Faculty of Electrical Engineering, Computer Science and Mathematics
Department Software Engineering
Warburger Straße 98
D-33098 Paderborn

Java Metadata Interface (JMI)

-Seminar Paper-

Thorsten Pivl

Paderborn, March 2005

Table of Contents

1. General Idea 3
1.1. Introduction 3
1.2. MOF 3
1.3. From MOF to JMI 5
2. Use Cases 8
2.1. Data Warehousing 8
2.2. Software Development 9
3. Usage of JMI 10
3.1. The generated API: 10
3.1.1. Class Proxy Interface and instance interface 10
3.1.2. Association Interface 11
3.1.3. Package Proxy Interface 11
3.2. Usage of the generated Interfaces 12
3.3. The JMI reflective interface 13
4. Conclusion 15
5. Bibliography 16

List of Figures

Figure 1 MOF Layers 4
Figure 2 Generated Java inheritance patterns. 5
Figure 3 Package-, class- and instance-level JMI objects and their dependencies. 6
Figure 1 Metadata Exchanges in a Data Warehouse……………………………………………..8
Figure 2 UML model of Simple XML example…………………………………………...10

1. General Idea

In this chapter, a motivation to Java Metadata Interfaces is given. After that, the idea of the Meta Object Facility (MOF) is explained, which is mandatory in order to understand the concept of JMI. In the last part of the chapter it will be discussed how MOF is mapped to JMI.

1.1. Introduction

In today's computer world it is quite difficult to exchange data. Each form of saving data has its own model, so there is simply no standard between these models. A major question is that all these data are provided in digital form, but there is still a barrier to exchange these data due to the lack of a common standard. For example, a bank which does business in form of money transfers with other banks faces one difficulty: The data it receives from the other bank may not comply with their own standard of data management. One solution is to use data extraction tools, but the rules which are formulated with these tools are very rigid and can corrupt an entire set of data.

A more intelligent solution for this problem is to use a standard for defining metadata. A common example of metadata is the database schema. The schema does not show the actual data but shows the definition of the data and how the different entities are related.

One approach to define these metadata is provided by the Object Management Group (OMG). They have defined a Meta-Object-Facility (MOF) which defines a common method for developing and accessing metamodels and an XMI-based mechanism for interchanging metamodel information among applications. A short description of MOF will be given later.

In order to use MOF in Java, Java Metadata Interface (JMI) was introduced. JMI generates for any MOF compliant metamodel a set of Application Programming Interfaces (API) which allows Java applications to specify, store, access, and interchange metadata using standard metadata services. The Java API is likely to increase the adoption of standards-based metadata and hence accelerate the creation of robust applications and solutions in which there are no barriers to information exchange.

1.2. MOF

The Meta-Object Facility (MOF) Specification defines an abstract language and a framework for specifying, constructing, and managing technology neutral metamodels. A metamodel is in this context an abstract language for some kind of metadata. Good examples for these metamodels are UML, CWM, and the MOF itself. The MOF also defines a framework for implementing repositories that hold metadata (e.g., models) described by the metamodels. This framework uses standard technology mappings to transform MOF metamodels into metadata APIs. This gives consistent and interoperable metadata repository APIs for different vendor product and different implementation technologies. For example, the MOF IDL mapping has

been applied to the MOF meta-metamodel and the UML metamodel to produce CORBA APIs for representing MOF metamodels and UML models respectively.
The MOF Specification includes the following:

- a formal definition of the MOF meta-metamodel or the abstract language for specifying MOF metamodels
- a mapping from arbitrary MOF metamodels to CORBA IDL that produces IDL interfaces for managing any kind of metadata
- a set of "reflective" CORBA IDL interfaces for managing metadata independent of the metamodel,
- a set of CORBA IDL interfaces for representing and managing MOF metamodels

The MOF model consists of four different layers. Each layer describes how the data have to be defined on the level beneath it. A common example to illustrate the idea behind MOF is a database. The example given shows the idea from the bottom layer to the top layer:

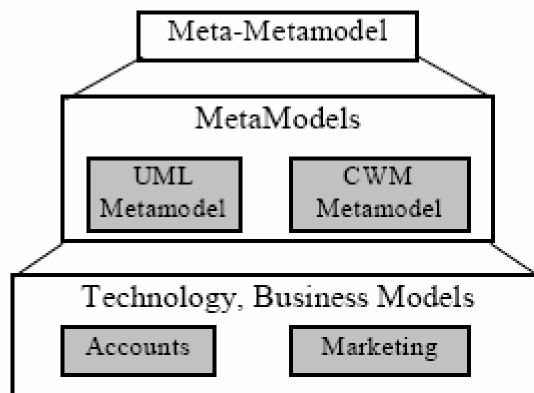

Figure 3MOF Layers

M0-Level (Information): On this layer is the actual information, e.g. in case of a database of employees the name of a person.

