

Java™ Metadata Interface(JMI) Specification

JSR 040

Technical Comments:
jmi-comments@sun.com

Version 1.0 Update
Public Review Draft Specification
20-November-01

Specification Lead:
Sridhar Iyengar,
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

Copyright © 2001 Unisys Corporation. All rights reserved.
Copyright © 2001 Hyperion Solutions. All rights reserved.
Copyright © 2001 IBM Corporation. All rights reserved.
Copyright © 2001 Oracle. All rights reserved.
Copyright © 2001 SAS Institute. All rights reserved.
Copyright © 2001 Sun Microsystems, Inc.. All rights reserved.
Copyright © 2001 Rational Software. All rights reserved.
Copyright © 2001 Sybase. All rights reserved.
Copyright © 2001 Novosoft. All rights reserved.
Copyright © 2001 Adaptive Ltd.. All rights reserved.
Copyright © 2001 IONA. All rights reserved.
Copyright © 2001 DSTC. All rights reserved.
Copyright © 2001 Perfekt-UML. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or documentation may be reproduced in any form by any means without prior written authorization of the copyright holders, or any of their licensors, if any. Any unauthorized use may be a violation of domestic or international law.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the U.S. Government and its agents is subject to the restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

Trademarks

Sun, Sun Microsystems, the Sun logo, Java, JavaBeans, Enterprise JavaBeans, JavaChip, JavaStation, JavaOS, Java Studio, Java WorkShop, Solaris, Solaris for Intranets, Solaris for ISPs, Solstice Enterprise Manager, Sun Internet Administrator, Sun Internet FTP Server, Sun Internet Mail Server, Sun Internet News Server, Sun Internet Services Monitor, SunScreen, Sun WebServer, and The Network Is The Computer are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. Information subject to change without notice.

OMG, OBJECT MANAGEMENT GROUP, CORBA, CORBA ACADEMY, CORBA ACADEMY & DESIGN, THE INFORMATION BROKERAGE, OBJECT REQUEST BROKER, OMG IDL, CORBAFACILITIES, CORBASERVICES, CORBANET, CORBAMED, CORBADOMAINS, GIOP, IIOP, OMA, CORBA THE GEEK, UNIFIED MODELING LANGUAGE, UML, and UML CUBE LOGO are registered trademarks or trademarks of the Object Management Group, Inc.

PostScript is a registered trademark of Adobe Systems, Inc.

All other product or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.



Please
Recycle



Adobe PostScript

Contents

1. Introduction	1
Metadata Interoperability	1
The Java Metadata Interface Specification	1
Platforms	2
Target Audience	2
JMI Expert Group	3
Acknowledgements	3
2. JMI Overview	5
The MOF Four-Layered Architecture	5
The MOF Interfaces	6
Introduction to JMI	7
Some JMI Use-Cases	7
3. An Overview of the MOF Model	9
The MOF Model	10
MOF Model Elements	10
MOF Model Associations	20
Discrepancies between JMI and MOF	23
XMI	23
4. MOF to Java Mapping	25
Metaobjects and Interfaces	25
Metaobject Type Overview	25
The Metaobject Interface Hierarchy	26
Computational Semantics for the Java Mapping	28
Equality in the Java Mapping	28
The Java NULL Value	28
JMI Collection Semantics	28
Lifecycle Semantics for the Java Mapping	29
Association Access and Update Semantics for the Java Mapping	31
Attribute Access and Update Semantics for the Java Interface Mapping	32
Reference Semantics for the Java Mapping	35
Cluster Semantics for the Java Mapping	36
Atomicity Semantics for the Java Mapping	36
The Supertype Closure Rule	36
Primitive Data Type mapping	37
Exception Framework	37

Preconditions for Java Interface Generation	38
Standard Tags for the Java Mapping	38
Tags for Specifying Package Prefix	39
Tags for Providing Substitute Identifiers	40
Tag for specifying prefix for generated methods	41
Tags for Capturing JavaDocs	41
Java Generation Rules	41
Rules for Generating Identifiers	42
Rules for Splitting MOF Model.ModelElement Names into Words	42
Literal String Values	43
Generation Rules for Attributes, AssociationEnds, References, Constants, and Parameters	43
Java Mapping Templates	44
Package Interface Template	45
Class Proxy Template	47
Instance Template	49
Association Template	50
Attribute Template	53
Reference Template	56
Operation Template	60
Exception Template	62
Constant Template	62
AliasType Template	62
CollectionType Template	63
StructureType Template	63
EnumerationType Template	63
Constraint Template	65
Annotation Template	65
JavaDocs Template	65
5. MOF Reflective Package	67
Introduction	67
The Reflective Classes and Interfaces	68
RefBaseObject	68
RefFeatured	72
RefAssociation	74
RefPackage	77
RefClass	83
RefObject	85
Reflective Interfaces for Data Types	88
RefEnum	88
RefStruct	90
RefAssociationLink	91
RefException Class	92
The Exception Framework	93
JmiException	93
AlreadyExistsException	94
ClosureViolationException	95

CompositionCycleException	96
ConstraintViolationException	96
DuplicateException	97
InvalidCallException	98
InvalidObjectException	98
NotFoundException	99
OutOfBoundsException	100
TypeMismatchException	101
WrongSizeException	102
XMI Import/Export in JMI	103
XMIWriter	103
XmiReader	104
F. The JMI APIs for the MOF	105
G. Accessing a JMI Service using the Connector Architecture	107
H. Examples	109

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 current draft of the MOF 1.4 specification is located at:

<http://cgi.omg.org/cgi-bin/doc?ptc/2001-08-22>

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:

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

1.5 JMI Expert Group

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

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

1.6 Acknowledgements

The JMI Expert Group wishes to thank the OMG MOF Revision Task Force for their valuable contributions, and for their efforts to reconcile the discrepancies between JMI and MOF.

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 a layered metadata architecture consisting of a single meta-metamodel (M3), metamodels (M2) and models (M1) of information (see FIGURE 2-1).

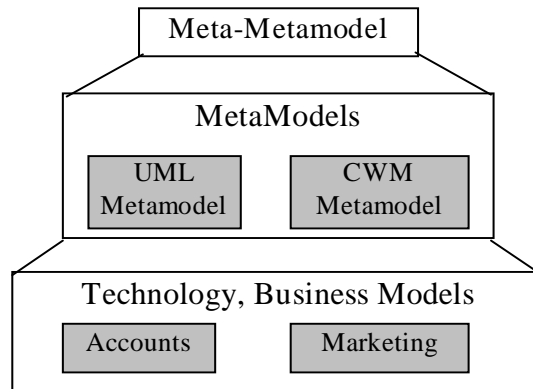


FIGURE 2-1 The layered architecture of the MOF.

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:

n *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.

n *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.

n *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.

n *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 its public metadata is a JMI service.

JMI provides the following to the J2EE environment:

- n A metadata framework that provides a common Java programming model for accessing metadata.
- n An integration and interoperability framework for Java tools and applications.
- n 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 Some JMI Use-Cases

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

The Data Warehouse Management Scenario

Note – To be completed for final specification.

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 manage 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:

- n 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.
- n The complexity of integration has been reduced from $N \times N$ where N is the number of tools being integrated, to $M \times M$ 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.

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 metamodel 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://cgi.omg.org/cgi-bin/doc?ptc/01-08-22>>.

3.1 The MOF Model

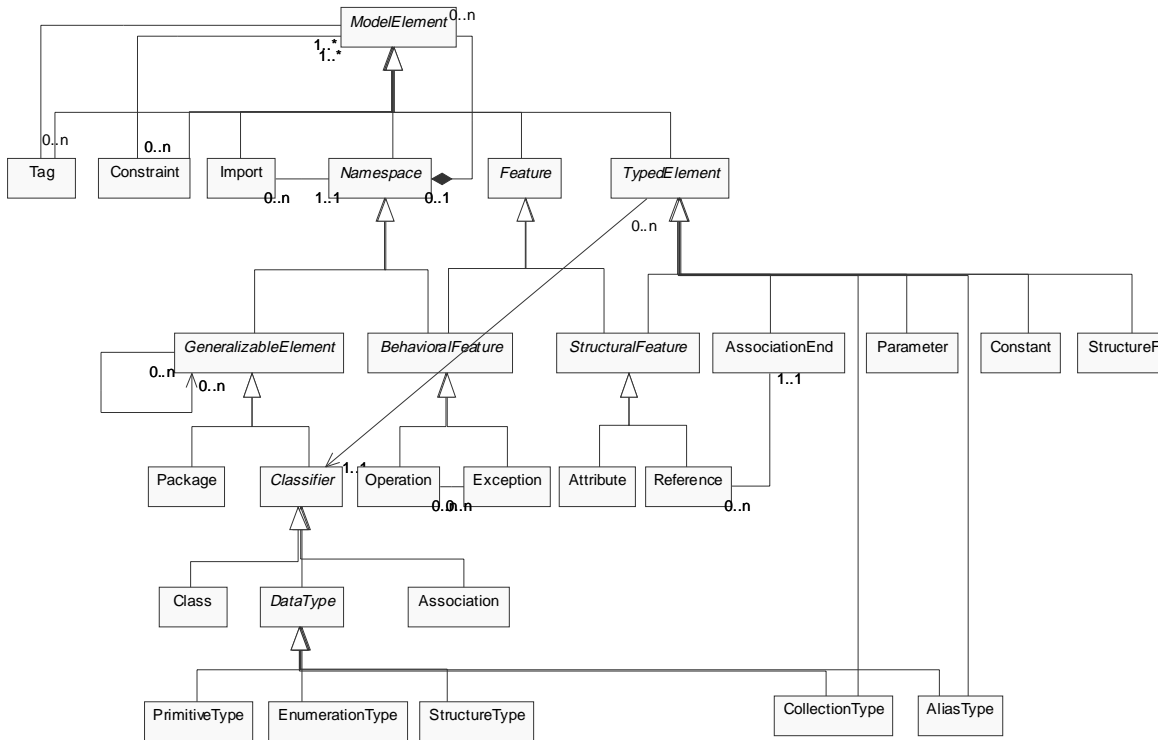


FIGURE 3-2 The MOF Model - overview

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.

