AndroMDA カスタマイズガイド

第１．２版

２００６／０１／２４

本書は公開されている情報に基づいて(株)エクサとその協力会社が共同して新規に書き起こしたものであり、権利は(株)エクサが保有します。

本書の複製を内部ネットワーク等の媒体で 2 次配布する場合は本書が(株)エクサによって公開された文書であることを明記してください。

本書で使用する製品名はそれぞれ各社の商標、または登録商標です。本書の内容についてはできるかぎり正確であるよう努力しています。しかしながら、本書の内容に基づく結果については責任を負いかねますのでご了承ください。

本書は無償で広く公開しておりますが、AndroMDA の利用方法や問題の解決などに関し、個別のお問い合わせには回答していません。

Java およびすべての Java 関連の商標は米国 Sun Microsystems, Inc.の登録商標または商標です。
MagicDraw UML は、米国 No Magic, Inc.の登録商標です。
JBoss は、米国 JBoss Inc.の登録商標です。
Object Management Group, OMG, Model Driven Architecture, Unified Modeling Language は Object Management Group, Inc. の登録商標です。
MDA、UML、OCL、XMI は Object Management Group, Inc. の商標です。
その他の会社名ならびに製品名は、各社の商標、もしくは登録商標です。

０ はじめに

本ガイドは AndroMDA のカスタマイズ方法について解説したものである。

AndroMDA のカスタマイズは大きく分けて３つのレベルがある。レベル１はテンプレートのカスタマイズであり、レベル２はカートリッジのカスタマイズである。最後のレベル３のカスタマイズはプロファイルのカスタマイズである。０．２章で３つのカスタマイズの概要と適用範囲について説明する。そして、以降の章で各カスタマイズの方法について説明する。１章ではカートリッジの概要、２章ではテンプレートカスタマイズ、３章ではカートリッジカスタマイズ、５章ではプロファイルカスタマイズについて詳しく解説する。また、４章では独自にカスタマイズしたカートリッジの例を示す。

本ガイドを理解することで、読者は AndroMDA をカスタマイズし、標準でサポートされていないフレームワークに AndroMDA を対応させることもできるようになる。

０．１　前提条件・環境

本ガイドは、AndroMDA の基礎的なことを理解しており、AndroMDA でシステムを構築できる人を対象としている。そうでない人は株式会社エクサが公開している「AndroMDA 入門ガイド」などを参考にし、AndroMDA の基本的な使用方法について理解してから本ガイドを読むことを推奨する。

本ガイドは、下記のソフトウェアとライブラリを使用し、動作確認を行っている。

- J2SDK1.4.2_04（J2SDK1.4 以上）
  - プロジェクトをビルドするときに使用する。
- Maven1.0.2
  - プロジェクトをビルドするときに使用する。
- JBoss4.0.1sp1
  - AndroMDA で構築したアプリケーションを動作させるときに使う J2EE コンテナ。
- AndroMDA3.0
  - AndroMDA 本体（※
- MagicDraw UML 9.5 Personal Edition
  - モデリングツール。AndroMDA が処理できる XMI ファイルを生成することが可能。
- MagicDraw UML 9.5 Professional Edition（オプション）
  - モデリングツール。オリジナルの Profile を作成する場合は必要になる。

※）2006 年 1 月現在 AndroMDA の最新バージョンは 3.1 であるが、このバージョンでは動作確認を行っていない。

各種ソフトウェアのセットアップについては、本ガイドでは説明しない。株式会社エクサが公開している「AndroMDA 入門ガイド」（http://www.exa-corp.co.jp/techinfo/）などを参考にしていただきたい。

０．２　AndroMDA のカスタマイズ

AndroMDA は Matthias Bohlen 氏と The AndroMDA Project による、MDA パラダイムにのっとったコード生成フレームワークである。UML モデリングツール(Magic Draw)から出力された XMI 形式ファイルを入力とし、様々なプラグインと連携して半完成のアプリケーションコンポーネントを出力する。

AndroMDA はカートリッジと呼ばれる変換ルールをプラグインすることで、様々なプラットフォームに対応する多様なプログラミング言語のソースコードや様々なファイルを自動生成することができる。

以下に AndroMDA のカートリッジと出力ファイルについての関係を簡単な図で示す。

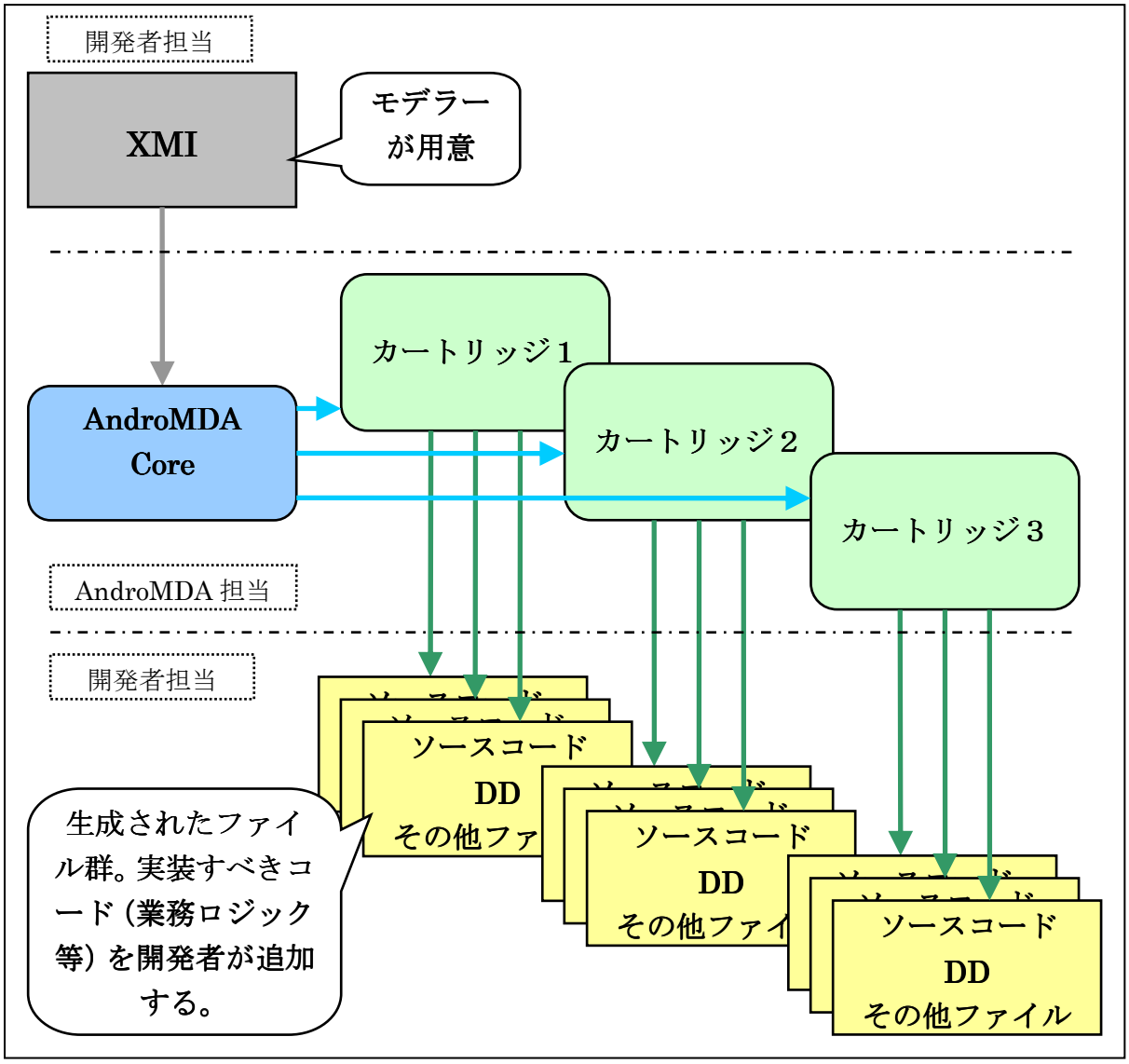


図０－１．　AndroMDA カートリッジと出力ファイルの関係

AndroMDA が提供しているカートリッジだけでは、限定されたフレームワーク（JBoss、Hibernate 等）に対応するファイル（DD、ソースコード、Mapping 定義ファイル等）しか生成できない。つまり、独自のフレームワーク等を使用する場合、AndroMDA の強力な MDA 機能を生かすことができない。他のフレームワークに対応させたファイルを生成したい場合は、カートリッ

ジを作成（または変更）する必要がある。カスタマイズについて説明する前に、カートリッジ内部の構成を説明する。下図でカートリッジの構成とファイル生成の順序を示す。

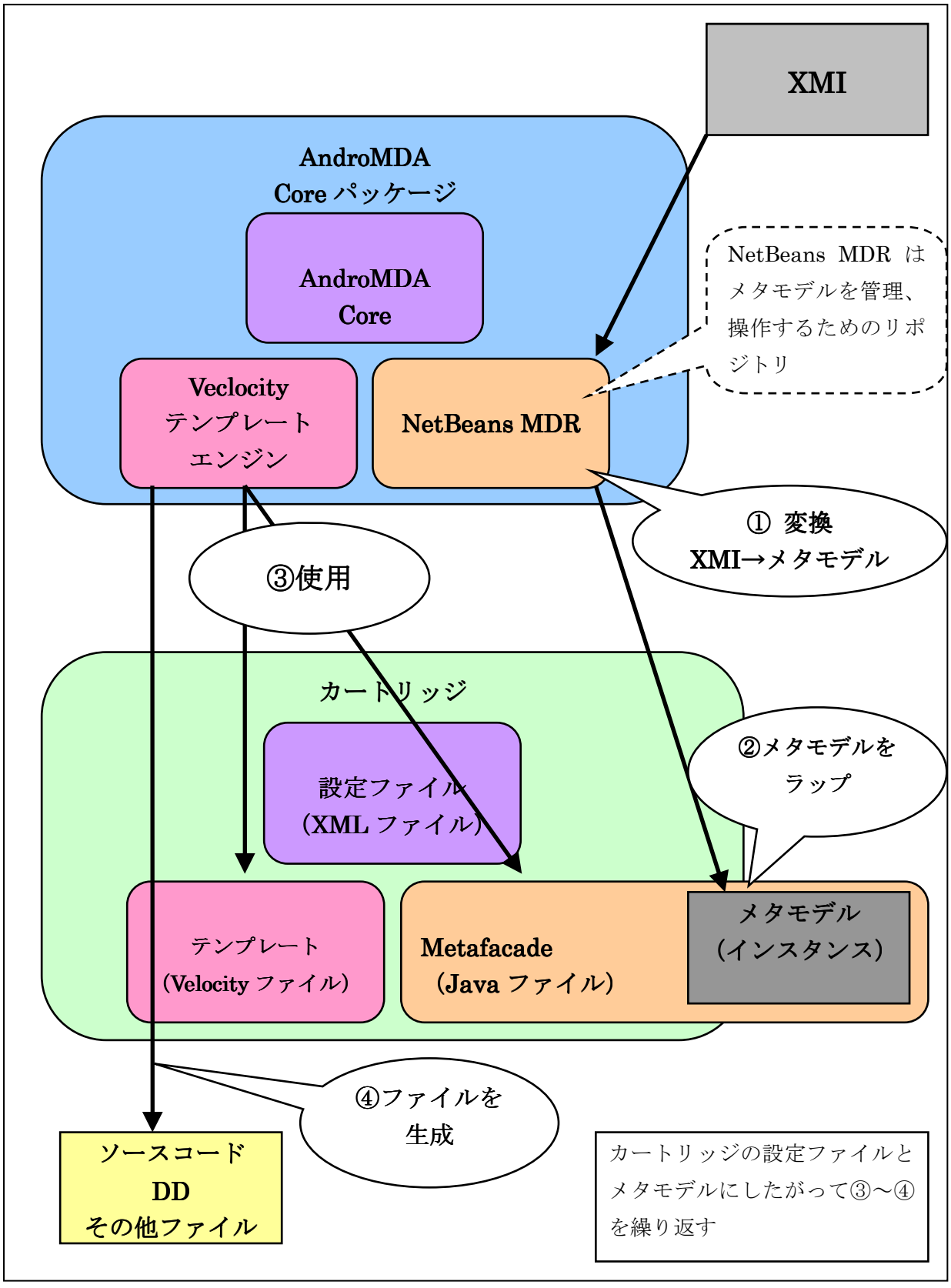


図０－２． カートリッジ内部の構造

AndroMDA のカスタマイズには、３つのレベルがある。

レベル１．　「テンプレートのカスタマイズ」。テンプレートカスタマイズは既存のカートリッジ内のテンプレートファイルを修正、または新しく追加することである。テンプレートを新しく追加する場合は、設定ファイルも修正する。一般的に、「テンプレートのカスタマイズ」は難易度が低く、カスタマイズのための手間も比較的小さい。しかし、カスタマイズ可能な範囲は、出力されるファイルのフォーマットを修正する程度である。モデル（XMI）から独自の情報を取得し、出力されるファイルに反映させたい場合は対応できない。つまり、カスタマイズできる範囲が限られており、カスタマイズに対する要求を満たせない可能性がある。

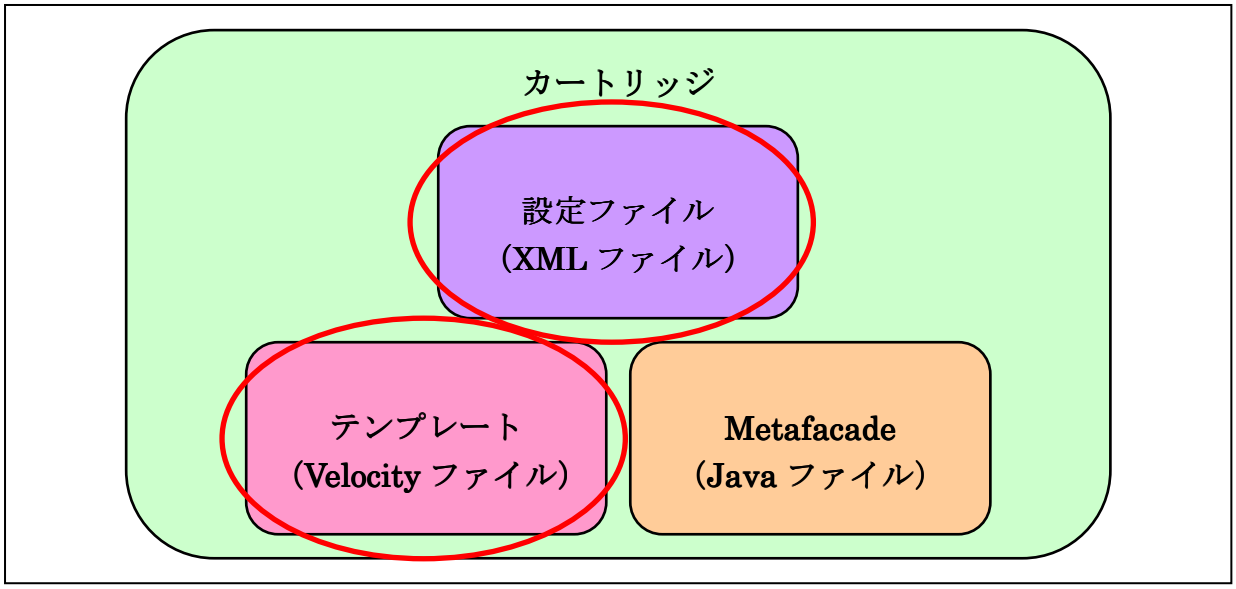


図０－３．　カートリッジのカスタマイズ範囲[テンプレートのカスタマイズ]

「テンプレートのカスタマイズ」には、カートリッジ自体を修正しないもう１つの方法がある。これはマージファイルを AndroMDA プロジェクト内に配置し、そのファイルでカートリッジが生成したファイルを置き換えるというものである。置き換えるには AndroMDA プロジェクトのカートリッジ設定ファイルにマージの宣言を記述する。これはカートリッジ自身が提供している機能で、カートリッジによってはこの機能を有していないものもある。

レベル２．　「カートリッジのカスタマイズ」。既存のカートリッジを修正する場合と、まったく新しいカートリッジを１から作る場合の２つの方法がある。変更する内容はレベル１の「テンプレートのカスタマイズ」＋AndroMDA Metafacade を修正、または新しく追加することになる。一般的に「カートリッジのカスタマイズ」は難易度が高く、カスタマイズのための作業もかなり多くなる。しかし、カスタマイズ可能な範囲はほぼ無制限であり、カスタマイズに対するほとんどの要求を満たすことが出来る。

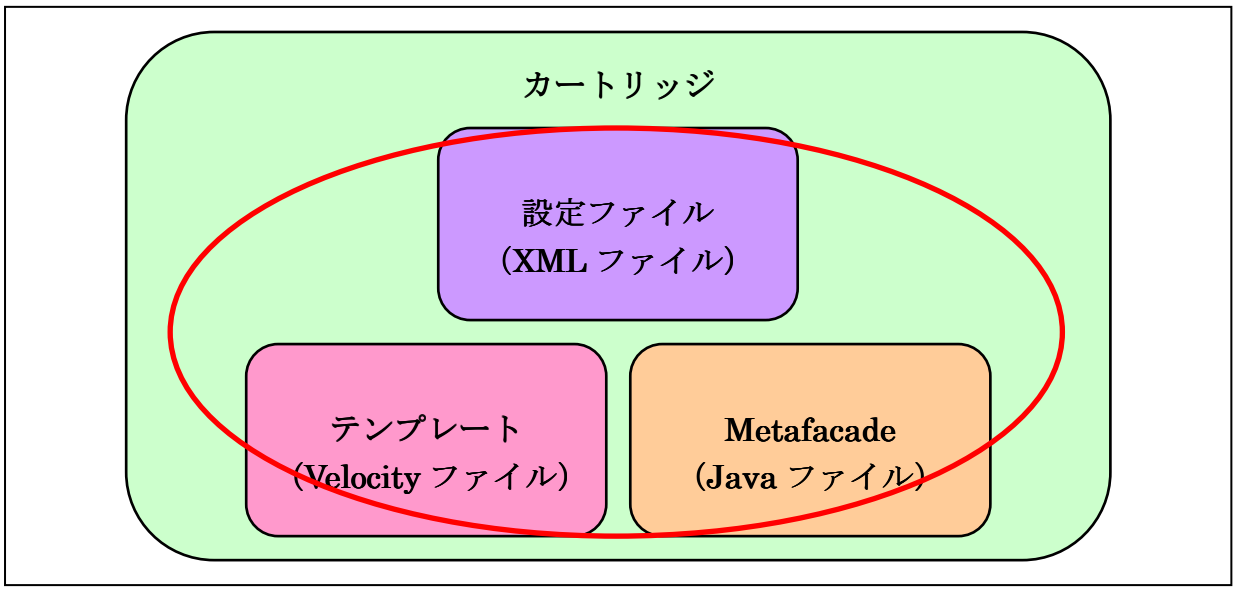


図０－４． カートリッジのカスタマイズ範囲[カートリッジのカスタマイズ]

レベル３． 「プロファイルのカスタマイズ」。レベル２の「カートリッジのカスタマイズ」とモデリングの際に必要なプロファイルを変更（または新規作成）することである。新しいステレオタイプやタグ付き値を定義したい場合に、このレベルのカスタマイズを行う。プロファイルはモデルを作成する時とビルドするときに必要になる。また、プロファイルはカートリッジ内部には含まれない。レベル２の「カートリッジのカスタマイズ」を行う場合は、レベル３のカスタマイズとセットで行われることが多い。

以上３つのレベルがあり、それぞれコストとカスタマイズ可能な範囲が異なっている。一般的に、簡易性とカスタマイズ性は次の図のようになる。

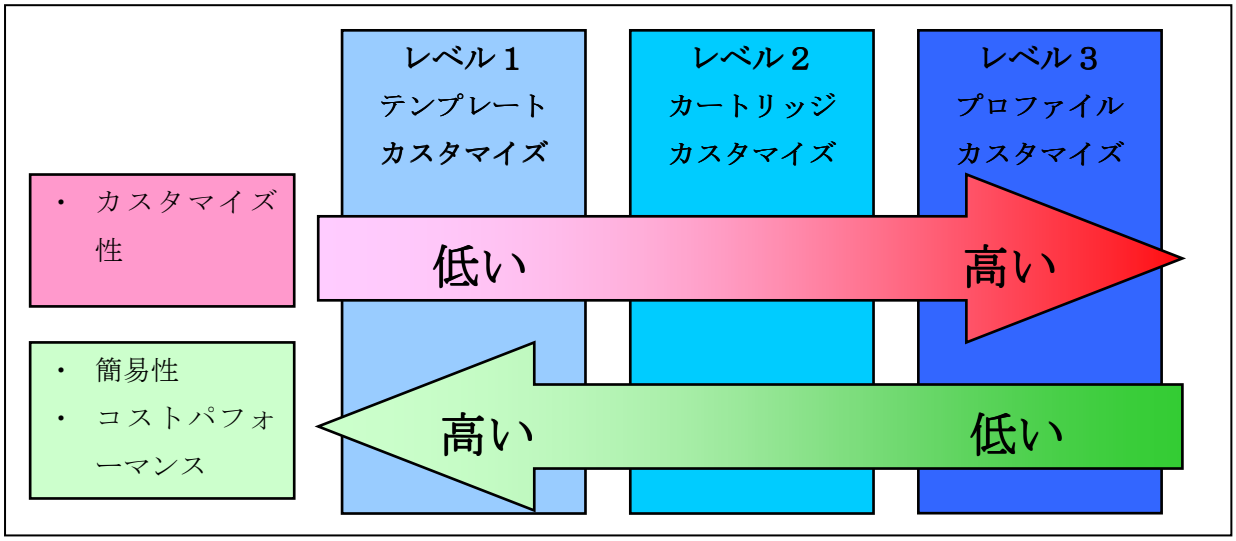


図０－５． 各カスタマイズのカスタマイズ性と簡易性

カスタマイズの要求に応じて３つのカスタマイズ方法のうち、どれかを選択する必要がある。

1 カートリッジとは？

カートリッジとは、AndroMDA によるファイル生成の変換ルールをまとめたものである。既存の変換ルールは主にオープンソースのフレームワークにあわせたものを生成するカートリッジが多い（例えば、Struts や Hibernate など）。

カートリッジ自体もひとつのプロジェクトである。この章では、カートリッジプロジェクトの構成と、ファイル生成時のカートリッジの役割、カートリッジを構成する要素の概要を説明する（カートリッジを構成する要素の詳細は３章に記述する）。

１．１　カートリッジプロジェクトの構成

カートリッジ自体プロジェクトとなっている。ここではカートリッジプロジェクトの構成を記述する。下図は Spring カートリッジのプロジェクト構造である。

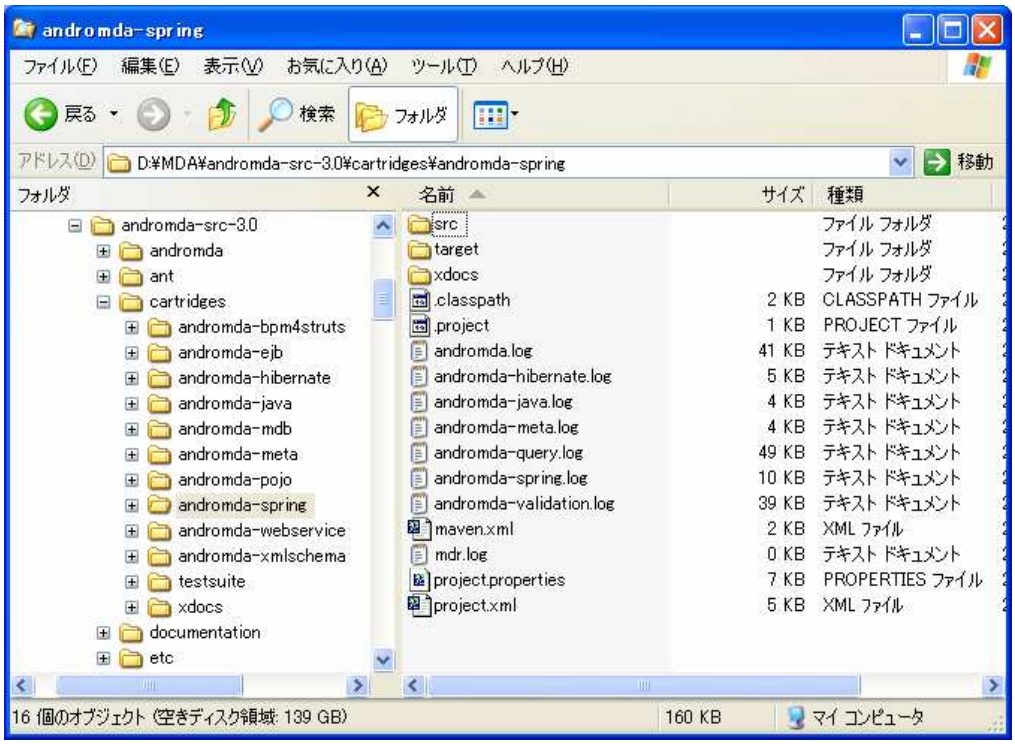


図１－１．　カートリッジのプロジェクト構成

基本的にどのカートリッジも同様の構成である。表１－１に構成の内容を記述する。

表１－１．プロジェクトフォルダの内容

フォルダ/ファイル	内容
src	カートリッジのモデルや Metafacade クラスのソースファイル、テンプレートファイル、設定ファイルなどを格納している
target	Metafacade のクラスやカートリッジの jar ファイルなどを格納している
xdocs	カートリッジのドキュメントを作成するためのファイルを格納している
maven.xml	カートリッジのビルドに実行する処理を記述した設定ファイル
project.properties	カートリッジのビルド時に使用する値を記述したプロパティファイル
project.xml	カートリッジをビルドする際に参照するリソースを記述する設定ファイル

下記に表１－１の「src」、「target」フォルダの構成を記述する

表１－２．src フォルダの内容

フォルダ/ファイル	内容
java	カートリッジの Metafacde のソースファイルを格納している
META-INF	カートリッジの設定ファイルを格納している
templates	カートリッジのテンプレートファイルを格納している
test	カートリッジテストを格納している
uml	カートリッジの Metafacade クラスのモデルファイルを格納している

表１－３．target フォルダの内容

フォルダ/ファイル	内容
cartridge-test	生成したカートリッジテストのファイルと検証用ファイルを格納している
classes	Metafacade のクラスファイルを格納している
src	Metafacade のソースファイルを格納している
test-classes	テスト用のファイルすべてを格納している
test-reports	テスト結果を格納している
andromda-カートリッジ名-cartridge-バージョン番号.jar	カートリッジの jar ファイル Spring カートリッジであれば andromda-spring-cartridge-3.0.jar

１．２　カートリッジの構成要素

ここでは、簡単ではあるがカートリッジを構成する上で重要な要素の３つを説明する。各要素は、３章で詳細を説明する。

１．２．１　　Metafacade クラス

Metafacade クラスとは、UML のモデルに対するアクセス手段を提供する Facade クラスである。UML モデルから目的のファイルを生成するには、UML モデルの内容を正しく解釈する必要がある。AndroMDA が内部で利用しているライブラリに UML のモデルを Java のクラスへマッピングするものがある。本ガイドではそのクラスを「**metaclass**」と呼ぶ。metaclass と UML のモデルの関係は下図のようになる。UML モデルはステレオタイプ「**UMLClass**」がついているものである。

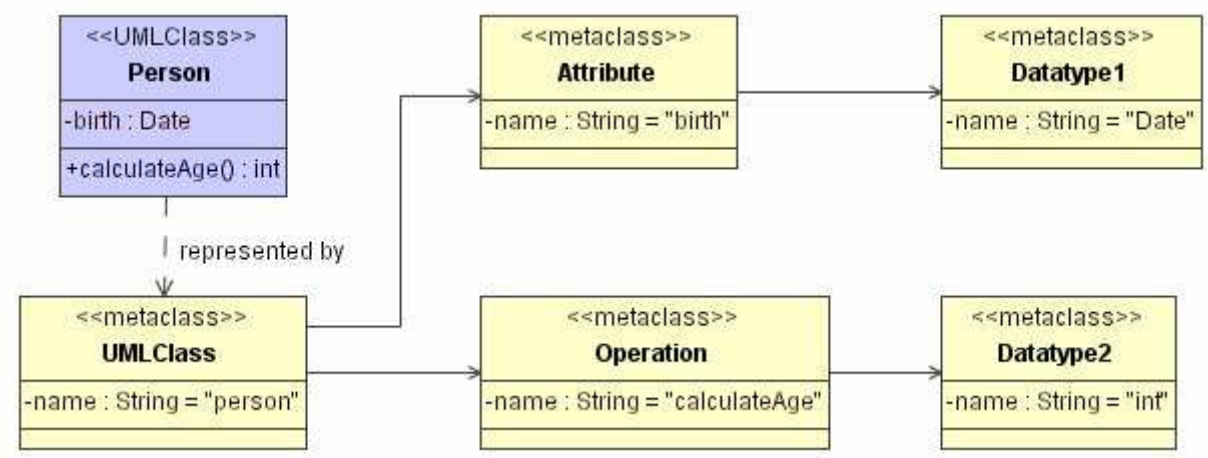


図１－２．UML モデルと metaclass の関係

ステレオタイプ<<metaclass>>がついているものが metaclass である。図からもわかるとおり、UML モデルの１つのクラスは複数の metaclass によって表現される。

Metafacade クラスとは、metaclass の Facade の役割をするクラスである。AndroMDA ではその Metafacade クラスをファイル生成に利用している。

尚、Metafacade クラス自体も UML でモデリングし、AndroMDA の Meta カートリッジによって生成するものである。

１．２．２　　テンプレートファイル

カートリッジによって生成するファイルの定型文を記述しているファイル。実態はテンプレートエンジンである Velocity のテンプレートファイルである。設定ファイルでテンプレートファイルと Metafacade クラスをマッピングし、ファイル生成時に利用する。

１．２．３　　設定ファイル

カートリッジによるファイル生成時の様々な情報を設定するファイルである。主に下記の３つのファイルがある。

- andromda-cartridge.xml

  生成するファイルのテンプレートを記述しているテンプレートファイルのことを記述す

る設定ファイル（要変更）

- andromda-metafacades.xml
  metaclass とマッピングする Metafacade クラスについて記述する設定ファイル
- andromda-profile.xml
  Metafacade クラスなどで参照する固定値などを記述する設定ファイル

各ファイルの詳細については、２章で説明する。

１．３　ファイル生成時のカートリッジ利用の様子

下図がビルド時にカートリッジがどのように影響しているかの様子である。

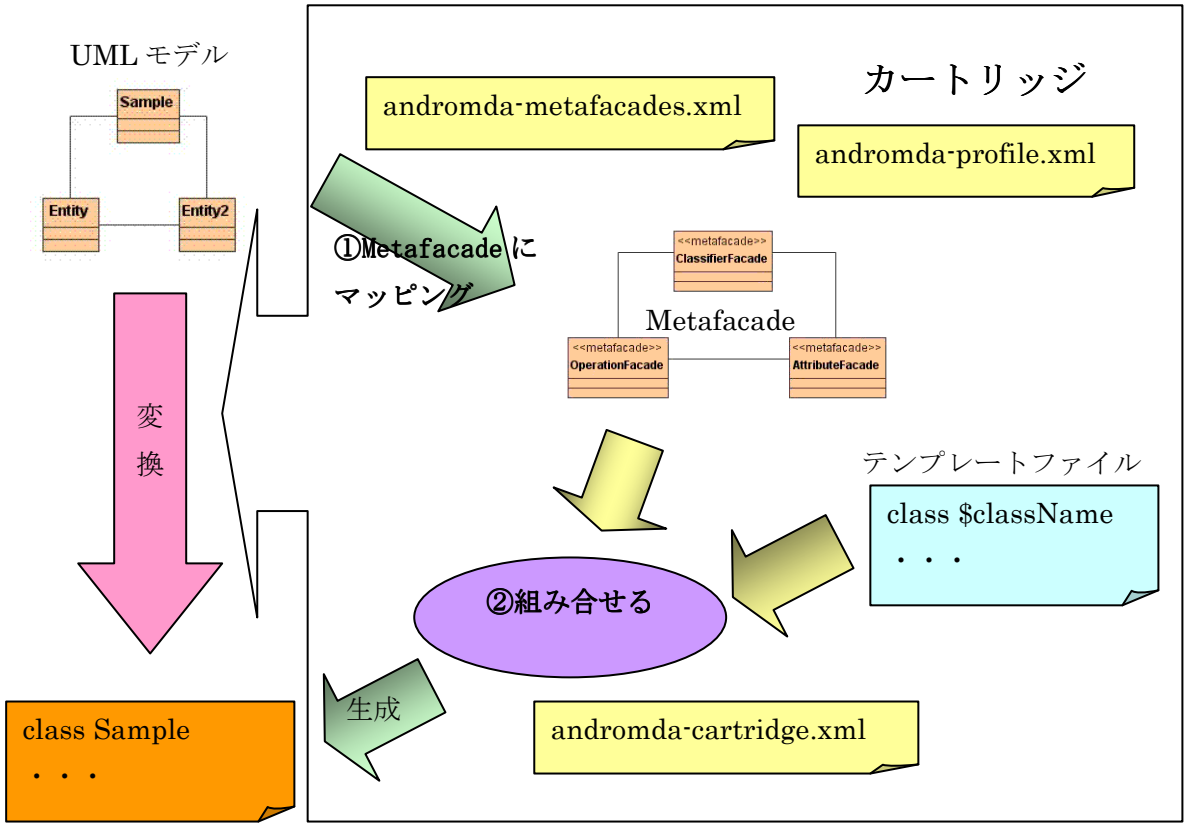


図１－３．カートリッジによるファイル生成の様子

上記図の左にある「変換」矢印の内部処理が図の右の部分である。

下記の処理は図中の番号と対応している。

① UML モデル図の要素を Metafacade クラスへマッピングする

② Metafacade クラスと対応するテンプレートファイルを組み合わせてファイルを生成する

１．４　既存カートリッジについて

AndroMDA にはいくつものカートリッジが用意されている。下記にそれぞれのカートリッジについて簡単に説明する。

表１－４．AndroMDA の既存カートリッジ一覧

カートリッジ	説明	備考
BPM4Struts	Web アプリケーションフレームワークである Struts を利用したソースコードや設定ファイルなどを生成するカートリッジ	
EJB	CMP の EJB エンティティを生成するカートリッジ	
Hibernate	O/R マッピングツールである Hibernate を利用したソースコードや設定ファイルなどを生成するカートリッジ	Hibernate だけでなく、EJB の SessionBean も利用する
Java	基本的な Java クラスを生成するカートリッジ	
Meta	AndroMDA の Metafacade を生成するカートリッジ	
Spring	DI コンテナである Spring Framework を利用したソースコードや設定ファイルを生成するカートリッジ	Spring だけでなく、Hibernate、EJB も利用する。ただし、EJB については利用しないことも可能。
WebService	Apache Axis 用の設定ファイルを生成するカートリッジ	Web Service Deployment Descriptor のファイル群と WSDL を生成する
XmlSchema	クラスモデルから XmlSchema を生成するカートリッジ	

これらの既存カートリッジについては AndroMDA のサイトにて説明があるので、詳しくはそちらを参照していただきたい。プレゼンテーション層は BPM4Struts を使用し、永続化層については、EJB、Hibernate、Spring カートリッジのいずれかより選択する。尚、Meta カートリッジはカートリッジを作成・変更する上で重要なカートリッジである。すべてのカートリッジは、Meta カートリッジを利用して生成される。

2 テンプレートのカスタマイズ

この章ではテンプレートのカスタマイズについて解説する。この章では説明のため、AndroMDA によって生成されるプロジェクトのルートディレクトリを「**ProjectRoot**」とする。

２．１ AndroMDA でのビルド時の処理とプロジェクトの構成

カスタマイズの説明をする前に AndroMDA でのビルド時の処理と作成されるプロジェクトの構成について説明する。

２．１．１ ビルド時の処理

AndroMDA ではプロジェクトをビルドする際に、ユーザがモデリングした UML のモデルを元にして各カートリッジが Java ソースや JSP などのファイルを作成する。その際に、作成するファイルは、各カートリッジが持っているテンプレートファイルを利用して作成される。
カートリッジは、いくつか用意されており作成するアプリケーションによって使用するカートリッジを選択する。下記に用途別に利用されるカートリッジを記す。

表２－１．用途別に利用されるカートリッジ

用途	カートリッジ	備考
Web アプリケーション	・ BPM4Struts カートリッジ ・ Hibernate カートリッジ ・ Spring カートリッジ ・ EJB カートリッジ ・ Java カートリッジ	Hibernate、Spring、EJB カートリッジはユーザが選択する
UI を持たないアプリケーション	・ Hibernate カートリッジ ・ Spring カートリッジ ・ EJB カートリッジ ・ Java カートリッジ	Hibernate、Spring、EJB カートリッジはユーザが選択する
WebService を利用するアプリケーション	・ Web アプリケーションで使用されるカートリッジ ・ WebService カートリッジ	・ BPM4Struts カートリッジは使用するか選択する ・ Hibernate、Spring、EJB カートリッジはユーザが選択する