M1-Level (Model): This layer describes how the information must be written on level M0. The metadata specifies, for example, a table definition in a database schema. A complete database schema combines many metadata definitions to construct a database model. The M1 layer represents instances of one or more metamodels.

M2-Level (Metamodel): This level is also called the Meta-Model layer. It provides a common language how to describe a Model or, to be more precise, to describe the structure and semantics of the metadata. For the example of a database it specifies the format of a table definition.
The most common Metamodel to describe a MOF compliant model is UML.

M3-Level (Meta-Metamodel): The top level of the MOF model is called the meta-metamodel layer. It provides a common language which should describe all information models.

1.3. From MOF to JMI

As mentioned before, MOF defines a framework for implementing repositories that can hold metadata. In order to get the step from MOF to UML, a small example is given. A MOF compliant metamodel can be defined using the UML syntax. On the left side in fig 2 there are two packages defined P1 and P2, where P2 inherits from P1.

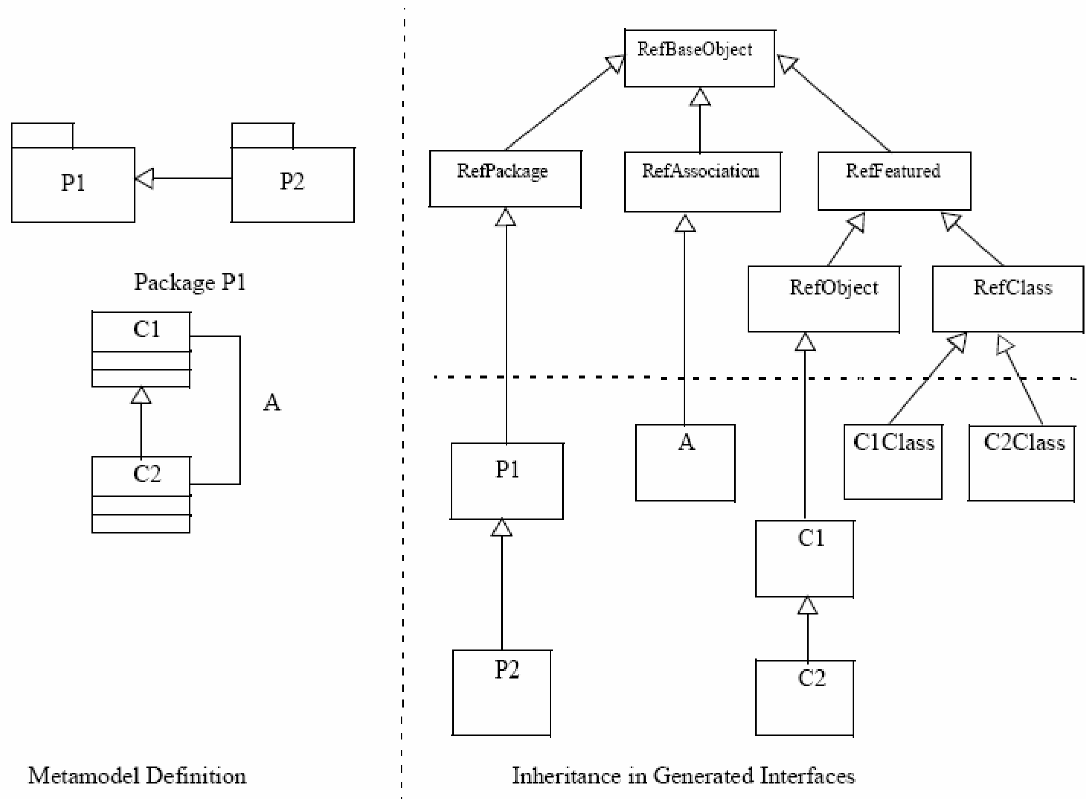

Figure 4 Generated Java inheritance patterns.

