Java™ Metadata Interface(JMI) Specification

JSR 040
Java Community Process
http://www.jcp.org/

Version 1.0
Final Specification
07-June-2002

Technical comments:
jmi-comments@sun.com

Specification Lead:
Ravi Dirckze,
Unisys Corporation

DISCLAIMER

This document and its contents are furnished "as is" for informational purposes only, and are subject to change without notice. Unisys Corporation (Unisys) does not represent or warrant that any product or business plans expressed or implied will be fulfilled in any way. Any actions taken by the user of this document in response to the document or its contents shall be solely at the risk of the user.

UNISYS MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, WITH RESPECT TO THIS DOCUMENT OR ITS CONTENTS, AND HEREBY EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR USE OR NON-INFRINGEMENT. IN NO EVENT SHALL UNISYS BE HELD LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING FROM THE USE OF ANY PORTION OF THE INFORMATION.

Copyright

Trademarks

Please
Recycle

Contents

CHAPTER 1

Introduction

1.1 Metadata Interoperability

Today's Internet-driven economy has accelerated users' expectations for unfettered access to information resources and transparent data exchange among applications. One of the key issues limiting data interoperability today is that of incompatible metadata. Metadata can be defined as information about data, or simply data about data. In practice, metadata is what most tools, databases, applications and other information processes use to define the structure and meaning of data objects.

Unfortunately, most applications are designed with proprietary schemes for modeling metadata. Applications that define data using different semantics, structures, and syntax are difficult to integrate, impeding the free flow of information across application boundaries. This lack of metadata interoperability is hampering the development and efficient deployment of numerous business solutions. These solutions include data warehousing, business intelligence, business-to-business exchanges, enterprise information portals, and software development. Standardizing on XML Document Type Definitions (DTDs), which many industries are attempting to do as a solution to this problem, is insufficient, as DTDs do not have the capability to represent complex, semantically rich, hierarchical metadata.

1.2 The Java Metadata Interface Specification

The Java™ Metadata Interface (JMI) Specification defines a dynamic, platform-neutral infrastructure that enables the creation, storage, access, discovery, and exchange of metadata. JMI is based on the Meta Object Facility (MOF) specification from the Object Management Group (OMG), an industry-endorsed standard for metadata management.

The MOF standard provides an open-ended information modeling capability, and consists of a base set of metamodeling constructs used to describe technologies and application domains, and a mapping of those constructs to CORBA IDL (Interface Definition Language) for automatically generating model-specific APIs. The MOF also defines a reflective programming capability that allows for applications to query a model at run time to determine the structure and semantics of the modeled system. JMI defines a Java mapping for the MOF.

As the Java language mapping to MOF, JMI provides a common Java programming model for metadata access for the Java platform. JMI provides a natural and easy-to-use mapping from a MOF-compliant data abstraction (usually defined in UML) to the Java programming language. Using JMI, applications and tools which specify their metamodels using MOF-compliant Unified Modeling Language (UML) can have the Java interfaces to the models automatically generated. Further, metamodel and metadata interchange via XML is enabled by JMI's use of the XML Metadata Interchange (XMI) specification, an XML-based mechanism for interchanging metamodel information among applications. Java applications can create, update, delete, and retrieve information contained in a JMI compliant metadata service. The flexibility and extensibility of the MOF allows JMI to be used in a wide range of usage scenarios.

This document details the JMI 1.0 specification, which is based on the MOF 1.4 specification. The MOF 1.4 specification is located at:

http://www.omg.org/cgi-bin/doc?formal/02-04-03

1.3 Platforms

The JMI specification is defined as an extension to the Java platform. JMI facilities can be accessed from Java applications, applets, or Enterprise JavaBeans™ (EJB) applications. Details on the integration of JMI with the Java™ 2 Platform, Enterprise Edition (J2EE) can be found in Appendix A.

The JMI 1.0 specification is contained in a single Java package, javax.jmi, which is composed of three sub-packages: javax.jmi.model, javax.jmi.reflect, and javax.jmi.xmi. The contents of these packages are described in subsequent sections.

1.4 Target Audience

This specification is targeted primarily towards the vendors of:

- Metadata based solutions
- Data warehouse products
- Software integration platforms
- Software development tools

1.5 JMI Expert Group

The JMI 1.0 specification expert group consisted of the following members:

- Don Baisley, Unisys
- Stephen Brodsky, IBM
- Dan Chang, IBM
- Stephen Crawley, DSTC
- Ravi Dirckze, Unisys
- Bill Flood, Sybase
- David Frankel, IONA
- Petr Hrebejk, Sun
- Sridhar Iyengar, IBM
- Claes-Fredrik Mannby, Rational
- Martin Matula, Sun
- Dave Mellor, Oracle
- Davide Mora, Perfekt-UML
- Chuck Mosher, Sun
- Constantine Plotnikov, Novosoft
- John Poole, Hyperion
- Barbara Price, IBM
- Pete Rivett, Adaptive
- Peter Thomas, Oracle
- Barbara Walters, SAS Institute

1.6 Acknowledgements

The JMI expert group whishes to thank the OMG MOF Revision Task Force for their valuable contributions, and for their efforts to reconcile the discrepancies between JMI and MOF.

The expert group also thanks Jennifer Rodoni, Mingwai Cheung, and G. K. Khalsa who have worked tirelessly on the Technology Compatibility Kit (TCK), and Patrick Simpson who has worked tirelessly on the Reference Implementation (RI).

Finally, the Expert Group thanks the JCP Program Management Office (PMO) for all their help and support with respect to JCP matters.

CHAPTER 2

JMI Overview

2.1 The MOF Four-Layered Architecture

The goal of MOF is to provide a framework and services to enable model and metadata driven systems. The MOF is typically described using a layered metadata architecture consisting of a single meta-metamodel (M3), metamodels (M2) and models (M1) of information (see FIGURE 2-1).

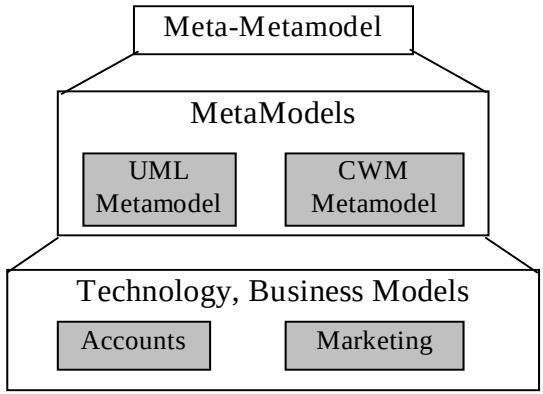

FIGURE 2-1 The layered architecture of the MOF.

Each meta level is an abstraction of the meta level below it. These levels of abstraction are relative, and provide a visual reference of MOF based frameworks. To put these terms in perspective, metamodeling is generally described using a four-layer architecture. These layers represent different levels of data and metadata. The four layers are:

- *Information*

  The information layer (also known as the M0 or data layer) refers to actual instances of information. These are not shown in the figure, but would be instances of a particular database, application objects, etc.

- *Model*

  The model layer (also known as the M1 or metadata layer) defines the information layer, describing the format and semantics of the data. The metadata specifies, for example, a table definition in a database schema that describes the format of the M0 level instances. A complete database schema combines many metadata definitions to construct a database model. The M1 layer represents instances (or realizations) of one or more metamodels.

- *Metamodel*

  The metamodel layer (also known as the M2 or meta-metadata layer) defines the model layer, describing the structure and semantics of the metadata. The metamodel specifies, for example, a database system table that describes the format of a table definition. A metamodel can also be thought of as a modeling language for describing different kinds of data. The M2 layer represents abstractions of software systems modeled using the MOF Model. Typically, metamodels describe technologies such as relational databases, vertical domains, etc.

- *Meta-metamodel*

  The meta-metamodel (M3) layer defines the metamodel layer, describing the structure and semantics of the meta-metadata. It is the common “language” that describes all other models of information. Typically, the meta-metamodel is defined by the system that supports the metamodeling environment.

2.2 The MOF Interfaces

In addition to the information modeling infrastructure, the MOF specification defines an IDL mapping for manipulating metadata. That is, for any given MOF compliant metamodel, the IDL mapping generates a set of IDL-based APIs for manipulating the information contained in any instance of that metamodel. Note that the MOF model itself is a MOF compliant model. That is, the MOF Model can be described using the MOF. As such, the APIs used to manipulate instances of the MOF Model (i.e., metamodels) conform to the MOF to IDL mapping.

A key goal of MOF models is to capture the semantics of the system of technology being modeled in a language and technology independent manner. It is also an abstraction of a system or technology rendered as a model. As such, it helps an architect deal with the complex systems by helping the architect visualize the metadata that is available. Beyond the visualization, the APIs provide the common programming model for manipulating the information. The MOF does not, however, prescribe how the information is to be stored or persisted.

The MOF also defines a set of reflective APIs. Similar to Java reflection, MOF reflection provides introspection for manipulating complex information. The MOF reflective interfaces allow a program to discover and manipulate information without using the tailored APIs rendered using the MOF to IDL mapping (or in the case of JMI, the MOF to Java mapping).

The Object Management Group's related XML Metadata Interchange (XMI) standard provides a mapping from MOF to XML. That is, information that has been modeled in MOF can be rendered in XML DTDs and XML documents using the XMI mapping.

2.3 Introduction to JMI

JMI is the Java rendition of the MOF. It can be viewed as an extensible metadata service for the Java platform that provides a common Java programming model for accessing metadata. Many software applications expose metadata that can be used by other applications for various purposes, such as decision support, interoperability and integration. Any system that provides a JMI compliant API to it's public metadata is a JMI service.

JMI provides the following to the J2EE environment:

- A metadata framework that provides a common Java programming model for accessing metadata.
- An integration and interoperability framework for Java tools and applications.
- Integration with OMG modeling and metadata architecture.

As the Java rendition of the MOF, the JMI specifies a set of rules that generate, for any given MOF compliant metamodel, a set of Java APIs for manipulating the information contained in the instances of that metamodel. The JMI specification also contains a Java implementation of MOF reflection.

2.3.1 JMI and the J2EE Platform

What is the value of JMI to the Java/J2EE platform? Quite simply, JMI is the common metadata infrastructure - a model based framework to represent and share metadata descriptions in the J2EE environment. In today's distributed, heterogeneous and autonomous environment, there are tremendous benefits to having a common metadata infrastructure some of which are described below.

Implementations of the JMI specification will provide a metadata management infrastructure that will greatly facilitate the integration of applications, tools, and services. Previously, complete interoperability and integration of disparate systems has been difficult because there has been no standard way to represent their unique characteristics. JMI provides the metadata framework that captures these semantics. Enterprise JavaBeansTM (EJBs) have proven to be highly effective at masking the complexities of the computing platform and enabling developers to build components without needing to directly handle transactions, security, resource pooling, and a host of other low-level programming tasks. Similarly, JMI will allow developers to mask complexities in their business logic by creating high-level models of their specific technology and business domains. Application-level integration is largely unaddressed by the J2EE framework, which addresses (and solves) mostly platform-level integration issues. JMI on the other hand, reduces the complexity of application level interoperability and integration by providing: 1 - a framework for defining technology/domain metamodels (i.e., a common domain "language" with which one can define the semantics of an application or component of that domain); 2 - a common programming model (APIs for metadata access which are automatically generated from the metamodel); and 3 - a common XML based interchange format (that corresponds to the metamodel).

A metadata framework must also be able to describe artifacts (applications, technologies, etc.) from multiple perspectives (i.e., views), and at different levels of abstraction (each representing a different level of detail). This allows one to focus on the part of the problem that is of interest to

him or her, and to include only the details that are relevant. When you think about metadata, note that one system's data may be another system's metadata from which, one may derive yet another set of metadata. Therefore, what is really needed is a meta-metamodel - a model for defining metamodels, or in other words, a language for defining metadata at a level of abstraction of interest to the user.

This is what the Meta Object Facility (MOF) specification from the Object Management Group (OMG) provides. It defines a small set of concepts (such as classes, methods, attributes, operations, references, data types, associations, and constraints) that allow one to define and manipulate models of metadata. Since the MOF uses UML notation, it is easy for people familiar with the UML to begin to use it to define and characterize their metadata. The MOF however, is only half the story, since we also need to define mappings from this level and abstraction to concrete languages and technologies. JMI is the mapping from MOF to the Java language. It defines how the constructs used by the MOF to model metadata map to the Java language.

As the Java mapping for MOF, JMI introduces OMG's Model Driven Architecture (MDA) to the Java/J2EE environment. MOF is the core of MDA - it the framework used to define the various technology and domain metamodels that constitute the MDA framework. These metamodels which can be represented using the UML since MOF used the UML notation (hence the term "model driven"), will allow one to define all aspects of the computing environment from different perspectives and from different levels of abstraction. JMI enables MDA in the Java/J2EE environment by providing the mapping from Java/J2EE to MOF, thereby allowing one to derive multiple perspectives of Java/J2EE artifacts at any level of detail of interest to the user.

2.3.2 Some JMI Use-Cases

Given below are a couple of use-cases that illustrate the advantages of JMI.

The Data Warehouse Management Scenario

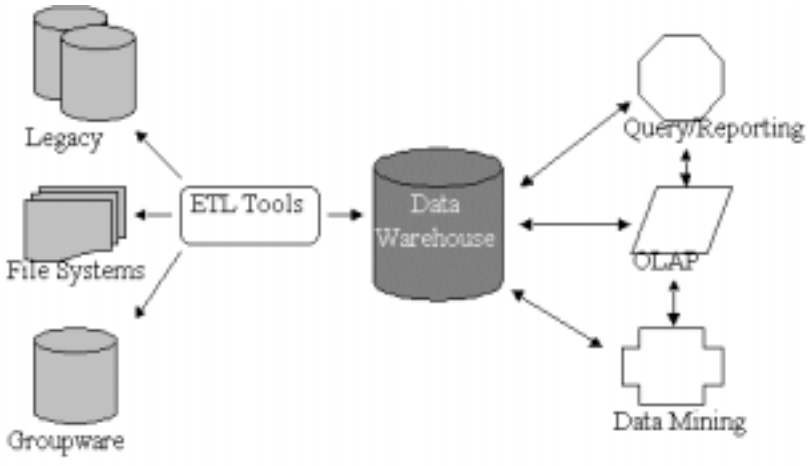

FIGURE 2-2 Data Warehousing Process

Data warehousing applications, because of the variety of inter-operational challenges they face, well illustrate some of the problems encountered by the lack of a common metadata infrastructure. The Data warehousing process has to deal with many different data sources, data warehouse formats, and analytical tools. For example, data could be collected from numerous sources, including databases, files that have captured web site click stream data, applications such as ERP or CRM systems, etc.. The first step of the data warehouse process is to *Extract*, *Transform*, and *Load* (a.k.a ETL process) this data into a data warehouse. There are over 250 vendors in the industry today that provide ETL tools for different kinds of data source on the input end, and analytical tools on the output end. An organization that is putting a data warehouse solution in place typically uses multiple ETL vendors to address their unique combination of data sources, and a collection of reporting tools to analyze the data that has been distilled into the data warehouse to generate the required reports. To address these complex inter-operational challenges, the data warehousing community requires a common metadata infrastructure that provides a common programming model, and a common interchange format. In fact, the data warehousing community is well on its way to defining a common metadata infrastructure based on the Common Warehouse Metamodel (CWM) and JMI (see JSR-69 “Java OLAP Interface” and JSR-73 “Data Mining API” for details).

The Software Development Scenario

This scenario illustrates using JMI as a platform for integrating heterogeneous software development tools to provide a complete software development solution. In most cases, the development team will be using a different tool for each task within the development process, or may even use different tools for the same task. Let’s take, for example, the development of a large Enterprise JavaBeans™ (EJB) application. Here, it is likely that the development team will use

one or more modeling tools, such as UML tools, to “model” the application, one or more Integrated Development Environments (IDEs) to develop the Java source, and one or more EJB deployment tools to mange the deployment of the application.

For this scenario, an EJB development solution can be built around JMI using three metamodels that represent the domains of the different tasks, i.e., the UML metamodel, the Java metamodel, and the EJB metamodel. Each tool would then participate in this integrated solution through an adapter that maps the tool specific APIs to the JMI APIs for the respective metamodel. Services that span multiple tasks, such as keeping the model and source code in sync, are then developed using the JMI APIs. The primary advantages of this solution over a hand crafted solution are:

- Services that span multiple domains, or even extensions to all tools of a single domain, can be developed using the common programming model provided by JMI. Note that such services and extensions are tool independent.
- The complexity of integration has been reduced from N x N where N is the number of tools being integrated, to M x M where M is the number of domains spanning the problem space (in this case, modeling, coding, and deployment). Adding a new tool would only require the development of a single adapter — all services that span the domain of that tool will then work with the new tool as well.

This example illustrates the advantages of abstraction (i.e., metamodels) and the common Java programming model for accessing metadata, in any integration framework. Integration platforms developed using JMI are inherently extensible.

CHAPTER 3

An Overview of the MOF Model

MOF is a three-layered model-based conceptual architecture for describing metadata. At the top of this architecture is the MOF Model (a.k.a. the metametamodel or M3 for short). This is the “abstraction language” used to define metamodels. The M3 is used to define information models for metadata (referred to as metamodels, or M2 for short).

This chapter provides an overview of the MOF Model. For a detailed description of MOF, the interested reader is referred to the MOF specification available on the OMG web page <http://www.omg.org/cgi-bin/doc?formal/02-04-03>.

3.1 The MOF Model

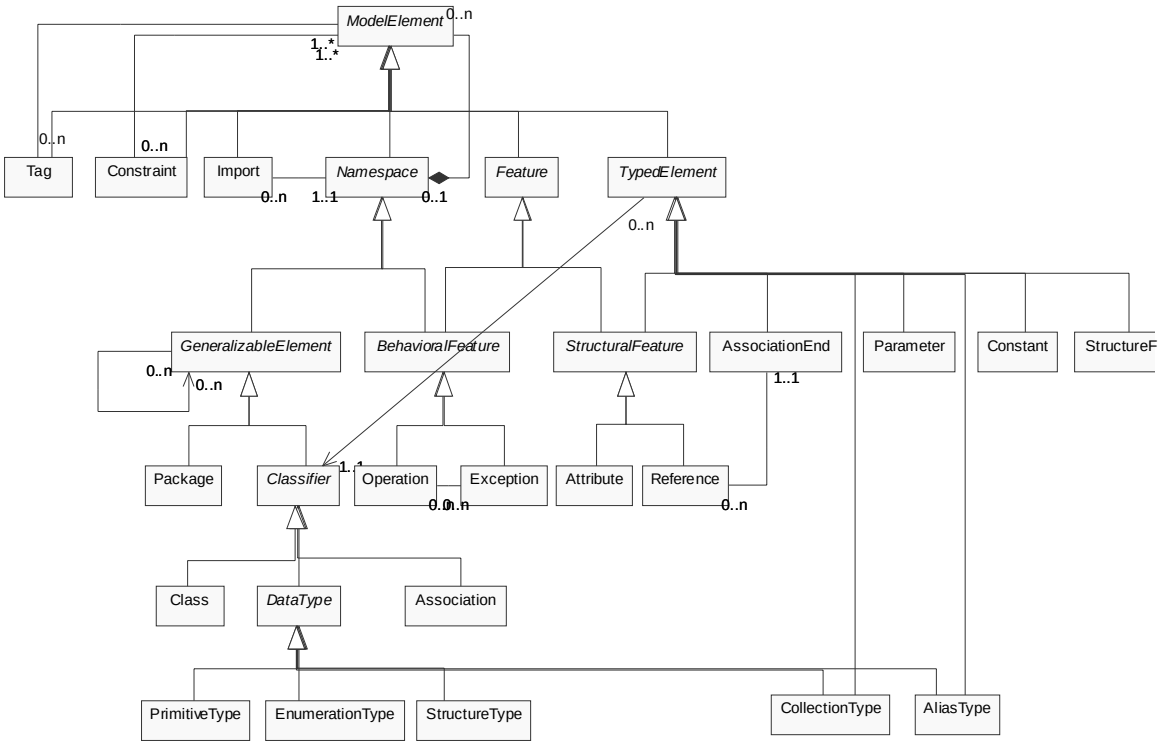

FIGURE 3-1 The MOF Model - overview

The MOF Model provides a set of modeling elements that are used for constructing metamodels, including rules for their use. Although the JMI specification contains the set of APIs used to manipulate MOF models, these interfaces do not provide the semantic information necessary to understand the behavior of MOF. Therefore, it is essential to understand MOF in terms of the model and related semantics, not just its interfaces.

The sections below describe the MOF metamodel in more detail.

- Section 3.1.1, “Common Superclasses” through Section 3.1.5, “Tags” break the MOF metamodel down into smaller, related groups of elements and describe them.

- Section 3.1.6, “MOF Model Elements” and Section 3.1.7, “MOF Model Associations” provide a detailed reference for each element and association in MOF.

3.1.1 Common Superclasses

FIGURE 3-1 “The MOF Model - overview” on page 12 shows the inheritance hierarchy for MOF. There is a common superclass, ModelElement from which everything inherits. FIGURE 3-1 “The MOF Model - overview” on page 12 shows some of the higher-level superclasses in more detail to illustrate what gets inherited:

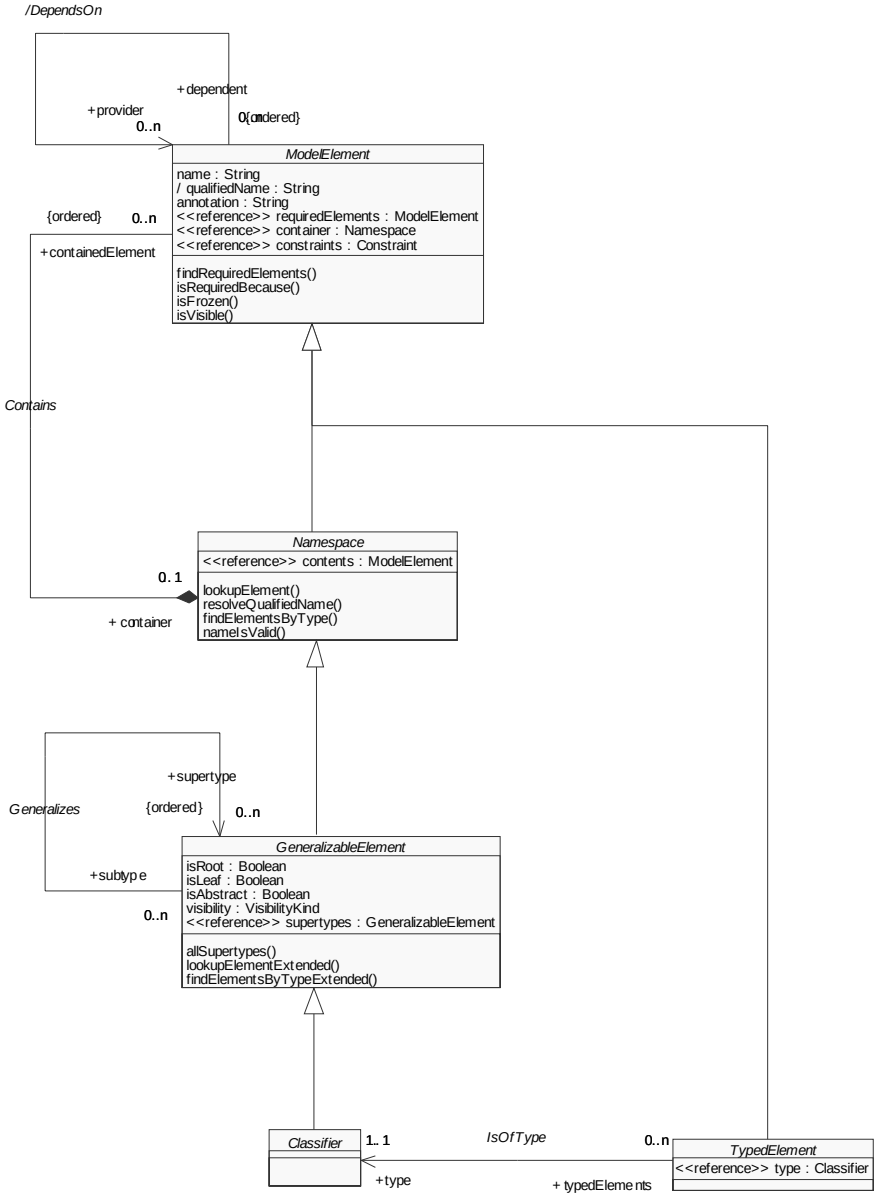

FIGURE 3-2 MOF Common Superclasses

From ModelElement, all elements will inherit a name and an annotation (or description). In addition all ModelElements are contained in one and only one Namespace (just a general container type which has several specialized subtypes for different things which can act like a container - for example a Class acts as a container for its Attributes and Operations). This model is supplemented with a detailed set of constraints that control which subtypes of Namespace can contain which other types. These rules are summarized in Section 3.1.2, “Containment Hierarchy,” on page 15.

