NetBeans Metadata Repository

Martin Matula
(martin.matula@sun.com)
3/3/2003

This document gives an overview of the NetBeans Metadata Repository (MDR) and provides basic information regarding related standards. It describes the basics of how the MDR works, the motivation for creating it, and some of its uses.

Introduction (MDR and Related Standards)

To put it briefly, the MDR is an extended implementation of the Meta-Object Facility, XML Metadata Interchange, and Java Metadata Interface standards. To understand what that means, let's look closer at what these standards define.

Meta-Object Facility

Meta-Object Facility (known as MOF) is a standard from the **Object Management Group (OMG)** that specifies an abstract language for describing other languages. In this context a language means an abstract syntax of a language (i.e. MOF is not used do describe language grammar - it is used to describe structure of objects that can be represented in a given language). MOF is also often referred to as a meta-metamodel and the abstract syntaxes it is used to describe are called metamodels. This is because MOF corresponds to the M3 level of the four-layered metamodel architecture:

Meta-level	Description	Examples
M0	Data / instances	Records in a DB table, instances of Java classes ("abc" - instance of java.lang.String)
M1	Metadata / models	Tables, columns in a database (records in a system catalog), concrete classes, methods, fields in Java program (java.lang.String – instance of Java Class language construct)
M2	Meta-metadata / metamodels / languages	Description of database (definition of things like Table, Schema, Column, etc.), description of language constructs (definition of Class, its attributes, contained elements – methods, fields, etc.)
M3	Meta-metamodel	MOF

In the table above, each M(x) level (where x<3) is described using constructs defined in M(x+1) level. M3 is the last level as MOF is described using MOF, thus there is no need for M(x>3).

To illustrate how MOF is used to define metamodels, let's try to use it to define XML.

An XML document can be described as a set of “element nodes”, “attribute nodes”, and “text nodes” as in the following example XML document:

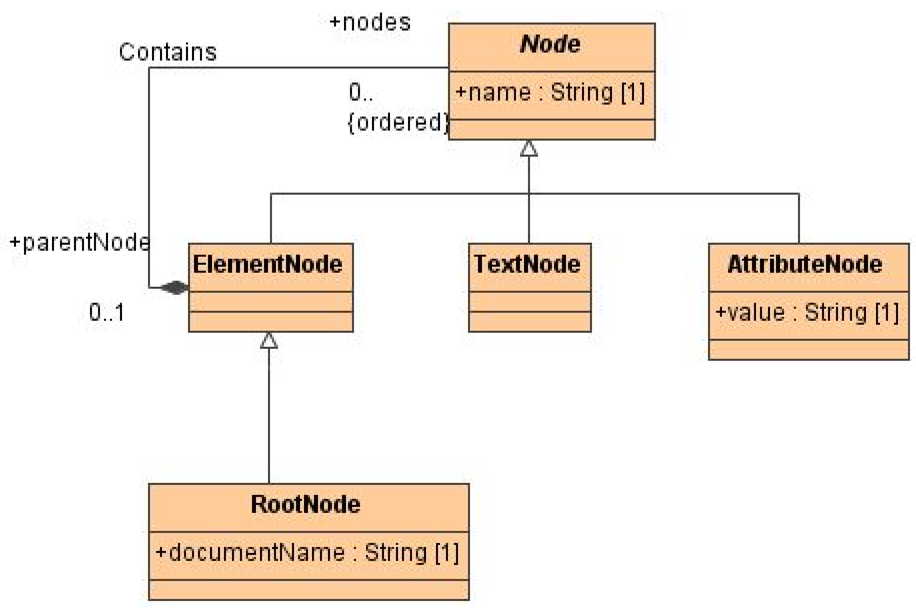
```
<acronyms>
    <item acronym=”MOF” meaning=”Meta-Object Facility”/>
    <item acronym=”XMI”>
        <meaning>XML Metadata Interchange</meaning>
    </item>
</acronyms>
```

The document contains the following element nodes (in order of appearance): acronyms, item, item, meaning. The attribute nodes in the document are: acronym, meaning, acronym. There is only one text node in the document: XML Metadata Interchange.

