

A New Specification for Managing Metadata

Chuck Mosher

Sun Microsystems

Introduction

Today's Internet-driven economy has accelerated users' expectations for unfettered access to information resources and transparent data exchange among applications. This exchange requires a deep knowledge of how each application, tool, or service structures and interprets the information it uses. The term *metadata* is used to describe both how this information is structured (its *syntax*) and what it means (its *semantics*).

Unfortunately, most applications define metadata differently. Each uses slightly different programming structures, syntax, and semantics to model their metadata. These incompatibilities make it challenging for one application to discover and interact with the data maintained by another application. To overcome this hurdle, companies typically spend lots of developer time and money building hard-wired interfaces between applications so they can share data in sophisticated ways. This costly and ongoing effort is the result of a lack of common metadata - or more specifically, a common model for creating metadata (a *metamodel*). This stifles the development of robust solutions to common business problems that require combining or using data from multiple, heterogeneous applications. Examples of such business problems include enterprise application integration, data warehousing, business intelligence, business-to-business exchanges, information portals, and software development. Metadata management is a gating function in the creation of integrated, interoperable applications. Standardizing on XML DTDs or XML Schemas, which many industries are attempting to do as a solution to this problem, is insufficient, as they do not have the capability to model complex, semantically rich, hierarchical metadata. This problem becomes more critical as the number and complexity of applications, services, data sources, interchange protocols, and deployment platforms continue to increase in the e-economy.

To facilitate metadata interoperability, a host of vendors have teamed up through the Java Community ProcessSM (JCP) to provide a platform-independent specification for metadata. It is the JavaTM Metadata Interface (JMI) specification, which is JSR-40 on the JCP website at jcp.org. The JMI specification defines a dynamic, platform-neutral infrastructure that enables the creation, storage, access, discovery, and exchange of metadata using Java interfaces. JMI is based on the Meta Object Facility (MOF) specification from the Object Management Group (OMG), an industry-endorsed standard for metadata management.

This paper describes what metadata is, why its management is so important, what is required of a metadata management infrastructure, and how JMI satisfies these requirements.

What is metadata?

Metadata can be defined as information about data, or simply data about data. In practice, metadata is what tools, databases, applications and other information services use to define the structure and meaning

of their objects, services, and other computing artifacts. An obvious example of metadata is a database schema, which describes not only how data entries are laid out in a relational database, but what they mean. A less obvious example is the meaning of object attributes and methods in a Java class file. Another example would be an Enterprise JavaBean™ (EJB) deployment descriptor, which describes the metadata an application server needs in order to deploy and use the EJB. These examples refer to what is called technical metadata. There is also business metadata (also called process metadata), which is used to capture semantic content about such things as business rules, business nomenclature and business terminology. A glossary is an example of business metadata. Then there are systems that provide a means of notating metadata. XML DTDs are an example of a way to notate and interchange semantic information about structured documents. The Unified Modeling Language (UML) defines a way to formally describe the function, structure, and behavior of a system. We will show later how UML can be used as part of a metadata management infrastructure.

