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.

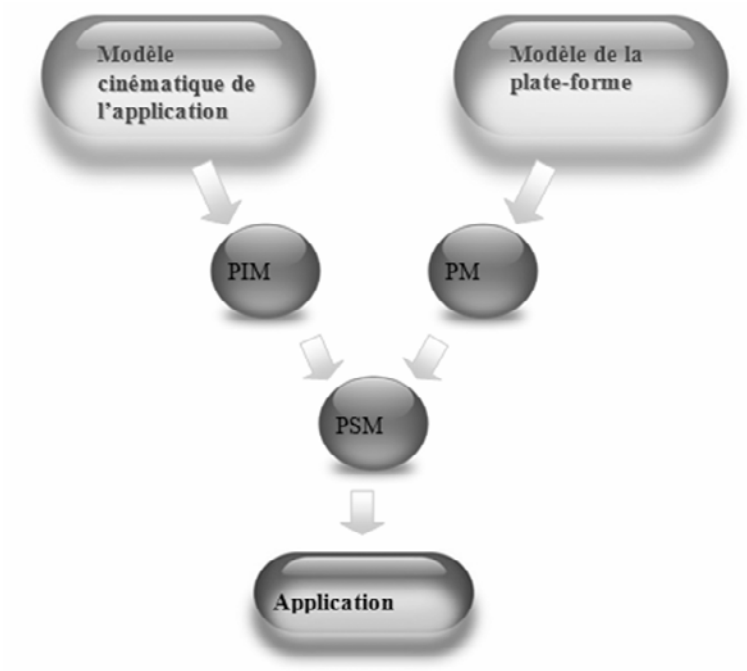
Cette approche de développement couvre l'ensemble du cycle du projet, de la conception au déploiement. Elle s'appuie sur une association des modèles créés, ou PIM (Platform Independant Model), qui représentent l'aspect métier, avec des modèles PM (Platform Model),

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

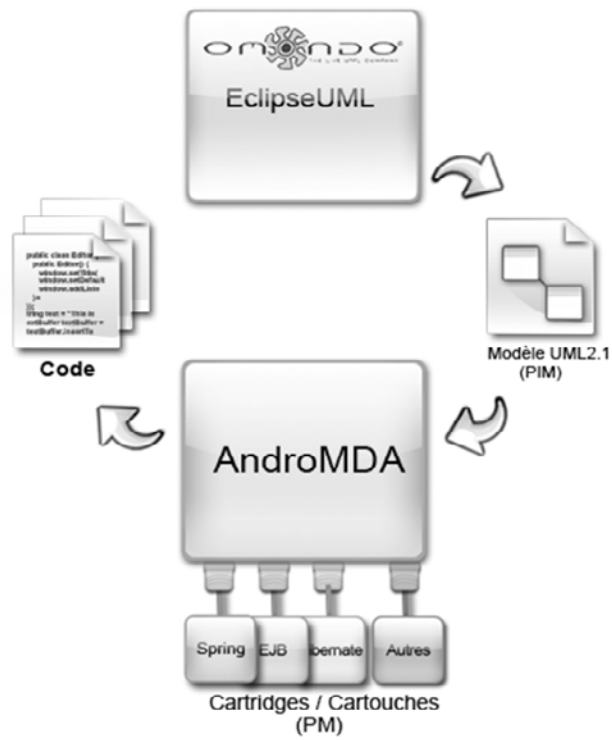
EclipseUML est un projet Eclipse développé sous la tutelle de la société Omondo, membre contributif de l'OMG (Object Management Group). L'outil est décliné en deux éditions, Free et Studio.

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.

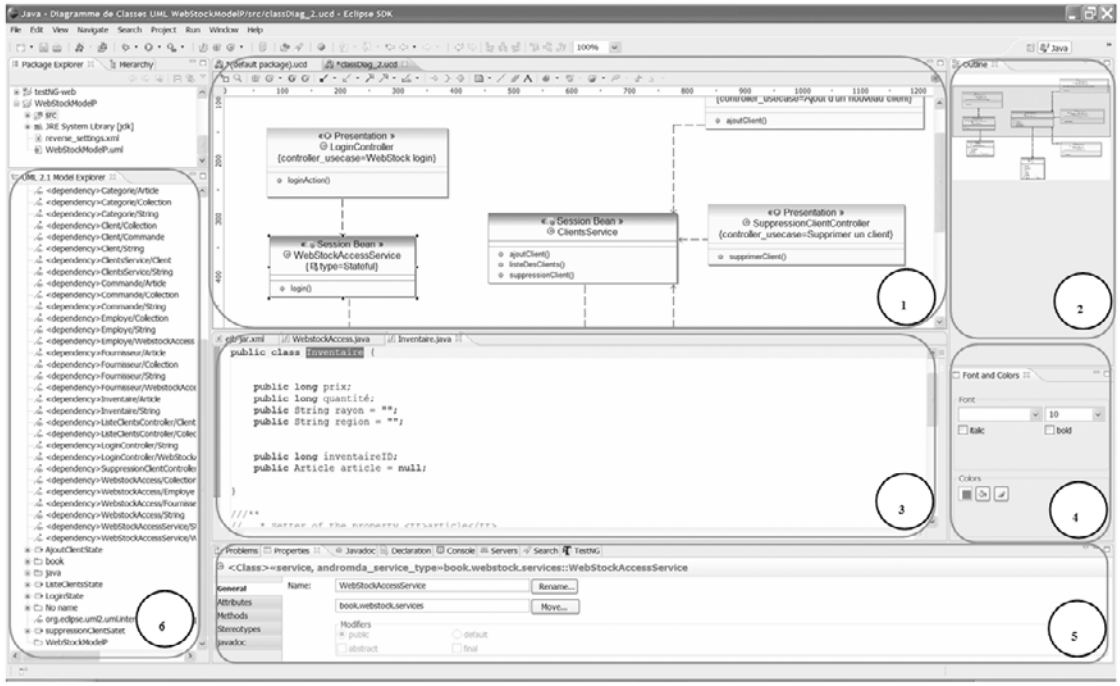


Figure 12.3

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.

