AndroMDA 解体新書

第１．１版

２００６／３／０１

0 はじめに

近年、オープンソースの MDA ツールとして有名になってきた AndroMDA は UML からプログラミング言語や Hibernate、Spring Framework 等の設定ファイルを生成するコード生成のフレームワークである。 AndroMDA はカートリッジ（コード生成の変換ルール）を追加開発することにより、機能を拡張(カスタマイズ)することができるように設計されている。本書は AndroMDA をカスタマイズしたい技術者向けに書かれている。AndroMDA　をカスタマイズするためには、OMG が制定する MetaObject Facility(MOF)や UML メタモデルなどの標準と AndroMDA が利用しているオープンソースの理解が必要である。そこで、第１章では、これら標準と NetBeans MDR と Velocity を中心に説明する。第２章では NetBeans MDR のみを用いて擬似的なコード生成が可能であることを示し、次に、AndroMDA を使用することにより、容易に本格的なコード生成が可能になることを示す。第３章では簡単な AndroMDA のカスタマイズ例を示す。

1 原理編

AndroMDA はユーザが UML ツールを使って作成したモデルを、XML ファイルを経由して読み込み、ソースコードとフレームワークやミドルウェアなどの環境設定ファイルを生成する。AndroMDA は様々なオープンソースを内部で利用している。図 1－1 に AndroMDA の内部構造と利用しているオープンソースを図示する。

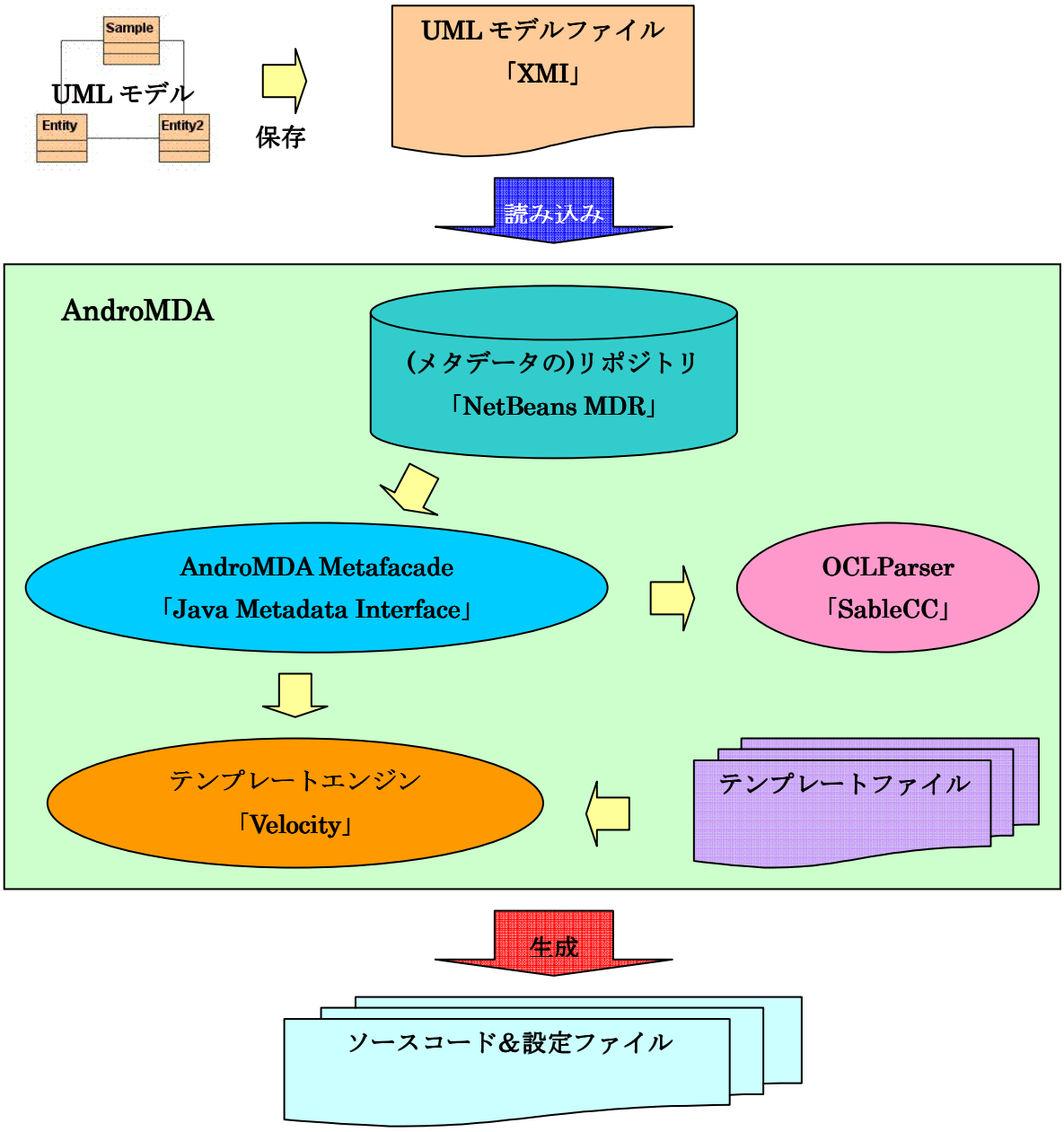

図 1－1 AndroMDA の構成図

図 1-1に図示している各要素について下記に記す。

- **UML モデル**

  人事、会計、生産管理などのアプリケーションの UML モデル。UML モデリングツールによって作成できる。なお、AndroMDA に適合する UML モデリングツールはいくつかあるが、推奨されているのは No Magic 社の MagicDraw である。その他の UML モデリングツールと AndroMDA との相性は下記のページを参照していただきたい。

  http://galaxy.andromda.org/docs/case-tools.html

- **UML モデルファイル**

  上記 UML モデルを保存した(シリアライズ)XML 形式のファイル。この XML 形式は OMG がモデルを交換するために制定した XML Metadata Interchange(XMI)という標準で規定されている。XMI はモデルをシリアライズする方法を規定しているともいえる。なお、AndroMDA で扱う XMI のバージョンは 1.2 である。

- **リポジトリ**

  リポジトリは UML モデルファイルを読み込み、メモリ上に復元する。また、逆にメモリ上にあるモデルを UML モデルファイルとして出力することもできる(ただし、AndroMDA には不要な機能である)。また、プログラムからメモリ上に読み込んであるモデルへのアクセスを提供する。モデルの検索や追加、更新、削除などができることから、簡単に言えばリポジトリはモデルのデータベースである。AndroMDA はリポジトリに Sun Microsystems が開発した NetBeans MDR(Metadata Repository)を用いている。

- **AndroMDA Metafacade**

  AndroMDA Metafacade は AndroMDA がコード生成する際にリポジトリの複雑な構造を隠蔽し、容易にアクセスする手段を提供するクラス群である。簡単に言えば、リポジトリからコードを生成するためのアプリケーションである。また、AndroMDA がリポジトリを交換できるようにリポジトリを抽象化している。

- **OCL Parser**

  AndroMDA はモデルに含まれる Object Constraint Language(OCL)を Java のコードに翻訳することができる(現状、OCL の select のみがサポートされている)。OCL を翻訳するためには OCL を構文解析するパーサが必要である。AndroMDA は Java で書かれているため、パーサは Java でなければならない。AndroMDA はコンパイラー・コンパイラーである SableCC が生成したパーサを使用している。パースツリーから Java コードを生成するジェネレータは AndroMDA 独自である。一般的に SableCC は文法を定義したファイルから Java によるパーサを自動生成する。従って OCL の文法を定義したファイルからは、Java による OCL パーサが生成される。また、SableCC はパースツリーを生成するクラスとパースツリーを走査するクラスも自動生成する特徴がある。古くからある OCL に関するツールキット「Dresden OCL Toolkit」でも SableCC を使用している。OCL から Java に翻訳するオープンソース「Octopus」はコンパイラー・コンパイラーである JavaCC を使用している。なお、AndroMDA はパーサを抽象化してあるため、コンパイラー・コンパイラーが交換可能になっている。

- **テンプレートエンジン&テンプレートファイル**

  AndroMDA はモデルの情報をテンプレートファイルと呼ばれるコードの定型ファイルを元に Java クラスや各種設定ファイルを生成する。一般に、テンプレートファイルは固定文字列と変数が含まれている。テンプレートエンジンは変数に値を代入してファイルを生成する。AndroMDA はテンプレートエンジンとして Velocity を利用している。Velocity に変数名と値の組み情報を格納する VelocityContext というオブジェクトを与えることにより、テンプレートに値がマージされる。

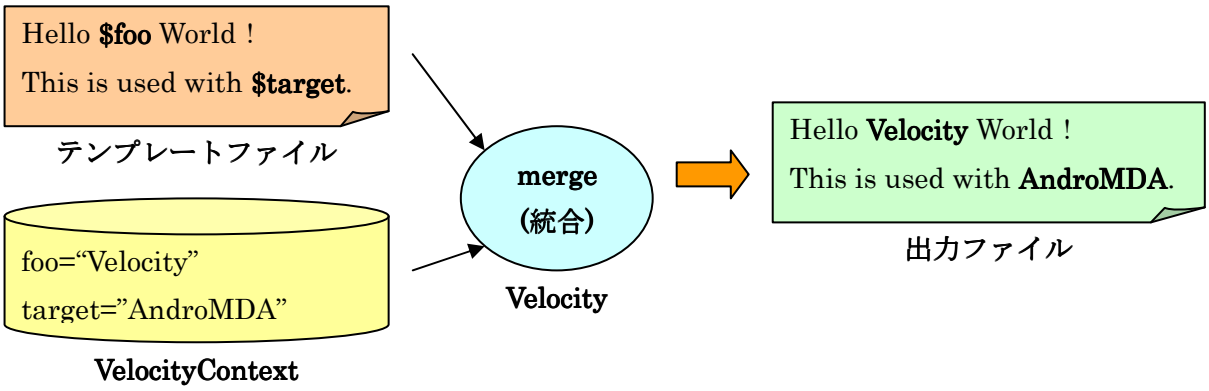

図 1-2 Velocity のファイル生成イメージ

- **ソースコード&設定ファイル**

  これは AndroMDA が生成する出力ファイルである。モデル要素と 1 対 1 のものもあれば複数のモデル要素から 1 つのファイルを生成することもある。

次に、MOF と個別のオープンソースについて詳しく述べる。

１．１　MetaObject Facility (MOF)

１．１．１　　MOF のメタデータ構造

MDA を実現するため、Object Management Group(OMG)は、MetaObject Facility(MOF)を定義している。 MOF は一般的なモデルの交換を可能とするため、図 １-３に示す階層構造になっている。階層構造になる理由を販売管理のビジネスシステムのリレーショナルデータベースを例にして説明する。販売管理のビジネスシステムのデータモデル(スキーマ)には、顧客、注文、商品等のエンティティがあるだろう。次にリレーショナルデータベース管理システム自体のデータモデルを考える。このデータモデルには表、属性関係、整合性制約などのエンティティがあるだろう。表のインスタンスとしては顧客、注文、商品がある。関係のインスタンスとして、1 対多の顧客-注文関係、多対多の注文-商品関係などがある。これらを整理すると、リレーショナルデータベース管理システム自体のモデルは販売管理のビジネスシステムのモデルを管理して蓄えることができる。データベース管理システム自体のモデルは販売管理のビジネスシステムのモデルのモデル(これをメタモデルという)である。更にリレーショナルデータベース管理システム自体のモデルに対してのメタモデルは唯一のモデルである MOF モデル(メタ-メタモデルともいう)に行き着く。このようにメタな関係を考えるとモデルは階層構造になる。メタな関係を逆に言えば、インスタンス化の関係になる。MOF モデルからインスタンス化の関係を追えば「MOF モデル→リレーショナルデータベース管理システムのモデル→販売管理のビジネスシステムのモデル」となる。オブジェクト指向においても同様な関係がある。前の例では「MOF モデル→UML メタモデル→UML で表現された販売管理のビジネスシステムのモデル(UML モデル)」となる。MOF モデルと UML メタモデルは OMG によって標準化されている。MDA ツールを開発する技術者は UML メタモデルを熟知する必要がある。なお、MOF モデルと UML メタモデルは静的な構造は類似している。

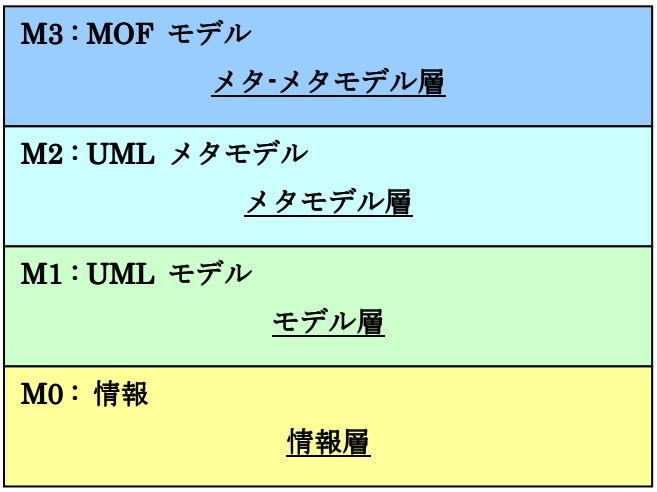

図 １-３　４階層構造

図 1－3の各層の詳細を下記に記す。

・情報層

M0 またはデータ層ともいう。情報層は情報の実際のインスタンスである。例えば、販売管理のビジネスシステムでは、「顧客：山田太郎」、「商品：DVD プレイヤー」である。

・モデル層

M1 またはメタデータ層ともいう。ビジネスや技術分野におけるモデルである。例えば販売管理システムのモデルや生産管理システムのモデルがある。なお、モデルは情報層を定義するデータでもあるため、通常のデータと異なることを強調してメタデータともいう。また、オブジェクト指向の立場ではデータでなくオブジェクトなので、メタデータをメタオブジェクトと呼ぶ。

・ メタモデル層

M2 またはメタ-メタデータ層ともいう。OMG は UML メタモデル、Common Warehouse Metamodel(CWM)、CORBA Component Model(CCM)の３つのモデルを標準化している。
UML メタモデルはクラス、属性、関連、継承など UML の概念をモデル化したものである。

・ メタ-メタモデル層

M3 または MOF モデルともいう。M3 のメタモデルは M3 自身である。即ち M3 は自身で完結している(閉じている)といえる。MOF モデルはメタモデリング環境をサポートするシステムによって内部的に定義される。即ち可変なデータとして読み込まれるのではなく、ハードコーディングされている。M3 モデルの重要性は M2 以下を定義する汎用性にある。即ち M3 のインスタンスを蓄えるメモリ領域を準備してから M2 のモデルを XMI 経由でメモリ領域に読み込む。次に M2 のインスタンスを蓄えるメモリ領域を準備してから M1 を XMI 経由でメモリ領域に読み込む。つまり、ブートストラップを適用することが可能になる。更に、ブートストラップと Java クラスを実行時に生成する技法(on the fly)を組み合わせると、きわめて汎用性が高いモデル蓄積機構を構築することができる。これが１．２で述べる NetBeans MDR である。

各層間の関係を図示する。モデルは XMI によって交換されるデータでもある。この観点でモデルはメタデータとも呼ばれる。

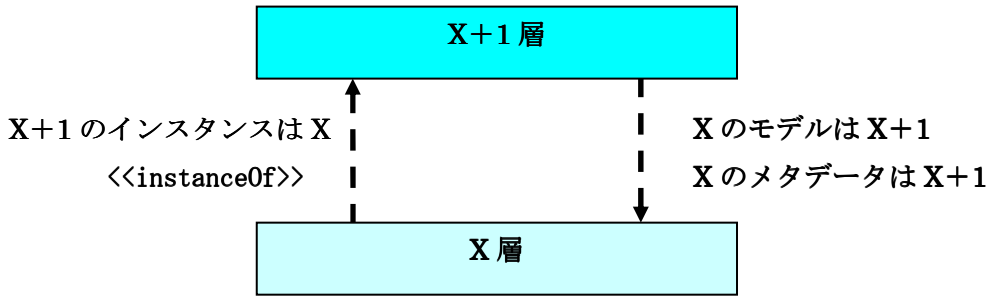

図 1－4 各層間の関係

次に、実際の UML モデル(Person クラス)を例にとり、４階層構造を示す。

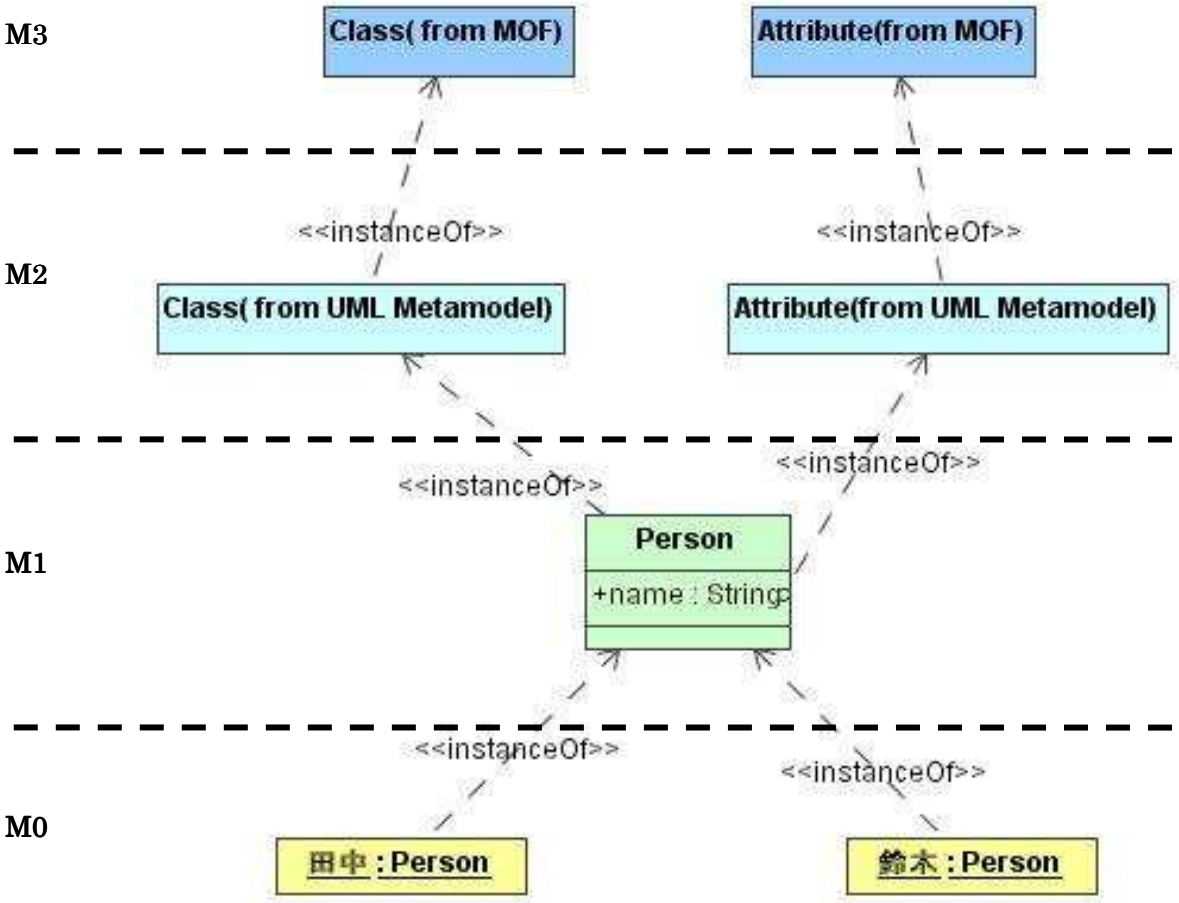

図 1－5 UML を例にした構造図

UML ツールなどを用いて作成する UML モデルは M1 の「Person」である。M1 の「Person」クラスをインスタンス化すれば M0 のオブジェクトになる。M1 の「Person」クラス は M2 の「Class」のインスタンスである。また、「Person」クラスの「name」属性は M2 の「Attribute」のインスタンスである。M2 のモデルは M3 の MOF のモデルによって定義される。このように M0 から M3 にかけて上位の層が下位の層の定義をしている。

１．１．２　　MOF モデル

MOF モデルは MOF の階層の最上位のモデルであり、下位モデルを規定する。ここでは MOF モデルの概要を説明する。下記に MOF モデルの概要図を図示する。

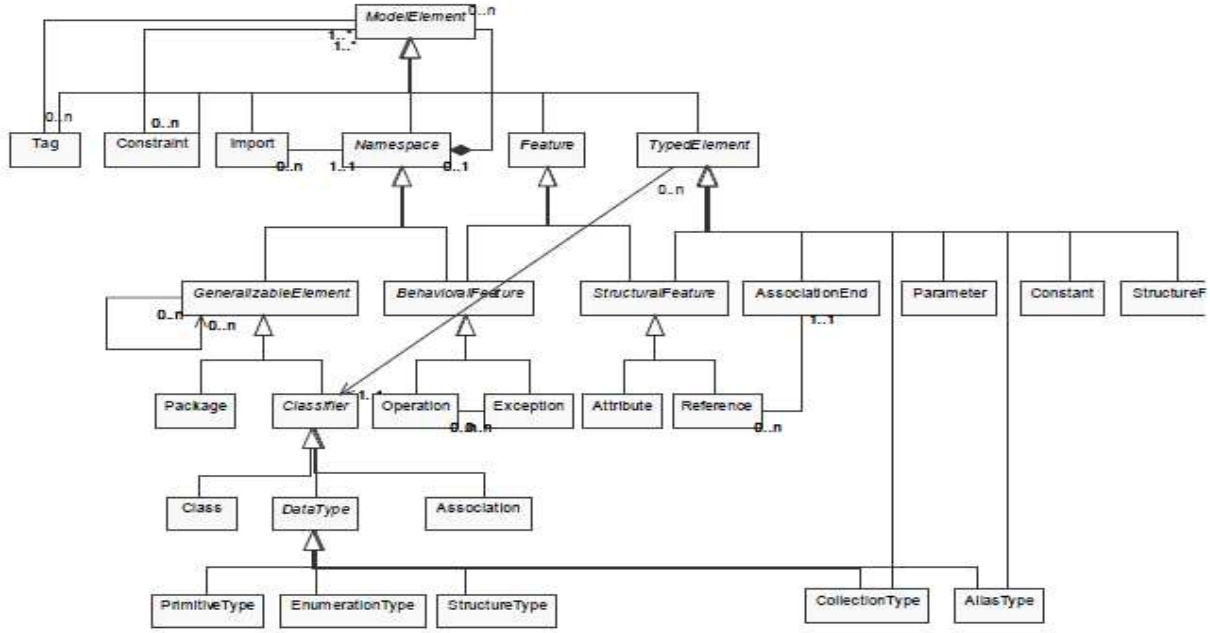

図 １－６ MOF モデルの概要図(Java Metadata Interface(JMI) Specification より引用)

