Hypersonic DB, comme illustré à la figure 11.6.

L'implémentation de référence de TopLink JPA est disponible en téléchargement sur le site de l'éditeur Oracle à l'adresse *http://www.oracle.com/technology/products/ias/toplink/jpa/download.html.*

Pour la configurer, il suffit d'ouvrir une sessions DOS et de saisir la commande suivante (selon la version de TopLink JPA, ici la v2, build 41) :

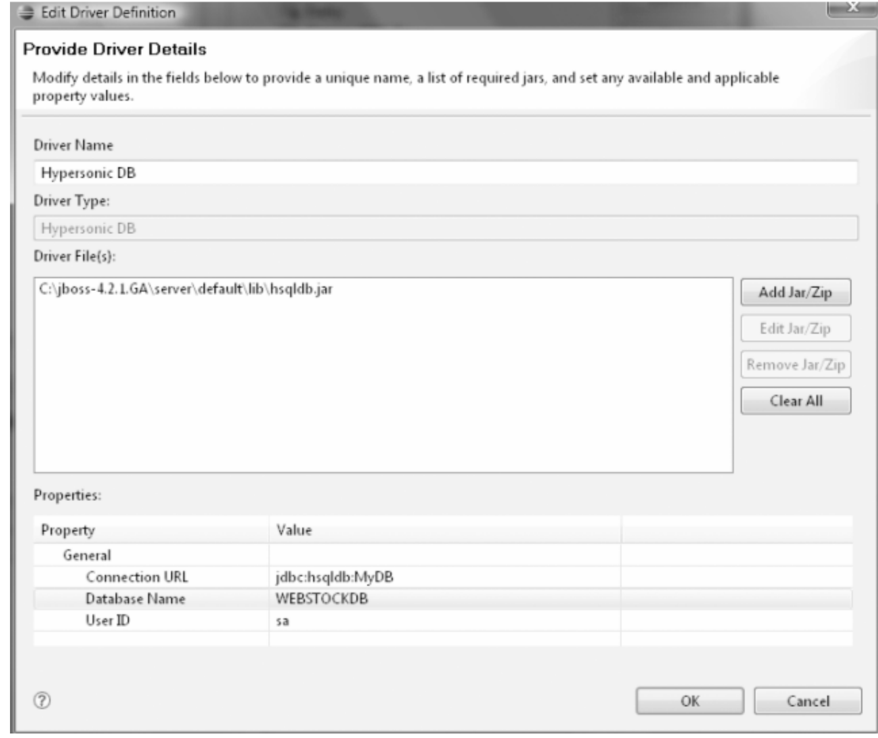
```
java -jar glassfish-persistence-installer-v2-41.jar
```

L'acceptation des conditions de licence a pour effet de décompresser dans le répertoire courant l'ensemble des bibliothèques TopLink requises.

Il faut ensuite démarrer le serveur JBoss pour amorcer le démarrage de la base HSQLDB.

Figure 11.6

Configuration du driver Hypersonic DB



Création du projet JPA et connexion à une source de données

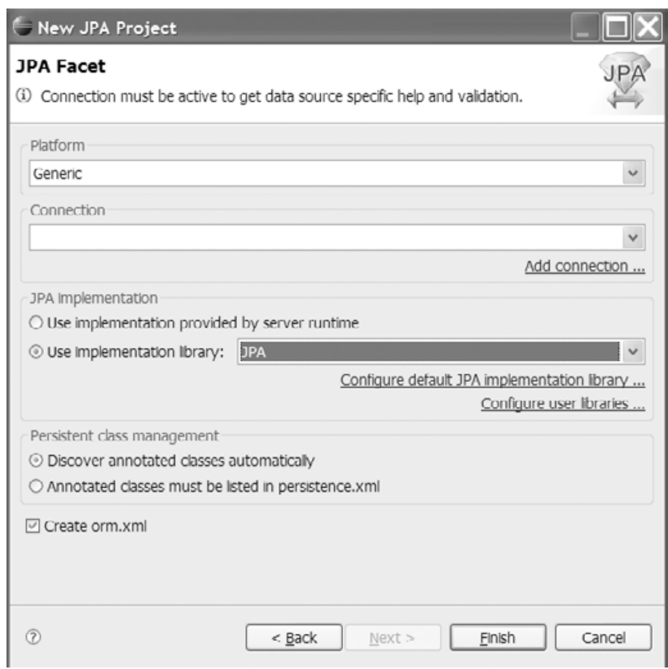
Dans cette première étape, vous allez créer un projet supportant JPA et configurer votre connexion à une source HSQLDB, la base de données livrée et embarquée pour la distribution de JBoss 4.2.

1. Cliquez sur File et New project.
2. Dans l'assistant de création de projet, sélectionnez JPA project.
3. Cliquez sur Next, et saisissez MonProjetJPA. Laissez les autres options inchangées (Target runtime et Utility Configuration en particulier, désignant le serveur d'applications cible et la configuration choisie, soit le support JPA avec JSE 5).

   Si vous disposez déjà d'une connexion existante configurée, vous pouvez la réutiliser en la spécifiant dans le champ Configurations de l'assistant de création de projet JPA.
4. Cliquez sur Next, et laissez les projets Facet proposés par défaut (Java/Java Persistence et Utility Module). Cliquez sur Suivant.
5. L'écran suivant permet de configurer la source de données associée et d'activer la validation. Sélectionnez la bibliothèque JPA comme illustré à la figure 11.7, et cliquez sur Add connection pour configurer la connexion à la base de données en laissant les autres options inchangées.
6. Sélectionnez HSQLDB Connection Profile dans la liste proposée par l'assistant de création de nouvelles connexions (voir figure 11.8), puis cliquez sur Next.

Figure 11.7

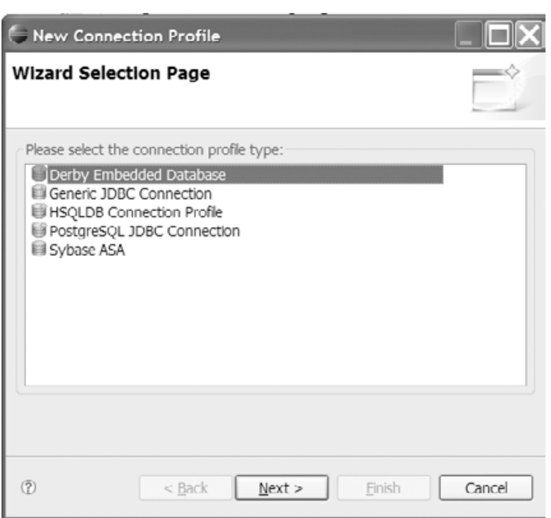
Configuration du projet JPA avec Dali



> **Persistent class management**
>
> Dans la zone « Persistent class management », Dali permet d'indiquer si le runtime JPA doit découvrir les entités dynamiquement ou si elles doivent être listées dans le fichier persistence.xml. La case à cocher Create orm.xml permet de spécifier un fichier de mapping XML au lieu des annotations standards.

Figure 11.8

Liste des sources de données supportées par Dali



7. Dans l'assistant de création de profil de connexion, donnez un nom à cette connexion, par exemple HSQLDBConn, et cochez l'option « Auto-connect at startup » pour une connexion automatique à la source de données (voir figure 11.9). Cliquez sur Suivant.

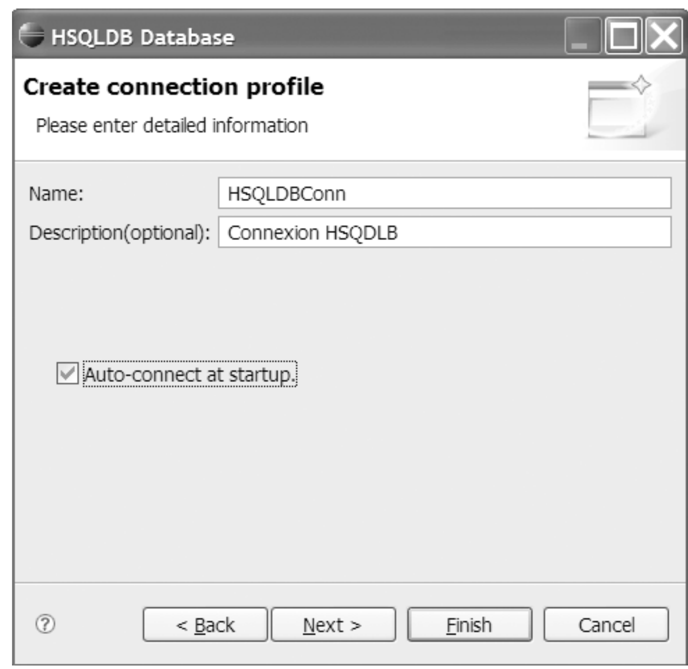


Figure 11.9

Création d'un profil de connexion (1/2)

8. L'écran suivant permet de configurer la connexion à la base HSQLDB, ici SAMPLE (voir figure 11.10).

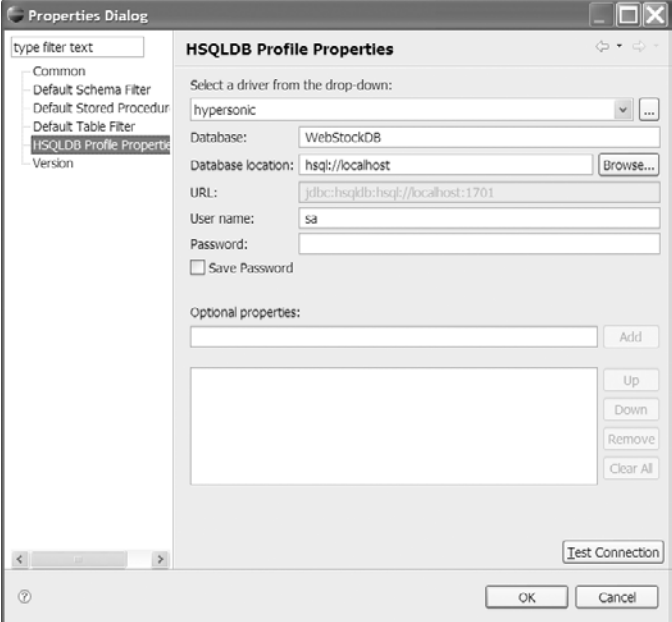


Figure 11.10

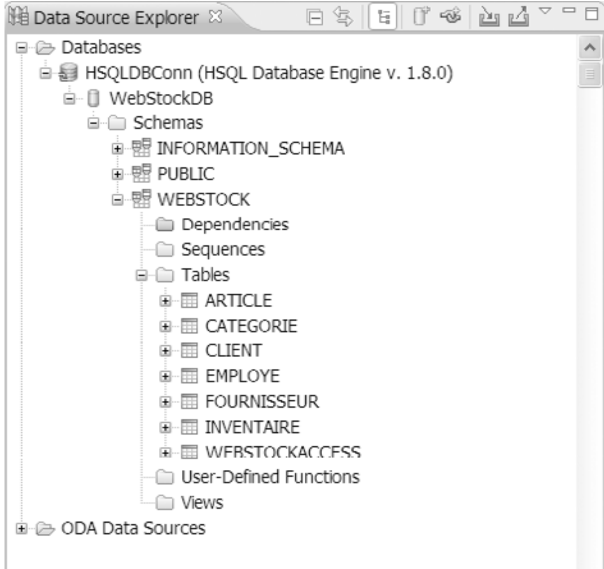
Création d'un profil de connexion (2/2)

9. Cliquez sur le bouton Test Connection pour valider la connexion, puis cliquez sur OK pour terminer la configuration de la connexion.

Vous pouvez vérifier le succès de la connexion dans la vue Data Source Explorer, qui contient une référence à la source de données Hypersonic définie précédemment et la liste des tables du modèle webstock, comme l'illustre la figure 11.11.

Figure 11.11

Vue Explorer de source de données



Remarquez la création du fichier persistence.xml suivant sous le répertoire src\META-INF :

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
➥xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation
➥="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/
➥persistence_1_0.xsd">

<persistence-unit name="MonProjetJPA-Unit">
</persistence-unit>

<properties>
    <property name="toplink.logging.level" value="FINEST"/>
    <property name="toplink.jdbc.driver"
   value="org.apache.derby.jdbc.ClientDriver"/>
    <property name="toplink.jdbc.url"
   value="jdbc:derby://localhost:1527/WebStockDB;create=true"/>
    <property name="toplink.jdbc.user" value="dali"/>
    <property name="toplink.jdbc.password" value="dali"/>
</properties>
</persistence-unit>

</persistence>
```

Le nom de l'unité de persistance est par défaut celui de votre projet. Ce dernier est associé à la connexion définie plus haut.

Vous avez achevé la définition de l'unité de persistance du projet avec les propriétés associées à la connexion à la base WebStockDB.

Votre projet est prêt à utiliser l'API JPA avec Dali.

Création des entités persistantes du modèle et mapping vers la base

Votre modèle contient trois entités, `Inventaire`, `Article` et `Commande`, que vous allez intégrer dans votre projet et qui vont servir de cadre à la mise en œuvre de l'outillage Dali.

1. Créez un package appelé com.webstock.chap11.model.
2. Cliquez sur New et Class, puis saisissez le nom de la classe `Inventaire`.
3. Répétez cette étape pour `Article` et `Commande`.
4. Ajoutez les champs récapitulés au tableau 11.1 pour chacune des entités.

Tableau 11.1 Structure du modèle physique de la base WebStockDB

Table	Attribut	Type java	Colonne associée	Type de donnée SQL
Inventaire	numInventaire	long	numeroinventaire	double (clé primaire)
	article	article	articleid	integer not null
	quantite	int	quantite	integer not null
	prix	double	prix	decimal (8,2)
	rayon	string	rayon	varchar (2)
	region	string	region	varchar (15)
Article	articleid	int	articleid	integer not null (clé primaire)
	nomArticle	string	nomarticle	varchar (30) not null
	articleCategorie	string	articlecategorieid	varchar (5)
	fournId	integer	fournisseurid	integer
	desc	string	description	varchar (50)
	poids	double	poids	decimal (10,2)
	image_url	string	image_url	varchar (50)
Commande	numCommande	long	numerocommande	integer not nul (clé primaire)
	dateCommande	date	datecommande	date not null
	article	article	articleid	integer not null (clé étrangère vers article)
	owner	string	clientid	integer not null
	quantite	int	quantite	integer not null
	etat	string	etatcommande	varchar (10)
Client	clientId	int	clientid	integer not null (clé primaire)
	userid	int	userid	integer (clé étrangère)
	clientNom	string	clientnom	varchar (15)
	clientPrenom	string	clientprenom	varchar (15)
	adresse	string	adresse	varchar (25)
	telNumero	string	tel	varchar (10)
	comment	string	comment	varchar (25)
	version	int	version	int

5. Pour la classe `Inventaire`, générez les getters et setters pour les champs suivants, sauf pour l'attribut numInventaire :

```
public class Inventaire {

private long numInventaire;
protected Article article;
protected double prix;
protected int quantite;
protected String rayon;
protected String region;
protected int version ;

public double getCost() {
    return cost;
}
public void setCost(double cost) {
    this.cost = cost;
}
public Article getArticle() {
   return article;
}
public void setArticle(Article article) {
    this.article = article;
}
public double getPrix() {
    return prix;
}
public void setPrix(double prix) {
    this.prix = prix;
}
// A completer…
```

6. Pour la classe `Article`, générez les getters et setters pour les champs suivants :

```
public class Article {

protected int articleId;
protected String nomArticle;
protected String articleCategorie;
protected int fournId;
protected String desc;
protected double poids;
protected String image_url;

}
```

7. Pour la classe `Commande`, générez les getters et setters pour les champs suivants :

```
public class Commande {

protected long numCommande;
protected Date dateCommande;
protected List <Article> articles;
protected int quantite;
protected String etat;
 protected Client owner ;

}
```

8. Pour la classe `Client`, générez les getters et setters pour les champs suivants :

```
public class Client {

protected int clientId;

protected int userid;

protected String clientNom;
protected String clientPrenom;
protected String adresse;
protected String telNumero;
protected String comment;
protected int userid;
protected int version ;

protected List <Commande> commandes ;



}
```

Votre modèle est créé. Vous pouvez passer à la création des entités persistantes et les associer aux tables correspondantes de la base.

Mapping des classes avec le modèle physique de données

Vous allez transformer chacune des classes en entité persistante en associant chaque entité du modèle avec sa table de la base :

1. Ouvrez le fichier Article.java dans la vue Package Explorer d'Eclipse. Vous pouvez voir s'afficher la vue JPA Structure correspondante.
2. Sélectionnez la classe `Article`, puis, dans la vue JPA Details, sélectionnez dans la liste déroulante Map As la valeur Entity, comme illustré à la figure 11.12.

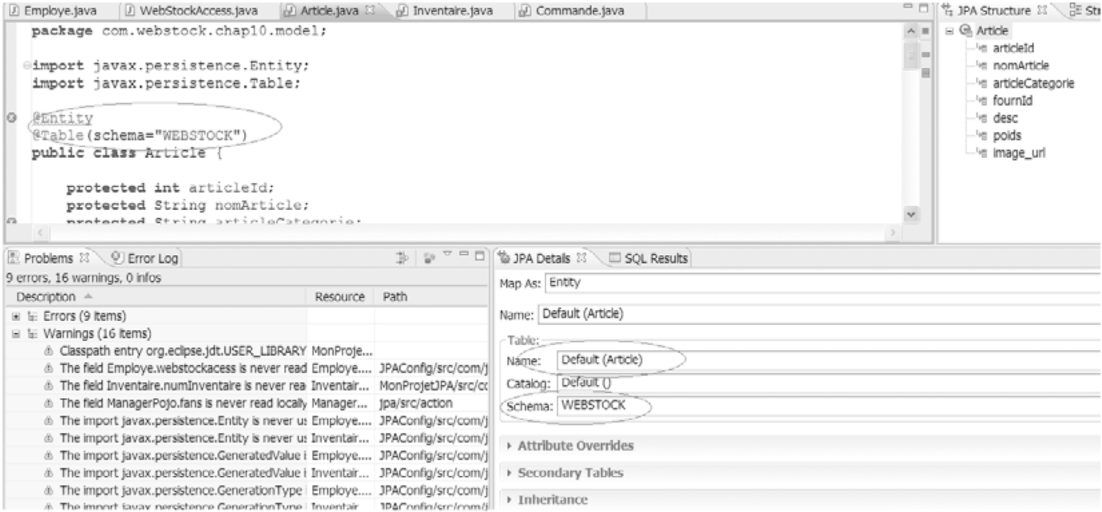


Figure 11.12

Détails de la vue JPA associée à l'entité Article

> **Schema**
> Il ne faut pas omettre de préciser le schéma webstock dans l'option Schema de la vue JPA Details afin de mapper exactement la table de la base à l'entité.

Vous pouvez constater qu'une balise @Entity a été automatiquement ajoutée dans le code de la classe `Article`, indiquant qu'elle est devenue une entité persistante.

3. Dans la vue Package Explorer, faites un clic droit sur le fichier persistence.xml, et sélectionnez JPA Tools puis Synchronize Classes. Cela a pour effet de mettre à jour le fichier descripteur avec la nouvelle entité ajoutée (voir balise <class>). Notez que Dali a automatiquement associé l'entité `Article` à la table `Article` correspondante de la base.

   Vous devez à ce stade voir s'afficher des erreurs dans la vue Problems d'Eclipse, du fait que certaines colonnes ne sont pas identiques au nom des colonnes associées dans la base. Vous résoudrez un peu plus loin ce problème de mapping.

4. Répétez les mêmes étapes pour les deux autres entités du modèle webstock extrait (`Inventaire` et `Commande`) et leurs tables associées.

Mapping des champs de la classe

Dans cette étape, vous allez mapper les attributs des classes avec ceux de la table de la base.

Pour cette application, vous utiliserez les types de mappings suivants :

- identifiant de clé
- de base
- One-to-One
- One-to-Many

Création des mappings d'identifiant de clé

Vous devez spécifier l'identifiant de clé primaire associé à chaque entité.

1. Dans l'explorateur de package, ouvrez la classe Article.java.
2. Sélectionnez le champ articleId dans la vue JPA Structure. La vue JPA Details affiche les propriétés du champ.
3. Dans la liste déroulante Map As, sélectionnez Id. Cela a pour effet d'ajouter l'annotation @Id dans le code de la classe `Article`, permettant de définir cette colonne comme clé primaire.
4. Positionnez Insertable et Updatable à false. Cela a pour effet d'ajouter les annotations suivantes :

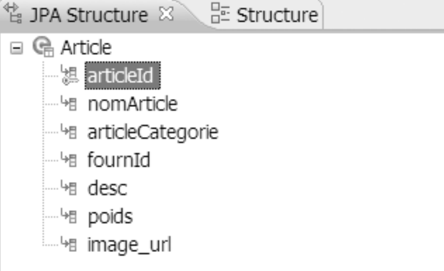
```
@Column(insertable=false, updatable = false)
```

   Ces annotations JPA permettent de spécifier si la colonne doit être utilisée en mode insert ou update.

   Remarquez dans la vue JPA Structure que le champ articleId a été identifié comme clé primaire avec le symbole associé (voir figure 11.13).

Figure 11.13

Structure JPA de l'entité Article



5. Répétez les mêmes étapes pour l'entité `Inventaire` associée à la clé numInventaire pour l'entité et à la colonne numeroInventaire et pour l'entité `Commande` associée à la clé numCommande et à la colonne numeroCommande.

Pour ces deux champs, comme le nom de la colonne dans la base et celui de l'attribut dans l'entité sont différents, il faudra générer l'annotation @Column *via* la liste déroulante Column/Name, comme illustré à la figure11.14.

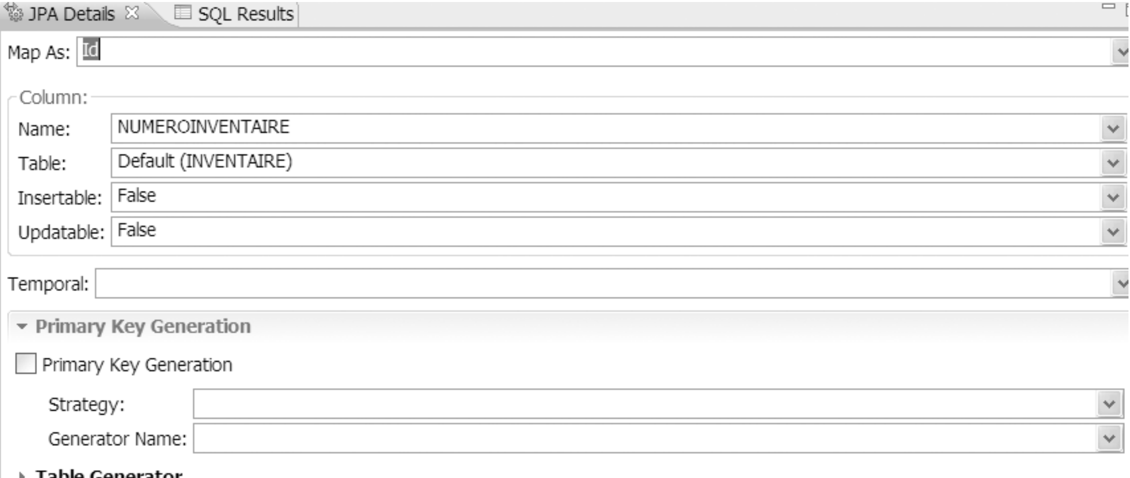


Figure 11.14

Mapping JPA avec Dali

Cela aura pour effet de générer le code suivant (pour l'entité `Commande`) :

```
@Id
@Column(name="NUMEROCOMMANDE", insertable = false, updatable = false)
```

Stratégie de génération des clés primaires

Dali permet de générer le code annoté associé au mode de génération des clés primaires à l'aide de la balise @GeneratedValue.

Cela s'effectue dans la vue JPA Details *via* la configuration des listes déroulantes Strategy et Table Generator, comme illustré à la figure 11.15.

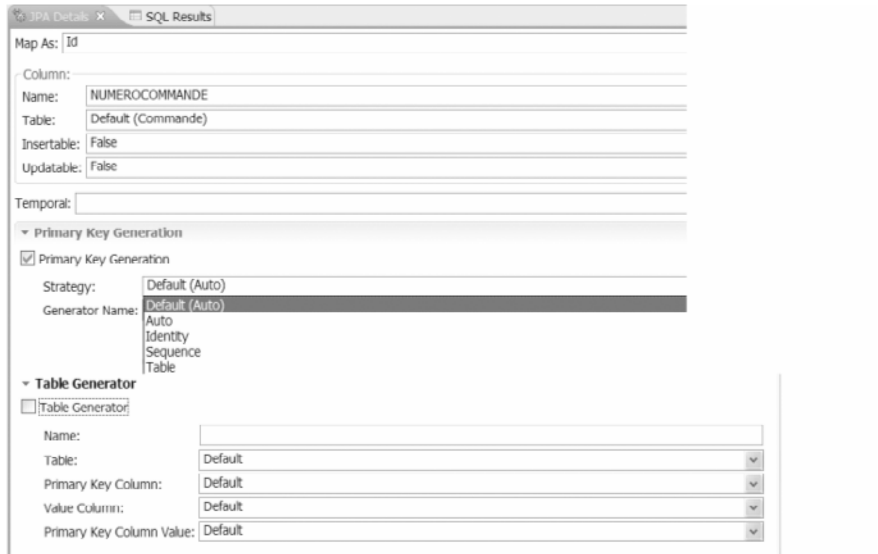


Figure 11.15

Assistant de support à la génération de clés primaires Dali

Le tableau 11.2 récapitule les propriétés de configuration associées à la mise en œuvre d'une stratégie de génération de clés primaires.

Tableau 11.2 Configuration d'une stratégie de génération de clés primaires

Propriété	Description	Valeur par défaut
Primary Key Generation	Définit comment la clé primaire est générée. Ce champ correspond à l'annotation @Generated Value.	
Strategy	– Auto – Sequence : les valeurs d'incrément sont assignées par le biais d'une table sequence. – Identity : les valeurs d'incrément sont assignées par une colonne Identity de la colonne de la base de données. – Table : les valeurs d'incrément sont assignées par une table de la base.	Auto
Generator Name	Nom unique pour la valeur générée	
Table Generator : les champs qui suivent vont définir les tables de la base utilisées pour générer la clé primaire et qui vont être associées à l'annotation @TableGenerator. Ces champs ne s'appliqueront que si la stratégie est de type Table.		
Name	Nom unique du générateur	
Table	Table qui va stocker les valeurs de séquence générées.	
Primary Key Column	Colonne dans la table de génération qui va contenir la clé primaire.	
Value Column	Colonne qui va stocker la valeur générée.	
Primary key Column value	Valeur associée à la colonne clé primaire dans la table servant de génération.	
Sequence Generator : ces champs définissent la séquence spécifique utilisée pour générer la clé primaire et correspondent à l'annotation @SequenceGenerator. Les champs qui suivent ne s'appliquent que lorsque la stratégie est Sequence.		
Name	Nom de la table séquence à utiliser	
Sequence	Nom unique de la séquence	

Pour les trois entités de votre portion de modèle webstock, `Article`, `Commande` et `Client`, vous devez spécifier un mode de stratégie de génération de clés primaires de type Auto (génération automatique), qui est le mode par défaut.

Pour ce faire, procédez comme suit :

1. Cochez l'option Primary Key Generation
2. Spécifiez la stratégie par défaut, Auto.
3. Laissez le champ Generator Name vide.
4. Mappez les attributs restants avec leurs correspondants dans la table de la base (en particulier, pour l'entité `Article`, les attributs articleCategorie avec `ARTICLECATEGO-RIEID`, fournId avec `FOURNISSEURID` et desc avec `DESCRIPTION`, et, pour l'entité `Commande`, l'attribut etat avec la colonne `ETATCOMMANDE`).

Vous avez achevé la première étape de la configuration. Vous pouvez passer au mapping des types de données de chacun des attributs des entités concernées.

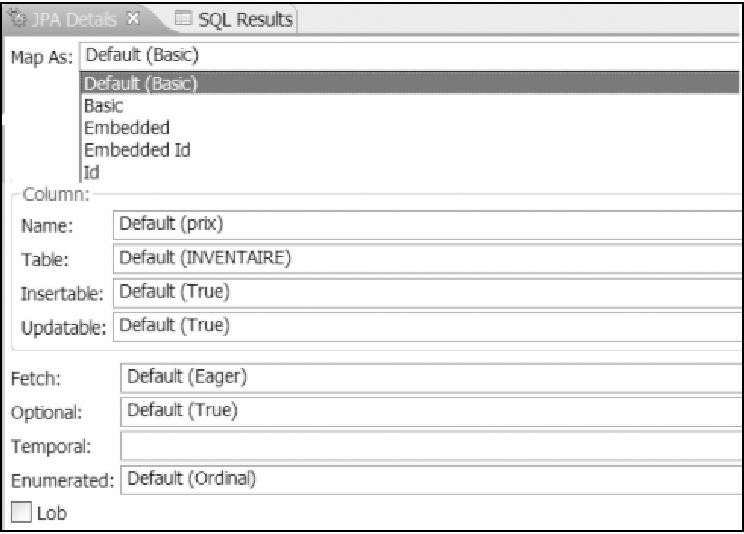
Création des mappings de base

Le mapping de type « basic » permet de mapper des attributs directement avec une colonne de la base. Ce type de mapping ne peut être utilisé qu'avec les types de données primitifs et les « wrappers », par exemple les types string, byte, char, character, date, calendar, time, timestamp, bigDecimal et bigInteger, ainsi qu'avec tout autre type implémentant l'interface Serializable.

Dali propose l'assistant de mapping illustré à la figure 11.16 pour configurer individuellement, selon le type de mapping (champ Map As) choisi, chaque colonne de la table.

Figure 11.16

Assistant de mapping Dali



Le champ Map As propose les valeurs suivantes :

- Default (Basic)
- Basic

- Embedded
- Embedded Id
- Id
- Many to Many
- Many to One
- One to Many
- One to One
- Transient
- Version

Le tableau 11.3 détaille chacune de ces propriétés et fournit des exemples d'application associée à chaque situation avec le code annoté généré. Cela vous sera utile pour l'application des différents types de mapping associés à votre portion de modèle webstock avec Dali.

Tableau 11.3 Propriétés du champ Map As

Propriété	Description	Valeur par défaut	S'applique au mapping suivant

Map As	Définit trois types de mappings :
	– Basic Mapping : correspond à l'annotation @Basic et ne concerne que les types de données Java de base et les wrappers : string, byte, char, character, date, calendar, time, timestamp, bigDecimal, bigInteger ainsi que tout autre type qui implémente l'interface Serializable.
	– Embedded Mapping : à utiliser lorsque vous souhaitez mapper un attribut de l'entité à une instance de type embeddable class, ou classe embarquée. Une classe embarquée est une classe dont les instances sont stockées comme une partie de l'entité mère et qui partage l'identifiant de cette dernière. Cela correspond à l'utilisation d'@Embedded combinée à @AttributeOverride, qui permet de surcharger la colonne d'un objet embarqué pour une entité donnée et sur une propriété particulière. Exemple :
	`@Embeddable`
	`    public class EmploymentPeriod {`
	`        java.util.Date startDate;`
	`        java.util.Date endDate;`
	`    ...`
	`}`
	`@Entity`
	`public class Employee implements Serializable {`
	`    ...`
	`    @Embedded`
	`    @AttributeOverrides({`
	`        @AttributeOverride(name="startDate",`
	`column=@Column("EMP_START")),`
	`        @AttributeOverride(name="endDate",`
	`column=@Column("EMP_END"))`
	`    )`
	`    public EmploymentPeriod getEmploymentPeriod() {`
	`        ...`
	La classe `EmploymentPeriod` peut être embarquée dans la classe entité `Employee` en utilisant les attributs annotés @AttributeOverrides.
		Basic	

Tableau 11.3 Propriétés du champ Map As *(suite)*

<table>
<tr><th>Propriété</th><th>Description</th><th>Valeur par défaut</th><th>S’applique au mapping suivant</th></tr>
<tr><td>Map As (suite)</td><td>

– Embedded Id : permet de spécifier la clé primaire d’un identifiant d’entité de type Embeddable. Ce type de mapping correspond à @EmbeddedId. Constitué d’un ensemble composite de clés primaires appartenant à l’entité, il se rencontre lorsque le mapping est effectué à partir de systèmes dit legacy, dont la clé est constituée de plusieurs colonnes. Il s’applique à des entités de type embarqué. Exemple :

