Visual Modeler

実装ガイド

リリース 9.1

Visual Modeler

実装ガイド

リリース 9.1

お願い

本書および本書で紹介する製品をご使用になる前に、 133 ページの『特記事項』に記載されている情報をお読みください。

目次

第 1 章 実装方法

Visual Modeler の実装方法は、完了まで実装を計画および追跡できるようなフェーズで構成されています。

Visual Modeler の実装方法の表には、フェーズおよび各フェーズで完了するアクティビティーのサマリーが記載されています。各フェーズの追跡には、標準的なドキュメント・セットを使用できます。

実装フェーズ
: **説明**

計画
: 実装の計画立案: タイムライン、マイルストーンの設定、およびリスクと依存関係の特定

分析
: 編成と管理、ビジネス・ルール、ユーザー・インターフェース、メッセージング・プロトコル、データ・ソース、e-commerce フロー計画立案、トレーニング要件、ロールアウト戦略、環境準備、運用計画立案の定義

設計と構成
: インストール、構成、統合、単体テスト、およびトレーニング開発

テストとデプロイ
: サーバー構成、エンタープライズからパートナーへの通信、パートナーからエンタープライズへの通信のテスト、実動システムへの切り替え、ディストリビューター・トレーニング、文書の送信、サポート

向上
: 進行中の機能拡張アクティビティー、パートナー・トレーニング、およびサポート

第 2 章 Visual Modeler 統合の実装

Visual Modeler は、e-commerce ネットワークにチャネル・パートナーを統合するように設計されています。ネットワーク内の組織は、エンタープライズおよびパートナーの役割を果たします。エンタープライズの役割を果たす組織はそれぞれ、自らのエンタープライズ・サーバーをインストールして、そのチャネル・パートナーに情報を伝達します。

販売店または流通業者はそれぞれ、複数のエンタープライズと取引を行う場合があり、インストールしたエンタープライズ・サーバーは、さまざまなエンタープライズ・サーバーから送信されたメッセージを受信して応答できる必要があります。

次の表は、Visual Modeler を実装するときの主なアクティビティーをまとめたものです。

実装フェーズ
タスク

計画 プロジェクト分析

分析

- 構成分析
- 統合分析
- 要件分析

設計と構成

- サーブレット・コンテナー環境の準備
- 知識ベースのインストール
- 知識ベースのセットアップ
- Visual Modeler の構成
- 役割とセキュリティーの定義
- システム管理者の認証
- XML スキーマの作成
- BizAPI、BLC、およびコントローラーのカスタマイズ
- JSP ページのカスタマイズ

テストとデプロイ

- 製品の統合
- サーバー構成のテスト
- エンタープライズからパートナーへの通信のテスト
- パートナーからエンタープライズへの通信のテスト
- 実動システムへのリリース

向上 評価と機能拡張

第 3 章 実装手順

Visual Modeler を実装するときに実行する主なタスクは、以下のとおりです。

- *プロジェクト分析*：実装プロジェクトに対して、タイムラインを設定したスケジュールに同意します。実装の進行状況を測定するためのマイルストーンを指定し、実装が予定どおりに完了しなくなる原因となる依存関係およびリスクを洗い出します。
- *構成分析*：Visual Modeler の適切な構成を決定します (使用されるマシンの数、およびファイアウォールとプロキシー・サーバーを基準とした、内部ネットワークでの各マシンの場所)。クラスター化された実装環境について詳しくは、「*VM インストール・ガイド*」の『*High Availability and Load Balancing*』を参照してください。
- *統合分析*：既存の e-commerce システムとの統合ポイントを特定します。
- *要件分析*：ハードウェアとソフトウェアの要件を調べて、予想されるトラフィックおよび要求される応答時間をサポートするうえで、マシンの処理能力が十分であることを確認します。
- *Visual Modeler のインストール*：指定したマシンに Visual Modeler をインストールします。詳しくは、「*VM インストール・ガイド*」の『*Installation Overview*』を参照してください。
- *知識*ベースのセットアップ：
  1. 知識ベースのインストール: 指定したデータベース・サーバーに知識ベース・スキーマをインストールします。
  2. 知識ベースのセットアップ: 知識ベースのデータベース・サーバーへの接続を確認して、 e-commerce に関連する情報をすべて入力します。これには、パートナーのパートナー・プロファイル、製品カタログ、および価格リスト情報を含める必要があります。

     詳しくは、「*VM インストール・ガイド*」を参照してください。
- *Visual Modeler の構成*：構成ファイルを変更して、実稼働環境でのシステム構成を定義します。
- *役割とセキュリティーの定義*：グループと役割を定義し、それに従って構成ファイルと ACL スクリプトを変更します。これによって、エンタープライズ・サーバーのユーザーのセキュリティー特権が決定します。
- *スキーマの作成*：ビジネス・オブジェクトのスキーマを作成して、データ・ソース情報を提供します。データ層は、エンタープライズ・サーバーと外部システムとの間のアクセスを管理します。
- *BLC およびコントローラーのカスタマイズ*：ビジネス・ロジックとコントローラー・クラスを変更して、ご使用のビジネス・ロジックをサポートします。場合によっては、組織に固有のビジネス・プロセスを実装するために、 Java クラスを変更することが必要になります。
- *JSP ページのカスタマイズ*：テンプレートを変更して、「ルック・アンド・フィール (look-and-feel)」、検索、静的ページの各要件を満たします。Visual Modeler

に用意されている JSP ページは、ブラウザーのページ表示に使用されます。また、組織のニーズに応じてカスタマイズすることもできます。

- *製品の統合* : 知識ベースに製品情報をインポートするか、またはパンチアウト統合 (業界標準のプロトコルを使用してシステムと統合すること) を実現します。実装環境で IBM 以外の製品からの製品順序付けをサポートする場合は、製品データを Visual Modeler に統合する手段を提供する必要があります。
- *サーバー構成のテスト* : Visual Modeler をデプロイする前に、システムを徹底的にテストします。主要な機能コンポーネントをテストするスクリプトが数多く用意されています。
- *エンタープライズからパートナーへの通信のテスト* : 使用しているエンタープライズ・サーバーから他のエンタープライズ・サーバーにテスト・メッセージを送信します。
- *パートナーからエンタープライズへの通信のテスト* : 他のエンタープライズ・サーバーから使用しているエンタープライズ・サーバーにテスト・メッセージを送信します。
- *評価と機能拡張* : Visual Modeler をデプロイしたら、その使用とパフォーマンスを分析する進行中のプロセスに向けて計画を立案する必要があります。

第 4 章 Visual Modeler と Sterling Selling and Fulfillment Foundation の統合

Visual Modeler と Sterling Selling and Fulfillment Foundation の統合

場合によっては、複雑な製品をお客様の購入前に構成しておくことが必要になります。それ以外の場合では、そのような製品には、お客様が要件に基づいて構成できるオプション製品が用意されていることもあります。Visual Modeler では、製品の構成可能なオプションを定義するモデルを作成したり、それらのモデルに製品を関連付けたりすることができます。IBM Sterling Configurator は、構成可能な製品を、エンド・ユーザーが使用できるオプションとともに表示するために使用されるツールです。

Visual Modeler と IBM Sterling Selling and Fulfillment Foundation で情報を交換できるようにするには、この 2 つを統合する必要があります。この統合は、Sterling Selling and Fulfillment Foundation に保持されている正しい製品情報が、Visual Modeler でのモデル定義に使用されるようにするために必要です。製品に適用される価格は、価格リスト、およびゲスト・ユーザーに関連付けられる通貨に基づいています。価格リストの関連付けについて詳しくは、「*Business Center 価格設定管理ガイド*」を参照してください。

Visual Modeler を Sterling Selling and Fulfillment Foundation に統合するには、Visual Modeler アプリケーションおよび Applications Manager で一部の構成を実行する必要があります。

Visual Modeler のプロパティーの構成

このタスクについて

Visual Modeler で Sterling Selling and Fulfillment Foundation から正しい製品情報を取得できるようにするには、Visual Modeler の一部のプロパティーの値を構成する必要があります。

Visual Modeler のプロパティーを構成するには:

手順

1. ブラウザーで次の URL を指します。

   ```
   http://<hostname>:<port>/<context_root>/en/US/enterpriseMgr/admin
   ```

   ここで、hostname は IP アドレス、port は Visual Modeler がインストールされているマシンのリスニング・ポート、および context_root はホストされている Visual Modeler アプリケーションのコンテキスト・ルートです。

   「ログイン (Login)」ページが表示されます。
2. ログイン ID とパスワードを入力し、「**ログイン (Log In)**」をクリックして、管理者としてログインします。

3. 「**システム・サービス (System Service)** 」ハイパーリンクをクリックします。「システム・プロパティー (System properties)」ページが表示されます。
4. 「**配送 (Fulfillment)**」ハイパーリンクをクリックします。「配送のプロパティー (Properties for Fulfillment)」ページが表示されます。
5. Sterling Order Fulfillment System の URL プロパティーを `http://<hostname>:<port>/smcfs/interop/InteropHttpServlet` に設定します。この URL は、Sterling Selling and Fulfillment Foundation の相互運用性サーブレットに関係しています。
6. Sterling Configurator の URL プロパティーを次のように設定します。

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

   ここで、hostname は Sterling Selling and Fulfillment Foundation がインストールされているマシンの IP アドレス、port は Sterling Selling and Fulfillment Foundation がインストールされているマシンのリスニング・ポートです。
7. 以下のプロパティーを適宜設定します。
   - Sterling Fulfillment システムのユーザー名
   - Sterling Fulfillment システムのパスワード

   これらのプロパティーの値によって、 Sterling Selling and Fulfillment Foundation サーバーとの通信に使用されるユーザー名およびパスワードが決定します。

Sterling Configurator のルールの構成

このタスクについて

Sterling Configurator で Visual Modeler から製品のモデル情報を入手できるようにするには、モデルの場所、プロパティー、およびモデルに関係するルールを Applications Manager で指定する必要があります。

Sterling Configurator のルールを構成するには:

手順

1. 「サインイン (Sign In)」ページで、ログイン ID とパスワードを入力し、「**サインイン (Sign In)** 」をクリックして、管理者としてログインします。アプリケーション・コンソールのホーム・ページが表示されます。
2. メニュー・バーから、「**構成**」 > 「**Applications Manager の起動**」にナビゲートします。ブラウザーの新しいウィンドウで Applications Manager が起動します。
3. Applications Manager のメニュー・バーから、「**アプリケーション**」 > 「**アプリケーション・プラットフォーム**」にナビゲートします。「アプリケーション・ルール (Application Rules)」サイド・パネルが表示されます。
4. 「アプリケーション・ルール (Application Rules)」サイド・パネルで、「**システム管理**」 > 「**製品コンフィギュレーター**」を選択します。
5. モデル、プロパティー・ファイル、およびルールが格納されている場所へのパスを指定します。

タスクの結果

注:

- モデル・リポジトリーに対して Applications Manager で指定したパスはすべて、Sterling Selling and Fulfillment Foundation と Visual Modeler で共有されます。Sterling Selling and Fulfillment Foundation と Visual Modeler が別々のマシンにある場合は、パスは両方からアクセス可能なドライブにマウントする必要があります。モデル・リポジトリーについて詳しくは、「*Selling and Fulfillment Foundation アプリケーション・プラットフォーム構成ガイド*」を参照してください。
- IBM Sterling Business Center では、モデルはバンドル製品のアイテム定義に割り当てることができます。モデル名はアイテム定義に保存されています。モデル名がアイテム定義に保存された後でそのモデル名を変更した場合は、変更後のモデル名を指すようにアイテム定義を変更する必要があります。この状況は、ユーザーが in Visual Modeler でモデル定義を編集するときに発生する場合があります。

第 5 章 J2EE Web アプリケーションの概要

このトピックでは、Java 2 Platform, Enterprise Edition (J2EE) の概要、および Web アプリケーションをデプロイするために J2EE がどのように使用されるかについて示します。このアーキテクチャーに既に習熟している場合には、このトピックはスキップしてかまいません。

アーキテクチャー

Visual Modeler は、Sun Microsystems, Inc. 発行の「*Java 2 Platform Enterprise Edition Specification, v 1.2*」で定義されている Java 2 Platform, Enterprise Edition (J2EE) アーキテクチャーに準拠するように設計されています。

Visual Modeler は、一連の Java クラスに、構成ファイル、HTML テンプレート、および JSP (JavaServer Pages) ページが付随した Web アプリケーションとしてデプロイされます。これは、J2EE 規格に適合したサーブレット・コンテナーにインストールする必要があります。

Web アプリケーション

J2EE Web アプリケーションは、J2EE 規格に適合するように構築されています。Web アプリケーション・アーカイブ (WAR) ファイルというパッケージに格納されている J2EE サーブレット・コンテナーに Web コンポーネントを追加します。WAR ファイルとは、JAR (Java アーカイブ) ファイル形式で圧縮したファイルです。

WAR ファイルには通常、Web コンポーネント以外にも、以下のような他のリソースが含まれます。

- サーバー・サイドのユーティリティー・クラス
- 静的な Web リソース (構成ファイル、HTML ページ、画像ファイル、音声ファイルなど)
- クライアント・サイドのクラス (アプレットとユーティリティー・クラス)

WAR ファイルとしてデプロイされている Web アプリケーションのディレクトリーおよびファイル構造は、正確な構成に適合しています。WAR ファイルには、特定の階層ディレクトリー構造があります。WAR ファイルの最上位のディレクトリーは、アプリケーションの*ドキュメント・ルート* です。ドキュメント・ルートとは、JSP ページ、クライアント・サイドのクラスとアーカイブ、および静的な Web リソースが格納されているディレクトリーです。ドキュメント・ルートには **WEB-INF/** というサブディレクトリーがあり、ここには以下のファイルとディレクトリーが格納されています。

- **web.xml**: Web アプリケーションのデプロイメント記述子。これは、Web アプリケーションの構造を記述したものです。
- タグ・ライブラリー記述子ファイル。
- **classes/**: サーブレット、ユーティリティー・クラス、Java Bean コンポーネントという、サーバー・サイドのクラスを格納したディレクトリー。

- **lib/**: ライブラリー (タグ・ライブラリー、およびサーバー・サイドのクラスで呼び出される任意のユーティリティー・ライブラリー) の JAR アーカイブを格納したディレクトリー。

web.xml ファイル

サーブレット・コンテナーにデプロイされたあらゆる Web アプリケーションでは、その **WEB-INF/** ディレクトリーに **web.xml** ファイルが存在している必要があります。あらゆる **web.xml** の構造は、J2EE 規格の一部として公開されている DTD に適合しています。

web.xml の目的は、J2EE 規格で必要とされる Web アプリケーションの全般的な構成を指定することです。具体的な内容は、以下のとおりです。

- Web アプリケーションの初期化パラメーター値が提供されます
- Web アプリケーションで使用されるサーブレット・クラスは、宣言して名前を指定することができます
- それぞれのサーブレット・クラスは 1 つ以上の URL パターンにマップされ、**web.xml** ファイルで定義されたパターンと URL が一致する要求をサーブレット・コンテナーが受信すると、対応するサーブレットがその要求の処理に使用されます
- 必要に応じて、各サーブレットに初期化パラメーター値が提供されます
- セッション情報 (タイムアウトなど)
- JSP ページで使用されるカスタム・タグ・ライブラリーの場所

JSP ページ

初期の Java ベースの Web アプリケーションでは、ユーザーの Web ブラウザーに返送された HTML の生成にはサーブレットのみを使用していました。時間の経過とともにテンプレート・メカニズムが導入され、Web 開発者が、テンプレートを使用して動的なコンテンツを生成し、HTML を生成できるようになりました。そのようなテンプレート・システムはいくつも使用可能ですが、J2EE アーキテクチャーは、コンテンツの表示に JSP (JavaServer Pages) ページを使用することに落ち着きました。

J2EE アプリケーションは、ユーザーのブラウザーから要求を受信すると、最初にその要求を処理して、要求からパラメーターを抽出して、要求によって開始されるビジネス・ロジックを実行します。この処理が完了したら、Web アプリケーションは要求を JSP ページにディスパッチする必要があります。この処理に使用するのが要求ディスパッチャー です。通常の場合、サーブレット・コンテキストがディスパッチャーにターゲット JSP ページを渡すことによって要求ディスパッチャーを呼び出し、要求ディスパッチャーによって要求オブジェクトと応答オブジェクトが転送 されます。

- JSP ページは、HTML、JSP タグ、スクリプトレット などのスクリプト要素の組み合わせで構成されています。
- HTML: JSP ページには、通常の HTML をいくらでも含めることができます。このコンテンツは、変更なしでブラウザー・ページにそのまま渡されます。
- JSP タグ: タグは、動的に生成される HTML に、ページの生成時に計算される値を移入します。<jsp:getProperty>、<jsp:include>、 <jsp:forward> などの標準的な

JSP タグがあります。これらは、JSP ページを作成する人間であれば誰でも使用できます。また、Web アプリケーションでカスタム・タグ・ライブラリーを 1 つ以上使用するように指定することもできます。それぞれのカスタム・タグ・ライブラリーは、Web アプリケーションの **web.xml** ファイルで宣言する必要があります。また、その宣言では、タグ・ライブラリーの URI とタグ・ライブラリー記述子 (TLD) ファイルの場所の両方を指定する必要があります。

注: Visual Modeler では、タグ・ライブラリーの使用は非推奨になりました。パフォーマンス上の理由により、スクリプトレットを使用することをお勧めします。一部の既存のアプリケーションまたは専門化された統合タスクでは、JSP タグは引き続き使用できます。

- スクリプト要素: JSP ページには HTML タグと JSP タグを随所に挿入することができます。挿入するときには、スクリプトレットの開始タグ <% (または <jsp:scriptlet>) と終了タグ %> (または </jsp:scriptlet>) の間に Java コードを含めます。JSP ページで複雑なフロー制御を管理する場合には、スクリプトレットが最も広く使用されています。ほとんどの JSP スクリプト要素は、次の表に示す簡易書式を使用して呼び出すことができます。

  簡易書式

	XML 形式
<%	<jsp:scriptlet>
<%=	<jsp:expression>
<%!	<jsp:declaration>
<%@	<jsp:directive>

  .

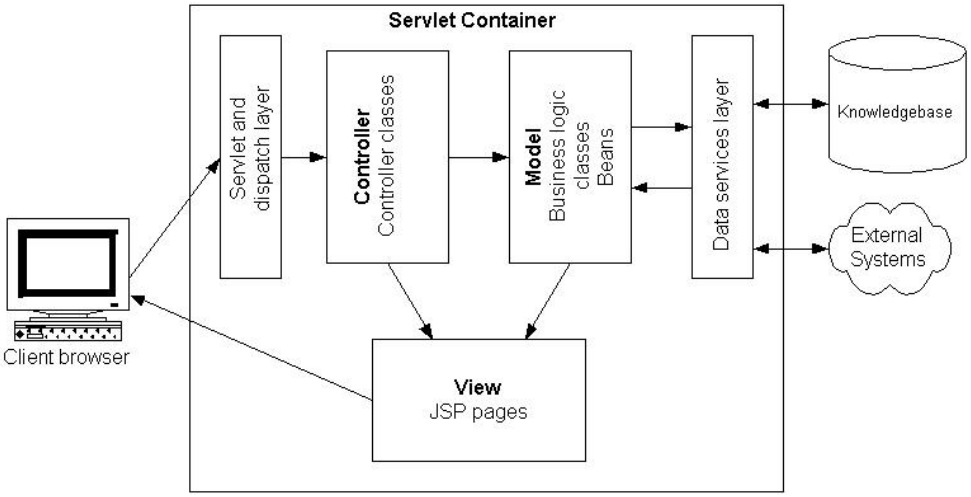
データは各種のメカニズムを使用して JSP ページに渡されますが、その最も重要なメカニズムが暗黙的なオブジェクトと Bean です。

- 暗黙的なオブジェクト: どの JSP ページでも、Web 開発者に対して、生成された HTML ページでのデータ表示に使用できるオブジェクトが提供されます。これらの中で最も重要なのが、ページ、要求、セッション、構成、アプリケーションの各オブジェクトです。
- Bean: アプリケーションのビジネス・ロジックによって生成されたデータのほとんどは、上に列挙した暗黙的なオブジェクトのいずれかに Java Bean を追加することによって、JSP ページに渡されます。

モデル 2 アーキテクチャー

Visual Modeler は、Sun の「モデル 2 (Model 2)」アーキテクチャーに適合するように設計されています。このアーキテクチャーでは、モデル、ビュー、およびコントローラー (MVC) という 3 つの機能コンポーネントによって、Web アプリケーションの機能が論理的に別々のコンポーネントに分割されます。

次の図は、このモデルのアーキテクチャーを示しています。



- モデル : このコンポーネントは、システムで使用されるデータ・オブジェクトおよびビジネス・オブジェクトを管理します。
- ビュー : このコンポーネントは、ユーザーに表示されるコンテンツの生成を担当します。
- コントローラー : このコンポーネントは、アプリケーションの論理フローを決定します。これは、モデルに対してどのようなアクションが実行されるかを決定し、モデル・コンポーネントとビュー・コンポーネントの間の通信を管理します。

コントローラー

モデル 2 アーキテクチャーでは、コントローラーは、インバウンド要求の処理を管理し、その要求を適切な JSP ページに転送するように意図された Java クラスです。Visual Modeler のコントローラーの基本構造は、次の形式に従っています。

```
public class GenericController extends Controller
{
   public void execute() throws Exception
   {
      //Dispatch some business logic
      BizObjs resultBizObjects = calculate();
      //Generate the beans
      Vector beans = generateBeans(resultBizObjs);
      //Attach the beans to the request
      attachBeans(beans);
      // Dispatch to JSP page
      String pageName = choosePageLogic();
      // Dispatch to JSP page
      Dispatcher rd = request.getDispatcher(pageName);
      rd.forward(request, response);
   }

   protected BizObjs calculate() throws Exception
   {
      //do some processing
      return resultBizObjs;
   }

   protected Vector generateBeans(BizObjs bizObjs)
   {
      //create beans from business objects
      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()
   {
      //logic to determine where to forward the request
      return pageString;
   }
}
```

モデル

モデル 2 アーキテクチャーでは、システム内のデータを表すオブジェクトは、モデル・コンポーネントによって保持されます。ビジネス・オブジェクトは、JSP ページで使用される Bean と区別することが一般的です。

ビジネス・ロジックがビジネス・オブジェクトの作成と変換を完了したら、コントローラー・クラスは、ビジネス・オブジェクトを対応する Bean に変換します。続いて Bean は、表示のために JSP ページに渡されます。

ビュー

Web アプリケーションのユーザー・インターフェースが、JSP ページを使用してブラウザーに提供されます。データが Bean の形式で各 JSP ページに渡されます。これらは、JSP ページ上のロジックによって、以下のような一般的な形式のタグを使用して値を取り出せるようにした accessor メソッドが定義されたクラスです。

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

ページ・レイアウトとデータ表示の管理には、スクリプトレット、簡易 JSP タグ、それより高度なカスタム・タグの組み合わせを使用することができます。

参考文献

Web アプリケーション、J2EE、サーブレット、および JSP ページに関しては、膨大な文献が発行されています。お勧めの参考文献を以下に示します。

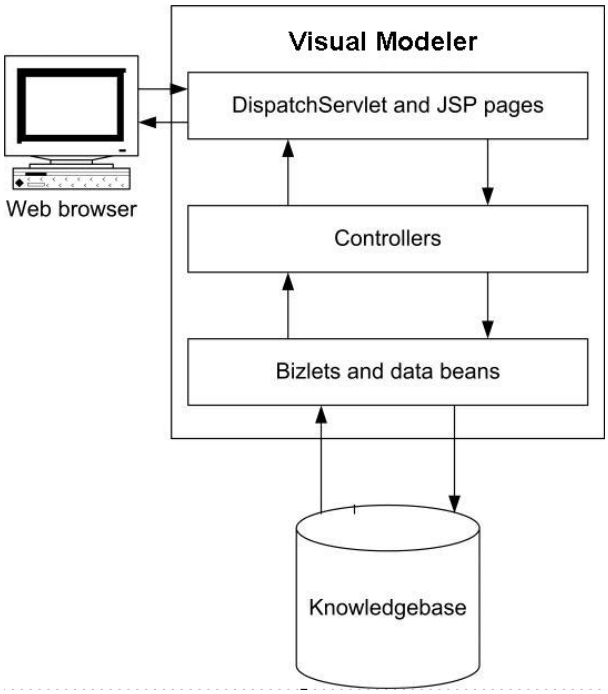
- Hall「*Core Servlets and JavaServer Pages*, Second Edition, Prentice Hall」 (2003)
- Hunter「*Java Servlet Programming*, Second Edition, O'Reilly」 (2001)
- Fields and Kolb「*Web Development with JavaServer Pages*, Second Edition, Manning」 (2001)

第 6 章 システム・アーキテクチャー

このトピックでは、Visual Modeler のアーキテクチャーについて説明し、Visual Modeler とそのアプリケーションで使用する重要な Java クラスをいくつか紹介します。ここでは、読者が J2EE アーキテクチャーを熟知していると想定しています。

このトピックの目的は、既存のアプリケーションの変更または拡張、あるいはアプリケーションの新規作成をやりやすくすることです。ただし、Visual Modeler の中には、アーキテクチャーに関するここでの記述に適合しない部分もあります。

次の図は、Visual Modeler のアーキテクチャーを示したものです。



Visual Modeler Web アプリケーション

Visual Modeler をサーブレット・コンテナーにインストールすると、**Sterling.war** という WAR ファイルとしてインストールされます。WAR ファイルがデプロイされたら、**Sterling/** というディレクトリーに解凍されます。**WEB-INF/** サブディレクトリーには、アプリケーションの **web.xml** ファイルが格納されています。

このファイルにおける最も重要な構成設定は、以下のとおりです。

- InitServlet および DispatchServlet の定義は、以下のとおりです。
  - InitServlet は、サーブレット・コンテナーが起動したときにロードします。InitServlet は、propertiesFile 要素の値 (デフォルトでは **Comergent.xml**) を使用して、Visual Modeler の構成情報をすべて読み込みます。