この図は UML のクラス図で表現した MOF モデルの概略である。モデル要素の責務を １．１．２．６章 MOF　Model Elements に纏めたので、その時にこの図を参照されたし。なお、MOF モデルは UML メタモデルに似ている。UML2.0 の制定において統一が議論されたことがあるほどである（図 １－６と Page30 の図 １－１５を比較されたし)。

１．１．２．１　Common Superclass

図　１－７は図　１－６の継承関係の一部を詳細化したものである。図中にある「ModelElement」はすべてから継承されるスーパークラスである。「ModelElement」はモデル要素の依存関係を表現する。「Namespace」はコンポジットパターンによりモデル要素の抱合関係を表現する。「GeneralizedElement」はモデル要素の汎化関係を表現する。

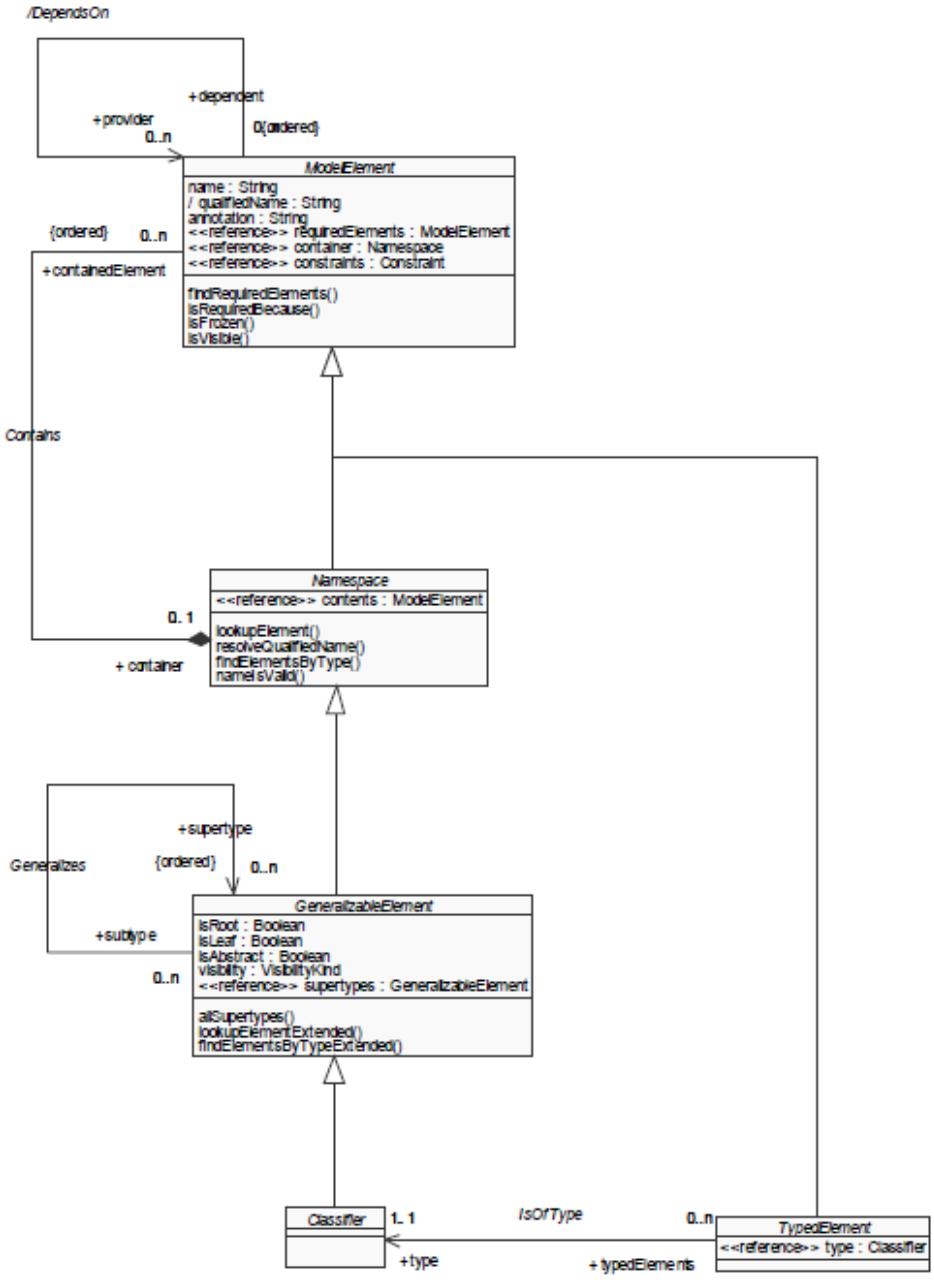

図　１－７　MOF Common SupperClass(Java Metadata Interface(JMI) Specification より引用)

「ModelElement」からすべての要素は「name」と「annotation」を受け継ぐ。そしてすべての「ModelElement」は唯一の「Namespace」に含まれる。

１．１．２．２　Containment Hierarchy

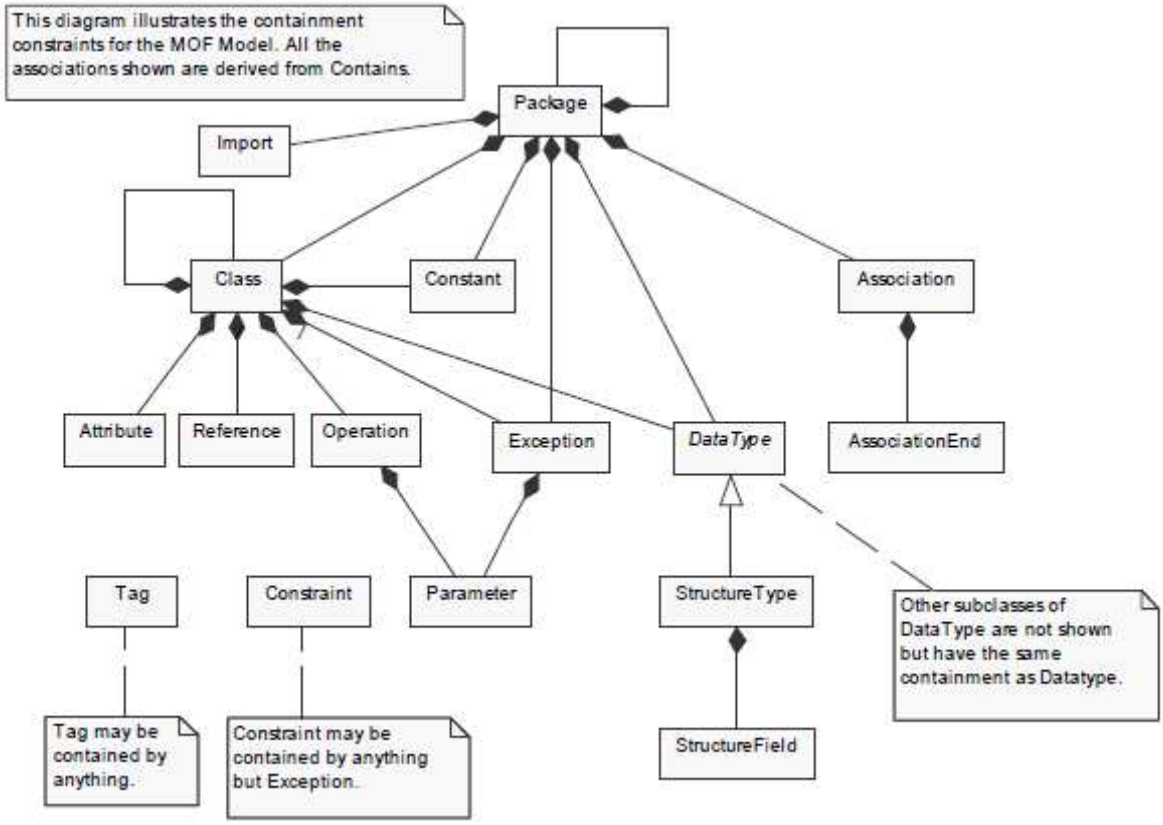

図　1－8　MOF の集約関係の階層(Java Metadata Interface(JMI) Specification より引用)

図　1－8は　MOF モデルの具象クラスの集約関係を抜き出したものである。「Package」が最上位のコンテナであり、他の要素に含むことができない唯一の要素である。各モデル要素は何らかの「Package」に含まれることになる。

１．１．２．３　Types

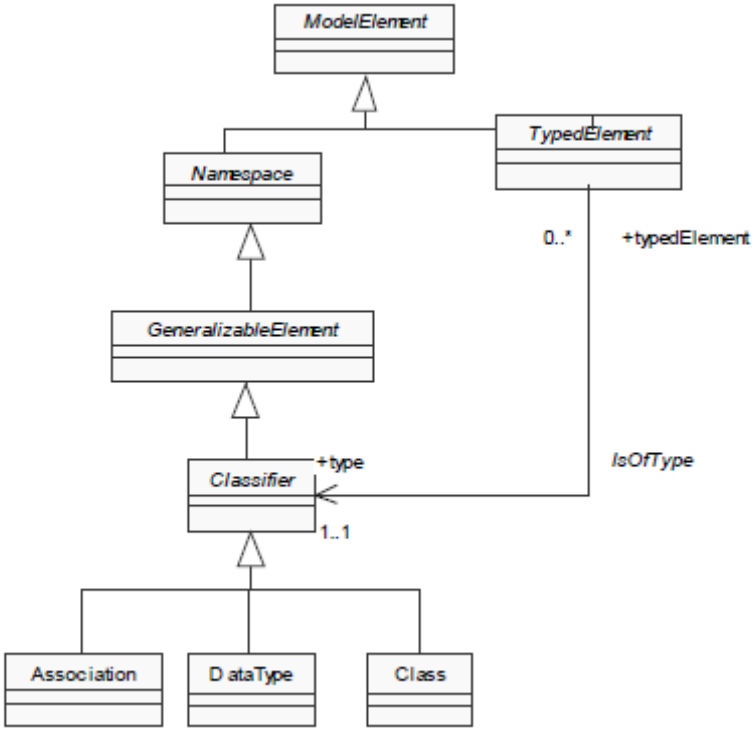

図　１－９　MOF の Type 要素(Java Metadata Interface(JMI) Specification より引用)

「Classifier」は、「Class」、「Association」、「DataType」を抽象化したクラスであり、型付けの責務がある。

１．１．２．４　Features

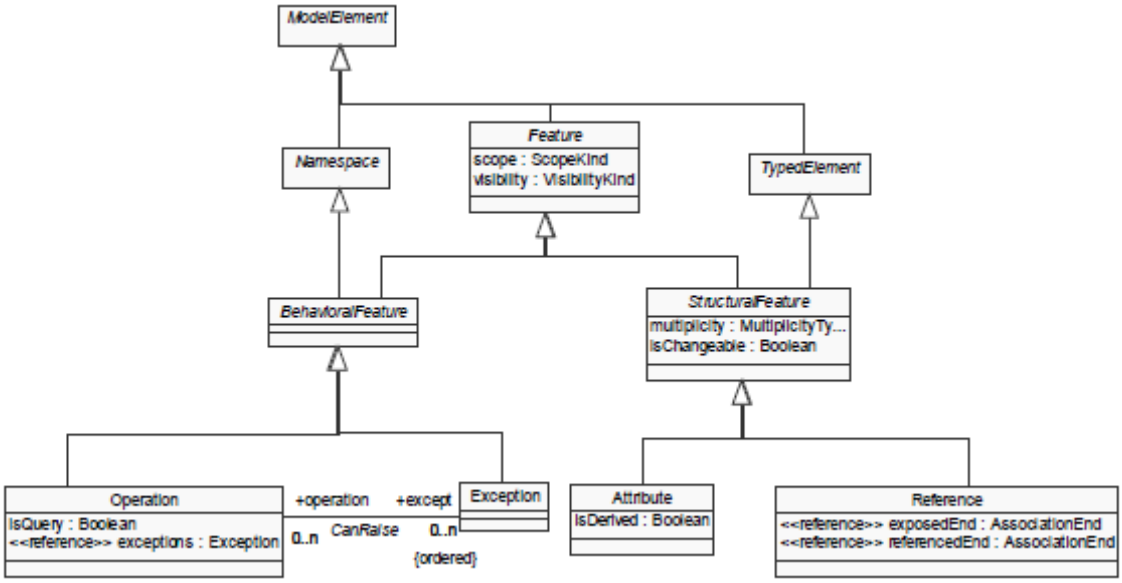

図　１－１０　MOF の Feature 要素(Java Metadata Interface(JMI) Specification より引用)

UML におけるクラスは振る舞いと構造を有する。振る舞いには操作があり、構造には属性と参照がある。図　１－１０に示すように MOF においても同様な構造がある。すなわち、「Feature」クラスには「StructuralFeature」と「BehavioralFeature」がある。「StructuralFeature」は、それを含む「ModelElement」の静的特徴である属性や参照を抽象化したクラスである。「BehavioralFeature」は、それを含む「ModelElement」の振る舞いである操作と例外を抽象化したクラスである。

１．１．２．５　Tags

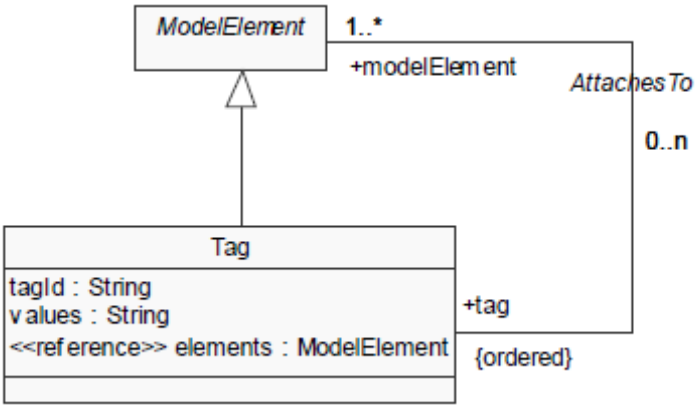

図　１－１１　MOF の Tags 要素(Java Metadata Interface(JMI) Specification より引用)

Tag モデル要素は、純粋な MOF モデルを拡張または変更するのを可能にする仕組みの基礎要素である。全てのモデル要素に複数のタグを付けることができる。

１．１．２．６　MOF Model Elements

１．１．１で紹介した MOF のモデルには様々な要素が存在する。ここではいくつかの要素について紹介する。.

- ModelElement

内容	すべての MOF 要素のルートである。モデルの基本構成概念を象徴する。ModelElement は自己参照により依存関係を表現する。
属性	name、annotation、qualifiedName
参照	container、requiredElements、constraints
操作	isFrozen、isVisible、isRequiredBecause
スーパークラス	なし
関連	Namespace、Constraint、Tag

- Namespace

内容	他の要素を包括することができる要素である。
属性	なし
参照	contents
操作	lookupElement 、 resolveQualifiedName 、 nameIsValid 、 findElementByType
スーパークラス	ModelElement
関連	ModelElement、Import

- Constraint

内容	1 つ以上の要素の状態または動きを制限する規則を定義する。
属性	expression、language、evaluationPolicy
参照	constrainedElements
操作	なし
スーパークラス	ModelElement
関連	なし

- Tag

内容	任意の名前と値の組である。Tag は最上位のスーパークラスである ModelElement に付加できるため、それを継承する全クラスは Tag を付加できることになる。また、Tag に Tag を付加することも可能である。
属性	tagId、values
参照	elements
操作	なし
スーパークラス	ModelElement
関連	ModelElement

- Feature

内容	モデル要素の特徴(振る舞いと構造)を抽象化したクラス
属性	visiblity、scope
参照	なし
操作	なし
スーパークラス	ModelElement
関連	なし

- TypedElement

内容	定義の一部としてタイプを必要とするモデル要素の抽象概念である。TypedElement 自体はタイプを定義しない。しかし、Classifier と関連する。
属性	なし
参照	type
操作	なし
スーパークラス	ModelElement
関連	Classifier

- GeneralizableElement

内容	継承関係を表すクラスである。
属性	visibility、isAbstract、isRoot、isLeaf
参照	supertypes
操作	allSupertypes、lookupElementExtended、 findElementByTypeExtended
スーパークラス	Namespace
関連	GeneralizableElement

- Classifier

内容	含まれる特徴（属性や参照、操作)によってインスタンスオブジェクトを分類する。
属性	なし
参照	なし
操作	なし
スーパークラス	GeneralizableElement
関連	TypedElement

- Association

内容	1 組の Classifier モデル要素の関連を表す。
属性	isDerived
参照	なし
操作	なし
スーパークラス	Classifier
関連	AssociationEnd

- DateType

責務	データの値のタイプを表す。オブジェクトと異なり生存期間や固有のアイデンティティを持たない。このクラスには用途に応じて様々なタイプのサブクラスがある。
属性	なし
参照	なし
操作	なし
スーパークラス	Classifier
関連	Class

- Class

内容	含まれる特徴によって具象化される Classifier モデル要素
属性	isSingleton
参照	なし
操作	なし
スーパークラス	Classifier
関連	Class、Package、Constant、Attribute、Reference、Operation、Exception、DataType

- BehavioralFeature

内容	モデル要素の振る舞い(操作と例外)を抽象化したクラス。
属性	なし
参照	なし
操作	なし
スーパークラス	Feature、Namespace
関連	なし

- StructuralFeature

内容	モデル要素の構造的な特徴(属性と関連)を抽象化クラス。
属性	なし
参照	なし
操作	なし
スーパークラス	Feature、TypedElement
関連	なし

- Operation

内容	モデル要素の操作を表現するクラス
属性	isQuery
参照	exceptions
操作	なし
スーパークラス	BehavioralFeature
関連	Exception

- Attribute

内容	値または値の集合などを含む静的な特徴を定義する。即ち属性。
属性	isDervied
参照	なし
操作	なし
スーパークラス	StructuralFeature
関連	なし

- Reference

内容	Classifier に参照付く関連オブジェクトの情報と関連のリンクセットへのアクセスを定義する。
属性	なし
参照	referencedEnd、exposedEnd
操作	なし
スーパークラス	StructuralFeature
関連	AssociationEnd

- Constant

内容	単純なデータ型の一定値(定数)を定める。
属性	value
参照	なし
操作	なし
スーパークラス	TypedElement
関連	Class、Package

- Parameter

内容	BehavioralFeature とやり取りするためのパラメータを定義する。
属性	direction、multiplicity
参照	なし
操作	なし
スーパークラス	TypedElement
関連	Operation、Exception

- AssociationEnd

内容	Association の一端を表す。つまり Association は 2 つの AssociationEnd からなる。
属性	aggregation、multiplicity、isNavigable、isChangeable
参照	なし
操作	otherEnd
スーパークラス	TypedElement
関連	Reference

※ 本書では MOF のバージョン 1.4 の内容を記述しているが、現在は 2.0 を提案中である。

１．２　NetBeans MDR

NetBeans MDR は Java による MOF の実装である。また、XMI によるメタモデルの交換を実現している。MOF と Java とのマッピングについては JCP が規定する JSR-040：Java Metadata Interface(JMI)という仕様(インタフェース)に従っている。XMI から読み込んだ任意の MOF 対応のメタモデル(M3 のインスタンスである任意の M2、またはその M2 の任意のインスタンス)に対して、NetBeans MDR はモデルに対応した Java オブジェクトを動的に生成することができる。すなわち、任意の MOF 対応のメタモデルをノンプログラムでメモリにロードすることが可能になる。下記に NetBeans MDR の主要な機能を記す。

- MOF1.4 と MOF1.3 による XMI1.1 または 1.2 のドキュメントのインポートをサポートする
- リポジトリに保存されるデータの XMI1.2 形式のドキュメントのエクスポートをサポートする。
- MOF 対応の任意のメタモデルに対して、JMI 仕様に従った Java インタフェース(後述)を生成することが可能である。
- MOF 対応の任意のメタモデルに対して、Java のインスタンスを生成できる。
- 生成する API はランタイムに自動生成される。