3.1.2 Containment Hierarchy

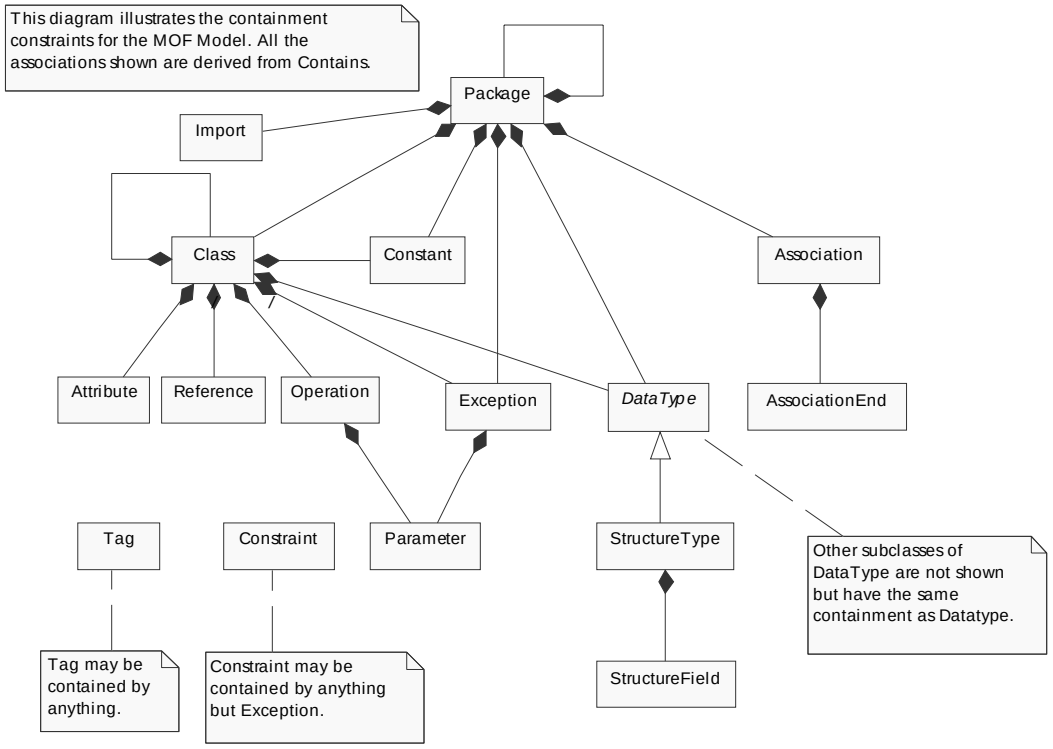

FIGURE 3-3 MOF Containment Hierarchy

The most important relationship in the MOF Model is the Contains Association. Containment is a utility Association that is used to relate (for example) Classes to their Operations and Attributes, Operations to their Parameters and so on. While the class diagram shows that ModelElement objects which are subtypes of Namespace can contain any other ModelElements, the MOF Model restricts the legal containments.

FIGURE 3-3 “MOF Containment Hierarchy” on page 15, shows all non-abstract subclasses of Namespace (i.e., possible containers) and the elements that they contain. Package is the 'highest level' container, and Packages may be nested inside each other. In fact Package is the only element that can really be 'top level' (or 'root') and not contained in something else. Each metamodel will have one such top-level package and this is used to represent the whole metamodel (at M2 level). Instances of that package are likewise used to represent whole models (at M1 level) and these are called package extents or repositories.

3.1.3 Types

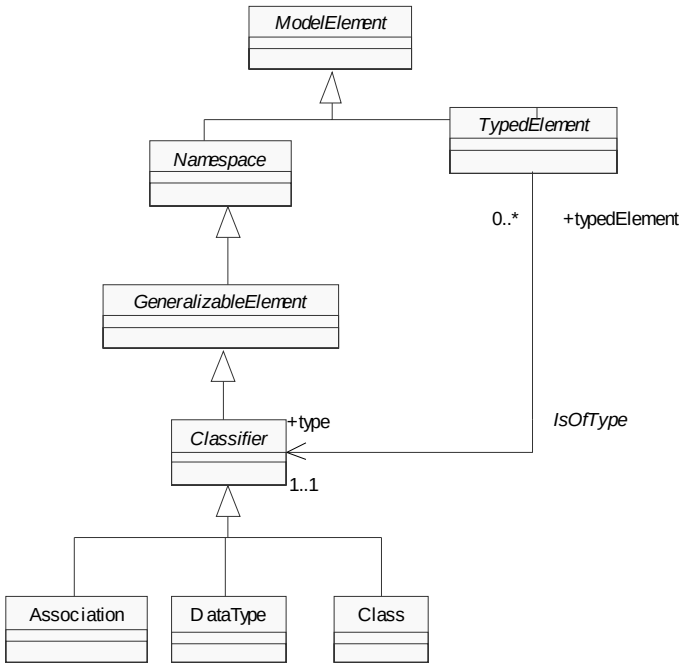

FIGURE 3-4 MOF Types

Classifier is the abstract superclass for MOF types - Class, Association, and DataType.

Class

A Class defines a classification over a set of object instances by defining the state and behavior they exhibit. This is represented through operations, attributes, references, participation in associations, constants, and constraints. Similar concepts are used in other environments for representing Classes and their implementations. However, in MOF the class characteristics are modeled in an implementation-independent manner.

Association

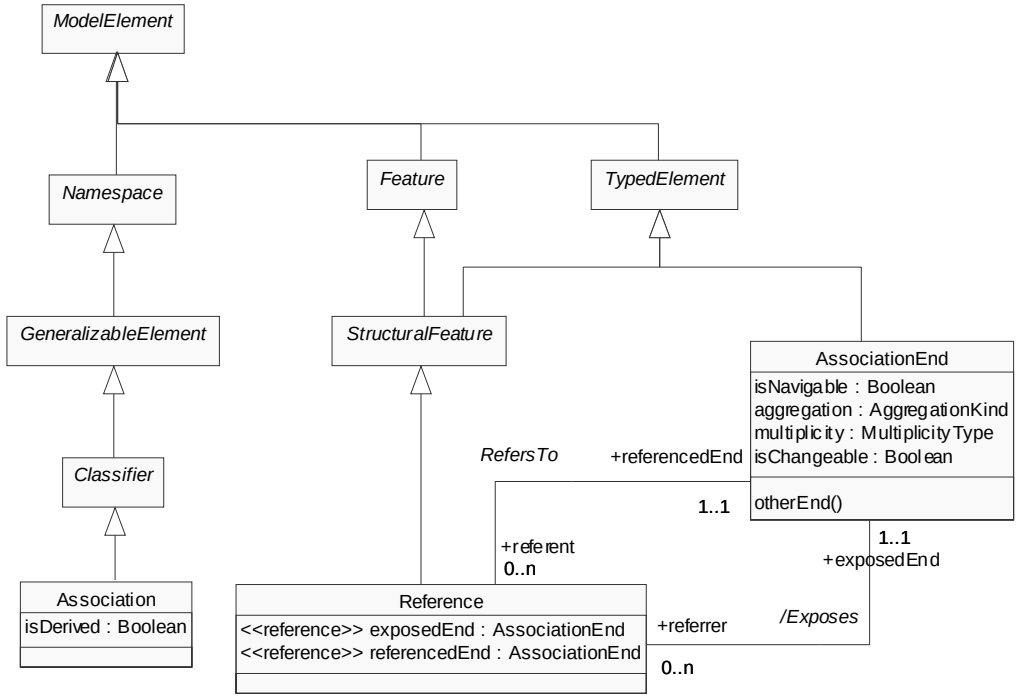

FIGURE 3-5 MOF Associations

FIGURE 3-5 “MOF Associations” on page 17 shows relationships (called Associations) in more detail. In MOF, as in UML, there's a 'duality' whereby the same stored link can either be accessed/updated from the perspective of the Association, or from the perspective of a class at either end (through defining a Reference as part of the Class that 'exposes' one of the AssociationEnds). Note that the link between Association and AssociationEnd (not shown here) is again via Contains inherited from Namespace.

DataTypes

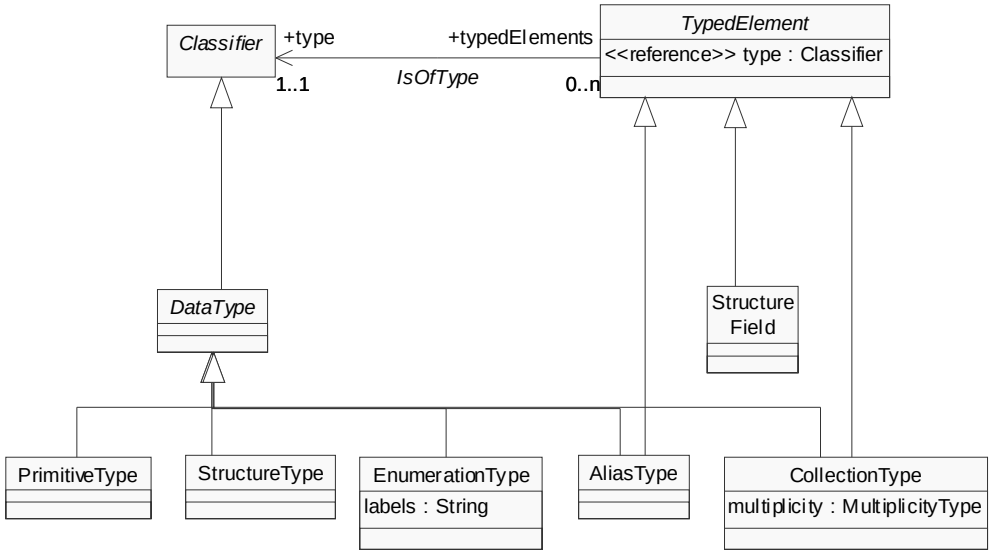

FIGURE 3-6 MOF DataTypes

The subtypes of DataType are new at MOF version 1.4, and were introduced to make MOF less dependent on CORBA and unified with the approach taken in JMI.

3.1.4 Features

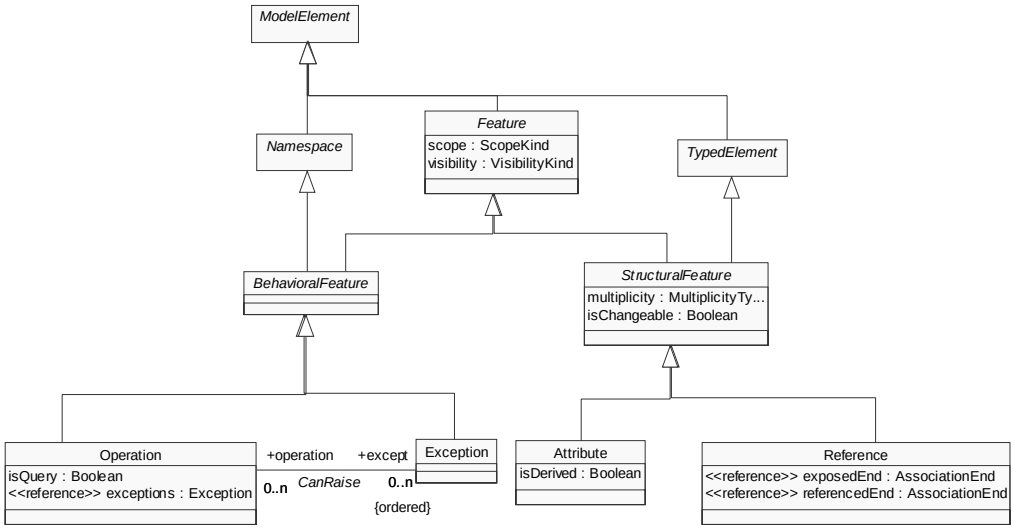

FIGURE 3-7 MOF Features

A Feature defines a characteristic of the ModelElement that contains it. Specifically, Classes are defined largely by a composition of Features. The Feature Class and its subclasses (StructuralFeature and BehavioralFeature) are illustrated in FIGURE 3-7 “MOF Features” on page 19.

A StructuralFeature defines a static characteristic of the ModelElement that contains it. The attributes and references of a Class define structural properties, which provide for the representation of the state of its instances.

A BehavioralFeature - Operation or Exception - defines a dynamic characteristic of the ModelElement that contains it.

As elsewhere in MOF, inheritance is used extensively to factor out common aspects of structure and behavior.

3.1.5 Tags

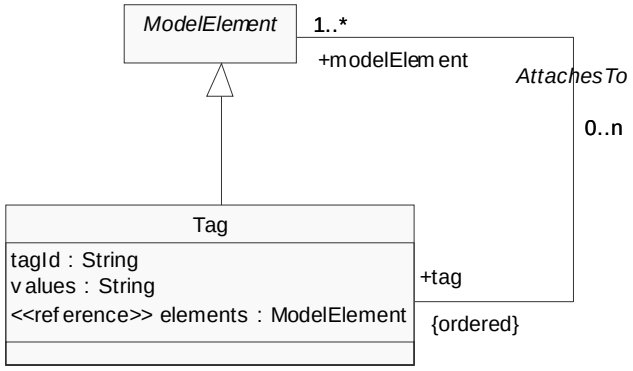

FIGURE 3-8 MOF Tags

The Tag model element is the basis of a mechanism that allows a “pure” MOF meta-model to be extended or modified. A Tag consists of:

• a name that can be used to denote the Tag in its container,

• a “tag id” that denotes the Tag’s kind,

• a collection of zero or more “values” associated with the Tag, and

• the set of other model elements that the Tag is “attached” to.

The meaning of a model element is (notionally) modified by attaching a Tag to it. The Tag’s “tag id” categorizes the intended meaning of the extension or modification. The “values” then further parameterize the meaning.

Section 4.6, “Standard Tags for the Java Mapping”, defines the standard tags that apply to the MOF to Java mapping.

3.1.6 MOF Model Elements

This section describes the 28 modeling elements that comprise the MOF Model.

t ModelElement

"ModelElement" is the root of all MOF elements. It represents the elementary constructs of model elements.

Abstract	Yes.
Super classes	None.
Attributes	**name**: unique name supplied by meta-modeler. **annotation**: description of the model element. **qualifiedName:** fully qualified name within the context of its outermost package extent.
References	**container**: identifies the Namespace element that contains this element. **requiredElements**: identifies the ModelElements whose definition this object depends on. **constraints**: identifies the set of Constraint objects that apply to this object.
Operations	**isFrozen**: reports whether the object is mutable. **isVisible**: this operation is reserved for future use. **isRequiredBecause**: checks whether the object depends on another object, and if so, returns the type of dependency.
MOF Constraints	C-1, C-2, C-3, and C-4.

t Import

An "Import" allows a package to reference model elements defined in some other package.

Abstract	No.
Super classes	ModelElement.
Attributes	**visibility**: this attribute is reserved for future use. **isClustered**: specifies whether the import represents simple package importation, or package clustering.
References	**importedNamespace**: returns the namespece object that it references.
Operations	None.
MOF Constraints	C-45, C-46, C-47, C-48, and C-49.

t Namespace

The “Namespace” element represents model elements that can contain other model elements.

Abstract	Yes.
Super classes	ModelElement.
Attributes	None.
References	**contents**: identifies the set of elements that the namespace object contains.
Operations	**lookupElement**: searches for a specified element within its contained elements. **resolveQualifiedName**: searches for a specified (fully qualified) element within its contained elements. **nameIsValid**: checks whether the given name can be used within the namespece. **findElementByType**: returns all elements identified by type, contained within the namespace.
MOF Constraints	C-5.

t Constraint

A “Constraint” defines a rule that restricts the state or behavior of one or more elements.

Abstract	No.
Super classes	ModelElement.
Attributes	**expression**: an expression that represents the constraint. **language**: the used to express the constraint. **evaluationPolicy**: the evaluation policy for the constraint.
References	**constrainedElements**: the elements that the constraint applies to.
Operations	None.
MOF Constraints	C-50, C-51.

t Tag

A “Tag” is an arbitrary name/value pair that can be attached to most elements. Tags provide an easy extension mechanism by allowing users to add information to a metamodel element.

Abstract	No.
Super classes	ModelElement.
Attributes	**tagId**: specifies the meaning of the tag. **values**: (ordered) specifies the string value(s) for the tag.
References	**elements**: the elements that this tag is attached to.
Operations	None.
MOF Constraints	None.

t Feature

A “Feature” defines a characteristic (e.g, operation or attribute) of a model element.

Abstract	Yes.
Super classes	ModelElement.
Attributes	**visibility**: this attribute is reserved for future use. **scope**: specifies whether the feature is a classifier scoped or instance scoped feature.
References	None.
Operations	None.
MOF Constraints	None.

t TypedElement

“TypedElement” is the abstraction of model elements that require a type as part of their definition (i.e., elements that represent types). The TypedElement itself does not define a type, but is associated with a “Classifier”.

Abstract	Yes.
Super classes	ModelElement.
Attributes	None.
References	**type**: provides the type represented by this element.
Operations	None.
MOF Constraints	None.

t GeneralizableElement

"GeneralizableElement" represents elements that can be generalized through supertyping and specialized through subtyping.

Abstract	Yes.
Super classes	Namespace.
Attributes	**visibility**: this attribute is reserved for future use. **isAbstract**: specifies whether the element is abstract. **isRoot**: specifies whether the element can be generalized (i.e., can have supertypes). **isLeaf**: specifies whether the element can be specialized (i.e., can have subtypes).
References	**supertypes**: identifies the set of supertypes for this element.
Operations	**allSupertypes**: returns the list of all (direct and indirect) supertypes. **lookupElementExtended**: returns an element whose name matches the supplied name. **findElementByTypeExtended**: returns all elements identified by type, contained within this element and all of its superclasses.
MOF Constraints	C-6, C-7, C-8, C-9, C-10, C-11, and C-12.

t Package

A "Package" represents an organizational construct in modeling. A package contains model elements.

Abstract	No.
Super classes	GeneralizableElement.
Attributes	None.
References	None.
Operations	None.
MOF Constraints	C-43, C-44.

t Classifier

"Classifier" represents a generalized modeling element that classifies instance objects by the features that they contain.

Abstract	Yes.
Super classes	GeneralizableElement.
Attributes	None.
References	None.
Operations	None.
MOF Constraints	None.

t Association

An "Association" represents a modeling element that classifies a set of links.

Abstract	No.
Super classes	Classifier.
Attributes	**isDerived**: specifies whether the association contains links or whether the link set is derived.
References	None.
Operations	None.
MOF Constraints	None.

t DataType

The "DataType" element represents a type for data values - which, unlike 'objects', do not have a lifetime or identity independent of their value.

Abstract	Yes.
Super classes	Classifier.
Attributes	None.
References	None.
Operations	None.
MOF Constraints	C-19, C-20.

t PrimitiveType

The “PrimitiveType” element represents a native/atomic data type. Six instances are predefined, in a separate package called PrimitiveTypes. These are: Integer, Boolean, String, Long, Double, Float.

Abstract	No.
Super classes	DataType.
Attributes	None.
References	None.
Operations	None.
MOF Constraints	None.

t EnumerationType

The “EnumerationType” element represents a data type with an enumerated set of possible string values.

Abstract	No.
Super classes	DataType.
Attributes	**labels**: the strings representing the enumerated values constituting the enumeration
References	None.
Operations	None.
MOF Constraints	None.

t StructureType

The “StructureType” element represents a data type that is an ordered ‘tuple’ of named StructureFields which are contained by the StructureType.

Abstract	No.
Super classes	DataType.
Attributes	None.
References	None.
Operations	None.
MOF Constraints	C-59.

t CollectionType

The “CollectionType” element represents a data type that is a finite collection of instances of another type (indicated by the inherited ‘type’ reference).

Abstract	No.
Super classes	DataType, TypedElement.
Attributes	**multiplicity**: describes the characteristics of the collection type.
References	None.
Operations	None.
MOF Constraints	None.

t AliasType

The “AliasType” element represents a different usage of another type (indicated by the inherited ‘type’ reference). It may constrain or just rename the type for a different purpose.

Abstract	No.
Super classes	DataType, TypedElement
Attributes	None.
References	None.
Operations	None.
MOF Constraints	None.

t Class

The “Class” element represents a (realizable) modeling element that classifies instance objects by the features that they contain.

Abstract	No.
Super classes	Classifier.
Attributes	**isSingleton**: specifies whether no more than one instance object or any number of instance objects may exist.
References	None.
Operations	None.
MOF Constraints	C-15, C-16.

t BehavioralFeature

The “BehavioralFeature” element defines a dynamic characteristic (e.g., an operation) of a model element.

Abstract	Yes.
Super classes	Feature, Namespace.
Attributes	None.
References	None.
Operations	None.
MOF Constraints	None.

t StructuralFeature

The “StructuralFeature” elements defines a static characteristic (e.g., an attribute) of a model element.

Abstract	Yes.
Super classes	Feature, TypedElement.
Attributes	**multiplicity**: defines the cardinality of an attribute/reference. **isChangeable**: specifies whether the attribute values are immutable (through the generated APIs).
References	None.
Operations	None.
MOF Constraints	None.

t Operation

The “Operation” element defines a dynamic feature that offers a service, i.e., an operation.

Abstract	No.
Super classes	BehavioralFeature.
Attributes	**isQuery**: specifies whether the behavior of the operation alters the state of the object.
References	**exceptions**: specifies the exceptions that the operation may raise.
Operations	None.
MOF Constraints	C-28, C-29, C-30.

t Exception

The “Exception” element defines an exception (or some abnormal condition).

Abstract	No.
Super classes	BehavioralFeature.
Attributes	None.
References	None.
Operations	None.
MOF Constraints	C-31, C-32.

t Attribute

The “Attribute” element defines a structural feature that contains a value (or values), i.e., an attribute.

Abstract	No.
Super classes	StructuralFeature.
Attributes	**isDerived**: specifies whether the value is part of the state of the object, or whether it is derived.
References	None.
Operations	None.
MOF Constraints	None.

t StructureField

The “StructureField” element represents a named and typed value within a StructureType.

Abstract	No.
Super classes	TypedElement.
Attributes	None.
References	None.
Operations	None.
MOF Constraints	None.

t Reference

A “Reference” defines a classifier’s knowledge of an association object which references that classifier, and access to the link set of the association.

Abstract	No.

Super classes	StructuralFeature.
Attributes	None.
References	**referencedEnd**: specifies the association end of principle interest to the reference. **exposedEnd**: the association end representing the end of the reference's owning classifier.
Operations	None.
MOF Constraints	C-21, C-22, C-23, C-24, C-25, C-26, C-27.

t Constant

The "Constant" element defines constant values of simple data types.