In Package P1 there are two classes; C2 inherits from C1 and there is an association between these two classes as well. The definition of this Metamodel takes place on the M2-Level of the MOF model.
Now with a specified metamodel, JMI is able to generate the MOF compliant APIs from this model. To do this, JMI provides a set of Java interfaces to map the MOF compliant model to Java interfaces. All the generated APIs are on M1-Level of the MOF model. The right side of figure 1 shows from which interfaces of the JMI package the generated APIs inherit in order to be MOF compliant. The root of the inheritance graph is a group of predefined interfaces that make up the Reflective package. These interfaces provide several operations to implement object identity and introspection.

The MOF to Java interface mapping and the Reflective package are based on an object-centric model of metadata with four kinds of M1-level metaobjects which are the following:

Package Objects and Package Creation
A package object can be associated with a “directory” of operations that give access to a collection of metaobjects described by a metamodel. The outermost package extent represents the “Root package” of the object-centric model of the metadata. This is illustrated in fig 3. The root package has to be initialized only once. Based on the provided manager, the root package creates a basic accessor which is used for all object management (get, create, modify). The root package serves as factory for retrieving model-specific packages which are retrieved by refPackage(). All objects managed by the JMI package (class-level and instance-level objects) delegate to the basic accessor of the root package. This allows a consistent unit of work policy even if multiple accessors delegate to the same manager. As shown in the figure each instance-level JMI object wraps a RefObject which is managed by the basic accessor. Transactions are controlled by the manager.

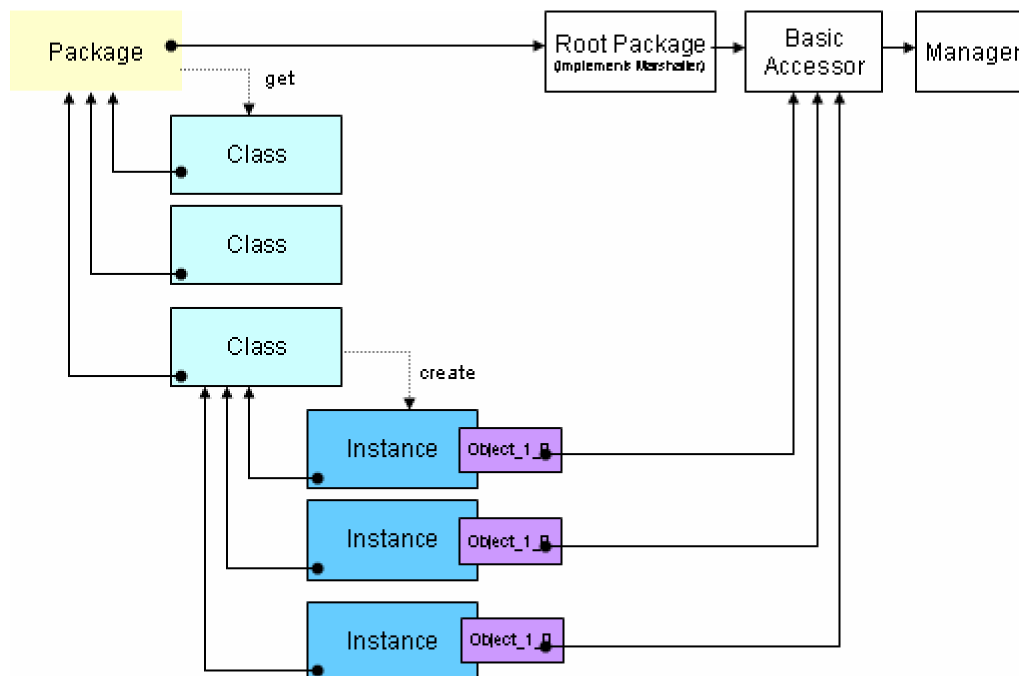

Figure 5 Package-, class- and instance-level JMI objects and their dependencies.

Class Proxy Objects
A class proxy object provides operations to create instance objects within the package. At the same time it serves as a container for the created instance object. The class proxy objects implement operations for creating instances of a corresponding class apart. Methods for accessing classifier-level attributes and using classifier-operations are implemented in this object as well.

Instance Objects
Instance objects are always tied to a class proxy object. The class proxy provides a factory operation for creating instance objects and is stored in a class proxy container as well.
The interface for an instance object provides:

- Operations to access and update the instance level and class level attributes.