下記に NetBeans MDR の構成を図示する。

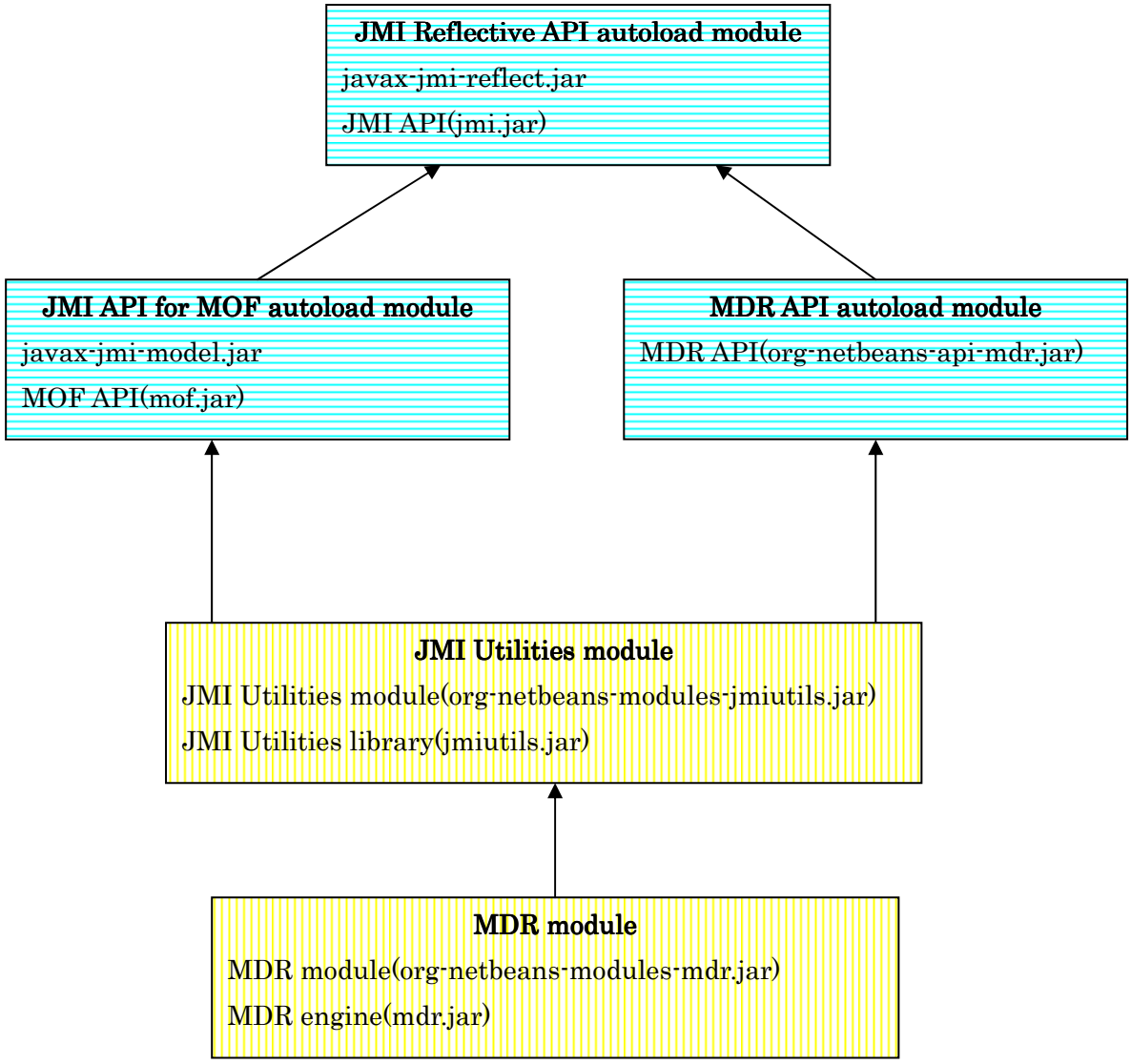

図 1－1 2 NetBeans MDR のライブラリ構成

図 1-１２の横線模様の箱のモジュールはランタイムにインタフェースを実装したクラスを自動生成するモジュールである。縦線模様の箱は、JMI や NetBeans MDR を使用する際に利用するモジュールである。下記に図中のモジュールの説明を記す。

- **JMI Reflective API**
  JMI のリフレクション機能はメタデータを規定するメタモデルに関する詳細な知識がなくとも NetBeans MDR によって管理されているメタデータにアクセスするためのインタフェースを提供する。例えば、顧客クラスから自身を規定する UML メタモデルの「Class」クラスの情報を取得することができる。NetBeans MDR は JMI Reflective API を自動生成する。
- **JMI API for MOF**
  MOF1.4 のメタモデルをマッピングしたクラスを生成する。(MOF のメタモデルに対応している)メタデータにアクセスするために使用する。例えば、MOF の Classifier や Operation クラスの情報にアクセスするためのメソッドを提供する。NetBeans MDR は API for MOF を自動生成する。
- **MDR API**
  NetBeans MDR の API はメタデータリポジトリに対する API の生成を定義する。この API は JMI API の拡張である。
- **JMI Utilities**
  JMI のリフレクション API を操作するための JMI ユーティリティライブラリである。このライブラリは XMI の読書きの機能や JMI のマッピングに対してのユーティリティ、その他の機能を提供する。すべてのユーティリティはどのような JMI 対応のリポジトリであっても実行が可能である。
- **MDR**
  NetBeans MDR の実装のコアライブラリである。

１．３　Java Metadata Interface (JMI)

JMI は MOF モデルを Java にマッピングする仕様である。また、メタデータの作成、保存、アクセス、検索、変更の仕様も定義している。NetBeans MDR は JMI の仕様(インタフェース)を実装している。下記に JMI が定義する機能を記す。

- JMI の処理はモデル駆動である。即ち、メタモデルの情報からメタデータにアクセスするためのすべての Java インタフェースが自動的に生成される。NetBeans MDR は実行時に動的にクラスを生成する技法(on the fly)でこの機能を実装している。しかも、動的に生成されたクラスのインスタンス化も自動的に実行される。この際、メタモデルに従ったインスタンスの関連付けも行われる。AndroMDA ではこの NetBeans MDR が生成したインスタンスを使用してコードを生成するのだが、このインスタンスは実行時に生成されるため、ソースもなければクラスファイルも存在しない。したがって、インスタンスを利用するプログラムの開発に支障が生じる(NetBeans MDR によりクラスファイルを生成する)。また、コード生成の観点では UML メタモデルは複雑すぎる。そこで、AndroMDA は UML モデルを内部に隠蔽し、扱いやすいインタフェースを提供するクラスとして AndroMDA Metafacade クラスを定義している。下記に AndroMDA Metafacade、JMI、NetBeans MDR のクラスの関係を図示する。

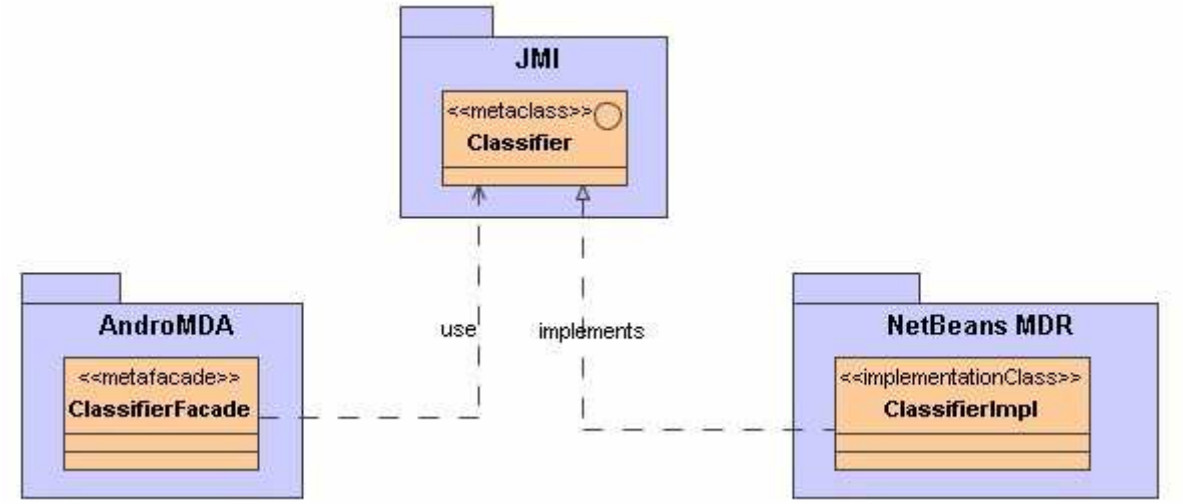

図 １－１３ JMI、AndroMDA Metafacade、NetBeans MDR のクラスの関係

- JMI はリフレクション機能を提供する。この機能はアプリケーションがモデル化したシステムの構造や意味をランタイムで問い合わせすることを可能にするためのものである。Java 言語におけるリフレクション機能により、M0 から M1 の情報を部分的ではあるが得ることができる。しかし、M1 から M2、また、M2 から M3 の情報を与えることはしない。そこで、JMI はその機能(Reflective Package)を提供した。

- JMI はライフサイクル管理を提供する。
例えば、「Foo」という名のモデルからは２つのインタフェース「Foo.java」と「FooClass.java」

が生成される。「FooClass」は「Foo」クラスのインスタンスの生成と消滅を管理する。なぜなら、例えば「Foo」クラスのインスタンスはメタモデルに従った関連付けが必要である。従って、通常のインスタンスを生成するようにnewで生成するだけでは処理が不足している。この生成に係わる処理の複雑さを「FooClass」のファクトリーメソッドが隠蔽してくれる。「FooClass」はインスタンスのライフサイクルを担当する「Foo」クラスのスタティック部分と考えられるため、「FooClass」と命名されているのである。ライフサイクル管理はデータベースの代用であると考えてもよい。一般に、OR-Mappingを実装する際、永続化機能を担当するクラスが必要になる。「FooClass」はOR-Mappingにおける永続化機能を担当するクラスに対応する。クラスが抽象クラスとしてモデル化されている場合は、ファクトリーメソッドは生成されない。「Foo」インタフェースは「Foo」クラスのすべてのインスタンスによって実装される。これらのインタフェースは要求されるモデル間での関連のナビゲーション機能も含まれる。この機能をJMIリファレンスと呼ぶ。JMIリファレンスは、インスタンスの関連を操作し、ナビゲーションする機能を提供する。

- JMIは、XMLのインポートとエクスポートの機能を提供する。JMIのXMLマッピングは、XML DTDとしてモデルをどのように表現するかを定義する。JMIのXML Utility APIを使用することによって、特定のJMIモデルパッケージ範囲内のデータのすべてまたは一部のインポートとエクスポートを可能にする。JMIで扱うXMLはXMI形式である。

JMI には４種のメタオブジェクトがある。

- Package Object

  「package object」はその名のとおりパッケージである。このメタオブジェクトはメタモデルに記述されるメタオブジェクトのコレクションに対してアクセスする操作を提供している。最も外側のパッケージは、メタデータが表している層の”ルート”を意味する。他のすべてのオブジェクト(「instance object」、「class proxy object」、「association object」、入れ子になった「package object」)は、最も外側のパッケージ内に含まれて、MOFによって提供されるアクセス手段を使用して生成される。「package object」のインタフェースは、JMIの「javax.jmi.reflect.RefPackage」を継承し「パッケージ名＋”Package”」という名前になる。

- Class Proxy Object

  「class proxy object」のインタフェースはクラス属性の状態にアクセスし、更新するためのメソッドを提供する。また、インタフェースは「instance object」を生成するファクトリーメソッドを提供する。前述のライフサイクル管理の説明に出てきた「FooClass」がこのオブジェクトにあたる。「class proxy object」のインタフェースは、JMIの「javax.jmi.reflect.RefClass」を継承し「クラス名＋”Class”」という名前になる。

- Instance Object

  「instance object」は、インスタンス属性や定数を保持する。また JMI リファレンスを利用して関連を操作する。一般的に複数の「instance object」が「package object」に含まれる。「instance object」は常に「class proxy object」によって管理されている。「class proxy object」は「instance object」を生成するためのメソッドを提供する。「instance object」が生成される際に、「instance object」は自動的に「class proxy object」のコンテナに追加される。逆に「instance object」が削除される際は、コンテナよりインスタンスを取り除かれる。前述のライフサイクル管理の説明に出てきた「Foo」インタフェースがこのオブジェクトにあたる。「instance object」のインタフェースは、JMI の「javax.jmi.reflect.RefObject」を継承し「クラス名」という名前になる。

- Association Object

  「association object」はメタモデルで定義されている関連をリンクとして保持する。「association object」は管理するという意味で「class proxy object」に類似している。「association object」は関連端の「instance object」のインスタンスの取得や関連の追加、修正、削除の操作を提供する。尚、リンクとは、「association object」によって接続された２つのクラス(「instance object」)のインスタンス間の物理的なリンクを表す「association object」のインスタンスのことである。「association object」のインタフェースは、JMI の「javax.jmi.reflect.RefAssociation」を継承し「関連名」という名前になる。

モデルとこれら４種類のメタオブジェクトまたはインタフェースの関係を図 1-１４に示す。図の左側はモデルにパッケージ P1 と P2、クラス C1 と C2 が含まれる場合を示す。すると、図の右側に示すインタフェースが生成される。即ち P1 から P1Package, P2 から P2Package, C1 から C1 と C1Class,C2 から C2 と C2Class が生成される(ステレオタイプでどのモデルから生成されたかを図 1-１４で表している)。

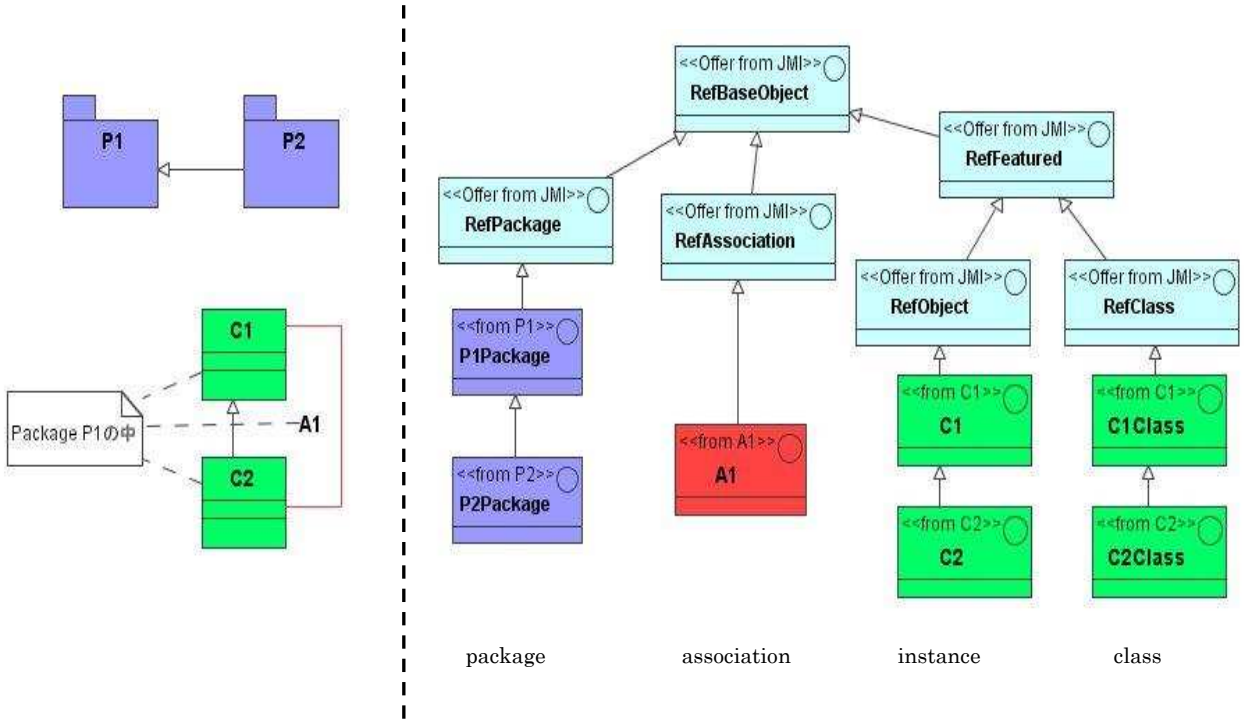

図 1-14 モデルと生成されるメタオブジェクト(インタフェース)

ステレオタイプ<<Offer from JMI>>がついたインタフェースは JMI が提供しているインタフェースであり、JMI のリフレクション機能を提供している(Reflective Package と呼ぶ)。各種インタフェースは次の継承パターンに従う。

- 「instance object」の場合：他の「instance object」を継承しない限りは、「javax.jmi.reflect.RefObject」を継承する。
- 「package object」の場合：他の「package object」を継承しない限りは、「javax.jmi.reflect.RefPackage」を継承する。
- すべての「class proxy object」は「javax.jmi.reflect.RefClass」を継承する。
- すべての「association object」は「javax.jmi.reflect.RefAssociation」を継承する。

上記で説明しているメタオブジェクトは JMI の実装(例えば NetBeans MDR)によってメタモデルから自動的に生成され、ライフサイクル管理の機能やリフレクションの機能を持つ。なお、XML のインポートとエクスポート機能については JMI の「javax.jmi.xmi」パッケージにてインタフェースが提供されている。AndroMDA で利用している UML メタモデルは NetBeans MDR の XML(XMI)ファイル出力機能を持つクラス(JMI のインタフェースを実装している)によって生成されている。その XML ファイルは Maven リポジトリ(AndroMDA を使用していて特に変更していなければ「C:/Documents and Settings/ログインユーザ名/.maven/repository」)の

「jmi/jar/jmiuml-1.4di.jar」に含まれている。その jar ファイルには XML より生成したインタフェースのクラスファイルも含まれている。jar に含まれているインタフェースというのがメタオブジェクトである。

これでメタモデルからどのようなメタオブジェクトが生成されるか理解できたかと思う。次に上記で説明したメタモデルから生成されるメタオブジェクトの実際のコード内容を説明する。ここでは AndroMDA で利用する UML メタモデルを用いて説明する。まず、メタモデルである UML メタモデルの一部(Core パッケージの一部)を下記に図示する。

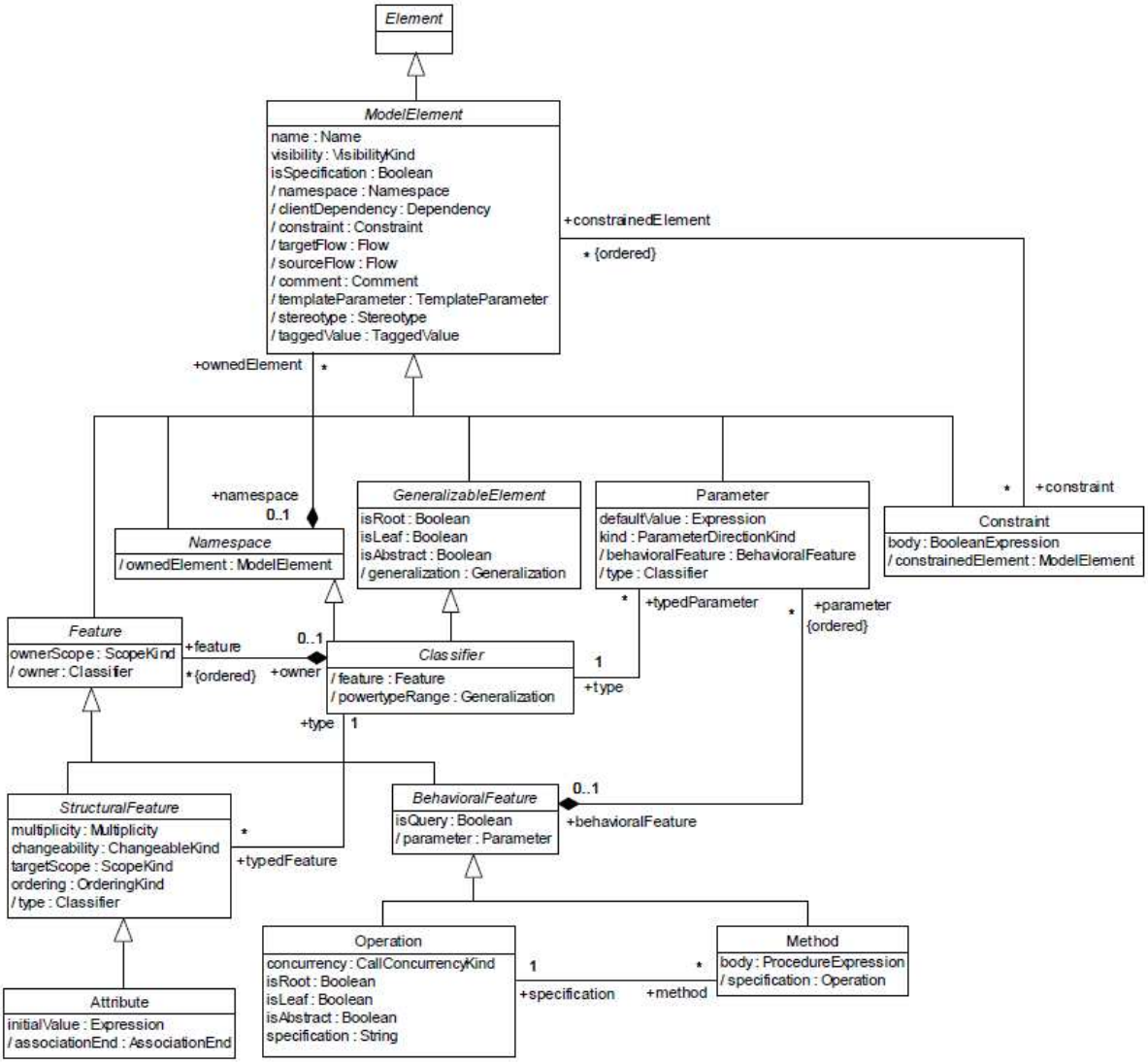

図 1-15 UML メタモデル(Core Package)(OMG Unified Modeling Language Specification より抜粋)

この UML メタモデルの図は、1．1 で説明した 4 階層構造の M2 層であるメタモデル層にあたる。NetBeans MDR が UML メタモデルを定義する XMI ファイル(具体的なファイル名は 2 章)を読み込んだ際にモデル要素が自動的に生成される。ここでは図 1-15 のモデル要素「Method」を例にして説明する。

```
・・・
<Model:Class xmi.id = 'a761' name = 'Method' annotation = '' isRoot = 'false'
  isLeaf = 'false' isAbstract = 'false' visibility = 'public_vis' isSingleton = 'false'>
  <Model:Namespace.contents>
    <Model:Attribute xmi.id = 'a762' name = 'body' annotation = '' scope = 'instance_level'
      visibility = 'public_vis' isChangeable = 'true' isDerived = 'false'>
      <Model:StructuralFeature.multiplicity>
        <XMI.field>1</XMI.field>
        <XMI.field>1</XMI.field>
        <XMI.field>false</XMI.field>
        <XMI.field>false</XMI.field>
      </Model:StructuralFeature.multiplicity>
      <Model:TypedElement.type>
        <Model:Class xmi.idref = 'a763'/>
      </Model:TypedElement.type>
    </Model:Attribute>
    <Model:Reference xmi.id = 'a764' name = 'specification' annotation = ''
      scope = 'instance_level' visibility = 'public_vis' isChangeable = 'true'>
      <Model:StructuralFeature.multiplicity>
        <XMI.field>1</XMI.field>
        <XMI.field>1</XMI.field>
        <XMI.field>false</XMI.field>
        <XMI.field>false</XMI.field>
      </Model:StructuralFeature.multiplicity>
      <Model:TypedElement.type>
        <Model:Class xmi.idref = 'a215'/>
      </Model:TypedElement.type>
      <Model:Reference.referencedEnd>
        <Model:AssociationEnd xmi.idref = 'a765'/>
      </Model:Reference.referencedEnd>
    </Model:Reference>
  </Model:Namespace.contents>
  <Model:GeneralizableElement.supertypes>
    <Model:Class xmi.idref = 'a579'/>(→ <Model:Class/>の「BehavioralFeature」)
  </Model:GeneralizableElement.supertypes>
</Model:Class>
・・・
```

```
<Model:Association xmi.id = 'a856' name = 'A_specification_method' annotation = ''
  isRoot = 'true' isLeaf = 'true' isAbstract = 'false' visibility = 'public_vis'
  isDerived = 'false'>
  <Model:Namespace.contents>
    <Model:AssociationEnd xmi.id = 'a765' name = 'specification' annotation = ''
      isNavigable = 'true' aggregation = 'none' isChangeable = 'true'>
      <Model:AssociationEnd.multiplicity>
        <XMI.field>1</XMI.field>
        <XMI.field>1</XMI.field>
        <XMI.field>false</XMI.field>
        <XMI.field>false</XMI.field>
      </Model:AssociationEnd.multiplicity>
      <Model:TypedElement.type>
        <Model:Class xmi.idref = 'a215'/>(→ <Model:Class/>の「Operation」)
      </Model:TypedElement.type>
    </Model:AssociationEnd>
    <Model:AssociationEnd xmi.id = 'a857' name = 'method' annotation = '' isNavigable =
'true' aggregation = 'none' isChangeable = 'true'>
      <Model:AssociationEnd.multiplicity>
        <XMI.field>0</XMI.field>
        <XMI.field>-1</XMI.field>
        <XMI.field>false</XMI.field>
        <XMI.field>true</XMI.field>
      </Model:AssociationEnd.multiplicity>
      <Model:TypedElement.type>
        <Model:Class xmi.idref = 'a761'/>(→ <Model:Class/>の「Method」)
      </Model:TypedElement.type>
    </Model:AssociationEnd>
  </Model:Namespace.contents>
</Model:Association>
・・・
```