Abstract	No.
Super classes	TypedElement.
Attributes	**value**: the value of the constant.
References	None.
Operations	None.
MOF Constraints	C-52, C-53.

t Parameter

The "Parameter" element defines parameters used to communicate with BehavioralFeatures.

Abstract	No.
Super classes	TypedElement.
Attributes	**direction**: specifies the direction of information exchange (i.e., to pass a value into, to receive a value from, or both). **multiplicity**: defines the cardinality of the parameter.
References	None.
Operations	None.
MOF Constraints	None.

t AssociationEnd

An "AssociationEnd" represents one end of an association object. That is, an association is composed of two AssociationEnds.

Abstract	No.

Super classes	TypedElement.
Attributes	**multiplicity**: defines the cardinality of the association end. **aggregation**: defines whether the association end is constrained by “aggregate” semantics. **isNavigable**: specifies whether the association end supports navigation. **isChangeable**: specifies whether the association end can be updated (using the generated APIs).
References	None.
Operations	**otherEnd**: returns the other “AssociationEnd”.
MOF Constraints	C-39, C-40, C-41, C-42.

3.1.7 MOF Model Associations

t DependsOn

"DependsOn" is a derived association that identifies the collection of model elements that a given model element's structure depends on.

end1: **dependent**	This end identifies the dependent element. end1Class: ModelElement multiplicity: zero or more
end2: **provider**	This end identifies the elements that the "dependent" end depends on. end2Class: ModelElement multiplicity: zero or more

t AttachesTo

The "AttachesTo" association associates tags with model elements.

end1: **modelElement**	Identifies the model element that the tag is attached to. end1Class: ModelEelement multiplicity: one or more
end2: **tag**	Identifies the tags attached to a model element. end2Class: Tag multiplicity: zero or more

t Contains

The "Contains" composite association defines the model elements contained by a namespace.

end1: **container**	Identifies the composing container. end1Class: Namespace multiplicity: zero or one
end2: **containedElement**	Identifies the contained elements. end2Class: ModelElement multiplicity: zero or more

t Aliases

"Aliases" identifies the imported Namespace.

end1: **importer**	Identifies the element that imports a namespace. end1Class: Import multiplicity: zero or more
end2: **imported**	The namespace that is imported. end2Class: Namespace multiplicity: exactly one

t Constrains

"Constrains" identifies the constraints, if any, on a model element.

end1: **constraint**	Identifies the constraints. end1Class: Constraint multiplicity: zero or more
end2: **constrainedElement**	Identifies the constrained elements. end2Class: ModelElement multiplicity: one or more

t Generalizes

The "Generalizes" association identifies a supertype/subtype relationship.

end1: **supertype**	Identifies the supertype (i.e., generalized element). end1Class: GeneralizableElement multiplicity: zero or more
end2: **subtype**	Identifies the subtype (i.e., the specialized element). end2Class: GeneralizableElement multiplicity: zero or more

t IsOfType

The "IsOfType" association identifies the type of a typed element.

end1: **type**	Identifies the typed element. end1Class: Classifier multiplicity: exactly one
end2: **typedElement**	Identifies the set of typed elements supported by the classifier. end2Class: TypedElement multiplicity: zero or more

t CanRaise

The “CanRaise” association identifies the exceptions that can be raised by an operation.

end1: **operation**	The set of operations that can raise this exception. end1Class: Operation multiplicity: zero or more
end2: **except**	The set of exceptions that this operation can raise. end2Class: Exception multiplicity: zero or more

t RefersTo

The “RefersTo” association defines the association end that a reference refers to.

end1: **referent**	The reference object that refers to an association end. end1Class: Reference multiplicity: zero or more
end2: **referencedEnd**	The association end being referenced. end2Class: AssociationEnd multiplicity: exactly one

t Exposes

“Exposes” defines the opposite association end of the association end that a reference refers to.

end1: **referer**	Identifies the referencing reference. end1Class: Reference multiplicity: zero or more
end2: **exposedEnd**	The reference’s owning classifier’s end in the association. end2Class: AssociationEnd multiplicity: exactly one

3.2 Discrepancies between JMI and MOF

The current JMI specification is a Java language mapping for the OMG MOF version 1.4 specification. In order to support a Java “friendly” API, the following changes have been introduced:

- The Supertype Closure rule has been modified to reflect the changes to the reflective framework.
- The underflow constraint on classifier level and instance level attributes is changed from immediate to deferred in order to support default factory operations that take no arguments.
- Collections are ‘live’ (see “JMI Collection Semantics” on page 40).

In addition, the JMI reflective interfaces and the exception framework are significantly different from their IDL counterparts. These changes however, do not alter the semantics of the MOF.

3.3 XMI

JMI provides APIs for stream based information exchange between JMI services. Such streams shall be in the XML Metadata Interchange (XMI) version 1.2 format. The XMI 1.2 specification is located at:

http://www.omg.org/cgi-bin/doc?formal/02-01-01

CHAPTER 4

MOF to Java Mapping

This chapter defines the standard mapping from a MOF compliant metamodel to Java interfaces. The resulting interfaces are designed to allow a user to create, update and access instances of the metamodel using Java client programs.

Note – For any MOF model, JMI defines the templates for generating the Java APIs for accessing and updating the metadata, and optionally, for generating the APIs for managing the metadata. The javax.jmi.ignoreLifecycle tag (see "Tag for controlling Lifecycle API generation" on page 52) is used to control the API generation, that is, when this tag is set to 'true', APIs dealing with the management of the metadata are suppressed.

4.1 Metaobjects and Interfaces

This section describes the different kinds of metaobjects that represent MOF based metadata, and how they relate to each other.

4.1.1 Metaobject Type Overview

The MOF to Java interface mapping and the Reflective package share a common, object-centric model of metadata with four kinds of M1-level metaobjects; i.e. "instance" objects, "class proxy" objects, "association" objects, and "package" objects.

Package Objects and Package Creation

A package object is little more than a "directory" of operations that give access to a collection of metaobjects described by a metamodel. The outermost package extent (a.k.a. outermost extent) represents the "root" of the object-centric model of the metadata. All other objects (i.e., instance objects, class proxies, associations, and (nested) package objects) are contained within some outermost extent and are created using the accessors provided by the MOF.

Class Proxy Objects

A class proxy object serves a number of purposes:

- It is a factory object for producing instance objects within the Package extent.
- It is the intrinsic container for instance objects.
- It holds the state of any classifier-scoped attributes for the class.
- It provides operations corresponding to classifier-scoped operations.

The interface of a class proxy object provides operations for accessing and updating the classifier-scoped attribute state. The interface also provides factory operations that allows the client to create instance objects.

Instance Objects

An instance object holds the state corresponding to the instance-scoped attributes, and any other "hidden" state implied by the class specification. Generally speaking, many instance objects can exist within a given package object.

Instance objects are always tied to a class proxy object. The class proxy provides a factory operation for creating instance objects. When an instance object is created, it is automatically added to the class proxy container. An instance is removed from the container when it is destroyed.

The interface for an instance object provides:

- Operations to access and update the instance-scoped and classifier-scoped attributes.
- Operations corresponding to instance-scoped and classifier-scoped operations.
- Operations to access and update associations via reference.

Association Objects

An association object holds a collection of links (i.e. the link set) corresponding to an association defined in the metamodel. The association object is a "static" container object (similar to a class proxy object) that is contained by a package object. Its interface provides:

- Operations for querying the link set.
- Operations for adding, modifying and removing links from the set.
- An operation that returns the entire link set.

A link is an instance of an association object that represents a physical link between two instances of the classes connected by the association object.

4.1.2 The Metaobject Interface Hierarchy

This section describes the patterns of inheritance of the Java interfaces generated by the MOF to Java mapping. The patterns are illustrated in FIGURE 4-1. This figure shows an example MOF metamodel expressed in UML (on the left) that consists of two Packages, P1 and P2. The first Package P1 contains Classes C1 and C2, where C2 is a subclass of C1 and an Association A that connects C1 and C2. The second Package P2 is then defined as a subpackage of P1.

The UML class diagram (on the right) shows the inheritance graph for the generated interfaces corresponding to the example metamodel.

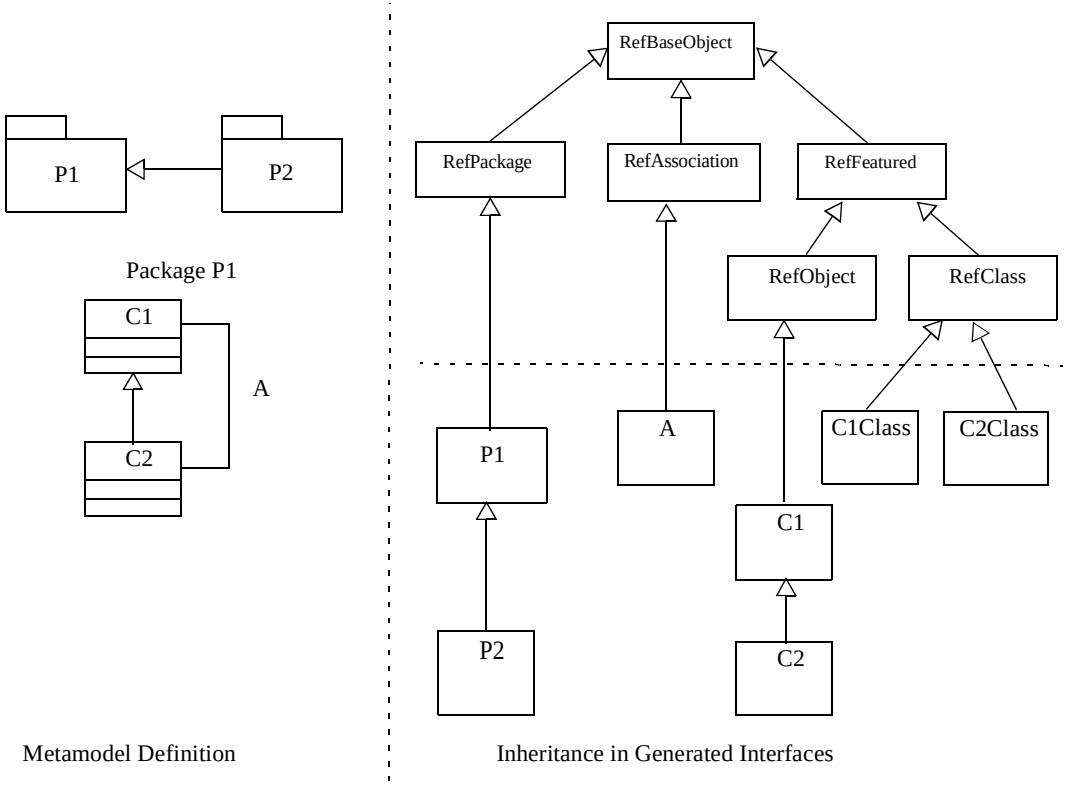

FIGURE 4-1 Generated Java inheritance patterns.

The root of the inheritance graph is a group of predefined interfaces that make up the Reflective package (see Chapter 5, *MOF Reflective Package*). These interfaces collectively provide:

- Operations that implement object identity.
- Operations that provide introspection.
- Operations for exercising the functionality of an object independent of its (metamodel-specific) generated interface.

Note – The interfaces in the Reflective package are all designed to be “abstract”; i.e. it is not anticipated that they should be the “most derived” type of any metaobject.

The interfaces for the Package objects, Association objects, Class Proxy objects and Instance objects provide functionality as described previously. The inheritance patterns are as follows:

- An Instance object that has no supertypes extends RefObject; all other Instance objects extend their supertypes.
- A Package object that has no supertypes extends RefPackage; all other Package objects extend their supertypes.
- All Class Proxy objects extend RefClass.
- All Associations extend RefAssociation.

4.2 Computational Semantics for the Java Mapping

This section describes the MOF's general computational semantics for the MOF to Java interface mapping.

4.2.1 Equality in the Java Mapping

JMI maps all MOF objects to Java object that implements the (reflective) RefBaseObject interface. Equality of JMI objects should be implemented as follows:

- Two JMI objects are equal if and only if the "refMofId" operation defined in the javax.jmi.reflect.RefBaseObject interface returns the same string for both objects.

4.2.2 The Java NULL Value

For an attribute, reference, or parameter whose multiplicity lower bound is 0 and upper bound is 1, Java NULL indicates that the attribute (reference or parameter) has no value, that is, it has a cardinality of zero. For an attribute, reference, or parameter whose multiplicity lower bound is 0 and upper bound is greater than 1, the empty collection indicates that it has no value.

4.2.3 JMI Collection Semantics

The java.util.Collection interface and its specializations (specifically, java.util.List) are used throughout the JMI APIs. Unless otherwise specified, Java collections that appear in the generated and reflective APIs have live semantics as opposed to copy semantics. That is, unless otherwise stated in the specification or prohibited by the metamodel (i.e., metamodel specifies that the source is immutable), the source collection can be updated using:

- Operations defined in the Java collection interfaces.
- Operations defined in the Iterator interface.

In addition, the collections behave as follows:

- Any change in the source collection is immediately reflected in the live Collection.
- Any change in the live collection is immediately reflected in the source collection.
- The behavior of an iterator becomes unspecified when its underlying collection is modified in any way other than through the iterator. In such situations, it may even throw an exception.
- Operations defined in the collection and iterator interfaces are executed as atomic operations.

JMI does not however specify the semantics of Collections returned by modeled operations (i.e., operations explicitly listed in the metamodel). It is the responsibility of the modeler/implementor of the operation to choose the required semantics.

Note – When JMI collection and iterator operations are used within the context of some concurrency control mechanism such as transactions, the semantics of the specific concurrency control mechanism override the semantics defined above.

4.2.4 Lifecycle Semantics for the Java Mapping

This section defines the Java interface mapping's computational model for metaobject creation and deletion. It also gives definitions of copy semantics, though these should currently be viewed as indicative rather than normative.

Package Object Creation and Deletion Semantics

The JMI specification does not specify how a outermost package extent is created. This is due to the fact that the concept of a JMI service has not been defined in the current version of the specification. However, it must be noted that normative APIs for a JMI service will be defined in a future version of the specification.

When an outermost extent is created, instances of the following dependent objects are automatically created along with the package object:

- A package object is created for each nested package within the outermost package extent.
- A package object is created for each clustered package within the outermost package extent.
- A class proxy object is created for each class within the outermost package extent.
- A association object is created for each association within the outermost package extent.

The dependent packages and class proxy objects are initialized so that the reflective refOutermostPackage and refImmediatePackage operations return the appropriate objects.

When an M1-level class proxy object is created, the values of the non-derived classifier-level attributes are initialized. Initially, the collections returned by the reflective refAllOfType and refAllOfClass operations will be empty, since no M1-level instance objects will have been created in the class proxy extent.

Instance Object Lifecycle Semantics

An instance object can be created by invoking the appropriate create operation declared in the class proxy object. An instance object is created within the extent of the respective class proxy object.

The class proxy object may not be found if the object violates the Supertype Closure Rule (see "The Supertype Closure Rule" on page 48). Creation of an instance object will also fail if the corresponding class is abstract (i.e., has isAbstract set to true). Similarly, it will fail if the class is a "singleton" class and an instance object for that class already exists within the class proxy's extent. In the above cases, an exception is raised.

When an instance object is (successfully) created within the extent of a class proxy object, it becomes part of the collection returned by the class proxy object's reflective refAllOfClass and refAllOfType operations. The instance object remains a member of that collection for its lifetime; i.e. until it is deleted.

An instance object will be deleted in the following three situations:

- When a client invokes the reflective "refDelete" operation on the instance object.
- When the outermost extent containing the instance object is deleted.
- When the instance object is a component of a "composite" instance that is deleted. This applies to composites formed by both associations and attributes.

When an instance object is deleted, the following things must occur:

- The binding between the instance object and its object reference(s) must be revoked.
- The instance object must be removed from its class proxy object's container.
- Any instance objects that are components of the object being deleted must also be deleted.
- Links involving the deleted instance object should be deleted as per the "Link Lifecycle Semantics" specification below.

Link Lifecycle Semantics

Links can be created and deleted in various ways. These include:

- Using the operations on the association objects.
- Using the operations corresponding to references on instance objects.
- By deleting one or other linked instance objects.
- When the service notices that a linked instance object no longer exists.

A link is created within the extent of an association object, and becomes part of the collection returned by the association object's reflective refAllLinks() operation. A link remains within the extent in which it was created for the lifetime of the link; i.e. until it is deleted. When a link is deleted, it is removed from the association's extent.

Deletion of an instance object *may* causes certain links to that object to become invalid references. Ideally, a well-formed association instance should not contain such links. In practice however, the immediate removal of invalid links from an association instance cannot always be implemented, particularly, in the case of links that cross outermost package extent boundaries.

Instead, a JMI service is required to behave as follows. When an instance object is deleted:

- All links referring to the instance object that belong to association instances within the same outermost package extent as the instance object *must* also be deleted.
- Any links referring to the instance object that belong to association instances in another outermost package extent as the instance object *may* also be deleted.

Note – The above semantics mean that an association instance can legally contain links that refer to defunct instance objects in other outermost package extents.

4.2.5 Association Access and Update Semantics for the Java Mapping

This section describes the computational semantics of the association object access and update operations defined in the MOF to Java Interface Mapping. With a couple of exceptions, these semantics transform one *Well-formed State* (as defined in "A Mathematical Model of Association State" in Chapter 4 of the MOF specification) to another. The exceptions are as follows:

- Deletion of an instance object in another outermost package extent may cause a association link set to contain links that are no longer valid.
- Deletion of an instance object can cause a link set to contain fewer links than is required.

Since an association requires that links connect instance objects, it is not legal to pass a null object reference as a parameter to any operation on an association.

Access Operations

There are two kinds of link access operations in the association interface generated by the Java Interface mapping:

- The "get<associationEndName>" operations return a collection containing the projection of the corresponding end of the association's link set.
- The "exists" operation tests for the existence of a given link in the association's link set.

These operations are defined to be side-effect free; i.e. they do not modify the *State* of the Association instance.

Link Addition Operations

The set of operations for adding links to an associations extent vary, depend on whether it has an ordered "AssociationEnd":

- For an unordered association, the "add" operation adds a link to the association's link set.
- For an ordered association, the "add" operation adds a link to the association's link set. The new link is added the end of the existing link set.

A number of constraints apply to the link addition operations:

- When adding a link, the link must reference existing instance objects.
- An operation cannot add a link that already exists.
- An operation cannot add a link if it would violate the multiplicity of either end of the association.
- An operation cannot add a link that creates a Composition cycle, or that violates the Composition or Reference Closure rules.

Link Modification Operations

Multivalued association ends can be modified using the operations defined in the Collection (and List) APIs

A number of constraints apply to Collection operations that result in links being modified:

- The link being modified need not be a valid link. That is, it can modify an existing link that has been made invalid (by say, the deletion of an instance object).
- The modified link must be a valid link.
- The modified link cannot already be a member of the link set.
- The operations cannot produce a link that creates a Composition cycle, or that violates the Composition or Reference Closure rules.

Link Removal Operations

The “remove” operation can be used to delete an existing link from the link set. The constraints that apply to the link removal operation are:

- The operation cannot remove a link that is not a member of the link set. However, it should succeed if the link is not a valid link.

Derived Associations

Setting “isDerived” to be true for an association object is a “hint” that the association’s link set should be computed from other information in the model. Apart from this, the Java Interface mapping makes no distinction between derived and non-derived associations. Equivalent Java interfaces are generated in each case, and the semantics are defined to be equivalent. If a derived association’s operations are coded by hand, it is the programmer’s responsibility to ensure that they implement the required semantics.

4.2.6 Attribute Access and Update Semantics for the Java Interface Mapping

The Java interface mapping maps attributes to a variety of operations, depending on the attribute’s “multiplicity” settings. There are three major cases; i.e. single-valued with bounds of [1..1], optional with bounds of [0..1], and multivalued.

Note – The lifecycle semantics for attributes in the Java interface mapping mean that an accessor operation can return a reference for a non-existent object.

Single-Valued Attributes

A single-valued attribute is mapped to two Java interface operations; i.e. a “get<*attributeName*>” operation that gets its value and a “set<*attributeName*>” operation that sets its value.

The get<*attributeName*> operation returns the current value of the attribute, which is a single instance of the attribute’s base type as mapped by the Java mapping. The set<*attributeName*> operation replaces the current value of the attribute with a new value. As before, the new value is a single instance of the attribute’s base type as mapped by the Java mapping.

The behavior of the set<*attributeName*> operation for an attribute whose type is a class is constrained as follows:

- The new value supplied must be an existing instance object.

- The new value (i.e. the component instance) must not already be a component of another instance object.
- The composite and component instance objects must belong to the same outermost package extent, i.e., the Composition Closure rule must not be violated.

“Optional Attributes

An optional attribute maps to the same operations as for single-valued with the added capability that the attribute can be set to null.

Multivalued Attributes

The interfaces and semantics for multivalued attributes depend on the settings of the “isOrdered” and “isUnique” fields of the attribute’s “multiplicity” property.

For multivalued attributes, only a “get<*attributeName*>” operation is provided. If “isOrdered” is false, the operation returns a java.util.Collection; if “isOrdered” is true, the operation returns a java.util.List. The base type of each element in the returned collection is the attribute’s base type.

Note – The return type of the “get<*attributeName*>” operation is determined by the “isOrdered” field. The “isUnique” field however, has no affect on the return type. This is because orderedness is a characteristic of a multivalued attribute (or reference), while uniqueness is a constraint.

All other operations to manipulate multivalued attributes are provided through the Collection and List interfaces.

A number of restrictions apply when setting a value of a multivalued attribute:

- If the attribute’s multiplicity has the “isUnique” flag set to true, no two instances in the collection may be equal. If the user attempts to add a duplicate instance, the JMI “DuplicateException” must be thrown.
- If the attribute’s multiplicity has a “upper” value other than the “UNBOUNDED” value, there can be at most that many elements in the collection.

If a multivalued attribute’s type is a class, the following restrictions also apply:

- The new element must be an existing object of the attribute’s base type.
- The new element must not be a component of another object.
- The composite and every component instance objects must belong to the same outermost package extent; i.e. the Composition Closure rule must not be violated.

Changeability and Derivedness

If “isChangeable” is set to false for a single-valued or optional-valued attribute, mutator operations are not generated. For multi-valued attributes, the value of the “isChangeable” flag has no effect on the generated Java APIs. However, any operation on the collection or list interface that attempts to change the source data must fail.

The lack of mutator operations for single and optional-valued attributes, or the restriction on the set operations in the Collection interface for multi-valued attributes does not preclude the existence of other mechanisms for updating the attribute's state.

The value of the "isDerived" flag has no effect on the generated Java APIs. The operations for the derived and non-derived cases are equivalent and they are defined to have equivalent semantics. If a derived attribute's operations are coded by hand, it is the programmer's responsibility to ensure that they implement the required semantics.

Classifier Scoped Attributes

An attribute whose "scope" is "classifier_level" differs from one whose "scope" is "instance_level" in the following respects:

- When the attribute has aggregation semantics of "composite":
  - The Composition Closure rule means that the class proxy object and attribute value instances must belong to the same extent.
  - Checking for composition cycles is unnecessary. The class proxy object is not an instance object, and thus, cannot be a "component".

Inherited Attributes

The semantics of an inherited attribute is equivalent to that of one defined in that class.

Life-cycle Semantics for Attributes

The previous semantic descriptions say nothing about how an attribute gets its initial value or values. In the Java mapping, attributes get their initial value as follows:

- If the "create<*ClassName*>" operation that takes an initial value for all attributes of that class is used, then the attributes are initialized from the parameters to the create operation.
- If the default "create<*ClassName*>" operation (that takes no arguments) is used, then the attributes will be initialized as follows:
  - If the attributes value can be null (i.e., in the Java mapping it gets mapped to some Java object), then it will be initialized to null.
  - If the attribute value cannot be null (i.e., in the Java mapping, it gets mapped to some Java scalar type such as "int") then it will get initialized to an implementation specific value.
- "classifier_level" attribute initialization follows the same rules as those defined for the default "create<*ClassName*> operation.