- DispatchServlet は、インバウンド要求の処理に使用される主なサーブレットです。サーブレット・マッピング・セクションで定義されている URL のほとんどは、解決されて DispatchServlet になります。

- サーブレット・マッピング・セクションによって、ほとんどの URL パターンが DispatchServlet にマップされます。要求を MessagingServlet にマップするときには「/msg/*」が使用されるため、インバウンド XML メッセージはこのサーブレット・クラスによって処理されるようになります。
- セッション構成要素によって、セッションのタイムアウト値が 30 (分) に設定されます。Visual Modeler を実装するたびに、このパラメーターの適切な値を慎重に検討する必要があります。以下の点を念頭においてください。
  - システムのエンド・ユーザーは、席を立つときにブラウザーを無人のまま放置することがあります。セッションがまだ有効なときに悪意のあるユーザーがブラウザーにアクセスできたら、システムにアクセスできてしまいます。
  - エンド・ユーザーは、Visual Modeler の使用中に他の外部システムに移動することがあります。セッションのタイムアウト値には、ユーザーが他のシステムに移動して戻ってくるのに十分な時間を指定する必要があります。
  - システム・リソースはセッションごとに使用されます。セッションのタイムアウト値が大きければ大きいほど、システムのメモリー使用量が増えます。
- Comergent タグ・ライブラリー記述子 (TLD) ファイルの場所が指定されます。

要求の処理

Visual Modeler は、ユーザーのブラウザーから要求を受信すると、その要求の処理方法、および処理結果をユーザーに表示する方法を決定する必要があります。この処理に使用するのが **MessageTypes.xml** 構成ファイルです。これらのファイルによって、要求と使用されているロジック処理クラスおよび JSP ページの間のマッピングが決定します。

1. 要求が受信されたら、そのメッセージ・タイプが識別され、適切なコントローラーが呼び出します。
2. ビジネス・ロジック・クラスまたは bizAPI クラスを使用して、追加のビジネス・ロジックを呼び出すことができます。
3. 次にコントローラーは、指定された JSP ページに要求を転送し、要求元のユーザーのブラウザーに出力をレンダリングします。

Comergent.xml ファイルの GeneralObjectFactory 要素の messageTypeFilename 要素で、メッセージ・タイプの指定に使用される **MessageTypes.xml** ファイルのコンマで区切られたリストを指定します。それぞれの **MessageTypes.xml** ファイルで、メッセージ・グループ順に編成されたメッセージ・タイプのリストを宣言します。

それぞれの要求によって、メッセージ・タイプが cmd パラメーターに指定されます。例えば、URL の形式が次のようになっているとします。

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

この場合、メッセージ・タイプの名前は「search」となります。

それぞれのメッセージ・タイプは、その MessageType 要素の Name 属性によって識別されます。Name 属性では、ユーザーが URL をクリックしたときにどのメッセージ・タイプが要求されるかが識別されます。

注: メッセージ・グループとメッセージ・タイプのそれぞれで、名前が固有であることを確認する必要があります。**MessageTypes.xml** ファイルの集合を調べ、名前の同じメッセージ・グループとメッセージ・タイプを定義していないことを確認する必要があります。このルールの例外については、『MessageType 定義のオーバーライド』 を参照してください。メッセージ・タイプの名前が重複していないかどうかを迅速に識別する手段として、メッセージ・グループ内の名前ごとに、メッセージ・タイプをアルファベット順で一覧表示することをお勧めします。

MessageType 要素には、以下の子要素が 1 つ以上あります。

- BizletMapping: メッセージ処理に使用され、Bizlet クラスとこのクラスのメソッドを関連付けてメッセージを処理します。
- ControllerMapping: 要求の処理に使用されるコントローラーを関連付けます。メッセージ処理を行う場合は、BizRouter クラスを指定して Bizlet クラスを呼び出し、メッセージを処理することができます。
- JSPMapping: 要求の処理結果を表示するために使用される JSP ページを関連付けます。

MessageType 要素では、これら 3 つの要素の任意の組み合わせを指定できます。

- ControllerMapping 要素が指定されていない場合は、デフォルトで ForwardController クラスが使用されます。このクラスは、JSPMapping 要素で指定されている JSP ページに要求を転送するのみです。 JSPMapping 要素が見つからないか、または指定された JSP ページが欠落している場合には、エラー・ページが表示されます。
- カスタム・コントローラーが指定されている場合は、そのコントローラーによって要求自体を処理することも (22 ページの『コントローラー・クラス』を参照)、AppExecutionEnv クラスの *runAppJob()* メソッドを使用してビジネス・ロジック・クラスを呼び出すこともできます (27 ページの『AppExecutionEnv クラス』を参照)。
- JSPMapping 要素が指定されていない場合は、どの JSP ページを使用するかを、ビジネス・ロジック・クラスまたはコントローラーで指定する必要があります。

当該メッセージ・タイプをユーザーが実行できることを検証するために、それぞれの要求またはメッセージが資格システムに対して妥当性検査されます。すべてのユーザーがすべてのメッセージ・タイプを実行できるわけではありません。

MessageType 定義のオーバーライド

MessageType 要素には、IsOverlay という任意指定の属性があります。この属性が「true」に設定されていると、messageTypeFilename 要素に既に一覧表示されている MessageTypes.xml ファイルでこのメッセージ・タイプの定義が前もって与えられている場合は、この MessageType 定義によってオーバーライドされます。

同じメッセージ・タイプに複数の定義が与えられていて、isOverlay 属性を指定する定義がない場合は、初期化エラーが表示され、メッセージ・タイプの最初の定義が使用されます。

IsOverlay 属性によって MessageType の場所が変更されるわけではないことに注意してください。この場所を決定するのが、最初の定義が属するメッセージ・グループ、またはメッセージ・タイプを参照する MessageTypeRef 要素であることに変わりはありません。

例えば、adirectLogin メッセージ・タイプの定義をオーバーライドするには、以下のようにして要素を定義できます。

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

IsOverlay 属性は、MessageGroup 宣言に使用してメッセージ・グループの定義を上書きできるようにすることも可能ですが、そのような使用はしないことをお勧めします。

デフォルトの要素

メッセージ・グループごとに、デフォルトの BizletMapping 要素、ControllerMapping 要素、および JSPMapping 要素を指定できます。これらは、そのメッセージ・グループに属するメッセージ・タイプにマッピングが指定されていない場合に使用されます。

一般的に、メッセージ・グループでデフォルトのマッピングが指定されていない場合は、現在のメッセージ・グループの親メッセージ・グループでデフォルトのマッピングが探されます。メッセージ・グループ・ツリーのどこにもマッピングが見つからない場合は、MessageGroupDefaults メッセージ・グループで指定されている値が使用されます。

主要な Java クラス

図式のレベルでは、Visual Modeler のアプリケーションはすべて構造が同じで、コントローラー、ビジネス・オブジェクトと bizlet、および JSP ページで構成されています。

ラッパー・クラス

J2EE Web アプリケーションで使用されている以下の標準的なクラスのいくつかは、ラッパー・クラスに包まれていて、サポートされているサーブレット・コンテナーの間に軽度の特異性があれば管理されます。

ComergentContext

このクラスは、サーブレット・コンテナー・コンテキストを包含するために使用されます。これは、環境情報の Env オブジェクトを取り出すために使用できます。設定されているコンテキスト属性はいずれも、シリアライズ可能である必要があります。シリアライズ可能でない属性を設定しようとすると、エラーがスローされます。

getResourceAsStream() メソッドが提供されます。このメソッドは、読み取り専用アクセスのために、ストリームとしてファイルにアクセスするときに使用できます。ファイルへの書き込みアクセスには、LegacyFileUtils クラスの *adjustFileName()* メソッドを使用する必要があります。

ComergentDispatcher

このクラスは標準的な RequestDispatcher クラスの軽量のラッパーであり、*forward()* メソッドと *include()* メソッドが用意されています。

ComergentRequest

このクラスは、標準的な HttpRequest クラスを包含し、ヘルパー・メソッドを提供して、インバウンド要求およびメッセージを解析します。

ComergentResponse

このクラスは、標準的な HttpResponse クラスを包含しています。新しいメッセージ・タイプを使用して要求を渡すために、*localRedirect()* メソッドが用意されています。例えば、あるコントローラーで要求を処理してから、その結果を別のコントローラーに渡すこともできます。その場合は、次を呼び出します。

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

これには、HTTP 要求として受信されたかのように、要求を DispatchServlet に送信するという効果があります。

ComergentSession

このクラスは、標準的な HttpSession クラスを包含しています。ユーザーが最初にログインすると、User データ Bean が作成され、ComergentSession オブジェクトに追加されます。ComergentSession *getUser()* メソッドを使用してユーザー情報にアクセスすることができます。

例:

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

これで、現在のユーザーのキーが返されます。

```
session.getUser().getPartnerKey()
```

ユーザーの属するパートナーのキーが返されます。

ComergentSession オブジェクトは、ユーザーのセッションの複数の要求にわたって持続する必要のある情報を格納するために使用します。セッションにオブジェクトを設定する場合は *setAttribute(String s, Object o)* メソッド、そのオブジェクトを取り出す場合は *getSession(String s)* を使用します。セッションに格納されたオブジェクトは、シリアライズ可能なインターフェースを実装する必要があります。また、生成されたデータ Bean がすべて、このインターフェースを実装するため、これらのオブジェクトがセッションに格納される場合があります。

ComergentSession クラスには *logout()* メソッドも用意されており、このメソッドを呼び出すと、サーブレット・コンテナー・セッションがすぐに無効になります。

サーブレット

使用される主なサーブレットは、以下のとおりです。

- InitServlet: このサーブレットは、サーブレット・コンテナーが起動したときにロードします。その *init(ServletConfig config)* メソッドによって、ComergentAppEnv クラスが初期化されます。
- DispatchServlet: このサーブレットは、Visual Modeler で処理される要求のほとんどすべてにサービスを提供するために使用します。その原則的なメソッド呼び出しは、以下のとおりです。

  ```
  void dispatch(HttpServletRequest request, HttpServletResponse response)
  ```

  このメソッドでは、要求の処理に使用するコントローラーが作成されます。

  ```
  Controller controller createController(ComergentRequest comergentRequest)
  ```

  さらに、以下を呼び出します。

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

  createController() メソッドで作成されるコントローラー・クラスのインスタンスは、要求のファンクションです。コントローラーは GeneralObjectFactory クラスによって作成されるため、要求メッセージ・タイプによってコントローラー・クラスが決定されます。GeneralObjectFactory では、要求メッセージ・タイプからコントローラー・クラスへのマップに **MessageTypes.xml** ファイルが使用されます。
- DebsDispatchServlet: このサーブレットは、別のシステムから Visual Modeler にポストされた XML メッセージを処理するために使用します。要求のコンテンツ・タイプの先頭が「application/x-icc-xml」または「text/xml」の場合は、要求の処理には MessagingController が呼び出されます。

コントローラー・クラス

Visual Modeler には、コントローラーを使用して要求を処理する方法として、以下の 2 つが用意されています。

カスタム・コントローラー

com.comergent.dcm.caf.controller.Controller クラスを拡張することによって、独自のコントローラー・クラスを記述することができます。その場合には、アプリケーション・ロジックを提供して、要求の転送先とする JSP ページを決定する必要があります。例:

```
boolean processingSuccess = false;
/*
*
* ビジネス・ロジックによって要求が処理され、処理に成功すると
processingSuccess が
* true に設定されます。
```

```
*/
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

アプリケーション・ロジックからの出口点が 1 つしかない場合には、SimpleController クラスを拡張して要求を処理できます。SimpleController では、要求のメッセージ・タイプを使用して、アプリケーション・ロジックが終了したら要求が転送される JSP ページを決定します。 SimpleController クラスを拡張するには、*calculate()* メソッドを上書きします。

MessagingController

このクラスは、(他のシステムからの、価格と可用性の転送要求やショッピング・カートの転送要求などの) XML 要求の処理に使用されます。

DataBean クラス

Visual Modeler でのデータへのアクセスは、データ・オブジェクトを使用して管理します。このデータ・オブジェクトは、パートナー、ユーザー、製品などのビジネス・エンティティーを記述した XML 文書です。これらには、データ・オブジェクトのフィールドとともに、知識ベース内のデータベース表へのマップに関する情報が記述されています。データ・オブジェクトごとに XML ファイルが使用され、対応する DataBean Java クラスが生成されます。

DataBean クラスは、Visual Modeler で各ビジネス・エンティティーを表すために使用される主なクラスです。ユーザー、パートナー、製品などの各ビジネス・エンティティーは、メモリー内では、適切な DataBean クラスのインスタンスで表されます。詳しくは、 47 ページの『Visual Modeler におけるデータ Bean』を参照してく

ださい。レガシー・アプリケーションの中には、BusinessObject クラスをまだ使用しているものがあるかもしれませんが、一般的には BusinessObject クラスの使用は非推奨です。

データを JSP ページに渡すには、 DataBean クラスも使用されます。Visual Modeler の XML スキーマにデータ・オブジェクト定義があれば、generateBean ターゲットを実行することによって DataBean クラスを生成する際に使用できます (詳しくは、 87 ページの『Visual Modeler の実装をカスタマイズするための Software Development Kit の使用』 を参照)。

DataBean クラスは汎用的な抽象クラスであり、生成されたデータ Bean クラスはすべて、このクラスを拡張したものです。各 DataBean クラスには、データベース内のデータを取り出す *restore()* メソッドと、データベースにデータを保存する *persist()* メソッドが用意されています。

一部のアプリケーションでは、アプリケーション Bean を利用しています。これらの Bean の使用法については、 54 ページの『アプリケーション Bean、エンティティー Bean、およびプレゼンテーション Bean』 を参照してください。

ObjectManager クラスと OMWrapper クラス

DataBean クラスは、そのコンストラクターを使用してインスタンス化しないでください。その代わり、アプリケーションの要件に従い、 ObjectManager クラスおよび OMWrapper クラスを使用して、オブジェクトの新規インスタンスを作成します。これらのクラスは、必要に応じてオブジェクト・インスタンスを生成するように設計されたクラスを提供するときの Factory パターンに従います。これによって、オブジェクトの作成と使用を行うアプリケーション・コードを変更しないで、あるオブジェクト・クラスから別のオブジェクト・クラスに切り替えることができます。

オブジェクトの作成

一般的には、ObjectManager クラスより OMWrapper クラスを使用するようにしてください。ただし、どちらも使用可能です。これらのクラスを使用してオブジェクトを作成するときには、以下のメソッドを使用します。

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

または

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

オブジェクト・クラスへのオブジェクト名のマッピング

ObjectManager クラスと OMWrapper クラスでは、 (***debs_home*/Sterling/WEB-INF/properties/** の中にある) **ObjectMap.xml** 構成ファイルを使用して、 *getObject()* メソッドで指定されたオブジェクト名から、どのタイプのオブジェクトが作成されるかを判別します。

注: ObjectMap.xml ファイルにはコメントを追加しないでください。追加すると初期化時にエラーが発生する可能性があります。

それぞれの要素の形式は、以下のようになっています。

```
<Object ID="ObjectName">
<ClassName>ObjectClass</ClassName>
</Object>
```

getObject("ObjectName") メソッドが呼び出されると、ObjectClass クラスのインスタンスが返されます。*ObjectName* は、Java クラスまたはインターフェースの名前である必要があり、*ObjectClass* は、 *ObjectName* クラスのサブクラス (クラス自体の場合もあり) または *ObjectName* インターフェースを実装するクラスである必要があります。

ID 属性が ObjectName パラメーターと一致しているオブジェクト要素が **ObjectMap.xml** ファイルにない場合は、ObjectManager または OMWrapper によって、 ObjectName クラスのインスタンスが作成されます。つまり、次の形式の要素があるかのように動作します。

```
<Object ID="ObjectName">
<ClassName>ObjectName</ClassName>
</Object>
```

例えば、ObjectMap.xml ファイルに次の要素があるとします。

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

次に、以下のメソッドを呼び出すと、MatrixProductBean クラスのインスタンスが作成されます。

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

MatrixProductBean で ProductBean クラスを拡張する必要があります。拡張しないと、実行時に ClassCastException がスローされます。ただし、ID 属性が com.comergent.bean.productMgr.ProductBean になっている要素がない場合は、同じメソッドを呼び出しても、返されるのは com.comergent.bean.productMgr.ProductBean クラスのインスタンスです。

制約事項

あるオブジェクト要素の ClassName 要素に指定されたクラスが別のオブジェクト要素の ID 属性となるようなオブジェクト定義は作成できません。このルールの例外は、このクラスが、1 つのオブジェクト要素の ID と ClassName 両方の値として使用されているときです。特に、データ・オブジェクトを拡張する場合は (60 ページの『データ・オブジェクトの拡張』を参照)、以下のようにします。

1. 以下のようにして、拡張後のクラスを拡張前のクラスにマップするオブジェクト要素を定義します。

```
<Object ID="<Extended class>">
<ClassName><Extending class></ClassName>
</Object>
```

2. 任意の ClassName 要素に拡張後のデータ・オブジェクトに対する参照があれば、拡張前のデータ・オブジェクトに置換してください。

パラメーターの引き渡し

オブジェクト・コンストラクターにパラメーターを渡す必要がある場合は、次の OMWrapper メソッドも使用できます。

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

この形式では、メソッドの呼び出しに最大で 10 個のパラメーターを渡すことができます。以下の OMWrapper メソッドおよび ObjectManager メソッドを呼び出すと、オブジェクトの配列として、パラメーターをいくつでも渡すことができます。

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

または

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

例えば、ObjectMap.xml ファイルに次の要素があるとします。

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

ここで、MatrixOrderBean クラスは OrderBean クラスのサブクラスです。MatrixOrderBean に、MatrixOrderBean(CartBean cb) という形式のコンストラクターがあるとします。

次に、以下のメソッドを呼び出すと、パラメーターとして CartBean クラスのインスタンスを使用して、 OrderBean クラスのインスタンスが作成されます。

```
Cart temp_CartBean = (CartBean)
OMWrapper.getObject("com.comergent.bean.partnerMkt.CartBean");
/*
Cart Bean を処理するコード
*/
OrderBean temp_OrderBean = (OrderBean)
OMWrapper.getObjectArg("com.comergent.bean.productMgr.OrderBean",
temp_CartBean);
```

オブジェクトのプール

あるオブジェクトのクラスが頻繁に作成および使用されると予想される場合は、ObjectManager クラスおよび OMWrapper クラスを使用して、オブジェクトのプールを作成することができます。(ID 属性で識別される) 親オブジェクトは、プール可能なインターフェースを実装する必要があります。このインターフェースは、com.comergent.dcm.objmgr パッケージの一部です。ここでは、実装が必要な 1 つのメソッド reset() を宣言します。

プール可能なオブジェクトの操作が終了したら、以下のような *return()* メソッドを使用して、オブジェクト・プールに戻すことができます。

1. プールされたクラスの **ObjectMap.xml** エントリーで、以下のようにして、MaxPoolSize 属性に、プールに作成するオブジェクトの数を設定します。

   ```
   <Object ID="ObjectName" MaxPoolSize="n">
   <ClassName>ObjectClass</ClassName>
   </Object>
   ```

2. 上述のように、OMWrapper と ObjectManager を使用して、オブジェクト・クラスのインスタンスを作成します。
3. オブジェクトの操作が終了したら、以下を使用して、インスタンスをプールに戻します。

   ```
   OMWrapper.return(temp_ObjectClass);
   ```

4. または

   ```
   temp_ObjectManager.return(temp_ObjectClass);
   ```

26 ページの『パラメーターの引き渡し』に記述したようにパラメーターを渡してオブジェクトを作成した場合は、プールにあるオブジェクトを再利用するのではなく、新規オブジェクトが作成されます。

AppExecutionEnv クラス

ビジネス・ロジック・クラスの実行には AppExecutionEnv クラスを使用できます。ただし、ビジネス・ロジック・クラスの使用は非推奨になっているので、このクラスを使用するのは、レガシー・アプリケーションをサポートする場合に限定してください。静的メソッド *runAppObj()* を使用して、ビジネス・ロジック・クラスの作成を呼び出し、その prolog メソッドと service メソッドを実行します。

最も一般的な形式としては、以下のように使用できます。

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

AppExecutionEnv クラスによって、messageType ストリングで判別されるビジネス・ロジック・クラスが呼び出されます。このとき、ビジネス・オブジェクトの BizObjTable ベクトルが、入力ビジネス・オブジェクトとして使用されます。

AppsLookupHelper クラス

Visual Modeler には、データ・オブジェクトのステータスが、ルックアップ・コードを使用して管理される状況が数多くあります。例えば、オーダーの発行全体を通じて、オーダーのオーダー・ステータスが何度も変わることがあります。また、詳細に定義された値をいくつも使用することがあったり、さまざまなロケールで管理する必要のある表示フィールドの例もいくつもあります (例: ユーザーの肩書など)。このデータは、知識ベースのデータベース・スキーマの CMGT_LOOKUPS テーブルに格納されています。

それぞれのルックアップ・タイプには、ルックアップ・コードが 1 つまたは複数あることもあり、それぞれのコードには、説明ストリングが関連付けられています。例:

ルックアップ・タイプ	ルックアップ・コード	説明
AddressType	10	請求
AddressType	20	配送

ルックアップ・コードを説明ストリングにマップするには、AppsLookupHelper クラスを使用できます。AppsLookupHelper クラスの適切なメソッドを呼び出すことにより、パラメーターとしてルックアップ・コードを渡すと、対応するストリングが返されます。目的のルックアップ・タイプに応じて、そのルックアップ・タイプに適したメソッドを選択します。CMGT_LOOKUPS テーブルからルックアップ・コードを取り出すときにどのルックアップ・タイプが使用されるかは、使用したメソッドによって決まります。例えば、オーダー・ステータス・コード・ストリングを取り出すには、以下のように記述できます。

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

逆に、以下のようにすると、ルックアップ・コードを取り出すことができます。

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

常にではありませんが、ほとんどの場合、ルックアップ・タイプには、ヘルパー・メソッドが定義されています。詳しくは、AppsLookupHelper クラスに関する Java 文書を参照してください。詳しくは、32 ページの『ルックアップ・コードのサポート』 を参照してください。

ComergentAppEnv クラス

ComergentAppEnv クラスは、アプリケーションに固有の環境情報をコードに提供する場合に使用します。これには、以下のような役に立つメソッドが用意されています。

- *adjustFileName()*: このメソッドは、LegacyFileUtils クラスに移動されています。30 ページの『LegacyFileUtils クラス』 を参照してください。

- *constructExternalURL()*: このメソッドは、クライアントを元のサーバーにリダイレクトすることができる URL を構成する場合に使用します。本来このメソッドは、サーバーがセッション情報を復元できるようなリダイレクト URL を生成する場合に使用します。
- *getEnv()*: このメソッドを呼び出すと、環境オブジェクトが返されます。
- *getContext()*: このメソッドを呼び出すと、アプリケーション・コンテキストが返されます。

グローバル・クラス

このクラスの使用は非推奨です。そのロギング機能は log4j API に置換されました。詳しくは、 71 ページの『Visual Modeler でのロギング: 概要』 を参照してください。これは、プロパティーの値を取り出す処理をサポートしていますが、このサポート機能は、プリファレンス・メカニズムに置換されました。グローバル・クラスを使用したコードを引き続き使用する必要がある場合は、それぞれの使用個所を LegacyPreferences クラスに置換します。

GlobalCache インターフェース

このインターフェースは、Visual Modeler のすべてのアプリケーションで使用される、キャッシュに入れられたオブジェクトへのアクセス権限を提供するキャッシュを定義する場合に使用します。これは、Visual Modeler が複数のマシンで稼働しているクラスター化された環境をサポートするために使用できます。

GlobalCache インターフェースを実装したキャッシュ・クラスを使用するには、そのインターフェースのメソッドを実装する必要があります。キャッシュ・クラスは、InitServlet *init()* メソッドが呼び出されたときにロードされます。このクラスの名前を、**Comergent.xml** ファイルの General.globalCacheImplClass 要素として提供する必要があります。デフォルトの実装環境には、Visual Modeler: com.comergent.dcm.cache.impl.AppContextCache が提供されます。

GlobalCache インターフェースの実装環境にアクセスするには、以下のようにします。

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

このインターフェースがサポートしているメソッドは、以下のとおりです。

- *public String store(Serializable entry)*: グローバル・キャッシュにオブジェクトを格納します。このオブジェクトは、アプリケーションによってクリーンアップされるまで残ります。
- *public boolean store(String id, Serializable entry)*: グローバル・キャッシュにオブジェクトを格納します。このオブジェクトは、アプリケーションによってクリーンアップされるまで残ります。
- *public String cache(Serializable entry)*: グローバル・キャッシュにオブジェクトを格納します。このオブジェクトは、アプリケーションで使用しているうちは利用可能ですが、キャッシュ・システムによって自動的にクリーンアップされます。
- 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)*: 固有のオブジェクトがキャッシュに含まれているかどうかを調べます。
- *public Object get(String id)*: キャッシュに入れることのできるオブジェクトを取り出します。
- *public Object remove(String id)*: キャッシュに入れることのできるオブジェクトを削除します。
- *public boolean gc()*: このメソッドは、クーロン・ジョブで呼び出して、未使用のエントリーをキャッシュがクリーンアップできるようにする必要があります。

LegacyFileUtils クラス

LegacyFileUtils クラスには、ファイル操作用のヘルパー・メソッドが用意されています。その使用は非推奨ですが、ComergentAppEnv クラスで以前に用意されていた以下のメソッドに対するサポートは提供されています。

- *adjustFileName()*: ファイルの実際のパス名を返します。このメソッドは、ファイルにアクセスして、読み取りと書き込みのいずれかを行う場合に使用します。*getRealPath()* メソッドは、NULL を返すので使用しないでください。クラスター化された環境では、*adjustFileName()* メソッドによって、クラスターのメンバーが全員、同じファイルにアクセスするようになります。このメソッドは、以下の 4 つのパラメーターとともに使用する必要があります。

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

  このメソッドを、パラメーター 1 つだけの形式で使用することは非推奨です。ブール・パラメーターを使用すると、**web.xml** ファイルの WritableDirectory 要素に指定された構成パラメーターを使用して、ファイルの場所が判別されます。

OutOfBandHelper クラス

OutOfBandHelper クラスは、テンプレートとして JSP ページを使用して出力ストリームを生成する手段となります。その使用例を以下に示します。

