Visual Modeler

Guide d'implémentation

Release 9.1

Visual Modeler

Guide d'implémentation

Release 9.1

Remarque

Avant d'utiliser le présent document et le produit associé, prenez connaissance des informations figurant à la section «Remarques», à la page 123.

Table des matières

Chapitre 1. Méthodologie d'implémentation 1

Chapitre 2. Implémentation de l'intégration de Visual Modeler 3

Chapitre 3. Étapes d'implémentation . . 5

Chapitre 4. Intégration entre Visual Modeler et Sterling Selling and Fulfillment Foundation 7
Intégration de Visual Modeler avec Sterling Selling and Fulfillment Foundation 7
Configuration des propriétés dans Visual Modeler . . 7
Configuration des règles dans Sterling Configurator 8

Chapitre 5. Présentation des applications Web J2EE 11

Chapitre 6. Architecture du système . . 17
Application Web Visual Modeler 17
Traitement des demandes 18
Remplacement des définitions MessageType . . 19
Éléments par défaut 20
Principales classes Java 20
Classes d'encapsuleur 20
ComergentContext 20
ComergentDispatcher 20
ComergentRequest 21
ComergentResponse 21
ComergentSession 21
Servlets 21
Classes Cde contrôleur 22
Classes DataBean 23
Classes ObjectManager et OMWrapper 23
Classe AppExecutionEnv 26
Classe AppsLookupHelper 27
Classe ComergentAppEnv 27
Classe Global 28
Interface GlobalCache 28
Classe LegacyFileUtils 28
Classe OutOfBandHelper 29
Classe Preferences 30
Transactions 31
Prise en charge des codes de recherche 31

Chapitre 7. Modularité de la plateforme 33
Présentation de la modularité de la plateforme Visual Modeler 33
Modules de plateforme 33
Modularité de la plateforme : Interfaces des modules 34
Description des modules de plateforme 34
Configuration du module de consignation 38
Consignateurs 38
Ajouteurs 39
Agencements 39
Moniteur de mémoire 40
Autorisation selon les types de messages 40
Gestionnaire d'objets 40
Réponse hors bande 40
Service Préférences 40
Bibliothèques de balises 41
Gestion des unités d'exécution 41
Convertisseur de messages XML 42
Service de messagerie XML 42
Services XML 43

Chapitre 8. Présentation des beans de données et des objets métier dans Visual Modeler 45
Beans de données dans Visual Modeler 45
Cycle de vie d'un bean de données 45
Définition d'un bean de données 46
Définition de la structure d'un objet de données . . 46
Création de beans de données et d'objets métier . . 47
DataContext 48
Beans de données de liste 51
Beans entity, application et présentation 51
Utilisation de procédures stockées 52
Méthodes de beans de données 53
Méthodes IData 53
Méthodes des interfaces IRd et IAcc 54
Restauration et persistance des données 54
Méthode DataBean restore() 55
Méthode DataBean persist() 56
Autres méthodes 56
Objets de données enfants 56
Extension d'objets de données 57
Exemple de bean de données 59
Création d'une définition d'objet de données . . 59
Arborescence DsElement 63
Éléments DsElement 63
`Métadonnées des éléments DsElement 64
Méthodes BusinessObject 65
Méthode BusinessObject restore() 65
Méthode BusinessObject persist() 65

Chapitre 9. Connexion à Visual Modeler 67
Présentation de la consignation dans Visual Modeler 67
Propriété système log4j.debug 67
Audit des modifications apportées aux objets de données 68

Chapitre 10. Modularité et interfaces générées 71

Chapitre 11. Modules dans Visual Modeler 73
Présentation des modules dans Visual Modeler . . 73
Interfaces des modules 73
Appel des interfaces 74

Chapitre 12. Interfaces générées. . . . 77

Chapitre 13. Classes logiques dans Visual Modeler 79
Implémentation des classes de logique 79
Concepts clés des classes logiques 79
Classes de logique d'application 79
Schéma XML 80
Service d'attribution de noms 80

Chapitre 14. Kit de développement de logiciels Visual Modeler 83
Utilisation du kit de développement de logiciels (SDK) pour personnaliser l'implémentation de Visual Modeler 83
Organisation des projets 83
Emplacements des fichiers et des répertoires de projet 83
Fichiers source Java. 83
Pages JSP 84
Fichiers de schéma 84

Chapitre 15. Localisation dans Visual Modeler 87
Présentation de Visual Modeler Localization . . . 87
Paramètres régionaux de présentation et de session 88
Pages JSP et fichiers de propriétés 89
Comportement de basculement 90
Comportement de basculement des regroupements de ressources 91
Comportement de basculement des pages JSP . . 91
Méthodes de récupération des paramètres régionaux 92
Utilisation de fichiers de propriétés dans le code . . 92
Données pour l'internationalisation 92
Modèles d'e-mail 93
Pages HTML 94
Images 94
JavaScript 94
Localisation dans Visual Modeler : Pages JSP . . . 95
Feuilles de style 96
Propriétés du système 97
Regroupements de ressources et formats. 97

Chapitre 16. Personnalisation des contrôles. 99
Modification d'un contrôle 99
Ajout d'un contrôle 100

Chapitre 17. Personnalisation des gestionnaires de fonction 101
Ajout d'une classe de gestionnaire de fonction . . 101

Chapitre 18. Exceptions 103
Hiérarchie ComergentException 103
Groupement de sous-systèmes. 103
Politique d'exception sous-système par sous-système 104
Chaînage des exceptions. 104
Lancement, levée et consignation d'exceptions . . 105
Situations qui justifient le lancement d'exceptions 105
Lancement d'exceptions d'exécution ou de compilation 105
Clauses catch et déclarations throws. 105
Exceptions de consignation 106
Affichage des exceptions 106

Chapitre 19. Travaux cron 109
Implémentation de travaux cron dans Visual Modeler 109
CronManager et CronScheduler 109
Interface CronJob 109
Création d'un travail cron dans Visual Modeler . . 111

Chapitre 20. Présentation des filtres 113

Chapitre 21. Filtres Visual Modeler 115

Chapitre 22. Gestion et affichage des zones restreintes 117

Index 119

Remarques 123

Chapitre 1. Méthodologie d'implémentation

La méthodologie d'implémentation de Visual Modeler comporte plusieurs phases qui permettent de s'assurer que l'implémentation peut être planifiée et contrôlée jusqu'à sa fin.

La table Méthodologie d'implémentation de Visual Modeler fournit un récapitulatif des différentes phases et décrit les activités effectuées pendant chaque phase. Chaque phase peut être contrôlée à l'aide d'un ensemble de documents standard.

Phase d'implémentation
: **Description**

Planification
: Planification de l'implémentation : définir un calendrier, poser des jalons et identifier les risques et les dépendances

Analyse
: Organisation et administration, définition des règles métier, de l'interface utilisateur, des protocoles de messagerie, des sources de données, planification des flux de commerce électronique, des besoins en formation, de la stratégie de déploiement, préparation de l'environnement, planification des opérations

Conception et configuration
: Installation, configuration, intégration, test d'unité et développement de formations

Test et déploiement
: Test de la configuration du serveur, de la communication d'entreprise à partenaire et de partenaire à entreprise ; passage aux systèmes de production, formation des distributeurs, mise en place de la documentation, support

Amélioration
: Actions continues d'amélioration, formation des partenaires et support

Chapitre 2. Implémentation de l'intégration de Visual Modeler

Visual Modeler est conçu pour intégrer des partenaires distributeurs dans un réseau de commerce électronique. Les organisations de ce réseau agissent comme des entreprises et des partenaires. Chaque organisation qui agit comme une entreprise installe un exemplaire du serveur d'entreprise pour transférer des informations à ses partenaires distributeurs en toute transparence.

Chaque revendeur ou distributeur peut collaborer avec plusieurs entreprises et son installation du serveur d'entreprise doit pouvoir recevoir et répondre à des messages provenant de différents serveurs d'entreprise.

Le tableau suivant résume les principales activités d'une implémentation de Visual Modeler :

Phase d'implémentation
Tâche

Planification
Analyse du projet

Analyse
- Analyse de la configuration
- Analyse de l'intégration
- Analyse de la configuration requise

Conception et configuration
- Préparation de l'environnement de conteneur de servlet
- Installation de Knowledgebase
- Configuration de Knowledgebase
- Configuration de Visual Modeler
- Définition des rôles et de la sécurité
- Authentification des administrateurs système
- Création du schéma XML
- Personnalisation des BizAPI, des classes de logique métier et des contrôleurs
- Personnalisation des pages JSP

Test et déploiement
- Intégration des produits
- Test de la configuration du serveur
- Test de la communication d'entreprise à partenaire
- Test de la communication de partenaire à entreprise
- Passage aux systèmes de production

Amélioration
Évaluation et amélioration

Chapitre 3. Étapes d'implémentation

Les principales tâches d'implémentation de Visual Modeler sont les suivantes :

- *Analyse du projet* : Fixez un planning pour le projet d'implémentation en définissant un calendrier d'exécution. Posez des jalons pour mesurer la progression de l'implémentation et identifiez les relations de dépendance et les risques susceptibles de retarder l'exécution de l'implémentation.
- *Analyse de la configuration* : Déterminez la configuration appropriée pour Visual Modeler (le nombre de machines utilisées et leur emplacement sur les réseaux internes par rapport aux pare-feu et aux serveurs proxy). Voir *High Availability and Load Balancing* dans *VM Installation Guide* pour plus d'informations sur une implémentation groupée.
- *Analyse de l'intégration* : Identifiez les points d'intégration avec les systèmes de commerce électronique existants.
- *Analyse de la configuration requise* : Vérifiez la configuration matérielle et logicielle requise pour vous assurer que les machines sont suffisamment puissantes pour prendre en charge le trafic prévu et les temps de réponse requis.
- *Installation de Visual Modeler* : Installez Visual Modeler sur la ou les machines appropriées. Pour plus d'informations, voir *Installation Overview* dans *VM Installation Guide*.
- *Configuration de Knowledgebase* :
  1. Installation de Knowledgebase : Installez le schéma Knowledgebase sur le serveur de base de données approprié.
  2. Configuration de Knowledgebase : Vérifiez la connectivité au serveur de base de données où Knowledgebase est installé et fournissez l'ensemble de vos informations relatives au commerce électronique. Vous devez notamment inclure les profils de vos partenaires, votre catalogue produits ainsi que des informations tarifaires.

     Pour plus d'informations, voir *VM Installation Guide*.
- *Configuration de Visual Modeler* : Modifiez les fichiers de configuration pour définir la configuration système dans votre environnement de production.
- *Définition des rôles et de la sécurité* : Définissez des groupes et des rôles et modifiez les fichiers de configuration et les scripts de liste de contrôle d'accès en conséquence. Les groupes et les rôles déterminent les privilèges de sécurité dont disposent les utilisateurs du serveur d'entreprise.
- *Création du schéma* : Créez le schéma d'objet métier pour fournir des informations sur les sources de données. La couche de données gère l'accès entre le serveur d'entreprise et les systèmes externes.
- *Personnalisation des classes de logique métier et de contrôleur* : Modifiez les classes de logique métier et de contrôleur pour prendre en charge votre propre logique métier. Dans certains cas, vous devez modifier les classes Java pour implémenter des processus métier spécifiques à votre organisation.
- *Personnalisation des pages JSP* : Modifiez les modèles pour les adapter à votre apparence générale, à vos besoins en termes de recherche et de pages statiques. Les pages JSP fournies par Visual Modeler sont utilisées pour afficher les pages de navigateur et peuvent être personnalisées en fonction des besoins de votre organisation.
- *Intégration des produits* : Importez les informations produit dans Knowledgebase ou effectuez une intégration avec les catalogues électroniques d'autres

fournisseurs (fonction "punch out"). Si votre implémentation doit prendre en charge la commande de produits à partir de sites non IBM, vous devez fournir un moyen d'intégration des données produit avec Visual Modeler.

- *Test de la configuration du serveur* : Avant de déployer Visual Modeler, testez rigoureusement le système. Nous fournissons un certain nombre de scripts permettant de tester les principaux composants fonctionnels.
- *Test de la communication d'entreprise à partenaire* : Envoyez des messages de test à partir du serveur d'entreprise vers d'autres serveurs d'entreprise.
- *Test de la communication de partenaire à entreprise* : Envoyez des messages de test à partir d'autres serveurs d'entreprise vers votre propre serveur d'entreprise.
- *Évaluation et amélioration* : Après le déploiement, vous devez prévoir un processus continu d'analyse de l'utilisation et des performances de Visual Modeler.

Chapitre 4. Intégration entre Visual Modeler et Sterling Selling and Fulfillment Foundation

Intégration de Visual Modeler avec Sterling Selling and Fulfillment Foundation

Certains produits complexes doivent être configurés avant de pouvoir être achetés par les clients. Dans d'autres cas, ces produits peuvent comporter des composants en option configurables par les clients en fonction de leurs besoins. Visual Modeler permet de créer des modèles qui définissent les options configurables d'un produit et d'associer des produits à ces modèles. IBM Sterling Configurator est un outil permettant d'afficher les produits configurables et les différentes options disponibles pour l'utilisateur final.

L'intégration entre Visual Modeler et IBM Sterling Selling and Fulfillment Foundation est nécessaire pour que l'échange d'informations soit possible. Cette intégration permet de s'assurer que les modèles définis dans Visual Modeler utilisent des informations produit appropriées, telles qu'elles sont indiquées dans Sterling Selling and Fulfillment Foundation. Les prix appliqués aux produits sont basés sur la liste de prix et la devise associées à l'utilisateur invité. Pour plus d'informations sur l'association de la liste de prix, voir *Business Center : Guide d'administration de la tarification*.

Pour intégrer Visual Modeler avec Sterling Selling and Fulfillment Foundation, certaines configurations doivent être effectuées dans l'application Visual Modeler et dans Applications Manager.

Configuration des propriétés dans Visual Modeler

Pourquoi et quand exécuter cette tâche

Vous devez configurer les valeurs de certaines propriétés dans Visual Modeler afin de lui permettre d'obtenir les informations produit appropriées à partir de Sterling Selling and Fulfillment Foundation.

Pour configurer les propriétés dans Visual Modeler, procédez comme suit :

Procédure

1. Entrez l'URL suivante dans votre navigateur :

   `http://<nom_hôte>:<port>/<racine_contexte>/en/US/enterpriseMgr/admin`

   nom_hôte correspond à l'adresse IP, port correspond au port d'écoute de la machine dans laquelle Visual Modeler est installé et racine_contexte correspond à la racine de contexte de l'application Visual Modeler hébergée.

   La page Connexion s'affiche.
2. Connectez-vous en tant qu'administrateur avec votre ID et votre mot de passe et cliquez sur **Connexion**.
3. Cliquez sur le lien hypertexte **Services système**. La page Propriétés du système s'affiche.
4. Cliquez sur le lien hypertexte **Exécution**. La page Propriétés pour Exécution s'affiche.

5. Définissez la propriété URL Sterling Order Fulfillment System sur `http://<nom_hôte>:<port>/smcfs/interop/InteropHttpServlet`. Cette URL fait référence au servlet Interop de Sterling Selling and Fulfillment Foundation.
6. Définissez la propriété URL Sterling Configurator sur :

   `http://<nom_hôte>:<port>/sbc/configurator/configure.action`

   nom_hôte correspond à l'adresse IP de la machine dans laquelle Sterling Selling and Fulfillment Foundation est installé et port correspond au port d'écoute de la machine dans laquelle Sterling Selling and Fulfillment Foundation est installé.
7. Définissez correctement les propriétés suivantes :
   - Nom d'utilisateur pour le système Sterling Fulfillment
   - Mot de passe pour le système Sterling Fulfillment

     Les valeurs de ces propriétés déterminent le nom d'utilisateur et le mot de passe à utiliser pour communiquer avec le serveur Sterling Selling and Fulfillment Foundation.

Configuration des règles dans Sterling Configurator

Pourquoi et quand exécuter cette tâche

Pour permettre à Sterling Configurator d'obtenir les informations de modèle des produits à partir de Visual Modeler, vous devez indiquer l'emplacement des modèles, des propriétés et des règles relatives aux modèles dans le gestionnaire d'application.

Pour configurer les règles de Sterling Configurator, procédez comme suit :

Procédure

1. Dans la page de connexion, connectez-vous en tant qu'administrateur avec votre ID et votre mot de passe et cliquez sur **Ouverture de session**. La page d'accueil de la console d'application s'affiche.
2. À partir de la barre de menus, cliquez sur **Configurations** > **Lancer Applications Manager**. Applications Manager est lancé dans une nouvelle fenêtre de navigateur.
3. À partir de la barre de menus d'Applications Manager, cliquez sur **Applications** > **Application Platform**. Le panneau latéral Règles d'application s'affiche.
4. Dans le panneau latéral Règles d'application, sélectionnez **Administration système** > **Configurateur d'articles**.
5. Indiquez les chemins d'accès aux emplacements où sont stockés les modèles, les fichiers de propriétés et les règles.

Résultats

Remarques :

- Tous les chemins indiqués comme référentiel de modèles dans Applications Manager sont partagés avec Sterling Selling and Fulfillment Foundation et Visual Modeler. Si Sterling Selling and Fulfillment Foundation et Visual Modeler résident sur des machines distinctes, les chemins doivent être montés sur une unité accessible à ces deux systèmes. Pour plus d'informations sur les référentiels de modèles, voir *Selling and Fulfillment Foundation: Application Platform Configuration Guide*.

- Dans IBM Sterling Business Center, un modèle peut être affecté à la définition d'article d'une offre groupée. Le nom du modèle est enregistré dans la définition d'article. Si vous modifiez un nom de modèle qui a été déjà enregistré dans la définition d'article, vous devez également modifier la définition de l'article pour faire référence au nouveau nom de modèle. Cette situation peut se produire lorsqu'un utilisateur modifie la définition du modèle dans Visual Modeler.

Chapitre 5. Présentation des applications Web J2EE

Cette section présente l'architecture Java 2 Plateform Enterprise Edition (J2EE) et explique comment vous pouvez l'utiliser pour déployer des applications Web. Si vous connaissez déjà cette architecture, vous pouvez ignorer cette section.

Architecture

Visual Modeler est conçu pour être conforme à l'architecture Java 2 Platform Enterprise Edition (J2EE), telle que définie dans la spécification *Java 2 Platform Enterprise Edition Specification, v 1.2* publiée par Sun Microsystems, Inc.

Visual Modeler est déployé comme une application Web comprenant un ensemble de classes Java associées à des fichiers de configuration, des modèles HTML et des pages JSP (JavaServer Pages). Il doit être installé dans un conteneur de servlet conforme au standard J2EE.

Applications Web

Une application Web J2EE est conçue pour être conforme à une spécification J2EE. Pour ajouter des composants Web à un conteneur de servlet J2EE, vous utilisez un package appelé fichier WAR (Web Application Archive). Un fichier WAR est un fichier compressé JAR (archive Java).

Outre les composants Web, il contient en général les ressources suivantes :

- Classes utilitaires côté serveur
- Ressources Web statiques (fichiers de configuration, pages HTML, fichiers image et son, etc.)
- Classes côté client (applets et classes utilitaires)

La structure de répertoires et de fichiers d'une application Web déployée en tant que fichier WAR doit répondre à des règles précises. Un fichier WAR a une structure de répertoire hiérarchique spécifique. Le répertoire supérieur d'un fichier WAR correspond au *répertoire document racine* de l'application. Ce répertoire document racine sert à stocker les pages JSP, les classes et les archives côté client et les ressources Web statiques. Il contient un sous-répertoire appelé **WEB-INF/**, où résident les fichiers et les répertoires suivants :

- **web.xml** : le descripteur de déploiement de l'application Web. Il décrit la structure de l'application Web.
- Fichiers descripteurs de la bibliothèque de balises.
- **classes/** : répertoire contenant les classes côté serveur, telles que le servlet, les classes utilitaires et les beans Java.
- **lib/** : répertoire contenant les archives JAR des bibliothèques (bibliothèques de balises et autres bibliothèques d'utilitaires appelées par les classes côté serveur).

Fichier web.xml

Le fichier **web.xml** doit figurer dans le répertoire **WEB-INF/** de chaque application Web déployée dans un conteneur de servlet. La structure de chaque fichier **web.xml** respecte une DTD publiée dans le cadre de la spécification J2EE.

Ce fichier **web.xml** indique la configuration générale de l'application Web en conformité avec le standard J2EE. Plus particulièrement :

- Il fournit les valeurs du paramètre d'initialisation pour l'application Web.
- Il peut déclarer et affecter des noms aux classes de servlet utilisées par l'application Web.
- Chaque classe de servlet est mappée à un ou plusieurs masques d'URL. Lorsque le conteneur de servlet reçoit une demande dont l'URL correspond à un masque défini dans le fichier **web.xml**, le servlet correspondant est utilisé pour traiter cette demande.
- Il fournit les valeurs du paramètre d'initialisation pour chaque servlet, si nécessaire.
- Il indique les informations de session (par exemple, le délai d'expiration).
- Il indique l'emplacement des bibliothèques de balises personnalisées utilisées par les pages JSP.

Pages JSP

Dans le passé, les applications Web basées sur Java généraient le code HTML renvoyé aux navigateurs Web des utilisateurs uniquement à l'aide de servlets. Des mécanismes de modèle ont été introduits au fil du temps, permettant aux développeurs Web de créer du contenu dynamique en générant le code HTML à l'aide de modèles. Plusieurs systèmes de masques d'URL sont disponibles, mais l'affichage du contenu dans l'architecture J2EE se fait à travers des pages JSP (JavaServer Pages).

Lorsqu'une application J2EE reçoit une demande du navigateur d'un utilisateur, elle la traite d'abord pour en extraire les paramètres et pour exécuter la logique métier lancée par cette demande. Une fois le traitement de la demande terminé, l'application Web doit répartir la demande vers une page JSP ; pour ce faire, elle utilise un *répartiteur de demande*. En général, le contexte de servlet appelle un répartiteur de demande en lui transmettant la page JSP cible. Le répartiteur de demande se charge ensuite de *transférer* les objets de demande et de réponse.

- Une page JSP est formée d'une combinaison de code HTML, de balises JSP et d'éléments de scriptage tels que des *scriptlets*.
- HTML : Une page JSP peut inclure n'importe quelle quantité de code HTML standard. Ce contenu est transmis directement à la page de navigateur sans aucune modification.
- Balises JSP : Ces balises remplissent le code HTML généré dynamiquement avec les valeurs calculées pendant la génération de la page. Les balises JSP standard sont <jsp:getProperty>, <jsp:include> et <jsp:forward>. Ces balises sont disponibles pour toutes les personnes qui créent des pages JSP. En outre, vous pouvez indiquer que votre application Web utilise une ou plusieurs bibliothèques de balises personnalisées. Chaque bibliothèque de balises personnalisées doit être déclarée dans le fichier **web.xml** correspondant à l'application Web ; la déclaration doit inclure l'URI de la bibliothèque de balises et l'emplacement du fichier TLD (descripteur de librairie de balises).

  Remarque : Dans Visual Modeler, l'utilisation des bibliothèques de balises est désormais obsolète. Pour optimiser les performances, il est recommandé d'utiliser des scriptlets. Vous pouvez continuer d'utiliser des balises JSP dans certaines applications existantes ou dans le cadre de certaines tâches d'intégration spécialisées.

- Éléments de scriptage : Vous pouvez insérer un bloc de code Java entre les balises HTML et JSP d'une page JSP en le plaçant entre la balise d'ouverture <% (ou <jsp:scriptlet>) et la balise de fermeture de scriptlet %> (ou </jsp:scriptlet>). Les scriptlets sont le plus souvent utilisés pour gérer le contrôle de flux complexe dans une page JSP. Notez que la plupart des éléments de scriptage JSP peuvent être appelés à l'aide d'une forme abrégée, comme indiqué dans le tableau suivant :

Forme abrégée	**Forme XML**
<%	<jsp:scriptlet>
<%=	<jsp:expression>
<%!	<jsp:declaration>
<%@	<jsp:directive>

  .

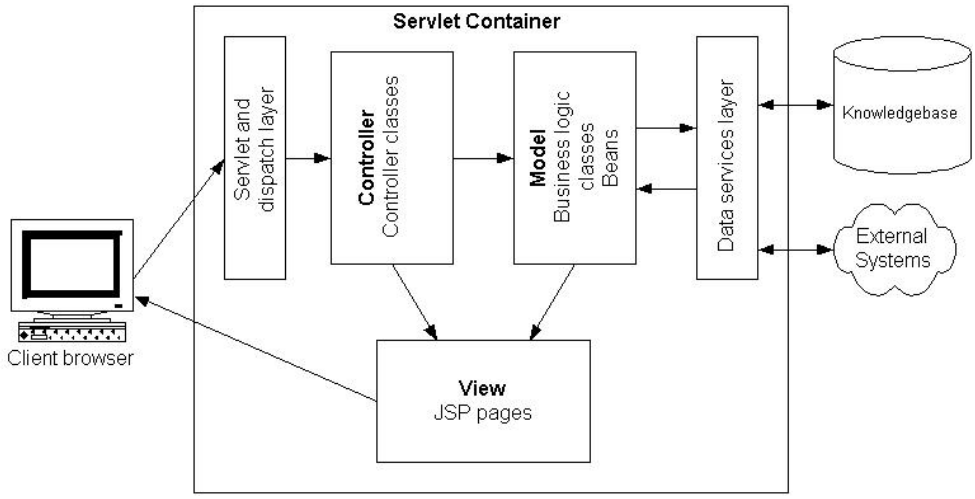
Les données sont transmises à la page JSP à l'aide de divers mécanismes, dont les plus importants sont les objets implicites et les beans.

- Objets implicites : Chaque page JSP fournit au développeur Web des objets permettant d'afficher des données sur la page HTML générée. Les principaux objets implicites sont les objets Page, Request, Session, Config et Application.
- Beans : La plupart des données générées par la logique métier de l'application sont transmises à la page JSP en ajoutant des beans Java à l'un des objets implicites répertoriés ci-dessus.

Architecture Model 2

Visual Modeler est conçu pour être conforme à l'architecture Model 2 de Sun. Dans cette architecture, trois composants fonctionnels désignés sous le nom MVC (Modèle-Vue-Contrôleur) divisent le fonctionnement de l'application Web en plusieurs composants logiques distincts.

La figure suivante montre l'architecture du modèle :



- *Modèle* : Ce composant gère les données et les objets métier utilisés par le système.

- *Vue* : Ce composant génère le contenu affiché pour l'utilisateur.
- *Contrôleur* : Ce composant contrôle le flux logique de l'application. Il détermine les actions effectuées sur le modèle et gère la communication entre les composants Modèle et Vue.

Contrôleurs

Dans l'architecture Model 2, les contrôleurs sont des classes Java destinées à gérer le traitement d'une demande interne et de dirigercelle-ci vers une page JSP appropriée. La structure de base d'un contrôleur dans Visual Modeler se présente comme suit :

```
public class GenericController extends Controller
{
   public void execute() throws Exception
   {
      //Répartir des logiques métier
      BizObjs resultBizObjects = calculate();
      //Générer les beans
      Vector beans = generateBeans(resultBizObjs);
      //Associer les beans à la demande
      attachBeans(beans);
      //Répartir la demande vers une page JSP
      String pageName = choosePageLogic();
      //Répartir la demande vers une page JSP
      Dispatcher rd = request.getDispatcher(pageName);
      rd.forward(request, response);
   }

   protected BizObjs calculate() throws Exception
   {
      //Effectuer une partie du traitement
      return resultBizObjs;
   }

   protected Vector generateBeans(BizObjs bizObjs)
   {
      //Créer des beans à partir des objets métier
      return beans;
   }

   protected void attachBeans(Vector beans)
   {
      Iterator it = beans.iterator();
      while (it.hasNext())
      {
         DataBean bean= (DataBean) it.next();
         request.setAttribute (beanName, bean);
      }
   }

   protected String choosePageLogic()
   {
      //Logique permettant de déterminer vers quelle page diriger la demande
      return pageString;
   }
}
```

Modèle

Dans l'architecture Model 2, les objets représentant les données du système sont stockés par le composant Modèle. Il est courant d'établir une distinction entre les objets métier et les beans utilisés dans les pages JSP.

Une fois que les objets métier sont créées et transformés par la logique métier, la classe de contrôleur transforme les objets métier en beans correspondants. Les beans sont alors transmis à la page JSP à des fins de présentation.

Afficher

L'interface utilisateur de l'application Web est transmise au navigateur à l'aide des pages JSP. Les données sont transmises à chaque page JSP sous forme de beans. Ces derniers sont des classes comportant des méthodes d'accès, qui permettent à la logique de la page JSP de récupérer des valeurs à l'aide de balises ayant le format suivant :

```
<%
DataBean dataBean = request.getAttribute("nomDeBean");
String stringProperty =
dataBean.getNamedProperty("nomDePropriété");
%>
```

Notez qu'il est possible d'utiliser une combinaison de scriptlets, de balises JSP standard et de balises personnalisées plus sophistiquées pour gérer la mise en page et l'affichage des données.

Pour aller plus loin

La documentation sur les applications Web, J2EE, les servlets et les pages JSP est très abondante. Si vous souhaitez aller plus loin, vous pouvez consulter les ouvrages suivants :

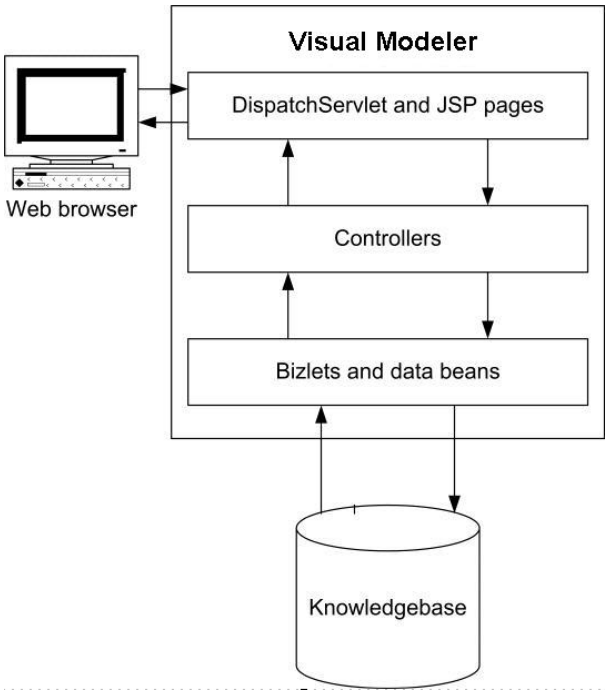
- Hall, *Core Servlets and JavaServer Pages*, Second Edition, Prentice Hall, 2003
- Hunter, *Java Servlet Programming*, Second Edition, O'Reilly, 2001
- Fields and Kolb, *Web Development with JavaServer Pages*, Second Edition, Manning, 2001

Chapitre 6. Architecture du système

Cette section décrit l'architecture de Visual Modeler et présente quelques classes Java principales utilisées par Visual Modeler et ses applications. Cette section part du principe que vous disposez déjà d'une bonne connaissance de l'architecture J2EE.

Cette section vous permet de modifier ou d'étendre des applications existantes ou d'écrire de nouvelles applications. Notez que toutes les parties de Visual Modeler ne respectent pas cette architecture.

La figure suivante présente l'architecture de Visual Modeler.



Application Web Visual Modeler

Lorsque vous installez Visual Modeler dans votre conteneur de servlet, il est installé en tant que fichier WAR (**Sterling.war**). Lors du déploiement du fichier WAR, celui-ci est décompressé dans un répertoire appelé **Sterling/**. Le sous-répertoire **WEB-INF/** contient le fichier **web.xml** de l'application.

Les principaux paramètres de configuration stockés dans ce fichier sont les suivants :

- La définition des servlets InitServlet et DispatchServlet :
  - Le servlet InitServlet est chargé lors du démarrage du conteneur de servlet. Il lit toutes les informations de configuration de Visual Modeler à l'aide de la valeur de l'élément propertiesFile. Par défaut, cette valeur correspond à **Comergent.xml**.