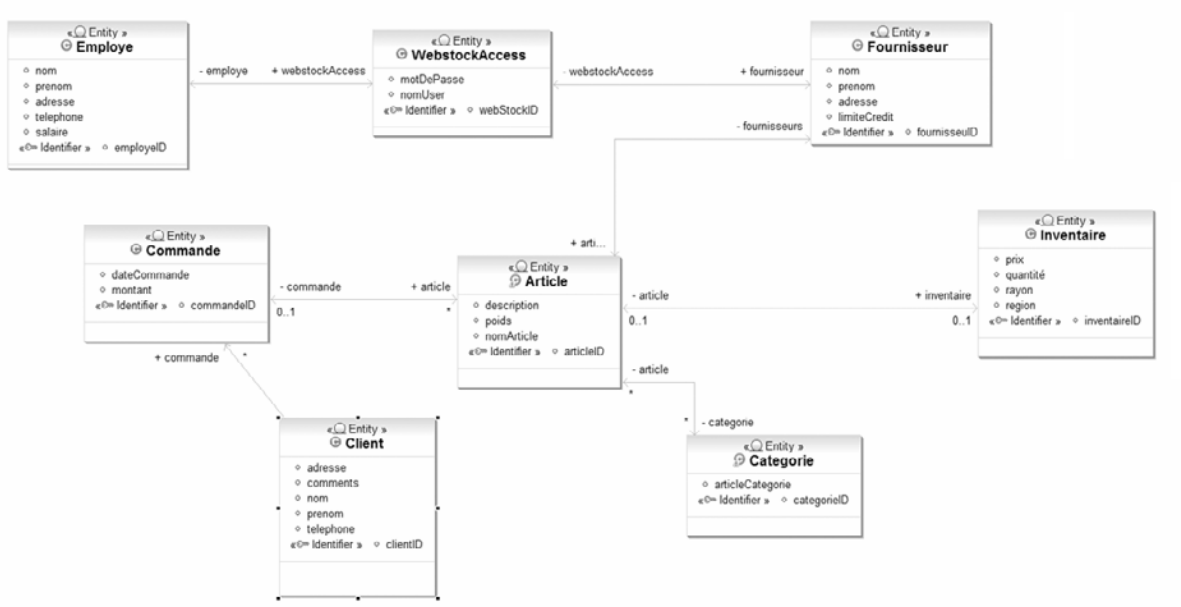


Figure 12.4

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.

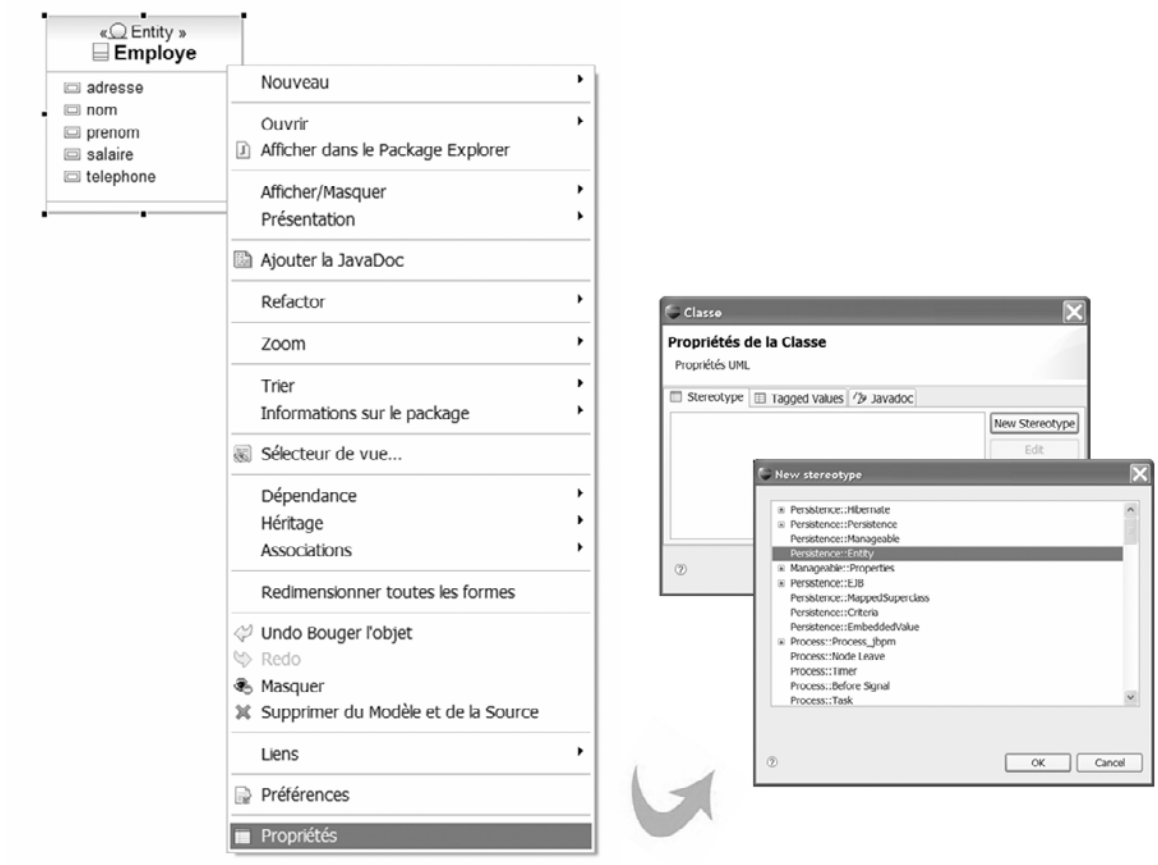


Figure 12.5

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.