```
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);
/*
* SendSMTP を初期化し、ストリームを
* 使用してメッセージの本文を設定します
*/
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);
//Send the mail
smtp.send( from, to, cc, subject, message, mimeType);
```

この例では、既存の要求オブジェクトと応答オブジェクト、および出力ストリームを使用して、OutOfBandHelper クラスがどのように初期化されるかがわかります。その *callJSP()* メソッドは、メッセージ・タイプ・パラメーターで決定された JSP ページに要求オブジェクトおよび応答オブジェクトを渡すことによって、出力ストリームを生成します。この出力ストリームは、コンテンツを取り出すためにアプリケーションによって使用できます。

OutOfBandHelper クラスは、メッセージ・タイプを JSP ページにマップするときに、セッションとコンテキストの情報を利用します。したがって、ブラウザー要求の場合と同様に、異なるロケールには異なる JSP ページを使用できます。また、OutOfBandHelper クラスは、どのロケールの JSP ページを使用するかを解決し、同じフェイルオーバー・ロジックを適用します。

プリファレンス・クラス

プリファレンス・モジュールは、Visual Modeler のプロパティーにアクセスするためのメカニズムです。これは、プラットフォームのモジュールに用意されているモジュールの 1 つであり、詳しくは 43 ページの『設定サービス』を参照してください。プリファレンス API の基本的な使用法は、以下のとおりです。

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

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

プロパティーを取り出すためにサポートしている主なメソッドは、以下のとおりです。

- 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)

それぞれの *get*Type*()* メソッドには、例えば次のように、対応する *put*Type*()* メソッドがあります。

- public void putString(String key, String value)

パラメーターを指定しないで *getPreferences()* メソッドを呼び出した場合は、Visual Modeler でサポートされているシングルトン・プリファレンス・オブジェクトが取り出されます。クラスの名前を渡した場合 (例えば、getPreferences(MyClass.class)) は、取り出したオブジェクトがスコープ指定されます。つまり、プリファレンス・オブジェクトを使用して値を取り出したプロパティーの名前に、指定したプロパティー名が先頭に付加されたクラスのパッケージ・パスが含まれます。

例えば、com.comergent.myApplication パッケージに MyClass が入っているとします。その場合、以下に示すコードのフラグメントが等価となります。

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

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

および

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

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

トランザクション

Visual Modeler には、トランザクションのサポートが用意されています。ここでいうトランザクションとは、1 つ以上のアトミック・オペレーションにわたるデータベース・アクションです。一般的には、トランザクション・クラスを使用するのは、複数のデータ・オブジェクトがともに存在している必要があり、そのうちの 1 つに障害が発生したら全部に障害が発生するような状況です。

ルックアップ・コードのサポート

Visual Modeler は、ルックアップ・コードを使用して、ロケール固有のストリングを保持してユーザーに表示するメカニズムを備えています。それぞれのルックアップ・タイプには、ルックアップ・コードを 1 つ以上定義でき、それぞれのルックアップ・コードには、サポートされているロケールごとの文字列を定義できます。

Visual Modeler に用意されているルックアップ・サポートは何ですか

Visual Modeler には、コード値と対応するストリングの間のルックアップ、およびルックアップ・コード・ストリングからコード値へのルックアップを自動的に提供する機能が備えられています。

「コード (code)」 DsElement が設定されている場合は、ルックアップ・キャッシュから「ストリング (string)」が自動的に移入されます。「ストリング (string)」値が設定されている場合は、ストリング値を使用して「コード (code)」がルックアップされます。

ストリング値はローカライズされますか

はい。コードからストリングをルックアップする場合、このメカニズムでは、ユーザーのロケールを使用して、使用するストリング値を判別します。ストリングからコードをルックアップする場合、このメカニズムでは、ストリング値を検索するときにユーザーのロケールを使用して、対応するコードを見つけます。

コードとストリングのマッピングはどのようにして定義しますか

コードとストリングの関係は、**DsDataElement.xml** スキーマ・ファイルで定義します。そのとき、データ・オブジェクトで「コード (code)」と「ストリング (string)」両方の DsDataElement が使用されている場合は、コードとストリングのマッピングが自動的に処理されます。

以下に示すのは。DataElement のコードとストリングのペアの例です。

```
<DataElement Name="OrderStatus" Description="Order Status"
DataType="LONG" MaxLength="20" LookupType="OrderStatus"
LookupString="OrderStatusString"/>
<DataElement Name="OrderStatusString" Description="Order Status"
DataType="STRING" MaxLength="260" LookupType="OrderStatus"
LookupCode="OrderStatus"/>
```

XML メッセージにルックアップは実行されますか

はい。メッセージングに使用されるデータ・オブジェクトにコードとストリング値のペアが含まれている場合は、ストリング値がコードのルックアップに自動的に使用されます。

ルックアップ・キャッシュはどのようにロードされますか

ルックアップ・キャッシュは、システムの起動時にロードされます。

第 7 章 プラットフォームのモジュール性

Visual Modeler プラットフォームのモジュール性の概要

Visual Modeler モジュラー・アーキテクチャーは、実装のカスタマイズおよびアップグレードを容易にするために設計されています。Visual Modeler プラットフォーム・アーキテクチャーにより、よりモジュール式にプラットフォームをビルドできるため、プラットフォームに対して変更およびアップグレードをより素早く簡単に実行できるようになり、それらのモジュールを使用して作成された異なる製品をサポートするために、モジュールを再利用することが可能になります。

プラットフォームの機能をプラットフォーム・モジュールの形で提供する手段を供給し、それらのモジュールが外部インターフェース領域を通してのみ互いに呼び出しを行うようにすることの利点は、以下のとおりです。

- アプリケーションの機能の区分化を簡単にします。
- Visual Modeler 各部の依存関係の理解および管理を簡単にします。
- 単一モジュールへのカスタマイズを含めたり、モジュール内に加えられた変更がシステム全体にどのような影響をもたらすのか把握しやすくなります。
- アップグレードにより起こりうる影響を最小限に抑えながら、それぞれのモジュールを個別に、より簡単にアップグレードできます。
- カスタマイズされていないモジュールへのアップグレードが、その他のモジュールに加えられたカスタマイズに影響を及ぼしません。
- Visual Modeler の既存のデプロイメントになる可能性のあるモジュールの形で、新しい機能を伝達できます。

プラットフォーム・モジュール

Visual Modeler プラットフォームは、共通の編成上の構造に準拠する相互依存型のモジュールのセットとして開発されました。通常、各プラットフォーム・モジュールは、サービスのような Visual Modeler の機能コンポーネントや、 Visual Modeler プラットフォームのコンポーネントに対応します。プラットフォーム・モジュールは、他のモジュールに対して Java API を提供します。一部のモジュールは、その他多数のモジュールによって使用される一連の「ヘルパー (helper)」クラスを提供します。

通常、各プラットフォーム・モジュールには次の構造があります。

- Java クラス。次のツリーに編成されています。ビルド時には、モジュールのディレクトリーは単一の JAR ファイルにアセンブルされます。
  - com.comergent.api.*モジュール名*: 外部 API インターフェース: このモジュールによって提供される機能にアクセスするために他のモジュールによって使用されます。一般に、あるモジュールが別のモジュールのクラスを呼び出す場合、その他のモジュールの外部 API を使用して呼び出しを行う必要があります。これは、モジュールの com.comergent.api パッケージです。

– com.comergent.モジュール名: 実装クラス: 外部 API インターフェースの実装です。別のモジュールがこのモジュールの外部 API の呼び出しを行う場合、使用される実際のクラスはこのモジュールのインターフェースの実装クラスです。実装パッケージには、内部クラスが含まれます。内部クラスは、実装クラスによって使用されますが、外部には公開されず、サポートされる Javadoc の一部ではありません。

- プロパティー・ファイルなど、モジュールに固有の構成ファイル。これらのファイルは、*getResource()* 呼び出しを介して参照できるよう、クラス階層に置かれることになっています。

プラットフォームのモジュール性: モジュールのインターフェース

各プラットフォーム・モジュールは、モジュール内の Java クラスやインターフェースに対するすべての呼び出しがインターフェースを介して呼び出せるよう、外部インターフェースを備えている必要があります。この外部インターフェースは、他モジュールの書き込みプログラムが外部インターフェースを確実かつ簡単に使用できるよう、広範囲にわたる Javadoc ページを提供します。

外部インターフェースは、com.comergent.api パッケージ下で編成されています。このパッケージには、モジュールでサポートされるすべての外部 API が含まれています。これらは、com.comergent.api.converter、com.comergent.api.logging などのモジュールによって編成されます。

インターフェースの呼び出し

オブジェクトまたは子インターフェースをインターフェースへキャストすることで Java クラスからインターフェースを呼び出すことができ、その後インターフェースが宣言するメソッドを呼び出すことができます。各モジュールは、このどちらかの方法を使用します。両方は使用しません。既存のモジュールで作業をしたり、新規モジュールを作成する場合は、一貫した方法でインターフェースを呼び出してください。 これにより、別の作業者も同じモジュールで作業しやすくなります。

通常、常に com.comergent.api パッケージで提供されるインターフェースを使用して作業するようにしてください。これらのインターフェースは、インターフェースの基礎となる実装が変更された場合でも、あるリリースから次のリリースまでプラットフォーム・モジュールによってサポートされます。

プラットフォーム・モジュールの説明

このセクションでは、各プラットフォーム・モジュールの目的とその使用例について簡単に説明します。

アクセス・ポリシー

このモジュールは、アクセス・ポリシーのチェックに使用されるサービスを提供します。

認証

このモジュールは、資格情報およびユーザーを認証するために使用される API を提供します。

Base64

このモジュールは、データを Base 64 表記に変換、または Base 64 表記から変換するために使用されるクラスを提供します。

クラスパス・アペンダー

このモジュールは、クラスパスにパスを追加するために使用されるクラスを提供します。

暗号化サービス

このモジュールは、データを暗号化および暗号化解除するために使用されるサービスを提供します。

データ・サービス

このモジュールは、既存のデータ・サービスの機能の再パッケージ化とクリーンアップを提供します。このモジュールの API は、別の com.comergent.api.dataservices パッケージに移動しました。データ・サービスでは、 Visual Modeler の残りの部分がそのプロパティーを管理するのに使用するのと同じプリファレンス・メカニズムを使用するようになりました。接続プールは 1 つのプールに統一され、調整可能です。ページ付けは更新され、現在ではファイル・システムに書き込まれるページ付けファイルに依存することはありません。

ディスパッチ許可

このモジュールはアクセス・チェックを管理し、各ユーザーがアプリケーション内の、自分にアクセス権が付与された部分のみを表示するようにできます。

ディスパッチ・フレームワーク

このモジュールは、サーブレット要求、応答、コンテキスト、およびセッション・クラスをディスパッチ機構によって使用されるベース・コントローラー・クラスとともにラップする、 Visual Modeler クラスのディスパッチ・フレームワークを管理します。

E メール・サービス

このモジュールは、Visual Modeler からの E メールの送信を開始する基本の API を提供します。

イベント・サービス

このモジュールは、EventBus および Events によって使用されるクラスを提供します。

例外サービス

このモジュールは、Visual Modeler によって使用される基本の例外フレームワークおよびクラスを提供します。

グローバル・キャッシュ・サービス

このモジュールは、キャッシュにアクセスするのに使用される API を提供します。

ヘルプ

このモジュールは ComergentHelpBroker クラスを提供します。これは、JavaHelp 2.0 実装の ServletHelpBroker クラスに対する単純なラッパー・クラスです。

初期化サービス

初期化モジュールは初期化サービスを提供します。これは、クラスおよびメソッドの一貫したフレームワークを使用して、Visual Modeler を初期化するのに役立つパッケージです。

Initialization Manager は、以下の作業を実行できるフォーカル・ポイントとなります。

- 初期化タスクの定義
- 失敗した初期化のポリシーの強制
- 構成の断片の集約

Initialization Manager の主要な役割は、初期化タスクのリストを、明確で予測可能な方法で実行することです。これは、次のような番号付きリストが作成されることを意味します。

- プログラム的に定義可能、または
- XML フォーマット・ファイルとして指定可能

次のコード抽出は、InitManager クラスの使用法の典型的な例を示しています。

```
InitManager initManager = InitManager.getInitManager();
try
{
String resourceName = args[0];
initManager.init(resourceName);
// or programatically created
//List modules = initModules();
//ResourceLocator resourceLocator = createNewResourceLocator();
//initManager.init(modules, resourceLocator);
}
catch (InitManagerException ime)
{
log.error(ime, ime);
System.exit(1);
```

```
}
// Initialization completed. OK to go on //
...
```

初期化プロセスを、構成ファイルを使用して指定できます。以下にサンプル・ファイルを示します。

```
<?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>
<!-- it is allowable to have no initClass -->
<module name="custom1" >
<config name="ObjectManager">
/com/comergent/module1/overlay/objectMap.xml
```

```
</config>
</module>
</initializationManager>
```

この例では、次のメソッドが Initialization Manager によって呼び出される場合に、

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

次の情報が使用可能になります。

- initParams には、param0-param0Value のキーと値のペアのリストがあります。
- configFragments には以下のリストがあります。
  - `/com/comergent/module1/objectMap.xml`
  - `/com/comergent/module12/objectMap.xml`
- resourceLocator は、 /a/b/c、現行パス、および現行のクラスパスに従ってリソースを検索できます。

国際化対応

このモジュールは、Visual Modeler によって提供される国際化対応機能の基本のサポートを提供します。

ロギング

このモジュールは、Visual Modeler 内のアクティビティーを記録するために使用されるロギング・サービスへのアクセスを提供します。このモジュールのプロパティー・ファイル **log4j.properties** は、ロギング・サービスの振る舞いを構成するために使用されます。このモジュールは log4j オープン・ソース・プロジェクトに基づいており、以下に示すようにその構成に使用されるものと同じ構文を使用します。

Log4j には、*ロガー*、アペンダー、および *レイアウト*のメイン・コンポーネントがあります。これら 3 つのタイプのコンポーネントは協働して、開発者がメッセージのタイプおよびレベルに従ってメッセージをログ記録したり、実行時にこれらのメッセージがフォーマットされる方法や報告される場所を制御したりすることを可能にします。

ロギング・モジュールの構成

log4j.properties 構成ファイルを使用して、そのロガー、アペンダー、およびレイアウトのプロパティーを指定することにより、ロギング・プラットフォーム・モジュールを構成できます。例えば、次のスニペットはルート・ロガーと CMGT アペンダーを構成するために使用されます。

```
# Set root category priority
#log4j.rootCategory=info, CMGT
log4j.rootCategory=info, STDOUT
#log4j.rootCategory=info, CMGT, RTS
### START - CMGT
# CMGT appender
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

 # The log format defaults to the "classic" format. This format is
# recommended for actual deployment to allow a log analyzer to
# work correctly.
log4j.appender.CMGT.layout.ConversionPattern=%d{yyyy.MM.dd HH:mm:ss:SSS}
Env/%t:%p:%c{1} %m%n
```

ロガー

ロガーとは、名前付きエンティティーです。ロガー名は大/小文字が区別され、階層的命名規則に従います。ロガー名の後ろにドットを付けたものが、子孫のロガー名の接頭部である場合、あるロガーを別のロガーの祖先であると言います。あるロガーと子孫のロガーの間に別の祖先がいない場合は、そのロガーを子ロガーの親であると言います。

例えば、com.foo というロガーは、 com.foo.Bar というロガーの親です。同様に、java は java.util の親であり、java.util.Vector の祖先です。この命名体系は多くの開発者にとってなじみの深いもののはずです。

ルート・ロガーはロガー階層の最上位に位置し、次の 2 つの点で例外的な存在です。

- 常に存在する。
- 名前で取り出せない。

クラスで静的な *Logger.getRootLogger()* メソッドを呼び出すと、ルート・ロガーが取り出されます。その他のすべてのロガーは、クラスで静的な *Logger.getLogger(String name)* メソッドによりインスタンス化され、取り出されます。

このメソッドは、必要なロガーの名前をパラメーターとします。

ロガーにはレベルを割り当てることが可能です。割り当て可能なレベルは、DEBUG、 INFO、WARN、ERROR、および FATAL で、org.apache.log4j.Level クラスで定義されています。あるロガーにレベルが割り当てられない場合、それは割り当てられたレベルを持つ、最も近い祖先からレベルを継承します。より正式な定義は以下のようになります。

レベルの継承: 任意のロガーの継承されるレベルは、ロガー階層内において、そのロガーからルート・ロガーまでの最初のヌルでないレベルと等しい。

すべてのロガーが最終的にレベルを継承できるよう、ルート・ロガーには必ず割り当てられたレベルがあります。

アペンダー

ロガーに基づいて選択的にロギング要求を有効または無効にできることは、全体像のほんの一部に過ぎません。複数のアペンダーをロガーに追加できます。

addAppender メソッドは、任意のロガーにアペンダーを追加します。任意のロガーに対して有効にされたロギング要求はそれぞれ、そのロガー内のすべてのアペンダ

ーおよびより高い階層に属するアペンダーに転送されます。言い換えると、アペンダーはロガー階層から加算的に継承されていきます。例えば、コンソール・アペンダーをルート・ロガーに追加すると、有効なロギング要求はすべて、少なくともコンソールには出力されることになります。さらに、ファイル・アペンダーをあるロガーに追加すると、そのロガーとその子に対する有効なロギング要求は、ファイルとコンソールに追加されることになります。additivity フラグを false に設定することで、このデフォルトの動作を上書きして、アペンダーが加算的に累積されないようにすることができます。

アペンダーの加法性 (additivity) を管理する規則を以下にまとめます。

- ロガー C のログ・ステートメントは、C と C の祖先のすべてのアペンダーに出力されます。これが「アペンダーの加法性 (additivity)」という用語の意味です。
- ただし、ロガー C のある祖先で additivity フラグが false に設定されている場合、C の出力は C のすべてのアペンダーとその祖先までは送られますが、その祖先の上位の祖先のアペンダーには送られません。
- ロガーの additivity フラグはデフォルトで true に設定されています。

レイアウト

出力の宛先だけでなく、出力フォーマットもカスタマイズすることが必要になる場合があります。これは、レイアウトをアペンダーに関連付けることによって可能になります。レイアウトには、必要に応じてロギング要求をフォーマットする役割があるのに対し、アペンダーはフォーマットされた出力を宛先に送信する責任を担います。標準の log4j 配布の一部である PatternLayout を使用すると、C 言語の printf 関数と類似した変換パターンに従って出力フォーマットを指定できます。

例えば、PatternLayout に以下の変換パターンを指定すると、

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

次のような出力が得られます。

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

最初のフィールドは、プログラムの開始時より経過した時間 (ミリ秒単位) を示します。2 番目のフィールドは、ログ要求を実行するスレッドです。3 番目のフィールドは、ログ・ステートメントのレベルです。4 番目のフィールドは、ログ要求に関連付けられたロガーの名前です。「-」の後のテキストは、ステートメントのメッセージです。

メモリー・モニター

このモジュールは、メモリー使用量をモニターおよびログ記録するために使用するクラスを提供します。

メッセージ・タイプの資格

このモジュールは、ユーザーにメッセージ・タイプを呼び出す資格があるかどうかをチェックするサービスを提供します。

インターフェースは、com.comergent.api.dispatchAuthorization パッケージで定義されます。このパッケージには、サービスに必要なファクトリー・クラス、インターフェース、および例外が含まれます。実装クラスは、com.comergent.dispatchAuthorization パッケージにあります。

このモジュールのメインのエントリー・ポイントは、EntitlementRepository クラスにあります。このクラスのインスタンスは、EntitlementFactory クラスから取得されます。アプリケーションで、EntitlementRepository クラスの名前付きインスタンスを作成できます。名前付きインスタンスを使用すると単体テストが容易になるため、代替デプロイメント環境で役立つ場合があります。

ディスパッチ・ルールまたはその他のメッセージ・タイプの資格オブジェクトを指定する必要のあるアプリケーションは、以下のようなロジックを実行します。

```
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);
```

オブジェクト・マネージャー

このモジュールは、オブジェクトをインスタンス化するために使用されるクラスを提供します。詳細は、 24 ページの『ObjectManager クラスと OMWrapper クラス』を参照してください。

アウト・オブ・バンド応答

このモジュールは、標準の JSP ページではなく、出力ストリームに出力を送信するために使用されます。

設定サービス

設定モジュールは、Visual Modeler によって使用される構成プロパティーの取り出しおよび設定に使用されます。以下のようにプロパティーを取り出せます。