- n 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.

- n 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 3-2 “The MOF Model - overview” on page 10 shows the inheritance hierarchy for MOF. There is a common superclass, ModelElement from which everything inherits. FIGURE 3-2 “The MOF Model - overview” on page 10 shows some of the higher-level superclasses in more detail to illustrate what gets inherited:

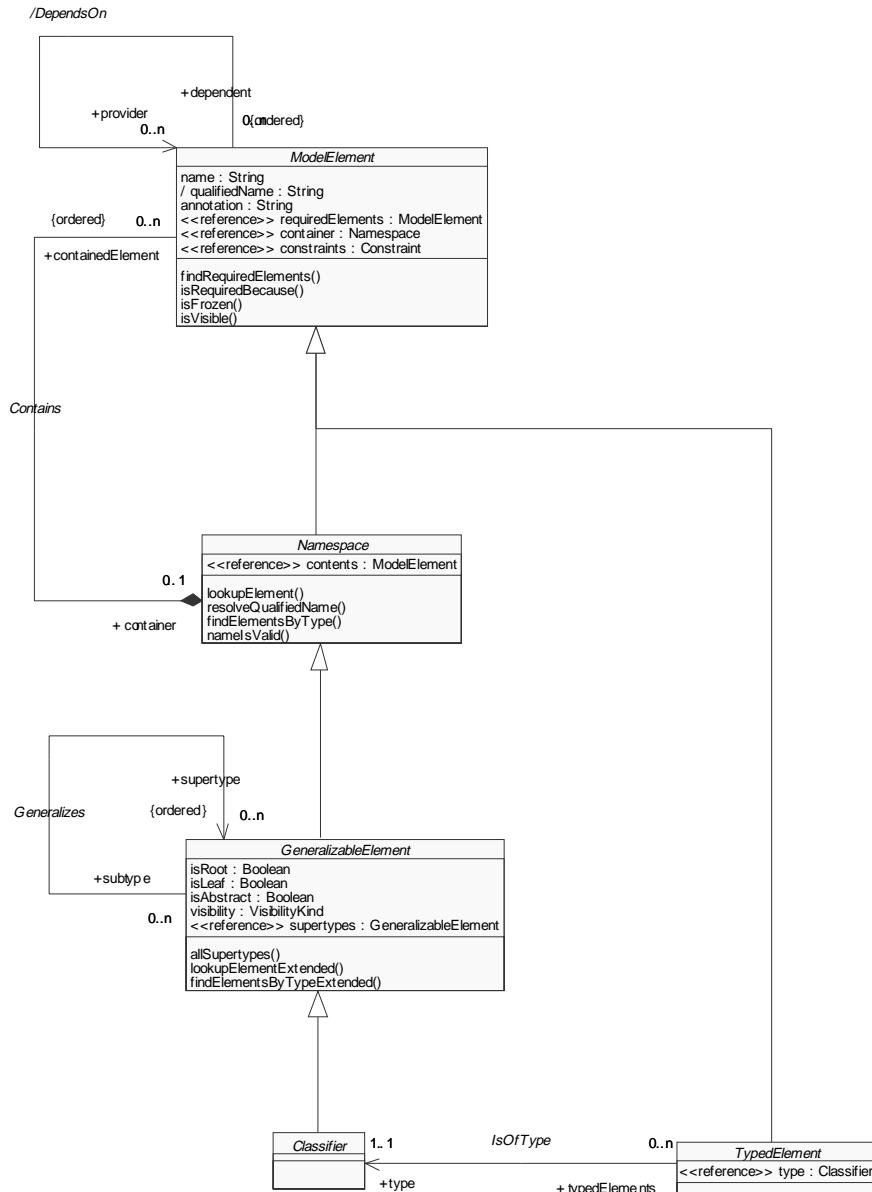


FIGURE 3-3 MOF Common Superclasses

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 3-13.

3.1.2 Containment Hierarchy

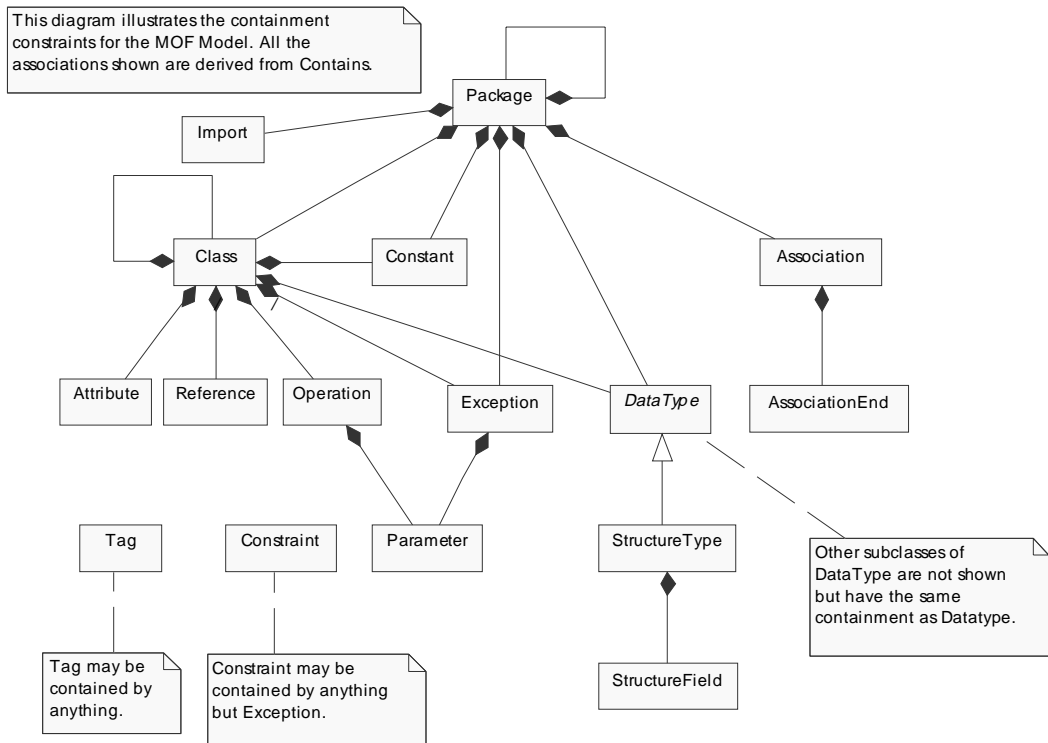


FIGURE 3-4 MOF Containment Hierarchy

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 3-4 “MOF Containment Hierarchy” on page 13, 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