  - Le servlet DispatchServlet est le principal servlet utilisé pour traiter les demandes entrantes. La plupart des URL définies dans la section de mappage de servlet sont basées sur le servlet DispatchServlet.
- La section de mappage de servlet mappe la plupart des masques d'URL au servlet DispatchServlet. Notez que l'élément "/msg/*" est utilisé pour mapper les demandes au servlet MessagingServlet ; cela permet de s'assurer que les messages XML entrants sont traités par cette classe de servlet.
- L'élément de configuration de session définit le délai d'expiration de session sur 30 (minutes). La valeur de ce paramètre doit être examinée attentivement dans chaque implémentation de Visual Modeler. Gardez à l'esprit les considérations suivantes :
  - Les utilisateurs finals d'un système peuvent laisser leur navigateur sans surveillance lorsqu'ils s'éloignent de leur bureau. Si d'autres utilisateurs peuvent accéder au navigateur lorsqu'une session est encore active, ils peuvent accéder au système dans un dessein peu scrupuleux.
  - Pendant l'utilisation de Visual Modeler, les utilisateurs finals peuvent quitter le site et se connecter à d'autres systèmes externes pour acheter dans différents autres catalogues (fonction "punch out"). La valeur du délai d'expiration de session doit être suffisamment élevée pour permettre aux utilisateurs de quitter le site et de se connecter à d'autres systèmes externes pour acheter dans différents autres catalogues (fonction "punch out"), puis de revenir à Visual Modeler.
  - Chaque session utilise des ressources système. Plus la valeur du délai d'expiration de session est élevée, plus l'utilisation de la mémoire du système est importante.
- L'emplacement du fichier descripteur de bibliothèque de balises Comergent est fourni.

Traitement des demandes

À la réception d'une demande provenant d'un navigateur de l'utilisateur, Visual Modeler doit déterminer comment traiter cette demande et comment afficher le résultat pour l'utilisateur. Il utilise à cet effet les fichiers de configuration **MessageTypes.xml**. Ces fichiers déterminent le mappage entre une demande et la logique qui traite les classes et les pages JSP utilisées.

1. Lorsqu'une demande est reçue, Visual Modeler identifie le type de message et appelle le contrôleur approprié.
2. Des logiques métier supplémentaires peuvent être appelées à l'aide d'une classe de logique métier ou d'une bizAPI.
3. Le contrôleur dirige la demande vers la page JSP indiquée pour afficher les résultats de la demande dans le navigateur de l'utilisateur.

L'élément messageTypeFilename de l'élément GeneralObjectFactory dans le fichier **Comergent.xml** contient une liste délimitée par des virgules de fichiers **MessageTypes.xml** utilisés pour spécifier les types de message. Chaque fichier **MessageTypes.xml** déclare une liste de types de messages organisés par groupe de messages.

Chaque demande indique le type de message dans le paramètre cmd. Par exemple, si l'URL a la forme suivante :

```
../Sterling/catalog/matrix?cmd=search
```

le nom du type de message est "search".

Chaque type de message est identifié par l'attribut Name de son élément MessageType. L'attribut Name identifie le type de message demandé lorsqu'un utilisateur clique sur une URL.

Remarque : Chaque groupe de messages et chaque type de message doivent posséder des noms uniques. Vérifiez la liste de fichiers **MessageTypes.xml** pour vous assurer que le même nom n'est pas défini pour plusieurs groupes de messages ou plusieurs types de message. Pour une exception à cette règle, voir «Remplacement des définitions MessageType». Vous pouvez trier les types de messages par ordre alphabétique du nom dans les groupes de messages afin d'identifier rapidement les éventuels noms de types de message en double.

Les éléments MessageType ont un ou plusieurs éléments enfants parmi les éléments suivants :

- BizletMapping : Cet élément associe une classe Bizlet et une méthode de cette classe pour traiter des messages.
- ControllerMapping : Associe le contrôleur à utiliser pour traiter la demande. Pour le traitement des messages, vous pouvez appeler la classe BizRouter.
- JSPMapping : Associe la page JSP à utiliser pour afficher le résultat de la demande.

Un élément MessageType peut contenir une combinaison de ces trois éléments.

- Si l'élément ControllerMapping n'est pas spécifié, la classe ForwardController est utilisée par défaut. Cette classe dirige la demande vers la page JSP indiquée par l'élément JSPMapping. Si l'élément JSPMapping n'est pas trouvé ou si la page JSP indiquée est manquante, une page d'erreur s'affiche.
- Si un contrôleur personnalisé est indiqué, il peut traiter lui-même la demande (voir «Classes Cde contrôleur», à la page 22) ou il peut appeler une classe de logique métier à l'aide de la méthode *runAppJob()* de la classe AppExecutionEnv (voir «Classe AppExecutionEnv», à la page 26).
- Si l'élément JSPMapping n'est pas spécifié, la classe de logique métier ou le contrôleur doivent indiquer la page JSP à utiliser.

Chaque demande ou message est validé par rapport au système d'autorisations d'utilisation afin de vérifier si l'utilisateur dispose des droits nécessaires pour exécuter ce type de message. Tous les types de message ne peuvent pas être exécutés par tous les utilisateurs.

Remplacement des définitions MessageType

L'élément MessageType comporte l'attribut facultatif IsOverlay. Si cet attribut est défini sur true, la définition de l'élément MessageType remplace la définition précédente de ce type de message fournie dans les fichiers MessageTypes.xml antérieurs indiqués dans l'élément messageTypeFilename.

Si deux ou plusieurs définitions sont disponibles pour le même type de message sans que l'attribut isOverlay soit indiqué, une erreur d'initialisation s'affiche et la première définition du type de message est utilisé.

Notez que l'attribut IsOverlay ne modifie pas l'emplacement de l'élément MessageType, qui est déterminé par le groupe de messages auquel appartient la première définition ou par l'élément MessageTypeRef qui référence le type de message.

Par exemple, pour remplacer la définition du type de message adirectLogin, vous pouvez définir l'élément suivant :

```
<MessageType Name="adirectLogin" IsOverlay="true">
<ControllerMapping>
com.comergent.apps.common.controller.MyLoginController
</ControllerMapping>
<JSPMapping>../common/adirectPageLoader.jsp</JSPMapping>
</MessageType>
```

L'attribut IsOverlay peut également être utilisé pour les déclarations MessageGroup afin de remplacer la définition d'un groupe de messages, mais cette utilisation est déconseillée.

Éléments par défaut

Pour chaque groupe de messages, vous pouvez indiquer les éléments par défaut BizletMapping, ControllerMapping et JSPMapping. Ces éléments sont utilisés lorsqu'aucun mappage n'est indiqué pour un type de message appartenant au groupe de messages.

En général, si aucun mappage par défaut n'est indiqué dans un groupe de messages, le système cherche un mappage par défaut dans le groupe de messages parent du groupe de messages en cours. Si aucun mappage n'est trouvé dans l'ensemble de l'arborescence de groupes de messages, les valeurs indiquées dans le groupe de messages MessageGroupDefaults sont utilisées.

Principales classes Java

D'un point de vue schématique, toutes les applications Visual Modeler ont une structure identique. Elles sont composées de contrôleurs, d'objets métier, de bizlets et de pages JSP.

Classes d'encapsuleur

Plusieurs classes standard utilisées dans les applications Web J2EE sont encapsulées dans des classes d'encapsuleur afin de permettre une gestion détaillée de toutes les particularités des conteneurs de servlet pris en charge.

ComergentContext

Cette classe permet d'encapsuler le contexte du conteneur de servlet. Vous pouvez l'utiliser pour récupérer l'objet Env afin d'obtenir des informations sur l'environnement. Notez que tous les attributs de contexte définis doivent être sérialisables. Une exception est lancée si vous tentez de définir un attribut non sérialisable.

Elle fournit la méthode *getResourceAsStream()*, qui permet un accès en lecture seule à un fichier comme s'il était un flux. Pour un accès en écriture, vous devez utiliser la méthode *adjustFileName()* de la classe LegacyFileUtils.

ComergentDispatcher

Cette classe est un encapsuleur léger de la classe standard RequestDispatcher. Elle fournit les méthodes *forward()* et *include()*.

ComergentRequest

Cette classe encapsule la classe standard HttpRequest et fournit des méthodes d'auxiliaire pour effectuer une analyse syntaxique des demandes et des messages entrants.

ComergentResponse

Cette classe encapsule la classe standard HttpResponse. Elle fournit la méthode *localRedirect()* permettant de transmettre une demande avec un nouveau type de message. Par exemple, si vous souhaitez que la demande soit traitée par un contrôleur et que le résultat soit transmis à un autre contrôleur, vous pouvez appeler la méthode suivante :

```
response.localRedirect(request, "messageType");
```

L'appel de cette méthode a pour effet de soumettre la demande à DispatchServlet comme si elle avait été reçue en tant que demande HTTP.

ComergentSession

Cette classe encapsule la classe standard HttpSession. Lors de la première connexion d'un utilisateur, un bean de données Utilisateur est créé et ajouté à l'objet ComergentSession. Vous pouvez accéder aux informations utilisateur à l'aide de la méthode ComergentSession *getUser()*.

Par exemple :

```
session.getUser().getUserKey()
```

renvoie la clé de l'utilisateur en cours ; et

```
session.getUser().getPartnerKey()
renvoie la clé du partenaire auquel l'utilisateur appartient.
```

L'objet ComergentSession est utilisé pour stocker des informations qui doivent être persistantes pour plusieurs demandes provenant de la session d'un utilisateur. Utilisez la méthode *setAttribute(String s, Object o)* pour définir un objet dans la session session et la méthode *getSession(String s)* pour récupérer cet objet. Les objets stockés dans la session doivent implémenter l'interface sérialisable ; tous les beans de données générés implémentent cette interface et peuvent ainsi être stockés dans la session.

La classe ComergentSession fournit également la méthode *logout()*, qui permet d'invalider immédiatement la session de conteneur de servlet.

Servlets

Les principaux servlets utilisés sont les suivants :

- InitServlet : Ce servlet est chargé lors du démarrage du conteneur de servlet. Sa méthode *init(ServletConfig config)* initialise la classe ComergentAppEnv.
- DispatchServlet : Ce servlet est utilisé pour répartir la plupart des demandes traitées par Visual Modeler. Sa principale méthode est la suivante :

  ```
  void dispatch(HttpServletRequest request, HttpServletResponse response)
  Cette méthode crée un contrôleur pour gérer la demande à l'aide des lignes suivantes :
  Controller controller createController(ComergentRequest comergentRequest)
  puis appelle :
  ```

```
controller.init(comergentContext, comergentSession,
comergentRequest, comergentResponse);
controller.execute();
```

Notez que l'instance de la classe Controller créée par la méthode *createController()* est une fonction de la demande. La classe Controller est déterminée par le type de message de la demande, parce que le contrôleur est créé par la classe GeneralObjectFactory. La classe GeneralObjectFactory utilise le fichier **MessageTypes.xml** pour mapper le type de message de la demande à une classe Controller.

- DebsDispatchServlet : Ce servlet est utilisé pour traiter les messages XML publiés dans Visual Modeler à partir d'un autre système. Si le type de contenu de la demande commence par "application/x-icc-xml" ou "text/xml", la classe MessagingController est appelée pour traiter la demande.

Classes Cde contrôleur

Visual Modeler offre deux façons d'utiliser les contrôleurs pour traiter les demandes :

Contrôleurs personnalisés

Vous pouvez écrire votre propre classe Controller en étendant la classe com.comergent.dcm.caf.controller.Controller. Lors de cette opération, vous devez fournir la logique d'application pour déterminer la page JSP vers laquelle la demande doit être dirigée. Par exemple :

```
boolean processingSuccess = false;
/*
*
* La logique métier traite la demande et définit processingSuccess
* sur true en cas de réussite.
*/
if (processingSuccess)
{
callJSP("SuccessMessageType");
}
else
{
callJSP("FailureMessageType");
}
protected void callJSP(String messageType) throws
ControllerException, ICCException, IOException
{
String resource = getJSPName(messageType);
ComergentDispatcher rd =
request.getComergentDispatcher(resource);
rd.forward(request, response);
}
protected String getJSPName(String messageType) throws ICCException
{
JSPObjectID id = new JSPObjectID(messageType);
return GeneralObjectFactory.getGeneralObjectFactory().-
getMapping(id);
}
```

SimpleController

Vous pouvez étendre la classe SimpleController pour traiter la demande s'il y a un seul point d'exit pour la logique d'application. La classe SimpleController utilise le type de message de la demande pour déterminer la page JSP vers laquelle la demande doit être dirigée une fois la logique d'application terminée. Pour étendre la classe SimpleController, écrasez la méthode *calculate()*.

MessagingController

Cette classe est utilisée pour traiter des demandes XML (par exemple des demandes de prix et de disponibilité ou de transfert de panier à partir d'autres systèmes).

Classes DataBean

L'accès aux données dans Visual Modeler est géré via les objets de données, qui sont des documents XML décrivant des entités métier telles que des partenaires, des utilisateurs, des produits, etc. Ils contiennent les zones de l'objet de données et fournissent des informations sur le mappage entre ces zones et les tables de base de données dans Knowledgebase. Chaque fichier XML d'objet de données est utilisé pour générer une classe Java DataBean correspondante.

Les classes DataBean sont les principales classes utilisées pour représenter chaque entité métier dans Visual Modeler. Chaque entité métier tel qu'un utilisateur, un partenaire, un produit, etc. est représentée dans la mémoire par une instance de la classe DataBean correspondante. Pour plus d'informations, voir «Beans de données dans Visual Modeler», à la page 45. Bien que certaines applications existantes utilisent encore la classe BusinessObject, son utilisation est généralement obsolète.

Les classes DataBean sont également utilisées pour transmettre des données aux pages JSP. Toutes les définitions d'objets de données figurant dans le schéma XML de Visual Modeler peuvent être utilisées pour générer une classe DataBean en exécutant la cible generateBean. Pour plus d'informations, voir «Utilisation du kit de développement de logiciels (SDK) pour personnaliser l'implémentation de Visual Modeler», à la page 83.

La classe DataBean est une classe abstraite générale et toutes les classes de bean de données générées étendent cette classe. Chaque classe DataBean fournit les méthodes *restore()* et *persist()*, qui permettent respectivement de récupérer et d'enregistrer des données dans la base de données.

Certaines applications utilisent des beans application. Pour plus d'informations sur l'utilisation de ces beans, voir «Beans entity, application et présentation», à la page 51.

Classes ObjectManager et OMWrapper

Vous ne devez pas instancier les classes DataBean à l'aide de leurs constructeurs. Lorsque cela est nécessaire, vous devez utiliser les classes ObjectManager et OMWrapper pour créer de nouvelles instances d'objets. Ces classes respectent le modèle de fabrique, fournissant une classe pour générer des instances d'objets lorsque de nouvelles instances sont requises. Elles permettent de passer d'une classe d'objets à une autre sans modifier le code d'application qui crée et utilise les objets.

Création d'objets

En général, vous devez utiliser la classe OMWrapper plutôt que la classe ObjectManager, même si les deux sont possibles. Pour créer des objets, vous pouvez utiliser ces classes avec les méthodes suivantes :

```
ObjectClass temp_ObjectClass =
(ObjectClass) OMWrapper.getObject("NomObjet");
```

ou

```
ObjectManager temp_ObjectManager = ObjectManager.getInstance();
ObjectClass temp_ObjectClass =
(ObjectClass) temp_ObjectManager.getObject("NomObjet");
```

Mappage des noms d'objets aux classes d'objets

Les classes ObjectManager et OMWrapper utilisent le fichier de configuration **ObjectMap.xml** (situé sous ***rép_base_debs*/Sterling/WEB-INF/properties/**) pour déterminer quel type d'objet doit être créé à partir du nom d'objet fourni dans la méthode *getObject()*.

Remarque : N'ajoutez pas de commentaires au fichier **ObjectMap.xml**, car cela peut entraîner des erreurs lors de l'initialisation.

Chaque élément Object se présente comme suit :

```
<Object ID="NomObjet">
<ClassName>ClasseObjets</ClassName>
</Object>
```

Lorsque la méthode *getObject("NomObjet")* est appelée, une instance de la classe ClasseObjets est renvoyée. Le *NomObjet* doit correspondre au nom de la classe ou de l'interface Java et *ClasseObjets* doit être une sous-classe de (ou correspondre à) la classe *NomObjet* ou une classe qui implémente l'interface *NomObjet*.

Si le fichier **ObjectMap.xml** ne contient pas un élément Object dont l'attribut ID correspond au paramètre NomObjet, la classe ObjectManager ou OMWrapper crée une instance de la classe NomObjet. Autrement dit, elle se comporte comme s'il y avait un élément de la forme suivante :

```
<Object ID="NomObjet">
<ClassName>NomObjet</ClassName>
</Objet>
```

Par exemple, supposons que le fichier ObjectMap.xml contient l'élément suivant :

```
<Object ID="com.comergent.bean.productMgr.ProductBean">
<ClassName>
com.comergent.bean.productMgr.MatrixProductBean
</ClassName>
</Object>
```

L'appel de la méthode suivante crée alors une instance de la classe MatrixProductBean :

```
ProductBean temp_ProductBean = (ProductBean)
OMWrapper.getObject("com.comergent.bean.productMgr.ProductBean");
```

Notez que MatrixProductBean doit étendre la classe ProductBean, autrement une exception ClassCastException est lancée lors de l'exécution. Cependant, s'il n'existe pas d'élément dont l'attribut ID correspond à com.comergent.bean.productMgr.ProductBean, le même appel renvoie une instance de la classe com.comergent.bean.productMgr.ProductBean.

Restrictions

Lorsque vous définissez des éléments Object, la classe spécifiée par l'élément ClassName d'un élément Object ne peut pas correspondre à l'attribut ID d'un autre élément Object. La seule exception à cette règle concerne la situation où la classe est utilisée à la fois comme valeur d'ID et comme valeur de l'élément ClassName pour le même élément Object. Plus particulièrement, si vous étendez un objet de données (voir «Extension d'objets de données», à la page 57), procédez comme suit :

1. Définissez un élément Object qui mappe la classe étendue à la classe d'extension :

   ```
   <Object ID="<Classe étendue>">
   <ClassName><Classe d'extension></ClassName>
   </Object>
   ```

2. Assurez-vous de remplacer toutes les références à l'objet de données étendu dans l'ensemble des éléments ClassName par une référence à l'objet de données d'extension.

Passage de paramètres

Si vous souhaitez passer des paramètres aux constructeurs d'objets, vous pouvez utiliser la méthode OMWrapper suivante :

```
ObjectClass temp_ObjectClass = (ObjectClass)
OMWrapper.getObjectArg("NomObjet", Object arg1, ... ,
Object arg10);
```

Sous cette forme, vous pouvez passer jusqu'à dix paramètres en tant qu'objets dans l'appel de méthode. Les méthodes OMWrapper et ObjectManager permettent de passer un nombre de paramètres illimité en tant que tableau d'objets :

```
ObjectClass temp_ObjectClass = (ObjectClass)
OMWrapper.getObject("NomObjet", Object[] args);
```

ou

```
ObjectClass temp_ObjectClass = (ObjectClass)
temp_ObjectManager.getObject("NomObjet", Object[] args);
```

Par exemple, supposons que le fichier ObjectMap.xml contient l'élément suivant :

```
<Object ID="com.comergent.bean.productMgr.OrderBean">
<ClassName>com.comergent.bean.matrix.MatrixOrderBean</ClassName>
</object>
```

Dans cet exemple, la classe MatrixOrderBean est une sous-classe de la classe OrderBean. Supposons que MatrixOrderBean comporte un constructeur de la forme MatrixOrderBean(CartBean cb).

L'appel de la méthode suivante crée alors une instance de la classe OrderBean en passant comme paramètre une instance de la classe CartBean :

```
Cart temp_CartBean = (CartBean)
OMWrapper.getObject("com.comergent.bean.partnerMkt.CartBean");
/*
Code permettant de traiter l'objet bean de panier
*/
OrderBean temp_OrderBean = (OrderBean)
OMWrapper.getObjectArg("com.comergent.bean.productMgr.OrderBean",
temp_CartBean);
```

Pool d'objet

Si vous prévoyez de créer une classe d'objet que vous utiliserez souvent, vous pouvez utiliser les classes ObjectManager et OMWrapper pour créer un pool d'objets. L'objet parent (identifié par l'attribut ID) doit implémenter une interface pouvant être placée dans un pool. Cette interface fait partie du package com.comergent.dcm.objmgr. Elle déclare une méthode reset() que vous devez implémenter.

Lorsque vous avez terminé la création de l'objet à placer dans un pool, vous pouvez le renvoyer au pool d'objet en appelant la méthode *return()* comme suit :

1. Dans le fichier **ObjectMap.xml**, modifiez l'entrée qui correspond à la classe à placer dans un pool en définissant l'attribut MaxPoolSize sur le nombre d'objets que vous souhaitez créer dans ce pool :
   ```
   <Object ID="NomObjet" MaxPoolSize="n">
   <ClassName>ClasseObjet</ClassName>
   </Object>
   ```
2. Créez des instances de cette classe d'objets à l'aide des classes OMWrapper et ObjectManager, comme indiqué précédemment.
3. Une fois que vous avez terminé la création de l'objet, renvoyez l'instance au pool à l'aide des lignes suivantes :
   ```
   OMWrapper.return(temp_ObjectClass);
   ```
4. ou
   ```
   temp_ObjectManager.return(temp_ObjectClass);
   ```

Notez que si vous créez un objet en passant des paramètres comme indiqué dans «Passage de paramètres», à la page 25, Visual Modeler crée un nouvel objet au lieu de réutiliser un objet du pool.

Classe AppExecutionEnv

La classe AppExecutionEnv permet d'exécuter les classes de logique métier. Cependant, l'utilisation des classes de logique métier étant obsolète, il est recommandé d'utiliser cette classe pour prendre en charge les applications existantes uniquement. Vous pouvez utiliser les méthodes statiques *runAppObj()* pour appeler la création d'une classe de logique métier et exécuter les méthodes prolog et service de celle-ci.

En général, cette classe est utilisée comme suit :

```
AppExecutionEnv.runAppObj(String messageType, BizObjTable bizObjects)
```

La classe AppExecutionEnv appelle la classe de logique métier indiquée par la chaîne messageType, qui prend le vecteur d'objets métier BizObjTable comme objet métier d'entrée.

Classe AppsLookupHelper

Dans de nombreuses situations, l'état d'un objet de données dans Visual Modeler est géré à l'aide d'un code de recherche. Par exemple, l'état d'une commande peut changer plusieurs fois au cours de son traitement. Les zones d'affichage peuvent également être différentes, comme le Titre de l'utilisateur, qui peut avoir plusieurs valeurs prédéfinies et doit être géré en fonction de différents paramètres régionaux. Ces données sont stockées dans la table CMGT_LOOKUPS du schéma de base de données Knowledgebase.

Pour chaque type de recherche, vous pouvez définir un ou plusieurs codes de recherche et chaque code est associé à une chaîne de description. Par exemple :

Type de recherche	Code de recherche	Description
Type d'adresse	10	Facturation
Type d'adresse	20	Expédition

Vous pouvez utiliser la classe AppsLookupHelper pour mapper un code de recherche à une chaîne de description. En appelant la méthode appropriée de la classe AppsLookupHelper, passez comme paramètre le code de recherche pour renvoyer la chaîne correspondante. La méthode que vous devez appeler varie en fonction du type de recherche. La méthode appelée détermine le type de recherche utilisé pour récupérer le code de recherche à partir de la table CMGT_LOOKUPS. Par exemple, pour récupérer une chaîne de code d'état d'une commande, vous pouvez saisir :

```
String orderStatusString =
AppsLookupHelper.getOrderStatusForCode(orderStatusCode);
```

```
Inversement, vous pouvez récupérer le code de recherche en saisissant :

int orderStatusCode =
AppsLookupHelper.getCodeForOrderStatus(orderStatusString);
```

Des méthodes d'auxiliaire sont définies pour la plupart des types de recherche. Consultez la section sur la classe AppsLookupHelper dans la documentation Java pour obtenir plus de détails. Pour plus d'informations, voir «Prise en charge des codes de recherche», à la page 31.

Classe ComergentAppEnv

Utilisez la classe ComergentAppEnv pour inclure dans votre code des informations sur l'environnement spécifiques à l'application. Elle fournit les méthodes suivantes :

- *adjustFileName()* : Cette méthode à été déplacée dans la classe LegacyFileUtils. Voir «Classe LegacyFileUtils», à la page 28.
- *constructExternalURL()* : Cette méthode construit une URL permettant de rediriger le client vers le serveur. Elle est principalement utilisée pour générer une URL de redirection afin de permettre au serveur de restaurer les informations de session.
- *getEnv()* : Cette méthode renvoie l'objet d'environnement.

- *getContext()* : Cette méthode renvoie le contexte d'application.

Classe Global

L'utilisation de cette classe est obsolète. La fonction de consignation est désormais assurée par l'API log4j. Pour plus d'informations, voir «Présentation de la consignation dans Visual Modeler», à la page 67. La fonction de récupération des valeurs des propriétés a été remplacée par le mécanisme Preferences. Si vous devez utiliser un code existant qui contient la classe Global, remplacez chaque occurrence de cette classe par la classe LegacyPreferences.

Interface GlobalCache

Utilisez cette interface pour définir un cache qui fournit l'accès aux objets mis en cache utilisés par toutes les applications Visual Modeler. Elle permet de prendre en charge un environnement groupé dans lequel Visual Modeler est exécuté sur plusieurs machines.

Pour utiliser une classe de cache qui implémente l'interface GlobalCache, vous devez implémenter les méthodes de l'interface. La classe de cache est chargée lorsque la méthode InitServlet *init()* est appelée. Vous devez spécifier le nom de la classe en tant que valeur de l'élément General.globalCacheImplClass dans le fichier **Comergent.xml**. Visual Modeler fournit l'implémentation par défaut com.comergent.dcm.cache.impl.AppContextCache.

Pour accéder à l'implémentation de l'interface GlobalCache, procédez comme suit :

```
GlobalCache globalCache = GlobalCacheManager.getGlobalCache();
```

L'interface prend en charge les méthodes suivantes :

- *public String store(Serializable entry)* : Stocke un objet dans le cache global jusqu'à ce qu'il soit nettoyé par l'application.
- *public boolean store(String id, Serializable entry)* : Stocke un objet dans le cache global jusqu'à ce qu'il soit nettoyé par l'application.
- *public String cache(Serializable entry)* : Stocke un objet dans le cache global. L'objet est disponible tant qu'il est utilisé par l'application, mais il est nettoyé automatiquement par le système de cache.
- public String cache(Serializable entry, long lease)
- public boolean cache(String id, Serializable entry)
- public boolean cache(String id, Serializable entry, long lease)
- *public boolean contains(String id)* : Vérifie si un objet spécifique est présent dans le cache.
- *public Object get(String id)* : Récupère un objet pouvant être mis en cache.
- *public Object remove(String id)* : Supprime un objet pouvant être mis en cache.
- *public boolean gc()* : Cette méthode doit être appelée par un travail cron pour nettoyer les entrées inutilisées du cache.

Classe LegacyFileUtils

La classe LegacyFileUtils fournit des méthodes d'auxiliaire pour l'utilisation de fichiers. Son utilisation est obsolète, mais elle prend en charge les méthodes fournies précédemment par la classe ComergentAppEnv :

- *adjustFileName()* : Renvoie le chemin d'accès réel d'un fichier. Utilisez cette méthode pour accéder aux fichiers en lecture ou en écriture ; n'utilisez pas la

méthode *getRealPath()*, qui peut renvoyer la valeur null. Dans un environnement groupé, la méthode *adjustFileName()* permet de s'assurer que tous les membres du cluster accèdent au même fichier. Vous devez utiliser cette méthode avec quatre paramètres :

```
adjustFileName(String fileName, boolean share, boolean xPublic,
boolean xLoadable);
```

L'utilisation de cette méthode avec un seul paramètre est obsolète. Les paramètres booléens permettent de déterminer l'emplacement du fichier à l'aide des paramètres de configuration spécifiés dans l'élément WritableDirectory du fichier **web.xml**.

Classe OutOfBandHelper

La classe OutOfBandHelper permet de générer un flux de sortie à l'aide d'une page JSP utilisée en tant que modèle. Voici un exemple d'utilisation de cette classe :

```
ComergentRequest request = ComergentAppEnv.getRequest();
ComergentResponse response = ComergentAppEnv.getResponse();
ByteArrayOutputStream stream = new ByteArrayOutputStream();
OutOfBandHelper outOfBandHelper = new OutOfBandHelper(request,
response, stream);
outOfBandHelper.getRequest().setAttribute(
ComergentRequest.COMERGENT_SESSION_ATTR,
request.getComergentSession());
outOfBandHelper.callJSP(messageType);
/*
* Lancer SendSMTP et utiliser le flux pour définir le corps
* du message
*/
String mimeType = "text/html";
String smtpHost = Global.getString(
"C3_Commerce_Manager.SMTP.SMTPHost");
SendSMTP smtp = new SendSMTP(smtpHost);
StringBuffer sb = new StringBuffer(subject);
String message = null;
String enc = ComergentI18N.getComergentEncoding();
message = stream.toString(enc);
//Envoyer l'e-mail
smtp.send( from, to, cc, subject, message, mimeType);
```

Cet exemple montre l'initialisation de la classe OutOfBandHelper à l'aide des objets demande et réponse existants et d'un flux de sortie. Sa méthode*callJSP()* génère le flux de sortie en passant les objets demande et réponse à la page JSP déterminée par le paramètre de type de message ; le flux de sortie peut être utilisé par l'application pour récupérer du contenu.

La classe OutOfBandHelper utilise les informations de contexte et de session lors du mappage d'un type de message à une page JSP. Par conséquent, vous pouvez utiliser des pages JSP distinctes pour des paramètres régionaux distincts, comme vous le faites pour traiter des demandes de navigateur ; la classe OutOfBandHelper déterminera quelle page JSP utiliser en fonction de vos paramètres régionaux et appliquera la même logique de basculement.

Classe Preferences

Le module Preferences fournit un mécanisme permettant d'accéder aux propriétés de Visual Modeler. Il est fourni avec les modules de plateforme. Pour plus d'informations, voir «Service Préférences», à la page 40. Voici un exemple typique d'utilisation de l'API Preferences :

```
private static Preferences temp_Preferences =
Preferences.getPreferences();

String temp_MyPropertyString =
temp_Preferences.getString("MyProperty");
```

Les principales méthodes prises en charge pour récupérer des propriétés sont les suivantes :

- public String getString(String key, String def)
- public boolean getBoolean(String key, boolean def)
- public double getDouble(String key, double def)
- public float getFloat(String key, float def)
- public int getInt(String key, int def)
- public long getLong(String key, long def)

Des méthodes *put*Type*()* correspondantes sont disponibles pour chaque méthode *get*Type*()*. Par exemple :

- public void putString(String key, String value)

Si vous appelez la méthode *getPreferences()* sans indiquer de paramètre, vous récupérez l'objet singleton Preferences pris en charge par Visual Modeler. Si vous passez le nom d'une classe (par exemple getPreferences(MyClass.class)), l'objet récupéré est étendu. Cela signifie que le nom des propriétés dont vous récupérez les valeurs à l'aide de l'objet Preferences a le même chemin d'accès au package que la classe ajoutée en préfixe au nom de propriété que vous avez fourni.

Par exemple, supposons que myClass se trouve dans le package com.comergent.myApplication. Les deux fragments de code suivants sont alors équivalents :

```
private static Preferences temp_Preferences =
Preferences.getPreferences();

String temp_MyPropertyString =
temp_Preferences.getString("com.comergent.myApplication.MyProperty");
```

et :

```
private static Preferences temp_Preferences =
Preferences.getPreferences(com.comergent.myApplication.MyClass.class);

String temp_MyPropertyString =
temp_Preferences.getString("MyProperty");
```

Transactions

Visual Modeler prend en charge les transactions, qui sont définies comme des actions de bases de données qui s'étendent sur une ou plusieurs opérations atomiques. En général, vous devez utiliser la classe Transaction pour gérer les situations dans lesquelles vous devez faire persister plusieurs objets de données ensemble et dans lesquelles tous les objets de données doivent échouer en cas d'échec d'un objet de données.

Prise en charge des codes de recherche

Visual Modeler utilise des codes de recherche pour fournir un mécanisme de stockage et d'affichage des chaînes spécifiques aux paramètres régionaux. Pour chaque type de recherche, vous pouvez définir un ou plusieurs codes de recherche, et pour chaque code de recherche vous pouvez définir une chaîne dans chaque ensemble de paramètres régionaux pris en charge.

Quels types de recherches sont prises en charge dans Visual Modeler ?

Visual Modeler offre une fonction de recherche automatique de chaînes à partir des valeurs de code correspondantes et inversement.

Si l'élément DsElement "code" est défini, l'élément "string" est rempli automatiquement à partir du cache de recherche. Si la valeur de l'élément "string" est définie, elle est utilisée pour rechercher la valeur de l'élément "code".

Les valeurs des chaînes sont-elles localisées ?

Oui. Pour la recherche d'une chaîne à partir d'un code, le mécanisme utilise les paramètres régionaux de l'utilisateur pour déterminer quelle valeur de chaîne utiliser. Pour la recherche d'un code à partir d'une chaîne, le mécanisme utilise les paramètres régionaux de l'utilisateur pour déterminer le code en fonction de la valeur de la chaîne.