```
Embeddable
public class EmployeePK implements Serializable
{
    private String name;
    private long id;

    public EmployeePK()
    {
    }

    public String getName()
    {
        return name;
    }

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

    public long getId()
    {
        return id;
    }
    public void setId(long id)
    {
        this.id = id;
    }

    public int hashCode()
    {
        return (int) name.hashCode() + id;
    }

    public boolean equals(Object obj)
    {
        if (obj == this) return true;
        if (!(obj instanceof EmployeePK)) return false;
        if (obj == null) return false;
        EmployeePK pk = (EmployeePK) obj;
        return pk.id == id && pk.name.equals(name);
    }
}
@Entity
public class Employee implements Serializable
{
    EmployeePK primaryKey;

    public Employee()
    {
    }
```

</td><td></td><td></td></tr>
</table>

Tableau 11.3 Propriétés du champ Map As *(suite)*

Propriété	Description	Valeur par défaut	S'applique au mapping suivant

Map As *(suite)*	`@EmbeddedId`
	`public EmployeePK getPrimaryKey()`
	`{`
	`    return primaryKey;`
	`}`
	
	`public void setPrimaryKey(EmployeePK pk)`
	`{`
	`    primaryKey = pk;`
	`}`
	Les relations de type Many-to-One, Many-to-Many, One-to-Many, One-to-One ont étés décrites dans ce chapitre. Elles génèrent les annotations de support aux relations correspondantes.
		Basic	

Column	Colonnes de la table mappées aux attributs de l'entité (correspond à l'annotation @Column)	Par défaut, les colonnes sont supposées avoir le même nom que les attributs de l'entité.	
Table	Nom de la table de la BD qui contient le nom de la colonne sélectionnée.		

Fetch	Définit la stratégie de chargement des données de la BD :
	– Eager : les données sont chargées au préalable avant leur utilisation.
	– Lazy : les données sont chargées seulement au besoin.
	Exemple:
	`@Entity`
	`public class Employee implements Serializable {`
	`    ...`
	`    @Basic(fetch=LAZY)`
	`    protected String getName() {`
	`        return name;`
	`  }`
		Eager	

Optional	Spécifie si le champ peut être null.	True	

Temporal	Spécifie si le champ mappé appartient à un des types suivants :
	– Date : java.sql.Date
	– Time : java.sql.Time
	– Timestamp : java.sql.Timestamp
	Ce champ correspond à l'annotation @Temporal. Exemple :
	`@Entity`
	`public class Employee {`
	`    ...`
	`@Temporal(DATE)    protected java.util.Date`
	`  startDate;`
	`    ...`
	`}`		

Tableau 11.3 Propriétés du champ Map As *(suite)*

<table>
<tr><th>Propriété</th><th>Description</th><th>Valeur par défaut</th><th>S'applique au mapping suivant</th></tr>
<tr><td>Enumerated</td><td>Spécifie la manière dont les types de données Enum vont persister dans la base :
– Soit vers une colonne ordinale (en stockant le numéro ordinal de l'enum).
– Soit vers une colonne de type chaîne de caractères (en stockant la chaîne de caractères représentant l'Enum).
La représentation de la persistance, par défaut ordinale, peut être surchargée grâce à l'annotation @Enumerated, comme l'illustre la propriété PayScale de l'exemple suivant, dans laquelle l'Enum persiste en tant que string :
<pre>
public enum EmployeeStatus {FULL_TIME, PART_TIME, CONTRACT}

public enum SalaryRate {JUNIOR, SENIOR, MANAGER, EXECUTIVE}

@Entity
public class Employee {
    ...
    public EmployeeStatus getStatus() {
    ...
    }

    @Enumerated(STRING)
    public SalaryRate getPayScale() {
    ...
  }
</pre></td><td></td><td></td></tr>
<tr><td>Lob</td><td>Indique que la propriété devrait être persistée dans un Blob ou un Clob selon son type : java.sql.Clob, Character[], char[] et java.lang.String seront persistés dans un Clob. java.sql.Blob, Byte[], byte[] et les types sérialisables seront persistés dans un Blob. Ce champ correspond à l'annotation @Lob. Exemple :
<pre>
@Lob
public String getFullText() {
    return fullText;
}

@Lob
public byte[] getFullCode() {
    return fullCode;
}
</pre></td><td></td><td></td></tr>
<tr><td>Target Entity</td><td>Entité persistante sur laquelle l'attribut est mappé.</td><td></td><td>Tout type de mapping de relation</td></tr>
<tr><td>Mapped By</td><td>Champ de la table qui possède la relation. L'association peut être bidirectionnelle, et, dans ce cas, une des extrémités doit être responsable de la mise à jour des colonnes de l'association. C'est là que se mesure l'utilité de cet attribut, qui doit être spécifié avec @OneToOne dans l'entité qui ne définit pas de colonne de jointure, comme dans l'exemple suivant entre l'entité <code>Employé</code> et <code>Badge</code> :
<pre>
@Entity
public class Employe {
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="badge_fk")
    public Badge getBadge() {
        ...
    }
</pre></td><td></td><td>Mapping de relations de type mono valuées</td></tr>
</table>

Tableau 11.3 Propriétés du champ Map As *(suite)*

Propriété	Description	Valeur par défaut	S'applique au mapping suivant

Mapped By *(suite)*	`@Entity`
	`public class Badge {`
	`    @OneToOne(mappedBy = "badge")`
	`    public Employe getOwner(){`
	`    ...`
	`}`
			Mapping de relations de type mono valuées

Optional	Spécifie si le champ peut être null.	Yes	Mapping de type multi valué
Join Columns	Spécifie une colonne mappée pour joindre une entité. Ce champ correspond à l'attribut @JoinColumn et correspond à la colonne de jointure. Fonctionne si l'attribut « Override Default » est coché.		Mapping de type mono valué
Cascade	Spécifie quels types d'opérations sont propagées à travers l'entité : – All : toutes les opérations. – Persist : effectue en cascade l'opération de persistance (création) sur les entités associées si `persist()` est appelée ou si l'entité est supervisée (par le gestionnaire d'entités). – Merge : effectue en cascade l'opération de fusion sur les entités associées si `merge()` est appelée ou si l'entité est supervisée. – Remove : effectue en cascade l'opération de suppression sur les entités associées si `delete()` est appelée.		Mapping de type multi valuées et mono valuées
Order by	Spécifie l'ordre des objets retournés par une requête : – No Ordering (liste non ordonnée). – Primary Key Order (liste triée par la clé primaire). – Custom Ordering (ordre spécifié).	Primay Key	Tout type de mapping, excepté la relation One-to-One
Inheritance	Une entité peut hériter des propriétés d'autres entités comme dans un modèle objet classique. Dali permet de spécifier une stratégie spécifique pour gérer ce type de relation : – Strategy : Dali propose trois types de stratégies d'héritage : Single Table (par défaut) : toutes les classes dans la hiérarchie sont mappées vers une et une seule table ; Joined table : la racine de l'arborescence est mappée vers une seule table, et toutes les entités filles mappent vers leur propre table ; One Table per class (une table par classe concrète) : chaque classe de la hiérarchie d'héritage est mappée vers une table. Cette stratégie prend en charge les associations de un vers plusieurs bidirectionnelles mais présente un certain nombre d'inconvénients, en particulier pour les relations polymorphes. La stratégie choisie est déclarée au niveau de la classe de l'entité la plus haute dans la hiérarchie en utilisant l'annotation @Inheritance. – Discriminator Column : utilisée pour spécifier le nom de la colonne discriminante si la stratégie d'héritage utilisée est de type Single Table ou Joined table. – Discriminator Type : utilisée pour positionner le type de différenciateur à Char ou Integer. La propriété Discriminator Value doit se confirmer à ce type (par défaut string). – Discriminator Value : spécifie la valeur discriminante utilisée pour différencier une entité dans la hiérarchie d'héritage (string, char, integer). La valeur doit être conforme à la valeur spécifiée dans Discriminator Type. Valeur par défaut string. Ce champs correspond à l'annotation @DiscriminatorValue.	Single Table	

Tableau 11.3 Propriétés du champ Map As *(suite)*

Propriété	Description	Valeur par défaut	S'applique au mapping suivant

Inheritance *(suite)*	Exemple
	`@Entity`
	`@Inheritance(strategy=InheritanceType.SINGLE_TABLE)`
	`@DiscriminatorColumn(`
	`    name="Vehiculetype",`
	`    discriminatorType=DiscriminatorType.STRING`
	`)`
	
	`@DiscriminatorValue("Vehicule")`
	`public class Vehicule { ... }`
	
	`@Entity`
	`@DiscriminatorValue("Passat")`
	`public class Xantia extends Vehicule { ... }`
	Dans cet exemple, la classe parente Vehicule définit la stratégie d'héritage (single table) et la colonne discriminante à l'aide de l'annotation @DiscriminatorColumn. L'annotation @DiscriminatorValue définit la valeur utilisée pour différencier la classe dans la hiérarchie. Le nom de la colonne discriminante est VehiculeType. @Inheritance et @DiscriminatorColumn devraient seulement être définies sur l'entité la plus haute de la hiérarchie.
		Single Table	

Primary Key Join Columns	Définit la clé primaire de la table de la classe fille jointe. Ce champ correspond à @PrimaryKeyJoinColumn.		

Dali identifie automatiquement par défaut les colonnes des entités construites comme mapping par défaut.

Il faut laisser le mapping basic pour chaque attribut des entités `Inventaire`, `Commande` et `Article`. Pour l'attribut dateCommande de l'entité `Commande`, il faut positionner le champ Temporal dans la vue JPA Details à Date. La balise @Temporal (DATE) est automatiquement ajoutée.

Mapping des relations interentités

À ce stade, vous devez modéliser les relations entre les différentes entités de notre modèle.

Relations mono-valuées

Mapping One-to-One

Dans votre modèle, le champ `article` de l'entité `Inventaire` possède une relation One-to-One vers l'entité `Article`, chaque article d'inventaire possédant un et un seul article.

1. Dans la vue Package Explorer, ouvrez la classe Inventaire.java.
2. Dans la même vue, sélectionnez le champ article de l'entité `Inventaire`. La vue JPA Details affiche les propriétés du champ.
3. Dans la liste déroulante Map As de la vue JPA Detail, sélectionnez la propriété One-to-One, et laissez les autres propriétés par défaut.

4. Cochez l'option Overide default, afin de spécifier la colonne de jointure. Dali propose la définition de jointure article_ARTICLEID->ARTICLEID.
5. Cliquez sur Edi pour éditer la jointure. La boîte de dialogue illustrée à la figure 11.17 s'affiche alors.

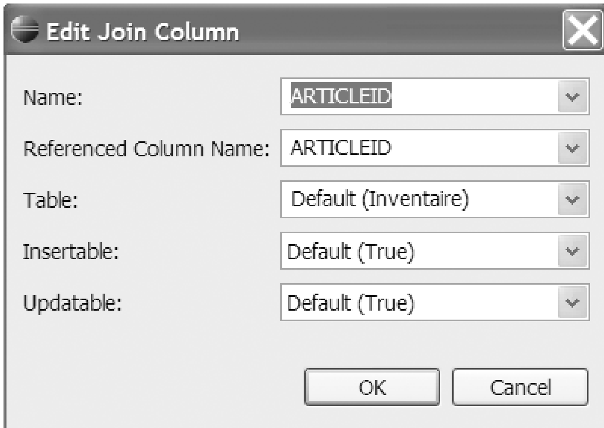


Figure 11.17

Assistant de définition de jointure Dali

6. Spécifiez `ARTICLEID` dans le champ Name. Dali génère le code suivant :

```
@Entity
@Table(schema="WEBSTOCK")
public class Inventaire {

@Id
@Column(table="Inventaire", name = "NUMEROINVENTAIRE", insertable = false, updatable = false)
private long numInventaire;

@OneToOne
@JoinColumn(name="ARTICLEID", referencedColumnName = "ARTICLEID")
protected Article article;
…
}
```

7. Dans la vue JPA Structure, un symbole s'affiche sur l'attribut `article`, preuve que la relation One-to-One a été prise en compte.
8. Sauvegardez la classe Article.java.

Pour transformer cette relation en One-to-One bidirectionnelle, vous devez ajouter un champ relation à l'entité `Article` afin de pointer en retour sur `Inventaire` :

```
@Entity
public class Article {
//…
@OneToOne (mappedBy="article")
protected Inventaire inventaire;
…
}
```

Mapping One-to-Many

Dans votre modèle, l'attribut commandes de l'entité `Client` possède une relation de type One-to-Many vers l'entité `Commande` (chaque client peut avoir plusieurs commandes). L'annotation @OneToMany est ajoutée a un attribut relation de type Collection, où l'entité à l'autre bout de la relation possède ou non un champ relation ou dispose d'une relation monovaluée (Many-to-One) qui pointe en retour sur l'entité.

1. Sélectionnez l'entité `Client` dans la vue Package Explorer.
2. Dans la vue JPA Structure, sélectionnez le champ commandes, puis, dans le champ Map As, choisissez One-to-Many.
3. Dans la liste Mapped By, sélectionnez le champ owner de l'entité `Commande`, comme illustré à la figure 11.18.

Figure 11.18

Mise en œuvre du mapping One-to-Many avec Dali

Remarquez le code généré par Dali :

```
@Entity
@Table(schema="WEBSTOCK")
public class Client {


    …
    @OneToMany(mappedBy="owner")
    protected Collection <Commande> commandes;

public Client() {
        commandes = new ArrayList<Commande>();
    }


…


}
```

La vue JPA Structure reflète le type de mapping One-to-Many associé au champ commandes (voir figure 11.19).

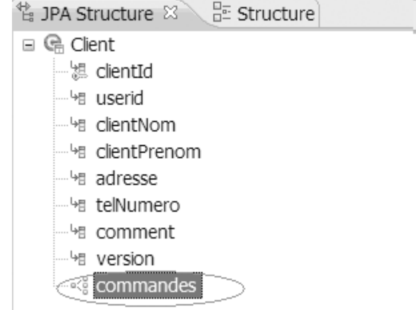


Figure 11.19

Vue JPA Structure après application du mapping One-to-Many

Mapping Many-to-One

Vous avez défini une relation de type « mapping retour » (back mapping), à partir de la relation One-to-Many définie précédemment. Ce type de mapping sera de type Many-to-One. Vous pouvez ainsi disposer d'une relation bidirectionnelle entre les deux entités.

Ce type d'association sera porté par l'attribut owner de l'entité `Commande`, que l'on pourrait résumer de la façon suivante : « Il existe plusieurs commandes qu'un client peut passer. »

1. Éditez l'entité `Commande` à partir de la vue Package Explorer, si ce n'est déjà fait.
2. Dans la vue JPA Structure, sélectionnez le champ owner de l'entité `Commande`.
3. Dans le champ Map As, sélectionnez Many-to-One, et cochez l'option Override Default.
4. Éditez la colonne de jointure pour qu'elle fasse référence explicitement à la clé étrangère CLIENTID de l'entité `Client` (voir figure 11.20), et laissez les autres champs inchangés.

JPA Details | SQL Results
Map As: Many to One
Target Entity: Default (com.webstock.chap10.model.Client)
Fetch: Default (Eager)
Optional: False
Cascade: All, Persist, Merge, Remove, Refresh
Override Default
Join Columns
CLIENTID -> clientId

Figure 11.20

Assistant d'édition de jointure externe Dali

5. Sauvegardez vos modifications.
6. Vérifiez la prise en compte par Dali de la relation ainsi définie en examinant le symbole porté par le champ owner.

Voici le code généré par Dali en fonction de la relation Many-to-One :

```
Public class Commande {
  @Id
  @Column (name= ?NUMEROCOMMANDE?)
  @GeneratedValue
  protected long numCommande;
  @Temporal (DATE)
  protected Date dateCommande;
  protected Article article;
  protected int quaantite;
  @Column (name=?ETATCOMMANDE?)
  protected String etat;
  @ManyToOne(optional=false)
 @JoinColumn(name="CLIENTID", referencedColumnName = "clientId")
 protected Client owner;

…
}
```

En utilisant un type de collection générique, comme `<Commande>`, Dali est capable de déterminer le type d'entité à l'autre bout de la relation, avantage appréciable pour la productivité du développeur JEE. Tout ce qui reste à résoudre en termes de référence de mapping pour la partie @OneToMany de la relation est le nom de la propriété sur cette entité, dans votre cas owner.

Mise en œuvre du mapping de version

Vous avez déjà rencontré l'annotation @Version, qui permet d'ajouter un contrôle de concurrence optimiste à un bean entité.

Dans cette section, vous allez la mettre en œuvre sur les différentes entités qui composent votre modèle. Ce type de stratégie d'optimisation doit être pensé en fonction des contraintes de votre projet et réexaminé ensuite lors des tests de recette et d'exploitation de l'application.

1. Sélectionnez l'entité `Client` dans la vue Package Explorer.
2. Dans la vue JPA Structure, sélectionnez le champ version de l'entité `Commande`. Cela a pour effet d'afficher les détails de la propriété dans la vue JPA Details.
3. Dans le champ Map As de la vue JPA Details, sélectionnez dans la liste déroulante la valeur Version. Le code suivant est ajouté à l'entité `Client` :

```
@Version
protected int version ;
```

Cette modification a pour effet d'ajouter un symbole devant l'attribut Version dans la vue JPA Structure.

Vous avez achevé le mapping des entités de votre modèle. Vous pouvez passer à l'étape de définition des requêtes nommées (named queries) associées aux entités du modèle métier.

Définition des requêtes nommées de l'interface Query

Au chapitre précédent, vous avez abordé les requêtes dynamiques avec Query à l'aide de l'interface EntityManager. Dans ce chapitre, vous allez découvrir les requêtes nommées.

Une requête nommée est définie par l'annotation @NamedQuery, qui définit le nom de la requête. Contrairement aux requêtes dynamiques, conçues à l'aide de l'API Query supportée par JPA, celles-ci ont l'avantage de pouvoir être précompilées lors du déploiement.

Vous pouvez mapper des requêtes JPQL/HQL en utilisant les annotations. @Named-Query et @NamedQueries peuvent être définies au niveau de la classe ou dans un fichier JPA XML. Leurs définitions sont globales au scope de la Session Factory/Entity Manager Factory.

Une requête nommée est définie par son nom et la chaîne de caractères de la requête réelle précédée de l'attribut query, comme dans l'exemple suivant :

```
@Entity
@NamedQueries({
  @NamedQuery(name = "Commande.findAll",
              query = "select o from Commande o"),
  @NamedQuery(name = "findCommandeByNumCommande",
              query = "select o from Commande o where o.numcommande = :numcommande"),
  @NameQuery (name = "findCommandeByEtat",
              Query="select o from Commande o where o.etat=?4")
})
@Table(name="COMMANDE")
public class Commmande
{
…
}
```

Une entité peut déclarer des instructions JPQL au sein des annotations @NamedQuery pour définir des requêtes « réutilisables ». Les noms @NamedQuery définis doivent être uniques au niveau de l'unité de persistance. La requête findCommndeByNumCommande possède un paramètre, numcommande, mais elle peut également prendre un paramètre indexé, comme la requête findCommandeByEtat précédente.

Liaison des paramètres de la requête

Les requêtes nommées peuvent prendre des paramètres lors de leur invocation ou des paramètres indexés. En supposant que la requête FindClientByUserid définie plus haut soit appelée à partir d'un code client de type bean session, vous aurez :

```
@Stateless
public class GestionCommandeClient implements CommandeMgr
{

  @PersistenceContext(unitName = "MonProjetJPA-Unit")
  private EntityManager em;
…

  /** <code>select o from Commande o</code> */
  public List<Commande> CommandeFindAll() {
    return em.createNamedQuery("Commande.findAll").getResultList();
  }
```

```
  /** <code>select o from Commande o where numcommande = :numcommande</code> */
  public List <Commande> FindByNumCommande(Object numcommande) {
    return em.createNamedQuery("Client.findCommandeByNumeroCommande")
     ➥.setParameter("numcommande",numcommande).getResultList();
  }
}
```

Intégration des entités du modèle logique et mise en œuvre d'un bean client façade

Une fois le modèle logique conçu avec les différentes entités correspondantes selon une approche de conception top-down ou bottom-up (conception partant du modèle logique ou inversement partant du schéma physique sous-jacent), il importe d'invoquer vos entités conçues avec Dali à partir d'un client Java.

Pour ce type de conception, vous vous appuierez sur un classique bean session, qui va servir de façade pour vos entités du modèle. Le bean session `CommandeManager` (voir le code source complet sur la page Web dédiée à l'ouvrage) expose les opérations CRUD sous forme de service, permettant aux clients d'accéder aux entités `Client` et `Commande`.

Vous pourrez implémenter selon le même canevas les opérations de traitement CRUD entre les entités `Inventaire` et `Article`, mais en utilisant cette fois la relation One-to-One. Les services offerts par le bean session façade autorisent la gestion transactionnelle et le maintien de la liaison entre les entités du domaine et la base sous-jacente.

Entités Client et Commande

Rappelons les relations de mapping supportées par les entités `Client` et `Commande` du modèle :

```
@Entity
@NamedQueries({
@NamedQuery(name = "Client.findAll", query = "select o from Client o"),
@NamedQuery(name = "Client.findByUserid", query = "select o from Client o where
➥userid = :userid")
})
public class Client
{

@Idw
@Column(table="Client")
@GeneratedValue
protected int clientId;
protected int userid;
protected String clientNom;
protected String clientPrenom;
protected String adresse;
@Column(name="TEL")
protected String telNumero;
protected String comment;
@Version
protected int version ;

@OneToMany(mappedBy="owner")
```

```
protected Collection <Commande> commandes;


public int getClientId() {
    return clientId;
}

public void setClientId(int clientId) {
    this.clientId = clientId;
}

public int getUserid() {
    return userid;
}

public void setUserid(int userid) {
    this.userid = userid;
}

public String getClientNom() {
    return clientNom;
}

public void setClientNom(String clientNom) {
    this.clientNom = clientNom;
}

public String getClientPrenom() {
    return clientPrenom;
}

public void setClientPrenom(String clientPrenom) {
    this.clientPrenom = clientPrenom;
}

public String getAdresse() {
    return adresse;
}

public void setAdresse(String adresse) {
    this.adresse = adresse;
}

public String getTelNumero() {
    return telNumero;
}

public void setTelNumero(String telNumero) {
    this.telNumero = telNumero;
}

public String getComment() {
    return comment;
}

public void setComment(String comment) {
    this.comment = comment;
}
```

```
public int getVersion() {
    return version;
}

public void setVersion(int version) {
    this.version = version;
}

public Collection <Commande> getCommandes() {
    return commandes;
}

public void setCommandes(Collection <Commande> commandes) {
    this.commandes = commandes;
}


}
@Entity
@Table(schema="WEBSTOCK")
public class Commande {

@Id
@Column(name="NUMEROCOMMANDE")
@GeneratedValue
protected long numCommande;
@Temporal(DATE)
protected Date dateCommande;
protected Article article;
protected int quantite;
@Column(name="ETATCOMMANDE")
protected String etat;

@ManyToOne(optional=false)
@JoinColumn(name="CLIENTID", referencedColumnName = "clientId")
protected Client owner;


public long getNumCommande() {
    return numCommande;
```

```
}
public void setNumCommande(long numCommande) {
    this.numCommande = numCommande;
}
public Date getDateCommande() {
    return dateCommande;
}
public void setDateCommande(Date dateCommande) {
    this.dateCommande = dateCommande;
}
public Article getArticle() {
    return article;
}
public void setArticle(Article article) {
    this.article = article;
}
```

```
public int getQuantite() {
    return quantite;
}
public void setQuantite(int quantite) {
    this.quantite = quantite;
}
public String getEtat() {
    return etat;
}
public void setEtat(String etat) {
    this.etat = etat;
}
public Client getOwner() {
    return owner;
}
public void setOwner(Client owner) {
    this.owner = owner;
}

}
```

Le bean session CommandeManager

Le bean session `CommandeManager` sert de façade aux beans entité `Client` et `Commande` vus précédemment. Il offre une interface aux opérations du gestionnaire d'entités `persist()`, `merge()` et `remove()` :

```
@Stateless(mappedName="CommandeManager")
public class CommandeManager
  implements CommandeMgr
{
  @PersistenceContext(unitName = "MonProjetJPA-Unit")
  private EntityManager em;

  public CommandeManager() {
  }

  public Object mergeEntity(Object entity) {
    return em.merge(entity);
  }

  public Object persistEntity(Object entity) {
    em.persist(entity);
    return entity;
  }

  /** <code>select o from Client  o</code> */
  public List<Client> queryClientFindAll() {
    return em.createNamedQuery("Client.findAll").getResultList();
  }

  public void removeClient(Client client) {
    customer = em.find(Client.class, client.getClientId());
    em.remove(client);
  }
```

```
  /** <code>select o from Client o</code> */
  public List<Commande> queryCommandeFindAll() {
    return em.createNamedQuery("Commande.findAll").getResultList();
  }

  public void removeCommande(Commande commande) {
    commande = em.find(Commande.class, commande.getNumCommande());
    em.remove(commande);
  }

    /** <code>select o from Client o where userid = :userid</code> */
  public List<Client> queryClientFindByUserid(Object userid) {
    return em.createNamedQuery("client.findByuserid").setParameter("userid",
    userid).getResultList();
  }
}


import java.util.List;
import javax.ejb.Remote;

@Remote
public interface CommandeMgr
{
  Object mergeEntity(Object entity);

  Object persistEntity(Object entity);

  List<Client> queryClientFindAll();

  void removeClient(Client client);

  List<Commande> queryCommandeFindAll();

  void removeCommande(Commande commande);

    List<Client> queryClientFindByuserid(Object userid);
}
```

Voici le fichier de persistance associé, adapté à un déploiement final sur la base de données Derby :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi
➥="http://www.w3.org/2001/XMLSchema-instance" version="1.0" xsi:schemaLocation
➥="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/
➥persistence_1_0.xsd">
<persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
<class>org.eclipse.dali.example.jsf.inventory.model.Inventory</class>
    <class>com.webstock.chap10.Article</class>
         <class> com.webstock.chap10..Inventaire>
    <class> com.webstock.chap10.Commande </class>
         <classe> com.webstock.chap10.Client </class>

    <properties>
        <property name="toplink.logging.level" value="FINEST"/>
        <property name="toplink.jdbc.driver" value
        ➥="org.apache.derby.jdbc.ClientDriver"/>
```

```
        <property name="toplink.jdbc.url" value="jdbc:derby://localhost:1527/WebStockDB;
        ➥create=true"/>
        <property name="toplink.jdbc.user" value="dali"/>
        <property name="toplink.jdbc.password" value="dali"/>
    </properties>
</persistence-unit>

</persistence>
```

Enfin, voici un extrait du code client de test invoqué à partir d'une instance d'une JVM sur les méthodes distantes du bean :

```
public class CommandeClient
{
  public static void main(String [] args) {
    try {
      final Context context = getInitialContext();
      CommandeMgr commandeMgr = (CommandeMgr)context.lookup
      ➥("java:comp/env/ejb/CommandeManager");
      // Invocation des methodes distantes du bean
      commandeMgr.mergeEntity(  entity );
      commandeMgr.persistEntity(  entity );
      System.out.println(commandeMgr.queryClientFindAll(  ) );
      System.out.println(commandeMgr.queryCommandeFindAll(  ) );
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }
…
}
```

Vous pouvez récupérer sur la page Web dédiée à l'ouvrage le code complet de ces extraits et les déployer sur le serveur JBoss 4 en utilisant les scripts Ant de déploiement prévus.

En résumé

Ce chapitre vous a permis de mettre en œuvre l'outillage Dali pour le mapping O/R ainsi que les concepts ayant trait à l'utilisation de l'API JPA.

Le chapitre suivant vous fournira l'occasion de mettre à profit tous ces concepts à la lumière d'une approche de développement centrée sur les modèles et automatisée, avec la solution EclipseUML de l'éditeur Omondo.