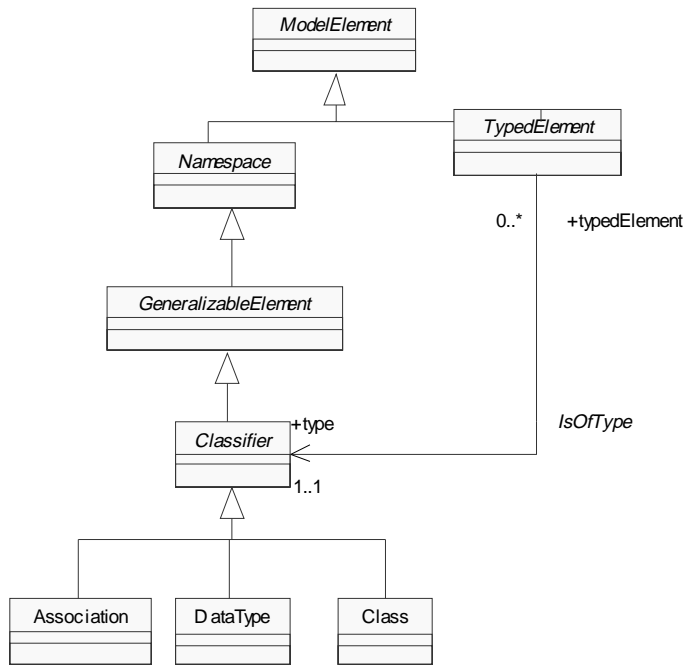


FIGURE 3-5 MOF Types

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

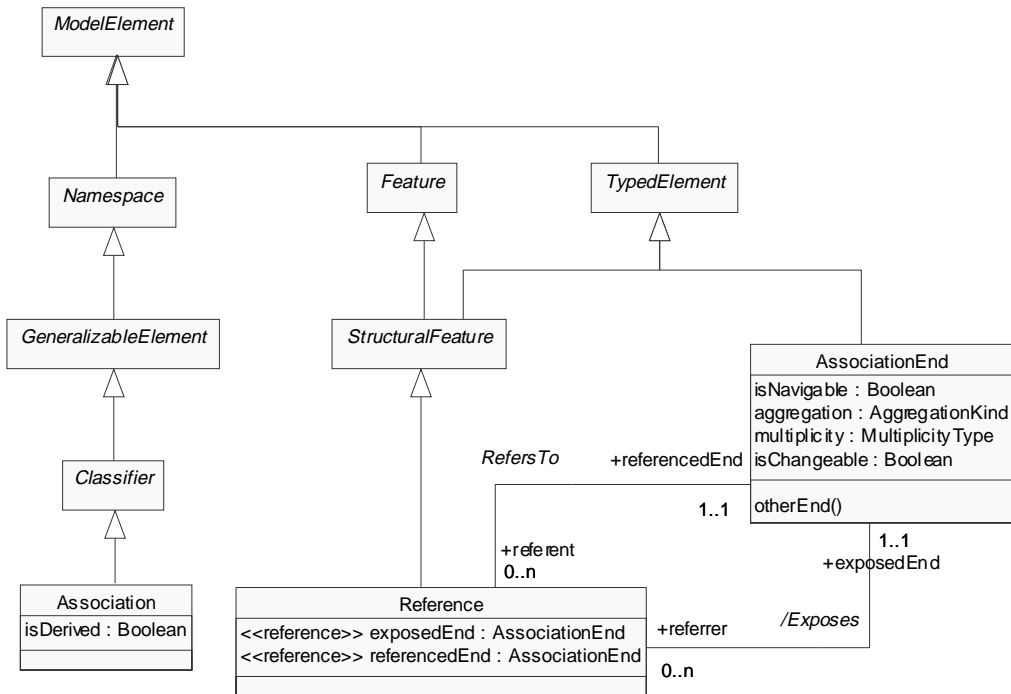


FIGURE 3-6 MOF Associations

FIGURE 3-6 “MOF Associations” on page 15 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

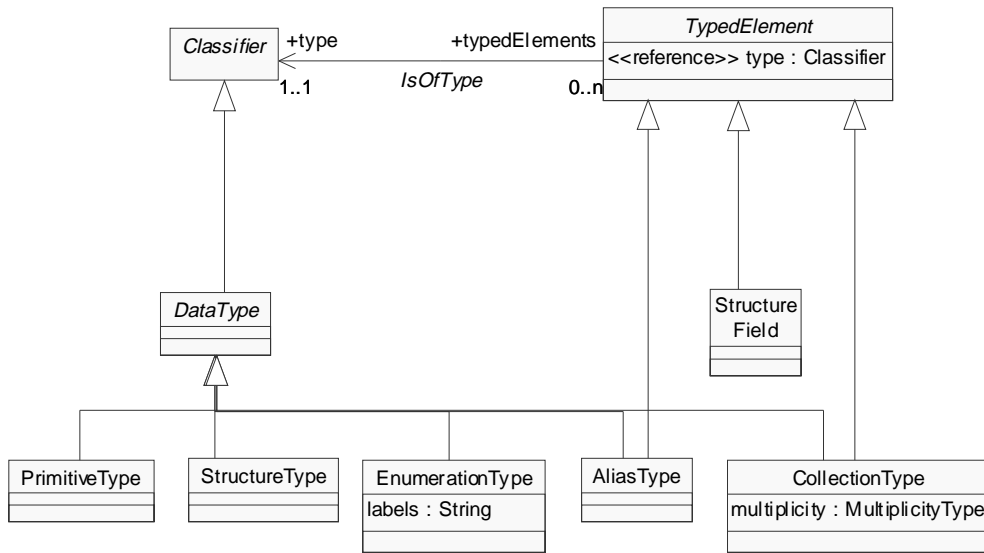


FIGURE 3-7 MOF DataTypes

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

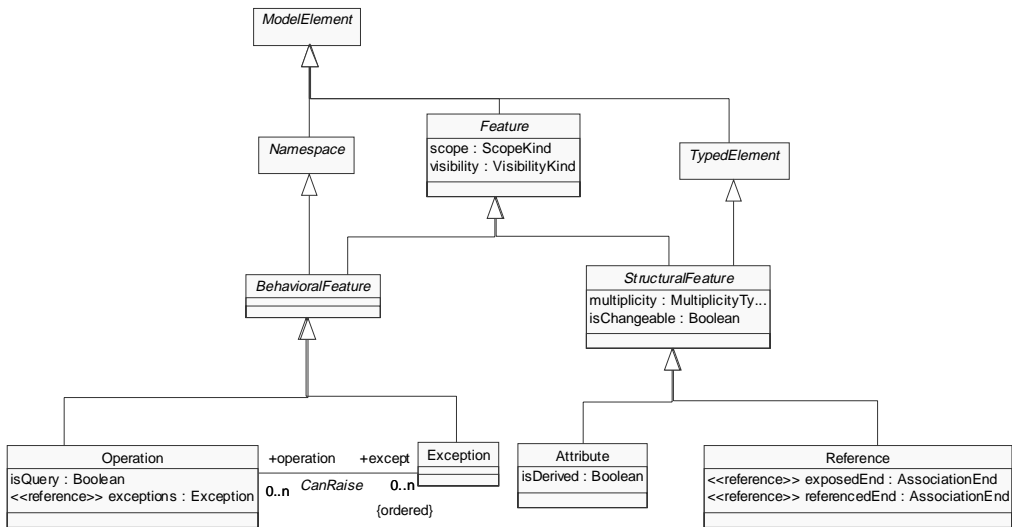


FIGURE 3-8 MOF Features

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 3-8 “MOF Features” on page 17.

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

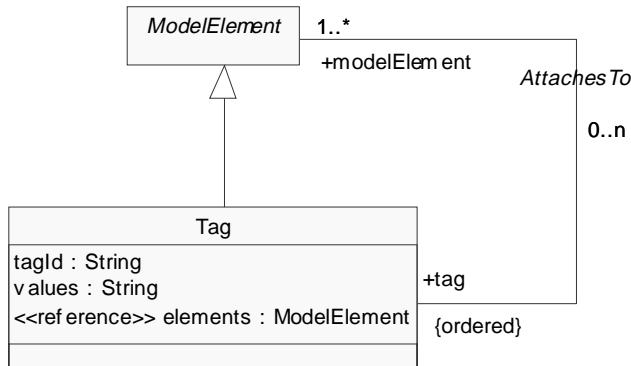


FIGURE 3-9 MOF Tags

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

The ModelElement is the root class within the MOF. It represents the elementary constructs of models.

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 outer most 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 namespace 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 namespace. 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

The “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

The “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 is comprised of model elements.

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

t Classifier

The “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 from the information in the metamodel.
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

t EnumerationType

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

Abstract No.

Super classes DataType.

Attributes None.

References None.

Operations None.

MOF Constraints None.

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” 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” 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” 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. isChangable : 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 from the information in the metamodel.
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” 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. isChangable : 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 elements structure depends on.

end1: **dependent** This end identifies the dependent element.

end1Class : ModelElement

multiplicity : zero or more

end2: This end identifies the elements that the “dependent” end depends on.

end1Class : 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 : ModelElement

multiplicity : one or more

end2: **tag** Identifies the tags attached to a model element.

end1Class : 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.

end1Class : ModelElement

multiplicity : zero or more

t Aliases

“Aliases” identifies the imported Namespace.

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

t Constrains

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

<i>end1</i> : constraint	Identifies the constraints. end1Class : Constraint multiplicity : zero or more
<i>end2</i> : constrainedElement	Identifies the constrained elements. end1Class : ModelElement multiplicity : one or more

t Generalizes

The “Generalizes” association identifies a supertype/subtype relationship.

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

t IsOfType

The “IsOfType” association identifies the type of a typed element.

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

t CanRaise

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

<i>end1</i> : operation	The set of operations that can raise this exception. end1Class : Operation multiplicity : zero or more
<i>end2</i> : except	The set of exceptions that this operation can raise. end1Class : Exception multiplicity : zero or more

t RefersTo

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

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

t Exposes

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

<i>end1</i> : referer	Identifies the referencing reference. end1Class : Reference multiplicity : zero or more
<i>end2</i> : exposedEnd	The reference’s owning classifier’s end in the association. end1Class : 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:

- n The Supertype Closure rule has been modified to reflect the changes to the reflective framework.
- n 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.
- n Collections are ‘live’ (see “JMI Collection Semantics” on page 38”).

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 available on the OMG web page <<http://cgi.omg.org/cgi-bin/doc?ptc/01-08-27>>.