```
private static final Preferences prefs =
Preferences.getPreferences(MyClass.class);
// implict scope of "com.comergent.apps.module.MyClass"
int max = prefs.getInt("PromotionManager.maxValue", 100);
int min = prefs.getInt("PromotionManager.minValue", 1);
```

getInt() 呼び出し内の 2 番目のパラメーターは、該当する名前を持つプロパティーが見つからなかった場合に返される値を指定します。プロパティーが定義される構成ファイルは、クラスパス上に存在するものとされます。例えば、**com.comergent.apps.module.Preferences.xml** ファイルなどです。XML プロパティ

ー・ファイルが設定サービスを使用して読み取られる場合は、XML ファイルで Comergent ルート要素が使用されるようにしてください。以下に例を示します。

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

以下のような要素を追加して WEB-INF/properties/init.xml 構成ファイルをカスタマイズすることにより、設定サービスがプロパティーを初期化するために使用されるようにできます。

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

設定クラスは、プロパティーの値を取得および配置するメソッドを提供します。以下に例を示します。

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

putObject() メソッドを使用する場合、オブジェクトは XMLEncoder API の要件を満たしている必要があります。つまりオブジェクトのフィールドには必ず getter/setter メソッドが必要です。

タグ・ライブラリー

Visual Modeler によって提供されるタグ・ライブラリーは、プラットフォーム・モジュールとして生成されます。

スレッド管理

このモジュールは、スレッドの作成、スレッドの状況の取得、および再利用など、スレッドの処理のための一元化された機能を提供します。このモジュールは backport-util-concurrent.jar ライブラリーによって提供されます。通常、アプリケーション開発者は以下を呼び出す必要はなくなります。

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

代わりに、一元化された機能によって、以下のことを実行できるようになります。

- 必要に応じてスレッドをプールおよび再利用
- 実行中のすべてのスレッドをトラックすることにより CPU およびリソースの使用のアカウンティングを向上
- 単純な状況レポートを提供 (スコアボード戦略: 実行中のスレッドがその状況を報告できる、中央の共有される場所)

- *Thread.interrupt()* の呼び出しによる単純な中止および割り込みの信号を提供することにより、長時間実行中の (ループ状態にある) スレッドを早期に終了させることが可能

このモジュールは、以下の機能を提供します。

1. スレッドのプールおよび再利用を透過的に提供
2. 管理機能として、スレッド管理者によって追跡されるすべての実行中のスレッドを照会する手段を提供
3. スレッド・サービスのユーザーに対して、現在のスレッドの状況を共通のスコアボードに報告する手段を提供
4. 単純なループのフォローまたは割り込み状況のプロトコルのチェックを実行するためのガイダンスを提供することにより、長時間実行中/ループ状態のスレッドを早期に終了させることが可能
5. タイマー機能の提供により、タイマーの期限が切れたときに実行中のスレッドに通知を送信可能。これは、単純なタイムアウトまたはタイム・シェア・ポリシーを実装する際に使用できます。

API は引き続き Runnable() パターンをフォローします。アプリケーションはスレッドのようなオブジェクトを取得して、それを使用して実行します。

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

XML メッセージ・コンバーター

このモジュールは、XML 文書をあるメッセージ・カテゴリー (ファミリーまたはバージョン) から別のカテゴリーに変換する機能を提供します。API のパッケージ名は、実装クラスの com.comergent.api.converter および com.comergent.converter です。

API パッケージには以下が含まれます。

- ConverterFactory: これは、コンバーターを作成するための Factory クラスです。
- Converter: これは、文書をあるメッセージ・カテゴリーから別のカテゴリーに変換するクラスです。このクラスは、変換のソースおよびターゲットとして文書またはストリームの形を取ることができます。

XML メッセージ・サービス

このモジュールは、アウトバウンド・メッセージを XML 文書として作成およびポストするために使用されます。API には、com.comergent.api.msgService パッケージ内の MsgContext インターフェース、MsgService インターフェース、MsgServiceFactory クラス、および MsgServiceException クラスが含まれ、実装クラスは com.comergent.msgService パッケージ内に存在します。

MsgService インターフェースには、データ Bean および XML 文書をメッセージ・コンテキストで指定されたようにポストするための汎用の *service()* メソッドが含まれます。

一般的な使用パターンを以下に示します。

1. MsgContextFactory を使用して MsgContext インスタンスを作成します。
2. コンテキスト・オブジェクト上で適切な属性を設定します。
3. ターゲット・メッセージ・ファミリーに対して MsgService インスタンスを作成します。
4. データ Bean およびメッセージ・コンテキストで service メソッドを呼び出して、メッセージをポストします。

以下に例を示します。

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

XML サービス

このモジュールは、XML 構文解析、XSL 変換、DOM ラッパー、およびユーティリティー・クラスのための機能をカプセル化します。

第 8 章 Visual Modeler データ Bean およびビジネス・オブジェクトの導入

Visual Modeler におけるデータ Bean

データ Bean とは、実世界のエンティティーの Visual Modeler における表現であり、データ・ソースに依存しません。Visual Modeler では、外部スキーマ (XML ファイルのセットとして定義されたもの) に基づいて各データ Bean タイプの構造が定義されます。例えばデータ Bean は、ユーザー、製品問い合わせリスト、パートナー、製品、およびワークスペースのデータ構造体として使用されます。

- DataBean クラスのインスタンスの作成には、OMWrapper クラスと ObjectManager クラスを使用します。詳しくは、 24 ページの『ObjectManager クラスと OMWrapper クラス』を参照してください。
- DataBean は DataManager を使用して作成できます。DataManager のメソッド *getDataBean(String beanName)* を呼び出すことによって、指定したタイプの DataBean が作成されます。該当する DataBean クラスが存在しない場合、このメソッドは InvalidBizobjException をスローします。

  注: このメソッドは、データ・オブジェクトの拡張をサポートしていないため、非推奨になりました。

データ Bean のライフサイクル

通常、データ・オブジェクト処理の基本的なフローは次のとおりです。

1. OMWrapper クラスを使用してデータ Bean オブジェクトをインスタンス化します。
2. set メソッドを使用して各データ・フィールドに値を直接挿入することによって、Bean にデータを追加します。
3. データ Bean を永続化し、新しいデータ・オブジェクトをそのデータ・ソースに初めて保存します。
4. その後、同じデータ・オブジェクトを取り出すことができます。キー・フィールドに値を設定し、そのデータ Bean に対して *restore()* を実行することによって、そのデータ・ソースから現在のフィールド値を取得できます。
5. そのデータ Bean に対して必要な任意のビジネス・ロジックを実行します。それによってメモリー内フィールドの値が変わる場合がありますが、そのデータ Bean のデータ・ソースに格納されている値は変わりません。
6. データ Bean をそのデータ・ソースに対して永続化することによって、データ Bean の変更を保存します。
7. その後、そのデータ・オブジェクトが不要になり、削除する必要が生じることがあります。
8. 最終的に不要な場合は、そのデータ・オブジェクトを消去することによって、データ・ソースからデータを完全に削除します。

データ・オブジェクトの基礎となるデータ・ソースがデータベースの場合に使用する Java メソッド呼び出しと、対応して呼び出される SQL メソッドを次の表に示します。

ステップ	Java メソッド	SQL メソッド
データ・オブジェクトのインスタンス化	*OMWrapper.getObject()*	
データ・フィールドの設定	*setDataField()*	
データ・フィールドの設定	*setDataField()*	
最初の永続化	*persist()*	INSERT
データ・オブジェクトの取得	*restore()*	SELECT
ビジネス・ロジックによるフィールド値の更新	*getDataField()* *setDataField()*	
変更内容の保存	*persist()*	UPDATE
データ・オブジェクトの削除	*delete()*	UPDATE **注:** 削除操作を実行すると、基礎となるデータベース表の該当する行の ACTIVE_FLAG 列が更新されます。表からレコードが削除されるわけではありません。
データ・オブジェクトの消去	*erase()*	DELETE

データ Bean の定義

データ Bean は XML スキーマを使用して定義します。データ Bean は個々のデータ・フィールドの値の取得および設定を実行する accessor メソッドを備えています。通常、Visual Modeler アプリケーションをカスタマイズするときには、データ Bean を使用します。

データ・オブジェクトの構造の定義

Visual Modeler で個々のデータ・オブジェクトのインスタンスを作成できるようにするには、そのデータ・オブジェクトの構造が定義されていなければなりません。データ・オブジェクトの構造はそのスキーマ XML ファイルに定義します。そのデータ・オブジェクトに含まれる各フィールドと、そのデータ・オブジェクトが子オブジェクトを持っているかどうかをこのファイルで指定します。

各データ・オブジェクトは、DataBean クラスのいずれかの拡張 Java クラスに対応しています。これらをデータ Bean クラスと呼びます。各データ Bean クラスは SDK マージ・プロセスの一部として自動的に生成されます。対応するデータ Bean クラスを生成すると、データ・オブジェクトの XML ファイルに宣言されているフィールドと子データ Bean にアクセスするためのメソッドが提供されます。

XML スキーマ・ファイルを編集して、その XML スキーマの定義を変更できます。したがって、このファイルの編集により、データ・オブジェクトおよび対応するデータ Bean クラスの定義を変更できます。

データ・オブジェクトとそのデータ・ソースとのリンク設定には、**DsRecipes.xml** 構成ファイルを使用します。またこのファイルでは、データ・オブジェクトの順序性が「1」か「n」かを指定します。データ・オブジェクト・ファイルは、データ・オブジェクトの正確な構造を指定するために使用し、**DsDataElements.xml** 構成ファイルは、各要素のデータ型 (LIST、LONG、STRING など) の指定に使用します。

拡張データ・オブジェクト

XML スキーマ・ファイルでデータ・オブジェクトを定義する際には、Extends 属性を使用して、そのデータ・オブジェクトが別のデータ・オブジェクトの拡張であることを宣言できます。この機能の使用方法には、次の 2 つの方法があります。

- 1 つのデータ・オブジェクトを、同じデータ・フィールド・セットを共有する複数の異なる拡張データ・オブジェクトの親として使用します。例えば、Visual Modeler のデータ・オブジェクトの多くが C3PrimaryRW データ・オブジェクトの拡張です。このデータ・オブジェクトは、アクセス制御の管理に使用される基本データ・フィールド OwnedBy および AccessKey を備えています。
- データ・オブジェクトをカスタマイズするには、そのデータ・オブジェクトの拡張データ・オブジェクトを作成します。拡張データ・オブジェクトにデータ・フィールドを追加することによって、カスタマイズの一部として使用する必要がある属性を追加できます。ObjectManager の使用により、拡張元タイプのデータ・オブジェクトの作成をシステムが要求されたときに、拡張データ・オブジェクトが確実に作成されるようにすることができます。既存のコードで拡張元データ・オブジェクト・インスタンスのインスタンス化に ObjectManager を使用している場合、このコードが呼び出されると、拡張データ・オブジェクトのインスタンスが作成されますが、このインスタンスでは拡張元データ・オブジェクトのインターフェースが継続してサポートされるため、既存のコードが今までどおり機能します。

DataManager は、*レシピ* とデータ・オブジェクト に基づいて、データ Bean またはビジネス・オブジェクトの要素構造と、その要素値を提供するデータ・ソースの場所を判別します。データ・オブジェクト定義を変更した場合や、新しい定義を作成した場合は、SDK ターゲット generateDTD および generateBean を再実行して DataBean クラスを作成およびコンパイルする必要があります。詳しくは、 87 ページの『Visual Modeler の実装をカスタマイズするための Software Development Kit の使用』を参照してください。データ・オブジェクト拡張のその他の方法については、 60 ページの『データ・オブジェクトの拡張』を参照してください。

データ Bean とビジネス・オブジェクトの作成

データ Bean は Visual Modeler の ObjectManager クラスと OMWrapper クラスによって作成され、ビジネス・ロジック・クラスとコントローラーによって処理されます。詳しくは、 24 ページの『ObjectManager クラスと OMWrapper クラス』を参照してください。

ビジネス・ロジック・クラスはコントローラーによって呼び出されます。各コントローラーは、メッセージおよびそのメッセージ・タイプに応じて、どのビジネス・ロジック・クラスを作成する必要があるか (作成する必要がある場合) を判別する役割を担います。

ビジネス・オブジェクトと BusinessObject クラスの使用は非推奨になりました。できるだけデータ Bean クラスを使用するようにし、ビジネス・オブジェクトはレガシー・コードの保守時にのみ使用してください。

DataContext

restore() メソッドは DataContext クラスのインスタンスをパラメーターとして受け取ります。DataContext クラスは、*restore()* 操作の実行のコンテキストに関する情報を指定するために使用されます。このクラスを使用して、返す結果の最大数の指定や、ページごとの結果数の設定 (ページ付け) を行えます。また、このクラスは、*restore()* 操作の結果に対してアクセス・チェックを実行するかどうか指定する目的で使用することもできます。デフォルトでは、アクセス・チェックが実行されます。

例えば、次のコード・スニペットでは、DataContext を作成し、いくつかのコンテキスト値を設定した後、そのコンテキストとクエリーを使用してデータ Bean をリストアします。

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

DataContext オブジェクトは、初期化時に DataServices.General.MaxResults および DataServices.General.NumPerCachePage 要素の値を構成ファイルから取得して、リストア操作に対してこれらのパラメーターを設定します。デフォルトでは、どちらのパラメーターについても制限は設定されません。DataContext の動作を変更する必要がある場合は、accessor メソッドを利用できます。詳しくは、DataContext の Javadoc を参照してください。

DataContext クラスは、ページ付けをサポートする *setCacheId(String cacheId)* メソッドを備えています。このメソッドでは、使用する特定のキャッシュを指定します。

DataContext クラスとは

DataContext クラスは、リストア操作と永続化操作の動作を制御する目的で使用します。

制御可能な動作

DataContext インスタンスでは次の動作を制御できます。

- 1 ページに表示するクエリー結果の数
- 処理するクエリー結果の最大数
- データ Bean タイプおよびセッションごとの、複数のページ・セットの使用

キャッシュ ID メソッドの用途

キャッシュ ID のメソッドを使用すると、アプリケーションで結果セットのページ付けに対して固有の ID を指定できます。この新しい機能により、アプリケーションで特定のデータ Bean およびセッションに対して複数の異なる結果セットを維持することが可能になります。

アプリケーションでキャッシュ ID を指定しないと、Bean 名とセッション ID の組み合わせに基づいてキャッシュが特定されます。この場合、同じセッション内の同じデータ Bean に対する後続のリストア試行が発生した場合は、あらゆる結果が上書きされます。

DataContext クラスは、データ Bean のリストア要求に対するキャッシュ ID の制御用に次のメソッドを備えています。

- void setCacheId(String cacheId): 新しいキャッシュ ID を設定します。この文字列を Bean 名およびセッション ID と組み合わせて使用することで、固有の ID が生成されます。
- String getCacheId( ): 現在のキャッシュ ID (または、キャッシュ ID が設定されていない場合は NULL) を返します。

最大結果数と 1 ページあたりの件数の動作

最大結果数の設定は、リストア時に取得可能な最大レコード数を決定します。この数に達すると、要求は解放されます。

1 ページあたりの件数の設定は、各結果キャッシュ・ページに保存するレコード数を決定します。検出されたレコード数が 1 ページあたりの件数の設定値より少ない場合、結果キャッシュは何も作成されません。

アプリケーションでこれらの属性を組み合わせて使用することで、取得する最大レコード数を指定しながら、ページ付けした結果セットを取得することが可能です。

DataContext クラスは、データ Bean のリストアおよび永続化要求に対する最大結果数と 1 ページあたりの件数の指定用に次のメソッドを備えています。

- void setMaxResults(int maxResults): 返す最大結果数 (ページ付けなしの結果) を設定します。
- int getMaxResults( ): 返す最大結果数 (ページ付けなしの結果) を取得します。
- void setMaxPaginatedResults(int maxResults): 返す最大結果数 (ページ付けした結果) を設定します。
- int getMaxPaginatedResults( ): 返す最大結果数 (ページ付けした結果) を取得します。
- void setNumPerPage(int numPerPage)
- int getNumPerPage( )

アプリケーションでデータ・サービスのデフォルトの制限を使用する場合は、DataContext の該当するプロパティーを **DsConstants.USE_DEFAULT** に設定する必要があります。デフォルト値を以下に示します。

- maxResults: 125

- maxPaginatedResults: 125
- numPerPage: 25

アプリケーションで numPerPage の値を指定しないと、prefs.xml に指定された値が使用されます。アプリケーションで値が指定されず、prefs.xml ファイルにも値が設定されていない場合は、要求をページ付けしないことを意味する値 -1 が使用されます。

また、以下のメソッドが提供する結果セット制限は、SQL クエリーの一部としてデータベースに直接渡されます。Visual Modeler のアクセス・ポリシー・チェック (ユーザーはそのデータを表示する権限を与えられているか、など) の一部として結果が破棄されることがあるため、これらのメソッドを使用して上位の結果セット制限を設定できます。

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

このほか、DataServices.General.LimitDBResults プリファレンスも設定できます。LimitDBResults を TRUE に設定すると、MaxResults (またはページ付けした結果の場合 MaxPaginatedResults) で許容されている数に結果が自動的に制限されます。このメカニズムを使用するには、アクセス・ポリシーが SQL として表現されていなければなりません。Oracle データベースでは、LimitDBResults プリファレンスを TRUE に設定しないでください。

IBM のアクセス・ポリシーは次の 2 つのいずれかの方法で処理されます。多くのものは SQL の WHERE 節に変換され、クエリーに適用されます。それにより、データベースでそのアクセス・ポリシーを処理できるようになります。ポリシーが複雑すぎる場合 (例えば、ポリシーがパートナーの階層に依存している場合など)、そのアクセス・ポリシーの適用が可能なのは、データベースからの結果の処理時に限られます。このようなポリシーは SQL に変換できません。

Oracle では、SQL の生成時に XML スキーマでの列の別名の定義を必要とする場合があります。これが必要となるのは、同じ列名を使用する複数の表をクエリーで結合する場合に限られます。この問題は SQL Server および DB2® には関連しません。

DataContext インスタンスのインスタンス化方法

現在、新しい DataContext インスタンスは、以下の標準の「new」メカニズムを使用してインスタンス化します。

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

new DataContext のデフォルト設定の内容

「new DataContext( )」の呼び出し時に、各属性は以下のデフォルト値を受け取ります。

属性　デフォルト値

doAccessCheck
TRUE

maxResults
DataServices.xml 内の maxResults プロパティー

numPerPage
DataServices.xml 内の numPerPage プロパティー

CacheId
NULL

doAccessCheck
TRUE

リスト・データ Bean

リスト・データ *Bean* およびリスト・ビジネス・オブジェクト と呼ばれる特殊なビジネス・オブジェクト・クラスがあります。このようなクラスは、同じタイプのデータ・オブジェクトのリストの管理に使用します。レシピ要素内でデータ・オブジェクト要素が順序性を「n」として宣言されると、常にリスト・データ Bean が作成されます。この場合も、アクセス資格は単一のビジネス・オブジェクト・レベルで管理されます。

注: 以前のバージョンのデータ・オブジェクトでは、データ・オブジェクトの定義ファイルで順序性を定義していました。現在では、データ・オブジェクトの順序性はレシピ・ファイルで設定します。バージョン 6.0 のデータ・オブジェクトでは、子、参照、および組み込みデータ・オブジェクトを宣言する目的で、順序性 (Ordinality) 属性を引き続き使用します。

通常、リスト・データ・オブジェクト用の DataBean を作成する必要はありません。DataBean は自動的に作成されます。詳しくは、23 ページの『DataBean クラス』を参照してください。この DataBean では、データ・オブジェクトのリストを返す、自動生成されるメソッドをサポートしています。例えば、次のコード片はユーザー・リストのリストア方法を示しています。「context」で指定された DataContext オブジェクトと、「query」として指定された DsQuery オブジェクトを使用して、*restore()* 呼び出しで返すユーザーを制限しています。

```
UserListBean userList = (UserListBean)
OMWrapper.getObject("com.comergent.bean.simple.UserListBean");
// Restore the list.
userList.restore(context, query);
// Return immediately if no results found.
if (userList.getUserCount() == 0)
{
return;
}
// At least one user in list, so walk through the list of users
ListIterator userIterator = userList.getUserIterator();
while (userIterator.hasNext())
{
UserBean user = (UserBean) userIterator.next();
// Perform any business logic on each user.
}
```

注: *restore()* メソッドでの DataContext および DsQuery パラメーターの使用: これらは知識ベースに対するクエリーの実行方法を管理する目的で使用します。

アプリケーション Bean、エンティティー Bean、およびプレゼンテーション Bean

Visual Modeler で使用される Bean の主な種類として、データ Bean、アプリケーション Bean、エンティティー Bean、およびプレゼンテーション Bean があります。このセクションでは、これらの Bean の主な相違点について説明します。

- データ Bean は、ビジネス・オブジェクトの XML スキーマ記述から自動的に作成される Java クラスです。generateBean SDK ターゲットを実行すると、各データ Bean のソース・コードが生成されます。これらの Bean の集まりが com.comergent.bean.simple パッケージになります。

  できるだけ、*instanceof* コマンドを使用してデータ Bean クラスを判別するようにし、ビジネス・オブジェクト・タイプについてクエリーを実行するのは避けてください。
- アプリケーション Bean は、単純 Bean では対応していない機能を追加する目的で作成される Java クラスです。例えば、アプリケーション Bean によって、自動生成されない追加メソッドを確保したり、複数の単純 Bean を結合してデータを JSP ページに渡したりすることができます。アプリケーション Bean はアプリケーションごとに構成され、各アプリケーションに、そのアプリケーション Bean 用のパッケージが作成されます。パッケージ名は、com.comergent.apps.<アプリケーション名>.bean です。

  アプリケーション Bean は単純 Bean のサブクラスのこともありますが、多くの場合、1 つ以上の単純 Bean をメンバー変数として含む Java クラスです。

  例えば、アプリケーション Bean クラス com.comergent.appservices.productService.productMgr.BizProductBean は、com.comergent.bean.simple.IDataProduct インターフェースを実装するメンバー変数を含む Java クラスです。アプリケーション Bean クラス BizProductBean は、*getProductID()* などのメソッドを com.comergent.bean.simple.IDataProduct メンバー変数に委任しますが、そのほかに製品の機能、交換チェーン、および価格を取得するメソッドを提供します。ProductDataBean 自体ではなく IDataProduct インターフェースを使用する点に注意してください。これはクラスではなく、生成されたインターフェースを使用する例です。これらのインターフェースの生成と使用について詳しくは、81 ページの『第 12 章 生成インターフェース』を参照してください。

  慣例から、データ Bean をラップするアプリケーション Bean を作成する場合は、対象のデータ Bean を取得するメソッド *getDataBean()* を準備する必要があります。
- プレゼンテーション Bean も JSP ページへのデータの受け渡しに使用されます。一般に、この Bean とアプリケーション Bean との違いは、プレゼンテーション Bean の場合、ビジネス・ロジックを提供しないことです。プレゼンテーション Bean では、複数のデータ Bean を使いやすいように 1 つのクラスに集約したり、フォーマット情報を提供したりすることができます。プレゼンテーション

Bean は、アプリケーション Bean と同じく、基礎となるデータ Bean にアクセスするためのメソッドを備えていなければなりません。例えば、IPresProduct インターフェースは *getIRdProduct()* メソッドを備えています。このメソッドは IRdProduct インターフェースを返します。これを基礎となるデータ Bean または必要に応じて拡張元データ Bean にダウンキャストすることができます。

- エンティティー Bean は、以前のリリースの Visual Modeler で使用されていたもので、アプリケーション Bean と同じ役割を果たしていました。現在、この使用は非推奨となっています。

ストアード・プロシージャーの使用

データ・オブジェクトのリストに、ストアード・プロシージャーを活用できます。ストアード・プロシージャー名は、データ・オブジェクトの ExternalName 要素で宣言します。

データ・オブジェクトを定義する際には、SourceType 属性の指定に注意してください。この属性には次の値を設定できます。

- 「1」: 基礎となるデータ・ソースで表を使用します。これはデフォルト値です。
- 「2」: 基礎となるデータ・ソースでストアード・プロシージャーを使用します。

SourceType 属性を定義しないと、データ・オブジェクトの基礎となるソース・タイプが表であることを意味するデフォルト値が使用されます。

データ Bean のメソッド

通常は、各データ Bean で生成される付属のインターフェースを活用するようにします。このインターフェースによって、データ・オブジェクトのライフサイクルを通してオブジェクトへのアクセスを管理するのに役立つ accessor メソッドとデータ・メソッドが構成されます。詳しくは、 81 ページの『第 12 章 生成インターフェース』を参照してください。

アクセス制御を行うには、アクセス・ポリシー・セキュリティー・メカニズムを使用します。

IData のメソッド

IData インターフェースは以下に示す重要なメソッドを備えています。

- *copyBean()*: このメソッドでは、Bean 間でデータ・フィールドの値をコピーできます。引数を 1 つ取ります。これは、このメソッドの呼び出し元 Bean と同じクラスのインスタンスである Bean またはサブクラスの Bean でなければなりません。
- *delete()*: このメソッドは対応するデータ・オブジェクトを削除済みとしてマーク付けします。データ・オブジェクトが永続化されたときに、そのデータ・オブジェクトに対応するデータベース表の ACTIVE_FLAG 列が「N」に設定されます。*delete()* の呼び出し後に *persist()* を呼び出す必要があります。この呼び出しを行わないと、削除は有効になりません。
- *erase()*: このメソッドはビジネス・オブジェクトに対応するデータベース・レコードを削除します。レコードをデータベース表から削除した場合、その削除レコー

ド内のキーを別の表が参照していると、データの保全性に問題が発生することがあります。通常このメソッドの使用は、削除対象のレコードおよびそのキーのすべての使用先を把握しており、対応するレコードを他の表から削除できる場合に限定してください。

- *generateKeys()*: このメソッドはデータ Bean のキー・フィールドにデータを設定します。このメソッドは、*persist()* の呼び出しなしで呼び出せます。このメソッドの呼び出しにより、生成されたキーを使用して、そのキーを必要とする他のオブジェクトを作成できます。
- *setDataContext()*: このメソッドは、*restore()* および *persist()* の呼び出しで、ページ付けされたデータ・セットの 1 ページあたりの結果数などのパラメーターに正しい値が使用されるようにするためのデータ・コンテキストを設定します。DataContext クラスについて詳しくは、50 ページの『DataContext』を参照してください。
- *persist()*: このメソッドはデータ Bean 内のデータをそのデータ・ソースに保存します。
- *prune()*: このメソッドは、メモリー内で Bean を削除対象としてマーク付けするために使用します。*prune()* の後に *restore()* を呼び出した場合、その Bean の基礎となるデータ・ソースには何も影響を及ぼしません。
- *restore()*: このメソッドはデータ Bean をそのデータ・ソースから取得します。*restore()* メソッドでの DataContext クラスの使用については、50 ページの『DataContext』 を参照してください。
- *update()*: このメソッドは対象のビジネス・オブジェクトに対応するデータベース・レコードを更新します。

状態を変更するメソッド呼び出しでは、いずれも、そのメソッドの後に *persist()* を呼び出す必要があります。この呼び出しを行わないと、データベース・レコードの実際の変更は行われません。

このほか、IData インターフェースは、オブジェクトがリストア可能かをチェックする *isRestorable()* メソッドと、永続化可能かをチェックする *isPersistable()* メソッドも備えています。

IRd および IAcc インターフェースのメソッド

IRd インターフェースは、データ・オブジェクト・フィールドに対する読み取り専用の accessor メソッドを備えています。IAcc インターフェースは、各データ・フィールドに対する set accessor メソッドを追加した、IRd インターフェースの拡張です。これら 2 つのインターフェースを区別することによって、クライアント・アプリケーションまたは JSP ページに読み取り専用オブジェクトを渡すことができます。

例えば、条件データ・オブジェクト・ファイル **Condition.xml** で DataField 要素が次のように指定されているとします。

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

その後、自動生成される IRdCondition インターフェースは次のようなメソッドを保有します。

public Long getControlType()

自動生成される IAccCondition インターフェースは次のようなメソッドを保有します。

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

これらの accessor メソッドのシグニチャーは、**DsDataElements.xml** ファイル内の対応する DataElement 定義によって決まります。

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

注: データ・フィールドの Writable 属性を「n」に設定した場合、対応する *set*DataField*()* メソッドは生成されません。

データのリストアおよび永続化

データ・オブジェクトに対して、*delete()*、*persist()*、および *restore()* という重要な操作を実行できます。

- データ・オブジェクトで *delete()* メソッドを呼び出すと、そのオブジェクトは削除済みとしてマーク付けされ、他のアプリケーションから取得されなくなります。基礎となるデータベース表で、ACTIVE_FLAG 列の値が「N」に設定されます。この場合、そのデータ・オブジェクトがデータ・ソースから削除されるわけではありません。データ・オブジェクトの基礎となるデータベース表に ACTIVE_FLAG 列がない場合は、*delete()* メソッドを使用しないでください。代わりに、*erase()* メソッドを使用してこのようなデータ・オブジェクトを知識ベースから削除できます。
- データ Bean に対して *persist* を実行すると、そのデータ・オブジェクトの DsElement ツリーに保持されているデータが Visual Modeler によってその外部データ・ソースに保存されます。Visual Modeler では、*persist()* メソッドにより、既存のデータ・オブジェクトの更新と新しいデータ・オブジェクトの作成の両方が処理されます。
- データ Bean またはビジネス・オブジェクトに対して *restore* を実行すると、そのデータが Visual Modeler によって外部データ・ソースから取得されます。*restore()* メソッドでクエリー・オブジェクトを指定しないと、キー・フィールドの値がデータ Bean の対応する値に一致するすべてのデータ・オブジェクトがリストアされます。
  - リスト・データ Bean 以外のものに対して *restore()* を呼び出すときは、そのデータが、そのキー・フィールドに含まれている値セットから一意的に取得可能であることを前提としてください。*restore()* 呼び出しの発行時、単一レコードのみの取得を検証するチェックは行われないため、取得された最初のレコードがデータ Bean に設定されます。レコードがまったく取得されないと、*restore()* 呼び出しから ICCException がスローされます。

– リスト・データ Bean で *restore()* を呼び出すときは、通常 DsQuery を指定する必要があります。DsQuery をまったく指定しないと、リストアされたリスト・データ Bean にはこのタイプのすべてのデータ Bean が取り込まれます。

DataBean restore() メソッド

このセクションでは、DataBean *restore()* メソッドの主な形式について説明します。

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

restore() メソッドの基本形です。dsQuery パラメーターを使用して、リストア操作で実行するクエリーを指定します。dataContext パラメーターは、返されるオブジェクトの最大数と、1 ページあたりの結果数 (ページ付けを行う場合) を決定します。この dataContext パラメーターを使用して、現在のユーザーがこの操作を実行するための適切な資格を持っているかどうかのチェックを実行するかどうかを指定します。デフォルトでは、アクセス・チェックが実行されるため、このチェックが行われないようにするには、*disableAccessCheck()* メソッドを使用してアクセス・チェックをオーバーライドします。

public void restore(DataContext dataContext)

これは *restore(dataContext, null)* を呼び出すのと同じになります。

次に、DataContext クラスと DsQuery クラスを一緒に使用して *restore()* の呼び出しを制御する例を示します。