図 1-16 UML メタモデルの XML 記述の一部

図 1-16はUMLメタモデルを定義するXMIの一部である。M2における「Method」クラスはM3の「Class」に対応する。M3の「Class」は図 1-9からわかるように「Namespace」と「GeneralizableElement」を継承しているため、それらの属性値を与える必要がある。まず、「Namespace.contents」より「Method」クラスは”body”と命名されたM3における「Attribute」と”spacification”と命名された「Reference」の包括を定義する。また、「GeneralizableElement.supertypes」により、汎化関係を定義している。

この定義から生成された「instance object」のインタフェースを抜き出して記す。

- **Method (instance object)**

```
package org.omg.uml.foundation.core;
public interface Method extends BehavioralFeature {
   public ProcedureExpression getBody();
   public void setBody(ProcedureExpression body);
   public Operation getSpecification();
   public void setSpecification(Operation specification);
}
```

属性や参照名よりGetter/Setterが生成されている。ただし、すべての属性や操作、参照が定義されるわけではない(図 1-16の場合「Model:Attribute」タグや「Model:Reference」タグにある「visibility」属性の値が「public_vis」であり「scope」属性が「instance_level」のもののみ定義される)。前述したが、「instance object」は「javax.jmi.reflect.RefObject」を継承する。しかし、「Method」は「BehavioralFeature」を継承している(「Model:GeneralizableElement.supertypes」で指定されている)ため「javax.jmi.reflect.RefObject」ではなく「BehavioralFeature」の「instance object」を継承する。なお、パッケージ名「org.omg.uml.fundation.core」はUMLメタモデルを定義するXMIファイルに記されている。

次に XMI から生成された「class proxy object」インタフェースを抜き出して記す。

・ MethodClass (class proxy object)

```
package org.omg.uml.foundation.core;
public interface MethodClass extends javax.jmi.reflect.RefClass {
   public Method createMethod();
   public Method createMethod(String name,
                              VisibilityKind visibility,
                              boolean isSpecification,
                              ScopeKind scopeKind,
                              boolean isQuery,
                              ProcedureExpression body);
}
```

このクラスは「javax.jmi.reflect.RefClass」を継承し、「instance object」の Factory メソッドを定義していることが分かる。Factory メソッドには、デフォルトコンストラクタで生成するメソッドと、モデル要素の属性を引数に取るコンストラクタで生成するメソッドの 2 種類がある。ただし、属性を引数に取るコンストラクタは生成対象の「instance object」が他の「instance object」を継承している場合は、継承元の属性も引数に取る。なお、モデル要素が抽象モデル要素だった場合は、Factory メソッドは生成されない。

次に図 1-15より「Method」クラスは「Operation」クラスと関連を持つの分かる。この関連についても図 1-16の XMI に定義されている。この定義から生成される「association object」インタフェースを抜き出して記す。

・ ASpecificationMethod (association object)

```
package org.omg.uml.foundation.core;
public interface ASpecificationMethod extends javax.jmi.reflect.RefAssociation {
    public boolean exists(Operation operation, Method method);
    public Operation getSpecification(Method method);
    public Collection getMethod(Operation operation);
    public boolean add(Operation operation, Method method);
    public boolean remove(Operation operation, Method method);
}
```

このクラスは「javax.jmi.reflect.RefAssociation」を継承し、関連についての操作を定義しているのが分かる。各関連端の多重度(図 1-16の「Model:AssociationEnd.multiplicity」のはじめの 2 つの要素「XMI.field」)により各関連端を取得するメソッドの戻り値の型が決定する(「Method」であれば 0、-1 のため Collection または List、「Operation」は 1、1 のため要素の型)。

次に生成された「package object」として、インタフェース「CorePackage」を示す。

・ CorePackage (package object)

```
package org.omg.uml.foundation.core;
public interface CorePackage extends javax.jmi.reflect.RefPackage {
    public MethodClass getMethod();
    public ClassifierClass getClassifier();
    public FeatureClass getFeature();
    public ASpecificationMethod getASpecificationMethod();
・・・
}
```

各メソッドの戻り値から、「CorePackage」パッケージ配下の「package object」、「class proxy object」、「association object」のスタティック情報を統括しているディレクトリと考えることができる。

これらの生成されるメタオブジェクトを見ればわかるとおり、メタモデルの内容によって生成されるメタオブジェクトの内容も変化する。そのため、JMI の実装(AndroMDA では NetBeans MDR)ではメタモデルから自動的にメタオブジェクトを生成する機能を提供する。「class proxy object」や「instance object」からわかるようにインスタンスの生成や変更など、ライフサイクル管理の機能も提供する。そして生成する情報が記述されている XML ファイル(XMI 形式)についても JMI の実装によってインポートおよびエクスポートが可能になっている。

最後に JMI のリフレクション機能について説明する。JMI のリフレクション機能は、アプリケーションでモデルを扱う際にランタイムで問い合わせることを可能にする。実際にリフレクション機能を利用すると、自分を定義している層、例えば、M2(メタモデル)層であれば M3(メタ-メタモデル)層にアクセスすることが可能になる。また、ライフサイクル管理の説明で紹介した「class proxy object」から自身が生成することができる「instance object」の一覧を取得することなどが可能となる。実際にリフレクション機能を提供しているものは、メタオブジェクトの説明で何度か出てきた「javax.jmi.reflect」パッケージのインタフェースのことである。

ここでは「class proxy object」の自身が生成することができる「instance object」の一覧を取得する機能を利用して、読み込んだ UML モデルのクラス要素のすべての名前を表示するプログラムを紹介する。なお、UML のクラス要素は、UML メタモデルの「Class」要素で定義されている。UML メタモデルの「Class」要素から生成される「instance object」や「class proxy object」をここでは「UmlClass」、「UmlClassClass」とする。読み込ませる UML モデルを下記に図示する。

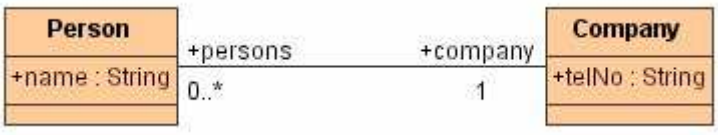

図 1-17 読み込む UML モデル

クラス要素の名前を表示するメソッドの内容を下記に記す。

```
import org.omg.uml.foundation.core.CorePackage;
import org.omg.uml.foundation.core.UmlClassClass;
import org.omg.uml.foundation.core.UmlClass;
・・・
    public void printAllUMLClass(CorePackage core) {
        // UmlClass の proxy object を取得する
        UmlClassClass umlClassClass = core.getUmlClass();
        // UmlClassClass で生成する instance object をすべて取得する
        Collection instances = umlClassClass.refAllOfType();
        //  すべての UmlClass の名前を表示する
        for (Iterator itr = instances.iterator();itr.hasNext();) {
                UmlClass umlClass = (UmlClass)itr.next();
                System.out.println("UmlClass = " + umlClass.getName());
        }
    }
```

図 1-18 クラス要素名表示メソッド

このメソッドは、「CorePackage」インタフェースを引数にとり、「class proxy object」である「UmlClassClass」を取得して、「UmlClassClass」が生成できる「instance object」の「UmlClass」をすべて取得してその名前を表示する。名前を表示するメソッドは、「UmlClass」の継承元をたどると見つかる「ModelElement」のインタフェースで定義されている(getName メソッド)。「class proxy object」である「UmlClassClass」に対して「refAllOfType」メソッドを呼び出している部分が JMI のリフレクション機能を利用している。「refAllOfType」メソッドは JMI のリフレクション機能であり、この「class proxy object」に生成することが可能なインスタンスのすべてを返すメソッドである。このようなリフレクション機能を定義するメソッドなどが JMI の仕様で定義されている。下記にリフレクション機能(Reflective Package)で提供しているインタフェースとそのメソッドの一部を紹介する。

- RefBaseObject

  RefPackage や RefFeature、RefAssociation などのスーパーインタフェースである。メソッドは以下の通りである。

  ・refMofId

  このメタオブジェクトのユニークな識別子を返す。

  ・refMetaObject

  このメタオブジェクトで規定されている RefObject を返す。

  ・refImmediatePackage

  このメタオブジェクトが属しているまたはこのメタオブジェクトを集約するパッケージの「package object」を返す。

  ・refOutermostPackage

  このメタオブジェクトを含んでいる最も外側のパッケージの「package object」を返す。

- RefFeature

  RefObject や RefClass のスーパーインタフェース。このメタオブジェクトは特徴(属性、操作、参照)を表している。メソッドは以下の通りである。

  ・refGetValue

  このメタオブジェクトが表す属性や参照の値を返す。

  ・refSetValue

  このメタオブジェクトが表す属性や参照に値を割り当てる。

  ・refInvokeOperation

  このメタオブジェクトが表す操作を実行する。

- RefAssociation

  関連のメタオブジェクト(association object)を表す。関連のインスタンスであるリンクについての操作を提供する。メソッドは以下の通りである。

  ・refAllLinks

  このメタオブジェクトが管理するリンクのすべてを返す。コレクションで返ってくるが中身は RefAssociationLink というすべての関連のリンクによって実装されているリフレクションインタフェースであり、RefAssociation のインスタンスである。

  ・refLinkExists

  このメタオブジェクトが管理するリンクに指定のメタオブジェクト２つを関連端に持つのリンクが存在するかを返す。指定するメタオブジェクトは関連端のメタオブジェクト２つである。

  ・refAddLink

  このメタオブジェクトに新しいリンクを追加する。

  ・refRemoveLink

  このメタオブジェクトから指定のリンクを削除する。

- RefPackage

  RefClass や RefAssociation、RefPackage を保有する「package object」を表す。メソッドは以下の通りである。

  ・refClass

  このメタオブジェクトが保有する指定の「class proxy object」を返す。

  ・refAssociation

  このメタオブジェクトが保有する指定の「association object」を返す。

  ・refPackage

  このメタオブジェクトが保有する指定の「package object」を返す。

  ・refAllPackages

  このメタオブジェクトが保有するすべての「package object」を返す。

  ・refAllClasses

  このメタオブジェクトが保有するすべての「class proxy object」を返す。

  ・refAllAssociations

  このメタオブジェクトが保有するすべての「association object」を返す。

  ・refDelete

  このメタオブジェクトを削除する。このメタオブジェクトが保有するメタオブジェクトも削除される。

- RefClass

  「class proxy object」を表す。メソッドは以下の通りである。

  ・refCreateInstance

  このメタオブジェクトで生成できる「instance object」を生成する。

  ・refAllOfType

  このメタオブジェクトで生成できるすべて(サブタイプも含める)の「instance object」を返す。

  ・refAllOfClass

  このメタオブジェクトで生成できるすべて(サブタイプは含まない)の「instance object」を返す。

- RefObject

  「instance object」を表す。メソッドは以下の通りである。

  ・refClass

  このメタオブジェクトを生成する「class proxy object」を返す。

  ・refDelete

  このメタオブジェクトに含まれるメタオブジェクトも含めて削除する。

１．４　XML Metadata Interchange (XMI)

XML Metadata Interchange(XMI)は、MOF の仕様に基づいたどのような種類のメタデータでも置き換えることをサポートする仕様である。なお、XMI の仕様は XML に基づく。一般的には、異なる UML ツール間でのモデルのメタデータを交換するために利用する。XMI の仕様には２つの主要構成要素がある。

- XMI のメタデータエンコードのための生成規則を XML DTD を用いて作成している。XML DTD は XMI ドキュメントのために用いられ、一般的な XML ツールが XMI ドキュメントの構成および有効性の確認に利用する。
- XML 互換性を持つフォーマットでメタデータをエンコードするための XML ドキュメント生成規則である。生成規則は、XMI ドキュメントをデコードし、メタデータを再構築するために利用することもできる。

主要構成要素である XMI の DTD に定まっている要素のいくつかを下記に記す。

- XMI.header
  XMI のヘッダ情報を記述する要素。この要素の下にも XMI ドキュメントの生成物やバージョンなどを記述する「XMI.document」、XMI で表しているメタモデルの情報を記述する「XMI.metamodel」などがある。
- XMI.content
  実際のモデル(XMI.header の XMI.metamodel)の内容を記述する要素。この要素の内部に UML や MOF のモデル情報などを記述する。
- XMI.extensions
  自由に拡張できる要素である。例えば UML ツール独自の情報(モデルの座標情報など)をこの要素の内部に記述する。この内部に記述することによって他のツールはこの要素を無視して読み込みなどができるようになる。

以上の XMI の DTD に付加する形で MOF 用、UML 用にも DTD が存在する。尚、MOF の DTD はメタモデルの M3 層、UML の DTD はメタモデルの M2 層に対応する。 下記に XMI の XML ファイルと UML の XMI 記述での XML ファイルの例を図示する。

```
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
<XMI xmi.version = '1.2' xmlns:Model = 'org.omg.xmi.namespace.Model' timestamp = 'Wed Aug
28 15:03:50 CEST 2002'>
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>Netbeans XMI Writer</XMI.exporter>
      <XMI.exporterVersion>1.0</XMI.exporterVersion>
    </XMI.documentation>
  </XMI.header>
  <XMI.content>
    <Model:Package xmi.id = 'a1' name = 'X M L Model' annotation = '' isRoot = 'false'
      isLeaf = 'false' isAbstract = 'false' visibility = 'public_vis'>
      <Model:Namespace.contents>
        <Model:Class xmi.id = 'a2' name = 'Node' annotation = '' isRoot = 'false'
          isLeaf = 'false' isAbstract = 'false' visibility = 'public_vis' isSingleton =
'false'>
          <Model:Namespace.contents>
            <Model:Reference xmi.id = 'a3' name = 'elements' annotation = ''
            scope = 'instance_level' visibility = 'public_vis' isChangeable = 'true'>
              <Model:StructuralFeature.multiplicity>
                <XMI.field>0</XMI.field>
                <XMI.field>-1</XMI.field>
                <XMI.field>true</XMI.field>
                <XMI.field>true</XMI.field>
              </Model:StructuralFeature.multiplicity>
              <Model:TypedElement.type>
                <Model:Class xmi.idref = 'a4'/>
              </Model:TypedElement.type>
              <Model:Reference.referencedEnd>
                <Model:AssociationEnd xmi.idref = 'a5'/>
              </Model:Reference.referencedEnd>
            </Model:Reference>
・・・
</XMI>
```

図 1-19 XMI の XML ファイル例(一部)

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE XMI SYSTEM 'UML_1.4_XMI_1.1.dtd'>
<XMI xmi.version='1.2' xmlns:UML='omg.org/UML/1.4'>
  <XMI.header>
    <XMI.metamodel xmi.name='UML' xmi.version='1.4'/>
  </XMI.header>
  <XMI.content>
    <UML:Model xmi.id='S.1' name='Employment Model' visibility='public'
     isSpecification='false' isRoot='false' isLeaf='false' isAbstract='false'>
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id='S.2' name='Person' visibility='public'
          isSpecification='false' namespace='S.1' isRoot='true' isLeaf='true'
          isAbstract='false' isActive='false'/>
        <UML:Class xmi.id='S.3' name='Business' visibility='public'
          isSpecification='false' namespace='S.1' isRoot='true' isLeaf='true'
          isAbstract='false' isActive='false'/>
        <UML:Association xmi.id='G.1' name='Employment' visibility='public'
         isSpecification='false' isRoot='false' isLeaf='false' isAbstract='false'>
          <UML:Association.connection>
            <UML:AssociationEnd name='employer' visibility='public'
              isSpecification='false' isNavigable='true' ordering='unordered'
              aggregation='none' targetScope='instance'
             changeability='changeable' participant='S.3' association='G.1'>
・・・
</XMI>
```

図 1-20 UML の XMI 形式の XML ファイル例(一部)(OMG Unified Modeling Language Specification より引用)

下記に異なるツール間でのファイルの交換可能の様子を図示する。

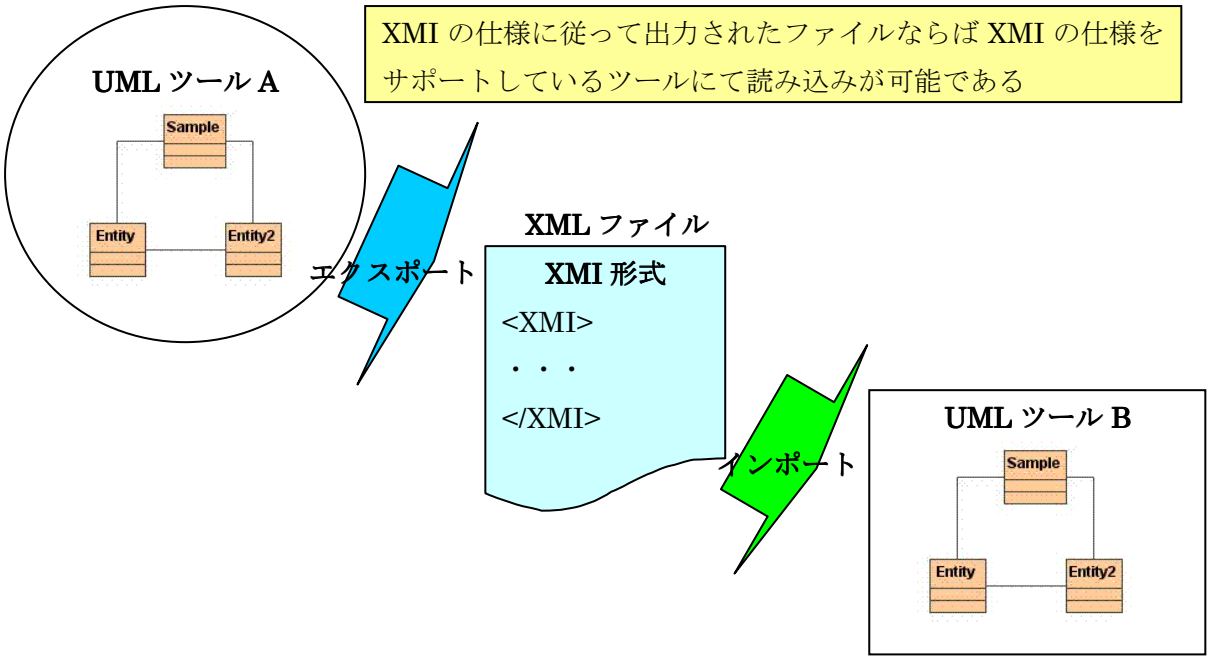

図 1-21 異なるツール間での XML ファイルの交換の様子

※ 本書で説明している XMI のバージョンは 1.2 であるが、XMI の最新バージョンは 2.1 である。2.1 では DTD でなく XMLSchema で定義されている。

２ AndroMDA を分解する

本章では、AndroMDA が使用する要素技術についてサンプルコードを交えて解説する。本章を読み、サンプルコードを追うことで AndroMDA の要素技術がどのようなものであるのか理解できる。また、要素技術を駆使し独自の MDA ツールを作ることも可能になる。

本章は２．１～２．５章まであり、２．１章ではメタデータのリポジトリである「NetBeans MDR」についてと「NetBeans MDR」を用いて UML モデルを読み込む方法を解説する。２．２章では、２．１章で読み込んだモデルを扱うための方法について解説する。２．３章では「AndroMDA Metafacade」を使い、モデルを読み込む方法について解説する。また、「NetBeans MDR」との関係についても解説する。２．５章では、「Velocity」について解説し、「Velocity」と「AndroMDA Metafacade」を使いモデルからクラスを生成するサンプルコードを示す。

２．１　NetBeans MDR (MetaData Repository)

２．１．１　　概要

NetBeans MDR は、モデルを扱うためのパッケージである。MDA で扱いたいモデルは UML モデルなどの M1 であるが、このモデルを扱うには M1 を定義する M2、さらにその M2 を定義する M3 が必要になる。NetBeans MDR には MDRRepository というメタモデルを管理するリポジトリクラスがある。このリポジトリクラスを使い、M1 である UML モデルを読み込む。ただし、UML モデルを読み込むには M2 の UML メタモデルを事前に読み込む必要があり、その UML メタモデルを読み込む為には M3 の MOF モデルを読み込む必要がある。各々のモデルは図 2-1 に示す XMI ファイルで記述されている。NetBeans MDR は M3 を自動的に読み込む(MOF モデルはハードコードされていると思えば良い)ので、UML メタモデル(jmiuml-1.4di.jar 中の M2_DiagramInterchangeModel.xml)と UML モデルの順番でモデルを読み込むことになる。

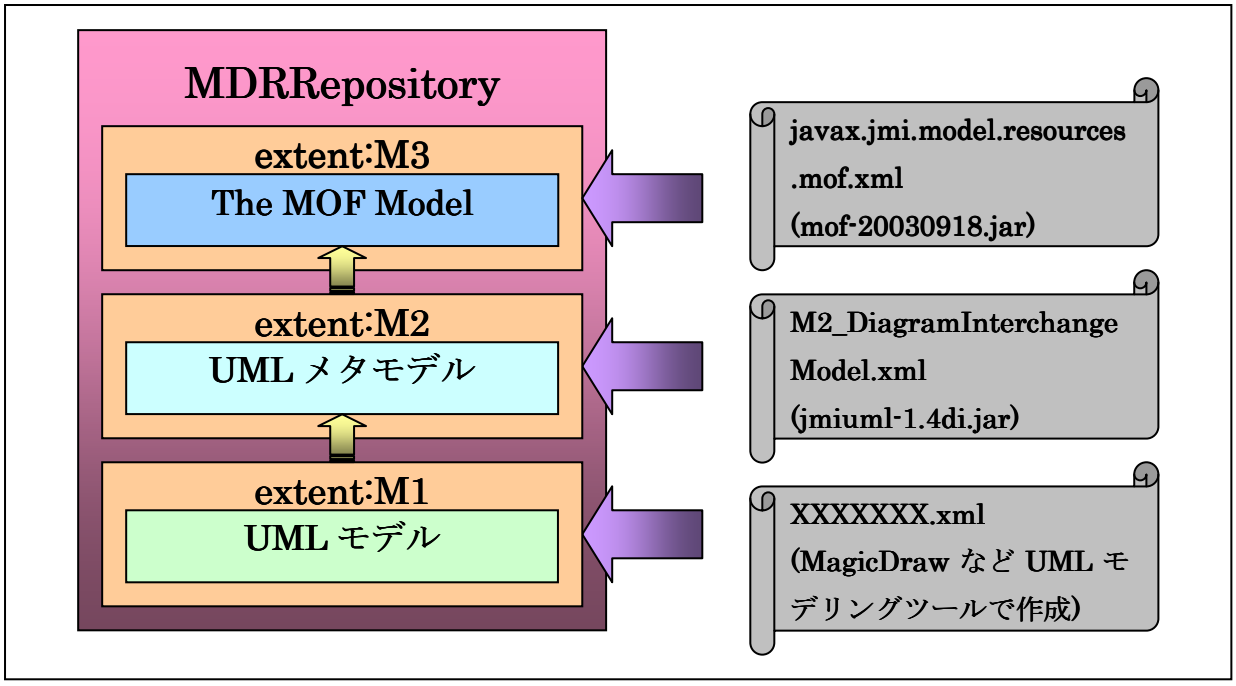

図 2 - 1 MDRRepository

MDRRepository には extent というモデルを管理し、蓄える領域がある。この領域はデータベースにおけるデータ領域に相当する。M3 の extent は MDRRepository が自動的に作成して MOF モデルを蓄積する。そして、M3 の extent に、M2 を読み込む。次に、M2 にある UML メタモデルに従った extent を作成し、そこに、M1 の UML モデルを読み込む手順になる。

２．１．２　　サンプルコード

NetBeans MDR を使い、XMI を読み込むサンプルコード（**MOFRepositry.java**）を以下に示す。また、XMI を読み込むには「org.netbeans.lib.jmi.xmi.InputConfig インタフェース」と「org.netbeans.lib.jmi.xmi.XmiContext」を実装したクラスが必要になるので、今回は「org.andromda.repositories.mdr.MDRXmiReferenceResolver クラス」と「org.andromda.repositories.mdr.MDRXmiReferenceResolverContext クラス」を使用する。モ

デルは、MagicDraw で記述されたモデルなら何でもよいが、ZIP のままではなく解凍し、XML ファイルとして読み込む必要がある。

このサンプルコードの処理の流れを示す。

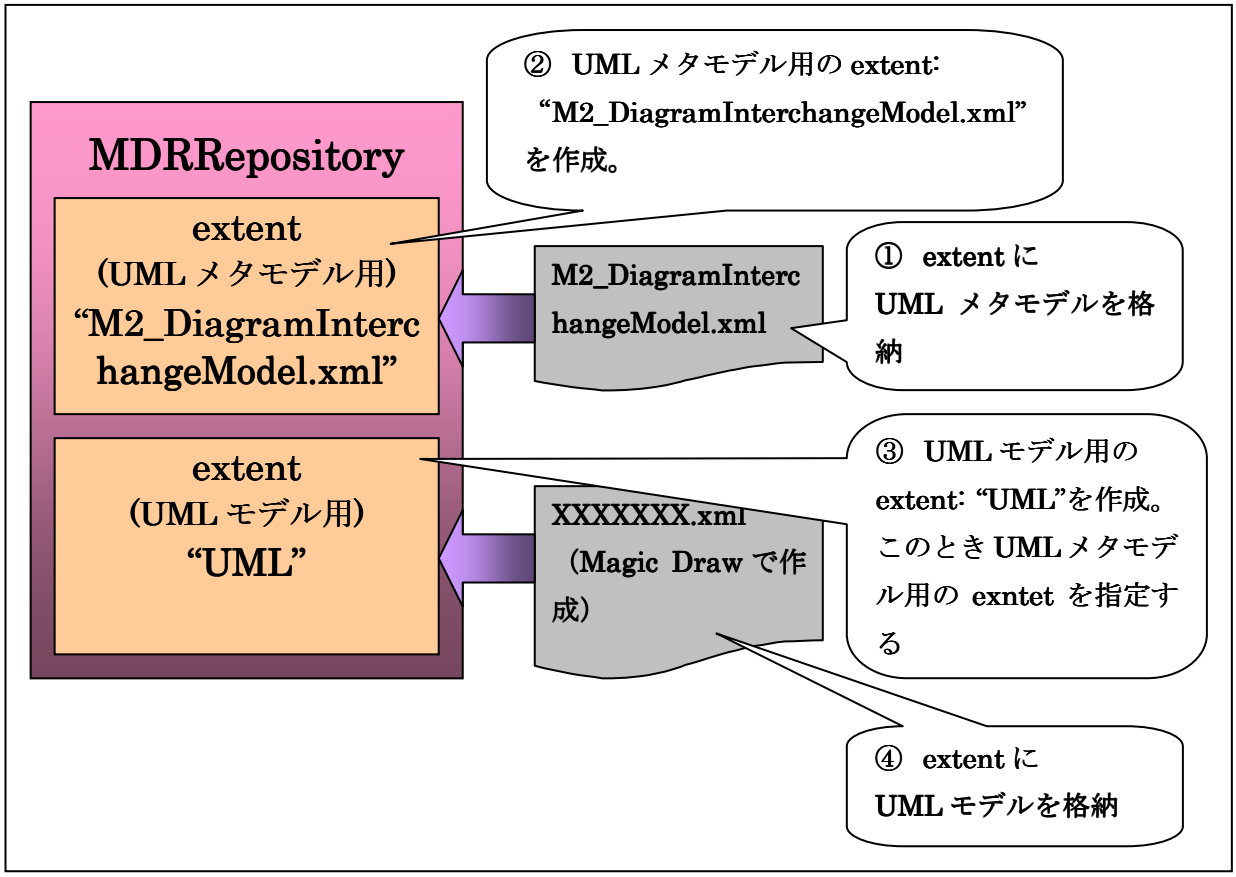

図 2－2 サンプルコードの処理の流れ

MOFRepositry.java