12

Mise en œuvre de l'application webstock

Dans ce chapitre, vous allez mettre en œuvre l'étude de cas webstock à l'aide des technologies EJB3/JSF, en suivant une démarche de développement MDA (Model Driven Architecture), ou pilotée par le modèle.

Vous utiliserez pour l'outil de conception EclipseUML, de l'éditeur Omondo *(http://www.omondo.com)*. Ce dernier couvre désormais l'intégralité du cycle de développement JEE et permet de générer une grande partie du code de l'application grâce au générateur de code Java AndroMDA.

Afin de construire l'étude de cas avec EJB3, vous vous appuierez sur le projet Web Tools. Vous compléterez ainsi votre panoplie du « parfait développeur » JEE.

Rappelons qu'EclipseUML Studio, disponible sur plate-forme Windows/Linux et Eclipse 3.3 Europa, est entièrement gratuit. La version payante Eclipse Studio offre toutefois la possibilité d'exporter certains diagrammes et de générer une documentation projet.

Un certain nombre de concepts clés comme la notion de méthodologie MDA, de stéréotypes, de profils UML et de modélisation UML2 seront rappelés. Pour en savoir plus, voir la documentation mentionnée en annexe.

L'architecture MDA

La démarche de développement MDA, concept très en vogue dans les projets J2EE/JEE actuels, sera mise en œuvre dans ce chapitre avec l'atelier EclipseUML for JEE.

Soutenue par l'OMG (Object Management Group), MDA (Model Driven Architecture) vous permet de modéliser votre application indépendamment de son implémentation cible (niveau matériel ou logiciel), permettant ainsi une bonne réutilisation de vos modèles.

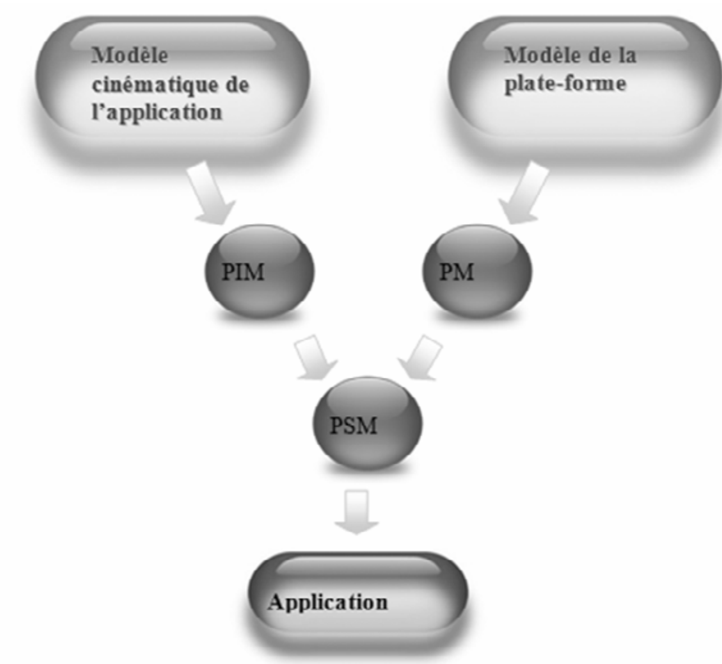
Cette approche de développement couvre l'ensemble du cycle du projet, de la conception au déploiement. Elle s'appuie sur une association des modèles créés, ou PIM (Platform Independant Model), qui représentent l'aspect métier, avec des modèles PM (Platform Model),

transformés pour obtenir *in fine* un modèle d'application spécifique, dit PSM (Platform Specific Model).

La figure 12.1 illustre l'architecture de l'approche MDA.

Figure 12.1

Approche MDA



Des outils de génération automatique de code permettent de créer le programme directement à partir des modèles.

En résumé, la mise en œuvre de MDA est entièrement fondée sur les modèles et leurs transformations. Le fait qu'elle soit indépendante de toute plate-forme assure un développement portable des langages et technologies sous-jacentes.

Méthodologie de mise en œuvre

L'approche MDA se déroule en deux grandes étapes :

1. **Conception.** Consiste à réaliser un modèle indépendant de toute plate-forme (PIM) exprimé en UML. Pour obtenir ce modèle, il faut choisir la ou les plates-formes d'exécution (plusieurs plates-formes peuvent être utilisées pour mettre en œuvre un même modèle).

2. **Transformation.** Consiste à projeter un modèle PIM suffisamment détaillé vers un modèle spécifique PSM. Les caractéristiques d'exécution et les informations de configuration qui ont été définies de façon générique sont alors converties pour tenir compte des spécificités de la plate-forme cible (dans notre cas EJB3/JSF/JPA).

La génération du code s'effectue à l'aide de composants cartridge (cartouches), chaque cartouche étant conçue pour générer du code Java pour les technologies J2EE/JEE standards (Struts, Spring, etc) et d'autres plates-formes.

EclipseUML for JEE

EclipseUML est un projet Eclipse développé sous la tutelle de la société Omondo, membre contributif de l'OMG (Object Management Group). L'outil est décliné en deux éditions, Free et Studio.

EclipseUML s'appuie sur le standard UML2 de l'OMG. Les premières générations de cet outil étaient destinées à la modélisation UML et à la génération de code Java. Il propose aujourd'hui neuf diagrammes UML, en particulier les diagrammes de classes, les cas d'utilisation et les diagrammes d'état. En juin 2007, EclipseUML Europa a vu le jour pour offrir au monde JEE un atelier MDA pratique et complet, couvrant l'intégralité du cycle de développement et permettant un reverse JEE à partir d'un modèle existant.

La figure 12.2 illustre le cycle de génération de code avec EclipseUML incluant AndroMDA.

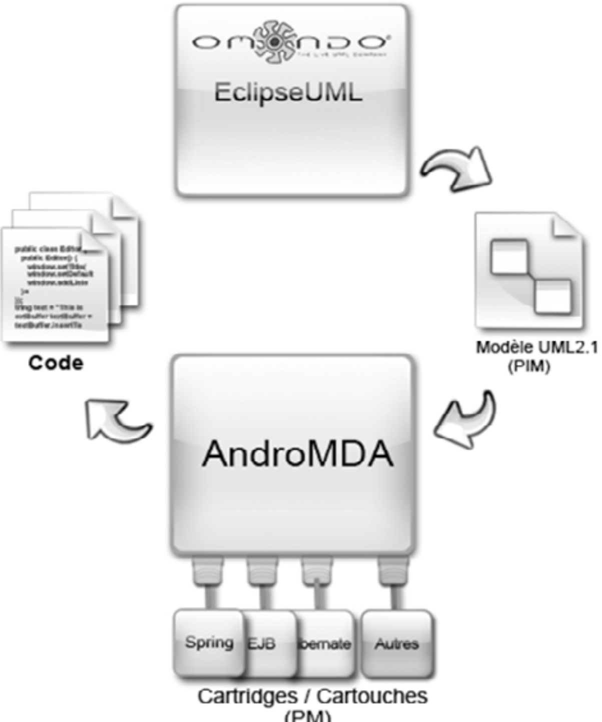


Figure 12.2

Architecture d'EclipseUML (source Omondo)

AndroMDA est un puissant générateur, qui permet, à partir d'un modèle métier spécifié en UML, de générer une grande partie des couches nécessaires à la construction d'une application Java, et ce indépendamment de la technologie sous-jacente.

La génération de code commence par la modélisation d'un PIM (Plateform Independant Model), généralement à l'aide d'UML. Ce PIM est ensuite traduit en PSM à l'aide des cartouches du générateur de code (templates). Le PSM représente le code de l'application.

Les cartouches génèrent du code en fonction du stéréotype et des valeurs balisées appliquées aux éléments du modèle. Comme vous le verrez, un ensemble de profils UML ont été développés pour désigner ces éléments et les classifier.

> **Notion de profil**
>
> Un profil est une extension du métamodèle UML permettant d'ajouter de nouveaux types et d'inclure des caractéristiques relatives à un domaine particulier. C'est un jeu de stéréotypes, de valeurs balisées et de contraintes OCL (Object constraint langage) permettant d'affiner les éléments du modèle. AndroMDA a défini un profil pour chacune des couches des applications JEE. Vous les utiliserez au fur et à mesure de la modélisation de l'application webstock.

La figure 12.3 donne un aperçu de l'atelier EclipseUML :

- Zone 1 : éditeur de classes de l'atelier.
- Zone 2 : vue Outline panoramique de l'éditeur ouvert en mode édition.
- Zone 3 : affiche le code Java de la classe sur laquelle l'utilisateur opère. Rappelons qu'EclipseUML assure une synchronisation entre le code et le modèle.
- Zone 4 : vue Front and Colors permettant d'éditer l'aspect visuel des éléments des différents diagrammes.
- Zone 5 : vue Properties offrant un accès rapide aux propriétés des éléments du diagramme de classes.
- Zone 6 : vue Explorateur des modèles des projets de modélisation.

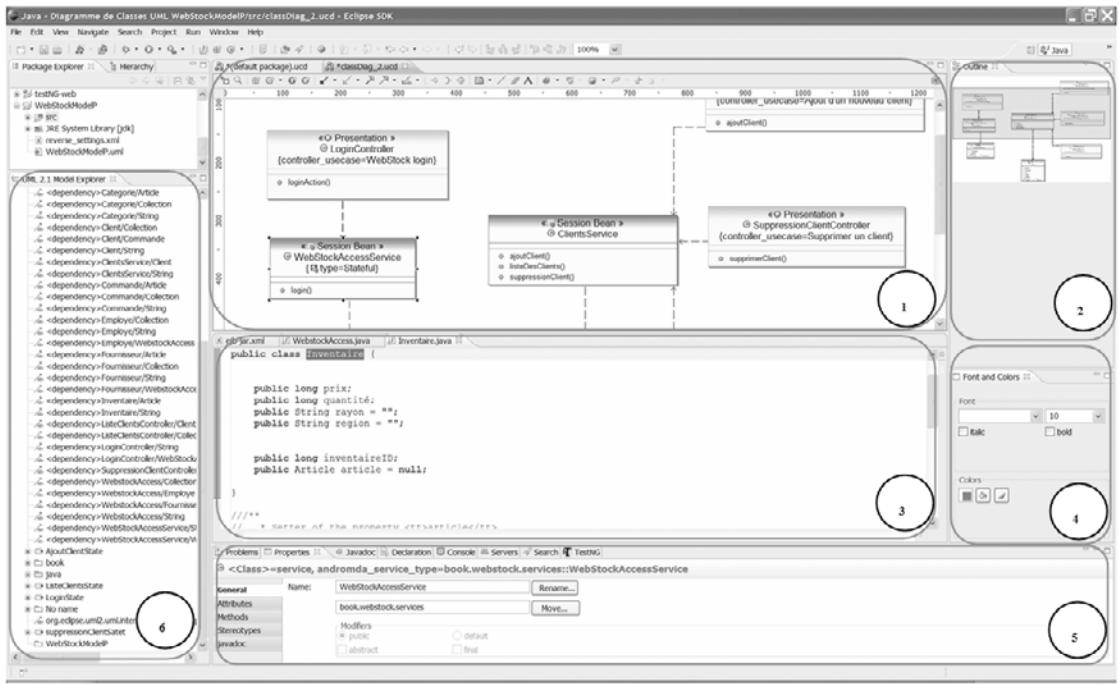


Figure 12.3

Atelier EclipseUML

> **AndroMDA**
>
> Il est recommandé de se documenter sur l'outil de génération AndroMDA *(http://www.andromda.org)*, qui s'appuie sur un certain nombre de stéréotypes. Voir en particulier l'excellente traduction de *Getting Started with AndroMDA for Java* par Sébastien Arbogast, disponible à l'adresse *ftp://ftp-developpez.com/sarbogast/fichiers/andromda-intro.pdf.*

L'application webstock

Webstock est une application simple de gestion d'un stock d'articles destinée à illustrer les aspects avancés du développement JEE.

Cette application couvre les besoins de la chaîne de vente d'articles et de pièces détachées informatiques WebStore. Ces besoins peuvent s'élargir à la consultation du système central de la chaîne de magasins.

Dans le cadre des étapes de création de l'application, le cycle MDA devrait comprendre les itérations suivantes :

- Modélisation UML du contexte métier du domaine webstock
- Sérialisation et export XMI du modèle UML
- Génération de code
- Déploiement et tests

Dans un premier temps, vous allez vous focaliser sur la modélisation de l'application et sur le modèle métier correspondant.

Environnement de travail

Votre environnement de développement sera Eclipse 3.3 avec Web Tools.

Pour les besoins de l'étude de cas, les prérequis sont les suivants :

- EclipseUML version free pour Europa, téléchargeable à l'adresse *http://www.ejb3.org/.*
- Web Tools 2, la version compatible Eclipse 3.3, récupérable à partir du site Eclipse, à l'adresse *http://www.eclipse.org/Web Tools/.*
- JBoss 4.2.1, pour un déploiement supportant EJB3.

Modélisation avec EclipseUML

Dans cette section, vous allez définir la conception de l'application webstock avec l'éditeur EclipseUML.

L'architecture *n*-tiers de l'application sera illustrée par différents diagrammes.

Architecture *n*-tiers

À la génération du code, EclipseUML fournit l'ossature d'une application *n*-tiers selon les bonnes pratiques JEE (en particulier en s'appuyant sur les principaux modèles de conceptions existants, dont un certain nombre sont abordés dans ce chapitre).

Le modèle MVC est inhérent à votre application. Pour une meilleure performance de la démarche MDA, vous vous focaliserez sur la modélisation des différents comportements de chacune des couches. Pour ce faire, vous utiliserez les trois diagrammes de base suivants :

- diagrammes de use cases, pour expliciter les points d'entrée de l'application et son interaction avec l'utilisateur physique ;
- diagrammes de classes, pour exprimer le modèle métier de l'application ainsi que les DAO ;
- diagrammes d'état, pour expliciter le déroulement du processus métier pour chacun des use cases du modèle.

Modèle métier

Pour créer votre modèle métier de l'application webstock composée de beans entité EJB3 (voir figure 12.4), commencez par créer un projet Java usuel à l'aide d'Eclipse.

1. Dans le répertoire source, créez un nouveau répertoire appelé, par exemple, book.webstock.entities. Ce dossier contiendra l'ensemble de vos entités.
2. Dans la vue Package Explorer, faites un simple clic sur le dossier que vous venez de créer.
3. Cliquez sur File, New, Other, UML Diagrams et UML Class Diagram.
4. Saisissez le nom de votre diagramme.
5. Dans l'éditeur qui s'affiche, modélisez vos entités comme illustré à la figure 12.3.

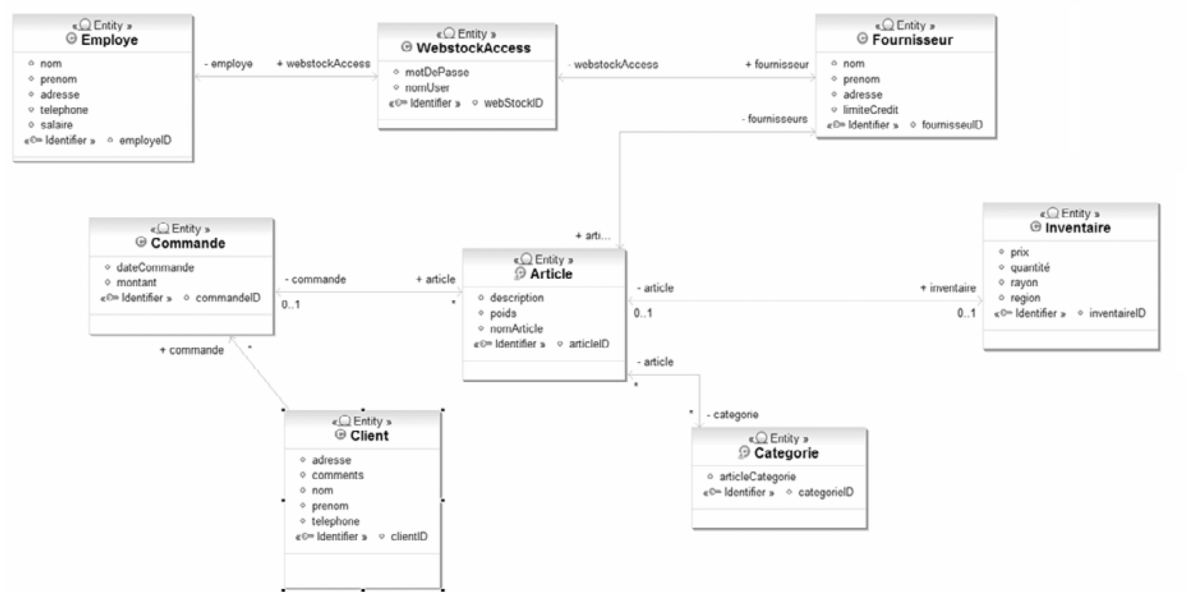


Figure 12.4

Modèle métier de l'application webstock

Création des beans entité EJB3

Un bean entité EJB3 sous EclipseUML est une classe portant le stéréotype Persistence::Entity.

1. Dans la palette de l'éditeur du diagramme de classes en cours, cliquez sur le bouton Création d'une classe.
2. Dans la fenêtre qui s'affiche, entrez le nom de votre entité.
3. Pour appliquer le stéréotype correspondant, faites un clic droit sur la classe EJB.
4. Dans le menu qui s'ouvre, cliquez sur properties.
5. Dans l'onglet Stereotype de la fenêtre properties, cliquez sur New Stereotype pour choisir le stéréotype à appliquer : pour les entités Persistence::Persistence.

Ces étapes sont résumées à la figure 12.5.

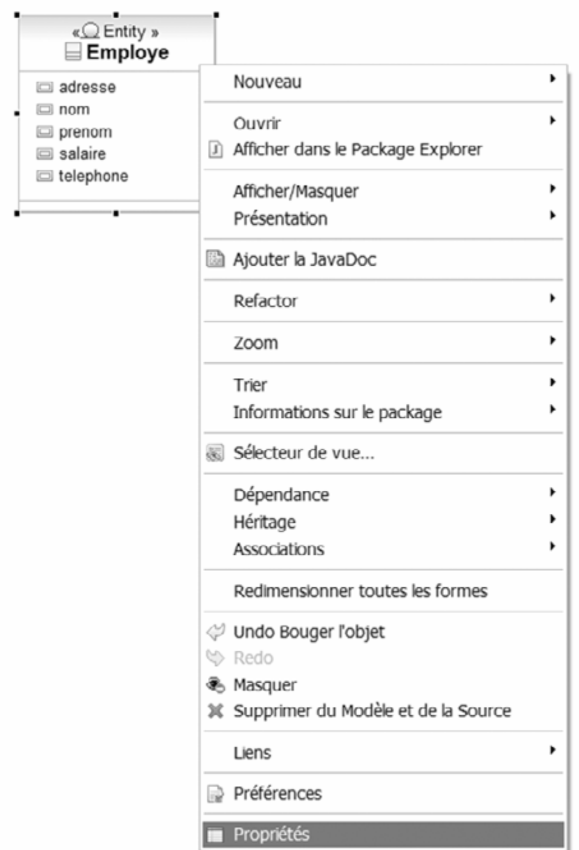


Figure 12.5

Création d'un bean entité

Cas d'utilisation

Vous allez décomposer votre modèle en un ensemble cohérent de cas d'utilisation.

Tout cas d'utilisation doit porter le stéréotype FrontEndUseCase. Un seul d'entre eux sera apte à porter le stéréotype FrontEndApplication. Ce stéréotype désignera la page d'accueil, home ou index, de votre application, ici la page Liste des clients.

Plusieurs cas d'utilisation sont à prévoir pour l'application webstock, notamment les suivants :

- Ajout/Suppression et affichage de chacune des entités de l'application WebStock.
- Login, permettant de se connecter *via* un login/password à l'application.

Pour créer ces use cases avec EclipseUML, procédez comme suit :

1. Cliquez sur File, New, Other et UML Use Case Diagram.
2. Entrez le nom du diagramme dans la fenêtre qui s'affiche.
3. Cliquez sur le bouton Package de la palette de l'éditeur du diagramme de use case pour créer un nouveau package destiné à contenir un de vos cas d'utilisation.
4. Une fois le package créé, dessinez le use case à l'intérieur de celui-ci.
5. Appliquez le stéréotype Presentation::FrontEndUseCase à partir de la vue properties.
6. Appliquez le stéréotype Presentation::FrontEndApplication pour désigner le use case représentant le point d'entré par défaut de votre application (voir figure 12.6).

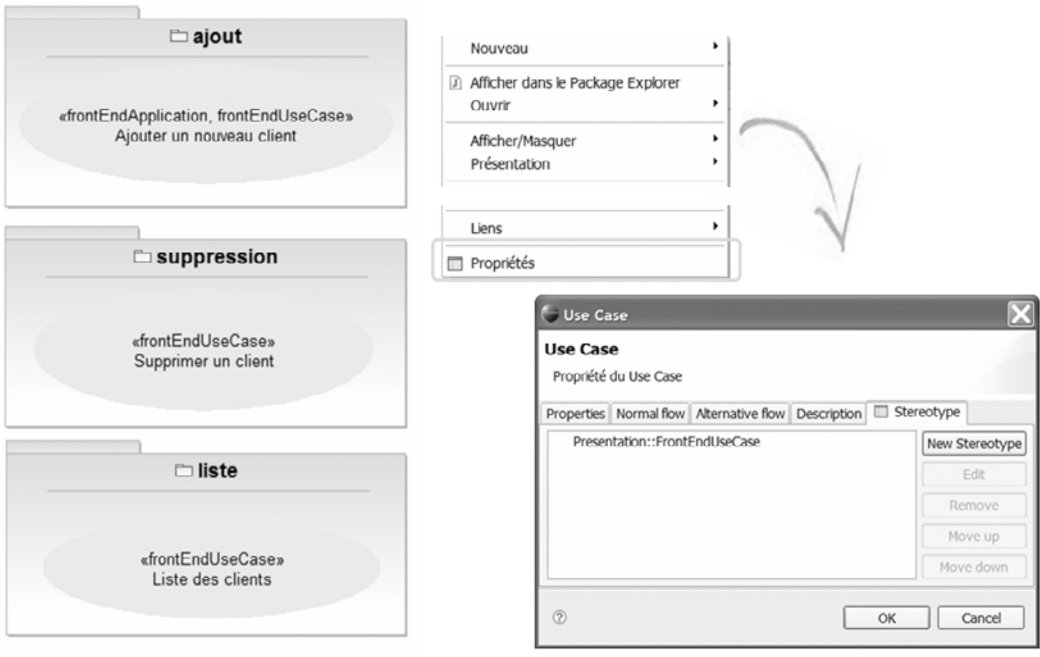


Figure 12.6

Stéréotype d'un cas d'utilisation

Pensez à donner des noms explicites à vos use cases, car ceux-ci vont représenter les éléments du menu général de l'application.

Services de l'application webstock

L'utilisation de beans session permet de proposer des services à la couche métier.

Vous utiliserez le classique pattern façade qu'implémentent les beans session pour fournir une interface pour les beans entité afin de limiter le nombre d'appels distants à ces objets.

Vous créerez, pour chacun des beans entité, un bean « service » qui interfacera tous leurs accès. Les services représenteront la passerelle entre la couche présentation et la couche métier de l'application.

Citons parmi eux les services ClientsService et WebStockAccessService :

- ClientService est un bean session sans état interfaçant l'entité `Client` pour encapsuler les appels CRUD à cet objet.
- WebStockAccessService est un bean session avec état permettant de se connecter à l'application webstock *via* un login/mot de passe.

> **Palette EclipseUML**
> Pour modéliser ces beans, EclipseUML offre une palette intuitive et configurable mettant à disposition de nouveaux éléments déjà stéréotypés pour EJB, Hibernate, JSF, Struts et les Web Services.

Pour activer un de ces menus, il suffit de faire un clic droit sur le fond du diagramme puis, dans le menu Préférences PSM, de choisir les technologies qui vous intéressent. Dans votre cas, vous activerez EJB pour AndroMDA et JSF.

La figure 12.7 illustre la palette EJB de l'éditeur de diagramme de classes EclipseUML.

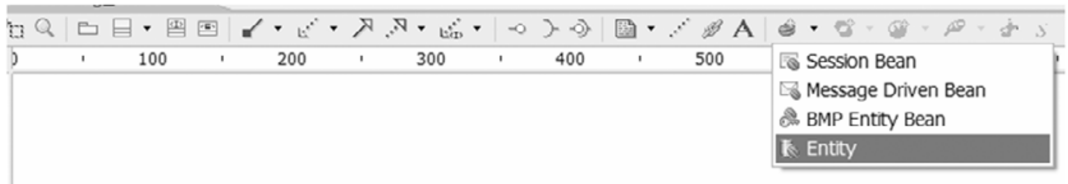


Figure 12.7

Palette EJB de l'éditeur de diagramme de classes EclipseUML

Pour modéliser ces services, vous pouvez également dessiner une classe Java usuelle en lui appliquant le stéréotype Service::Service à partir du menu Properties.

Contrôleurs de l'application webstock

Les contrôleurs sont des classes permettant de faire communiquer la couche Web et la couche métier. Vous allez modéliser un contrôleur pour chacune des requêtes susceptibles d'être lancées par l'utilisateur.

Vous disposerez ainsi d'un contrôleur pour chaque use case, comme l'illustre la figure 12.8.

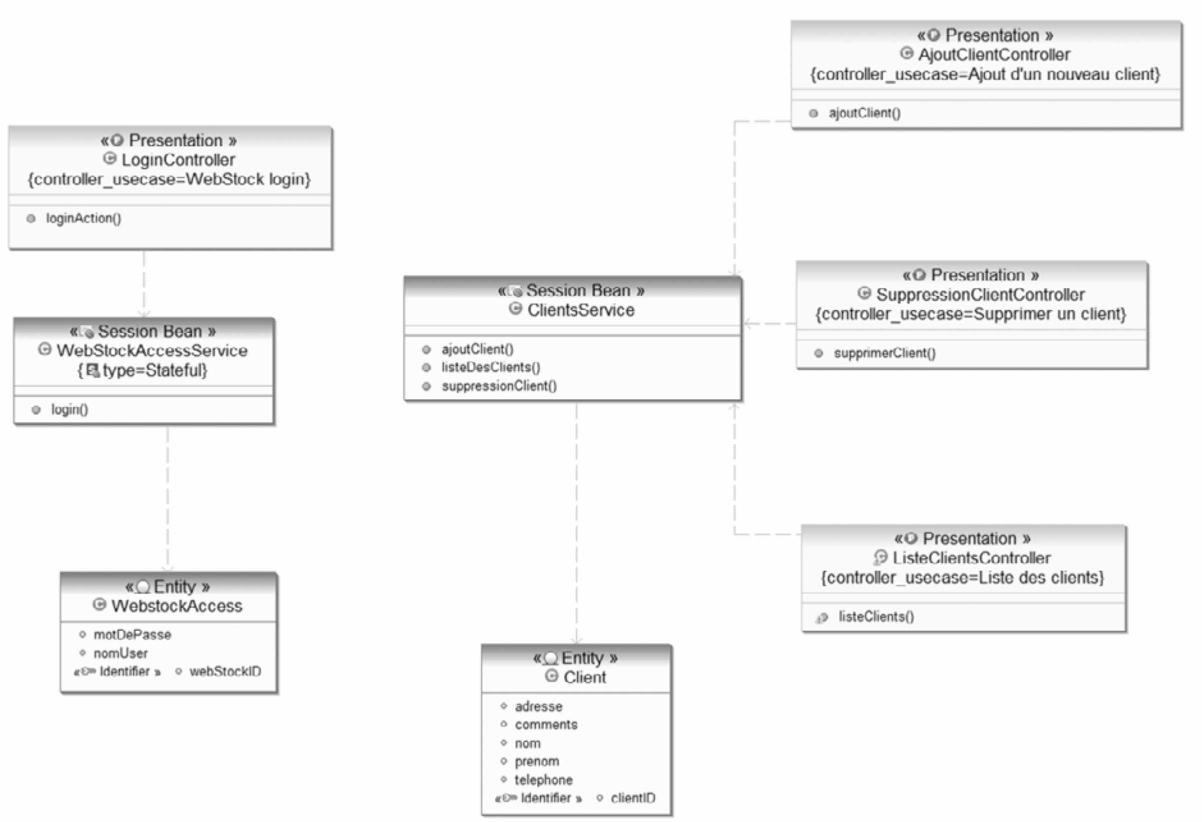


Figure 12.8

Entités, services et contrôleurs

1. Faites un clic droit sur la classe représentant le contrôleur dans le diagramme de classes.
2. Dans le menu pop-up qui s'affiche, choisissez Liens et Lien Interface Web.
3. Dans la fenêtre qui s'ouvre, cherchez dans le menu déroulant le use case en question pour le sélectionner.
4. Cliquez sur OK. Un nouveau stéréotype va être appliqué à cette classe, portant l'information du use case dont elle assure le traitement.

La figure 12.9 illustre l'attribution du contrôleur ListeClientsController au cas d'utilisation Liste des clients.

Figure 12.9

Attribution du contrôleur ListeClientsController au cas d'utilisation Liste des clients

Diagrammes d'état

Vous allez réaliser un diagramme d'état pour chacun de vos cas d'utilisation afin d'expliciter leur comportement et de redessiner l'aspect dynamique du système.

La palette graphique EclipseUML pour les diagrammes d'état est assez riche. Elle donne accès à plusieurs des éléments illustrés à la figure 12.10.

Figure 12.10

Palette graphique de l'éditeur de diagrammes d'état EclipseUML

Le tableau 12.1 récapitule les éléments de la palette.

Tableau 12.1 Éléments de la palette graphique de l'éditeur de diagrammes d'état

Symbole	Fonction
	Création d'un point de départ
	Création d'un point de fin
	Création d'un état
	Création d'un history
	Création d'un point d'option
	Création d'un point de sortie
	Création d'un terminer
	Création d'un point d'entrée
	Création d'une jointure
	Création d'un fork
	Création d'une transition
	Création d'une nouvelle note

Rappelons quelques notions de base sur les diagrammes d'état :

- Un diagramme d'état doit obligatoirement avoir un point de départ, lequel doit posséder une ou deux transitons qui en dérivent.
- Chaque état du diagramme représente un état donné du processus métier à modéliser.
- Les états qui demandent une interaction utilisateur avec le système doivent porter le stéréotype Presentation::FrontEndView.
- Les états qui désignent un comportement côté serveur ne sont pas censés porter un stéréotype.
- Vous devez ajouter un événement, ou « effect », aux transitions issues d'un état stéréotypé FrontEndView. Un effect doit passer des paramètres de l'état source à l'état destination.
- Les stéréotypes appliqués sur les attributs d'une transition d'un état côté client spécifient le type d'affichage de ceux-ci dans le formulaire correspondant.
- S'il existe un état final, il doit porter le nom du use case implémenté par le diagramme.