```
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));
}
```

DataBean persist() メソッド

このセクションでは、DataBean *persist()* メソッドの主な形式について説明します。

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

dataContext でアクセス・チェックの実行が指定されている場合、この形式の *persist()* メソッドは操作の実行前にアクセス・チェックを実行します。ユーザーが適切な資格を持っていないと、操作は実行されません。

その他のメソッド

getBizObj() メソッド

データ・オブジェクトのビジネス・オブジェクト表現とそのデータを取得するには、*getBizObj()* メソッドを呼び出します。このメソッドは、オブジェクトの内部構造を表示する必要がある場合に役立ちます。次に例を示します。

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

このメソッドは、現在は非推奨となっています。

writeExternal() メソッド

このメソッドは、データ Bean の XML 表現とそのデータを書き出すときに使用します。

子データ・オブジェクト

データ・オブジェクトの多くでは、子データ・オブジェクトの宣言に ChildDataObject 要素を使用します。例えば、ShoppingCart データ・オブジェクトでは、子データ・オブジェクトとして LineItem を次のように宣言します。

```
<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>
```

この Relationship 要素には、親の更新時に子オブジェクトをどのように処理するかと、子の変更時に親を更新するかどうかを定義する属性があります。JoinKey 要素では、子データ・オブジェクトのリストア方法を定義します。通常、この定義では、親データ・オブジェクト内の値をどのように使用して子データ・オブジェクトを設定するかを指定します。

親データ Bean の生成時には、子データ Bean を格納する ListIterator オブジェクトを返す *get*ChildDataObject*Iterator()* メソッドが生成されます。各オブジェクトを反復処理することによって、それぞれのオブジェクトの子データ Bean を順番にチェックし、標準の accessor メソッドを使用して子データ Bean のフィールドにアクセスできます。

例えば、ShoppingCartBean クラスは getLineItemIterator() メソッドをサポートしています。次のコード行は、明細アイテムのフィールドを取得する方法を示しています。

```
/*
shoppingCartBean is a ShoppingCartBean object that has already been
restored
*/
ListIterator lineItemIterator =
shoppingCartBean.getLineItemIterator();
LineItemBean lineItemBean =
(LineItemBean) lineItemIterator.getLineItemBean(0);
Long quantity = lineItemBean.getQuantity();
```

子データ・オブジェクトは、親データ・オブジェクトのリストア時にはリストアされません。子データ・オブジェクトは、上記の方法でアプリケーションからアクセスされたときに初めてリストアされます。

データ・オブジェクトの拡張

このタスクについて

Visual Modeler のどの実装環境でも、データ・オブジェクトへのデータ・フィールドの追加や、既存のデータ・オブジェクトを拡張するデータ・オブジェクトの作成が必要になることがよくあります。

追加データは新しいデータベース表に格納することをお勧めします。そして、新しい表にアクセスする DataObject を新たに定義してください。その後、新しい IncludeDataObject を追加して、元の DataObject を拡張する別の DataObject を新たに定義します。

例えば、Order データ・オブジェクトに、「ホスト (hosted)」オーダー (店舗パートナーが受けたオーダー) を追跡するための新しいデータ・フィールドを追加する必要が生じたとします。追加データ・フィールドは店舗パートナーのパートナー・キ

ーです。この場合の推奨アプローチを以下に示します。

手順

1. 厳密に 2 つのフィールド OrderKey と PartnerKey を持つ、HostedPartner という新しいデータ・オブジェクトを作成します。ORDER_KEY と PARTNER_KEY という 2 つの列を持つ表 CMGT_ORDER_X_PARTNER を指すように、このデータ・オブジェクトを設定します。

```
<?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. Order を拡張する新しいデータ・オブジェクト HostedOrder を作成します。**HostedOrder.xml** ファイルの内容は次のようなものです。

```
<?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>
```

   以下に示す 3 つの基本アプローチを使用できます。

3. 拡張を使用して新しい DataField を単純に追加し、表名をオーバーライドします。それにより、新しい表にすべてのデータを組み込めます。このアプローチが最も役立つのは、データがまったく同じ内容のコピーが別に必要な場合です (必要に応じて、Order を HostedOrder に変える前に Order がどのようなものか示すスナップショットを維持しておきます)。

4. Order を拡張して HostedOrder 用の IncludedDataObject を追加します。この HostedOrder では、別の表に格納する追加データのみを定義します。この場合、元の Order DataField の変更については、Order 表に引き続き永続化し、HostedOrder の追加データは別の表に永続化することになります。これは上述の推奨アプローチです。
5. Order が IncludedDataObject であることを指定して、HostedOrder を定義します。それにより、2 番目のアプローチと同じ結果になります。このアプローチの問題点は、HostedOrder が Order の拡張ではないため、アプリケーション・コードで Order として扱えなくなることです。

   注: 2 つの表を使用すると、パフォーマンスの面で多少難がありますが、クエリーの実行は問題領域とはなっていません。2 つの表を使用することで、(要件の内容によっては) データの冗長性が軽減される場合があります。

タスクの結果

カスタマー拡張を頻繁に参照するわけでない場合は、ChildDataObject を使用して遅延リンク・メカニズムを活用する方法もあります。

データ Bean の例

このセクションでは、データ・オブジェクトの定義および使用プロセスについて説明します。データ・オブジェクトを使用して、顧客からの単純な問い合わせを表現する場合を考えてみましょう。このオブジェクトは以下のもので構成されます。

- 顧客の E メール・アドレス
- 問い合わせ日
- 応答日 (オプション)
- 問い合わせ内容
- 応答の内容 (オプション)
- 問い合わせに関連する製品 ID (オプション)

データ・オブジェクト定義の作成

このタスクについて

データ・オブジェクトの定義を作成する手順は次のとおりです。

手順

1. Enquiry というビジネス・オブジェクト (BusinessObject) 要素を作成し、この要素を **DsBusinessObjects.xml** ファイルに追加します。

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

   Version 属性では、そのビジネス・オブジェクトのさまざまなバージョンが同時に使用される場合に各バージョンを管理できます。Version 属性は、アクセス・チェックを自動的に実行するか (バージョン 5.0 以上) どうかの設定にも使用されます。

2. このビジネス・オブジェクトのレシピ (Recipe) を作成し、このレシピを **DsRecipes.xml** ファイルに追加します。

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

レシピの Name 属性は、ビジネス・オブジェクトの Name 属性と完全に一致させる必要があります (大/小文字は区別されます)。リリース 9.1 では、1 つのレシピに対して、複数のビジネス・オブジェクトをデータ・オブジェクト・リストに定義できますが、そのうち*書き込み可能* データ・オブジェクトとすることができるのは 1 つだけです。これらのデータ・オブジェクトでは、それぞれのデータ・オブジェクト (DataObject) 要素の属性としてデータ・ソース名を定義します。これらのエントリーに基づいて、ビジネス・オブジェクトのデータの取得元とする各ソースと、永続化先として使用可能なソースが決まります。

3. データ・オブジェクトを定義する **Enquiry.xml** というファイルを作成します。データ・オブジェクト (DataObject) 要素の Name 属性は、レシピ (Recipe) 要素の DataObject の設定に定義されている Name 属性と完全に一致させる必要があります (大/小文字は区別されます)。

この例では、これらのデータ・オブジェクトのデータを CMGT_ENQUIRY というデータベース表に保持し、各 DataField 要素の ExternalFieldName 属性で DataField 値の取得に使用する列を指定しています。例えば、CMGT_ENQUIRY 表の EMAIL_ADDRESS 列には問い合わせに関連付けられた E メール・アドレス値が保持されます。

```
<?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>
```

TimeToRespond データ・フィールドの定義に注目してください。このデータ・フィールドはデータベースの列に対応していないため、定義に ExternalFieldName 属性が含まれていません。このフィールドの値は実行時に計算され、フィールド値を表示できるように EnquiryBean に設定されます。

4. **DsDataElements.xml** で、DataElement として Enquiry および EnquiryList を定義します。

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

5. **DsDataElements.xml** で、各 DataField の DataElement を定義します。DataElement では、DataManager がこのビジネス・オブジェクト・タイプのデータの取得または保存時に使用するデータ型情報を指定します。次に例を示します。

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

EmailAddress と SKU については、DataElement を組み込んでいない点に注目してください。これらの DataField の DataElement は既に定義されており、DataElement は何回でも再利用できます (ただし、データ型が常に同じでなければなりません)。

6. このデータ Bean の **ObjectMap.xml** ファイルを設定します。次に例を示します。

```
<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. 最後に、DataObject 要素で定義した DataSourceName 属性に対応するデータ・ソース (DataSource) 要素を定義します。このデータ・ソースは、スキーマの一部として DsDataSources.xml ファイルに定義します。ほとんどの場合、このデータ・ソースは既に定義されています。新しいデータ・ソースを定義する必要があるのは、別のデータベース、または知識ベースの他の部分とは異なるデータ・ソースを使用する場合に限られます。次に例を示します。

   ```
   <DataSource Name="ENTERPRISE" Version="2.0">
   <Primary Type="SQL" DataService="JdbcService"
   SubType="ORACLE"
   ConnectionString="jdbc:<driver>:<server>:<port>:<sid>"
   UserId="userid" Password="password" />
   <Alternate Type="SQL" DataService="JdbcService"
   SubType="MSSQL"
   ConnectionString="jdbc:<driver>:<server>:<port>:<sid>"
   UserId="userid" Password="password" />
   </DataSource>
   ```

   Primary 要素と Alternate 要素の DataService 属性では、EnquiryBean の *restore()* メソッドと *persist()* メソッドの処理に使用するクラスを設定します。その他の属性では、外部ソースへの厳密なアクセス方法を設定します。

8. generateBean SDK ターゲットを実行して、新しいデータ Bean EnquiryBean および EnquiryListBean とそれぞれの対応するインターフェースのソース・コードを生成します。これらのインターフェースについて詳しくは、81 ページの『第 12 章 生成インターフェース』を参照してください。

タスクの結果

これで、Enquiry データ Bean および対応するインターフェースをビジネス・ロジック・クラスで使用できるようになりました。Enquiry データ Bean のインスタンスを作成するには、次の形式のメソッドを呼び出します。

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

このメソッドにより、EnquiryBean データ Bean が返されます。このデータ Bean は、Enquiry DataObject で指定された構造になっています。QueryBean クラスのイ

ンスタンスが得られたら、次にそのキー・フィールドにデータを設定し、この Bean をリストアして Bean のデータを取得します。

```
int queryIndex = 0;
try
{
String queryKey = request.getParameter("querykey");
queryIndex = Integer.parseInt(queryKey);
}
catch (Exception e)
{
//Throw exception if parameter not valid
}
QueryBean queryBean = (QueryBean)
OMWrapper().getObject("com.comergent.bean.simple.EnquiryBean");
queryBean.setKey(queryIndex);
queryBean.restore();
```

問い合わせリストの取得方法は次のとおりです。

```
// Use default settings for DataContext parameters
DataContext context = new DataContext();
// Retrieve enquiries relating to a particular product ID, MXWS-7000
DsQuery query =
QueryHelper.newWhereClause("SKU", DsQueryOperators.EQUALS,
"MXWS-7000");
EnquiryListBean enquiryList = (EnquiryListBean)
OMWrapper.getObject("com.comergent.bean.simple.EnquiryListBean");
// Restore the list.
enquiryList.restore(context, query);
// Walk through the list...
ListIterator enquiryIterator = enquiryList.getEnquiryIterator();
while (enquiryIterator.hasNext())
{
boolean isModified = false;
EnquiryBean enquiry = (EnquiryBean) enquiryIterator.next();
// Process each enquiry
}
```

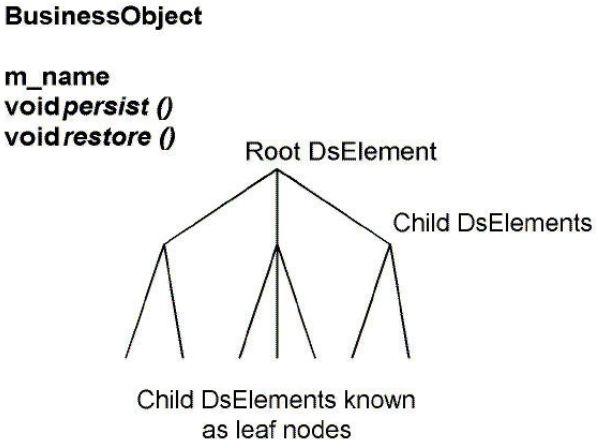
通常、EnquiryBean を使用するアプリケーションでは、データ Bean 自体ではなく、生成されたインターフェースの 1 つを使用するようにします。それにより、アプリケーションでデータ・オブジェクトの実装をそのインターフェースから切り離し、そのオブジェクトのデータに対してアプリケーションが持つアクセス権の内容を管理できます。IAccEnquiry インターフェースを実装するクラスのインスタンスを取得するには、次のようにします。

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

DsElement ツリー

このセクションでは、データ Bean のメタデータを取得するメソッドについて説明します。また、データ・オブジェクトおよびビジネス・オブジェクトのクラスのデータの格納に使用される DsElement ツリーについても説明します。ここでの説明は、レガシー・アプリケーションのサポートを目的とするものです。データ Bean クラスを使用する新しいアプリケーションについては、いずれもこの説明は適用されません。

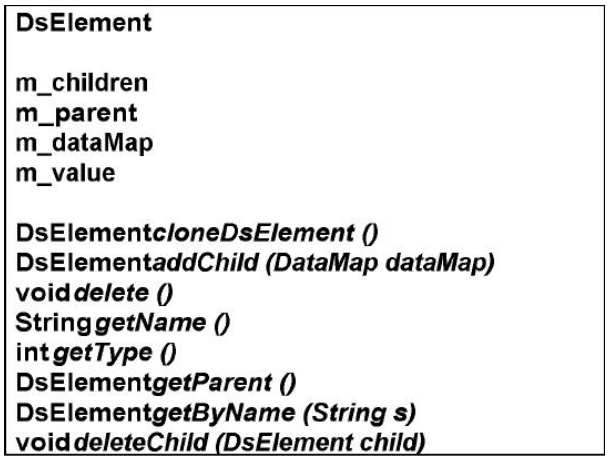
データ・オブジェクトは、データ Bean クラスのオブジェクトとして作成されます。各データ・オブジェクトの内容は、DsElement というコンポーネント・ツリーとして、そのオブジェクト内に格納されます (『DsElement』を参照)。この内容は、XML スキーマと、その XML スキーマで定義されているレシピおよびデータ・ソースに基づいて、外部システムから取得されます。次の図はビジネス・オブジェクトを示したものです。



DataManager は、データ Bean またはビジネス・オブジェクトを作成するときに、XML スキーマに基づいてその DsElement ツリーの構造を判定します。DsElement ツリーはビジネス・オブジェクトの構造の Java 表現です。また、リーフ・ノードで挿入可能なデータ型と、ノードの値に制限を適用するかどうかも、スキーマによって決まります。DsElement ツリーにアクセスするには、ビジネス・オブジェクトの *getRootElement()* メソッドを呼び出します。

DsElement

各 DsElement には、データと、データがそのデータ・ソースにどのように対応するかを定義する DataMap が格納されます。個々の DsElement は別の DsElement (その親) の子である可能性があります。DsElement ツリーは DsElement の集合であるため、1 つを除くすべての要素がツリー内の別の要素 (親) の子要素です。当然ながら、親が NULL の DsElement がルート DsElement です。次の図は DsElement のメソッドを示したものです。



DsElement クラスは、DsElement ツリー内の移動をサポートするさまざまな追加メソッドを備えています。代表的なものとして、特定の DsElement の子 DsElement のイテレーターを返す *children()* があります。このビジネス・オブジェクト・クラスは、*getRootElement()* のほかに、ツリー内の指定された DsElement に直接アクセスする *getElementByName()* メソッドを備えています。

同じ名前 (例えば、child_name) を持ち、かつ別の DsElement の子となっている DsElement はすべて、<child_name>List という名前の親を持っていなければなりません。XML スキーマでは、順序性を「1」ではなく「n」として定義することによって、このような要素を指定します。DsElement の子は m_children という Vector に維持されます。

DsElement は以下の重要なメソッドを持っています。

- *addChild()*: その DsElement の DataMap に定義されている新しい DsElement を追加します。
- *cloneDsElement()*: その DsElement のコピーを返します。
- *delete()*: DsElemState を DsElemState.DELETED に設定します。
- *deleteChild()*: DsElement として子を指定することによって、Vector m_children からその子を削除します。
- *getName()*: その MetaData に定義されているその要素の名前を返します。
- *getParent()*: その DsElement の親を返します。
- *getType()*: その DataMap に定義されているその要素のタイプを返します。

DsElement MetaData

データ・フィールドおよびその基礎となる DsElement に関する情報を取得すると便利なことがあります。この目的で、IData インターフェースの *getMetaData(String elementName)* メソッドを使用できます。このメソッドは IMetaData インターフェースを実装するオブジェクトを返します。このインターフェースは次のメソッドをサポートしています。

- public int getDataType(): DsDataTypes に定義されている値を返します。
- public long getMaxLength(): 最大長をバイト数で返します。
- public long getMaxCharLength(Locale locale): 最大長を文字数で返します。

- public Object getMinValue(): 許容される最小値 (最小値がなければ NULL) を返します。
- public Object getMaxValue(): 許容される最大値 (最大値がなければ NULL) を返します。
- public int getCountAllowedValues()
- public ListIterator getAllowedValueIterator()
- public Object getDefaultValue()

注: 生成される各 DataBean クラスには IData インターフェースが実装されるため、これらのメソッドは生成されるすべてのデータ Bean で使用できます。

BusinessObject メソッド

ビジネス・オブジェクトの使用は非推奨となりました。このセクションでは、参考用として、ビジネス・オブジェクトの一部のメソッドに関する情報を提供します。

BusinessObject restore() メソッド

このセクションでは、BusinessObject *restore()* メソッドの主な形式について説明します。

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

restore() メソッドの基本形です。queryObj パラメーターを使用して、リストア操作で実行するクエリーを指定します。maxResults パラメーターは、返されるオブジェクトの最大数を決定します。accessCheck パラメーターを使用して、現在のユーザーがこの操作を実行するための適切な資格を持っているかどうかのチェックを実行するかどうかを指定します。アクセス・チェックが完了すると、*restore(BusinessObject queryObj, int maxResults)* が呼び出されます。

public void restore(BusinessObject queryObj, int maxResults)

このメソッドは、基礎となるデータ・オブジェクトの *restore()* メソッド *restore(this, queryObj, maxResults, false)* を呼び出します。

public void restore(BusinessObject queryObj)

これは、*restore(queryObj, 0)* を呼び出すのと同じになります。

```
public void restore()
```

このメソッド形式は *restore(null, 0)* メソッドを呼び出します。

BusinessObject persist() メソッド

このセクションでは、BusinessObject *persist()* メソッドの主な形式について説明します。

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

各 boolean パラメーターはそれぞれ、永続化操作を同期化するか、基礎となるデータ・ソースにコミットするか、永続化前にアクセス・チェックを実行するかどうかを決定します。

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

このメソッド形式は、Version 属性が 4.0 以下のビジネス・オブジェクトでは *persist(synch, commit, false)* と同じになります。Version 属性が 5.0 以上のビジネス・オブジェクトでは *persist(synch, commit, true)* と同じになります。

```
public void persist()
```

このメソッド形式は *persist(false, true)* を呼び出します。

BusinessObject クラスは以下のメソッドも備えています。

- *delete()*: ビジネス・オブジェクトの DsElement ツリーを削除することによって、そのビジネス・オブジェクトを空にします。
- *getRootElement()*: DsElement ツリーのルート DsElement を返します。
- *getType()*: DsElement ツリーのルート要素の名前を返します。これはビジネス・オブジェクトのタイプです。
- *setRootElement()*: このビジネス・オブジェクトのルート要素を設定します。

第 9 章 Visual Modeler でのロギング

Visual Modeler でのロギング: 概要

Visual Modeler によるロギング・メカニズムを使用すると、アプリケーションの書き込み機能により Visual Modeler のアクティビティーのログ記録が可能になります。 ロギング動作を設定するには、log4j API および **log4j.properties** 構成ファイルが使用されます。ロギング機能は、データ・オブジェクトに対する変更の監査もサポートしています。詳しくは、72 ページの『データ・オブジェクトへの変更の監査』を参照してください。

log4j API は、Visual Modeler のロギング動作を管理するための柔軟で拡張可能なロギング・フレームワークを提供します。本セクションでは、Visual Modeler をカスタマイズおよび拡張する場合のフレームワークの使用について説明します。

本フレームワークは Visual Modeler で使用された以前のフレームワークとは異なることに注意してください。今回のフレームワークはグローバル・クラスの *log*Level*()* メソッドです。これらは現在推奨されていません。

log4j API を使用するには、次の行に従って、各クラス・ファイルに Logger クラスを作成する必要があります。

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

ログ・エントリー呼び出しを作成する場合は、次のようにします。

```
log.info("This is a log entry");
```

呼び出すメソッドは、メッセージを記録するロギング・レベルにより異なります。次のメソッドが使用できます。

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

log(priority, message) も使用することはできますが、通常は上記のメソッドで十分です。

log4j.debug システム・プロパティー

log4j.debug システム・プロパティーを TRUE に設定すると、現在のログ設定をエコー出力できます。例えば、サーブレット・コンテナー始動スクリプトに次をインクルードします。

```
-Dlog4j.debug=true
On startup, you should see logging output like this:
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.
```

データ・オブジェクトへの変更の監査

多くの実装環境で、 Visual Modeler のデータへの変更を追跡する監査証跡の提供を求められる場合があります。データ・オブジェクトへのあらゆる変更をログに記録することで、この要望を実現することができます。DataBean クラスでロギング・レベルを INFO 以上に設定すると、このクラスのインスタンスで *persist()* がいつ呼び出されても、ログ・メッセージは Logger に書き出されます。例えば、次の行は、パートナーに対して変更が加えられた場合に書き出される行の例です。

```
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
```

管理 UI を使用すれば、Visual Modeler のどのクラスのロギング・レベルも随時変更することができます。ただし、この場合、ロギング・レベルへの変更は一時的なもので、サーブレット・コンテナーが再起動されると変更は無効になります。さらに、ロギングが書き出されるのはセキュアでない可能性のある標準のアペンダーです。

log4j.properties 構成ファイルをカスタマイズして、監査ロギングを指定する必要があります。これにより確実に、サーブレット・コンテナーが再起動しても監査は続行され、監査情報を処理するためにカスタム・アペンダーを指定できるようになり

ます。例えば、アペンダーがロギング・メッセージをリモート Web サーバーに通知するよう指定することができます。リモート Web サーバーは、Visual Modeler とは関係なく保護されています。

例として、次の **log4j.properties** 構成ファイル内の項目は、UserContact データ・オブジェクトに対するすべての変更が監査されたことを確認します。