Comment définir un mappage du code à la chaîne ?

Les relations de code à chaîne sont définies dans le fichier de schéma **DsDataElement.xml**. Si les éléments DsDataElement "code" et "string" sont tous deux utilisés dans un objet de données, le mappage du code à la chaîne est géré automatiquement.

Voici un exemple de paire code-chaîne définie dans un élément DataElement :

```
<DataElement Name="EtatCommande" Description="État de la commande"
DataType="LONG" MaxLength="20" LookupType="OrderStatus"
LookupString="OrderStatusString"/>
<DataElement Name="ChaineEtatCommande" Description="État de la commande"
DataType="STRING" MaxLength="260" LookupType="OrderStatus"
LookupCode="OrderStatus"/>
```

Y a-t-il des recherches les les messages XML ?

Oui. Si un objet de données utilisé pour la messagerie contient une paire code-chaîne, la valeur de la chaîne est utilisée automatiquement pour rechercher le code.

Comment le cache de recherche est-il chargé ?

Le cache de recherche est chargé au démarrage du système.

Chapitre 7. Modularité de la plateforme

Présentation de la modularité de la plateforme Visual Modeler

L'architecture modulaire de Visual Modeler est conçue pour faciliter la personnalisation et la mise à niveau des implémentations. Grâce à la construction modulaire de la plateforme Visual Modeler, vous pouvez la modifier et la mettre à niveau plus rapidement et plus facilement et réutiliser des modules pour la construction d'autres produits.

La fonctionnalité de plateforme fournie à travers les modules de plateforme et la possibilité pour les modules d'appeler d'autres modules uniquement via leur interface externe présentent les avantages suivants :

- Un compartimentage plus facile du fonctionnement des applications.
- Une meilleure compréhension et une gestion plus facile des relations de dépendance entre les différentes parties de Visual Modeler.
- Personnalisation plus facile de modules individuels et meilleure compréhension des effets provoqués par la modification d'un module sur l'ensemble du système.
- Mise à niveau plus facile des modules indépendamment les uns des autres, ce qui permet de réduire les éventuels effets d'une mise à niveau.
- La mise à niveau des modules non personnalisés n'affecte pas les personnalisations effectuées dans d'autres modules.
- Vous pouvez fournir une nouvelle fonctionnalité en plaçant un module dans un déploiement existant de Visual Modeler.

Modules de plateforme

La plateforme Visual Modeler a été développée comme un ensemble de modules interdépendants qui respectent une structure organisationnelle commune. En général, chaque module de plateforme correspond à un composant fonctionnel de Visual Modeler, tel qu'un service ou un composant de la plateforme Visual Modeler. Les modules de plateforme fournissent une API Java aux autres modules. Certains modules fournissent un ensemble de classes d'auxiliaire utilisées par d'autres modules.

En général, tous les modules de plateforme ont la structure suivante :

- Classes Java : Elles sont organisées selon les arborescences ci-après. Lors de la phase de création, les répertoires du module sont réunis en un seul fichier JAR.
  - com.comergent.api.*module* - Interfaces API externes : Elles sont utilisées par les autres modules pour accéder à la fonctionnalité fournie par un module. En général, lorsqu'un module effectue un appel vers une classe d'un autre module, il doit utiliser l'API externe de l'autre module. Il s'agit du package com.comergent.api du module.
  - com.comergent.*module* - Classes d'implémentation : Elles sont utilisées pour l'implémentation des API externes. Lorsqu'un autre module effectue un appel vers l'API externe du module, les classes utilisées sont les classes d'implémentation de l'interface du module. Les packages d'implémentation peuvent inclure des classes internes, qui sont utilisées par les classes d'implémentation ; ces classes ne sont pas publiques et ne font pas partie de la Javadoc fournie.

- Fichiers de configuration spécifiques au module, tels que des fichiers de propriétés. Ils sont stockés dans la hiérarchie de classes pour pouvoir être référencés à l'aide des appels *getResource()*.

Modularité de la plateforme : Interfaces des modules

Chaque module de plateforme doit fournir une interface externe utilisée pour effectuer tous les appels vers les classes et les interfaces Java du module. Un ensemble complet de pages Javadoc est disponible pour cette interface externe, afin qu'elle puisse être utilisée facilement et de façon fiable par les développeurs d'autres modules.

Les interfaces externes sont organisés dans le package the com.comergent.api. Ce package contient toutes les API externes prises en charge par les modules. Les API sont organisées par module : com.comergent.api.converter, com.comergent.api.logging, etc.

Appel des interfaces

Vous pouvez appeler une interface à partir d'une classe Java en convertissant n'importe quel objet ou interface enfant vers l'interface souhaitée, puis en appelant une méthode déclarée par cette interface. Chaque module utilise l'une ou l'autre de ces techniques, mais pas les deux. Lorsque vous utilisez un module existant ou créez un nouveau module, veillez à conserver une cohérence dans les noms d'interfaces. Vos collègues pourront ainsi travailler plus facilement avec le même module.

En général, vous devez toujours essayer d'utiliser les interfaces fournies par les packages com.comergent.api ; ces interfaces sont prises en charge par les modules de plateforme d'une version à l'autre, sans tenir compte des éventuelles modifications apportées aux implémentations sous-jacentes.

Description des modules de plateforme

Cette section décrit brièvement l'objectif de chaque module de plateforme et fournit des exemples de son utilisation.

Règle d'accès

Ce module fournit le service utilisé pour contrôler les règles d'accès.

Authentification

Ce module fournit les API utilisées pour authentifier les données d'identification et les utilisateurs.

Base64

Ce module fournit les classes utilisées pour convertir des données au format Base 64.

Ajouteur de chemin d'accès aux classes

Ce module fournit des classes utilisées pour ajouter des chemins au chemin d'accès aux classes.

Service de cryptographie

Ce module fournit les services utilisés pour chiffrer et déchiffrer les données.

Services de données

Ce module fournit une version réorganisée et nettoyée de la fonction de service de données existante. Son API a été déplacée vers un package distinct, com.comergent.api.dataservices. Pour gérer ses propriétés, les services de données utilisent désormais le même mécanisme de préférences que les autres fonctions de Visual Modeler. Le pool de connexions a été réuni dans un seul pool réglable. La pagination a été mise à jour et n'est plus basée sur l'écriture de fichiers pbtagination sur le système de fichiers.

Autorisation de répartition

Ce module gère les contrôles d'accès, qui sont utilisés pour s'assurer que chaque utilisateur ne peut accéder qu'aux parties de l'application pour lesquelles il dispose des autorisations appropriées.

Infrastructure de répartition

Ce module gère l'infrastructure de répartition des classes Visual Modeler qui encapsulent les classes de demande, de réponse, de contexte et de session du servlet, ainsi que les classes de contrôleur de base utilisées par le mécanisme de répartition.

Service d'e-mail

Ce module fournit les API de base permettant l'envoi de courrier électronique à partir de Visual Modeler.

Service d'événement

Ce module fournit les classes utilisées par les API EventBus et Event.

Service d'exception

Ce module fournit les classes et l'infrastructure d'exception de base utilisées par Visual Modeler.

Service de cache global

Ce module fournit les API à utiliser pour accéder au cache.

Aide

Ce module fournit la classe ComergentHelpBroker, qui est une classe d'encapsuleur standard de la classe ServletHelpBroker de l'implémentation JavaHelp 2.0.

Service d'initialisation

Le module d'initialisation fournit le service d'initialisation. Ce package permet d'initialiser Visual Modeler à l'aide d'une infrastructure de classes et de méthodes cohérente.

Initialization Manager fournit un point focal permettant de :

- Définir des tâches d'initialisation
- Imposer des stratégies en cas d'échec de l'initialisation
- Agréger des fragments de configuration

La principale fonction d'Initialization Manager est d'agir sur une liste de tâches d'initialisation de manière prévisible et bien définie. Cela implique d'avoir une liste ordonnée qui :

- peut être définie par programmation
- ou peut être incluse dans un fichier au format XML

Le fragment de code suivant fournit un exemple typique d'utilisation de la classe InitManager.

```
InitManager initManager = InitManager.getInitManager();
try
{
String resourceName = args[0];
initManager.init(resourceName);
//ou créé par programmation
//List modules = initModules();
//ResourceLocator resourceLocator = createNewResourceLocator();
//initManager.init(modules, resourceLocator);
}
catch (InitManagerException ime)
{
log.error(ime, ime);
System.exit(1);
}
//Initialisation terminée. Vous pouvez continuer//
...
```

Vous pouvez défini le processus d'initialisation à l'aide d'un fichier de configuration. Voici un exemple de fichier que vous pouvez utiliser :

```
<?xml version="1.0" encoding="UTF-8"?>
<initializationManager>
<resourceLocator>
<path>/a/b/c</path>
<path>.</path>
<path>CLASSPATH</path>
</resourceLocator>
<module name="ObjectManager"
initClass="com.comergent.objectManager.InitHelper>
<config name="Preferences">
/com/comergent/objectManager/preferences.xml
</config>
<init-param name="param0">param0Value</init-param>
</module>
<module name="module1" initClass="com.comergent.module1.InitHelper>
<config name="ObjectManager">
/com/comergent/module1/objectMap.xml
</config>
<config name="MessageTypes">
/com/comergent/module1/messageTypes.xml</config>
```

```
<config name="Preferences">
/com/comergent/modules1/preferences.xml
</config>
<init-param name="param1">param1Value</init-param>
</module>
<module name="module2" initClass="com.comergent.module2.InitHelper>
<config name="ObjectManager">
/com/comergent/module2/objectMap.xml
</config>
<config name="MessageTypes">
/com/comergent/module2/messageTypes.xml
</config>
<config name="Preferences">
/com/comergent/modules2/preferences.xml
</config>
<init-param name="param2">param2Value</init-param>
</module>
<!-- la classe initClass peut être absente -->
<module name="custom1" >
<config name="ObjectManager">
/com/comergent/module1/overlay/objectMap.xml
</config>
</module>
</initializationManager>
```

Dans cet exemple, lorsque la méthode suivante est appelée par Initialization Manager :

```
com.comergent.objmgr.ObjManagerInitHelper.init(initParams,
configFragments, resourceLocator)
```

les informations suivantes sont disponibles :

- initParams comporte une liste de paires clé-valeur : param0-param0Value
- configFragments comporte une liste composée de :
  - `/com/comergent/module1/objectMap.xml`
  - `/com/comergent/module12/objectMap.xml`
- resourceLocator peut trouver la ressource sous /a/b/c, sous le répertoire actuel et dans le chemin d'accès aux classes actuel.

Internationalisation

Ce module fournit une prise en charge de base des fonctions d'internationalisation offertes par Visual Modeler.

Consignation

Ce module fournit l'accès au service de consignation utilisé pour enregistrer l'activité de Visual Modeler. Le comportement du service de consignation est configuré à l'aide de son fichier de propriétés, **log4j.properties**. Ce module repose sur le projet open source log4j et utilise la même syntaxe de configuration :

Log4j est composé de *consignateurs*, d'*ajouteurs* et d'*agencements*. Ces trois types de composants fonctionnent ensemble, permettant aux développeurs de consigner les messages en fonction de leur type et de leur niveau et de contrôler leur formatage et l'emplacement où ils sont stockés lors de l'exécution.

Configuration du module de consignation

Vous pouvez configurer le module de plateforme de consignation en indiquant les propriétés des consignateurs, des ajouteurs et de l'agencement correspondants dans le fichier de configuration log4j.properties. Par exemple, le fragment suivant est utilisé pour configurer le consignateur racine et l'ajouteur CMGT :

```
# Définir la priorité de la catégorie racine
#log4j.rootCategory=info, CMGT
log4j.rootCategory=info, STDOUT
#log4j.rootCategory=info, CMGT, RTS
### START - CMGT
# Ajouteur CMGT
log4j.appender.CMGT=com.comergent.logging.ComergentRollingFileAppender
#log4j.appender.CMGT=com.comergent.logging.ComergentDailyRollingFileAppender

 #log4j.appender.CMGT.layout=org.apache.log4j.PatternLayout
log4j.appender.CMGT.layout=com.comergent.logging.ConversionPattern

 # Par défaut, le journal utilise le format "classique". Ce format est
# recommandé afin que l'analyseur de journal puisse fonctionner
# correctement dans le déploiement effectif.
log4j.appender.CMGT.layout.ConversionPattern=%d{yyyy.MM.dd HH:mm:ss:SSS} Env/%t:%p:%c{1} %m%n
```

Consignateurs

Les consignateurs sont des entités nommées. Les noms de consignateurs sont sensibles à la casse et respectent une règle d'attribution de noms hiérarchique : un consignateur est considéré comme l'ancêtre d'un autre consignateur si son nom suivi d'un point est le préfixe du nom d'un consignateur descendant. Un consignateur est considéré comme le parent d'un consignateur enfant s'il n'existe aucun ancêtre entre lui et le consignateur descendant.

Par exemple, le consignateur "com.foo" est le parent du consignateur "com.foo.Bar". De la même manière, "java" est le parent de "java.util" et l'ancêtre de "java.util.Vector". Ce schéma d'attribution de noms est généralement familier à la plupart des développeurs.

Le consignateur racine réside en haut de la hiérarchie de consignateurs. Il se distingue des autres consignateurs par le fait que :

- Il existe toujours ;
- Il ne peut pas être récupéré à l'aide de son nom.

Pour le récupérer, vous devez appeler la méthode de classe statique *Logger.getRootLogger()*. Tous les autres consignateurs sont instanciés et récupérés à l'aide de la méthode de classe statique *Logger.getLogger(String nom)*.

Cette méthode prend comme paramètre le nom du consignateur souhaité.

Vous pouvez affecter des niveaux aux consignateurs. Les différents niveaux possibles (DEBUG, INFO, WARN, ERROR et FATAL) sont définis dans la classe org.apache.log4j.Level. Si aucun niveau n'est affecté à un consignateur, il hérite du niveau de son ancêtre le plus proche ayant un niveau. En d'autres termes :

Héritage de niveau : Un consignateur donné hérite du premier niveau autre que null de la hiérarchie de consignateurs, en partant de ce consignateur et en continuant vers le haut de la hiérarchie jusqu'au consignateur racine.

Un niveau est toujours défini pour le consignateur racine pour garantir que tous les consignateurs peuvent hériter d'un niveau.

Ajouteurs

La possibilité d'activer et de désactiver des demandes de consignation individuelles en fonction de leur consignateur n'est qu'un aspect parmi d'autres. Vous pouvez associer plusieurs ajouteurs à un consignateur.

La méthode addAppender permet d'ajouter un ajouteur à un consignateur donné. Chaque demande de consignation activée pour un consignateur donné sera transférée vers l'ensemble de ses ajouteurs et vers les ajouteurs situés plus haut dans la hiérarchie. En d'autres termes, les ajouteurs sont hérités de façon additive de la hiérarchie d'un consignateur. Par exemple, si un ajouteur console est ajouté au consignateur racine, toutes les demandes de consignation activées seront imprimées au moins dans la console. Si un ajouteur fichier est également associé au consignateur, les demandes de consignation activées pour ce consignateur et ses enfants seront imprimées dans un fichier et dans la console. Il est possible de redéfinir ce comportement par défaut pour que le cumul d'ajouteurs ne fonctionne pas de façon additive ; pour ce faire, définissez l'indicateur d'additivité sur false.

Les règles régissant l'additivité des ajouteurs sont résumées ci-dessous :

- La sortie d'une instruction de consignation d'un consignateur C est transmise à tous les ajouteurs de C et à ses ancêtres. C'est le sens du terme "additivité des ajouteurs".
- Cependant, si un ancêtre du consignateur C, nommé P, a l'indicateur d'additivité défini sur false, alors la sortie du consignateur C sera dirigée vers tous ses ajouteurs et vers ses ancêtres jusqu'à P inclusivement, mais pas vers les ajouteurs des ancêtres de P.
- Par défaut, l'indicateur d'additivité des consignateurs est défini sur true.

Agencements

Outre la destination de sortie, vous pouvez aussi personnaliser le format de sortie. Pour ce faire, vous devez associer un agencement à un ajouteur. L'agencement s'occupe du formatage de la demande de consignation selon vos besoins, tandis que l'ajouteur envoie les résultats formatés à la destination. L'agencement PatternLayout, fourni dans le cadre de la distribution standard de log4j, permet de spécifier le format de sortie selon des modèles de conversion similaires à la fonction printf du langage C.

Par exemple, si vous appliquez le modèle de conversion :

```
%r [%t] %-5p %c - %m%
```

à l'agencement PatternLayout, la sortie se présentera comme suit :

```
176 [main] INFO Translator - got current date: 10/22/2005.
```

La première zone correspond au nombre de millisecondes écoulées depuis le démarrage du programme. La deuxième zone correspond à l'unité d'exécution qui envoie la demande de consignation. La troisième zone correspond à l'instruction de niveau de consignation. La quatrième zone correspond au nom du consignateur associé à la demande de consignation. Le texte suivant le signe "-" correspond au message de l'instruction.

Moniteur de mémoire

Ce module fournit des classes permettant de surveiller et de consigner la consommation de mémoire.

Autorisation selon les types de messages

Ce module fournit le service qui vérifie si les utilisateurs sont autorisés à appeler des types de message donnés.

Les interfaces sont définies dans le package com.comergent.api.dispatchAuthorization. Ce package contient les classes de fabrique, les interfaces et les exceptions requises pour ce service. Les classes d'implémentation sont définies dans le package com.comergent.dispatchAuthorization.

Le principal point d'entrée de ce module est la classe EntitlementRepository. Une instance de cette classe est obtenue à partir de la classe EntitlementFactory. Les applications peuvent créer des instances nommées de la classe EntitlementRepository. Les instances nommées facilitent le test d'unité et peuvent s'avérer utiles dans d'autres environnements de déploiement.

L'exemple suivant présente la logique exécutée par une application qui doit spécifier des règles de répartition ou d'autres objets d'autorisation selon le type de message :

```
import com.comergent.api.dispatchAuthorization.EntitlementRepository;
import com.comergent.api.dispatchAuthorization.EntitlementFactory;
import javax.xml.dom.Document;
...
Document document = ...;
...
EntitlementRepository repository =
EntitlementFactory.getEntitlementRepository();
repository.setRules(document);
```

Gestionnaire d'objets

Ce module fournit les classes permettant d'instancier des objets. Pour plus d'informations, voir «Classes ObjectManager et OMWrapper», à la page 23.

Réponse hors bande

Ce module est utilisé pour envoyer une sortie aux flux de sortie autres que les pages JSP standard.

Service Préférences

Le module Préférences permet de récupérer et de définir les propriétés de configuration utilisées par Visual Modeler. Pour récupérer des propriétés, utilisez les lignes suivantes :

```
private static final Preferences prefs =
Preferences.getPreferences(MyClass.class);
//Portée implicite de "com.comergent.apps.module.MyClass"
```

```
int max = prefs.getInt("PromotionManager.maxValue", 100);
int min = prefs.getInt("PromotionManager.minValue", 1);
```

Le second paramètre dans les appels *getInt()* indique la valeur à renvoyer si aucune propriété ayant ce nom n'est trouvée. On considère que le fichier de configuration contenant la définition de la propriété se trouve sur le chemin d'accès aux classes, par exemple dans le fichier **com.comergent.apps.module.Preferences.xml**. Si le fichier de propriétés XML est lu à l'aide du service Préférences, assurez-vous que ce fichier utilise l'élément racine Comergent. Par exemple :

```
<Comergent>
<PromotionManager>
<maxValue>50</maxValue>
<minValue>20</minValue>
</PromotionManager>
</Comergent>
```

Pour vous assurer que les propriétés sont initialisées à l'aide du service Préférences, personnalisez le fichier de configuration WEB-INF/properties/init.xml en ajoutant un élément à l'aide des lignes suivantes :

```
<module name="PromotionMgr">
<config name="Preferences">
com/comergent/reference/apps/mktMgr/controller/Init.xml
</config>
</module>
```

La classe Preferences fournit des méthodes permettant de récupérer et de placer des valeurs de propriétés. Par exemple :

```
prefs.putInt("PromotionManager.maxValue", 25);
prefs.putObject("currentShoppingCart", cartBean);
```

Lorsque vous utilisez la méthode *putObject()*, assurez-vous que l'objet respecte les spécifications de l'API XMLEncoder, qui requiert notamment que les zones de l'objet fournissent les méthodes getter et setter.

Bibliothèques de balises

Visual Modeler fournit des bibliothèques de balises en tant que modules de plateforme.

Gestion des unités d'exécution

Ce module fournit une fonction centralisée permettant de gérer les unités d'exécution ; il sert notamment à créer, réutiliser et obtenir l'état des unités d'exécution. Il est fourni par la bibliothèque backport⌂util⌂concurrent.jar. En général, les développeurs d'applications n'ont plus à appeler la méthode suivante :

```
Thread t = new Thread(new MyRunnable());
```

Le fait de disposer d'une fonction centralisée permet de :

- Placer une unité d'exécution dans un pool et la réutiliser si nécessaire
- Contrôler toutes les unités d'exécution en cours d'utilisation pour optimiser l'utilisation de l'unité centrale et des ressources

- Générer des rapports d'état standard (Visual Modeler utilise une stratégie de tableau indicateur, qui est un emplacement partagé centralisé dans lequel une unité d'exécution peut stocker son état).
- Fournir un signal d'abandon et d'interruption standard via les appels *Thread.interrupt()*, pour forcer la fermeture d'une unité d'exécution de longue durée (exécutée en boucle).

Le module offre les fonctions suivantes :

1. Pool et réutilisation transparente des unités d'exécution.
2. La possibilité pour les administrateurs d'interroger toutes les unités d'exécution en cours d'utilisation contrôlées par le gestionnaire d'unités d'exécution.
3. La possibilité pour les utilisateurs du service de l'unité d'exécution d'envoyer des rapports d'état vers un tableau indicateur commun.
4. Assistance pour suivre une boucle standard ou vérification du protocole d'état interrompu pour permettre la fermeture anticipée d'unité d'exécution de longue durée ou exécutée en boucle.
5. Fonction de temporisateur permettant la notification de l'unité d'exécution en cours d'utilisation lors l'expiration d'un temporisateur. Cette fonction permet d'implémenter une politique standard de délai d'attente ou de temps partagé.

L'API continue d'utiliser le modèle Runnable() : l'application obtient un objet de type unité d'exécution qu'elle utilise pour l'exécution.

```
Excutor executor = ExecutorFactory.getPooledExecutor();
executor.execute(new MyComergentRunnable());
```

Convertisseur de messages XML

Ce module fournit une fonction permettant de convertir des documents XML d'une catégorie de message (famille et version) vers une autre. Le nom du package est com.comergent.api.converter pour l'API et com.comergent.converter pour les classes d'implémentation.

Le package API comprend :

- ConverterFactory : Il s'agit de la classe de fabrique qui crée des convertisseurs.
- Converter : Il s'agit de la classe qui convertit un document d'une catégorie de message vers une autre. Vous pouvez spécifier des documents ou des flux en tant que sources et cibles de la conversion.

Service de messagerie XML

Ce module est utilisé pour créer et publier des messages sortants au format XML. L'API inclut les interfaces MsgContext et MsgService, la classe MsgServiceFactory et les classes MsgServiceException dans le package com.comergent.api.msgService ; les classes d'implémentation se trouvent dans le package com.comergent.msgService.

L'interface MsgService contient une méthode *service()* générique permettant de publier un bean de données et un document XML tel qu'indiqué dans le contexte de message.

Le modèle typique d'utilisation est le suivant :

1. Créer une instance de MsgContext à l'aide de MsgContextFactory ;
2. Définir les attributs appropriés de l'objet contextuel ;

3. Créer une instance MsgService pour la famille du message cible ;
4. Publier un message en appelant la méthode service avec un bean de données et un contexte de message.

Par exemple :

```
MsgContext ctx = new MsgContext();
ctx.setMessageType("ERPOrderCreateRequest");
ctx.setURL("http://www.serveur.com");
ctx.setMessageCategory("ERPOrderCreateRequest");
ctx.setContentType("text/xml");
ctx.setRemoteUser(NomUtilisateur);
ctx.setRemotePassword(MotdePasse);
MsgService msgService =
MsgServiceFactory.getMsgService(ctx.getMessageCategory());
resultBean = msgService.service(requestBean, ctx);
```

Services XML

Ce module offre des fonctions d'analyse syntaxique XML, de transformation XSL, d'encapsuleurs DOM et de classes utilitaires.

Chapitre 8. Présentation des beans de données et des objets métier dans Visual Modeler

Beans de données dans Visual Modeler

Un bean de données est la représentation d'une entité réelle dans Visual Modeler, indépendante de la source de données. Visual Modeler utilise un schéma externe (formé d'un ensemble de fichiers XML) pour définir la structure de chaque type de bean de données. Par exemple, les beans de données sont utilisées pour structurer les données des utilisateurs, des listes de demandes sur les produits, des partenaires, des produits et des espaces de travail.

- Les classes OMWrapper et ObjectManager permettent de créer des instances des classes DataBean. Pour plus d'informations, voir «Classes ObjectManager et OMWrapper», à la page 23.
- Vous pouvez créer un bean de données à l'aide de DataManager. Appelez la méthode DataManager *getDataBean(String nomBean)* pour créer un bean de données d'un type donné. Cette méthode lance une exception InvalidBizobjException si cette classe DataBean n'existe pas.

  Remarque : L'utilisation de cette méthode est obsolète car elle ne prend pas en charge les extensions de l'objet de données.

Cycle de vie d'un bean de données

En général, le flux d'utilisation d'un objet de données est le suivant :

1. Instanciez un bean de données à l'aide de la classe OMWrapper.
2. Ajoutez de données au bean à l'aide des méthodes set qui permettent d'insérer des valeurs directement dans les zones de données.
3. Faites persister le bean de données pour enregistrer le nouvel objet de données dans sa source de données pour la première fois.
4. Vous pouvez récupérer ultérieurement cet objet de données en définissant les valeurs des zones clés et en effectuant une opération *restore()* sur le bean de données pour récupérer les valeurs actuelles des zones de données à partir de la source de données.
5. Exécutez les logiques métier requises sur le bean de données. Cette action peut modifier les valeurs des zones stockées en mémoire, mais pas les valeurs stockées dans la source de données du bean de données.
6. Enregistrez les modifications apportées au bean de données en le faisant persister dans sa source de données.
7. Lorsque l'objet de données n'est plus utilisé, vous pouvez le supprimer.
8. Enfin, vous pouvez effacer l'objet de données pour supprimer intégralement les données de la source de données.

Pour les objets de données ayant comme source une base de données, le tableau suivant répertorie les méthodes Java et les méthodes SQL correspondantes que vous pouvez appeler :

Étape	Méthode Java	Méthode SQL
Instancier un objet de données	*OMWrapper.getObject()*	

Étape	Méthode Java	Méthode SQL
Remplir les zones de données	*setDataField()*	
Remplir les zones de données	*setDataField()*	
Faire persister un objet de données pour la première fois	*persist()*	INSERT
Récupérer un objet de données	*restore()*	SELECT
Logique métier permettant de mettre à jour les valeurs des zones	*getDataField()* *setDataField()*	
Enregistrer les modifications	*persist()*	UPDATE
Supprimer un objet de données	*delete()*	UPDATE **Remarque :** L'opération de suppression met à jour la colonne ACTIVE_FLAG pour la ligne de la table de base de données sous-jacente ; elle ne supprime pas l'enregistrement de la table.
Effacer un objet de données	*erase()*	DELETE

Définition d'un bean de données

Les beans de données sont définis à l'aide d'un schéma XML. Les beans de données fournissent des méthodes d'accès pour récupérer et définir les valeurs de zones de données spécifiques. Les beans de données sont généralement utilisés lors de la personnalisation des applications Visual Modeler.

Définition de la structure d'un objet de données

La structure de chaque objet de données doit être définie afin de permettre à Visual Modeler de créer une instance de l'objet de données. La définition de la structure d'un objet de données se trouve dans sont fichier de schéma XML ; elle spécifie les différentes zones de l'objet de données et ses éventuels objets enfants.

Chaque objet de données correspond à une classe Java qui étend une classe DataBean. Ces classes s'appellent des classes de bean de données. Les classes de bean de données sont générées automatiquement dans le cadre du processus de fusion de l'outil SDK. Lorsque vous générez la classe de bean de données correspondante, celle-ci fournit des méthodes permettant d'accéder aux zones et aux beans de données enfants déclarés dans le fichier XML de l'objet de données.

Vous pouvez modifier la définition du schéma XML et, par conséquent, la définition des objets de données et des classes de bean de données correspondantes en modifiant les fichiers de schéma XML.

Le fichier de configuration **DsRecipes.xml** est utilisé pour relier chaque objet de données à sa source de données. Il indique également si l'ordinalité de l'objet de données correspond à "1" ou à "n". Le fichier de l'objet de données contient la

structure précise de l'objet de données et le fichier de configuration **DsDataElements.xml** indique le type de données (LIST, LONG, STRING, etc.) de chaque élément.

Extension des objets de données

Lorsque vous définissez un objet de données à l'aide d'un fichier de schéma XML, vous pouvez déclarer qu'il étend un autre objet de données en spécifiant l'attribut Extends. Cette fonction est utilisée de deux façons :

- Vous pouvez utiliser un objet de données en tant que parent de plusieurs objets de données d'extension qui partagent un ensemble commun de zones de données commun. Par exemple, de nombreux objets de données dans Visual Modeler étendent l'objet de données C3PrimaryRW ; ce dernier fournit les zones de données de base OwnedBy et AccessKey, utilisées pour la gestion du contrôle d'accès.
- Vous pouvez personnaliser un objet de données en créant un objet de données qui l'étend. L'ajout de zones de données à l'objet de données d'extension permet d'ajouter des attributs dont vous avez besoin dans le cadre de la personnalisation. Grâce à ObjectManager, vous pouvez vous assurer que l'objet de données d'extension est créé lorsque le système reçoit la demande de création d'un objet de données de type étendu. Si le code existant utilise ObjectManager pour instancier les instances de l'objet de données, l'appel de ce code déclenche la création des instances de l'objet de données d'extension ; ces instances prennent en charge les interfaces de l'objet de données étendu, ce qui permet au code existant de continuer de fonctionner.

DataManager utilise une *recette* et un *objet de données* pour déterminer la structure d'élément du bean de données ou de l'objet métier, ainsi que l'emplacement de la source de données qui stocke les valeurs de l'élément. Lorsque vous modifiez la définition des objets de données ou vous créez de nouvelles définitions, vous devez réexécuter les cibles SDK generateDTD et generateBean pour créer et compiler les classes DataBean. Pour plus d'informations, voir «Utilisation du kit de développement de logiciels (SDK) pour personnaliser l'implémentation de Visual Modeler», à la page 83. Pour plus d'informations sur les autres méthodes d'extension des objets de données, voir «Extension d'objets de données», à la page 57.

Création de beans de données et d'objets métier

Les classes ObjectManager et OMWrapper de Visual Modeler permettent de créer des beans de données, qui seront traités par les classes de logique métier et les contrôleurs. Pour plus d'informations, voir «Classes ObjectManager et OMWrapper», à la page 23.

Les classes de logique métier sont appelées par des contrôleurs ; chaque contrôleur détermine quelle classe de logique métier doit être créée (le cas échéant) en réponse à un message et à un type de message.

L'utilisation d'objets métier et de la classe BusinessObject est obsolète. Lorsque cela est possible, vous devez utiliser des classes de beans de données et limiter l'utilisation des objets métier aux situations où vous vous devez maintenir le code existant.

DataContext

La méthode *restore()* prend comme paramètre une instance de la classe DataContext. La classe DataContext est utilisée pour indiquer des informations sur le contexte dans lequel l'opération *restore()* est effectuée. Vous pouvez spécifier le nombre maximum de résultats à renvoyer et le nombre de résultats affichés par page pour gérer la pagination. Cette classe permet également d'indiquer si les résultats de l'opération *restore()* doivent faire l'objet d'un contrôle d'accès. Par défaut, un contrôle d'accès est effectué.

Par exemple, le fragment de code suivant crée un objet DataContext, définit certaines valeurs de contexte et utilise ce contexte avec une requête pour restaurer un bean de données :

```
DataContext temp_DataContext = new DataContext();
temp_DataContext.setMaxResults(DsConstants.NO_LIMIT);
temp_DataContext.setNumPerPage(-1);
skuMappingListBean.restore(temp_DataContext, query);
```

Lors de l'initialisation d'un objet DataContext, celui-ci récupère à partir des fichiers de configuration les valeurs des éléments DataServices.General.MaxResults et DataServices.General.NumPerCachePage et définit ces paramètres pour l'opération de restauration. Par défaut, aucune limite n'est définie pour ces éléments. Si vous devez modifier le comportement de l'objet DataContext, vous pouvez utiliser les méthodes d'accès disponibles. Pour plus d'informations, voir la documentation Java sur l'objet DataContext.