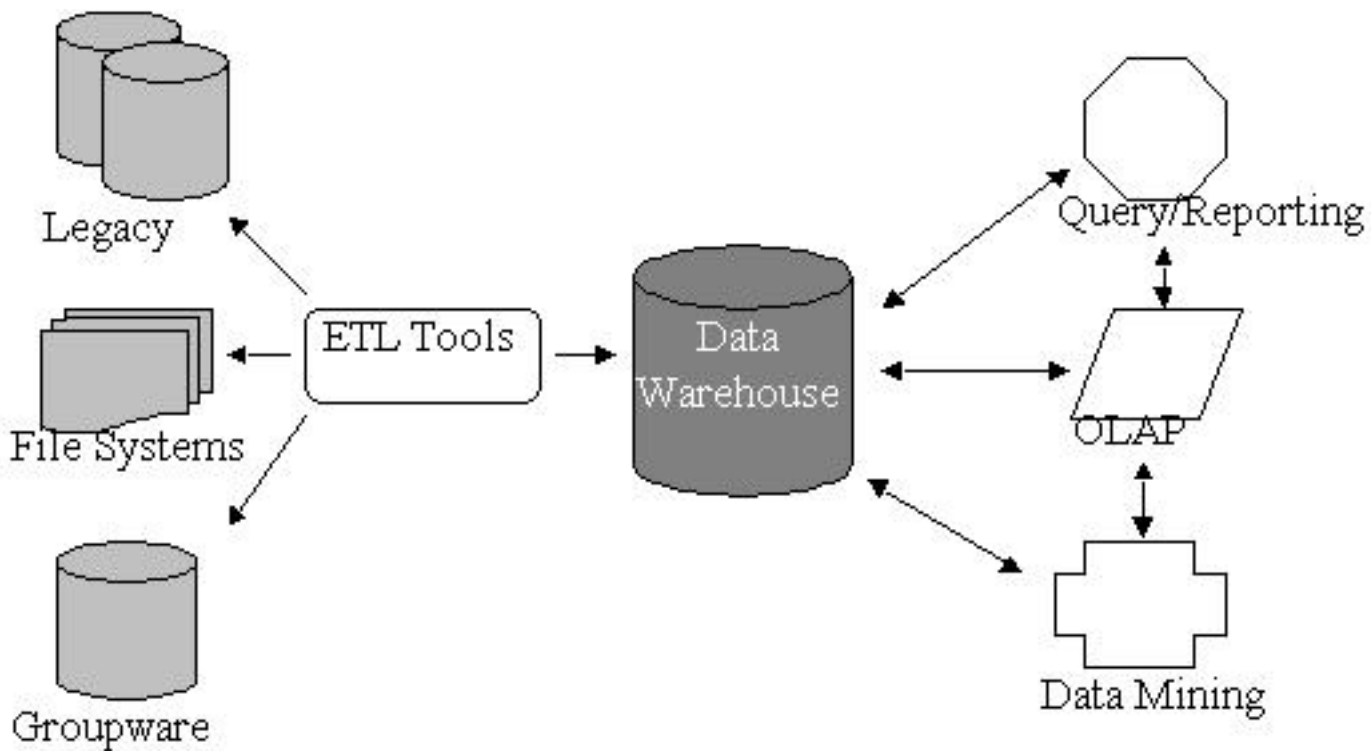
Why is metadata important?

Without metadata, or semantics, one has no way of knowing what a data object represents. The integer value "39" encountered in a program, for example, could mean almost anything. Currently, metadata management is made extremely difficult by the fact that many, perhaps most, of these semantics tend to be embedded in the systems that use them. The lack of a common way to represent and share metadata thus leads to great difficulties in sharing even the simplest data, let alone in the integration of complex applications, components, and systems.

People are excited about XML in part because at one level (using data encapsulated within descriptor tags) it appears to solve this problem. However, XML has significant limitations as a general solution to metadata representation and management. The current flurry of activity in various industries to standardize on a set of XML DTDs to represent the semantics of their particular artifacts, while a valuable and worthwhile first step, is really focused at the wrong level of abstraction for enabling true metadata interchange.

Where is metadata management required? Well, basically wherever there are significant metadata artifacts to be found. And this turns out to be just about everywhere. Application development tools and frameworks need to manage and understand models, record definitions, and database definitions. Component-based development environments must deal with a host of interfaces, classes, and components, possibly in different languages. Data warehouses and information portals have to manage data that is organized as tables, columns, schemas, cubes, or flat files in different formats. These are just a few examples.

Finally, comprehensive metadata management is a necessary prerequisite for enabling the kind of services-based architecture that we are all moving towards. You cannot share or use a service unless you understand the semantics of the objects and features that it provides. For example, the higher-level artifacts of ebXML (a standard for enabling e-commerce) such as business objects, business processes, company profiles, and trading partner agreements all need to be described in a platform and implementation-independent way. The reason we point out ebXML is because the people working on it are using the metadata technologies we will be describing to characterize their artifacts.