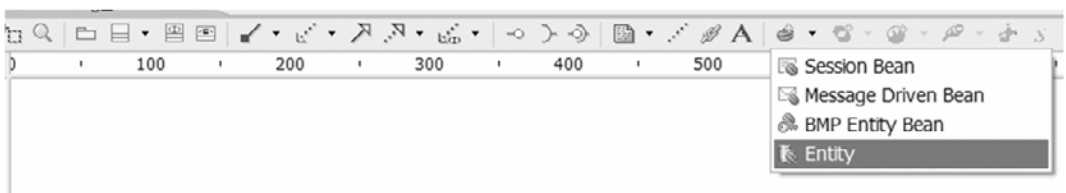


Figure 12.7

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.

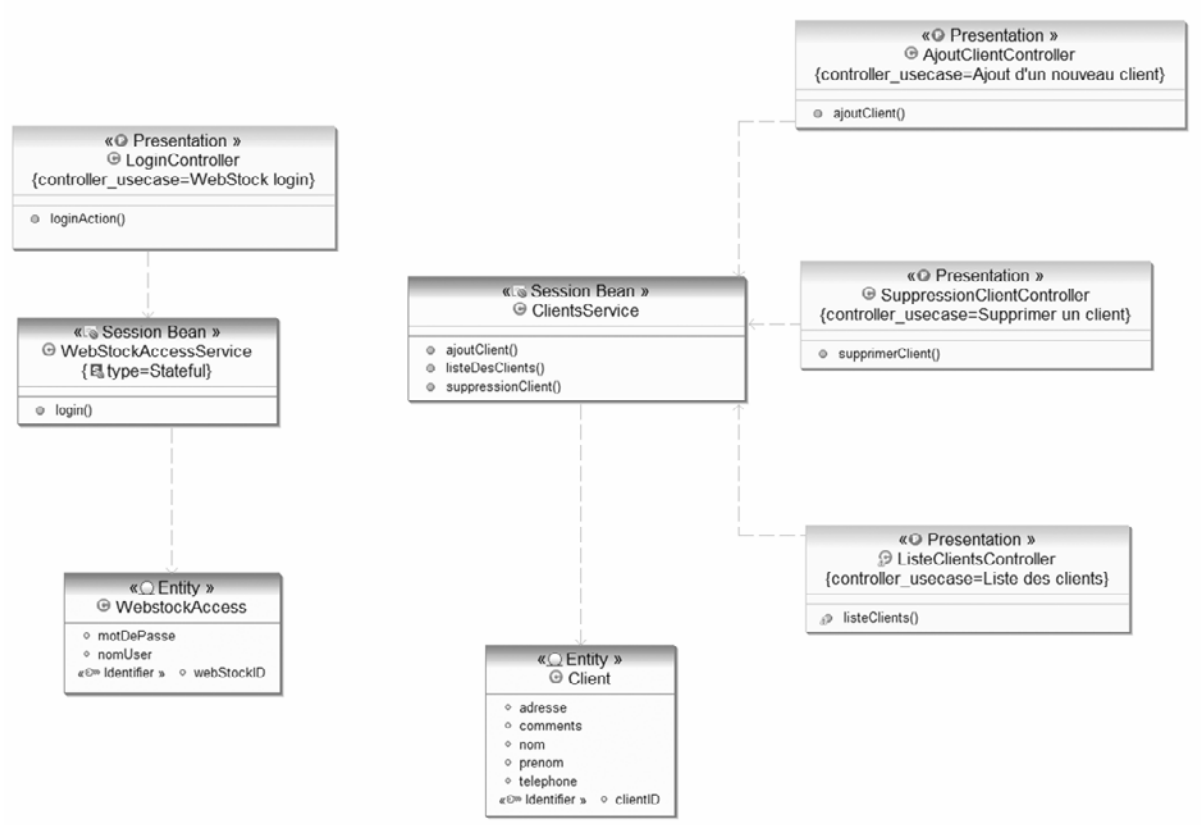


Figure 12.8

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.