下記に AndroMDA のプロジェクトをビルドしたときの処理概要を記す。

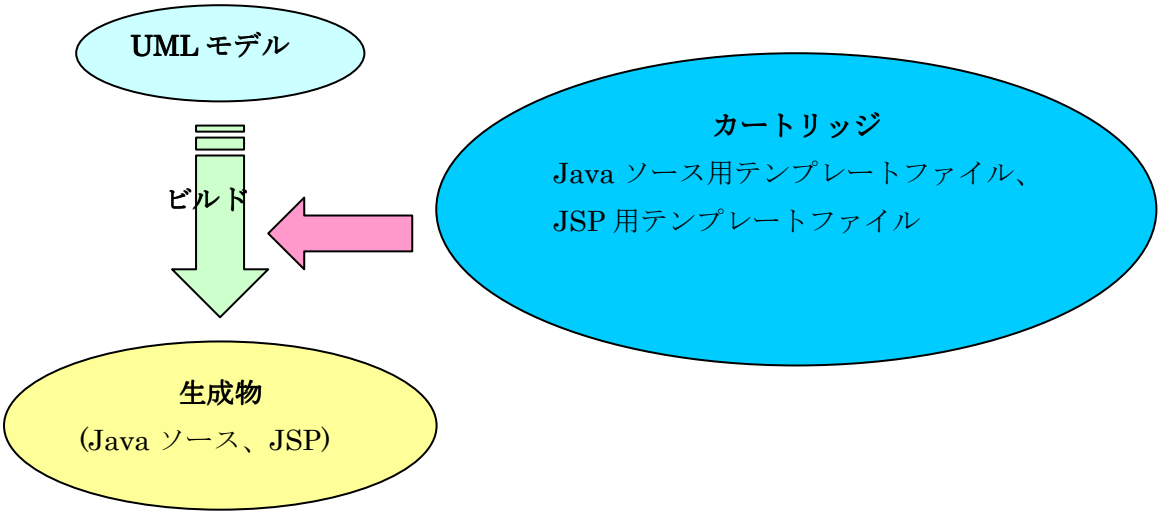


図２－１．ビルド時の処理の概要図

ファイルを生成する際利用するテンプレートファイルは、テンプレートエンジンである Jakarta の Velocity が利用されている。テンプレートファイルには Velocity テンプレートファイルである拡張子「vsl」のものと、Velocity 用マクロのファイルである拡張子「vm」のものがある。

尚、本ガイドでは Velocity についての説明は行わないので、Velocity については Web サイトなどを参照していただきたい。

２．１．２　　プロジェクトの構成

構成が理解できないと、カスタマイズを行うのは困難である。下記に実際にプロジェクトを作成した際の構成を表す。

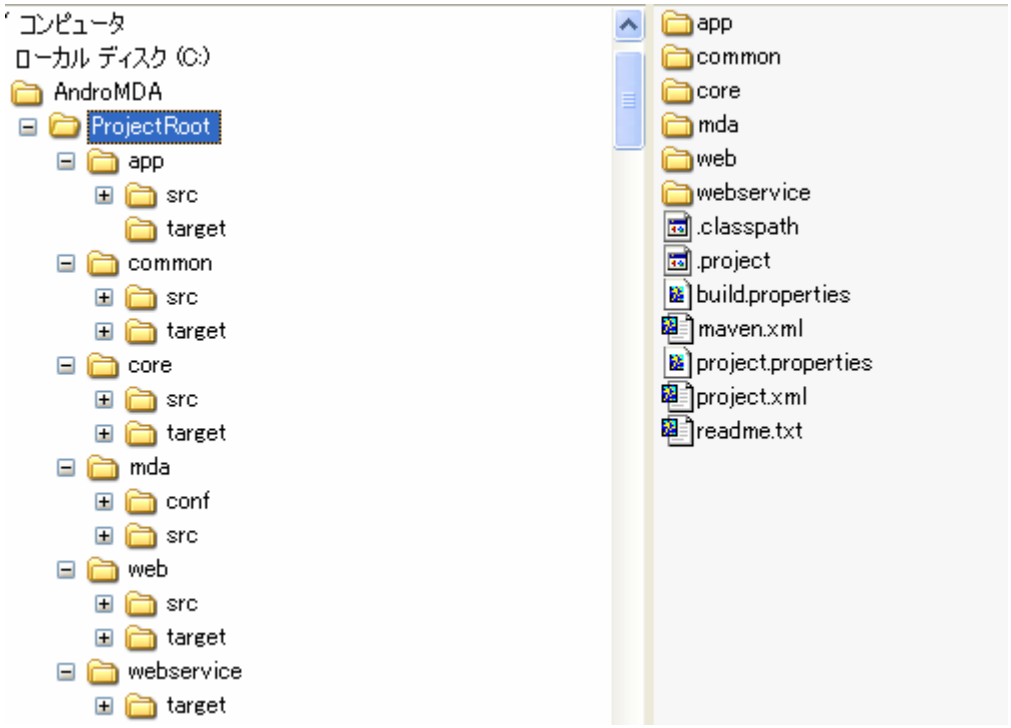


図２－２．プロジェクト構成

プロジェクトルートディレクトリは「ProjectRoot」である。「ProjectRoot」直下にある各ディレクトリをこれ以降、本章では**モジュール**と記述する。
モジュールの説明を下記に記す。

表２－２．プロジェクトの各モジュールの内容

モジュール	内容	ファイルを出力するカートリッジ	備考
app	ear ファイルを作成するためのリソースとクラスを集める。	すべて	プロジェクト作成時に「ear」or「war」を作成するという質問で「ear」を選択した場合にのみ作成される
common	他のモジュールで共有されるリソースとクラスを集める	すべて	このモジュールに作成されるファイルはすべてユーザによる変更が不可なファイル
core	サーバ側のリソースとクラスを集める	BPM4Struts カートリッジ以外	
mda	アプリケーションを作成するために必要な情報を集める	なし	アプリケーションを作成するための設定ファイルや UML モデルが存在する
web	プレゼンテーション側のリソースとクラスを集める。war ファイルを作成する	BPM4Struts カートリッジ	
webservice	WebService を提供するためのリソースとクラスを集める	WebService カートリッジ	

各モジュールは、以下のような構成になっている。

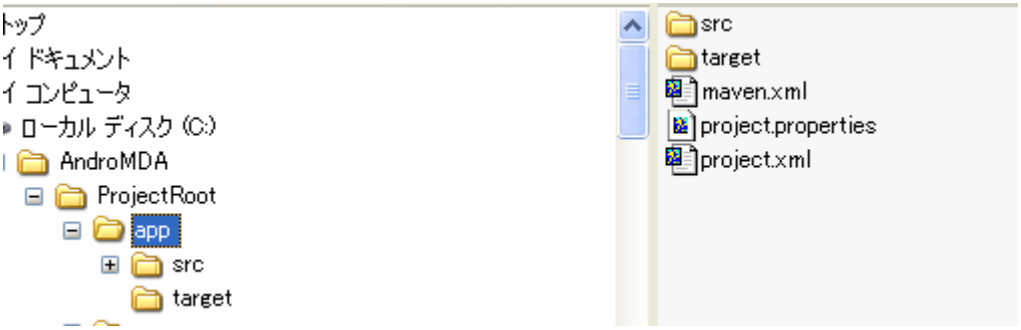


図２－３．各モジュールの構成

モジュール以下の構成の説明を下記に記す。

表２－３．モジュールの構成

ディレクトリ/ファイル	内容	備考
src	ビルド時に出力ファイルが存在しない場合だけ生成されるファイルの配置場所	ユーザが変更できるソースなどの配置場所
target	ビルド時に毎回出力されるファイルの配置場所	ユーザが変更できないソースファイルや JSP ファイルなどの配置場所
maven.xml	ビルドに実行される処理を記述した設定ファイル	
project.properties	ビルド時に使用される値を記述したプロパティファイル	
poject.xml	モジュールをビルドする際に参照するリソースを記述する設定ファイル	
conf	ビルド時のプロジェクト全体の設定を記述するファイルの配置場所	mda モジュールにのみ存在する

本章で説明するのは基本的にファイル生成に使用されるテンプレートファイルに対してのカスタマイズとなる。

２．２　目的別カスタマイズ方法

テンプレートのカスタマイズは目的によって実現方法や適用範囲は異なる。ここでは、目的と実現方法についての対応を表にまとめる。

表２－４．目的別カスタマイズ方法

目的	実現方法	適用範囲	備考
web.xml や struts-config.xml に設定を追加したい	設定ファイルのマージ	BPM4Struts カートリッジ	
テンプレートファイルの差し替えをしたい	テンプレートファイルのマージ	各カートリッジ単位	
アプリケーションにファイルの上書きや追加をしたい	アプリケーションへファイルのマージ	プロジェクトの各モジュール単位	
アプリケーションが参照する Jar ファイルを追加したい	参照 jar ファイルの追加	プロジェクトの各モジュール単位	テンプレートのカスタマイズではないが、カスタマイズを行ううえで必要な知識

これ以降に実現方法の詳細について記述する。

２．３　設定ファイルのマージ

AndroMDA の BPM4Struts カートリッジは、Web アプリケーションの設定ファイル「**web.xml**」や「**struts-config.xml**」ファイルを出力している。これらのファイルに設定を追加できるように BPM4Struts カートリッジにはマージポイントと呼ばれる設定の追加可能領域が設けられている。よって、各 xml ファイルに設定を追加したい場合はマージポイントに記述すればよい。
下記に用意されているマージポイントとその内容を記述する。

表２－５．マージポイント

マージポイント	設定ファイルでのタグ	内容
context-param	<context-param>	コンテキストの初期化パラメータ
filter	<filter>	クライアントとの送受信データのフィルタ
filter-mapping	<filter-mapping>	どの送受信データにフィルタ処理を適用するか指定する
listener	<listener>	何らかのイベントにより動作する listener クラス
error-page	<error-page>	HTTP エラーコード、Java 例外クラスに対応したエラーページを設定する
servlet	<servlet>	初期化パラメータ、メモリロードのタイミングなどサーブレットクラスに対する振る舞いを指定する
servlet-mapping	<servlet-mapping>	URL でアクセスする際のサーブレットクラスの名称を指定
welcome-file-list	<welcome-file-list>	URL がディレクトリ指定だった場合に返す初期ファイルを指定する
security-constraint	<security-constraint>	コンテナに認証を付与する
security-role	<security-role>	認証で使用するロールの定義
global-forwards	<global-forwards>	struts-config の global-forwards
mime-type	<mime-mapping>	ファイル拡張子と MIME タイプのマッピングを設定する
custom-messages	リソースファイルに追記される	マージするメッセージファイルのパス
taglib	web.xml の<taglib>	JSP の振る舞いを指定する

以降、「**filter**」と「**filter-mapping**」設定の追加を例に取り説明する。尚、設定するフィルタクラスについては、**ProjectRoot/web/src/java** 以下に「**filter.CharacterEncodingFilter**」という名前のクラスで「**encoding**」というパラメータを持つものと仮定する。

２．３．１　　マージポイント有効の確認

ProjectRoot/mda/project.xml を開き、以下の BPM4Struts カートリッジの設定に太字部分が記述されていることを確認する。

```
<dependency>
    <groupId>andromda</groupId>
    <artifactId>andromda-bpm4struts-cartridge</artifactId>
    <version>${andromda.version}</version>
    <properties>
        ・・・
        <mergeMappingsUri>
            file:${maven.conf.dir}/mappings/Bpm4StrutsMergeMappings.xml
        </mergeMappingsUri>
    </properties>
</dependency>
```

図２－４．マージポイント有効定義

太字の部分にマージポイントを記述するファイルが指定されているので、記述されていればマージポイントは有効になっている。

※プロジェクトを作成するとデフォルトでは記述されており有効になっている。

２．３．２　　マージポイントの設定

「ProjectRoot/mda/conf/mappings 」に２．３．１で確認したマージポイントの設定ファイル「Bpm4StrutsMergeMappings.xml」があるので開く。

開くと各マージポイントがコメントで記述されているので、追加したいマージポイントに記入する。ここではフィルタクラスの設定を追加するので、「filter merge-point」と「filter-mapping merge-point」部分に記入する。

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<mappings name="Bpm4StrutsMerge">
・・・
    <mapping>
        <from><![CDATA[<!-- filter merge-point -->]]></from>
        <to>
            <![CDATA[
<filter>
  <filter-name>CharacterEncodingFilter</filter-name>
  <filter-class>filter.CharacterEncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
</filter>
            ]]>
        </to>
    </mapping>
    <mapping>
        <from><![CDATA[<!-- filter-mapping merge-point -->]]></from>
        <to>
            <![CDATA[
<filter-mapping>
  <filter-name>CharacterEncodingFilter</filter-name>
  <url-pattern>*.do</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>CharacterEncodingFilter</filter-name>
  <url-pattern>*.jsp</url-pattern>
</filter-mapping>
            ]]>
        </to>
    </mapping>・
・・・
</mappings>
```

図２－５．マージポイントファイルの記述例

太字の部分が追加したフィルタクラスと、クラスを利用するパターンの設定である。これで、

マージポイントの設定は完了する。

２．３．３　　マージの確認

プロジェクトをビルドし、「ProjectRoot/web/target/src/WEB-INF」の「web.xml」ファイルを開き設定が追加されているのを確認する。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">
  <filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-class>filter.CharacterEncodingFilter</filter-class>
    <init-param>
      <param-name>encoding</param-name>
      <param-value>UTF-8</param-value>
    </init-param>
  </filter>
 ・・・
  <filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>*.do</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>*.jsp</url-pattern>
  </filter-mapping>
 ・・・
</web-app>
```

図２－６．設定ファイルのマージ結果

２．４　テンプレートファイルのマージ

実際にアプリケーションを開発する際に標準のテンプレートで出力されるファイルでは足りない記述、または余分な記述がある場合がある。そのようなときは、カートリッジが使用するテンプレートファイルを書き換え、差し替えることによって解決する。

各カートリッジは UML のモデルから Java のソースファイルや JSP ファイル、設定ファイルをテンプレートファイルから出力する。

このカスタマイズは、テンプレートファイルを差し替えるのでカートリッジから差し替えたいテンプレートファイルを取得する必要がある。テンプレートファイルは、各カートリッジの jar ファイルまたはソースディレクトリに含まれている。

下記に jar ファイルの場合と、ソースディレクトリの場合のテンプレートファイルの場所を記述する。

・ jar ファイル

各カートリッジの jar ファイルを解凍すると下記のような構成になっている(カートリッジによっては存在しないディレクトリもある、ただし templates ディレクトリは基本的に存在する)。

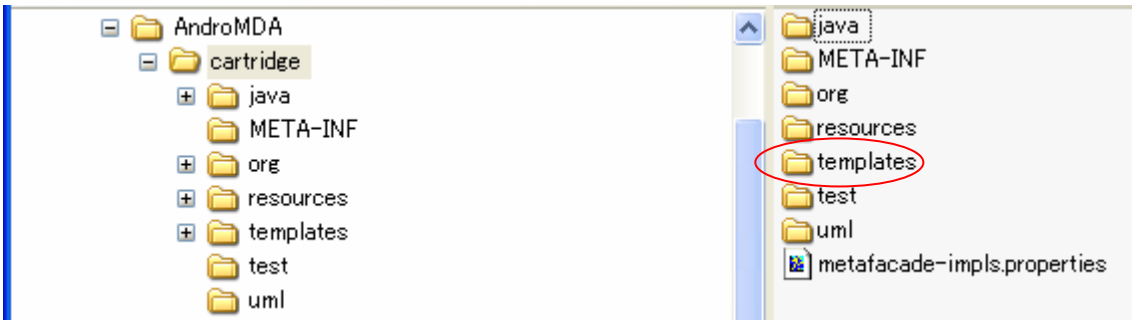


図２－７．jar ファイルを解凍した場合のカートリッジ構成

解凍後に生成された「templates」ディレクトリ以下にテンプレートファイルが存在する。

・ ソースディレクトリ

AndroMDA のソースファイルの中にある「andromda-src-3.0/cartridges」ディレクトリ以下に各カートリッジのソースディレクトリが存在する。下記のような構成になっている(カートリッジによっては存在しないディレクトリもある、ただし templates ディレクトリは基本的に存在する)。

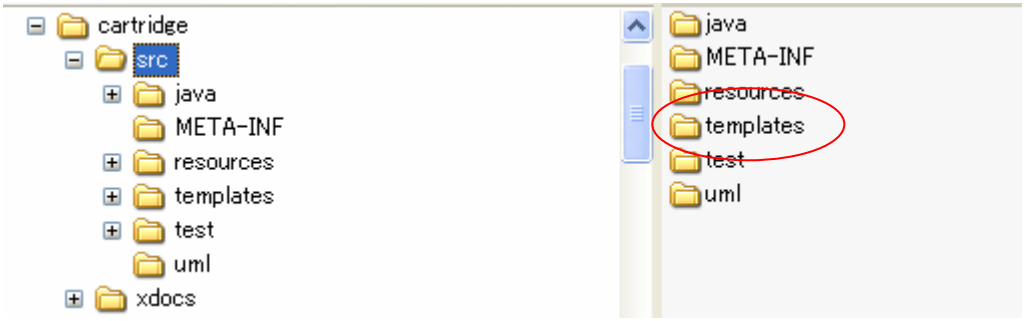


図２－８．ソースディレクトリの場合のカートリッジ構成

「templates」ディレクトリ以下にテンプレートファイルが存在する。

本章ではカートリッジのひとつ BPM4Struts カートリッジの JSP のテンプレートファイルを差し替える方法を例にとり説明する。

２．４．１　　テンプレートファイルの配置場所の設定

テンプレートファイルを差し替えるには、差し替えるテンプレートファイルをどこに置くのかを設定する必要がある。ここでは、その設定について説明する。

「ProjectRoot/mda/project.xml」を開くと各カートリッジの定義および設定内容が記述されている。<dependency/>タグ内の<artifactId/>タグの値が「andromda-bpm4struts-cartridge」のものが、BPM4Struts カートリッジの設定箇所である。差し替えテンプレートファイルのパスを設定するには、<mergeLocation/>タグを使用する。

下記の太字のようにテンプレートファイルの配置場所を設定する。

```
<dependency>
    <groupId>andromda</groupId>
    <artifactId>andromda-bpm4struts-cartridge</artifactId>
    <version>${andromda}</version>
    <properties>
      ・・・
      <mergeLocation>custom</mergeLocation>
      ・・・
    </properties>
</dependency>
```

図２－９．テンプレートファイルの配置場所設定

<mergeLocation/>タグに配置場所を ProjectRoot からのパスで記述する。
ここでは配置場所を「custom」とする。

２．４．２　　差し替え用テンプレートファイルの配置

差し替え用のテンプレートファイルを２．４．１で設定した場所へ配置する。ここでは BPM4Struts カートリッジの「page.jsp.vsl」ファイルを差し替える対象とする。page.jsp.vsl ファイルは、BPM4Struts カートリッジの「templates」ディレクトリから「templates/bpm4struts/pages」というパスにある。

２．４．１で設定したディレクトリへ「templates」ディレクトリ以下からコピーする。コピーした結果は、「custom/templates/bpm4struts/pages/page.jsp.vsl」というパスになる。

図２－１０．差し替え用テンプレートファイルの配置場所

２．４．３　　テンプレートファイルの編集

コピーした「page.jsp.vsl」を開き、ファイルの最後あたりに下記の太字部分を追加する。

```
・・・
        <div id="pageHelpSection">
            <blockquote>
#if ($jsp.validationRequired ・・・)
                <bean:message key="required.fields.asterisk"/>
#end
#if ($onlineHelp == "true")
                <a href=""・・・>
                    <bean:message key="online.help.href"/>
                </a>
                <html:img page="/layout/help.gif" style="display:inline;"/>
#end
<h1>Screen customizing</h1>
            </blockquote>
        </div>
    </tiles:put>
</tiles:insert>
```

図２－１１．テンプレートの編集例

２．４．４　　カスタマイズ結果の確認

テンプレートの差し替えが成功しているかを確認する。プロジェクトをビルドし、アプリケーションを起動する。下記にテンプレートファイルの差し替え前と後の結果を示す。

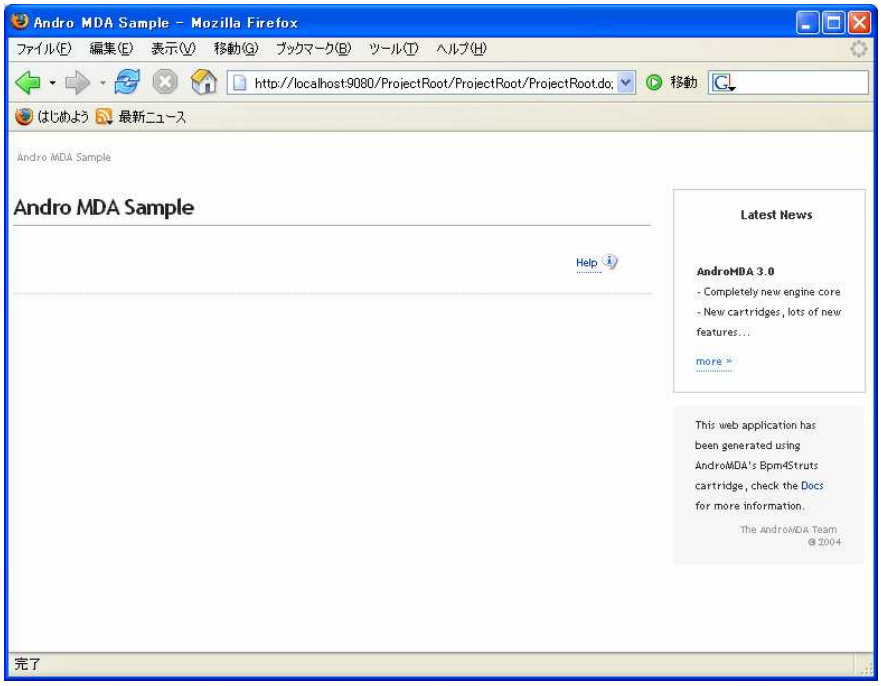


図２－１２．差し替え前

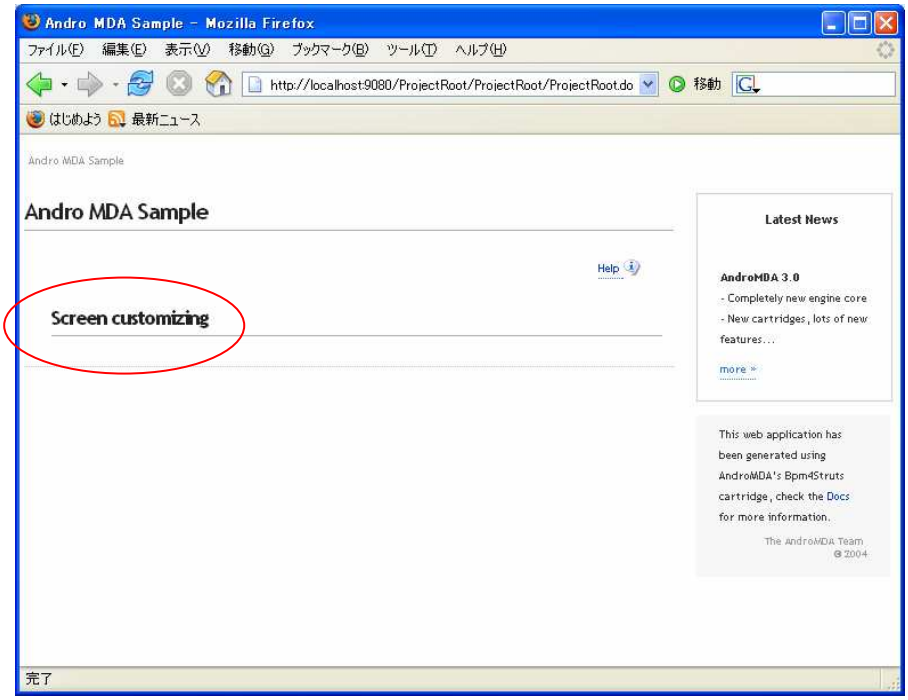


図２－１３．差し替え後

２．５　アプリケーションへファイルのマージ

AndroMDA はプロジェクトのサーバ側(core モジュール)やプレゼンテーション側(web モジュール)など各モジュールの「**src/java**」以下に java ソースファイルを置くとビルド時にコンパイルをし、jar や war ファイルに含めてくれる。

しかし、実際のアプリケーションで AndroMDA が自動生成できないファイルが必要になる場合がある。また、AndroMDA で出力する JSP では Web アプリケーションに適さないレイアウトになる場合がほとんどである。そこで、独自に作成したファイルをアプリケーションに含める、また AndroMDA で出力されるファイルを上書きする必要がある。

本章では、Web アプリケーションに独自の JSP を含める方法を例にとり説明する。

２．５．１　　maven.xml の編集とビルド結果

「**ProjectRoot/web/maven.xml** 」を開き編集する。

```
<!-- copying custom web files over the generated ones (or just adding them) -->
<!-- uncomment this section and put files in src/jsp to have them copied over the generated
ones
<preGoal name="war:init">
   <ant:copy todir="${maven.war.webapp.dir}" overwrite="true">
      <ant:fileset dir="${maven.src.dir}/jsp">
         <include name="**/*"/>
      </ant:fileset>
   </ant:copy>
</preGoal>
-->
```

図２－１４．maven.xml の編集

「**web**」モジュールにある「**maven.xml**」にはあらかじめ war にファイルを含める設定がコメントで記述されている。上記の太字部分のコメントをはずす。

記述内容について簡単に説明すると、war ファイルを作成する Goal が実行される前に、「**${maven.src.dir}/jsp**」にあるディレクトリ以下すべてを上書き許可で「**${maven.war.webapp.dir}**」へコピーする ant の設定になっている。よって、上書きするファイルおよび追加するファイルを「**${maven.src.dir}/jsp**」へ置いておけばファイルの上書き・追加が可能になる。

尚、「**${maven.src.dir}**」は「**ProjectRoot/web/src**」ディレクトリのことで、「**${maven.war.webapp.dir}**」は「**ProejctRoot/web/target/プロジェクト名-web-バージョン.war**」ディレクトリ(AndroMDA で自動生成されるディレクトリ)のことである。

下記に「**${maven.src.dir}/jsp**」へファイルを置いてビルドした結果を示す。ここでは、追加するファイルを **AddFile.jsp** とする。

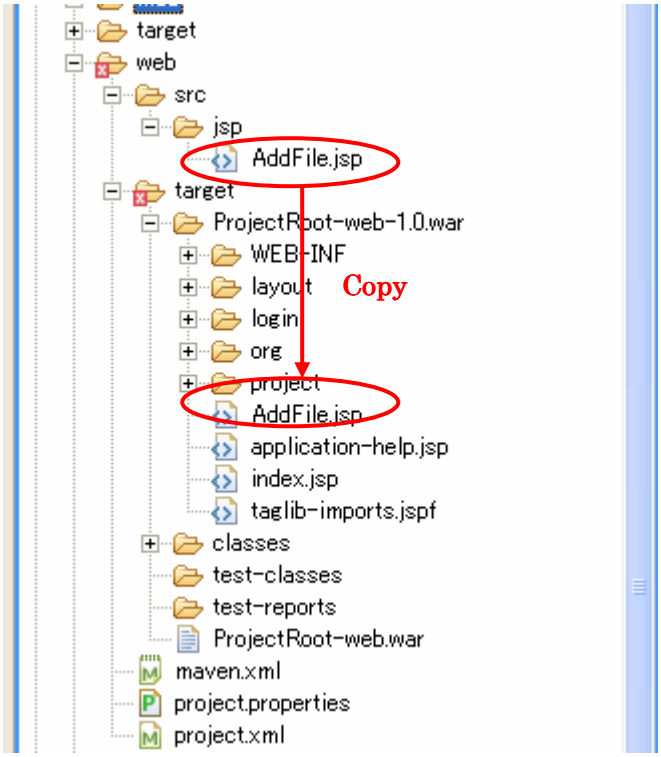


図２－１５．ビルド結果

２．５．２　　その他

上書きや追加するファイルは、何でも良い。コピー時にすでに同一名ファイルがある場合は、上書きされ、ない場合は追加される。

サーバ側(core モジュール)の「maven.xml」にはコピーする設定は記述されていないが、名前を「ejb:ejb」とした preGoal を<project/>タグ内に作成して、そこにコピーする記述をすることによって、サーバ側(core モジュール)でもファイルの上書き、追加が可能となる。例えば、下記のような記述を追加する。

```
<preGoal name="ejb:ejb">
    <ant:copy todir="${maven.build.dest}">
        <ant:fileset dir="${maven.src.dir}/java">
            <ant:include name="**/*.ini"/>
        </ant:fileset>
    </ant:copy>
</preGoal>
```

図２－１６．サーバ側のファイル上書き設定例

「${maven.src.dir}/java」以下にある ini ファイルを「${maven.build.dest}」へコピーする。「 ${maven.src.dir} 」 は 「 ProjectRoot/core/src 」 、 「 ${maven.build.dest} 」 は「ProjectRoot/core/target/src」のことである。

２．６　参照 jar ファイルの追加

カスタマイズする際に、AndroMDA がデフォルトで参照する jar 以外の他の jar を参照している場合、ビルド時に jar を参照、アプリケーション(ear や war)作成時に jar 含める必要がある。ここでは、ビルド時の jar 参照設定方法、アプリケーション作成時の jar の組み込み方法を JavaMail の jar の参照を追加する方法を例にとって説明する。

２．６．１　　ビルド時の参照 jar の追加設定

参照する jar の追加が必要なプロジェクトファイルを編集する。ここでは、サーバ側(core モジュール)で JavaMail を利用してメール送信の実装がされていると仮定する。

「ProjectRoot/core/project.xml」を開き編集する。

```
<dependencies>
 ・・・
   <dependency>
      <groupId>mail</groupId>
      <artifactId>mail</artifactId>
      <version>1.3</version>
      <type>jar</type>
      <properties/>
   </dependency>
 ・・・
</dependencies>
```

図２－１７．ビルド時の参照 jar の追加設定記述例

上記のように<dependencies/>の中に<dependency/>を追加する。各タグの意味は下記の表のとおりである。

表２－６．dependency タグの内容

タグ	意味
groupId	リポジトリ(デフォルトのリポジトリは「C:/Documents and Settings/ユーザ名/.maven/repository」)にあるディレクトリ名。 例：リポジトリ/mail
artifactId	ファイル名の先頭部分。 ファイルは下記の形式になっている。 <artifactId>-<version>.jar 例：mail-1.3.jar
version	ファイル名の後尾部分。 ファイルは下記の形式になっている。 <artifactId>-<version>.jar 例：mail-1.3.jar
type	ファイルのタイプ 指定できる値は「jar」、「pom」、「plugin」、「xml.zip」、「ejb」
properties	必要なプロパティ 例： <properties> <ear.bundle>true</ear.bundle> </properties>

２．６．２　　アプリケーション作成時の jar の含め方

ここでは、生成されるアプリケーションのファイル ear ファイルに jar を追加する方法を説明する。アプリケーション作成時の jar の参照設定が記述されている「ProjectRoot/app/project.xml」を開き編集する。

```
<dependencies>
 ・・・
    <dependency>
        <groupId>mail</groupId>
        <artifactId>mail</artifactId>
        <version>1.3</version>
        <type>jar</type>
        <properties>
            <ear.bundle>true</ear.bundle>
        </properties>
    </dependency>
 ・・・
</dependencies>
```

図２－１８．アプリケーション作成時の参照 jar の追加設定記述例

上記のように<dependencies/>の中に<dependency/>を追加する。各タグの内容は、基本的に２．６．１．の内容と変わらない。唯一異なるのは、<properties/>に<ear.bundle/>が追加されていることぐらいであり、この<ear.bundle/>の値によって ear に含めるか否かを設定できる。

尚、デフォルトは「false」であり、タグを省略すると ear には含まれない。また、war ファイルに対して jar を追加する方法を説明する。war ファイルが作成される「ProjectRoot/web 」の「project.xml」を開き編集するが、編集内容は ear ファイルに含めるときとほぼ同じである。違う点は、<properties/> に <ear.bundle>true</ear.bundle> タグを追加するのではなく <war.bundle>true</war.bundle>タグを追加する点である。これで、war ファイルへ jar を追加することができる。war ファイルへ追加する jar は「WEB-INF/lib」に追加される。

２．７　テンプレートカスタマイズサンプル

ここではテンプレートのカスタマイズを利用した簡単なサンプルアプリケーションについて説明する。サンプルアプリケーションは Web コンテンツの情報漏えい防止を JavaScript やスタイルシートで実現する。

２．７．１　　情報漏えい防止機能の内容

今回のサンプルアプリケーションで防止する機能として下記のものがある。

・ 右クリックの禁止

Html の<body/>タグの「**oncontextmenu**」属性に「**return false**」と設定することによって画面上でマウスの右クリックをしてもポップアップメニューが表示できないようにする。

```
<body oncontextmenu="return false;" >
```

・ 印刷の禁止(印刷はできるが内容が印刷されない)

スタイルシートで body に「**display:none;**」と設定することによって、印刷しても内容が印刷できないようにする。

```
body{
        display:none;
}
```

・ テキストの選択防止(Internet Explorer のみ)

Html の<body/>タグの「**onselectstart**」属性に「**return false**」と設定することによって画面上でマウスをドラッグしてもテキストが選択できないようにする。

```
<body onselectstart="return false;" >
```

・ テキストのコピー防止(Internet Explorer のみ)

Html の<body/>タグの「**onCopy**」属性に「**return false**」と設定することによって画面上でテキストがコピーできないようにする。

```
<body onCopy="return false;" >
```

２．７．２　　プロジェクトの作成・モデリング

新規プロジェクトを作成する。プロジェクト作成時に Web アプリケーションを利用するという質問で yes と答える以外は、自由に指定しても良い。下記はこのガイドでの指定である。

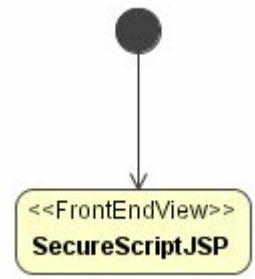
表２－７．プロジェクト作成時の質問内容と回答

質問	回答
名前	Taro
プロジェクト名	Secure JSP
プロジェクト ID	secure_jsp
バージョン	1.0
ベースパッケージ名	secure.jsp
EAR か WAR か？	war
Web サービスを利用するか？	no

MagicDraw でモデルを作成する。このガイドでは、作成するものを war にしたのでプレゼンテーション側のモデルのみ作成する。プレゼンテーション側のコントローラクラスを１つとアクティビティ図に画面をひとつ作成した。

・クラス図(ClassDiagram)-コントローラクラス(SecureJSPController)

・アクティビティ図(ActivityDiagram)-画面(SecureScriptJSP)



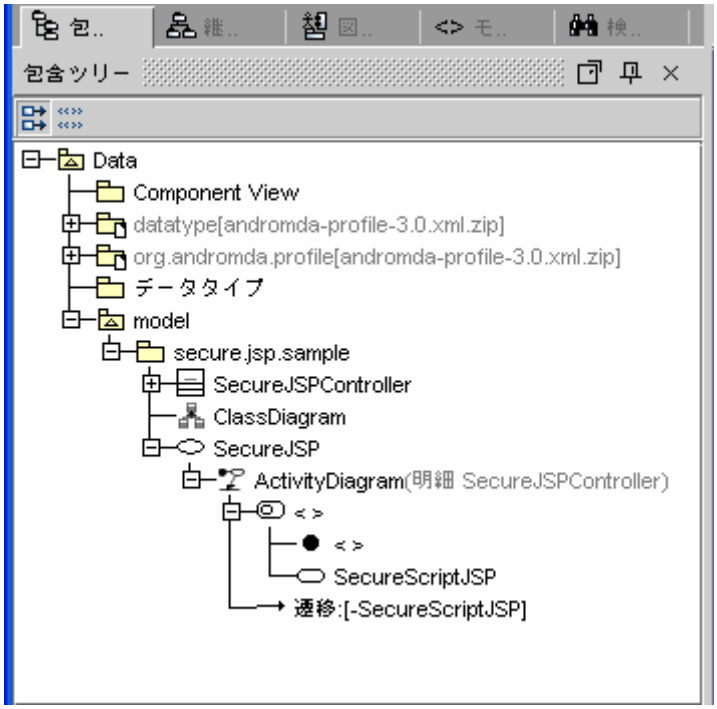


図２－１９．モデル構成

２．７．３　　カスタマイズ

プロジェクトが完成したので、次に情報漏えい防止の機能を追加する。

利用するカスタマイズは、本ガイドの２．４「テンプレートファイルのマージ」と２．５「アプリケーションへファイルのマージ」で説明したカスタマイズ方法を利用する。

- JSP テンプレートファイルの差し替え

  JSP にスタイルシートファイルを読み込む記述を追加したいため、BPM4Struts カートリッジのテンプレートファイルを差し替える。差し替えるテンプレートファイルは、「**templates/bpm4struts/pages/main-layout.jsp.vsl**」ファイルである。本ガイドの２．３の手順どおりに、「**secure_jsp/mda/project.xml**」ファイルを開き、<**mergeLocation/**>に「**custom**」と値を記述する。プロジェクトルートの直下に「**custom**」フォルダを作成してテンプレートファイルをそこへディレクトリごとコピーする。