La figure 12.11 illustre le diagramme d'état associé au cas d'utilisation Ajout d'un nouveau client.

La figure illustre vos trois états intermédiaires : Début Ajout Client, Formulaire ajout client et Persister nouveau client. Le deuxième représente un formulaire de saisie des paramètres du nouveau client et est donc associé à la couche de présentation. Pour cette raison, lui est appliqué le stéréotype Presentation::FrontEndView.

Dans le troisième état, vous allez exécuter la méthode ajoutClient du contrôleur AjoutClientController, qui implémente le cas d'utilisation Ajout d'un nouveau client.

Pour avoir accès à cette méthode *via* ce diagramme d'état, il faut d'abord attribuer ce diagramme au cas d'utilisation qu'il représente. Il suffit pour cela de faire un clic droit sur le cas d'utilisation, puis, dans le menu déroulant, de choisir Liens et Liens Struts/JSF, et enfin de sélectionner le diagramme d'état désigné.

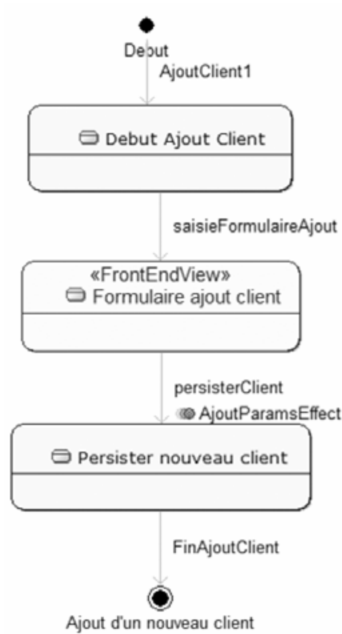


Figure 12.11

Diagramme d'état du cas d'utilisation Ajout d'un nouveau client

La figure 12.12 illustre ce processus.

Pour réaliser votre diagramme d'état, procédez de la façon suivante :

1. Commencez par dessiner les différents états du diagramme en utilisant les éléments de la palette de l'éditeur correspondant.
2. Créez les différents états (environ cinq) concernant le processus Ajout client, y compris les états initiaux et finaux.
3. Appliquez le stéréotype FrontEndView à l'état Formulaire ajout client *via* le menu Properties accessible par clic droit sur celui-ci (voir figure 12.12).

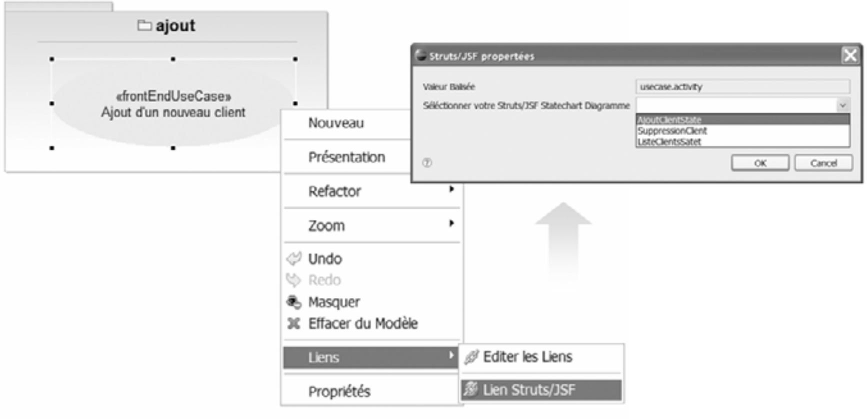


Figure 12.12

Attribution d'un diagramme d'état à un cas d'utilisation

4. Créez les transitions entre les différents états.

   Puisque l'état Formulaire ajout client représentera une page Web, il devra passer des paramètres à l'état suivant, lesquels représentent les champs du formulaire affichés sur cette page. Pour ce faire, vous devez ajouter un effet sur la transition qui en dérive.

5. Pour ajouter un effet à la transition persisterClient, faites un clic droit sur la transition, puis choisissez Ajouter Behavior et Effect.

6. Donnez un nom à l'effet, par exemple AjoutParamsEffect.

   Dans l'état Persister Nouveau Client, vous allez exécuter la méthode AjoutClient du contrôleur AjoutClientController. Cette méthode utilisera les paramètres transmis par la transition entrante à laquelle vous venez d'attribuer l'effet.

7. Ouvrez la fenêtre Properties de l'état en question, accessible *via* son menu pop-up correspondant.

8. Choisissez l'onglet Connection pour ajouter une nouvelle activité qui représentera la méthode du contrôleur à exécuter en franchissant l'état en cours.

9. Dans la fenêtre de définition de la nouvelle activité, saisissez le nom de l'activité et son type, et spécifiez le moment de son exécution et l'opération à exécuter.

10. Cliquez sur New Opération pour pointer vers l'opération AjoutClient de la classe AjoutClientController.

La figure 12.13 illustre le processus d'ajout d'une nouvelle activité à un état du diagramme.

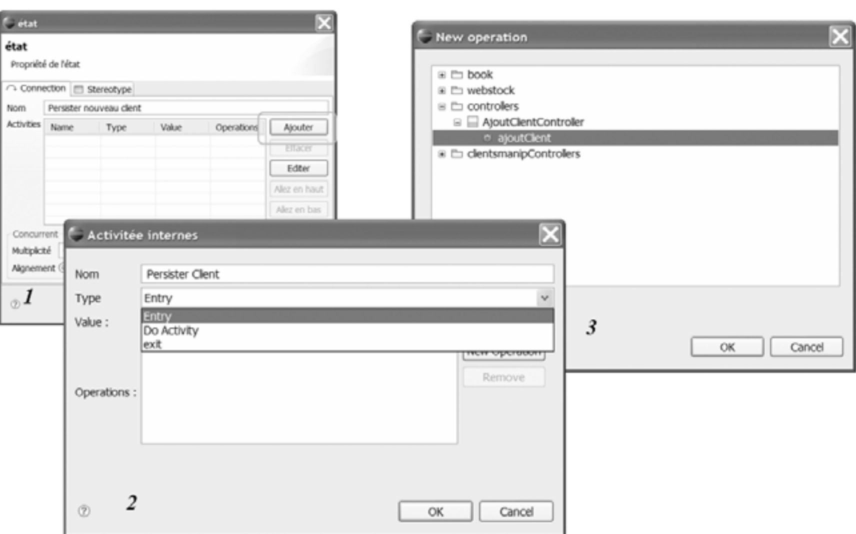


Figure 12.13

Exécution d'une opération à l'entrée d'une transition

Vous allez maintenant vérifier l'effet que vous venez d'ajouter à la transition issue de l'état Formulaire ajout client.

11. Choisissez Properties dans le menu pop-up de l'effet.

    Observez les paramètres de l'effet (au nombre de quatre) : nom, prénom, adresse et téléphone (voir figure 12.14). Ces paramètres sont ceux que prend la méthode

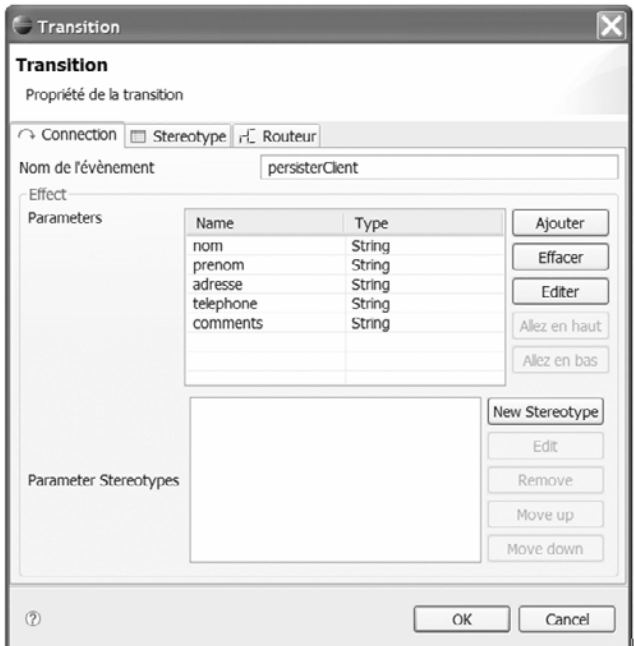
AjoutClient qui va s'exécuter dans l'état cible de cette transition et qui devront être saisis dans le formulaire de l'état source.

Par défaut, le générateur génère pour l'état Formulaire ajout client un formulaire dont les champs sont des textes. Il est possible de changer le type des champs en appliquant des valeurs balisées sur ces derniers.

12. Pour ce faire, sélectionnez le paramètre en question et, dans la partie Parameter Stereotypes, choisissez une valeur balisée du stéréotype Presentation::Presentation (voir figure 12.14).

Figure 12.14

Paramètres de l'effet AjoutParamsEffect



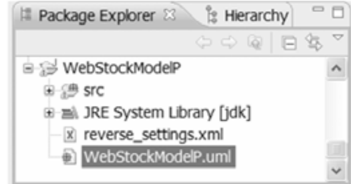
Export XMI du modèle

La phase de modélisation est achevée. En vertu du schéma fonctionnel d'EclipseUML et du cycle MDA, la seconde étape consiste à exporter le modèle en un XMI (XML Metadata Interchange) standard. EclipseUML le fera pour vous.

Comme vous avez pu le constater lors de la modélisation, chaque diagramme de classes est exporté sous la forme d'un fichier .uml portant son nom. On trouve également à la racine du projet un fichier .uml portant le nom du projet et sérialisant tous ses diagrammes (voir figure 12.15).

Figure 12.15

Sérialisation XMI avec EclipseUML



Cette phase de sérialisation est importante. C'est dans cette forme standard que le modèle va subir les transformations du générateur de code pour vous fournir le corps de votre application.

> **XMI (XML Metadata Interchange)**
>
> EclipseUML utilise EMF (Eclipse Modeling Farmework) pour sauvegarder ses modèles en XMI. XMI est un des standards sur lesquels repose MDA. C'est une forme standard définie par l'OMG, qui suppose une structuration des modèles sous forme XML permettant l'échange et le partage des modèles et des méta-données entre les différents outils de modélisation.

Nouveau projet JEE

Pour aboutir à la phase de génération de code, vous devez disposer d'un projet JEE destiné à accueillir le code de votre application.

Pour ce faire, vous allez utiliser l'assistant EclipseUML pour générer un projet JEE signé AndroMDA. Ce projet sera décomposé en *n*-tiers, où chaque tiers sera représenté par un projet Eclipse autonome, compatible avec le projet Web Tools.

Pour créer le projet, procédez de la façon suivante :

1. Cliquez sur File, New Project, AndroMDA et JEE Project.
2. Dans la première page de l'assistant, spécifiez le nom du projet, sa description ainsi que la paquetage de base où seront générées les ressources communes et non modélisées par l'utilisateur. Mettez à titre d'exemple book.webstock comme package de base.
3. Choisissez la technologie de la couche métier ainsi que le packaging à utiliser : EJB3 et la case EAR.
4. Spécifiez la base de données à utiliser ainsi que les paramètres de connexion : Hypersonic, la base native de JBoss.
5. Spécifiez la technologie Web désirée : JSF puis Facets.
6. Comme vous utilisez EJB3 et JSF, activez le support de la plate-forme JBoss Seam.
7. Cliquez sur Finish pour lancer la génération du projet JEE.

La figure 12.16 illustre l'assistant EclipseUML de création d'un nouveau projet JEE et la figure 12.17 le projet JEE tel qu'il est généré par EclipseUML.

À ce stade, vous disposez de cinq projets Eclipse ayant chacun une nature spécifique :

- EJB3_JSF_WebStock-app : le projet EAR qui va contenir les différents modules de l'application JEE.
- EJB3_JSF_WebStock-common : le projet Java qui va contenir les différentes classes d'exception de l'application. Comme son nom l'indique, ce projet est commun à toutes les autres couches de l'application.
- EJB3_JSF_WebStock-core : ce projet va contenir les composants EJB de l'application.
- EJB3_JSF_WebStock-mda : ce projet sans nature va contenir les fichiers de configuration ainsi que le modèle UML de l'application.
- EJB3_JSF _WebStock-web : ce projet Web correspond à la couche Web de l'application ; il va regrouper les pages Web et les composants JSF de l'application.

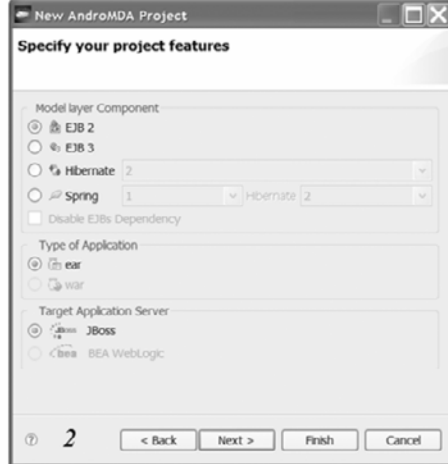
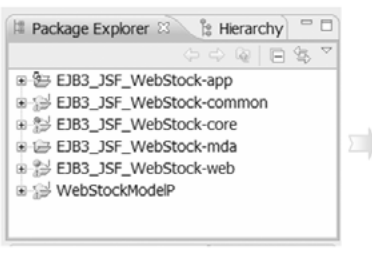


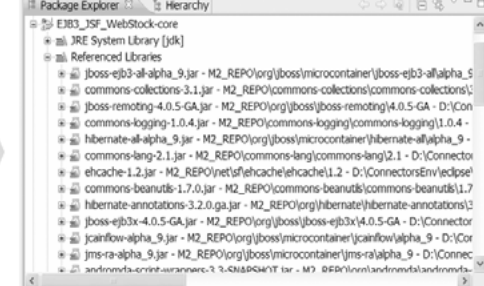
Figure 12.16

Assistant EclipseUML de création d'un nouveau projet JEE

Figure 12.17

Projet JEE généré par EclipseUML





> **Projet JEE**
>
> Chacune des couches du projet JEE est compatible Web Tools et n'a donc pas à être configurée afin de lui ajouter les bibliothèques et descripteurs indispensables. Le générateur EclipseUML le fait pour vous.

Génération de code

Cette étape correspond à la génération du code pour un déploiement sur le serveur JBoss.

1. Faites un clic droit sur le fichier .uml du projet de modélisation, puis choisissez JEE UML Code Generation et Build all components (voir figure 12.18).
2. Dans l'assistant en cours, choisissez le projet que vous venez de créer (c'est le projet qui sera choisi pour contenir le code généré), puis cliquez sur Finish.

Vous donnez ainsi la main à AndroMDA, qui va lancer son processus de génération. Ce dernier peut prendre plusieurs minutes. Si aucune erreur de validation du modèle n'est notifiée, la génération s'effectue avec succès.

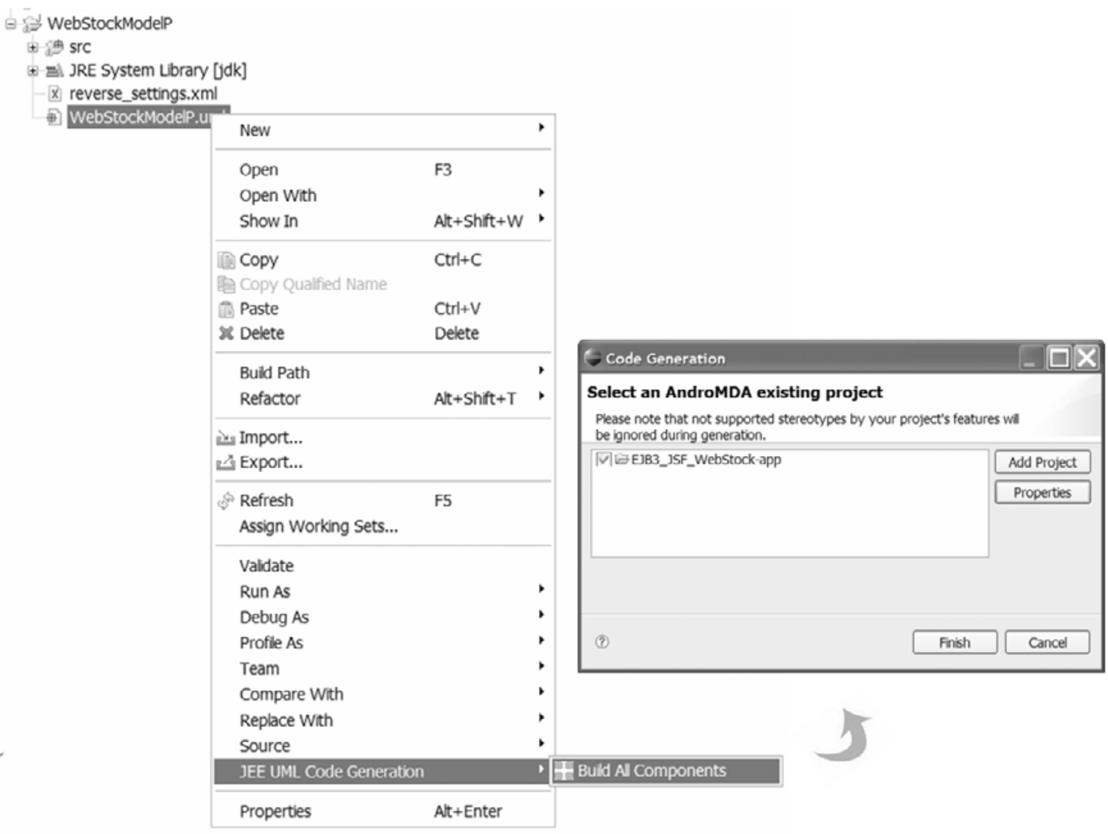


Figure 12.18

Assistant EclipseUML pour la génération de code EJB3

Code généré

Nous n'entrons pas ici dans le détail du code généré, car les concepts fondamentaux d'EJB3 ont été abordées au chapitre précédent.

Par contre, nous donnons quelques extraits de code pour illustrer comment ces notions ont été appliquées *via* les spécifications UML.

Beans entité

Comme vous venez de modéliser vos entités métier, vous allez passer à la concrétisation de votre application et à la génération de code.

Pour une meilleure visibilité, la figure 12.19 illustre les classes générées pour l'entité `Client`. Le diagramme a été obtenu par Reverse UML du code généré par EclipseUML.

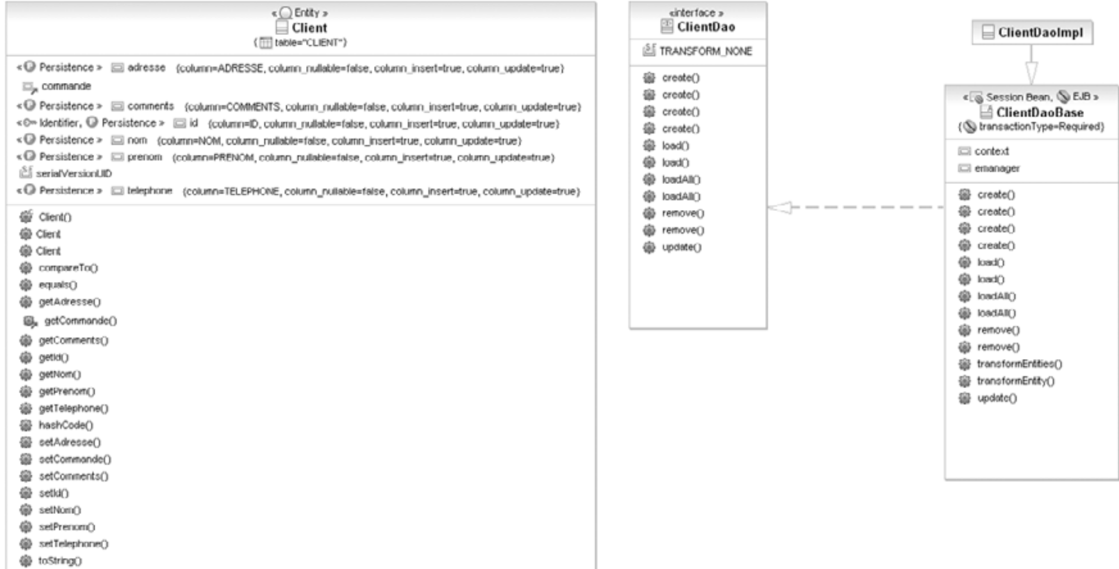


Figure 12.19

Classes générés pour l'entité EJB3

> **Reverse UML**
>
> EclipseUML supporte plusieurs nouvelles fonctionnalités, dont le Reverse Engineering à partir du code source pour obtenir le modèle. Le reverse est accessible *via* le menu pop-up des projets Java UML, puis Create et Update UML Model. Cette fonctionnalité vous permet de garder trace de votre modèle, non seulement à travers XMI mais aussi à travers son code métier. Vous pouvez ainsi « reverser » votre code pour obtenir un modèle que vous pourrez facilement migrer vers d'autres plates-formes cibles.

Le reverse cible plusieurs plates-formes, dont EJB3/2 et Hibernate 3/2. Pour plus d'informations, consultez la documentation EclipseUML *(http://www.ejb3.org).*

La figure 12.20 illustre l'assistant EclipseUML pour le reverse JEE.

Le reverse est assez fidèle à votre modèle de base, tout en étant plus détaillé. Le générateur génère plusieurs notions pour vous sans que vous ayez à les modéliser ou à ajouter des valeurs balisées supplémentaires.

Le reverse intercepte ces détails et les reporte au niveau du diagramme. Citons à titre d'exemple la génération des ID (identificateurs) pour chaque entité, sachant que, pour modéliser une telle propriété pour un bean entité, vous devez ajouter une nouvelle propriété portant la valeur balisée Persistence::Identifier. Rappelez-vous cependant que

vous n'avez pas effectué de telles manipulations sur vos entités et que le générateur a automatiquement généré un identificateur ainsi que d'autres notions, comme les Named-Query.

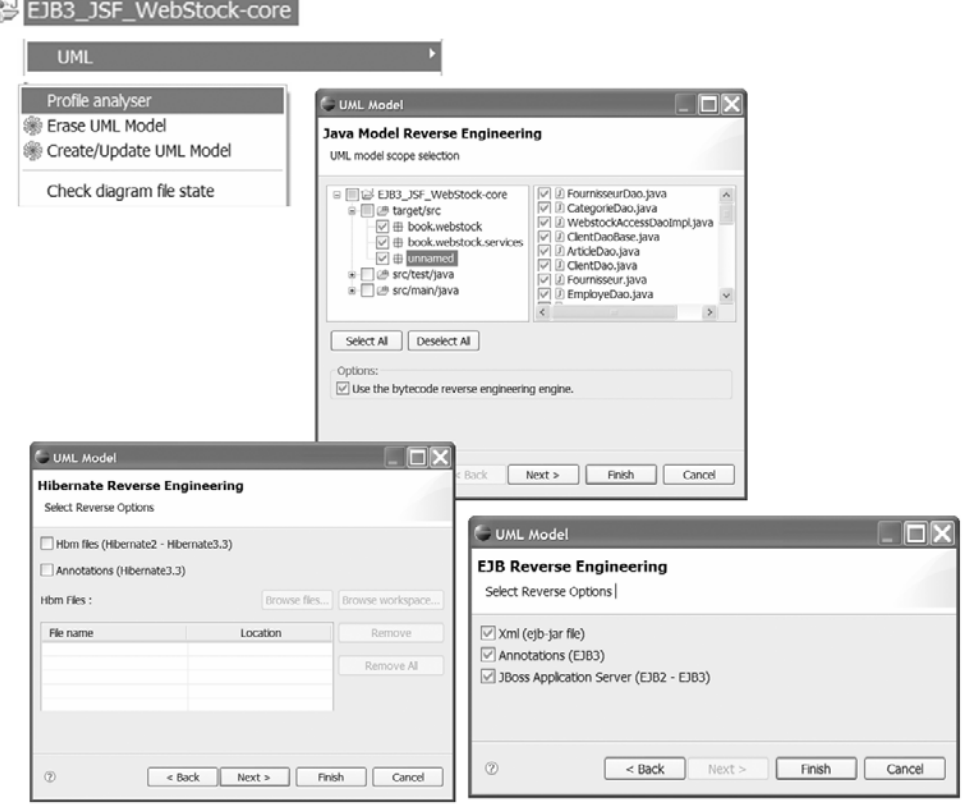


Figure 12.20

Reverse JEE

EclipseUML a généré quatre classes pour chaque entité modélisée :

- La classe du bean entité portant son nom.
- Un bean session portant le nom du bean suivi du préfixe DAOBase.
- L'interface locale du bean session avec le préfixe Dao.
- Une dernière classe étendant la classe du bean session, que l'utilisateur peut utiliser pour ajouter son code propre afin de personnaliser le comportement de la session. Cette classe porte le préfixe DaoImpl.

La classe la plus intéressante en termes de code généré est la classe `Commande`. Celle-ci possède plusieurs relations avec d'autres beans, dont des relations un à plusieurs, un à un et plusieurs à plusieurs.

Voici le code de l'entité `Article` :

```
@javax.persistence.Entity
@javax.persistence.Table(name = "ARTICLE")
```

```
@javax.persistence.NamedQuery(name = "Article.findAll", query = "select article from
➥Article AS article")
public class Article
    implements java.io.Serializable, Comparable<Article>
{

    private static final long serialVersionUID = 1016923637013396942L;

    // ----------- Attribute Definitions ------------

    private float poids;
    private java.lang.String nomArticle;
    private java.lang.String description;
    private java.lang.Long id;


    // --------- Relationship Definitions -----------

    private java.util.Set<unnamed.Fournisseur> fournisseur = new java.util.TreeSet
    ➥<unnamed.Fournisseur>();
    private java.util.Set<unnamed.Inventaire> inventaire = new java.util.TreeSet
    ➥<unnamed.Inventaire>();
    private java.util.Set<unnamed.Categorie> categorie = new java.util.TreeSet
    ➥<unnamed.Categorie>();
    private java.util.Set<unnamed.Commande> commande = new java.util.TreeSet
    ➥<unnamed.Commande>();

    /**
     * Default empty constructor
     */
    public Article() {}

    /**
     * Implementation for the constructor with all POJO attributes except auto
     ➥incremented identifiers.
     * This method sets all POJO fields defined in this class to the values provided by
     * the parameters.
     *
     * @param poids Value for the poids property
     * @param nomArticle Value for the nomArticle property
     * @param description Value for the description property
     */
     public Article(float poids, java.lang.String nomArticle, java.lang.String description)
    {
        setPoids(poids);
        setNomArticle(nomArticle);
        setDescription(description);
    }

    /**
     * Constructor with all POJO attribute values and CMR relations.
     *
     * @param poids Value for the poids property
     * @param nomArticle Value for the nomArticle property
     * @param description Value for the description property
     * @param fournisseur Value for the fournisseur relation
     * @param inventaire Value for the inventaire relation
```

```
     * @param categorie Value for the categorie relation
     * @param commande Value for the commande relation
     */
    public Article(float poids, java.lang.String nomArticle, java.lang.String
    ➥description, java.util.Set<unnamed.Fournisseur> fournisseur,
    ➥java.util.Set<unnamed.Inventaire> inventaire, java.util.Set<unnamed.Categorie>
    ➥categorie, java.util.Set<unnamed.Commande> commande)
    {
        setPoids(poids);
        setNomArticle(nomArticle);
        setDescription(description);
        setFournisseur(fournisseur);
        setInventaire(inventaire);
        setCategorie(categorie);
        setCommande(commande);
    }

@javax.persistence.Column(name = "POIDS", nullable = false, insertable = true, updatable
➥= true)
    public float getPoids()
    {
        return poids;
    }

    /**
     * Set the poids property.
     * @param value the new value
     */
    public void setPoids(float value)
    {
        this.poids = value;
    }
…

// l'implémentation des autres accesseurs

    /**
     * Get the id property.
     *
     * @return java.lang.Long The value of id
     */
    @javax.persistence.Id
    @javax.persistence.GeneratedValue(strategy = javax.persistence
    ➥.GenerationType.AUTO)
    @javax.persistence.Column(name = "ID", nullable = false, insertable = true, updatable
    ➥= true)
    public java.lang.Long getId()
    {
        return id;
    }

    /**
     * Set the id property.
     * @param value the new value
     */
    public void setId(java.lang.Long value)
    {
```

```
        this.id = value;
    }

// ------------- Relations ------------------

    /**
     * Get the fournisseur Collection
     *
     * @return java.util.Set<unnamed.Fournisseur>
     */
    @javax.persistence.ManyToMany(mappedBy = "articles")
    public java.util.Set<unnamed.Fournisseur> getFournisseur()
    {
        return this.fournisseur;
    }

    /**
     * Set the fournisseur
     *
     * @param fournisseur
     */
    public void setFournisseur (java.util.Set<unnamed.Fournisseur> fournisseur)
    {
        this.fournisseur = fournisseur;
    }

    /**
     * Get the inventaire Collection
     *
     * @return java.util.Set<unnamed.Inventaire>
     */
    @javax.persistence.OneToMany(mappedBy = "article", fetch = javax.persistence
    ➥.FetchType.EAGER)
    public java.util.Set<unnamed.Inventaire> getInventaire()
    {
        return this.inventaire;
    }

    /**
     * Set the inventaire
     *
     * @param inventaire
     */
    public void setInventaire (java.util.Set<unnamed.Inventaire> inventaire)
    {
        this.inventaire = inventaire;
    }

    /**
     * Get the categorie Collection
     *
     * @return java.util.Set<unnamed.Categorie>
     */
    @javax.persistence.ManyToMany()
    @javax.persistence.JoinTable
    (
        name = "ARTICLE2CATEGORIE",
```

```
        joinColumns = {@javax.persistence.JoinColumn(name = "ARTICLE_IDC",
        ➥referencedColumnName = "ID")},
        inverseJoinColumns = {@javax.persistence.JoinColumn(name = "CATEGORIE_IDC",
        ➥referencedColumnName = "ID")}
    )
    public java.util.Set<unnamed.Categorie> getCategorie()
    {
        return this.categorie;
    }

    /**
     * Get the commande Collection
     *
     * @return java.util.Set<unnamed.Commande>
     */
    @javax.persistence.OneToMany()
    public java.util.Set<unnamed.Commande> getCommande()
    {
        return this.commande;
    }

..................
//reste de l'implémentation
} //fin classe
```

La deuxième classe intéressante est le bean session `ArticleDaoBase` (généré pour le bean entité `Article`), qui implémente les méthodes CRUD donnant accès à ce bean entité.