The additional statements that can be made about the XML document structure are as follows:

- Each node has a name (for text nodes the name is the actual text of the node)
- Element nodes can contain other element nodes, attribute nodes or text nodes (in the document above, the second “item” element node contains an “acronym” attribute node and a “meaning” element node; the “meaning” element node contains an “XML Metadata Interchange” text node)
- Attribute nodes have a value (e.g. the value of the first “acronym” attribute node in the document above is “MOF”)
- For each XML document there is exactly one root element node (in the document above it is the “acronyms” element node)

The following picture shows, how the above information can be expressed in MOF:



The boxes in the picture represent MOF classes – there is an abstract class called “Node” with a “name” attribute - this is a root class for all kinds of nodes and captures the fact that all nodes have a name. The “Node” class is subtyped by 3 non-abstract classes representing 3 different kinds of nodes – “ElementNode”, “TextNode” and “AttributeNode”. Between the “ElementNode” class and the “Node” class, there is an MOF association named “Contains” which captures the fact that every ElementNode may contain zero to many other nodes. To express the fact that every document contains exactly one root element node, there is a separate subclass of ElementNode representing root nodes (the name of their home document is stored in an attribute called “documentName”). The whole metamodel captures all the information we are interested in with respect to the structure of XML documents, thus we can call it a metamodel of XML. As you can see, MOF is not used to describe the concrete representation (grammar) for XML (i.e. it does not define that each element node starts with a node's name enclosed in <> brackets, etc.) - it only describes the abstract syntax (i.e. structure) of the language.

Note: *The MOF specification itself does not define any textual or graphical representation (concrete syntax) for MOF. What you see above is in fact a model expressed using UML notation (UML = Unified Modeling Language – a modeling language standardized by OMG), not MOF. UML notation can be used to represent MOF metamodels thanks to the* ***UML Profile for MOF****, which is a standard that defines the mapping between MOF and UML.*

Within OMG, MOF is used to formally define abstract syntaxes for several other standards. These include **UML (Unified Modeling Language)**, **CWM (Common Warehouse Metamodel)** and MOF itself (MOF is self-described). The advantage of this approach is not only the formalism itself (and thus unambiguity of the specification) but also the possibility of defining other generic standards on top of MOF, which can then be automatically applied to any standard formally expressed using MOF.

XML Metadata Interchange

One excellent example of such a standard is **XML Metadata Interchange (XMI)**. XMI is a mapping of MOF to XML – i.e. a standard used to automatically derive the XML interchange format for instances of any language described in MOF. The XMI standard defines production rules for generating DTDs and XML Schemas from MOF metamodels and XML serialization rules for instances of those metamodels. The instances serialized using these rules then conform to the XML Schemas/DTDs generated from the metamodels.

So, when UML was standardized, there was no need to create a separate standard for UML interchange format. The standard interchange format was inferred by applying XMI on the MOF description (metamodel) of UML. The resulting DTD, and the rules for serializing UML models into XML (so that it conforms to the DTD) enable interoperability between UML tools - any UML tools that implement the XML serialization rules correctly (according to XMI specification) are able to exchange UML models.

Because MOF is in fact also a metamodel (it is described using MOF itself), XMI can be applied on it, too. As a result, we have a standard interchange format not only for metamodel instances, but for the metamodels themselves (as they are also instances of a metamodel, which is MOF itself). So, when applied to MOF itself, the XMI standard defines the concrete XML-based textual syntax for MOF.

Java Metadata Interface

Similarly to XMI, one can define standards that enable interoperability on the API level (specific to a given platform) rather than on the XML level (documents interchange). Let's take the example of a UML modeling tool implemented in Java. If we want to make the tool extensible – enable other vendors to build plugins for it – we need to add a set of APIs into the tool and expose it to the 3rd party plugins so that they can use it to access the UML metadata (information about the UML model that a user is currently working on) in our UML tool. We already know what the abstract syntax of UML is (as it is expressed in MOF and used for deriving the interchange format of UML models). So why don't we use it to generate the set of APIs that can be than used to access the information about UML models programmatically? Standard rules for generating such API would make it possible for a plugin written for a UML tool "A" to work also with a UML tool "B" (if both tools expose the API conforming to the hypothetical standard). This standard is not hypothetical! It already exists for Java. (It also exists for IDL, but that's beyond the scope of this document.)