Data Warehousing Example

An Example: Challenges of Metadata Management in Data Warehousing

Data warehousing applications, because of the variety of interoperational challenges they face, well illustrate some of the problems encountered by the lack of a common metadata standard. Data warehousing applications typically deal with many different data sources. Each data source will of course have its own unique metadata (its schema). Not only do data warehousing applications have to integrate these different databases, but the databases themselves usually capture different aspects of the business (i.e., not only are the schemas structurally different, they often refer to different things). Further, these databases are often located on different systems that span the company. Additionally, other data sources can contribute, for example, files that have been generated that capture web site click-stream data. Also, applications themselves can be sources of data, such as the output of ERP or CRM systems. All of these different kinds of data need to be *Extracted, Transformed, and Loaded* into a data warehouse. This is what *ETL* stands for in the middle box in the figure above. This process is so complex that there are over 250 ETL vendors in the industry today that make their living by writing separate interfaces to each different kind of data source on the input end and each data warehouse on the output end. And any one company that is putting a data warehouse solution in place cannot typically use only one ETL vendor, but needs many to handle their own unique set of data source requirements. Then we even have a problem on the output end, where one would like to take the data that has been distilled into a data warehouse and analyze it with different reporting tools. Of course, these tools themselves also expect their data to be in a certain (different and unique) format.

What is required to manage metadata?

We have seen that there are tremendous benefits to be had in managing metadata. If one has a standard

way to represent and share metadata descriptions, then application-level integration is greatly facilitated. Application-level integration is largely unaddressed by the Java 2 Platform, Enterprise Edition (J2EE) framework, which addresses (and solves) mostly platform-level integration issues. So what is required to describe and manage metadata?

The first thing we need is a common, standard way to represent metadata. Another way of saying this is that we need a way to *model* metadata. A well-defined model will have very precise definitions of what the features and attributes of particular model instances (the metamodels) mean. These precise definitions will then allow us to define exact and unambiguous mappings of the model features to particular languages and interchange formats. Also, this model must be able to describe metadata at different levels of abstraction, for one can always move up and down in meta-levels. One system's data is another system's metadata is another system's meta-metadata. When you think about it, what we really need is a meta-metamodel. In other words, a model for defining metamodels that describe metadata (whew!). In effect, a language for defining metadata at any level of abstraction.

This is what the Meta Object Facility (MOF) specification from the Object Management Group (OMG) provides. It defines a small set of concepts (such as classes, methods, attributes, operations, references, data types, associations, and constraints) that allow one to define and manipulate models of metadata. These concepts are described using UML notation, but can be used to, for example, model the UML itself. Since the MOF uses UML notation, it is easy for people familiar with the UML to begin to use it to define and characterize their metadata. The MOF standard was approved by the OMG in November 1997. However, the MOF is only half the story, since we also need to define a mapping from this admittedly very high-level and abstract definition to actual language semantics. This is where JMI comes in. It defines how the constructs used by the MOF to model metadata map to the Java language.

Thus, one can define metadata clearly and unambiguously with UML-like semantics, and JMI will automatically generate the Java APIs that will create, manage, access, and query metadata instances. Additionally, JMI uses the XML Metadata Interchange (XMI) standard to define precise mappings from MOF-based metamodels to XML DTDs, and mappings from metadata instances to XML streams, enabling standard XML-based interchange.

What about XML?

We mentioned previously that XML alone is not sufficient to model metadata. Why is that the case? Isn't XML a language for defining other languages, and as such could be used to describe metadata? While it is true that XML is great for data and metadata interchange, it turns out that it is not rich enough for capturing high level semantic information. Features like hierarchical relationships, associations, data types, and other critical information necessary for describing metadata can only be represented in XML by extending XML itself. Some extensions like these are found in XML Schema, but that still does not provide a common metadata modeling environment. For example, although any one XML Schema can represent the metadata for a particular application, component, or service, it does not give a general solution to modeling metadata across many different domains. What we need, again, is a set of high-level constructs that can be used to build models of any kind of metadata. The MOF model provides these constructs.