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.

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 outer most package extent (a.k.a. outer most 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 outer most extent and are created using the accessors provided by the MOF.

Class Proxy Objects

A class proxy object serves a number of purposes:

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

- n Operations to access and update the instance-scoped and classifier-scoped attributes.
- n Operations corresponding to instance-scoped and classifier-scoped operations.
- n 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:

- n Operations for querying the link set.
- n Operations for adding, modifying and removing links from the set.
- n 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 4-10. 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.

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:

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

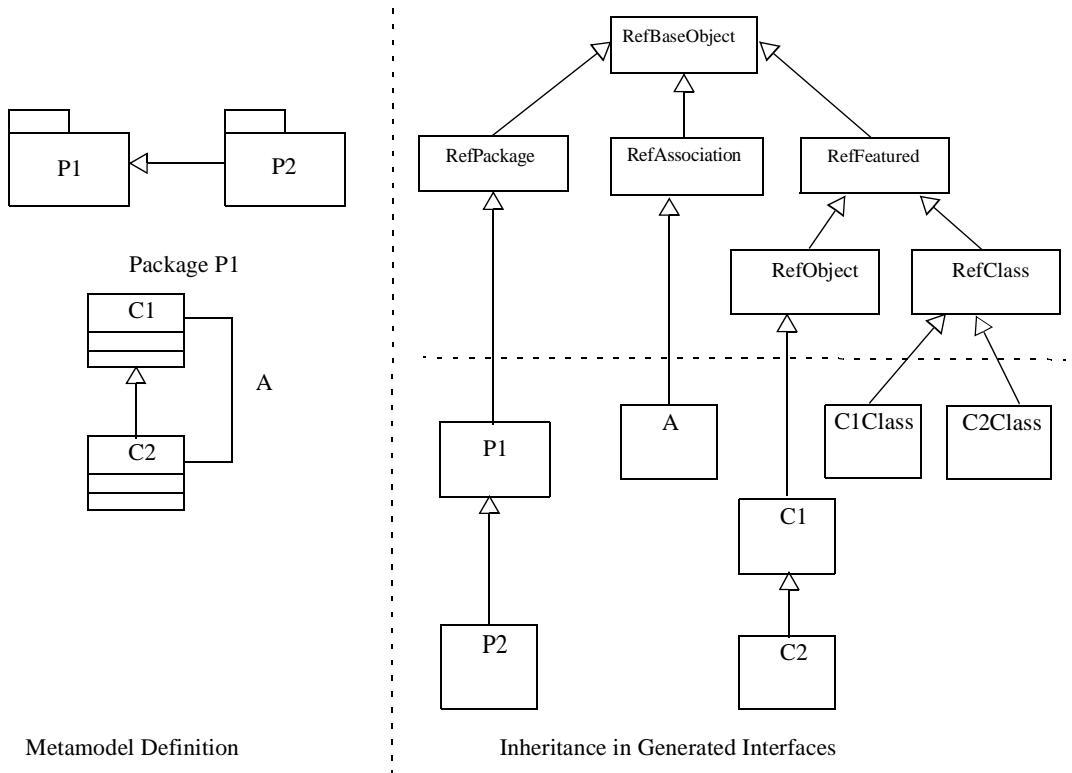


FIGURE 4-10 Generated Java inheritance patterns.

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:

- n An Instance object that has no supertypes extends RefObject; all other Instance objects extend their supertypes.
- n A Package object that has no supertypes extends RefPackage; all other Package objects extend their supertypes.
- n All Class Proxy objects extend RefClass.
- n 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 implement the (reflective) `RefBaseObject` interface. Equality of JMI objects should be implemented as follows:

- n 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:

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

In addition, the collections behave as follows:

- n Any change in the source collection is immediately reflected in the live Collection.
- n Any change in the live collection is immediately reflected in the source collection.
- n 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 through an exception.
- n 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 outer most 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 outer most extent is created, instances of the following dependent objects are automatically created along with the package object:

- n A package object is created for each nested package within the outer most package extent.
- n A package object is created for each clustered package within the outer most package extent.
- n A class proxy object is created for each class within the outer most package extent.
- n A association object is created for each association within the outer most 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 46). 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:

- n When a client invokes the reflective "refDelete" operation on the instance object.
- n When the outer most extent containing the instance object is deleted.
- n 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:

- n The binding between the instance object and its object reference(s) must be revoked.
- n The instance object must be removed from its class proxy object's container.
- n Any instance objects that are components of the object being deleted must also be deleted.
- n 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:

- n Using the operations on the association objects.
- n Using the operations corresponding to references on instance objects.
- n By deleting one or other linked instance objects.
- n 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 outer most package extent boundaries.

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

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

- n Deletion of an instance object in another outer most package extent may cause a association link set to contain links that are no longer valid.
- n 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:

- n The “get<associationEndName>” operations return a collection containing the projection of the corresponding end of the association’s link set.
- n 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”:

- n For an unordered association, the “add” operation adds a link to the association’s link set.
- n For an ordered association, the “add” and “addBefore” operations both add a link to the association’s link set. In the “add” case, the new link is added the end of the existing link set. In the “addBefore” case, the new link is added immediately before the link selected by the “before” argument

A number of constraints apply to the link addition operations:

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

Link Modification Operations

There are two “modify” operations for modifying an existing link in the association’s link set. The semantics of the “modify” operations depend on whether the association has an ordered “AssociationEnd”:

- n If the association is ordered and the *ordered* end is being modified, then the position of the link being modified does not change.
- n In all other cases (i.e., the association is unordered, or it is ordered and the unordered end is being modified), the modify operation is equivalent to deleting the existing link and adding the modified link.

A number of constraints apply to the link modification operations:

- n The modify operation modifies existing links. However, the link need not be a valid link. That is, the modify operation can modify an existing link that has been made invalid (by say, the deletion of an instance object).
- n The modified link must be a valid link.
- n The modified link cannot already be a member of the link set.
- n A “modify” 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:

- n The operation cannot remove a link if doing so would violate the multiplicity constraint on either end.
- n 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:

- n The new value supplied must be an existing instance object.
- n The new value (i.e. the component instance) must not already be a component of another instance object.
- n The composite and component instance objects must belong to the same outer most 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:

- n 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.
- n 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:

- n The new element must be an existing object of the attribute’s base type.

- n The new element must not be a component of another object.
- n The composite and every component instance objects must belong to the same outer most package extent; i.e. the Composition Closure rule must not be violated.

Changeability and Derivedness

If “isChangeable” is set to true 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:

- n When the attribute has aggregation semantics of “composite”:
 - n The Composition Closure rule means that the class proxy object and attribute value instances must belong to the same extent.
 - n 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:

- n 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.
- n If the default “create<ClassName>” operation (that takes no arguments) is used, then the attributes will be initialized as follows:
 - n 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.
 - n 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.
- n “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	<code>i.get<referenceName>();</code>	<code>a.<referencedEndName>(i);</code>
optional and single-valued	<code>i.set<referenceName>(new);</code>	<code>old = a.<referenceEndName>(i);</code> if <code>old.size > 0</code> then <code>a.modify<referenceEndName>(i, old[0], new);</code> else <code>a.add(i, new);</code>
multivalued	<code>c.addAll(new)</code>	<code>old = a.<refEndName>(i);</code> for (<code>int j = 0; J < old.size; j++</code>) <code>a.remove(i, old[j]);</code> for (<code>int j = 0; J < old.size; j++</code>) <code>a.add(i, new[j]);</code>
multivalued	<code>c.add(new)</code>	<code>a.add(i, new);</code>
multivalued	<code>c.remove(old)</code>	<code>a.remove<referenceEndName>(i, old);</code>

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:

- n If an operation succeeds, state changes required by the specification should be made, except as noted below:
 - n When an instance object is deleted, deletion of any component instance objects may occur asynchronously.
 - n When an instance object is deleted, removal of links to the deleted instance object may occur asynchronously.
- n 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.C1 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. A

Primitive Types	Default Java Type
Boolean	boolean
Byte	byte
Boolean	boolean
Double	double
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:

- ⁿ *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.
- ⁿ *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 103.

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:

- n The MOF Model constraints must all be satisfied for the input metamodel.
- n The input metamodel must be structurally consistent.
- n The visible names within a namespace must conform to the standard Java identifier syntax, or they must have a valid Java substitute name.
- n The visible names within a namespace must be unique after name substitution and other name mangling specified in the mapping.
- n A class may not be nested within another class.
- n A class may not be imported.
- n Model Elements in a metamodel cannot be cyclically dependent except as follows:
 - n A dependency cycle consisting of one or more classes is legal, provided they all have the same container.
 - n 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 is 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
<i>tag id:</i>	A string that denotes the semantic category for the tag.
<i>attaches to:</i>	Gives the kind(s) of Model.ModelElement that this category of tag can be meaningfully attached to.

TABLE 4-3 Notation for describing Standard Tags

Tag	Properties
<i>values:</i>	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.)
<i>meaning:</i>	Describes the meaning of the tag in this context.
<i>Java interface generation:</i>	Defines the tag's impact on the generated Java interfaces.
<i>restrictions:</i>	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 – There is currently an issue as to whether a tag attached to a metamodel element but not contained by that metamodel should be considered in the Java mapping.