  パスは「**custom/templates/bpm4struts/pages/main-layout.jsp.vsl**」となる。

  次に今コピーした差し替え用のテンプレートファイルを編集する。

・32,33 行目あたりに太字の部分を追加

```
--%>
<link rel="stylesheet" type="text/css" media="screen"
       href="<html:rewrite page="/layout/default-application.css"/>"></link>
<link rel="stylesheet" type="text/css" media="screen"
       href="<html:rewrite page="/layout/default.css"/>"></link>
<link rel="stylesheet" type="text/css" media="print"
       href="<html:rewrite page="/layout/print_secure.css"/>"></link>
<script type="text/javascript" language="Javascript1.1"
       src="<html:rewrite page="/layout/layout-common.js"/>"></script>
<script type="text/javascript" language="Javascript1.1"
       src="<html:rewrite page="/layout/key-events.js"/>"></script>
<%-- to be uncommented later, when supporting struts-menu
<script type="text/javascript" language="Javascript1.1"
       src="<html:rewrite page="/layout/menu/menu-expandable.js"/>"></script>
--%>
```

図２－２０．テンプレートファイルの編集１

・ 43,44 行目あたりの<body/>タグに太字のように属性を追加

```
<body onselectstart="return false;" oncontextmenu="return false;"
        onCopy="return false;">
    <div id="container">
      <div id="sidebar">
        <tiles>
・・・
```

図２－２１．テンプレートファイルの編集２

- スタイルシートファイルをアプリケーション(war)に追加
  今回印刷の禁止を実現するためにスタイルシートを利用しているが、そのスタイルシートが記述されたファイルをアプリケーションに追加する。
  「secure_jsp/web/maven.xml」を開き、コメントになっている太字部分のコメントをはずす。

```
<!-- copying custom web files over the generated ones (or just adding them) -->
<!-- uncomment this section and put files in src/jsp to have them copied over the
generated ones
    <preGoal name="war:init">
        <ant:copy todir="${maven.war.webapp.dir}" overwrite="true">
            <ant:fileset dir="${maven.src.dir}/jsp">
                <include name="**/*"/>
            </ant:fileset>
        </ant:copy>
    </preGoal>
-->
```

図２－２２．設定ファイルの編集

その後、「secure_jsp/web/src/jsp/layout/print_secure.css」ファイルをディレクトリごと作成する。作成した「print_secure.css」ファイルに下記の内容を記述する。

```
body{
        display:none;
}
```

図２－２３．CSS ファイルの編集

２．７．４　　プロジェクトのビルドと動作確認

プロジェクトをビルドしてアプリケーションを作成する。その後、アプリケーションを起動させる。

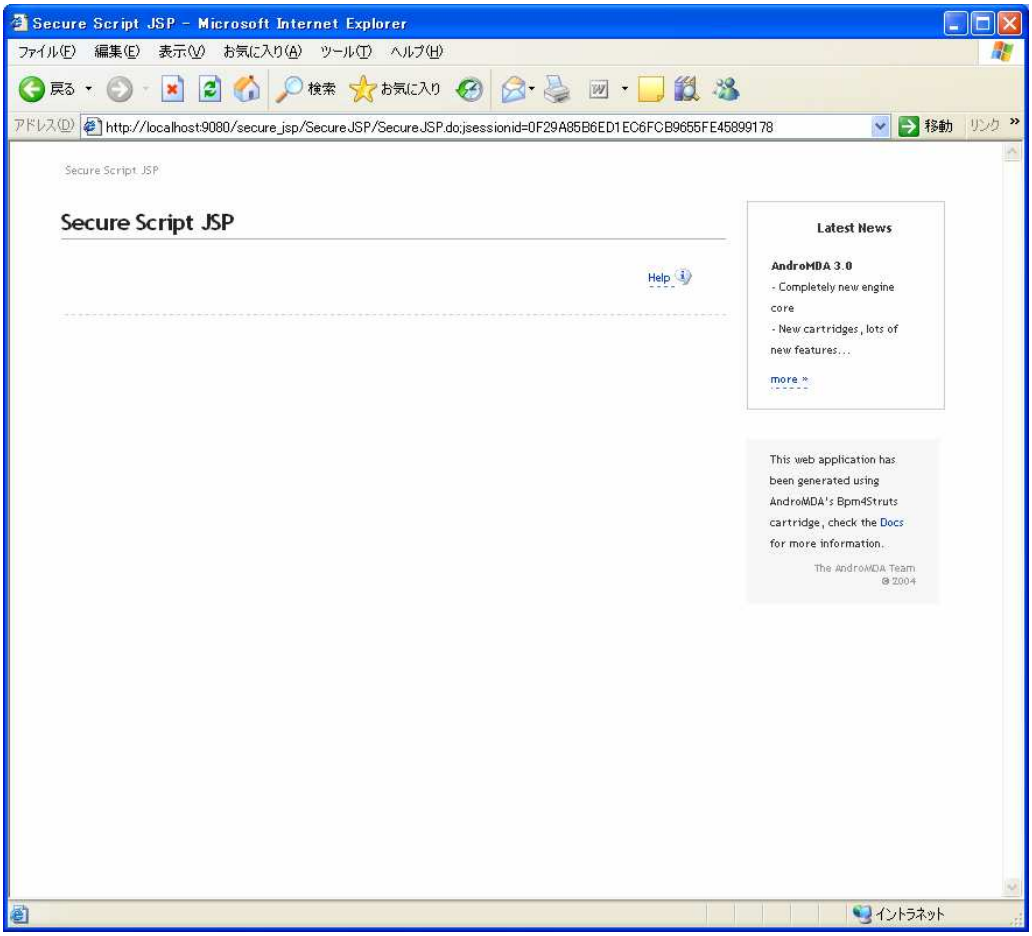


図２－２４．アプリケーション動作結果

表示された画面上で右クリックが不能になり、印刷しても内容が印刷されなくなる。Internet Explorer のみだが、テキストの選択やコピーすることが禁止されている。

３ カートリッジのカスタマイズ

３．１ カートリッジの内容

この章ではカートリッジの内容について記述する。３．１章では既存カートリッジである Spring カートリッジを例にとって説明する。尚、３．１章での「ProjectRoot」は「andromda-spring」のことを指している。

３．１．１　　カートリッジ設定ファイル

カートリッジを動作する上で必要な情報を記述している設定ファイルが４つある。カートリッジプロジェクトに対する設定ファイルが１つ、残りの３つは変換時に利用する設定ファイルである。各カートリッジの適用範囲を下図に示す。

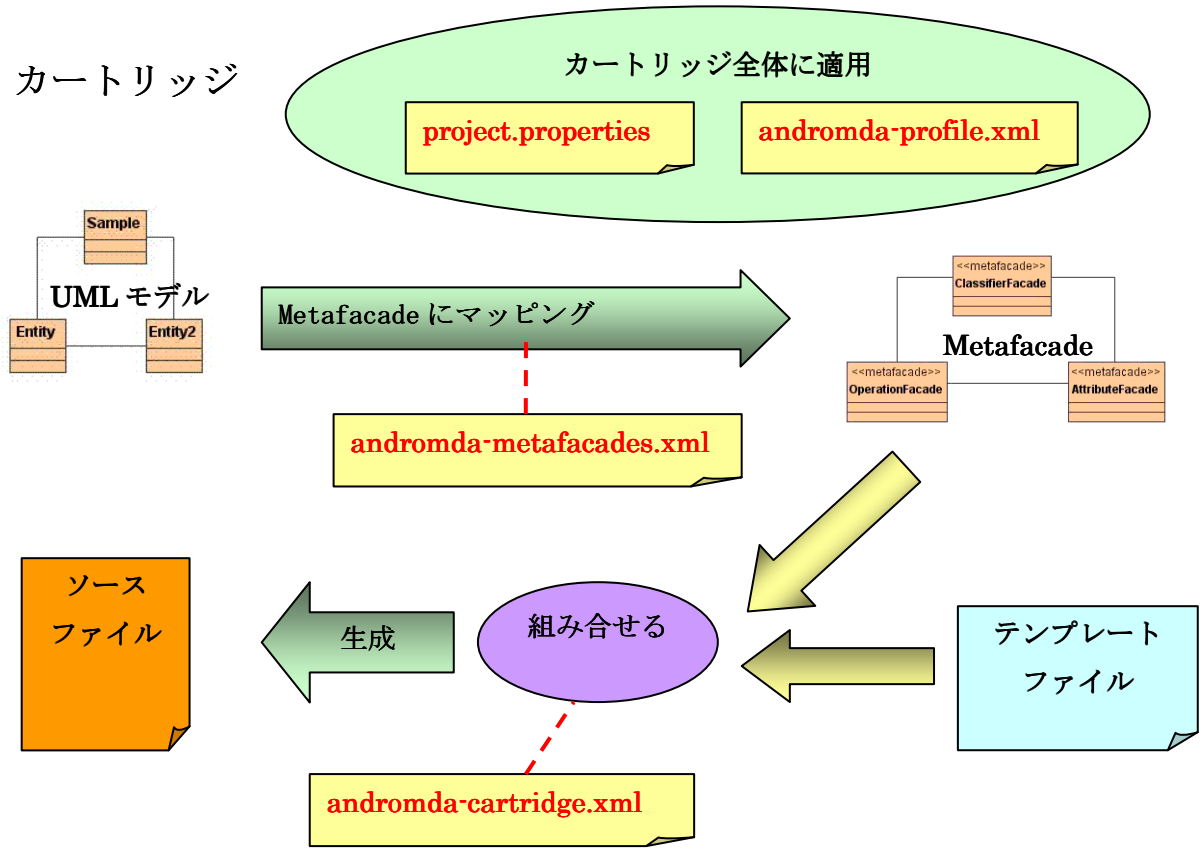


図３－１．各カートリッジ設定ファイルの適用範囲

ここでは、各設定ファイルの記述内容について説明する。

３．１．１．１　project.properties ファイル

「**project.properties**」ファイルは、カートリッジプロジェクトの情報を設定するファイルである。このファイルは、**ProjectRoot** に存在する。下記に記述内容を挙げる。

```
maven.xdoc.includeProjectDocumentation=yes

# ----------------------------------------------------------------
# Cartridge Metafacade Model Properties
# ----------------------------------------------------------------

metafacade.model.file=${maven.src.dir}/uml/SpringMetafacadeModel.xml.zip
maven.andromda.model.uri=jar:file:${metafacade.model.file}!/SpringMetafacadeModel.xml
metafacade.cartridgeFilter=meta

# specify that we don't want to process the package containing the core metafacades
# (since by default it is set to process all of them)
maven.andromda.modelPackage.0.name=org::andromda::metafacades::uml
maven.andromda.modelPackage.0.shouldProcess=false
 ・・・
```

図３－２．project.properties の記述例

この設定ファイルで重要な項目は「**metafacade.model.file**」と「**maven.andromda.model.uri**」である。「**metafacade.model.file**」はカートリッジの UML モデルアーカイブファイルへのパスを設定する。「**${maven.src.dir}**」は「**ProjectRoot/src**」のことを指している。「**maven.andromda.model.uri**」は UML モデルアーカイブファイルに格納されている xml ファイルへの URI を設定する。

３．１．１．２　andromda-cartridge.xml ファイル

この設定ファイルは、カートリッジがファイル生成する際に利用する Metafacade クラスとテンプレートファイルのマッピングやテンプレートから参照するプロパティやオブジェクトの設定をするファイルである。このファイルは、「**ProjectRoot/src/META-INF**」上に存在する。下記に記述例を挙げる。

```
<cartridge name="spring">
  <templateEngine>
    <macrolibrary name="templates/spring/RenderPreconditions.vm"/>
  </templateEngine>
  ・・・
  <templateObject name="stringUtils" className="org.apache.commons.lang.StringUtils"/>
  ・・・
  <property reference="driver" default=""/>
  <property reference="username" default=""/>
  ・・・
  <template
    path="templates/spring/SpringDao.vsl"
    outputPattern="$generatedFile"
    outlet="daos"
    overwrite="true"
    required="false">
    <modelElements variable="entity">
      <modelElement>
        <type name="org.andromda.cartridges.spring.metafacades.SpringEntity"/>
      </modelElement>
    </modelElements>
  </template>
  ・・・
</cartridge>
```

図３－３．andromda-cartridge.xml の記述例

各タグや属性の主な内容は以下のとおりである。

- **cartridge** タグ

  cartridge タグは、この設定ファイルのルートとなるタグである。このカートリッジの名前を設定する。属性は下記のとおりである。

表３－１．cartridge タグの属性

属性	内容	必須	備考
name	カートリッジの名前を指定する	○	

・ templateEngine タグ
テンプレートファイル内で使用されるマクロのファイルを<macrolibrary/>の name 属性で設定する。このタグの省略は可能である。

・ templateObject タグ
すべてのテンプレートファイルで共通的に使用するオブジェクトを設定する。属性は下記のとおりである。

表３－２．templateObject タグの属性

属性	内容	必須	備考
name	テンプレートファイル内で参照する名前	○	テンプレートファイル内では「${name の値}」で参照する。
className	使用するオブジェクトの完全修飾名	○	

・ property タグ
すべてのテンプレートファイル内で参照が可能な値を設定する。尚、このタグは後述するカートリッジを利用するプロジェクトの設定ファイルで値を設定する。属性は下記のとおりである。

表３－３．property タグの属性

属性	内容	必須	備考
reference	テンプレート内で参照する名前	○	テンプレートファイル内では「${reference の値}」で参照する。
default	プロパティ値が設定されていない場合のデフォルト値	○	

・ template タグ

このタグは生成するファイルのテンプレートファイルについて設定する。属性は下記のとおりである。

表３－４．template タグの属性

属性	内容	必須	備考
path	テンプレートファイルのパス	○	
outputPattern	生成ファイルのファイル名パターン	○	
outlet	生成ファイルの出力先	○	このプロパティの値が空の場合は対象のテンプレートファイルを使用したファイル生成を行わない
generateEmptyFiles	テンプレートファイルで生成するファイルがない場合に空のファイルを出力するかの有無		・デフォルトは false ・テンプレートファイルで生成するファイルがない場合とは、テンプレートファイルが参照する Metafacade クラスが存在しない(モデルで作成されていない)場合
overwrite	生成ファイルがすでに存在していた場合に上書きするかの有無	○	true(上書き)/false(上書きしない)で設定する
required	生成ファイルの出力先が設定されていない場合の警告の有無		true(警告する)/false(警告しない)で設定する(デフォルトは true)

・ **modelElements** タグ

<template/>の中に設定する。このタグにテンプレートファイル内で参照する Metafacade クラスの設定をする。属性は下記のとおりである。

表３－５．modelElements タグの属性

属性	内容	必須	備考
variable	テンプレートファイル内で参照する Metafacade クラスの参照名		テンプレートファイル内では「${variable の値}」で参照する。

・ **modelElement** タグ

テンプレートファイル内で参照する Metafacade クラスの設定をする。このタグは<modelElements/>内に設定する必要がある。

・ **type** タグ

このタグはテンプレートファイル内で参照する Metafacade クラスの設定をする。属性は下記のとおりである。

表３－６．type タグの属性

属性	内容	必須	備考
name	参照する Metafacade クラスの完全修飾名	○	

３．１．１．３　andromda-metafacades.xml ファイル

この設定ファイルはカートリッジがファイル生成をする際に利用する Metafacade クラスと metaclass とのマッピング情報や使用するプロパティを設定するファイルである。下記に記述例を挙げる。

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- contains the hibernate cartridge metafacade mappings -->
<metafacades namespace="spring">
    <property reference="daoNamePattern" default="{0}Dao"/>
    <property reference="daoBaseNamePattern" default="{0}DaoBase"/>
・・・
    <metafacade class="org.andromda.cartridges.spring.metafacades.SpringEntityLogicImpl"
contextRoot="true">
        <mapping class="org.omg.uml.foundation.core.UmlClass$Impl">
            <stereotype>ENTITY</stereotype>
        </mapping>
    </metafacade>
    <metafacade
class="org.andromda.cartridges.spring.metafacades.SpringEntityOperationLogicImpl">
        <mapping class="org.omg.uml.foundation.core.Operation$Impl">
            <context>org.andromda.cartridges.spring.metafacades.SpringEntity</context>
        </mapping>
        <property reference="implementationOperationNamePattern" default="handle{0}"/>
    </metafacade>
・・・
</metafacades>
```

図３－４．andromda-metafacades.xml 記述例

各タグや属性の主な内容は以下のとおりである。

- **metafacades** タグ

  metafacades タグは、この設定ファイルのルートとなるタグである。この Metafacade 構成の名前を設定する。属性は下記のとおりである。

表３－７．metafacades タグの属性

属性	内容	必須	備考
name	この Metafacade 構成の名前	○	

・ property タグ

すべての Metafacade クラスが持つメソッドで取得が可能な値を設定する。尚、このタグは後述するカートリッジを利用するプロジェクトの設定ファイルで値を設定する。属性は下記のとおりである。

表３－８．property タグの属性

属性	内容	必須	備考
reference	メソッドで取得する際のキー	○	getConfiguredProperty(reference)で取得できる
default	プロパティ値が設定されていない場合のデフォルト値		

・ metafacade タグ

Metafacade クラスを構成するタグ、タグ内部に属性やタグで Metafacade クラスの詳細を設定する。属性は下記のとおりである。

表３－９．metafacade タグの属性

属性	内容	必須	備考
class	生成する Metafacade クラスの実装クラス完全修飾名	○	Metafacade クラス名の後ろに「LogicImpl」がつく
contextRoot	ルートとなる Metafacade の有無		・デフォルトは「false」 ・３．１．１．２の andromda-cartridge.xml にあるテンプレートで参照する Metafacade に指定するクラスが「true」となる

・ mapping タグ

このタグは<metafacade/>の中で使用する。<metafacade/>で宣言した Metafacade クラスがどの UML モデルの metaclass とマッピングするかを設定している。属性は下記のとおりである。

表３－１０．mapping タグの属性

属性	内容	必須	備考
class	マッピングする metaclass の完全修飾名	○	後ろについている「$Impl」は対象の metaclass の実装クラスを指している

・ stereotype タグ

このタグは<mapping/>の中で使用する。<mapping/>で Metafacade クラスにマッピングする metaclass を設定するが、このタグによってマッピングする metaclass に条件をつけることができる。このタグには属性はなく値だけである。値に設定したステレオタイプを持つ metaclass のみ Metafacade クラスへマッピングするように条件付ける。尚、このタグは<context/>と併用可能である。

・ context タグ

このタグは<mapping/>の中で使用する。このタグはある Metafacade クラスを生成する際に一緒に生成するように設定するためのタグである。値には Metafacade クラスの完全修飾名を設定する。記述例で説明すると、Metafacade クラス「SpringEntityLogicImpl」が生成される際に、Metafacade クラス「SpringEntityOperationLogicImpl」も生成される。尚、このタグは<stereotype/>と併用可能である。

・ property タグ

このタグは<metafacade/>の中で使用する。属性の内容は、前述で説明している<property/>タグと同じである。ただし、あちらは andromda-metafacades.xml で設定するすべての Metafacade クラス全体のプロパティだが、こちらは<metafacade/>で宣言した Metafacade クラスでのみ使用できるプロパティである。

３．１．１．４　andromda-profile.xml ファイル

カートリッジ内で使用されるステレオタイプやタグ付き値、その他の値を別名にマッピングするための設定ファイルである。下記に記述例を挙げる。

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!--
  The default Spring profile mappings.
-->
<mappings name="spring">
・・・
    <mapping>
        <from>EJB_TRANSACTION_TYPE</from>
        <to>@andromda.ejb.transaction.type</to>
    </mapping>
    <mapping>
        <from>LESS_THAN_COMPARATOR</from>
        <to>less</to>
    </mapping>
・・・
</mappings>
```

図３－５．andromda-profile.xml 記述例

各タグや属性の主な内容は以下のとおりである。

・ **mappings** タグ

mappings タグは、この設定ファイルのルートとなるタグである。このマッピング構成の名前を設定する。尚、３．１．１．２の andromda-cartridge.xml や３．１．１．３の andromda-metafacades.xml でのルートタグの属性はすべて同一の値にする必要がある。属性は下記のとおりである。

表３－１１．mappings タグの属性

属性	内容	必須	備考
name	このマッピング構成の名前	○	

・ **mapping** タグ

このタグは 1 組のマッピングを表すタグである。属性は持たず内部に<from/>と<to/>の組を持つ。<from/>には参照する値を設定し、<to/>に実際の値を設定する。Metafacade クラス内で参照する場合は、org.andromda.core.common.Profile クラスを生成(instance メソッド)し、get(<from/>の値)で<to/>の値を取得できる。

３．１．２　　Metafacade クラス

Metafacade クラスとは､UML のモデルに対するアクセス手段を提供する Facade クラスである。後述するテンプレートファイルはこの Metafacade クラスを参照し、ファイルを生成する。metaclass と Metafacade の関係は UML でモデリングする。その UML モデルは「**ProjectRoot/src/uml**」の下に置かれている。下記に metaclass と Metafacade クラスの関連例を挙げる。

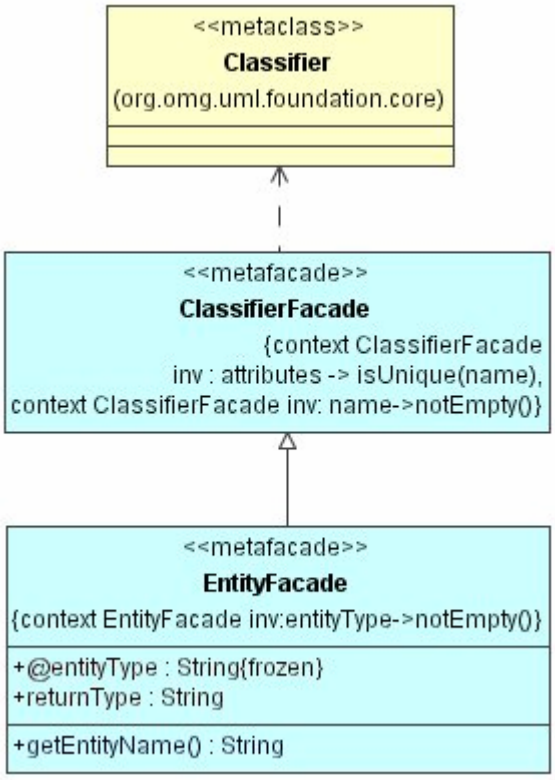


図３－６．metaclass と Metafacade クラスの関連モデル

ステレオタイプ<<metaclass>>がついているのが UML の metaclass である。ステレオタイプ<<metafacade>>がついているのが Metafacade クラスである。Metafacade クラスは、metaclass の Façade クラスではあるが、実際は基本的に metaclass の Getter メソッドをデレゲートし、必要であれば独自のインタフェースを持つ。Metfacade クラスは metaclass に対して依存の線(図の破線矢印)で結ぶことによって対象の metaclass の Façade クラスとなる。

Metafacade クラスに記述されている「**context ・・・**」は制約(OCL)のことである。これは、Metafacade クラスを生成する際にモデルが Metafacade クラスのルールに従っているかを判定する制約である。例えば、図３－６．中の「**EntityFacade**」の「**context EntityFacade inv:entityType->notEmpty()**」は、「**EntityFacade の entityType が Empty でないこと**」を表している。よって Metafacade クラス生成時に「**entityType**」が Empty ならばエラーとなる。

基本的に AndroMDA によって主要な metaclass の Façade である Metafacade クラスは用意されている。そのため、新たな Metafacade を作成する場合は、AndroMDA が用意している Metafacade クラスを継承して作成する(ClassifierFacade を EntityFacade が継承しているように)だけでよい。もし、AndroMDA が用意している Metafacade クラスにない metaclass の Façade が必要の場合に新規に作成する。

図３－６．中の EntityFacade から生成された Metafacade クラスは、下図のようになる。

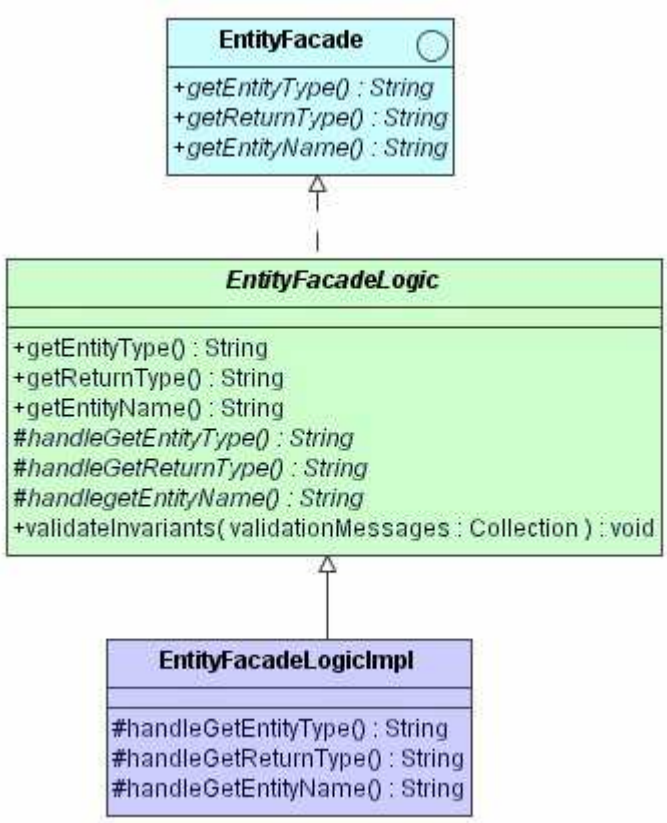


図３－７．生成された Metafacade クラスのモデル

Metafacade クラスからは、インタフェース、抽象クラス、実装クラスの３つが生成される。それぞれのクラス名は Metafacade クラスの名前をもとに、インタフェースは同じ、抽象クラスは名前の後ろに「**Logic**」が付加される、実装クラスは名前の後ろに「**LogicImpl**」が付加される。

属性・メソッド・制約(OCL)がどのような形で Metafacade クラスに生成されるのかを図３－６．をもとに説明する。

- **属性**

  属性は、Getter が生成される。実際の Getter の内容は実装クラスに頭文字に「**handle**」がついたメソッドの中身に実装する必要がある。尚、Metafacade の「**entityType**」には「**@**」や「**{frozen}**」とついているが、これは属性の可変性を表しており、「**frozen**」を指定すると、１度目の呼び出し時に定まった値を２度目以降は戻すような処理が生成される。

- **メソッド**

  メソッドは、定義したまま生成される。実際の内容は属性と同様、実装クラスに頭文字に「**handle**」がついたメソッドの中身に実装する必要がある。

- **制約(OCL)**

  制約(OCL)は、抽象クラスに生成メソッド「**validateInvariants**」の中で Java のプログラムに変換される。プログラムへの変換は AndroMDA が処理する。

ここでは説明のために図３－６．のようなモデルを例にしたが、実例として Spring カートリッジで利用している Entity の Metafacade クラス群のモデルを下図に表す。尚、Spring カートリッジの UML モデルは「**ProjectRoot /src/uml/SpringMetafacadeModel.xml.zip**」である。

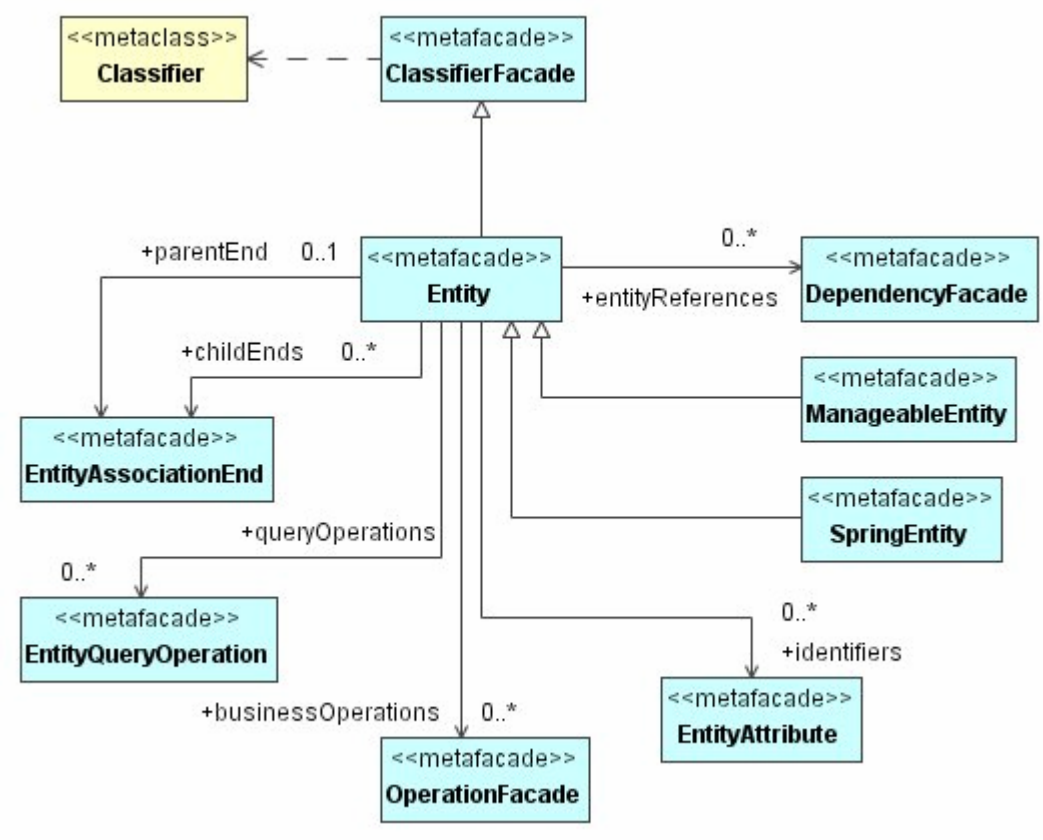


図３－８．Entity の Metafacade クラス群のモデル

図３－８．は Entity の Metafacade クラス群のモデルである。尚、この図では属性、メソッド、制約などを省略している。

３．１．３　　テンプレートファイル

テンプレートファイルとは、その名のとおりカートリッジが生成するファイルのテンプレートとなるファイルである。どのようなファイルを生成するときにどのテンプレートを使用するかについては　３．１．１３．１．１．２で説明した「**andromda-cartridge.xml**」ファイルに設定する。尚、既存の AndroMDA ではテンプレートエンジンとして Velocity を使用しているため、テンプレートファイルは Velocity のルールに従って記述する必要がある。下記にテンプレートファイルの一部を載せる。

```
 1:// Attention: Generated code! Do not modify by hand!
 2:// Generated by: SpringDao.vsl in andromda-spring-cartridge.
 3://
 4:#if ($stringUtils.isNotBlank($entity.packageName))
 5:package $entity.packageName;
 6:#end
 7:#set ($superclass = $entity.generalization)
 8:/**
 9: * @see $entity.fullyQualifiedEntityName
10: */
11:public interface $entity.daoName
12:#if($superclass)
13:    extends $superclass.fullyQualifiedDaoName
14:#end
15:{
16:・・・
17:#if (!$entity.abstract)
18:    /**
19:     * Creates an instance of $entity.fullyQualifiedEntityName and adds it to the
persistent store.
20:     */
21:    public $entity.root.fullyQualifiedEntityName
22:     create($entity.fullyQualifiedEntityName $argumentName);
23:・・・
```

図３－９．テンプレートファイルの記述例

Velocity についてはここでは詳細は述べない(Web サイトに多数の解説サイトが存在するため、そちらを参照していただきたい)。「**$**」で始まっている単語が外部から与えている変数であ

る。カートリッジでは設定ファイルのプロパティやMetafacadeクラスなどがそれにあたる。尚、単語の後に「.」続けて単語があるものは、対象クラスの変数やメソッド呼び出しを表している。

例えば、図中11行目は「**$entity.daoName**」を変数「**entity**」でマッピングしているクラスの「**daoName**」変数を参照し(実際はgetDaoNameというGetterを利用する)値を取得する。ここでは「**entity.daoName**」の値を「**SampleDao**」と仮定すると11行目は下記のようになる。

```
11:public interface SampleDao
```

テンプレートファイルはカートリッジで用意してあるものだが、カートリッジを利用するプロジェクトでカートリッジのテンプレートファイルを上書き(差し替え)することも可能である。この方法については２章のテンプレートのカスタマイズを参照していただきたい。

３．１．４　　カートリッジテスト

カートリッジテストとは、カートリッジのビルド時にカートリッジのファイル生成処理の妥当性をテストするためのものである。そのため、実際にプロジェクトでカートリッジを利用する時のようなUMLモデルを作成し、そのUMLモデルから生成するファイルが期待している出力ファイルと同一かどうかを判定する。カートリッジのテストは、必須ではなく省略することが可能である。

カートリッジテストで必要なものは「**テスト結果を作成するための UML モデル**」、「**期待される出力結果のアーカイブファイル**」、「**テスト実行時の設定情報を記述する設定ファイル**」の３つである。

- **テスト結果を作成するための UML モデル**
  UMLモデルは、カートリッジが生成するファイルのすべてを網羅するようなモデルを作成する。この UML モデルのファイルは「**ProjectRoot/src/test/uml**」以下に置き、ファイル名は任意である。
- **期待される出力結果のアーカイブファイル**
  期待される出力結果のアーカイブファイルは、テスト対象の UML モデルをもとにカートリッジで生成したときに生成されるだろうファイルをアーカイブにしたものを使用する。このアーカイブファイルは「**ProjectRoot/src/test/expected**」以下にファイル名「**cartridge-output.zip**」で置く。
- **テスト実行時の設定情報を記述する設定ファイル**
  テスト実行時の設定情報を記述する設定ファイルは、３．１．１３．１．１．１で説明したプロジェクトの設定ファイルと同じ ProjectRoot 上にある「**project.properties**」のことである。下記に記述例を挙げる。

```
maven.xdoc.includeProjectDocumentation=yes
 ・・・
# -------------------------------------------------------------------
# Cartridge Test Properties
# -------------------------------------------------------------------

# define the test model
test.model.file=${maven.src.dir}/test/uml/SpringCartridgeTestModel.xml.zip
andromda.cartridge.test.model.uri=jar:file:${test.model.file}!/SpringCartridgeTestModel.x
ml

test.output.dir=${andromda.cartridge.test.actual.dir}

# Define the properties of this cartridge to test
andromda.cartridge.test.property.0=languageMappingsUri
andromda.cartridge.test.property.languageMappingsUri=Java
andromda.cartridge.test.property.1=wrapperMappingsUri
andromda.cartridge.test.property.wrapperMappingsUri=JavaWrapper
andromda.cartridge.test.property.2=jdbcMappingsUri
andromda.cartridge.test.property.jdbcMappingsUri=JDBC
```

図３－１０．カートリッジテストの設定記述例

記述内容については下記の表のとおりである。

表３－１２．カートリッジテストの設定項目

キー名	値
test.model.file	カートリッジテストのモデルファイルへのパス
andromda.cartridge.test.model.uri	カートリッジテストのモデルファイルのURI
test.output.dir	テストモデルからの生成物出力先
andromda.cartridge.test.property.数値	・カートリッジが参照するプロパティの名前 ・数値部分を 0 からつける ・この値が次の項目の文字列部分になる
andromda.cartridge.test.property.文字列	・カートリッジが参照するプロパティの値 ・前の項目の値が文字列部分となる

設定項目のうち「andromda.cartridge.test.property.数値」、「andromda.cartridge.test.property.文字列」項目についてはもう少し説明をする。この二つの項目は２つで 1 組である。この組は３．１．１で説明した３つの xml ファイルのうち「andromda-cartridge.xml」、「andromda-metafacades.xml」に設定する<property/>の内容を記述する。つまり、<property/>の「**reference**」属性の値を「andromda.cartridge.test.property.数値」の値に設定し、その値を「andromda.cartridge.test.property.文字列」の値に設定する。

例えば「**reference**」属性に「**username**」という<property/>があり、その値が「**andromda**」だったとする。この場合に設定ファイルに記述すると、「andromda.cartridge.test.property.数値」は「andromda.cartridge.test.property.0=**reference**」、「andromda.cartridge.test.property.文字列」は「andromda.cartridge.test.property.**reference**= **andromda**」となる。

これは、後述する３．１．５のカートリッジを利用するプロジェクトで設定する設定ファイルに記述する内容と表現は異なるが同じことである。

３．１．５　　カートリッジを利用するプロジェクトの設定

ここでは、プロジェクトで実際にカートリッジを利用する際に設定するファイルの記述内容について説明する。

カートリッジの情報を記述するプロジェクトのファイルは、カートリッジを利用するプロジェクトのルートフォルダの下にある「**mda/project.xml**」ファイルである。下記に記述例を挙げる。

```
<?xml version="1.0" encoding="UTF-8"?>

<project>
    <extend>../project.xml</extend>
    <name>Sample Test</name>
    <artifactId>sampletest</artifactId>
    <shortDescription>${pom.name} Component</shortDescription>
    <description>Contains the ${pom.name} module</description>
    <dependencies>
・・・
        <dependency>
            <groupId>andromda</groupId>
            <artifactId>andromda-spring-cartridge</artifactId>
            <version>${andromda.version}</version>
            <type>jar</type>
            <properties>
                <dataSource>${dataSource}</dataSource>
                <hibernateDialect>${hibernate.db.dialect}</hibernateDialect>
                <hibernateShowSql>${hibernate.db.showSql}</hibernateShowSql>
                <hibernateUseQueryCache>false</hibernateUseQueryCache>
                 ・・・
            </properties>
        </dependency>
・・・
    </dependencies>
</project>
```