```
 1:import java.io.IOException;
 2:import java.net.URL;
 3:import java.util.Collection;
 4:import java.util.Iterator;
 5:import javax.jmi.model.ModelPackage;
 6:import javax.jmi.model.MofPackage;
 7:import javax.jmi.reflect.RefPackage;
 8:import javax.jmi.xmi.MalformedXMIException;
 9:import org.netbeans.api.mdr.CreationFailedException;
10:import org.netbeans.api.mdr.MDRManager;
11:import org.netbeans.api.mdr.MDRepository;
12:import org.netbeans.api.xmi.XMIReader;
13:import org.netbeans.api.xmi.XMIReaderFactory;
14:import org.omg.uml.UmlPackage;
15:import org.andromda.repositories.mdr.MDRXmiReferenceResolver;
16:public class MOFRepositry {
17:  // UML Model ( M2 ) from Magic Draw
18:  // jmiuml-1.4di.jar
```

```
19:  private final String UML_M2_MODEL = "/M2_DiagramInterchangeModel.xml";
20:  private final String UML_MODEL = "UML_MODEL";
21:  private final String UML_CORE = "Core";
22:  private final String UML_TYPE = "DataType";
23:  private final String UML_STATE = "State_Machines";
24:  private MDRepository repository = null;
25:  private URL metamodelURL = null;
26:  private MofPackage umlPackage = null; //    UML Model ( M2 )
27:
28:  public MOFRepositry(String url) throws Exception {
```

↑ 引き数として XMI ファイル（UML モデル）のパスを指定する。

```
29:              // Set Persitance to Memory
30:              System.setProperty(
31:                       "org.netbeans.mdr.storagemodel.StorageFactoryClassName",
32:                       "org.netbeans.mdr.persistence.memoryimpl.StorageFactoryImpl");
```

↑ モデルを蓄えるためのストレージクラスを指定する。本サンプルでは org.netbeans.mdr.persistence.memoryimpl.StorageFactoryImpl クラスを指定している。

```
33:              // Get Repository
34:              repository = MDRManager.getDefault().getDefaultRepository();
```

↑ モデルを格納するための repository を取得する。この repository にすべてのモデルが格納される。

```
35:              repository.beginTrans(true);
36:              // M2 UML Model
37:              try {
38:                       umlPackage = loadMetaModel(UML_M2_MODEL, UML_MODEL, null);
```

↑ loadMetaModel メソッドをコールし、extent にモデルを読み込む。ここで読み込む XMI ファイルは M2_DiagramInterchangeModel.xml というファイルで、UML メタモデルである。この XMI ファイルは jmiuml-1.4di.jar ファイルの中にある。また、M3 である The MOF Model は MDR が自動で読み込むので、明示的に読み込む必要はない。

```
39:                       System.out.println("MataData Loaded");
40:              } catch (Exception e) {
41:                       System.out.println("Metadata Load Error " + e);
42:                       e.printStackTrace();
43:                       System.exit(-1);
44:              }
45:              readModel(new URL(url));
```

↑ UML モデルを読み込むメソッド readModel をコールする。

```
46:  }
47:  public UmlPackage getUmlPackage() throws Exception {
48:              return (UmlPackage) repository.getExtent("UML");
```

↑ repository から UML モデル用の extent 取得し、リターンする。これが読み込んだ UML モデルになる。

```
49:  }
50:  private MofPackage loadMetaModel(String xmi, String model, MofPackage mof)
51:                       throws CreationFailedException, IOException, MalformedXMIException {
52:              URL url = MOFRepositry.class.getResource(xmi);
53:              if (url == null) {
```

```
54:                     throw new IOException("Can't find XML file: " + xmi);
55:             }
56:             metamodelURL = url;
57:             ModelPackage extent = (ModelPackage) repository.getExtent(url
58:                             .toExternalForm());
```

↑ リポジトリからUMLメタモデル(M2_DiagramInterchangeModel.xml)のextentを取得する。

```
59:             if (extent == null) {
60:                     if (mof == null) {
61:                             extent = (ModelPackage) repository.createExtent(url
62:                                             .toExternalForm());
```

↑ extentがない場合は新しいextentを作成する。The MOF Modelがない場合(mof == null)。

```
63:                     } else {
64:                             extent = (ModelPackage) repository.createExtent(url
65:                                             .toExternalForm(), mof);
```

↑ extentがない場合は新しいextentを作成する。The MOF Modelがある場合。

```
66:                     }
67:             }
68:             MofPackage pack = findPackage("UML", extent);
```

↑ ”UML”という名前のextentを検索し、取得する。

```
69:             if (pack == null) {
```

↑ MDRRepository内に”UML”というextentがない(UMLメタモデルが読み込まれていない)場合、UMLメタモデルを読み込む

```
70:                     XMIReader xmiReader = XMIReaderFactory.getDefault()
71:                                     .createXMIReader();
72:                     Collection elements = xmiReader.read(url.toExternalForm(), extent);
```

↑ urlのXMIを読み込み、extentに格納する。

```
73:                     pack = findPackage("UML", extent);
```

↑ 改めてextent内の”UML”というパッケージを取得する。

```
74:                     System.out.println("Find " + url.toExternalForm() + " = " + pack);
75:             }
76:             return pack;
77: }
78: private void readModel(URL url) throws Exception {
79:             RefPackage extent = repository.getExtent("UML");
```

↑ repositoryから”UML”のextentを取得する。どのextentを取得するかは引数で指定する。

```
80:             if (extent == null) {
81:                     System.out.println("Creating extent");
82:                     extent = repository.createExtent("UML", umlPackage);
```

↑ ”UML”のextentがない場合は、新しい”UML”のextentを作成する。このextentはUMLモデルを読み込むため、UMLメタモデルのextentが必要になる。そのため、umlPackage(38行目で取得している)を引き数として渡している。

```
83:             }
84:             String moduleSearchPath[] = { "./xml" };
85:             XMIReader xmiReader = XMIReaderFactory.getDefault().createXMIReader(
```

↑ XMIReader を作成する。XMIReader を作成するには InputConfg と XmiContext が必要になる。本サンプルではこれらのサブタイプで AndroMDA が提供する MDRXmiReferenceResolver と MDRXmiReferencesResolverContext を使用する。

```
86:                                     new MDRXmiReferenceResolver(new RefPackage[] { extent },
87:                                                     moduleSearchPath));
88:             System.out.println("Start Reading " + url + " .....");
89:             xmiReader.read(url.toExternalForm(), extent);
```

↑ 指定された url の UML モデルを”UML”という名前の extent に読み込ませる。

```
90:             return;
91: }
92: private MofPackage findPackage(String packageName, ModelPackage metaModel) {
93:             for (Iterator it = metaModel.getMofPackage().refAllOfClass().iterator(); it.hasNext();) {
94:                     javax.jmi.model.ModelElement temp = (javax.jmi.model.ModelElement) it.next();
95:                     if (temp.getName().equals(packageName)) {
96:                             return (MofPackage) temp;
97:                     }
98:             }
99:             return null;
100: }
101:}
```

↑ 引数の ModelPackage 以下にある MofPackage の一覧を走査し、引数の名前と一致する MofPackage を検索する。MofPackage の一覧は、ModelPackage#getMofPackage()で取得できる MofPackageClass の refAllOfClass という Reflective API を利用して取得する。

MDRXmiReferenceResolver.java(org.andromda.repositories.mdr パッケージ)

```
 1:package org.andromda.repositories.mdr;
 2:
 3:import org.netbeans.api.xmi.XMIReferenceResolver;
 4:import org.netbeans.lib.jmi.xmi.InputConfig;
 5:import javax.jmi.reflect.RefPackage;
 6:
 7:/**
 8: * @author Matthias Bohlen
 9: * @author Chad Brandon
10: */
11:public class MDRXmiReferenceResolver extends InputConfig
12:     private XMIReferenceResolver referenceResolver;
13:     public MDRXmiReferenceResolver(RefPackage extents[], String[] moduleSearchPath)
14:     {
15:                 this.referenceResolver = new MDRXmiReferenceResolverContext(extents, this,
    moduleSearchPath);
16:     }
17:     public void setReferenceResolver(XMIReferenceResolver arg0)
18:     {
19:         throw new IllegalStateException("MDRXmiReferenceResolver.setReferenceResolver must not
    be implemented!");
20:     }
21:     public XMIReferenceResolver getReferenceResolver()
22:     {
23:         return referenceResolver;
24:     }
25:}
```

MDRXmiReferenceResolverContext.java(org.andromda.repositories.mdr パッケージ)

```
 1:package org.andromda.repositories.mdr;
 2:
 3:import org.apache.commons.lang.StringUtils;
 4:import org.apache.log4j.Logger;
 5:import org.netbeans.api.xmi.XMIInputConfig;
 6:import org.netbeans.lib.jmi.xmi.XmiContext;
 7:import javax.jmi.reflect.RefPackage;
 8:import java.io.File;
 9:import java.io.InputStream;
10:import java.net.MalformedURLException;
11:import java.net.URL;
12:import java.util.HashMap;
13:
14:/**
15: * This class supports the expansion of XML HREF references to other modules within a model. The
    result of the resolver
16: * should be a valid URL. This is necessary for Magic Draw as it doesn't have the entire model referenced
    but just the
17: * archived model.
18: *
19: * @author Matthias Bohlen
20: * @author Chad Brandon
21: */
22:public class MDRXmiReferenceResolverContext
23:        extends XmiContext
24:{
25:    private String[] moduleSearchPath;
26:    private static Logger logger =
Logger.getLogger(MDRXmiReferenceResolverContext.class);
27:    private static final HashMap urlMap = new HashMap();
28:    public MDRXmiReferenceResolverContext(RefPackage[] extents, XMIInputConfig config, String[]
    moduleSearchPath)
29:    {
30:        super(extents, config);
31:        this.moduleSearchPath = moduleSearchPath;
32:    }
33:
34:    public URL toURL(String systemId)
35:    {
36:        if (logger.isDebugEnabled())
37:            logger.debug("attempting to resolve Xmi Href --> '" + systemId + "'");
38:
39:        final String suffix = getSuffix(systemId);
40:
41:        // if the model URL has a suffix of '.zip' or '.jar', get
42:        // the suffix without it and store it in the urlMap
43:        String exts = "\\.jar|\\.zip";
44:        String suffixWithExt = suffix.replaceAll(exts, "");
45:        URL modelUrl = (URL)urlMap.get(suffixWithExt);
46:
47:        // Several tries to construct a URL that really exists.
```

```
48:         if (modelUrl == null)
49:         {
50:             // If systemId is a valid URL, simply use it
51:             modelUrl = this.getValidURL(systemId);
52:             if (modelUrl == null)
53:             {
54:                 // Try to find suffix in module list.
55:                 String modelUrlAsString = findModuleURL(suffix);
56:                 if (StringUtils.isNotBlank(modelUrlAsString))
57:                 {
58:                     modelUrl = getValidURL(modelUrlAsString);
59:                 }
60:                 if (modelUrl == null)
61:                 {
62:                     // search the classpath
63:                     modelUrl = this.findModelUrlOnClasspath(systemId);
64:                 }
65:                 if (modelUrl == null)
66:                 {
67:                     // Give up and let superclass deal with it.
68:                     modelUrl = super.toURL(systemId);
69:                 }
70:             }
71:             // if we've found the module model, log it
72:             // and place it in the map so we don't have to
73:             // find it if we need it again.
74:             if (modelUrl != null)
75:             {
76:                 logger.info("Referenced model --> '" + modelUrl + "'");
77:                 urlMap.put(suffixWithExt, modelUrl);
78:             }
79:         }
80:         System.out.println("MDRXmiReferenceResolverContext#toURL:"+ systemId +"->"+ modelUrl );
81:         return modelUrl;
82:     }
83:     private String findModuleURL(String moduleName)
84:     {
85:         if (moduleSearchPath == null)
86:             return null;
87:
88:         if (logger.isDebugEnabled())
89:             logger.debug("findModuleURL: moduleSearchPath.length=" + moduleSearchPath.length);
90:         for (int i = 0; i < moduleSearchPath.length; i++)
91:         {
92:             File candidate = new File(moduleSearchPath[i], moduleName);
93:             if (logger.isDebugEnabled())
94:                 logger.debug("candidate '" + candidate.toString() + "' exists=" + candidate.exists());
95:             if (candidate.exists())
96:             {
97:                 String urlString;
98:                 try
99:                 {
100:                    urlString = candidate.toURL().toExternalForm();
```

```
101:                    }
102:                    catch (MalformedURLException e)
103:                    {
104:                        return null;
105:                    }
106:
107:                    if (moduleName.endsWith(".zip") || moduleName.endsWith(".jar"))
108:                    {
109:                        // typical case for MagicDraw
110:                                urlString = "jar:" + urlString + "!/" + moduleName.substring(0,
    moduleName.length() - 4);
111:                    }
112:                    return urlString;
113:                }
114:            }
115:            return null;
116:        }
117:        private String getSuffix(String systemId)
118:        {
119:            int lastSlash = systemId.lastIndexOf("/");
120:            if (lastSlash > 0)
121:            {
122:                String suffix = systemId.substring(lastSlash + 1);
123:                return suffix;
124:            }
125:            return systemId;
126:        }
127:        private URL findModelUrlOnClasspath(String systemId)
128:        {
129:            String modelName = StringUtils.substringAfterLast(systemId, "/");
130:            String dot = ".";
131:            // remove the first prefix because it may be an archive
132:            // (like magicdraw)
133:            modelName = StringUtils.substringBeforeLast(modelName, dot);
134:
135:            final ClassLoader loader = Thread.currentThread().getContextClassLoader();
136:
137:            URL modelUrl = loader.getResource(modelName);
138:            if (modelUrl == null)
139:            {
140:                if (CLASSPATH_MODEL_SUFFIXES != null && CLASSPATH_MODEL_SUFFIXES.length >
    0)
141:                {
142:                    int suffixNum = CLASSPATH_MODEL_SUFFIXES.length;
143:                    for (int ctr = 0; ctr < suffixNum; ctr++)
144:                    {
145:                        if (logger.isDebugEnabled())
146:                        logger.debug("searching for model reference --> '" + modelUrl + "'");
147:                        String suffix = CLASSPATH_MODEL_SUFFIXES[ctr];
148:                        modelUrl = loader.getResource( modelName + dot + suffix );
149:                        if (modelUrl != null)
150:                        {
151:                            break;
```