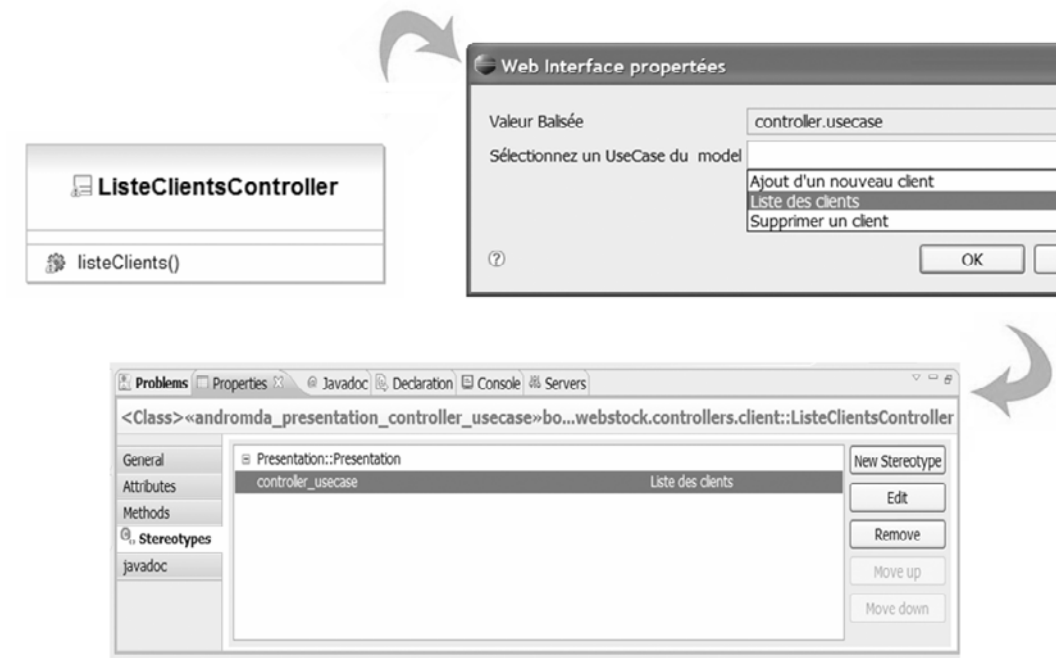


Figure 12.9

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.

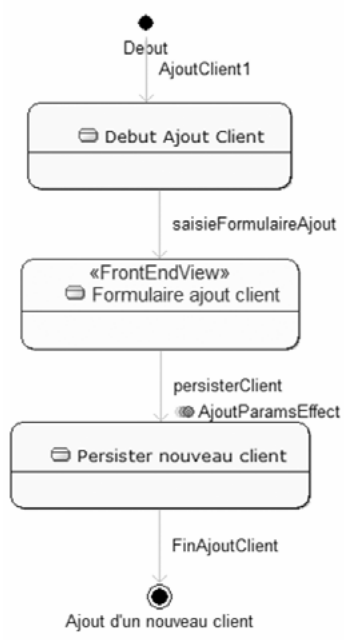
La figure illustre vos trois états intermédiaires : Début Ajout Client, Formulaire ajout client et Persister nouveau client. Le deuxième représente un formulaire de saisie des paramètres du nouveau client et est donc associé à la couche de présentation. Pour cette raison, lui est appliqué le stéréotype 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).

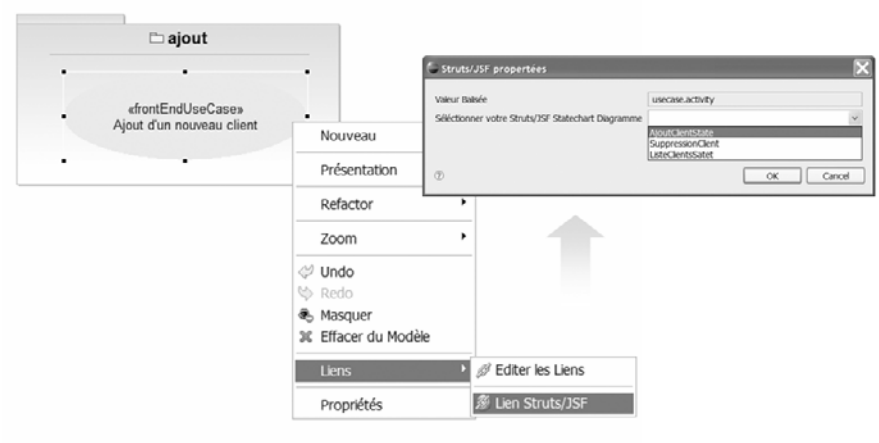


Figure 12.12

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.

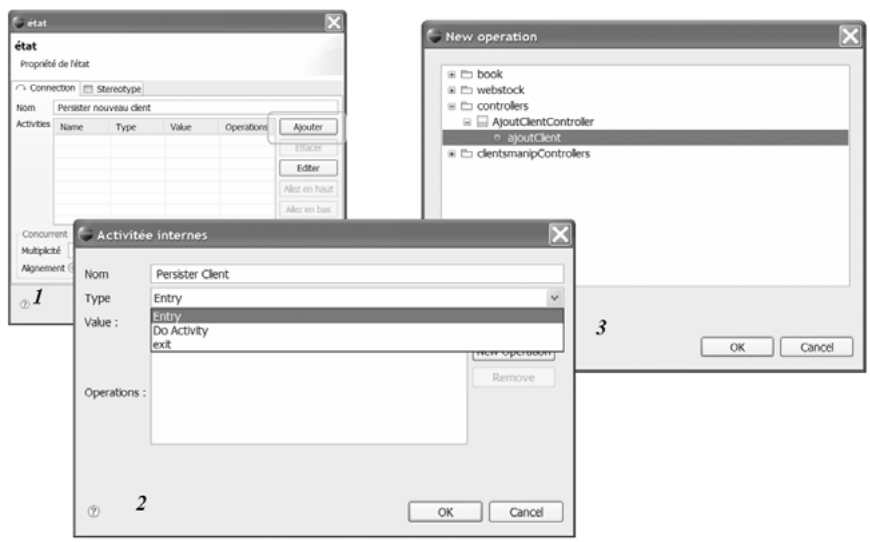
9. Dans la fenêtre de définition de la nouvelle activité, saisissez le nom de l'activité et son type, et spécifiez le moment de son exécution et l'opération à exécuter.