図３－１１．カートリッジの利用設定記述例

様々なタグが記述されているが、カートリッジについての設定は太字の<dependency/>部分である。下記に各タグの内容を記す。

表３－１３．カートリッジ設定のタグ内容

名前	内容
groupId	カートリッジがあるリポジトリのフォルダ名
artifactId	カートリッジのファイル名前の先頭
version	カートリッジのバージョン
type	カートリッジのファイルタイプ
properties	カートリッジで使用する各種プロパティ設定

<groupId/>に設定する値は、Maven のリポジトリフォルダ直下にあるフォルダ名を設定する。尚、特に変更していなければリポジトリのルートフォルダは「**C:/Documents and Settings/ログインユーザ名/.maven/repository**」である。

カートリッジのファイル名は、タグ<artifactId/>と<version/>の値を使って「**<artifactId/>-<version/>**」がファイル名になる。

<properties/> 内に記述するタグ名が「**andromda-cartridge.xml**」や「**andromda-metafacades.xml**」に記述される<property/>の属性「**reference**」とマッチする。そして、そのタグの値が property の値となる。この設定ファイルで設定しないプロパティについては、各設定ファイルの<property/>の属性「**default**」の値が使用される。

３．２ カートリッジ作成例

この章では簡単なカートリッジサンプルとして「POJO カートリッジ」を作る方法を説明する。POJO カートリッジは、モデルの<<Entity>>ステレオタイプを持つクラスを元に Java クラス（POJO）を生成するカートリッジである。手順は以下のようになる。

① **カートリッジプロジェクトの新規作成**
② **Metafacade モデルの作成**
③ **MetafacadeImpl クラスのコードの実装**
④ **テンプレートの作成**
⑤ **設定ファイルの編集**
⑥ **POJO カートリッジの使用**
⑦ **カートリッジのテスト**

カートリッジのテストファイルを手作業で作成するのは現実的ではない。そのため、一度テストモデルからコードを生成し、そのコードをテストコードとして使う。このため、本来ならテストが⑥番にくるはずだが、⑦番になっている。

３．２．１　　カートリッジプロジェクトの新規作成

カートリッジプロジェクトを新規に作成する。

・ 「andromda-src-3.0/cartridges /andromda-meta フォルダ」を「andromda-src-3.0/cartridges」へコピーし、リネームする。ここでは、「andromda-pojo」と命名する。今後３．２章では、この andromda-pojo フォルダを「ProjectRoot」とする。

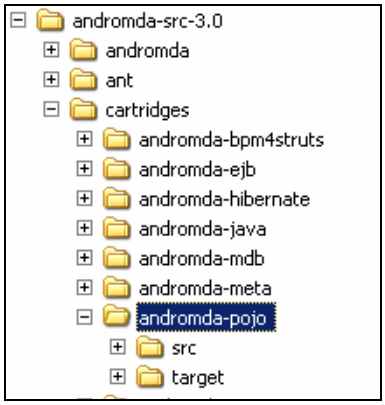


図３－１２．　カートリッジの配置フォルダ

・ 「ProjectRoot/project.xml」を編集する。編集する要素はカートリッジ名（<artifactId>要素）、プロジェクト名、開発者名である。「project.xml」の編集箇所を太字で示す。

```
<project>
    <extend>../project.xml</extend>
    <artifactId>andromda-pojo-cartridge</artifactId>
    <name>AndroMDA Pojo Cartridge</name>
```

```
    <shortDescription>Pojo Cartridge</shortDescription>
    <description>
                Produces metafacades.
    </description>
<issueTrackingUrl>${issue.tracking.location}/secure/BrowseProject.jspa?id=10050
</issueTrackingUrl>
    <developers>
        <developer>
            <name>Koji Iida</name>
            <id>iida</id>
            <email></email>
            <roles>
                <role>Developer</role>
            </roles>
        <!-- 省略 -->
```

図３－１３． projext.xml の記述例

・ 「ProjectRoot/xdocs」フォルダは必要ないので削除する（削除しなくても問題は無い）。「ProjectRoot/target/cartridge-test/actual」フォルダと「ProjectRoot/target/cartridge-test/expected」フォルダ以下は削除する（３．２．７で、新規に作成する）。

・ 確認のためここでビルドをする。ビルドする場合はコマンドプロンプトを起動し、ProjectRoot にカレントディレクトリを移し、maven を実行する。

```
> maven
```

ビルドに成功すると以下の結果が出力される。

```
～省略～
INFO  [App] BUILD SUCCESSFUL
INFO  [App] Total time: 30 seconds
INFO  [App] Finished at: Mon Dec 05 17:23:00 JST 2005
INFO  [App]
```

以上で新規カートリッジプロジェクト「andromda-pojo-cartridge」の雛形の作成が完了した。

３．２．２　　Metafacade モデルの作成

AndroMDA は、カートリッジをビルドすると Metafacade モデルから Metafacade クラスのスケルトンを生成する。Metafacade クラスは、使い易くするために NetBeans MDR パッケージのクラスをラップするクラスである。この章では Metafacade モデルを作成する。具体的に Metafacade モデルを作成する手順を示す。また、生成したスケルトンクラスの実装は３．２．３で行う。

・ 「**ProjectRoot/project.properties**」を開き以下のように編集する。編集する箇所を太字で示す。

```
#省略
metafacade.model.file=${maven.src.dir}/uml/PojoMetafacadeModel.xml.zip
maven.andromda.model.uri=jar:file:${metafacade.model.file}!/PojoMetafacadeModel.xml
#省略
#test.model.file=${maven.src.dir}/test/uml/MetaCartridgeTestModel.xml.zip
#andromda.cartridge.test.model.uri=jar:file:${test.model.file}!/MetaCartridgeTestMo
del.xml
```

図３－１４．　project.properties

・ モデルファイルの名前を変更。「**ProjectRoot/src/uml/MetaMetafacadeModel.xml.zip**」を「**PojoMetafacadeModel.xml.zip**」にリネームする。そして、Magic Draw でこのモデルを開く。開く時に、「**UMLMetaFacadeModel-3.0.xml.zip**」が必要になるので、「**mavan リポジトリ/andromda/xml.zips/UMLMetaFacadeModel-3.0.xml.zip**」を指定する。

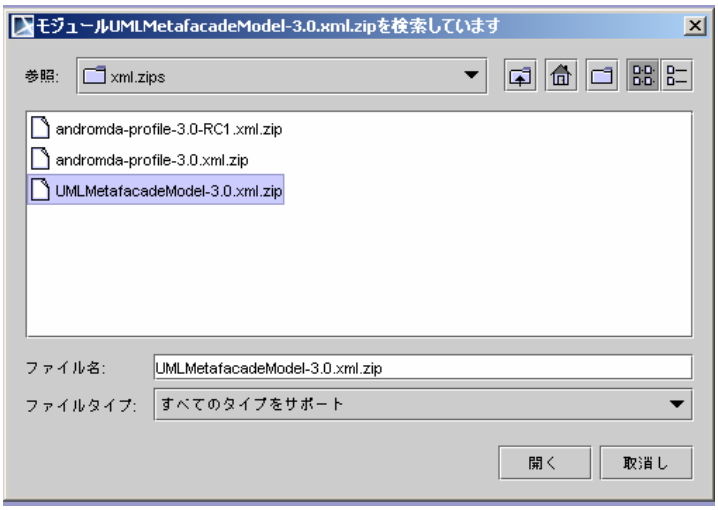


図３－１５．　UMLMetafacadeModel-3.0.xml.zip の指定

ここまで完了すると、Magic Draw にモデルが展開される。

・ モデルを展開したら、meta パッケージは必要ないので削除する。

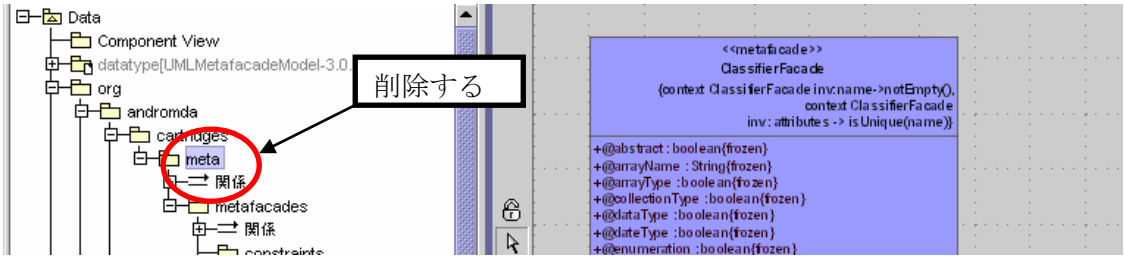


図３－１６． meta パッケージの削除

・ pojo.metafacades パッケージを作成する。（パッケージを作成するには cartridges パッケージを右クリックし、「新規エレメント」→「モデルパッケージ」を選択する）

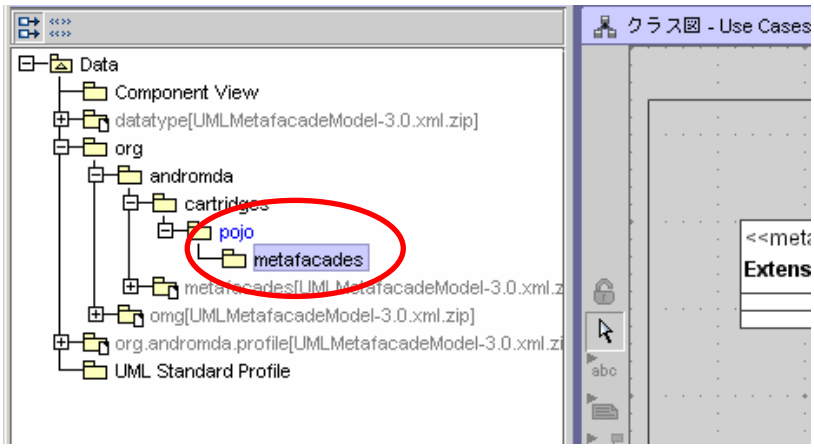


図３－１７．pojo.metafacade パッケージの作成

metafacades パッケージ内で独自のクラス図（MetaFacade クラス）を作成する。

・ metafacades パッケージに pojo クラス図を作成し、PojoEntity クラスと PojoAssociationEnd クラスを作成する。２つのクラスには<<metafacade>>ステレオタイプをつける。必ず<<metafacade>>をつけなければならない。

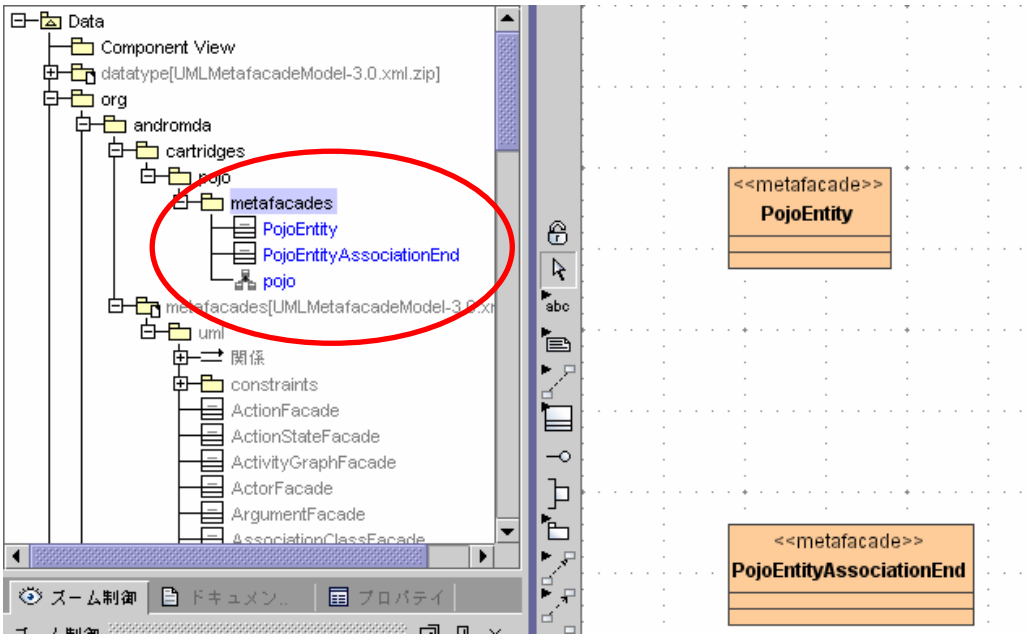


図３－１８． PojoEntity と PojoEntityAssociationEnd の作成

・ インポートした metafacades[UMLMetafacadeModel-3.0.xml.zip].uml.Entity クラスと EntityAssocationEnd クラスを pojo クラス図に表示させる（ドラッグする）。そして、下図のように継承関係を設定する。

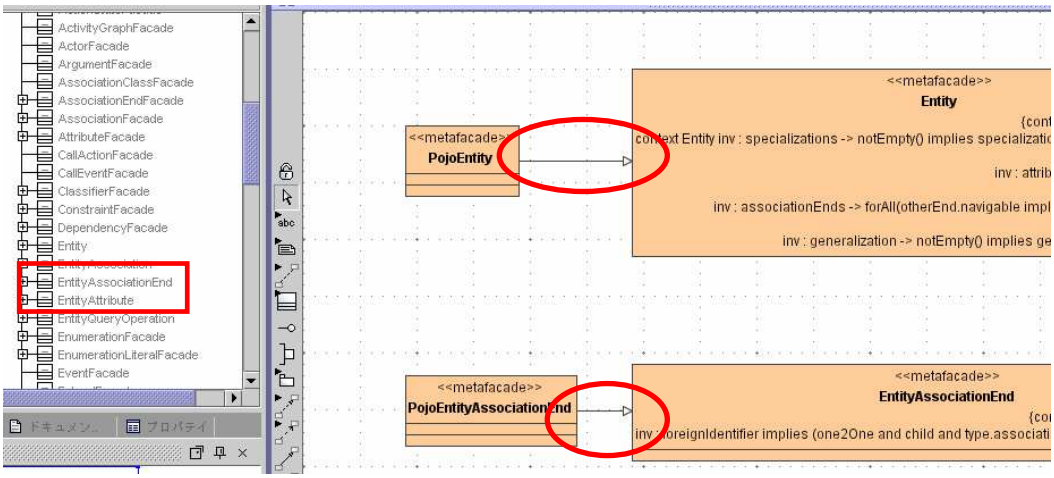


図３－１９． 継承関係の設定

・ PojoEntityAssocationEnd クラスに下図のよう getMultiplicityLower() メソッドと getMulitiplicityUpper メソッドを記述する。２つのメソッドは関連の多重度を取得するメソッドである。

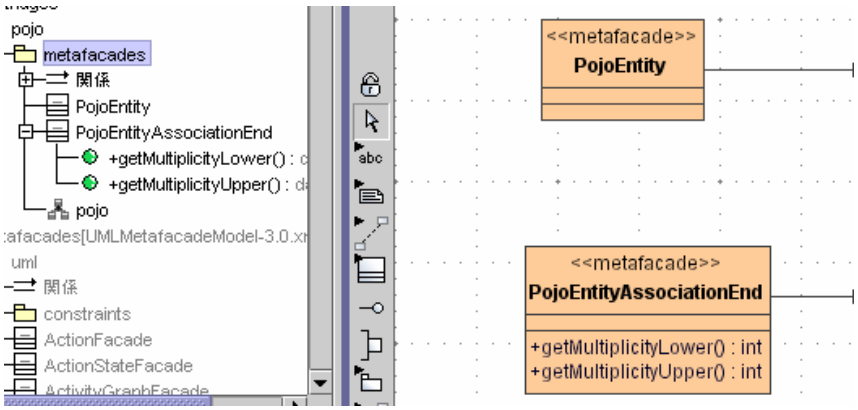


図３－２０． メソッドの記述

・ 確認のためここでビルドをする。ビルドする場合はコマンドプロンプトを起動し、**ProjectRoot** にカレントディレクトリを移し、maven を実行する。

```
> maven
```

ビルドに成功すると以下の結果が出力される。

```
～省略～
INFO  [App] BUILD SUCCESSFUL
INFO  [App] Total time: 22 seconds
INFO  [App] Finished at: Tue Dec 06 13:49:54 JST 2005
INFO  [App]
```

ビルドが完了すると、モデル上の 2 つの Metafacade クラスから 3 種類の Java コード（インタフェース、抽象クラス、具象クラス）が生成される。また、それらの生成されたファイルは「**org.andromda.cartridges.pojo.metafacades**」**パッケージ**に配置される。これで Metafacade モデリングが完了した。

３．２．３　　MetafacadeImpl クラスのコードの実装

この章では、３．２．２で生成した Java コードのうち具象クラス（＊＊＊Impl.java）に実装コードを記述する。実装するクラスは３．２．２でモデリングした PojoEntityAssociation クラスから生成された具象クラス PojoEntityAssociationEndLogicImpl である。

・ PojoEntityAssociationEndLogicImpl に下記の通り実装コードを記述する。記述する部分を太字で示す。

```
package org.andromda.cartridges.pojo.metafacades;
import java.util.Iterator;
import org.omg.uml.foundation.core.AssociationEnd;
import org.omg.uml.foundation.datatypes.Multiplicity;
import org.omg.uml.foundation.datatypes.MultiplicityRange;
public class PojoEntityAssociationEndLogicImpl
    extends PojoEntityAssociationEndLogic
{
    private  AssociationEnd associationEnd;
    public PojoEntityAssociationEndLogicImpl (Object metaObject, String context){
        super (metaObject, context);
        this.associationEnd = (AssociationEnd)metaObject;
    }
    protected int handleGetMultiplicityLower(){
        Multiplicity multiplicity = associationEnd.getMultiplicity();
        if( multiplicity != null ){
           Iterator i = multiplicity.getRange().iterator();
            if( i.hasNext() )return ((MultiplicityRange)i.next()).getLower();}
        return -1;
    }
    protected int handleGetMultiplicityUpper(){
        Multiplicity multiplicity = associationEnd.getMultiplicity();
        if( multiplicity != null ) {
            Iterator i = multiplicity.getRange().iterator();
            if( i.hasNext() ) return ((MultiplicityRange)i.next()).getUpper();}
        return -1;
    }
}
```

図３－２１．　PojoEntityAssociationEndLogicImpl.java

・ 確認のためここでビルドをする。ビルドする場合はコマンドプロンプトを起動し、ProjectRoot にカレントディレクトリを移し、maven を実行する。

```
> maven
```

ビルドに成功すると以下の結果が出力される。

```
～省略～
INFO  [App] BUILD SUCCESSFUL
INFO  [App] Total time: 21 seconds
INFO  [App] Finished at: Tue Dec 06 15:31:12 JST 2005
INFO  [App]
```

３．２．４　　テンプレートの作成

テンプレートファイルの作成について順に説明する。テンプレートファイルは、カートリッジが生成するファイルのテンプレートである。AndroMDA は Velocity をテンプレートエンジンとして使っているので、Velocity のルールに従ってテンプレートを作成する。

・ 「**ProjectRoot/src/templates/meta**」フォルダがあるので削除する。そして「**ProjectRoot/src/templates/pojo**」というフォルダを作成し、pojo フォルダに PojoEntity.vsl ファイルを作成する。ファイルはプレーンテキストである。

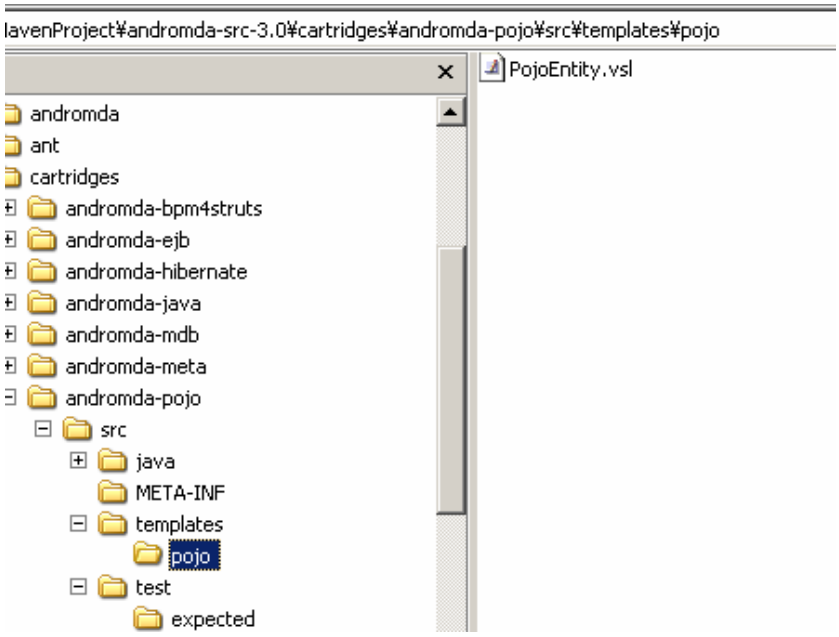


図３－２１．　PojoEntity.vsl の配置位置

・ 「**PojoEntity.vsl** 」は Velocity のテンプレートファイルである。Velocity の使用方法については詳しく解説しない。「**PojoEntity.vsl**」を以下の通り記述する。

```
// license-header java merge-point
//
// Attention: Generated code! Do not modify by hand!
// Generated by: PojoEntity.vsl in andromda-pojo-cartridge.
// AndroMDA: Metafacade = $class.class.name
/*
   Taged Values
#foreach ($tag in $class.taggedValues)
          ${tag.name} = ${tag.value}
#end
   Stereotypes
#foreach ($stereotype in $class.stereotypes)
          ${stereotype.name}
#end
*/

#if ($stringUtils.isNotBlank($class.packageName))
package $class.packageName;
#end

/**
$class.getDocumentation(" * ")
 */
public class $class.name
#if($class.generalization)
    extends ${class.generalization.fullyQualifiedName}
#end
#if ($serializable == 'true')
    implements java.io.Serializable
#end
{
/* ---------------------------------------------------------------------- */
```

次ページに続く

```
##  Attributes
#foreach ($attribute in $class.attributes)
#foreach ($stereotype in $attribute.stereotypes)
     /* <<${stereotype.name}>> */
#end
#foreach ($tag in $attribute.taggedValues)
     /* ${tag.name} = ${tag.value} */
#end
     private $attribute.getterSetterTypeName $attribute.name;
#end
/* ------------------------------------------------------------------------ */
    public ${class.name}()
    {
    }
## Getter Setter Method for Attributes
#foreach ($attribute in $class.attributes)
    /**
$attribute.getDocumentation("     * ")
     */
    $attribute.visibility                          $attribute.getterSetterTypeName
${attribute.getterName}()
    {
        return this.${attribute.name};
    }
    $attribute.visibility                                                  void
${attribute.setterName}($attribute.getterSetterTypeName $attribute.name)
    {
        this.${attribute.name} = $attribute.name;
    }

#end
/* ------------------------------------------------------------------------ */
#foreach ($operation in $class.businessOperations)
    $operation.visibility              $operation.returnType.fullyQualifiedName
$operation.signature $operation.exceptionList {
```

次ページに続く

```
#foreach ($tag in $operation.taggedValues)
    /* ${tag.name} = ${tag.value} */
#end
    }
#end
/* ------------------------------------------------------------------------ */
## Generate the Relation Methods.
#foreach ($associationEnd in $class.associationEnds)
// AndroMDA: AssociationEnd Metafacade = $associationEnd.class.name
//      AndroMDA:      Muliplicity      [$associationEnd.multiplicityLower      ,
$associationEnd.multiplicityUpper]
#set ($target = $associationEnd.otherEnd)
#if ($target.navigable)
    private $target.getterSetterTypeName $target.name;

    /**
     * Get the $target.name$target.getDocumentation("     * ")
     */
    public $target.getterSetterTypeName ${target.getterName}()
    {
        return this.${target.name};
    }
    /**
     * Set the $target.name
     */
    public void ${target.setterName}($target.getterSetterTypeName $target.name)
    {
        this.${target.name} = ${target.name};
    }
#end
#end
}
```

図３－２２．　PojoEntity.vsl

３．２．５　　設定ファイルの編集

設定ファイルを編集する。ここに Metafacade クラス、テンプレート、ステレオタイプ、タグ付き値などの設定情報を記述する。

・ 「**ProjectRoot/src/META-INF フォルダ**」以下の３つのファイルを編集する。

- **andromda-cartridge.xml**
- **andromda-metafacades.xml**
- **andromda-profile.xml**

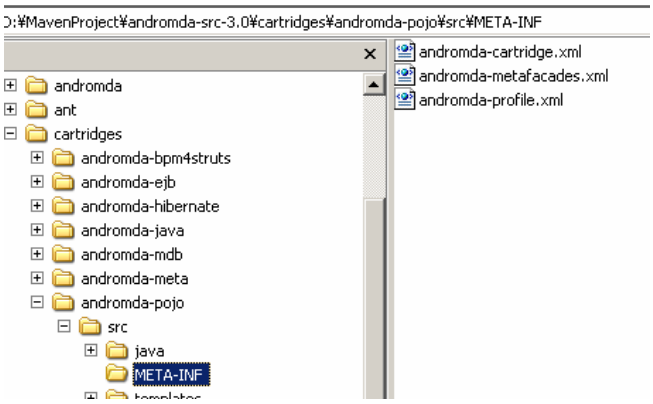


図３－２３．　設定ファイルの位置

・ 「**andromda-cartridge.xml**」を以下の通り編集する。「andromda-cartridge.xml」はカートリッジが生成するテンプレートファイルと生成に利用する Metafacade クラス、およびプロパティなどを記述した設定ファイルである

```
<cartridge name="pojo">

    <templateObject name="stringUtils"
                    className="org.apache.commons.lang.StringUtils"/>
    <!-- cartridge-templateObject merge-point-->

    <property reference="serializable" default="true"/>
    <!-- cartridge-property merge-point-->

    <!-- cartridge-resource merge-point -->
    <template
        path="templates/pojo/PojoEntity.vsl"
        outputPattern="{0}/{1}.java"
        outlet="generated-code"
        overwrite="true">
        <modelElements variable="class">
```

次ページに続く

```
            <modelElement>
                <type
name="org.andromda.cartridges.pojo.metafacades.PojoEntity"/>
            </modelElement>
        </modelElements>
    </template>
</cartridge>
```

図３－２４． andromda-cartridge.xml

・ 「**andromda-metafacades.xml**」を以下の通り編集する。「andromda-metafacades.xml」では、カートリッジで用意する Metafacade クラスがどのような条件で生成するのかを規定するため、およびプロパティの設定をする。

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- contains the hibernate cartridge metafacade mappings -->
<metafacades namespace="pojo">
    <metafacade
class="org.andromda.cartridges.pojo.metafacades.PojoEntityLogicImpl"
contextRoot="true">
        <mapping  class="org.omg.uml.foundation.core.UmlClass$Impl">
            <stereotype>ENTITY</stereotype>
        </mapping>
    </metafacade>
    <metafacade
class="org.andromda.cartridges.pojo.metafacades.PojoEntityAssociationEndLogicIm
pl">
        <mapping  class="org.omg.uml.foundation.core.AssociationEnd$Impl">

<context>org.andromda.cartridges.pojo.metafacades.PojoEntity</context>
        </mapping>
    </metafacade>
</metafacades>
```

図３－２５． andromda-metafacades.xml

・ 「**andromda-profile.xml**」を以下の通り編集する。「andromda-profile.xml」はカートリッジ内で使用されるステレオタイプやタグ付き値、その他の値を別名にマッピングするための設定ファイルである。

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!--
  The default Hibernate profile mappings.
-->
<mappings name="pojo">
    <!-- tagged values -->
    <mapping>
        <from>POJO_TABLE</from>
        <to>@jp.co.exa-corp.persistence.table</to>
    </mapping>
    <mapping>
        <from>POJO_COLUMN</from>
        <to>@jp.co.exa-corp.persistence.column</to>
    </mapping>
</mappings>
```

図３－２６． andromda-profile.xml

・ ビルドをする。ビルドする場合はコマンドプロンプトを起動し、**ProjectRoot** にカレントディレクトリを移し、maven を実行する。

```
> maven
```

ビルドに成功すると以下の結果が出力される。

```
～省略～
INFO  [App] BUILD SUCCESSFUL
INFO  [App] Total time: 21 seconds
INFO  [App] Finished at: Tue Dec 06 16:53:42 JST 2005
INFO  [App]
```

上記の結果を確認したら、andromda-pojo カートリッジ実装は完了である。生成されたカートリッジは jar ファイルにまとめられ、ローカル PC の maven のリポジトリに自動的にコピーされる。よって今後、AndroMDA プロジェクトで andromda-pojo カートリッジを使用することを宣言する（カートリッジを使用するプロジェクトの「mda/project.xml」に記述する）だけで andromda-pojo カートリッジが AndroMDA プロジェクト内で使用される。

また、カートリッジは **ProjectRoot/target/andromda-pojo-cartridge-3.0.jar** として jar ファイルにまとめられている。この jar ファイルを別 PC の maven リポジトリにコピーすることで、その PC でも POJO カートリッジが使用可能になる。

３．２．６　　POJO カートリッジの使用

AndroMDA プロジェクトを作成し、実際に POJO カートリッジを使用する。このプロジェクトでは POJO カートリッジを使い、２つ**<<Entity>>**クラスから２つの POJO クラスを生成する。

・ AndroMDA プロジェクトとして「Pojo Sample」というプロジェクトを作成する。コマンドプロンプトから以下のように実行する。

```
>maven andromdapp:generate

  _   _
 | ¥/  |_ _Apache_ __
 | |¥/| / _` ¥ V / -_) ' ¥   ~ intelligent projects ~
 |_|  |_¥__,_|¥_/¥__|_||_|   v. 1.0.2


Please enter your first and last name (i.e. Chad Brandon):
Koji Iida
Please enter the name of your J2EE project (i.e. Animal Quiz):
Pojo Sample
Please enter the id for your J2EE project (i.e. animalquiz):
pojosample
Please enter a version for your project (i.e. 1.0-SNAPSHOT):
1.0
Please enter the base package name for your J2EE project (i.e. org.andromda.samples):
jp.co.exa
Would you like an EAR or standalone WAR (enter 'ear' or 'war')?
ear
Please enter the type of transactional/persistence cartridge to use (enter 'hibernate', 'ejb', or
'spring'):
spring
Would you like a web application? (enter 'yes' or 'no'):
no
Would you like to be able to expose your services as web services? (enter 'yes'or 'no'):
no
```

図３－２７．　AndroMDA プロジェクト「Pojo Sample」の作成

実行すると、pojosample プロジェクトが作成される。

・ 「**pojosample/mda/project.xml**」ファイルの**<dependencies/>属性**に次の属性を追加する。これは POJO カートリッジを使用するための宣言である。追加する部分を太字で示す。また下線の部分（andromda-spring-cartridge 宣言～andromda-ocl-query-library 宣言）をコメントアウトし、それらのカートリッジを使用不可する。これは使用するステレオタイプが POJO カートリッジと競合するためである。

```
<dependencies>
    <!-- 省略 -->
    <!--
    <dependency>
        <groupId>andromda</groupId>
        <artifactId>andromda-spring-cartridge</artifactId>
    <!-- 省略 -->
    <dependency>
        <groupId>andromda</groupId>
        <artifactId>andromda-ocl-query-library</artifactId>
        <version>${andromda.version}</version>
    </dependency>
    -->
    <dependency>
        <groupId>andromda</groupId>
        <artifactId>andromda-pojo-cartridge</artifactId>
        <version>${andromda.version}</version>
        <type>jar</type>
        <properties>
            <generated-code>${maven.andromda.core.generated.dir}
                                                    </generated-code>
        </properties>
    </dependency>
</dependencies>
```

図３－２８． andromda-profile.xml の変更

「pojosample/mda/src/uml/PojoSampleModel.xmi」を Magic Draw で開き、モデルを編集する。jp.co.exa.domain パッケージを作成し、パッケージ以下に Customer クラスと Purchase クラスを作成する。そして２つのクラスに**<<Entity>>ステレオタイプ**を作成する。また、２つのクラスに関連を設定する。その他細かいモデリングは下図を参照していただきたい。

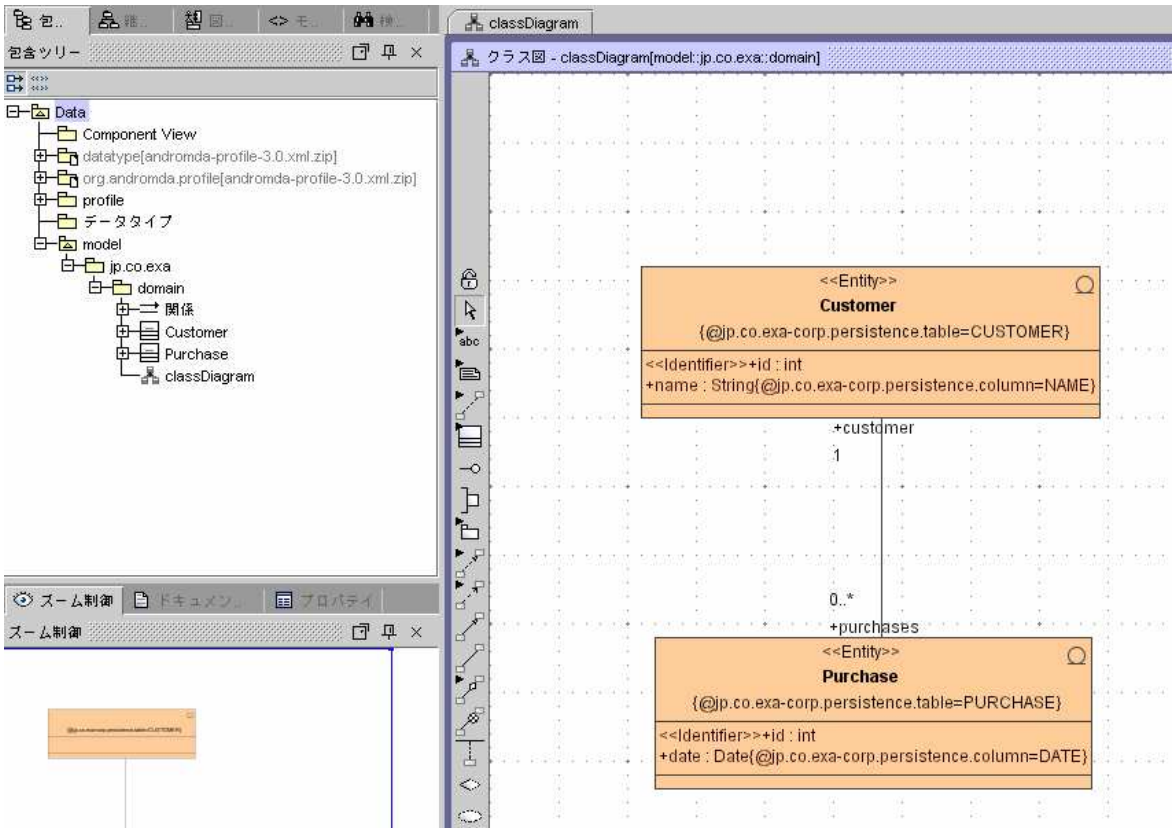


図３－２９． PojoSample のモデリング

・ コマンドプロンプトでカレントディレクトリを pojosample フォルダにし、maven を実行する。

```
> maven
```

ビルドに成功すると以下の結果が出力される。

```
～省略～
INFO  [App] BUILD SUCCESSFUL
INFO  [App] Total time: 21 seconds
INFO  [App] Finished at: Mon Dec 12 11:21:04 JST 2005
INFO  [App]
```

・ コマンドプロンプトでカレントディレクトリを pojosample フォルダにし、maven を実行する。

・ 「pojosample/core/target/src/jp/co/exa/domain」フォルダ以下に「Customer.java」と「Purchase.java」があるので、ファイルの内容を確認する。これらのファイルは pojo カー

トリッジによって生成されたファイルである。

３．２．７　　カートリッジのテスト

カートリッジのテストを作成する。AndroMDA は、カートリッジのビルド時にカートリッジを自動的にテストする機能を持っている。このテストは、自身のカートリッジを使用してモデルからコードを生成し、事前に用意されたコードとマッチしているかどうかを判定する。このテストでビルドしたカートリッジの妥当性を検証する。

・ andromda-meta-cartridge 用のテストモデルとコードを削除する。下記のファイルとフォルダを削除する。

- **ProjectRoot/src/test/uml/MetaCartridgeTestModel.xml.zip**
- **ProjectRoot/target/cartridge-test/actual 以下のファイルとフォルダ**

・ モデルを配置する。ここでは３．２．６で作成したモデルをそのまま利用する。「**PojoSampleModel.xmi**」を「**PojoCartridgeTestModel.xml**」にリネームし、ZIP で圧縮する。圧縮した ZIP ファイルを「**PojoCartridgeTestModel.xml.zip**」にリネームし、「**PojoCartridgeTestModel.xml.zip**」を「**ProjectRoot/src/test/uml**」**フォルダ**にコピーする。