```
152:                            }
153:                        }
154:                    }
155:            }
156:            return modelUrl;
157:        }
158:        private URL getValidURL(String systemId)
159:        {
160:            InputStream stream = null;
161:            URL url = null;
162:            try
163:            {
164:                url = new URL(systemId);
165:                stream = url.openStream();
166:                stream.close();
167:            }
168:            catch (Exception e)
169:            {
170:                url = null;
171:            }
172:            finally
173:            {
174:                stream = null;
175:            }
176:            return url;
177:        }
178:}
```

２．２　NetBeans MDR の利用

２．２．１　　概要

２．１では UML モデルを読み込み、メモリ上に展開する（MDRRepository に格納する）方法について解説した。本章ではメモリ上に展開されたモデルを利用する方法について解説する。

MDRRepository に格納されたモデルは、MDRRepository#getExtent(String)によって取り出す（String は extent 名）。取り出したモデルが UML モデルの場合は、UML モデルのルート要素の「org.omg.uml.UmlPackage$Impl」のインスタンスがリターンされる。このインスタンスをナビゲートすることで目的のクラスや属性にアクセスできる。また、「org.omg.uml.UmlPackage$Impl」がリターンされると記述したが、これは NetBeans MDR が UML メタモデルを読み込み、動的に生成したクラスである。

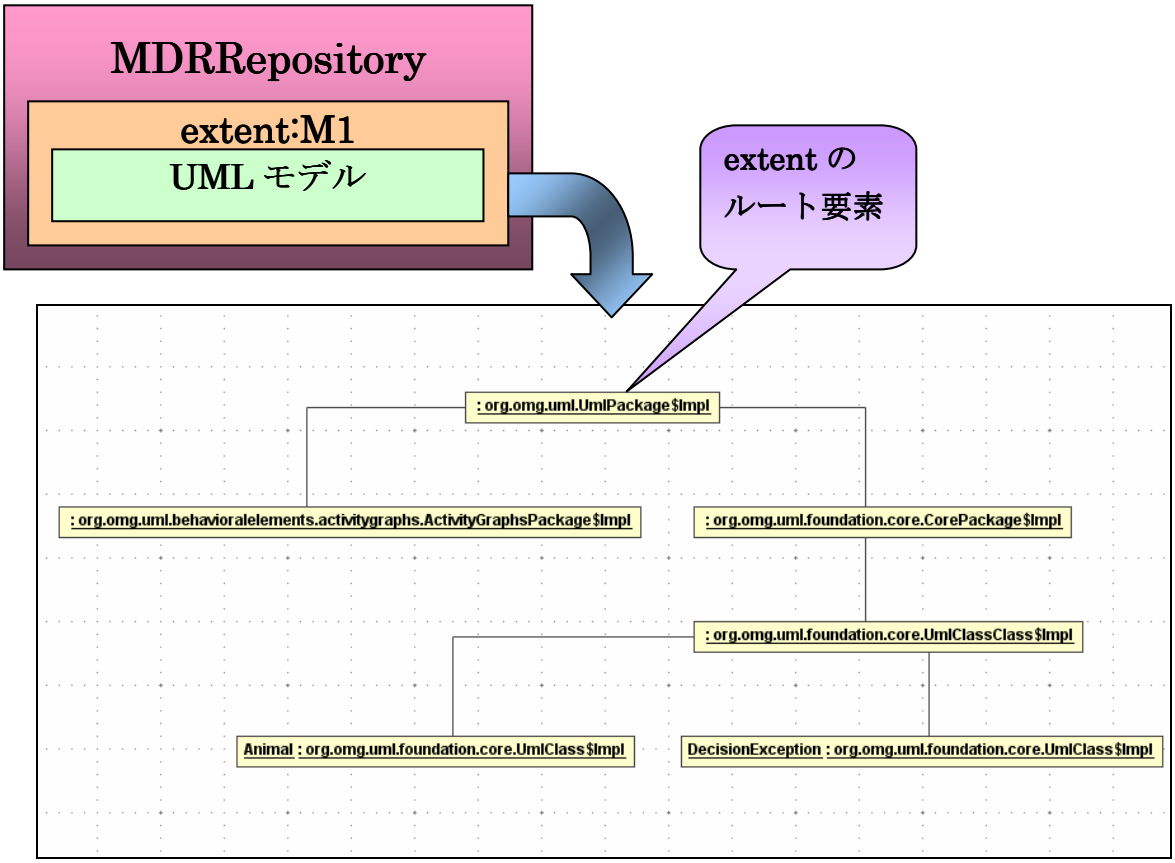

図２－３ UML モデルとして AnimalQuiz を読み込んだ場合のインスタンス図（一部）

２．２．２　　サンプルコード

２．１．２で読み込んだ UML モデルのクラス一覧とそのステレオタイプ、タグ付き値、フィールド、メソッド、関連を表示するサンプルコード（PrintModelClasses.java）を示す。サンプルコードを追うことで、メモリ上に展開されたモデルが Java オブジェクトとしてどのような構造になっているかを理解して欲しい。

また、このクラスは２．１．２の MOFRepository#getUmlPackage メソッドで受け取った UmlPackage をコンストラクタの引き数とする。

PrintModelClasses.java

```
 1:import java.util.Collection;
 2:import java.util.Iterator;
 3:import java.util.Vector;
 4:import org.apache.commons.collections.CollectionUtils;
 5:import org.apache.commons.collections.Predicate;
 6:import org.omg.uml.UmlPackage;
 7:import org.omg.uml.foundation.core.AssociationEnd;
 8:import org.omg.uml.foundation.core.Attribute;
 9:import org.omg.uml.foundation.core.CorePackage;
10:import org.omg.uml.foundation.core.EnumerationLiteral;
11:import org.omg.uml.foundation.core.GeneralizableElement;
12:import org.omg.uml.foundation.core.Generalization;
13:import org.omg.uml.foundation.core.Interface;
14:import org.omg.uml.foundation.core.ModelElement;
15:import org.omg.uml.foundation.core.Operation;
16:import org.omg.uml.foundation.core.Parameter;
17:import org.omg.uml.foundation.core.Stereotype;
18:import org.omg.uml.foundation.core.TaggedValue;
19:import org.omg.uml.foundation.core.UmlAssociation;
20:import org.omg.uml.foundation.core.UmlClass;
21:import org.omg.uml.foundation.datatypes.MultiplicityRange;
22:import org.omg.uml.foundation.datatypes.VisibilityKind;
23:import org.omg.uml.foundation.datatypes.VisibilityKindEnum;
24:
25:public class PrintModelClasses {
26:
27:     static final String[] moduleSearchPath = null;
28:     private UmlPackage umlPackage;
29:     private CorePackage corePackage;
30:     public PrintModelClasses(UmlPackage umlPackage){
31:          this.umlPackage = umlPackage;
32:          corePackage = umlPackage.getCore();
```

↑ 引き数の UmlPackage から CorePackage を取得する。この CorePackage に UML モデルのすべての情報が格納されている。CorePackage は１．３で解説した「**package object**」である。

```
33:     }
34:     public void print(){
35:          for (Iterator i = getUmlClasses().iterator(); i.hasNext();) {
```

↑ getUmlClasses()メソッドは、すべての UmlClass を取得するメソッドである。すべての UmlClass を取得し、Iterator で走査していく。UmlClass は１．３で説明した「**instance object**」である。

```
36:               UmlClass c = (UmlClass) i.next();
37:               System.out.println(getClassExp(c) + getExtentionExp(c) +
                                                          getTaggedValuesExp(c));
38:               for (Iterator j = getAttributes(c).iterator(); j.hasNext();) {
39:                    Attribute x = (Attribute) j.next();
40:                    System.out.println("     " + getAttributeExp(x) + getExtentionExp(x) +
                                                          getTaggedValuesExp(x));
41:               }
```

```
42:                 for (Iterator j = getOperations(c).iterator(); j.hasNext();) {
43:                     Operation x = (Operation) j.next();
44:                     System.out.println("     " + getOperationExp(x) + getExtentionExp(x) +
                                                                                    getTaggedValuesExp(x));
45:                 }
46:                 for (Iterator j = getAssociationEnds(c).iterator(); j.hasNext();) {
47:                     AssociationEnd x = getOtherAssociationEnd((AssociationEnd) j.next());
48:                     System.out.println("     " + getAssociationEndExp(x) +
                                                            getExtentionExp(x) + getTaggedValuesExp(x));
49:                 }
50:             }
51:         }
52:         private String getClassExp(UmlClass c) {
53:             StringBuffer b = new StringBuffer(getVisibilityExp(c));
54:             if (!c.isAbstract()) {
55:                 b.append(" class " + c.getName());
56:             } else {
57:                 b.append(" abstract class " + c.getName());
58:             }
59:             if (!c.isRoot()) {
60:                 if (c.getGeneralization().iterator().hasNext()) {
61:                     GeneralizableElement g = ((Generalization)
                                                        c.getGeneralization().iterator().next()).getParent();
62:                     if (!(g instanceof Interface)) {
63:                         b.append(" extends " + g.getName());
64:                     } else {
65:                         b.append(" implements " + g.getName());
66:                     }
67:                 }
68:             }
69:             return b.toString();
70:         }
71:         private String getAttributeExp(Attribute a) {
72:             return getVisibilityExp(a) + " " + a.getType().getName() + " " + a.getName();
73:         }
74:         private String getOperationExp(Operation m) {
75:             String r = "";
76:             StringBuffer b = new StringBuffer("( ");
77:             for (Iterator i = m.getParameter().iterator(); i.hasNext();) {
78:                 Parameter p = (Parameter) i.next();
79:                 if (i.hasNext()) {
80:                     b.append(p.getType().getName() + " " + p.getName() + ",");
81:                 } else {
82:                     r = p.getType().getName();
83:                     break;
84:                 }
85:             }
86:             if (b.charAt(b.length() - 1) == ',') {
87:                 b.deleteCharAt(b.length() - 1);
88:             }
89:             b.append(" )");
90:             return getVisibilityExp(m) + " " + r + " " + m.getName() + b.toString();
91:         }
```

```
92:     private String getAssociationEndExp(AssociationEnd a) {
93:         return getVisibilityExp(a) + " " + a.getName() + " -> " + getMultiplicityExp(a)
                                                                    + " " + a.getParticipant().getName();
94:     }
95:     private String getExtentionExp(ModelElement e) {
96:         StringBuffer b = new StringBuffer();
97:         for (Iterator i = e.getStereotype().iterator(); i.hasNext();) {
98:             Stereotype s = (Stereotype) i.next();
99:             b.append(" <<" + s.getName() + ">> ");
100:        }
101:        return b.toString();
102:    }
103:    private String getTaggedValuesExp(ModelElement e) {
104:        StringBuffer b = new StringBuffer();
105:        for (Iterator i = e.getTaggedValue().iterator(); i.hasNext();) {
106:            TaggedValue t = (TaggedValue) i.next();
107:            b.append(" " + t.getName() + "=");
108:
109:            if (t.getReferenceValue().size() == 0) { //タグ値が参照型ではない場合
110:                for (Iterator i2 = t.getDataValue().iterator(); i2.hasNext();) {
111:                    b.append(i2.next() + ",");
112:                }
113:            } else { //タグ値が参照型の場合
114:                for (Iterator i2 = t.getReferenceValue().iterator(); i2.hasNext();) {
115:                    b.append(((EnumerationLiteral) i2.next()).getName() + ",");
116:                }
117:            }
118:        }
119:        return b.toString();
120:    }
121:    private String getVisibilityExp(ModelElement e) throws IllegalArgumentException {
122:        VisibilityKind v = e.getVisibility();
123:        if (v == null) {
124:            return "public";
125:        } else if (v.equals(VisibilityKindEnum.VK_PUBLIC)) {
126:            return "public";
127:        } else if (v.equals(VisibilityKindEnum.VK_PRIVATE)) {
128:            return "private";
129:        } else if (v.equals(VisibilityKindEnum.VK_PROTECTED)) {
130:            return "protected";
131:        } else if (v.equals(VisibilityKindEnum.VK_PACKAGE)) {
132:            return "";
133:        } else {
134:            throw new IllegalArgumentException("getVisibilityExp:" + v);
135:        }
136:    }
137:    private String getMultiplicityExp(AssociationEnd e) {
138:        String range = "";
139:        if (e.getMultiplicity() != null) {
140:            Iterator i = e.getMultiplicity().getRange().iterator();
141:            if (i.hasNext()) {
142:                MultiplicityRange r = (MultiplicityRange) i.next();
143:                range = "[" + r.getLower() + "," + r.getUpper() + "]";
```

```
144:                    return range;
145:                }
146:            }
147:            return range;
148:        }
149:        private Collection getUmlClasses() {
150:            return corePackage.getUmlClass().refAllOfType();
151:        }
```

↑ corePackage.getUmlClass() は UmlClassClass を返す。UMLClassClass は１． ３で説明した「**class proxy object**」である。UmlClassClass は UmlClass のインスタンスの“入れ物”と考えれば良い。 データベースに例えるならば、 UmlClassClass は UmlClass のインスタンスを蓄える表である。また、UmlClassClass は UmlClass のクラス属性(インスタンスではなくクラスに属する)と考えることができるので、UmlClassClass と命名されている。 例えば、 RelationshipClass, StereotypeClass なども入れ物である。 UmlClassClass, RelationshipClass, StereotypeClass などがデータベースの表なら、一般的な検索方法が必要である。 これらクラスは javax.jmi.reflect.RefClass を継承している。このクラスには 自身が生成することが可能な総てのインスタンスを検索する refAllOfType() が備わっている。インスタンスの生成は個別クラスに備わっている。例えば、UmlClassClass には createUmlClass()がある。

```
152:        private Collection getInterfaces() {
153:            return corePackage.getInterface().refAllOfType();
154:        }
```

↑ corePackage.getInterface()は InterfaceClass を返す。InterfaceClass は１． ３で説明した「**class proxy object**」である。refAllOfType()は InterfaceClass 内のすべての Interface を返す。Interface は１． ３で説明した「**instance object**」である。

```
155:        private Collection getAssociations() {
156:            return corePackage.getUmlAssociation().refAllOfType();
157:        }
```

↑ corePackage.getUmlAssociation()は UmlAssociationClass を返す。UmlAssociationClass は１．３で説明した「**class proxy object**」である。refAllOfType()は UmlAssociationClass 内のすべての UmlAssociation を返す。UmlAssociation は１． ３で説明した「**instance object**」である。

```
158:
159:
160:
161:        private Collection getAttributes(final UmlClass c) {
162:            return CollectionUtils.select(c.getFeature(), new Predicate() {
163:                public boolean evaluate(Object x) {
164:                    return x instanceof Attribute;
165:                }
166:            });
167:        }
168:        private Collection getOperations(final UmlClass c) {
169:            return CollectionUtils.select(c.getFeature(), new Predicate() {
170:                public boolean evaluate(Object x) {
```

```
171:                    return x instanceof Operation;
172:                }
173:            });
174:    }
175:    private Collection getAssociations(final UmlClass c) {
176:            /*
177:             * MOF Core package の Model 図 を参照されたし
178:             */
179:            return CollectionUtils.select(getAssociations(), new Predicate() {
180:                public boolean evaluate(Object x) {
181:                    for (Iterator i = ((UmlAssociation) x).getConnection().iterator(); i.hasNext();) {
182:                        AssociationEnd e = (AssociationEnd) i.next();
183:                        if (e.getParticipant() == c)
184:                            return true;
185:                    }
186:                    return false;
187:                }
188:            });
189:    }
190:    private Collection getAssociationEnds(final UmlClass c) {
191:            /*
192:             * MOF Core package の Model 図 を参照されたし
193:             */
194:            Vector ends = new Vector();
195:            for (Iterator i = getAssociations(c).iterator(); i.hasNext();) {
196:                for (Iterator j = ((UmlAssociation) i.next()).getConnection().iterator();
                                                                                    j.hasNext();) {
197:                    AssociationEnd e = (AssociationEnd) j.next();
198:                    if (e.getParticipant() == c) {
199:                        ends.add(e);
200:                    }
201:                }
202:            }
203:            return ends;
204:    }
205:    private AssociationEnd getOtherAssociationEnd(AssociationEnd e) {
206:            /*
207:             * MOF Core package の Model 図 を参照されたし
208:             */
209:            for (Iterator i = e.getAssociation().getConnection().iterator(); i.hasNext();) {
210:                AssociationEnd x = (AssociationEnd) i.next();
211:                if (e != x)
212:                    return x;
213:            }
214:            return null; // should not come here
215:    }
216:}
```

読み込む UML モデルを AndroMDA のサンプルアプリケーション「AnimalQuiz」とし、このサンプルを実行すると、以下を出力する。

出力結果

```
public class GuessController
    public void getFirstQuestion( String question )
    public String nextDecisionItemAvailable( String question )
    public void rememberAnimal( String animal )
    public void rememberQuestion( String question )
    public boolean lastAnswerWasYes(  )
    public void rememberPositiveAnswer(  )
    public void rememberNegativeAnswer(  )
    public void initializeSession(  )
public class GuessSessionState <<FrontEndSessionObject>>
    public VODecisionItem lastDecisionItem
    public String lastAnimalName
    public String lastPrompt
    public String lastAnswerFromUser
public class Question extends DecisionItem <<Entity>>
    public String promptString
    public String getPrompt(  )
public abstract class DecisionItem <<Entity>>  @andromda.persistence.table=DecisionItem,
    public boolean rootItem
    public String getPrompt(  )
    public DecisionItem findRoot(  )
    public yesSuccessor -> [0,1] DecisionItem
    public null ->  DecisionItem
    public noSuccessor -> [0,1] DecisionItem
    public null ->  DecisionItem
public class Animal extends DecisionItem <<Entity>>
    public String name
    public String getPrompt(  )
public class DecisionService <<Service>>  <<WebService>>  @andromda.ejb.viewType=both,
    public VODecisionItem getFirstQuestion(  )
    public VODecisionItem getNextQuestion( Long itemId )
    public void addNewAnimalWithQuestion( String animalName,String promptForYes,Long
idOfLastNoDecision )
public class DecisionException <<ApplicationException>>
public class VODecisionItem <<ValueObject>>
    public Long id
    public String prompt
    public Long idYesItem
    public Long idNoItem
```

２．３　AndroMDA Metafacade

２．３．１　　概要

AndroMDA Metafacade は、NetBeans MDR のラッパークラスである。NetBeans MDR を直接扱うよりも、AndroMDA Metafacade を通じてモデルを扱うほうが便利である。AndroMDA Metafacade のクラス間の構造は、NetBeans MDR のそれとほとんど同じ構造となっている。しかし、AndroMDA Metafacade のみが持つユーティリティのようなメソッドが用意されている。これらのメソッドは MDA ツールとしてコードを生成するための利便性を向上させる為に用意されている。

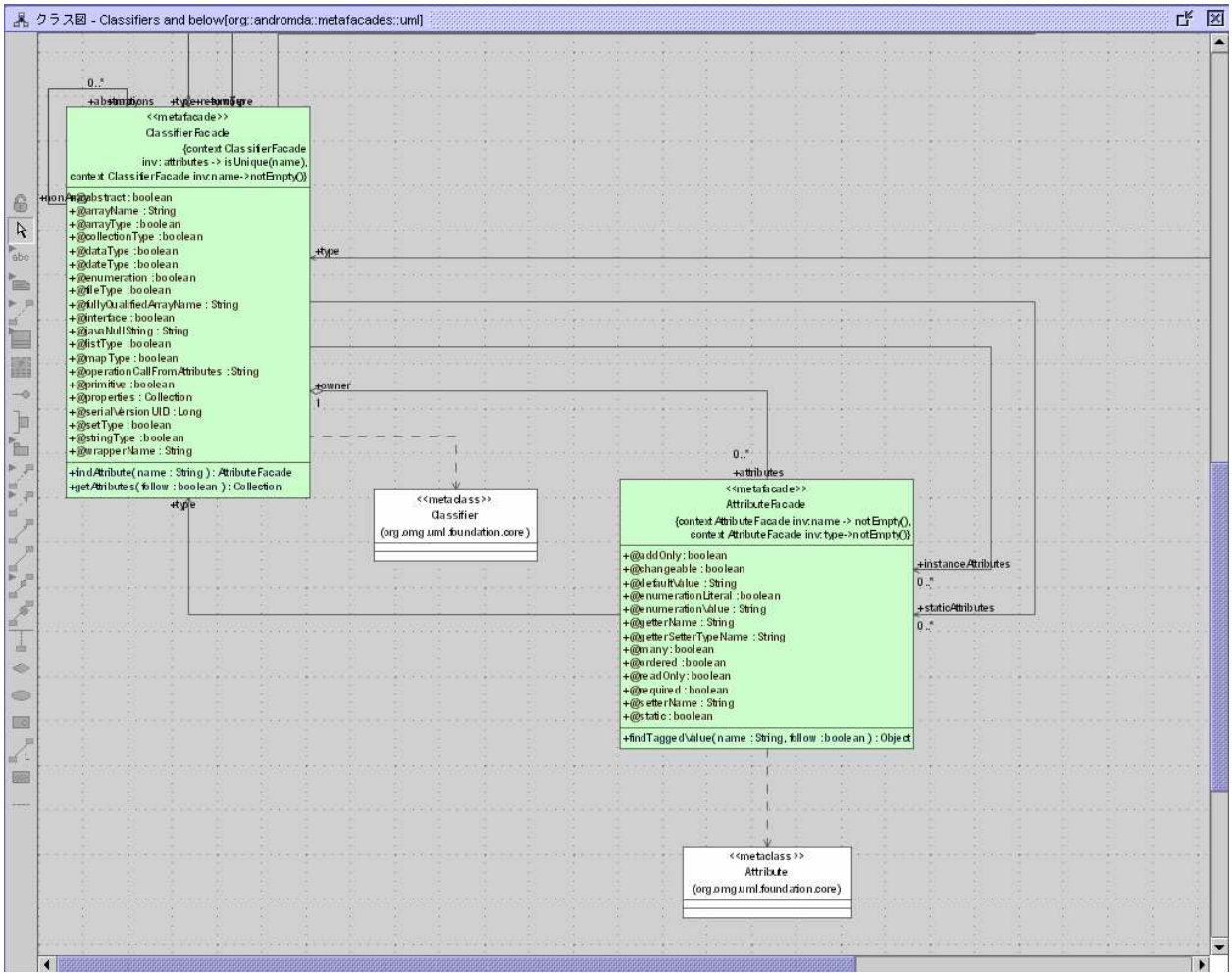

図２－４ AndroMDA Metafacade

また、各開発者が独自に AndroMDA Metafacade を定義することができる。定義の方法については３章で詳しく説明しているので、そちらを参照していただきたい。

２．３．２　　サンプルコード

AndroMDA Metafacade を使って UML モデルを読み込むサンプルコード（AndroMDAMetafacadeFactory.java）を示す。

AndroMDAMetafacadeFactory.java

```
 1:import java.net.URL;
 2:import org.andromda.core.common.AndroMDALogger;
 3:import org.andromda.core.common.ComponentContainer;
 4:import org.andromda.core.common.Namespace;
 5:import org.andromda.core.common.Namespaces;
 6:import org.andromda.core.common.Property;
 7:import org.andromda.core.common.XmlObjectFactory;
 8:import org.andromda.core.metafacade.MetafacadeFactory;
 9:import org.andromda.core.metafacade.ModelAccessFacade;
10:import org.andromda.core.repository.RepositoryFacade;
11:import org.apache.log4j.Logger;
12:
13:public class AndroMDAMetafacadeFactory {
14:
15:  private final String[] moduleSearchPath = null;
16:  static {
17:              Logger logger = Logger.getLogger(AndroMDAMetafacadeFactory.class);
18:              AndroMDALogger.configure();
19:  }
20:
21:  private ModelAccessFacade facade;
22:
23:  private RepositoryFacade repository;
24:
25:  public AndroMDAMetafacadeFactory(String url) throws Exception {
```

↑ 引き数として XMI ファイル（UML モデル）のパスを指定する。