The standard for mapping MOF to Java (i.e. generating the Java API from MOF metamodels) was developed in JCP (Java Community Process) as JSR-40 and is called the **Java Metadata Interface (JMI)**. In addition to rules for generating metamodel-specific Java API, it also defines a set of "reflective" interfaces that can be used similarly to the metamodel-specific API without prior knowledge of the metamodel. This is useful for building generic tools such as generic metadata browsers.

What is MDR?

Now we are finally getting to what the MDR really is. As its name suggests, MDR is a metadata repository. Because it implements MOF, it is able to load any MOF metamodel (description of metadata) and store instances of that metamodel (the metadata conforming to the metamodel). Metamodels and metadata can be imported into/exported from MDR using XML that conforms to the XMI standard. Metadata in the repository can be managed programmatically using the metamodel-specific or reflective JMI API.

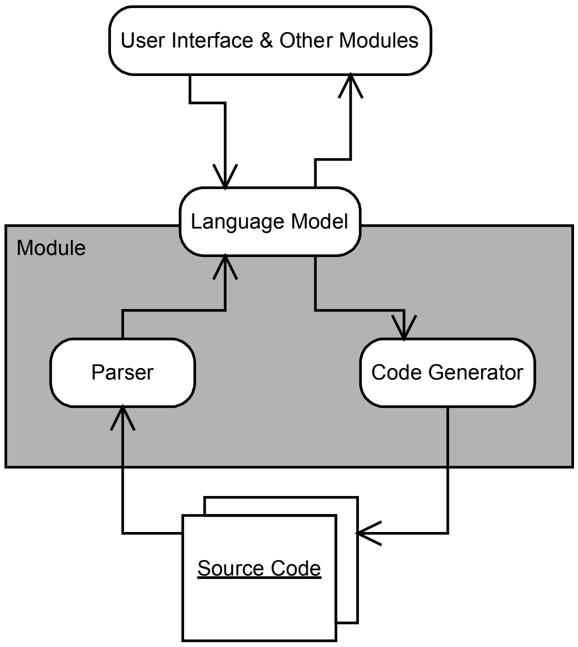
Motivation

MDR is being developed as part of NetBeans project. NetBeans can serve as a platform for basically any kind of Java application, but it is specifically very useful as a framework for modular integrated development tools. NetBeans itself provides a set of modules to support Java development. SunONE Studio builds on top of NetBeans to provide extended functionality – support for EJBs, web services, etc.

Every development environment needs to deal with metadata for the target platform/language. For example, a good Java IDE needs to be able to show what classes are in a user's project, what methods and fields each class has, etc. All of these are the metadata. They are needed for project explorers, object browsers, refactoring tools, and similar components of the IDE. Besides showing the metadata to the user, a modular IDE (such as NetBeans or SunONE Studio) needs to expose the metadata via API to other modules plugged into it. This enables better integration between modules and reduces duplication of efforts. To better understand this, consider the following example: In any

IDE that supports Java development, two typical components are the Java editor and the project explorer. Both of them need to work with Java metadata – the editor needs the metadata to implement code completion, and the project explorer needs it to be able to display the project structure (all Java packages in the project, Java classes in those packages, their methods and fields). It would obviously be very ineffective if both the editor and the project explorer had their own ways of extracting the metadata they need from the Java source code by having their own separate Java parsers, for example. A better approach is to have one module that takes care of providing metadata to the IDE for a particular language (Java in this case) and expose these metadata via API. The editor and project explorer would then use this API to work with the metadata.

The architecture of such a language support module would typically look like what is shown in the picture below.