La classe DataContext fournit la méthode *setCacheId(String cacheId)* pour prendre en charge la pagination ; cette méthode permet d'identifier le cache utilisé.

Qu'est-ce que la classe DataContext ?

La classe DataContext est utilisée pour contrôler le comportement des opérations restore et persist.

Quel comportement peut être contrôlé ?

Une instance DataContext peut contrôler :

- Le nombre de résultats de la requête affichés par page.
- Le nombre maximum de résultats de la requête pouvant être traités.
- La possibilité d'utiliser plusieurs ensembles de pages par session et type de bean de données.

À quoi servent les méthodes d'ID de cache ?

Les méthodes d'ID de cache permettent à une application d'indiquer un identificateur unique pour la pagination des ensembles de résultats. Grâce à cette nouvelle fonction, une application peut conserver plusieurs ensembles de résultats distincts pour un bean de données et une session donnés.

Si l'application ne spécifie pas d'ID de cache, le cache est identifié à l'aide d'une combinaison du nom du bean et de l'ID de session. Dans ce cas, toute nouvelle tentative de restauration du même bean de données dans le cadre de la même session écrasera tous les résultats existants.

La classe DataContext fournit les méthodes suivantes pour contrôler l'ID de cache pour les demandes de restauration du bean de données :

- void setCacheId(String cacheId) : Définit un nouvel ID de cache. Cette chaîne est utilisée en combinaison avec le nom du bean et l'ID de session pour générer un identificateur unique.
- String getCacheId( ) : Renvoie l'ID de cache actuel ou la valeur null si l'ID de cache n'est pas défini.

Comment fonctionnent les attributs maxResults et numPerPage ?

L'attribut paramètre maxResults détermine le nombre maximum d'enregistrements pouvant être récupérés lors d'une restauration. Lorsque ce nombre est atteint, la demande est libérée.

La valeur de numPerPage détermine le nombre d'enregistrements affichés dans chaque page de cache de résultats. Si le nombre d'enregistrements effectif est inférieur à la valeur de ce paramètre, le cache de résultats n'est pas créé.

Notez que cette combinaison d'attributs permet à l'application de récupérer un ensemble de résultats paginés, tout en offrant la possibilité d'indiquer le nombre maximum d'enregistrements à récupérer.

La classe DataContext fournit les méthodes suivantes aux attributs maxResults et numPerPage pour les demandes restore et persist effectuées sur les beans de données :

- void setMaxResults(int maxResults) permet de définir le nombre maximum de résultats renvoyés pour les résultats non paginés
- int getMaxResults( ) permet de récupérer le nombre maximum de résultats renvoyés pour les résultats non paginés
- void setMaxPaginatedResults(int maxResults) permet de définir le nombre maximum de résultats renvoyés pour les résultats paginés
- int getMaxPaginatedResults( ) permet de récupérer le nombre maximum de résultats renvoyés pour les résultats paginés
- void setNumPerPage(int numPerPage)
- int getNumPerPage( )

Pour utiliser les limites par défaut des services de données dans une application, la propriété correspondante de DataContext doit être définie sur **DsConstants.USE_DEFAULT**. Les valeurs par défaut sont les suivantes :

- maxResults : 125
- maxPaginatedResults : 125
- numPerPage : 25

Si l'application ne spécifique pas de valeur pour numPerPage, la valeur figurant dans le fichier prefs.xml est utilisée. Si aucune valeur n'est indiquée par l'application ou dans le fichier prefs.xml, la valeur -1 est utilisée, ce qui signifie que la demande ne sera pas paginée.

En outre, les méthodes suivantes fournissent des limites pour les ensembles de résultats qui sont transmises directement à la base de données dans le cadre de la requête SQL. Comme Visual Modeler peut supprimer des résultats dans le cadre

de la vérification de la règle d'accès (par exemple, si l'utilisateur n'est pas autorisé à afficher certaines données), ces méthodes permettent de définir une limite plus élevée pour les ensembles de résultats.

- public void setDBResultLimit(int limit)
- public int getDBResultLimit( )

Vous pouvez également définir la préférence DataServices.General.LimitDBResults. Si la préférence LimitDBResults est définie sur true, les résultats sont limités automatiquement au nombre autorisé par MaxResults (ou par MaxPaginatedResults, pour les résultats paginés). Pour utiliser ce mécanisme, les règles d'accès doivent être exprimées en langage SQL. Pour les bases de données Oracle, ne définissez pas la préférence LimitDBResults sur true.

Il existe deux façons de gérer les règles d'accès. De nombreuses règles d'accès sont converties en clauses SQL WHERE appliquées à la requête. Cela permet à la base de données de gérer la règle d'accès. Si la règle est trop complexe (par exemple, si elle repose sur une hiérarchie de partenaires), la règle d'accès peut être appliquée uniquement lors du traitement des résultats de la base de données. Ce type de règles ne peuvent pas être converties en SQL.

Dans certains cas, lorsque vous utilisez Oracle, la génération SQL requiert que les alias de colonne soient définis dans le schéma XML. Cela est nécessaire uniquement si la requête regroupe plusieurs tables utilisant le même nom de colonne. Cette opération n'est pas nécessaire si vous utilisez SQL Server ou DB2.

Comment instancier une instance DataContext ?

Actuellement, vous pouvez instancier une nouvelle instance DataContext à l'aide du mécanisme standard "new" :

```
DataContext dc = new DataContext();
```

Quels sont les paramètres par défaut de la nouvelle instance DataContext ?

Lorsque vous appelez "new DataContext( )", les valeurs par défaut suivantes sont affectées aux attributs :

Attribut
: **Valeur par défaut**

doAccessCheck
: true

maxResults
: Propriété DataServices.xml maxResults

numPerPage
: Propriété DataServices.xml numPerPage

CacheId
: null

doAccessCheck
: true

Beans de données de liste

Les *beans de données de liste* et les *objets métier de liste* sont une classe spéciale d'objets métier. Ces classes sont utilisées pour gérer une liste d'objets métier du même type. Chaque fois qu'un élément d'objet de données est déclaré avec l'ordinalité "n" dans un élément Recipe, un bean de données de liste est créé. Les autorisations d'accès continuent d'être gérées au niveau de l'objet métier individuel.

Remarque : Dans les versions antérieures, l'ordinalité des objets de données était spécifiée dans le fichier de définition de l'objet de données. L'ordinalité est désormais déterminée par le fichier de recette. Dans la version 6.0, les objets de données continuent d'utiliser l'attribut d'ordinalité pour déclarer les objets de données enfants, de référence et les objets de données inclus.

En général, vous n'avez pas à créer de beans de données pour les objets de données de liste, car ils sont créés automatiquement. Pour plus d'informations, voir «Classes DataBean», à la page 23. Ils prennent en charge les méthodes générées automatiquement qui renvoient une liste d'objets de données. Par exemple, le fragment de code suivant montre comment restaurer une liste d'utilisateurs. L'objet DataContext identifié par "context" et l'objet DsQuery identifié par "query" sont utilisés pour restreindre les utilisateurs renvoyés par la méthode *restore()* :

```
UserListBean userList = (UserListBean)
OMWrapper.getObject("com.comergent.bean.simple.UserListBean");
// Restaurer la liste.
userList.restore(context, query);
// Renvoyer immédiatement si aucun résultat n'est trouvé.
if (userList.getUserCount() == 0)
{
return;
}
// Au moins un utilisateur dans la liste, donc parcourir la liste
d'utilisateurs
ListIterator userIterator = userList.getUserIterator();
while (userIterator.hasNext())
{
UserBean user = (UserBean) userIterator.next();
// Exécuter les logiques métier requises sur chaque utilisateur.
}
```

Remarque : Les paramètres DataContext et DsQuery sont utilisés dans la méthode *restore()* pour gérer l'exécution de la requête par rapport à Knowledgebase.

Beans entity, application et présentation

Les principaux types de beans de données utilisés dans Visual Modeler sont les beans de données, les beans application, les beans entity et les beans présentation. Cette section explique les différences entre les différents types de beans.

- Les beans de données sont des classes Java créées automatiquement à partir de la description du schéma XML des objets métier. L'exécution de la cible SDK generateBean permet de générer le code source pour chaque bean de données. Ces beans composent le paquet com.comergent.bean.simple.

Lorsque cela est possible, vous devez utiliser la commande *instanceof* pour déterminer la classe d'un bean de données plutôt que d'interroger le type d'objet métier.

- Les beans application sont des classes Java créées pour ajouter des fonctions non prises en charge par les beans standard. Par exemple, un bean application peut fournir des méthodes supplémentaires ne pouvant pas être générées automatiquement ou combiner deux ou plusieurs beans standard afin d'envoyer des données à une page JSP. Les beans application sont organisés par application et se trouvent dans le paquet com.comergent.apps.<*nom_application*>.bean de chaque application.

  Les beans application peuvent être des sous-classes de beans standard, mais sont le plus souvent des classes Java contenant un ou plusieurs beans standard en tant que variables de membre.

  Par exemple, la classe de bean application com.comergent.appservices.productService.productMgr.BizProductBean est une classe Java contenant une variable de membre qui implémente l'interface com.comergent.bean.simple.IDataProduct. La classe de bean application BizProductBean délègue des méthodes telles que *getProductID()* à la variable de membre com.comergent.bean.simple.IDataProduct et fournit en outre des méthodes pour récupérer les fonctions du produit, sa chaîne de remplacement et les prix associés. Notez que l'interface IDataProduct est utilisée au lieu de la classe ProductDataBean : il s'agit ici d'un exemple typique d'utilisation d'une interface générée au lieu d'une classe. Pour plus d'informations sur la génération et l'utilisation de ces interfaces, voir Chapitre 12, «Interfaces générées», à la page 77.

  Par convention, si vous créez un bean application pour encapsuler un bean de données, vous devez fournir une méthode appelée *getDataBean()* qui permet de récupérer le bean de données.
- Les beans présentation permettent également d'envoyer des données aux pages JSP ; ils se distinguent des beans application par le fait qu'ils ne fournissent pas de logique métier. Ils peuvent regrouper plusieurs beans de données en une seule classe pour en simplifier l'utilisation ou fournir des informations de mise en forme. Comme les beans application, les beans présentation doivent fournir une méthode permettant d'accéder au bean de données sous-jacent. Par exemple, l'interface IPresProduct fournit la méthode *getIRdProduct()* qui renvoie l'interface IRdProduct ; si nécessaire, vous pouvez la transmettre au bean de données sous-jacent ou au bean de données étendu.
- Les beans entity étaient utilisés dans les éditions précédentes de Visual Modeler. Ils jouaient le même rôle que les beans application. Leur utilisation est désormais obsolète.

Utilisation de procédures stockées

Vous pouvez utiliser des procédures stockées pour restaurer des objets de données. Le nom de la procédure stockée est déclaré dans l'élément ExternalName de l'objet de données.

Lorsque vous définissez des objets de données, veillez à inclure l'attribut SourceType. Cet attribut peut prendre les valeurs suivantes :

- "1" : La source de données sous-jacente est une table. Il s'agit de la valeur par défaut.
- "2" : La source de données sous-jacente est une procédure stockée.

Si l'attribut SourceType n'est pas défini, on considère par défaut que la source sous-jacente de l'objet de données est une table.

Méthodes de beans de données

En général, vous devez utiliser les interfaces générées fournies par chaque bean de données ; celles-ci organisent les méthodes d'accès et les méthodes de données pour vous aider à gérer l'accès aux objets de données au cours de leur cycle de vie. Pour plus d'informations, voir Chapitre 12, «Interfaces générées», à la page 77.

Utilisez le mécanisme de sécurité de la règle d'accès pour effectuer un contrôle d'accès.

Méthodes IData

L'interface IData fournit les principales méthodes suivantes :

- *copyBean()* : Cette méthode est utilisée pour copier les valeurs des zones de données d'un bean vers un autre. Elle prend comme argument un bean qui est une instance de la même classe ou une sous-classe du bean qui appelle la méthode.
- *delete()* : Cette méthode marque l'objet de données correspondant comme supprimé ; la colonne ACTIVE_FLAG correspondante de la table de base de données à l'objet de données est définie sur "N" lorsque l'objet est persisté. Notez que vous devez appeler la méthode *persist()* après la méthode *delete()* ; dans le cas contraire, la suppression ne prend pas effet.
- *erase()* : Cette méthode supprime l'enregistrement de base de données correspondant à l'objet métier. Notez que la suppression d'enregistrements des tables de base de données peut entraîner des problèmes d'intégrité des données si d'autres tables font référence aux clés que vous avez supprimées. En général, vous devez utiliser cette méthode uniquement après avoir pris en compte l'ensemble des utilisations de l'enregistrement et de ses clés et si vous pouvez supprimer les enregistrements correspondants des autres tables.
- *generateKeys()* : Cette méthode remplit les zones clés du bean de données. Elle peut être appelée sans appeler la méthode *persist()*. Vous pouvez utiliser les clés générées à l'aide de cette méthode pour créer d'autres objets qui requièrent ces clés.
- *setDataContext()* : Cette méthode définit le contexte de données afin que les méthodes *restore()* et *persist()* puissent utiliser des valeurs appropriées pour des paramètres tels que le nombre de résultats par page dans un ensemble de données paginé. Pour plus d'informations sur la classe DataContext, voir «DataContext», à la page 48.
- *persist()* : Cette méthode enregistre les données du bean de données dans sa source de données.
- *prune()* : Cette méthode est utilisée pour marquer le bean pour suppression dans la mémoire. L'appel de la méthode *restore()* après *prune()* n'a aucun effet sur la source de données sous-jacente du bean.
- *restore()* : Cette méthode récupère les données du bean de données à partir de sa source de données. Pour plus d'informations sur l'utilisation de la classe DataContext dans la méthode *restore()*, voir «DataContext», à la page 48.
- *update()* : Cette méthode met à jour l'enregistrement de base de données correspondant à cet objet métier.

Notez que l'appel d'une méthode entraînant une modification d'état doit être suivi de l'appel de la méthode *persist()* afin d'appliquer la modification à l'enregistrement de base de données.

L'interface IData fournit également les méthodes *isRestorable()* et *isPersistable()*, qui permettent de vérifier si un objet de données peut être restauré ou persisté.

Méthodes des interfaces IRd et IAcc

L'interface IRd fournit les méthodes d'accès read-only aux zones de l'objet de données. L'interface IAcc étend l'interface IRd en ajoutant les méthodes d'accès set à chaque zone de données. La distinction entre ces deux interfaces permet de transmettre un objet en lecture seule à une application client ou à une page JSP.

Par exemple, supposons que le fichier de l'objet de données Condition, appelé **Condition.xml**, contient un élément DataField défini comme suit :

```
<DataField Name="ControlType"
Writable="y" Mandatory="y"
ExternalFieldName="CONTROL_TYPE"/>
```

Dans l'interface IRdCondition générée automatiquement, la méthode suivante est appelée :

public Long getControlType()

Dans l'interface IAccCondition générée automatiquement, la méthode suivante est appelée :

```
public void setControlType(Long value) throws ICCException
```

Les signatures de ces méthodes d'accès sont déterminées par la définition de l'élément DataElement correspondant dans le fichier **DsDataElements.xml** :

```
<DataElement Name="ControlType" DataType="LONG"
Description="Condition Control Type" MaxLength="20" />
```

Remarque : Si l'attribut Writable d'une zone de données est défini sur "n", la méthode *set*DataField*()* correspondante n'est pas générée.

Restauration et persistance des données

Les principales opérations que vous pouvez effectuer sur un objet de données sont les suivantes : *delete*(), *persist()* et *restore()*.

- En appelant la méthode *delete()* pour un objet de données, vous marquez celui-ci comme supprimé et il ne pourra pas être récupéré par une autre application. Sa valeur dans la colonne ACTIVE_FLAG de la table de base de données sous-jacente est définie sur "N". Notez que les données de l'objet de données ne sont pas supprimées de la source de données. Si la table de base de données de l'objet de données ne comporte pas de colonne ACTIVE_FLAG, n'utilisez pas la méthode *delete()*. Vous pouvez toutefois utiliser la méthode *erase()* pour supprimer les objets de données de Knowledgebase.
- Lorsque vous effectuez une opération *persist* sur un bean de données, Visual Modeler enregistre les données stockées dans l'arborescence DsElement d'un élément de données dans sa ou ses sources de données externes. Notez que

méthode *persist()* sert à la fois à gérer la mise à jour des objets de données existants et à créer de nouveaux objets de données.

- Lorsque vous effectuez une opération *restore* sur un bean de données ou un objet métier, Visual Modeler récupère ses données à partir de sa ou ses sources de données externes. Si aucun objet de requête n'est spécifié dans la méthode *restore()*, Visual Modeler restaure tous les objets de données qui ont des valeurs de zones clés identiques aux valeurs du bean de données.
  - Notez que si vous appelez la méthode *restore()* sur un bean de données autre qu'un bean de liste, vous devez prendre en considération le fait que ses données ne peuvent être récupérées qu'à partir des valeurs définies dans ses zones clés. Lorsque la méthode *restore()* est lancée, aucun contrôle n'est effectué pour vérifier qu'un seul enregistrement est récupéré ; le bean de données sera rempli avec le premier enregistrement récupéré. Si aucun enregistrement n'est récupéré, l'appel de la méthode *restore()* lance une exception ICCException.
  - En général, lorsque vous appelez la méthode *restore()* sur un bean de données de liste, vous devez spécifier la classe DsQuery. Si cette classe est spécifiée, le bean de données de liste restauré contiendra tous les beans de données de ce type.

Méthode DataBean restore()

Cette section décrit les principales formes de la méthode DataBean *restore()*.

```
public void restore(DataContext dataContext, DsQuery dsQuery)
```

Il s'agit de la principale forme de la méthode *restore()*. Utilisez le paramètre dsQuery pour indiquer la requête qui doit être exécutée par l'opération restore. Le paramètre dataContext détermine le nombre maximum d'objets renvoyés et, pour la pagination, le nombre de résultats par page. Utilisez le paramètre dataContext pour vérifier si l'utilisateur actuel dispose des autorisations nécessaires pour exécuter cette opération. Par défaut, un contrôle d'accès est effectué ; si vous ne souhaitez pas effectuer de contrôle d'accès, vous devez remplacer ce paramètre à l'aide de la méthode *disableAccessCheck()*.

public void restore(DataContext dataContext)

Cette méthode est équivalente à la méthode *restore(dataContext, null)*.

Voici un exemple d'utilisation conjointe des classes DataContext et DsQuery pour gérer l'appel de la méthode *restore()* :

```
try
{
DataContext dataContext = new DataContext();
if (doAccessCheck == true)
{
dataContext.enableAccessCheck();
}
else
{
dataContext.disableAccessCheck();
}
dataContext.setNumPerPage(pageSize);
DsQuery dsQuery = QueryHelper.newWhereClause("PartnerKey",
DsConstants.EQUALS, partnerKey);
LightWeightPartnerBean partnerBean =
```

```
(com.comergent.bean.simple.LightWeightPartnerBean)
com.comergent.dcm.util.OMWrapper.getObject(
"com.comergent.bean.simple.LightWeightPartnerBean");
partnerBean.restore(dataContext, dsQuery);
QueryHelper.freeQuery(dsQuery);
return partnerBean;
}
catch (ICCException e)
{
throw (new ProfileMgrException(e));
}
```

Méthode DataBean persist()

Cette section décrit les principales formes de la méthode DataBean *persist()*.

```
public void persist(DataContext dataContext)
```

Si dataContext indique qu'un contrôle d'accès est requis, cette forme de la méthode *persist()* effectue un contrôle d'accès avant d'exécuter l'opération. Si l'utilisateur ne dispose pas des autorisations nécessaires, l'opération n'est pas exécutée.

Autres méthodes

méthode getBizObj()

Pour récupérer une représentation d'objet métier de l'objet de données et de ses données, vous pouvez appeler la méthode *getBizObj()*. Cette méthode est utile lorsque vous souhaitez afficher la structure interne de l'objet. Par exemple :

```
BusinessObject bo = bean.getBizobj();
ComergentDocument doc = bo.serializeToXml();
doc.prettyPrint();
```

Cette méthode est désormais obsolète.

méthode writeExternal()

Cette méthode permet d'écrire une représentation XML du bean de données et de ses données.

Objets de données enfants

La plupart des objets de données déclarent des objets de données enfants à l'aide de l'élément ChildDataObject. Par exemple, l'objet de données ShoppingCart déclare l'objet de données enfant LineItem comme suit :

```
<DataObject Name="ShoppingCart" Extends="C3PrimaryRW"
ExternalName="CMGT_CARTS" ObjectType="JDBC" Version="6.0">
...
<ChildDataObject Access="RWID" Name="LineItem">
<Relationship CascadeDelete="y" CascadeErase="n"
ChangeUpdatesParent="y">
<JoinKeys>
<JoinKey DstJoinField="ShoppingCartKey"
SrcJoinField="ShoppingCartKey"/>
</JoinKeys>
</Relationship>
```

```
</ChildDataObject>
...
</DataObject>
```

L'élément Relationship comporte des attributs qui indiquent comment gérer les objets enfants lorsque le parent est mis à jour et si le parent doit être mis à jour en cas de modification de l'enfant. Les éléments JoinKey indiquent comment restaurer les objets de données enfants ; en général, ils indiquent comment utiliser les valeurs de l'objet de données parent pour définir des valeurs dans l'objet de données enfant.

Lors de la génération du bean de données parent, la méthode *get*ChildDataObject*Iterator()* est générée, qui renvoie un objet ListIterator contenant les beans de données enfants. En effectuant une itération sur les objets, vous pouvez examiner individuellement chaque bean de données enfant et accéder à ses zones de données à l'aide des méthodes d'accès standard.

Par exemple, la classe ShoppingCartBean prend en charge la méthode getLineItemIterator(). Les lignes de code suivantes montrent comment récupérer une zone de ligne article :

```
/*
shoppingCartBean est un objet qui a été déjà restauré.
*/
ListIterator lineItemIterator =
shoppingCartBean.getLineItemIterator();
LineItemBean lineItemBean =
(LineItemBean) lineItemIterator.getLineItemBean(0);
Long quantity = lineItemBean.getQuantity();
```

Les objets de données enfants ne sont pas restaurés lors de la restauration d'un objet de données parent. Ils sont restaurés uniquement lorsque l'application accède aux enfants comme décrit ci-dessus.

Extension d'objets de données

Pourquoi et quand exécuter cette tâche

Dans toutes les implémentations de Visual Modeler, vous êtes souvent amené à ajouter des zones de données à des objets de données ou à créer des objets de données qui étendent des objets de données existants.

Il est recommandé de stocker ces informations complémentaires dans une nouvelle table de base de données. Vous devez alors définir un nouvel objet de données qui accède à la nouvelle table. Un nouvel objet de données DataObject est défini ; celui-ci étend l'objet de données DataObject en ajoutant le nouvel objet IncludeDataObject.

Par exemple, supposons que vous souhaitez ajouter une nouvelle zone à l'objet de données Commande pour suivre les commandes "hébergées" qui sont passées dans des boutiques en ligne partenaires. Le nouvelle zone correspond à la clé de partenaire de la boutique en ligne partenaire. Nous vous recommandons de procéder comme suit :

Procédure

1. Créez un nouvel objet de données appelé HostedPartner comportant seulement deux zones : OrderKey et PartnerKey. Configurez-le pour faire référence à une table à deux colonnes appelée CMGT_ORDER_X_PARTNER (composée des colonnes ORDER_KEY et PARTNER_KEY).

```
<?xml version="1.0"?>
<DataObject Name="HostedPartner"
ExternalName="CMGT_ORDER_X_PARTNER" ObjectType="JDBC"
Version="6.0">
<KeyFields>
<KeyField Name="OrderKey" ExternalName="ORDER_KEY"/>
<KeyField Name="PartnerKey" ExternalName="PARTNER_KEY"/>
</KeyFields>
<DataFieldList>
<DataField Name="OrderKey" ExternalFieldName="ORDER_KEY"
Mandatory="n" Writable="y"/>
<DataField Name="PartnerKey"
ExternalFieldName="PARTNER_KEY"
Mandatory="n" Writable="y"/>
</DataFieldList>
</DataObject>
```

2. Créez un nouvel objet de données appelé HostedOrder qui étend l'objet Order. Le fichier **HostedOrder.xml** se présente comme suit :

```
<?xml version="1.0"?>
<DataObject Name="HostedOrder" Extends="Order" ObjectType="JDBC"
Version="6.0">
<IncludedDataObject Access="RWID" Name="HostedPartner"
Ordinality="1">
<Relationship CascadeDelete="y" CascadeErase="n"
ChangeUpdatesParent="y">
<JoinKeys>
<JoinKey DstJoinField="OrderKey"
SrcJoinField="OrderKey"/>
</JoinKeys>
</Relationship>
</IncludedDataObject>
</DataObject>
```

Trois approches de base peuvent être utilisées

3. Vous pouvez utiliser une extension pour ajouter facilement les nouvelles zones de données requises et remplacer le nom de la table. Cela permet d'ajouter l'ensemble des données dans une nouvelle table. Cette approche est particulièrement utile lorsque vous avez besoin des mêmes données, mais il vous faut un exemplaire distinct de ces données. (Vous pouvez, par exemple, stocker une image instantanée d'une commande avant qu'elle soit considérée comme une demande "hébergée".)
4. Vous pouvez étendre l'objet Order pour ajouter l'objet IncludedDataObject à HostedOrder, où HostedOrder définit uniquement les données supplémentaires à stocker dans une autre table. Cela signifie que les modifications apportées aux zones de données d'origine de l'objet Order seront toujours persistées dans la table Order, mais les informations supplémentaires pour HostedOrder seront persistées dans une autre table. Cette méthode recommandée est décrite dans l'exemple ci-dessus.
5. Vous pouvez définir un objet HostedOrder, en indiquant que l'objet Order correspond à IncludedDataObject. Cette approche produit le même résultat que

la deuxième méthode présentée ci-dessus. Cependant, cette approche pose un problème dans la mesure où l'objet HostedOrder n'étend pas l'objet Order et ne peut donc plus être considéré comme un objet Order par le code d'application.

Remarque : L'utilisation de deux tables présente un léger inconvénient en termes de performances, mais la requête est exécutée correctement. Par ailleurs, l'utilisation de deux tables peut réduire la redondance des données (en fonction de vos besoins).

Résultats

Si les références à l'extension client sont occasionnelles, vous pouvez utiliser un objet ChildDataObject pour tirer parti du mécanisme de lien différé.

Exemple de bean de données

Cette section explique comment définir et utiliser un objet de données. Supposons que vous souhaitez utiliser un objet de données pour représenter une demande standard d'un client. Vous devez fournir :

- l'adresse électronique du client
- la date d'envoi de la demande
- la date de réponse (facultatif)
- le contenu de la demande
- le contenu de la réponse (facultatif)
- l'ID produit du produit faisant l'objet de la demande (facultatif)

Création d'une définition d'objet de données

Pourquoi et quand exécuter cette tâche

Pour créer une définition d'objet de données, procédez comme suit :

Procédure

1. Créez l'élément d'objet métier Enquiry et ajoutez-le au fichier **DsBusinessObjects.xml**.

   ```
   <BusinessObject Name="Enquiry" Version="6.0"
   Description="Customer enquiry"/>
   ```

   Utilisez l'attribut Version pour gérer les versions d'objets métier pouvant être utilisées simultanément. Notez que l'attribut Version permet également de déterminer si un contrôle d'accès est effectué automatiquement (version 5.0 ou ultérieure).

2. Créez la recette de cet objet métier et ajoutez-la au fichier **DsRecipes.xml**.

   ```
   <Recipe Name="Enquiry" Version="6.0" Ordinality="n"
   Description="Customer enquiry">
   <DataObjectList>
   <DataObject Name="Enquiry"
   DataSourceName="ENTERPRISE" />
   </DataObjectList>
   </Recipe>
   ```

   L'attribut Name de la recette est sensible à la casse et doit correspondre exactement au nom de l'objet métier. Dans la version 9.1, la liste d'objets de données pour chaque recette peut contenir plusieurs objets de données, mais un seul objet de données peut être *accessible en écriture*. L'attribut de chaque

élément d'objet de données définit le nom de la source de données. Ces entrées déterminent les sources à partir desquelles sont récupérées les données de l'objet métier et la source vers laquelle il peut être persisté.

3. Créez un fichier appelé **Enquiry.xml** pour définir l'objet de données. Le nom de l'élément d'objet de données est sensible à la casse et doit correspondre exactement à l'attribut Name défini dans l'entrée DataObject de l'élément recipe.

   Dans l'exemple ci-après, les données de ces objets de données sont stockées dans une table de base de données appelée CMGT_ENQUIRY et l'attribut ExternalFieldName de chaque élément DataField indique quelle colonne utiliser pour récupérer la valeur DataField. Par exemple, la colonne EMAIL_ADDRESS de la table CMGT_ENQUIRY contient la valeur d'adresse e-mail associée à une demande.

```
<?xml version="1.0"?>
<DataObject Name="Enquiry" Extends="C3PrimaryRW" Version="6.0"
ExternalName="CMGT_ENQUIRY"
Access="R" ObjectType="JDBC">
<KeyFields>
<KeyField Name="Key" ExternalName="ENQUIRY_KEY"/>
</KeyFields>
<DataFieldList>
<DataField Name="EnquiryKey"
Writable="n" Mandatory="y"
ExternalFieldName="ENQUIRY_KEY"/>
<DataField Name="EmailAddress"
Writable="n" Mandatory="y"
ExternalFieldName="EMAIL_ADDRESS"/>
<DataField Name="EnquiryDate"
Writable="n" Mandatory="y"
ExternalFieldName="ENQUIRY_DATE"/>
<DataField Name="ResponseDate"
Writable="n" Mandatory="n"
ExternalFieldName="RESPONSE_DATE"/>
<DataField Name="TimeToRespond"
Writable="n" Mandatory="n"/>
<DataField Name="EnquiryContent"
Writable="n" Mandatory="y"
ExternalFieldName="ENQUIRY_CONTENT"/>
<DataField Name="ResponseContent"
Writable="y" Mandatory="n"
ExternalFieldName="RESPONSE_CONTENT"/>
<DataField Name="SKU"
Writable="n" Mandatory="n"
ExternalFieldName="SKU"/>
</DataFieldList>
</DataObject>
```

   Notez que la définition de la zone de données TimeToRespond ne comporte pas d'attribut ExternalFieldName car cet attribut ne correspond pas à une colonne de la base de données. Les valeurs de cette zone sont calculées lors de l'exécution et sont définies dans le bean EnquiryBean pour pouvoir être affichées.

4. Définissez les éléments DataElement Enquiry et EnquiryList dans le fichier **DsDataElements.xml** en procédant comme suit :

```
<DataElement Name="Enquiry" Description="Enquiry"
DataType="HEADER"/>
<DataElement Name="EnquiryList" Description="Enquiry list"
DataType="LIST"/>
```

5. Définissez un élément DataElement pour chaque zone de données du fichier **DsDataElements.xml**. Les éléments DataElement fournissent des informations sur le type de données utilisées par DataManager pour récupérer ou enregistrer des données pour ce type d'objet métier. Par exemple :

   ```
   <DataElement Name="EnquiryKey" LongName="Enquiry Key"
   DataType="LONG" MaxLength="20" />
   <DataElement Name="EnquiryDate" LongName="Enquiry Date"
   DataType="DATE" />
   <DataElement Name="ResponseDate" LongName="Date de réponse"
   DataType="DATE" />
   <DataElement Name="EnquiryContent" LongName="Enquiry content"
   DataType="STRING" MaxLength="256" />
   <DataElement Name="ResponseContent" LongName="Contenu de la réponse"
   DataType="STRING" MaxLength="256" />
   ```

   Notez que les zones EmailAddress et SKU ne comportent pas d'élément DataElement, car les éléments DataElement sont déjà définis pour ces zones. Vous pouvez réutiliser les éléments DataElement autant de fois que vous voulez, tant que le type de données reste identique dans chaque occurrence.