Attributes with "composite" aggregation semantics have special life-cycle semantics. When an object with a composite attribute is deleted, the instance object or objects that form its value are also deleted.

Note that unlike associations, when an instance object is deleted, the delete operation should make no attempt to tidy up "dangling references" to it.

Note – In order to support the default create operation in JMI, the underflow constraints on classifier_level and instance_level attributes *WILL* be deferred rather than immediate. The default create operation is provided as a convenience to the Java programmer. However, it is expected that programmer will use the set<*AttributeName*> (or equivalent) operation to set the value of (un-assigned) attributes in a timely manner. It must also be noted that, if an attribute is read-only (isChangable = false), or its visibility is private_vis or protected_vis, then no mutator operations will be generated, or in the case of multi-valued attributes, the mutator operations in the collection API are required to fail.

4.2.7 Reference Semantics for the Java Mapping

References combine attribute style interfaces with association access and update semantics. The Java mapping maps references to attribute style interface operations. Let *i* be an instance object containing a reference *ref*, *a* be the association object for *ref*, and *c* be the collection returned by the "get" operation for *ref* for the case where *ref* is a multivalued reference, then each reference operation maps fairly directly onto an association interface operation as shown below.

Multiplicity	Reference Operation	Association Operation(s)
optional, single- and multivalued	i.get<*referenceName*>();	a.<*referencedEndName*>(i);

optional and single-valued	i.set<*referenceName*>(new);	old = a.<*referenceEndName*>(i);
		if old.size > 0 then
		  a.remove(i, old);
		  a.add(i, new);

TABLE 4-1 Semantic mapping of reference operations to association operations.

In practice, an implementation also needs to transform exceptions reported for the association operations into exceptions that apply from the reference perspective.

Note – The above semantic mapping description is not intended to imply any particular implementation approach.

4.2.8 Cluster Semantics for the Java Mapping

A clustered Package behaves identically to a nested Package in terms of life-cycle and extent rules. The only significant difference is that clustering is not always a strict composition relationship. In the Java mapping, this means that two or more "get<*PackageName*>" operations may point at the same clustered package instance.

4.2.9 Atomicity Semantics for the Java Mapping

All operations defined by the Java mapping (including the Reflective versions) are required to be atomic:

- If an operation succeeds, state changes required by the specification should be made, except as noted below:
  - When an instance object is deleted, deletion of any component instance objects may occur asynchronously.
  - When an instance object is deleted, removal of links to the deleted instance object may occur asynchronously.
- If an operation fails (e.g. by raising an exception), no externally visible changes should be caused by the failed operation.

Note – The JMI specification *does not require* a transactional or persistent implementation of the metadata service.

4.2.10 The Supertype Closure Rule

The inheritance pattern for instance and class proxy interfaces has an important consequence when one metamodel class is a subclass of a second one. Problems arise when an M2-level class (e.g. P2.C2) has a superclass that is imported from another M2-level package (e.g. P1.C1). Here, operations corresponding to classifier level features of P1.CI are copied to the class proxy interface of P2.C2. If package P2 simply imports P1 where isClustered is set to false, then the operations corresponding to classifier level features of P1.C1 are no longer valid for the instance and class proxy interfaces of P2.C2.

The adopted solution to this problem is to add an extra restriction to the MOF computational semantics. This restriction is known as the *Supertype Closure Rule*:

Supertype Closure Rule: An M2 level package can be instantiated only if for each class contained in that package, all its (direct and indirect) supertypes are also contained within that package.

4.3 Primitive Data Type mapping

The default mapping of MOF primitive types to Java types is given below in Table 4-2.

Primitive Types	Default Java Type
Boolean	boolean
Double	double

TABLE 4-2 Default mapping of primitive types to Java types.

Primitive Types	Default Java Type
Float	float
Integer	int
Long	long
String	java.lang.String

TABLE 4-2 Default mapping of primitive types to Java types.

4.4 Exception Framework

In JMI, exceptions are raised in a variety of interfaces. These include the Reflective interfaces and the specific interfaces produced by the MOF to Java language mapping templates. Exceptions raised by a JMI service fall into two categories:

- *Modeled Exceptions* —These exceptions appear in the generated APIs as a result of them being modeled explicitly in the metamodel. The JMI service is not expected to know when to raise the modeled exceptions. Instead, the implementation of the APIs provided by the user need to explicitly raise these exceptions when the intended exception condition has occurred. All modeled exceptions extend the javax.jmi.reflect.RefException (which extends java.lang.Exception), and are designed to be checked exceptions.
- *Service Exceptions* —These exceptions represent exceptions that are raised by the JMI service as a result of some exception condition within the service. All service exceptions extend the javax.jmi.reflect.JmiException (which extends java.lang.RuntimeException), and are designed to be unchecked exceptions.

Note – For more details on modeled and service exceptions, see “The Exception Framework” on page 105.

4.5 Preconditions for Java Interface Generation

The Java mapping may not produce valid Java interfaces if any of the following preconditions on the input metamodel is not satisfied. Note that the following is not a complete list of preconditions:

- The MOF Model constraints must all be satisfied for the input metamodel.
- The input metamodel must be structurally consistent.
- The visible names within a namespace must conform to the standard Java identifier syntax, or they must have a valid Java substitute name.
- The visible names within a namespace must be unique after name substitution and other name mangling specified in the mapping.

- A class may not be nested within another class.
- A class may not be imported.
- Model Elements in a metamodel cannot be cyclically dependent except as follows:
  - A dependency cycle consisting of one or more classes is legal, provided they all have the same container.
  - A dependency cycle consisting of one or more classes and one or more data types or exceptions, is legal provided they all have the same container.

4.6 Standard Tags for the Java Mapping

This section defines the standard tags that apply to the MOF to Java mapping. Other tags may be attached to the elements of a metamodel, but the meaning of those tags are not specified. Similarly, this section does not specify the meaning of the tags outside the context of the MOF to Java mapping.

All standard tag identifiers for the Java language mapping start with the prefix string:

"javax.jmi."

TABLE 4-3 shows the conventions used to describe the standard tags and their properties.

Tag	Properties
tag id:	A string that denotes the semantic category for the tag.
attaches to:	Gives the kind(s) of Model.ModelElement that this category of tag can be meaningfully attached to.
values:	Gives the number and types of the tag's values (i.e. parameters), if any. (Tag parameters are expressed as an unordered collection of java.Object values.)
meaning:	Describes the meaning of the tag in this context.
Java interface generation:	Defines the tag's impact on the generated Java interfaces.
restrictions:	Tag usage restrictions; e.g. "at most one tag of this kind per element", or "tag must be contained by the metamodel".

TABLE 4-3 Notation for describing Standard Tags

Note – All tags attached to a metamodel element that are contained within that extent must be considered in the Java mapping.

4.6.1 Tag for Specifying Package Prefix

This tag allows the metamodeler to specify a prefix for the package name generated by the Java mapping.

Package Prefix

tag id:	“javax.jmi.packagePrefix”
attaches to:	Model.Package
values:	a String
meaning:	This tag supplies prefix to be added to the package name.
Java interface generation:	A prefix is applied to the package heading for the interfaces in the package.
restrictions:	A Prefix tag should only be attached to a non-nested Package.

4.6.2 Tag for Providing Substitute Identifiers

There are some situations when the Java identifiers produced by the Java mapping templates will result in name collisions. The following tag allows a metamodeler to provide a substitute for a model element’s name that will be used in the Java interface generation.

tag id:	“javax.jmi.substituteName”
attaches to:	Model.ModelElement
values:	a String
meaning:	The String is the substitute name to be used in place of the model element’s name.
java interface generation:	Wherever the Java mapping makes use of a model element’s name, the substitute name should be used in its place. This substitution occurs before applying any name mangling rules.
restrictions:	The preconditions defined in “Preconditions for Java Interface Generation” on page 49 apply to the substitute name; i.e. [1] it must be a syntactically valid Java identifier, and [2] all identifiers produced from it must be unique in their respective scopes after formatting and name mangling, as per the Java mapping specification. In addition, [3] there should be at most one substitute name tag per ModelElement.

Note – The javax.jmi.model.Class has the javax.jmi.substituteName tag set to “MofClass” so that the generated operation for the class proxy for javax.jmi.model.Class in javax.jmi.model.ModelPackage is not named “getClass” (as getClass is already defined in java.lang.Object).

4.6.3 Tag for specifying prefix for generated methods

This tag allows the modeler to provide a prefix that is applied to the names of all generated methods within a package. This tag applied to packages (i.e., instances of MOF Package). The default value is the empty string.

tag id:	“javax.jmi.methodPrefix”
attaches to:	Model.Package
values:	a string. Default value is the empty string.
meaning:	This tag supplies a user defined prefix for the generated operations for a model (package)
java interface generation:	The prefix is applied to the names of all generated operations of that package. The non-empty string tag of a nested package overrides the tag value of the containing package. The tag does not apply to modeled operations.
restrictions:	The tag value should be a valid prefix for a Java operation name.

4.6.4 Tag for controlling Lifecycle API generation

This tag which applies to packages (i.e., instances of MOF Package) is used to control the Java API generation. That is, when javax.jmi.ignoreLifecycle=true, all APIs dealing with the lifecycle aspects are suppressed. To determine the full impact of the javax.jmi.ignoreLifecycle tag, see Section 4.8, “Java Mapping Templates”.

tag id:	“javax.jmi.ignoreLifecycle”
attaches to:	Model.Package
values:	a string (true/false)
meaning:	Specifies whether lifecycle related APIs are to be generated.
restrictions:	Value must be either “true” or “false”

Note – When javax.jmi.ignoreLifecycle=true, not only does this suppress the generated APIs dealing with lifecycle semantics, it also affects the respective JMI reflective methods. For example, when javax.jmi.ignoreLifecycle=true the 'RefObject.refDelete()' method is required to throw java.lang.UnsupportedOperationException.

4.7 Java Generation Rules

During the design of the MOF to Java mapping, several design decisions were made which are explained in this section.

4.7.1 Rules for Splitting MOF Model.ModelElement Names into Words

In the MOF, the name of a ModelElement is, typically, formed from one or more words in some natural language and is an instance of the "NameType" i.e., any valid UTF-16 string. Since the MOF does not restrict the set of strings that can be used to name a ModelElement, not all ModelElement names transform directly to valid Java identifiers. For example, names that include graphic characters do not map to Java identifiers.

However, ModelElement names (or the javax.jmi.substituteName if one is provided) consisting only of ASCII letters, digits, hyphens ('-'), underscores ('_') and white-space characters, and conform to the following syntax are subject to a JMI identifier generation pattern that will result in standardized Java identifiers being generated.

MOF identifier syntax subject to JMI identifier mapping:

Let "ALPHA" be the set of upper-case alphabetic characters;

Let "alpha" be the set of lower-case alphabetic characters;

Let "num" be the set of numeric characters.

Then:

word ::= [ALPHA] [ALPHA | num]* [alpha | num]* | [alpha] [alpha | num]*

white-space::= SP, CR, LF, HT, VT

non-sig ::= { '_' | '-' | white-space }*

MOF-identifier ::= [non-sig] word { non-sig word }* [non-sig]

The above syntax defines a heuristic for splitting names into a sequence of words. The Java identifier is generated by reassembling the words as defined in Section 4.7.2, “Rules for Generating Identifiers”. The "non-sig" characters are non-significant to the Java identifier generation rules and are discarded.

Note – In order to generate a complete JMI API, all ModelElement names (or the javax.jmi.substituteName if one is provided) must conform to the above identifier syntax.

4.7.2 Rules for Generating Identifiers

Identifier naming is an important issue for automatically generated Java interfaces, especially when the interfaces are intended to be used by applications written by human programmers. The mapping has to reach a balance between conflicting requirements:

- Syntactic correctness — all identifiers in the mapped Java must conform to the Java syntax.
- User friendliness — identifiers should convey as much information as possible without being overly long.
- Conformance to existing conventions — identifiers should conform to existing stylistic conventions.

In JMI, all identifiers (i.e., names of packages, names of class proxies, names of operations, names of parameters, names of constants, and names enumeration literals) shall conform to the following rules:

Package names:	The identifier consists of lower-case alphabetic characters only.
Class proxy names:	The identifier consists of lower-case alphabetic characters with the following exceptions.The first letter of the identifier is capitalized. If the identifier consists of multiple words, the first letter of each word in the identifier is capitalized.
Operation names:	The identifier consists of lower-case alphabetic characters with the following exception. If the identifier consists of multiple words, the first letter of each word except the first word, is capitalized.
Attribute names:	The identifier consists of lower-case alphabetic characters with the following exception. If the identifier consists of multiple words, the first letter of each word except the first word, is capitalized.
Constants	The identifier consists of all upper-case alphabetic characters and the “_” character (used to separate words).
Enumeration literals	The identifier consists of all upper-case alphabetic characters and the “_” character (used to separate words).

4.7.3 Literal String Values

Literal string values (in string valued Constants) are not re-formatted and appear in the generated Java exactly as specified by the Constant's "value" attribute.

4.7.4 Generation Rules for Attributes, AssociationEnds, References, Constants, and Parameters

The MOF Model allows attributes, association ends, references and parameters to be single-, optional- or multivalued depending on the ModelElement's base type (i.e., the modeled MOF type) and its multiplicity. In addition, parameters are one of four "DirectionKinds" — IN_DIR, OUT_DIR, INOUT_DIR, or RETURN_DIR — depending on whether the parameter value is to be passed in and/or out, of returned as the result of the operation.

In the mapped interfaces, the multiplicity makes it necessary to pass the Java "Object" representation for the optional case, and pass collections of values for the multivalued case. The "direction" makes it necessary to pass arrays of values for the OUT_DIR and INOUT_DIR cases as in Java, parameters cannot return values.

This section defines the rules for generating the types of attributes, association ends, references and parameters.

Generation Rules for Attributes, AssociationEnds, and References

The type of a an attribute, assocationend, or reference is derived from its modeled type and its multiplicity. TABLE 4-4 defines the generation rules for attributes, association ends, and references:

Multiplicity	Generation rules for PrimitiveType	Generation rules for all other types
0..1	Java object type	Corresponding Java interface
1..1	If the type is a Java primitive type with scalar and object representations, then Java scalar type; else, Java object type.	Corresponding Java interface
other	Collection of Java object type	Collection of corresponding Java interface

TABLE 4-4 Generation rules for Attributes, AssociationEnds, and References

Generation Rules for Constants

The type of a constant is derived from its modeled type. The type of a constant must be some primitive type.

Generation Rules for Parameters

The type of a parameter is derived from its base type, its multiplicity, and its direction attributes. TABLE 4-5 defines the generation rules for parameters whose modeled type is a MOF primitive type. TABLE 4-6 defines the rules for parameters whose modeled type is any type other than a MOF primitive type.

Multiplicity	IN_DIR & RETURN_DIR	OUT_DIR and INOUT_DIR
0..1	Java object type	Array of object type
1..1	If the type is a Java primitive type with scalar and object representations, then Java scalar type; else, Java object type.	If the type is a Java primitive type with scalar and object representations, then array of Java scalar type; else, array of Java object type.
other	Collection of Java object type	Array collection of Java object type

TABLE 4-5 Generation rules for MOF PrimitiveType parameters.

Multiplicity	IN_DIR & RETURN_DIR	OUT_DIR and INOUT_DIR
0..1	Corresponding Java interface	Array of corresponding Java interface
1..1	Corresponding Java interface	Array of corresponding Java interface.
other	Collection of corresponding Java interface	Array collection of corresponding Java interface

TABLE 4-6 Generation rules for parameters that are not of MOF PrimitiveType.

4.8 Java Mapping Templates

Model specific Java interfaces are produced by traversing the containment hierarchy of an outermost package extent. The Java inheritance hierarchy of the resulting interfaces directly reflects the containment hierarchy of the source package.

The mapping rules are described in terms of Java interfaces. Each Java interface describes the full list of operations which could be generated when mapping MOF Model objects. In any specific case, the actual operations generated will depend on the properties of the corresponding MOF Model object.

The Template approach used here is a notational convenience, not a required or suggested implementation strategy.

4.8.1 Package Interface Template

A package interface is named <*PackageName*>**Package** and it contains operations that provide the dependent Package, Association and Class proxy objects for a Package object. This interface is not generated if the ignoreLifecycle tag (attached to the package containing this metamodel) is set to true.

Template

```
// if the ignoreLifecycle tag is not true, generate interface
<<ANNOTATION TEMPLATE>>
public interface <packageName>Package
// if Package has no super-Packages
extends javax.jmi.reflect.RefPackage
// else for each public super-Package (in order)
extends <superPackage>Package,... {

// for each imported package where:
//    isClustered == true and
//    Import.visibility == public_vis and
//    importedNamespace.visibility == public_vis
public <ClusteredPackageName>Package get<ImportName>();
// for each contained package where visibility = public_vis
public <NestedPackageName>Package get<NestedPackageName>();
// for each contained class visibility = public_vis
public <ClassName>Class get<ClassName>();
// for each contained association visibility = public_vis
public <AssociationName> get<AssociationName>();
// for each StructType directly contained by the package
public <StructTypeName> create<StructTypeName>(/*for each attribute
<AttributeType> <attributeName>*/) throws javax.jmi.reflect.JmiExcep-
tion;
};
```

Supertypes

If the M2-level Package inherits from other M2-level Packages with “visibility” of “public_vis”, the Package interface extends the interfaces for the supertype M2-level Packages. Otherwise, the Package interface extends **javax.jmi.reflect.RefPackage**.

Operations

t get<*ImportName*>()

An operation is generated for each clustered package in the current package.

reflective analog:

return type:	<*ClusteredPackageName*>**Package**
parameters:	none
exceptions	none

t get<*NestedPackageName*>()

An operation is generated for each nested package in the current package.

reflective analog:

return type:	<*NestedPackageName*>**Package**
parameters:	none
exceptions	none

t get<*ClassName*>()

An operation is generated for each class in the current package.

reflective analog:

return type:	<*ClassName*>**Class**
parameters:	none
exceptions	none

t get<*AssociationName*>()

An operation is generated for each association in the current package.

reflective analog:

return type:	<*AssociationName*>
parameters:	none
exceptions:	none

t create<*StructTypeName*>()

A create operation is generated for each StructType defined in the current package.

reflective analog:

return type:	<*StructTypeName*>
parameters:	one parameter for each attribute.
exceptions:	JmiException

4.8.2 Class Proxy Template

The class proxy template defines the Java interface generation rules for the <*className*>**Class** interface for a class whose “visibility” is “public_vis”. This interface has operations for any classifier-scoped attributes, constants, operations, along with a factory operation that give access to its instance objects. This interface is not generated if the ignoreLifecycle tag (attached to the package containing this metamodel) is set to true.

The class proxy interface declaration appears in the immediate package containing the class.

Template

```
// if the ignoreLifecycle tag is not true, generate interface
public interface <ClassName>Class
extends javax.jmi.reflect.RefClass {
//If isAbstract is not set to true, generate factory methods.
public <ClassName> create<ClassName> () throws javax.jmi.reflect.JmiEx-
ception;
public <ClassName> create<ClassName> (/* for each non-derived direct or
inherited instance-level attribute <AttributeType> <attributeName> */)
throws javax.jmi.reflect.JmiException;
// for each StructType directly contained by this class
public <StructTypeName> create<StructTypeName>(/*for each attribute
<AttributeType> <attributeName> */) throws javax.jmi.reflect.JmiExcep-
tion;
// for each attribute and operation contained in this class or one of its
supertypes with “classifier-level” scope, generate the appropriate oper-
ations
<<ATTRIBUTE TEMPLATE>>
<<OPERATION TEMPLATE>>
}; // end of interface <ClassName>Class
```

Operations

t create<*ClassName*>() taking no arguments

The create<*ClassName*> operation that takes no arguments is the default factory operation used to create instance object

reflective analog:	refCreateInstance(...)
return type:	<*className*>
parameters:	none
exceptions	JmiException (AlreadyExistsException, ClosureViolationException)

t create<*ClassName*>()

The create<ClassName> operation is the factory operation used to create instance objects

reflective analog:	refCreateInstance(...)
return type:	<className>
parameters:	in <AttrType> <attrName>,...
exceptions	JmiException (AlreadyExistsException, ClosureViolationException)

The parameters to this create operation provide initial values for the class's non-derived instance-level attributes. Parameter declarations are generated in an order defined by a recursive depth-first traversal of the inheritance graph. More precisely:

- A class's superclasses are processed before the class's attributes.
- Superclasses are processed in the order of the "Generalizes" association.
- The attributes of each class or superclass are processed in the order of the "Contains" association.
- When an attribute is encountered with a "scope" value of "classifier_level" or an "isDerived" value of true no parameter is generated.
- When an attribute is encountered a second or subsequent time, no additional parameter is generated.
- When the attribute multiplicity is not [1..1], the <*AttributeType*> has an appropriate type as specified the section titled "Generation Rules for Parameters" on page 56.

t create<*StructTypeName*>()

A create operation is generated for each StructType defined in this class.

reflective analog:	
return type:	*<StructTypeName>*
parameters:	one parameter for each attribute.
exceptions:	JmiException

4.8.3 Instance Template

The instance template defines the Java generation rules for the *<className>* interface for a class whose visibility is “public_vis”. This interface contains operations for the classes instance-scoped attributes and operations, along with any references.

The instance interface declaration appears in the same package as it’s class proxy declaration.

Template

```
<<ANNOTATION TEMPLATE>>
public interface <className>
// If the class has no super-types
extends javax.jmi.reflect.RefObject {
// else, for each super-class
extends <superClassName>, ...{
// for each constant, generate the appropriate code
<<CONSTANT TEMPLATE>>
// for each instance level attribute, reference, operation
// contained in this class, generate the appropriate code
<<ATTRIBUTE TEMPLATE>>
<<REFERENCE TEMPLATE>> // public only
<<OPERATION TEMPLATE>> // public_vis only
}; // end of interface <ClassName>
```

Supertypes

The instance interface for an class extends the instance interfaces for all of its superclasses.

4.8.4 Association Template

The association template defines the generation rules for the association interface corresponding to an association whose “visibility” is “public_vis”. This interface contains the interface operations for accessing and updating the association's link set. This interface is not generated if the ignoreLifecycle tag (attached to the package containing this metamodel) is set to true.