The module would contain a language parser that is used to extract the language metadata from the source code. The metadata would be exposed to other modules via a set of APIs – in the picture these APIs are called "Language Model", as they typically reflect the structure (abstract syntax) of the language (or its subset that is relevant to the IDE). The module would handle synchronizing any changes made to the source code with the metadata exposed by the language model (by watching the underlying source files, reparsing them, and sending change notifications to the module's clients in case of any changes). To support the opposite direction (synchronizing source code with changes made to the metadata), the language support module would contain a code generator. So if one of the language model's clients (for example, a project explorer with an editable property sheet for the displayed elements) would make changes to the metadata (e.g. the user changes name of a class in the property sheet), the code generator would take care of propagating these changes to the source code.

Let's look closer at the "Language Model" component in the diagram above. As noted earlier, it is basically a set of APIs that provide access to the language metadata. The language model is present in almost all modular IDEs (in NetBeans it is the org.openide.src package). Typically these APIs are hand-written, which may have several undesirable consequences, such as poor maintainability or inconsistencies across language models for different languages. (If the IDE supports more than one programming language, an API for each language is typically written by a different person, using different kinds of patterns, coding style, etc.). In addition, the metadata exposed by a language model are often not persistent. These metadata are extracted from the source code on demand and garbage-collected as soon as they are not used. This makes it impossible to perform more complicated queries on the metadata, such as "find where a given method is used in my project" and perform refactoring. The reason for this is that all of the project sources would need to be parsed for each such query and since the extracted metadata are not persistent, they would actually be lost immediately after returning the results of the query. Thus for the next similar query the whole parsing would need to be repeated. This problem can be solved by implementing a mechanism for keeping the metadata persistent. However, this mechanism is not simple to develop and maintain – definitely not if each different language support module has its own way of making the metadata persistent.

So here is where MDR comes into play.

Functionality Provided by MDR

As noted earlier, MDR is a repository for the metadata described in MOF. One can just model the language using UML (in a UML 1.4 XMI compliant tool, such as Poseidon for UML 1.5, or MagicDraw 6.0) similarly to what we did for XML in the above example. This model of the language is the only thing that a language support module developer has to do to create the "Language Model" component of the module. MDR generates the rest – using the UML2MOF tool (available from MDR project website) the UML model of the language can be translated into MOF, then be loaded into MDR. JMI interfaces for accessing the metadata for the given language can be generated from it. (These interfaces will be used by other modules to access the language metadata.)

Additionally, MDR also automatically provides implementation of the JMI interfaces generated from the model – this implementation automatically stores all the metadata using the storage implementation provided (currently a persistent b-tree and a transient in-memory implementations are available). To enable MDR clients to respond to any changes in the repository, the implementation of the JMI interfaces provided by MDR also handles notifying all the registered listeners of any changes in the metadata they are interested in.

Other features include simple support for transactions (multiple read-only or single read/write transaction at time), additional indexing (enables object indexing using a specified combination of values of several attributes/references), storage federation and others. For full list of the features and documentation on them please visit the MDR website (http://mdr.netbeans.org) or see the references at the end of this document.

Use Cases

As shown above, the main use case for MDR is as an integration platform for various plugins/modules in a modular environment. It can be used as a metadata interface between various pieces of software.

The MDR is of course also a repository – it can be used as a persistent storage for metadata of any kind.

Thanks to these features, MDR fits perfectly into architecture of modular IDEs such as NetBeans, enabling support for plugging in new kinds of metadata (such as new language models). For some languages the metamodels already exist – UML being the most important example. MDR is even more important for supporting languages like UML, as UML has no official textual syntax besides UML models stored in XML files conforming to XMI specification. MDR can automatically store/import all the metadata using XMI, so having MDR built into a tool enables interoperability with existing UML tools and other MOF or XMI-compliant software. Also having language metadata persistently stored in MDR makes it possible to easily implement modules for refactoring, code analysis, design patterns and so forth.