6. Créez des entrées dans le fichier **ObjectMap.xml** pour ce bean de données. Par exemple :

   ```
   <Object ID="com.comergent.bean.simple.EnquiryBean">
   <ClassName>com.comergent.bean.simple.EnquiryBean</ClassName>
   </Object>
   <Object ID="com.comergent.bean.simple.IRdEnquiry">
   <ClassName>com.comergent.bean.simple.EnquiryBean</ClassName>
   </Object>
   <Object ID="com.comergent.bean.simple.IAccEnquiry">
   <ClassName>com.comergent.bean.simple.EnquiryBean</ClassName>
   </Object>
   <Object ID="com.comergent.bean.simple.IDataEnquiry">
   <ClassName>com.comergent.bean.simple.EnquiryBean</ClassName>
   </Object>
   ```

7. Enfin, définissez un élément de source de données qui correspond à l'attribut DataSourceName défini dans l'élément DataObject. Cette source de données est définie dans le fichier DsDataSources.xml dans le cadre du schéma. Dans la plupart des cas, cette source de données est déjà définie ; vous devez la définir vous-même uniquement si vous utilisez une base de données distincte ou une source de données autre que Knowledgebase. Par exemple :

   ```
   <DataSource Name="ENTERPRISE" Version="2.0">
   <Primary Type="SQL" DataService="JdbcService"
   SubType="ORACLE"
   ConnectionString="jdbc:<pilote>:<serveur>:<port>:<IDsource>"
   UserId="IDutilisateur" Password="motdepasse" />
   <Alternate Type="SQL" DataService="JdbcService"
   SubType="MSSQL"
   ConnectionString="jdbc:<pilote>:<serveur>:<port>:<IDsource>"
   UserId="IDutilisateur" Password="motdepasse" />
   </DataSource>
   ```

L'attribut DataService des éléments Primary et Alternate déterminent la classe utilisée pour traiter les méthodes *restore()* et *persist()* du bean EnquiryBean. Les autres attributs déterminent les méthodes d'accès à la source externe.

8. Exécutez la cible SDK generateBean pour générer le code source des nouveaux beans de données EnquiryBean et EnquiryListBean, ainsi que leurs interfaces correspondantes. Pour plus d'informations sur ces interfaces, voir Chapitre 12, «Interfaces générées», à la page 77.

Résultats

Vous pouvez maintenant utiliser le bean de données Enquiry et ses interfaces dans les classes de logique métier. Pour créer une instance du bean de données Enquiry, vous devez appeler une méthode de la forme suivante :

OMWrapper().getObject("com.comergent.bean.simple.EnquiryBean")

Cette méthode renvoie le bean de données EnquiryBean, dont la structure est indiquée dans l'objet de données Enquiry. Une fois que vous avez créé l'instance de la classe QueryBean, remplissez ses zones clés et restaurez le bean pour récupérer ses données :

```
int queryIndex = 0;
try
{
String queryKey = request.getParameter("querykey");
queryIndex = Integer.parseInt(queryKey);
}
catch (Exception e)
{
//Lancer une exception si le paramètre n'est pas valide
}
QueryBean queryBean = (QueryBean)
OMWrapper().getObject("com.comergent.bean.simple.EnquiryBean");
queryBean.setKey(queryIndex);
queryBean.restore();
```

Pour récupérer une liste d'enquêtes, procédez comme suit :

```
// Utiliser les valeurs par défaut pour les paramètres DataContext
DataContext context = new DataContext();
// Récupérer les enquêtes relatives à un ID produit spécifique, MXWS-7000
DsQuery query =
QueryHelper.newWhereClause("SKU", DsQueryOperators.EQUALS,
"MXWS-7000");
EnquiryListBean enquiryList = (EnquiryListBean)
OMWrapper.getObject("com.comergent.bean.simple.EnquiryListBean");
// Restaurer la liste.
enquiryList.restore(context, query);
// Parcourir la liste...
ListIterator enquiryIterator = enquiryList.getEnquiryIterator();
while (enquiryIterator.hasNext())
{
boolean isModified = false;
EnquiryBean enquiry = (EnquiryBean) enquiryIterator.next();
// Traiter chaque demande
}
```

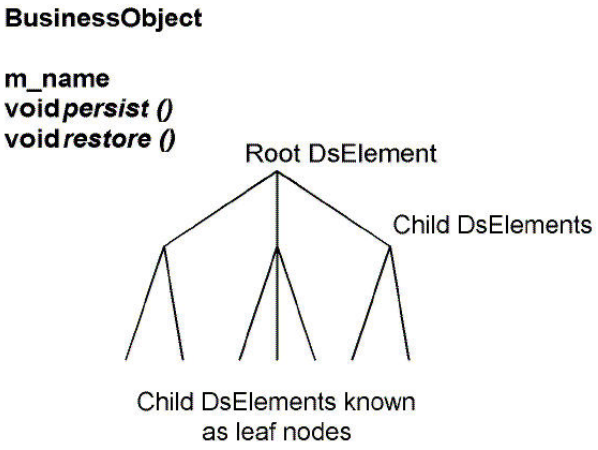
En général, vous devez vous assurer que les applications qui utilisent le bean EnquiryBean utilisent l'une des interfaces générées et non le bean de données lui-même. Les interfaces générées servent à séparer l'implémentation de l'objet de données de son interface, ce qui vous permet de gérer la façon dont l'application accède aux données de l'objet. Pour récupérer l'instance d'une classe qui implémente l'interface IAccEnquiry, vous pouvez utiliser les lignes suivantes :

```
IAccEnquiry temp_IAccEnquiry = (IAccEnquiry)
OMWrapper.getObject("com.comergent.bean.simple.IAccEnquiry");
```

Arborescence DsElement

Cette section présente les méthodes permettant de récupérer les métadonnées des beans de données. Elle décrit également l'arborescence DsElement utilisée pour stocker des données dans les objets de données et les classes d'objet métier. Cette arborescence est présentée uniquement pour illustrer le fonctionnement des applications existantes ; les nouvelles applications qui utilisent des classes de bean de données ne sont pas concernées.

Les objets de données sont créés en tant qu'objets de classes de bean de données. Le contenu de chaque objet de données est organisé sous la forme d'une arborescence de composants appelés éléments DsElements (voir «Éléments DsElement»). Le contenu de ces éléments est récupéré à partir de systèmes externes à l'aide du schéma XML ainsi que des recettes et des sources de données définies dans le schéma XML. La figure suivante présente l'objet métier :



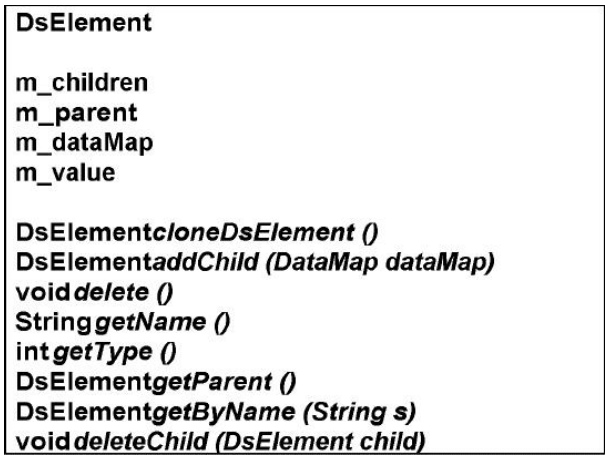
.

Lorsque DataManager crée un bean de données ou un objet métier, il détermine la structure de son arborescence DsElement à l'aide du schéma XML . L'arborescence DsElement est la représentation Java de la structure de l'objet métier. Le schéma détermine également les types de données pouvant être insérés au niveau des noeuds terminaux ainsi que les éventuelles contraintes liées aux valeurs du noeud. Vous pouvez accéder à l'arborescence DsElement en appelant la méthode d'objet métier *getRootElement()*.

Éléments DsElement

Chaque élément DsElement contient des données et un format de page qui définit le mappage entre ses données et sa source de données. Un élément DsElement peut être l'enfant d'un autre élément DsElement (ce dernier sera donc son *parent*). Une arborescence DsElement est une collection d'éléments DsElement dans laquelle tous les éléments, à l'exception d'un seul, ont comme parent un autre élément de

l'arborescence. Par définition, l'élément DsElement ayant un parent null est le DsElement *racine*. La figure suivante présente les méthodes des éléments DsElement :



La classe DsElement fournit plusieurs méthodes permettant de prendre en charge la navigation dans une arborescence DsElement, notamment la méthode *children()*, qui renvoie un élément itératif des éléments DsElement enfants d'un élément DsElement donné. Outre *getRootElement()*, la classe d'objet métier fournit également la méthode *getElementByName()* pour accéder directement à un élément DsElement donné de l'arborescence.

Tous les éléments DsElement portant le même nom, par exemple nom_enfant, et qui sont les enfants du même élément DsElement doivent avoir un parent appelé <nom_enfant>List. Pour identifier ces éléments dans le schéma XML, leur ordinalité est définie sur "n" par opposition à "1". Les enfants d'un élément DsElement sont conservés dans un vecteur appelé m_children.

Les principales méthodes de la classe DsElement sont les suivantes :

- *addChild()* : Ajoute un nouvel élément DsElement défini par le format de page de cet élément DsElement.
- *cloneDsElement()* : Renvoie une copie de cet élément DsElement.
- *delete()* : Définit DsElemState sur DsElemState.DELETED.
- *deleteChild()* : Supprime un enfant du vecteur m_children en le définissant en tant qu'élément DsElement.
- *getName()* : Renvoie le nom de l'élément tel que défini par ses métadonnées.
- *getParent()* : Renvoie le parent de cet élément DsElement.
- *getType()* : Renvoie le type de l'élément tel que défini par son format de page.

`Métadonnées des éléments DsElement

Il est parfois utile de récupérer des informations sur une zone de données et sur son élément DsElement sous-jacent. Pour ce faire, vous pouvez utiliser la méthode *getMetaData(String elementName)* de l'interface IData. Elle permet de renvoyer un objet qui implémente l'interface IMetaData. Cette interface prend en charge les méthodes suivantes :

- public int getDataType() : Renvoie les valeurs définies dans DsDataTypes
- public long getMaxLength() : Renvoie la longueur maximale en octets
- public long getMaxCharLength(Locale locale) : Renvoie la longueur maximale en caractères

- public Object getMinValue() : Renvoie la valeur minimale autorisée (ou la valeur null si aucune valeur minimale n'est définie)
- public Object getMaxValue() : Renvoie la valeur maximale autorisée (ou la valeur null si aucune valeur maximale n'est définie)
- public int getCountAllowedValues()
- public ListIterator getAllowedValueIterator()
- public Object getDefaultValue()

Remarque : Étant donné que chaque classe DataBean générée implémente l'interface IData, ces méthodes sont disponibles pour l'ensemble des beans de données générés.

Méthodes BusinessObject

L'utilisation des objets métier est obsolète. Cette section fournit des informations sur quelques méthodes BusinessObject à titre de référence uniquement.

Méthode BusinessObject restore()

Cette section décrit les principales formes de la méthode BusinessObject *restore()*.

```
public void restore(BusinessObject queryObj, int maxResults,
boolean accessCheck)
```

Il s'agit de la principale forme de la méthode *restore()*. Utilisez le paramètre queryObj pour indiquer la requête qui doit être exécutée par l'opération restore. Le paramètre maxResults détermine le nombre maximum d'objets renvoyés. Utilisez le paramètre accessCheck pour vérifier si l'utilisateur actuel dispose des autorisations nécessaires pour exécuter cette opération. Une fois le contrôle d'accès terminé, la méthode *restore(BusinessObject queryObj, int maxResults)* est appelée.

public void restore(BusinessObject queryObj, int maxResults)

Cette méthode appelle la méthode *restore() restore(this, queryObj, maxResults, false)* de l'objet de données sous-jacent.

public void restore(BusinessObject queryObj)

Cette méthode est équivalente à la méthode *restore(queryObj, 0)*.

```
public void restore()
```

Cette forme de la méthode appelle la méthode *restore(null, 0)*.

Méthode BusinessObject persist()

Cette section décrit les principales formes de la méthode BusinessObject *persist()*.

```
public void persist(boolean synch, boolean commit,
boolean accessCheck)
```

Les paramètres booléens déterminent respectivement si l'opération persist est synchronisée, si elle être validée dans la source de données sous-jacente et si un contrôle d'accès doit être effectué avant l'opération.

```
public void persist(boolean synch, boolean commit)
```

Cette forme de la méthode est équivalente à *persist(synch, commit, false)* pour les objets métier dont l'attribut Version correspond à 4.0 ou à une version antérieure. Elle est équivalente à *persist(synch, commit, true)* pour les objets métier dont l'attribut Version correspond à 5.0 ou à une version ultérieure.

```
public void persist()
```

Cette forme de la méthode appelle *persist(false, true)*.

La classe BusinessObject fournit également les méthodes suivantes :

- *delete()* : Vide l'objet métier en supprimant son arborescence DsElement.
- *getRootElement()* : Renvoie l'élément racine de l'arborescence DsElement.
- *getType()* : Renvoie le nom de l'élément racine de l'arborescence DsElement. Il s'agit du type de l'objet métier.
- *setRootElement()* : Définit l'élément racine de cet objet métier.

Chapitre 9. Connexion à Visual Modeler

Présentation de la consignation dans Visual Modeler

Le mécanisme de consignation fourni par Visual Modeler permet aux développeurs d'applications de consigner les activités dans Visual Modeler. Ce mécanisme est utilisé par l'API log4j et les fichiers de configuration **log4j.properties** pour configurer le comportement de consignation. La fonction de consignation permet également de prendre en charge l'audit des modifications apportées aux objets de données. Pour plus d'informations, voir «Audit des modifications apportées aux objets de données», à la page 68.

L'API log4j fournit une infrastructure de consignation souple et extensible pour gérer le comportement de consignation de Visual Modeler. Cette section explique comment utiliser cette infrastructure lors de la personnalisation et l'extension de Visual Modeler.

Notez que cette infrastructure remplace la précédente infrastructure utilisée par Visual Modeler, qui était basée sur la classe Global et ses méthodes *log*Level*()*. Leur utilisation est désormais obsolète.

Pour utiliser l'API log4j, vous devez créer une classe Logger dans chaque fichier de classe. Pour ce faire, entrez les lignes suivantes :

```
private static final org.apache.log4j.Logger log =
org.apache.log4j.Logger.getLogger(NomClasse.class);
```

Pour appeler une entrée de journal :

```
log.info("Ceci est une entrée de journal");
```

La méthode appelée dépend du niveau de consignation que vous souhaitez utiliser pour enregistrer le message. Vous pouvez utiliser les méthodes suivantes :

- *debug()*
- *error()*
- *fatal()*
- *info()*
- *warning()*

Vous pouvez également utiliser *log(priority, message)*, mais les méthodes répertoriées ci-dessus devraient être suffisantes.

Propriété système log4j.debug

En définissant la propriété système log4j.debug sur true, vous pouvez exécuter une commande echo sur les paramètres de journal actuels. Par exemple, ajoutez les lignes suivantes dans le script de démarrage du conteneur de servlet :

```
-Dlog4j.debug=true
La sortie de consignation suivante devrait s'afficher au démarrage :
log4j: Trying to find [log4j.xml] using context classloader
sun.misc.Launcher$AppClassLoader@136228.
log4j: Trying to find [log4j.xml] using sun.misc.Launcher$AppClassLoader@136228 class loader.
log4j: Trying to find [log4j.xml] using ClassLoader.getSystemResource().
```

```
log4j: Trying to find [log4j.properties] using context classloader
sun.misc.Launcher$AppClassLoader@136228.
log4j: Using URL [jar:file:/home/hle/ws/32-cmgt-modules/modules.cryptography-
tool/target/cmgt-cryptography-tool-2.0.0-SNAPSHOT-app.jar!/log4j.properties]
for automatic log4j configuration.
log4j: Reading configuration from URL jar:file:/home/hle/ws/32-cmgt-modules/modules.
cryptography-tool/target/cmgt-cryptography-tool-2.0.0-SNAPSHOT-app.jar!/log4j.properties
log4j: Parsing for [root] with value=[WARN, A1].
log4j: Level token is [WARN].
log4j: Category root set to WARN
log4j: Parsing appender named "A1".
log4j: Parsing layout options for "A1".
log4j: Setting property [conversionPattern] to [%-4r [%t] %-5p %c %x - %m%n].
log4j: End of parsing for "A1".
log4j: Parsed "A1" options.
log4j: Finished configuring.
```

Audit des modifications apportées aux objets de données

Dans de nombreuses implémentations, vous pouvez effectuer une analyse rétrospective qui contrôle les modifications apportées aux données de Visual Modeler. Il est nécessaire pour cela de consigner toutes les modifications apportées aux objets de données. Si vous définissez le niveau de consignation sur INFO ou sur un niveau supérieur dans une classe DataBean, chaque fois que vous appelez la méthode *persist()* sur une instance de la classe, un message de journal est écrit dans le consignateur pour cette classe. Voici un exemple de ligne écrite en cas de modification du partenaire :

```
2006.01.18 13:41:05:546 Env/http-8080-Processor23:INFO:PartnerBean Updating:
com.comergent.bean.simple.PartnerBean KeyFields - PartnerKey: 301 Changes -PartnerKey ->
old: 301 new: 301PartnerName -> old: Scalar2 new: Scalar2 LegalName ->
old: null new: null ParentCompany -> old: null new: nullStatus ->
old: A new: A DunBradID -> old: null new: nullBusinessID ->
old: Scalar2-001 new: Scalar2-001PartnerTypeCode -> old: 10 new: 10PartnerLevelCode ->
old: 20 new: 20XMLMessageVersion -> old: dXML 4.0 new: dXML 4.0BusinessTransaction ->
old: SELL new: SELL NetWorth -> old: null new: null NumEmployees ->
old: null new: null PotRevCurrFy -> old: null new: null PotRevNextFy ->
old: null new: null ReferenceUseFlag -> old: null new: null CotermDayMonth ->
old: null new: nullURL -> old: http:///www.scalar.com new: http:///www.scalar2.com LogoURL ->
old: null new: null DistiAccess -> old: null new: null YearEstd -> old: null new:
null AnalysisFy -> old: null new: null FyEndMonthCode -> old: null new: null AccountManagerKey ->
old: null new: null MessageURL -> old: null new: null EmailAddress ->
old: null new: nullCommerceCategory -> old: 2 new: 2 PartnerRefNum ->
old: null new: null ParentKey -> old: null new: null RootPartnerKey ->
old: null new: null ParentCode -> old: null new: null CustomField1 ->
old: null new: null CustomField2 -> old: null new: null CustomField3 ->
old: null new: null CustomField4 -> old: null new: null CustomField5 ->
old: null new: null PartnerCom -> old: null new: null Storefront ->
old: null new: null URLName -> old: null new: null ContentType ->
old: null new: nullPartnerStatusCode -> old: 10 new: 10OrganizationType ->
old: DirectPartner new: DirectPartner InheritedPartnerStatusCode ->
old: null new: nullCreditLimit -> old: 0.0000 new: 0.00AvailableCredit ->
old: 0.0000 new: 0.0000CreditCurrencyCode -> old: 23 new: 23 MaxAssignableReps ->
old: null new: null RemotePrices -> old: null new: null RemotePriceExpiryInterval ->
old: null new: nullCoopPercentage -> old: 0.000000 new: 0.000CoopAccountMax ->
old: 0.000000 new: 0.00 PartnerID -> old: null new: nullOwnedBy ->
old: 0 new: 0AccessKey -> old: 5601 new: 5601UpdateDate ->
old: 2006-01-18 13:39:33.0 new: 2006-01-18 13:41:05.484UpdatedBy ->
old: 0 new: 0CreateDate -> old: 2006-01-04 13:19:38.0 new: 2006-01-04 13:19:38.0CreatedBy ->
old: 0 new: 0
```

Le niveau de consignation peut être modifié de façon dynamique pour chaque classe de Visual Modeler à l'aide de l'interface utilisateur d'administration. Cependant, cette modification du niveau de consignation n'est pas persistante et

sera perdue au redémarrage du conteneur de servlet. En outre, la consignation est écrite dans l'ajouteur standard, qui n'est peut-être pas sécurisé.

Pour indiquer que la consignation doit être écrite dans le journal d'audit, vous devez personnaliser le fichier de configuration **log4j.properties**. Cela garantit que l'audit sera poursuivi après le redémarrage du conteneur de servlet et permet de spécifier un ajouteur personnalisé pour le traitement des informations d'audit. Par exemple, vous pouvez indiquer que l'ajouteur doit publier le message de journal sur un serveur Web distant qui peut être sécurisé indépendamment de Visual Modeler.

Par exemple, les entrées suivantes du fichier de configuration **log4j.properties** indiquent qu'un audit doit être effectué pour toutes les modifications apportées à l'objet de données UserContact :

```
log4j.logger.com.comergent.bean.simple.UserContactBean=info
log4j.appender.com.comergent.bean.simple.
UserContactBean=com.comergent.logging.ComergentRollingFileAppender
log4j.appender.com.comergent.bean.simple.
UserContactBean.layout = org.apache.log4j.PatternLayout
```

Si vous souhaitez qu'un serveur de consignation distant puisse se connecter en tant que client pour enregistrer les informations d'audit à partir de Visual Modeler, vous pouvez entrer les lignes suivantes :

```
log4j.appender.com.comergent.bean.simple.UserContactBean=org.apache.log4j.net.
SocketHubAppender
log4j.appender.com.comergent.bean.simple.UserContactBean.port=4321
```

Chapitre 10. Modularité et interfaces générées

Visual Modeler offre les fonctions suivantes pour faciliter la personnalisation et la mise à niveau des implémentations :

- Modules
- Interfaces générées

Ces fonctions sont associées dans la mesure où les interfaces sont organisées par modules et les modifications apportées aux interfaces peuvent être incluses dans les modifications apportées à des modules individuels.

La fonctionnalité fournie à travers les modules et la possibilité pour les modules d'appeler d'autres modules uniquement via leur interface externe présentent les avantages suivants :

- Un compartimentage plus facile du fonctionnement des applications.
- Une meilleure compréhension et une gestion plus facile des relations de dépendance entre les différentes parties de Visual Modeler.
- Personnalisation plus facile de modules individuels et meilleure compréhension des effets provoqués par la modification d'un module sur l'ensemble du système.
- Mise à niveau plus facile des modules indépendamment les uns des autres, ce qui permet de réduire les éventuels effets d'une mise à niveau.
- La mise à niveau des modules non personnalisés n'affecte pas les personnalisations effectuées dans d'autres modules.
- Vous pouvez fournir une nouvelle fonctionnalité en plaçant un module dans un déploiement existant de Visual Modeler.

Chapitre 11. Modules dans Visual Modeler

Présentation des modules dans Visual Modeler

Visual Modeler a été développé comme un ensemble de modules interdépendants qui respectent une structure organisationnelle commune. En général, chaque module correspond à un composant fonctionnel de Visual Modeler, tel qu'une application ou un composant de la plateforme Visual Modeler. Certains modules prennent en charge une API Java et une interface utilisateur, d'autres prennent en charge uniquement une API Java pouvant être fournie aux autres modules. Certains modules fournissent un ensemble de classes d'auxiliaire, des pages JSP et divers autres fichiers tels que des fichiers et des images JavaScript utilisés par d'autres modules.

En général, tous les modules ont la structure suivante :

- Classes Java : Elles sont organisées en trois arborescences. Lors de la phase de création, les répertoires de tous les modules sont réunis sous une seule arborescence, dans le package com.comergent.
  - Interfaces API externes : Elles sont utilisées par les autres modules pour accéder à la fonctionnalité fournie par un module. En général, lorsqu'un module effectue un appel vers une classe d'un autre module, il doit utiliser l'API externe de l'autre module. Il s'agit du package com.comergent.api du module. En outre, vous pouvez utiliser com.comergent.appservices.appServiceUtils.OFApiHelper pour appeler les XAPI de Sterling Selling and Fulfillment Foundation.
  - Classes d'implémentation : Elles sont utilisées pour l'implémentation des API externes. Lorsqu'un autre module effectue un appel vers l'API externe du module, les classes utilisées sont les classes d'implémentation de l'interface du module. Les packages d'implémentation peuvent inclure des classes internes, qui sont utilisées par les classes d'implémentation ; ces classes ne sont pas publiques et ne font pas partie de la Javadoc fournie. Il s'agit du package com.comergent.apps ou com.comergent.appservices du module.
  - Composants de référence : Les classes Controller et les pages JSP font toujours partie de l'implémentation de référence et leur source est fournie avec Visual Modeler. L'implémentation de référence fournit également les regroupements de ressources. Il s'agit du package com.comergent.reference du module.
- Pages JSP : Elles peuvent être organisées en répertoires, en fonction de l'organisation du module. Elles doivent toujours accéder aux classes des autres modules via les API externes de ces modules. Cela permet la réutilisation des pages JSP d'une version à l'autre, si les API externes sont prises en charge.
- Regroupements de ressources, fichiers JavaScript et fichiers statiques (images et fragments HTML).
- Fichiers de configuration spécifiques au module, tels que les fichiers **MessageTypes.xml** et les règles métier.

Interfaces des modules

Chaque module doit fournir une interface externe utilisée pour effectuer tous les appels vers les classes et les interfaces Java du module. Un ensemble complet de pages Javadoc est disponible pour cette interface externe, afin qu'elle puisse être utilisée facilement et de façon fiable par les développeurs d'autres modules.

L'interface externe de chaque module se présente en général comme une combinaison d'interfaces créées manuellement et d'interfaces générées automatiquement. La plupart des modules fournissent des interfaces créées manuellement pour les beans de présentation ; ces interfaces permettent aux beans de présentation d'effectuer des manipulations de données autres que celles autorisées par les méthodes d'accès standard fournies par les interfaces de bean de données générées. En général, les beans de présentation encapsulent un bean de données et implémentent les mêmes interfaces que le bean de données, mais ils implémentant, en outre, des méthodes d'auxiliaires et certaines logiques métier.

Les interfaces externes sont organisées dans les packages suivants :

- com.comergent.api : Ce package contient toutes les API externes du module. Elles sont divisées en :
  - apps : Ce sont des API d'application créées manuellement. Il s'agit en général d'interfaces de bean de présentation, d'interfaces utilitaires et de classes de fabrique.
  - appservices : Il s'agit des packages fournis par les modules de service utilisés par d'autres applications.
  - dcm : Il s'agit des API externes fournies par la plateforme Visual Modeler.
- com.comergent.bean.simple : Ce package contient toutes les interfaces de bean générées automatiquement, ainsi que les classes de bean de données.

Les interfaces générées sont fournies pour chaque objet de données déclaré dans les fichiers de schéma XML. Elles sont organisées de sorte à fournir des niveaux d'accès appropriés aux zones des beans de données sous-jacents. Cela permet de s'assurer qu'il y a une séparation nette entre la présentation et la logique métier dans Visual Modeler. Pour plus d'informations sur les interfaces générées, voir Chapitre 12, «Interfaces générées», à la page 77.

Appel des interfaces

Vous pouvez appeler une interface à partir d'une classe Java en convertissant n'importe quel objet ou interface enfant vers l'interface souhaitée, puis en appelant une méthode déclarée par cette interface. Pour ce faire, vous pouvez utiliser l'une des techniques suivantes dans Visual Modeler :

- «Utilisation du gestionnaire d'objets»
- «Utilisation des classes de fabrique», à la page 75

Chaque module utilise l'une ou l'autre de ces techniques, mais pas les deux. Lorsque vous utilisez un module existant ou créez un nouveau module, veillez à conserver une cohérence dans les noms d'interfaces. Vos collègues pourront ainsi travailler plus facilement avec le même module.

En général, vous devez toujours essayer d'utiliser les interfaces fournies par les packages com.comergent.api ; ces interfaces sont prises en charge par les modules d'une version à l'autre, sans tenir compte des éventuelles modifications apportées aux implémentations sous-jacentes.

Utilisation du gestionnaire d'objets

Vous pouvez utiliser la classe ObjectManager pour renvoyer une interface appropriée. Supposons que vous souhaitez récupérer l'interface IAccProduct pour définir les zones de données d'un produit. Effectuez un appel à l'aide des lignes suivantes :

```
IAccProduct temp_IAccProduct =

(com.comergent.bean.simple.IAccProduct)
com.comergent.dcm.util.OMWrapper.getObject(
  "com.comergent.bean.simple.IAccProduct");
```

Si le fichier **ObjectMap.xml** contient une entrée qui spécifie l'objet à renvoyer et si cet objet implémente l'interface IAccProduct, l'appel aboutira et vous pouvez appeler des méthodes sur l'interface. Par exemple, si le fichier **ObjectMap.xml** contient les lignes suivantes :

```
<Object ID="com.comergent.bean.simple.IAccProduct">
<ClassName>com.comergent.bean.simple.ProductBean</ClassName>
```

ObjectManager renvoie un objet com.comergent.bean.simple.ProductBean, qui peut être converti vers l'interface IAccProduct car la classe com.comergent.bean.simple.ProductBean implémente l'interface com.comergent.bean.simple.IAccProduct.

Utilisation des classes de fabrique

Les appels vers une interface peuvent être effectués par des classes de fabrique qui renvoient une instance de cette nterface. Par exemple, le package com.comergent.api.apps.commerce fournit l'interface publique IInquiryListFactory. Si un autre module requiert une instance de cette interface de fabrique, il appelle la méthode *getFactory(int i)* de la classe CommerceAPI. Le paramètre int détermine le type de classe de fabrique renvoyée. À son tour, le module appelant peut maintenant appeler des méthodes sur IInquiryListFactory pour renvoyer des interfaces de liste de demandes du type approprié. Par exemple, *getInquiryList(Long listKey, boolean bFillPrices)* renvoie un objet qui implémente l'interface IInquiryList.

Chapitre 12. Interfaces générées

Si vous souhaitez accéder aux données d'un objet de données spécifique, vous devez utiliser les interfaces générées fournies par chaque objet de données. Ces interfaces générées sont créées et compilées lors de l'exécution de la cible SDK generateBean dans le cadre du déploiement de Visual Modeler.

Pour chaque objet de données déclaré comme DataObject dans le fichier **DsRecipes.xml** et chaque objet de données parent, enfant ou de référence, les classes et les interfaces suivantes sont générées et compilées dans le package com.comergent.bean.simple :

- <*Nom*>.java : Il s'agit de la classe de bean de données. Elle implémente les interfaces répertoriées dans cette section. En outre, si l'objet de données étend un autre objet de données, le bean étend le bean <*Parent*>.java.
- IAcc<*Nom*>.java : Cette interface étend l'interface IRd<*Nom*>.java en fournissant les méthodes d'accès write (set) pour l'ensemble des zones de l'objet de données. En outre, si l'objet de données étend un autre objet de données, l'interface IAcc étend l'interface IAcc<*Parent*>.java.
- IData<*Nom*>.java : Cette interface étend l'interface IAcc<*Nom*>.java en fournissant les méthodes *restore()* et *persist()* pour l'objet de données. En outre, si l'objet de données étend un autre objet de données, l'interface IData étend l'interface IData<*Parent*>.java.
- IRd<*Nom*>.java : Cette interface fournit les méthodes d'accès read-only (get) pour les zones de l'objet de données. En outre, si l'objet de données étend un autre objet de données, l'interface IRd étend l'interface IRd<*Parent*>.java.
- Les beans de liste implémentent également l'interface IData<*Nom*>List.java. Chaque interface de liste étend l'interface IDataList.java ainsi que l'interface de liste de tout objet parent.

En général, vous devez utiliser l'interface IRd pour tous les objets que vous souhaitez transmettre aux pages JSP afin que ces objets soient effectivement en lecture seule. Utilisez des objets qui implémentent l'interface IData uniquement si vous savez que vous devez restaurer ou faire persister l'objet de données.

Exemple d'interface générée

Prenons l'objet de données de liste de contrôle d'accès. Le fichier **ACL.xml** se présente comme suit :

```
<?xml version="1.0"?>
<DataObject Name="ACL" Extends="C3PrimaryRW"
ExternalName="CMGT_ACLS"
Access="RWID" Ordinality="1"
ObjectType="JDBC" Version="5.0">
<KeyFields>
<KeyField Name="AccessKey" ExternalName="ACL_KEY"
KeyGenerator="ACLKey"/>
</KeyFields>
<DataFieldList>
<DataField Name="AccessKey"
Writable="n" Mandatory="n"
ExternalFieldName="ACL_KEY"/>
```

```
<DataField Name="ACLName"
Writable="y" Mandatory="n"
ExternalFieldName="NAME"/>
</DataFieldList>
<ChildDataObject Name="Access" />
</DataObject>
```

Par conséquent, la classe IRdACL.java déclare :

```
public interface IRdACL extends IRdC3PrimaryRW
```

et fournit les méthodes :

- public Long getAccessKey();
- public String getACLName();

La classe IAccACL.java déclare :

```
public interface IAccACL extends IAccC3PrimaryRW, IRdACL
```

et fournit les méthodes :

- public void setACLName(String value) throws ICCException;
- public void addAccess(AccessBean bean) throws ICCException;

La classe IDataACL.java déclare :

```
public interface IDataACL extends IAccACL,IDataC3PrimaryRW, IData
```

En général, cette interface ne déclare pas d'autres méthodes que celles déclarées dans l'interface IData, car cette dernière contient toutes les méthodes standard pour la lecture et l'écriture de données dans des sources de données externes.