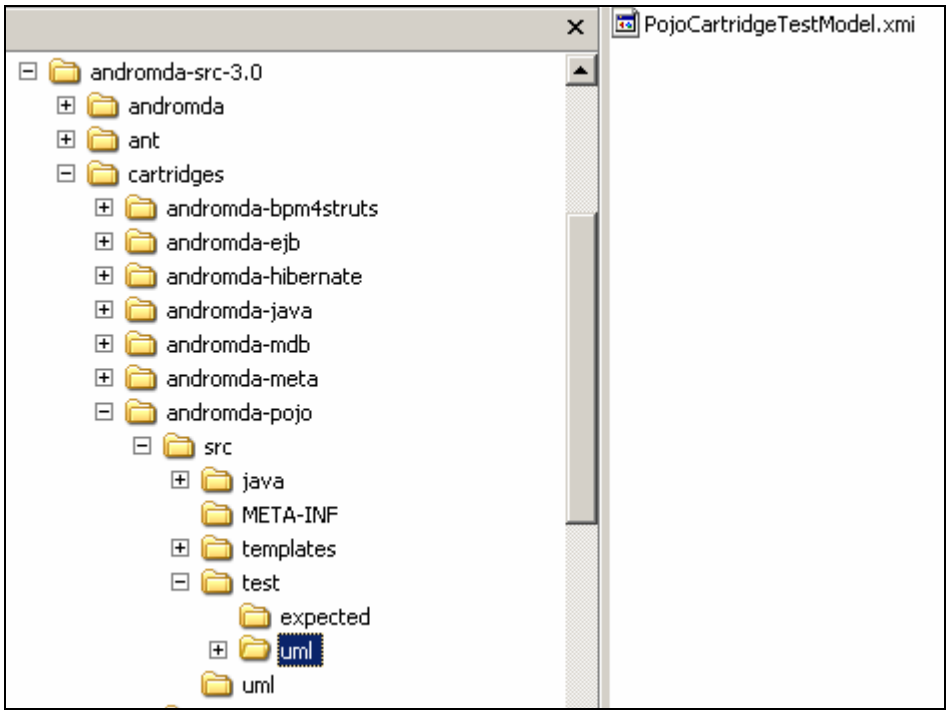


図３－３０．　モデルの配置

・ テスト用のモデルファイルを指定し、プロパティを設定する。「ProjectRoot /project.properties」を開き、太字の部分を追加する。

```
～省略～
# ----------------------------------------------------------------
# Cartridge Test Properties
# ----------------------------------------------------------------

# define the test model
test.model.file=${maven.src.dir}/test/uml/PojoCartridgeTestModel.xml.zip
andromda.cartridge.test.model.uri=jar:file:${test.model.file}!/PojoCartridgeTes
tModel.xml
test.output.dir=${andromda.cartridge.test.actual.dir}
# Define the properties of this cartridge to test
～省略～
andromda.cartridge.test.property.facade-logic-impls=${test.output.dir}
andromda.cartridge.test.property.6=languageMappingsUri
andromda.cartridge.test.property.languageMappingsUri=Java
andromda.cartridge.test.property.7=wrapperMappingsUri
andromda.cartridge.test.property.wrapperMappingsUri=JavaWrapper
andromda.cartridge.test.property.8=jdbcMappingsUri
andromda.cartridge.test.property.jdbcMappingsUri=JDBC
andromda.cartridge.test.property.9=sqlMappingsUri
andromda.cartridge.test.property.sqlMappingsUri=HypersonicSql
andromda.cartridge.test.property.10=generated-code
andromda.cartridge.test.property.generated-code=${test.output.dir}
```

図３－３１． project.properties の編集

・ カートリッジをビルドする。

```
> maven
```

ビルドに成功すると以下の結果が出力される。

```
～省略～
BUILD SUCCESSFUL
```

・ 「ProjectRoot /target/cartridge-test/actual」以下のフォルダ（図では jp フォルダ）を ZIP で圧縮し、「cartridge-output.zip」にリネームする。
そして、「ProjectRoot/src/test/expected/cartridge-output.zip」に上書きする。

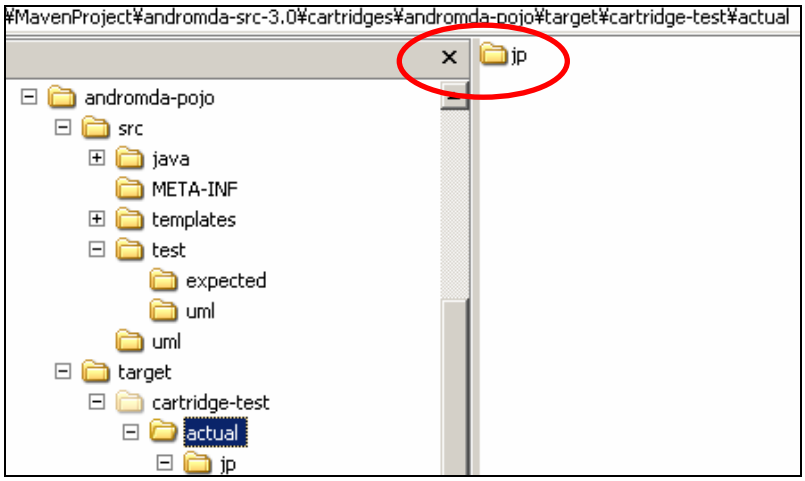


図３－３２．　テスト検証用データの圧縮

下記のファイルとフォルダを削除する。

- **ProjectRoot/target/cartridge-test/expected** 以下のファイルとフォルダ

・ もう１度カートリッジをビルドする。

```
> maven
```

maven のログにテストの実行結果が出力される。

```
～省略～
test:single:
    [junit] Running org.andromda.cartridges.testsuite.CartridgeTest
    [junit] INFO  [CartridgeTest]  --- Expecting 2 Generated Files ---
    [junit] INFO  [CartridgeTest] binary suffixes --> [jpg, jpeg, gif, png, jar,
 zip]
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.093 sec
```

ビルドに成功すると以下の結果が出力される。

```
～省略～
INFO  [App] BUILD SUCCESSFUL
INFO  [App] Total time: 21 seconds
INFO  [App] Finished at: Mon Dec 12 18:06:18 JST 2005
INFO  [App]
```

以上で、POJO カートリッジの完成である。

４ 実例紹介

これまでカートリッジ自体のことやカートリッジの内容、簡単なカートリッジ作成方法について説明してきた。ここでは３．２章のような簡単なカートリッジではなく実際に利用できるようなカートリッジの紹介をする。

AndroMDA を用いて開発するシステムに、異なるシステム(ear が異なるという意味)間でメッセージをやり取りする要件があり、メッセージのやり取りをする技術には JMS（Java Message Service）と MDB（message-driven Enterprise JavaBeans）を選択した。しかし、AndoraMDA ではこの２つの技術の出力をサポートしていなかった。

そこで JMS や MDB を生成する独自カートリッジ「MDB カートリッジ」を作成した。本章では MDB カートリッジの概要および Metafacade クラスのモデル、UML プロファイル、実際に適用した UML モデルと出力ファイルイメージを紹介する。

４．１　MDB カートリッジの概要

MDB カートリッジは JMS によるメッセージ送信部分と MDB によるメッセージ受信部分、送受信されるメッセージそのもの、MDB の設定を記述するアプリケーションサーバや EJB の設定ファイルを生成するカートリッジである。生成するうちユーザが実装を追加する部分はメッセージ受信後のメッセージを処理する部分、つまりビジネスロジックのみになる。MDB カートリッジはその用途から AndroMDA が標準で用意している EJB、Spring、Hibernate カートリッジと併用して利用する。各要素については下記のとおりである。

- **メッセージ要素**
  JMS と MDB 間で送られるメッセージ、このメッセージは単純な JavaBeans で構成している。JMS と MDB 間で送受信する際には JavaAPI により XML 形式のテキストで送受信する。
- **メッセージ受信要素**
  JMS によって送るメッセージを受信する MDB の実装部分、Generation Gap パターンを適用し、インタフェース、抽象クラス、実装クラスを生成する。MDB の実装についてユーザは意識する必要がなく、指定のビジネスロジックメソッドを実装するだけでよい。
- **メッセージ送信要素**
  JMS によるメッセージ送信の実装部分、ユーティリティクラスとすることにより JMS の実装についてユーザは意識する必要がなく、送信したいメッセージをメソッドに渡すだけのインタフェースを提供する。
- **JMS や MDB の設定要素**
  JMS や MDB について記述する設定ファイル、各アプリケーションサーバ固有の設定や EJB の設定を記述する設定ファイルを生成する。

下図の赤色円が MDB カートリッジの適用範囲である。

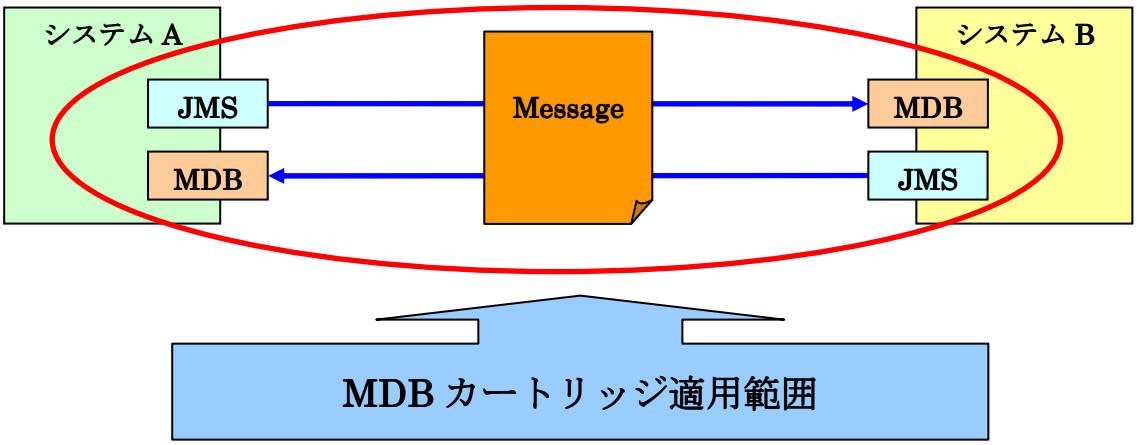


図４－１．MDB カートリッジ適用範囲

４．２　MDB カートリッジの Metafacade と利用する UML プロファイル

ここではMDBカートリッジのMetafacadeクラスのモデルと使用するプロファイルを紹介する。

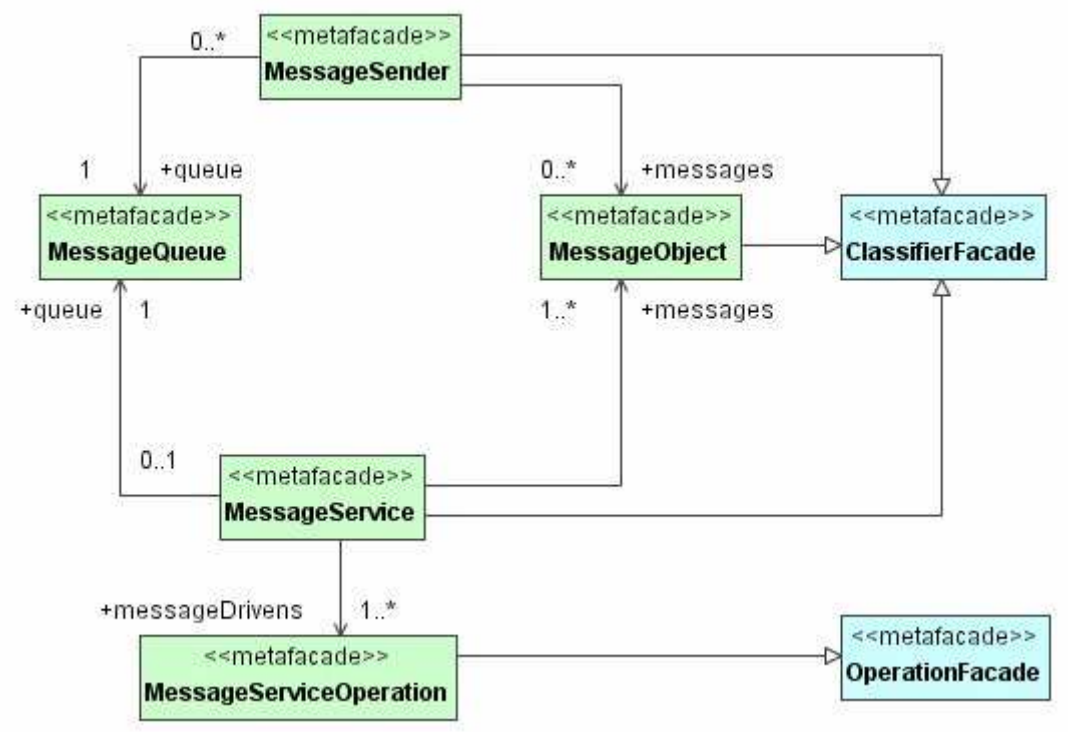


図４－２．MDB カートリッジの Metafacade クラスのモデル

「ClassifierFacade」および「OperationFacade」については AndroMDA で用意されている Metafacade である。残りの Metafacade が MDB カートリッジ独自の Metafacade である。下記に内容を記す。

- MessageObject

  JMS と MDB 間で受け渡すメッセージのクラスを生成する Metafacade クラス。

- MessageQueue

  メッセージの送信タイプを持つ Metafacade クラス。送信タイプには「Queue」と「Topic」がある。この Metafacade はメッセージ送信クラスや設定ファイル生成時に利用する。

- MessageSender

  メッセージを送信するクラスの生成に利用する Metafacade クラス。関連する「MessageQueue」や「MessageObject」を組み合わせてメッセージ送信クラスを生成する。

- MessageService

  メッセージを受信するクラスや設定ファイルの生成に利用する Metafacade クラス。関連する「MessageQueue」や「MessageObject」、「MessageServiceOperation」を組み合わせてメッセージ受信クラスや設定ファイルを生成する。

- MessageServiceOperation

  JMSからMDBへ送信されたメッセージのクラスを処理するユーザが実装する必要があるメソッド部分を生成する Metafacade クラス、メッセージクラスは送信時 XML 形式で送信されるが、この Metafacade で生成するメソッドでは XML ではなくメッセージクラスとして

処理できるようにメソッドを生成する。

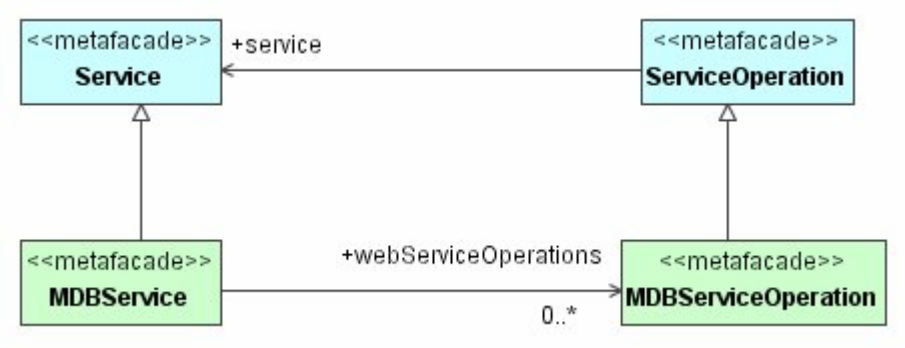


図４－３．他のカートリッジとバッティングする部分を生成する Metafacade クラスのモデル

MDB カートリッジは他のカートリッジと生成するファイルがバッティングするものが一部ある。図４－３.にある Metafacade はバッティングする部分を処理するための Metafacade である。

「Service」および「ServiceOperation」は AndroMDA で用意されている Metafacade である。「MDBService」、「MDBServiceOperation」Metafacade クラスは MDB カートリッジが他のカートリッジとバッティングするファイルを生成する際に利用する Metafacade である。内容は併用するカートリッジにあわせて併用するカートリッジが出力する内容と MDB カートリッジ自身が出力する内容をマージして出力する。

MDB カートリッジが利用するプロファイルを下記に記す。

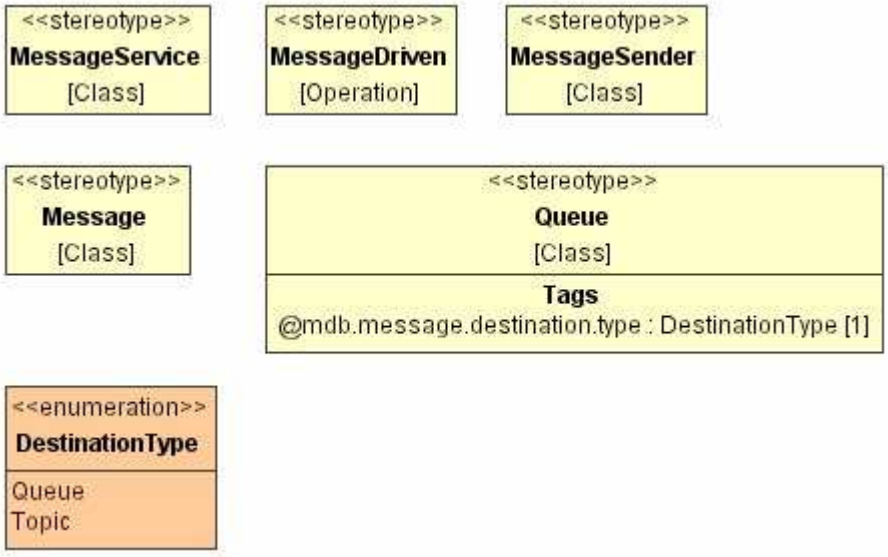


図４－４．MDB カートリッジが利用するプロファイル

各ステレオタイプは、Metafacade クラスに対応する。対応は下記のとおりである。

表４－１．ステレオタイプと Metafacade クラスの対応

ステレオタイプ	Metafacade クラス
Message	MessageObject
Queue	MessageQueue
MessageSender	MessageSender
MessageService	MessageService
MessageDriven	MessageServiceOperation

４．３　MDB カートリッジを適用したモデルとその出力イメージ

MDB カートリッジを適用したモデルを下図に記す。

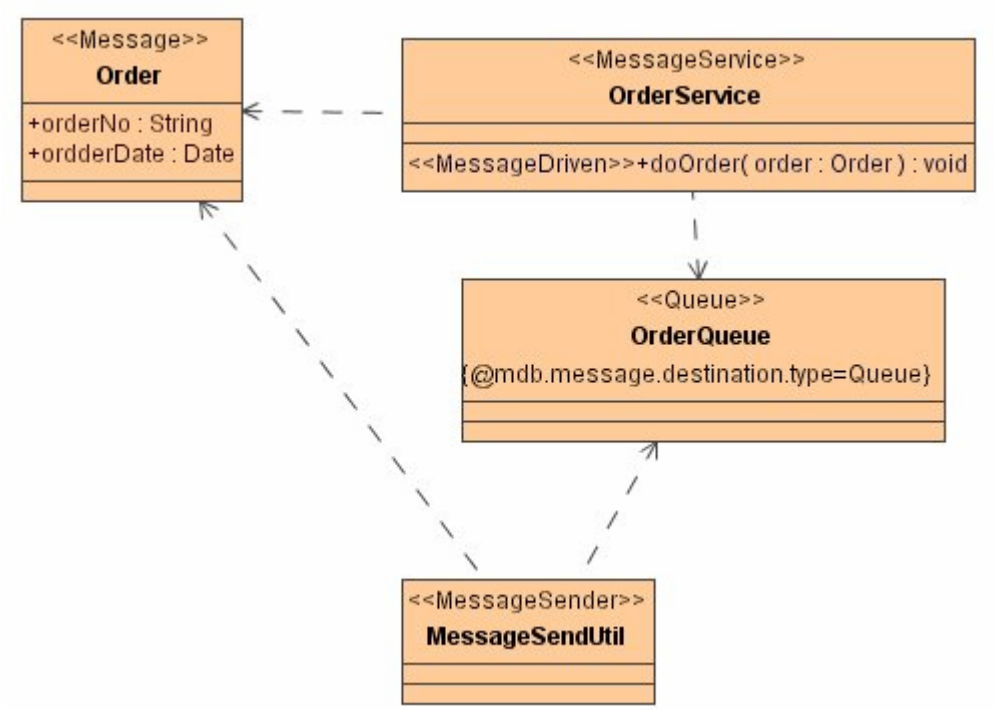


図４－５．MDB カートリッジを使用した UML モデル図

図４－５．を元にファイルを出力すると、下図のようなクラス群を生成する。

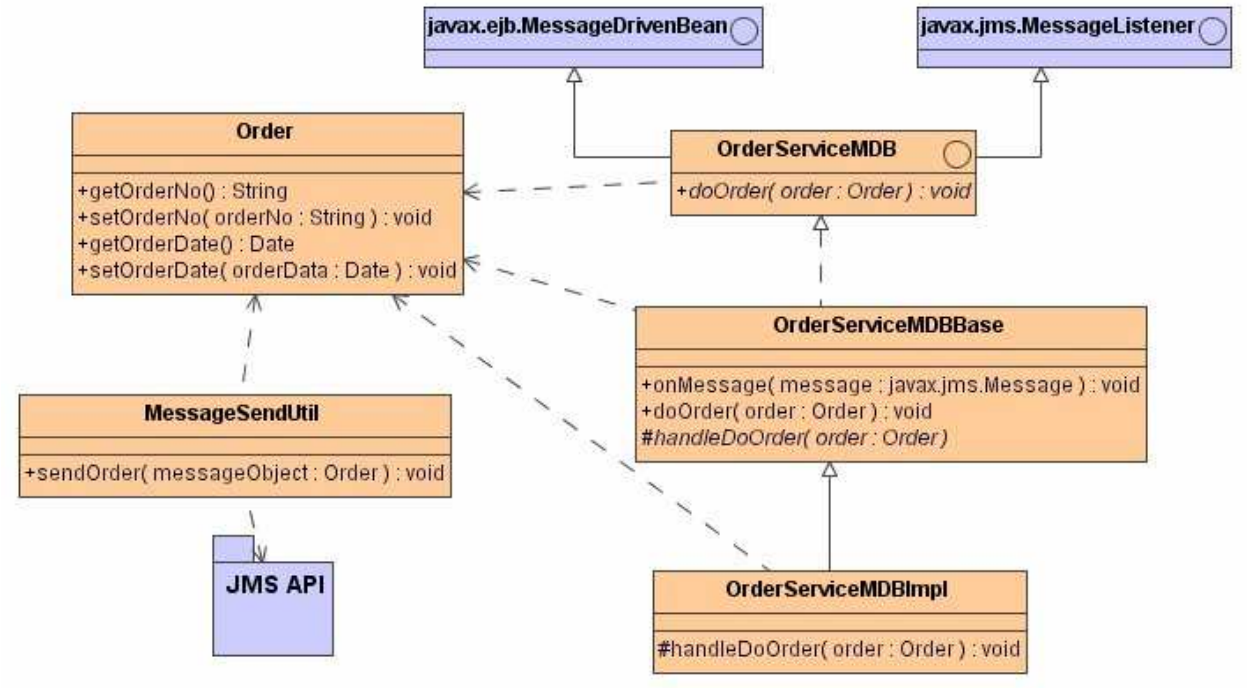


図４－６．MDB カートリッジによって生成されたクラス群

図４－６．には図４－５．のステレオタイプ<<Queue>>がついたクラスに対応するようなクラスがない、これはステレオタイプ<<Queue>>に対応するクラスの Metafacade クラスは設定ファイルやクラスの一部を生成するのに利用するだけで他の Metafacade のように対応したファイルを生成するようなことがないため、生成物のクラス群にはクラスが存在しない。

尚、MDB カートリッジの設定手順や使用方法については、別冊「MDB カートリッジ手順書」を参照していただきたい。

5 プロファイルのカスタマイズ

UML にてモデリングを行う際は、モデル要素に対して役割や用法などを装飾するためのステレオタイプや、任意情報をモデル要素に定義するためのタグ付き値を利用する。そしてそれらの拡張情報を構成しているものをプロファイルと呼ぶ。

AndroMDA にも標準のプロファイルが提供されている。そのプロファイルに AndroMDA が用意している既存カートリッジで使用するステレオタイプやタグ付き値、データ型などが含まれる。しかし、独自カートリッジを作成する際に新たなステレオタイプやタグ付き値、データ型が必要になることもある。その場合は、必要なステレオタイプやタグ付き値などを定義する独自プロファイルの作成が必要である。

この章では、独自プロファイルの作成および、そのプロファイルをプロジェクトへ適用する方法を説明する。尚、説明する題材として４章で紹介した MDB カートリッジ用のプロファイルを題材にする。

５．１　独自プロファイルの作成

ここでは、独自プロファイルの作成方法を説明する。尚、作成するために使用するツールとして MagicDraw9.5 のプロフェッショナル版を使用する。注意点として MagicDraw のパーソナル版ではプロファイルの作成はできないことに気をつけていただきたい。パーソナル版ではエクスポート関連の機能が制限されている。

５．１．１　　プロファイル用のプロジェクト作成

MagicDraw を起動し、新規プロジェクトを作成する。新規プロジェクトをロード後、「包含ツリー」に表示されている「Component View」以外のパッケージを、独自プロファイル作成には不要なため削除する。

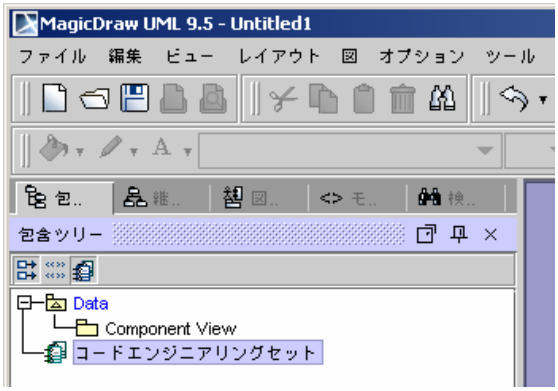


図５－１．不要なパッケージの削除

次に「Data」の下にプロファイル用に「jp.co.exa_corp.mdb.profile」という名前で「モデルパッケージ」を作成する。

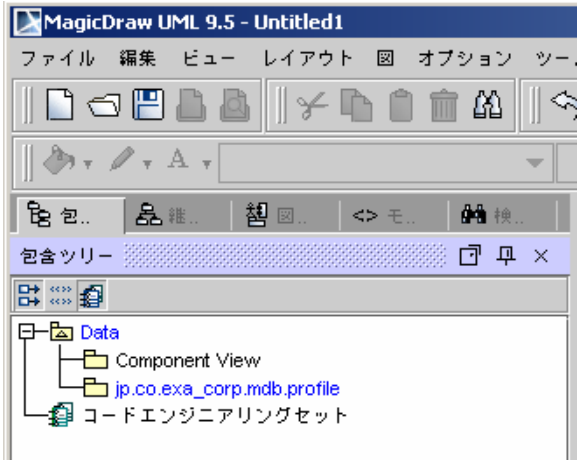


図５－２．プロファイル用のパッケージ作成

これでプロファイルを作成するためのプロジェクトの準備ができたこととなる。

５．１．２　　ステレオタイプの作成

まず、ステレオタイプを作成する。MDB カートリッジでは、以下の５つのステレオタイプを必要とする。ここでは、ステレオタイプ<<Message>>を例にとって作成方法を説明する。

表５－１．MDB カートリッジのステレオタイプ

ステレオタイプ	役割
Message	メッセージとなるクラスを表す
Queue	送信タイプを定義するクラスを表す
MessageSender	メッセージ送信を行うクラスを表す
MessageService	メッセージ受信を行うクラスを表す
MessageDriven	メッセージ処理を行うメソッドを表す

５．１．１で作成した「jp.co.exa_corp.mdb.profile」パッケージの仕様を開き「内部エレメント」タブを表示する。ステレオタイプを追加するには、「追加」ボタンを押下し、ステレオタイプを選択する。

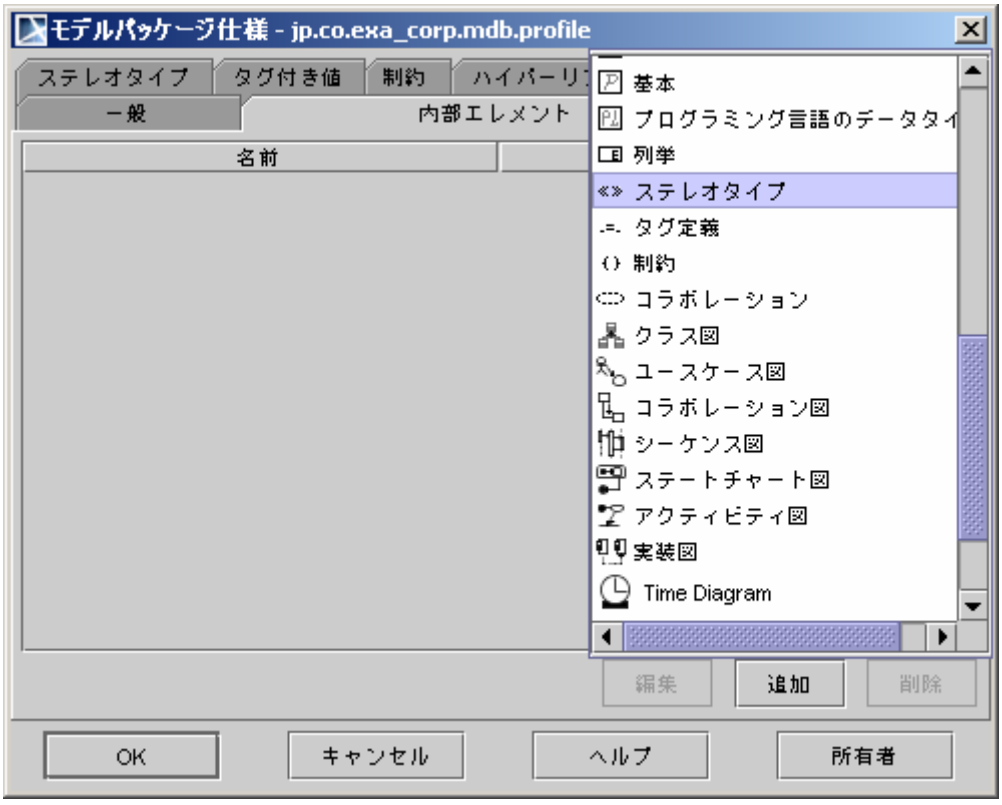


図５－３．ステレオタイプの追加

表示されたダイアログの「一般」タブの「名前」に「Message」と入力する。

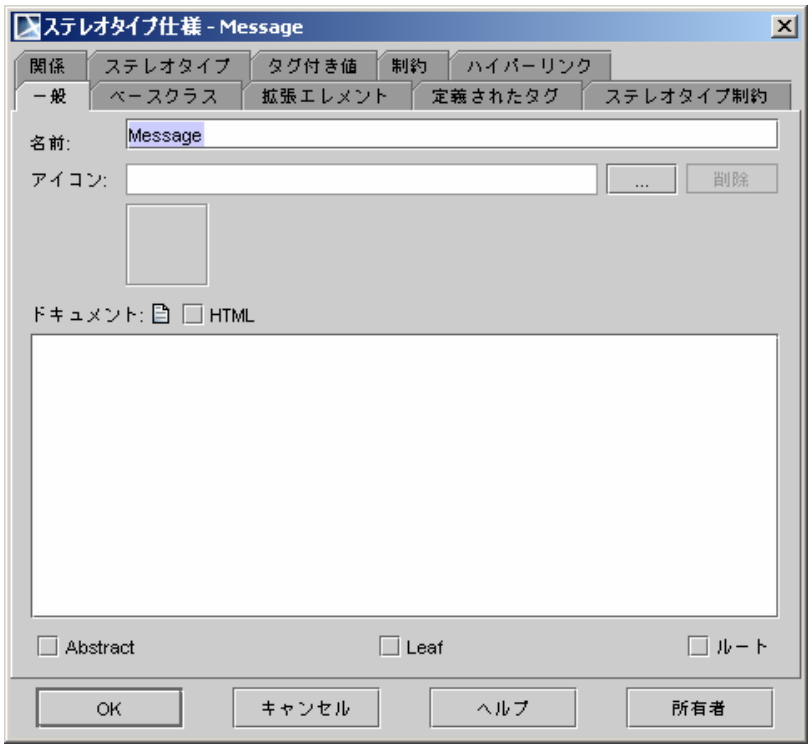


図５－４．ステレオタイプ名入力

次に「ベースクラス」タブを表示し、「選択されたアイテム」欄から「ModelElement」を削除して「Class」だけになるように追加する。ベースクラスは、どの部分につけるステレオタイプかを意味する。例えば「Class」であればクラスモデルに「Operation」であればメソッドにつく。

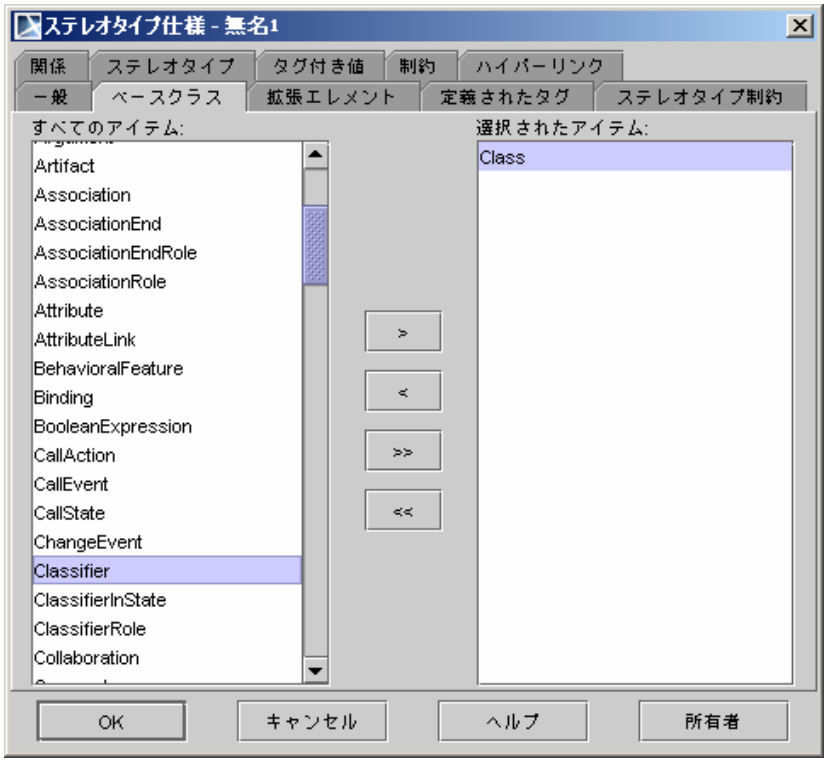


図５－５．ベースクラスの選択

その後、「OK」ボタンを押下して設定を確定する。

図５－６．ステレオタイプの追加後

これで、ステレオタイプの追加が完了する。ちなみに、MDB カートリッジのその他のステレオタイプは下記のような設定になる。

表５－２．MDB カートリッジのステレオタイプ設定内容

ステレオタイプ	ステレオタイプ名	ベースクラス
Message	Message	Class
Queue	Queue	Class
MessageSender	MessageSender	Class
MessageService	MessageService	Class
MessageDriven	MessageDriven	Operation

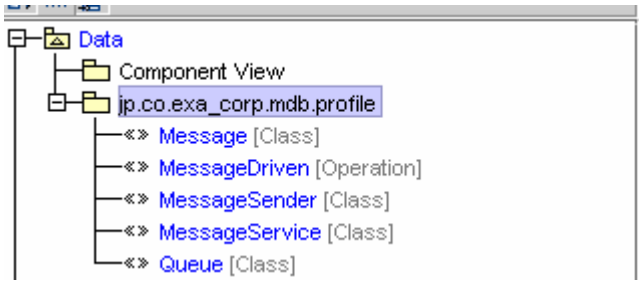


図５－７．すべてのステレオタイプ設定後

５．１．３　　データ型の作成

ステレオタイプの次はデータ型を作成する。データ型はモデルクラスの属性の型などに利用される。ここでは後述するタグ付き値の値として利用する列挙型を作成する。作成する列挙型はMDB カートリッジのメッセージ送信タイプの値として使用する。

ステレオタイプと同様、「**jp.co.exa_corp.mdb.profile**」パッケージの仕様を開き「内部エレメント」タブを表示する。その後「追加」ボタンを押下し、「列挙」を選択する。