```
26:
27:              XmlObjectFactory.setDefaultValidating(false);
28:              MetafacadeFactory.getInstance().initialize();
29:
30:              repository = (RepositoryFacade)ComponentContainer.instance().
                                                        findComponent(RepositoryFacade.class);
```

↑ RepositoryFacade のインスタンスを取得する。

```
31:              repository.open();
32:              repository.readModel(new URL(url), null);
```

↑ repository にモデルを展開する。ここで、UML モデルの読み込みは完了する。

```
33:
34:              MetafacadeFactory factory = MetafacadeFactory.getInstance();
```

↑ MetafacadeFactory のインスタンスを取得する。

```
35:
36:              { /* Set defalut Name Space */
37:                        String[] names = { "languageMappingsUri", "wrapperMappingsUri",
"maxSqlNameLength", "jdbcMappingsUri", "sqlMappingsUri" };
38:                        String[] value = { "file:xml/JavaMappings.xml",
"file:xml/JavaWrapperMappings.xml", null, null, null };
```

↑ namespace のためのパラメータを定義する。

```
39:                        Namespace namespace = new Namespace();
40:                        namespace.setName(Namespaces.DEFAULT);
```

```
41:                         for (int i = 0; i < names.length; i++) {
42:                                 Property property = new Property();
43:                                 property.setName(names[i]);
44:                                 if (value[i] == null) {
45:                                         property.setIgnore(true);
46:                                 } else {
47:                                         property.setValue(value[i]);
48:                                 }
49:                                 namespace.addProperty(property);
```

↑ ３７行目、３８行目で設定したパラメータを namespace に追加する。この namespace の情報に従って AndroMDA は AndroMDA Metafacade と repository 内に展開された UML モデルとのマッピングを行う。

```
50:                         }
51:                         Namespaces.instance().addNamespace(namespace);
52:             }
53:             factory.setActiveNamespace(Namespaces.DEFAULT);
54:             facade = repository.getModel();
```

↑ repository から UML モデルを取得する。

```
55:             factory.setModel(facade);
```

↑ 取得した UML モデルを MetafacadeFactory に設定する。

```
56: }
57:
58: /**
59:  * @see java.lang.Object#finalize()
60:  */
61: protected void finalize() throws Throwable {
62:             close();
63:             super.finalize();
64: }
65:
66: /**
67:  * リポジトリをクローズする。
68:  * クロース後は ModelAccessFacade にアクセスできない
69:  */
70: public void close() {
71:             repository.close();
72: }
73:
74: /**
75:  * ModelAccessFacade を取得
76:  * @return
77:  */
78: public ModelAccessFacade getModelAccessFacade() {
79:             return facade;
```

↑ AndroMDA Metafacade の１つである ModelAccessFacade をリターンする。ModelAccessFacade は、UML モデルのルートとなる AndroMDA Metafacade である。

```
80: }
81:}
```

２．４　AndroMDA Metafacade の利用

２．４．１　　概要

２．３では、AndroMDA での UML モデルを読み込む方法について解説した。この章では、読み込んだ UML モデルを AndroMDA Metafacade で操作する方法について解説する。

２．４．２　　サンプルコード

２．３．２で読み込んだ UML モデルのクラス一覧とそのステレオタイプ、タグ付き値、フィールド、メソッド、関連を表示するサンプルコード（**PrintModelClasses.java**）を示す。基本的に２．２．２のサンプルコードと同じ処理をする。

また、このクラスは２．３．２の AndroMDAMetafacadeFactory#getModelAccessFacade()メソッドの ModelAccessFacade を使用する。

```
 1:import java.util.ArrayList;
 2:import java.util.Collection;
 3:import java.util.Iterator;
 4:import java.util.List;
 5:
 6:import org.andromda.core.metafacade.ModelAccessFacade;
 7:import org.andromda.metafacades.uml.ActorFacade;
 8:import org.andromda.metafacades.uml.AttributeFacade;
 9:import org.andromda.metafacades.uml.ClassifierFacade;
10:import org.andromda.metafacades.uml.EnumerationLiteralFacade;
11:import org.andromda.metafacades.uml.OperationFacade;
12:import org.andromda.metafacades.uml.TaggedValueFacade;
13:import org.andromda.metafacades.uml.UseCaseFacade;
14:
15:public class PrintModelClasses {
16:
17:  public static void print(ModelAccessFacade modelAccessFacade){
```

↑　２．３．２の AndroMDAMetafacadeFactory#getModelAccessFacade()メソッドの ModelAccessFacade を引数として渡す。

```
18:             for(Iterator i = getClassFacade(modelAccessFacade).iterator(); i.hasNext() ; ){
19:                     ClassifierFacade c = (ClassifierFacade)i.next();
20:                     System.out.println(getClassExp(c) + "    " +
          getExtentionExp(c.getStereotypeNames()) + " " + getTaggedValuesExp(c.getTaggedValues()));
```

↑　クラスの表示

```
21:                     for(Iterator j = c.getAttributes().iterator(); j.hasNext(); ){
22:                             AttributeFacade a = (AttributeFacade)j.next();
23:                             System.out.println("    " + getAttributeExp(a) +
          getExtentionExp(a.getStereotypeNames()) + " " + getTaggedValuesExp(a.getTaggedValues()));
```

↑　属性の表示

```
24:                     }
25:                     for(Iterator j = c.getOperations().iterator(); j.hasNext(); ){
26:                             OperationFacade a = (OperationFacade)j.next();
27:                             System.out.println("    " + getOperationExp(a) +
          getExtentionExp(a.getStereotypeNames()) + " " + getTaggedValuesExp(a.getTaggedValues()));
```

↑ 操作の表示

```
28:                         }
29:             }
30: }
31:
32: private static Collection getClassFacade(ModelAccessFacade facade) {

33:             List classFacades = new ArrayList();
34:             for (Iterator i = facade.getModelElements().iterator(); i.hasNext();) {
35:                         Object x = i.next();
36:                         if (x instanceof ClassifierFacade && !((ClassifierFacade) x).isDataType()
                                        && !(x instanceof ActorFacade) && !(x instanceof UseCaseFacade)) {
37:                                     classFacades.add(x);
38:                         }
39:             }
40:             return classFacades;
41: }
```

↑ UML モデルクラスの ClassfierFacade をすべて取得

```
42:
43: private static String getClassExp(ClassifierFacade c){
44:             StringBuffer b = new StringBuffer();
45:             b.append(c.getVisibility() + " ");
46:           if (!c.isAbstract()) {
47:                b.append("class " + c.getName());
48:           } else {
49:                b.append("abstract class " + c.getName());
50:           }
51:           if(c.getGeneralization() != null){
52:             b.append("extends " + c.getGeneralization().getFullyQualifiedName());
53:           }
54:           String str = c.getGeneralizationList();
55:           return b.toString();
56: }
57:
58: private static String getAttributeExp(AttributeFacade a){
59:             StringBuffer b = new StringBuffer();
60:             b.append(a.getVisibility() + " " + a.getType().getFullyQualifiedName() + " " +
                                                                                    a.getName());
61:             return b.toString();
62: }
63:
64: private static String getOperationExp(OperationFacade o){
65:             StringBuffer b = new StringBuffer();
66:             b.append(o.getVisibility() + " ");
67:             if(o.isAbstract()){
68:                         b.append("abstract ");
69:             }
70:             b.append(o.getReturnType().getFullyQualifiedName() + " " + o.getName() + "( " +
                                                                    o.getTypedArgumentList() + " )");
71:             return b.toString();
72: }
73:
```

```
74:  private static String getExtentionExp(Collection stereotypes){
75:          StringBuffer b = new StringBuffer();
76:          for(Iterator i = stereotypes.iterator() ;i.hasNext();){
77:                  b.append("<<" + i.next() + ">>");
78:                  if(i.hasNext()){
79:                          b.append(",");
80:                  }
81:          }
82:          return b.toString();
83:  }
84:
85:  private static String getTaggedValuesExp(Collection taggedValues){
86:          StringBuffer b = new StringBuffer();
87:          for(Iterator i = taggedValues.iterator() ;i.hasNext();){
88:                  TaggedValueFacade t = (TaggedValueFacade)i.next();
89:                  b.append(t.getName() + "=");
90:                  for(Iterator j = t.getValues().iterator();j.hasNext();){
91:                          Object o = j.next();
92:                          if(o instanceof EnumerationLiteralFacade){
93:                                  EnumerationLiteralFacade e =
                                                                  (EnumerationLiteralFacade)o;
94:                                  b.append(e.getValue());
95:                          }else{
96:                                  b.append(o);
97:                          }
98:                          if(j.hasNext()){
99:                                  b.append(",");
100:                         }
101:                 }
102:         }
103:         return b.toString();
104: }
105:}
```

読み込む UML モデルを AndroMDA のサンプルアプリケーション「AnimalQuiz」とし、このサンプルを実行すると、以下を出力する。

```
private class GuessController
    public void getFirstQuestion( java.lang.String question )
    public java.lang.String nextDecisionItemAvailable( java.lang.String question )
    public void rememberAnimal( java.lang.String animal )
    public void rememberQuestion( java.lang.String question )
    public boolean lastAnswerWasYes(  )
    public void rememberPositiveAnswer(  )
    public void rememberNegativeAnswer(  )
    public void initializeSession(  )
private class GuessSessionState     <<FrontEndSessionObject>>
    public org.andromda.samples.animalquiz.decisiontree.VODecisionItem lastDecisionItem
    public java.lang.String lastAnimalName
    public java.lang.String lastPrompt
    public java.lang.String lastAnswerFromUser
private class Questionextends org.andromda.samples.animalquiz.decisiontree.DecisionItem     <<Entity>>
```

```
    public java.lang.String promptString
    public java.lang.String getPrompt(  )
private abstract class DecisionItem    <<Entity>> @andromda.persistence.table=DecisionItem
    public boolean rootItem
    public java.lang.Long id<<Identifier>>
    public abstract java.lang.String getPrompt(  )
    public org.andromda.samples.animalquiz.decisiontree.DecisionItem findRoot(  )
private class Animalextends org.andromda.samples.animalquiz.decisiontree.DecisionItem    <<Entity>>
    public java.lang.String name
    public java.lang.String getPrompt(  )
private class DecisionService    <<Service>>,<<WebService>> @andromda.ejb.viewType=,both
    public org.andromda.samples.animalquiz.decisiontree.VODecisionItem getFirstQuestion(  )
    public  org.andromda.samples.animalquiz.decisiontree.VODecisionItem  getNextQuestion(  java.lang.Long
itemId )
    public  void  addNewAnimalWithQuestion(  java.lang.String  animalName,  java.lang.String  promptForYes,
java.lang.Long idOfLastNoDecision )
private class DecisionException    <<ApplicationException>>
private class VODecisionItem    <<ValueObject>>
    public java.lang.Long id
    public java.lang.String prompt
    public java.lang.Long idYesItem
    public java.lang.Long idNoItem
```

２．５　Velocity

２．５．１　　概要

Velocity は、Apache Jakarta プロジェクトのサブプロジェクトの１つで、Java ベースのテンプレートエンジンである。テンプレートエンジンとは**雛形となるテンプレート（vsl、vm ファイル）**と **Java のオブジェクト（Velocity Context）**を組み合わせて、動的なテキスト（Web ページやメール等）を作成するものである。

図で表すと次のようなイメージになる。Velocity Context には、key 名と value オブジェクトを設定する。VM ファイルには$key 名が記述されており、統合（マージ）すると、$key 名が value オブジェクトで置換される。もし、value オブジェクト内のパラメータでマージしたい場合は、例えば次のようになる。

```
$object1.param1
```

これは、Velocity Context の key 名が object1 のオブジェクトを取得し、そのオブジェクトの getParam1()というメソッドをコールし、その戻り値を値として VM ファイルにマージする。

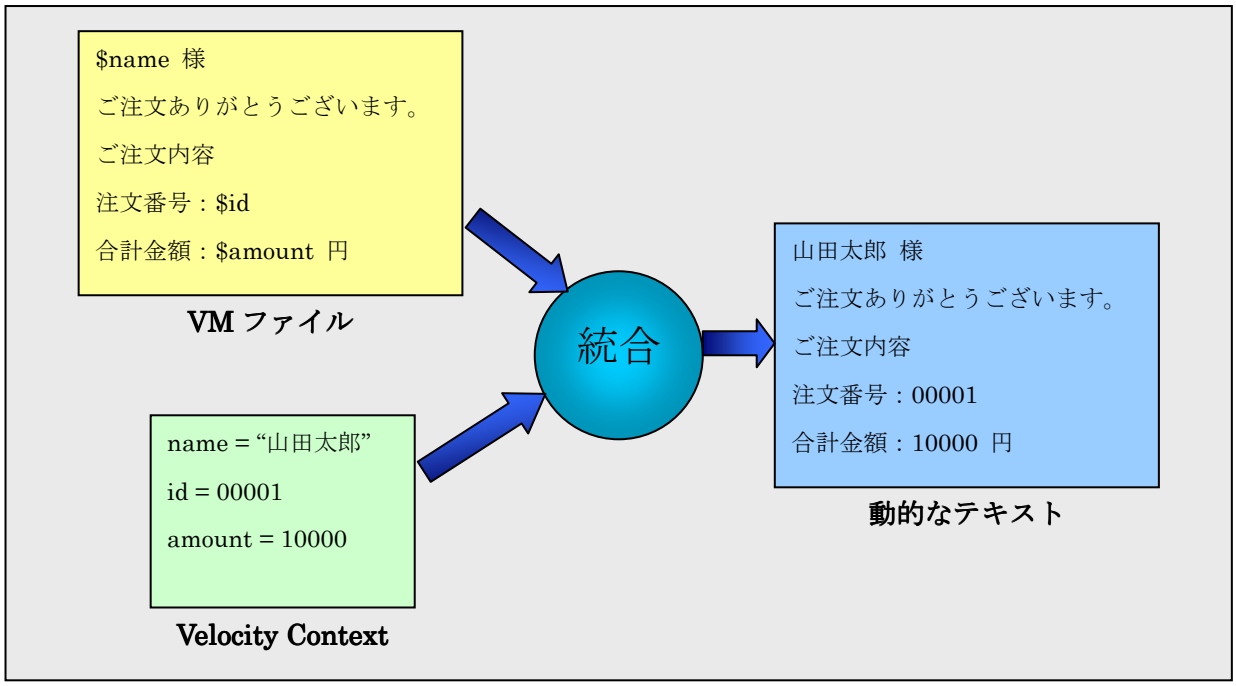

図２－５ Velocity でのファイル生成

AndroMDA では、ソースコード、DD、その他ファイルを MDA として生成するために Velocity を使っている。

２．５．２　　サンプルコード

２．３で読み込んだ AndroMDA Metafacade と Velocity を使い、Java クラスファイルを生成するサンプルコードを示す。このサンプルコードは、<<Entity>>クラスを POJO の Java クラスファイルとして生成するものである。Java クラスファイル生成のイメージを示す。

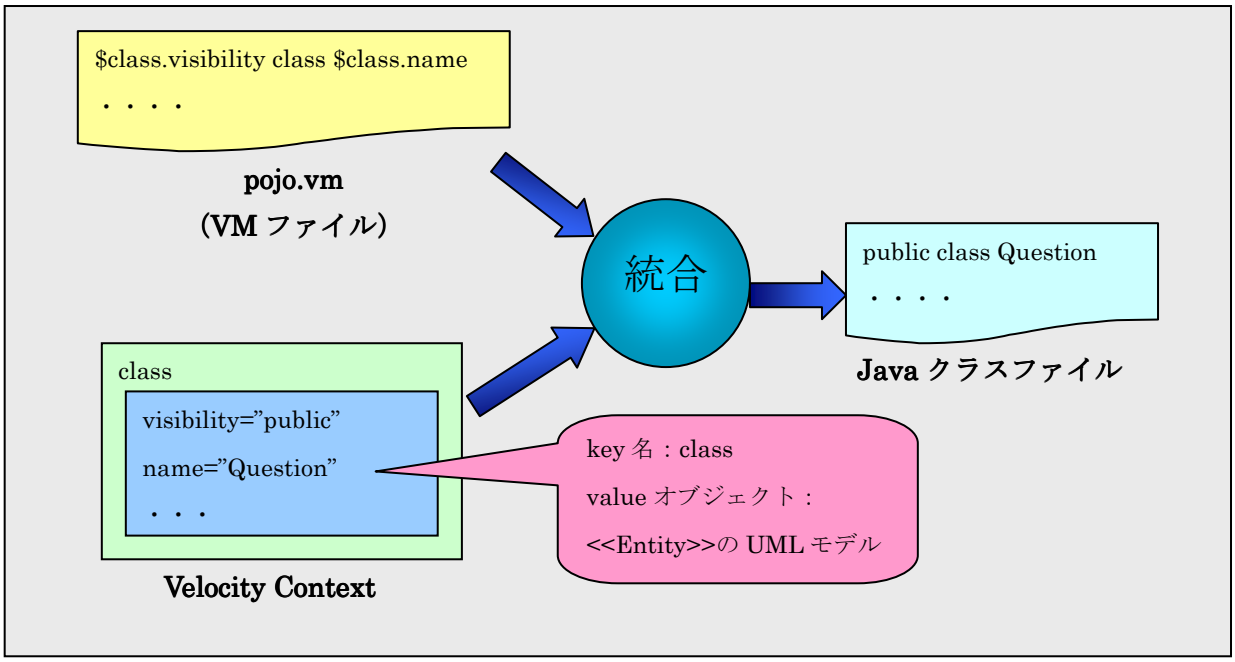

図 2 - 6 POJO 生成

template/pojo.vm

```
 1:#if(!($class.packageName == ""))
 2:package $class.packageName;
 3:#end
 4:/**
 5: * Auto Generated. Do not modify!!
 6: */
 7:
 8:#if(!$class.generalization)
 9:$class.visibility class $class.name {
10:#else
11:$class.visibility class $class.name extends $class.generalization.fullyQualifiedName {
12:#end
13:
14:#foreach( $attribute in $class.attributes )
15:     private $attribute.getterSetterTypeName $attribute.name;
16:
17:     $attribute.visibility $attribute.getterSetterTypeName ${attribute.getterName}(){
18:         return this.$attribute.name;
19:     }
20:
21:     $attribute.visibility void ${attribute.setterName}
                                        ($attribute.getterSetterTypeName $attribute.name){
22:         this.$attribute.name = $attribute.name;
23:     }
24:#end
25:}
```

PojoFactory.java

```
 1:import java.io.File;
 2:import java.io.FileWriter;
 3:import java.io.Writer;
 4:import java.util.ArrayList;
 5:import java.util.Collection;
 6:import java.util.Iterator;
 7:import java.util.List;
 8:import org.andromda.core.metafacade.ModelAccessFacade;
 9:import org.andromda.metafacades.uml.ClassifierFacade;
10:import org.apache.velocity.Template;
11:import org.apache.velocity.VelocityContext;
12:import org.apache.velocity.app.Velocity;
13:import org.apache.velocity.context.Context;
14:
15:public class PojoFactory {
16:
17:  private final String ENTITY = "Entity";
18:
19:  private final String GENERATED_CODE_DIR = "../generatedCode/";
20:
21:  /**
22:   * ModelAccessFacade をもとに Pojo クラスを作成
23:   * @param facade
24:   * @throws Exception
25:   */
26:  public void createPojoClass(ModelAccessFacade facade) throws Exception {
```

↑ このメソッドは、受け取った UML モデルから、すべての<<Entity>>クラスを探す。そして、それを元に Pojo クラスファイルを生成するメソッドである。引き数として ModelAccessFacade を指定する。２．３．２の ModelAccessFacade ををパラメータとして渡す。

```
27:            //テンプレート取得
28:            Velocity.init();
29:            Template template = Velocity.getTemplate("templates/pojo.vm");
```

↑ Velocity の Template インスタンスを取得する。テンプレートファイルとして「pojo.vm」を使用する。

```
30:
31:            for (Iterator i = getEntityFacades(facade).iterator(); i.hasNext();) {
```

↑ すべての<<Entity>>クラスを走査する。

```
32:                    ClassifierFacade c = (ClassifierFacade) i.next();
33:                    String ClassName = c.getName();
34:                    String packagePath = c.getPackagePath();
35:
36:                    Context context = new VelocityContext();
```

↑ Velocity Context を作成する。

```
37:                    context.put("class", c);
```

↑ 作成した Velocity Context に key 名=「class」、value オブジェクト=「<<Entity>>クラスモデルインスタンス」として設定する。

```
38:
39:                         File file = new File(GENERATED_CODE_DIR + packagePath);
40:                         file.mkdirs();
41:
42:                         Writer writer = new FileWriter(GENERATED_CODE_DIR + packagePath +
"/" + ClassName + ".java");
```

↑ ファイル名の設定、コード生成用のディレクトリ＋パッケージパス＋クラス名＋”.java”となる。

```
43:                         template.merge(context, writer);
```

↑ Template と Velcity Context をマージし、ファイルに書き込む。

```
44:                         writer.flush();
45:                         writer.close();
46:             }
47: }
48:
49: /**
50:  * <<Entity>>クラスの Facade を取得
51:  * @param facade
52:  * @return
53:  */
54: private Collection getEntityFacades(ModelAccessFacade facade) {
```

↑ このメソッドは、モデル内のすべてのクラスを調べ、すべての<<Entity>>クラスをリターンするメソッドである。

```
55:             List entities = new ArrayList();
56:             for (Iterator i = facade.getModelElements().iterator(); i.hasNext();) {
```

↑ すべてのクラスを走査する。

```
57:                         Object x = i.next();
58:                         if (x instanceof ClassifierFacade && !((ClassifierFacade) x).isDataType()) {
```

↑ モデルが ClassifierFacade でなおかつ、DataType でない（つまり唯の）クラスかどうかを調べる

```
59:                                 ClassifierFacade c = (ClassifierFacade) x;
60:                                 if (c.getStereotypeNames().contains(ENTITY)) {
61:                                         entities.add(c);
```

↑ ステレオタイプに Entity が存在するかどうかを判定し、Entity があれば List に追加する。