Voici le code de cette classe :

```
/**
 * <p>
 * Base EJB3 DAO Class: is able to create, update, remove, load, and find
 * objects of type <code>unnamed.Client</code>.
 * </p>
 *
 * @see unnamed.ClientDao
 */
@javax.ejb.TransactionAttribute(javax.ejb.TransactionAttributeType.REQUIRED)
@javax.ejb.Local({unnamed.ClientDao.class})
public abstract class ClientDaoBase
    implements unnamed.ClientDao
{
    // ------ Session Context Injection ------

    @javax.annotation.Resource
    protected javax.ejb.SessionContext context;

    // ------ Persistence Context Injection --------

    /**
     * Inject persistence context testNG
     */
    @javax.persistence.PersistenceContext(unitName = "testNG")
    protected javax.persistence.EntityManager emanager;

    /**
     * @see unnamed.ClientDao#load(int,)
```

```
 */
public Object load(final int transform, final java.lang.Long id)
    throws unnamed.ClientDaoException
{
    if (id == null)
    {
        throw new IllegalArgumentException(
            "Client.load - 'id' can not be null");
    }
    try
    {
        final Object entity = (unnamed.Client)emanager.
        ➥find(unnamed.Client.class, id);
        return transformEntity(transform, (unnamed.Client)entity);
    }
    catch (Exception ex)
    {
        throw new unnamed.ClientDaoException(ex);
    }
}

/**
 * @see unnamed.ClientDao#load()
 */
public unnamed.Client load( final java.lang.Long id)
    throws unnamed.ClientDaoException
{
    return (unnamed.Client)this.load(TRANSFORM_NONE, id);
}

/**
 * @see unnamed.ClientDao#loadAll()
 */
@SuppressWarnings({"unchecked"})
public java.util.Collection<unnamed.Client> loadAll()
    throws unnamed.ClientDaoException
{
    return (java.util.Collection<unnamed.Client>)this.loadAll(TRANSFORM_NONE);
}

/**
 * @see unnamed.ClientDao#loadAll(int)
 */
public java.util.Collection loadAll(final int transform)
    throws unnamed.ClientDaoException
{
    // implémentation
}

/**
 * @see unnamed.ClientDao#create(unnamed.Client)
 */
public unnamed.Client create(unnamed.Client client)
    throws unnamed.ClientDaoException
{
    return (unnamed.Client)this.create(TRANSFORM_NONE, client);
}
```

```
/**
 * @see unnamed.ClientDao#create(int transform, unnamed.Client)
 */
public Object create(final int transform, final unnamed.Client client)
    throws unnamed.ClientDaoException
{
    if (client == null)
    {
        throw new IllegalArgumentException(
            "Client.create - 'client' can not be null");
    }

    try
    {
        emanager.persist(client);
        emanager.flush();
        return this.transformEntity(transform, client);
    }
    catch (Exception ex)
    {
        throw new unnamed.ClientDaoException(ex);
    }
}

/**
 * @see unnamed.ClientDao#create(java.util.Collection<unnamed.Client>)
 */
@SuppressWarnings({"unchecked"})
public java.util.Collection<unnamed.Client> create(final java.util.Collection
➥<unnamed.Client> entities)
    throws unnamed.ClientDaoException
{
    return create(TRANSFORM_NONE, entities);
}

/**
 * @see unnamed.ClientDao#create(int, java.util.Collection<unnamed.Client>)
 */
@SuppressWarnings({"unchecked"})
public java.util.Collection create(final int transform, final java.util
➥.Collection<unnamed.Client> entities)
    throws unnamed.ClientDaoException
{
    if (entities == null)
    {
        throw new IllegalArgumentException(
            "Client.create - 'entities' can not be null");
    }
    java.util.Collection results = new java.util.ArrayList();
    try
    {
    for (final java.util.Iterator entityIterator = entities.iterator();
    ➥entityIterator.hasNext();)
    {
    results.add(create(transform, (unnamed.Client)
    ➥entityIterator.next()));
    }
}
```

```
catch (Exception ex)
{
    throw new unnamed.ClientDaoException(ex);
}
    return results;
}

/**
 * @see unnamed.ClientDao#create(java.lang.String, java.lang.String,
 ➥java.lang.String, java.lang.String, java.lang.String)
 */
public unnamed.Client create(
    java.lang.String comments,
    java.lang.String telephone,
    java.lang.String adresse,
    java.lang.String prenom,
    java.lang.String nom)
    throws unnamed.ClientDaoException
{
    return (unnamed.Client)this.create(TRANSFORM_NONE, comments, telephone,
    ➥adresse, prenom, nom);
}

/**
 * @see unnamed.ClientDao#create(int, java.lang.String, java.lang.String,
 ➥java.lang.String, java.lang.String, java.lang.String)
 */
public Object create(
    final int transform,
    java.lang.String comments,
    java.lang.String telephone,
    java.lang.String adresse,
    java.lang.String prenom,
    java.lang.String nom)
    throws unnamed.ClientDaoException
{
    unnamed.Client entity = new unnamed.Client();
    entity.setComments(comments);
    entity.setTelephone(telephone);
    entity.setAdresse(adresse);
    entity.setPrenom(prenom);
    entity.setNom(nom);
    return this.create(transform, entity);
}

/**
 * @see unnamed.ClientDao#update(unnamed.Client)
 */
public void update(unnamed.Client client)
    throws unnamed.ClientDaoException
{
    if (client == null)
    {
        throw new IllegalArgumentException(
            "Client.update - 'client' can not be null");
    }
    try
    {
```

```
            emanager.merge(client);
            emanager.flush();
        }
        catch (Exception ex)
        {
            throw new unnamed.ClientDaoException(ex);
        }
    }

    /**
     * @see unnamed.ClientDao#update(java.util.Collection<unnamed.Client>)
     */
    public void update(final java.util.Collection<unnamed.Client> entities)
        throws unnamed.ClientDaoException
    {
        // implémentation
    }

    /**
     * @see unnamed.ClientDao#remove(unnamed.Client)
     */
    public void remove(unnamed.Client client)
        throws unnamed.ClientDaoException
    {
        if (client == null)
        {
            throw new IllegalArgumentException(
                "Client.remove - 'client' can not be null");
        }
        try
        {
            emanager.remove(client);
            emanager.flush();
        }
        catch (Exception ex)
        {
            throw new unnamed.ClientDaoException(ex);
        }
    }

    /**
     * @see unnamed.ClientDao#remove(java.lang.Long)
     */
    public void remove(java.lang.Long id)
        throws unnamed.ClientDaoException
    {
        if (id == null)
        {
            throw new IllegalArgumentException(
                "Client.remove - 'id' can not be null");
        }
        try
        {
        final unnamed.Client entity = this.load(id);
        if (entity != null)
        {
            this.remove(entity);
        }
    }
```

```
catch (Exception ex)
{
    throw new unnamed.ClientDaoException(ex);
}
}

/**
 * @see unnamed.ClientDao#remove(java.util.Collection<unnamed.Client>)
 */
public void remove(java.util.Collection<unnamed.Client> entities)
    throws unnamed.ClientDaoException
{
    // implémentation
}

protected Object transformEntity(final int transform, final unnamed.Client entity)
{
          // Implémentation de la méthode

}

protected void transformEntities(final int transform, final java.util
➥.Collection entities)
{
    // Implémentation de la méthode
}}
```

Il est inutile de reporter le code généré pour les classes `ArticleDaoImpl` et `ArticleDao`, qui sont respectivement l'interface pour personnaliser le bean session `ArticleDaoBase` et l'interface locale de ce dernier. Le lecteur souhaitant s'y plonger peut se reporter au code complet associé à ce chapitre, disponible sur la page Web dédiée à l'ouvrage.

Beans session

La figure 12.21 illustre les classes générés pour le bean session `ClientsService`.

Figure 12.21

Classes générées pour le bean session ClientsService

Les classes générées sont les suivantes :

- `ClientsServiceBase` : classe du bean session.
- `ClientsServiceRemote` : interface distante du bean session.
- `ClientsServiceBean` : classe implémentant la classe du bean session. Remarquez dans l'explorateur de projets que cette classe se trouve dans le dossier source src/main/java. Cela signifie qu'elle peut être éditée par l'utilisateur et qu'elle ne sera donc pas écrasée lors de la prochaine génération de code.

La figure 12.22 illustre les classes générées pour la classe `ClientsService` modélisée.

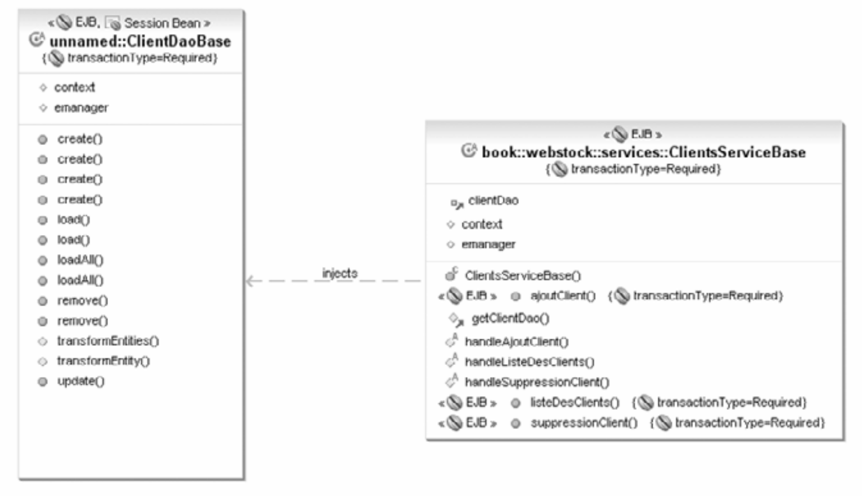


Figure 12.22

Injection de dépendances entre les beans session ClientsServiceBean et ClientDaoBase

Remarquez la notion d'injection de dépendances entre les beans `ClientDaoBase` et `ClientsServiceBean` dans le listing suivant :

```
@javax.ejb.TransactionAttribute(javax.ejb.TransactionAttributeType.REQUIRED)
@javax.ejb.Remote({book.webstock.services.ClientsServiceRemote.class})

    public abstract class ClientsServiceBase
    implements book.webstock.services.ClientsServiceRemote
    {
    // ------ Session Context Injection ------

    @javax.annotation.Resource
    protected javax.ejb.SessionContext context;

    // ------ Persistence Context Definitions --------

    /**
     * Inject persistence context testNG
     */
    @javax.persistence.PersistenceContext(unitName = "Ejb3_JSF_WebStock")
    protected javax.persistence.EntityManager emanager;
```

```
// ------ DAO Injection Definitions --------
/**
     * Inject DAO ClientDao
     */
    @javax.ejb.EJB
    private unnamed.ClientDao clientDao;

    // --------------- Constructors ---------------

    public ClientsServiceBase()
    {
        super();
    }

// ------ DAO Getters --------

    /**
     * Get the injected DAO ClientDao
     */
    protected unnamed.ClientDao getClientDao()
    {
        return this.clientDao;
    }

// -------- Business Methods  --------------

    @javax.ejb.TransactionAttribute(javax.ejb.TransactionAttributeType.REQUIRED)
    public void ajoutClient(java.lang.String nom, java.lang.String prenom,
    ➥java.lang.String adresse, java.lang.String telephone, java.lang.String comments)
    {
        // implémentation de la méthode
    }

    /**
     * Performs the core logic for {@link #ajoutClient(java.lang.String,
     ➥java.lang.String, java.lang.String, java.lang.String, java.lang.String)}
     */
    protected abstract void handleAjoutClient(java.lang.String nom, java.lang.String
    ➥prenom, java.lang.String adresse, java.lang.String telephone, java.lang.String
    ➥comments)
        throws java.lang.Exception;

@javax.ejb.TransactionAttribute(javax.ejb.TransactionAttributeType.REQUIRED)
    public void listeDesClients()
    {
        implémentation de la méthode
    }

    /**
     * Performs the core logic for {@link #listeDesClients()}
     */
    protected abstract void handleListeDesClients()
        throws java.lang.Exception;

 @javax.ejb.TransactionAttribute(javax.ejb.TransactionAttributeType.REQUIRED)
    public void suppressionClient(long id)
    {
```

```
            implémentation de la méthode
    }

    /**
     * Performs the core logic for {@link #suppressionClient(long)}
     */
    protected abstract void handleSuppressionClient(long id)
        throws java.lang.Exception;
    }
```

Descripteurs générés

Trois descripteurs sont générés pour la couche métier ou le projet EJB : ejb-jar.xml (non utile pour un développement EJB3), jboss.xml et persistence.xml, comme l'illustre la figure 12.23.

Figure 12.23

Descripteurs générés dans le cadre de l'application webstock

La figure 12.24 illustre le descripteur jboss.xml généré pour l'application webstock déclarant les beans session de la couche métier.

Figure 12.24

Descripteur jboss.xml

La figure 12.25 donne un aperçu du descripteur ejb-jar.xml de l'application webstock déclarant les différents beans (session et entité) de l'application.

Figure 12.25

Descripteur ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                        http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
    version="3.0">

    <description><![CDATA[No Description.]]></description>
    <display-name>Generated by AndroMDA EJB3 Cartridge</display-name>

    <enterprise-beans>

        <session>
            <description>
                <![CDATA[

                ]]>
            </description>
            <ejb-name>ClientsServiceBean</ejb-name>
            <remote>book.webstock.services.ClientsServiceRemote</remote>
            <ejb-class>book.webstock.services.ClientsServiceBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
```

La figure 12.26 illustre une partie du fichier de configuration persistence.xml généré permettant de configurer l'unité persistante utilisée pour l'application webstock.

Figure 12.26

Descripteur persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
     Attention: Generated code! Do not modify by hand!
     Generated by: persistence.xml.vsl in andromda-ejb3-cartridge.
  -->
<persistence>

<!--
<persistence
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">
-->
    <persistence-unit name="EJB3_JSF_WebStock">
        <jta-data-source>java:/DefaultDS</jta-data-source>
        <properties>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
        </properties>
    </persistence-unit>
</persistence>
```

Couche Web

Un ensemble de facelets sont générées pour chacune des pages Web de l'application.

Prenons comme exemple la page d'ajout d'un nouveau client dans la base webstock. Les deux facelets suivantes sont générées :

- formulaire-ajout-client.xhtml
- formulaire-ajout-client-ajout-client-effect.xhtml

Voici le code du facelet formulaire-ajout-client.xhtml :

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:c="http://java.sun.com/jstl/core">

<ui:composition template="/layout/layout.xhtml">
    <ui:define name="title">
        <c:set var="title" value="#{messages['formulaire.ajout.client.title']}"
        ➥scope="request"/>
    </ui:define>
    <ui:define name="content">
        <ui:include src="/book/webstock/clients/ajout/formulaire-ajout-client-ajout-
        ➥client-effect.xhtml"/>
    </ui:define>
</ui:composition>

</html>
```

Le code du facelet formulaire-ajout-client-ajout-client-effect.xhtml est généré comme suit :

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:c="http://java.sun.com/jstl/core"
      xmlns:t="http://myfaces.apache.org/tomahawk"
      xmlns:a="http://www.andromda.org/cartridges/jsf"
      xmlns:af="http://xmlns.oracle.com/adf/faces">
<ui:composition>
<h:form id="ajoutDUnNouveauClientFormulaireAjoutClientAjoutClientEffectForm"
➥enctype="multipart/form-data">
    <af:panelForm>
        <af:inputText id="nom" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.nom}"
        ➥label="#{messages['nom']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="prenom" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.prenom}"
        ➥label="#{messages['prenom']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="adresse" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.adresse}"
        ➥label="#{messages['adresse']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="telephone" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.telephone}"
        ➥label="#{messages['telephone']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="comments" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.comments}"
  label="#{messages['comments']}:" required="false" readOnly="false" rows="3"
  ➥columns="40">
        </af:inputText>
        <f:facet name="footer">
            <af:panelButtonBar>
```

```
                <af:commandButton text="#{messages['ajout.client.effect']}"
                ➥action="#{ajoutClientController.formulaireAjoutClient
                ➥AjoutClientEffect}"/>
            </af:panelButtonBar>
        </f:facet>
    </af:panelForm>
    <a:validator client="false"/>
</h:form>
</ui:composition>
</html>
```

L'action du bouton de validation de ce formulaire passe la requête à la classe `ajout-ClientController`. Cette classe sera personnalisée *via* son implémentation `ajoutClient-ControllerImpl`.

Personnalisation du code

Le code produit est suffisamment fonctionnel pour être déployé sur le serveur JBoss.

Prenons l'exemple du scénario d'insertion d'un nouveau client dans la base. Pour compléter ce processus, vous devez éditer un certain nombre de classes, dont Ajout-ClientControllerImpl.java et ClientsServiceBean.java. Les éditions de code étant mineures, nous ne donnons ici que les classes à implémenter.

Voici l'implémentation de la méthode `ajoutClient` du contrôleur `AjoutClientControllerImpl` :

```
    /**
     * @see book.webstock.clients.ajout.AjoutClientController
     */
     public class AjoutClientControllerImpl
      extends AjoutClientController
     {

     private static final long serialVersionUID = 1L;

     /**
      * @see book.webstock.clients.ajout.AjoutClientController#ajoutClient
      ➥(java.lang.String nom, java.lang.String prenom, java.lang.String adresse,
      ➥java.lang.String telephone, java.lang.String comments)
      */
      public void ajoutClient(AjoutClientForm form)
      {
            book.webstock.services.ClientsServiceRemote clientsService;
          try {
              clientsService =  (book.webstock.services.ClientsServiceRemote)
              ➥ServiceLocator.instance().getService("testNG-1.0.0/ClientsServiceBean/
              ➥remote");


          clientsService.ajoutClient(form.getNom(), form.getPrenom(),
          ➥form.getAdresse(), form.getTelephone(), form.getComments());

             } catch (Exception e) {
                 e.printStackTrace();
             }
      }
}
```

Vous devez également implémenter la méthode `handleAjoutClient` de la classe `ClientsServiceBean` en accédant au bean `ClientDao` injecté à travers le bean session `ClientServiceBase`. Cela permet d'accéder à sa méthode `create(Client)`, qui crée votre nouveau client et l'insère dans la base.

Voici le code de la méthode `handleAjoutClient` :

```
protected void handleAjoutClient(java.lang.String nom, java.lang.String prenom,
➥java.lang.String adresse, java.lang.String telephone, java.lang.String comments)
        throws java.lang.Exception
    {
    getClientDao().create(comments, telephone, adresse, prenom,   nom);  }
```

Packaging de l'application webstock

Une fois que vous avez édité votre code comme indiqué précédemment, vous devez relancer la génération de code. Celle-ci met à jour vos sources s'il y a eu des changements du modèle ; sinon, c'est une phase de compilation et de packaging (voir figure 12.27).

Figure 12.27

Packaging de l'application webstock

Consultez le dossier target du projet EAR de l'application EJB3_JSF_WebStock-app. Celui-ci contient l'archive EAR ainsi que le fichier XML de configuration de la datasource que vous devrez copier dans le dossier de déploiement de votre serveur.

Déploiement sur le serveur JBoss

Vous allez utiliser JBoss comme serveur d'applications. Vous choisirez la version 4.2, qui implémente la spécification EJB3. Cette version est la plus stable actuellement.

1. Copiez l'EAR et le ficher XML de configuration de la datasource sous les dossiers $JBOSS_HOME/server/default/deploy.
2. Après avoir démarré le serveur, vérifiez que le déploiement s'est achevé avec succès. Pour ce faire, voyez si les tables relatives à vos entités EJB3 ont été créées automatiquement, comme spécifié dans le descripteur persistence.xml.
3. Accédez pour cela à la base Hypersonic *via* la console JMX *(http://127.0.0.1 :8080/JMX-console/)* en cliquant sur `database=localDB,service=Hypersonic` puis `startDatabaseManager()`.

La figure 12.28 illustre les tables de l'application créées avec succès lors du déploiement.

4. Vous pouvez également vérifier si vos beans session ont été déployés avec succès en consultant la liste des références JNDI du serveur *via* service=JNDIView puis List of MBean operations à partir de la console JMX du serveur (voir figure 12.29).

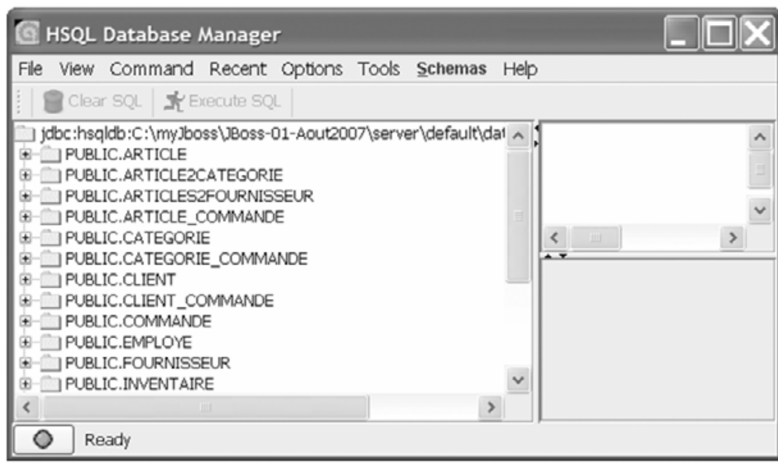


Figure 12.28

Création du schéma de la base de données de l'application webstock

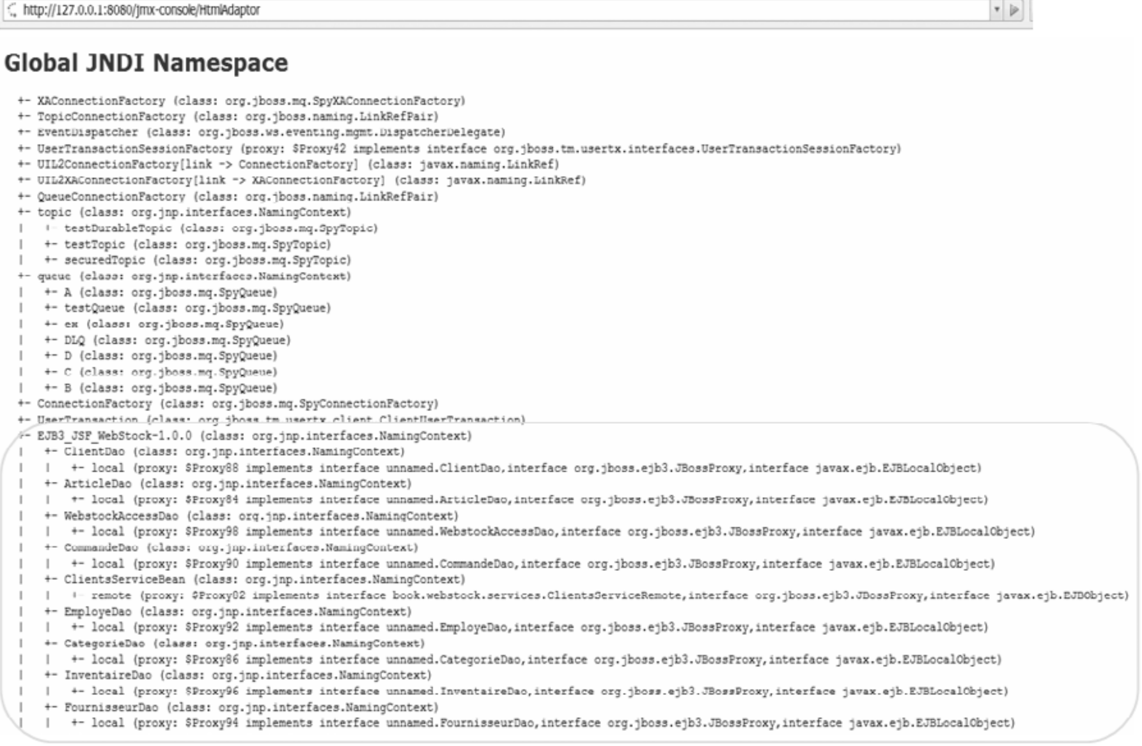


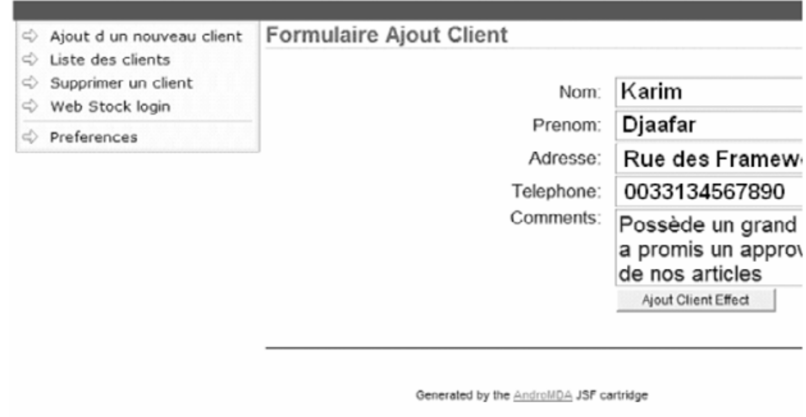
Figure 12.29

Références JNDI relatives à l'application EJB3_JSF_WebStock

5. Pour tester votre application, entrez l'adresse *http://127.0.0.1:8080/EJB3_JSF_WebStock*. La page d'ajout d'un nouveau client s'affiche comme page d'accueil de l'application (voir figure 12.30).

Figure 12.30

Exécution de l'application webstock



En résumé

Ce chapitre vous a permis de mettre en œuvre sur un exemple concret les concepts EJB3 abordés au cours des chapitres précédents. Le processus MDA a été mis en œuvre à l'aide de l'outil EclipseUML for JEE, qui vous a permis d'économiser vos efforts de codage pour vous concentrer sur la modélisation métier.

Le chapitre suivant traite des fonctionnalités avancées de la nouvelle plate-forme Seam de Jboss. Celles-ci permettent d'alléger le contexte de vos applications par une meilleure collaboration entre la couche Web et la couche métier.

13

Configuration EJB3 avancée avec Seam

Au chapitre précédent, vous avez bâti votre application sur EJB3 et JSF et mis en œuvre les innovations apportées par la nouvelle spécification.

Dans cet ultime chapitre, vous reprendrez l'application exemple webstock en partie générée à l'aide de l'atelier EclipseUML d'Omondo. Vous utiliserez pour ce faire le framework Seam, présenté au chapitre 8, mais cette fois appliqué au développement EJB3.

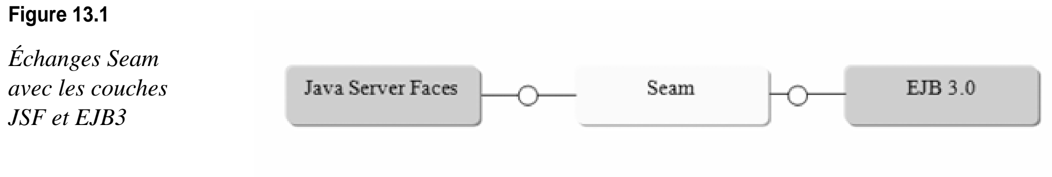
Vous découvrirez les aspects avancés de ce framework, en particulier pour la gestion des contextes, un des points forts de Seam.

Les contextes Seam

Comme expliqué au chapitre 8, JBoss Seam simplifie les échanges entre les différentes couches des applications JEE, en particulier entre la partie présentation, représentée par la technologie JSF, et la partie EJB3, comme illustré à la figure 13.1.

Figure 13.1

Échanges Seam avec les couches JSF et EJB3



La possibilité de gérer plusieurs conversations simultanées est une fonctionnalité remarquable de Seam permise par le support du contexte conversationnel, qui permet de simuler virtuellement des sessions multiples sans avoir à les créer physiquement.

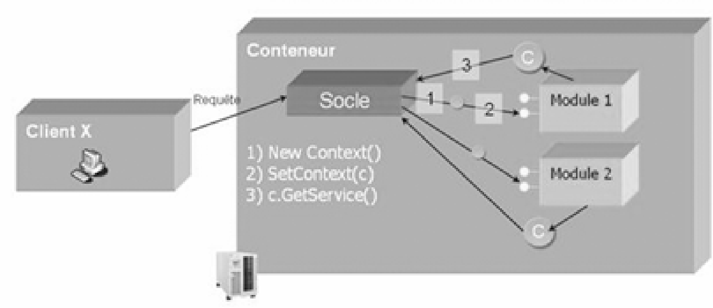
Pour Seam, tous les composants d'une application en cours d'exécution sont des objets avec état (stateful). Ces objets sont souvent des EJB, comme dans notre exemple.

Chaque instance d'un composant est associée à un contexte défini et possède un nom permettant de la référencer.

Par définition, un contexte est une vue sur les composants d'une application à base de conteneur en cours d'exécution, comme l'illustre la figure 13.2.

Figure 13.2

Représentation d'un contexte Seam

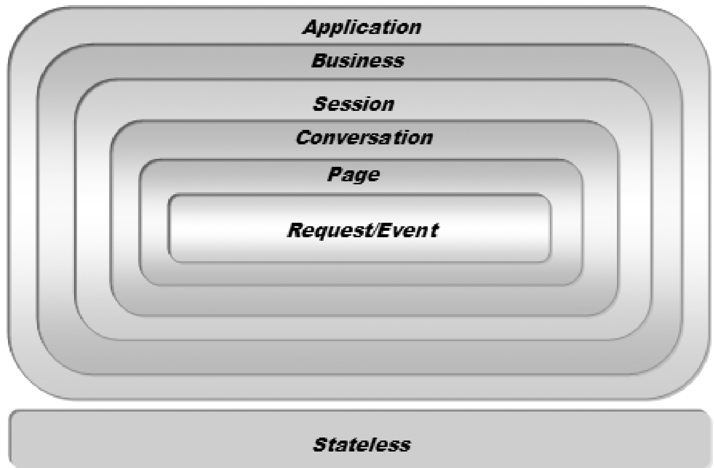


Une application Web classique comporte plusieurs contextes traditionnels, comme Session, Requête et Application. Seam ajoute quatre nouveaux contextes au modèle existant (voir figure 13.3) :

- Stateless
- Conversation
- Page
- Processus métier (Business)

Figure13.3

Modèle contextuel de Seam



Les contextes Seam sont créés et détruits par le conteneur JBoss. Ils ne sont plus contrôlés avec une API Java spécifique, mais gérés d'une manière explicite, déclarée à l'aide des annotation JDK5.