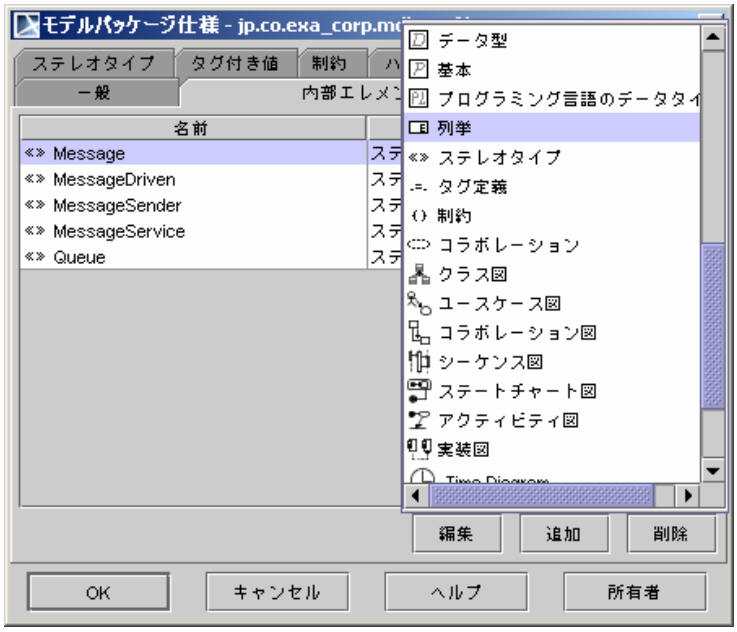


図５－８．列挙の追加

ダイアログが表示されたら、「一般」タブの「名前」に「**DestinationType**」と入力する。

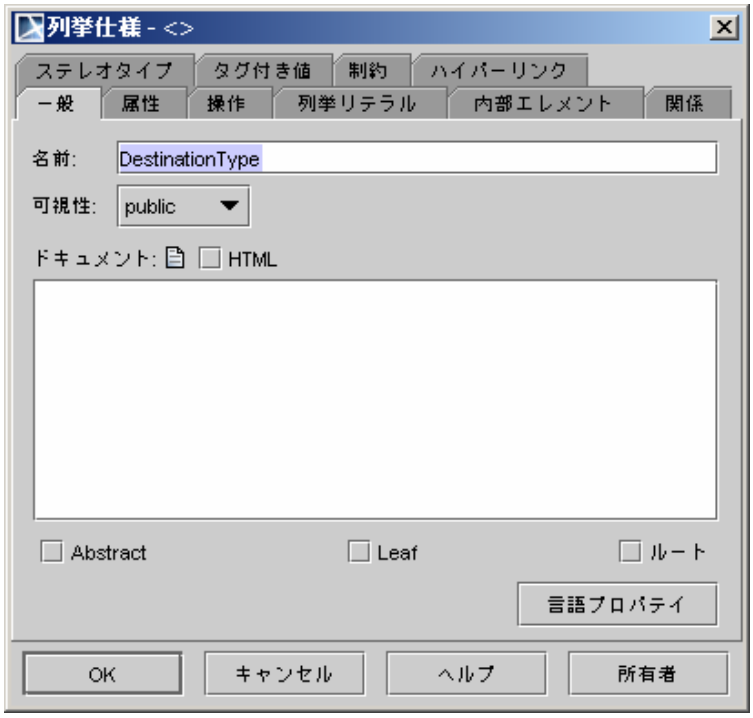


図５－９．列挙の名前入力

その後、「列挙リテラル」タブを表示し「追加」ボタンを押下する。すると列挙リテラルの仕様ダイアログが表示されるので「一般」タブの「名前」に「**Queue**」と入力する。

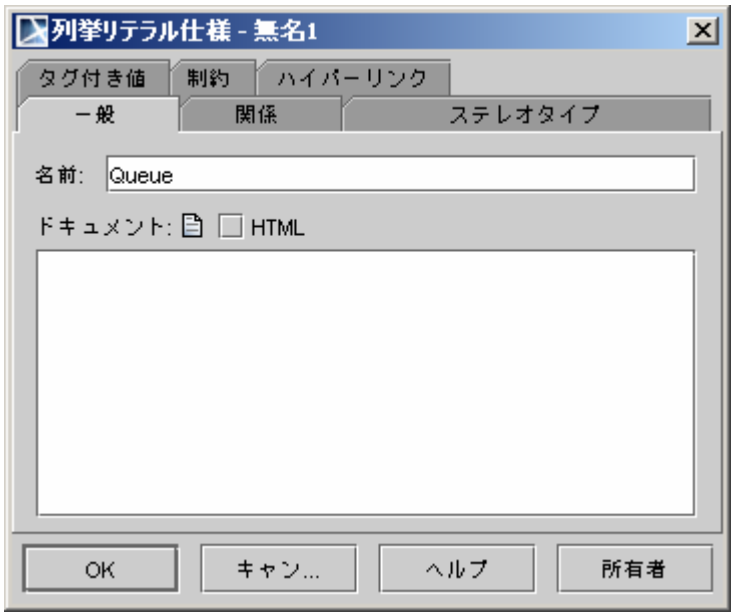


図５－１０．列挙リテラル名の入力

同様の操作で「**Topic**」も追加する。追加後は「OK」ボタンで確定する。

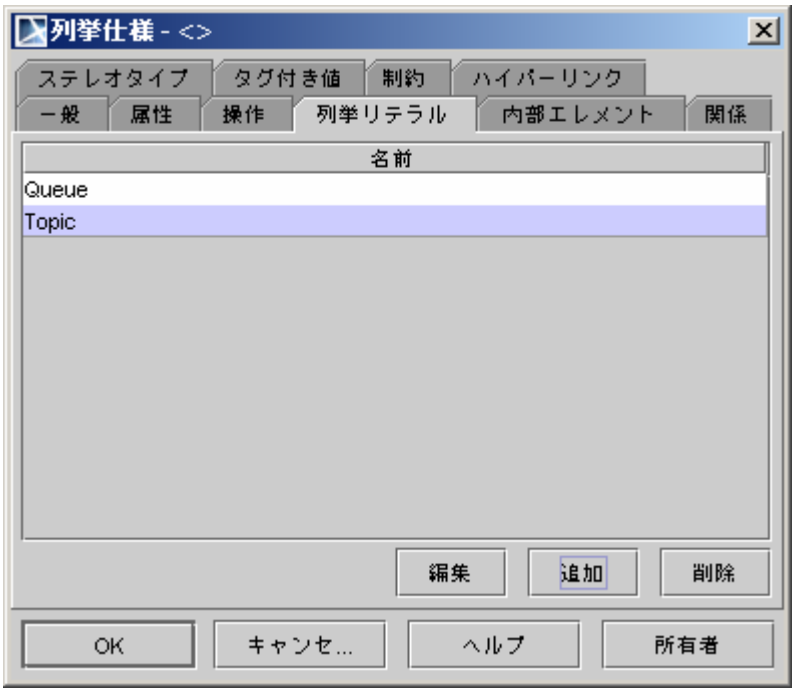


図５－１１．列挙リテラル追加後

これで列挙型の設定が完了する。

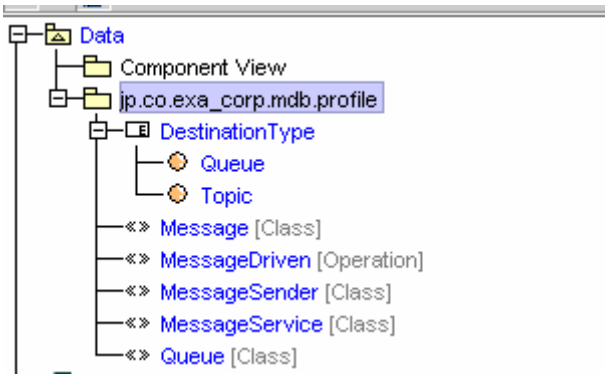


図５－１２．列挙型設定後

５．１．４　　タグ付き値の作成

最後にタグ付き値を作成する。MDB カートリッジでは１つだけタグ付き値を使用する。それは、メッセージの送信タイプの指定である。尚、MagicDraw ではタグ付き値の定義を「タグ定義」と呼んでいる。

ステレオタイプやデータ型と同様、「**jp.co.exa_corp.mdb.profile**」パッケージの仕様を開き「内部エレメント」タブを表示する。その後「追加」ボタンを押下し、「タグ定義」を選択する。

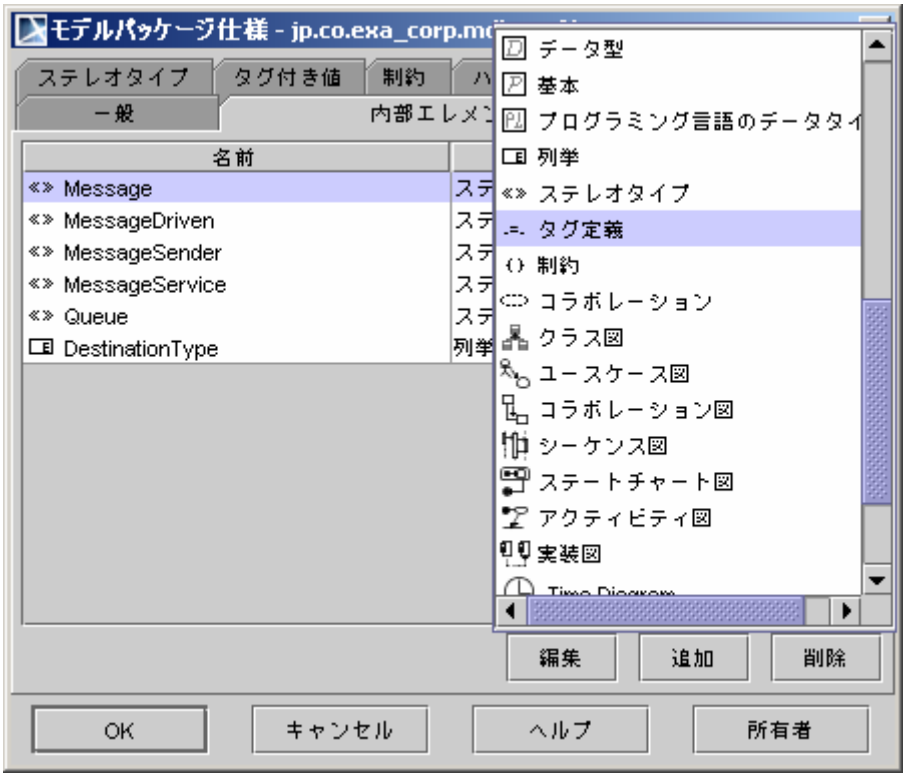


図５－１３．タグ定義の追加

表示されたダイアログの「一般」タブの「名前」に「**@mdb.message.destination.type**」、「タイプ」に５．１．３で作成したデータ型(列挙型)の「**DestinationType**」を指定する。タイプが列挙型のため「デフォルト値」を指定する。デフォルト値は「**Queue**」を指定する。

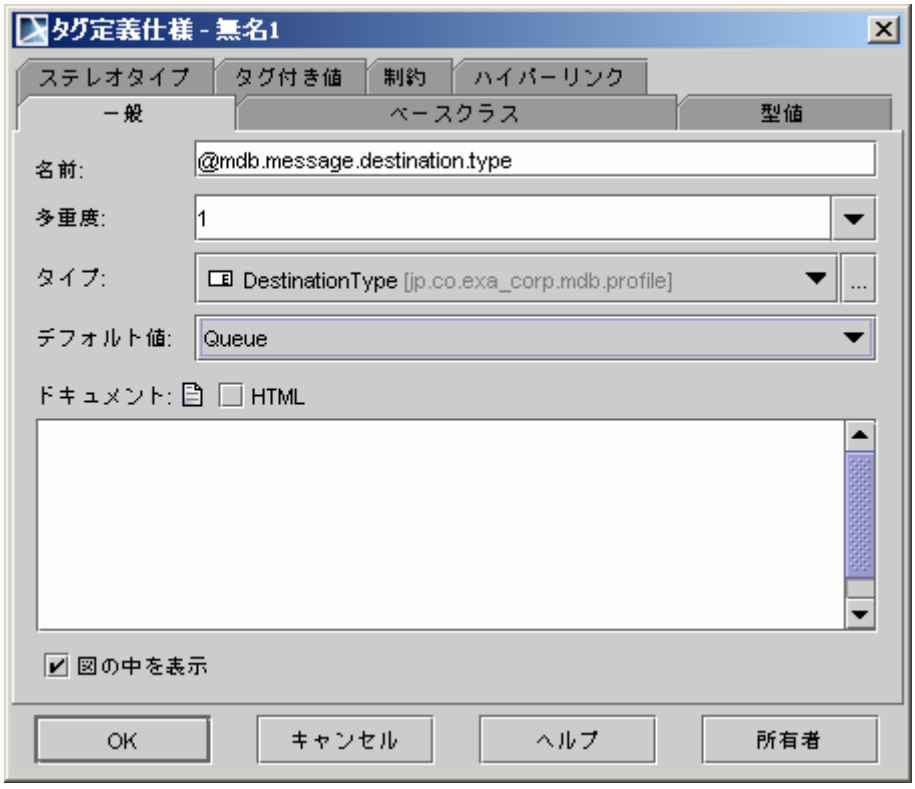


図５－１４．タグ定義の入力

これでタグ定義の設定が完了したので、「OK」ボタンを押下して確定する。

図５－１５．タグ定義完了後

タグ定義は完了したが、MDB カートリッジではこのタグ付き値はステレオタイプ<<Queue>>のついたクラスでのみ定義できるようにしている。よってこのタグ定義もステレオタイプ<<Queue>>に関連する。関連する方法は簡単である。タグ定義「**@mdb.message.destination.type**」をドラッグ＆ドロップでステレオタイプ<<Queue>>にあわせるだけである。それだけで、ステレオタイプ<<Queue>>にタグ定義を関連付けることが可能である。

図５－１６．タグ定義の関連付け

５．１．５　　プロファイルの保存

すべての設定が完了したのでプロファイルを保存する。しかし、このままただ保存してもプロファイルではなくただのプロジェクトになってしまう。そのため、まずはパッケージを共有化する。まず、「ファイル」メニューの「エクスポート」→「共有パッケージ」を選択する。

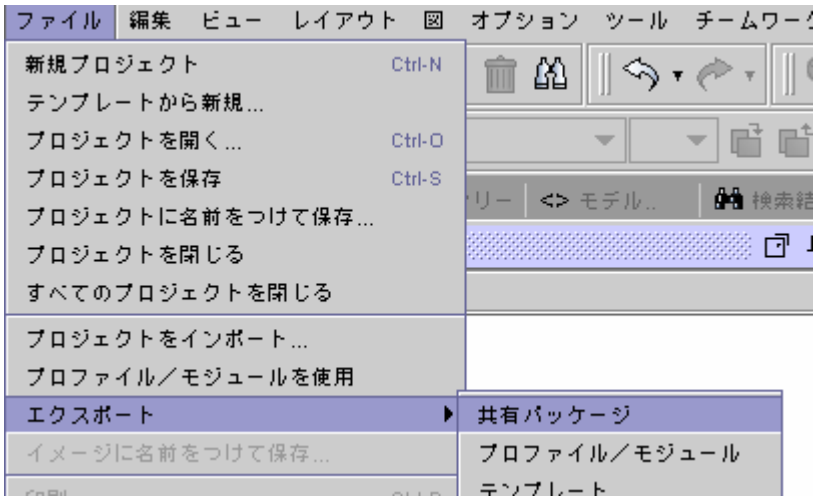


図５－１７．共有パッケージ選択

表示されたダイアログで「jp.co.exa_corp.mdb.profile」パッケージを「選択されたオブジェクト」欄へ追加する。

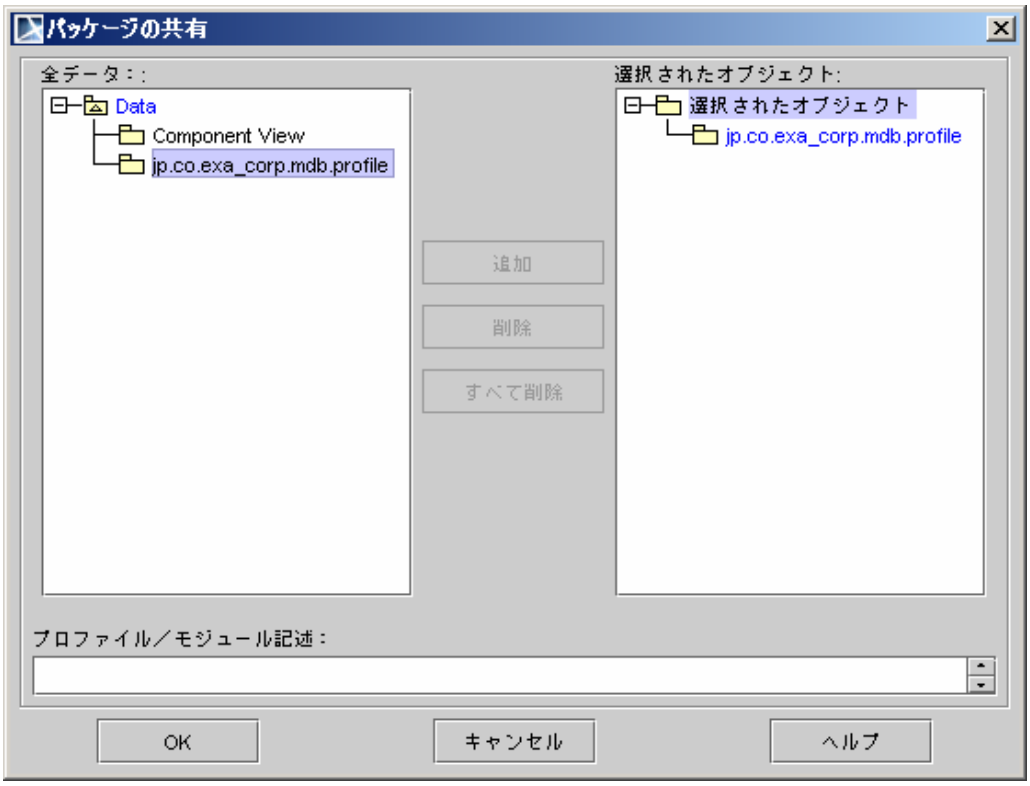


図５－１８．共有するパッケージの選択

その後、「OK」ボタンを押下して確定する。すると「包含ツリー」上の「jp.co.exa_corp.mdb.profile」パッケージのアイコンが変化し、パッケージの後ろに「[分割]」と表示される。

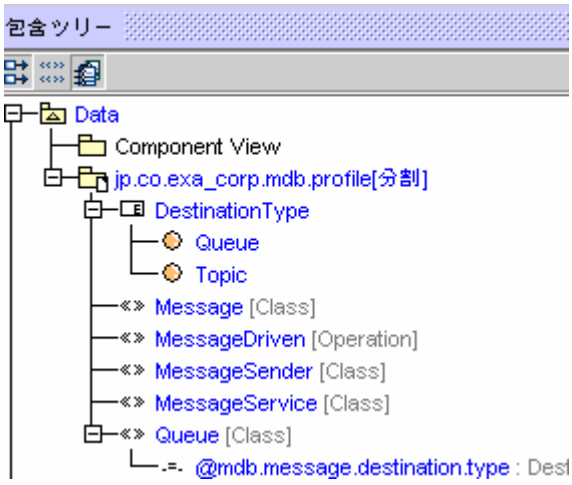


図５－１９．パッケージの共有後

パッケージの共有化が完了したので、プロジェクトを保存する。尚、保存ダイアログでは、「ファイルタイプ」を「拡張可能なマークアップランゲージ（*.xml）」にすること。もし、Xml ファイルで保存しない場合、AndroMDA のビルド時にエラーになってしまう。ここでは「mdb-profile.xml」という名前で保存する。

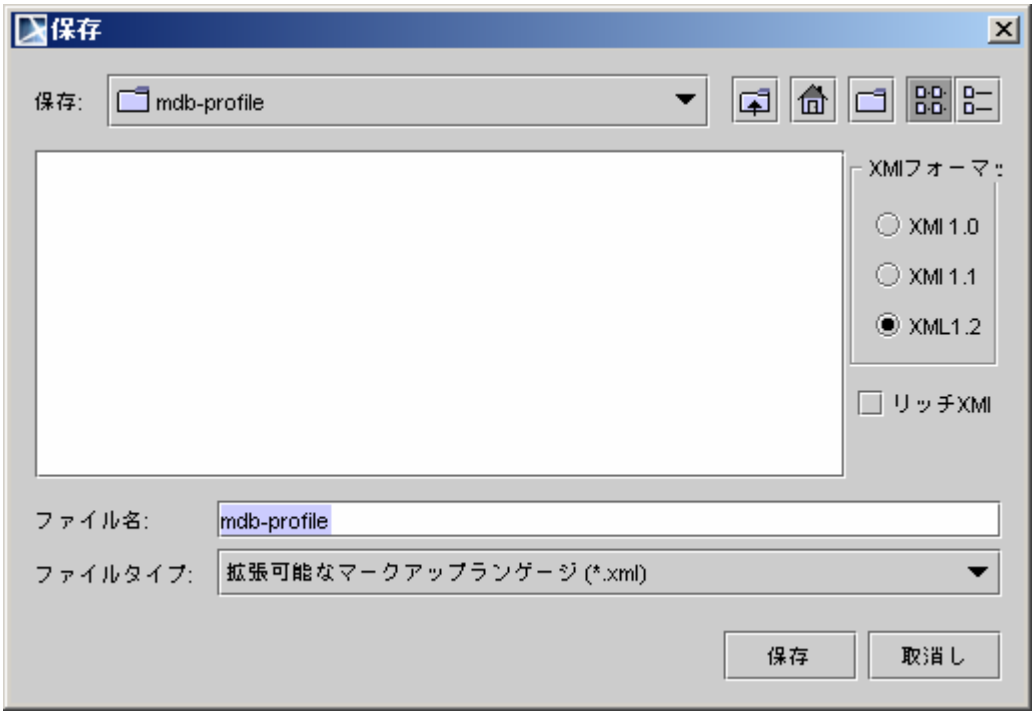


図５－２０．プロファイルの保存

５．２　プロファイルの適用方法

５．１にて独自プロファイルの作成ができた。ここでは作成した独自プロファイルをプロジェクトへ適用する方法を説明する。

５．２．１　　プロジェクトへの適用方法

まず、プロジェクトでの適用方法について説明する。適用方法自体は簡単である。独自プロファイル(*.xml ファイル)をプロジェクトの「**mda/src/uml**」以下にコピーする。これだけでプロジェクトに対しての適用は完了する。ここでは５．１で作成したプロファイル「**mdb-profile.xml**」をコピーしたものとする。

５．２．２　　UML モデリング時の適用方法

次に、MagicDraw でモデリングする際に独自プロファイルを適用する方法を説明する。適用したいプロジェクトの UML モデルを MagicDraw で読み込む。

読み込み後、「ファイル」メニューの「プロファイル／モジュールを使用」を選択する。

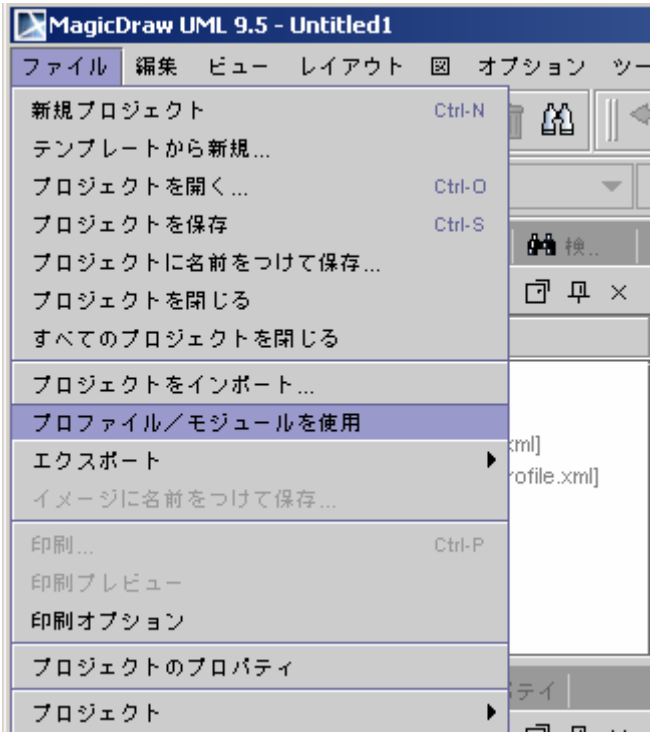


図５－２１．プロファイルの適用

表示される「プロファイル／モジュール使用」ダイアログの「モジュールパス」の右にある「...」ボタンを押下する。するとファイル(および／または)フォルダを選択するダイアログが表示されるので「追加」ボタンを押下し、独自プロファイルがあるフォルダを指定する。

指定するのは、５．２．１で独自プロファイルのプロジェクト適用をした「**mda/src/uml**」を指定する。すると、「パス変数を使用しますか？」という質問のダイアログが表示される。ここではパス変数を使用するので唯一選択できる「**<project.dir>**」を選択し、「選択を使用」ボタンを押下する。これでモジュールパスを選択したので「OK」ボタンを押下して確定する。

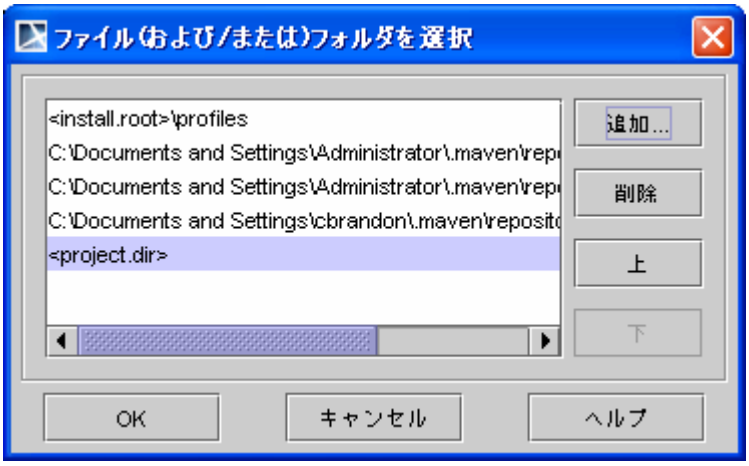


図５－２２．フォルダの選択

「プロファイル／モジュール使用」ダイアログの「モジュールパス」に追加したフォルダ(今回はパス「**<project.dir>**」)が表示されているはずである。このフォルダを選択すると、５．２．１でコピーしてある「**mdb-profile.xml**」が表示されるので選択し、「OK」ボタンを押下する。

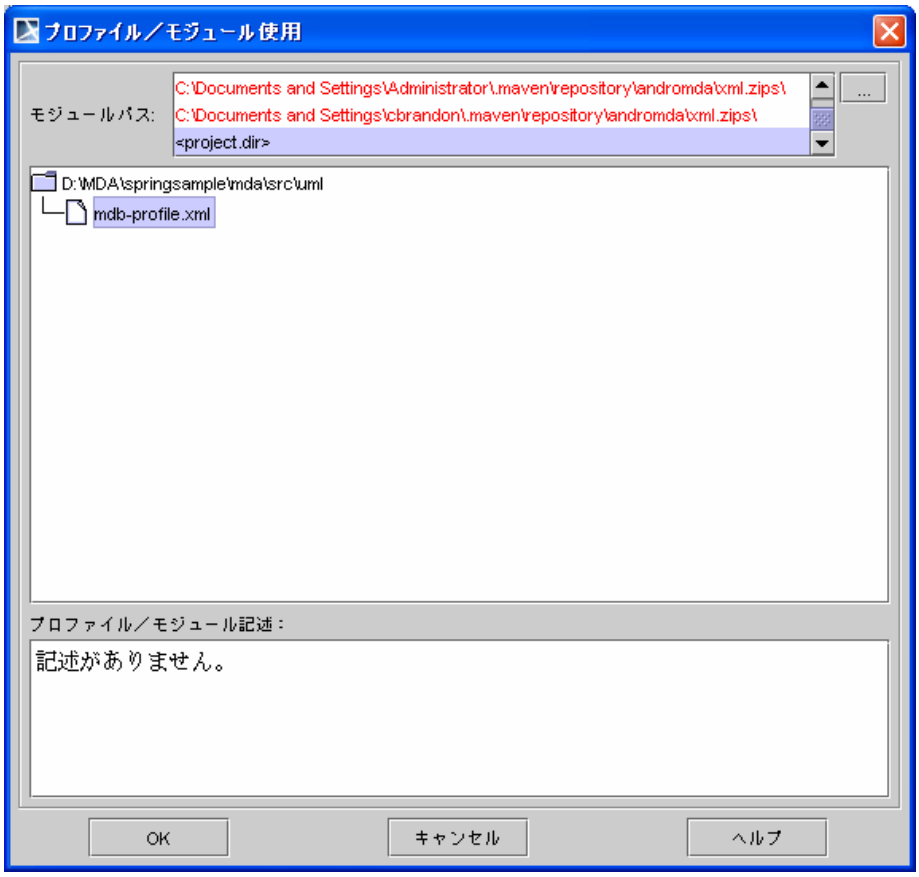


図５－２３．プロファイルの選択

これで、UML モデルに独自プロファイルの適用が完了する。

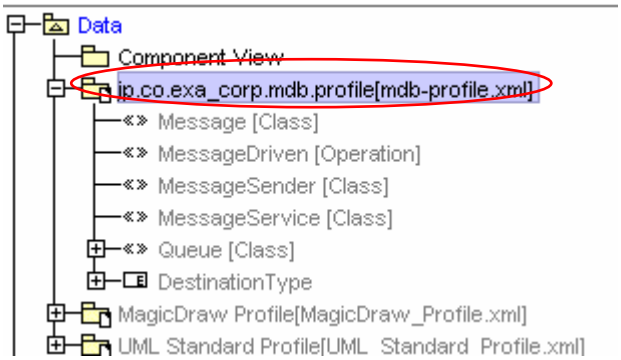


図５－２４．独自プロファイルの適用後

適用したプロファイルに定義されているステレオタイプやタグ付き値は、モデルに使用することができる。

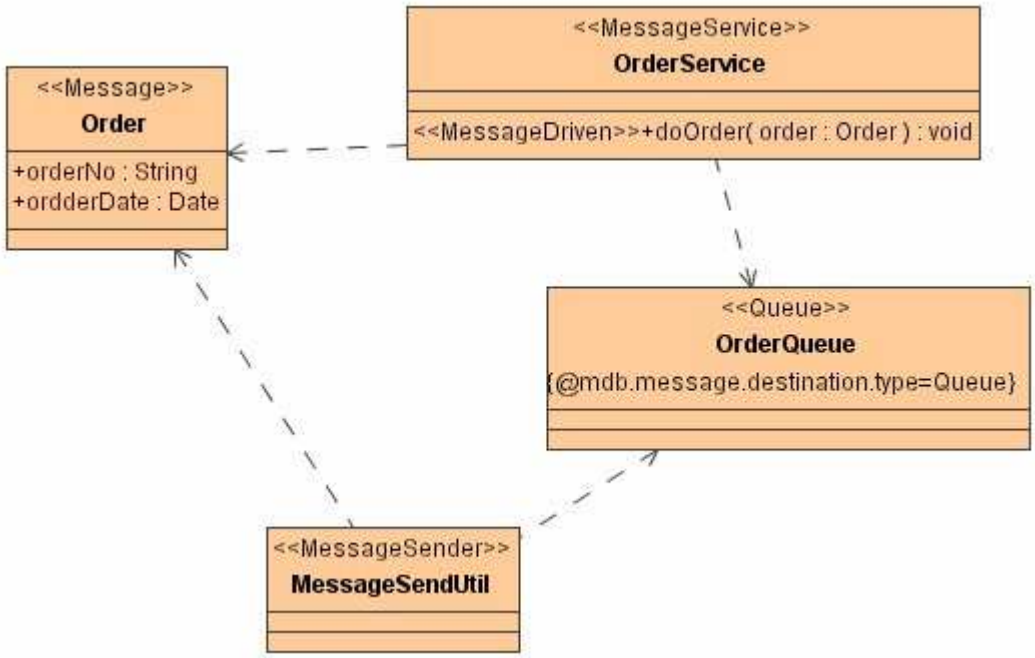


図５－２５．独自プロファイルのモデル適用

付録１ テンプレートファイルリスト

ここでは、AndroMDA や各カートリッジで使用されるテンプレートファイルについて簡単な対応表および、モデルとファイルの対応イメージを示す。尚、表中の「<<名前>>」はステレオタイプのことを表す。付録付録１－９．以降のものについてはカートリッジではなく AndroMDA プロジェクト作成時に利用されるテンプレートファイルである。

付録１－１． BPM4Struts カートリッジ

表１．BPM4Struts カートリッジテンプレートファイル一覧

テンプレートファイル	モデル	備考
Action.java.vsl	アクティビティ図上での遷移線	
ActionServlet.java.vsl	なし	
PatternMatchingExceptionHandler.java.vsl	なし	
StrutsValidator.java.vsl	なし	
CrudAction.java.vsl	<<Entity>>と<<Manageable>>をもつクラス	
	<<Entity>>と<<Manageable>>をもつクラスが参照している関連終端	
jboss-web.xml.vsl	なし	
menu-config.xml.vsl	なし	
struts-config.xml.vsl	<<FrontEndUseCase>> または <<FrontEndApplication>>をもつユースケース	ユースケースは、アクティビティ図の情報も含んでいる
	<<Entity>>と<<Manageable>>をもつクラス	
	<<Entity>>と<<Manageable>>をもつクラスが参照している関連終端	
tiles-defs.xml.vsl	なし	
validation.xml.vsl	<<FrontEndUseCase>> または <<FrontEndApplication>>をもつユースケース	ユースケースは、アクティビティ図の情報も含んでいる
validator-rules.xml.vsl	<<FrontEndUseCase>> または <<FrontEndApplication>>をもつユースケース	ユースケースは、アクティビティ図の情報も含んでいる
web.xml.vsl	<<FrontEndUseCase>> または <<FrontEndApplication>>をもつユースケース	ユースケースは、アクティビティ図の情報も含んでいる
	<<Entity>>と<<Manageable>>をもつクラス	
	<<Entity>>と<<Manageable>>をもつクラスが参照している関連終端	
struts-admin.xml.vsl	なし	
validation-admin.xml.vsl	なし	

Controller.java.vsl	アクティビティ図にアサインしているクラス	
ControllerFactory.java.vsl	アクティビティ図にアサインしているクラス	
ControllerImpl.java.vsl	アクティビティ図にアサインしているクラス	
SessionObject.java.vsl	<<FrontEndSessionObject>>をもつクラス	
TableDecorator.java.vsl	タグ付き値「@andromda.struts.view.table.decorator」が"true"の遷移パラメータ	
ActionForm.java.vsl	アクティビティ図上での遷移線	
ActionFormInterface.java.vsl	アクティビティ図にアサインしているクラスの操作	
CrudForm.java.vsl	<<Entity>>と<<Manageable>>をもつクラス	
	<<Entity>>と<<Manageable>>をもつクラスが参照している関連終端	
application-resources.properties.vsl	<<FrontEndUseCase>> または <<FrontEndApplication>>をもつユースケース	ユースケースは、アクティビティ図の情報も含んでいる
	<<Entity>>と<<Manageable>>をもつクラス	
	<<Entity>>と<<Manageable>>をもつクラスが参照している関連終端	
displaytag.properties.vsl	なし	
action.jsp.vsl	アクティビティ図上での遷移線	
application-help.jsp.vsl	<<FrontEndUseCase>> または <<FrontEndApplication>>をもつユースケース	ユースケースは、アクティビティ図の情報も含んでいる
breadcrumbs.jsp.vsl	なし	
calendar-js.jsp.vsl	なし	
default-application.css.vsl	なし	
form-validation.jsp.vsl	<<FrontEndUseCase>> または <<FrontEndApplication>>をもつユースケース	ユースケースは、アクティビティ図の情報も含んでいる
help-layout.jsp.vsl	なし	
index.jsp.vsl	<<FrontEndUseCase>> または <<FrontEndApplication>>をもつユースケース	ユースケースは、アクティビティ図の情報も含んでいる
	<<Entity>>と<<Manageable>>をもつクラス	
	<<Entity>>と<<Manageable>>をもつクラスが参照している関連終端	
login-form.jsp.vsl	なし	

main-layout.jsp.vsl	なし	
menu.jsp.vsl	<<FrontEndUseCase>> または <<FrontEndApplication>>をもつユースケース	ユースケースは、アクティビティ図の情報も含んでいる
	<<Entity>>と<<Manageable>>をもつクラス	
	<<Entity>>と<<Manageable>>をもつクラスが参照している関連終端	
messages.jsp.vsl	なし	
page-help.jsp.vsl	<<FrontEndView>>を持つアクション状態	
page-variables.jspf.vsl	<<FrontEndView>>を持つアクション状態	
page.css.vsl	<<FrontEndView>>を持つアクション状態	
page.jsp.vsl	<<FrontEndView>>を持つアクション状態	
taglib-imports.jspf.vsl	なし	
usecase-help.jsp.vsl	なし	
ActionForm.java.vsl	アクティビティ図上での遷移線	
ActionFormInterface.java.vsl	アクティビティ図にアサインしているクラスの操作	
CrudForm.java.vsl	<<Entity>>と<<Manageable>>をもつクラス	
	<<Entity>>と<<Manageable>>をもつクラスが参照している関連終端	
application-resources.properties.vsl	<<FrontEndUseCase>> または <<FrontEndApplication>>をもつユースケース	ユースケースは、アクティビティ図の情報も含んでいる
	<<Entity>>と<<Manageable>>をもつクラス	
	<<Entity>>と<<Manageable>>をもつクラスが参照している関連終端	
displaytag.properties.vsl	なし	
action.jsp.vsl	アクティビティ図上での遷移線	
application-help.jsp.vsl	<<FrontEndUseCase>> または <<FrontEndApplication>>をもつユースケース	ユースケースは、アクティビティ図の情報も含んでいる
breadcrumbs.jsp.vsl	なし	
calendar-js.jsp.vsl	なし	
default-application.css.vsl	なし	
default-manageable.css.vsl	<<Entity>>と<<Manageable>>をもつクラス	