So now we can see that the XML DTD standardization efforts referred to earlier are often focused at the wrong level of abstraction. These groups are essentially standardizing on an interchange format. While

this is very useful, their efforts would be even more effective if they could standardize on the actual objects used in their domain, the way the data warehousing industry has done with the Common Warehouse Metamodel (CWM). In this case, industry leaders have agreed on the high-level definitions of such industry-specific concepts as dimension, schema, column, relational table, and OLAP cube. By standardizing the metadata, data warehousing vendors are significantly increasing the ability of their systems to interoperate.

JMI - Enabling a Metadata Management Infrastructure

Implementations of the JMI specification will provide a metadata management infrastructure that will greatly facilitate the integration of applications, tools, and services. Formerly, complete interoperability and integration of disparate systems has been difficult because there has been no standard way to represent their unique characteristics. JMI provides the metadata framework that captures these semantics. Enterprise JavaBeans™ (EJBs) have proven to be highly effective at masking the complexities of the computing platform and enabling developers to build components without needing to directly handle transactions, security, resource pooling, and a host of other low-level programming tasks. Similarly, JMI will allow developers to mask complexities in their business logic by creating high-level models of their specific technology and business domains. JMI thus reduces the complexity of interoperability and integration by providing: 1) a common semantic model (with which to create the models), 2) a common programming model (APIs for metadata access which are automatically generated, accompanied by a rich set of lifecycle management interfaces), and 3) a common interchange format (through XML).

A host of companies are participating in the definition of the JMI specification. They are: Adaptive, DSTC, Hyperion Solutions Corporation, IBM Corporation, Iona, Novosoft, Oracle Corporation, Perfekt-UML, Rational Software Corporation, SAS Institute, Sun Microsystems, Sybase, and Unisys Corporation, which is the JCP Specification Lead. Many of these companies are producing systems that will implement or use the JMI standard. Compliance to the specification will be ensured by the JMI Technology Compatibility Kit (TCK), which will be available, along with a Reference Implementation of the specification, when the standard becomes final.

The Value of JMI

Let us examine all of the features that a JMI implementation provides:

First, the entire process is model driven; all the Java interfaces for metadata access are automatically generated from the information in the metamodel (metadata model). If the metamodel changes, the interfaces will be changed (generated) automatically to reflect it. This greatly facilitates maintaining alignment between the models of systems and their implementations. It is important to note that metamodels and the facilities to create them are independent of any implementation of a JMI service. The tools used to build such models are commonly used in the industry to design and build complex systems. JMI brings the power of this modelling capability directly into the Java platform.

Second, JMI provides reflection. In addition to the model-specific interfaces mentioned above, each interface to a metamodel artifact implements a JMI reflective interface specific to that artifact. These

interfaces provide all the capabilities provided by the generated interfaces, but in a highly generalized way. The JMI reflective interfaces augment the Java reflective interfaces as they provide JMI functionality that allows for applications to query a model at run time to determine the structure and semantics of the modeled system. This enables tools to implement generic browsing and discovery mechanisms.

Third, JMI provides lifecycle management. For example, for each class in a model named "Foo", there are two interfaces that get generated - the Foo.java interface and the FooClass.java interface. The FooClass.java interface is a factory object for instances of Foo. It is used to create and manage instances of Foo. If the class is modeled as an abstract class then the factory object does not get generated. The Foo interface is implemented by all instances of the Foo class. These generated interfaces contain all the navigational functionality required to traverse the links (instances of associations) in the model. JMI also has something called a JMI reference (not to be confused with the Java reference). A JMI reference provides accessors for manipulating and navigating links. It should be noted that lifecycle management is an optional component for JMI systems, as there exist systems that do not have repositories or persistence mechanisms behind them but still wish to provide access to their metadata via the JMI model-specific and/or reflective interfaces.

Fourth, JMI provides XML import/export capabilities. The JMI XML mapping specifies how models can be represented as XML DTDs, which can then be used to validate XML documents that represent instance data. Using the JMI XML utility APIs, all or part of the data within a particular JMI model package can be imported or exported.