Chapitre 13. Classes logiques dans Visual Modeler

Implémentation des classes de logique

Cette rubrique et les deux rubriques suivantes expliquent comment implémenter des classes de logique métier lors de l'implémentation de Visual Modeler. Avant de lire cette rubrique, il est nécessaire de disposer d'une bonne connaissance de l'architecture de base de Visual Modeler et de Java.

Remarque : L'utilisation des classes de logique métier est obsolète. En général, les nouvelles applications doivent implémenter la logique métier à l'aide de bizlets, de contrôleurs et de BizAPI.

Concepts clés des classes logiques

Pour bien comprendre le fonctionnement de l'application Visual Modeler, vous devez comprendre son architecture.

Une installation de Visual Modeler traite les demandes au fur et à mesure qu'elles sont reçues des navigateurs des utilisateurs, ainsi que les messages provenant d'autres installations de Visual Modeler et de systèmes externes. Vous devez configurer Visual Modeler pour le traitement de chaque type de demande et de message.

Le gestionnaire est le noyau de Visual Modeler. Souple et puissant, ce serveur est conçu pour intégrer de façon transparente un réseau de partenaires distributeurs et les systèmes externes qui forment l'environnement de commerce électronique de chaque partenaire.

Chaque serveur Visual Modeler du réseau de partenaires commerciaux fonctionne à la fois comme un serveur, car il reçoit les demandes entrantes des navigateurs, et comme un client, car il récupère des informations à partir d'autres serveurs Visual Modeler et de systèmes externes.

Pour personnaliser l'installation de Visual Modeler dans votre environnement, vous devez prendre en considération la manière dont le système récupère des données à partir de vos systèmes externes. En général, vous pouvez utiliser les classes Schema et Service pour récupérer des informations à partir d'une base de données locale ou d'un autre serveur Visual Modeler via l'échange de messages. Toutefois, pour récupérer des informations à partir d'un autre système externe, vous devez créer des classes de logique métier personnalisées.

Classes de logique d'application

Les classes de logique d'application sont implémentées en tant que classes bizAPI, logiques métier ou classes de contrôleur.

- Les classes bizAPI sont utilisées pour gérer la logique métier des objets métier. D'un point de vue conceptuel, chaque classe bizAPI correspond à un objet métier et ses méthodes correspondent aux actions pouvant être effectuées sur l'objet métier. Par exemple, la classe bizAPI OrderInquiryList fournit les méthodes *duplicate()*, *copyLineItem()* et *changeOwner()*, qui représentent les actions pouvant être effectuées sur une liste de demandes sur les produits. Elle implémente l'interface com.comergent.api.apps.orderMgmt.oil.IOrderInquiryList.

Les classes bizAPI sont définies dans les paquets com.comergent.apps.<*application*>.bizAPI. En général, elles implémentent une interface déclarée dans le paquet com.comergent.api.apps.<*application*> correspondant.

Par exemple, la classe bizAPI Order se trouve dans le paquet com.comergent.apps.orderMgmt.orders.bizAPI. Elle étend la classe OrderInquiryList et implémente l'interface com.comergent.api.apps.orderMgmt.orders.IOrder.

- Chaque classe de logique métier est une sous-classe de la classe abstraite BLC. Cette classe implémente l'interface ApplicationObject. Les classes de logique métier exécutent la logique métier de votre implémentation de Visual Modeler. Chaque classe de logique métier contient une table d'objets métier tels que la session, l'utilisateur et le panier. L'exécution de la méthode *service()* d'une classe de logique métier appelle les méthodes *persist()* et *restore()* des objets métier.

  Remarque : En général, l'utilisation des classes de logique métier est obsolète. Vous devez gérer la logique métier à l'aide de classes de contrôleur ou de classes bizAPI.
- Dans certaines installations de Visual Modeler, la logique métier est exécutée à l'aide de classes de contrôleur. Ces classes se trouvent dans les paquets com.comergent.reference.apps.<*application*>.controller de chaque application.

Visual Modeler est fourni avec plusieurs classes bizAPI, classes de logique métier, classes de contrôleur et pages JSP standard. Cependant, vous pourriez être amené à créer de nouvelles classes logiques ou à modifier les classes existantes.

Schéma XML

Vous devez gérer l'accès aux données à l'aide des classes Schema et Service.

Service d'attribution de noms

Pour récupérer les paramètres lors de l'exécution, Visual Modeler fournit un service d'attribution de noms permettant d'accéder à un fichier à plat ou à une base de données.

Les classes de logique d'application peuvent appeler un service d'attribution de noms à l'aide des méthodes *getInstance()* et *getInstance(int i)* de la classe statique NamingManager. Ces deux méthodes renvoient un objet qui implémente l'interface NamingService.

- Si aucun argument de type entier n'est spécifié, un objet de type par défaut est créé (un objet NamingServiceProperties ou NamingServiceDatabase).
- Si l'argument de type entier est la constante NamingManager.DATABASE, un objet NamingServiceDatabase est créé.
- Si l'argument de type entier est la constante NamingManager.PROPERTIES, un objet NamingServiceProperties est créé.
- Si l'argument de type entier n'est pas l'une de ces deux constantes, un objet de type par défaut est créé.

Dans tous les cas, Visual Modeler accède au fichier **Comergent.xml** pour déterminer précisément comment doit être créé l'objet NamingService :

- Si l'objet NamingServiceDatabase doit être créé, les entrées NamingManager.database sont utilisées pour définir la connexion à la base de données.

- Si l'objet NamingServiceProperties doit être créé, l'entrée NamingManager.properties est utilisé pour déterminer dans quel fichier de propriétés sont stockées les valeurs de paramètre.

Une fois l'objet NamingService créé, vous pouvez utiliser les méthodes répertoriées ci-dessous pour récupérer les paramètres en tant que classe NamingResult :

- public NamingResult get(int key)
- public NamingResult get(Long key)
- public NamingResult get(String key)

Le paramètre key permet de récupérer seulement les paramètres dont le nom commence par la chaîne clé.

La classe NamingResult fournit la méthode *get(String parameter)* permettant de renvoyer la valeur du paramètre.

Exemple d'utilisation de l'objet NamingService

Par exemple, le fragment de code suivant récupère la valeur du paramètre d'URL de message pour un distributeur identifié par une clé de partenaire.

```
NamingService namingService = NamingManager.getInstance();
NamingResult namingResult = namingService.get(partnerKey);
String url = namingResult.get(NamingResult.MESSAGE_URL);
```

Remarque : Par défaut, l'objet NamingService crée est un objet NamingServiceDatabase parce que l'élément NamingManager defaultType est défini sur "database" dans le fichier **Comergent.xml**.

Chapitre 14. Kit de développement de logiciels Visual Modeler

Utilisation du kit de développement de logiciels (SDK) pour personnaliser l'implémentation de Visual Modeler

Vous pouvez utiliser le kit de développement de logiciels (SDK) de Visual Modeler pour installer et personnaliser votre implémentation de Visual Modeler. La documentation HTML accompagnant votre version de l'outil SDK fournit une présentation de son fonctionnement et explique comment l'utiliser pour gérer des projets. Cette section décrit la structure de base d'un projet de personnalisation. Suivez ces instructions de personnalisation pour organiser votre projet.

Organisation des projets

Tous les projets créés à l'aide de l'outil SDK sont placés à la racine du répertoire d'installation de la version de Visual Modeler. Lorsque vous créez un projet à l'aide de la cible newproject, l'outil SDK crée un ensemble de fichiers de projet appropriés pour cette version. Vous devez ajouter des fichiers de projet pour chaque personnalisation que vous effectuez dans un projet. Il existe plusieurs manières d'ajouter des fichiers à un projet :

- Utilisez la cible customize pour copier un fichier depuis un répertoire de destination vers le projet. Lorsque vous utilisez la cible customize, le fichier est copié automatiquement dans le sous-répertoire approprié du projet.
- Créez manuellement le fichier dans le sous-répertoire approprié du projet.

Pour plus d'informations sur l'emplacement des fichiers, voir «Emplacements des fichiers et des répertoires de projet».

Emplacements des fichiers et des répertoires de projet

Cette section part du principe que vous avez créé un projet appelé *projet* et que vous avez un répertoire de projet appelé ***rép_base_sdk*/projects/*projet*/**. Assurez-vous que chaque fichier de projet est situé à un emplacement approprié sous le répertoire de projet, comme suit :

- Fichiers source Java : Ils doivent être placés sous le répertoire ***projet*/src/** et respecter l'organisation des packages de Visual Modeler.
- Pages JSP : Elles sont organisées par module et par paramètres régionaux sous le répertoire ***projet*/WEB-INF/web/** directory.
- Fichiers schéma : Ils comprennent les fichiers d'objet de données et les fichiers des services de données pris en charge. Toutes vos personnalisations doivent être stockées sous le répertoire ***projet*/WEB-INF/schema/custom/**. Assurez-vous que l'élément schemaRepositoryExtn est défini sur "WEB-INF/schema/custom".

Fichiers source Java

Dans le répertoire ***projet*/src/**, suivez les instructions suivantes pour organiser vos personnalisations de Visual Modeler :

- Utilisez les packages com/comergent/api/ pour ajouter vos extensions à l'API Visual Modeler. En général, vous devez créer de nouvelles classes qui étendent l'API existante : n'écrasez pas l'API de version car cela peut affecter les mises à niveau ultérieures.

- Utilisez les packages com/comergent/apps/ et com/comergent/appservices/ pour ajouter des classes d'implémentation ; vous pouvez ajouter des classes entièrement nouvelles ou de nouvelles classes qui étendent les classes d'implémentation existantes.
- Utilisez les packages com/comergent/reference/ pour les classes de contrôleur. Vous pouvez personnaliser des classes de contrôleur existantes ou en créer de nouvelles.

Pages JSP

Dans le répertoire ***projet*/WEB-INF/web/**, suivez les instructions suivantes pour organiser vos personnalisations de Visual Modeler :

- Lorsque cela est possible, utilisez l'organisation existante des pages JSP pour ajouter de nouvelles pages JSP ou personnaliser des pages existantes.
- Si vous ajoutez un module fournissant une nouvelle fonctionnalité, créez un nouveau répertoire dans les paramètres régionaux de ce module et respectez les conventions d'attribution de noms utilisées pour les classes Java créées pour le module.

Fichiers de schéma

Dans le répertoire ***projet*/WEB-INF/schema/custom/**, suivez les instructions suivantes pour organiser vos personnalisations de Visual Modeler :

- Pour ajouter de nouveaux objets de données, procédez comme suit :
  - Placez la définition XML de l'objet de données dans ***projet*/WEB-INF/schema/custom/**. Par exemple, créez le fichier ***projet*/WEB-INF/schema/custom/CustComment.xml**
  - Modifiez le fichier ***projet*/WEB-INF/schema/custom/DsBusinessObjects.xml** en ajoutant le nouvel objet métier. Par exemple :

    ```
    <?xml version="1.0"?>
    <Schema Name="Projet" Description="Schéma de projet personnalisé"
    Version="6.0">
    <BusinessObject Name="CommentaireClient" Version="6.0"
    Description="Objet métier CommentaireClient"/>
    </Schema>
    ```

  - Modifiez le fichier ***projet*/WEB-INF/schema/custom/DsDataElements.xml** en ajoutant les nouvelles données pour les objets de données d'en-tête et de liste, ainsi que tous les nouvelles zones déclarées par l'objet de données. Par exemple :

    ```
    <?xml version="1.0"?>
    <Schema Name="Projet" Description="Schéma de projet personnalisé"
    Version="6.0">
    <DataElement Name="CommentaireClient" Description="Objet de données
    CommentaireClient"
    DataType="HEADER"/>
    <DataElement Name="ListeCommentairesClient" Description="Donnes de la
    liste de commentaires client
    object" DataType="HEADER"/>
    <DataElement Name="CléCommentaireClient" Description="Clé du
    commentaire client"
    DataType="LONG" MaxLength="20"/>
    </Schema>
    ```

  - Modifiez le fichier ***projet*/WEB-INF/schema/custom/DsRecipes.xml** en ajoutant l'élément recipe. Par exemple :

```
<Schema Name="Projet" Description="Schéma de projet personnalisé"
Version="6.0">
<Recipe Name="CommentaireClient" BusinessObject="CommentaireClient"
Description="Élément de recette de la liste d'approbations par défaut"
Version="6.0">
        <DataObjectList>
<DataObject Name="CommentaireClient" Access="RWID"
DataSourceName="ENTERPRISE" Ordinality="n"
Version="6.0"/>
</DataObjectList>
</Recipe>
</Schema>
```

– Modifiez le fichier générateur de clés approprié, par exemple projet/WEB-INF/schema/custom/OracleKeyGenerators.xml, en ajoutant les nouvelles clés requises :

```
<?xml version="1.0"?>
<Schema Description="Schéma de projet personnalisé" Name="Projet"
Version="6.0">
<KeyGenerator Name="CléCommentaireClient"
KeyProcedureName="CLECOMMENTAIRECLIENT"
GeneratorType="PROCEDURE" />
</Schema>
```

Chapitre 15. Localisation dans Visual Modeler

Présentation de Visual Modeler Localization

Visual Modeler offre une prise en charge intégrée pour :

- plusieurs devises
- plusieurs langues
- plusieurs formats de nombre et de date
- plusieurs jeux de caractères

En outre, vous pouvez gérer d'autres aspects de la localisation pour des marchés spécifiques, parmi lesquels :

- les lois et réglementations locales
- le traitement des devises
- les informations sur l'expédition et l'exportation
- les taxes applicables

Les paramètres régionaux sont utilisés pour la prise en charge de l'internationalisation. Chaque ensemble de paramètres régionaux identifie une langue et un pays (ou région). En identifiant les paramètres régionaux à utiliser pour l'affichage de données à l'utilisateur, vous vous assurez que l'utilisateur visualise les informations en fonction de ses paramètres régionaux et que ces informations sont présentées sous une forme appropriée.

Lorsque les utilisateurs se connectent à Visual Modeler, un ensemble de paramètres régionaux est affecté à la session : il s'agit des paramètres régionaux préférés spécifiés dans le profil utilisateur. Les utilisateurs peuvent modifier leurs paramètres régionaux préférés dans leur profil utilisateur ; cette modification sera prise en compte lors de la prochaine connexion. Les administrateurs peuvent modifier les paramètres régionaux préférés d'un utilisateur comme tout autre élément de leurs profils utilisateur.

Les paramètres régionaux par défaut du système sont indiqués dans le fichier de configuration **Internationalization.xml** dans l'élément defaultSystemLocale. Vous pouvez spécifier un ensemble de paramètres régionaux par défaut pour chaque langue. Pour plus d'informations, voir «Comportement de basculement», à la page 90.

Visual Modeler offre une prise en charge complète du format Unicode pour la saisie et l'affichage de données.

La plupart des tâches de localisation peuvent être effectuées à l'aide de regroupements de ressources Java. Pour plus d'informations, voir «Regroupements de ressources et formats», à la page 97.

Prise en charge des paramètres régionaux

Si vous envisagez d'implémenter Visual Modeler pour prendre en charge des paramètres régionaux autres que en_US, vous devez créer des pages reflétant la langue de ces paramètres régionaux et fournir d'autres informations spécifiques

aux paramètres régionaux (par exemple, l'adresse des bureaux).

Paramètres régionaux de présentation et de session

Lorsqu'un utilisateur se connecte à Visual Modeler, le processus d'authentification récupère ses paramètres régionaux préférés définis dans son profil utilisateur. Le système utilise deux paramètres régionaux logiquement distincts :

- Paramètres régionaux de session : Ils déterminent quelles données récupérer à partir de Knowledgebase pour les objets de données.
- Paramètres locaux de présentation : Ils déterminent quelles pages JSP et quels regroupements de ressources utiliser pour l'affichage des pages HTML à l'utilisateur.

En général, l'ensemble de paramètres régionaux pris en charge en tant que paramètres régionaux de présentation doit être un sous-ensemble des paramètres régionaux de session possibles. Par exemple, vous choisissez de stocker les paramètres fr_CA, fr_CH et fr_FR en tant que paramètres régionaux de session, mais seuls les paramètres fr_FR et fr_CA sont pris en charge en tant que paramètres régionaux de présentation.

Lors de la première connexion d'un utilisateur, le système détermine quels paramètres régionaux utiliser pour la session utilisateur :

1. Si les paramètres régionaux préférés de l'utilisateur sont déclarés dans le fichier **web.xml** de Visual Modeler, ils sont définis en tant que paramètres régionaux de présentation.
2. Sinon, consultez le fichier **Internationalization.xml** : si l'élément useCountryDefaulting est défini sur true, identifiez le pays ou la région par défaut pour la langue des paramètres régionaux préférés de l'utilisateur. Vérifiez si le pays ou la région par défaut est déclaré dans le fichier **web.xml** pour cette langue. Le cas échéant, les paramètres régionaux de présentation sont définis sur ce pays.
3. Si l'élément useCountryDefaulting est défini sur false ou si les paramètres par défaut du pays ou de la région ne figurent pas dans le fichier **web.xml**, et si l'élément useGeneralDefaulting est défini sur true, alors définissez les paramètres régionaux de présentation de l'utilisateur sur les paramètres régionaux par défaut du système, indiqués dans l'élément defaultSystemLocale.
4. Si les éléments Defaulting sont définis sur false ou si le fichier **web.xml** ne contient aucun paramètre régional, les paramètres régionaux de présentation sont définis sur les paramètres régionaux de session.

Les paramètres régionaux de présentation déterminent l'expérience de l'utilisateur lors de la navigation dans Visual Modeler, en contrôlant les pages JSP et les fichiers de propriétés utilisés pour l'affichage des pages Web. En même temps, les paramètres régionaux préférés sont également définis en tant que *paramètres régionaux de session*. Les paramètres régionaux de session sont utilisés pour déterminer quelles données récupérer à partir de la base de données lorsque l'utilisateur affiche des objets de données localisés.

Remarque : Vous devez vous assurer que tous les paramètres régionaux que vous créez dans la base de données ont un ensemble d'entrées correspondant dans le fichier **web.xml** ou que ce fichier contient des entrées pour les paramètres régionaux par défaut du pays ou de la région et que l'affectation de valeur par défaut en fonction du pays ou de la région est activée. Dans le cas contraire, certains utilisateurs ne pourront peut-être pas à accéder au système.

Pages JSP et fichiers de propriétés

1. Pour chaque page JSP, au moins une page JSP doit figurer dans le sous-répertoire correspondant au module, sous le répertoire de paramètres régionaux par défaut du système. Lors de la première installation de Visual Modeler, les paramètres régionaux par défaut du système sont définis sur en_US. Un ensemble complet de pages JSP est fourni sous ***rép_base_debs*/SterlingWEB-INF/web/en/US/**. Si vous modifiez les paramètres régionaux par défaut du système, prenez soin de remplir intégralement les répertoires appropriées pour les nouveaux paramètres régionaux.
2. Tous les blocs de texte visibles sur chaque page sont déclarés à l'aide de la balise text de la bibliothèque de balises Comergent ou de la méthode *cmgtText()* correspondante. Par exemple :

   ```
   <cmgt:text
    id='cmgt_channelMgmt/channelCartDisplay/ChannelCartDisplayData_7'
   bundle='channelMgmt.channelCartDisplay.ChannelCartDisplayDataResources'
   >Build Product List </cmgt:text>
   ```

   ou

   ```
   String title = cmgtText("cmgt_commerce/search/AdvancedSearchBody_2",
   "Recherche dans la liste de demandes");
   ```

   L'attribut bundle doit correspondre à un fichier du package com.comergent.reference.jsp de l'arborescence de classes. Pour l'exemple ci-dessus, un fichier appelé **ChannelCartDisplayDataResource.properties** doit figurer dans le répertoire ***rép_base_debs*/Sterling/WEB-INF/classes/com/comergent/reference/jsp/channelMgmt/channelCartDisplay/**. L'attribut id doit être unique dans le fichier de propriétés. Pour l'exemple ci-dessus, il doit y avoir une ligne qui se présente comme suit :

   ```
   cmgt_channelMgmt/channelCartDisplay/ChannelCartDisplayData_7=Créer la liste de produits
   ```

3. Pour chaque ensemble de paramètres régionaux pris en charge (par exemple *la_CO*), vous devez copier les répertoires suivants depuis **rép_base_debs/Sterling/WEB⌂INF/web/en/US/** vers ***rép_base_debs*/Sterling/WEB⌂INF/web/*la/CO*/** :
   - **cic/**
   - **common/**
   - **home/**
4. Pour chaque ensemble de paramètres régionaux pris en charge (par exemple *la_CO*) et pour chaque page JSP, vous devez effectuer l'une des deux opérations suivantes :
   a. Créez une nouvelle page JSP pour ces paramètres régionaux et placez-la dans le répertoire correspondant de l'application Web, situé sous ***rép_base_debs*/Sterling/WEB⌂INF/web/*la/CO*/**. Si la même page peut être utilisée pour plusieurs paramètres régionaux pour la même langue (par exemple fr_FR et fr_CA), placez-la dans le répertoire des paramètres régionaux par défaut pour cette langue. Pour plus d'informations sur les paramètres régionaux par défaut des différentes langues, voir «Comportement de basculement», à la page 90.
   b. Préparez un fichier de propriétés contenant le texte approprié pour chaque ID. Il doit y avoir un fichier de propriétés par page ou fragment JSP.

      Les caractères HTML et JavaScript tels que "<", ">", """, etc. ne sont pas autorisés dans les valeurs des propriétés. Ces caractères doivent être précédés par les caractères d'échappement HTML ou JavaScript spécifiques. Par exemple, entrez "<" pour afficher le caractère "<" en HTML et "\'" pour afficher le caractère "'" en JavaScript.

Les fichiers de propriétés doivent respecter le standard Java relatif aux fichiers de propriétés utilisés par des regroupements de ressources. Plus particulièrement, ils doivent respecter la convention d'attribution de noms suivante : **<*Nom de la page JSP*>Resources_*la_CO*.properties**. Les fichiers de propriétés doivent être des fichiers texte contenant des lignes au format suivant :

```
     cmgt_module/package/JSPname_n=Texte d'affichage pour ces paramètres régionaux
Par exemple :
cmgt_channelMgmt/channelCartDisplay/
ChannelCartDisplayData_7=Créer la liste de produits
```

Tous les fichiers de propriétés se trouvent dans le répertoire ***rép_base_debs*/Sterling/WEB-INF/classes/com/comergent/reference/jsp/** ; dans ce répertoire, ils ont la même organisation par module que les pages JSP d'un module. Si vous souhaitez modifier l'emplacement de ces regroupements de ressources, vous devez personnaliser la balise text pour récupérer les regroupements de ressources à partir de leur nouvel emplacement.

Si vous ajoutez du texte à une page JSP, n'oubliez pas de mettre à jour les pages JSP ou les fichiers de propriétés des paramètres régionaux correspondants, soit en ajoutant du texte à une balise existante soit en ajoutant un nouvel ID.

Prenez connaissance des informations suivantes :

- La longueur du texte traduit peut être très différente du texte d'origine, ce qui peut affecter la mise en page d'une page Web.
- Les listes déroulantes et les fonctions JavaScript peuvent comporter du texte qui, une fois traduit, est susceptible d'affecter la logique de Visual Modeler. Voir «JavaScript», à la page 94 et «Pages JSP», à la page 84.
- L'affichage des informations doit parfois tenir compte de certaines réglementations locales (par exemple, l'affichage des prix en euros et en devise locale).
- Portez donc une attention particulière aux modifications que vous devez apporter au flux logique des pages pour respecter les réglementations locales en vigueur (par exemple, l'affichage d'une notice pour l'exportation ou d'informations fiscales).

Vous pouvez utiliser l'élément debugJSPResouceBundle du fichier de configuration **Internationalization.xml** pour identifier les chaînes manquantes. Lorsque cet élément est défini sur true, un message d'erreur s'affiche sur la page de navigateur si une chaîne du regroupement de ressources référencé est manquante. Dans les systèmes de production, vous devez définir cette valeur sur false.

Comportement de basculement

Cette section décrit le comportement de Visual Modeler lorsque des ressources (pages JSP ou propriétés) ne sont pas définies pour les paramètres régionaux de présentation actuels de l'utilisateur. Notez que les comportements de basculement des pages JSP et des regroupements de ressources sont légèrement différents :

- Les pages JSP peuvent basculer d'un ensemble de paramètres régionaux spécifique vers les paramètres de langue par défaut du pays ou de la région, puis vers les paramètres régionaux par défaut du système. Par exemple : fr_CA vers fr_FR, puis vers en_US.
- Le basculement des regroupements de ressources est déterminé par la spécification Java : ***_fr_CA.properties** vers ***_fr.properties** vers ***.properties**.

Deux propriétés dans le fichier de configuration **Internationalization.xml** sont utilisées pour gérer le comportement de basculement des pages JSP :

- useCountryDefaulting : Si cette propriété est définie sur true et si aucune ressource n'est disponible pour les paramètres régionaux de présentation, la page bascule par défaut vers le pays ou la région indiqué dans l'élément de langue correspondant.
- useGeneralDefaulting : Si cette propriété est définie sur true et si aucune ressource n'est disponible pour les paramètres régionaux de présentation, la page bascule par défaut vers les paramètres régionaux du système.

Comportement de basculement des regroupements de ressources

Il est inutile de traduire toutes les chaînes de texte dans tous les paramètres régionaux pris en charge. Si une chaîne de texte n'est pas disponible pour un ID donné dans le fichier de propriétés d'un regroupement de ressources, le processus de basculement Java standard est appliqué. Par exemple, si le fichier **ChannelCartDisplayDataResource_fr_CA.properties** ne contient pas la chaîne cmgt_channelMgmt/channelCartDisplay/ChannelCartDisplayData_7, le fichier **ChannelCartDisplayDataResource_fr.properties** est consulté. Si ce dernier n'existe pas ou ne contient pas d'entrée pour cet ID, le fichier **ChannelCartDisplayDataResource.properties** est consulté.

Comportement de basculement des pages JSP

Il n'est pas nécessaire que toutes les pages JSP soient disponibles pour tous les paramètres régionaux pris en charge. Par exemple, vous pouvez utiliser les pages en_US pour la plupart des pages affichées par les utilisateurs en_CA. Cette section décrit ce qui se passe lors du traitement d'un type de message donné :

La demande est dirigée vers la page JSP indiquée par l'élément JSPMapping du type de message dans le fichier **MessageTypes.xml** approprié.

1. S'il existe une page JSP pour les paramètres régionaux actuels, elle est utilisée pour générer la page Web.
2. Si la page JSP n'existe pas pour les paramètres régionaux actuels, le mécanisme de basculement identifie les paramètres régionaux par défaut pour la langue des paramètres régionaux actuels. Les paramètres régionaux par défaut sont déclarés par l'élément defaultCountry de la langue dans le fichier de configuration **Internationalization.xml**.
3. S'il existe une page JSP pour les paramètres régionaux par défaut de la langue, elle est utilisée pour générer la page Web. Par exemple, l'élément suivant du fichier **Internationalization.xml** indique que US est le pays ou la région par défaut pour la langue en ; si aucune page JSP n'est disponible pour les paramètres régionaux en_CA, la page JSP en_US correspondante est utilisée.
4. <en visible="false">
   <defaultCountry ...>US</defaultCountry>
   </en>
5. Si aucune page JSP n'existe pour le pays ou la région par défaut, le mécanisme de basculement identifie les paramètres régionaux par défaut du système. Les paramètres régionaux par défaut sont déclarés par l'élément defaultSystemLocale dans le fichier **Internationalization.xml**. S'il existe une page JSP pour les paramètres régionaux par défaut du système, elle est utilisée pour générer la page Web.
6. Enfin, si aucune page JSP n'existe pour les paramètres régionaux par défaut du système, une exception est lancée et une page d'erreur s'affiche.

Méthodes de récupération des paramètres régionaux

Dans la plupart des cas, Visual Modeler fournit un support intégré pour permettre l'affichage de contenu selon les paramètres régionaux des utilisateurs. Si vous avez besoin d'accéder manuellement aux paramètres régionaux, vous pouvez utiliser la classe ComergentI18N. Celle-ci fournit les méthodes suivantes :

- *getDefaultLocale()* : Renvoie les paramètres régionaux par défaut du système.
- *getComergentLocale(boolean b)*: Si b est défini sur true, renvoie les paramètres régionaux de présentation de l'utilisateur ; sinon, renvoie les paramètres régionaux de session de l'utilisateur.
- *findPresentationLocale(Locale sessionLocale)* : Permet de déterminer les paramètres régionaux de présentation à utiliser pour un ensemble de paramètres régionaux de session donné.

Utilisation de fichiers de propriétés dans le code

Vous pouvez utiliser des fichiers de propriétés dans votre code Java. Par exemple, pour récupérer la chaîne spécifique aux paramètres régionaux qui correspond à la chaîne keyString définie dans le fichier **com.comergent.reference.jsp.AdvisorBodyResources.properties**, entrez les lignes suivantes :

```
String temp_NamedPopertiesFile =
"com.comergent.reference.jsp.AdvisorBodyResources.properties";
ResourceBundle temp_ResourceBundle =
com.comergent.dcm.util.ComergentI18N.-
getBundle(temp_NamedPopertiesFile);
String temp_LocalisedString =
temp_ResourceBundle.getString("keyString");
```

Visual Modeler utilise les paramètres régionaux actuels de l'utilisateur stockés dans sa session utilisateur. Si vous souhaitez forcer l'utilisation d'un autre ensemble de paramètres régionaux, entrez les lignes suivantes :

```
Locale specific_Locale = new Locale("fr", "CA");
String temp_NamedPopertiesFile =
"com.comergent.reference.jsp.AdvisorBodyResources.properties";
ResourceBundle temp_ResourceBundle =
com.comergent.dcm.util.ComergentI18N.-
getBundle(temp_NamedPopertiesFile, specific_Locale);
String temp_LocalisedString =
temp_ResourceBundle.getString("keyString");
```

Données pour l'internationalisation

Si vous souhaitez que les utilisateurs entreprise et les utilisateurs finals puissent entrer des données comportant des caractères codés sur plusieurs octets, vous devez prendre en considération la longueur des zones de données et des colonnes correspondantes de la table de base de données. D'après notre expérience, lorsque des données comportant des caractères codés sur plusieurs octets sont entrées dans Visual Modeler, leur longueur dans la base de données peut être jusqu'à trois fois supérieure à la longueur des chaînes en_US correspondantes. Par conséquent, il est nécessaire de s'assurer qu'une longueur appropriée est définie pour les zones où les utilisateurs sont susceptibles d'entrer des données comportant des caractères codés sur plusieurs octets, notamment les zones Nom et Description.

Pour modifier la longueur des zones, gardez à l'esprit que cette modification doit être effectuée dans le fichier de configuration **DsDataElements.xml**, puis répercutée dans le script SQL utilisé pour la génération du schéma Knowledgebase.

Exemple

Pour agrandir la zone Description de l'objet de données Product afin de permettre la saisie de caractères codés sur plusieurs octets, procédez comme suit :

1. Déterminez dans quelle zone de données sont stockées les descriptions produit. Comme l'objet de données Product est localisable (Localized="y"), il correspond en effet à la zone Description de l'objet de données ProductLocale. CMGT_PRODUCT_LOCALE.DESCRIPTION est sa table de base de données et sa colonne correspondante.

   ```
   <DataField Name="Description" ExternalFieldName="DESCRIPTION"
   Mandatory="n" Writable="y"/>
   ```

2. Supposons que vous souhaitez autoriser une description comportant jusqu'à 240 caractères :

   <DataElement Name="Description" DataType="STRING" Description="Description" MaxLength="240" />

3. Modifiez l'instruction SQL correspondante qui crée la table CMGT_PRODUCT_LOCALE en définissant la colonne DESCRIPTION sur VARCHAR2(720) :

   DESCRIPTION VARCHAR2(720) DEFAULT 'Not available',

4. Exécutez les cibles SDK appropriées (merge et createDB) pour appliquer les modifications à votre implémentation de Visual Modeler.

Dans cet exemple, la zone de données Description est largement utilisée par de nombreux objets de données. La modification de sa définition dans le fichier de configuration **DsDataElements.xml** peut avoir des effets secondaires imprévus sur d'autres objets de données. Une autre méthode consiste à créer une nouvelle zone de données appelée ProductDescription et de l'utiliser dans l'objet de données ProductLocale. Vous pouvez par exemple ajouter les lignes suivantes dans le fichier **ProductLocale.xml** :

```
<DataField Name="ProductDescription"
ExternalFieldName="DESCRIPTION" Mandatory="n" Writable="y"/>
```

Ajoutez ensuite les lignes suivantes dans le fichier de configuration **DsDataElements.xml** :

```
<DataElement Name="ProductDescription" DataType="STRING"
Description="Cette zone contient la description du produit"
MaxLength="240" />
```

Remarque : Si vous utilisez une méthode JavaScript pour valider les données entrées dans les zones par les utilisateurs, vérifiez que la longueur des zones correspond à la longueur spécifiée dans l'élément DataElement associé.