Template

```
// if the ignoreLifecycle tag is not true, generate interface
public interface <AssociationName> extends javax.jmi.reflect.RefAssocia-
tion {
public boolean exists(<AssociationEnd1ClassName> <AssocationEnd1Name>,
AssociationEnd2ClassName> <AssocationEnd2Name>) throws
javax.jmi.reflect.JmiException;
//If associationEnd1 is single valued and isNavigable
public <AssociationEnd1ClassName> get<AssociationEnd1Name>
(ssociationEnd2ClassName> <AssocationEnd2Name>) throws
javax.jmi.reflect.JmiException;
//If associationEnd1 is multivalued, isOrdered is false, and isNavigable
public Collection get<AssociationEnd1Name> (ssociationEnd2ClassName>
<AssocationEnd2Name>) throws javax.jmi.reflect.JmiException;
//If associationEnd1 is multivalued, isOrdered, and isNavigable
public List get<AssociationEnd1Name> (ssociationEnd2ClassName>
<AssocationEnd2Name>) throws javax.jmi.reflect.JmiException;
//If associationEnd2 is single valued and isNavigable
public <AssociationEnd2ClassName> get<AssociationEnd2Name>
(ssociationEnd1ClassName> <AssocationEnd1Name>) throws
javax.jmi.reflect.JmiException;
//If associationEnd2 is multivalued, isOrdered is false, and isNavigable
public Collection get<AssociationEnd2Name> (ssociationEnd1ClassName>
<AssocationEnd1Name>) throws javax.jmi.reflect.JmiException;
//If associationEnd2 is multivalued, isOrdered, and isNavigable
public List get<AssociationEnd2Name> (ssociationEnd1ClassName>
<AssocationEnd1Name>) throws javax.jmi.reflect.JmiException;
//If associationEnd1 and associationEnd2 isChangable
public boolean add(AssociationEnd1ClassName> <AssocationEnd1Name>,
AssociationEnd2ClassName> <AssocationEnd2Name>) throws
javax.jmi.reflect.JmiException;
//If associationEnd1 and associationEnd2 isChangable
public boolean remove(<AssociationEnd1ClassName> <AssocationEnd1Name>,
AssociationEnd2ClassName> <AssocationEnd2Name>) throws
javax.jmi.reflect.JmiException;
};
```

Operations

t exists

The “exists” operation queries whether a link currently exists between a given pair of instance objects association’s link set.

reflective analog:	refLinkExists(...);
return type:	boolean
parameters:	*<AssocEnd1ClassName> <assocEnd1Name>* *<AssocEnd2ClassName> <assocEnd2Name>*
query:	yes
exceptions:	JmiException (InvalidObjectException)

The parameters to the “exists” operation are a pair of instance values of the appropriate type for the association. Since MOF link relationships are implicitly directional, the order of the parameters is significant.

t **get***<associationEnd1Name>*

The “get*<associationEnd1Name>*” operation queries the instance object or objects that are related to a particular instance object by a link in the current association’s link set.

reflective analog:	refQuery(...)
return type:	<AssociationEnd1ClassName>, Collection, or List
parameters:	in *<AssocEnd2ClassName> <assocEnd2Name>*
query:	yes
exceptions:	JmiException (InvalidObjectException)

Note – The result type of the operation depends on the multiplicity of *<AssociationEnd1>*. If it has bounds of [0..1] or [1..1], the result type is the instance type corresponding to the association end type. Otherwise, it is a collection of the association ends type.

---

t **get***<associationEnd2Name>*

This operation is the equivalent of get*<AssociationEnd1Name>*, with the ends interchanged.

t add

The add operation creates a link between the pair of instance objects in this association's link set.

reflective analog:	refAddLink(...);
return type:	boolean
parameters:	in <*AssocEnd1ClassName*> <*assocEnd1Name*> in <*AssocEnd2ClassName*> <*assocEnd2Name*>
exceptions:	JmiException (InvalidObjectException, DuplicateException, ClosureViolationException, CompositionCycleException)

The two parameters to the" add" operation give the instance objects at the two ends of the new link.

If one or other end of the association has "isOrdered" set to true, the new link must be added so that it is the last member of the projection for the ordered association end.

If the new link cannot be added for any reason other than that it already contains the link, an exception must be thrown (rather than returning false).

ClosureViolationException is raised when either the reference closure rule of composition closure rule is violated.

CompositionCycleViolation occurs when adding the new link would create a cycle of composite / component relationships such that one of the instance object parameters is a component of itself.

t remove

The "remove" operation removes a link between a pair of instance objects in the current association's link set.

reflective analog:	refRemoveLink(...)
return type:	boolean
parameters:	<*AssocEnd1ClassName*> <*assocEnd1Name*> <AssocEnd2ClassName> <assocEnd2Name>
exceptions:	JmiException

The two parameters to this operation give the instance objects at both ends of the link that is to be removed from the current association's link set.

If either AssociationEnd1 or AssociationEnd2 has "isOrdered" set to true, the "remove" operation must preserve the ordering of the remaining members of the corresponding projection.

Returns true if the link was successfully removed.

Collection Operations on multivalued AssociationEnds

For multivalued association ends, the operations in the java.util.Collection (and java.util.List) are used to modify the ends. In JMI some of these operations have additional semantics and may raise exceptions other than those defined in the Collection interface. The additional exception conditions for collection operations on multivalued association ends are defined below:

- NullPointerException must be thrown if a collection operation attempts to add or set a value of a multivalued association end to null.
- TypeMismatchException must be thrown if a collection operation attempts to add or set a value of a multivalued association end to an object of the wrong type.
- WrongSizeException must be thrown if the operation will result in the upper or lower bound being violated.
- ClosureViolationException and CompositionCycleException are only possible when the association has composite aggregation semantics.
  - ClosureViolationException must be thrown when a collection operation attempts to add an object that belongs to a different outermost package extent.
  - CompositionCycleException must be thrown when a collection operation would result in the object becoming a direct or indirect component of itself.

4.8.5 Attribute Template

The attribute template defines the generation rules for accessor and mutator operations for attributes whose visibility is public_vis. These operations appear on different interfaces, depending on the attribute's scope:

- Operations for instance-scoped attributes appear in the instance interface only.
- Operations for classifier-scoped attributes appear in the class proxy interface only.

The operations generated for an attribute and their signatures depend heavily on the attribute's properties. For the purposes of defining the generated Java code, attribute multiplicities fall into three groups:

- Single-valued attributes: multiplicity bounds are [1..1].
- Optional-valued attributes: multiplicity bounds are [0..1].
- Multivalued attributes: any other multiplicity.

In order to follow established Java naming patterns, the names of accessor and mutator methods for an attribute will be generated as follows:

Accessor Operations

For single-valued and optional-valued Boolean attributes whose <*AttributeName*> has as a prefix the word "is":

```
<AccessorName> = <AttributeName>
```

For single-valued and optional-valued Boolean attributes whose <*AttributeName*> does not have as a prefix the word "is":

```
<AccessorName> = is<AttributeName>
```

For all other (non-Boolean) single-valued and optional-valued attributes:

```
<AccessorName> = get<AttributeName>
```

Mutator Operations

For single-valued and optional-valued Boolean attributes whose <AttributeName> has as a prefix the word “is”

```
<MutatorName> = set<AttributeNameWithoutPrefixIs>
```

For all other (non-Boolean, and Boolean attributes whose names are not prefixed with the word “is”) single-valued and optional-valued attributes:

```
<MutatorName> = set<AttributeName>
```

Template

```
// if Attribute visibility is private or protected no Java code is
generated
<<ANNOTATION TEMPLATE>>
// Accessor Operations
// if optional-valued attribute
public <AttributeType> <AccessorName> () throws javax.jmi.reflect.JmiEx-
ception;
// if single-valued attribute
public <AttributeType> <AccessorName> () throws javax.jmi.reflect.JmiEx-
ception;
// if upper > 1 and isOrdered = false
public Collection get<AttributeName> () throws javax.jmi.reflect.JmiEx-
ception;
// if upper > 1 and isOrdered = true
public List get<AttributeName> () throws javax.jmi.reflect.JmiException;
// Mutator Operations
// if optional or single-valued, and isChangeable = true
public void <MutatorName> (<AttributeType> newValue) throws
javax.jmi.reflect.JmiException;
```

Operations

t *<AccessorName>*

The <*AccessorName*> (e.g., get<AttributeName>) operation returns the value of the named attribute.

reflective analog:	refGetValue(...);
return type:	[0..1], [1..1] - <*AttributeType*> [0..N] for N>1 and isOrdered = false - java.util.Collection [0..N] for N>1 and isOrdered = true- java.util.List
parameters:	none
query:	yes
exceptions:	JmiException

The signature of the <*AccessorName*> operation depends on the attribute's multiplicity as indicated above. Its behavior is as follows:

- In the [0..1] case, the operation returns either the attributes optional value or null. In the [1..1] case, the operation simply returns the attribute's single value.
- In other cases, the operation returns a Collection (or List, if the multivalued attribute is ordered). In the case where the collection is empty the result value will be an Collection C such that C.isEmply() = true. No exception is raised in this case.

t *<MutatorName>*

The "<*MutatorName*>" (e.g., set<AttributeName>) operation sets the value of the named attribute. This operation is not generated for mutivalued attributes.

reflective analog:	refSetValue(...);
return type:	none
parameters:	<*AttributeType*> newValue
exceptions:	JmiException (DuplicateException, InvalidObjectException, ClosureViolationException, CompositionCycleException)

ClosureViolationException and CompositionCycleException are only possible when the type of the attribute is a class, and the attribute has "composite" aggregation semantics

ClosureViolationException occurs when "newValue" or one of its members (in the multivalued case) belongs to a different outermost package extent to this object.

CompositionCycleViolation occurs when the operation would result in this object having itself as a direct or indirect component.

InvalidObjectException occurs when some instance object is found to be non-existent or inaccessible.

Collection Operations on multivalued Attributes

For multivalued attributes the operations in the java.util.Collection (and java.util.List) are used to modify the attribute. In JMI some of these operations have additional semantics and may raise exceptions other than those defined in the Collection interface. The additional exception conditions for collection operations on multivalued attributes are defined below:

- NullPointerException must be thrown if a collection operation attempts to add or set a value of a multivalued attribute to null.
- TypeMismatchException must be thrown if a collection operation attempts to add or set a value of a multivalued attribute to an object of the wrong type.
- WrongSizeException must be thrown if the operation will result in the upper or lower bound being violated.
- ClosureViolationException and CompositionCycleException are only possible when the type of the attribute is a class, and the attribute has "composite" aggregation semantics.
  - ClosureViolationException must be thrown when a collection operation attempts to add or set a value of a multivalued attribute to an object that belongs to a different outermost package extent.
  - CompositionCycleException must be thrown when a collection operation would result in the object becoming a direct or indirect component of itself.

4.8.6 Reference Template

The reference template defines the Java generation rules for a reference whose “visibility” is “public_vis”. The Java code generated for a reference is declared in the instance object interface definition. The Java code generated by the reference template provides the operations to return the value of the reference as well as operations to modify it. The code generated is dependent upon the multiplicity, mutability, and ordering of the specified reference.

The operations generated for a reference and their signatures depend heavily on the properties of the referenced association end which are also mirrored on the reference itself. For the purposes of defining the generated Java code, reference multiplicities fall into three groups:

- Single-valued references: multiplicity bounds are [1..1].
- Optional-valued references: multiplicity bounds are [0..1].
- Multivalued references: any other multiplicity.

The generated operations for a reference are designed to have similar signatures and behaviors to those for an instance-scoped attribute with the same multiplicity and changeability settings.

Note – A reference is “well formed” only if the referenced association end has “isNavigable” set to true. Similarly, a reference’s “isChangeable” can be true only if the referenced association end’s “isChangeable” is also true.

Template

```
// If the Reference has visibility of protected or private, no // Java is
generated
```

```
<<ANNOTATION TEMPLATE>>
// operations to return the Reference value
// if upper = 1
public <ReferenceClass> get<ReferenceName> () throws
javax.jmi.reflect.JmiException;
// if upper > 1 and isOrdered = false
public Collection get<ReferenceName> () throws javax.jmi.reflect.JmiEx-
ception;
// if upper > 1 and isOrdered = true
public List get<ReferenceName> () throws javax.jmi.reflect.JmiException;
// if upper = 1 and isChangeable
public void set<ReferenceName> (ReferenceClass> newValue) throws
javax.jmi.reflect.JmiException;
```

Operations

t get<*ReferenceName*>

The "get<*ReferenceName*>" operation returns the value of reference. The signature of the operation depends on the multiplicity of the Reference.

reflective analog:	refGetValue(...);
return type:	[0..1], [1..1] - <*ReferenceClass*> [0..N] for N>1 and isOrdered = false - java.util.Collection [0..N] for N>1 and isOrdered = true- java.util.List
parameters:	none
exceptions:	JmiException

The "get<*ReferenceName*>" operation's signature is determined by the multiplicity of the reference (i.e., the multiplicity of the referenced association end).

The operation calculates and returns the projection of the respective association end's link set as follows:

- In the [0..1] case, the operation returns the projected instance object if there is one; else it returns "null".
- In the [1..1] case, the operation normally returns a single instance object. However, if the projection contains no elements, this is signalled as a JmiException.
- In all other cases, the operation returns a collection. If the projection is empty the result is an empty collection.

Note – Under no circumstances should the "<*ReferenceName*>" operation return a collection that includes a null object reference.

---

t set<*ReferenceName*>

The “set<*ReferenceName*>” operation assigns a new value to a reference. This operation is not generated for multivalued references.

reflective analog:	refSetValue(...)
return type:	none
parameters:	in <*ReferenceClass*> newValue
exceptions:	JmiException (InvalidObjectException, DuplicateException, InvalidObjectException, ClosureViolationException, CompositionCycleException)

The “set<*ReferenceName*>” operation replaces the referenced association end.

InvalidObjectException occurs if any of the supplied instance objects is a non-existent, null or inaccessible instance object.

ClosureViolationException occurs when “newValue” belongs in a different outermost extent to “this” object.

CompositionCycleViolation occurs when the referenced association has composite aggregation semantics, and the update would make “this” object a component of itself.

Collection Operations on References

For multivalued references the operations in the java.util.Collection (and java.util.List) are used to modify the reference. In JMI some of these operations have additional semantics and may raise exceptions other than those defined in the Collection (or List) interface. The additional exception conditions for collection operations on multivalued attributes are defined below:

- NullPointerException must be thrown if a collection operation attempts to add or set a value of a multivalued reference to null.
- TypeMismatchException must be thrown if a collection operation attempts to add or set a value of a multivalued reference to an object of the wrong type.
- WrongSizeException must be thrown if the operation will result in the upper or lower bound being violated.
- ClosureViolationException and CompositionCycleException are only possible when the referenced association has composite aggregation semantics.
  - ClosureViolationException must be thrown when a collection operation attempts to add an object that belongs to a different outermost package extent.
  - CompositionCycleException must be thrown when a collection operation would result in the object becoming a direct or indirect component of itself.

Note – The “remove” operations should be able to cope with removal of a link when the object at the other end of a link is non-existent or inaccessible.

4.8.7 Operation Template

The operation template defines the Java generation rules for operations whose “visibility” is “public_vis”. It generates a Java interface operation within the scope of an instance object interface or class proxy interface, depending on the scope of the operation.

Template

```
// If the Operation has visibility of protected or private, no
// Java is generated
<<ANNOTATION TEMPLATE>>
// The <GeneratedReturnType> and <GeneratedParamType> should conform to
“Generation Rules for Parameter Type“
// if Operation contains no “return” Parameter
public void <OperationName>(
// else
public <GeneratedReturnType> <OperationName>(
// for each contained “in”, “out” or “inout” Parameter
<GeneratedParamType> <param_name>,...
) throws
// for each Exception raised by the Operation
<ExceptionName>,..., javax.jmi.reflect.JmiException;
```

t <*OperationName*>

An “<*OperationName*>” operation invokes an implementation specific method to perform the behavior implied by the operation model element.

reflective analog:	refInvokeOperation(...)
return type:	<*GeneratedReturnType*> <param_name>
parameters:	<*GeneratedParamType*> <*ParameterName*>,...
exceptions:	Modeled exceptions (if any). JmiException (DuplicateException, InvalidObjectException)

An “OperationName” operation invokes an implementation specific method. While the behavior of the method itself is beyond the scope of the Generated Java Interface mapping, the signature of the generated operation is defined by the mapping, along with some parameter checking semantics.

The return type for an “OperationName” operation is generated from the operation’s (optional) return parameter; i.e. the contained parameter object whose “direction” attribute has the value “RETURN_DIR”. The return type is as defined in the section “Generation Rules for Parameters” on page 56.

For each non-return parameter of the operation, in the defined order, the “OperationName” declaration has a parameter declaration as follows:

The parameter type is as defined in the section “Generation Rules for Parameters” on page 56.

The <*ParameterName*> is produced by rendering the parameter’s name from the definition.

The list of exceptions raised by an “<*OperationName*>” operation is generated from the operation’s modeled exceptions, followed by the javax.jmi.reflect.JmiException.

While modeled exceptions should be signalled by raising exceptions corresponding to the operation’s exceptions list, JmiException is used to signal the following structural errors relating to the values supplied by the caller for “in” and “inout” parameters.

- DuplicateException occurs when a multivalued parameter has “isUnique” set to true, and the supplied collection contains a duplicate.
- InvalidObjectException can occur if an instance object typed parameter value or element is a reference to a non-existent (i.e. deleted) or inaccessible object.

Like all other operations that have JmiException in their signature, an “<*OperationName*>” operation can use JmiException to signal constraint errors and semantic errors as well.

4.8.8 Exception Template

The exception declaration appears in the immediate package containing the exception.

In order to follow established Java naming patterns, the names of modeled exceptions will be generated as follows:

If the name of a modeled exception ends with 'Exception' then:

<GenExceptionName> = <ExceptionName>

Else,

<GenExceptionName> = <ExceptionName>**Exception**

```
<<ANNOTATION TEMPLATE>>
public class <GenExceptionName> extends java.jmi.reflect.RefException {
   // For each parameter
   private final <GeneratedParameterType> <parameterName>;
   // constructor
   public <ExceptionName> (/* for each parameter <GeneratedParameter-
   Type> <parameterName>, ... */){
     super();
   // for each parameter
   this.<ParameterName> = <parameterName>;
   }

   // for each parameter
   public <GeneratedParameterType> <AccessorName>() {
     return this.<parameterName>;
   }
 }
```

Note – *<GeneratedParameterType>* conforms to the rules defined in the section “Generation Rules for Parameters” on page 56. <AccessorName> conforms to the name generation pattern for attributes (see “Head3” on page 65).

4.8.9 Constant Template

```
<<ANNOTATION TEMPLATE>>
public final <ConstantType> <ConstantName> = <ConstantValue>;
```

4.8.10 AliasType Template

Instances of AliasType get mapped to the actual type that is being aliased. For example, if you define an AliasType XAlias to StructureType XStruct, then any attribute of type XAlias gets mapped to XStruct.

Note – If an AliasType that is an alias to an object is used as the type of an attribute, the composition semantics do not apply to that attribute.

4.8.11 CollectionType Template

All CollectionTypes are mapped to java.util.Collection or java.util.List.

Note – Composition semantics do not apply to members of a collection.

4.8.12 StructureType Template

The StructType interface declaration appears in the immediate package containing the structure type.

```
<<ANNOTATION TEMPLATE>>
public interface <StructName> extends javax.jmi.reflect.RefStruct {
  // For each attribute
  public <AttributeType> <AccessorName>()throws
    javax.jmi.reflect.JmiException;
}
```

Note – <AccessorName> conforms to the name generation pattern for attributes (see “Head3” on page 65).

Design Considerations: *The JMI expert group considered an alternate mapping for structs, i.e., one that generated classes. In order to support access to the metadata from the instance objects in an implementation independent manner, it was necessary to define interfaces instead.*

4.8.13 EnumerationType Templates

For each enumeration type defined in the metamodel, JMI generates an interface and an implementation class. The interface and class definitions appear in the immediate package containing the enumeration type.

```
<<ANNOTATION TEMPLATE>>
public interface <EnumerationName> extends javax.jmi.reflect.RefEnum {
}


public final class <EnumerationName>Enum implements <EnumerationName> {
  // for each enumeration literal
  public static final <EnumerationName>Enum <LITERAL_IDENTIFIER> = new
    <EnumerationName>Enum("<literalName>");

  private static final List typeName;
  private final String literalName;

  static {
    java.util.ArrayList temp = new ArrayList();
    // for each part of the fully qualified name
    temp.add("<fullyQualifiedNamePart>");
    typeName = java.util.Collections.unmodifiableList(temp);
  }

  private <EnumerationName>Enum(String literalName) {
    this.literalName = literalName;
  }

  public String toString() {
    return literalName;
  }

  public static <EnumerationName> forName(String value) {
    //for each literal
    if (value.equals("<literalName>") return <LITERAL_IDENTIFIER>;
    throw new IllegalArgumentException("implementation specific mes-
      sage");
   }

  public List refTypeName() {
    return typeName;
  }

  public int hashCode() {
     return literalName.hashCode();
  }
```

```
  public boolean equals(Object o) {
    if (o instanceof <EnumIdentifier>Enum) return (o == this);
    else if (o instanceof <EnumIdentifier>) return
      (o.toString().equals(literalName));
    else return ((o instanceof javax.jmi.reflect.RefEnum) &&
      ((javax.jmi.reflect.RefEnum) o).refTypeName().equals(typeName) &&
      o.toString().equals(literalName));
  }

  protected Object readResolve() throws java.io.ObjectStreamException {
    try {
       return forName(literalName);
    } catch (IllegalArgumentException iae) {
       throw new java.io.InvalidObjectException(iae.getMessage());
    }
  }

}
```

Design Considerations: *The interface is included to provide type checking. by defining the static fields representing literals in the class as opposed to the interface, the generated interface is not dependent on the generated class.This is done to avoid loading of the generated class and initializing the interface fields during the class loading which is undesirable if some JMI service does not intend to use the generated class.*

4.8.14 Constraint Template

Constraints specified in Object Constraint Language (OCL) will be implemented in Java. Constraints specified in Java will appear as is.

4.8.15 Annotation Template

The Annotation template optionally generates Java comments for an M2-level ModelElement. This template should be regarded as indicative rather than normative.

Template

```
// Annotation comments may optionally be suppressed by the
// Java code generator
// The Annotation template may use either "//" or "/**/ Java
// comment format
/* <Modelelement.Annotation> */
```

CHAPTER 5

MOF Reflective Package

5.1 Introduction

One of the advantages of the metadata API is that it provides a discovery mechanism that allows a program to use objects without prior knowledge of the objects' interfaces. In the MOF context, a metaobject allows a program to “discover” the semantics of any object. With this information in hand, the MOF’s reflective interfaces allow a program to do the following without using the “tailored” (i.e., generated) interfaces:

- Create, update, access, navigate and invoke operations on class proxy objects.
- Query and update links using association objects.
- Navigate -MOF package structure.

In essence, the reflective interfaces provide the complete functionality of the tailored interfaces.

Note – The reflective interfaces do not allow a program to access or update MOF objects contrary to their metaobject descriptions. For example, they cannot be used to create, access or update attributes that do not exist.

In addition, the reflective interfaces allow the program to:

- Find an object’s metaobject.
- Find an object’s container(s) and enclosing package(s).
- Test for object identity.
- Delete a object.

The MOF reflective package contains eight "abstract" interfaces that are extended by the generated interfaces. The eight interfaces are:

1. RefBaseObject interface — provides common operations for all MOF objects except for exceptions, enumerations and structure data types.
2. RefAssociation interface — provides common operations for association objects.
3. RefPackage interface — provides common operations for package objects.
4. RefFeatured interface — provides common operations for featured objects (i.e., instance and class proxy objects).
5. RefClass object interface — provides common operations for class proxy objects.
6. RefObject interface — provides common operations for instance objects.
7. RefStruct interface — provides common operations for StructType objects.

8. RefEnum interface — provides common operations for EnumType operations.

Note – The RefPackage, RefClass, and RefAssociation interfaces will not be available when the javax.jmi.ignoreLifecycle tag (attached to the package containing the metamodel) is set to true. Also note that methods in the remaining reflective interfaces that deal with lifecycle semantics will not be implemented (i.e., the java.lang.UnsupportedOperationException will be thrown).

The MOF reflective interfaces are declared in the “javax.jmi.reflect” package.

5.2 The Reflective Classes and Interfaces

This section describes the MOF reflective interfaces. They provide the same functionality as the tailored interfaces, although there are some important differences:

Reflective operations pass the values of parameters to operations and exceptions using the reflective counterparts. On the other hand, the model specific versions of these operations pass the values using the precise types specified in the metamodel. For example, reflective operations on associations pass instance objects with the type RefObject. The model specific versions of these operations pass instance objects using their specific interfaces.

Note – For reflective operations that take a meta element as an argument, two flavors of the operation, one that takes a RefObject argument and one that takes a String argument, are provided. For the operation taking the String argument, the simple modeled name of the meta element is expected.