```
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
```

Visual Modeler からの監査情報を保存するためにリモート・ログ・サーバーを asa クライアントに接続するよう指定したい場合は、次のように指定します。

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

第 10 章 モジュール性および生成インターフェース

Visual Modeler には次のような機能があり、実装環境のカスタマイズおよびアップグレードが簡単に実現できるよう設計されています。

- モジュール
- 生成インターフェース

これらの機能は、インターフェースがモジュールによって編成されている点と、インターフェースへの変更が個々のモジュール内の変更に含まれる可能性がある点で関連しています。

モジュールごとに機能の伝達手段を提供したり、他のモジュールへの呼び出しを外付けインターフェースを介してのみ要求することで、次のような利点があります。

- アプリケーションの機能の区分化を簡単にします。
- Visual Modeler 各部の依存関係の理解および管理を簡単にします。
- 単一モジュールへのカスタマイズを含めたり、モジュール内に加えられた変更がシステム全体にどのような影響をもたらすのか把握しやすくなります。
- アップグレードにより起こりうる影響を最小限に抑えながら、それぞれのモジュールを個別に、より簡単にアップグレードできます。
- カスタマイズされていないモジュールに対するアップグレードは、他のモジュールで行われたカスタマイズに影響しません。
- Visual Modeler の既存のデプロイメントになる可能性のあるモジュールの形で、新しい機能を伝達できます。

第 11 章 Visual Modeler のモジュール

Visual Modeler モジュール: 概要

Visual Modeler は、一般的な組織的構造に準拠する一連の相互依存型モジュールとして開発されました。通常は、モジュールはそれぞれ Visual Modeler の機能的なコンポーネント (Visual Modeler プラットフォームのアプリケーションやコンポーネントなど) に対応しています。モジュールには Java API とユーザー・インターフェースの両方をサポートしているものもあれば、その他のモジュールに提供された Java API のみをサポートしているものもあります。一部のモジュールは、"helper" クラス、JSP ページ、その他のファイル (他の多くのモジュールで使用される Javascript ファイルや画像) を提供しています。

通常は、各モジュールは次のような構造になっています。

- Java クラス: 3 つのツリー構造で編成されています。ビルド時、全モジュールのディレクトリーは com.comergent パッケージ下の単一ツリーにアセンブルされます。
  - 外部 API インターフェース: モジュールによって提供された機能にアクセスするため、他のモジュールによって使用されます。一般に、あるモジュールが別のモジュールのクラスを呼び出す場合、その他のモジュールの外部 API を使用して呼び出しを行う必要があります。これは、モジュールの com.comergent.api パッケージです。さらに、com.comergent.appservices.appServiceUtils.OFApiHelper は、Sterling Selling and Fulfillment Foundation XAPI の呼び出しに使用されます。
  - 実装クラス: 外部 API インターフェースの実装環境。別のモジュールがこのモジュールの外部 API の呼び出しを行う場合、使用される実際のクラスはこのモジュールのインターフェースの実装クラスです。実装環境のパッケージには内部クラスを含む場合があります。実装クラスで使用されますが、外部に非公開であり、サポート済み Javadoc の一部ではありません。これは、モジュールの com.comergent.apps または com.comergent.appservices パッケージです。
  - リファレンス・コンポーネント: コントローラー・クラスおよび JSP ページは常にリファレンス実装の一部を構成しており、そのソースは Visual Modeler に付属しています。また、リソース・バンドルもリファレンスの一部です。モジュールの com.comergent.reference パッケージです。
- JSP ページ: モジュールの編成によってはディレクトリーに編成されることもあります。その他のモジュールによって公開された外部 API を使用して、その他のモジュールのクラスに常にアクセスする必要があります。これにより、外部 API がサポートされていれば、JSP ページはリリース間で確実に再利用できるようになります。
- リソース・バンドル、Javascript、静的ファイル (画像および HTML フラグメントなど)。
- モジュールに特有の構成ファイル (**MessageTypes.xml** ファイルやビジネス・ルールなど)。

モジュール・インターフェース

各モジュールは、モジュール内の Java クラスやインターフェースに対するすべての呼び出しがインターフェースを介して呼び出せるよう、外部インターフェースを提供する必要があります。この外部インターフェースは、他モジュールの書き込みプログラムが外部インターフェースを確実かつ簡単に使用できるよう、広範囲にわたる Javadoc ページを提供します。

各モジュールの外部インターフェースは、通常は手作りのインターフェースと自動生成インターフェースの組み合わせになります。多くのモジュールは、プレゼンテーション Bean 用に手作りのインターフェースを提供します。これにより、プレゼンテーション Bean は、生成されたデータ Bean インターフェースの accessor メソッドより簡単にデータを操作できるようになります。通常プレゼンテーション Bean は、データ Bean を折り返し、同じインターフェースを実装しますが、さらにヘルパー・メソッドと一部のビジネス・ロジックを実装します。

外部インターフェースは、次のメイン・パッケージの下に編成されます。

- com.comergent.api: このパッケージには、すべてのモジュールの外部 API が含まれます。これらは、以下に編成されます。
  - apps: アプリケーションの手作りの API です。通常はプレゼンテーション Bean インターフェース、ユーティリティー・インターフェース、ファクトリー・クラスです。
  - appservices: 他のアプリケーションで使用されるサービス・モジュールによって提供されるパッケージです。
  - dcm: Visual Modeler プラットフォームにより提供される外部 API です。
- com.comergent.bean.simple: このパッケージには、自動生成 Bean インターフェースとデータ Bean クラスそのものがすべて含まれています。

生成インターフェースは、XML スキーマ・ファイルに宣言されたデータ・オブジェクトのそれぞれに対して提供されます。これらは、基礎となるデータ Bean のデータ・フィールドに対する適切なアクセス・レベルを備えるよう編成されます。これにより、Visual Modeler のプレゼンテーションとビジネス・ロジックとの間に明らかな区別があることを確実にするのに役立ちます。生成インターフェースについて詳しくは、81 ページの『第 12 章 生成インターフェース』 を参照してください。

インターフェースの呼び出し

オブジェクトまたは子インターフェースをインターフェースへキャストすることで Java クラスからインターフェースを呼び出すことができ、その後インターフェースが宣言するメソッドを呼び出すことができます。 Visual Modeler では、次のどちらかの方法を使用します。

- 79 ページの『オブジェクト・マネージャーの使用』
- 79 ページの『Factory クラスの使用』

各モジュールは、このどちらかの方法を使用します。両方は使用しません。既存のモジュールで作業をしたり、新規モジュールを作成する場合は、一貫した方法でインターフェースを呼び出してください。これにより、別の作業者も同じモジュールで作業しやすくなります。

通常、常に com.comergent.api パッケージで提供されるインターフェースを使用して作業するようにしてください。これらのインターフェースは、インターフェースの基礎となる実装が変更された場合でも、あるリリースから次のリリースまでプラットフォーム・モジュールによってサポートされます。

オブジェクト・マネージャーの使用

ObjectManager クラスを使用して、次のような適切なインターフェースを返すことができます。IAccProduct インターフェースを検索してプロダクトのデータ・フィールドを設定するとします。以下の行に従って呼び出しを行います。

```
IAccProduct temp_IAccProduct =

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

返されるオブジェクトを指定する **ObjectMap.xml** ファイルにエントリーがあるか、またはオブジェクトが IAccProduct インターフェースを実装しているならば、この呼び出しは成功し、インターフェースのメソッドが呼び出されます。例えば、**ObjectMap.xml** ファイルに次の行が含まれる場合、

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

com.comergent.bean.simple.ProductBean クラスは com.comergent.bean.simple.IAccProduct インターフェースを実装しているため、ObjectManager は com.comergent.bean.simple.ProductBean オブジェクトを返し、IAccProduct インターフェースへキャストできます。

Factory クラスの使用

インターフェースへの呼び出しは、インターフェースのインスタンスを返す Factory クラスにより提供できます。例えば、パッケージ com.comergent.api.apps.commerce はパブリック・インターフェース IInquiryListFactory を提供します。別のモジュールでこの Factory インターフェースのインスタンスが必要な場合は、 CommerceAPI クラスの *getFactory(int i)* メソッドを呼び出します。 int パラメーターは、どのような種類の Factory クラスが返されるのかを決定します。同様に、呼び出しモジュールは IInquiryListFactory でメソッドを呼び出し、適切な型の照会リスト・インターフェースを返すことができるようになります。例えば、*getInquiryList(Long listKey, boolean bFillPrices)* は IInquiryList インターフェースを実装するオブジェクトを返します。

第 12 章 生成インターフェース

特定のデータ・オブジェクトにあるデータにアクセスする必要がある場合、各データ・オブジェクトが提供する生成インターフェースを使用する必要があります。こうした生成インターフェースは、Visual Modeler のデプロイメントの一環として SDK generateBean ターゲットの実行時に作成およびコンパイルされます。

DsRecipes.xml ファイル内部で DataObject として宣言された各データ・オブジェクト、親、リファレンス、子データ・オブジェクトの場合は、次のクラスやインターフェースが com.comergent.bean.simple パッケージで生成されコンパイルされます。

- <*Name*>.java: データ Bean クラスです。記載のインターフェースを実装します。さらに、データ・オブジェクトを別のデータ・オブジェクトに拡張する場合、Bean により <*Parent*>.java Bean が拡張されます。
- IAcc<*Name*>.java: データ・オブジェクトのすべてのデータ・フィールドに対する書き込み (設定) の accessor メソッドを備えることで、IRd<*Name*>.java を拡張するインターフェースです。さらに、データ・オブジェクトを別のデータ・オブジェクトに拡張する場合、IAcc インターフェースにより IAcc<*Parent*>.java インターフェースが拡張されます。
- IData<*Name*>.java: データ・オブジェクトの *restore()* および *persist()* メソッドを提供することで、IAcc<*Name*>.java を拡張するインターフェースです。さらに、データ・オブジェクトを別のデータ・オブジェクトに拡張する場合、IData インターフェースにより IData<*Parent*>.java インターフェースが拡張されます。
- IRd<*Name*>.java: データ・オブジェクトのデータ・フィールドに対する読み取り専用の accessor メソッドを備えたインターフェースです。さらに、データ・オブジェクトを別のデータ・オブジェクトに拡張する場合、IRd インターフェースにより IRd<*Parent*>.java インターフェースが拡張されます。
- また、リスト Bean も IData<*Name*>List.java インターフェースを実装します。各リスト・インターフェースにより、IDataList.java インターフェースおよび親オブジェクトのリスト・インターフェースが拡張されます。

通常は、オブジェクトを効果的に読み取り専用にするために JSP ページに渡されるどのオブジェクトに対しても IRd インターフェースを使用する必要があります。データ・オブジェクトを復元または持続する必要があると判明している場合は、IData インターフェースを実装するオブジェクトのみを使用してください。

生成インターフェースの例

ACL.xml ファイルを例に、ACL データ・オブジェクトについて検証します。

```
<?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>
```

その結果、IRdACL.java クラスは次を宣言します。

```
public interface IRdACL extends IRdC3PrimaryRW
```

次のメソッドが公開されます。

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

IAccACL.java クラスは次を宣言します。

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

次のメソッドが公開されます。

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

IDataACL.java クラスは次を宣言します。

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

通常このインターフェースは、IData インターフェースに宣言された以上のメソッドを宣言することはありません。外部データ・ソースから読み取りおよび書き込みするすべての標準的なメソッドがこのインターフェース内に宣言されているためです。

第 13 章 Visual Modeler のロジック・クラス

ロジック・クラスの実装

本トピックおよび以降の 2 つのトピックでは、Visual Modeler 実装環境でのビジネス・ロジック・クラス (BLC) の実装方法について説明します。本トピックをご覧になる前に、 Visual Modeler および Java の基本的なアーキテクチャーの動作理解が必要です。

注: BLC の使用は推奨されません。通常は、新規アプリケーションには bizlet、コントローラー、BizAPI を使用してビジネス・ロジックを実装することをお薦めします。

ロジック・クラスの主要概念

アプリケーションとして Visual Modeler がどのように動作するのか十分理解するには、そのアーキテクチャーについて理解することが必要です。

Visual Modeler のインストール済み環境では、ユーザーのブラウザーから受け取った要求、およびその他の Visual Modelers や外部システムからのメッセージを処理します。各種の要求やメッセージを処理するために Visual Modeler を設定する必要があります。

Visual Modeler の中核を担うのは、Manager です。このパワフルかつフレキシブルなサーバーは、各パートナーの e-commerce 環境を構成するチャネル・パートナーや外部システムのネットワークをシームレスに統合するよう設計されています。

セールス・パートナーのネットワークにある各 Visual Modeler サーバーは、ブラウザーからのインバウンド要求に関連するサーバーとしても、およびその他の Visual Modeler サーバーや外部システムからの情報を取得するクライアントとしても機能しています。

お使いの環境にある Visual Modeler をカスタマイズするには、どのように外部システムからデータを取得するのかを考慮する必要があります。通常は、スキーマやサービス・クラスを使用して、ローカル・データベース・ソースからデータを取得したり、メッセージの送受信により別の Visual Modeler サーバーからデータを取得することができます。ただし、これら以外の外部システムから情報を取得するためには、カスタム BLC を作成する必要があります。

アプリケーション・ロジック・クラス

アプリケーション・ロジック・クラスは bizAPI、ビジネス・ロジック、コントローラー・クラスとして実装されています。

- bizAPI クラスは、ビジネス・オブジェクトのビジネス・ロジックの管理に使用されます。概念上、bizAPI クラスはそれぞれビジネス・オブジェクトに対応しています。また、そのメソッドはビジネス・オブジェクトで実行されるアクションに対応しています。例えば、OrderInquiryList bizAPI クラスは次のメソッドを提供し

ます: *duplicate()*, *copyLineItem()*, and *changeOwner()*。これは、プロダクト照会リストで実行されるアクションに対応しています。また、com.comergent.api.apps.orderMgmt.oil.IOrderInquiryList インターフェースを実装しています。

bizAPI クラスは、 com.comergent.apps.<*application*>.bizAPI パッケージに定義されています。通常は、対応する com.comergent.api.apps.<*application*> パッケージに宣言されたインターフェースを実装します。

例えば、Order bizAPI クラスは com.comergent.apps.orderMgmt.orders.bizAPI パッケージ内にあります。OrderInquiryList クラスを拡張し、com.comergent.api.apps.orderMgmt.orders.IOrder インターフェースを実装します。

- BLC はそれぞれ BLC 抽象クラスのサブクラスです。このクラスは、ApplicationObject インターフェースを実装します。BLC は、お使いの Visual Modeler の実装環境のビジネス・ロジックを実行します。各 BLC には、ビジネス・オブジェクトのテーブル (例: セッション、ユーザー、ショッピング・カートなど) が含まれています。BLC の *service()* メソッド実行時、こうしたビジネス・オブジェクトの *persist()* メソッドおよび *restore()* メソッドが呼び出されます。

  注: 一般に、BLC クラスの使用は推奨されません。ビジネス・ロジックの管理には、コントローラーまたは bizAPI クラスの使用をお薦めします。
- 一部の Visual Modeler では、ビジネス・ロジックの実行にコントローラー・クラスを採用しています。このクラスは、各アプリケーションの com.comergent.reference.apps.<*application*>.controller パッケージ内にあります。

Visual Modeler には、標準的な bizAPI クラス、BLC、コントローラー、JSP ページの多くが搭載されています。しかし、新たにロジック・クラスを作成したり、既存のクラスを修正する必要があるかもしれません。

XML スキーマ

スキーマやサービス・クラスを使用して、データ・アクセスを管理する必要があります。

ネーミング・サービス

実行時にパラメーターを検索するため、Visual Modeler はネーミング・サービスを提供してフラット・ファイルまたはデータベースのどちらかにアクセスしたり、パラメーターを復旧します。

アプリケーション・ロジック・クラスは、静的クラスの NamingManager メソッド *getInstance()* および *getInstance(int i)* を呼び出すことで、ネーミング・サービスを呼び出すことができます。どちらのメソッドも NamingService インターフェースを実装するオブジェクトを返します。

- 整数引数がない場合、デフォルト・タイプのオブジェクト (NamingServiceProperties オブジェクトまたは NamingServiceDatabase オブジェクトのいずれか) が作成されます。

- 整数引数が定数 NamingManager.DATABASE の場合は、NamingServiceDatabase オブジェクトが作成されます。
- 整数引数が定数 NamingManager.PROPERTIES の場合は、NamingServiceProperties オブジェクトが作成されます。
- 整数引数が上記のどちらでもない場合は、デフォルト・タイプのオブジェクトが作成されます。

どの場合も、Visual Modeler は **Comergent.xml** ファイルにアクセスし、NamingService オブジェクトがどのように作成される必要があるかを厳密に決定します。

- NamingServiceDatabase オブジェクトが作成される必要がある場合は、NamingManager.database エントリーを使用してデータベースへの接続を指定します。
- NamingServiceProperties オブジェクトが作成される必要がある場合は、NamingManager.properties エントリーを使用してパラメーター値を保持するプロパティー・ファイルを特定します。

一度 NamingService オブジェクトが作成されると、下記のメソッドを使用して NamingResult クラスとしてパラメーターを検索します。

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

キー・パラメーターには、キー・ストリングで始まる名前を持つパラメーターのみを検索する手段があります。

NamingResult クラスには、パラメーターの値を返すための *get(String parameter)* メソッドがあります。

NamingService の例

例えば、次のコード・フラグメントでは、パートナー・キーによって参照されるディストリビューター用にメッセージ URL パラメーターの値を回復します。

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

注: デフォルトでは、**Comergent.xml** の NamingManager defaultType 要素は "database" に設定されているため、作成された NamingService のタイプは NamingServiceDatabase オブジェクトになります。

第 14 章 Visual Modeler Software Development Kit

Visual Modeler の実装をカスタマイズするための Software Development Kit の使用

Visual Modeler の Software Development Kit (SDK) を使用して、Visual Modeler の実装およびカスタマイズができます。SDK の各バージョンに付属する HTML ドキュメントでは、SDK の動作方法の概要およびプロジェクト管理を目的とした使用方法を説明します。このトピックでは、カスタマイズするプロジェクトの基本的な構造を説明します。プロジェクトを、カスタマイズのガイドラインに沿うように編成するために、このガイドラインを遵守します。

プロジェクト編成

SDK を使って構築される各プロジェクトは、Visual Modeler のリリースをベースに作成されます。 newproject ターゲットを使用してプロジェクトを作成する場合、SDK ではリリースに適するプロジェクト・ファイルのセットを作成します。プロジェクトで行うすべてのカスタマイズは、ファイルをプロジェクトに追加することにより行います。ファイルは、以下の方法でプロジェクトに追加できます。

- customize ターゲットを使用して、ファイルをリリースからプロジェクトにコピーします。customize ターゲットを使用する場合、ファイルはプロジェクトの適切なサブディレクトリーに自動的にコピーされます。
- ファイルをプロジェクトの適切なサブディレクトリーに手動で作成します。

ファイルの配置が必要な場所については、『プロジェクト・ファイルおよびディレクトリーの場所』を参照してください。

プロジェクト・ファイルおよびディレクトリーの場所

このセクションでは、*project* という名のプロジェクトが作成されること、および ***sdk_home*/projects/*project*/** という名のプロジェクト・ディレクトリーがあることを前提にしています。以下に示すとおり、プロジェクト・ファイルのそれぞれが、プロジェクト・ディレクトリー配下の適切な場所に確実にあるようにします。

- Java ソース・ファイル: ***project*/src/** ディレクトリーの配下に配置される必要があり、 Visual Modeler のパッケージ編成に従います。
- JSP ページ: モジュールおよびロケールにより、 ***project*/WEB-INF/web/** ディレクトリーの配下で編成されます。
- スキーマ・ファイル: このファイルは、データ・オブジェクト・ファイルおよびサポートするデータ・サービス・ファイルで構成されます。すべてのカスタマイズは、 ***project*/WEB-INF/schema/custom/** ディレクトリー配下で保持される必要があります。schemaRepositoryExtn 要素は、必ず WEB-INF/schema/custom に設定されるようにします。

Java ソース・ファイル

***project*/src/** ディレクトリーで、以下のガイドラインに従って、カスタマイズを Visual Modeler に編成します。

- com/comergent/api/ パッケージを使用して、拡張機能を Visual Modeler API に追加します。一般的に、既存 API を継承する新規クラスの作成が必要です。リリースの API は、上書きするとアップグレードに支障が出るおそれがあるため、上書きしないでください。
- com/comergent/apps/ パッケージおよび com/comergent/appservices/ パッケージを使用して、実装クラスを追加します。これらのクラスは、全面的な新規クラス、または既存の実装クラスを継承する新規クラスのいずれも可能です。
- com/comergent/reference/ パッケージを制御クラスに使用します。既存の制御クラスをカスタマイズするか、新規制御クラスを作成できます。

JSP ページ

***project*/WEB-INF/web/** ディレクトリーで、以下のガイドラインに従って、カスタマイズを Visual Modeler に編成します。

- 適切な場合、JSP ページの既存の編成を使用して、新規 JSP ページを追加するか、既存 JSP ページをカスタマイズします。
- 新規機能モジュールを追加する場合、モジュールに関する適切なロケール (複数の場合あり) 配下に新規ディレクトリーを作成してから、モジュールに対して作成された Java クラスと同じ命名規約に従います。

スキーマ・ファイル

***project*/WEB-INF/schema/custom/** ディレクトリーで、以下のガイドラインに従って、カスタマイズを Visual Modeler に編成します。

- 新規データ・オブジェクトを追加するには、以下のようにします。
  - データ・オブジェクトの XML 定義を、***project*/WEB-INF/schema/custom/** に配置します。例えば、***project*/WEB-INF/schema/custom/CustComment.xml ファイルを作成します。**
  - 新規ビジネス・オブジェクトを追加することにより、***project*/WEB-INF/schema/custom/DsBusinessObjects.xml** を変更します。例えば、以下のようにします。

```
<?xml version="1.0"?>
<Schema Name="project" Description="project Custom Schema"
Version="6.0">
<BusinessObject Name="CustComment" Version="6.0"
Description="CustComment BusinessObject"/>
</Schema>
```

  - データ・オブジェクトにより宣言された新規フィールドとともに、ヘッダーおよびリストのデータ・オブジェクトに関する新規データ要素を追加することにより、***project*/WEB-INF/schema/custom/DsDataElements.xml** を変更します。例えば、以下のようにします。

```
<?xml version="1.0"?>
<Schema Name="project" Description="project Custom Schema"
Version="6.0">
```

```
<DataElement Name="CustComment" Description="Customer Comment data
object"
DataType="HEADER"/>
<DataElement Name="CustCommentList" Description="Customer Comment list
data
object" DataType="HEADER"/>
<DataElement Name="CustCommentKey" Description="Customer Comment Key"
DataType="LONG" MaxLength="20"/>
</Schema>
```

– レシピ要素を追加することにより、***project*/WEB-INF/schema/custom/DsRecipes.xml** を変更します。例えば、以下のようにします。

```
<Schema Name="project" Description="project Custom Schema"
Version="6.0">
<Recipe Name="CustComment" BusinessObject="CustComment"
Description="Default Approvals List Recipe" Version="6.0">
        <DataObjectList>

<DataObject Name="CustComment" Access="RWID"
DataSourceName="ENTERPRISE" Ordinality="n"
Version="6.0"/>
</DataObjectList>
</Recipe>
</Schema>
```

– 適切な鍵生成プログラム・ファイルを変更します。例えば、以下のように project/WEB-INF/schema/custom/OracleKeyGenerators.xml に必要な新規の鍵を追加します。

```
<?xml version="1.0"?>

<Schema Description="project Custom Schema" Name="project"
Version="6.0">

<KeyGenerator Name="CustCommentKey" KeyProcedureName="CUSTCOMMENTKEY"
GeneratorType="PROCEDURE" />
</Schema>
```

第 15 章 Visual Modeler のローカライズ

Visual Modeler のローカライズの概要

Visual Modeler では、以下に関するサポートが組み込まれています。

- 複数通貨
- 複数言語
- 数値形式および日付形式
- 文字セット

特定の市場に関して、以下のようなローカライズのその他の要素も管理できます。

- 現地の法律および規制
- 通貨処理
- 出入荷情報
- 税金

国際化対応のサポートは、ロケールを使用して管理されます。それぞれのロケールでは、言語および国または地域を識別します。ユーザーに対する情報表示に使用されるロケールを指定することにより、ロケール固有の情報およびユーザーが想定する表示形式で表示される情報を、ユーザーが確実に見られるようにします。

ユーザーが Visual Modeler にログインすると、ロケールがセッションに割り当てられます。このロケールは、ユーザーのプロファイルに指定された設定済みロケールです。ユーザーは設定済みのロケールをユーザー・プロファイルで変更できます。変更は、次回のログイン時から反映されます。ユーザー管理者は、ユーザーのプロファイルのその他の要素を変更できるように、ユーザーの設定済みロケールを変更できます。

システムのデフォルト・ロケールは、defaultSystemLocale 要素を使用する **Internationalization.xml** 構成ファイルに指定されます。各言語について、デフォルト・ロケールを指定できます。詳しくは、95 ページの『フェイルオーバーの動作』を参照してください。

Visual Modeler は、データの入力および表示に関してユニコードを完全にサポートします。

多数のローカライズを、 Java ResourceBundle を使用して行うことができます。詳しくは、101 ページの『リソース・バンドルおよび形式』を参照してください。

ロケールのサポート

Visual Modeler を実装して en_US 以外のロケールのサポートを計画する場合、ローカルの言語およびロケール固有の情報 (事業所の場所など) を反映するためのページを作成する必要があります。

プレゼンテーション・ロケールおよびセッション・ロケール

ユーザーが Visual Modeler にログインする場合、ユーザー・プロファイルに定義された設定済みロケールは、認証プロセスにより取得されます。システムでは、論理的に異なる以下の 2 つのロケールを使用します。

- セッション・ロケール: このロケールで、データ・オブジェクトに関して知識ベースから取得されるデータが決定されます。
- プレゼンテーション・ロケール: このロケールで、ユーザーに対して HTML ページを表示するのに使用される JSP ページおよびリソース・バンドルが決定されます。

一般的に、プレゼンテーション・ロケールとしてサポートするロケール・セットは、使用可能なセッション・ロケールのサブセットである必要があります。例えば、fr_CA、fr_CH、および fr_FR をセッション・ロケールとして保持しても、fr_FR および fr_CA のみをプレゼンテーション・ロケールとしてサポートすることを選択します。

ユーザーが最初にログインすると、システムではユーザー・セッションに関するプレゼンテーション・ロケールを以下のように計算します。

1. ユーザーの設定済みロケールが Visual Modeler の **web.xml** ファイルで宣言されている場合、このロケールをプレゼンテーション・ロケールに設定します。
2. 宣言されていない場合、 **Internationalization.xml** ファイルを参照します。useCountryDefaulting 要素が「TRUE」に設定されている場合、ユーザーの設定済みロケールの言語をデフォルトの国または地域ロケールに指定します。デフォルトの国または地域ロケールが **web.xml** ファイルで宣言されているかどうかを確認します。宣言されている場合、デフォルトの国ロケールをプレゼンテーション・ロケールに設定します。
3. useCountryDefaulting 要素に「FALSE」が設定されている場合、またはデフォルトの国または地域ロケールが **web.xml** ファイルにない場合のいずれかで、かつ、 useGeneralDefaulting 要素に「TRUE」が設定されている場合、defaultSystemLocale 要素で指定されたデフォルトのシステム・ロケールにユーザーのプレゼンテーション・ロケールを設定します。
4. デフォルトの要素に「FALSE」が設定されているか、 **web.xml** ファイルに宣言されたロケールがない場合、プレゼンテーション・ロケールにセッション・ロケールが設定されます。

このプレゼンテーション・ロケールは、ユーザーが Visual Modeler 内を遷移する際のユーザー・エクスペリエンスを決定するのに使用されます。それには、ユーザーが参照する Web ページを表示するのに使用される JSP ページおよびプロパティー・ファイルを制御します。それと同時に、設定済みロケールはセッション・ロケールとしても設定されます。このセッション・ロケールは、ローカライズされたデータ・オブジェクトがユーザーに表示される際に、データベースからどのデータを取得するかを判断するために使用されます。

注: データベース内に作成するすべてのロケールで、対応する項目のセットが **web.xml** ファイルにあるか、デフォルトの国または地域ロケールの項目が **web.xml**

ファイルにあるようにするとともに、国または地域のデフォルトを必ず有効にする必要があります。このようにしていない場合、一部のユーザーはシステムにアクセスできないおそれがあります。