Modèles d'e-mail

Si votre système prend en charge des langues autres que l'anglais et que votre installation de Visual Modeler utilise des modèles d'e-mail pour générer des messages envoyés aux utilisateurs, gardez à l'esprit que ces modèles doivent être traduits.

Depuis la version 6.4, il est possible d'utiliser des pages JSP pour générer des messages électroniques ; cette fonction permet d'internationaliser des messages électroniques à l'aide de l'infrastructure existante pour l'internationalisation des pages JSP.

Pour les applications existantes, vous pouvez utiliser les modèles par défaut fournis par Visual Modeler sous ***rép_base_debs*/Sterling/WEB-INF/templates/**.

Pages HTML

Les pages HTML statiques doivent être traduites lorsque cela est approprié. Si vous souhaitez prendre en charge plusieurs langues simultanément, vous devez produire des pages pour chacune des langues. Si ces pages sont stockées de façon cohérente dans la structure du répertoire de paramètres régionaux, les références relatives de ces pages pointeront toujours vers la page HTML appropriée.

Par exemple, le fragment JSP suivant génère de façon dynamique des URL pointant vers une page Exemple.html spécifique aux paramètres régionaux :

```
<A HREF="<cmgt:link app="catalog">
/static/Exemple.html
</cmgt:link>">
resourceBundle.getString("PageExemple")
</A>
```

Dans cet exemple, un regroupement de ressources est utilisé pour déterminer le texte affiché pour le lien.

Images

En général, vous devez utiliser des images qui ne comportent pas de texte intégré. Les mêmes images peuvent ainsi être utilisées dans plusieurs paramètres régionaux, en réduisant les coûts liés à la localisation et à la maintenance.

Toutefois, vous devez fournir des versions localisées des images lorsque cela est nécessaire. Comme pour les pages HTML statiques, vous pouvez utiliser des URL relatives pour vous assurer que les images spécifiques aux paramètres régionaux sont récupérées à partir de l'emplacement correspondant à la page JSP.

En particulier, n'oubliez pas que tous les boutons sur les pages publiques sont des boutons d'image qui comportent du texte. Lorsque cela est nécessaire, vous devez créer des versions localisées de chaque bouton. Les URL source des images peuvent être générées comme suit :

```
<IMG ALT="Placez ici le texte spécifique aux paramètres régionaux"
SRC="../images/bouton.gif"></A>
```

JavaScript

Le texte d'affichage utilisé dans votre code JavaScript doit être localisé. Par exemple, le texte affiché dans les boîtes de dialogue d'alerte doit correspondre aux paramètres régionaux de l'utilisateur.

- Certains fichiers JavaScript sont inclus dans les pages Web à l'aide des lignes suivantes :

```
<script language='JavaScript' src='../js/genericUtil.js'>
</script>
```

Ces fichiers JavaScript doivent être conservés pour tous les paramètres régionaux afin que le navigateur puisse les inclure correctement dans les pages Web générées.

- Si vous placez du code JavaScript dans une page ou un fragment JSP, le texte d'affichage doit être compris entre des balises text. Par exemple :

  ```
  alert("<cmgt:text id="*">ID produit manquant.</cmgt:text>");
  ```

  Lorsque ces balises sont traitées via l'outil SDK, l'attribut id est transformé en ID unique, qui est ensuite ajouté avec le corps de la balise au regroupement de ressources pour la page ou le fragment JSP.

Localisation dans Visual Modeler : Pages JSP

En général, la localisation des étiquettes, du texte d'explication, des listes et de l'affichage des dates et des devises en fonction des paramètres régionaux doit être reflétée dans les pages JSP créées pour un ensemble de paramètres régionaux.

Un principe d'organisation qui peut s'avérer utile consiste à créer une mappe de hachage de toutes les chaînes localisées d'une page et de faire référence à cette mappe dans le reste de la page. Par exemple :

```
HashMap localized = new HashMap();
localized.put("TaskListHeader",
cmgtText("cmgt_taskMgr/TaskWorkspaceData_3","Liste de tâches :"));
localized.put("QuickSearchTitle",
cmgtText("cmgt_taskMgr/TaskWorkspaceData_4","Recherche de tâches"));
localized.put("TaskID",
cmgtText("cmgt_taskMgr/TaskWorkspaceData_5","ID"));
localized.put("TaskName",
cmgtText("cmgt_taskMgr/TaskWorkspaceData_6","Nom"));
localized.put("Status",
cmgtText("cmgt_taskMgr/TaskWorkspaceData_7","État"));
localized.put("Priority",
cmgtText("cmgt_taskMgr/TaskWorkspaceData_8","Priorité"));
localized.put("CreateDate",
cmgtText("cmgt_taskMgr/TaskWorkspaceData_9","Date de crêation"));
request.setAttribute("localized", localized);
```

Vous pouvez faire référence à ces chaînes à l'aide des fonctions de scriptage utilisés dans les lignes suivantes :

```
<cic:span css="banner" value="${localized['TaskListHeader']}"/>
```

Cette technique a l'avantage de rendre les pages JSP plus lisibles, de faciliter la réutilisation des chaînes et de se rapprocher du modèle JSF.

Pour plus d'informations sur la localisation de ce composant de l'interface graphique, voir «Widget calendrier», à la page 96. Par exemple, pour remplir une liste déroulante des jours de la semaine dans un environnement en anglais, entrez les lignes suivantes :

```
<SELECT Name="DayOfWeek">
<OPTION VALUE=0>Sunday</OPTION>
<OPTION VALUE=1>Monday</OPTION>
<OPTION VALUE=2>Tuesday</OPTION>
```

```
<OPTION VALUE=3>Wednesday</OPTION>
<OPTION VALUE=4>Thursday</OPTION>
<OPTION VALUE=5>June</OPTION>
<OPTION VALUE=6>Friday</OPTION>
<OPTION VALUE=7>Saturday</OPTION>
</SELECT>
```

Vous pouvez également gérer les informations spécifiques aux paramètres régionaux à l'aide de regroupements de ressources. Par exemple, vous pouvez utiliser la méthode suivante pour remplir la liste déroulante des jours de la semaine dans le calendrier grégorien :

```
<SELECT Name="DayOfWeek">
<OPTION VALUE=0><%= resourceBundle.getString("dimanche") %></OPTION>
<OPTION VALUE=1><%= resourceBundle.getString("lundi") %></OPTION>
<OPTION VALUE=2><%= resourceBundle.getString("mardi") %></OPTION>
<OPTION VALUE=3><%= resourceBundle.getString("mercredi") %></OPTION>
<OPTION VALUE=4><%= resourceBundle.getString("jeudi") %></OPTION>
<OPTION VALUE=5><%= resourceBundle.getString("vendredi") %></OPTION>
<OPTION VALUE=6><%= resourceBundle.getString("samedi") %></OPTION>
</SELECT>
```

Widget calendrier

Le widget calendrier utilisé dans une page JSP doit être localisé. Pour ce faire, personnalisez le fichier JavaScript **I18N.js** situé dans le répertoire de paramètres régionaux.
***rép_base_debs*/Sterling//*la*/*CO*/js/**. Par exemple, pour prendre en charge les paramètres régionaux de_DE, créez un fichier appelé ***rép_base_debs*/Sterling/de/DE/js/I18N.js contenant les lignes suivantes :**

```
// Paramètres régionaux par défaut (anglais)
var MONTH_NAMES = new Array('Januar', 'Februar', 'Maerz', 'April', 'Mai',
'Juni', 'Juli', 'August', 'September', 'Oktober', 'November', 'Dezember',
'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Okt',
'Nov', 'Dez');
var DAYOFWEEK_HEADER_NAMES = new Array("So","Mo","Di","Mi","Do","Fr","Sa");
var WEEK_START_DAY = 0;
// Créer l'objet CalendarPopup
var popupCal = new CalendarPopup();
```

Feuilles de style

Visual Modeler utilise des feuilles de style en cascade pour définir le formatage des éléments HTML. Si vous utilisez des polices de caractères spécifiques des paramètres régionaux, assurez-vous que ces polices sont indiquées dans une feuille de style. Utilisez toujours le même emplacement relatif pour enregistrer les feuilles de style spécifiques aux paramètres régionaux.

Dans les pages JSP, vous pouvez inclure une feuille de style en cascade spécifique aux paramètres régionaux, par exemple **customer.css**, à l'aide de la ligne suivante :

```
<LINK rel="stylesheet" href="../css/customer.css" type="text/css">
```

Propriétés du système

En général, les fichiers de configuration ne présentent que des données aux administrateurs. Pour localiser ces fichiers, modifiez le texte d'aide des éléments sans modifier leur nom ou leur valeur. Notez qu'il existe un seul ensemble de fichiers de configuration par instance de Visual Modeler ; vous devez donc utiliser la langue des paramètres régionaux par défaut du système.

Regroupements de ressources et formats

Fichiers PropertyResourceBundles et Properties

Visual Modeler repose en grande partie sur des fichiers de propriétés pour la gestion des données spécifiques aux paramètres régionaux. Ces fichiers remplacent les classes Java ResourceBundle. Pour plus d'informations, voir «Présentation de Visual Modeler Localization», à la page 87.

Classes ResourceBundle

Les classes Java ResourceBundle sont un mécanisme utile pour la gestion de la localisation.

Remarque : L'utilisation des classes ResourceBundle dans Visual Modeler est obsolète. Vous devez désormais utiliser des fichiers de propriétés (voir «Présentation de Visual Modeler Localization», à la page 87).

Ces classes permettent de gérer les informations spécifiques aux paramètres régionaux. Toutes les classes ResourceBundle utilisées dans Visual Modeler étendent la classe ListResourceBundle. Elles définissent le mappage entre les chaînes de nom et les chaînes de valeur renvoyées lorsque la méthode *getString (String nameString)* est appelée.

Si vous respectez la convention d'attribution de noms pour les classes ResourceBundle, vous pouvez créer des classes ResourceBundle spécifiques à tous les paramètres régionaux pris en charge. Par exemple, vous pouvez créer les classes ResourceBundle suivantes pour les utiliser dans une nouvelle application appelée Inventory :

- InventoryResourceBundle
- InventoryResourceBundle_fr
- InventoryResourceBundle_fr_FR
- InventoryResourceBundle_fr_CA

Le scriptlet suivant permet de récupérer le regroupement de ressources à utiliser dans une page JSP :

```
<%
String baseName = "AdvisorResourceBundle";
ResourceBundle resourceBundle =
AdvisorResourceBundle.getBundle (baseName,
session.getLocale());
%>
```

Classes NumberFormat et DateFormat

Vous pouvez utiliser la classe NumberFormat pour afficher les nombres dans un format spécifique à un ensemble de paramètres régionaux. Pour créer une instance de la classe NumberFormat, passez les paramètres régionaux dans le constructeur.

Par exemple, le scriptlet suivant affiche le nombre total de paniers dans un format approprié pour les paramètres régionaux :

```
<%
NumberFormat numberFormat =
NumberFormat.getInstance(session.getLocale());
int number = request.getParameter("ShoppingCartsTotal");
%>
<P>Le nombre de paniers actifs en cours d'utilisation est :
<%= numberFormat.format(number) %>
</P>
```

De la même manière, vous pouvez utiliser la classe DateFormat pour afficher les dates dans un format spécifique à un ensemble de paramètres régionaux. Pour créer une instance de la classe DateFormat, passez les paramètres régionaux dans le constructeur.

Par exemple, le scriptlet suivant affiche la date du jour dans un format approprié pour les paramètres régionaux :

```
<%
DateFormat dateFormat =
DateFormat.getInstance(session.getLocale());
Date todaysDate = new Date();
%>
<P>Nous sommes le :
<%= dateFormat.format(todaysDate) %>
</P>
```

Chapitre 16. Personnalisation des contrôles

Les contrôles déterminent l'affichage et le comportement des classes d'option et des éléments d'option dans l'interface utilisateur. Vous pouvez modifier un contrôle existant ou ajouter un nouveau contrôle.

Chaque contrôle correspond à une page JSP et détermine le comportement des éléments d'option. Cette correspondance est définie dans le fichier de configuration `controls.properties`, sous le dossier `Comergent/WEB-INF/properties` situé dans le répertoire de déploiement.

Voici un exemple d'entrée définie dans le fichier `controls.properties` :

```
RADIO.name=Bouton radio
RADIO.jsp=controls/radio.jsp
RADIO.behavior=single
```

Dans cet exemple, pour le contrôle de bouton radio, la page JSP `radio.jsp` est utilisée pour afficher la classe d'option dans l'interface utilisateur. La propriété `behavior` indique comment les sélections pour ce contrôle seront gérées par Sterling Configurator. En fonction de la valeur de la propriété `behavior`, Sterling Configurator gère les sélections comme suit :

- entry - Valeur utilisée pour les contrôles entrés par l'utilisateur.
- expand - Développer tous les enfants de ce contrôle lorsque le contrôle est sélectionné.
- multiple - Autoriser la sélection d'un ou plusieurs éléments d'option de ce contrôle.
- single - Si un élément d'option est sélectionné, supprimer toute sélection antérieure de cette classe d'option.

Modification d'un contrôle

Pourquoi et quand exécuter cette tâche

Vous pouvez personnaliser un contrôle existant en modifiant l'entrée correspondante du fichier `controls.properties`.

Pour modifier un contrôle existant, procédez comme suit :

Procédure

1. Exécutez la cible suivante pour récupérer le fichier `controls.properties` que vous devez personnaliser :

   ```
   sdk customize WEB-INF/properties/controls.properties
   ```

   Cette commande place le fichier `controls.properties` dans votre projet de personnalisation.
2. Modifiez les entrées du fichier `controls.properties` selon vos besoins.
3. Exécutez la cible suivante pour fusionner les personnalisations :

   ```
   sdk merge
   ```
4. Si vous déployez l'application Visual Modeler en tant que fichier WAR, procédez comme suit :
   a. Exécutez la cible suivante pour recréer le fichier WAR :

```
sdk distWar
```

b. Déployez le fichier `.war` sur votre serveur d'applications.

Résultats

Une fois ces étapes terminées, vous devez effectuez les modifications requises dans Sterling Selling and Fulfillment Foundation. Pour plus d'informations, voir *Sterling Configurator: Application Guide*.

Ajout d'un contrôle

Pourquoi et quand exécuter cette tâche

Vous pouvez définir un nouveau contrôle en ajoutant le nom du contrôle à la liste de contrôles déclarés puis en définissant les propriétés du nouveau contrôle.

Pour ajouter un nouveau contrôle, procédez comme suit :

Procédure

1. Exécutez la cible suivante pour récupérer le fichier `controls.properties` que vous devez personnaliser :

   ```
   sdk customize WEB-INF/properties/controls.properties
   ```

   Cette commande place le fichier `controls.properties` dans votre projet de personnalisation.
2. Ajoutez le nom du nouveau contrôle à la liste séparée par des virgules de l'attribut `controls`.

   Par exemple, pour ajouter un nouveau contrôle appelé ABC_PERSO, vous pouvez définir l'attribut `controls` comme suit :

   ```
   controls=ABC_CUSTOM,RADIO,CHECKBOX,COMBOBOX,LISTBOX,MULTISELLISTBOX,ALLPICKED,UEV,DISPLAY
   ```
3. Définissez les propriétés du nouveau contrôle. Par exemple, vous pouvez définir les propriétés du nouveau contrôle ABC_PERSO comme suit :

   ```
   ABC_PERSO.name=Contrôle personnalisé de matrice
   ABC_PERSO.jsp=controls/ABCPerso.jsp
   ABC_PERSO.behavior=single
   ```
4. Exécutez la cible suivante pour fusionner les personnalisations :

   ```
   sdk merge
   ```
5. Si vous déployez l'application Visual Modeler en tant que fichier WAR, procédez comme suit :
   a. Exécutez la cible suivante pour recréer le fichier WAR :

      ```
      sdk distWar
      ```

   b. Déployez le fichier `.war` sur votre serveur d'applications.

Résultats

Une fois ces étapes terminées, vous devez effectuez les modifications requises dans Sterling Selling and Fulfillment Foundation. Pour plus d'informations, voir *Sterling Configurator: Application Guide*.

Chapitre 17. Personnalisation des gestionnaires de fonction

Les classes de gestionnaire de fonction sont des classes Java utilisées pour définir des fonctions personnalisées pouvant être appelées par le moteur de règles Sterling Configurator. Les classes de gestionnaire de fonction peuvent être personnalisées.

Les gestionnaires de fonction sont définis dans le fichier de configuration `functionHandlers.properties`, sous le dossier `Comergent/WEB-INF/properties` situé dans le répertoire de déploiement. Ce fichier comprend le nom de chaque gestionnaire de fonction, ainsi que le répertoire dans lequel se trouve la classe de gestionnaire de fonction.

Voici un exemple de fragment du fichier `functionHandlers.properties` :

```
WEB-INF/classes/com/comergent/apps/configurator/functionHandlers=
CheckLookupFunctionHandler,ChildSum,CountFunctionHandler,
IsSelectedHandler,LengthFunctionHandler,ListFunctionHandler,
LookupFunctionHandler,MaxFunctionHandler,MinFunctionHandler,
ParentFunctionHandler,PropValHandler,SumFunctionHandler,
ValueFunctionHandler,WebServiceLookupCheckLookupFunctionHandler=
com.comergent.apps.configurator.
function-Handlers.CheckLookupFunctionHandler
```

Ajout d'une classe de gestionnaire de fonction

Pourquoi et quand exécuter cette tâche

Vous pouvez ajouter une nouvelle classe de gestionnaire de fonction.

Pour ajouter une nouvelle classe de gestionnaire de fonction, procédez comme suit :

Procédure

1. Exécutez la cible suivante pour récupérer le fichier `functionHandlers.properties` que vous devez personnaliser :

   ```
   sdk customize WEB-INF/properties/functionHandlers.properties
   ```

   Cette commande place le fichier `functionHandlers.properties` dans votre projet de personnalisation.
2. Créez une nouvelle classe Java dans la déclaration du package com.comergent.apps.configurator.functionHandlers. La déclaration de classe doit indiquer qu'elle étend la classe AbstractRuleFunctionHandler.

   Remarque : La nouvelle classe Java doit être fournie dans le chemin d'accès aux classes de l'application Visual Modeler.

   La nouvelle classe Java doit implémenter les méthodes suivantes :

   - public String getFuncName() : Renvoie le nom de la fonction, par exemple "sum" ou "max". Cette méthode est sensible à la casse ; vous pouvez utiliser des gestionnaires de fonction distincts pour gérer les noms "sum" et "SUM".
   - public int getType() : Renvoie le type de valeur renvoyée par la fonction. Cette valeur doit être une constante définie dans la classe com.comergent.api.appsservices.rulesEngine.Value. La méthode de classe

AbstractRuleFunctionHandler renvoie un résultat au format Valeur.CHAINE. Vous devez donc remplacer cette méthode si la fonction renvoie un autre type de valeur.

- public Value handle(State state, String prop) : Renvoie la valeur calculée pour la fonction.
- public boolean isPublicHandler() : Renvoie la valeur true si le gestionnaire de fonction peut être utilisé par toutes les applications client ; dans le cas contraire, renvoie la valeur false. La méthode de classe AbstractRuleFunctionHandler renvoie la valeur true. Vous devez remplacer cette méthode uniquement si le gestionnaire de fonction est privé.

Résultats

Une fois ces étapes terminées, vous devez effectuez les modifications requises dans Sterling Selling and Fulfillment Foundation. Pour plus d'informations, voir *Sterling Configurator: Application Guide*.

Chapitre 18. Exceptions

Hiérarchie ComergentException

Classe racine d'exception

Cette section décrit les classes suivantes :

- ComergentException
- ICCException
- ComergentRuntimeException

ComergentException

Toutes les classes d'exception de compilation déclarées dans le logiciel de production doivent, en dernier ressort, hériter de la classe com.comergent.dcm.util.ComergentException. Cette classe étend la classe java.lang.Exception pour fournir le chaînage et générer un message utilisateur indépendant.

ICCException

ICCException fournit une sous-classe de la classe ComergentException. Plutôt que de créer un ensemble de classes d'exception pour un sous-système, vous pouvez utiliser la classe ICCException de manière uniforme dans ce sous-système.

ComergentRuntimeException

Toutes les classes d'exception d'exécution doivent hériter de la classe com.comergent.dcm.util.ComergentRuntimeException, qui étend la classe java.lang.RuntimeException pour fournir une fonction identique.

Groupement de sous-systèmes

Un sous-système de Visual Modeler peut être défini comme une application distincte et séparable, comme un niveau d'application ou ou comme service de niveau système. Un sous-système est une organisation logique. Il peut s'étendre sur plusieurs packages dans la hiérarchie de packages Java ou faire partie d'un package.

Chaque sous-système logique doit déclarer sa propre classe racine d'exception. Cette racine hérite de ComergentException et elle est la classe parent de toutes les exceptions de compilation produites dans le sous-système. Un sous-système peut être défini comme une application distincte et séparable, comme un niveau d'application ou ou comme service de niveau système. Un sous-système est une organisation logique. Il peut s'étendre sur plusieurs packages dans la hiérarchie de packages Java ou faire partie d'un package, mais en général la structure des packages doit respecter l'organisation des sous-systèmes logiques.

Prenons un sous-système appelé Foo. Il devrait comporter une classe `FooException` :

```
public class FooException extends ComergentException
{
public FooException(String msg)
{
super(msg);
}
public FooException(String msg, Exception ex)
{
super(msg, ex);
}
}
```

Supposons que Foo réponde à une erreur d'initialisation en lançant l'exception `BadInitializationException`. Cette exception héritera de `FooException`:

```
public class BadInitializationException extends FooException
{
...
}
```

Politique d'exception sous-système par sous-système

Chaque sous-système doit implémenter une politique de différenciation des exceptions cohérente. Il doit créer une sous-classe de la classe d'exception du sous-système pour chaque type d'exception distinct (politique Java standard) ou l'exception racine du sous-système doit hériter de l'exception ICCException ; par ailleurs, il doit définir le paramètre d'état de façon à établir une distinction entre les différentes exceptions (politique ICCException).

Par exemple, si le sous-système Foo utilise la politique d'exception Java FooException doit étendre ComergentException. Si le sous-système Bar utilise la politique ICCException, FooException doit étendre ICCException, qui étend à son tour ComergentException.

```
public class BarException extends ICCException
{
...
}
```

Chaînage des exceptions

Chaque sous-système devrait lancer à son appelant des exceptions provenant uniquement de son propre sous-système. Si un service sous-jacent lance une exception qui ne peut pas être gérée par un sous-système donné, le sous-système doit lever cette exception et lancer une autre exception significative dans son propre contexte. La nouvelle exception doit inclure l'exception initiale à l'aide d'un constructeur de chaînage, afin qu'elle ne soit pas perdue lorsque l'exception est finalement gérée et consignée.

Par exemple, supposons que le sous-système Foo tente d'ouvrir un fichier de propriétés, ce qui peut provoquer une exception d'entrée-sortie. S'il implémente une politique d'exception de type Java, il peut déclarer une nouvelle classe d'exception appelée FooPropertyFileException, qui étend FooException. L'instruction catch IOException lancerait alors une nouvelle exception FooPropertyFileException, à l'aide d'un constructeur qui transmet un message et l'exception d'entrée-sortie initiale.

```
try
{
...
Properties props = new Properties();
props.load(input);
...
}
catch (IOException, p. ex.)
{
// Chaînage de l'exception d'entrée-sortie
throw new FooPropertyFileException("Chargement du fichier" + nom_fichier,
p. ex.);
}
```

Lancement, levée et consignation d'exceptions

Situations qui justifient le lancement d'exceptions

Des exceptions doivent être lancées lorsque le contrat entre une méthode et l'appelant ne peut pas aboutir. Cette utilisation est définie dans la spécification du langage Java. Cependant, cette spécification n'est pas suffisamment détaillée, parce que l'on peut définir un contrat très général, dans lequel les exceptions seraient inutiles, ou, au contraire, un contrat très limité où les exceptions seraient générées très souvent. En règle générale, l'utilisation d'exceptions devrait essayer de concilier ces deux objectifs opposés :

Les exceptions ne doivent pas constituer une norme.

- Elles impliquent la création d'un objet supplémentaire, ce qui signifie que plus les exceptions sont fréquentes, moins les performances sont bonnes.
- Le mélange de données et de contrôles doit être évité. Le renvoi de la valeur null d'une méthode est une alternative au lancement d'une exception. La valeur renvoyée renferme deux significations : la réussite ou l'échec de la demande, ainsi que la signification des données renvoyées, le cas échéant. Les programmeurs devraient éviter autant que possible cette utilisation.
- Si la valeur null convient au but établi d'une méthode ou si vous prévoyez qu'une méthode échoue souvent dans le cadre d'un fonctionnement normal, il est raisonnable de renvoyer la valeur null pour indiquer l'échec de la méthode ; dans le cas contraire, il est préférable de lancer une exception.

Lancement d'exceptions d'exécution ou de compilation

Selon la spécification du langage Java, des exceptions d'exécution doivent être lancées si l'appelant a fourni une entrée non valide (autrement dit, il a enfreint le contrat de la méthode) et si la déclaration d'une exception de compilation serait une opération fastidieuse. Par exemple, si un appelant appelle une méthode en passant une valeur négative pour un paramètre d'indice de tableau, il convient de lancer une exception d'exécution. Dans les autres cas, les exceptions de compilation sont plus appropriées.

Clauses catch et déclarations throws

Les clauses catch et les déclarations throws ne doivent pas être trop générales. Si la méthode appelée lance, par exemple, l'exception FileNotFoundException, l'appelant doit lever l'exception FileNotFoundException, et non Exception ou Throwable. En effet, si le code sous-jacent est modifié pour lancer une nouvelle exception ou cesse

de lancer cette exception, il est souhaitable que la modification génère une erreur de compilation pour signaler cette nouvelle situation au programmeur.

Toutefois, des dérogations à cette règle sont possibles dans certains cas exceptionnels où le côté pratique doit prévaloir. Si un grand nombre d'exceptions peuvent être lancées et que la réponse fournie est identique dans tous les cas, il n'y a aucune raison de lever individuellement chaque exception.

Exceptions de consignation

Toute exception levée et gérée par une méthode (c'est-à-dire non relancée) doit être consignée par cette méthode. La méthode connaît normalement la signification de l'exception et est donc capable de la consigner avec le niveau de gravité Erreur ou un niveau de gravité inférieur. Les instructions catch vides doivent être considérées avec beaucoup de méfiance.

N'entrez jamais :

```
catch (ExempleException, p. ex.)
{
}
```

Mais :

```
catch (ExempleException, p. ex.)
{
Global.logVerbose(ex);
}
```

Ou encore :

```
catch (ExempleException, p. ex.)
{
ex.printStackTrace(Global.debugStream);
}
```

Il est inutile de consigner les exceptions provenant de sous-systèmes sous-jacents ou de packages tiers, qui sont détectées et enchaînées à une nouvelle exception. En remontant dans la hiérarchie, un processus va finalement lever, gérer et consigner ces exceptions.

Affichage des exceptions

En général, les exceptions ne devraient pas être visibles pour les utilisateurs de Visual Modeler ; elles doivent être gérées normalement par le sous-système approprié, en envoyant une réponse à la condition d'erreur.

Dans les pages d'erreur de Visual Modeler, la pile d'exception est placée entre des commentaires HTML. Vous pouvez lire la trace de pile en affichant la source de la page Web.

Si une trace de pile d'exception est envoyée à la page JSP, gardez à l'esprit que le message d'exception ne peut peut-être pas être envoyé intégralement à la page Web en raison des limites de la mémoire tampon de la page JSP. Si une longue trace de pile d'exception est envoyée à la page JSP, vous pouvez l'afficher en modifiant la taille de la mémoire tampon de la page JSP. Utilisez la balise buffer comme suit :

```
<%@ page buffer=1024kb %>
```

Une fois que vous avez diagnostiqué et corrigé la condition d'erreur, vous devez supprimer cette balise car elle entraîne une dégradation des performances.

Chapitre 19. Travaux cron

Implémentation de travaux cron dans Visual Modeler

Certaines tâches effectuées dans le cadre de l'implémentation de Visual Modeler ne sont pas lancées en réponse à une action de l'utilisateur. Par exemple, il est préférable que les données sur les commandes soient synchronisées toutes les heures avec un système externe ou que les données de catalogue soient importées toutes les semaines à partir d'un tiers sans intervention de l'utilisateur. Vous pouvez planifier l'exécution de ces travaux à des intervalles appropriés à l'aide de la fonction de planificateur de travaux fournie par Visual Modeler.

Les travaux cron peuvent être divisés en travaux cron système et travaux cron d'application.

- Un travail cron système est exécuté par Visual Modeler et n'est associé à aucun utilisateur. Il appelle directement les classes Visual Modeler. Un travail cron système doit être exécuté par une classe qui étend la classe abstraite SystemCron. En général, les travaux cron système exécutent des tâches telles que le nettoyage du cache.
- Chaque travail cron d'application est exécuté en tant qu'utilisateur ; le nom utilisateur et le mot de passe de l'utilisateur sont fournis lors de la création du travail cron à l'aide de l'interface du planificateur de travaux. Le fonctionnement des travaux cron d'application est basé sur la publication de messages XML dans Visual Modeler, qui sont ensuite traités par le système. Un travail cron d'application doit être exécuté par une classe qui étend la classe abstraite ApplicationCron. En général, les travaux cron d'application sont utilisés pour effectuer des tâches administratives qui impliquent des données utilisateur ou produit (par exemple, la synchronisation des commandes).

  Remarque : Un travail cron système ne doit pas tenter de lancer des opérations *restore()* et *persist()*. Dans ce cas, le contrôle d'accès intégré dans les méthodes d'accès aux données lancerait une exception, car aucun utilisateur n'est associé à la classe du travail cron.

CronManager et CronScheduler

La définition et la création de travaux cron sont gérées par la classe CronManager. Les informations de configuration des travaux cron sont représentées dans la mémoire par le bean de données CronConfigBean. La définition des travaux cron est conservée dans Knowledgebase.

La planification et l'exécution des travaux cron sont gérées par la classe CronScheduler. Cette classe singleton est instanciée au démarrage du serveur.

Interface CronJob

Chaque travail cron est une classe Java qui implémente l'interface CronJob :

```
public interface CronJob extends java.lang.Runnable
{
/**
* Indiquer l'objet de bean Cron Configuration.
```

```
*
* @param config L'objet de bean Cron Configuration.
*/
public void setCronConfiguration(CronConfigBean config);
/**
* Renvoyer l'objet de bean Cron Configuration.
*
* @return L'objet CronConfigBean.
*/
public CronConfigBean getCronConfiguration();
/**
* Fonction d'initialisation. Cette fonction est appelée
* immédiatement après la création de l'objet.
*
* @return True si l'initialisation réussit, false dans le cas contraire.
*/
public boolean init();
/**
* Renvoie l'heure planifiée actuelle.
*
* @return L'heure planifiée en cours dans l'objet Calendar.
*/
public Calendar getSchedule();
/**
* Replanifier le travail cron pour refléter les modifications apportées
* au paramètre de configuration. Cette fonction est appelée par
* le gestionnaire de travaux cron chaque fois que la configuration du
travail cron est modifiée.
*/
public void reschedule();
/**
* Indique si le travail doit être réexécuté. Cette fonction est
* utile si l'exécution actuelle présente un problème et
* et que vous souhaitez essayer de réexécuter le travail à l'heure
indiquée.
*
* @return True si le travail peut être réexécuté
* s'il n'a pas pu être effectué
* lors de la dernière exécution
*/
public boolean retry();
/**
* Indiquer s'il faut arrêter l'exécution de ce travail cron.
*
* @return True si ce travail ne doit pas être réexécuté
*/
public boolean stopRun();
/**
* Calculer l'heure de la prochaine exécution du travail cron. En général,
cette heure est basée sur
* l'intervalle d'exécution du travail cron.
*/
public void computeNextSchedule();
/**
* Vérifier si le travail cron est
* prêt à être exécuté correctement avant de déclencher le démarrage de
```

```
l'unité d'exécution.
*
* @return True ou false. La valeur true signifie que le travail cron est
prêt à être exécuté.
*/
public boolean isOKtoRun();
/**
* Est appelé lors du démarrage de l'unité d'exécution.
*
* @return False si le travail doit être arrêté. True si
* l'exécution doit continuer.
*/
public boolean service();
/**
* Vérifier si l'heure de la prochaine exécution est postérieure à l'heure
de fin de l'exécution en cours.
*
* @return True si l'heure de la prochaine exécution est postérieure à
l'heure de fin de l'exécution en cours.
*/
public boolean isExpired();
}
```

Création d'un travail cron dans Visual Modeler

Pourquoi et quand exécuter cette tâche

Pour créer un nouveau travail cron, procédez comme suit :

Procédure

1. Créez une classe CronJob en étendant la classe SystemCron ou ApplicationCron. Ces deux classes sont abstraites et étendent la classe abstraite AbstractCronJob.