4.6.1 Tags for Specifying Package Prefix

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

Package Prefix

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

4.6.2 Tags 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.

<i>tag id:</i>	“javax.jmi.substituteName”
<i>attaches to:</i>	Model.ModelElement
<i>values:</i>	a String
<i>meaning:</i>	The String is the substitute name to be used in place of the model element’s name.
<i>java interface generation:</i>	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.
<i>restrictions:</i>	The preconditions defined in “Preconditions for Java Interface Generation” on page 48 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.

<i>tag id:</i>	“javax.jmi.methodPrefix”
<i>attaches to:</i>	Model.Package
<i>values:</i>	a string. Default value is the empty string.
<i>meaning:</i>	This tag supplies a user defined prefix for the generated operations for a model (package)
<i>java interface generation:</i>	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.
<i>restrictions:</i>	The tag value should be a valid prefix for a Java operation name.

4.6.4 Tags for Capturing JavaDocs

This tag allows the modeler to store JavaDocs in the metamodel. It applies to packages (i.e., instances of MOF Package), classes (i.e., instances of Class), and operations (i.e., instances of Operation).

<i>tag id:</i>	“javax.jmi.javaDocs”
<i>attaches to:</i>	Model.Class, Model.Package, and Model.Operation
<i>values:</i>	a string
<i>meaning:</i>	The string value captures the JavaDocs associated with the respective object.
<i>java interface generation:</i>	The string precedes the class, package, or operation
<i>restrictions:</i>	Should be a valid JavaDocs comment

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 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:

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

<i>Package names:</i>	The identifier consists of lower-case alphabetic characters only.
<i>Class proxy names:</i>	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.
<i>Operation names:</i>	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.
<i>Attribute names:</i>	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.
<i>Constants</i>	The identifier consists of all upper-case alphabetic characters and the “_” character (used to separate words).
<i>Enumeration literals</i>	The identifier consists of all upper-case alphabetic characters and the “_” character (used to separate words).

4.7.2 Rules for Splitting MOF Model.ModelElement Names into Words

ToBeDone – This section needs to spell out the rules for splitting words as in the MOF specification.

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 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. The Java mapping needs to convert all ModelElement names to valid Java identifiers with minimum exceptions.

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, associationend, 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 outer most 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.

Template

```
<<JAVADOCS 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.JmiException;
// for each EnumType directly contained by the package
public <EnumTypeName> create<EnumTypeName>(String value) throws
javax.jmi.reflect.JmiException;
};
```

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

t `create<EnumTypeName>()`

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

reflective analog:

return type: <AssociationName>
parameters: String value.
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.

Template

```
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.JmiException;
public <ClassName> create<ClassName> (/* for each non-derived direct or
inherited attribute <AttributeType> <attributeName> */) throws
javax.jmi.reflect.JmiException;
// for each StructType directly contained by the package
public <StructTypeName> create<StructTypeName>(/*for each attribute
<AttributeType> <attributeName> */) throws javax.jmi.reflect.JmiException;
// for each EnumType directly contained by the package
public <EnumTypeName> create<EnumTypeName>(String value) throws
javax.jmi.reflect.JmiException;
```

```

// 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:

<i>return type:</i>	<className>
<i>parameters:</i>	none
<i>exceptions</i>	JmiException (AlreadyExistsException, ClosureViolationException)

t create<ClassName>()

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

reflective analog:

<i>return type:</i>	<className>
<i>parameters:</i>	in <AttrType> <attrName>,...
<i>exceptions</i>	JmiException (AlreadyExistsException, ClosureViolationException)

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

- n A class's superclasses are processed before the class's attributes.
- n Superclasses are processed in the order of the "Generalizes" association.
- n The attributes of each class or superclass are processed in the order of the "Contains" association.
- n When an attribute is encountered with a "scope" value of "classifier_level" or an "isDerived" value of true no parameter is generated.
- n When an attribute is encountered a second or subsequent time, no additional parameter is generated.
- n 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 54.

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

t create<EnumTypeName>()

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

reflective analog:

return type: <AssociationName>
parameters: String value.
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.

Template

```
<<JAVADOCS 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 a 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.

Template

```
public interface <AssociationName> extends javax.jmi.reflect.RefAssociation {
    public boolean exists(<AssociationEnd1ClassName> <AssociationEnd1Name>,
        AssociationEnd2ClassName> <AssociationEnd2Name>) throws
        javax.jmi.reflect.JmiException;
    //If associationEnd1 is single valued and isNavigable
    public <AssociationEnd1ClassName> get<AssociationEnd1Name>
        (ssociationEnd2ClassName> <AssociationEnd2Name>) throws
        javax.jmi.reflect.JmiException;
    //If associationEnd1 is multivalued, isOrdered is false, and isNavigable
    public Collection get<AssociationEnd1Name> (ssociationEnd2ClassName>
        <AssociationEnd2Name>) throws javax.jmi.reflect.JmiException;
    //If associationEnd1 is multivalued, isOrdered, and isNavigable
    public List get<AssociationEnd1Name> (ssociationEnd2ClassName>
        <AssociationEnd2Name>) throws javax.jmi.reflect.JmiException;
    //If associationEnd2 is single valued and isNavigable
    public <AssociationEnd2ClassName> get<AssociationEnd2Name>
        (ssociationEnd1ClassName> <AssociationEnd1Name>) throws
        javax.jmi.reflect.JmiException;
    //If associationEnd2 is multivalued, isOrdered is false, and isNavigable
    public Collection get<AssociationEnd2Name> (ssociationEnd1ClassName>
        <AssociationEnd1Name>) throws javax.jmi.reflect.JmiException;
    //If associationEnd2 is multivalued, isOrdered, and isNavigable
    public List get<AssociationEnd2Name> (ssociationEnd1ClassName>
        <AssociationEnd1Name>) throws javax.jmi.reflect.JmiException;
    //If associationEnd1 and associationEnd2 isChangable
    public void add(AssociationEnd1ClassName> <AssociationEnd1Name>,
        AssociationEnd2ClassName> <AssociationEnd2Name>) throws
        javax.jmi.reflect.JmiException;
    //If associationEnd1 and associationEnd2 isChangable
    public void remove(<AssociationEnd1ClassName> <AssociationEnd1Name>,
        AssociationEnd2ClassName> <AssociationEnd2Name>) throws
        javax.jmi.reflect.JmiException, javax.jmi.reflect.NotFoundException;
};
```

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 <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 <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: none

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.

DuplicateException occurs when the link set for the current association already contains the link whose creation is requested.

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: none

parameters: <AssocEnd1ClassName> <assocEnd1Name>
 <AssocEnd2ClassName> <assocEnd2Name>

exceptions: JmiException (NotFoundException)

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.

"NotFoundException" occurs if the link to be deleted does not exist in 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.

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:

- n Operations for instance-scoped attributes appear in the instance interface only.
- n 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:

- n Single-valued attributes: multiplicity bounds are [1..1].
- n Optional-valued attributes: multiplicity bounds are [0..1].
- n 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.JmiException;
// if single-valued attribute
public <AttributeType> <AccessorName> () throws javax.jmi.reflect.JmiException;
// if upper > 1 and isOrdered = false
public Collection get<AttributeName> () throws javax.jmi.reflect.JmiException;
// if upper > 1 and isOrdered = true
public List get<AttributeName> () throws javax.jmi.reflect.JmiException;
// Mutator Operations
// if optional-valued and isChangeable = true
public void <MutatorName> (<AttributeType> newValue) throws
javax.jmi.reflect.JmiException;
// if single-valued attribute and isChangeable = true
public void <MutatorName> (<AttributeType> newValue) throws
javax.jmi.reflect.JmiException;

```

Operations

t get<AttributeName>

The get<AttributeName> (or the equivalent <AccessorName>) operation returns the value of the named attribute.

<i>reflective analog:</i>	refGetValue(...);
<i>return type:</i>	[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
<i>parameters:</i>	none
<i>query:</i>	yes
<i>exceptions:</i>	JmiException

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

- n 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.
- n 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.isEmpty() = true. No exception is raised in this case.

The “*set<AttributeName>*” (or equivalent *<MutatorName>*) operation sets the value of the named attribute. This operation is not generated for multivalued attributes.

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

`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 outer most 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.

t *remove<AttributeName>*

The “*remove<AttributeName>*” operation removes an existing member from a multivalued attribute.

<i>reflective analog:</i>	<code>refRemoveValue(...)</code>
<i>return type:</i>	none
<i>parameters:</i>	<i><AttributeType></i> <code>oldElement</code>
<i>exceptions:</i>	<code>JmiException</code> (<code>NotFoundException</code>)

The “*remove<AttributeName>*” operation removes an occurrence of the value passed in the “`oldElement`” parameter.

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 semantics are defined below:

The “add(Object o) operation

n `NullPointerException` must be thrown if object “o” is null.

- n ClosureViolationException, CompositionCycleException are only possible when the type of the attribute is a class, and the attribute has “composite” aggregation semantics.
- n DuplicateException must be thrown if the attribute’s multiplicity has isUnique set to true, and the new element value is equal to an element of the attribute’s current value set.
- n ClosureViolationException must be thrown when the new object belongs to a different outer most package extent.
- n CompositionCycleException must be thrown when the operation would result in the object becoming a direct or indirect component of itself.
- n InvalidObjectException must be thrown when some instance object is found to be non-existent or inaccessible.

The “add(int index, Object element)” operation

- n NullPointerException, ClosureViolationException, DuplicateException, ClosureViolationException, CompositionCycleException, InvalidObjectException must be thrown in similar situations to those described for “add(Object o)” above.
- n NotFoundException must be thrown if the index is no longer valid.

The “set(int index, Object element)” operation

- n NullPointerException, ClosureViolationException, DuplicateException, ClosureViolationException, CompositionCycleException, and InvalidObjectException must be thrown in similar situations to those described for “add(Object o)” above.
- n NotFoundException must be thrown in similar situations to those described for “add(int index, Object element)” above.
- n If the attribute has “isOrdered” set to true, the operation must preserve the initial order of the collection’s elements.

The “remove(Object o)” operation

- n NullPointerException must be thrown if object “o” is null.
- n NotFoundException must be thrown if the “oldElement” value is not present in the attribute’s value list.

The “remove(int index)” operation

- n NotFoundException must be thrown if the index is no longer valid.

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:

- n Single-valued references: multiplicity bounds are [1..1].
- n Optional-valued references: multiplicity bounds are [0..1].
- n 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.

<i>reflective analog:</i>	refGetValue(...);
<i>return type:</i>	[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
<i>parameters:</i>	none
<i>exceptions:</i>	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:

- n In the [0..1] case, the operation returns the projected instance object if there is one; else it returns “null”.
- n 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.
- n 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.

<i>reflective analog:</i>	refSetValue(...)
<i>return type:</i>	none
<i>parameters:</i>	in <ReferenceClass> newValue
<i>exceptions:</i>	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 semantics are defined below:

The “add(Object o)” operation

- n NullPointerException must be thrown if object “o” is null.
- n DuplicateException must be thrown if the operation would create a duplicate link in the link set for the referenced association.
- n ClosureViolationException must be thrown when the new value belongs in a different outermost extent.
- n CompositionCycleException must be thrown when the referenced association has composite aggregation semantics, and the update would make the object a component of itself.

The “add(int index, Object o)” operation

- n NullPointerException, DuplicateException, ClosureViolationException and CompositionCycleException must be thrown in similar situations to those described for the “add(Object o)” operation above.
- n NotFoundException must be thrown if the index is no longer valid.
- n InvalidObjectException must be thrown if either new object is a non-existent, null or inaccessible instance object.

The “set(int index, Object element)” operation

- n NullPointerException, DuplicateException, ClosureViolationException and CompositionCycleException must be thrown in similar situations to those described for the “add(Object o)” operation above.
- n InvalidObjectException and NotFoundException must be thrown in similar situations to those described for the “add(int index, Object o)” operation above.

The “remove(Object o)” operation

- n NullPointerException must be thrown if object “o” is null.
- n NotFoundException must be thrown if the link to be deleted is not found.

The “remove(int index)” operation

n NotFoundException must be thrown if the index is no longer valid.

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
<<JAVADOC 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.

<i>reflective analog:</i>	refInvokeOperation(...)
<i>return type:</i>	<GeneratedReturnType> <param_name>
<i>parameters:</i>	<GeneratedParamType> <ParameterName>,...
<i>exceptions:</i>	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 54.

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 54.

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.

- n `DuplicateException` occurs when a multivalued parameter has “isUnique” set to true, and the supplied collection contains a duplicate.
- n `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

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

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

ToBeDone – Resolution to I0063 is that the empty constructor needs to be generated only for exceptions that do not have required parameters.

Note – *<GeneratedParameterType>* conforms to the rules defined in the section “Generation Rules for Parameters” on page 54. *<AccessorName>* conforms to the name generation pattern for attributes (see “Accessor Operations” on page 63).

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

```
<<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 “Accessor Operations” on page 63).

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 Template

For each enumeration type defined in the metamodel, JMI generates an interface and an implementation class.

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

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

```

private static final List typeName;
private final String literalName;

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

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

public String toString() {
    return literalName;
}

public List refTypeName() {
    return typeName;
}

public boolean equals(Object o) {
    if (o == null) {
        return false;
    } else {
        return ((o instanceof javax.jmi.reflect.RefEnum) &&
            ((javax.jmi.reflect.RefEnum) o).refTypeName().equals(typeName)
            && ((javax.jmi.reflect.RefEnum) o).toString().equals(literal-
            Name));
    }
}
}
}

```

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> */
```

4.8.16 JavaDocs Template

The JavaDocs template generates JavaDocs for instances of Model.Package, Model.Class, and Model.Operation. If the javax.jmi.JavaDocs tag is an empty or null string but the ModelElement.Annotation contains some value, then the JavaDocs template may be replaced by the annotation template.

Template

```
// JavaDocs may optionally be suppressed by the Java code
// generator
// If the javax.jmi.JavaDocs tag contains a value
/**
 * <javax.jmi.JavaDocs>
 */
// else if ModelElement.Annotation contains some value, the
// JavaDocs template may be replaced by the annotation
// template defined above.
```


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:

- n Create, update, access, navigate and invoke operations on class proxy objects.
- n Query and update links using association objects.
- n 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:

- n Find an object's metaobject.
- n Find an object's container(s) and enclosing package(s).
- n Test for object identity.
- n 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.

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.

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

n None (root object)

Operations

t refMofId

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

<i>specific analog:</i>	none
<i>return type:</i>	string
<i>isQuery:</i>	yes
<i>parameters:</i>	none
<i>exceptions:</i>	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.

<i>specific analog:</i>	none
<i>return type:</i>	RefObject
<i>isQuery:</i>	yes
<i>parameters:</i>	none
<i>exceptions:</i>	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.

<i>specific analog:</i>	none
<i>return type:</i>	RefPackage
<i>isQuery:</i>	yes
<i>parameters:</i>	none
<i>exceptions:</i>	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.

t refOutermostPackage

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

<i>specific analog:</i>	none
<i>return type:</i>	RefPackage
<i>isQuery:</i>	yes
<i>parameters:</i>	none
<i>exceptions:</i>	none

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

t refVerify

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

<i>specific analog:</i>	none
<i>return type:</i>	Collection
<i>isQuery:</i>	yes
<i>parameters</i>	boolean deepVerify
<i>exceptions:</i>	none

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

When deepVerify is true, the refVerify 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, then an empty collection is returned; otherwise, a list of all RefObjects, within the specified containment, that violate constraints is returned.

The Collection returned from the refVerify 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 refVerify(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

n 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.

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

The result for the "refGetValue" operation is encoded as per section "Generation Rules for Parameters" on page 54.

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

t refSetValue

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

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

The “value” parameter must be encoded as per the section “Generation Rules for Parameters” on page 54.

InvalidCallException is raised when one of the following conditions occur:

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

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.

<i>specific analog:</i>	none
<i>return type:</i>	java.lang.Object
<i>parameters:</i>	RefObject requestOperation (or String operationName), List args
<i>exceptions:</i>	JmiException (InvalidCallException, 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 54. `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`” (or “`operationName`”) does not designate an operation that can be invoked.

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 JmiException;
    public java.lang.Object refGetValue(String featureName) throws JmiException;
    public java.lang.Object refInvokeOperation(RefObject requestedOperation, 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 manage 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

n RefBaseObject

Operations

t refAllLinks

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

<i>specific analog:</i>	none
<i>return type:</i>	Collection
<i>isQuery:</i>	yes
<i>parameters:</i>	none
<i>exceptions:</i>	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.

<i>specific analog:</i>	exists(...);
<i>return type:</i>	boolean
<i>isQuery:</i>	yes
<i>parameters:</i>	RefObject endOne, RefObject endTwo
<i>exceptions:</i>	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”.

<i>specific analog:</i>	<code><AssociationEndName> (...);</code>
<i>return type:</i>	Collection
<i>isQuery:</i>	yes
<i>parameters:</i>	RefObject queryEnd (or String queryEndName) RefObject queryObject
<i>exceptions:</i>	JmiException (InvalidCallException, 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.

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.

<i>specific analog:</i>	<code>add(...);</code>
<i>return type:</i>	none
<i>parameters:</i>	RefObject endOne, RefObject endTwo.
<i>exceptions:</i>	JmiException (InvalidCallException, WrongSizeException, DuplicateException, ClosureViolationException, CompositionCycleException, TypeMismatchException, InvalidObjectException, java.lang.NullPointerException)

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

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.

<i>specific analog:</i>	remove(...);
<i>return type:</i>	none
<i>parameters:</i>	RefObject endOne, RefObject endTwo.
<i>exceptions:</i>	JmiException (WrongSizeException, TypeMismatchException, java.lang.NullPointerException)

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

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(AssociationEnd queryEnd, RefObject queryOb-
        ject) throws JmiException;
    public Collection refQuery(String queryEndName, RefObject queryObject)
        throws JmiException;
    public void refAddLink(RefObject endOne, RefObject endTwo) throws
        JmiException;
    public void refRemoveLink(RefObject endOne, RefObject endTwo) throws
        JmiException, NotFoundException;
}
```