Model-Driven Development

Increasingly, software developers are looking to find ways to capture and leverage best practices when developing their systems. They are doing this with *patterns*, high-level descriptions of software solutions to recurring problems. Some well known examples are the Model-View-Controller pattern, which decouples the presentation, control, and data management aspects of an application; or a Data Access Object, which encapsulates loosely-coupled access to back-end data resources. These patterns allow for reusability in a much broader context than the particular domain in which the pattern was first identified.

As artifacts that are captured at multiple levels of abstraction, patterns are most often characterized using UML notation. These patterns are just another kind of metamodel. Coupled with the facilities that JMI provides of automatically generating the Java interfaces and XML DTDs to the models, patterns have the potential to drastically speed up the design and development of well-built, successful systems. The OMG has based its future architecture and development on just such a model-based approach, known as the Model Driven Architecture (MDA), in which MOF and JMI are core components.

Other Specifications Leveraging JMI

There are a number of specifications that are using the metadata management facilities of JMI:

- The UML/EJB Mapping Specification (JSR-26) defines a set of profiles in UML that enable the precise mapping of EJBs to UML and vice versa. The specification also defines a mechanism for using these UML models stored in an EJB-JAR to describe the contents of the jar. This will let enterprise tool and framework vendors use the UML models stored in the jar files for automation and reflection. JMI will be providing the APIs for accessing those models.

- The Java API for OLAP Specification (JSR-69) defines a standard Java API for data warehousing and online analytical processing, or OLAP. It is based on the Common Warehouse Metamodel (CWM), and will also be relying on JMI to generate the interfaces to the data warehouse objects..
- The Java API for Data Mining (JSR-73) defines a standard Java API for data mining. The expert group is establishing the metamodel descriptions of the objects and artifacts necessary to implement an interface to a data mining engine, and then letting JMI generate the APIs.

We continue to have discussions with other JSR Specification Leads to explore other areas where this technology can be leveraged.

Roadmap

The JMI specification has been released for public review, and can be found at <http://jcp.org/jsr/detail/40.jsp>. Unisys, Sun, IBM, Adaptive, and other companies are in the process of implementing the specification. Others, such as Hyperion and Oracle, will be implementing JMI interfaces and functionality as part of embedded metadata management services within their Java for OLAP (JSR-69) and Java for Data Mining (JSR-73) product offerings.

The final release of the JMI specification, along with its Reference Implementation and compatibility test suite is expected by July 2002. An early implementation of JMI that works with the NetBeans open-source IDE is the Metadata Repository (MDR). See the References section below for the link.

Summary

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 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.

We have seen how a standard for metadata management is becoming increasingly important for enabling

the integration and interoperability of applications, tools, and services. We have seen that a common way to represent metadata is required, and that this representation must be both capable of modeling multiple levels of abstraction, as well as use well-defined artifacts that are precisely defined. With these precisely defined artifacts, unambiguous mappings to specific languages such as Java, or interchange mechanisms such as XML can be defined.

The JMI specification provides this standard. JMI will allow Java applications to specify, store, access, and interchange metadata using standard metadata services. By describing metadata using the widely-adopted UML notation, developers can take advantage of the capabilities of the JMI framework to automatically generate the APIs necessary to manage and interchange their metadata.

The specification 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. The ability to reuse modules and systems is also increased, because the components can be well characterized. Finally, comprehensive metadata management is an enabler for the services-based architecture. A standard for describing objects and services is essential in building services that can query, negotiate, and interact with each other.

References

JMI: java.sun.com/products/jmi

Meta Object Facility (MOF): <http://www.omg.org/cgi-bin/doc?formal/00-04-03>

Common Warehouse Metamodel (CWM): <http://www.omg.org/technology/cwm/> and <http://cwmforum.org>

OMG's Model-Driven Architecture: <http://www.omg.org/mda/>

NetBeans JMI Implementation: <http://mdr.netbeans.org/>

Java for OLAP: <http://jcp.org/jsr/detail/69.jsp>

Java for Data Mining: <http://jcp.org/jsr/detail/73.jsp>

Author: chuck.mosher@sun.com