5.2.1 RefBaseObject

Abstract

The RefBaseObject interface is extended by all other reflective interfaces. It provides common operations for testing for object identity, returning an object's metaobject, and returning its facility container as required for implementing structural constraints such as the MOF's type closure rule and composition restrictions.

Supertypes

- None (root object)

Operations

t refMofId

The "refMofId"” operation returns this object’s permanent unique identifier string.

specific analog:	none
return type:	string
isQuery:	yes
parameters:	none
exceptions:	JmiException

Every MOF object has a permanent, unique MOF identifier associated with it. This identifier is generated and bound to the object when it is created and cannot be changed for the lifetime of the object. The primary purpose of the MOF identifier is to serve as a label that can be compared to definitively establish an object’s identity.

A MOF implementation must ensure that no two distinct MOF objects within the extent of an outermost Package object ever have the same MOF identifier. This invariant must hold for the lifetime of the extent. A group of outermost Package extents can only be safely federated if the respective implementations can ensure the above invariant applies across the entire federation. A federation of extents in which the invariant does not hold is not MOF compliant.

t refMetaObject

The "refMetaObject"” operation returns the RefObject object that describes this object in its metamodel specification.

specific analog:	none
return type:	RefObject
isQuery:	yes
parameters:	none
exceptions:	none

If the object’s metaobject is unavailable, the return value may be a Java null object reference.

t refImmediatePackage

The "refImmediatePackage"" operation returns the package object for the package that most immediately contains or aggregates this object.

specific analog:	none
return type:	RefPackage
isQuery:	yes
parameters:	none
exceptions:	none

If this object has no containing or aggregating package (i.e. it is the RefPackage object for an outermost package), then the return value is a Java null object reference. In complex cases where there is more than one immediate aggregating package, the return value may be any of them.

If the ignoreLifecycle tag (attached to the package containing the metamodel) is set to true, this method throws the java.lang.UnsupportedOperationException.

t refOutermostPackage

The "refOutermostPackage"" operation returns the package object for the package that ultimately contains this object.

specific analog:	none
return type:	RefPackage
isQuery:	yes
parameters:	none
exceptions:	none

If this object is the RefPackage object for an outermost package then the return value is this object.

If the ignoreLifecycle tag (attached to the package containing the metamodel) is set to true, this method throws the java.lang.UnsupportedOperationException.

t refVerifyConstraints

The “refVerifyConstraints” operation verifies that an object and its properties satisfy all constraints defined on it

specific analog:	none
return type:	Collection
isQuery:	yes
parameters	boolean deepVerify
exceptions:	none

When deepVerify is false (i.e., a shallowVerify), the refVerifyConstraints method checks all constraints on that object and its properties.

When deepVerify is true, the refVerifyConstraints method carries out a shallowVerify on that object and a deep verify through its containment hierarchy. If the object is a extent object (i.e., class proxy, package, or association object), the containment hierarchy includes all objects in its extent.

If no constraint is violated, the null value is returned; otherwise, a list of javax.jmi.reflect.JmiException objects (each representing a constraint violation) is returned.

The Collection returned from the refVerifyConstraints operation has copy semantics. That is, it does not reflect any changes to the source after the operation is executed, and it cannot be used to update the source.

Interface

```
package javax.jmi.reflect;

import java.util.*;
public interface RefBaseObject {
  public RefObject refMetaObject();
  public RefPackage refImmediatePackage();
  public RefPackage refOutermostPackage();
  public String refMofId() throws JmiException;
  public Collection refVerifyConstraints(boolean deepVerify);
}
```

5.2.2 RefFeatured

Abstract

The RefFeatured interface provides the metaobject description of instances and class proxy objects. It provides a range of operations for accessing and updating the object's features in a model-independent way.

The model assumed by the interface is that an object has structural features and operations. The model allows structural features to have single values or collection values. In the latter case, the collection values may have ordering or uniqueness semantics. There is provision for creation of new object instances, and for obtaining the set of objects that exist in a context.

Supertypes

- RefBaseObject

Operations

t refGetValue

The "refGetValue" operations fetch the current value of the attribute or reference denoted by the "feature" (or featureName) argument. If this object is a class proxy, only classifier scoped attributes can be fetched.

specific analog:	get<*ReferenceName*>**()**; get<*AttributeName*>**()**;
return type:	java.lang.Object
isQuery:	yes
parameters:	RefObject feature (or String featureName)
exceptions:	JmiException (InvalidCallException, InvalidNameException)

The result for the "refGetValue" operation is encoded as per section "Generation Rules for Parameters" on page 56.

InvalidCallException is raised when the "feature" argument does not denote an attribute or reference accessible from this object.

InvalidNameException is raised when the "featureName" does not denote a valid feature name.

t refSetValue

The “setRefValue” operations assign a new value to an attribute or reference for an object.

specific analog:	set<*ReferenceName*>**(**...**)**; set<*AttributeName*>**(**...**)**;
return type:	none
parameters:	RefObject feature (or String featureName), java.lang.object value
exceptions:	JmiException (InvalidCallException, InvalidNameException, ClosureViolationException, CompositionCycleException, InvalidObjectException, java.lang.NullPointerException)

The “value” parameter must be encoded as per the section “Generation Rules for Parameters” on page 56.

InvalidCallException is raised when one of the following conditions occur:

- The “feature” does not denote an attribute or reference accessible from this object.
- The “feature” (or featureName) denotes a multivalued attribute.

InvalidNameException is raised when the “featureName” does not denote a valid feature name.

ClosureViolationException occurs when the Composition Closure or Reference Closure rule has been violated.

CompositionCycleException occurs when the Composition Cycle rule has been violated.

t refInvokeOperation

The “refInvokeOperation” operations invoke a metamodel defined operation on the instance or class proxy object with the arguments supplied.

specific analog:	none
return type:	java.lang.Object
parameters:	RefObject requestOperation (or String operationName), List args
exceptions:	JmiException (InvalidCallException, InvalidNameException, DuplicateException, WrongSizeException, TypeMismatchException), RefException.

The “args” parameter is used to pass the values of all of the operation’s parameters. There must be a distinct parameter value (real or dummy) in the “args” list for every parameter. WrongSizeException is raised if this is not so.

The parameter values in “args” must appear in the order of the operation’s parameters as defined in the metamodel.

The “args” member values provided by the caller for parameter positions must be encoded depending on the parameter’s type and multiplicity as per the “Generation Rules for Parameters” on page 56. TypeMismatchException or WrongSizeException is raised if this is not so.

If the operation defines a result, the result for a “refInvokeOperation” call gives the result value.

InvalidCallException is raised when the “requestedOperation” does not designate an operation that can be invoked.

InvalidNameException is raised when the “operationName” does not denote a valid operation name.

Interface

```
package javax.jmi.reflect;

import java.util.*;
public interface RefFeatured extends RefBaseObject {
  public void refSetValue(RefObject feature, java.lang.Object value)
    throws JmiException;
  public void refSetValue(String featureName, java.lang.Object value)
    throws JmiException;
  public java.lang.Object refGetValue(RefObject feature) throws JmiEx-
    ception;
  public java.lang.Object refGetValue(String featureName) throws JmiEx-
    ception;
  public java.lang.Object refInvokeOperation(RefObject requestedOpera-
    tion, List args) throws JmiException, RefException;
  public java.lang.Object refInvokeOperation(String operationName, List
    args) throws JmiException, RefException;
};
```

5.2.3 RefAssociation

Abstract

The RefAssociation interface provides the metaobject description of an association. It also provides generic operations querying and updating the links that belong to the association.

The model of association supported by this interface is of collection of two ended asymmetric links between objects. The links may be viewed as ordered on one or other of the ends, and there may be some form of cardinality constraints on either end.

The RefAssociation interface is designed to be used with associations that contain no duplicate links, though this is not an absolute requirement. There is no assumption that different association objects for a given association type are mutually aware. Links are modeled as having no object identity.

(A data model that required “heavy weight” links with object identity (e.g., so that attributes could be attached to them) would need to represent them as RefObject instances. The RefAssociation interface could be used to mange light weight links between the heavy weight link objects they connect. Similar techniques could be used to represent n-ary associations. However, in both cases better performance would be achieved using a purpose built reflective layer.)

Supertypes

- RefBaseObject

Operations

t refAllLinks

The “refAllLinks” operation returns all links in the link set for this Association object.

specific analog:	none
return type:	Collection
isQuery:	yes
parameters:	none
exceptions:	none

This operation returns the current link set for the current association extent as defined for the specific version of this operation.

The Collection returned from this operation is an immutable live collection. This is, the collection will reflect any changes to the source, however, the operations in the Collection interface cannot be used to update the source.

t refLinkExists

The “refLinkExists” operation returns true if and only if the supplied link is a member of the link set for this association object.

specific analog:	exists**(**...**)**;
return type:	boolean
isQuery:	yes
parameters:	RefObject endOne, RefObject endTwo
exceptions:	JmiException (TypeMismatchException)

TypeMismatchException is raised if the parameters do not match the types of the respective association ends.

t refQuery

The “refQuery” operations return a list containing all instance objects that are linked to the supplied “queryObject” by links in the extent of this association object, where the links all have the “queryObject” at the “queryEnd”.

specific analog:	<*AssociationEndName*> **(**...**)**;
return type:	Collection
isQuery:	yes
parameters:	RefObject queryEnd (or String queryEndName) RefObject queryObject
exceptions:	JmiException (InvalidCallException, InvalidNameException, TypeMismatchException, InvalidObjectException, java.lang.NullPointerException)

The “queryEnd” (or “queryEndName) parameter must designate an association end for this association object. InvalidCallException is raised if this is not so.

The “queryObject” parameter must be an instance object whose type is compatible with the type of the “queryEnd” (or “queryEndName) of the association. TypeMismatchException is raised if this is not so.

InvalidCallException is raised if the “queryEnd” parameter does not designate a valid object.

InvalidNameException is raised when the “queryEnd” does not denote a valid object name.

InvalidObjectException or NullPointerException is raised if the “queryObject” (or “queryEndName) is non-existent, null, or inaccessible.

t refAddLink

The “refAddLink” operation adds “newLink” into the set of links in the extent of this association object. If one or other of the association’s ends is ordered, the link is inserted after the last link with respect to that ordering.

specific analog:	add**(**...);
return type:	boolean
parameters:	RefObject endOne, RefObject endTwo.
exceptions:	JmiException (InvalidCallException, WrongSizeException, ClosureViolationException, CompositionCycleException, TypeMismatchException, InvalidObjectException, java.lang.NullPointerException)

Both RefObject members of the “newLink” parameter should be valid instance objects.

If the new link cannot be added for any reason other than that it already contains the link, an exception must be thrown (rather than returning false).

The instance objects must be compatible with the association. TypeMismatchException is raised otherwise.

Both instance objects must be valid instance objects. InvalidObjectException, NullPointerException is raised otherwise.

t refRemoveLink

The “refRemoveLink” operation removes the existing link from the association.

specific analog:	remove**(**...**)**;
return type:	boolean
parameters:	RefObject endOne, RefObject endTwo.
exceptions:	JmiException (WrongSizeException, TypeMismatchException, java.lang.NullPointerException)

The instance objects passed in must be compatible with the association. TypeMismatchException is raised otherwise.

Returns true if the link was successfully removed.

WrongSizeException is raised if removing the link causes the association end to violate the specified multiplicity.

Interface

```
package javax.jmi.reflect;

import java.util.*;
public interface RefAssociation extends RefBaseObject {
 public Collection refAllLinks();
 public boolean refLinkExists(RefObject endOne, RefObject endTwo) throws
   JmiException;
 public Collection refQuery(RefObject queryEnd, RefObject queryObject)
   throws JmiException;
 public Collection refQuery(String queryEndName, RefObject queryObject)
   throws JmiException;
 public boolean refAddLink(RefObject endOne, RefObject endTwo) throws
   JmiException;
 public boolean refRemoveLink(RefObject endOne, RefObject endTwo) throws
   JmiException;
 }
```

5.2.4 RefPackage

Abstract

The RefPackage interface is an abstraction for accessing a collection of objects and their associations. The interface provides an operation to access the metaobject description for the package, and operations to access the package instance's class proxy objects and its association objects

Supertypes

- RefBaseObject

Operations

t refClass

The “refClass” operations return the class proxy object for a given class.

specific analog:	get<*ClassName*>()
return type:	RefClass
isQuery:	yes
parameters:	RefObject type (or String className)
exceptions:	JmiException (InvalidCallException, InvalidNameException)

The “type” (or “className”) parameter should designate the class whose class proxy object is to be returned.

InvalidCallException is raised if the “type” parameter does not designate a valid class.

InvalidNameException is raised when the “className” does not denote a valid class name.

t refAssociation

The “refAssociation” operations return an association object for a given association.

specific analog:	get<*AssociationName*>()
return type:	RefAssociation
isQuery:	yes
parameters:	RefObject association (or String associationName)
exceptions:	JmiException (InvalidCallException, InvalidNameException)

The “association” (or “associationName”) parameter should designate the (M2) association whose association object is to be returned.

InvalidCallException is raised if the “association” parameter does not designate a valid association.

InvalidNameException is raised when the “associationName” does not denote a valid association name.

t refPackage

The “refPackage” operations return a package object for a nested or clustered package.

specific analog:	get<*PackageName*>()
return type:	RefPackage
isQuery:	yes
parameters:	RefObject nestedPackage (or String nestedPackageName)
exceptions:	JmiException (InvalidCallException, InvalidNameException)

The "nestedPackage" (or “nestedPackageName”) parameter should designate the package whose package object is to be returned. It must either be nested within the package for this package object, or imported with “isCluster” set to true.

InvalidCallException is raised if the “nestedPackage” parameter does not designate a valid package.

InvalidNameException is raised when the “nestedPackageName” does not denote a valid nested package name.

t refAllPackages

The “refAllPackages” operation returns all packages directly contained or clustered by this package.

specific analog:	None
return type:	Collection of RefPackages
isQuery:	yes
parameters:	None
exceptions:	JmiException

Returns a (possible empty) collection of RefPackages directly contained by this package.

The Collection returned from this operation is an immutable live collection. This is, the collection will reflect any changes to the source, however, the operations in the Collection interface cannot be used to update the source.

t refAllClasses

The “refAllClasses” operation returns all class proxies directly contained by this package.

specific analog:	None
return type:	Collection of RefClasses
isQuery:	yes
parameters:	None
exceptions:	JmiException

Returns a (possible empty) collection of RefClasses directly contained by this package.

The Collection returned from this operation is an immutable live collection. This is, the collection will reflect any changes to the source, however, the operations in the Collection interface cannot be used to update the source.

t refAllAssociations

The “refAllAssociation” operation returns all associations directly contained by this package.

specific analog:	None
return type:	Collection of RefAssociations
isQuery:	yes
parameters:	None
exceptions:	JmiException

Returns a (possible empty) collection of RefAssociations directly contained by this package.

The Collection returned from this operation is an immutable live collection. This is, the collection will reflect any changes to the source, however, the operations in the Collection interface cannot be used to update the source.

t refCreateStruct

This “refCreateStruct” operations create a new instance of a struct data type.

specific analog:	create<*Struct*>(...);
return type:	RefStruct
parameters:	RefObject structType (or String structName), List args
exceptions:	JmiException (WrongSizeException, TypeMismatchException, InvalidObjectException, InvalidCallException, InvalidNameException, java.lang.NullPointerException)

The “refCreateStruct” operation creates an instance of a struct data type defined by the metaobject “structType” (or “structName”) whose attribute values are specified by the ordered collection “args”.

The members of the “args” list correspond 1-to-1 to the parameters for the specific create operation. They must be encoded as per “Generation Rules for Parameters” on page 56.

InvalidCallException is raised if the “structType” parameter does not designate a struct type.

InvalidNameException is raised when the “structName” does not denote a valid struct name.

t refGetEnum

This “refGetEnum” returns the enumeration object representing the enumeration literal.

specific analog:	none.
return type:	RefEnum
parameters:	RefObject enumType (or String “enumName”) String literalName
exceptions:	JmiException (TypeMismatchException, InvalidCallException, InvalidNameException, java.lang.NullPointerException)

The “refGetEnum” operation returns the instance of an enumeration (i.e., an enumeration literal) whose value is described by the value of “literalName”. Note that the type of enumeration is defined by the metamobject that owns the metaLiteral object.

InvalidCallException is raised if the “enumType” parameter does not designate a valid enumeration.

InvalidNameException is raised when the “enumName” does not denote a valid enum name.

t refDelete

The "refDelete" operation destroys this package, including the objects it contains directly or transitively.

specific analog:	none
return type:	none
parameters:	none
exceptions:	JmiException

Deletion of an outermost package causes all objects within its extent to be deleted.

Interface

```
package javax.jmi.reflect;

import java.util.*;
public interface RefPackage extends RefBaseObject {
  public RefClass refClass(RefObject type) throws JmiException;
  public RefClass refClass(String className) throws JmiException;
  public RefPackage refPackage(RefOject nestedPackage)throws JmiExcep-
    tion;
  public RefPackage refPackage(String nestedPackageName)throws JmiExcep-
    tion;
  public RefAssociation refAssociation(RefObject association)throws
    JmiException;
  public RefAssociation refAssociation(String associationName)throws
    JmiException;
  public Collection refAllPackages();
  public Collection refAllClasses();
  public Collection refAllAssociations();
  public RefStruct refCreateStruct(RefObject structType, List args)
    throws JmiException;
  public RefStruct refCreateStruct(String structName, List args) throws
    JmiException;
  public RefEnum refGetEnum(RefObject enumType, String literalName)
    throws JmiException;
  public RefEnum refGetEnum(String enumName, String literalName) throws
    JmiException;
  public void refDelete() throws JmiException;
}; // end of interface RefPackage
```

5.2.5 RefClass

Abstract

The RefClass interface provides the metaobject description of a class proxy object, and a range of operations for accessing and updating an object's classifier scoped features.

Supertypes

- RefFeatured

Operations

t refCreateInstance

This "refCreateInstance" operation creates a new instance of the class for the class proxy's most derived interface. The "args" list gives the initial values for the new instance object's instance scoped, non-derived attributes.

specific analog:	create<*ClassName*>(...);
return type:	RefObject
parameters:	List args
exceptions:	JmiException (WrongSizeException, DuplicateException, ClosureViolationException, AlreadyExistsException, TypeMismatchException)

The value of the "args" parameter must be either null, or a list such that: 1 - the list of arguments correspond 1-to-1 to the parameters for the specific create operation; and 2 - each argument must be encoded as per the section "Generation Rules for Parameters" on page 56. If "args" is null, then the operation corresponds to the default constructor that takes no arguments.

t refAllOfType

The “refAllOfType” operation returns the set of all instances in the current extent whose type is given by this object’s class or one of its sub-classes.

specific analog:	None.
return type:	Collection (unique; unordered)
isQuery:	yes
parameters:	None
exceptions:	none

The Collection returned from this operation is an immutable live collection. This is, the collection will reflect any changes to the source, however, the operations in the Collection interface cannot be used to update the source.

t refAllOfClass

The “refAllOfClass” operation returns the set of all instances in the current extent whose type is given by this object’s class (instances of sub classes are not included).

specific analog:	None.
return type:	Collection (unique; unordered)
isQuery:	yes
parameters:	None
exceptions:	none

The Collection returned from this operation is an immutable live collection. This is, the collection will reflect any changes to the source, however, the operations in the Collection interface cannot be used to update the source.

t refCreateStruct

The “refCreateStruct” operations create a new instance of a struct data type.

specific analog:	create<*Struct*>(...).
return type:	RefStruct
parameters:	RefObject structType (or String structName), List args
exceptions:	JmiException (WrongSizeException, TypeMismatchException, InvalidObjectException, InvalidCallException, InvalidNameException, java.lang.NullPointerException)

The “refCreateStruct” operation creates an instance of a struct data type defined by the metaobject “structType” (or “structName”) whose attribute values are specified by the ordered collection “args”.

The members of the “args” list correspond 1-to-1 to the parameters for the specific create operation. They must be encoded as per the section titled “Generation Rules for Parameters” on page 56.

InvalidCallException is raised if the “structType” parameter does not designate a valid struct type.

InvalidNameException is raised when the “structName” does not denote a valid struct name.

t refGetEnum

This “refGetEnum” returns the enumeration object representing the enumeration literal.

specific analog:	none.
return type:	RefEnum
parameters:	RefObject enumType (or String enumName) String literalName
exceptions:	JmiException (TypeMismatchException, InvalidCallException, InvalidNameException, java.lang.NullPointerException)

The “refGetEnum” operation returns an instance of an enumeration (i.e., an enumeration literal) whose value is described by the value of “literalName”. Note that the type of enumeration is defined by the metamobject that owns the metaLiteral object.

InvalidCallException is raised if the “enumType” parameter does not designate a valid enumeration.

InvalidNameException is raised when the “enumName” does not denote a valid enumeration name.

Interface

```
package javax.jmi.reflect;

import java.util.*;
public interface RefClass extends RefFeatured {
 public RefObject refCreateInstance(List args) throws JmiException;
 public Collection refAllOfType();
 public Collection refAllOfClass();
 public RefStruct refCreateStruct(RefObject structType, List args)
   throws JmiException;
 public RefStruct refCreateStruct(String structName, List args) throws
   JmiException;
 public RefEnum refGetEnum(RefObject enumType, String literalName)
   throws JmiException;
 public RefEnum refGetEnum(String enumName, String literalName) throws
   JmiException;
}; // end of interface RefClass
```

5.2.6 RefObject

Abstract

The RefObject interface provides the metaobject description of an instance object, and a range of operations for accessing and updating the object's features.

Supertypes

- RefFeatured

Operations

t refIsInstanceOf

This operation tests whether this RefObject is an instance of the class described by the “objType” metaobject. If the “considerSubtypes” argument is true, an object whose class is a subclass of the class described by “objType” will be considered as an instance of the class.

specific analog:	none
return type:	boolean
isQuery:	yes
parameters:	RefObject objType boolean considerSubtypes
exceptions:	none

t refClass

This operation returns the RefObject’s class proxy object

specific analog:	none
return type:	RefClass
isQuery:	yes
parameters:	none
exceptions:	none

If the ignoreLifecycle tag (attached to the package containing the metamodel) is set to true, this method throws the java.lang.UnsupportedOperationException.

t refImmediateComposite

The “refImmediateComposite” operation returns the immediate composite object for this instance as specified below.

specific analog:	none
return type:	RefFeatured
isQuery:	yes
exceptions:	none

The immediate composite object C returned by this operation is an instance object such that:

- C is related to this object via a relation R defined by an attribute or association.
- The aggregation semantics of the relation R are “composite”.

- This object fills the role of “component” in its relationship with C.

If the immediate object C does not exist, or if “this” object is a class proxy object rather than an instance object, a Java null object reference is returned.

Note – If the composite relationship R corresponds to a “classifier-level” scoped attribute, the immediate composite object C will be the class proxy object that holds the attribute value.

---

t refOutermostComposite

The “refOutermostComposite” operation returns the “outermost composite” for this object as defined below.

specific analog:	none
return type:	RefFeatured
isQuery:	yes
exceptions:	none

The outermost composite object C returned by this operation is an instance object such that:

- There is a chain of *zero or more* immediate composite relationships (as described for "refInvokeOperation" above) connecting “this” object to C, and
- C does not have an immediate composite.

The above definition is such that if "this" object is not a component of any other object, it will be returned.

t refDelete

The "refDelete"” operation destroys this object, including the objects it contains directly or transitively.

specific analog:	none
return type:	none
parameters:	none
exceptions:	JmiException

Deletion of an instance object deletes it and its component closure.

If the ignoreLifecycle tag (attached to the package containing the metamodel) is set to true, this method throws the java.lang.UnsupportedOperationException.