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

n RefBaseObject

Operations

t refGetClass

The “refGetClass” 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)

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

InvalidCallException is raised if the “type” (or “className”) parameter does not designate a valid class. |

t refGetAssociation

The “refGetAssociation” 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)

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

InvalidCallException is raised if the “association” (or “associationName”) parameter does not designate a valid association. |

t refGetPackage

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

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

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” (or “nestedPackageName”) parameter does not designate a valid package.

t refGetAllPackages

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

<i>specific analog:</i>	None
<i>return type:</i>	Collection of RefPackages
<i>isQuery:</i>	yes
<i>parameters:</i>	None
<i>exceptions:</i>	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 refGetAllClasses

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

<i>specific analog:</i>	None
<i>return type:</i>	Collection of RefClasses
<i>isQuery:</i>	yes
<i>parameters:</i>	None
<i>exceptions:</i>	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 refGetAllAssociations

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

<i>specific analog:</i>	None
<i>return type:</i>	Collection of RefAssociations
<i>isQuery:</i>	yes
<i>parameters:</i>	None
<i>exceptions:</i>	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.

<i>specific analog:</i>	<code>create<Struct>(…);</code>
<i>return type:</i>	RefStruct
<i>parameters:</i>	RefObject structType (or String structName), List args
<i>exceptions:</i>	JmiException (WrongSizeException, TypeMismatchException, InvalidObjectException, 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 54.

t refCreateEnum

This “refCreateEnum” operations create a new instance of the enumeration.

<i>specific analog:</i>	<code>create<EnumTypeName>(…);.</code>
<i>return type:</i>	RefEnum
<i>parameters:</i>	RefObject enumType (or String “enumName”) String literalName
<i>exceptions:</i>	JmiException (TypeMismatchException, java.lang.NullPointerException)

The “refCreateEnum” operation creates 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.

t refDelete

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

<i>specific analog:</i>	none
<i>return type:</i>	none
<i>parameters:</i>	none
<i>exceptions:</i>	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 RefObject refGetClass(RefObject type) throws JmiException;
    public RefObject refGetClass(String className) throws JmiException;
    public RefPackage refGetPackage(RefObject nestedPackage) throws JmiException;
    public RefPackage refGetPackage(String nestedPackageName) throws JmiException;
    public RefAssociation refGetAssociation(RefObject association) throws JmiException;
    public RefAssociation refGetAssociation(String associationName) throws JmiException;
    public Collection refGetAllPackages();
    public Collection refGetAllClasses();
    public Collection refGetAllAssociations();
    public RefStruct refCreateStruct(RefObject structType, List args) throws JmiException;
    public RefStruct refCreateStruct(String structName, List args) throws JmiException;
    public RefEnum refCreateEnum(RefObject enumType, String literalName) throws JmiException;
    public RefEnum refCreateEnum(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

n 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.

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

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 “Generation Rules for Parameters” on page 54.

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.

<i>specific analog:</i>	None.
<i>return type:</i>	Collection (unique; unordered)
<i>isQuery:</i>	yes
<i>parameters:</i>	None
<i>exceptions:</i>	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).

<i>specific analog:</i>	None.
<i>return type:</i>	Collection (unique; unordered)
<i>isQuery:</i>	yes
<i>parameters:</i>	None
<i>exceptions:</i>	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.

<i>specific analog:</i>	create<Struct>(…).
<i>return type:</i>	RefStruct
<i>parameters:</i>	RefObject structType (or String structName), List args
<i>exceptions:</i>	JmiException (WrongSizeException, TypeMismatchException, InvalidObjectException, 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 54.

t refCreateEnum

The “refCreateEnum” operations create a new instance of the enumeration.

<i>specific analog:</i>	<code>create<Enumeration>(int).</code>
<i>return type:</i>	<code>RefEnum</code>
<i>parameters:</i>	<code>RefObject enumType (or String enumName)</code> <code>String literalName</code>
<i>exceptions:</i>	<code>JmiException (TypeMismatchException,</code> <code>java.lang.NullPointerException)</code>

The “refCreateEnum” operation creates 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.

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 refCreateEnum(RefObject enumType, String literalName)
        throws JmiException;
    public RefEnum refCreateEnum(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

n 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.

<i>specific analog:</i>	none
<i>return type:</i>	boolean
<i>isQuery:</i>	yes
<i>parameters:</i>	RefObject objType boolean considerSubtypes
<i>exceptions:</i>	none

t refGetClass

This operation returns the RefObject’s class proxy object

<i>specific analog:</i>	none
<i>return type:</i>	RefClass
<i>isQuery:</i>	yes
<i>parameters:</i>	none
<i>exceptions:</i>	none

t refImmediateComposite

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

<i>specific analog:</i>	none
<i>return type:</i>	RefFeatured
<i>isQuery:</i>	yes
<i>exceptions:</i>	none

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

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

<i>specific analog:</i>	none
<i>return type:</i>	RefFeatured
<i>isQuery:</i>	yes
<i>exceptions:</i>	none

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

- n There is a chain of *zero or more* immediate composite relationships (as described for “refInvokeOperation” above) connecting “this” object to C, and
- n 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.

<i>specific analog:</i>	none
<i>return type:</i>	none
<i>parameters:</i>	none
<i>exceptions:</i>	JmiException

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

Interface

```
package javax.jmi.reflect;

public interface RefObject extends RefFeatured {
    public boolean refIsInstanceOf(RefObject objType, boolean considerSub-
        types);
    public RefClass refGetClass();
}
```

```
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:

<i>return type:</i>	List
<i>isQuery:</i>	yes
<i>parameters:</i>	none
<i>exceptions:</i>	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:

<i>return type:</i>	boolean
<i>isQuery:</i>	yes
<i>parameters:</i>	java.lang.Object other
<i>exceptions:</i>	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

n java.io.Serializable

Operations

t refGetValue

The “refGetValue” operation returns the value of an attribute.

<i>specific analog:</i>	get<AttributeName>;
<i>return type:</i>	jav.lang.Object
<i>isQuery:</i>	yes
<i>parameters:</i>	javax.jmi.model.StructureField field
<i>exceptions:</i>	JmiException (InvalidCallException)

This operation returns the value of the specified attribute. The “field” parameter represents the metaobject of the attribute. InvalidCallException occurs in cases where the “field” does not denote a valid attribute.

t refTypeName

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

<i>specific analog:</i>	none
<i>return type:</i>	List
<i>isQuery:</i>	yes
<i>parameters:</i>	none
<i>exceptions:</i>	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.

<i>specific analog:</i>	none
<i>return type:</i>	boolean
<i>isQuery:</i>	yes
<i>parameters:</i>	java.lang.Object other
<i>exceptions:</i>	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 Object refGetValue(RefObject field) throws JmiException;
    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

n 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.

n 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.

```
package javax.jmi.reflect;
public abstract class JmiException extends java.lang.RuntimeException {
    private final RefObject elementInError;
    // default constructor that takes no arguments
    public JmiException() {
        this(null);
    }
}
```

```

// constructor that takes the element in error
public JmiException(RefObject elementInError) {
    this.elementInError = elementInError;
}
// constructor that takes element in error and exception message.
public JmiException(RefObject elementInError,String msg) {
    super(msg);
    this.elementInError = elementInError;
}

public RefObject getElementInError() {
    return elementInError;
}
}

```

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 does not take an error message
    public AlreadyExistsException(RefObject existing) {
        super(existing.refMetaObject());
        this.existing = existing;
    }
    // constructor taking detailed 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

t getViolatingObject

The “getViolatingObject” operation returns the object violating the closure rule.

return type: RefObject

parameters: None

```
package javax.jmi.reflect;
public class ClosureViolationException extends JmiException {
    private final RefObject violatingObject;
    // constructor that takes violating-object and element-in-error as arguments
    public ClosureViolationException(RefObject violatingObject, RefObject
elementInError) {
        super(elementInError);
        this.violatingObject = violatingObject;
    }
    // construct that takes violating-object, element-in-error and
detailed message
    public ClosureViolationException(RefObject violatingObject, RefObject
elementInError, String msg) {
        super(elementInError, msg);
        this.violatingObject = violatingObject;
    }

    public RefObject getViolatingObject() {
        return violatingObject;
    }
}
```

5.4.4 CompositionCycleException

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

Operations

t getCyclicObject

The “getCyclicObject” operation returns the object that is a component of itself.

return type: RefObject

parameters: None

```
package javax.jmi.reflect;
public class CompositionCycleException extends JmiException {
    private final RefObject cyclicObject;
    // constructor taking cyclic-object and element-in-error as arguments
    public CompositionCycleException(RefObject cyclicObject, RefObject
elementInError) {
        super(elementInError);
        this.cyclicObject = cyclicObject;
    }

    // constructor taking cyclic-object, element-in-error and detailed
message
    public CompositionCycleException(RefObject cyclicObject, RefObject
elementInError, String msg) {
        super(elementInError, msg);
        this.cyclicObject = cyclicObject;
    }

    public RefObject getCyclicObject() {
        return cyclicObject;
    }
}
```

5.4.5 ConstraintViolationException

The ConstraintViolationException is thrown when a constraint is violated.

Operations

None.