- Operations corresponding to instance level and class level operations.
- Operations to access and update associations via reference.

The instance interface contains methods for accessing instance-level attributes and references and invokes instance-level operations

Association Objects
An association object manages a collection of links corresponding to an association defined in the metamodel. Similar to a class proxy object, the association object is a “static” container object that is contained by a package object. Its interface provides:

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

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

2. Use Cases

Now that the basic ideas of JMI were explained in the previous chapter, a few use cases will be presented which should illustrate how JMI can be used in practice.

2.1. Data Warehousing

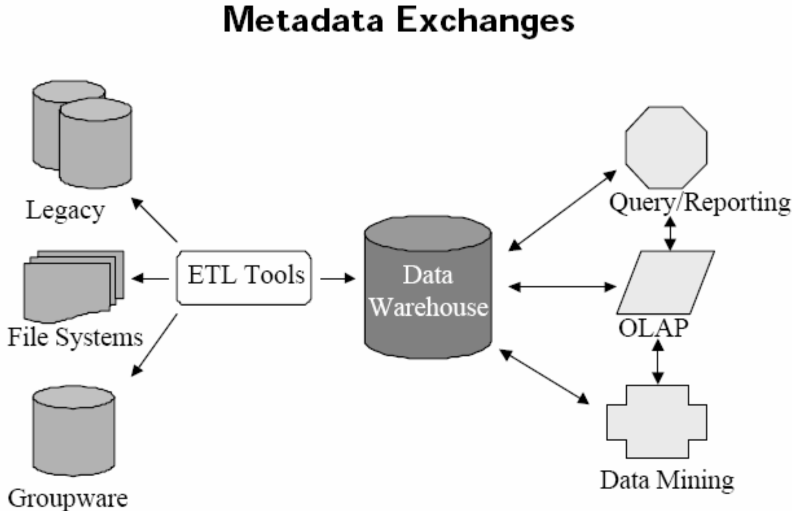

Figure 6 Metadata Exchanges in a Data Warehouse

One example where it is helpful to use JMI is Data Warehousing. The basic idea of Data Warehousing is that there are so called ETL Tools (Extraction, Transformation and Loading Tools) which extract data from operational systems. These systems can be different sources like databases, files that have captured web site click stream data, or applications such as ERP or CRM applications. These data are now transformed into the data warehouse. The problem with these tools is that they all use different metadata and therefore an interface has to be implemented for each tool which is very tedious and can cost a business a lot of resources and money. The same problem occurs with the tools that are used to analyze the data that has been distilled into the data warehouse to generate the required reports. These tools also use different metadata standards, hence hard wired interfaces have to be implemented for each tool.
To solve this problem, a common metadata infrastructure must be created which provides a common programming model, and a common interchange format.

This lack of metadata interoperability also impedes data analysis activities. For example, a data analyst creates 25 different segments of customers represented in the company's data warehouse. A sales analyst might then calculate sales forecasts for these customer segments across various regions and product segments using an OLAP tool (Online Analytical Processing Tool). He might also like to visualize the customer segments and the sales forecasts using a three-dimensional visualization tool.
But since the data mining, OLAP, and data visualization tools do not share metadata constructs, analysts cannot move data seamlessly among the tools. One option might be to purchase the tools from a single vendor. This, however, does not guarantee interoperability, especially if the vendor has acquired data analysis tools from

different companies. It also restricts users from potentially selecting the different products which might be better than the one being used. The other option is for the analysts or in-house programmers to write code to integrate the tools and alter the database schema to reflect the 25 customer segments.
To solve this problem, it is sensible to standardize the exchange format of these tools. This can be achieved by creating a MOF compliant metamodel and implementing it in Java using the JMI technology.

2.2. Software Development

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 modelling tools, such as UML tools, to “model” the application, one or more Integrated Development Environments (IDEs) to develop the Java source, and one or more EJB deployment tools to mange the deployment of the application.
For this scenario, an EJB development solution can be built around JMI using three metamodels that represent the domains of the different tasks, i.e., the UML metamodel, the Java metamodel, and the EJB metamodel. Each tool would then participate in this integrated solution through an adapter that maps the tool specific APIs to the JMI APIs for the respective metamodel. Services that span multiple tasks, such as keeping the model and source code in sync, are then developed using the JMI APIs. The primary advantages of this solution over a hand crafted solution are:

- Services that span multiple domains, or even extensions to all tools of a single domain, can be developed using the common programming model provided by JMI. Note that such services and extensions are tool independent.
- The complexity of integration has been reduced from N x N where N is the number of tools being integrated, to M x M where M is the number of domains spanning the problem space (in this case, modelling, 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.