Interface

```
package javax.jmi.reflect;


public interface RefObject extends RefFeatured {
public boolean refIsInstanceOf(RefObject objType, boolean considerSub-
types);
public RefClass refClass();
public RefFeatured refImmediateComposite();
public RefFeatured refOutermostComposite();
public void refDelete() throws JmiException;
};
```

5.3 Reflective Interfaces for Data Types

This section describes the reflective interfaces for data types that do not get mapped directly to Java primitive types, i.e., enumerations and structs. It also describes the RefAssociationLink interface — the reflective interface implemented by the instances of associations, and the reflect.RefException class — the superclass which is extended by all modeled exceptions.

5.3.1 RefEnum

Abstract

The RefEnum interface is the reflective interface for enumerations. It provides generic operations for querying the enumeration.

Supertypes

- java.io.Serializable

Operations

This operation returns the integer representation of the enumeration literal.

t toString

The “toString” operation returns the enumeration literal

specific analog:

return type:	String
isQuery:	yes
parameters:	none
exceptions:	None

This operation returns the enumeration literal.

t refTypeName

The “refTypeName” operation returns the fully qualified name of the enumerations metaobject

specific analog:

return type:	List
isQuery:	yes
parameters:	none
exceptions:	None

This operation returns the fully qualified name of the enumeration object’s metaobject.

The Collection returned from this operation has copy semantics. That is, it does not reflect any changes to the source after the operation is executed, and it cannot be used to update the source.

t equals

The “equals” operation compares the enumeration object with another object.

specific analog:

return type:	boolean
isQuery:	yes
parameters:	java.lang.Object other
exceptions:	None

The comparison for enumerations is based on literal value. If two enumerations are of the same type and represent the same literal, then they are equal.

Interface

```
package javax.jmi.reflect;

import java.io.*;
import java.util.*;
public interface RefEnum extends Serializable {
  public String toString();
  public List refTypeName();
  public boolean equals(Object other);
}
```

5.3.2 RefStruct

Abstract

The RefStruct interface is the reflective interface for struct data types. It provides generic operations for querying structs.

Supertypes

- java.io.Serializable

Operations

t refFieldNames

The “refFieldNames” operation returns the list of field names in this struct

specific analog:	None
return type:	java.util.List
isQuery:	yes
parameters:	None
exceptions:	None

This operation returns the list of (String) field names contained in this struct.

t refGetValue

The “refGetValue” operation returns the value of the specified field

specific analog:	<AccessorName>
return type:	Object
isQuery:	yes
parameters:	String fieldName
exceptions:	JMIException (InvalidNameException)

This operation returns the list of (String) field names contained in this struct.

t refTypeName

The “refTypeName” operation returns the fully qualified name of the structs metaobject.

specific analog:	None
return type:	List
isQuery:	yes
parameters:	None
exceptions:	None

This operation returns the fully qualified name of the struct object’s metaobject.

The List returned from this operation has copy semantics. That is, it does not reflect any changes to the source after the operation is executed, and it cannot be used to update the source.

t equals

The “equals” operation compares the struct object with another object.

specific analog:	none
return type:	boolean
isQuery:	yes
parameters:	java.lang.Object other
exceptions:	None

The comparison for structs is based on attribute values. If two structs are of the same type and all attributes have the same value, then they are equal.

Interface

```
package javax.jmi.reflect;

import java.io.*;
import java.util.*;

public interface RefStruct extends Serializable {
 public List refFieldNames();
 public Object refGetValue(String fieldName) throws JmiException;
 public List refTypeName();
 public boolean equals (Object other);
}
```

5.3.3 RefAssociationLink

Abstract

The RefAssociationLink interface is the reflective interface implemented by all association links, i.e., instances of RefAssociation. It provides generic operations for querying the association link.

Supertypes

- None

Operations

t refFirstEnd

The “refFirstEnd” operation returns object at the first end of the link

specific analog:	
return type:	RefObject
isQuery:	yes
parameters:	none
exceptions:	none

This operation returns the first end (as appearing in the metamodel) of the link.

t refSecondEnd

The “refSecondEnd” operation returns object at the second end of the link

specific analog:	
return type:	RefObject
isQuery:	yes
parameters:	none
exceptions:	none

This operation returns the second end (as appearing in the metamodel) of the link.

Interface

```
package javax.jmi.reflect;
public interface RefAssociationLink {
  public RefObject refFirstEnd();
  public RefObject refSecondEnd();
}
```

5.3.4 RefException Class

The RefException class is the superclass that is extended by instances of modeled exceptions. That is, all M1 instances of exceptions modeled in the respective M2 (i.e., instances of RefException), must extend the reflective RefException class.

RefException is intended to be a checked exception, and as such, it extends java.lang.Exception.

Supertypes

- java.lang.Exception.

Operations

None.

Class Definition

```
package javax.jmi.reflect;
public class RefException extends java.lang.Exception {
  // default constructor
  public RefException() {
  }
  // constructor that takes an error message
```

```
  public RefException(String msg) {
    super(msg);
  }
}
```

5.4 The Exception Framework

This section describes the non-modeled exceptions thrown by a JMI service. That is, the exceptions that are not described in the metamodel but thrown as a result of some violation of the MOF, or some exception that occurred within the JMI service. For example, the "AlreadyExistsException" is thrown by the class factory operation when a client attempts to create a second instance of a class that is designated as a singleton (i.e., the class can have no more than one instance). This exception does not appear in the metamodel, but appears in the generate API as a result of it being required by the Java API generation template for class proxies.

5.4.1 JmiException

At the root of the exception framework is JmiException, which is the superclass of all other JMI exceptions. JmiException is intended to be an unchecked exception and as such, extends java.lang.RuntimeException.

The JmiException contains two private attributes - elementInError and objectInError. The elementInError is the ModelElement of the instance for which the error is being reported. The objectInError is the object for which the error is being reported.

Operations

t getElementInError

The "getElementInError" operation returns the element-in-error.

return type:	RefObject
parameters:	None

t getObjectInError

The “getObjectInError” operation returns the object-in-error.

return type:	Object
parameters:	None

```
package javax.jmi.reflect;
public abstract class JmiException extends java.lang.RuntimeException {
  private final RefObject elementInError;
  private final Object objectInError;
  // default constructor that takes no arguments
  public JmiException() {
        this.elementInError = null;
        this.objectInError = null;
  }
  // constructor that takes the element-in-error
  public JmiException(RefObject elementInError) {
        this.elementInError = elementInError;
        this.objectInError = null;
  }
  // constructor that takes element-in-error and message.
  public JmiException(RefObject elementInError, String msg) {
        super(msg);
        this.elementInError = elementInError;
        this.objectInError = null;
  }
  // constructor that takes the object-in-error, element-in-error
  public JmiException(Object objectInError, RefObject elementInError) {
        this.elementInError = elementInError;
        this.objectInError = objectInError;
  }
  // constructor that takes object-in-error, element-in-error, and mes-
sage.
  public JmiException(Object objectInError, RefObject elementInError,
String msg) {
        super(msg);
        this.elementInError = elementInError;
        this.objectInError = objectInError;
  }

// constructor that takes error message.
```

```
  public JmiException(String msg) {
     super(msg);
  }

  public RefObject getElementInError() {
        return elementInError;
  }

  public Object getObjectInError() {
    return objectInError;
  }
}
```

5.4.2 AlreadyExistsException

The AlreadyExistsException is raised when a client attempts to create a second instance of an M2 class whose isSingleton is set to true.

Operations

t getExistingInstance

The “getExistingInstnace” operation returns the existing instance of singleton.

return type:	RefObject
parameters:	None

```
package javax.jmi.reflect;
public class AlreadyExistsException extends JmiException {
  private final RefObject existing;
  // constructor that takes existing-object
  public AlreadyExistsException(RefObject existing) {
        super(existing.refMetaObject());
        this.existing = existing;
  }
  // constructor that takes existing-object and error message
  public AlreadyExistsException(RefObject existing, String msg) {
        super(existing.refMetaObject(), msg);
        this.existing = existing;
  }

  public RefObject getExistingInstance() {
        return existing;
```

```
    }
}
```

5.4.3 ClosureViolationException

The ClosureViolationException is thrown when Composition Closure or Reference Closure rules are violated. Note that the Supertype Closure rule can never be violated in JMI.

Operations

None.

```
package javax.jmi.reflect;
public class ClosureViolationException extends JmiException {
// constructor that takes object-in-error, and element-in-error
  public ClosureViolationException(Object objectInError, RefObject ele-
mentInError) {
        super(objectInError, elementInError);
  }
  // construct that takes object-in-error, element-in-error, and message
  public ClosureViolationException(Object objectInError, RefObject ele-
mentInError, String msg) {
        super(objectInError, elementInError, msg);
  }
}
```

5.4.4 CompositionCycleException

The CompositionCycleException is thrown when an instance object is a component of itself.

Operations

None.

```
package javax.jmi.reflect;
public class CompositionCycleException extends JmiException {
  // constructor taking object-in-error and element-in-error
  public CompositionCycleException(Object objectInError, RefObject ele-
    mentInError) {
      super(objectInError, elementInError);
  }

    // constructor taking object-in-error, element-in-error, and message
```

```
 public CompositionCycleException(Object objectInError, RefObject ele-
   mentInError, String msg) {
    super(objectInError, elementInError, msg);
   }
 }
```

5.4.5 CompositionViolationException

CompositionViolationException is thrown when an operation attempts to add a value that is already a component in a composite relation, as a component of another composite relation.

Operations

None.

```
package javax.jmi.reflect;
public class CompositionViolationException extends JmiException {
 // constructor taking object-in-error and element-in-error
 public CompositionViolationException(Object objectInError, RefObject
   elementInError) {
     super(objectInError, elementInError);
 }

   // constructor taking object-in-error, element-in-error, and message
 public CompositionViolationException(Object objectInError, RefObject
   elementInError, String msg) {
     super(objectInError, elementInError, msg);
 }
}
```

5.4.6 ConstraintViolationException

The ConstraintViolationException is thrown when a constraint is violated.

Operations

None.

```
package javax.jmi.reflect;
public class ConstraintViolationException extends JmiException {
// constructor taking object-in-error and element-in-error
```

```
  public ConstraintViolationException(Object objectInError, RefObject
elementInError) {
super(objectInError, elementInError);
  }
  // constructor taking object-in-error, element-in-error, and message
  public ConstraintViolationException(Object objectInError, RefObject
elementInError, String msg) {
        super(objectInError, elementInError, msg);
  }
}
```

5.4.7 DuplicateException

The DuplicateException is thrown when a duplicate value is added to a multivalued attribute whose isUnique property is set to true.

Operations

None

```
package javax.jmi.reflect;
public class DuplicateException extends JmiException {
// constructor taking object-in-error, element-in-error
  public DuplicateException(Object objectInError, RefObject elementInEr-
ror) {
        super(objectInError, elementInError);
}
  // constructor taking object-in-error, element-in-error, an message
public DuplicateException(Object object-in-error, RefObject elementInEr-
ror, String msg) {
        super(objectInError, elementInError, msg);
  }

}
```

5.4.8 InvalidCallException

The InvalidCallException is thrown when an operation is invoked incorrectly using the reflective API.

Operations

None.

```
package javax.jmi.reflect;
```

```
public class InvalidCallException extends JmiException {
// constructor taking object-in-error, element-in-error
  public InvalidCallException(Object objectInError, RefObject elementIn-
Error) {
        super(objectInError, elementInError);
  }
  // constructor taking object-in-error, element-in-error, and message
  public InvalidCallException(Object objectInError, RefObject elementIn-
Error, String msg) {
        super(objectInError, elementInError, msg);
  }
}
```

5.4.9 InvalidNameException

InvalidNameException is thrown when the user passes an invalid feature name to a reflective method.

Operations

t getInvalidName

The “getInvalidName” operation returns the invalidName.

return type:	String
parameters:	None

```
package javax.jmi.reflect;
public class InvalidNameException extends JmiException {
   private final String invalidName;
// constructor taking invalidName
  public InvalidNameException(String invalidName) {
        this.invalidName = invalidName;
  }
  // constructor taking invalidName and message
  public InvalidNameException(String invalidName, String msg) {
        super(msg);
        this.invalidName = invalidName;
  }

   public String getInvalidName() {
      return invalidName;
```

```
    }
}
```

5.4.10 InvalidObjectException

InvalidObjectException is thrown when an object detects a non-existing (i.e., deleted) object.

Operations

None.

```
package javax.jmi.reflect;
public class InvalidObjectException extends JmiException {
// constructor taking element-in-error
  public InvalidObjectException(RefObject elementInError) {
        super(elementInError);
  }
  // constructor taking element-in-error and message
  public InvalidObjectException(RefObject elementInError, String msg) {
        super(elementInError, msg);
    }
}
```

5.4.11 TypeMismatchException

TypeMismatchException is thrown when the value has the wrong type for the context in which it was supplied.

Operations

t getExpectedType

The “getExpectedType” operation returns the Class of the expected type.

return type:	Class
parameters:	None

```
package javax.jmi.reflect;
public class TypeMismatchException extends JmiException {
private final Class expectedType;
// constructor taking expected-type, object-in-error, and element-in-
error
```

```
  public TypeMismatchException(Class expectedType, Object objectInError,
RefObject elementInError) {
        super(objectInError, elementInError);
this.expectedType = expectedType;
  }
  // constructor taking expected-type, objectInError, element-in-error,
and message
  public TypeMismatchException(Class expectedType, Object objectInError,
RefObject elementInError, String msg) {
        super(objectInError, elementInError, msg);
this.expectedType = expectedType;
  }

  public Class getExpectedType() {
        return expectedType;
    }
}
```

5.4.12 WrongSizeException

WrongSizeException is thrown when the lower bound or upper bound is violated.

Operations

None.

```
package javax.jmi.reflect;
public class WrongSizeException extends JmiException {
// constructor taking element-in-error
  public WrongSizeException(RefObject elementInError) {
        super(elementInError);
  }
  // constructor taking element-in-error and message
  public WrongSizeException(RefObject elementInError, String msg) {
        super(elementInError, msg);
  }
}
```

5.5 XMI Import/Export in JMI

A JMI service will provide APIs for streaming metadata in the XML Metadata Interchange (XMI) format. The XmiWriter and XmiReader interfaces import and export XML documents to and from a JMI service. The methods in these interfaces read and write XMI elements, i.e., XML elements with the "XMI" tag (containing valid XMI). An XMI element is all the information contained within the <XMI> tag and its corresponding <\XMI> tag.

5.5.1 XMIWriter

The XmiWriter writes XMI elements to a stream. The stream is left open upon completion of the write, and a separate XMI element is written for each write operation. The 'xmiVersion' parameter specifies the version of XMI. The values for the ‘xmiVersion’ parameter are the values placed in the 'version' attribute of the XMI element as defined in the XMI specification, e.g., "1.2" or "2.0". A JMI service is required to support only XMI version 1.2 for which the xmiVersion value is "1.2". In the future and upon standardization of version 2.0, vendors may choose to support version 2.0 as well.

Operations

- write (RefPackage)

The “write” operation (which takes a RefPackage as the second argument) writes an entire extent to a stream.

return type:	None
parameters:	java.io.OutputStream stream RefPackage extent String xmiVersion
exceptions:	java.io.IOException

- write (Collection)

The “write” operation (which takes a Collection as the second argument) writes MOF objects contained in the collection to a stream.

return type:	None
parameters:	java.io.OutputStream stream java.util.Collection objects String xmiVersion
exceptions:	java.io.IOException

```
package javax.jmi.xmi;
```

```
import javax.jmi.reflect.*;

public interface XmiWriter {
    public void write(java.io.OutputStream stream, RefPackage extent,
      String xmiVersion) throws java.io.IOException;

    public void write(java.io.OutputStream stream, java.util.Collection
      objects, string xmiVersion) throws java.io.IOException;
}
```

5.5.2 XmiReader

The XmiReader reads an entire XMI element from a stream into a RefPackage. The stream is left open upon completion of the read.

Operations

t read

The “read” operation reads an entire XMI element from a specified URI into an outermost package. If the operation succeeds, the read operation returns a collection of RefObjects which are the outermost objects in the XMI content.

return type:	Collection (of RefObjects)
parameters:	String URI RefPackage extent
exceptions:	java.io.IOException, MalformedXMIException

t read

This “read” operation reads an entire XMI element from an InputStream into a package extent. Here, the String URI argument specifies the logical name given to the document to be read, and can be used to resolve relative links, if any. If the InputStream is null, then this methods is equivalent to the read operation described above.

return type:	Collection (of RefObjects)
parameters:	java.io.InputStream String URI RefPackage
exceptions:	java.io.IOException, MalformedXMIException

```
package javax.jmi.xmi;

import javax.jmi.reflect.*;
```

```
public interface XmiReader {
 public java.util.Collection read(String URI, RefPackage extent) throws
   java.io.IOException, javax.jmi.xmi.MalformedXMIException;

 public java.util.Collection read(java.io.InputStream stream, String
   URI, javax.jmi.reflect.RefPackage extent) throws java.io.IOException,
   javax.jmi.xmi.MalformedXMIException;
}
```

5.5.3 MalformedXMIException

The MalformedXMIException is thrown when the XMIReader is given a malformed XMI element.

Operations

None.

```
package javax.jmi.xmi;
public class MalformedXMIException extends java.lang.Exception {
 // default constructor taking no arguments
 public MalformedXMIException() {
 }

 // constructor taking error message
 public MalformedXMIException(String msg) {
    super(msg);
 }


}
```

APPENDIX **A**

The JMI APIs for the MOF

The JMI reflective APIs and the generated APIs for the MOF (generated by applying the JMI mapping to the MOF model) are contained in the companion jmi.zip file.

APPENDIX**B**

Accessing a JMI Service using the Connector Architecture

A JMI Service can benefit from the J2EE Connector Architecture (JCA) by using it to provide connectivity to the service in a pluggable way. The Connector Architecture provides seamless integration of data sources, in this case a JMI service, with application servers. The following example illustrates how a JMI service may be accessed via JCA by treating it as a generic Enterprise Information System (EIS).

Note – This example illustrates one possible strategy for connecting to a JMI service. JMI itself does not prescribe how clients connect to JMI services.

In this example, an EIS ("MyEIS") uses the JCA-prescribed patterns for the Common Client Interface (CCI) to define its own Connection and ConnectionFactory interfaces. A client of MyEIS obtains an instance of the ConnectionFactory interface via JNDI, which is registered as a node in a local JNDI tree. The client then uses one of two variants of the getConnection() method defined by the ConnectionFactory interface to obtain an actual instance of the Connection interface. This is illustrated below:

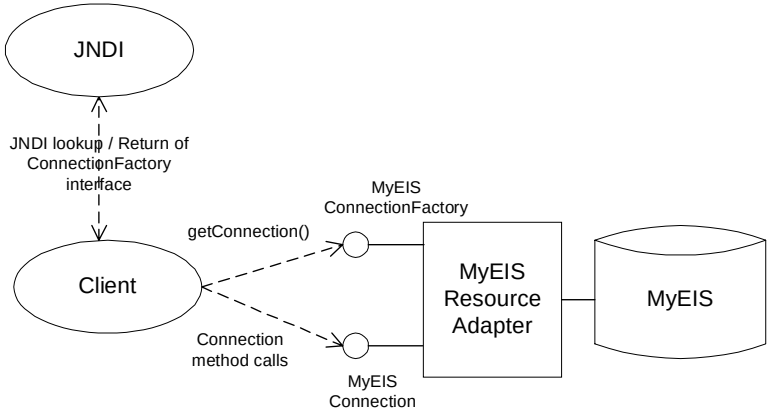

FIGURE B-1 JMI and the Connection Architecture.

The ConnectionFactory and Connection interfaces are defined by MyEIS as follows:

```
public interface com.myeis.ConnectionFactory extends java.io.Serial-
izable, javax.resource.Referenceable {
  public com.myeis.Connection getConnection() throws com.myeis.Resource-
    Exception;
  public com.myeis.Connection getConnection(javax.resource.cci.Connec-
    tionSpec properties) throws com.myeis.ResourceException;
  public javax.resource.cci.ConnectionSpec createConnectionSpec() throws
    com.myeis.ResourceException;
  public javax.resource.cci.ResourceAdapterMetaData getMetaData() throws
    com.myeis.ResourceException;
}
```

```
public interface com.myeis.Connection {
  public void close() throws com.myeis.ResourceException;
  public javax.resource.cci.ConnectionMetaData getMetaData() throws
    com.myeis.ResourceException;
  public javax.jmi.reflect.RefPackage getTopLevelPackage() throws
    com.myeis.ResourceException;
}
```

MyEIS's custom version of ConnectionFactory provides two factory methods for creating Connection instances:

```
com.myeis.Connection getConnection()
```

```
com.myeis.Connection getConnection(javax.resource.cci.Connection-
Spec properties)
```

The first variant does not require the client to supply any properties. This variant is used in cases where a client component defers connection management to its container. The second variant accepts a JCA ConnectionSpec interface as a means of specifying client-level connection properties. ConnectionSpec, as defined by JCA, is essentially a marker or tag interface for a JavaBean implementation class that provides resource-specific properties required for establishing client-managed connections to JCA-enabled EIS resources. A client introspects ConnectionSpec to determine the supported properties. These properties are accessible as JavaBean-style getter/setter methods. Default properties for ConnectionSpec, as defined by the JCA Specification, are UserName and Password. An EIS may define additional properties, if necessary.

MyEIS uses an "embedded" JMI service as a means of providing advanced metadata management via JMI. Access to JMI services implemented by MyEIS is provided through the Connection interface's ‘getTopLevelPackage’ method:

```
javax.jmi.reflect.RefPackage getTopLevelPackage()
```

This method returns a single object that implements the javax.jmi.reflect.RefPackage interface. This object is a outermost (or root-level) JMI Package. Starting with this top-level package, a client application embarks upon a process of discovery and access of the metadata contained within this package using the JMI generated and reflective interfaces.

Given below is the complete Java program that connects to the “MyEIS” JMI service and obtains a handle to the outermost package.

```
import java.util.*;
import javax.naming.*;
import com.myeis.*;

public class EISTest {

 public static void main (String[] args) {
   EISTest eisTest = new EISTest();
   // connect to a EIS resource instance
   eisTest.connectToEISResource();
 } // main

 public void connectToEISResource() {
   try {
     // Obtain the ConnectionFactory from the JNDI tree
     Context initCtx = new InitialContext();
     com.myeis.ConnectionFactory cxf = (com.myeis.ConnectioF-
       tory)initCtx.lookup("java:comp/env/eis/MyEIS");
     javax.resource.cci.ConnectionSpec cxs= (javax.resource.cci.Connec-
       tionSpec)cxf.createConnectionSpec();

     // Set the user name and password values
     ((com.myeis.ConnectionSpecImpl)cxs).setUserName("guest");
     ((com.myeis.ConnectionSpecImpl)cxs).setPassword("password");

     // Establish a connection to the EIS resource
     com.myeis.Connection cx =(com.myeis.Connection)cxf.getConnec-
       tion(cxs);

     // Obtain the top-level package extent from the Connection
       javax.jmi.reflect.RefPackage tlp = (javax.jmi.reflect.RefPack-
       age)cx.getTopLevelPackage();

     // JMI specific code to access/modify metadata in the RefPackage

     // Close the connection to the EIS resource
     cx.close();
```

```
    } // try
    catch (javax.naming.NamingException ne) {
      ne.printStackTrace();
    }
    catch (com.myeis.ResourceException re) {
      re.printStackTrace();
    }
  } // connectToEISResource
}
```

APPENDIX**C**

Example: Simple XML model

This example shows a typical usage of JMI service. It shows what the generated JMI API look like for a simple metamodel, and how it can be used (together with the JMI reflective API) to create, access and update metadata.

t Simple Metamodel

As JMI is mapping from MOF 1.4 to Java, in order to be able to generate and use JMI API for some kind of metadata, we need to describe the metadata using MOF. For the purpose of this example we will consider a very simple metamodel of XML.