```
package javax.jmi.reflect;
public class ConstraintViolationException extends JmiException {
    // constructor taking element-in-error as an argument
    public ConstraintViolationException(RefObject elementInError) {
    }
    // constructor taking element-in-error and a detailed message as arguments
    public ConstraintViolationException(RefObject elementInError, String
msg) {
        super(elementInError, msg);
    }
}
```

5.4.6 DuplicateException

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

Operations

t getDuplicatedElement

The “getDuplicatedElement” operation returns the element that was duplicated.

return type: java.lang.Object

parameters: None

```
package javax.jmi.reflect;
public class DuplicateException extends JmiException {
    private final RefObject duplicatedElement;
    // constructor taking duplicate-element and element-in-error
    // as arguments
    public DuplicateException(java.lang.Object duplicatedElement, RefOb-
ject elementInError) {
        super(elementInError);
        this.duplicatedElement = duplicatedElement;
    }
    // constructor taking duplicate-element, element-in-error and
    // detailed message
    public DuplicateException(java.lang.Object duplicatedElement, RefObject
elementInError, String msg) {
        super(elementInError, msg);
        this.duplicatedElement = duplicatedElement;
    }
}
```

```

    public java.lang.Object getDuplicatedElement() {
        return duplicatedElement;
    }
}

```

5.4.7 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 element-in-error as an argument
    public InvalidCallException(RefObject elementInError) {
        super(elementInError);
    }
// constructor taking element-in-error and detailed message
    public InvalidCallException(RefObject elementInError, String msg) {
        super(elementInError, msg);
    }
}

```

5.4.8 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 as an argument
    public InvalidObjectException(RefObject elementInError) {
        super(elementInError);
    }
// constructor taking element-in-error and detailed message
    public InvalidObjectException(RefObject elementInError, String msg) {
        super(elementInError, msg);
    }
}

```

```
}
```

5.4.9 NotFoundException

NotFoundException is thrown by the `addBefore`, `modify`, and `remove` methods to indicate that provided element does not exist.

Operations

t `getMissingElement`

The “`getMissingElement`” operation returns a reference to the element that could not be found in the collection.

return type: `java.lang.Object`

parameters: `None`

```
package javax.jmi.reflect;
public class NotFoundException extends JmiException {
    private final Object missingObject;
    // default constructor taking no arguments
    private NotFoundException() {
        missingObject = null;
    }
    // constructor taking missing-object and element-in-error as arguments
    public NotFoundException(Object missingObject, RefObject elementInError) {
        super(elementInError);
        this.missingObject = missingObject;
    }
    // constructor taking missing-object, element-in-error, and
    // detailed error message
    public NotFoundException(Object missingObject, RefObject elementInError, String msg) {
        super(elementInError, msg);
        this.missingObject = missingObject;
    }

    public Object getMissingObject() {
        return missingObject;
    }
}
```

5.4.10 OutOfBoundsException

OutOfBoundsException is thrown by `addAt`, `modifyAt` and `removeAt` operations when value of parameter representing position of element is out of range.

Operations

t `getPosition`

The “`getPosition`” operation returns the invalid position that caused the exception.

return type: int
parameters: None

t `getSize`

The “`getSize`” operation returns size of the collection.

return type: int
parameters: None

```
package javax.jmi.reflect;
public class OutOfBoundsException extends JmiException {
    private final int position;
    private final int size;
    // default constructor taking no arguments
    private OutOfBoundsException() {
        this(0, 0, null);
    }
    // constructor taking position, size, and element-in-error as
    // arguments
    public OutOfBoundsException(int position, int size, RefObject elementInError) {
        super(elementInError);
        this.position = position;
        this.size = size;
    }
    // constructor taking position, size, element-in-error and detailed
    // message
    public OutOfBoundsException(int position, int size, RefObject elementInError, String msg) {
        super(elementInError, msg);
        this.position = position;
        this.size = size;
    }
}
```

```

    }

    public int getPosition() {
        return position;
    }
    public int getSize() {
        return size;
    }
}

```

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

t `getInvalidValue`

The “getInvalidValue” operation returns the value that caused the exception.

return type: Object

parameters: None

```

package javax.jmi.reflect;
public class TypeMismatchException extends JmiException {

    private final Object invalidValue;
    private final Class expectedType;
    // constructor taking invalid-value, expected-type, and element-in-error
    public TypeMismatchException(Object invalidValue, Class expectedType,
    RefObject elementInError) {
        super(elementInError);
        this.invalidValue = invalidValue;
        this.expectedType = expectedType;
    }
}

```

```

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

    public Object getInvalidValue() {
        return invalidValue;
    }

    public Class getExpectedType() {
        return expectedType;
    }
}

```

5.4.12 WrongSizeException

WrongSizeException is thrown when a collection containing fewer elements than the lower bound or more elements than the upper bound is passed in as argument.

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 detailed message
    public WrongSizeException(RefObject elementInError, String msg) {
        super(elementInError, msg);
    }
}

```

5.5 XMI Import/Export in JMI

In JMI, the XML Metadata Interchange format (XMI) will be used to represent metadata in a (XML) stream format. The following XmiWriter and XmiReader interfaces are used to import and export XMI elements from a JMI service. An XMI element is an XML element with the “XMI” tag (containing valid XMI).

Note – The following XmiWriter and XmiReader interfaces are currently under specified. The final version of the JMI specification will include a complete description of these APIs.

5.5.1 XMIWriter

The XmiWriter writes XMI elements to a stream. One XMI element is written for each write operation performed. The stream is left open upon completion of the write.

Operations

t 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
RefPackage

t write (Collection)

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

return type: None
parameters:

```
package javax.jmi.xmi;  
  
import javax.jmi.reflect.*;  
public interface XmiWriter {  
    public void write(java.io.OutputStream stream, RefPackage extent);  
    public void write(java.io.OutputStream stream, java.util.Collection  
objects);  
}
```

5.5.2 XmiReader

The XmiReader reads an entire XMI element from a stream into a RefPackage. The read operation returns a collection of RefObjects which are the outermost objects in the XMI content. The stream is left open upon completion of the read.

Operations

t read

The “read” operation reads an entire XMI element into a RefPackage.

return type: Collection (of RefObjects)

parameters: java.io.InputStream
RefPackage

```
package javax.jmi.xmi;

import javax.jmi.reflect.*;

public interface XmiReader {
    public java.util.Collection read(java.io.InputStream stream, RefPack-
age extent);
}
```


A

The JMI APIs for the MOF

The Java APIs for the MOF generated by applying the JMI mapping to the MOF model is contained in the companion `jmi.zip` file.

B

Accessing a JMI Service using the Connector Architecture

A JMI Service can benefit from the J2EE Connector Architecture by using it to provide connectivity to a JMI service in a pluggable way. The Connector Architecture provides seamless integration of data sources, in this case a JMI service, with application servers. It requires the data source to provide a Resource Adapter that implements the specified System Contracts.

The Resource Adapter can be used in two scenarios: the managed case, and the non-managed case. In the managed case, the user connects to an application server and uses the connection to the application server to access the JMI service. In this scenario, the application server manages security, transaction boundaries, etc.. In the non-managed case, the use obtains a connection from the Resource Adapter directly and uses it to access the JMI service. In this scenario, the needs to explicitly state transaction boundaries, etc..

Although the typical connection would represent a heavyweight connection to a data source such as a database instance, a connection to a JMI service represents a connection to a package extent (i.e., RefPackage). This limitation is due to the fact that, in the current JMI specification, the root of the normative APIs is the package extent. However, it must be noted that, in a future revision of the specification the connection to the JMI service is likely to represent the complete JMI service.

As the connection to a JMI service represents a package extent, the RefPackage interface will be the root interface available through the connection.

Note – The JMI specification does not compel JMI services to implement the above J2EE Connection Architecture specific extensions.

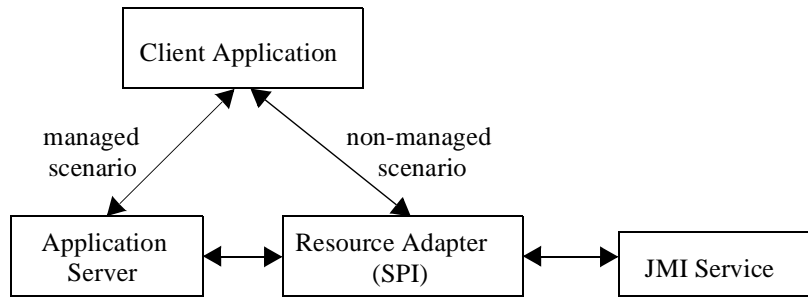


FIGURE B-1 JMI connection management.

C

Examples

Note – Examples will be included in the final version of the specification.