Model-Driven Development

As J2EE and other technologies get more complicated and integration between various technologies (EJBs, web services, databases, etc.) gets more and more important, it is obvious that things are no longer manageable at the code level. A model-driven approach to software development is needed. OMG recognized this fact, thus the **Model-Driven Architecture (MDA)** – a framework of standards that enable model-driven development – became its mainstream vision. Since MOF is a key component in this framework, implementation of MOF becomes an essential component of MDA tools. As the metadata for all languages an IDE supports are stored in one place (in an MOF repository such as MDR), it is easier to build transformation/synchronization modules between, for example, UML models (capturing an application logic at a higher level of abstraction) and the technologies the application should be deployed to – Java, JSPs, EJBs, database, etc.

Architecture

Let's look at the high-level architecture of MDR. MDR consists of several libraries. Although it is developed as part of the NetBeans open source project, its core is completely independent from NetBeans and can be used by any tool (even if the tool is independent of NetBeans) as a standalone MOF-compliant metadata repository. Here are the components that the MDR consists of:

- JMI tools – Standalone set of tools that should work with any JMI implementation. This includes XMI importer/exporter, JMI interfaces generator, and some other generic utilities not specific to MDR or NetBeans.
- MDR engine – Standalone metadata repository that implements basically all MDR functionality. Internally uses JMI Tools.
- MDR module – NetBeans module that integrates MDR engine with the NetBeans platform.
- NetBeans MDR Explorer – NetBeans module that enables browsing and managing the metadata stored in the MDR.

- Other MDR-based tools – MDR Ant tasks for generating JMI interfaces, importing/exporting XMI, UML2MOF Tool for converting UML models to MOF, etc.

Current Status

MDR has been available on open source since January, 2002, hosted on the NetBeans.org open source project. It is now stable in terms of functionality – it passes all 1086 TCK tests for JMI and it is already used by several companies in their commercial products (such as Poseidon for UML 1.5 from Gentleware). However, APIs for MDR as it stands today are still in flux and should be stabilized shortly as part of an upcoming release of NetBeans.

The MDR team is now focusing on the integration of MDR with the NetBeans Java module (a module that adds support for Java language into NetBeans) so that the metadata the current Java module produces are persistently stored in MDR and exposed via JMI API. This will make it possible to extend the current Java module functionality by adding features such as renaming classes, methods and fields without breaking the code (this involves changing the name of the feature on all its occurrences) and advanced navigation (such as finding all implementations of an interface or all the places where a value of a given field is changed or read).

Summary

MDR is a standalone metadata repository implementing MOF, XMI and JMI standards. It provides a runtime access to the metadata via a set of generated metamodel-specific APIs or the metamodel-independent reflective API. The metadata managed by MDR can either be persistent or transient.

MDR serves as an integration facility used by individual parts (components) of a software to communicate with each other. Additionally, MDR as a persistent storage makes it possible to implement advanced IDE features like code analysis, refactoring or model-to-model transformations (such as generation of Java code from UML models – i.e. transformations of UML metadata to Java metadata).

MDR as an implementation of MOF is a key technology needed for tools to implement MDA.

References

http://mdr.netbeans.org/docs.html
- Documents section of the MDR Project website

http://www.netbeans.org
- NetBeans.org open source project

http://www.omg.org/technology/documents/formal/mof.htm
- Meta-Object Facility (MOF) specification

http://www.omg.org/technology/documents/formal/xmi.htm
- XML Metadata Interchange (XMI) specification

http://java.sun.com/products/jmi
- Java Metadata Interface (JMI) specification

http://mdr.netbeans.org/uml2mof/profile.html
- UML Profile for MOF

http://www.omg.org/mda
- Model-Driven Architecture (MDA) website

http://www.omg.org/technology/documents/formal/uml.htm
- Unified Modeling Language (UML) specification

http://www.omg.org/technology/documents/formal/cwm.htm
- Common Warehouse Metamodel (CWM) specification (metamodels for data warehousing – includes metamodels for relational databases, XML, multi-dimensional databases, etc.)

http://www.w3.org/XML
- XML standard

http://www.gentleware.com
- Poseidon for UML website

http://www.magicdraw.com
- MagicDraw website