JSP ページおよびプロパティー・ファイル

1. 各 JSP ページには、システムのデフォルト・ロケール・ディレクトリー配下の適切なモジュール・サブディレクトリーに、少なくとも 1 つの JSP ページが必要です。最初に Visual Modeler をインストールする場合、デフォルトのシステム・ロケールには en_US が設定されます。したがって、JSP ページのフルセットが ***debs_home*/SterlingWEB-INF/web/en/US/** 配下で提供されます。デフォルトのシステム・ロケールを変更する場合は、新規ロケールの対応するディレクトリーに完全にデータ設定するように注意します。
2. 各ページに表示されるすべてのテキストは、 Comergent タグ・ライブラリーのテキスト・タグまたは対応する *cmgtText()* メソッドを使用して宣言されます。例えば、以下のとおりです。

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

   または

   ```
   String title = cmgtText("cmgt_commerce/search/AdvancedSearchBody_2",
   "Inquiry Lists Search");
   ```

   バンドル属性は、クラス・ツリーの com.comergent.reference.jsp パッケージ内のファイルに対応する必要があります。上記の例では、**ChannelCartDisplayDataResource.properties** という名のファイルが ***debs_home*/Sterling/WEB-INF/classes/com/comergent/reference/jsp/channelMgmt/channelCartDisplay/** ディレクトリーにある必要があります。 ID 属性は、プロパティー・ファイル内で固有にする必要があります。上記の例では、次の形式の行が必要です。

   ```
   cmgt_channelMgmt/channelCartDisplay/ChannelCartDisplayData_7=Build Product List
   ```

3. サポート対象の各追加ロケール (仮に、*la_CO*) に関して、以下のディレクトリーを **debs_home/Sterling/WEB-INF/web/en/US/** から ***debs_home*/Sterling/WEB-INF/web/*la*/*CO*/** にコピーする必要があります。
   - **cic/**
   - **common/**
   - **home/**
4. サポート対象の各追加ロケール (仮に、*la_CO*) および各 JSP ページに関して、以下を行う必要があります。
   a. ロケールに対して新規 JSP ページを作成して、 Web アプリケーション内の対応するディレクトリーの場所に配置します。ディレクトリーの場所は、***debs_home*/Sterling/WEB-INF/web/*la*/*CO*/** 配下です。同じページが同じ言語内の複数のロケールで使用できる場合 (例えば、fr_FR および fr_CA)、そのページを必ず該当の言語のデフォルト・ロケールに配置します。言語のデフォルト・ロケールについて詳しくは、 95 ページの『フェイルオーバーの動作』を参照してください。

b. あるいは、各 ID に関して適切なテキストを含むプロパティー・ファイルを用意します。プロパティー・ファイルは、各 JSP ページおよび JSP フラグメントに対して 1 つになるように編成されます。

<、>、'、などの HTML 文字および Javascript 文字などは、プロパティーの値に含んではなりません。これらの文字は、エスケープ文字に対する HTML または Javascript の仕組みを使用して回避する必要があります。例えば、HTML では "<" を "<" の代わりに、Javascript では "\'" を "'" の代わりに使用します。

プロパティー・ファイルは、リソース・バンドルに使用されるプロパティー・ファイル向けの Java 標準に従う必要があります。特に、次のような命名規約 (**<*Name of JSP page*>Resources_*la_CO*.properties**) に従う必要があります。プロパティー・ファイルはテキスト・ファイルにする必要があり、さらにファイル内の各行は次の形式にする必要があります。

```
cmgt_module/package/JSPname_n=Display text for this locale
```

例えば、以下のとおりです。

```
cmgt_channelMgmt/channelCartDisplay/
ChannelCartDisplayData_7=Build Product List
```

プロパティー・ファイルは、すべて ***debs_home*/Sterling/WEB-INF/classes/com/comergent/reference/jsp/** ディレクトリーにあり、モジュール JSP ページがモジュール内で編成されるのと同じ方法で、このディレクトリー内のモジュールにより編成されます。これらのリソース・バンドルの場所を変更する場合、新しい場所からリソース・バンドルを取得するためにテキスト・タグのカスタマイズが必要な点に注意してください。

テキストを JSP ページに追加する場合、既存のタグ ID に対してテキストを修正するか新規 ID を追加することにより、対応するロケール JSP ページまたはプロパティー・ファイルを注意しながら更新します。

以下の点に注意します。

- 変換されたテキストの長さは大幅に異なります。したがって、Web ページのレイアウトに影響する可能性があります。
- 変換が Visual Modeler のロジックに影響する場合、ドロップダウン・リストおよび Javascript 機能ではテキストを使用できます。 99 ページの『Javascript』および 88 ページの『JSP ページ』を参照してください。
- 地域の規則が情報の表示 (ユーロおよび現地通貨での価格表示など) に影響を及ぼすことがあります。
- ページの論理的なフローが、ローカルでの慣習 (輸出に関する注意の表示または税金の情報の表示など) を反映するために変更が必要な場合、特に注意してください。

Internationalization.xml 構成ファイルの debugJSPResouceBundle 要素を使用して、ストリングの欠落を識別をしやすくできます。この要素に「TRUE」を設定すると、参照されたリソース・バンドルからストリングが欠落している場合、エラー・メッセージがブラウザー・ページに表示されます。実動システムでは、この要素の値に「FALSE」の設定が必要です。

フェイルオーバーの動作

このセクションでは、ユーザーの現在のプレゼンテーション・ロケールにリソース (JSP ページまたはプロパティー) が定義されていない場合に発生する内容を説明します。フェイルオーバーの動作は、以下の点で JSP ページおよびリソース・バンドルとはわずかに異なる点に注意してください。

- JSP ページでは、特定のロケールから言語ロケールのデフォルトの国または地域にフェイルオーバーし、さらにシステムのデフォルト・ロケールにフェイルオーバーできます。例えば、fr_CA から fr_FR を経由して en_US などです。
- リソース・バンドルでは、Java の指定 (***_fr_CA.properties** to ***_fr.properties** to ***.properties**) に従ってフェイルオーバーします。

JSP ページのフェイルオーバーの動作を管理するために、**Internationalization.xml** 構成ファイル内の以下の 2 つのプロパティーが使用されます。

- useCountryDefaulting: これに「TRUE」が設定されていれば、プレゼンテーション・ロケールにリソースがない場合、適切な言語要素に指定された国または地域をデフォルトにします。
- useGeneralDefaulting: これに「TRUE」が設定されていれば、プレゼンテーション・ロケールで使用可能なリソースがない場合、システム・ロケールをデフォルトにします。

フェイルオーバーの動作 (リソース・バンドル)

すべてのテキスト・ストリングをそれぞれのロケールに変換する必要はありません。テキスト・ストリングがリソース・バンドル・プロパティー・ファイル内の特定 ID に指定されていない場合は、標準の Java フェイルオーバー・プロセスに従って処理されます。例えば、cmgt_channelMgmt/channelCartDisplay/ChannelCartDisplayData_7 ストリングが存在するものの **ChannelCartDisplayDataResource_fr_CA.properties** に定義がない場合は、**ChannelCartDisplayDataResource_fr.properties** ファイルが参照されます。このファイルが存在しない、またはこのファイルに該当する ID の項目がない場合、**ChannelCartDisplayDataResource.properties** ファイルが参照されます。

フェイルオーバーの動作 (JSP ページ)

すべての JSP ページがサポート対象のすべてのロケールに対して使用できる必要はありません。例えば、すべてのページに対して en_US ページを使用しても、一部のページは en_CA ユーザーから参照されるように選択することができます。このセクションでは、メッセージ・タイプが処理される際の内容を以下で説明します。

要求は、適切な **MessageTypes.xml** 内のメッセージ・タイプの JSPMapping 要素で指定された JSP ページに転送されます。

1. JSP ページが現在のロケールに存在する場合、このページが Web ページの生成に使用されます。
2. JSP ページが現在のロケールに存在しない場合、フェイルオーバーの仕組みでは、現在のロケールの言語に対するデフォルト・ロケールを識別します。これは、言語に対する defaultCountry 要素として、**Internationalization.xml** 構成ファイル内で宣言されます。

3. JSP ページが言語のデフォルト・ロケールに存在する場合、このページが Web ページの生成に使用されます。例えば、 **Internationalization.xml** 内の後続の要素で US が en 言語ロケールのデフォルトの国または地域として指定されており、JSP ページが en_CA ロケールにない場合、対応する en_US JSP ページが使用されます。
4. <en visible="false">
   <defaultCountry ...>US</defaultCountry>
   </en>
5. デフォルトの国または地域の JSP ページが存在しない場合、フェイルオーバーの仕組みでは、デフォルトのシステム・ロケールを識別します。これは、**Internationalization.xml** ファイルの defaultSystemLocale 要素の値として宣言されます。 JSP ページがシステムのデフォルト・ロケールに存在する場合、このページが Web ページの生成に使用されます。
6. 最終的に、JSP ページがデフォルトのシステム・ロケールに存在しない場合、例外がスローされてエラー・ページが表示されます。

ロケール取得メソッド

大半の場合、Visual Modeler の組み込みサポートを使用し、ユーザーのロケールに適するコンテンツでユーザーに表示できます。手動でロケールにアクセスする必要がある場合、ComergentI18N クラスを使用できます。このクラスでは以下のメソッドを提供します。

- *getDefaultLocale()*: システムのデフォルト・ロケールを戻します。
- *getComergentLocale(boolean b)*: b が「TRUE」の場合、ユーザーのプレゼンテーション・ロケールを戻し、「TRUE」以外の場合、ユーザーのセッション・ロケールを戻します。
- *findPresentationLocale(Locale sessionLocale)*: 任意のセッション・ロケールに対して使用されるべきプレゼンテーション・ロケールを求めるのに使用されます。

コードでのプロパティー・ファイルの使用

プロパティー・ファイルを Java コード内でも使用できます。例えば、**com.comergent.reference.jsp.AdvisorBodyResources.properties** ファイル内で定義されたストリング keyString に対応するロケール固有のストリングを取得するには、以下のように使用します。

```
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");
```

ここでは、ユーザーのセッションに保管された、ユーザーの現在のロケールを使用します。別のロケールを強制的に使用する場合、以下のように使用します。

```
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");
```

国際化対応のデータ

企業ユーザーおよびエンド・ユーザーがデータをマルチバイト文字で入力することを想定する場合、データ・フィールドの長さおよびそれに対応するデータベース表のカラムを考慮する必要があります。当社の経験では、マルチバイト文字を使用して Visual Modeler に入力されたデータは、en_US ロケールで使用されるストリングと比べ、データベース内で最大で 3 倍の長さになることがあります。したがって、マルチバイト文字でのデータ入力が想定されるフィールド (特に名前フィールドと説明フィールド) の長さを検討する必要があります。

フィールドの長さを変更する場合、 **DsDataElements.xml** 構成ファイル内の両方のフィールドの変更が必要であること、および、知識ベース・スキーマの生成に使用される SQL スクリプトにも対応する変更が必要であることを覚えておいてください。

例

製品データ・オブジェクトの説明フィールドの長さをマルチバイト文字に最適な長さにするには、以下の手順を実行する必要があります。

1. 製品説明の保持に使用されるデータ・フィールドを識別します。製品データ・オブジェクトは、ローカライズ可能なデータ・オブジェクト (Localized="y") であるため、これが ProductLocale データ・オブジェクトの説明フィールドです。製品データ・オブジェクトに対応するデータベース表およびカラムは CMGT_PRODUCT_LOCALE.DESCRIPTION です。

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

2. 説明に対して、以下のように最大で 240 文字の長さを許可すると想定します。

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

3. CMGT_PRODUCT_LOCALE 表の作成に対応する SQL ステートメントを変更して、次のように DESCRIPTION カラムが VARCHAR2(720) に設定されるようにします。

   ```
   DESCRIPTION VARCHAR2(720) DEFAULT 'Not available',
   ```

4. 適切な SDK ターゲット (merge および createDB) を実行して、Visual Modeler の実装に対して変更します。

この例で、説明データ・フィールドは多くのさまざまなデータ・オブジェクトにより広範囲に使用されるため、 **DsDataElements.xml** 構成ファイル内でこのフィールドの定義を変更すると、他の個所で予期せぬ影響を与えるおそれがあります。代わりの方法は、 ProductDescription の名前で新規データ・フィールドを作成して、このフィールドを ProductLocale データ・オブジェクト内で使用することです。新規データ・フィールドを、以下のように **ProductLocale.xml** ファイルに設定できます。

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

次に、以下のように **DsDataElements.xml** 構成ファイルに設定します。

```
<DataElement Name="ProductDescription" DataType="STRING"
Description="This is the product description field"
MaxLength="240" />
```

注: ユーザーが有効なデータをフィールドに入力したことを検査するために Javascript メソッドを提供する場合、フィールドの長さを検査するには、対応する DataElement に指定された長さに対して検査します。

E メール・テンプレート

ご使用のシステムが英語以外の言語をサポートしており、ユーザーに送信されるメッセージを生成するために Visual Modeler のインストール済み環境で E メール・テンプレートを使用する場合、翻訳が必要なことを覚えておいてください。

リリース 6.4 では JSP ページを使用して E メール・メッセージを生成する機能が導入されています。この機能により、JSP ページの国際化対応用の既存フレームワークを使用して E メール・メッセージの国際化対応をサポートします。

既存のアプリケーションの場合、Visual Modeler により提供されるデフォルトのテンプレートを使用できます。デフォルトのテンプレートは ***debs_home*/Sterling/WEB-INF/templates/** にあります。

HTML ページ

静的な HTML ページには翻訳が必要です (適切な場合)。複数言語を同時にサポートする場合、それぞれの言語に対してページの作成が必要な点に注意が必要です。これらのページの場所を、ロケール・ディレクトリー構造の全体で一貫して維持する場合、これらのページに対して相対参照することにより、必ず正しい HTML ページを参照できます。

例えば、以下の JSP フラグメントでは、動的に URL を生成してロケール固有の Example.html ページを指し示します。

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

この例で、リソース・バンドルは、リンクに関して表示されるテキストを判断するのに使用されます。

イメージ

一般的に、イメージは、埋め込みのテキストがないイメージを使用します。これにより、同じイメージを複数のロケールで使用することで、ローカライズおよびメンテナンスのコストを確実に低減できます。

ただし、必要な場合は、イメージのローカライズ・バージョンの提供が必要です。静的 HTML ページの場合のように、相対 URL を使用して、ロケール固有のイメージが JSP ページに相対する正しい場所から取得されるようにします。

特に海外向けのページにあるすべてのボタンはテキスト付きのイメージ・ボタンであることを忘れないようにしてください。必要な場合、各ボタンのローカライズ・バージョンの作成が必要です。イメージ・ソースの URL は以下のように生成できます。

```
<IMG ALT="Locale-specific alternate text goes here"
SRC="../images/button.gif"></A>
```

Javascript

Javascript で使用される表示テキストのローカライズには注意が必要です。例えば、アラート・ダイアログ・ボックスでは、ユーザーのロケールを表示テキストに反映する必要があります。

- 一部の Javascript ファイルは、以下の行に従って Web ページ内に記述されます。

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

  各ロケールに対するこれらの Javascript ファイルは、ブラウザーがこのファイルを、生成された Web ページ内に適切に含められるように保持する必要があります。

- Javascript が JSP ページ内または記述された JSP フラグメント内で定義される場合、表示テキストはテキスト・タグでラップされる必要があります。例えば、次のとおりです。

  ```
  alert("<cmgt:text id="*">Product ID is missing.</cmgt:text>");
  ```

  これらのタグが SDK ツールの一部として処理される場合、ID の属性は固有 ID に変更され、この ID およびタグのボディが JSP ページまたはフラグメントに関するリソース・バンドルに追加されます。

Visual Modeler のローカライズ (JSP ページ)

一般的に、すべてのローカライズ (ラベル、説明のためのテキスト、読み込みリスト、および日付と通貨に関するロケール固有形式など) は、ロケールに対して作成された JSP ページに反映される必要があります。

編成の有用な考え方は、ローカライズされたすべてのストリングのハッシュ・マップをページ上で作成し、このハッシュ・マップをその他のページ全体で参照することです。例えば、以下のとおりです。

```
HashMap localized = new HashMap();
localized.put("TaskListHeader",
cmgtText("cmgt_taskMgr/TaskWorkspaceData_3","Task List:"));
localized.put("QuickSearchTitle",
cmgtText("cmgt_taskMgr/TaskWorkspaceData_4","Search for Tasks"));
localized.put("TaskID",
cmgtText("cmgt_taskMgr/TaskWorkspaceData_5","ID"));
localized.put("TaskName",
cmgtText("cmgt_taskMgr/TaskWorkspaceData_6","Name"));
localized.put("Status",
cmgtText("cmgt_taskMgr/TaskWorkspaceData_7","Status"));
localized.put("Priority",
cmgtText("cmgt_taskMgr/TaskWorkspaceData_8","Priority"));
localized.put("CreateDate",
cmgtText("cmgt_taskMgr/TaskWorkspaceData_9","Create Date"));
request.setAttribute("localized", localized);
```

これらのストリングは、次の行に従うスクリプト機能を使用して参照できます。

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

この手法には、JSP ページがさらに読みやすくなるため、ローカライズされたストリングを容易に再使用でき、さらに JSF モデルに近くなるというメリットがあります。

この UI コンポーネントのローカライズについては、101 ページの『カレンダー・ウィジェット』を参照してください。例えば、フランス語ロケールの場合に曜日のドロップダウン・リストを設定するには以下のようにします。

```
<SELECT Name="DayOfWeek">
<OPTION VALUE=0>dimanche</OPTION>
<OPTION VALUE=1>lundi</OPTION>
<OPTION VALUE=2>mardi</OPTION>
<OPTION VALUE=3>mercredi</OPTION>
<OPTION VALUE=4>jeudi</OPTION>
<OPTION VALUE=5>juin</OPTION>
<OPTION VALUE=6>vendredi</OPTION>
<OPTION VALUE=7>samedi</OPTION>
</SELECT>
```

リソース・バンドルを使用しても、ロケール固有の表示情報を管理できます。例えば、以下はグレゴリオ・カレンダーでの曜日のドロップダウン・リストを設定する代替方式です。

```
<SELECT Name="DayOfWeek">
<OPTION VALUE=0><%= resourceBundle.getString("Sunday") %></OPTION>
<OPTION VALUE=1><%= resourceBundle.getString("Monday") %></OPTION>
```

```
<OPTION VALUE=2><%= resourceBundle.getString("Tuesday") %></OPTION>
<OPTION VALUE=3><%= resourceBundle.getString("Wednesday") %></OPTION>
<OPTION VALUE=4><%= resourceBundle.getString("Thursday") %></OPTION>
<OPTION VALUE=5><%= resourceBundle.getString("Friday") %></OPTION>
<OPTION VALUE=6><%= resourceBundle.getString("Saturday") %></OPTION>
</SELECT>
```

カレンダー・ウィジェット

カレンダー・ウィジェットを JSP ページで使用する場合、ローカライズが必要です。ローカライズは、**I18N.js** Javascript ファイルがロケール・ディレクトリー (***debs_home*/Sterling//*la*/*CO*/js/**) で検出されるようにカスタマイズすることで行います。例えば、de_DE ロケールをサポートするには、以下を読み取るファイルを ***debs_home*/Sterling/de/DE/js/I18N.js の名称で作成します。**

```
// DEFAULT LOCALE (English)
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;
// Create CalendarPopup object
var popupCal = new CalendarPopup();
```

スタイル・シート

Visual Modeler では、カスケーディング・スタイル・シートを使用して、 HTML 要素の書式を設定します。特定ロケールのフォントを使用する場合、これらのフォントを指定するスタイル・シートを必ず作成します。各ロケールに関して、このロケール固有のスタイル・シートを同じ相対位置に保存します。

JSP ページで、ロケール固有のカスケーディング・スタイル・シート (仮に **customer.css**) を以下のように記述できます。

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

システム・プロパティー

一般的に、構成ファイルのデータは管理者のみに表示されます。構成ファイルをローカライズするには、要素の名前または値を変更する必要はありませんが、要素のヘルプ・テキストの変更を検討する必要があります。各 Visual Modeler に対する構成ファイルは 1 セットしかないため、これらのファイルにはシステムのデフォルト・ロケールの言語を使用する必要がある点に注意してください。

リソース・バンドルおよび形式

PropertyResourceBundle およびプロパティー・ファイル

Visual Modeler では、プロパティー・ファイルを拡張して使用し、ロケール固有データを管理します。これは、 ResourceBundle Java クラスを使用する代わりです。

詳しくは、 91 ページの『Visual Modeler のローカライズの概要』を参照してください。

ResourceBundle

ローカライズを管理するための有用な仕組みは、Java ResourceBundle を使用することです。

注: リソース・バンドル・クラスの Visual Modeler での使用は、推奨されません。 91 ページの『Visual Modeler のローカライズの概要』に説明があるように、プロパティー・ファイルを使用する必要があります。

ロケール固有情報を管理するクラスがあります。Visual Modeler で使用されるすべての ResourceBundle クラスは、 ListResourceBundle を継承します。これらのクラスでは、 *getString (String nameString)* メソッド呼び出し時に戻される名前ストリングと値ストリングの間のマッピングを定義します。

ResourceBundle の命名規約に従い、サポートが必要なすべてのロケールに対してロケール固有の ResourceBundle を作成できます。例えば、以下の ResourceBundle を作成して、Inventory という名称の新規アプリケーションで使用できます。

- InventoryResourceBundle
- InventoryResourceBundle_fr
- InventoryResourceBundle_fr_FR
- InventoryResourceBundle_fr_CA

以下のスクリプトレットは、JSP ページで使用する場合、適切なリソース・バンドルを取得できます。

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

NumberFormat および DateFormat

NumberFormat クラスを使用して、数値をロケール固有の方法で表示しやすくします。NumberFormat のインスタンスは、ロケールをコンストラクターに引き渡すことにより作成します。

例えば、以下のスクリプトレットでは、ショッピング・カードの合計数をロケールに対して適切な形式で表示します。

```
<%
NumberFormat numberFormat =
NumberFormat.getInstance(session.getLocale());
int number = request.getParameter("ShoppingCartsTotal");
%>
```

```
<P>The number of active shopping carts in use is:
<%= numberFormat.format(number) %>
</P>
```

同様に、DateFormat クラスを使用して、日付をロケール固有の方法で表示しやすくします。DateFormat のインスタンスは、ロケールをコンストラクターに引き渡すことにより作成します。

例えば、以下のスクリプトレットでは、現在の日付をロケールに対して適切な形式で表示します。

```
<%
DateFormat dateFormat =
DateFormat.getInstance(session.getLocale());
Date todaysDate = new Date();
%>
<P>It is now:
<%= dateFormat.format(todaysDate) %>
</P>
```

第 16 章 コントロールのカスタマイズ

コントロールは、ユーザー・インターフェース (UI) でのオプション・クラスおよびオプション・アイテムの表示方法と動作方法を判断するのに使用されます。既存コントロールを変更するか、新規コントロールを追加できます。

各コントロールは、JSP ページおよびオプション・アイテムの動作に対応します。この対応付けは、デプロイメント・ディレクトリーにある Comergent/WEB-INF/properties フォルダー配下の controls.properties 構成ファイルに定義されます。

以下は、controls.properties ファイルに定義されたサンプルの項目です。

```
RADIO.name=Radio Button
RADIO.jsp=controls/radio.jsp
RADIO.behavior=single
```

この例で、ラジオ・ボタン・コントロールの場合、radio.jsp JSP ページが、UI でのオプション・クラスの表示に使用されます。behavior プロパティーでは、Sterling Configurator がこのコントロール内のピックを処理する方法を決定します。behavior プロパティーの定義方法に基づき、 Sterling Configurator ではピックを以下のように処理します。

- 入力 - ユーザー入力のコントロールに使用されます。
- 拡張 - コントロール自体がピックされた場合、このコントロールのすべての子を拡張します。
- 複数 - このコントロールから 1 つ以上のオプション・アイテムがピックされるのを許可します。
- 単一 - オプション・アイテムがピックされる場合、このオプション・クラスから以前にピックされたオプション・アイテムがあれば削除します。

コントロールの変更

このタスクについて

既存のコントロールは、controls.properties ファイル内の対応する項目を変更することによりカスタマイズできます。

既存のコントロールを変更するには、以下の手順を実行します。

手順

1. 次のターゲットを実行して、カスタマイズのために controls.properties ファイルを取得します。

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

   このターゲットを実行すると、controls.properties ファイルがカスタマイズ・プロジェクトに配置されます。
2. 必要に応じて、controls.properties ファイル内の項目を変更します。

3. 次のターゲットを実行して、カスタマイズをビルドにマージします。

   ```
   sdk merge
   ```

4. Visual Modeler アプリケーションを WAR ファイルとしてデプロイする場合、以下の手順を実行します。
   a. 次のターゲットを実行して、WAR ファイルを再作成します。

      ```
      sdk distWar
      ```

   b. .war ファイルをアプリケーション・サーバーにデプロイします。

タスクの結果

これらの手順完了後、必要な変更を Sterling Selling and Fulfillment Foundation で行う必要があります。詳しくは、「*Sterling Configurator アプリケーション・ガイド*」を参照してください。

コントロールの追加

このタスクについて

新規コントロールは、宣言されたコントロールのリストにコントロール名を追加してから、新規コントロールのプロパティーを定義することにより定義できます。

新規コントロールを追加するには、以下の手順を実行します。

手順

1. 次のターゲットを実行して、カスタマイズのために controls.properties ファイルを取得します。

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

   このターゲットを実行すると、controls.properties ファイルがカスタマイズ・プロジェクトに配置されます。

2. 新規コントロールの名前を、 controls 属性のコンマ区切りの値のリストに追加します。

   例えば、新しい ABC_CUSTOM コントロールを追加するには、controls 属性を次のように定義できます。

   ```
   controls=ABC_CUSTOM,RADIO,CHECKBOX,COMBOBOX,LISTBOX,MULTISELLISTBOX,
   ALLPICKED,UEV,DISPLAY
   ```

3. 新規コントロールのプロパティーを定義します。例えば、新しい ABC_CUSTOM コントロールのプロパティーを以下のように定義できます。

   ```
   ABC_CUSTOM.name=Matrix Custom Control
   ABC_CUSTOM.jsp=controls/ABCCustom.jsp
   ABC_CUSTOM.behavior=single
   ```