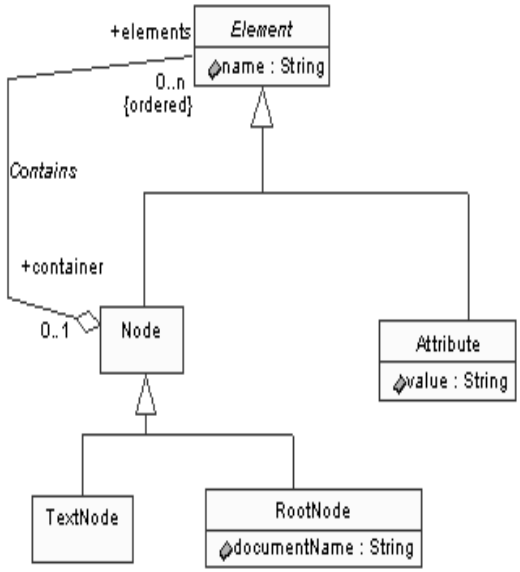

FIGURE C-2 Simple XML Metamodel

The whole metamodel is defined in a MOF package named XMLModel. Each XML document consists of nodes (in the model represented by Node class) and each node can contain either attributes (represented by Attribute class) or nested nodes. Both attribute and node have name (this

and the Contains association are the reasons why there is a root abstract class named Element), in addition attribute has a value. Contains association expressing the fact that a node can contain both attributes and nodes is ordered on the elements end – this is because order of nodes is significant in XML. Text nodes are represented by a separate subclass of Node named TextNode. As each XML document consists of exactly one root node, it is convenient to define a separate subclass of Node class called RootNode for representing a root node in a document. This class also represents the document itself – it contains an additional attribute documentName.

Note – Please see the JSR-40 website for the complete XML description of the above metamodel (in XMI 1.2 format for MOF 1.4) and the corresponding generated JMI interfaces. Additional examples are also available for reference.

---

t Generated JMI API

Let's look at the generated JMI API for our metamodel of XML.

JMI defines that for each metamodel class two interfaces are generated. One representing a static context of the class (class proxy interface) and the other representing instances of the class (instance interface).

The instance interface is very straightforward. It contains methods for accessing instance-level attributes and references and invoking instance-level operations.

Following is the instance interface for Element class:

```
public interface Element extends javax.jmi.reflect.RefObject {
  public String getName();
  public void setName(String newValue);
  public Node getContainer();
  public void setContainer(Node newValue);
}
```

The interface contains methods for getting and setting name of an element (which is a modeled attribute of type String) and for getting and setting container (reference of type Node).

The class proxy interface contains operations for creating instances of a corresponding class (if the class is not abstract) and methods for accessing classifier-level attributes and invoking classifier-level operations of the class.

For Attribute class from our metamodel it looks as follows:

```
public interface AttributeClass extends javax.jmi.reflect.RefClass {
  public Attribute createAttribute();
  public Attribute createAttribute(String name, String value);
}
```

As the class has no classifier-level features, the interface contains only the factory methods for creating new instances. Let's look at how the class proxy interface looks like for the Element class which is abstract:

```
public interface ElementClass extends javax.jmi.reflect.RefClass {
}
```

As you can see the interface is empty. This is because the class contains no classifier-level features and the factory methods do not make sense for abstract classes.

Another kind of interfaces are the association proxy interfaces. These are generated for each association in a model. In our simple metamodel of XML there is only one association – Contains. The generated association proxy interface for it looks like this:

```
public interface Contains extends javax.jmi.reflect.RefAssociation {
 public boolean exists(Element element, Node container);
 public java.util.List getElements(Node container);
 public Node getContainer(Element element);
 public boolean add(Element element, Node container);
 public boolean remove(Element element, Node container);
}
```

Finally there is a package proxy interface generated for each package in a metamodel. In our case for the XMLModel package. It contains methods for accessing all proxy objects for all directly nested or clustered packages and contained classes and associations:

```
public interface XMLModelPackage extends javax.jmi.reflect.RefPack-
age {
 public NodeClass getNode();
 public AttributeClass getAttribute();
 public ElementClass getElement();
 public RootNodeClass getRootNode();
 public Contains getContains();
}
```

t Using generated JMI interfaces

Let's suppose that someone wants to create a utility that takes an XML document as input, parses it and creates metadata for it in the JMI repository generated from the metamodel of XML described above. Such a tool would typically use some existing XML parser and use its output to populate the repository using the generated JMI interfaces.

As JMI defines no standard API for obtaining access to the metadata, for simplicity we will suppose that our tool already obtained a reference to a package proxy for the XML metamodel – this will serve us as an entry point to the metadata service:

```
XMLModelPackage service = <obtain reference to the package proxy>
```

The reference to the package proxy is obtained in a vendor specific way (typically using some kind of lookup mechanism).

The first thing that the tool needs to do when it starts to create metadata for some XML document is to create the document's root node. This can be done by the following piece of code:

```
RootNode root = service.getRootNode().createRootNode(<name>, <docu-
mentName>);
```

The code above first obtains a reference to the class proxy object of RootNode class (by calling getRootNode on the package proxy) and then creates a new instance by calling createRootNode method on the class proxy and passing name of the node and name of the document to it.

Now the tool needs to create other nodes and attributes as it goes through the XML document. The node creation consists of two steps. First a new instance of Node has to be created and then it needs to be added to its parent node as a contained element:

```
Node node = service.getNode().createNode(<name>);
node.setContainer(<parentNode>);
```

Similarly a new attribute can be created:

```
Attribute attr = service.getAttribute().createAttribute(<name>,
<value>);
attr.setContainer(<parentNode>);
```

Let's suppose we have the following XML document named test.xml:

```
<test><node attr1 = "value1" attr2 = "value2">text</node></test>
```

Metadata for this particular document can be created by the following sequence of JMI calls:

```
// create the root node
RootNode root = service.getRootNode().createRootNode("test",
"test.xml");
// create the first node
Node node = service.getNode().createNode("node");
// set root as its container
node.setContainer(root);
// now create the two attributes
Attribute attr1 = service.getAttribute().createAttribute("attr1",
"value1");
Attribute attr2 = service.getAttribute().createAttribute("attr2",
"value2");

// add the two attributes as contained elements of node
```

```
// (this is an alternative way of setting the container)
node.getElements().add(attr1);
node.getElements().add(attr2);
// create the text node
TextNode text = service.getTextNode().createTextNode("text");
// set node as its container
text.setContainer(node);
```

Besides writing a tool for populating the repository by XML metadata, we may also want to write a tool for generating XML documents from existing metadata in a repository. Here is a simple example of how it is possible to use the generated JMI API for traversing through the existing metadata:

```
public void generateXML(Node parent, java.io.PrintStream stream)
  throws java.io.IOException {
  // write the current node
  if (parent instanceof TextNode) {
    // write text node
    stream.print(parent.getName());
  }
  else {
    // write node opening
    stream.print("<" + parent.getName() + ">");
    // write contained nodes
    for (Iterator it=parent.getElements().iterator(); it.hasNext();) {
      Object element = it.next();
      if (element instanceof Node) {
        generateXML((Node) element, stream);
      }
    }
    // write node ending
    stream.print("</" + parent.getName() + ">");
  }
}
```

The example above shows a method that generates XML described by the metadata of the passed node into the print stream passed as the second parameter. To keep it simple this method generates only nodes (i.e. it ignores the attributes). Also everything is generated into a single line of text without any indentation.

t Using JMI reflective interfaces

All the functionality exposed by the generated JMI interfaces is also available via the JMI reflective interfaces. The reflective interfaces are less convenient to use, however they enable someone to write generic JMI tools (such as XMI Reader/Writer or a generic metadata browser) that operate on any metamodel. Sample code below shows how it is possible to use the reflective API to read values of all attributes of a given object:

```
import javax.jmi.reflect.RefObject;
import javax.jmi.model.MofClass;
import javax.jmi.model.ModelElement;
import java.util.Iterator;

public void readAttributes(RefObject object) {
  // metaobject of any RefObject is an instance of MofClass
  MofClass metaObject = (MofClass) object.refMetaObject();

  // explore the object's metamodel to get object's attrs.
  for (Iterator it=metaObject.getContents().iterator(); it.hasNext();) {
    ModelElement element = (ModelElement) it.next();
    if (element instanceof javax.jmi.model.Attribute) {
      // print the attribute's name
      System.out.print(element.getName() + " = ");
      // get the value of the attribute
      Object value = object.refGetValue(element);
      // print the attribute value
      System.out.println(value.toString());
    }
  }
}
```

Please note that the code above ignores the inherited attributes. In addition the values of multivalued attributes are not printed correctly (the code should check the multiplicity and if the attribute is multivalued it should iterate through the returned collection).

For the root node from one of the previous examples the output of the readAttributes would look like this:

```
documentName = test.xml
```

If the readAttributes would be improved to go also through all inherited attributes, the output would also include the following line:

```
name = test
```

The examples discussed in this section were aimed to give a short overview of basic usage of JMI generated and reflective API.

APPENDIX**D**

Example: Data Warehousing Scenario

This scenario illustrates how JMI can be used in the implementation of a data warehouse. Specifically, this example illustrates how a data warehouse can be modeled and automatically constructed using metadata-driven tools, where JMI is the pervasive access mechanism for managed metadata. This scenario consists of five distinct steps, and the example illustrates how JMI is used to support each step:

1. Metadata service initialization. A centralized metadata service supporting a common metamodel is generated and brought online.
2. Model construction. The metadata service is used to create shared metadata based on the common metamodel. This metadata is then published to the rest of the environment.
3. Tool initialization. Each tool constructs its own internal information structures and links to other tools, based on shared metadata published via the central metadata service.
4. Metamodel interoperability. A new tool that supports a different metamodel is introduced to the data warehouse. JMI reflection is used to reconcile differences between the tool-specific metamodel and the common metamodel used by the rest of the data warehouse.
5. Data warehouse information flow. The overall flow of data and control through the data warehouse is illustrated. These flows are metadata-driven and facilitated by JMI programmatic interfaces (both metamodel-specific and reflective) and by the XMI bulk import/export mechanism supported by JMI.

t Metadata Service Initialization

In the first step of the scenario, a data warehouse administrator initializes a JMI-enabled metadata service as the central metadata store for the data warehouse. The metadata service is some general-purpose tool that provides a complete implementation of the JMI specification. An implementation of the JMI specification might, for example, consist of some server process that realizes the JMI interfaces corresponding to some MOF-compliant metamodel, the JMI reflective interfaces, and the XMI reader and writer interfaces. Precisely how a JMI-enabled metadata service is implemented is not, of course, prescribed by the JMI specification.

The administrator's warehouse management tool connects to the JMI service via the J2EE Connector Architecture's Custom Client Interface. This example assumes that the Connection interface has several custom methods that facilitate access to the JMI service. These methods are implementation-specific (i.e., not prescribed by JMI itself) and are defined as follows:

```
public interface com.mdservice.Connection {
  public void close() throws com.mdservice.ResourceException;
```

```
  public javax.resource.cci.ConnectionMetaData getMetaData() throws
    com.mdservice.ResourceException;
  public javax.jmi.reflect.RefPackage getTopLevelPackage() throws
    com.mdservice.ResourceException;
  public javax.jmi.xmi.XmiReader getXmiReader() throws com.mdser-
    vice.ResourceException;
  public javax.jmi.xmi.XmiWriter getXmiWriter() throws com.mdser-
    vice.ResourceException;
}
```

For the purpose of this example, assume that the getTopLevelPackage() returns an empty outer-most extent that is an instance of Model.Package (i.e., a package for holding a MOF compliant metamodel). The Connection interface also provides the custom factory methods getXmiReader() and getXmiWriter(). These methods are used for instantiating the XmiReader and XmiWriter interfaces, respectively, within the context of the connected metadata service.

A data warehouse administration client first connects to the JMI service via a J2EE Connector and obtains an empty outer-most extent by calling the getTopLevelPackage() method. Next, the client creates an instance of XmiReader and invokes the read() method on the XmiReader interface, supplying both the RefPackage returned by the previous call to getTopLevelPackage(), and a URL pointing to an XMI document containing the MOF metamodel to be loaded. In this case, the XMI document contains the definition of the Object Management Group's Common Warehouse Metamodel (CWM), rendered as MOF Model instance.

Once the CWM metamodel is loaded, the JMI service generates a complete set of metamodel-specific (tailored) Java interfaces, according to the JMI mapping templates, as well as a corresponding library of Java implementation classes. These implementation classes constitute the realization of the CWM server. Note, once again, that this particular implementation strategy of the JMI service is not itself defined by the JMI specification.

The sequence of JCA and JMI calls required to initialize the JMI service as described above are given by the following code fragment:

```
ConnectionFactory cxf = new ConnectionFactory();
Connection cx = cxf.getConnection(properties);
RefPackage msTlp = Connection.getTopLevelPackage();
XmiReader xmiReader = Connection.getXmiReader();
xmiReader.read("http://www.omg.org/cwm/1.0/cwm_1.0.xmi", msTlp);
```

t Model Construction

Now that the JMI service has been initialized with the CWM metamodel, we can begin to construct a model of our data warehouse based on CWM. This is illustrated in the figure below:

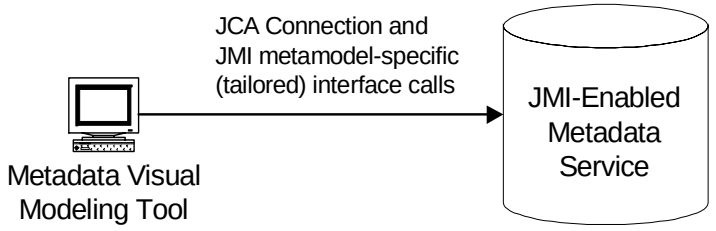

FIGURE D-1 Model Construction

Here, a CWM-aware visual modeling tool is used to construct the model instances. The modeling tool connects to the JMI service via JCA. The modeling tool queries the JMI service via the Connection for its outermost RefPackage extent. From this point forward, the modeling tool directs its activities against local (client-side) JMI interfaces that are specific to the CWM metamodel. The user might, for example, drill-down on the outermost package to discover the metamodel-specific packages clustered in this package. These would consist of packages such as Warehouse Process, Warehouse Operation, Transformation, OLAP, etc. Each is a RefPackage instance corresponding to one of CWM metamodel packages shown in the block diagram of Figure D-2. The end user constructs a CWM model by selecting various meta classes from each of the CWM packages and creating instances of them using the JMI interfaces. For example, the user might create an OLAP Dimension by selecting a Dimension meta class icon from a diagram representing the OLAP metamodel, and then dragging it onto a display area.

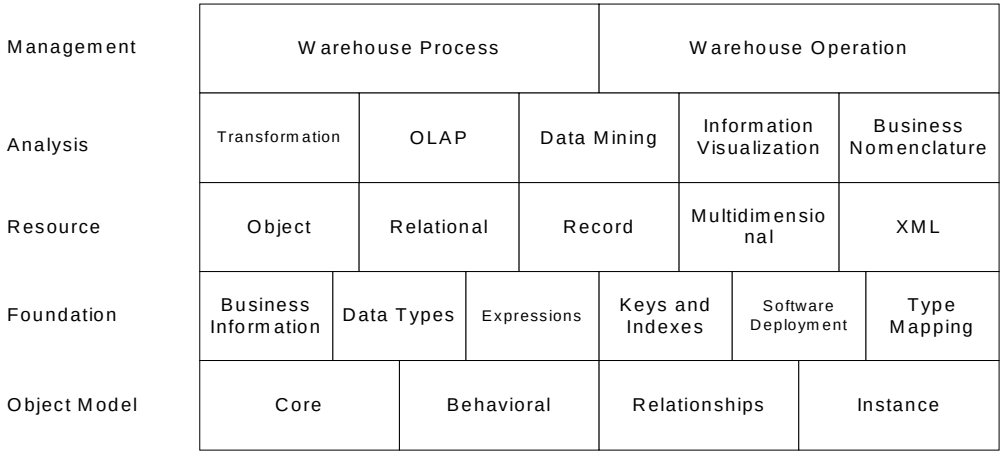

FIGURE D-2 CWM Metamodel

The user, therefore, constructs a CWM model (that is, an instance of the CWM metamodel) of the complete data warehouse, through navigation and discovery of the metamodel structure of CWM. Internally, the modeling tool performs local calls against the CWM metamodel-specific JMI interfaces. For example, the following code fragment illustrates several of the JMI calls that might be performed in the construction of a CWM OLAP Dimension instance:

```
// Get the DimensionClass proxy
org.omg.java.cwm.analysis.olap.DimensionClass dc = olapPkg.getDimen-
sion();
// Create a Time Dimension
org.omg.java.cwm.analysis.olap.Dimension timeDim = dc.createDimension();
timeDim.setName("Time");
timeDim.setTime(true);
```

Now, let's assume that the completed data warehouse model consists of instances of modeling elements defined in the following CWM packages:

- Record package: Use to model the raw data feeds supplying information to the data warehouse.
- Relational package: Used to construct models of both the operational data store (ODS) and dimensional star-schema analysis database of the data warehouse.
- Transformation package: Used to model the data transformations going from the raw data source to the ODS, as well as from the ODS to the dimensional star-schema database. These models serve as the primary descriptions of any extract, transform, and load (ETL) process that the data warehouse might implement.
- OLAP package: Used to model the OLAP (Online Analytical Processing) abstractions exposed by the data warehouse for analysis and reporting. This model includes a mapping to the relational star-schema model, as a source of OLAP data.
- Warehouse Process and Warehouse Operation packages: Used to model the data warehouse control processes that manage and track any ETL activities that might be performed, based on the Transformation models.

The complete data warehouse model is stored in a single RefPackage that is an instance of the CWM metamodel, and is now available to the rest of the data warehousing environment via the JMI interfaces of the central JMI service.

t Tool Initialization

Now that the data warehouse schema has been completely defined in terms of a centrally stored CWM model, the data warehouse can be physically generated. This is done through the metadata initialization of a number of JMI-enabled data warehousing tools. This is illustrated in the figure below.

A data warehouse administration tool (possibly a back-end to the modeling tool described earlier) is used to initialize each of the installed data warehousing tools with CWM metadata. Each Tool would peruse the content of the outermost RefPackage extent, navigating the model structure through JMI interface calls, load the particular metadata that it requires (i.e., once each tool has discovered an appropriate RefPackage instance within the CWM model), via either the JMI programmatic interfaces or the XmiReader and XmiWriter interfaces.

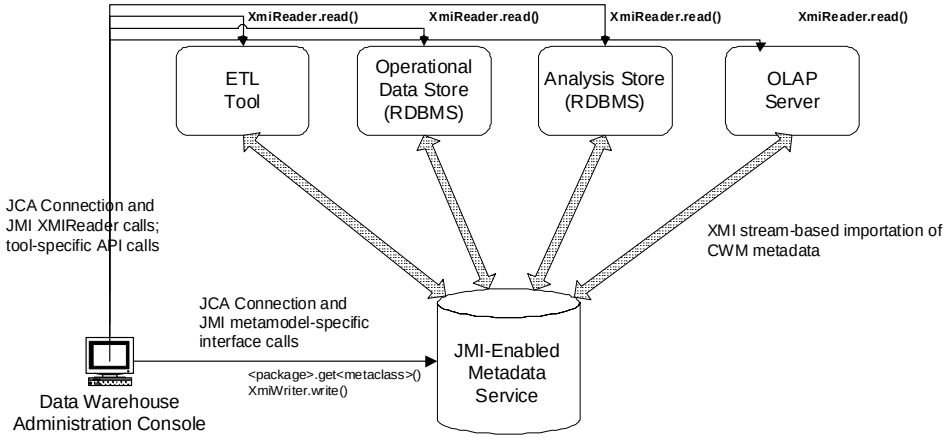

FIGURE D-3 Schema Generation

t Metamodel Interoperability

This portion of the data warehouse scenario describes advanced functionality that can be realized through MOF/JMI reflection. In this case, the users of the OLAP server have acquired an advanced, multidimensional visualization/reporting software package. Like the other tools comprising the data warehouse, the new visualization tool is also JMI-enabled. However, it uses a different MOF metamodel than CWM; specifically, some MOF-compliant metamodel that represents very specific aspects of advanced, multidimensional, visual analysis. The new tool needs to be integrated with the rest of the data warehouse, and, in particular, since the tool supports metadata-driven data integration with OLAP servers, it must be integrated at the metadata level with the CWM OLAP model used by the OLAP server.

To effectively integrate the new visualization tool with the rest of the environment, the data warehouse modeler first connects to the JMI service and then creates an instance of the CWM Visualization model to represent the visualization concepts, as they relate to the OLAP model. That is, visualization metadata is constructed and linked to the various OLAP model objects that are to be visualized (e.g., Cube, Dimension).

Now, the new visualization tool needs to be integrated at the metadata-level with the CWM Visualization model just created. However, although the visualization tool is JMI/MOF-compliant, it does not understand CWM. It is driven by its own MOF metamodel. The data warehouse modeler uses the visualization tool's set up screen/front end to first build the tool's metadata. The modeler then reconciles both types of metadata by constructing a programmatic script (written in

Java) that connects to the JMI service and uses JMI Reflective programming to acquire equivalent metaobjects from the CWM model. In this manner, the visualization tool is driven by its own metadata, but can dynamically map to an equivalent CWM Visualization model, and therefore indirectly to the OLAP model elements that are to be rendered. This level of indirection provided by JMI Reflection enables the advanced visualization/reporting tool to perform metadata-directed processing of OLAP data, even though its metamodel is different from CWM.

t Summary: Data Warehouse Information Flow

The diagram below shows the overall flow of information through the integrated data warehouse. The clear arrows represent the general flow or progression of data through the data warehouse, all the way from the ODS to the advanced visualization/reporting software. This data flow is inherently metadata-driven, and metadata in this environment has a single and centralized representation in terms of JMI. Shared metadata is defined by a MOF-compliant metamodel (i.e., CWM), but the JMI-enabled metadata service is not tied to any particular metamodel, and is capable of loading the CWM metamodel and dynamically generating an internal implementation of CWM.

Communication of shared metadata is achieved through the JMI interfaces. XmiReader and XmiWriter interfaces are used to transfer complete models or specific packages of models in a bulk format for loading into tools. On the other hand, metamodel-specific (CWM, in this case) JMI interfaces are used by client tools for browsing and possibly creating or modifying existing metadata structures. Finally, JMI Reflection is used to facilitate metadata integration between tools whose metamodels differ, but otherwise are MOF-compliant.

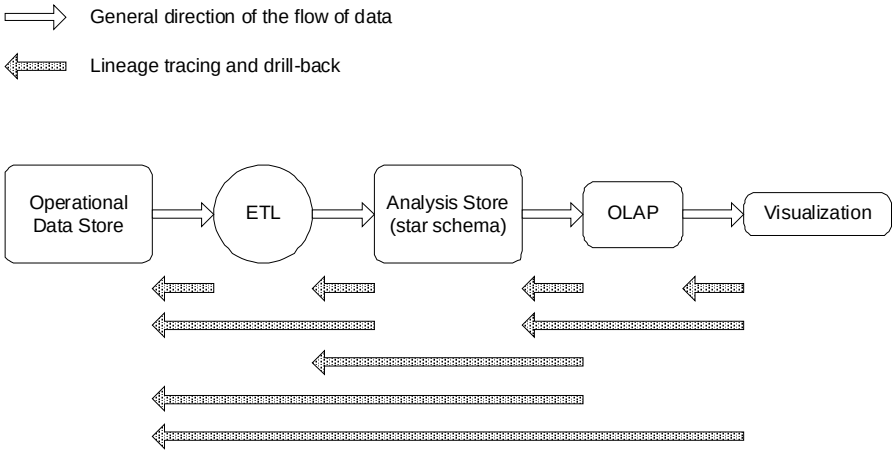

FIGURE D-4 Data Warehouse Information Flow