3. Usage of JMI

A simple example on how to use the JMI will be given in this section. The idea of this example is to create a metamodel of a XML scheme. From this example the JMI APIs will be created. These APIs can now be used to create a small application.

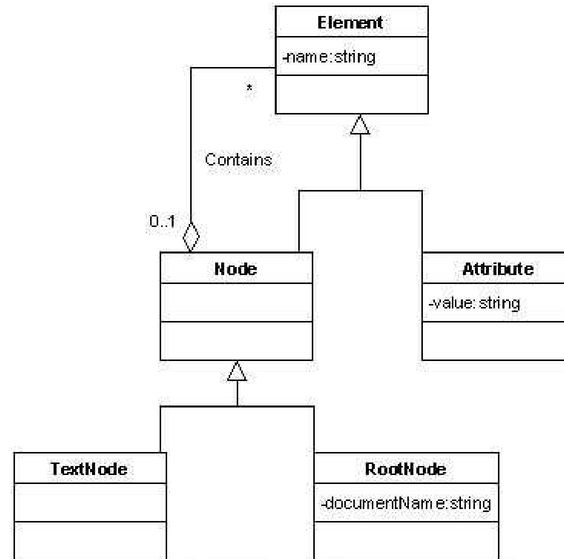

Figure 7 UML model of Simple XML example

The whole metamodel is defined in a MOF package named XMLModel. Each XML document consists of nodes (in the model represented by Node class) and each node can contain either attributes represented by Attribute class or nested nodes. Both attribute and node have name. In addition attribute can have value. Contains association expressing the fact that a node can contain both attributes and nodes are ordered on the elements end – this is because order of nodes is significant in XML. Text nodes are represented by a separate subclass of Node named TextNode. As each XML document consists of exactly one root node, it is convenient to define a separate subclass of Node class called RootNode for representing a root node in a document. This class also represents the document itself – it contains an additional attribute documentName.

3.1. The generated API:

3.1.1. Class Proxy Interface and instance interface

As mentioned before, JMI is using four kinds of interfaces. Furthermore, two kinds of interfaces are generated for each class in the metamodel:

1. One is the static context of the class (class proxy interface),
2. The other one represents the instances of this class (instance interface).

The class proxy interface implements operations for creating instances of a corresponding class apart from abstract classes. Methods for accessing and using attributes on the class level are implemented in this interface as well. For the class Attribute, the class proxy interface looks as follows:

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

Due to the simplicity of this example, there are no operations needed in this interface to access and use the attributes, but as mentioned before, methods for creating an instance of this class are needed.

The class proxy interface for the class Element is also pretty straight forward:

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

Again there are no classifier-operations and even no operation to create an instance for this class is defined as abstract.

The instance interface contains methods for accessing instance-level attributes and references and invokes instance-level operations:

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

The generated interface has a get and set method for the attribute Name, for it is the instance-level attribute. The method of a container simply gets or sets the parent node in the XML scheme.

3.1.2. Association Interface

The third interface which is generated by JMI is the Association interface. It used for all associations in a metamodel – in the example there is only the *Contain* association which is generated as follows:

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

The methods implemented can add, remove or check for the existence of an association.

3.1.3. Package Proxy Interface

Finally, the fourth generated interface is the package proxy interface. This interface is generated always for each package in a metamodel. In this example the XMLModel package. The methods in this interface give access to all proxy objects for all directly nested or clustered packages and contained classes and associations:

```
public interface XMLModelPackage extends javax.jmi.reflect.RefPackage
{
public NodeClass getNode();
```

```
public AttributeClass getAttribute();
public ElementClass getElement();
public RootNodeClass getRootNode();
public Contains getContains();
}
```

3.2. Usage of the generated Interfaces

With the generated interfaces, and the MOF compliant metamodel, the question remains how to use these interfaces. Since we are using a XML example, the following scenario should serve as an example: A tool is to be created which gets an XML document as input. The task is now to create metadata for it in the JMI repository generated from the metamodel of XML described above. To do this, the document has to be parsed from the application and this output will be used to populate the repository using the generated JMI interfaces.