```
62:                                 }
63:                         }
64:             }
65:             return entities;
66: }
67:}
```

図 2-7 にあるようなモデルを読み込ませた場合に出力される Java クラスファイルを示す。

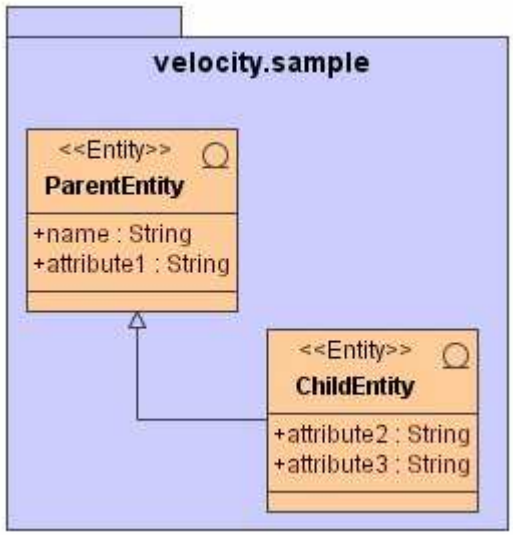

図 2－7 読み込ませるモデル

ParentEntity.java

```
 1: package velocity.sample;
 2:
 3: /**
 4:    * Auto Generated. Do not modify!!
 5:    */
 6:
 7: public class ParentEntity {
 8:
 9:        private String name;
10:
11:        public String getName(){
12:              return this.name;
13:        }
14:
15:        public void setName(String name){
16:              this.name = name;
17:        }
18:        private String attribute1;
19:
20:        public String getAttribute1(){
21:              return this.attribute1;
22:        }
23:
24:        public void setAttribute1(String attribute1){
25:              this.attribute1 = attribute1;
26:        }
27: }
```

ChildEntity.java

```
 1: package velocity.sample;
 2:
 3: /**
 4:  * Auto Generated. Do not modify!!
 5:  */
 6:
 7: public class ChildEntity extends ParentEntity {
 8:
 9:     private String attribute2;
10:
11:     public String getAttribute2 (){
12:         return this.attribute2;
13:     }
14:
15:     public void setAttribute2(String attribute2){
16:         this.attribute2 = attribute2;
17:     }
18:     private String attribute3;
19:
20:     public String getAttribute3(){
21:         return this.attribute3;
22:     }
23:
24:     public void setAttribute3(String attribute3){
25:         this.attribute3 = attribute3;
26:     }
27: }
```

3 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。

3．1 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章では、この「andromda-pojo」フォルダを「ProjectRoot」とする。

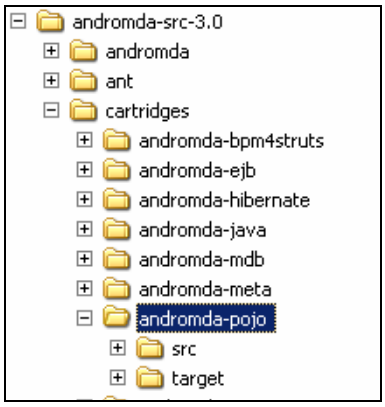

図 3－1 カートリッジの配置フォルダ

・ 「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</issu
eTrackingUrl>
    <developers>
        <developer>
            <name>Koji Iida</name>
            <id>iida</id>
            <email></email>
            <roles>
                <role>Developer</role>
            </roles>
      <!-- 省略 -->
```

図 3 - 2 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（３．７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　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３．３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}!/MetaCartridgeTestModel.xml
```

図 3－3 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、MagicDraw 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。

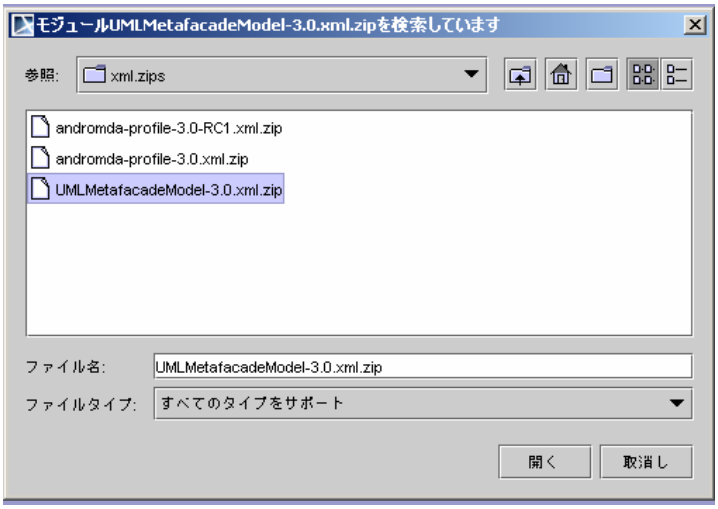

図 3－4 UMLMetafacadeModel-3.0.xml.zip の指定

ここまで完了すると、MagicDraw 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。

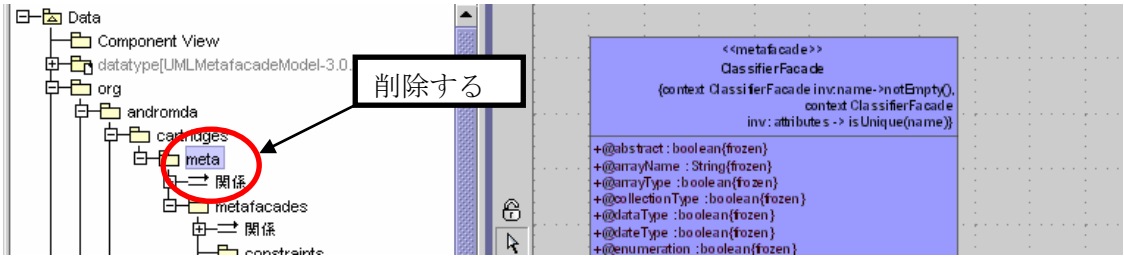

図 3－5 meta パッケージの削除

・ 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）

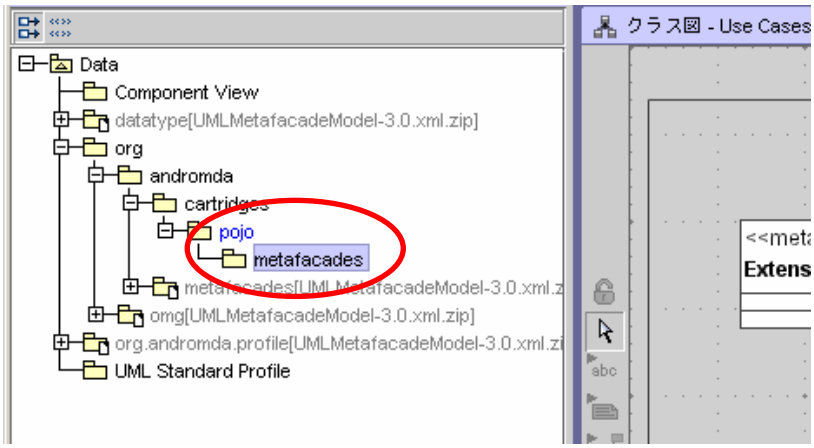

図 3－6 pojo.metafacade パッケージの作成

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。

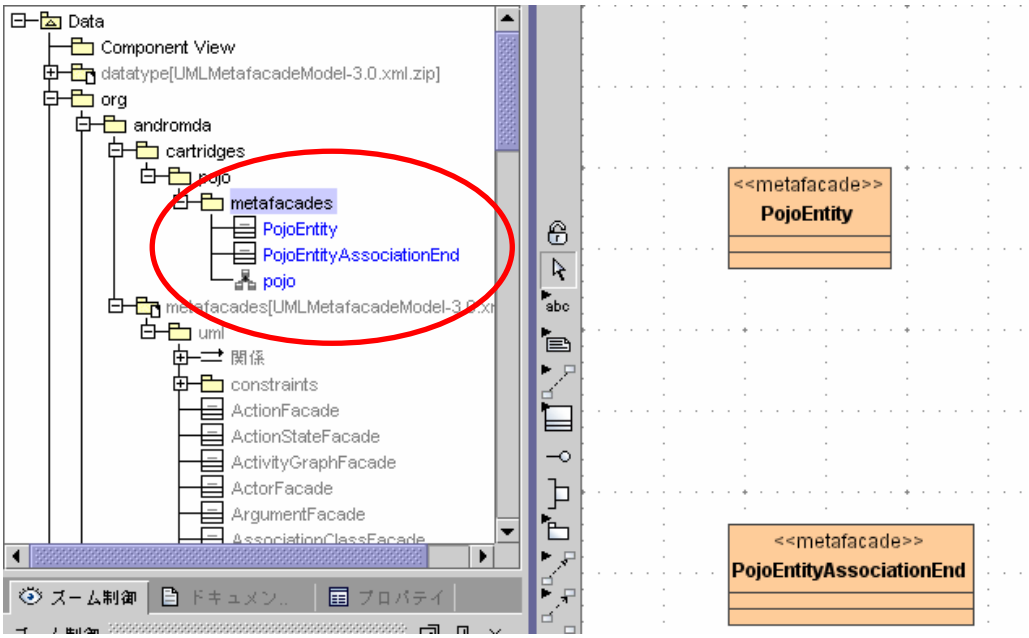

図 3－7 PojoEntity と PojoEntityAssociationEnd の作成

・ 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。

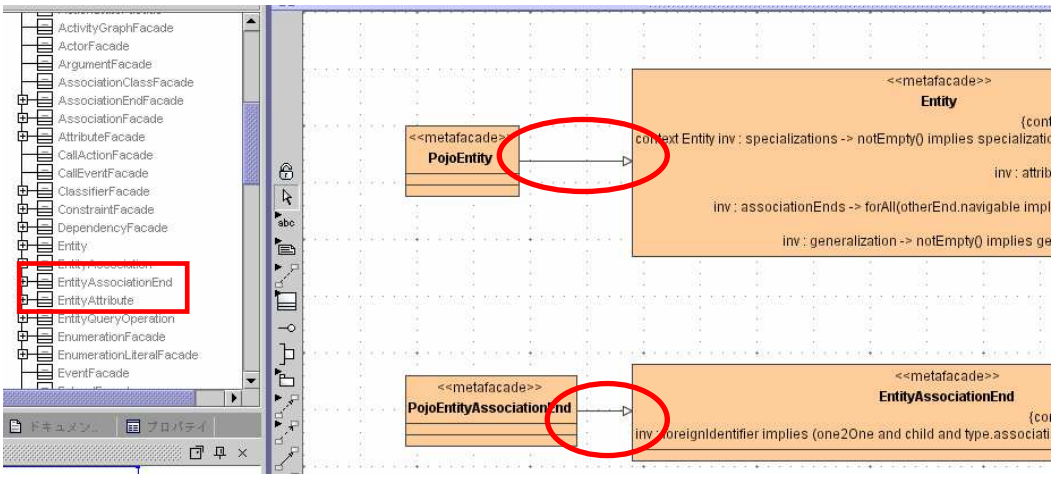

図 3－8 継承関係の設定

・ 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。

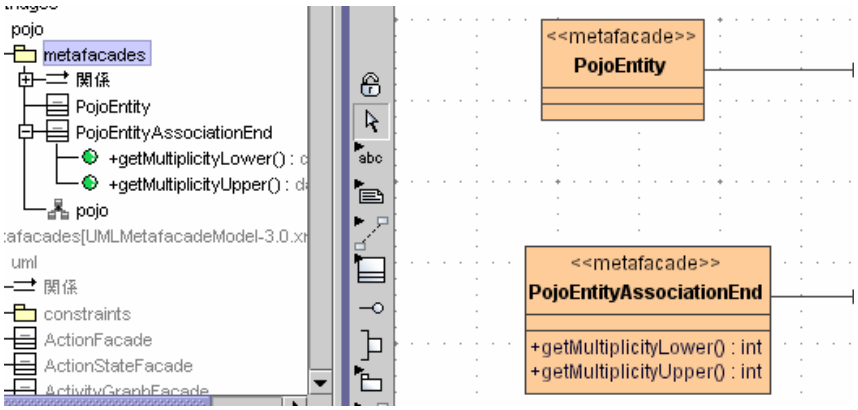

図 3－9 メソッドの記述

・ 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。

３．３　MetafacadeImpl クラスのコードの実装

この章では、３．２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 3－10 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]
```

３．４　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。

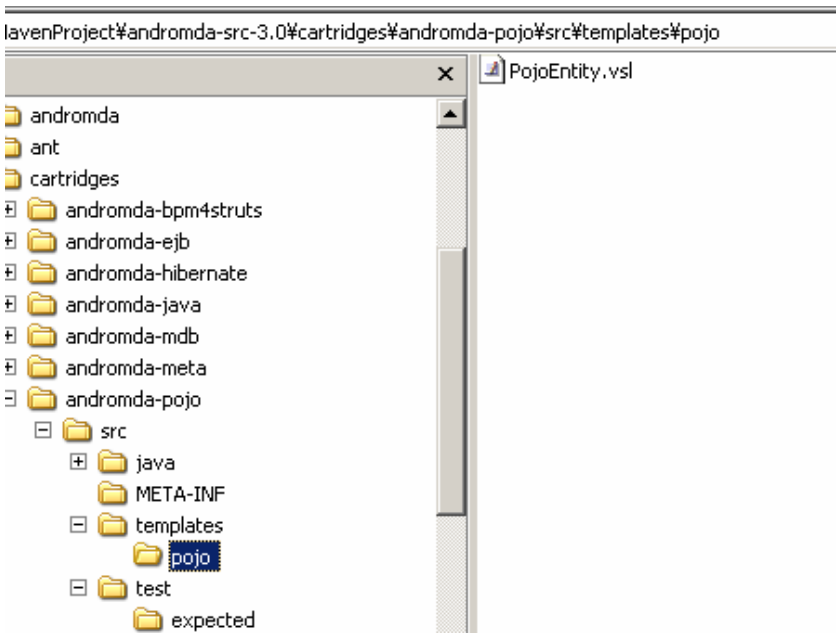

図 3－11 PojoEntity.vsl の配置位置

・ 「**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
{
/* ---------------------------------------------------------------------- */
##   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
$operation.exceptionList {
#foreach ($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 - 12 PojoEntity.vsl

3．5　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

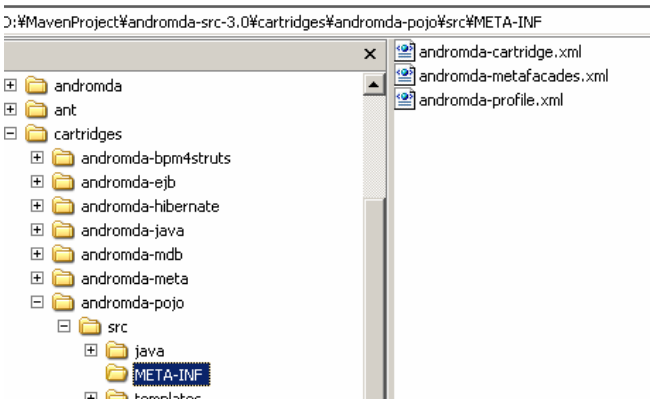

図 3 - 13 設定ファイルの位置

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

図 3 - 14 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.PojoEntityAssociationEndLogicImpl">
        <mapping  class="org.omg.uml.foundation.core.AssociationEnd$Impl">
              <context>org.andromda.cartridges.pojo.metafacades.PojoEntity</context>
        </mapping>
    </metafacade>
</metafacades>
```

図 3 - 15 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 - 16 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。

３．６　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 3－17 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。追加する部分を波線で示す。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 - 18 andromda-profile.xml の変更

「pojosample/mda/src/uml/PojoSampleModel.xmi」を MagicDraw 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。

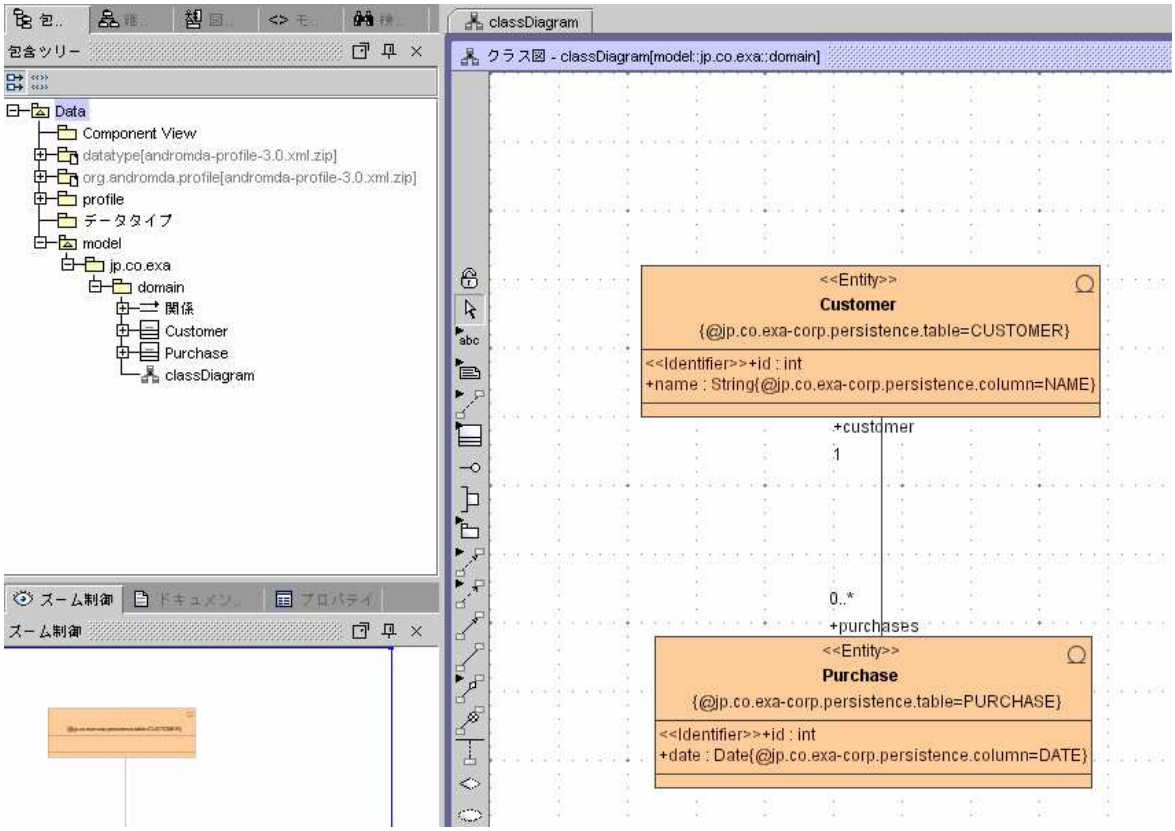

図 3 - 19 PojoSample のモデリング

・ 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]
```

・ 「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。

３．７　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。

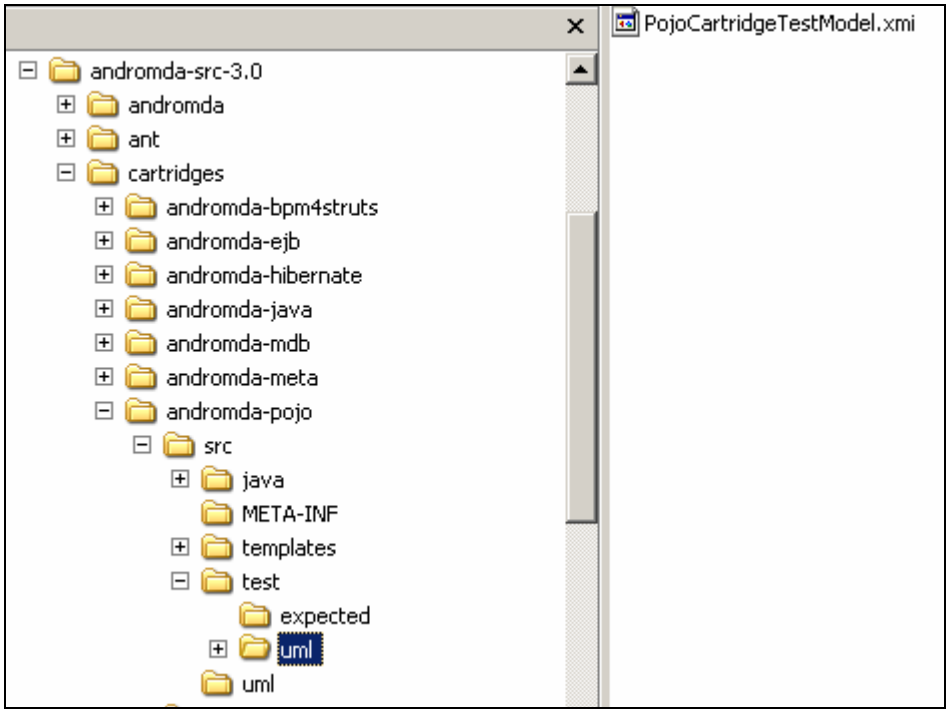

図 3－20 モデルの配置

・ 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}!/PojoCartridgeTestMode
l.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 3－21 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。

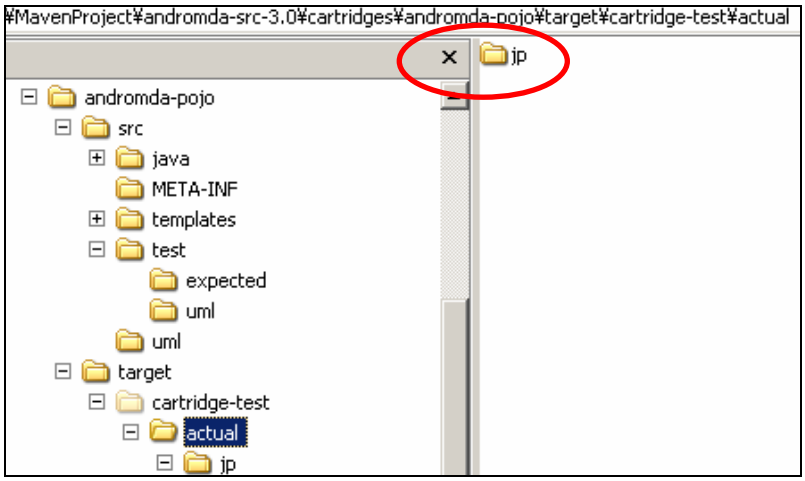

図 3－22 テスト検証用データの圧縮

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。

参考文献

[1]JMI Java doc

http://java.sun.com/products/jmi/jmi-1_0-fr-doc/javax/jmi/model/package-tree.html

[2]JMI Specification JSR-40

http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html

[3]UML Model

http://www.omg.org/docs/formal/03-03-01.pdf

[4]OMG XML Metadata Interchange (XMI) Specification

http://www.omg.org/docs/formal/02-01-01.pdf

[5]NetBeans MDR (Architecture)

http://mdr.netbeans.org/architecture.html