	<<Entity>>と<<Manageable>>をもつクラスが参照している関連終端	
entity-switcher.jsp.vsl	<<Entity>>と<<Manageable>>をもつクラス	
	<<Entity>>と<<Manageable>>をもつクラスが参照している関連終端	
manageable.jsp.vsl	<<Entity>>と<<Manageable>>をもつクラス	
	<<Entity>>と<<Manageable>>をもつクラスが参照している関連終端	
roles.properties.vsl	<<FrontEndUseCase>> または <<FrontEndApplication>>をもつユースケース	ユースケースは、アクティビティ図の情報も含んでいる
default-manageable.css.vsl	<<Entity>>と<<Manageable>>をもつクラス	
	<<Entity>>と<<Manageable>>をもつクラスが参照している関連終端	
users.properties.vsl	なし	
Action.java.vm	ファイル生成で利用される Velocity マクロ	
populateCurrentFormOnError.vm	ファイル生成で利用される Velocity マクロ	
page.jsp.vm	ファイル生成で利用される Velocity マクロ	

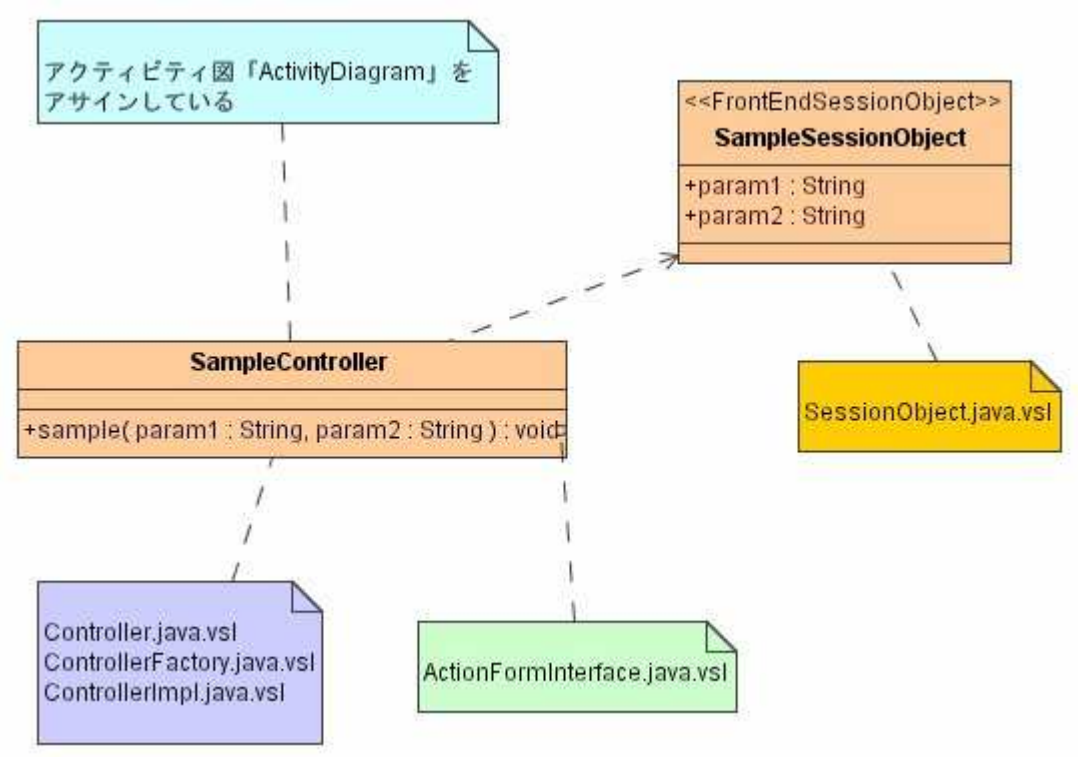


図１．プレゼンテーション側のクラス図とテンプレートファイルの対応図

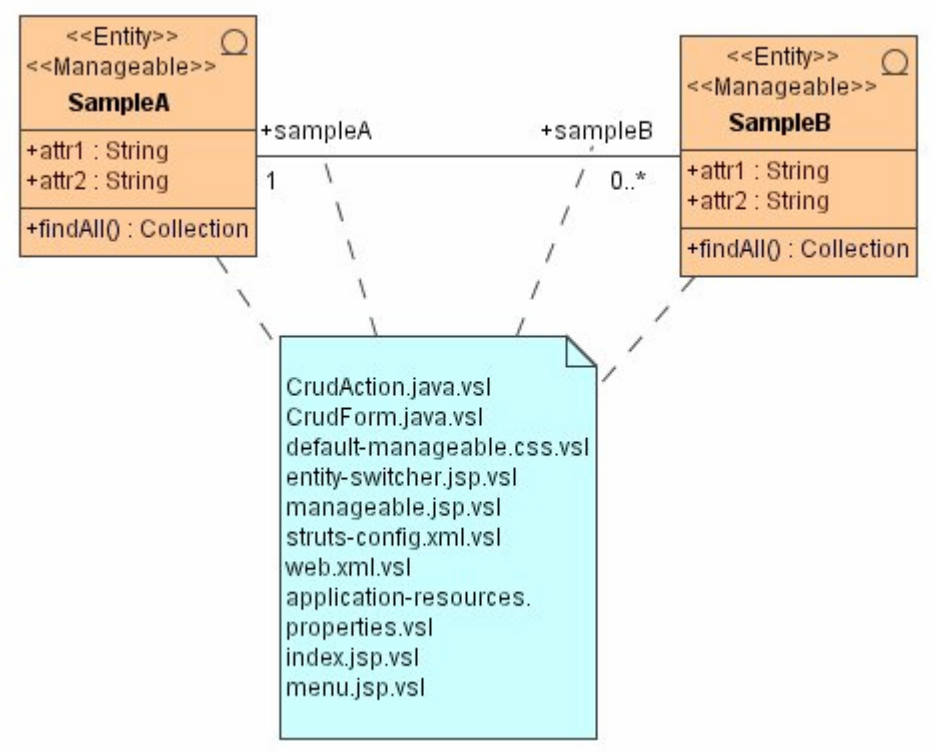


図２．サーバ側のクラス図とテンプレートファイルの対応図

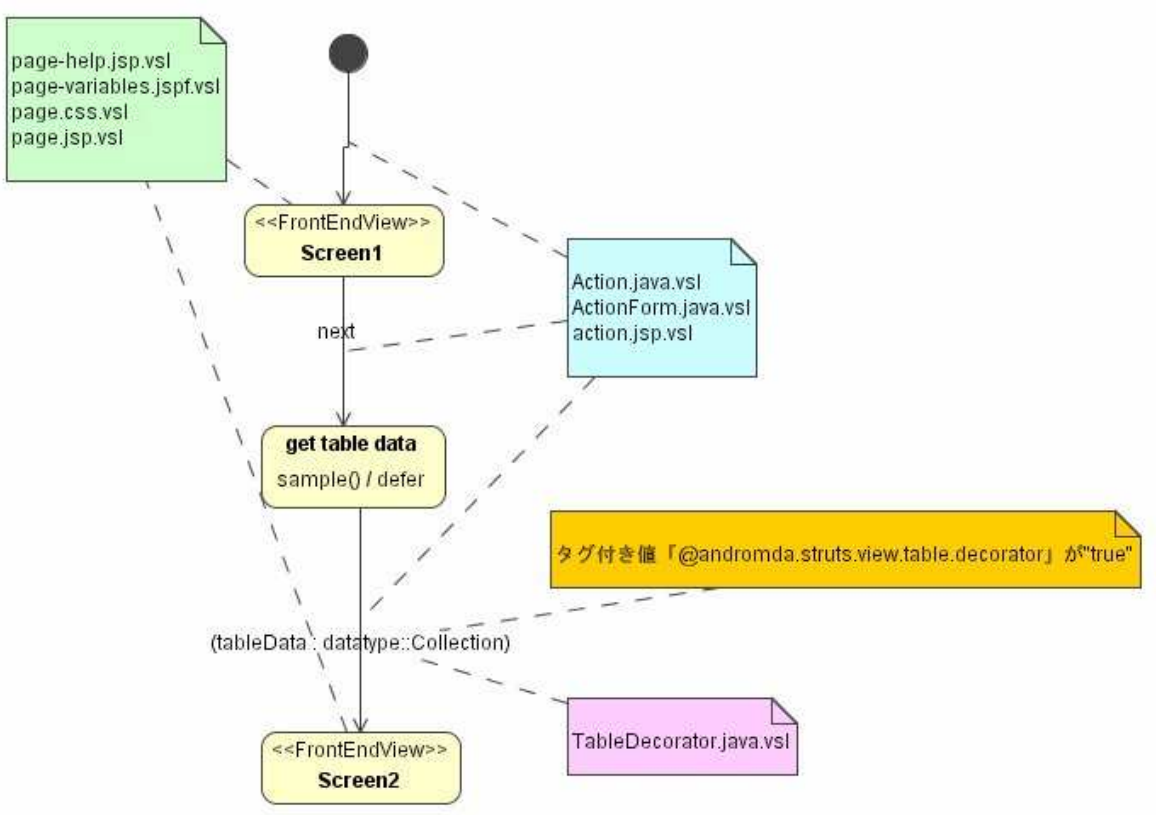


図３．プレゼンテーション側のアクティビティ図とテンプレートファイルの対応図

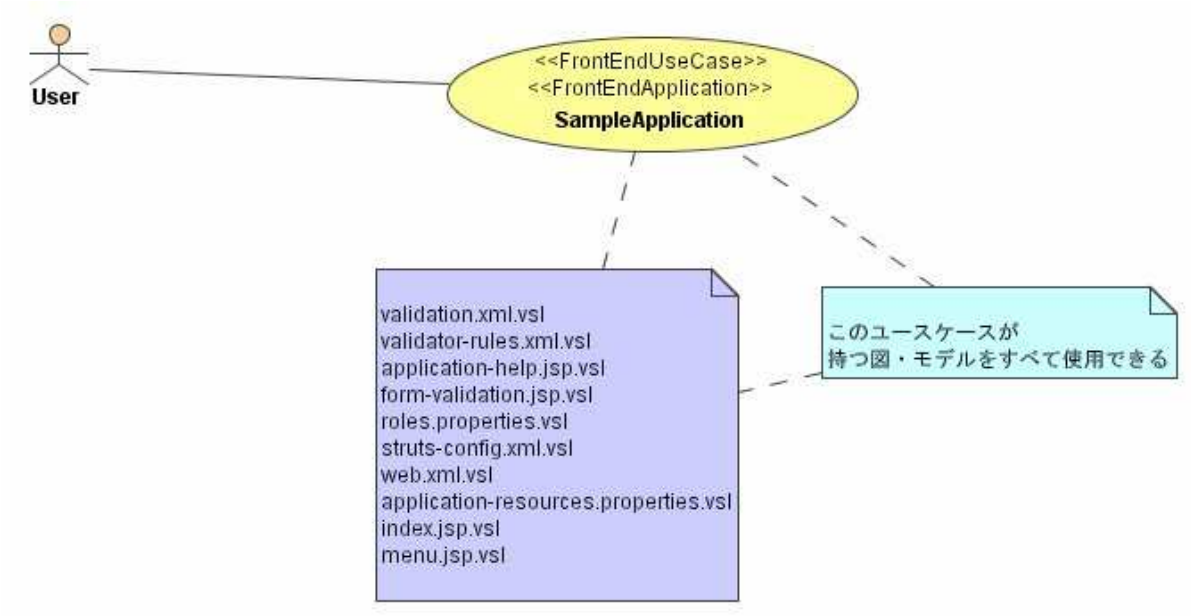


図４．ユースケースとテンプレートファイルの対応図

付録１－２． EJB カートリッジ

表２．EJB カートリッジのテンプレートファイル一覧

テンプレートファイル	モデル	備考
ejb-jar.xml.vsl	なし	
EntityBean.vsl	<<Entity>>がついているクラス	クラスが持つ属性や操作も利用される
	<<Entity>>がついているクラスが参照しているクラスの関連終端	
	<<Entity>>がついているクラスが参照しているクラスの関連	
EntityBeanImpl.vsl	<<Entity>>がついているクラス	クラスが持つ属性や操作も利用される
	<<Entity>>がついているクラスが参照しているクラスの関連終端	
	<<Entity>>がついているクラスが参照しているクラスの関連	
EntityHome.vsl	<<Entity>>がついているクラス	クラスが持つ属性や操作も利用される
	<<Entity>>がついているクラスが参照しているクラスの関連終端	
	<<Entity>>がついているクラスが参照しているクラスの関連	
EntityLocal.vsl	<<Entity>>がついているクラス	クラスが持つ属性や操作も利用される
	<<Entity>>がついているクラスが参照しているクラスの関連終端	
	<<Entity>>がついているクラスが参照しているクラスの関連	
jboss.xml.vsl	なし	
SessionBean.vsl	<<Service>>がついているクラス	クラスが持つ属性や操作も利用される
SessionBeanImpl.vsl	<<Service>>がついているクラス	クラスが持つ属性や操作も利用される
SessionLocal.vsl	<<Service>>がついているクラス	クラスが持つ属性や操作も利用される
SessionLocalHome.vsl	<<Service>>がついているクラス	クラスが持つ属性や操作も利用される
SessionRemote.vsl	<<Service>>がついているクラス	クラスが持つ属性や操作も利用される
SessionRemoteHome.vsl	<<Service>>がついているクラス	クラスが持つ属性や操作も利用される
ValueObject.vsl	<<ValueObject>>がついているクラス	
Globals.vm	ファイル生成で利用される Velocity マクロ	

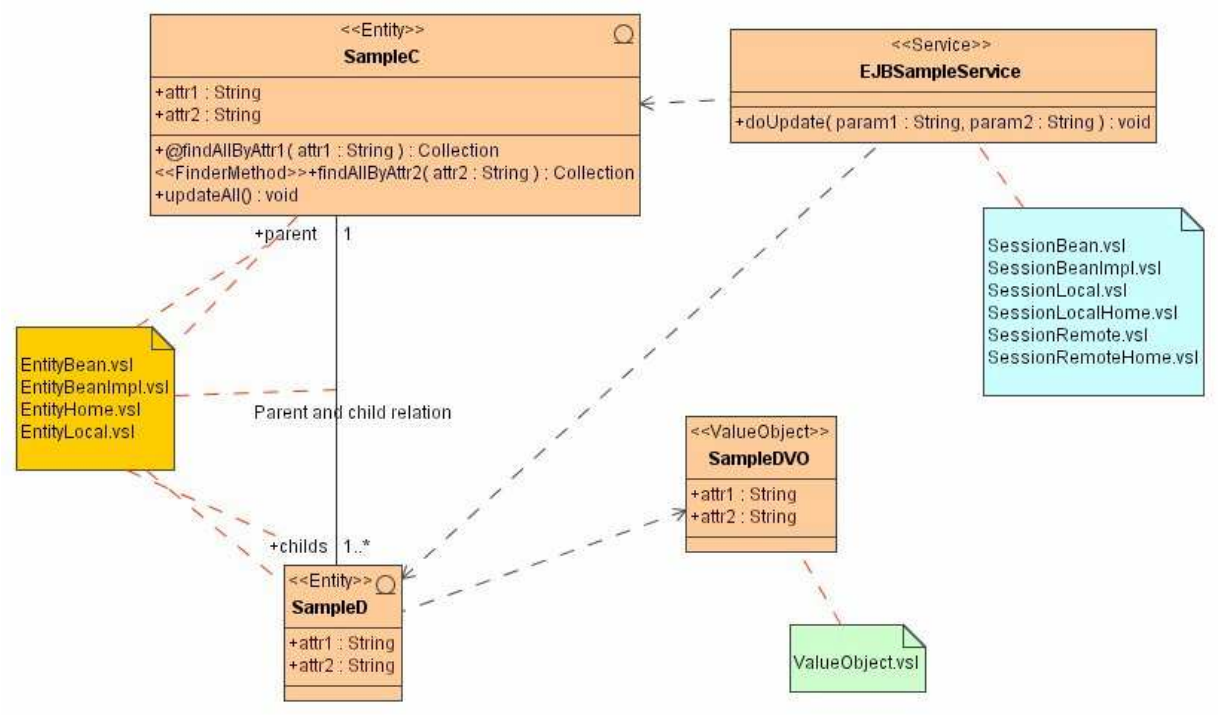


図５．サーバ側のクラス図とテンプレートファイルの対応図

付録１－３． Hibernate カートリッジ

表３．Hibernate カートリッジのテンプレートファイル一覧

テンプレートファイル	モデル	備考
ehcache.xml.vsl	<<Entity>>がついているクラス	クラスが持つ属性や操作も利用される
	<<Entity>>がついているクラスが参照しているクラスの関連終端	
	<<Entity>>がついているクラスが参照しているクラスの関連	
hibernate.cfg.xml.vsl	<<Entity>>がついているクラス	クラスが持つ属性や操作も利用される
	<<Entity>>がついているクラスが参照しているクラスの関連終端	
	<<Entity>>がついているクラスが参照しているクラスの関連	
hibernate.hbm.xml.vsl	<<Entity>>がついているクラス	クラスが持つ属性や操作も利用される
	<<Entity>>がついているクラスが参照しているクラスの関連終端	
	<<Entity>>がついているクラスが参照しているクラスの関連	
HibernateEmbeddedValue.vsl	<<EmbeddedValue>>がついているクラス	クラスが持つ属性も利用される
HibernateEmbeddedValueImpl.vsl	<<EmbeddedValue>>がついているクラス	クラスが持つ属性も利用される
HibernateEntity.vsl	<<Entity>>がついているクラス	クラスが持つ属性や操作も利用される
	<<Entity>>がついているクラスが参照しているクラスの関連終端	
	<<Entity>>がついているクラスが参照しているクラスの関連	
HibernateEntityImpl.vsl	<<Entity>>がついているクラス	クラスが持つ属性や操作も利用される
	<<Entity>>がついているクラスが参照しているクラスの関連終端	

	<<Entity>>がついているクラスが参照しているクラスの関連	
HibernateEntityImplManual.vsl	<<Entity>>がついているクラス	クラスが持つ属性や操作も利用される
	<<Entity>>がついているクラスが参照しているクラスの関連終端	
	<<Entity>>がついているクラスが参照しているクラスの関連	
HibernateEnumeration.vsl	<<Enumeration>>がついているクラス	
ejb-jar.xml.vsl	<<Service>>がついているクラス	クラスが持つ操作も利用される
HibernateEntityFactory.vsl	<<Entity>>がついているクラス	クラスが持つ属性や操作も利用される
	<<Entity>>がついているクラスが参照しているクラスの関連終端	
	<<Entity>>がついているクラスが参照しているクラスの関連	
HibernateSession.vsl	<<Service>>がついているクラス	クラスが持つ操作も利用される
HibernateSessionBean.vsl	<<Service>>がついているクラス	クラスが持つ操作も利用される
HibernateSessionBeanImpl.vsl	<<Service>>がついているクラス	クラスが持つ操作も利用される
HibernateSessionEJBLocator.vsl	<<Service>>がついているクラス	クラスが持つ操作も利用される
HibernateSessionHome.vsl	<<Service>>がついているクラス	クラスが持つ操作も利用される
HibernateUtils.vsl	なし	
jboss.xml.vsl	<<Service>>がついているクラス	クラスが持つ操作も利用される
HibernateByteBlobType.vsl	なし	
HibernateStringClobType.vsl	なし	
hibernate.hbm.xml.vm	ファイル生成で利用される Velocity マクロ	
hibernate.java.vm	ファイル生成で利用される Velocity マクロ	
HibernateSessionEJBGlobals.vm	ファイル生成で利用される Velocity マクロ	

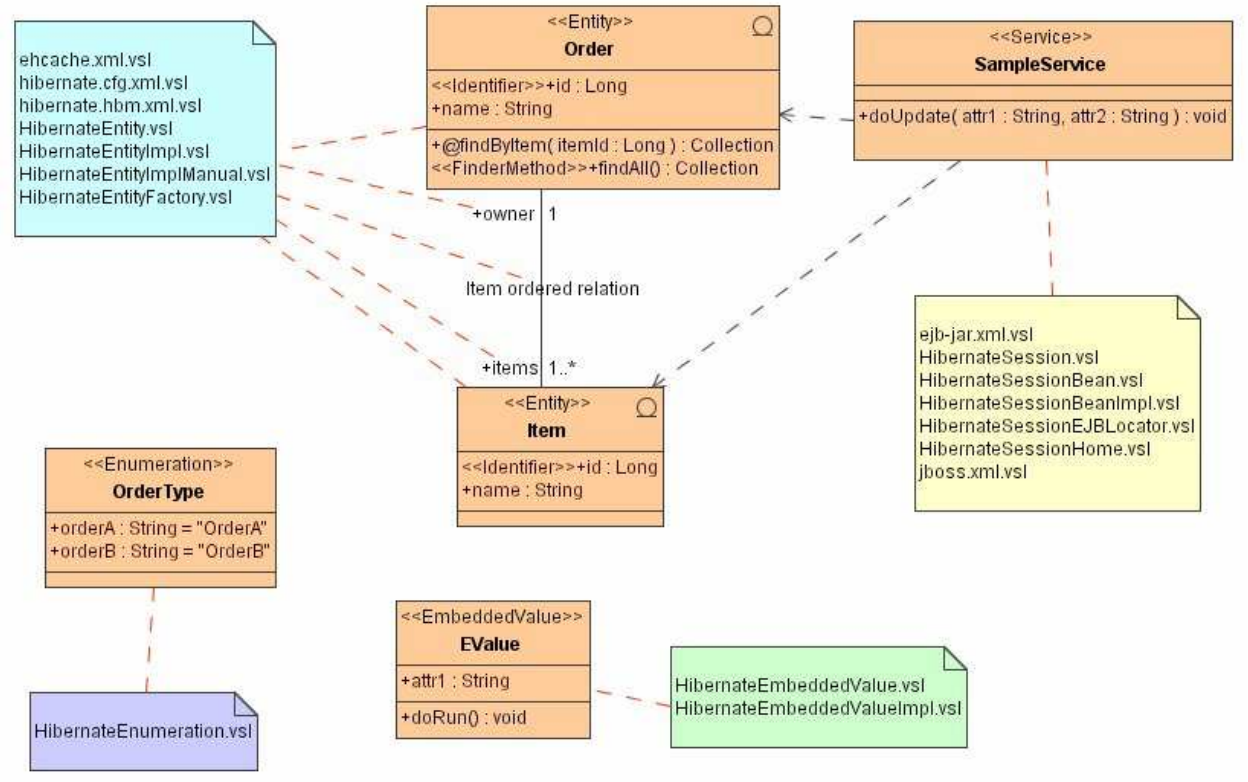


図６．サーバ側のクラス図とテンプレートファイルの対応図

付録１－４．　Java カートリッジ

表４．Java カートリッジのテンプレートファイル一覧

テンプレートファイル	モデル	備考
ApplicationException.vsl	<<Exception>> または <<ApplicationException>>がついているクラス	
Enumeration.vsl	<<Enumeration>>がついているクラス	
Service.vsl	<<Service>>がついているクラス	クラスが持つ操作も利用される
ServiceImpl.vsl	<<Service>>がついているクラス	クラスが持つ操作も利用される
UnexpectedException.vsl	<<UnexpectedException>>がついているクラス	
ValueObject.vsl	<<ValueObject>>がついているクラス	
ExceptionUtils.vm	ファイル生成で利用される Velocity マクロ	
ExceptionUtilsImports.vm	ファイル生成で利用される Velocity マクロ	

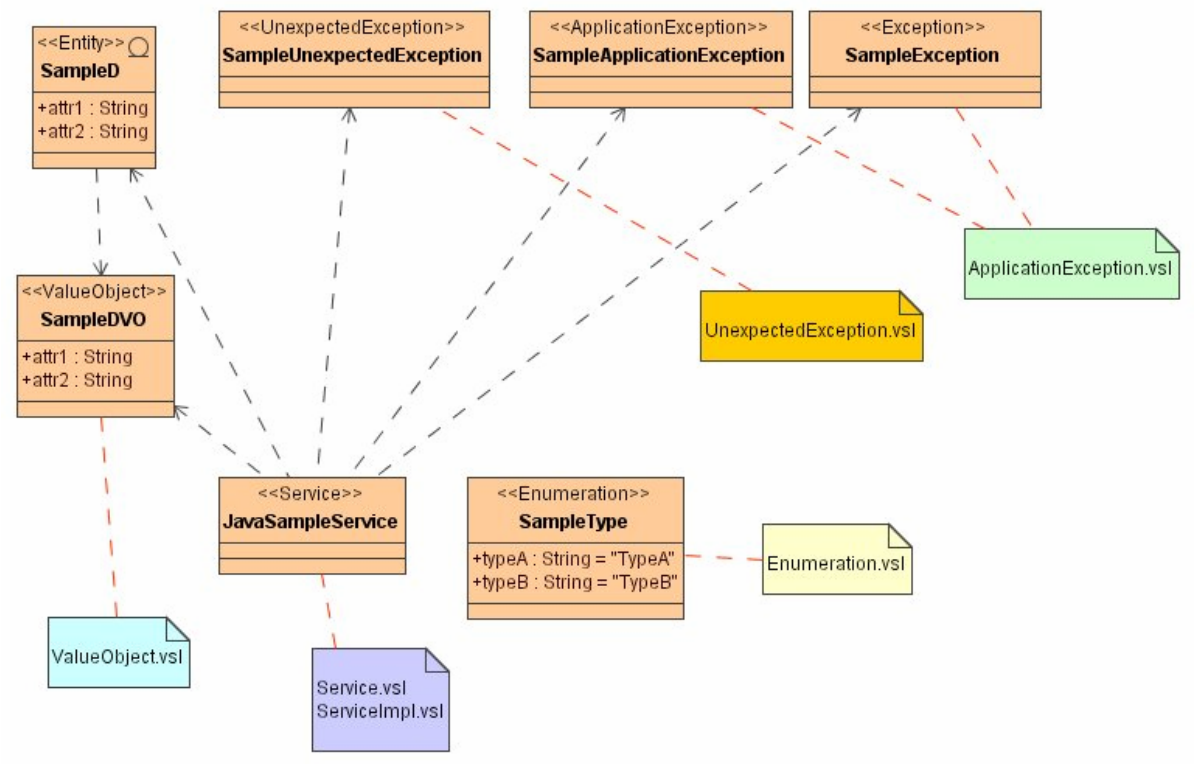


図７．サーバ側クラスとテンプレートファイル対応図

付録１－５．　Meta カートリッジ

表５．Meta カートリッジのテンプレートファイル一覧

テンプレートファイル	モデル	備考
Metafacade.vsl	<<Metafacade>>を持つクラス	クラスの属性や操作も利用される
	<<Metafacade>>を持つクラスが参照する関連終端	
	<<Metafacade>>を持つクラスの継承関連	
MetafacadeLogic.vsl	<<Metafacade>>を持つクラス	クラスの属性や操作も利用される
	<<Metafacade>>を持つクラスが参照する関連終端	
	<<Metafacade>>を持つクラスの継承関連	
MetafacadeLogicImpl.vsl	<<Metafacade>>を持つクラス	クラスの属性や操作も利用される
	<<Metafacade>>を持つクラスが参照する関連終端	
	<<Metafacade>>を持つクラスの継承関連	
MetafacadesToImpls.vsl	<<Metafacade>>を持つクラス	クラスの属性や操作も利用される
	<<Metafacade>>を持つクラスが参照する関連終端	
	<<Metafacade>>を持つクラスの継承関連	
RenderTranslation.vm	ファイル生成で利用される Velocity マクロ	

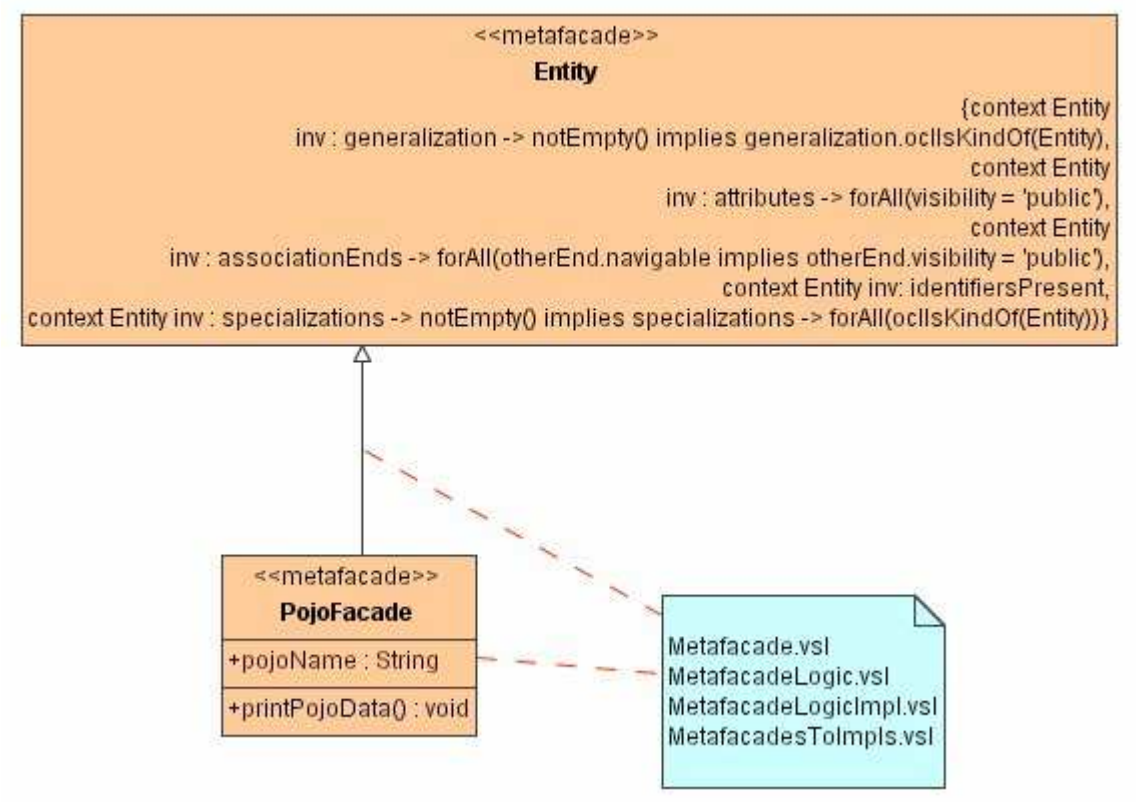


図８．クラス図とテンプレートファイルの対応図

付録１－６． Spring カートリッジ

表６．Spring カートリッジのテンプレートファイル一覧

テンプレートファイル	モデル	備考
applicationContext-dataSource.xml.vsl	<<Entity>>がついているクラス	クラスの操作も利用される
	<<Service>>がついているクラス	
applicationContext.xml.vsl	<<Entity>>がついているクラス	クラスの操作も利用される
	<<Service>>がついているクラス	
	<<Entity>>と<<Manageable>>がついているクラス	
	<<Entity>>と<<Manageable>>がついているクラスが参照している関連終端	
beanRefFactory.xml.vsl	<<Entity>>がついているクラス	クラスの操作も利用される
	<<Service>>がついているクラス	
	<<Entity>>と<<Manageable>>がついているクラス	
	<<Entity>>と<<Manageable>>がついているクラスが参照している関連終端	
clientContext.xml.vsl	<<Service>>がついているクラス	クラスの操作も利用される
DefaultServiceException.vsl	<<Service>>がついているクラス	クラスの操作も利用される
serverContext.xml.vsl	<<Service>>がついているクラス	クラスの操作も利用される
SpringClientExceptionHandlingAdvice.vsl	<<Service>>がついているクラス	クラスの操作も利用される
SpringClientServiceLocator.vsl	<<Service>>がついているクラス	クラスの操作も利用される
SpringDao.vsl	<<Entity>>がついているクラス	クラスの操作も利用される
SpringPrincipalStore.java.vsl	なし	
SpringService.vsl	<<Service>>がついているクラス	クラスの操作も利用される
SpringServiceBase.vsl	<<Service>>がついているクラス	クラスの操作も利用される
SpringServiceImpl.vsl	<<Service>>がついているクラス	クラスの操作も利用される
SpringServiceLocator.vsl	<<Service>>がついているクラス	クラスの操作も利用される
SpringWebServiceDelegator.vsl	なし	

applicationContext-manageable.xml.vsl	<<Entity>>と<<Manageable>>がついているクラス	
	<<Entity>>と<<Manageable>>がついているクラスが参照している関連終端	
SpringCrudDao.vsl	<<Entity>>と<<Manageable>>がついているクラス	
	<<Entity>>と<<Manageable>>がついているクラスが参照している関連終端	
SpringCrudDaoBase.vsl	<<Entity>>と<<Manageable>>がついているクラス	
	<<Entity>>と<<Manageable>>がついているクラスが参照している関連終端	
SpringCrudService.vsl	<<Entity>>と<<Manageable>>がついているクラス	
	<<Entity>>と<<Manageable>>がついているクラスが参照している関連終端	
SpringCrudServiceBase.vsl	<<Entity>>と<<Manageable>>がついているクラス	
	<<Entity>>と<<Manageable>>がついているクラスが参照している関連終端	
SpringCrudServiceLocator.vsl	<<Entity>>と<<Manageable>>がついているクラス	
	<<Entity>>と<<Manageable>>がついているクラスが参照している関連終端	
SpringCrudValueObject.vsl	<<Entity>>と<<Manageable>>がついているクラス	
	<<Entity>>と<<Manageable>>がついているクラスが参照している関連終端	
ejb-jar.xml.vsl	<<Service>>がついているクラス	クラスの操作も利用される
jboss.xml.vsl	<<Service>>がついているクラス	クラスの操作も利用される

SpringSession.vsl	<<Service>>がついているクラス	クラスの操作も利用される
SpringSessionBean.vsl	<<Service>>がついているクラス	クラスの操作も利用される
SpringSessionEJBLocator.vsl	<<Service>>がついているクラス	クラスの操作も利用される
SpringSessionHome.vsl	<<Service>>がついているクラス	クラスの操作も利用される
HibernateSearch.vsl	<<Entity>>がついているクラス	クラスの操作も利用される
HibernateSearchConfiguration.vsl	<<Entity>>がついているクラス	クラスの操作も利用される
HibernateSearchParameter.vsl	<<Entity>>がついているクラス	クラスの操作も利用される
SearchCriteria.vsl	<<Criteria>>がついているクラス	
SpringHibernateDaoBase.vsl	<<Entity>>がついているクラス	クラスの操作も利用される
SpringHibernateDaoImpl.vsl	<<Entity>>がついているクラス	クラスの操作も利用される
SpringHibernateDaoImplManual.vsl	<<Entity>>がついているクラス	クラスの操作も利用される
RenderPreconditions.vm	ファイル生成で利用される Velocity マクロ	
SpringGlobals.vm	ファイル生成で利用される Velocity マクロ	
SpringSessionEJBGlobals.vm	ファイル生成で利用される Velocity マクロ	