To make things easier, we assume that the tool already knows how to get access to the package proxy interface:

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

With the picture of the metamodel in mind, we first have to create root node which can be done with the following instruction:

```
RootNode root = service.getRootNode().createRootNode(<name>, <documentName>);
```

This piece of code first obtains a reference to the class proxy object of RootNode and creates a new instance of RootNode. With a Root node instantiated it is now possible and necessary to create other nodes. The principle is that node is created and then added to a container. This container functions as a “parent node” similar to a binary tree:

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

The principle of creating an attribute works in the same way like creating a node:

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

Now a concrete code of XML is given and shown how metadata is created using the generated JMI APIs:

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

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

```
// create the root node
RootNode root = service.getRootNode().createRootNode("test", "test.xml");
// create the first node
Node node = service.getNode().createNode("node");
// set root as its container
node.setContainer(root);
// now create the two attributes
Attribute attr1 = service.getAttribute().createAttribute("attr1",
"value1");
Attribute attr2 = service.getAttribute().createAttribute("attr2",
"value2");
// add the two attributes as contained elements of node
// (this is an alternative way of setting the container)
```

```
node.getElements().add(attr1);
node.getElements().add(attr2);
// create the text node
TextNode text = service.getTextNode().createTextNode("text");
// set node as its container
text.setContainer(node);
```

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

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

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

3.3. The JMI reflective interface

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

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

public void readAttributes(RefObject object) {
// metaobject of any RefObject is an instance of MofClass
MofClass metaObject = (MofClass) object.refMetaObject();
// explore the object's metamodel to get object's attributes.
for (Iterator it=metaObject.getContents().iterator(); it.hasNext();) {
       ModelElement element = (ModelElement) it.next();
       if (element instanceof javax.jmi.model.Attribute) {
              // print the attribute's name
              System.out.print(element.getName() + " = ");
       // get the value of the attribute
       Object value = object.refGetValue(element);
// print the attribute value
System.out.println(value.toString());
}
}
}
```

4. Conclusion

In this paper the technology of JMI was introduced. It is based on the Meta Object Facility (MOF) specification which can be considered as the standard model for metadata. The relationship between JMI and MOF was shown in chapter one. JMI is basically the implementation of the MOF specification in the programming Java. To achieve this, a MOF compliant metamodel is created using the UML syntax. Based on this UML model, a set of interfaces are generated which are based on an object-centric model of metadata with four kinds of M1-level metaobjects:

- Package Objects
- Class Proxy Objects
- Instance Objects
- Association Objects

The goal of JMI is to have a standardized access to metadata and to enable the interoperability of different applications to simplify the exchange of Information. This reduces the development costs of applications and provides a faster development for the metamodels can be reused. JMI is capable to generate the Java classes directly from the MOF compliant metamodel which makes the development of applications pretty easy. Two use cases were explained in Chapter two to understand the benefits of JMI in practice.

Finally, in chapter three a simple example of JMI was provided. It showed how the generated interfaces look like and how they could be used in an application.

5. Bibliography

1. Ravi Dirckze: *Java Metadata Interface (JMI) Specification Version 1.0, 2002* (http://www.jcp.org/jsr/detail/40.jsp)

2. *Meta Object Facility (MOF) – Specification* Version 1.4, April 2002
   http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pdf

3. Model-Driven Architecture: Vision, Standards And Emerging Technologies, John D. Poole, Hyperion Solutions Corporation, April 2001

4. Chuck Mosher: *A New Specification for Managing Metadata*
   http://java.sun.com/developer/technicalArticles/J2EE/JMI/

5. Benoy Jose: *Java Metadata Interface*
   http://javaboutique.internet.com/articles/JMI/index.html

6. Java Dokumentation der JMI-Pakete
   http://java.sun.com/products/jmi/jmi-1_0-fr-doc/

7. XMI-Dokumentation der OMG
   http://www.omg.org/technology/documents/formal/xmi.htm

8. XML-S Spezifikation
   http://www.w3.org//XML/Schema

9. Carsten Amelunxen, Andy Schürr: *Codegenerierung für Assoziationen in MOF 2.0*
   http://www.es.tu-darmstadt.de/english/research/publications/download/modellierung2004.pdf