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

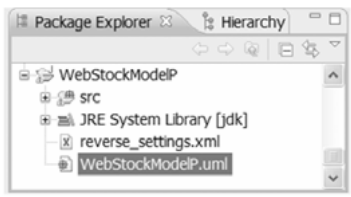
Export XMI du modèle

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.

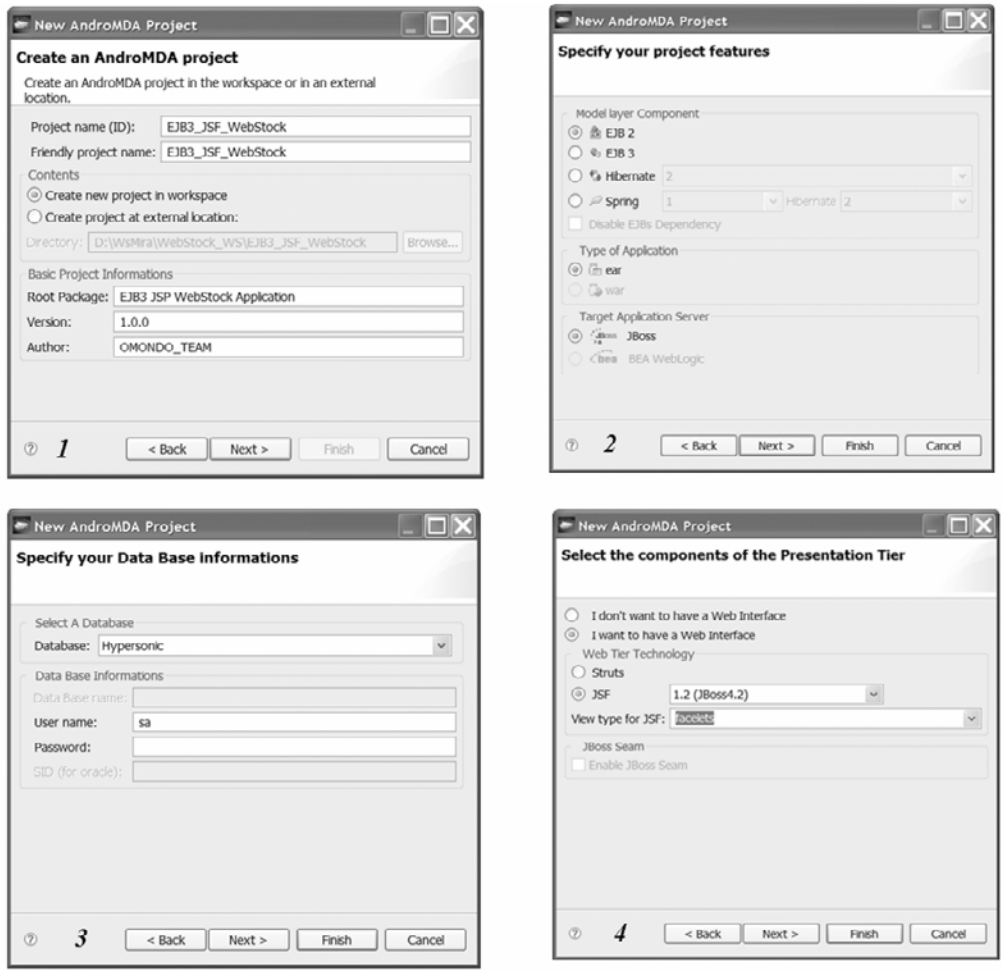
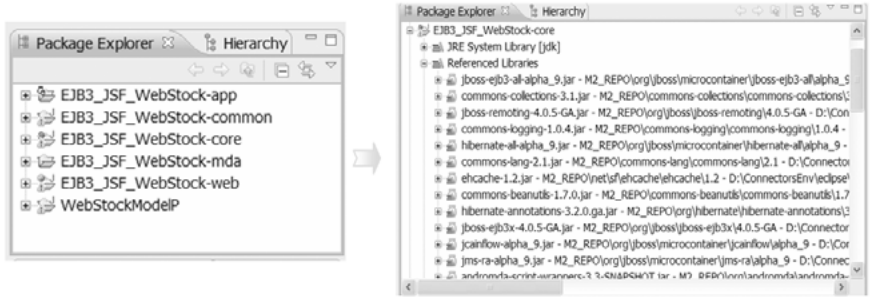


Figure 12.16

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.