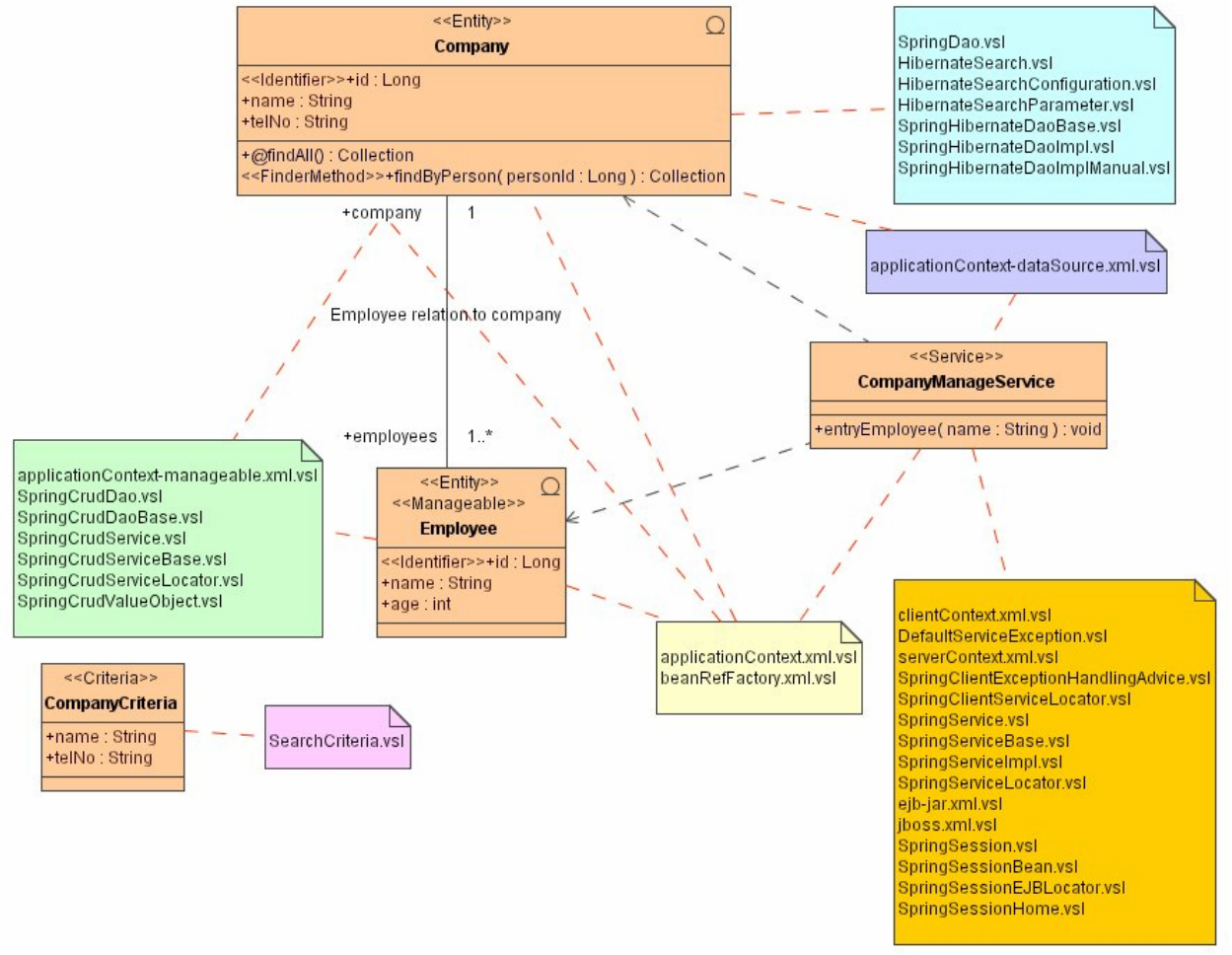


図９．クラス図とテンプレートファイルの対応図

付録１－７． WebService カートリッジ

表７．WebService カートリッジのテンプレートファイル一覧

テンプレートファイル	モデル	備考
jboss-web.xml.vsl	<<Service>>または<<WebService>>がついているクラス	クラスの操作も利用される
server-config.wsdd.vsl	<<Service>>または<<WebService>>がついているクラス	クラスの操作も利用される
web.xml.vsl	<<Service>>または<<WebService>>がついているクラス	クラスの操作も利用される
WebServiceTest.vsl	<<Service>>または<<WebService>>がついているクラス	クラスの操作も利用される
WebServiceTestImpl.vsl	<<Service>>または<<WebService>>がついているクラス	クラスの操作も利用される
WebServiceTestServiceLocator.vsl	<<Service>>または<<WebService>>がついているクラス	クラスの操作も利用される
crypto.properties.vsl	<<Service>>または<<WebService>>がついているクラス	クラスの操作も利用される
wrapped-wsdl.vsl	<<Service>>または<<WebService>>がついているクラス	クラスの操作も利用される
EJB-Globals.vm	ファイル生成で利用される Velocity マクロ	
Globals.vm	ファイル生成で利用される Velocity マクロ	
RPC-Globals.vm	ファイル生成で利用される Velocity マクロ	
AxisGlobals.vm	ファイル生成で利用される Velocity マクロ	
EJB-provider.vm	ファイル生成で利用される Velocity マクロ	
RPC-provider.vm	ファイル生成で利用される Velocity マクロ	

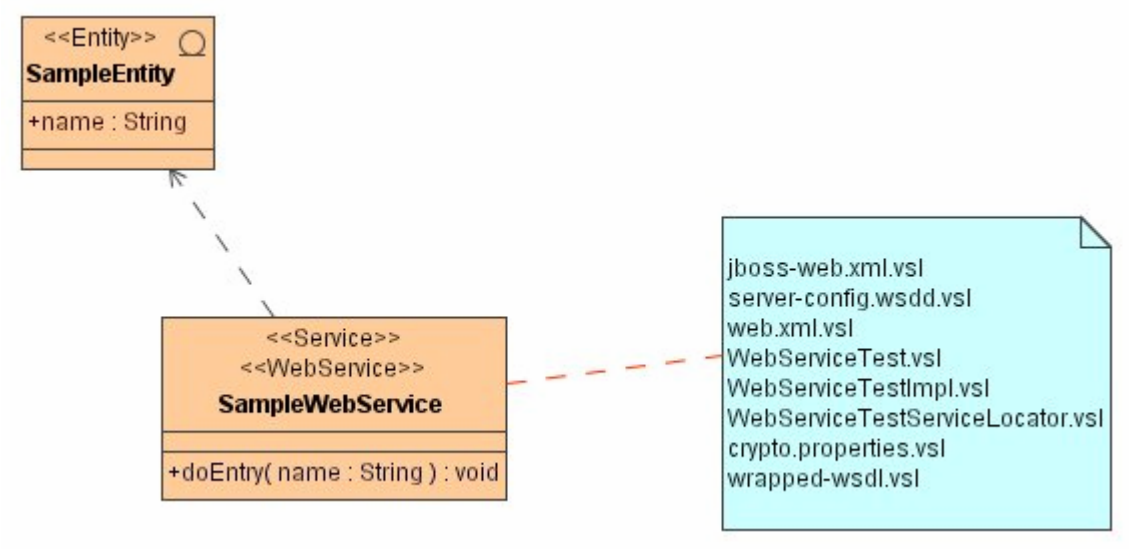


図１０．クラス図とテンプレートファイルの対応図

付録１－８．　Xmlschema カートリッジ

表８．Xmlschema カートリッジのテンプレートファイル一覧

テンプレートファイル	モデル	備考
XmlSchema.vsl	<<XmlSchemaType>>を持つクラス	クラスの属性も利用される

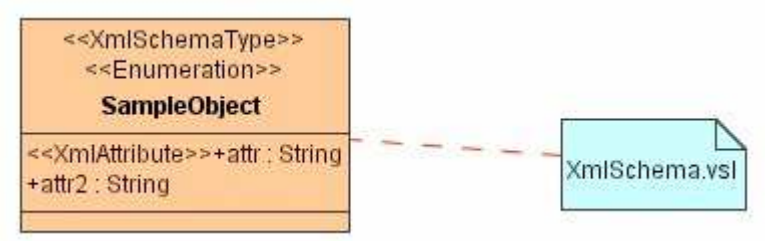


図１１．クラス図とテンプレートファイルの対応図

付録１－９． Ant

表９．Ant で AndroMDA プロジェクトの生成に利用されるテンプレートファイル一覧

templates ディレクトリ以下のファイルパス	備考
build.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/build.properties.vsl	プロジェクトのプロパティを参照する
j2ee-app/build.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/jboss.properties.vsl	プロジェクトのプロパティを参照する
j2ee-app/readme.txt.vsl	プロジェクトのプロパティを参照する
j2ee-app/app/build.properties.vsl	プロジェクトのプロパティを参照する
j2ee-app/app/build.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/app/src/META-INF/application.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/app/src/META-INF/jboss-app.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/common/build.properties.vsl	プロジェクトのプロパティを参照する
j2ee-app/common/build.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/ejb/build.properties.vsl	プロジェクトのプロパティを参照する
j2ee-app/ejb/build.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/ejb/src/META-INF/MANIFEST.MF.vsl	プロジェクトのプロパティを参照する
j2ee-app/hibernate/build.properties.vsl	プロジェクトのプロパティを参照する
j2ee-app/hibernate/build.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/hibernate/db/build.xml.vsl	プロジェクトのプロパテ

	ィを参照する
j2ee-app/hibernate/db/src/java/org/andromda/ant/hibernate/db/PersistenceUtil.java.vsl	プロジェクトのプロパティを参照する
j2ee-app/hibernate/db/src/META-INF/jboss-service.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/hibernate/db/src/xml/hibernate.cfg.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/hibernate/ejb/build.properties.vsl	プロジェクトのプロパティを参照する
j2ee-app/hibernate/ejb/build.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/hibernate/ejb/src/META-INF/MANIFEST.MF.vsl	プロジェクトのプロパティを参照する
j2ee-app/mda/build.properties.vsl	プロジェクトのプロパティを参照する
j2ee-app/mda/build.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/web/build.properties.vsl	プロジェクトのプロパティを参照する
j2ee-app/web/build.xml.vsl	プロジェクトのプロパティを参照する

付録１－１０．　Maven

表１０．Maven で AndroMDA プロジェクトの生成に利用されるテンプレートファイル一覧

templates ディレクトリ以下のファイルパス	備考
j2ee-app/build.properties.vsl	プロジェクトのプロパティを参照する
j2ee-app/eclipse-project.vsl	プロジェクトのプロパティを参照する
j2ee-app/maven.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/project.properties.vsl	プロジェクトのプロパティを参照する
j2ee-app/project.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/readme.txt.vsl	プロジェクトのプロパティを参照する
j2ee-app/app/project.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/app/src/META-INF/jboss-app.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/common/project.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/ejb/project.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/hibernate/project.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/mda/project.properties.vsl	プロジェクトのプロパティを参照する
j2ee-app/mda/project.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/mda/conf/mappings/Bpm4StrutsMergeMappings.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/spring/project.properties.vsl	プロジェクトのプロパティを参照する
j2ee-app/spring/project.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/web/maven.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/web/project.properties.vsl	プロジェクトのプロパティを参照する
j2ee-app/web/project.xml.vsl	プロジェクトのプロパティを参照する
j2ee-app/webservice/project.xml.vsl	プロジェクトのプロパティを参照する

付録１－１１．　Translation libraries

表１１．Translation libraries で利用されるテンプレートファイル一覧

テンプレートファイル	モデル	備考
EJB-QL.vsl	なし	プロジェクトのプロパティを参照する
Hibernate-QL.vsl	なし	プロジェクトのプロパティを参照する
Globals.vm	ファイル生成で利用される Velocity マクロ	
Patterns.vm	ファイル生成で利用される Velocity マクロ	

付録２ AndroMDA3.1 での変更点

本ガイドの２．３や２．４などで「**/mda/project.xml**」を編集しているが、AndroMDA3.1 では編集項目がすべて「**/mda/conf/andromda.xml**」ファイルへ移行されている。内容自体に違いはないので、本ガイドの内容を AndroMDA3.1 で実施する場合は、「**/mda/project.xml**」を「**/mda/conf/andromda.xml**」に置き換えてカスタマイズを行う必要がある。

付録３ andromda-cartridge.xml の詳細

このファイルはカートリッジが生成するテンプレートファイルと生成に利用する Metafacade クラス、およびプロパティなどを記述した設定ファイルである。ここでは各タグの詳細を記述する。

```
<cartridge name="ejb">
 <templateEngine>
  <!-- library of macros used in template engine -->
  <macrolibrary name="templates/EJB.vm"/>
 </templateEngine>
 <!-- define the template objects that are made availble to the template -->
 <templateObject
  name="str"
  className="org.andromda.core.common.StringUtilsHelper"
 />

 <resource path="templates/*.properties" outlet= "entity-beans" overwrite="true">
 </resource>

 <template
  path="templates/EntityBean.vsl"
  outputPattern="{0}/{1}Bean.java"
  outlet="entity-beans"
  overwrite="true"
 >
  <modelElements variable="entity">
   <modelElement>
    <type name="org.andromda.cartridges.ejb.metafacades.EJBEntity"/>
   </modelElement>
  </modelElements>
 </template>
...
</cartridge>
```

図１． andromda-cartridge.xml の記述例

- <cartridge/>

  このタグはこのカートリッジ設定ファイルのルート要素であり、カートリッジに名前をつける。

表１．<cartridge/>タグの属性

属性	内容	必須	備考
name	カートリッジの名前を指定する	○	

- <property/>

  このタグはカートリッジが参照するプロパティを設定する。このプロパティはテンプレートからファイル作成をする間参照可能である。

表２．<property/>タグの属性

属性	内容	必須	備考
reference	プロパティの名前を指定する	○	
default	カートリッジを利用するプロジェクトで値が設定されない場合のデフォルト値を指定する		

- <template/>

  このタグはファイルを生成する際に使用するテンプレートについて設定する。

表３．<template/>タグの属性

属性	内容	必須	備考
path	テンプレートファイル(*.vsl)のパスを指定する	○	
outputPattern	java.text.MessageFormat の構文で出力ファイルのファイル名パターンを指定する	○	
outlet	ファイルの出力先を論理名で指定する	○	論理名はプロパティで指定する
overwrite	ファイル生成時にすでにファイルが存在する場合の上書きの有無を指定する	○	true/false で指定する
generateEmptyFiles	テンプレートによるファイル生成が発生しなかった場合に、空のファイルを出力するかを指定する		true/false で指定する(デフォルトは false)
outputToSingleFile	生成するファイルは複数の Metafacade で１つであるかどうかを指定する		true/false で指定する(デフォルトは false)
outputOnEmptyElements	テンプレートファイルで利用する Metafacade クラスが存在しない場合に空のファイルを出力するかを指定する		・true/false で指定する(デフォルトは true) ・この属性は「outputToSingleFile」の値が「true」のときのみ有効になる。
required	このテンプレートでファイルを生成しない場合に警告を出力するかの有無を指定する		true/false で指定する(デフォルトは true)

- **<modelElements/>**

  このタグは<template/>の中で定義する。テンプレートで使用する Metafacade クラスの参照名を指定する。

表４．<modelElements/>タグの属性

属性	内容	必須	備考
variable	テンプレートが Metafacade を参照する際の値を指定する		

- **<modelElement/>**

  このタグは<modelElements/>の中で定義する。テンプレートで使用する Metafacade クラスを指定する。

表５．<modelElement/>タグの属性

属性	内容	必須	備考
stereotype	テンプレートで使用する Metafacade が持つステレオタイプを指定する	△	<type/>を指定しない限りは必須
variable	テンプレートが Metafacade を参照する際の値を指定する。ただし、<template/>の「outputToSingleFile」属性に「true」を指定したときのみ、この値は有効になる。		

- **<type/>**

  このタグは<modelElement>の中で定義する。テンプレートで使用する Metafacade クラスを指定する。

表６．<type/>タグの属性

属性	内容	必須	備考
name	テンプレートで使用する Metafacade クラスの完全修飾名を指定する。	△	<modelElement/> の「stereotype」属性を指定しない限りは必須

- **<property/>**

  このタグは<type/>タグの中で定義する。<type/>で指定している Metafacade クラスを使用するための条件をつける。属性の欄に「value」とあるが実際に属性があるわけではなく。下記のように開始タグと終了タグで囲んだ値のことを指す。

```
...
<modelElements variable="enumeration">
    <modelElement>
        <type name="org.andromda.metafacades.uml.ClassifierFacade">
            <property name="enumeration">true</property>
        </type>
    </modelElement>
</modelElements>
...
```

表７．<property/>タグの属性

属性	内容	必須	備考
name	Metafacade クラスにあるプロパティ名	○	
value	Metafacade クラスが有効かどうかを判定するために属性「name」で指定したプロパティの値を指定する		省略時は、対象のプロパティが有効(notNull、notEmpty)

- <resource/>

  このタグで指定されているファイルは、テンプレートエンジンで処理されず単純にそのまま出力する。

表８．<resource/>タグの属性

属性	内容	必須	備考
path	出力するファイルのパスを指定する	○	
outputPattern	java.text.MessageFormat の構文で出力ファイルのファイル名パターンを指定する	○	
outlet	ファイルの出力先を論理名で指定する	○	論理名はプロパティで指定する
overwrite	ファイル生成時にすでにファイルが存在する場合の上書きの有無を指定する	○	true/false で指定する
required	このテンプレートでファイルを生成しない場合に警告を出力するかの有無を指定する		true/false で指定する(デフォルトは true)

付録４ andromda-metafacades.xml の詳細

カートリッジで用意する Metafacade クラスがどのような条件で生成するのかを規定するため、およびプロパティの設定ファイル。ここでは、各タグの詳細を説明する。

```
<metafacades namespace="webservice">
    ...
    <property reference="arrayNamePrefix" default="ArrayOf"/>
    <property reference="schemaTypeMappingsUri"/>
    <metafacade
class="org.andromda.cartridges.webservice.metafacades.WSDLTypeLogicImpl"
contextRoot="true">
        <mapping class="org.omg.uml.foundation.core.UmlClass$Impl"/>
    </metafacade>
    <metafacade
class="org.andromda.cartridges.webservice.metafacades.WSDLTypeAttributeLogicImpl">
        <mapping class="org.omg.uml.foundation.core.Attribute$Impl">

<context>org.andromda.cartridges.webservice.metafacades.WSDLType</context>
        </mapping>
    </metafacade>
    <metafacade
class="org.andromda.cartridges.webservice.metafacades.WebServiceLogicImpl"
contextRoot="true">
        <mapping class="org.omg.uml.foundation.core.Classifier$Impl">
            <stereotype>WEBSERVICE</stereotype>
        </mapping>
        <property reference="ejbInterfacePattern" default="{0}.{1}"/>
    </metafacade>
    ...
</metafacades>
```

図３． andromda-metafacades.xml の記述例

- <metafacades/>

  このタグは andromda-metafacades.xml ファイルのルートとなるタグである。

表９．<metafacades/>タグの属性

属性	内容	必須	備考
namespace	名前空間を指定する。ほとんどの場合は、指定した値とカートリッジの名前は同一である。	○	
shared	この名前空間の Metafacade を共有するか指定する		true/false で指定する（デフォルトは false）

- <property/>

  このタグはカートリッジが参照するプロパティを設定する。このプロパティはこの設定ファイルで定義するすべての Metafacade クラスから参照可能である。

表１０．<property/>タグの属性

属性	内容	必須	備考
reference	プロパティの名前を指定する	○	
default	カートリッジを利用するプロジェクトで値が設定されない場合のデフォルト値を指定する		

- <default/>

  このタグは<metafacades/>タグ内でひとつのみ指定が可能なタグである。後述する<metafacade/>タグで指定する Metafacade クラスに適用されないような metaclass をマッピングする Metafacade クラスを指定する。

表１１．<default/>タグの属性

属性	内容	必須	備考
class	Metafacade の実装クラスの完全修飾名を指定する	○	

- <metafacade/>

  このタグは metaclass とマッピングする Metafacade クラスの設定をするためのタグである。

表１２．<metafacade/>タグの属性

属性	内容	必須	備考
class	Metafacade の実装クラスの完全修飾名を指定する	○	
contextRoot	この Metafacade が他の Metafacade のコンテキストルートになるかどうかを指定する。詳細は図４．を参照。		true/false で指定する(デフォルトは false)

```
...
<metafacade
 class="org.andromda.cartridges.webservice.metafacades.WebServiceLogicImpl"
 contextRoot="true"
>
    <mapping class="org.omg.uml.foundation.core.UmlClass$Impl">
        <stereotype>WEBSERVICE</stereotype>
    </mapping>
    ...
</metafacade>
...
<metafacade
 class=
"org.andromda.cartridges.webservice.metafacades.WebServiceOperationLogicImpl"
>
    <mapping class="org.omg.uml.foundation.core.Operation$Impl">
        <context>
          org.andromda.cartridges.webservice.metafacades.WebService
        </context>
    </mapping>
    ...
</metafacade>
...
```

図４．contextRoot 属性の記述例

属性「contextRoot」が「true」の場合は、図中の Metafacade「org.andromda.cartridges.webservice.metafacades.WebServiceLogicImpl」が生成されるときに、<context/>でこの Metafacade を指定している Metafacade(org.andromda.cartridges.webservice.metafacades.WebServiceOperationLogicImpl)も共に生成される。

- **<property/>**
  このタグは、設定内容については前述の<property/>と同じである。ただし、あちらはすべての Metafacade クラスで参照されるのに対し、こちらは対象の Metafacade でのみ参照可能になる。属性の詳細は、前述の<property/>を参照していただきたい。
- **<mapping/>**
  このタグは Metafacade がどの metaclass とマッピングするのかを定義する。このタグで内包する以下の要素を組み合わせてマッピング時の条件を定義する。

- stereotypes
  ひとつ以上の<stereotype/>で条件を構成する
- context
  コンテキストとなる Metafacade クラスで条件を構成する
- properties
  適用するひとつ以上の<property/>で条件を構成する

表１２．<mapping/>タグの属性

属性	内容	必須	備考
class	Metafacade にマッピングする metaclass を指定する	○	

➢ **<stereotype/>**

このタグは metaclass と Metafacade クラスのマッピングの際に metaclass が持つステレオタイプを指定する。このタグは複数指定が可能であり、指定方法によって条件の解釈が変化する。下記にその指定方法を記す。

- **<stereotype>タグをひとつだけ指定**

```
・・・
<mapping class="org.omg.uml.foundation.core.Classifier$Impl">
    <stereotype>SERVICE</stereotype>
</mapping>
・・・
```

上記のようにひとつだけステレオタイプを指定した場合は、指定したステレオタイプを持つ metaclass のみ Metafacade にマッピングされる。

- **<stereotype>タグを二つ以上指定**

```
・・・
<mapping class="org.omg.uml.foundation.core.Classifier$Impl">
    <stereotype>WEBSERVICE</stereotype>
    <stereotype>SERVICE</stereotype>
</mapping>
・・・
```

上記のように二つ以上ステレオタイプを指定した場合は、指定したステレオタイプをすべて持つ metaclass のみ Metafacade にマッピングされる。

- **指定した<stereotype/>のどちらを持つ場合の指定**

```
・・・
<metafacade ・・・>
  <mapping class="org.omg.uml.foundation.core.Classifier$Impl">
    <stereotype>WEBSERVICE</stereotype>
  </mapping>
</metafacade>
<metafacade ・・・>
  <mapping class="org.omg.uml.foundation.core.Classifier$Impl">
    <stereotype>SERVICE</stereotype>
  </mapping>
</metafacade>
・・・
```

いずれか一方のステレオタイプを持つ metaclass をマッピングしたい場合は、上記のように<metafacade/>の定義をステレオタイプごとに分けて定義する。こうすることによって上記の記述例では、ステレオタイプ<<WEBSERVICE>>または<<SERVICE>>を持つ metaclass を Metafacade へマッピングすることが可能になる。

- **<context/>**

このタグは指定した Metafacade クラスの生成時に共に生成するようするためのタグである。下記に記述例を挙げる。

```
...
<metafacade class=
"org.andromda.metafacades.uml14.EntityQueryOperationFacadeLogicImpl">
    <mapping class="org.omg.uml.foundation.core.Operation$Impl">
        <context>org.andromda.metafacades.uml.Entity</context>
        <property name="query"/>
    </mapping>
</metafacade>
...
```

上記、記述例の場合は、Metafacade「org.andromda.metafacades.uml.Entry」クラスが生成されるときに、この Metafacade クラスを共に生成するという記述になる。

- <property/>

  このタグは生成する Metafacade が指定のプロパティを持ち、かつ指定の値または、条件を満たしているかを指定する。値を省略時のマッピング有効条件とは下記の三つのことである。

  - このプロパティの値は Not Null である
  - もしプロパティの型が Collection であれば、Not Empty である
  - もしプロパティの型が boolean であれば、Not False である

表１２．<property/>タグの属性

属性	内容	必須	備考
name	評価するプロパティの名前を指定する。尚、このプロパティは Metafacade が持っている必要である。	○	
value	評価するプロパティの値を指定する。この属性は指定方法が二通りある。指定方法は図５．を参照		省略時は前述した有効条件で判定する

```
...
<metafacade
 class=
"org.andromda.cartridges.spring.metafacades.BusinessOperationLogicImpl"
>
    <mapping class="org.omg.uml.foundation.core.Operation$Impl">
        <property name="query" value="false"/>
    </mapping>
</metafacade>
...
上、または下の記述はどちらも同じ意味になる
...
<metafacade
 class=
"org.andromda.cartridges.spring.metafacades.BusinessOperationLogicImpl"
>
    <mapping class="org.omg.uml.foundation.core.Operation$Impl">
        <property name="query">false</property>
    </mapping>
</metafacade>
...
```

図５．<property/>の指定方法

付録５ andromda-profile.xml の詳細

カートリッジ内で使用されるステレオタイプやタグ付き値、その他の値を別名にマッピングするための設定ファイルである。下記に記述例を挙げる。

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<mappings name="uml">
<!-- stereotypes -->
    <mapping>
        <from>ENTITY</from>
        <to>Entity</to>
    </mapping>
・・・
<!-- tagged values -->
    <mapping>
        <from>PERSISTENCE_COLUMN_LENGTH</from>
        <to>@andromda.persistence.column.length</to>
    </mapping>
・・・
<!-- datatypes -->
    <mapping>
        <from>COLLECTION_TYPE</from>
        <to>datatype::Collection</to>
    </mapping>
・・・
</mappings>
```

図６． andromda-profile.xml の記述例

マッピング定義は<mapping/>で指定する。<from/>がカートリッジ内で参照する値になり、<to/>が実際の値になる。

尚、この Profile はカートリッジを利用するプロジェクトで値の上書きが可能である。上書き用の xml ファイルを用意し、カートリッジを利用しているプロジェクトの「mda/project.xml」ファイルにあるカートリッジ定義部分の<properties/>内に、下記の記述例のように記述する。

```
<dependency>
    <groupId>andromda</groupId>
    <artifactId>maven-andromda-plugin</artifactId>
    <version>${andromda.version}</version>
    <type>plugin</type>
    <properties>
        ...
        <profileMappingsUri>
          file:${basedir}/mda/conf/mappings/CustomizedProfileMappings.xml
        </profileMappingsUri>
        ...
    </properties>
</dependency>
```

図７．Profile の上書き設定の記述例

太字の部分が上書きの設定である。<profileMapping/>の値に上書き用の設定ファイルへのパスを指定する。これで、プロジェクトのビルド時に上書き指定した Profile が適用される。

付録６ 代表的な Metafacade

カートリッジでは Metafacade が重要な要素のひとつである。ここでは、AndroMDA で用意されている Metafacade クラスについて代表的なものを紹介する。尚、Metafacade クラスのモデルも図示するが、属性や操作、制約などは省略してある。

- **ClassifierFacade クラス**

  基本的な UML のクラス表現する Metafacade クラス。Pojo な Java クラスなどを作成する場合などは、このクラスを継承して利用する。基本的な、属性、操作の Metafacade の関連はすでに持っている。主な周りの関連のイメージを下図に記す。

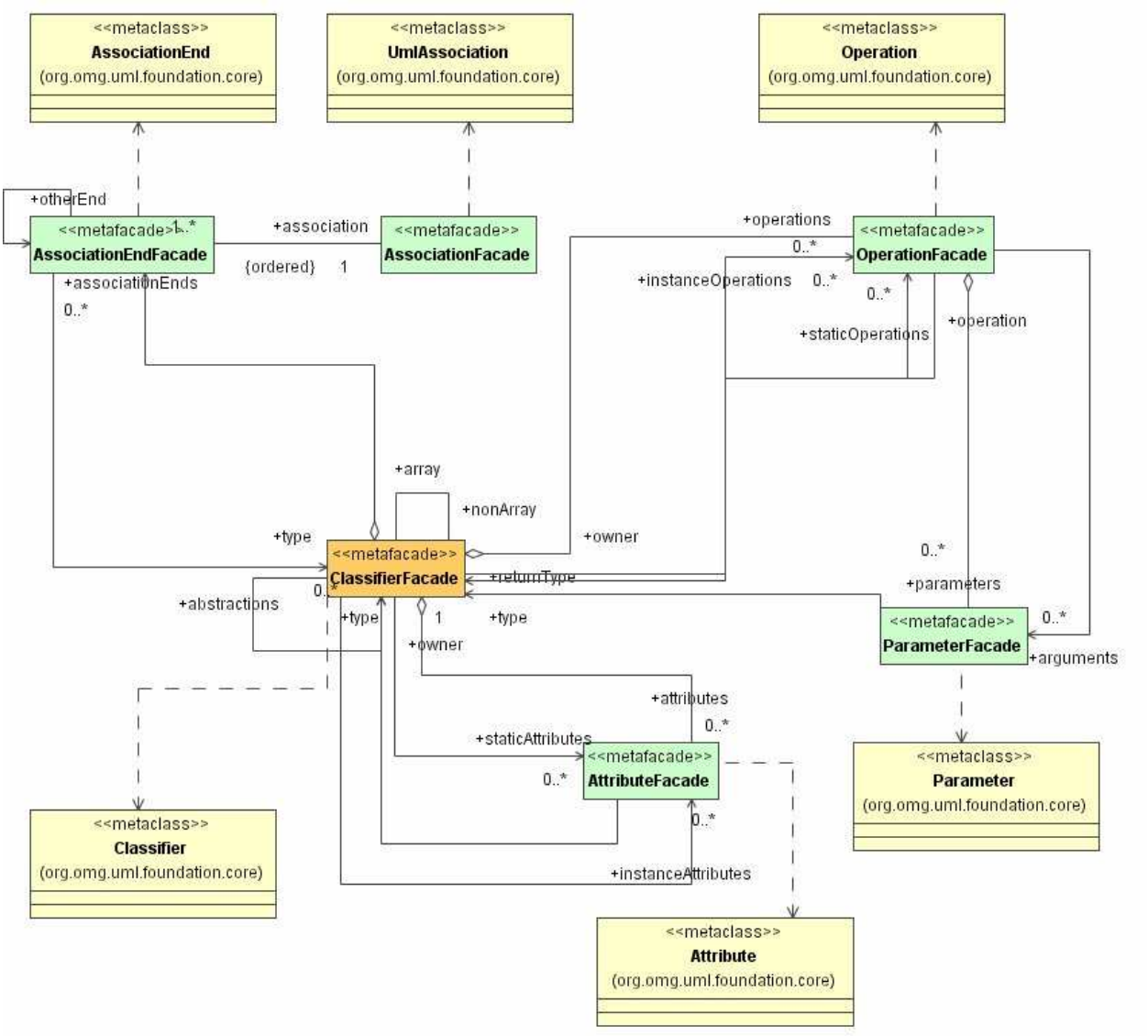


図８．ClassifierFacade クラスとその関連 Metafacade

- **Entity クラス**

  このクラスは EJB、Hibernate、Spring カートリッジなど永続化層で利用するカートリッジで利用している Metafacade クラスである。DB のテーブルをマッピングするクラスなどを作成する Metafacade が利用(継承)する。主な周りの関連のイメージを下図に記す。

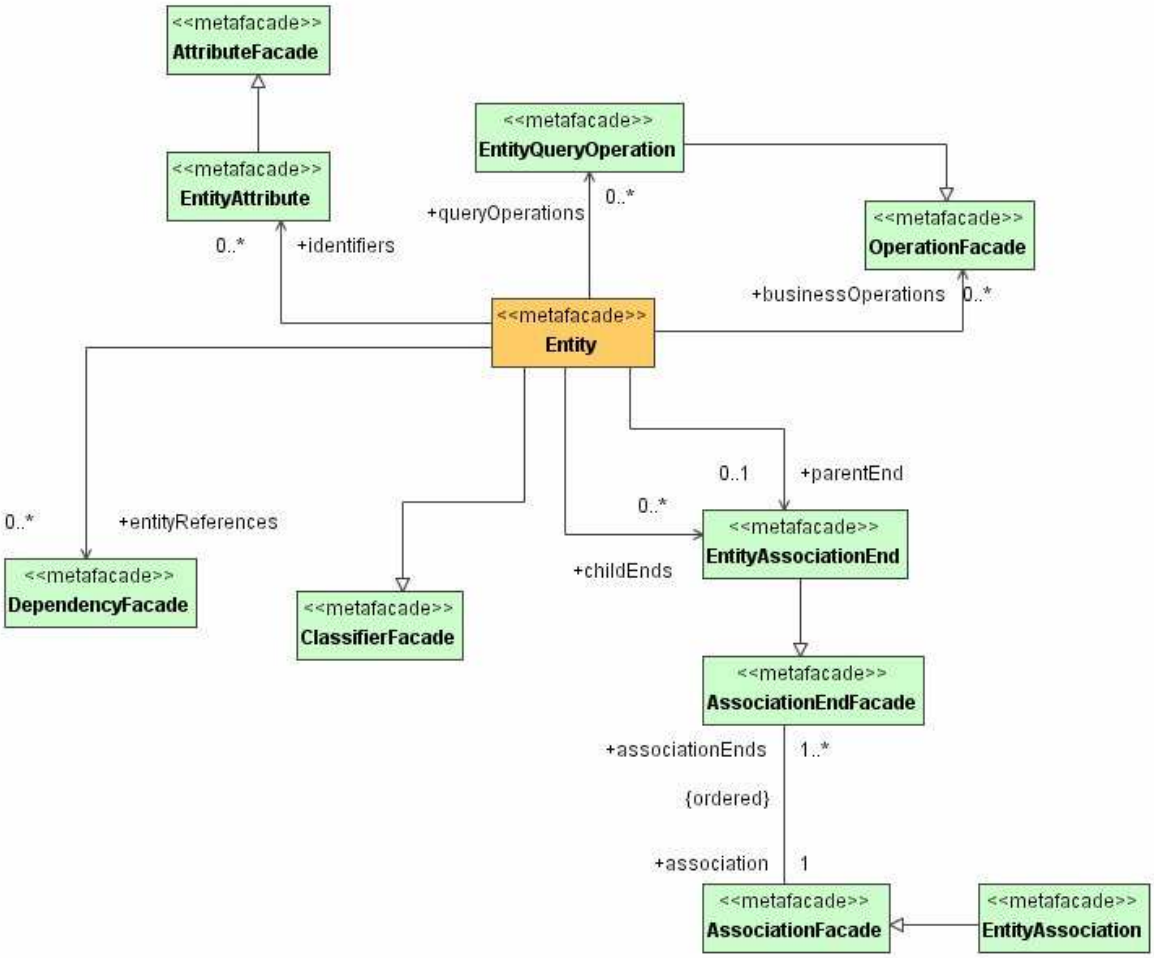


図９．Entity クラスとその関連 Metafacade

- **Service クラス**

  このクラスは EJB、Hibernate、Spring カートリッジでビジネスロジック記述部分を作成するために利用する Metafacade クラスである。ビジネスロジック部分を生成する Metafacade クラスを作成する際には、この Metafacade を継承すると良い。主な周りの関連のイメージを下図に記す。

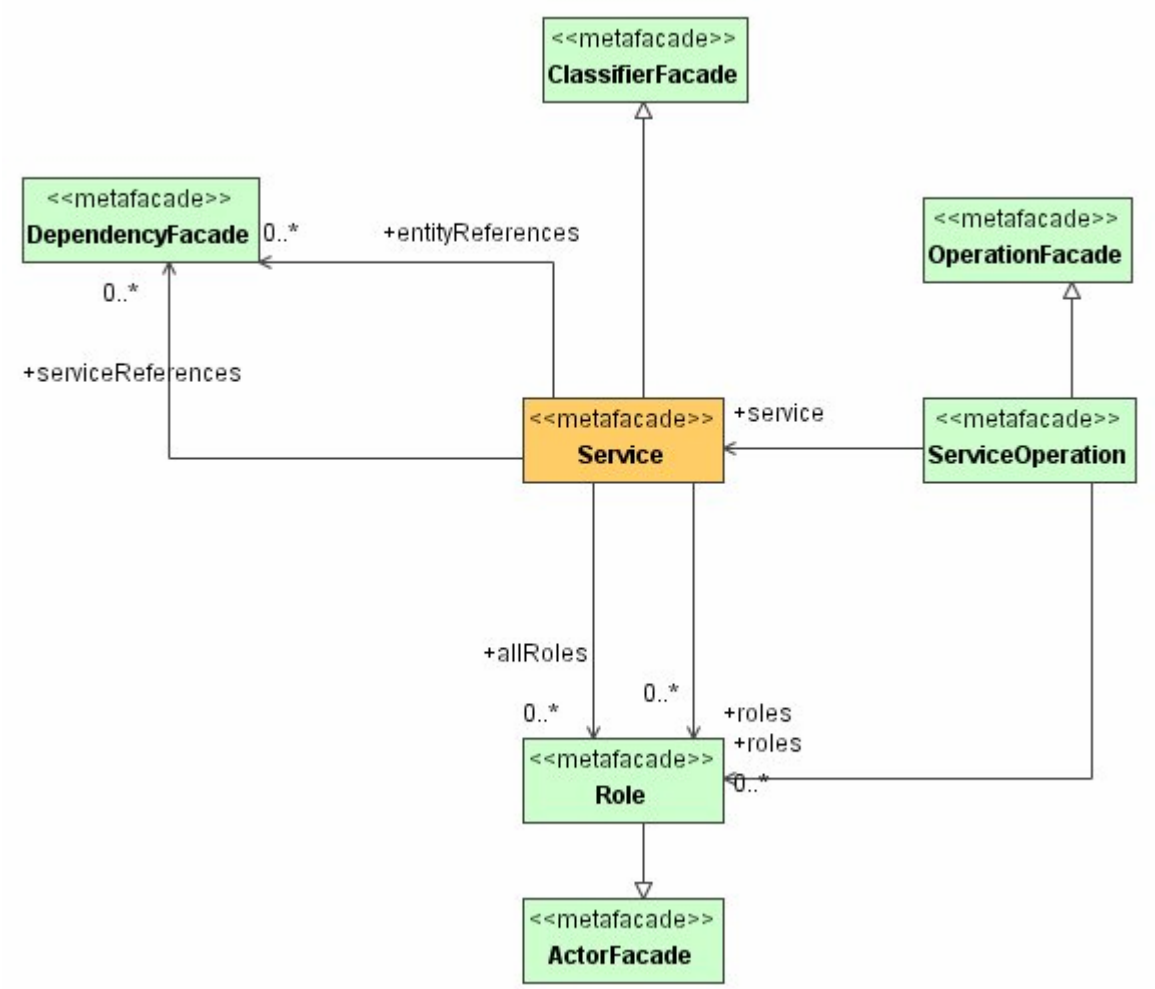


図１０．Service クラスとその関連 Metafacade