   La seule méthode que vous devez implémenter est *service()*. Celle-ci permet de traiter la publication entrante lancée par CronScheduler.

   - Si les paramètres passés dans le travail sont définis via l'interface utilisateur du planificateur de travaux, vous pouvez récupérer les paramètres à l'aide des méthodes *getParameter(String s)* et *getParameters()* de la classe AbstractCronJob. Ces méthodes se comportent de la même façon que les méthodes correspondantes de la classe HttpServletRequest.
   - Si vous souhaitez enregistrer le résultat du travail dans la base de données, la méthode *service()* doit appeler la méthode *setExecutionOutcome(String s)*.
   - Vous pouvez indiquer que le travail cron doit être réexécuté ultérieurement en appelant la méthode *setRetry(Calendar c)* de la classe AbstractCronJob. Utilisez le paramètre Calendar pour spécifier la date de réexécution du travail.

2. À l'aide de l'interface utilisateur du planificateur de travaux fournie avec l'application d'administration de système, définissez le travail cron en indiquant sa classe, sa planification d'exécution, ainsi que tous les paramètres qui doivent être passés dans travail cron lors de l'exécution. Si le travail cron doit être exécuté en tant que travail cron d'application, vous devez également fournir le nom d'utilisateur et le mot de passe de l'utilisateur.

   La syntaxe pour passer des paramètres dans le travail cron est la même que pour les demandes HTTP. Par exemple : Nom1=Valeur1&Nom2=Valeur2.

Chapitre 20. Présentation des filtres

Un filtre est un objet qui effectue des tâches de filtrage sur la demande envoyée à une ressource (un servlet ou un contenu statique), sur la réponse fournie par une ressource, ou les deux. Les filtres sont définis dans le cadre de la spécification J2EE 2.3.

Le filtrage est exécuté à l'aide de la méthode *doFilter()*. Chaque filtre a accès à un objet FilterConfig, à partir duquel il peut obtenir ses paramètres d'initialisation ; il s'agit d'une référence au contexte de servlet qu'il peut utiliser, par exemple, pour charger les ressources requises pour les tâches de filtrage.

Les filtres sont configurés dans le descripteur de déploiement d'une application Web. Voici quelques exemples typiques de filtres :

- Filtres d'authentification
- Filtres de consignation et d'audit
- Filtres de conversion d'images
- Filtres de compression de données
- Filtres de chiffrement
- Filtres de segmentation
- Filtres qui déclenchent des événements d'accès aux ressources
- Filtres XSLT
- Filtres de chaîne de type MIME

Chapitre 21. Filtres Visual Modeler

Visual Modeler fournit les filtres suivants, inclus dans le package com.comergent.dcm.core.filters :

- «DosFilter»
- «WSDLFilter»

DosFilter

Ce filtre peut être utilisé comme filtre de base pour protéger l'application contre les attaques par saturation.

Pour utiliser ce filtre, créez une classe qui étend la classe com.comergent.dcm.core.filters.DosFilter ; dans cette classe, remplacez la méthode *isRequestDenied()* pour implémenter la logique que vous souhaitez utiliser pour identifier et bloquer les attaques par saturation.

Ensuite, modifiez le fichier **web.xml** pour déclarer votre classe d'implémentation en tant que filtre :

```
<filter>
<filter-name>DosFilter</filter-name>
<filter-class>
com.comergent.dcm.messaging.CustomDosFilter
</filter-class>
</filter>
```

```
et
<filter-mapping>
   <filter-name>DosFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

WSDLFilter

La classe WSDLFilter est utilisée pour transformer les WSDL du service Web, si l'accès aux WSDL se fait via des URL standard, par exemple http://serveur:port/s/dXML/5.0/OrderInterface.wsdl.

Chapitre 22. Gestion et affichage des zones restreintes

Les zones de données restreintes ne peuvent prendre qu'une valeur parmi plusieurs. C'est par exemple le cas des niveaux de partenariat ("Gold", "Silver", etc.), de l'implantation géographique des partenaires ("Nord-Ouest", "Benelux", etc.) et des niveaux de compétences ("Expert", "Certifié", etc.). Ces zones de données peuvent être gérées de plusieurs manières dans Visual Modeler. Ce choix dépend de l'emplacement où vous souhaitez les stocker et de l'utilisation que vous voulez en faire.

Options

Les options suivantes sont disponibles pour spécifier une zone de données restreinte et les valeurs autorisées pour cette zone :

- Stocker la zone de données comme un ensemble de valeurs dans une table de base de données. Affecter des valeurs aux objets métier à l'aide d'une table de références croisées ou en définissant des références à une clé pour chaque valeur de la table d'objets métier.
- Stocker les valeurs dans l'élément constraint du schéma XML (déclaré dans le fichier **DsConstraints.xml**). Indiquer la contrainte comme attribut de l'élément DataElement associé à la zone de données.
- Définir les valeurs autorisées en tant que valeurs de l'élément de masque <SELECT> dans un modèle HTML.

Il est recommandé de stocker les valeurs autorisées d'une zone de données dans une table de base de données, sauf si :

- les valeurs ne seront pas être modifiées lors de l'exécution
- la zone de données ne peut prendre qu'une valeur dans chaque objet métier
- les valeurs peuvent être affichées dans un ordre intrinsèque, déterminé par les valeurs elles-mêmes (par exemple, par ordre alphabétique).

L'utilisation de la troisième option est déconseillée pour les raisons suivantes :

- La mise à jour des modèles ou du code d'application pose un problème de maintenance si vous souhaitez modifier la liste de valeurs de données autorisées.
- Cette option pose aussi un problème de sécurité, car les utilisateurs peuvent modifier le code HTML pour utiliser des valeurs interdites. Vous devez ajouter un script JavaScript (qu'un utilisateur peut supprimer) permettant de valider la sélection ou valider la valeur renvoyée dans le cadre de la logique métier.

Critères

Votre choix doit se baser sur le fonctionnement de la zone de données. Vous devez vous poser les questions suivantes relatives à l'utilisation de la zone de données :

1. Une zone de données restreinte peut-elle prendre une ou plusieurs valeurs dans un objet métier ?

   Si vous souhaitez que plusieurs valeurs puissent être affectées au même objet métier (par exemple, un partenaire qui peut être présent dans plusieurs régions), vous devez *obligatoirement* définir les valeurs de la zone dans une table de base de données et définir les valeurs de l'objet métier dans une table de références croisées.

2. Lors de la création d'un nouvel objet métier, pouvez-vous entrer de nouvelles valeurs pour la zone de données ou devez-vous vérifier qu'une la valeur entrée pour la zone est un membre valide de l'ensemble de contraintes ?

   Si seules les valeurs uniques sont autorisées et que vous souhaitez qu'il soit possible d'entrer de nouvelles valeurs, vous devez *obligatoirement* utiliser une table de base de données pour stocker les valeurs de la zone. Cependant, vous ne devez pas utiliser une table de références croisées pour affecter des valeurs de zones de données aux objets métier. L'interface actuelle de Visual Modeler ne permet pas d'ajouter dynamiquement des valeurs à la liste de valeurs autorisées d'un élément constraint.

   Les valeurs possibles de la zone de données restreinte sont-elles stockées dynamiquement ou sont-elles lues une seule fois au démarrage ?

3. Si vous avez répondu que seules les valeurs uniques sont autorisées (Question 1) et que les nouvelles valeurs ne sont pas autorisées (Question 2) mais vous devez exécuter une mise à jour dynamique, vous devez *obligatoirement* utiliser une table de base de données. Si les valeurs de la zone restreinte ne sont pas modifiées après le démarrage de Visual Modeler, vous pouvez utiliser un élément constraint.

   Avez-vous besoin de trier les valeurs affichées pour les zones restreintes ? Si votre réponse est oui, souhaitez-vous qu'elles soient triées de manière intrinsèque (par exemple, par ordre alphabétique) ou dans un autre ordre qui ne peut pas être déduit des valeurs proprement dites ?

4. Si les valeurs de la zone de données doivent être triées dans un ordre qui n'est pas lié aux valeurs proprement dites, vous devez fournir les informations concernant cet ordre de tri dans une table de base de données. En revanche, si vous souhaitez trier les valeurs selon un ordre intrinsèque (par exemple, par ordre alphabétique), vous pouvez utiliser l'élément constraint.

Index

A

AbstractCronJob, classe 111
abstraite BLC, classe 80
ACTIVE_FLAG, colonne 54
  utiliser pour marquer des objets comme supprimés 53
addChild, méthode 64
adjustFileName, méthode 20, 27, 28
Alternate, élément 62
analyse rétrospective 68
API log4j 67
API Preferences 30
AppContextCache, classe 28
AppExecutionEnv, classe 19, 26
ApplicationCron, classe 109, 111
AppsLookupHelper, classe 27
attributs
  DataService 62
  DataSourceName 61
  ExternalFieldName 60
  ID 24
  IsOverlay 19
  MaxPoolSize 26
  Name 19, 59, 60
  Version 59, 66
attributs de contexte sérialisables 20
autorisations d'accès 51

B

bean entity 51
beans application 23, 51, 52
beans présentation 51
bibliothèques de balises 12
bibliothèques de balises personnalisées 12
bizAPI, classes 79
Bizlet, classe 19
BizletMapping
  valeur par défaut pour le groupe de messages 20
BizletMapping, élément 19
BizRouter, classe 19
bundle, attribut 89
BusinessObject, classe 66

C

C3PrimaryRW, objet de données 47
calendrier 96
calendrier, widget
  localisation 96
callJSP, méthode 29
caractères codés sur plusieurs octets 92
ChildDataObject, élément 56
children, méthode 64
cibles
  generateBean 23, 47, 51, 62, 77
  generateDTD 47
classe de gestionnaire de fonction
  ajout 101
classes 22
  AbstractCronJob 111
  AppExecutionEnv 19, 26
  ApplicationCron 109, 111
  Bizlet 19
  BizobjBean 51
  BizRouter 19
  BusinessObject 66
  ComergentAppEnv 21, 27
  ComergentContext 20
  ComergentDispatcher 20
  ComergentException 103
  ComergentRequest 21
  ComergentResponse 21
  ComergentRuntimeException 103
  ComergentSession 21
  CronConfigBean 109
  DataBean 23
  DataContext 48
  DataManager 61, 63
  DataMap 64
  DataService 62
  DebsDispatchServlet 22
  DispatchServlet 17, 21
  DsElement, élément 64
  Env 20
  Exception 103
  GeneralObjectFactory 22
  HttpRequest 21
  HttpResponse 21
  HttpServletRequest 111
  HttpSession 21
  ICCException 103
  InitServlet 17, 21, 28
  MessagingController 22, 23
  métadonnées 64
  NamingManager 80
  NamingResult 81
  NamingServiceDatabase 80
  NamingServiceProperties 80
  ObjectManager 23, 45, 47
  OMWrapper 23, 45
  RequestDispatcher 20
  ResourceBundle 97
  RuntimeException 103
  SimpleController 23
  SystemCron 109, 111
classes de contrôleur 22
classes de logique métier 47, 79
  implémentation 79
ClassName, élément 24, 25
cloneDsElement, méthode 64
CMGT_LOOKUPS, table 27
cmgtText, méthode 89
codes de recherche 27, 31
  mappage aux chaînes 27
com.comergent.api.dataservices, package 35
com.comergent.api.dispatchAuthorization, package 40
com.comergent.api.msgservice, package 42
com.comergent.dcm.caf.controller.Controller, classe 22
com.comergent.dcm.core.filters, package 115
com.comergent.dcm.objmgr, package 26
com.comergent.dispatchAuthorization, package 40
com.comergent.msgservice, package 42
Comergent.xml, fichier de configuration 18
ComergentAppEnv, classe 21, 27
ComergentContext, classe 20
ComergentDispatcher, classe 20
ComergentHelpBroker, classe 35
ComergentI18N, classe 92
ComergentRequest, classe 21
ComergentResponse, classe 21
ComergentSession, classe 21
commande
  instanceof 52
comportement de basculement 91
configuration requise 5
contexte
  définition des attributs 20
contexte de servlet
  définition des attributs 20
contrôle d'interface utilisateur
  ajout, nouveau 100
  modification 99
Controller, classes
  dans le cadre de l'implémentation de référence 73
ControllerMapping
  valeur par défaut pour le groupe de messages 20
ControllerMapping, élément 19
ConverterFactory, classe 42
copyBean, méthode 53
createController, méthode 22
CronConfigBean, classe 109
CronJob, interface 109
CronManager, classe 109
CronScheduler, classe 109
customize, cible 83

D

DataBean, classe 23
DataContext, classe 48
  utilisée pour la restauration 51
DataField, élément 60
DataObject, élément 61
DataService, attribut 62
DataService, classe 62
DataServices.General.LimitDBResults, préférence 50
DataSourceName, attribut 61

dates 95
débogage des regroupements de ressources JSP 90
DebsDispatchServlet, classe 22
debug, méthode 67
debugJSPResouceBundle, élément 90
defaultSystemLocale, élément 87, 88, 91
defaultType, élément 81
delete, méthode 53, 64, 66
deleteChild, méthode 64
demandes 79
descripteur de bibliothèque de balises 12, 18
devises 87, 95
disableAccessCheck, méthode 55
DispatchServlet, classe 21
doFilter, méthode 113
DosFilter, classe 115
DsElement 63
DsElement, arborescence 63
  applications existantes uniquement 63
DsElement, élément
  enfant 63
  parent 63
  racine 63
DsQuery, classe 55
  utilisée pour la restauration 51

E

élément defaultCountry 91
élément JoinKey 57
éléments
  Alternate 62
  BizletMapping 19
  ControllerMapping 19
  DataElements 61
    réutilisation 61
  DataField 60, 61
  DataObject 61
  defaultSystemLocale 87
  ExternalName 52
  GeneralObjectFactory 18
  globalCacheImplClass 28
  JSPMapping 19
  MessageType 19
  messageTypeFilename 18
  Primary 62
  propertiesFile 17
éléments de scriptage 12
enfant, objets de données 56
EntitlementFactory, classe 40
Env, classe 20
environnement groupé 28
erase, méthode 53
error, méthode 67
exceptions 103
  affichage 106
exemples de code
  utilisation de fichiers de propriétés de paramètres régionaux 92
Extends, attribut 47
ExternalFieldName, attribut 60
ExternalName, élément 52

F

fatal, méthode 67
feuilles de style en cascade 96
fichier de configuration
  DsDataElements.xml
    définir la longueur des zones de données 93
fichier de configuration
  Internationalization.xml 90, 91
fichier de configuration
  log4j.properties 67
fichiers de configuration 5, 11
  Comergent.xml 17, 18
  DsBusinessObjects.xml 59
  DsConstraints.xml 117
  DsRecipes.xml 59
  Internationalization.xml 87
  MessageTypes.xml 18, 22
  ObjectMap.xml 24
  web.xml 11, 12, 17
fichiers de déploiement
  Sterling.war 17
filtres
  filtres J2EE 113
findPresentationLocale, méthode 92

G

GeneralObjectFactory, classe 22
GeneralObjectFactory, élément 18
generateBean, cible 23, 47, 51, 62, 77
generateDTD, cible 47
generateKeys, méthode 53
get, méthode 81
getAllowedValueIterator, méthode 65
getBizObj, méthode 56
getBoolean, méthode 30
getCacheId, méthode 49
getComergentLocale, méthode 92
getCountAllowedValues, méthode 65
getDataBean, méthode 52
getDataType, méthode 64
getDefaultLocale, méthode 92
getDefaultValue, méthode 65
getDouble, méthode 30
getElementByName, méthode 64
getFloat, méthode 30
getInstance, méthode 80
getInt, méthode 30, 41
getIRdProduct, méthode 52
getLong, méthode 30
getMaxCharLength, méthode 64
getMaxLength, méthode 64
getMaxPaginatedResult 49
getMaxResults, méthode 49
getMaxValue, méthode 65
getMetaData, méthode 64
getMinValue, méthode 65
getName, méthode 64
getNumPerPage, méthode 49
getObject, méthode 24
getParameter, méthode 111
getParameters, méthode 111
getParent, méthode 64
getPreferences, méthodes 30
getRealPath, méthode 28
getResourceAsStream, méthode 20
getRootElement, méthode 63, 64, 66
getSession, méthode
  ComergentSession, classe 21
getString, méthode 30
getType, méthode 64, 66
Global, classe
  remplacée par Preferences 28
  utilisation obsolète pour la consignation 67
GlobalCache, interface 28
groupes de messages 18
  utilisé pour indiquer les mappages par défaut 20

H

HttpRequest, classe 21
HttpResponse, classe 21
HttpServletRequest, classe 111
HttpSession, classe 21

I

IAcc, interface 54
id, attribut
  utilisé dans la balise text 89
ID, attribut 24, 25
IData, interface 53, 54
  accès aux métadonnées 64
IMetaData, interface 64
info, méthode 67
InitManager, classe 36
InitServlet, classe 21
instanceof, commande 52
interface sérialisable 21
interfaces
  GlobalCache 28
  IAcc 54
  IData 53
  IRd 54
  NamingService 80
  pouvant être placées dans un pool 26
internationalisation
  feuilles de style en cascade 96
  mécanisme de basculement des pages JSP 91
  mécanisme de basculement des regroupements de ressources 91
Internationalization.xml, fichier de configuration 87
IRd, interface 54
IsOverlay, attribut 19
isPersistable, méthode 54
isRequestDenied, méthode 115
IsRestorable, méthode 54

J

J2EE 11
Java 2 Platform Enterprise Edition 11
JSPMapping
  valeur par défaut pour le groupe de messages 20
JSPMapping, élément 19, 91

K

kit de développement de logiciels 83
Knowledgebase 109

L

langues 87
LegacyFileUtils, classe 20, 28
LegacyPreferences, classe 28
liste, objets métier 51
localisation
  images 94
  JavaScript 94
localRedirect, méthode 21
log, méthode 67
logLevel, méthodes 67
logout, méthode 21
longueur des zones de données 92

M

masques d'URL
  mappage aux servlets 12
MaxPoolSize, attribut 26
MaxResults, élément 48
mécanisme de basculement des pages JSP 91
mécanisme de basculement des regroupements de ressources 91
messages 79
messages XML 22
MessageType, élément 19
  éléments enfants 19
messageTypeFilename, élément 18, 19
MessageTypeRef, élément 19
MessageTypes.xml, fichier de configuration 18
MessagingController 22
MessagingController, classe 22, 23
MessagingServlet, classe 18
métadonnées
  pour les zones de données 64
méthodes
  addChild 64
  adjustFileName 27
  calculate 23
  children 64
  cloneDsElement 64
  constructExternalURL 27
  copyBean 53
  createController 21, 22
  delete 53, 64, 66
  deleteChild 64
  dispatch 21
  erase 53
  forward 20
  generateKeys 53
  get 81
  getContext 28
  getDataBean 52
  getElementByName 64
  getEnv 27
  getInstance 80
  getName 64
  getObject 24
  getParameter 111
méthodes *(suite)*
  getParameters 111
  getParent 64
  getPartnerKey 21
  getRootElement 63, 64, 66
  getType 64, 66
  getUser 21
  getUserKey 21
  include 20
  init 22, 28
  isPersistable 54
  IsRestorable 54
  persist 23, 53, 54, 56, 62, 65, 80
  prune 53
  reset 26
  restore 23, 53, 54, 55, 62, 65, 80
  return 26
  runAppJob 19
  runAppObj 26
  service 80, 111
  setCacheId 48
  setDataContext 53
  setRetry 111
  setRootElement 66
  update 53
méthodes d'accès
  effet de l'attribut Writable 54
méthodes de consignation
  debug 67
  error 67
  info 67
  log 67
  warning 67
méthodes setExecutionOutcome 111
modèle de fabrique 23
modèles d'e-mail 93
  emplacement 94
MsgContext, interface 42
MsgService, interface 42
MsgServiceException, classe 42
MsgServiceFactory, classe 42

N

Name, attribut 19, 59, 60
NamingManager, classe 80
NamingResult, classe 81
NamingServiceDatabase, classe 80
NamingServiceProperties, classe 80
newproject, cible 83
NumPerCachePage, élément 48

O

Object, élément 24, 25
ObjectManager, classe 23, 45, 47
ObjectMap.xml, fichier de configuration 24
objets, pool 26
objets de données 47
  accès aux objets de données enfants 56
  extension 25, 47
  ordinalité 46
  personnalisation 47
  procédures stockées 52
objets métier
  listes 51
  Utilisateur 21
OMWrapper, classe 23, 45
org.apache.log4j.Level, classe 38
OutOfBandHelper, classe 29

P

package com.comergent.reference.jsp 89
packages
  com.comergent.dcm.objmgr 26
pages JSP 11
  dans le cadre de l'implémentation de référence 73
  débogage de la localisation 90
  localisation 95
  mémoire tampon de la page 106
  utilisées dans les modèles de courrier électronique 29
paramètres régionaux
  paramètres régionaux préférés 87
  présentation 88
  session 88
paramètres régionaux par défaut
  mécanisme de basculement 91
persist, méthode 23, 53, 54, 56, 62, 65, 80
  appeler après la méthode delete 53
placer dans un pool, interface 26
plusieurs formats de nombre et de date 87
plusieurs jeux de caractères 87
polices de caractères 96
pools d'objet 26
présentation, paramètres régionaux 88
Primary, élément 62
procédures stockées 52
propriété système log4j.debug 67
prune, méthode 53
putInt, méthode 41
putString, méthode 30

R

recettes 47
Recipe, élément
  déclaration de l'ordinalité 51
redirection d'une demande 21
regroupements de ressources 90
Relationship, élément 57
répartiteur de demande 12
représentations XML des beans de données 56
RequestDispatcher, classe 20
reset, méthode 26
restore, méthode 23, 53, 55, 62, 65, 80
  exemple d'utilisation de DataContext et de DsQuery 55
  procédures stockées 52
  utilisée dans les beans de liste 51
return, méthode 26
rôles 5
runAppJob, méthode 19

S

schéma XML 63
schemaRepositoryExtn, élément 83
scriptlets 12, 13
SDK 83
sécurité 5
service, méthode 42, 80, 111
service d'attribution de noms 80
session, paramètres régionaux 88
setAttribute, méthode
  ComergentSession, classe 21
setCacheId, méthode 48, 49
setDataContext, méthode 53
setExecutionOutcome, méthode 111
setMaxPaginatedResult 49
setMaxResults, méthode 49
setNumPerPage, méthode 49
setRetry, méthode 111
setRootElement, méthode 66
SimpleController, classe 23
SourceType, attribut 52
sous-système 103
SystemCron, classe 109, 111

T

text, balise 89
TLD, voir descripteur de bibliothèque de balises 12
Transaction, classe 31
type de contenu 22
types de message 18
types de recherche 27, 31

U

Unicode, prise en charge 87
update, méthode 53
usecountry ou regionDefaulting, élément 88
useCountryDefaulting, élément 88, 91
useGeneralDefaulting, élément 88, 91
utilisateurs 21
  récupérer à partir de la session 21
utilisation de la méthode restore dans les beans de liste 51
utilisation des pages JSP en tant que modèles 29

V

Version, attribut 66

W

warning, méthode 67
web.xml, fichier de configuration 115
Writable, attribut 54
WritableDirectory, élément 29
writeExternal, méthode 56
WSDLFilter, classe 115

Z

zones de données, métadonnées 64

Remarques

Le présent document peut contenir des informations ou des références concernant certains produits, logiciels ou services IBM non annoncés dans ce pays. Pour plus de détails, référez-vous aux documents d'annonce disponibles dans votre pays, ou adressez-vous à votre partenaire commercial IBM. Toute référence à un produit, logiciel ou service IBM n'implique pas que seul ce produit, logiciel ou service puisse être utilisé. Tout autre élément fonctionnellement équivalent peut être utilisé, s'il n'enfreint aucun droit d'IBM. Il est de la responsabilité de l'utilisateur d'évaluer et de vérifier lui-même les installations et applications réalisées avec des produits, logiciels ou services non expressément référencés par IBM.

IBM peut détenir des brevets ou des demandes de brevet couvrant les produits mentionnés dans le présent document. La remise de ce document ne vous donne aucun droit de licence sur ces brevets ou demandes de brevet. Si vous désirez recevoir des informations concernant l'acquisition de licences, veuillez en faire la demande par écrit à l'adresse suivante :

IBM Director of Licensing

IBM Corporation

North Castle Drive

Armonk, NY 10504-1785

U.S.A.

Les informations sur les licences concernant les produits utilisant un jeu de caractères double octet peuvent être obtenues par écrit à l'adresse suivante :

Intellectual Property Licensing

Legal and Intellectual Property Law

IBM Japan Ltd.

1623-14, Shimotsuruma, Yamato-shi

Kanagawa 242-8502 Japan

Le paragraphe suivant ne s'applique ni au Royaume-Uni, ni dans aucun pays dans lequel il serait contraire aux lois locales : LE PRÉSENT DOCUMENT EST LIVRÉ "EN L'ÉTAT" SANS AUCUNE GARANTIE EXPLICITE OU IMPLICITE. IBM DÉCLINE NOTAMMENT TOUTE RESPONSABILITÉ RELATIVE À CES INFORMATIONS EN CAS DE CONTREFAÇON AINSI QU'EN CAS DE DÉFAUT D'APTITUDE À L'EXÉCUTION D'UN TRAVAIL DONNÉ. Certaines juridictions n'autorisent pas l'exclusion des garanties implicites, auquel cas l'exclusion ci-dessus ne vous sera pas applicable.

Le présent document peut contenir des inexactitudes ou des coquilles. Ce document est mis à jour périodiquement. Chaque nouvelle édition inclut les mises à jour. IBM peut, à tout moment et sans préavis, modifier les produits et logiciels décrits dans ce document.

Les références à des sites Web non IBM sont fournies à titre d'information uniquement et n'impliquent en aucun cas une adhésion aux données qu'ils contiennent. Les éléments figurant sur ces sites Web ne font pas partie des éléments du présent produit IBM et l'utilisation de ces sites relève de votre seule responsabilité.

IBM pourra utiliser ou diffuser, de toute manière qu'elle jugera appropriée et sans aucune obligation de sa part, tout ou partie des informations qui lui seront fournies.

Les licenciés souhaitant obtenir des informations permettant : (i) l'échange des données entre des logiciels créés de façon indépendante et d'autres logiciels (dont celui-ci), et (ii) l'utilisation mutuelle des données ainsi échangées, doivent adresser leur demande à :

IBM Corporation

J46A/G4

555 Bailey Avenue

San Jose, CA 95141-1003

U.S.A.

Ces informations peuvent être soumises à des conditions particulières, prévoyant notamment le paiement d'une redevance.

Le logiciel sous licence décrit dans ce document et tous les éléments sous licence disponibles s'y rapportant sont fournis par IBM conformément aux dispositions de l'ICA, des Conditions internationales d'utilisation des logiciels IBM ou de tout autre accord équivalent.

Les données de performance indiquées dans ce document ont été déterminées dans un environnement contrôlé. Par conséquent, les résultats peuvent varier de manière significative selon l'environnement d'exploitation utilisé. Certaines mesures évaluées sur des systèmes en cours de développement ne sont pas garanties sur tous les systèmes disponibles. En outre, elles peuvent résulter d'extrapolations. Les résultats peuvent donc varier. Il incombe aux utilisateurs de ce document de vérifier si ces données sont applicables à leur environnement d'exploitation.

Les informations concernant des produits non IBM ont été obtenues auprès des fournisseurs de ces produits, par l'intermédiaire d'annonces publiques ou via d'autres sources disponibles. IBM n'a pas testé ces produits et ne peut confirmer l'exactitude de leurs performances ni leur compatibilité. Elle ne peut recevoir aucune réclamation concernant des produits non IBM. Toute question concernant les performances de produits non IBM doit être adressée aux fournisseurs de ces produits.

Toute instruction relative aux intentions d'IBM pour ses opérations à venir est susceptible d'être modifiée ou annulée sans préavis, et doit être considérée uniquement comme un objectif.

Tous les tarifs indiqués sont les prix de vente actuels suggérés par IBM et sont susceptibles d'être modifiés sans préavis. Les tarifs appliqués peuvent varier selon les revendeurs.

Ces informations sont fournies uniquement à titre de planification. Elles sont susceptibles d'être modifiées avant la mise à disposition des produits décrits.

Le présent document peut contenir des exemples de données et de rapports utilisés couramment dans l'environnement professionnel. Ces exemples mentionnent des noms fictifs de personnes, de sociétés, de marques ou de produits à des fins illustratives ou explicatives uniquement. Toute ressemblance avec des noms de personnes, de sociétés ou des données réelles serait purement fortuite.

LICENCE DE COPYRIGHT :

Le présent logiciel contient des exemples de programmes d'application en langage source destinés à illustrer les techniques de programmation sur différentes plateformes d'exploitation. Vous avez le droit de copier, de modifier et de distribuer ces exemples de programmes sous quelque forme que ce soit et sans paiement d'aucune redevance à IBM, à des fins de développement, d'utilisation, de vente ou de distribution de programmes d'application conformes aux interfaces de programmation des plateformes pour lesquels ils ont été écrits ou aux interfaces de programmation IBM. Ces exemples de programmes n'ont pas été rigoureusement testés dans toutes les conditions. Par conséquent, IBM ne peut garantir expressément ou implicitement la fiabilité, la maintenabilité ou le fonctionnement de ces programmes. Les programmes exemples sont fournis "en l'état", sans garantie d'aucune sorte. IBM ne sera en aucun cas responsable des dommages liés à l'utilisation de ces programmes exemples.

Toute copie totale ou partielle de ces programmes exemples et des oeuvres qui en sont dérivées doit comprendre une notice de copyright, libellée comme suit :

© IBM 2011. Des segments de code sont dérivés des Programmes exemples d'IBM Corp. © Copyright IBM Corp. 2011.

Si vous visualisez ces informations en ligne, il se peut que les photographies et illustrations en couleur n'apparaissent pas à l'écran.

Marques

IBM, le logo IBM et ibm.com sont des marques d'International Business Machines Corp. dans de nombreux pays. Les autres noms de produits et de services peuvent appartenir à IBM ou à des tiers. La liste actualisée de toutes les marques d'IBM est disponible sur la page Web “Copyright and trademark information” à l'adresse http://www.ibm.com/legal/copytrade.shtml.

Adobe, le logo Adobe, PostScript et le logo PostScript sont des marques d'Adobe Systems Incorporated aux États-Unis et/ou dans certains autres pays.

IT Infrastructure Library est une marque de The Central Computer and Telecommunications Agency qui fait désormais partie de The Office of Government Commerce.

Intel, le logo Intel, Intel Inside, le logo Intel Inside, Intel Centrino, le logo Intel Centrino, Celeron, Intel Xeon, Intel SpeedStep, Itanium, et Pentium sont des marques d'Intel Corporation ou de ses filiales aux Etats-Unis et dans certains autres pays.

Linux est une marque de Linus Torvalds aux Etats-Unis et/ou dans certains autres pays.

Microsoft, Windows, Windows NT et le logo Windows sont des marques de Microsoft Corporation aux Etats-Unis et/ou dans certains autres pays.

ITIL est une marque de The Office of Government Commerce et est enregistrée au bureau américain Patent and Trademark Office.

UNIX est une marque enregistrée de The Open Group aux Etats-Unis et/ou dans certains autres pays.

Java ainsi que tous les logos et toutes les marques incluant Java sont des marques d'Oracle et/ou de ses filiales.

Cell Broadband Engine est une marque de Sony Computer Entertainment, Inc. aux États-Unis et/ou dans certains autres pays, et est utilisée sous license.

Linear Tape-Open, LTO, le logo LTO, Ultrium et le logo Ultrium Logo sont des marques de HP, IBM Corp. et Quantum aux États-Unis et/ou dans certains autres pays.

Connect Control Center, Connect:Direct, Connect:Enterprise, Gentran, Gentran:Basic, Gentran:Control, Gentran:Director, Gentran:Plus, Gentran:Realtime, Gentran:Server, Gentran:Viewpoint, Sterling Commerce, Sterling Information Broker et Sterling Integrator sont des marques de Sterling Commerce, Inc., une filiale d'IBM Company.

Les autres noms de sociétés, de produits et de services peuvent appartenir à des tiers.

Imprimé en France