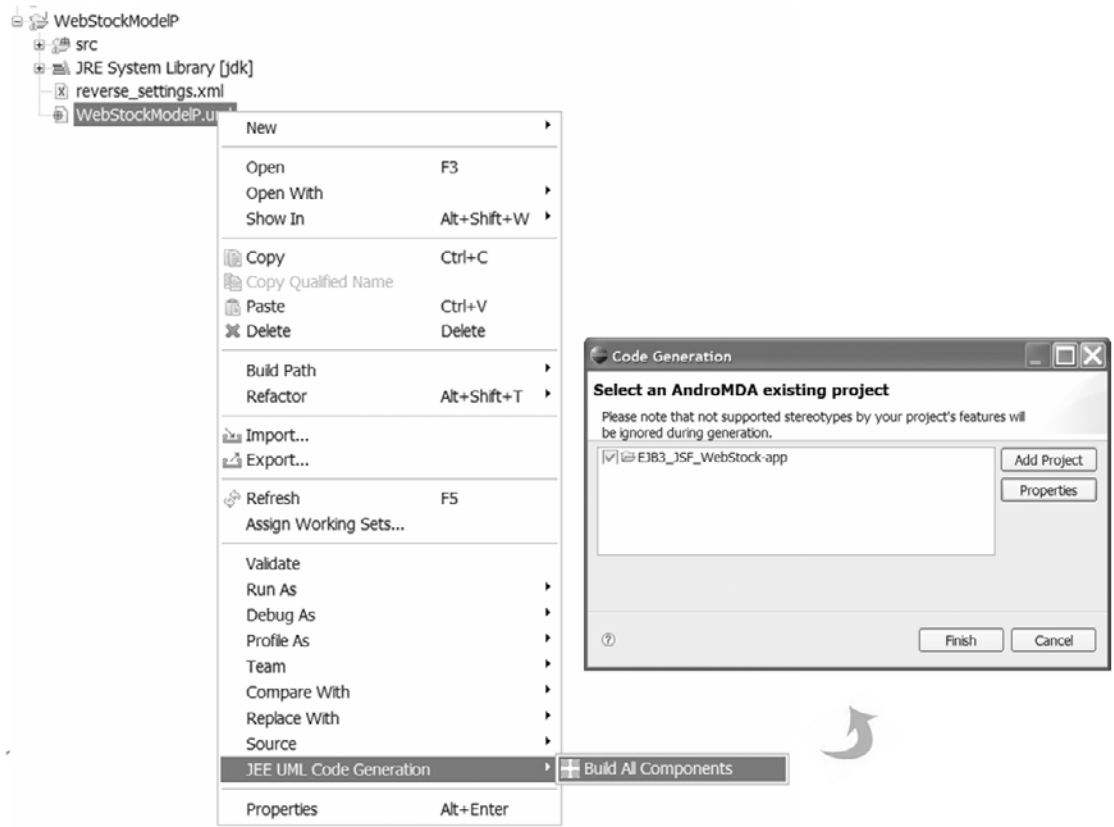


Figure 12.18

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.

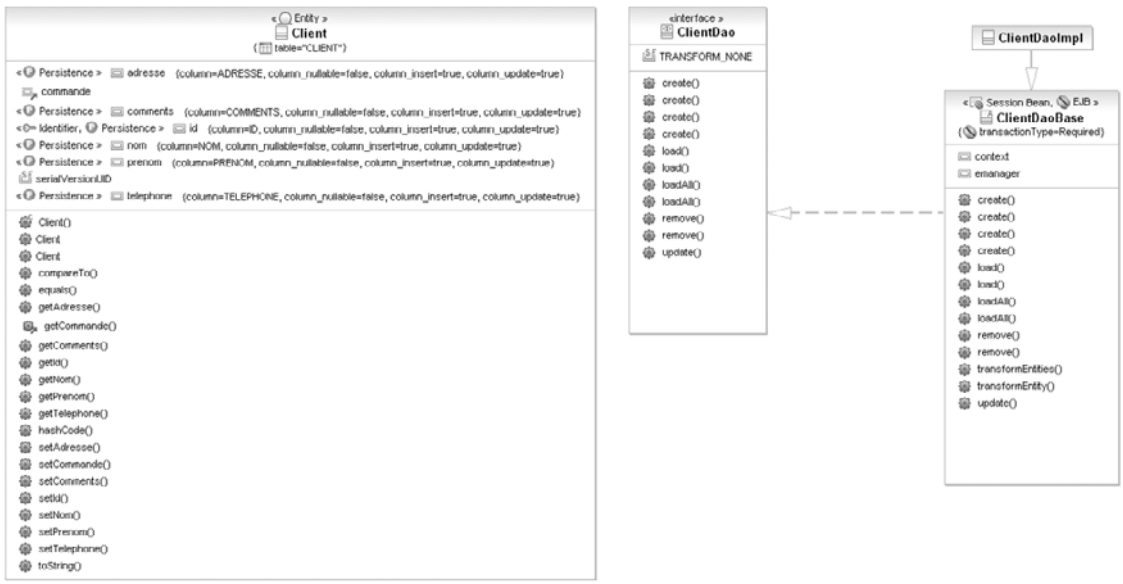


Figure 12.19

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 NamedQuery.