Chaque type de composant Seam possède un contexte par défaut, que le développeur peut modifier à l'aide de l'annotation @Scope.

Le tableau 13.1 récapitule les contextes par défaut de Seam.

Tableau 13.1 Contextes par défaut de Seam

objet	contexte seam par defaut
Beans stateless	STATELESS
Beans entité	CONVERSATION
Beans session stateful	CONVERSATION
JavaBeans (POJO)	EVENT

Rappelons brièvement les rôles des contextes Seam :

- **Stateless.** Les objets sans état vivent dans le contexte Stateless traditionnel d'une application Web. Tout JavaBean peut être désigné comme étant un objet Seam sans état. Ce contexte est utilisé lorsqu'il n'est pas nécessaire de persister l'état du contexte durant plusieurs appels de la classe.
- Voici un exemple de JavaBean helloworld stateless :

```
@Name ("statelessHelloWorld")
@Scope (ScopeType.STATELESS)
public class StatelessHelloWord {
  @RequestParameter ("nom")
  String nom;

  public String getText() {
      return "Hello World ! " + nom ;
   }
}
```

- **Event.** C'est le contexte le plus simple et le plus utilisé dans les applications Web. Les objets enregistrés dans le cadre de ce contexte sont détruits à la fin de la requête. Les JavaBeans sont stockés par défaut dans ce contexte événementiel.
- **Page.** Les composants de ce contexte sont attachés à une page spécifique. Vous avez accès à ces composants pour tout événement émis à partir de cette page.
- **Conversation.** Ce contexte est détaillé ultérieurement dans ce chapitre.
- **Session.** Gère l'état associé à la session utilisateur afin de garder trace de certaines informations relatives à cet utilisateur et des données susceptibles d'être partagées entre différentes conversations.
- **Business.** C'est le contexte associé aux processus métier de longue durée. L'état de ce contexte est géré et persisté par le moteur de workflow JBoss jBPM (JBoss Business Process Management). Rappelons qu'un processus métier est un ensemble d'actions réparties sur différents types d'utilisateurs. L'état de ce contexte est partagé entre ces différents utilisateurs selon une réglementation définie.
- **Application.** Utilisé pour garder les informations statiques relatives à une application donnée, comme les métamodèles ou les objets de référence. Par exemple, Seam persiste ses configurations et métamodèles dans le contexte de l'application.

Gestion des états de composants Seam

Les instances de composants Seam sont associées à une variable de contexte. Le nom du composant est attribué *via* l'annotation @Name et le contexte auquel il sera associé à l'aide de l'annotation @Scope.

Contexte conversationnel

La conversation est un concept bien connu du développement J2EE. JBoss Seam est toutefois le premier framework à l'implémenter complètement et à l'enrichir avec le scope de conversation.

Une conversation peut être vue comme un genre de pas-à-pas. C'est une action qui s'étend sur plusieurs petites actions et qui permet a l'utilisateur de se promener aisément d'avant en arrière, sans se soucier de la perte de ses données, mais sans pour autant les persister dans la base.

Toutes les informations utilisées pendant une conversation sont stockées dans le scope de conversation. La notion de conversation est introduite par Seam à travers le support de ses nouveaux contextes.

Pour Seam, une conversation est une requête utilisateur qui s'étend sur plusieurs pages avant d'être close. Le panier d'achats en ligne en est un exemple. Chaque cycle requête/réponse est aussi une conversation élémentaire entre la page qui expose le formulaire de la requête et la page de réponse. En d'autres termes, une conversation est une unité de travail élémentaire : ce qu'on fait à un moment donné sur une page donnée.

Par défaut, les objets Seam avec état (stateful) ont une conversation de type Scope. Le contexte conversationnel par défaut se résume ainsi : le composant est instancié quand la première page est émise puis est détruit quand la page de réponse est définitivement affichée.

Les composants utilisent des conversations temporaires, qui ont lieu tout au long de la requête avant d'être détruites à l'affichage de la page de réponse.

POJO

Un POJO Seam qui ne porte que l'annotation @Name est un objet avec état et est enregistré dans le contexte conversationnel du conteneur du serveur d'applications.

Mise en œuvre de l'étude de cas avec Seam

Dans cette section, vous allez « refactorer » le code de l'application webstock afin de bien cerner l'intégration de Seam dans l'architecture MVC et la valeur ajoutée de cette plate-forme. Vous pourrez ainsi mesurer à quel point elle allège le contexte de l'application suivant la bonne pratique dite DRY.

DRY (Don't Repeat Yourself)

Le concept DRY vise à éviter les redondances et les duplications de code d'une application afin d'en augmenter la cohérence ainsi que l'évolutivité et la maintenance.

Intégration de Seam dans l'architecture MVC

Le diagramme de séquences illustré à la figure 13.4 illustre un cycle JSF complet intégrant Seam.

Seam utilise différents intercepteurs pour contrôler ses objets à travers les différentes couches de l'application. Le but est de ne pas encombrer l'architecture de l'application en ajoutant des servlets spécifiques, comme le font les autres plates-formes, et de ne pas perturber le cycle JSF.

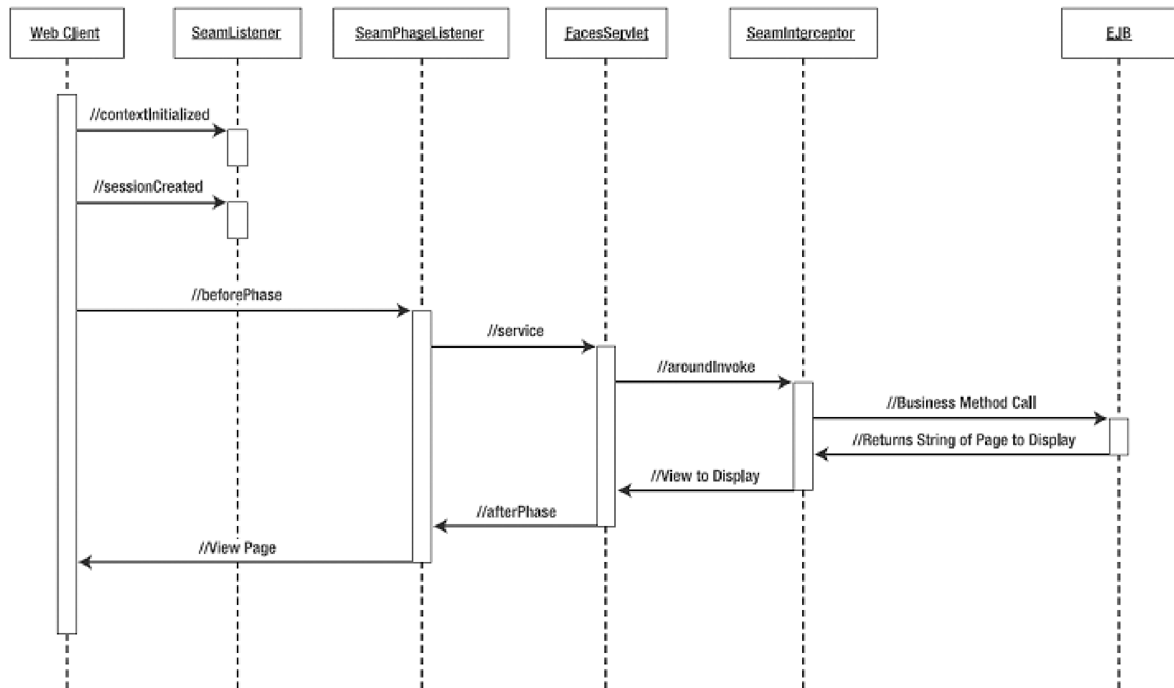


Figure 13.4

Diagramme de séquences du cycle JSF intégrant Seam

Le premier listener appelé est SeamListener (org.jboss.seam.servlet.SeamListener), que vous devez déclarer dans le fichier descripteur de l'application web.xml (voir listing suivant). Ce listener est appelé à chaque instanciation d'une nouvelle session.

```
<listener>
   <listener-class>
      org.jboss.seam.servlet.SeamListener
   </listener-class>
</listener>

<!—Faces Servlet Maping -- >
<servlet-mapping>
    <servlet-name>faces Servlet</servlte-name>
    <url-pattern>*.seam</url-pattern>
</servlet-mapping>
```

C’est à travers cet objet que vous invoquerez la servlet FacesServlet. Ce listener fait partie du cycle de vie de la servlet FacesServlet, comme décrit dans l’extrait suivant (notez le package org.andromda.presentation utilisé par AndroMDA pour la génération JSF avec EclipseUML) :

```
<lifecycle>
  <phase-listener> org.andromda.presentation.jsf.MessagePhaseListener
  </phase-listener> org.jboss.seam.jsf.SeamPhaseListener</phase-listener>
</lifecycle>
```

Pour intégrer Seam à la couche métier, vous devrez utiliser SeamInterceptor afin d’intercepter les appels aux EJB et composants métier :

```
< ?xml version="1.0" encoding="UTF-8" ?>
<ejb-jar xmlns=http://java.sun.com/xml/ns/javaee
   xmlns :xsi=http://www.w3.org/2001/XMLSchema-instance
   xsi :schemaLocation="http://java.sun.com/xml/ns/javaee
     http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
   version="3.0"
  <interceptors>
     <interceptor>
        org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
     </interceptor>
   <interceptors>
```

La figure 13.5 illustre ce principe d’intégration « d’intercepteurs » à travers les couches Web et métier des applications EJB3/JSF.

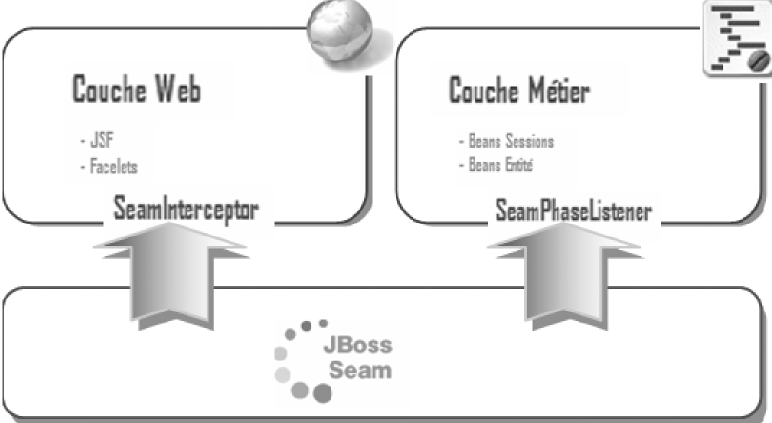


Figure 13.5

Intercepteurs Seam

Configuration de Seam

L’installation du framework Seam est décrite en annexe de l’ouvrage. Dans le cadre de ce chapitre, vous aurez besoin des JAR jboss-seam.jar et jboss-seam-ui-jar que vous copiez sous la racine de votre EAR, sans oublier d’ajouter ces dépendances au fichier application.xml :

```
<module>
<java>jboss-seam.jar</java>
</module>
```

Vous disposez à présent d'une structure projet prête à fonctionner avec Seam.

La section suivante reprend certains cas d'utilisation de l'application webstock afin de les adapter à cette plate-forme.

Cas d'utilisation « Ajout d'un nouveau client »

Vous allez reprendre le cas d'utilisation « Ajout d'un nouveau client » de l'application webstock pour l'adapter au contexte Seam (voir la figure 12.11 du chapitre 12 décrivant le diagramme d'état associé à ce cas d'utilisation).

Rappelons que le cycle JSF sans Seam de ce use case se résume ainsi :

1. Une page xHTML présente le formulaire de saisie des paramètres du nouveau client, qui invoque la servlet AjoutNouveauClient.java.
2. La servlet fait appel au contrôleur AjoutClientController.java pour enregistrer les paramètres dans un bean managé sous forme de POJO, qui sert à contenir les différentes valeurs des paramètres saisis sur le formulaire d'ajout.
3. Le contrôleur se charge d'invoquer la méthode métier d'ajout du nouveau client sur le bean session.

La figure 13.6 illustre le processus d'une requête JSF dans le cadre de l'application webstock avant l'intégration de Seam.

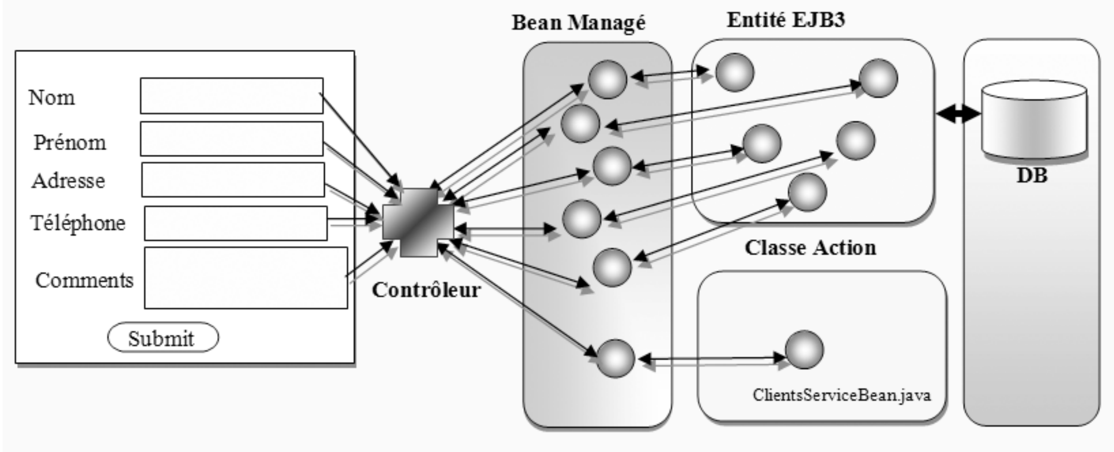


Figure 13.6

Use case « Ajout d'un nouveau client » avant l'intégration de Seam

Vous allez reprendre cette chaîne d'invocation des objets EJB3 afin de la refactorer selon le principe Seam.

La méthodologie Seam stipule qu'il faut supprimer tout bean managé pour « laisser la main » à la plate-forme. Il vous est simplement demandé d'ajouter les annotations nécessaires.

Voici les listing des différentes classes invoquées lors du déroulement du cas d'utilisation après leurs adaptation pour une intégration Seam.

DataModel `Client` :

```
@Name("Client")
@javax.persistence.Entity
@javax.persistence.Table(name = "CLIENT")
@javax.persistence.NamedQuery(name = "Client.findAll", query = "select client from
➥Client AS client")
public class Client
    implements java.io.Serializable, Comparable<Client>
{

    // ----------- Attribute Definitions ------------

    private java.lang.String comments;
    private java.lang.String telephone;
    private java.lang.String adresse;
    private java.lang.String prenom;
    private java.lang.String nom;
    private java.lang.Long id;


    // --------- Relationship Definitions -----------

    private java.util.Set<unnamed.Commande> commande = new java.util.TreeSet
    ➥<unnamed.Commande>();

@NotNull
@Range(min=3, max=100,
message="Prenom must be between 3 and 100")

@javax.persistence.Column(name = "PRENOM", nullable = false, insertable = true,
➥updatable = true)
    public java.lang.String getPrenom()
    {
        return prenom;
    }

    @NotNull
@Pattern(regex="^[a-zA-Z.-]+ [a-zA-Z.-]+",
message="Vous devez un commentaire")


    @javax.persistence.Column(name = "COMMENTS", nullable = false, insertable
    ➥= true, updatable = true)
    public java.lang.String getComments()
    {
        return comments;
    }

    @javax.persistence.Column(name = "TELEPHONE", nullable = false, insertable
    ➥= true, updatable = true)
    public java.lang.String getTelephone()
    {
        return telephone;
    }

@Length(max=250)
```

```
@javax.persistence.Column(name = "ADRESSE", nullable = false, insertable = true,
➥updatable = true)
    public java.lang.String getAdresse()
    {
        return adresse;
    }
```

La classe Client.java porte l'annotation @Name lui permettant d'avoir un nom dans le contexte Seam. Aucun tag @Scope n'étant déclaré, le contexte par défaut est utilisé. Dans ce cas, l'entité `client` est enregistrée comme étant un objet nommé `Client` dans le contexte conversationnel Seam.

Certaines annotations abordées au chapitre 8 sont utilisées ici pour la validation. Elles permettront de déclencher la validation JSF des attributs qu'elles annotent. Le formulaire « Ajout d'un nouveau client » en est un exemple.

La seconde classe à modifier est le bean session `ClientsServiceBean` qui déclare toutes les méthodes CRUD relatives au bean `Client` :

```
@Name("ClientsServiceBean")
@Scope(ScopeType.CONVERSATION)
public class ClientsServiceBean
    extends book.webstock.services.ClientsServiceBase
{
@In(required=true,create=true) @Out
    Client client;

public Client getClient() {
    return client;
}

public void setClient(Client client) {
    this.client = client;
}

    public void ajouterClient(){
    try {
        getClientDao().create(client);
    } catch (Exception e) {
        e.printStackTrace();
    }
    }
    // -------------- Constructors ---------------

    public ClientsServiceBean()
    {
        super();
    }

…
}
```

Le service "ClientsServiceBean" est enregistré comme étant un objet Seam appelé `ClientsServiceBean` vivant dans le scope conversationnel. Remarquez bien l'injection du paramètre « client » dans le bean session.

Maintenant que vous avez enregistré vos deux composants dans le contexte conversationnel, vous pouvez les invoquer directement sans passer par les beans managés habituels de JSF. Pour ce faire, reprenez votre facelet ajoutClient.xhtml pour une refactorisation Seam.

Voici le code du facelet avant refactoring :

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:c="http://java.sun.com/jstl/core"
      xmlns:t="http://myfaces.apache.org/tomahawk"
      xmlns:a="http://www.andromda.org/cartridges/jsf"
      xmlns:af="http://xmlns.oracle.com/adf/faces">
<ui:composition>
<h:form id="ajoutDUnNouveauClientFormulaireAjoutClientAjoutClientEffectForm"
➥enctype="multipart/form-data">
    <af:panelForm>
        <af:inputText id="nom" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.nom}" label
        ➥="#{messages['nom']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="prenom" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.prenom}" label
        ➥="#{messages['prenom']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="adresse" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.adresse}" label
        ➥="#{messages['adresse']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="telephone" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.telephone}" label
        ➥="#{messages['telephone']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="comments" value="#{ajoutDUnNouveauClient
        ➥FormulaireAjoutClientAjoutClientEffectForm.comments}" label
        ➥="#{messages['comments']}:" required="false" readOnly="false" rows="3"
        ➥columns="40">
        </af:inputText>
        <f:facet name="footer">
            <af:panelButtonBar>
                <af:commandButton text="#{messages['ajout.client.effect']}"
                ➥action="#{ajoutClientController.formulaireAjoutClient
                ➥AjoutClientEffect}"/>
            </af:panelButtonBar>
        </f:facet>
    </af:panelForm>
    <a:validator client="false"/>
</h:form>
</ui:composition>
</html>
```

Le code du facelet après sa refactorisation est le suivant (remarquez au passage que le code est plus compact) :

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
```

```
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:c="http://java.sun.com/jstl/core"
        xmlns:t="http://myfaces.apache.org/tomahawk"
        xmlns:a="http://www.andromda.org/cartridges/jsf"
        xmlns:af="http://xmlns.oracle.com/adf/faces">
<ui:composition>
<h:form >
    <af:panelForm>
        <af:inputText id="nom" value="#{Client.nom}" label="#{messages['nom']}:"
        ➥required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="prenom" value="#{Client.prenom}"
        ➥label="#{messages['prenom']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="adresse" value="#{Client.adresse}"
        ➥label="#{messages['adresse']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="telephone" value="#{Client.telephone}"
        ➥label="#{messages['telephone']}:" required="false" readOnly="false">
        </af:inputText>
        <af:inputText id="comments" value="#{Client.comments}"
        ➥label="#{messages['comments']}:" required="false" readOnly="false"
        ➥rows="3" columns="40">
        </af:inputText>
        <f:facet name="footer">
            <af:panelButtonBar>
                <af:commandButton text="#{messages['ajout.client.effect']}"
                ➥action="#{ClientsServiceBean.ajouterClient}"/>
            </af:panelButtonBar>
        </f:facet>
    </af:panelForm>
    <a:validator client="false"/>
</h:form>
</ui:composition>
</html>
```

Maintenant que tout est prêt, il ne vous reste qu'à ajouter le fichier components.xml pour JBoss Seam, comme dans le listing suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
                "http://jboss.com/products/seam/core http://jboss.com/products/seam/
                 ➥core-1.1.xsd
                 http://jboss.com/products/seam/components http://jboss.com/
                 ➥products/seam/components-1.1.xsd">

<core:ejb installed="false"/>
<component
        name="client"
        class="unnamed.Client"
        scope="CONVERSATION"/>
<component
        name="ClientsServiceBean"
```

```
        class="book.webstock.services.ClientsServiceBean"
        scope="CONVERSATION"
  jndi-name="testNG-1.0.0/ClientsServiceBean/remote"/>
    <!-- seam-components merge-point -->
</components>
```

Ce fichier doit se trouver sous le répertoire META-INF du projet EJB. Il utilise ici une description minimale pour déclarer les objets Seam (noms, classes et contexte correspondant). Le code complet est disponible sur la page Web dédiée à l'ouvrage.

Tout est prêt pour exécuter l'application webstock avec son « revêtement » Seam. Vous n'avez plus besoin de passer par les beans managés ni par les contrôleurs, évitant ainsi les redondances et laissant Seam gérer en toute transparence les synchronisations entre les couches métier et Web.

La figure 13.7 illustre votre use case après l'intégration Seam.

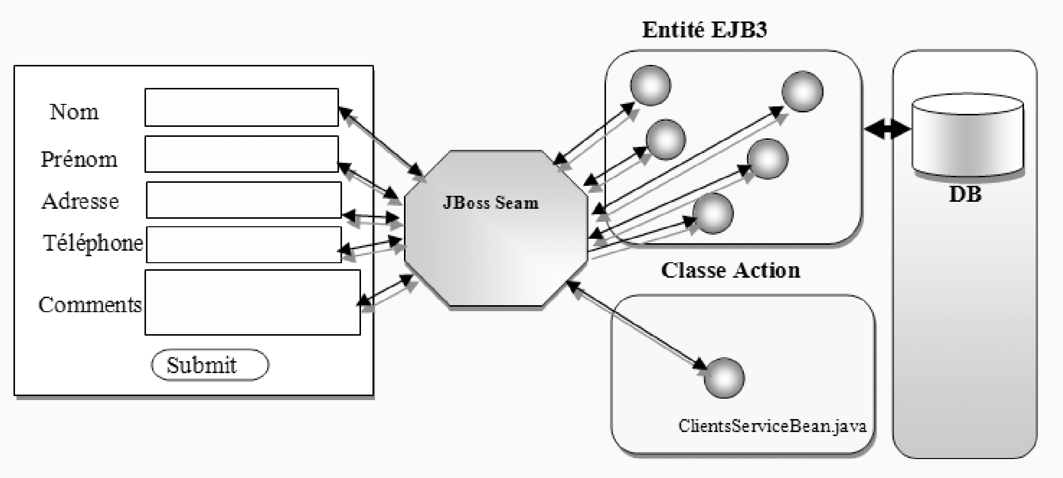


Figure 13.7

Use case « Ajout d'un nouveau client » après l'intégration Seam

Conversation Seam

Vous allez à présent appliquer une conversation Seam à l'application WebStock.

Vous prendrez cette fois le cas classique du Login, ou identification à l'application.

Voici le listing du bean session WebStockAccessServiceBean :

```
@javax.ejb.TransactionAttribute(javax.ejb.TransactionAttributeType.REQUIRED)
@javax.ejb.Remote({book.webstock.services.WebStockAccessServiceRemote.class})

@Scope(ScopeType.CONVERSATION)
@Name("WebStockAcess")

public class WebStockAccessServiceBean
    extends book.webstock.services.WebStockAccessServiceBase
{
```

```
// ------ Session Context Injection ------

@javax.annotation.Resource
protected javax.ejb.SessionContext context;

@In @Out
WebstockAccess user;

@In
FacesMessages facesMessages;


@RequestParameter("login")
String login;

@RequestParameter ("password")
String password;

// ------ Persistence Context Definitions --------

/**
 * Inject persistence context EJB3_JSF_SEAM_Webstock
 */
@javax.persistence.PersistenceContext(unitName = "EJB3_JSF_SEAM_Webstock")
protected javax.persistence.EntityManager emanager;

// ------ DAO Injection Definitions --------

/**
 * Inject DAO WebstockAccessDao
 */
@javax.ejb.EJB
private WebstockAccessDao webstockAccessDao;

// -------------- Constructors ---------------

public WebStockAccessServiceBase()
protected WebstockAccessDao getWebstockAccessDao()
{
    return this.webstockAccessDao;
}

// -------- Business Methods  --------------

/**
 *
 */
@Begin
@javax.ejb.TransactionAttribute(javax.ejb.TransactionAttributeType.REQUIRED)
public boolean login(java.lang.String login, boolean password)
{
    try
    {
        return this.handleLogin(login, password);
    }
    catch (Throwable th)
    {
```

```
            throw new book.webstock.services.WebStockAccessServiceException(
                "Error performing 'book.webstock.services.WebStockAccessService
                ➥.login(java.lang.String login, boolean password)' --> " + th,
                th);
        }
    }

    /**
     * Rechercher le user ayant ce login et ce password pour voir   si ça existe
     */

    Public WebStockAccess getUser(login, password){
        return getWebstockAccessDao.findUser(login, password);
    }

    @End
    public void displayLoginResult(){
    if (user==null)
        facesMessages.add("Error login action");
    else{facesMessages.add("you're logged in");}
    }

    /**
     * Performs the core logic for {@link #login(java.lang.String, boolean)}
     */
    protected abstract boolean handleLogin(java.lang.String login, boolean password)
        throws java.lang.Exception;

    public void logOut(WebstockAccess user){
    try
        {
            handlelogOut(user);
        }
        catch (Throwable th)
        {
            throw new book.webstock.services.WebStockAccessServiceException(
                "Error performing 'book.webstock.services.WebStockAccessService
                ➥.logout(user)' --> " + th,
                th);
        }
    }

    protected abstract boolean handlelogOut(WebstockAccess user)
    throws java.lang.Exception;


    /**
     * Remove lifecycle method
     */
    @Remove @Destroy
    public void destroy() {
        handleDestroy();
    }