4. 次のターゲットを実行して、カスタマイズをビルドにマージします。

   ```
   sdk merge
   ```

5. Visual Modeler アプリケーションを WAR ファイルとしてデプロイする場合、以下の手順を実行します。
   a. 次のターゲットを実行して、 WAR ファイルを再作成します。

      ```
      sdk distWar
      ```

b. .war ファイルをアプリケーション・サーバーにデプロイします。

タスクの結果

これらの手順完了後、必要な変更を Sterling Selling and Fulfillment Foundation で行う必要があります。詳しくは、「*Sterling Configurator アプリケーション・ガイド*」を参照してください。

第 17 章 関数ハンドラーのカスタマイズ

関数ハンドラー・クラスは、 Sterling Configurator ルール・エンジンで呼び出し可能なカスタム関数の定義に使用される Java クラスです。関数ハンドラー・クラスは、カスタマイズ可能です。

関数ハンドラーは、デプロイメント・ディレクトリーにある Comergent/WEB-INF/properties フォルダー配下の functionHandlers.properties 構成ファイルに定義されます。このファイルには、各関数ハンドラーの名前および関数ハンドラー・クラスがあるディレクトリーが記述されています。

以下は、functionHandlers.properties ファイルのサンプル・フラグメントです。

```
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
```

関数ハンドラー・クラスの追加

このタスクについて

新規関数ハンドラー・クラスを追加できます。

新規関数ハンドラー・クラスを追加するには、以下の手順を実行します。

手順

1. 次のターゲットを実行して、カスタマイズのために functionHandlers.properties ファイルを取得します。

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

   このターゲットを実行すると、functionHandlers.properties ファイルがカスタマイズ・プロジェクトに配置されます。

2. com.comergent.apps.configurator.functionHandlers パッケージ宣言を使用して、新しい Java クラスを作成します。クラス宣言では、このクラスが AbstractRuleFunctionHandler クラスを継承することを宣言する必要があります。

   注: 新しい Java クラスは、 Visual Modeler アプリケーションのクラスパス内で提供される必要があります。

   新しい Java クラスでは、以下のメソッドを実装する必要があります。

   - public String getFuncName(): sum または max などの関数名を戻します。これは大/小文字を区別するため、sum および SUM を管理するために別の関数ハンドラーを使用できます。

- public int getType(): 関数が戻す値のタイプを戻します。タイプは、com.comergent.api.appsservices.rulesEngine.Value クラスに定義された定数であることが必要です。AbstractRuleFunctionHandler クラス・メソッドは Value.STRING を戻します。したがって、関数がその他のタイプを戻す場合、このメソッドをオーバーライドする必要があります。
- public Value handle(State state, String prop): 関数に対して計算された値を戻します。
- public boolean isPublicHandler(): 関数ハンドラーがいかなるクライアント・アプリケーションでも使用できる場合は「TRUE」を戻し、それ以外は「FALSE」を戻します。AbstractRuleFunctionHandler クラス・メソッドは、「TRUE」を戻します。したがって、関数ハンドラーが private の場合、このメソッドのオーバーライドのみ必要です。

タスクの結果

これらの手順完了後、必要な変更を Sterling Selling and Fulfillment Foundation で行う必要があります。詳しくは、「*Sterling Configurator アプリケーション・ガイド*」を参照してください。

第 18 章 例外

ComergentException 階層

例外ルート

このトピックでは、以下を説明します。

- ComergentException
- ICCException
- ComergentRuntimeException

ComergentException

実動ソフトウェアで宣言された、コンパイル時のすべての例外クラスは、根本的に com.comergent.dcm.util.ComergentException クラスを継承する必要があります。この継承元のクラスは、java.lang.Exception を継承して、チェーニングおよび独立したユーザー・メッセージを提供します。

ICCException

ICCException は、ComergentException の便利なサブクラスを提供します。サブシステムに対して例外クラスのセットを作成する代わりに、サブシステム全体で一律に ICCException クラスを作成できます。

ComergentRuntimeException

すべての実行時例外クラスは、java.lang.RuntimeException を継承して同一機能を提供する com.comergent.dcm.util.ComergentRuntimeException を継承する必要があります。

サブシステムのグループ化

Visual Modeler のサブシステムは、別個で分割可能なアプリケーション、アプリケーション・レベル・サービス、またはシステム・レベル・サービスのいずれかになるように定義されます。サブシステムは、論理的に編成されています。Java パッケージ階層内で複数のパッケージにまたがるか、パッケージの一部を構成できます。

それぞれの論理サブシステムでは、それ自体の例外ルート・クラスを宣言することが想定されます。このルートは、ComergentException を継承するとともに、サブシステム内でのコンパイル時のすべての例外の親クラスです。サブシステムは、別個で分割可能なアプリケーション、アプリケーション・レベル・サービス、またはシステム・レベル・サービスのいずれかになるように定義されます。サブシステムは、論理的に編成されています。Java パッケージ階層内で複数のパッケージにまたがるか、パッケージの一部を構成できますが、論理的なサブシステム編成に合うようにパッケージ構造を編成する必要があります。

例えば、Foo という名称のサブシステムがあるとします。以下のように、FooException クラスが必要です。

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

Foo が、不適切な初期化状態に対して BadInitializationException を後続のすべての要求にスローすることで応答するとします。この例外は、以下のように FooException から継承されます。

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

サブシステム例外ポリシー別のサブシステム

それぞれのサブシステムでは、例外を識別するために一貫性のあるポリシーを実装する必要があります。異なる例外タイプごとにサブシステムの例外クラスをサブクラスにする必要がある (これは Java スタイル・ポリシーの標準) か、またはサブシステムのルート例外が ICCException を継承して、例外を識別するために状況パラメーターを設定する必要があります (これは ICCException ポリシー)。

例えば、サブシステム Foo で Java スタイルの例外ポリシーが選択されると、FooException は ComergentException を継承する必要があります。サブシステム Bar で ICCException ポリシーが選択されると、FooException は ICCException (同様に ComergentException を継承) を継承する必要があります。

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

例外チェーン

各サブシステムでは、サブシステム固有のサブシステムからサブシステムの呼び出し元に対して例外のみスローすることが想定されています。基礎となるサービスが、任意のサブシステムで処理できない例外をスローする場合、任意のサブシステムは、該当の例外をキャッチし、それ自体のコンテキスト内で有用な例外を再度スローすることが想定されています。新しい例外では、チェーニング・コンストラク

ターを使用して元の例外を含める必要があります。これにより、最終的に例外が処理されて記録される場合、元の例外が失われません。

例えば、サブシステム Foo がプロパティー・ファイルのオープンを試行すると、IO 例外が発生するとします。サブシステム Foo が Java スタイルの例外ポリシーを実装していれば、新規の例外クラスである FooPropertyFileException を宣言できます。この例外クラスは FooException を継承します。IO 例外のキャッチ・ステートメントは、メッセージおよび元の IO 例外を渡すコンストラクターを使用して、新規の FooPropertyFileException をスローします。

```
try
{
...
Properties props = new Properties();
props.load(input);
...
}
catch (IOException ex)
{
// chain the io exception
throw new FooPropertyFileException("Loading file" + filename, ex);
}
```

例外のスロー、キャッチ、およびロギング

例外をスローするタイミング

例外は、メソッドとメソッド呼び出し元の間の契約が実行されない場合に、スローされる必要があります。これは、 Java 言語仕様に示される使用法です。残念ながら、この言語仕様ではわずかなガイダンスしか提供されないため、契約は例外を不要にするために広義に定義するか、例外が頻繁に発生するように狭義に定義できます。一般的な経験則として、例外を使用するには、以下の 2 つの相反する目的の間でバランスを取る必要があります。

例外を標準にしてはなりません。

- 例外は追加オブジェクトの作成に関連します。このため、パフォーマンスの観点のみからすると、例外が頻繁に発生する可能性がある場合は問題です。
- データおよびコントロールの混合は避ける必要があります。例外スローの代替策は、メソッドからヌル値を戻す場合が多くあります。これは、戻り値には 2 つの意味 (成否、およびデータ表示内容) が包含されていることを表します。実際のプログラミングでは、この使用方法を避けるのが賢明です (可能な場合)。
- メソッドの示す目的に対してヌルが適切な値である場合、または通常動作で頻繁に失敗することがメソッドに想定される場合、失敗を示すためにヌルを戻すのは適切です。それ以外は、例外をスローするのが適切です。

実行時例外またはコンパイル時例外のスロー

Java 言語仕様に従うと、呼び出し元が誤った入力を渡した場合、実行時例外がスローされる必要があり (基本的にメソッド契約破棄)、コンパイル時例外を宣言するの

は負荷が高いとされます。例えば、配列添字のパラメーターに対して負の値を渡すメソッドを呼び出し元が呼び出す場合、実行時例外をスローするのは適切です。それ以外の場合、コンパイル時例外をスローします。

キャッチ節およびスローの宣言

キャッチ節およびスローの宣言は、過度に汎用的にしないようにする必要があります。呼び出されたメソッドが、例えば FileNotFoundException をスローすると、呼び出し元は (Exception でも Throwable でもなく) FileNotFoundException をキャッチする必要があります。この理由は、基礎となるコードが新しい例外をスローするように変更される場合、またはこの例外のスローを停止される場合、変更によりコンパイル・エラーが発生してプログラマーに新しい状況を検討するように知らせることが望ましいためです。

このルールに対する例外があり、実際に広く普及しています。スロー可能なさまざまな例外のデータが大きく、すべての場合に応答が同じ場合、個別にキャッチする必要はありません。

例外のロギング

メソッドが例外をキャッチして処理する (すなわち、例外を再スローしない) 場合、例外を記録する必要があります。このメソッドでは例外の重要度を認識し、エラーの重大度またはその他の重大度の低いレベルとともに、例外を記録するかどうか認識していると考えられます。しかし、Empty キャッチ・ステートメントは、疑いをもって見る必要があります。

以下のようにしてはなりません。

```
catch (SomeException ex)
{
}
```

以下のようにします。

```
catch (SomeException ex)
{
Global.logVerbose(ex);
}
```

または、以下のようにします。

```
catch (SomeException ex)
{
ex.printStackTrace(Global.debugStream);
}
```

基礎となるサブシステムまたはサード・パーティー・パッケージからの例外がキャッチされて、新しい例外にチェーニングされた場合、新しい例外は記録する必要がありません。階層上位の一部のプロセスは、最終的に例外をキャッチして処理するとともに例外の記録方法を認識しています。

例外の表示

一般的に、 Visual Modeler のユーザーが例外を目にすることはありません。適切なサブシステムが、エラー状態に対して適正に応答することで、必ず例外を迅速に処理するためです。

Visual Modeler のエラー・ページでは、例外スタック・トレースを HTML コメントの間に配置します。表示された Web ページのソースを見ることにより、スタック・トレースを読むことができます。

例外スタック・トレースが JSP ページに渡される場合、JSP ページのバッファー制限により Web ページにすべての例外メッセージが渡されない場合があることを覚えておいてください。長い例外スタック・トレースが JSP ページに渡される場合、ユーザーは JSP ページのバッファーを変更することにより見ることができます。バッファー・タグを次のように使用します。

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

このタグはパフォーマンスに影響があるため、エラー状態を問題判別して修正が完了すると、削除する必要があります。

第 19 章 クーロン・ジョブ

Visual Modeler クーロン・ジョブの実装

Visual Modeler の実装のうち、特定のタスクはユーザー入力に対して開始されるわけではありません。例えば、オーダー・データと外部システムとの 1 時間ごとの同期、またはカタログ・データのサード・パーティーからの 1 週間ごとのインポートは、ユーザーの介入なしで適切に実行されます。これらのジョブは、Visual Modeler が提供するジョブ・スケジューラー機能を使用して、適切な間隔での実行をスケジュールできます。

クーロン・ジョブは、システム・クーロン・ジョブまたはアプリケーション・クーロン・ジョブのいずれかとして定義できます。

- システム・クーロン・ジョブは Visual Modeler により実行され、ユーザーには一切関連付けられません。システム・クーロン・ジョブは、Visual Modeler クラスを直接呼び出します。システム・クーロン・ジョブは、SystemCron 抽象クラスを継承するクラスにより実行される必要があります。通常、システム・クーロン・ジョブは、キャッシュの削除などのタスクを実行します。
- 各アプリケーション・クーロン・ジョブは、ユーザーとして実行されます。ユーザーのユーザー名およびパスワードは、ジョブ・スケジューラーのユーザー・インターフェースを使用してクーロン・ジョブが作成される際に、入力されます。アプリケーション・クーロン・ジョブは、XML メッセージを Visual Modeler にポストすることにより動作し、システムにより処理されます。アプリケーション・クーロン・ジョブは、ApplicationCron 抽象クラスを継承するクラスにより実行される必要があります。通常、アプリケーション・クーロン・ジョブは、ユーザーまたは製品データに関する必要な管理タスク (オーダーの同期化など) の実行に使用します。

  注: システム・クーロン・ジョブは、*restore()* および *persist()* 操作自体を試行してはなりません。ユーザーがクーロン・ジョブ・クラスに関連付けられていないため、データ・アクセス・メソッドに組み込まれたアクセス検査が例外をスローします。

CronManager および CronScheduler

クーロン・ジョブの定義および作成は、CronManager クラスにより管理されます。クーロン・ジョブの構成情報は、CronConfigBean データ Bean によりメモリーに反映されます。クーロン・ジョブの定義は、知識ベースに保持されます。

クーロン・ジョブのスケジューリングおよび実行は、CronScheduler クラスにより管理されます。この singleton クラスはサーバー起動時にインスタンス化されます。

CronJob インターフェース

各クーロン・ジョブは、以下のように CronJob インターフェースを実装する Java クラスです。

```
public interface CronJob extends java.lang.Runnable
{
/**
* Specify the Cron Configuration bean object.
*
* @param config Cron configuration bean object.
*/
public void setCronConfiguration(CronConfigBean config);
/**
* Return the Cron Configuration bean object.
*
* @return CronConfigBean object.
*/
public CronConfigBean getCronConfiguration();
/**
* Initialization function. This function is called
* immediately after the object is created.
*
* @return true if initialization success, false otherwise.
*/
public boolean init();
/**
* Return the current scheduled time.
*
* @return Current schedule time in Calendar object.
*/
public Calendar getSchedule();
/**
* Reschedule the cron to reflect the changes made to the
* cronfiguration parameter. This function is called by the
* Cron Manager whenever cron configuration changes.
*/
public void reschedule();
/**
* Whether the job needs to be run again. This function is
* useful if there is some problem in the current run and you
* want to retry at specified time.
*
* @return true if the job is allowed to retry if the job
* did not run successfully
* on the last time of execution
*/
public boolean retry();
/**
* Determines whether to stop this cron job from running.
*
* @return true if the job has been slated to not run again
*/
```

```
public boolean stopRun();
/**
* Compute next cron run time: this is usually based on the cron
* run interval.
*/
public void computeNextSchedule();
/**
* Check to determine if the cron job is
* in a good state to run before triggering the thread to run.
*
* @return true or false. True means ready to run.
*/
public boolean isOKtoRun();
/**
* Is called when the thread starts.
*
* @return false if the job needs to be stopped. Return true to
* continue running.
*/
public boolean service();
/**
* Checks whether the next run time is later than the end run date.
*
* @return true if next run time greater than end run time
*/
public boolean isExpired();
}
```

Visual Modeler クーロン・ジョブの作成

このタスクについて

新しいクーロン・ジョブを作成するには、以下の手順を実行します。

手順

1. CronJob クラスの作成: SystemCron クラスまたは ApplicationCron クラスのいずれかを継承する必要があります。この両方のクラスは、抽象クラスであり、抽象クラス AbstractCronJob を継承します。

   実装が必要な唯一のメソッドは、*service()* です。これは、CronScheduler により開始されるインバウンド・ポストを処理するメソッドです。

   - ジョブ・スケジューラーのユーザー・インターフェースを使用して定義されたパラメーターがジョブに渡されると、AbstractCronJob クラスの *getParameter(String s)* メソッドおよび *getParameters()* メソッドを使用して、パラメーターを取得できます。これらのメソッドは、HttpServletRequest クラスの対応するメソッドと同様に動作します。
   - ジョブの結果をデータベースに保存する場合、*service()* メソッドが *setExecutionOutcome(String s)* メソッドを呼び出すようにする必要があります。

- AbstractCronJob クラスの *setRetry(Calendar c)* メソッドを呼び出すことにより、クーロン・ジョブを後から必ず再実行するように指定できます。Calendar パラメーターを使用して、ジョブの再実行が必要な時期を指定します。

2. システム管理アプリケーションの一部として提供される、ジョブ・スケジューラーのユーザー・インターフェースを使用して、クーロン・ジョブを定義します。それには、クーロン・ジョブのクラス、クーロン・ジョブの実行時期を判断するスケジュール、および実行時にクーロン・ジョブに渡される任意のパラメーターを指定します。クーロン・ジョブがアプリケーション・クーロン・ジョブとして実行される場合、ユーザーのユーザー名およびパスワードも指定する必要があります。

   パラメーターは、HTTP 要求パラメーターと同じ構文を使用してクーロン・ジョブに渡されます。例えば、Name1=Value1&Name2=Value2 です。

第 20 章 フィルターの概要

フィルターはオブジェクトであり、リソースに対する要求 (サーブレットまたは静的コンテンツ) 時、またはリソースからの応答時、またはその両方でフィルタリング・タスクを実行します。フィルタリング・タスクは J2EE 2.3 仕様の一部として定義されます。

フィルターは、*doFilter()* メソッドでフィルタリングが実行されます。あらゆるフィルターは、FilterConfig オブジェクトにアクセスし、このオブジェクトから初期化パラメーターを取得できます。また、ServletContext を参照し、(例えば) フィルタリング・タスクに必要なリソースのロードに使用できます。

フィルターは、Web アプリケーションのデプロイメント記述子内で構成できます。標準的なフィルター例としては、以下があります。

- 認証フィルター
- ロギングおよび監査のフィルター
- イメージ変換フィルター
- データ圧縮フィルター
- 暗号化フィルター
- トークン化フィルター
- リソース・アクセスのイベント・トリガーのフィルター
- XSLT フィルター
- MIME タイプ・チェーン・フィルター

第 21 章 Visual Modeler フィルター

Visual Modeler では、以下のフィルターを提供します。これは、次の com.comergent.dcm.core.filters パッケージの一部です。

- 『DosFilter』
- 『WSDLFilter』

DosFilter

このフィルターは、 Web アプリケーションをサービス妨害攻撃から保護するためのフィルターの基本として使用できます。

このフィルターを使用するには、com.comergent.dcm.core.filters.DosFilter クラスを継承するクラスを作成します。そして、その中で、*isRequestDenied()* メソッドをオーバーライドして、サービス妨害攻撃を識別してブロックするのに使用するロジックを実装します。

次に、 **web.xml** 構成ファイルを変更して、以下のように、実装クラスをフィルターとして宣言します。

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

および

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

WSDLFilter

WSDLFilter クラスは、標準の URL (http://server:port/s/dXML/5.0/OrderInterface.wsdl) などを使用してこのクラスにアクセスされる場合に、 Web サービス WSDL を変換するのに使用されます。

第 22 章 コンストレインド・フィールドの管理および表示

コンストレインド・データ・フィールドでは、多数の値の中から 1 つのみ採用できます。例としては、パートナー・レベル (「ゴールド (Gold)」、「シルバー (Silver)」など)、パートナーの地域 (「北西 (North-west)」、「ベネルクス (Benelux)」など)、およびスキル・レベル (「エキスパート (Expert)」、「認定 (Certified)」など) があります。Visual Modeler では、これらのデータ・フィールドをさまざまな方法で管理できます。管理方法の選択は、データ・フィールドの維持方法および使用方法により決まります。

オプション

コンストレインド・データ・フィールドおよび許可データ・フィールドを指定するには、以下のオプションがあります。

- データ・フィールドを、データベース表で値のセットとして保持します。相互参照表により値をビジネス・オブジェクトに割り当てるか、ビジネス・オブジェクト表内の値ごとにキーを参照することにより、値をビジネス・オブジェクトに割り当てます。
- 値を制約要素として XML スキーマ (**DsConstraints.xml** ファイル内で宣言) に保持します。制約を、データ・フィールドに関連付けられた DataElement の属性として指定します。
- 許可された値を、HTML テンプレート内の要素からの <SELECT> の値として組み込みます。

フィールドに対する許可値は、以下の場合を除いてデータベース表として保持することを推奨します。

- 値が実行時に変更されない予定である。
- データ・フィールドは、各ビジネス・オブジェクトで 1 つの値のみ取得できる。
- 値を、値自体により決まる自然順 (アルファベット順など) で表示できる。

以下の理由から 3 番目のオプションの使用を推奨します。

- 許可されたデータ値のリストを変更する場合、テンプレートまたはアプリケーション・コードの更新がメンテナンス上の問題になります。
- ユーザーが、HTML を変更して禁止値を戻すことができるため、セキュリティーに問題があることを表します。選択を検査するために Javascript (ユーザーが削除可能) を追加するか、ビジネス・ロジックの一部として戻り値を検査する必要があります。

基準

オプションの選択は、データ・フィールドの機能により決まります。以下の質問を自問して、データ・フィールドの使用方法を確定します。

1. ビジネス・オブジェクトを、コンストレインド・データ・フィールドの 1 つの値のみ、または複数の値に割り当てられますか。

回答が、同じビジネス・オブジェクトに対して複数の値が割り当て可能 (例えば、パートナーが複数の地域で業務を行う可能性がある) であれば、フィールドの値にはデータベース表を使用して、ビジネス・オブジェクトへの値の割り当てには相互参照表を使用する必要があります。

2. 新規ビジネス・オブジェクト作成時にデータ・フィールドに新しい値を入力できますか。またはそのデータ・フィールドに入力された値が制約セットの有効なメンバーであることを検査する必要がありますか。

   単一の値のみ許可されており、質問 2 に対する回答が、新しい値が許可されるということであれば、フィールド値を保持するためにデータベース表の使用が必要です。ただし、データ・フィールドの値をビジネス・オブジェクトに割り当てるのに相互参照表を使用する必要はありません。現在の Visual Modeler のインターフェースを使用して、制約要素の許可値のリストに値を動的に追加することはできません。

   コンストレインド・データ・フィールドで取り得る値は、動的に保守されるか、または始動時に 1 回読み込まれるだけですか。

3. 質問 1 に対する回答が単一の値で、質問 2 に対する回答が新しい値は許可されないであるが、動的な更新が必要な場合、データベース表を使用する必要があります。Visual Modeler が開始されると制約のある値が変わらない場合、制約要素を使用できます。

   コンストレインド・データ値を表示用にソートする必要がありますか。必要がある場合、値 (仮にアルファベット順) でソートされるか、値自体から推測不能な定義された順でソートされますか。

4. 最終的に、データ・フィールド値で値本来とは異なる順でのソートが必要であれば、この順序付けの情報はデータベース表に保持が必要です。ただし、自明の順序付け (アルファベット順など) によってのみ値を順序付ける場合、制約要素を選択できます。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

[カ行]

[サ行]

[タ行]

[ナ行]

[ハ行]

[マ行]

[ヤ行]

[ラ行]

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒103-8510

東京都中央区日本橋箱崎町19番21号

日本アイ・ビー・エム株式会社

法務・知的財産

知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation

J46A/G4

555 Bailey Avenue

San Jose, CA 95141-1003

U.S.A.

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

表示されている IBM の価格は IBM が小売り価格として提示しているもので、現行価格であり、通知なしに変更されるものです。卸価格は、異なる場合があります。

本書はプランニング目的としてのみ記述されています。記述内容は製品が使用可能になる前に変更になる場合があります。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。これらのサンプル・プログラムは特定物として現存するままの状態で提供されるものであり、いかなる保証も提供されません。IBM は、お客様の当該サンプル・プログラムの使用から生ずるいかなる損害に対しても一切の責任を負いません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© IBM 2011。このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。 © Copyright IBM Corp. 2011。

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

商標

IBM、IBM ロゴおよび ibm.com は、世界の多くの国で登録された International Business Machines Corp. の商標です。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、http://www.ibm.com/legal/copytrade.shtml をご覧ください。

Adobe、Adobe ロゴ、PostScript、PostScript ロゴは、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

IT Infrastructure Library は英国 Office of Government Commerce の一部である the Central Computer and Telecommunications Agency の登録商標です。

Intel、Intel ロゴ、Intel Inside、Intel Inside ロゴ、Intel Centrino、Intel Centrino ロゴ、Celeron、Intel Xeon、Intel SpeedStep、Itanium、Pentium は、Intel Corporation または子会社の米国およびその他の国における商標または登録商標です。

Linux は、Linus Torvalds の米国およびその他の国における商標です。

Microsoft、Windows、Windows NT および Windows ロゴは、Microsoft Corporation の米国およびその他の国における商標です。

ITIL は英国 Office of Government Commerce の登録商標および共同体登録商標であって、米国特許商標庁にて登録されています。

UNIX は The Open Group の米国およびその他の国における登録商標です。

Java およびすべての Java 関連の商標およびロゴは Oracle やその関連会社の米国およびその他の国における商標または登録商標です。

Cell Broadband Engine, Cell/B.E は、米国およびその他の国における Sony Computer Entertainment, Inc. の商標であり、同社の許諾を受けて使用しています。

Linear Tape-Open、LTO、LTO ロゴ、Ultrium および Ultrium ロゴは、米国およびその他の国における HP、IBM Corp. および Quantum の商標です。

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®、および Sterling Integrator® は、Sterling Commerce, Inc.、IBM Company の商標です。

Printed in Japan

日本アイ・ビー・エム株式会社
〒103-8510 東京都中央区日本橋箱崎町19-21