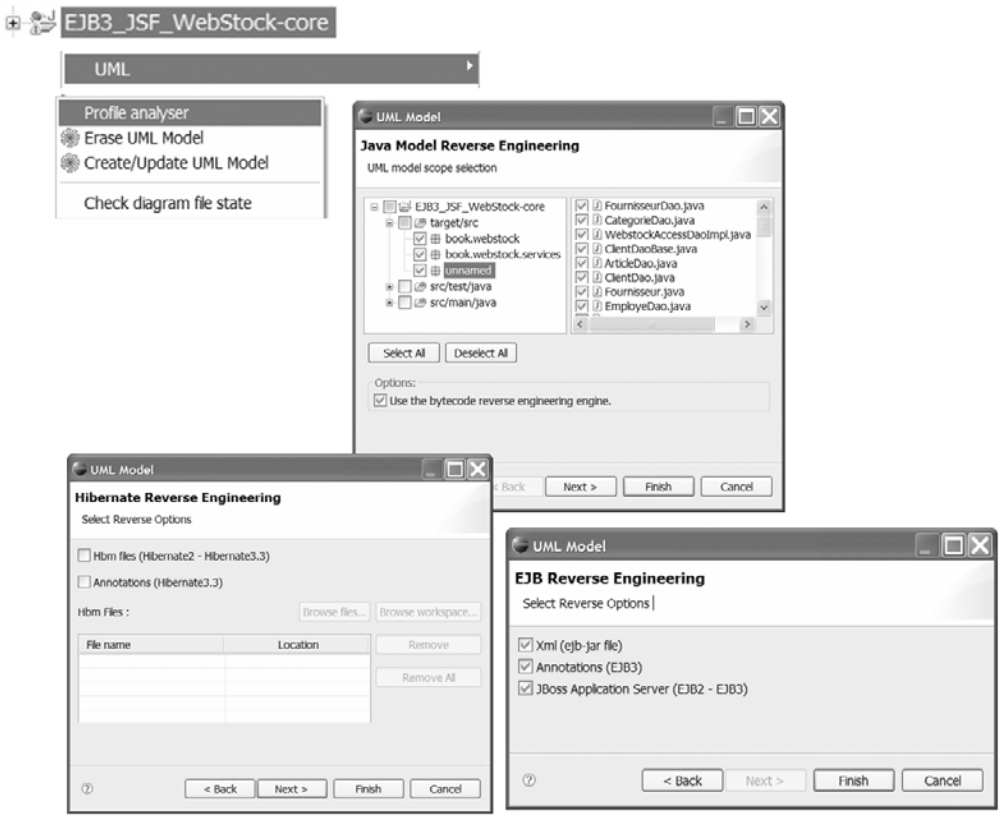


Figure 12.20

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
}}
```

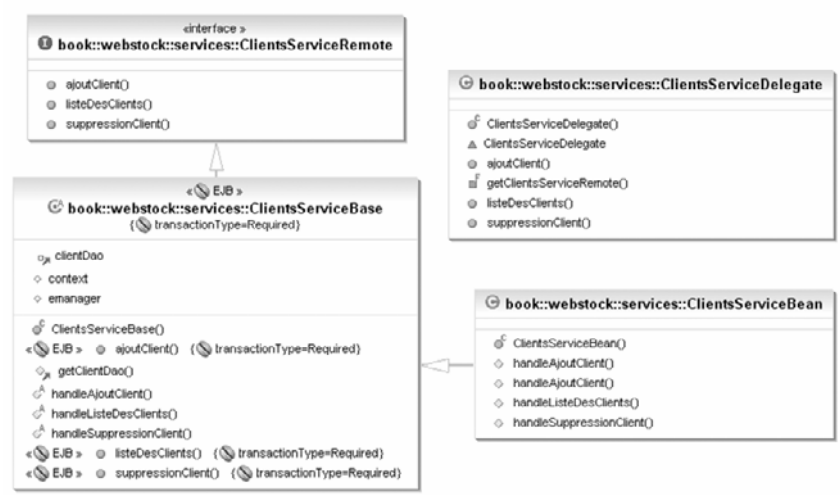
Il est inutile de reporter le code généré pour les classes `ArticleDaoImpl` et `ArticleDao`, qui sont respectivement l'interface pour personnaliser le bean session `ArticleDaoBase` et l'interface locale de ce dernier. Le lecteur souhaitant s'y plonger peut se reporter au code complet associé à ce chapitre, 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



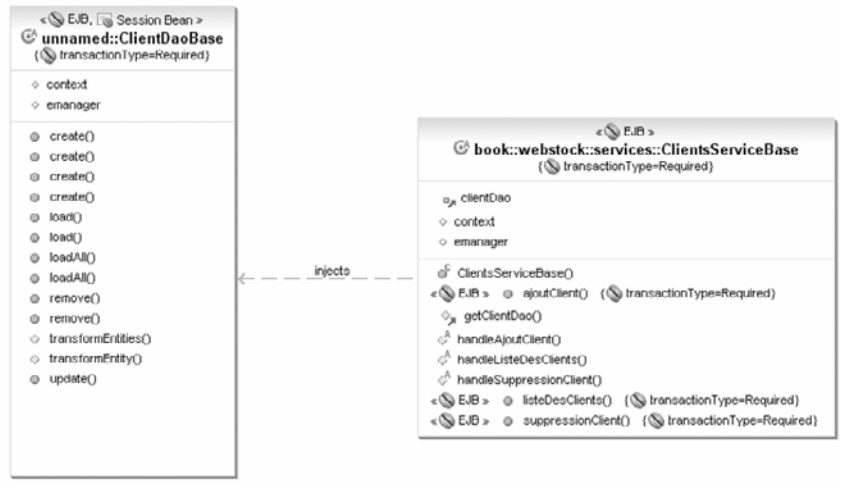
Les classes générées sont les suivantes :

- `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

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

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 `ajoutClientController`. Cette classe sera personnalisée *via* son implémentation `ajoutClientControllerImpl`.

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 AjoutClientControllerImpl.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` :