    /**
     * Performs the core logic for {@link #destroy())
     */
    protected abstract void handleDestroy()}}
```

Le service WebStockAccessServiceBean est un bean session avec état qui implémente la logique métier de connexion à l'application webstock *via* un nom d'utilisateur et un mot de passe. La requête de login est élue pour être conversationnelle selon les termes Seam.

Le tableau 13.2 récapitule les annotations utilisées jusqu'ici dans la gestion du contexte Seam (voir en annexe pour plus de détails).

Tableau 13.2 Annotations de gestion des contextes Seam

Annotation Seam	Description
@Scope(ScopeType.CONVERSATION)	Permet au bean d'être enregistré dans le contexte conversationnel Seam.
@Begin	Permet de débuter la conversation Seam si la méthode qu'elle marque est invoquée. Dans le cas du Bean WebStockAccessBean, c'est la méthode `login()`.
@End	Permet d'arrêter la conversation si la méthode qu'elle annote est invoquée. Dans ce cas, la méthode `displayLoginResult()` arrête la conversation.
@Destroy	Utilisée pour tout clean-up effectué par Seam
@Remove	Informe la plate-forme que le bean session doit être détruit après l'invocation de la méthode qu'elle marque.

@Destroy et @Remove

Dans Seam, tous les beans session avec état doivent déclarer une méthode marquée @Destroy et @Remove. C'est la méthode EJB Remove() qui est invoquée quand Seam détruit le contexte Session.

En résumé

Tout au long de cette partie consacrée au développement EJB3 avec la plate-forme JBoss, vous avez pu apprécier toute la souplesse et la simplicité de la nouvelle norme EJB3, complétée par la richesse de l'API JPA et des sous-projets associés (Dali en particulier).

Avec l'apport du framework Seam, conjugué à une démarche de développement centrée sur le modèle avec MDA voir SOA, la spécification EJB3 devrait atteindre toute sa puissance pour le développement d'applications Web 2.0 complexes d'aujourd'hui et de demain.

Annexes

Cette partie comporte trois annexes regroupant un ensemble d'informations synthétiques destinées à servir de complément à cet ouvrage :

- Annexe A : procédures d'installation des produits utilisés dans l'ouvrage : Eclipse Europa, SVN et Tortoise SVN, EclipseUML, JBoss 4.2 et Apache Derby.
- Annexe B : description détaillée des principales annotations du framework Seam.
- Annexe C : principaux stéréotypes et valeurs balisées du profil EJB 3.0 pour AndroMDA utilisé par l'outil EclipseUML de l'éditeur Omondo.

Procédures d'installation

Eclipse Europa

Plate-forme de développement :

- Windows 2000 Professionnel, avec ServicePack 2 ou ultérieur.
- Windows XP Professionnel, Windows Vista.
- Windows NT Workstation ou Serveur Version 4.0 avec ServicePack 6a ou ultérieur.
- Linux.
- MacOS.

JRE (Java Runtime Environment) **:**

- JDK 5
- JDK 6

Serveur d'applications :

- JBoss 4.0 et JBoss 4.2

Base de données :

- Derby, Hypersonic

Europa est le nom de code de la dernière version finalisée d'Eclipse 3.3, disponible depuis le 29 juin 2007.

Plusieurs profils sont offerts pour aider le développeur à installer Eclipse avec les outils qui coïncident avec ses préférences. Dans notre cas, nous avons choisi le profil du développeur JEE pour Windows. Nous aurons donc un Eclipse enrichi de l'outillage Java et comportant WTP, ainsi que l'outillage de développement de plug-ins et d'applications RCP.

Téléchargement à l'adresse :

http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/20070927/eclipse-jee-europa-fall-win32.zip

La figure A.1 récapitule les plug-ins disponibles pour les différents profils Eclipse (doubles coches : inclus avec le code source ; simples coches : inclus ; coche estompée : partiellement inclus).

	Java	JEE	C/C++	RCP/Plugin	Classic
RCP/Platform	✓	✓	✓	✓✓	✓✓
CVS	✓	✓	✓	✓	✓✓
EMF	(✓)	✓		(✓)	
GEF	✓	✓		✓	
JDT	✓	✓		✓	✓✓
Mylyn	✓	✓		✓	
WST	(✓)	✓		(✓)	
PDE		✓		✓✓	✓✓
Datatools		✓			
JST		✓			
CDT			✓		

Figure A.1
Plug-ins disponibles selon le type de profil JST

Subversion et client TortoiseSVN sous Windows

1. Allez sur le site de Subversion *(http://subversion.tigris.org)*, et téléchargez le plus récent : svn-x.y.z-setup.exe.
2. Allez sur le site *http://svnservice.tigris.org/*, et téléchargez SVNService 0.5.2.zip si vous souhaitez démarrer votre serveur SVN par le biais des services Windows.
3. Allez sur le site *http://tortoisesvn.net/*, et téléchargez l'installateur le plus récent.

Serveur et du client en ligne de commande

1. Lancez svn-x.y.z-setup.exe, et suivez les étapes d'installation (Dans cet ouvrage, Subversion est installé dans le répertoire D:\Programmes\SVN (voir figure A.2).

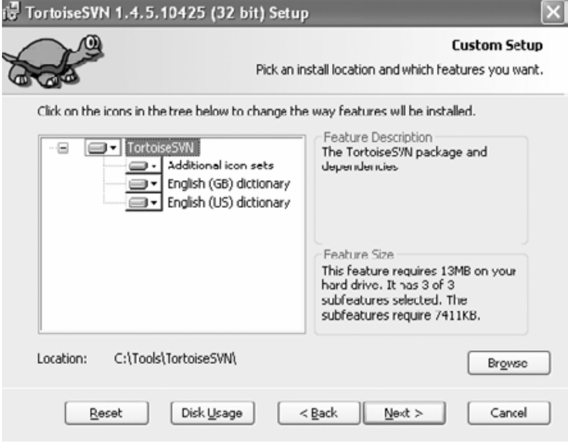


Figure A.2
Installation de TortoiseSVN

2. Cliquez sur Next puis Install, et attendez la fin du processus d'installation.
3. Cliquez sur Finish.

L'assistant vous propose de redémarrer votre poste pour que les changements prennent effet. Reprenez ensuite les étapes de configuration du repository ainsi que du démarrage du serveur sous forme de service (voir le chapitre 3).

Installation et configuration d'Apache Derby

L'installation de Derby est relativement simple :

1. Assurez-vous d'avoir installé et configuré un JRE 1.4 au minimum.
2. Récupérez Derby sur le site d'Apache, à l'adresse (http://db.apache.org/derby/derby_downloads.html), et décompressez la distribution.
3. Positionnez la variable système DERBY_INSTALL sous le répertoire ou Derby a été décompressé.
4. Ajoutez les fichiers JAR derby.jar et derbytools.jar (répertoire lib de DERBY_INSTALL) dans la variable classpath que vous aurez configurée *via* le Panneau de configuration de Windows.
5. Lancez la classe org.apache.derby.tools.sysinfo pour vérifier que le paramétrage de Derby est correct avec la commande java.org.apache.derby.tools.sysinfo, dont le résultat indique les versions de Derby et du JDK utilisées.
6. Saisissez la commande Java org.Apache.derby.tools.ij.
7. Le prompt ij doit apparaître. Pour créer une base de données nommée test et vous y connecter, entrez la commande suivante :

```
Ij> connect 'jdbc :derby :test;create=true' ;
```

8. La base de données est créée dans le répertoire de lancement de la JVM.
9. Pour exécuter un script SQL, par exemple MonScript.sql , entrez :

```
Ij> run 'monScript.sql' ;
```

10. Entrez 'exit' pour vous déconnecter de la base de données.

EclipseUML

Pour installer EclipseUML Europa, vous devez télécharger l'installateur Omondo, à l'adresse *http://www.ejb3.org/download_studio_JEE_eclipse_3.3.html*.

1. Double-cliquez sur le jar récupéré pour démarrer l'installation.
2. Si vous n'observez pas la première page de l'assistant, ouvrez une console DOS, et entrez la commande java –jar, suivie de l'adresse vers votre installateur, comme illustré à la figure A.3.

```
Sélectionner C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\OMONDO_TEAM>java -jar "F:\builds 30-10-2007\eclipseUML
_E330_2007_studioEdition_JEE_3.3.0.v20071030.jar"
```

Figure A.3

Démarrage de l'installateur EclipseUML

3. La page qui s'affiche permet à l'utilisateur de choisir la langue de l'installateur, mais pas d'EclipseUML (attention), comme l'illustre la figure A.4.

Figure A.4
Sélection de la langue de l'installateur

4. Cliquez sur OK. Vous êtes redirigé vers la page de bienvenue de l'installateur.
5. Cliquez sur Suivant.
6. La page suivante vous propose des informations sur la licence EclipseUML. Acceptez-en les termes pour poursuivre l'installation.
7. Cliquez sur Suivant.
8. Mentionnez le chemin d'installation de votre Eclipse Europa.

Figure A.5
Sélection du chemin d'installation du plug-in EclipseUML

9. La boîte de dialogue de configuration des paquetages à installer s'affiche comme illustré à la figure A.6. Notez bien si vous avez installé Eclipse.
10. Le processus d'installation proprement dit débute.
11. Cliquez sur Quitter une fois l'installation terminée.

Figure A.6

Sélection des packages

12. Pour vérifier qu'EclipseUML est bien installé, sélectionnez Help puis About Eclipse Plateform (voir figure A.7).

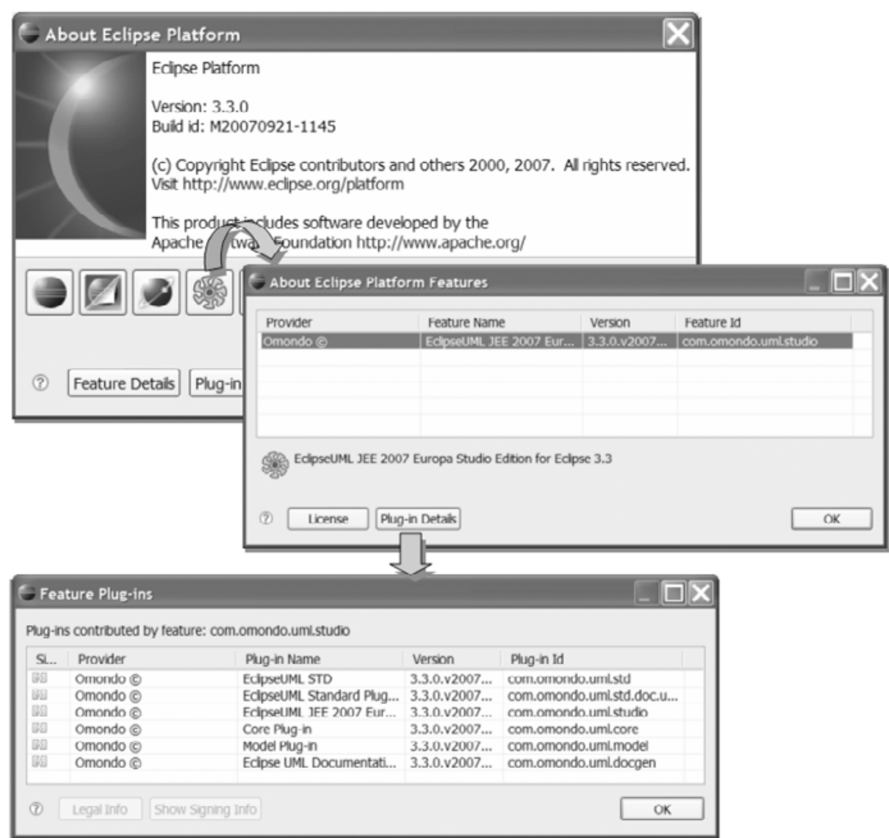


Figure A.7

Finalisation de l'installation d'EclipseUML

Paramétrage JVM pour Eclipse

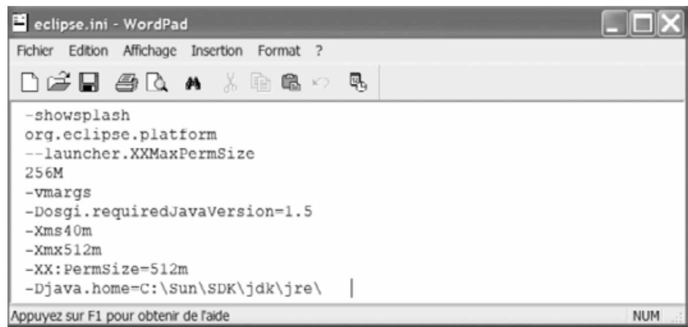
Pour commencer à travailler avec EclipseUML sans crash de mémoire ni problèmes lors de la génération, les configurations mineures suivantes sont requises :

1. Éditez le fichier $ECLIPSE_HOME/eclipse.ini, où sont déclarées les variable de lancement de l'éditeur. Ajoutez les variables suivantes à votre fichier :

xmx	Limite l'utilisation de la mémoire à la valeur indiquée. Attention aux problèmes de dépassement de mémoire.
PermSize	Redéfinit la mémoire permanente utilisée.
java.home	Indique le chemin associé à la variable JAVA_HOME.

2. Faites en sorte que le fichier eclipse.ini ressemble à celui illustré à la figure A.8.

Figure A.8
Fichier eclipse.ini



JBoss 4.2

La version 4.2 de JBoss supporte nativement la spécification EJB3, contrairement à a version JBoss 4.0, qui nécessitait une installation spécifique du conteneur EJB3.

1. Si vous souhaitez installer la version 4.0.x, utilisez l'installateur graphique *(http://prdownloads.sourceforge.net/jboss/jems-installer-1.2.0.CR1.jar?download)*.
2. Une fois téléchargé, exécutez l'installateur en double-cliquant sur le fichier (si les fichiers JAR ne sont pas associés à Java, l'exécution de l'installateur peut se faire *via* la commande java -jar ems-installer-1.2.0.CR1.jar.

Pour les versions de JBoss supérieures à 4.2, l'installation de JBoss 4.2.x.GA, disponible sur le site de sourceforge *(http://labs.jboss.com/jbossas/downloads/)* se limite à une simple décompression de l'archive sous le répertoire, en supposant un JDK5 installé (positionnement de la variable JAVA_HOME préalable avant installation du serveur JBoss).

Seam

1. Pour installer Seam, assurez-vous que vous possédiez un JDK1.5, Ant (ou Eclipse) et une installation JBoss avec support d'EJB3.
2. Téléchargez l'archive Seam (*http://www.jboss.com/products/list/downloads#seam*), et décompressez-la sur votre disque en dehors de l'arborescence serveur JBoss que vous venez d'installer.
3. Éditez le fichier build.properties se trouvant sous la racine du projet Seam afin qu'il pointe la variable JBOSS_HOME vers le chemin d'installation du serveur JBoss.
4. À partir d'Eclipse ou à travers une console DOS (si vous avez Ant proprement installé), exécutez le script build.xml situé sous la racine du projet Seam.

5. À la fin de l'exécution du script, un nouveau répertoire est créé, jboss-seam, suivi de la date du jour, par exemple : jboss-seam-Octobre 18 2007. Ce répertoire comporte les jars Seam nécessaires pour développer votre projet.
6. Dans le cadre de l'application exemple de l'ouvrage, vous n'avez besoin que du fichier jboss-seam.jar, que vous devez copier sous la racine de votre ear (sans oublier d'ajouter cette dépendance au fichier application.xml) :

```
<application>
    <display-name> Seam pour les debutants </display-mode>

    <module>
       <web>
         <web-uri>MonAppliWebSeam.war</web-uri>
         <context-root>/monAppliSeam</context-root>
       </web>
     </module>
     <module>
        <ejb>monEJBs.jar</ejb>
     </module>
     <module>
   <java>jboss-seam.jar</java>
    </module>
</application>
```

7. Si vous déployez votre application sous la forme de fichier war, vous n'avez qu'à copier le fichier jboss-seam.jar dans le répertoire WEB-INF/lib de l'archive. En plus de ce fichier il existe plusieurs bibliothèques de support que vous pouvez ou non inclure à l'archive de l'application, selon la manière dont vous utilisez Seam dans vos projets.

Les différents fichiers de configuration sont récapitulés au tableau suivant.

Nom de la ressource	Description	Contrainte
META-INF/application.xml	Descripteur de déploiement de l'application JEE définissant les modules et les war de l'application ear	Non utilisé par Seam
META-INF/ejb-jar.xml	Descripteur de déploiement EJB3 pour les EJB.jar définissant les différents intercepteurs	Nécessaire pour déclarer l'Intercepteur Seam-Interceptor de Seam pour la couche métier
META-INF/persistence.xml	Permet de configurer l'unité persistante.	Non utilisé par Seam
META-INF/jboss-app.xml	Utilisé pour configurer le ClassLoader JBoss	Non utilisé par Seam
META-INF/jboss-beans.xml	Descripteur de déploiement pour le micro-conteneur JBoss	Non utilisé par Seam
WEB-INF/web.xml	Descripteur de déploiement pour le war	Non utilisé par Seam
WEB-INF/faces-config.xml	Déclare les règles de navigation et les composants JSF.	Non utilisé par Seam
WEB-INF/jboss-web.xml	Descripteur de déploiement JBoss pour un war	Non utilisé par Seam
WEB-INF/pages.xml	Définit les action Seam (Seam page action) avant d'afficher une page donnée.	Peut être vide.
WEB-INF/events.xml	Événements Seam	Peut être vide.
Seam.properties	Fichier de configuration des composants Seam	Peut être vide.
Hibernate.cfg.xml	Utilisé pour la configuration d'Hibernate	Non utilisé par Seam
Jbpm.cfg.xml	Configuration JBPM	Non utilisé par Seam
Classes/messages.properties	Fichier de ressource Seam	Peut être vide.

Installation des bibliothèques EJB3 embarquées

Vous pouvez utilisez le conteneur JBoss EJB 3.0 embarqué avec Seam dans votre application. Le fait d'utiliser le conteneur embarqué EJB 3.0 dans votre application vous permet d'exécuter vos composants EJB 3.0 sans avoir à vous soucier des fonctionnalités de votre serveur d'applications.

Vous pouvez de la sorte exécuter différentes applications sur votre serveur, comme certaines embarquant un environnement EJB 2.1, d'autres le conteneur embarqué EJB 3.0.

Les bibliothèques spécifiques requises sont les suivantes (localisées dans le répertoire lib de la distribution Seam) :

- hibernate-all.jar, qui inclut toutes les classes Hibernate que le conteneur embarqué utilise pour l'implémentation JPA.
- thidparty-all.jar, qui inclut des fichiers utilitaires supplémentaire utilisés par le conteneur.
- jboss-embedded-all.jar, qui contient toutes les classes implémentant le conteneur EJB 3.0 proprement dit.

Ces bibliothèques peuvent être incluses dans le répertoire racine de votre fichier ear ou dans le répertoire WEB-INF/lib de votre fichier WAR, en fonction du mode de packaging et de déploiement de votre application.

Structure projet des applications Web POJO

Le chapitre 8 a présenté un cas pratique pour l'utilisation de Seam dans le cadre d'une application Web sans EJB (POJO).

La structure d'un projet Web utilisant Seam pour une application Web POJO est illustrée à la figure A.9.

Figure A.9

Structure d'un projet Web POJO

Les composants Seam sont ajoutés dans le dossier source src. Les classes de test pour testNG sont ajoutées dans le dossier source test (il faut également configurer les fichiers de configuration de tests components.xml et testNG.xml).

Dans le META-INF du projet Web, on trouve le descripteur JPA persistence.xml pour la configuration de la source de données du projet.

Sous le WEB-INF, on trouve le reste des descripteurs pour Seam et pour l'application Web.

Structure projet pour les applications EJB3

Au chapitre 13, nous avons utilisé Seam dans un contexte EJB3. La structure de projet utilisée dans ce cadre est générée par EclipseUML mais aurait pu l'être à l'aide d'un fichier de script build.xml (les différents scripts Ant nécessaires sont disponibles sur la page Web dédiée à l'ouvrage).

La figure A.10 illustre la structure projet correspondante.

Figure A.10

Structure d'un projet Web EJB3

La structure projet est identique à la précédente, à quelques descripteurs près.

B

Annotations Seam

Nous présentons ici les principales annotations Seam, dont un certain nombre sont définies dans la spécification EJB 3.0.

Annotations au niveau composant

Les annotations suivantes permettent de définir un composant Seam. Elles apparaissent au niveau de la classe du composant.

@Name

Définit le nom du composant pour la classe (requise pour tous les composants Seam).

```
@Name (" MonComposant" )
```

@Scope

Définit le contexte par défaut du composant. Les valeurs possibles sont définies par l'énumération ScopeType, soit EVENT, PAGE, CONVERSATION, SESSION, BUSINESS_PROCESS , APPLICATION, STATELESS. Lorsque la portée n'est pas explicitement spécifiée, la valeur par défaut dépend du type de composant. Pour les beans session stateful, la valeur par défaut est CONVERSATION. Pour les composants Java-Beans la valeur par défaut EVENT.

```
@Scope(ScopeType.CONVERSATION)
```

@Role

Permet à un composant Seam d'être relié à plusieurs variables contexte.

Propriétés

- name : nom de la variable contexte.
- scope : portée de la variable contexte.

```
@Role (name=" monRole" , scope=ScopeType.SESSION)
```

@BypassInterceptors

Désactive tous les intercepteurs Seam sur un composant particulier ou une méthode d'un composant.

@JndiName

Spécifie que la portée du composant est conversationnelle, ce qui signifie qu'aucune méthode du composant ne peut être appelée à moins qu'une conversation longue ne soit active.

```
@JndiName("ejb/session/MonBeanSession")
Public class MonBeanSession{
}
```

@Startup

Utilisée en combinaison avec @Scope, spécifie qu'un composant de portée application est démarré immédiatement lors de l'initialisation de l'application (utile par exemple pour démarrer certaines ressources critiques, comme les sources de données).

Paramètres

- depends : spécifie le nom du composant qui doit être démarré en premier.

```
@Scope (SESSION) @Startup (depends= "org.jboss.seam.bpm.jbpm")
```

@Install

Spécifie si le composant doit être ou non installé par défaut. L'absence de cette annotation indique que le composant doit être installé.

Propriétés

- dependencies : spécifie que le composant doit être installé si les composants listés comme dépendants sont aussi installés.
- genericDependencies : spécifie qu'un composant doit être installé si un composant implémenté avec une classe particulière est installé.
- classDependencies : spécifie la précédence du composant. S'il existe plusieurs composants de même nom, c'est celui ayant la plus haute priorité qui est installé. Les valeurs de priorités sont les suivantes (ordre ascendant) :
  - BUILT_IN : priorité sur tous les composants Seam installés.
  - FRAMEWORK : priorité d'utilisation pour les composants des frameworks qui étendent Seam.
  - APPLICATION : priorité sur les composants de l'application (priorité par défaut).
  - DEPLOYMENT : priorité d'utilisation pour les composants qui surchargent les composants de l'application dans un déploiement spécifique.
  - MOCK : priorité pour les objets simulés (MOCK) utilisés pour les tests.

```
@Install(precedence=BUILT_IN)
```

@Synchronized

Spécifie qu'un composant est accédé de manière concurrente par plusieurs clients et que Seam doit sérialiser les requêtes. Si la requête est dans l'impossibilité d'obtenir un verrou sur le composant pendant la période spécifiée par le paramètre timeout, une exception est levée.

Propriétés

- timeout : durée en seconde de l'attente du verrou à partir de laquelle une exception est levée.

```
@Synchronized (timeout=2000)
```

@ReadOnly

Spécifie qu'un composant JavaBean ou une méthode d'un composant ne nécessite pas de réplication de l'état à la fin de l'invocation.

@AutoCreate

Spécifie qu'un composant sera automatiquement créé, même si le client ne spécifie pas create=true.

Annotations pour la bijection

@In

Spécifie que l'attribut d'un composant est injecté dans une variable de contexte au début de chaque invocation du composant. Si la variable de contexte est nulle, une exception est levée.

Propriétés

- value : spécifie le nom de la variable de contexte qui peut être substituée par une expression JSF EL, délimitée par le caractère #{…}.
- create : spécifie que Seam doit instancier le composant avec le même nom que la variable de contexte si cette dernière est indéfinie pour tous les contextes.
- required : spécifie que Seam doit lever une exception si la variable de contexte est indéfinie pour tous les contextes.

```
@In(required=false)
@In
private User user
```

@Out

Spécifie que l'attribut du composant est *outjected* vers une variable de contexte une fois la méthode appelée (principe d'injection de dépendances). La variable spécifiée par l'annotation dans le contexte est automatiquement substituée après chaque appel. Si l'attribut est nul, une exception est levée.

Propriétés

- value : spécifie le nom de la variable de contexte, par défaut le nom de l'attribut du composant.

- required : spécifie que l'attribut du composant Seam est *outjected* vers la variable de contexte à la fin de l'invocation. L'attribut peut être null (required=false) durant l'outjection.
- scope : spécifie qu'un attribut du composant qui n'est pas du type de composant Seam doit être outjected vers un scope spécifique à la fin de l'invocation. Si aucun scope n'est explicitement spécifié, le scope du composant avec l'attribut @Out est utilisé (ou le scope EVENT si le composant est stateless).

```
@In(create=true) @Out private User currentUser;
```

@Factory

Spécifie que la méthode du composant est utilisée pour initialiser la valeur de la variable de contexte spécifiée, lorsque la variable de contexte n'a pas de valeur.

Propriétés

- value : spécifie le nom de la variable de contexte, par défaut le nom de l'attribut du composant.
- required : spécifie que l'attribut du composant Seam est *outjected* vers la variable de contexte à la fin de l'invocation. L'attribut peut être null (required=false) durant l'outjection.
- scope : spécifie qu'un attribut du composant non-Seam doit être outjected vers un scope spécifique à la fin de l'invocation. Si aucun scope n'est explicitement spécifié, le scope du composant avec l'attribut @Out est utilisé (ou le scope EVENT si le composant est stateless).

```
@Factory("items")
    public void getItems() {
        if ((items==null)  || (index != firstItem) ){
            getNextItems();
        }
```

Annotations pour les méthodes du cycle de vie du composant

Ces annotations permettent au composant de réagir à son propre cycle de vie des événements sur les méthodes du composant.

@Create

Spécifie que la méthode doit être invoquée lorsqu'une instance du composant est instanciée par Seam. Les méthodes create sont uniquement applicables pour les JavaBeans et les beans session stateful.

```
    @Name("DatabaseUtils")
Public class DatabaseUtils{

    @Create()
    public void initDatabase(){
    }
}
```

@Destroy

Spécifie que la méthode doit être appelée lorsque le contexte se termine et que les variables de contexte associées au contexte sont supprimées. Les méthodes destroy sont uniquement supportées par les JavaBeans et les beans session stateful.

```
@Name("DatabaseUtils")
Public class DatabaseUtils{

    @Destroy()
    public void closeDbResources(){
    }
}
```

@Observer

Spécifie que la méthode doit être appelée lorsqu'un événement sur un composant survient.

Paramètres

- value : spécifie le nom de la variable de contexte, par défaut le nom de l'attribut du composant.
- create : si l'instance n'existe pas et que create est à false, l'événement produit est ignoré. Par défaut à true.

```
@Observer(value="ModificationEffectuee",create=false)
```

Annotations pour la démarcation du contexte

Ces annotations s'appliquent à la gestion déclarative du contexte conversationnel.

@Begin

Marque la méthode comme démarrant une conversation « longue » Seam.

Paramètres

- join : valeur true ou false. Spécifie que si une conversation longue est déjà en progression, le contexte conversationnel est simplement propagé. Si false (défaut), l'invocation de la méthode begin dans le scope d'une conversation en cours provoque une exception.
- Nested : valeur true ou false. Si actif (valeur true), et si une conversation est en cours, démarre une conversation imbriquée, au lieu de continuer dans le contexte de la conversation existante.
- Flushmode : positionne le mode flush pour chaque session Hibernate Seam managée ou contexte persistant JPA créé durant la conversation (par défaut AUTO).
- Pageflow : nom de la définition du processus jBPM définie pour le flux de la page pour une conversation particulière.

```
// Specify that this method starts a long running conversation
    @Begin
    public String find() {
        hotel = null;
        String searchPattern = searchString == null ? "%" : '%' +
        ➥searchString.toLowerCase().replace('*', '%') + '%';
```

```
        hotels = em.createQuery("from Hotel where lower(name) like :search or
        ➥lower(city) like :search or lower(zip) like :search or lower(address)
        ➥like :search")
                .setParameter("search", searchPattern)
                .setMaxResults(50)
                .getResultList();

        log.info(hotels.size() + " hotels found");

        return "main";
    }
```

@End

Spécifie qu'une conversation « longue » Seam se termine et supprime les variables de contexte associées.

Paramètres

- Beforeredirect : valeur true ou false. Par défaut (valeur=false), la conversation n'est pas réellement détruite, à moins qu'une redirection n'ait été effectuée. Sinon (valeur true), spécifie que la conversation est détruite à la fin de la requête courante et que la redirection est traitée dans un nouveau contexte de conversation temporaire.

```
@End @Destroy @Remove
    public String confirm() {
        if (booking == null || hotel == null) return "main";
        em.persist(booking);
        log.info("booking confirmed");
        return "confirmed";
    }
```

@StartTask

Démarre une tâche jPBM.

Paramètres

- taskIdParameter : nom du paramètre de la requête sous lequel localiser l'identifiant de la tâche qui doit être lancée.
- flushmode : positionne le mode flush pour n'importe quel gestionnaire d'entité utilisé pour cette conversation.

@BeginTask

Reprend la tâche jBPM précédemment démarrée.

Propriétés

- taskIdParameter : nom du paramètre de la requête sous lequel localiser l'identifiant de la tâche qui doit être reprise.
- flushmode : positionne le mode flush pour n'importe quel gestionnaire d'entité utilisé pour cette conversation.

@EndTask

Termine une tâche jBPM.

Propriétés

- transition : nom de la transition jBPM déclenchée lorsque la tâche est terminée (par défaut, nom de la transition par défaut).
- beforeRedirect (valeur true ou false) : permet de spécifier si la conversation est détruite avant toute redirection. Le comportement par défaut propage la conversation à travers le redirect et la détruit à la fin de la requête de redirection.

@CreateProcess

Crée une nouvelle instance de processus jBPM.

Propriétés

- definition : nom de la définition du processus jPBM déployé *via* org.jboss.seam.bpm.jbpm.processDefinitions.

@ResumeProcess

Entre à nouveau dans le scope d'une instance jBPM existante.

Propriétés

- processIdparameter : nom du paramètre de la requête sous lequel localiser l'identifiant du processus qui doit être repris.

@Transition

Spécifie une méthode pour la transition jBPM après que celle-ci retourne un résultat non null et sans exception.

Propriétés

- value : nom de la transition (par défaut, nom de la méthode).

```
@Transition("cancel")
```

Annotations pour l'utilisation des composants JavaBean Seam dans un environnement J2EE

Les annotations suivantes permettent de spécifier la stratégie de démarcation transactionnelle.

@Transactional

Spécifie qu'un composant JavaBean doit avoir un comportement transactionnel similaire au comportement par défaut d'un composant bean session. Les invocations de méthodes doivent prendre place dans la transaction. Si aucune transaction n'existe lorsque la méthode est invoquée, une transaction est démarrée juste pour cette méthode.

Propriétés

- value : type de programmation transactionnelle.

Annotations pour le support des exceptions

Ces annotations permettent de spécifier comment le framework Seam doit gérer une exception qui se propage en dehors d'un composant Seam.

@Redirect

Provoque un débranchement vers la page spécifiée par le paramètre viewId.

Propriétés

- viewId : spécifie l'identifiant de la page JSF pour la redirection. Vous pouvez également utiliser une expression JBoss EL (Expression Language).
- message : message qui sera affiché.
- end : spécifie qu'une conversation longue doit se terminer (par défaut, false).

```
@Redirect (viewId="error.jsp")
```

@HttpError

Provoque une erreur HTTP.

Propriétés

- errorcode : le code erreur http (par défaut, 500).
- message : message qui sera envoyé avec l'erreur HTTP (par défaut, le message d'exception).
- end : spécifie qu'une conversation longue doit se terminer (par défaut, false).

```
@HttpError (errorCode=404)
```

Annotations pour Seam Remoting

Le framework Seam fournit un mécanisme rapide et puissant pour l'accès aux composants à partir d'une page Web, en utilisant AJAX (Asynchronous Javascript and XML). Cela s'effectue avec le framework Seam Remoting, dont l'emploi nécessite que l'interface locale d'un bean session soit annotée avec @WebRemote.

@WebRemote

Spécifie que la méthode distante annotée peut être appelée à partir d'un client JavaScript.

Propriétés

- exclude : propriété optionnelle permettant aux objets d'être exclus du graphe d'objets produit.

```
@Local
  public interface HelloLocal {
    @WebRemote
    public String sayHello(String name);
  }
```

Annotations pour le support des intercepteurs Seam

L'annotation @Interceptor n'est pas une annotation Seam spécifique. Elle est utilisée par Seam pour mettre en œuvre les mécanismes de bijection, de validation et d'interception des invocations de composant.

@Interceptor

Met en œuvre le mécanisme d'interception Seam.

Propriétés

- Stateless : valeur true ou false. Spécifie que l'intercepteur est stateless.
- type : spécifie que l'intercepteur est de type client et qu'il est invoqué avant le conteneur EJB (type=CLIENT).
- around : spécifie l'ordre de précédence de l'intercepteur.
- within : spécifie que l'intercepteur est positionné plus profondément dans la pile que les intercepteurs spécifiés.