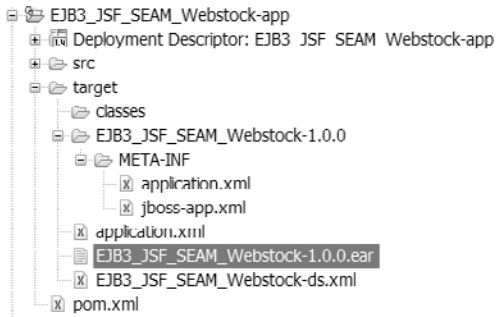
```
protected void handleAjoutClient(java.lang.String nom, java.lang.String prenom,
➥java.lang.String adresse, java.lang.String telephone, java.lang.String comments)
        throws java.lang.Exception
    {
    getClientDao().create(comments, telephone, adresse, prenom,   nom);  }
```

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.

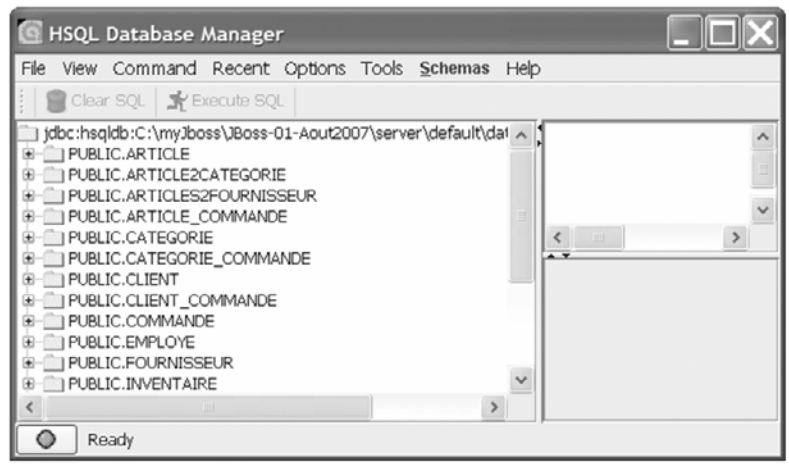
1. Copiez l'EAR et le ficher XML de configuration de la datasource sous les dossiers $JBOSS_HOME/server/default/deploy.
2. Après avoir démarré le serveur, vérifiez que le déploiement s'est achevé avec succès. Pour ce faire, voyez si les tables relatives à vos entités EJB3 ont été créées automatiquement, 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



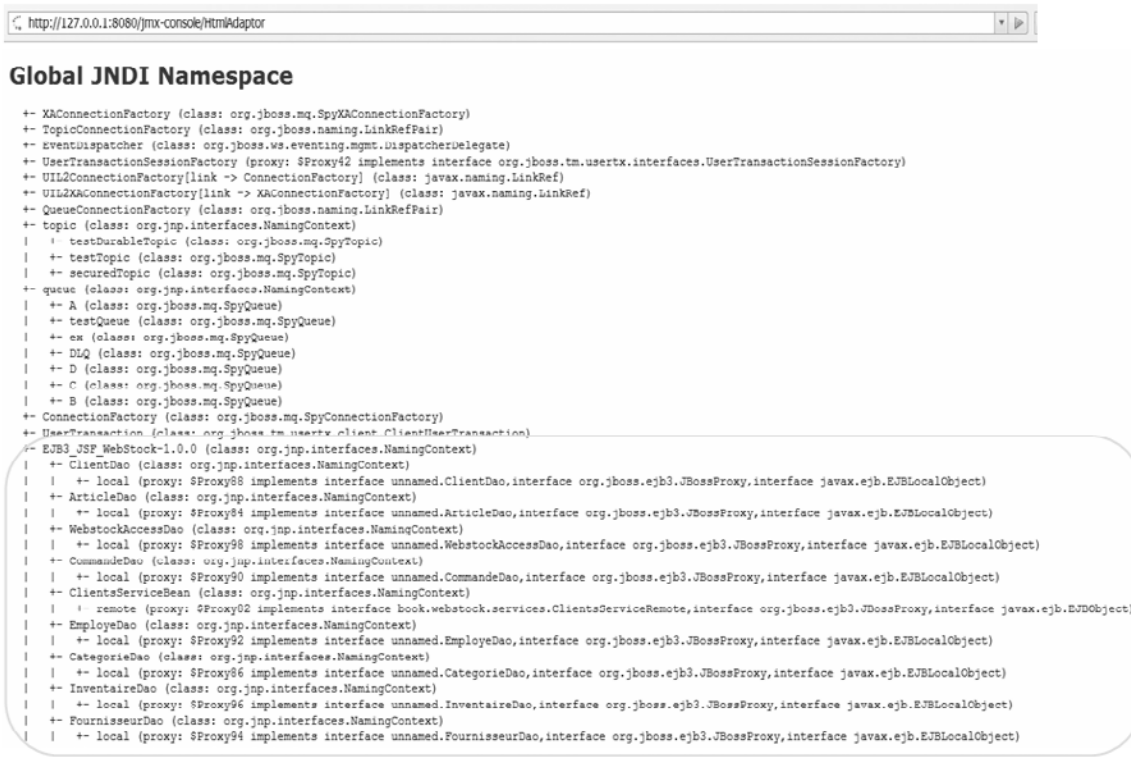


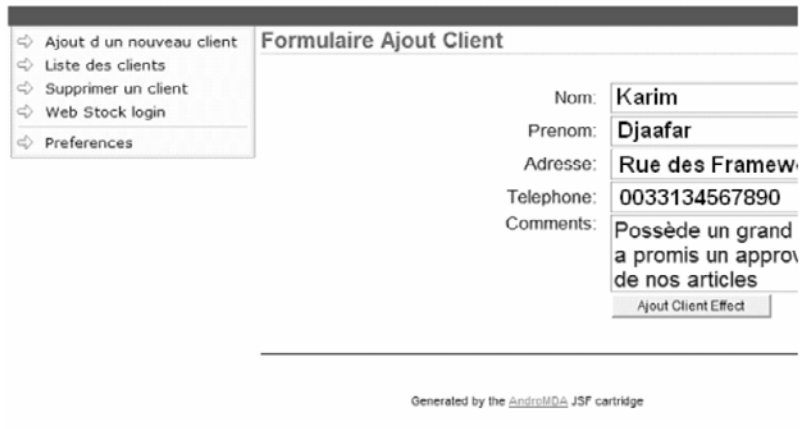
Figure 12.29

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.