```
@Interceptor(around={unIntercepteur.class, unAutreIntercepteur.class})

@Interceptor(within={unIntercepteur.class, unAutreIntercepteur.class})
```

Annotations pour la gestion des méthodes asynchrones

L'annotation @Asynchronous est utilisée pour déclarer des méthodes asynchrones.

@Asynchronous

Spécifie que l'appel de la méthode est effectué de manière asynchrone. Ce mécanisme s'adapte parfaitement à l'utilisation du framework AJAX lorsque le client peut interroger le serveur de manière automatique pour recevoir le résultat d'une tâche fonctionnant en arrière-plan.

Propriétés

- Les annotations @Duration, @Expiration, @IntervalDuration s'utilisent conjointement avec @Asynchronous pour permettre une planification des méthodes asynchrones associées.

```
@Local
public interface PaymentHandler
{
    @Asynchronous
    public void processScheduledPayment(Payment payment, @Expiration Date date);

    @Asynchronous
    public void processRecurringPayment(Payment payment, @Expiration Date date,0
    ➥@IntervalDuration Date interval)'
}
```

Annotations pour l'utilisation avec JSF

Les annotations suivantes facilitent l'implémentation de listes sélectionnables qui s'appuient sur les beans session stateful.

@DataModel

Outjecte une propriété (la propriété devenant ainsi accessible aux variables JSF) de type List, Map, Set ou Objects, comme type DataModel JSF dans la portée du composant propriétaire (ou dans la portée event si le composant propriétaire est stateless). Dans le cas particulier d'un type Map, chaque ligne de DataModel est une Map.Entry.

Propriétés

- value : nom de la variable de contexte de conversation (par défaut, nom de l'attribut).
- scope : si scope=ScopeType.PAGE est explicitement spécifiée, DataModel est conservé dans le contexte PAGE.

```
@DataModel
private List<Book> books;
```

@DataModelSelection

Injecte la ligne de donnée sélectionnée de l'objet DataModel à utiliser conjointement avec l'annotation @DataModel.

Propriétés

- value : nom de la variable de contexte de conversation (non requis s'il existe une annotation @DataModel dans le composant).

```
@DataModel
private List<Book> bookList;
@DataModelSelection
Book book;
```

@DataModelSelectionIndex

Injecte la ligne de donnée sélectionnée de l'objet DataModel à utiliser conjointement avec l'annotation @DataModel.

Propriétés

- value : nom de la variable de contexte de conversation (non requis s'il existe une seule annotation @DataModel dans le composant).

```
@DataModel
private List<Book> bookList;
@DataModelSelection
Book book;
```

Annotations pour l'intégration avec le conteneur de servlets

Ces méta-annotations permettent d'implémenter des fonctionnalités similaires à @DataModel et @dataModelSelection pour les autres structures de données indépendamment des listes.

@DataBinderClass

Spécifie que l'annotation est de type databinding, c'est-à-dire qu'elle résulte de l'outjection d'une représentation « enveloppe » de la valeur de l'attribut du composant annoté.

```
@DataBinderClass(DataModelBinder.class)
```

@DataSelectorClass

Spécifie que l'annotation est une annotation de sélection de donnée, c'est-à-dire qu'elle résulte de l'injection de l'élément sélectionné de type de donnée databound.

```
@DataSelectorClass(DataModelSelector.class)
```

Annotations pour le packaging

Ces annotations fournissent un mécanisme de déclaration pour un ensemble de composants qui seront packagés ensemble. Peut être appliqué à chaque package Java.

@Namespace

Spécifie la configuration de l'espace de nom du package Java contenant l'ensemble des composants Seam.

Propriétés

- value : espace de nom de la configuration pour le packaging.
- prefix (en option) : spécifie le préfixe du nom du composant à appliquer pour les noms de composants spécifiés dans le fichier de configuration XML.

```
@Namespace(value="http://jboss.com/products/seam/core", prefix="org.jboss.seam.core")
```

C

Annotations AndroMDA pour le profile EJB3

Cette section présente les principaux stéréotypes et valeurs balisées du profil EJB3 pour AndroMDA mis en œuvre avec l'atelier de modélisation MDA EclipseUML for JEE.

Rappelons qu'un stéréotype est un nom qui apparaît sur un diagramme. Souvent désigné par la chaîne <<MonStereotype>>, il peut être appliqué à n'importe quel élément d'un diagramme UML.

Les l ments du profil sont organisés selon trois éléments clés du diagramme de classes, à savoir la description des éléments du profil applicables pour les classes, les attributs et les méthodes.

Les tableaux suivants récapitulent les éléments principaux du profil EJB 3.0 pour AndroMDA. Davantage d'informations sont disponibles à l'adresse *http://web.aanet.com.au/persabi/andromda/profile.html.*

Valeurs balisées du profil EJB3 Persistence::EJB pour les classes

Valeur balisée	S'applique à	Valeurs possibles	Description
noSyntheticCreateMethod : @andromda.ejb.noSyntheticCreateMethod	Bean entité	Chaîne de caractères	Utilisée sur les entités pour permettre leur création synthétique (autogénération)
automaticKey : @andromda.ejb.automaticKey	Bean entité ou attribut	true false	Si true, génération automatique de la clé primaire

Valeurs balisées du profil EJB3 Persistence::Persistence pour les classes

Valeur balisée	S'applique à	Valeurs possibles	Description
inheritance @andromda.persistence.inheritance	Entité	SINGLE_TABLE TABLE_PER_ClASS JOINED	utilisée pour redéfinir l'héritage par défaut pour les entités
Entity_cache : @andromda.persistence.entity.cache	Entité	NONE NONSTRICT_READ_WRITE READ_ONLY READ_WRITE TRANSACTIONAL	Définit la stratégie du cache pour les entités.
discriminator_column_name : @andromda.persistence.discriminator.column.name	Entité	string	Optionnelle : définit le nom de la colonne discriminante.
discriminator_column_definition : @andromda.persistence.discriminator.column.definition	Entité	fragment SQL	Optionnelle : indique le fragment SQL utilisé pour la génération du DDL pour la colonne discriminante.
Table : @andromda.persistence.table	Entité ou association Many-to-Many	Chaine de caractères	Définit la table relationnelle utilisée pour l'entité persistée.
discriminator_type : @andromda.persistence.discriminator.type	Entité	string char integer	Utilisée pour redéfinir le type de discriminateur défini dans la propriété entityDiscriminatorType du namespace.
discriminator_column_length : @andromda.persistence.discriminator.column.length	Entité	Longueur de la colonne	Indique la longueur de la colonne des discriminateurs en chaînes de caractères. Ignorée pour les autres types de discriminateurs.

Stéréotypes du profil EJB3 Service::Service pour les classes

Stéréotype	S'applique à	Description
Service	Bean session	Stéréotype utilisé pour désigner un bean session
PersistenceContext	Classe	Identifie un contexte de persistence.
Listener	Classe	Définit une classe Listener pour les beans entité, session ou MDB.
UserTransaction	Classe	Utilisé pour injecter une ressource javax.transaction.UserTransaction
MessageDriven	Classe	Produit une façade de MDB JMS qui reçoit les messages.

Valeurs balisées du profil EJB3 Persistence::Service pour les classes

Valeur balisée	S'applique à	Valeurs possibles	Description
persistence_context_unit_name : @andromda.service.persistence.context.unit.name	Bean session ou classe stéréotypée PersistenceContext	Chaîne de caractères	Utilisée pour définir le nom du contexte de persistence
persistence_context_datasource : @andromda.service.persistence.context.datasource	Bean session ou classe stéréotypée PersistenceContext	Nom JNDI de la source de données	Utilisée pour définir le nom JNDI de la source de données du contexte de persistence
jndi_remote : @andromda.service.jndi.remote	Bean session	Nom JNDI de l'interface distante	Spécifie manuellement le nom JNDI de l'interface distante.
jndi_local @andromda.service.jndi.local	Bean session	Nom JNDI de l'interface locale	Spécifie manuellement le nom JNDI de l'interface locale.
security_permitall : @andromda.service.security.permitAll	Bean session	true false	Donne des permissions sur l'exécution de l'opération du Bean.
security_denyAll : @andromda.service.security.denyAll	Bean session ou une de ses opérations	true false	Interdit les permissions sur l'exécution de l'opération du Bean.

Valeurs balisées du profil EJB3 Service::EJB pour les méthodes

Valeur balisée	S'applique à	Valeurs possibles	Description
transactionType : @andromda.ejb.transaction.type	Opération d'un service ou attribut d'une entité	NotSupported Supports Required RequiresNew Mandatory Never	Définit le type de transaction.
viewType : @andromda.ejb.viewType	Service ou opération d'un service	remote (default) local both	Définit le type de vue pour un bean session au niveau classe et au niveau de l'opération pour chaque méthode session.

Valeurs balisées du profil EJB3 Persistence::Persistence pour les attributs

Valeur balisée	S'applique à	Valeurs possibles	Description
generator_name : @andromda.persistence.generator.name	Attribut d'une entité	string	Définit un nom unique qui sera référencé par une ou plusieurs classes pour être le générateur d'un bean entité.
column_precision : @andromda.persistence.column.precision	Attribut d'une entité	Valeur décimale	Spécifie la précision pour une colonne décimale.
generator_type : @andromda.persistence.generator.type	Attribut d'une entité	auto (defaut) sequence identity table generic none	Spécifie la stratégie de génération de la clé primaire.
column_definition : @andromda.persistence.column.definition	Attribut d'une entité	SQL	Spécifie le SQL pour créer le DDL pour la propriété de l'entité.

Valeurs balisées du profil EJB3 Persistence::Persistence pour les attributs *(suite)*

Valeur balisée	S'applique à	Valeurs possibles	Description
collection_type : @andromda.persistence.collection.type	Association ends entre deux entités	set map list bag	Utilisée pour définir le mapping des collections EJB3
column : @andromda.persistence.column	Attribut d'une entité	Chaine de caractères	Représente le nom de la colonne utilisée pour l'attribut.
temporal_type : @andromda.persistence.temporal.type	Attribut d'une entité ou paramètre d'une méthode taggée (finderMethod)	date (default) time timestamp none	Cette valeur balisée est spécifiée sur les attributs de type java.util.Date ou java.util.Calendar ou les paramètres des méthodes Finder.
optional : @andromda.persistence.optional	Attribut d'une entité	true false	Définit l'annotation non-null relationship. Si elle est mise à false, une relation non nulle doit toujours exister.
Column_update @andromda.persistence.column.update	Attribut d'une Entity	true false	Spécifie si une colonne doit être incluse dans le sql update. true est la valeur par défaut.
generator_initial_value : @andromda.persistence.generator.initial.value	Attribut d'une Entity		Spécifie la séquence de départ pour la génération du ID pour le générateur de PK d'une entité.
order_by : @andromda.persistence.orderBy	Association End	Nom de colonne	Spécifie le(s) nom(s) de colonne(s) qui seront utilisées pour le tri des collections retournées ; avec ASC ou DESC optionnellement. Sans valeur, la clé primaire est assumée.
enumeration_type : @andromda.persistence.enumeration.type	Attribut d'une Entity	ordinal (default) string	Spécifie le type d'énumération du champ persistant.
generator_allocation_size : @andromda.persistence.generator.allocation.size	Attribut	Un nombre	Spécifie le nombre à incrémenter à l'allocation d'un id pour le générateur.
generator_source_name : @andromda.persistence.generator.source.name	Attribut d'une Entity	Nom de table	Spécifie le nom de table qui stocke les identifiants générés ou le nom de la séquence de base de données utilisée pour obtenir les identifiants.
column_nullable : @andromda.persistence.column.nullable	Attribut d'une Entity	true false	Spécifie si l'attribut peut être null.
fetch_type : @andromda.persistence.fetch.type	Attribut d'une Entity ou association End	lazy eager	Définit la valeur du Fetch type
column_scale : @andromda.persistence.column.scale	Attribut d'une Entity	Décimal	Indique l'échelle à utiliser pour une colonne décimale.
association_cache : @andromda.persistence.association.cache	Association End entre deux entités	none nonstrict_read_write read_only read_write transactional	Définit la stratégie du cache pour une association entre deux entités de type collection.
column_insert : @andromda.persistence.column.insert	Attribut d'une Entity	true false	Spécifie si une colonne doit être incluse dans le SQL INSERT.

Valeurs balisées du profil EJB3 Persistence::EJB pour les attributs

Valeur balisée	S'applique a	Valeurs possibles	Description
queryUseCache : @andromda.ejb.query.useCache	operations Query	true false	Active/désactive le cache pour les requêtes.
viewType : @andromda.ejb.viewType	Services ou méthodes d'une classe service	remote (default) local both	Définit le type d'accès au bean session (remote, local...).

Stéréotypes du profil EJB3 Persistence::EJB pour les attributs

Stéréotype	S'applique à	Description
FinderMethod	Méthodes d'une classe Entity	Associe cette méthode à une requête SQL pour retrouver cette entité.
PrePpersist/PostPersist (stéréotype)	Méthodes d'une classe Entity	Spécifie que l'opération associée est une méthode de callback pour un Bean entité et qu'elle sera invoquée après que l'entité aura été crée en base et qu'elle sera appliquée en cascade pour toutes les entités reliées.
PostLoad	Méthodes d'une classe Entity	Spécifie que l'opération associée est une méthode de callback pour un bean entité et qu'elle sera invoquée juste après le chargement des données à partir de la base et leur association avec l'entité.
PostUpdate	Méthodes d'une classe Entity	Spécifie que l'opération associée est une méthode de callback pour un bean entité et qu'elle sera invoquée après la mise à jour de la base.
CreateMethod	Méthodes d'une classe Entity ou Session	Utilisée pour annoter la méthode Create d'un EJB
PreUpdate	Méthodes d'une classe Entity	Spécifie que l'opération associée est une méthode de callback pour un bean entité et qu'elle sera invoquée avant la mise à jour de la base.
PostRemove/PreRemove	Méthodes d'une classe Entity	Spécifie que l'opération associée est une méthode de callback pour un bean entité, qu'elle sera invoquée après/avant la suppression de l'entité de la base et qu'elle sera appliquée en cascade pour toutes les entités reliées.
PostConstruct	Opérations des beans session et MDB	Spécifie que l'opération associée est une méthode de callback pour le bean entité et qu'elle sera invoquée après que l'entité aura été créée en base.

Index

A

Actuate 127
AJAX 21, 179
AndroMDA 291, 293, 334
annotations 178, 197
  de persistance 228
Apache
  Derby 130, 141
  Geronimo 201
  Struts 166
API
  JPA 225
  Query 248
ATF (Ajax Toolkit Framework) 11, 104

B

BEA (WebLogic_10) 202
bean
  entité 198, 228
    annotations de persistance 228
    interface Entity Manager 242
    invocation à partir d'un client Java et d'un client Web 252
    managé 247
    mapping des relations 235
    méthodes de callback 251
    méthodes de cycle de vie 243
    mise à jour 249
    persistance 246
    principales opérations disponibles 246
    relations 234
    requêtes JPQL 247
    suppression 249
    tables de jointure 241
  MDB 198
  message 213
  session 202
    avec état 209
    événements de callback 211
    façade 285
    sans état 203
    stateful 198
    stateless 198
BIRT (Business Intelligence and Reporting Tools) 13
  ajout de règles métier 13
  mise en forme 13
  récupération des données 13
  transformation des données 13

C

CDT (C/C++ Development Tools) 19
client
  léger 21
  riche
    exemple d'architecture 22
Cloudscape 131
CMF (Connectivity Management Framework) 128
Computer Associates 87
Compuware 87
  QaLoad 88
conception du modèle de données 127
conversations 332
couverture des tests 87
CVS (Concurrent Versions System) 55
  adaptateur Eclipse 58
  ajout d'utilisateurs avec pserver 65
  branches 58
  configuration du référentiel dans Eclipse 65
  conflits 72
  fonctionnalités 61
  installation et configuration de CVSNT 62
  intégration dans Eclipse 55
  mise en œuvre dans Eclipse 61
  référentiel d'équipe 59
  scénario de développement
    parallèle 60
    séquentiel 60
  synchronisation du référentiel 69
  validation 73
  versionnement des ressources d'un projet 74
CVSNT 59
  installation 62

D

Dali 10, 104, 215, 259
  configuration 260
  création
    d'un projet JPA et connexion à une source de données 263
    des entités persistantes du modèle et mapping vers la base 267
    des mappings de base 273
  définition des requêtes nommées de l'interface Query 284
  intégration des entités du modèle logique 285
  mapping
    de version 283
    des relations interentités 279

Many-to-One 282
One-to-Many 281
One-to-One 279
mise en œuvre
d'un bean client façade 285
de l'API JPA 262
scénarios de développement 260
Dali JPA Tools 11
Derby 14, 130, 141, 162, 289
design d'application Web 141
développement
collaboratif avec CVS 55
EJB3
avec Eclipse et Web Tools 195
avec JBoss et Web Tools 215
Web
Seam 165
Web Tools 101, 139
DI (Dependency Injection) 200
DLTK (Dynamic Languages Toolkit) 10
Draw2D 16
DTP (Data Tools Platform) 14, 127
architecture 128
CMF 128
connexion à la base Apache Derby 130
création d'un profil de connexion 133
génération du DDL 137
mise en œuvre 130
Model Base 128
script SQL de création du schéma de base de données 135
sous-projets 128
SQL Development Tools 129
vue SQL Results 136
DTP 1.5 128

E

Eclipse
architecture plugable 58
assistant de génération de getters/setters 153
configuration du référentiel CVS 65
console CVS 70
contrôle de version 70
décorateurs 69
développement en équipe 55
Europa 178
gestion de configuration intégrée 57
historique local 56
mise à jour des ressources CVS 72
mise en œuvre de CVS 61
perspective Référentiels CVS 74
référentiel partageable 57
SCMS 59
support de CVS 55
Web Tools 101, 168
Eclipse RCP 21
ajout de fonctionnalités 32
architecture 22
classes principales 29
démarrage de l'application 28
développement d'une application 25
exemples d'applications 24
mise en œuvre 25
points d'extension de plug-ins 43
Eclipse Studio 291
Eclipse 3.3 1, 90, 108, 114, 215
EclipseUML Studio 291
utilisation de svn 83
Web Tools 295
EclipseUML 291, 293, 334
Europa 293
EclipseUML Studio 291
Ecore 15
EJB 3.0 178
JPA 225
EJB3 178, 195
annotations 199
bean
entité 228
message 213
session
avec état 209
sans état 203
création de beans entité 297
descripteur de la persistance des données 199
différence fondamentale avec EJB2 198
étude de cas 291
événements de callback 211
fondamentaux 197
injection des dépendances 200
Java Persistence API 198
méthodes de cycle de vie d'une entité 243
mise en œuvre avec Web Tools et JBoss 215
principales différences avec EJB2 212
principales nouveautés 199
Seam (configuration avancée) 329
spécification 197
support des intercepteurs 201
synthèse des frameworks à succès 201
EMF (Eclipse Modeling Framework) 14, 89, 128
EMF 2.3 90
Entity Manager 242
ESB (Enterprise Service Bus) 178
Europa 1, 4, 90, 108, 114, 259
DTP 1.5 128
EclipseUML 293
EclipseUML Studio 291
EMF 15
import de projets avec copie 7
intégration des flux RSS 7
liens vers des ressources fichier 7
nettoyage de code 8
nouveautés V, 3, 345
restructuration de scripts 8
Seam 178
sous-projets 4, 9
support de la langue française 25
vue
Cheat Sheet 7
explorateur de projets 4
Filtres 6
historique 5
Nettoyage 8
Restructuration 8
TPTO 12
Europa 3.3 262

F

facelets Seam 182
facets 111, 245

G

GEF (Graphical Editor Framework) 16
 afficheur de diagrammes UML 16
GEF 3.3 90
Geronimo 2.0.1 201
gestion des entités 225
gestionnaire de versions Subversion 75
Glassfish 202
GMF (Graphical Modeling Framework) 16
 cinématique d'utilisation sous Eclipse 17
 éditeur UML 17
 générateur de modèles 17
 plug-in Visual Editor 18
Google Maps 21

H

Hibernate 104, 178, 197, 198
 gestion de la persistance 226
 JPA 183
Hibernate_3.0 201
HSQLDB 262
Hyades 12, 87
Hypersonic 130, 306
Hypersonic DB 262

I

IBM 10, 87, 127, 131
 alphaworks 18
 Notes 24
 WebSphere_7 202
 Workplace Client Technology Rich Edition 22
injection de dépendances 178, 200
Intel 87

J

J2EE 1.4 103
J2SE 5.0 227
JAAS (Java Authentication and Authorization Service) 105
JBI (Java Business Integration) 178
JBoss 103, 162
 AS 4.2.x et 5.x 181
 Dali 259
 déploiement de l'application 163
 EL (Expression Language) Enhancement 181
 HSQLDB 262
 jBPM 179
 mise en œuvre du développement EJB3 215
 Seam 165, 178, 329
JBoss Portal 179
JBoss 4.0 111
JBoss 4.0.5 110, 117
JBoss 4.2 144, 215
 HSQLDB 262
 support de JSF_1.2 168
JBoss 4.2.0
 GA 179
JBoss 4.2.1 295
JBoss 5 201
jBPM (JBoss Business Process Management) 331
JCM (J2EE Core Model) 105
JDBC (exécution d'une requête) 139
JDK5 215, 228
 annotations 197
JDO 2.0 201
JEE5 104, 178
 EJB3 198
 nouvelle API de persistance 201
JFace 23
JNDI 105
JPA (Java Persistence API) 183, 225
 architecture 226
 beans entité 228
 caractéristiques 227
 contexte de persistance 246
 création d'un projet et connexion à une source de données 263
 Dali 259
 gestionnaire d'entités 243
 mapping O/R 227
 mise en œuvre avec Dali 262
 TopLink 246
 TopLink Essentials 260
JPQL (Java Persistence Query Language) 248
JSE6 198
JSF (JavaServer Faces) 104, 166
 approche orientée composant 167
 configuration de la cinématique des pages 174
 éditeur de configuration Web Tools 175
 fondamentaux 167
 limitations 167
 mise en œuvre avec Web Tools 168
JSF Tools (JavaServer Faces Tools) 11
JSF 1.1 121
JSF 1.2 11, 168
JSP (JavaServer Pages) 166
JST (J2EE Standard Tools) 10, 103, 104
 architecture 105
 assistant de création de projet 108
 gestion des serveurs 109
 outillage J2EE standard 106
 serveurs d'applications supportés 110
 standards JCP supportés 104
JVMPI (Java Virtual Machine Profiler Interface) 90

L

loadRuner 88

M

mapping
 cardinalité 234
 de version 283
 directionnalité de la relation 234
 JPA avec Dali 259
 Many-to-Many 240
 Many-to-One 237, 282
 One-to-Many 238, 281
  bidirectionnel 239
 One-to-One 236, 279
  bidirectionnel 237
 tables de jointure 241
MDA (Model Driven Architecture) 291
Mercury (loadRuner) 88
Message Driven Beans 213
mesure des performances 87

métadonnées 199, 230
Model Base 128
modèle de données 127
MOF (MetaModel Facility) 15
MVC2 166
Mylyn 10
MySQL 5 131

O

ObjectWeb 10
Omondo 15, 195
  EclipseUML 291, 293
Oracle 11
  Dali 259
  TopLink 226
  TopLink Essentials 259
OSGI (Open Services Gateway Initiative) 23

P

patterns 139
  Commande 154
persistance des données 199
POA (programmation orientée aspect) 201
POJO (Plain Old Java Objects) 178, 198
profiling de code 87, 88

Q

QaLoad 88

R

Rational 16
RCP (Rich Client Platform) 21
référentiel d'équipe (CVS) 59
Reverse UML 309
RIA (Rich Internet Applications) 21
Ruby 10

S

SAP 87
Scapa Technologies 87
SCMS (Software Configuration Management System) 57, 59
Seam 165, 178, 228, 306
  AJAX 179
  annotations 178
    associées à JSF 191
    de gestion des contextes 343
    de validation des champs de saisie 193
  Ant_1.6 181
  caractéristiques 178
  composants fondamentaux de l'architecture 179
  configuration 334
    avec les composants POJO 183
    EJB3 avancée 329
  contexte 329
    conversationnel 332
  contextes 179
  conversation 329, 340
  développement fondé sur les POJO 178
  DI (Dependency Injection) 178
  Hibernate 181
  intégration dans l'architecture MVC 333
  intercepteurs 333
  mise en œuvre 181
  tiers présentation JSF 190
  utilisation des facelets 182
SOA Tool Platform 10
Spring 165, 197, 201
  injection de dépendances 200, 226
SQL Development Tools 129
Struts 165
  limitations 167
Subclipse 83
Subversion 75
Sun 166
  Glassfish 179, 202
  JPA 225
  JSF_1.2 168
SWT 16
Sybase 127

T

Tomcat 104, 179, 226
TopLink 198, 201, 246
  JPA 226
TopLink Essentials 259
TortoiseSVN 79
TPTP (Test & Performance Tools Platform) 11, 87
  analyse des informations de profiling 98
  architecture 88
  composants 89
  mise en œuvre 90
  Monitoring Tools Project 12, 87
  ouverture de la vue de profiling 94
  Platform Project 12, 87
  Test Tools Project 12, 87
  Tracing and Profiling Tools Project 12, 87
  vues de profiling 91

U

UML Visual Editor 18

V

Velocity 104
VisualAge for Java 56

W

Web Tools 101, 103, 295
  artefacts du projet et vues JEE 125
  assistant de publication 143
  complétion de code 146
  configuration
    d'un projet de développement Web 118
    de l'environnement d'exécution 115
    de la source de données du serveur 162
    des bibliothèques additionnelles spécifiques 117
  création
    de la page d'accueil 142
    de la servlet LoginServletTask 148
    des classes d'accès aux données 154
    des composants servlets et JSP 157
    des pages Login.jsp et LoginSuccess.jsp 145
  Dali 259
  design
    de l'application Web 141
    de la partie métier 153
  éditeur
    de configuration 174
    de JSP 147
    HTML intégré 142

étude de cas 291
exécution d'une requête JDBC 139
GMF 18
mise en œuvre 114
de JSF 168
du développement EJB3 215
patterns Commande et Singleton 154
propriétés du projet 123
Seam 165, 178
Singleton 154
sous-projets
JST 104
WST 111
structure et contenu du projet 124
Web Tools 2.0 90, 120, 262
Dali 259
WebLogic_10 202
WebSphere_7 202
Widgets Toolkit 22
Working Set 6
WST (Web Standard Tools) 10, 103, 111
assistant de création et de publication de services Web 113
monitoring TCP/IP 114
support
des facets projet 111
HTML, CSS et JavaScript 112
Web Services 113
WTP (Web Tools Platform) 10, 103
WTP 2.0 114

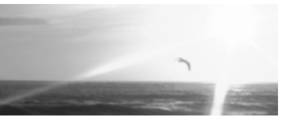
X

XDoclet 104, 105, 121, 217
XDoclet 1.2.3 122
XMI (XML Metadata Interchange) 305

Y

Yahoo! Mail 21
Yoxos 215

Développement JEE 5 avec Eclipse Europa



K. Djaafar

Résolument pratique, cet ouvrage montre comment utiliser de manière cohérente et efficace les différents outils de la suite Eclipse Europa, combinés à des frameworks tels que JBoss Seam, pour couvrir l'ensemble du cycle de développement d'une application Java EE 5 et tirer parti des innovations de la spécification EJB 3.

L'ouvrage met l'accent sur la conception du modèle de données avec Data Tools, le développement JSF avec Web Tools et Seam, le nouveau standard EJB 3.0, le mapping objet-relationnel avec Java Persistence API et le framework Dali, l'approche de développement piloté par le modèle avec EclipseUML et AndroMDA, les fonctionnalités avancées du framework Seam et le déploiement sur le serveur JBoss.

Cette approche est illustrée au travers de l'étude de cas WebStock, application e-commerce construite au fil de l'ouvrage et dont le code source est disponible sur www.editions-eyrolles.com. Le lecteur acquerra ainsi une parfaite maîtrise des outils et une solide méthodologie de développement reposant sur un ensemble de « best practices » en matière de conception d'applications EJB 3.

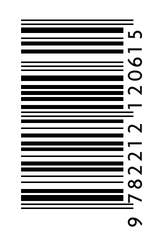
Karim Djaafar est consultant expert en technologies Java/JEE après plus d'une dizaine d'années d'expérience dans le domaine du développement logiciel et du conseil au sein de SSII. Aujourd'hui directeur technique de la SSII Jasmine Conseil, il assure des missions de conseil sur des projets Java/JEE mettant en œuvre des frameworks Open Source tels que Hibernate, Spring ou Seam. Il intervient aussi bien sur les phases de conception des architectures applicatives, que sur les problématiques de test, de déploiement sur les principaux serveurs d'applications JEE et d'optimisation des performances.

Au sommaire

Nouveautés d'Eclipse 3.3 (Europa) • Support du client riche avec Eclipse RPC • Développement collaboratif et gestion de configuration avec CVS et Subversion sous Eclipse • Profiling de code avec TPTP • Web Tools et ses sous-projets • Conception du modèle de données avec Data Tools • Développement JSP avec Web Tools et les design patterns • JSF et le framework Seam • Développement EJB 3 avec Web Tools et JBoss • Gestion de la persistance avec JPA • Mapping JPA avec Dali • Mise en œuvre de l'étude de cas WebStock : modélisation MDA et génération de code avec EclipseUML, déploiement sur JBoss • Configuration EJB 3 avancée avec Seam • Annexes : procédures d'installation (Eclipse Europa, JBoss…), annotations Seam, profil EJB 3 pour AndroMDA.

Sur le site www.editions-eyrolles.com
- Dialoguez avec l'auteur
- Téléchargez le code source de l'étude de cas WebStock
- Découvrez les mises à jour et compléments

Code éditeur : G12061
ISBN : 978-2-212-12061-5



Conception : Nord Compo

www.editions-eyrolles.com
Groupe Eyrolles | Diffusion Geodif | Distribution Sodis

EYROLLES