In the second part of this series on UML and XML, Benoît introduces the UML metamodel. He proceeds to XMI, the XML-based specification for the exchange of models. He then shows how to map from the metamodel to XML schema. As an illustration, he includes two stylesheets that provide simple round-trip engineering between UML and XML. Share your thoughts on this article with the author and other readers in the accompanying discussion forum. (You can also click Discuss at the top or bottom of the article to access the forum.)

View more content in this series

This column is currently focused on modeling, UML, and XML. More specifically, I am exploring the use of UML modeling for XML development and in particular how XSLT stylesheets can help through automatic derivation.

As XML has become a common feature in development projects, many developers have grown interested in integrating XML with the rest of their development. While many organizations still rely on ad-hoc tools for XML development, the trend is towards adopting the same methodology for XML -- or at least one common set of tools -- that is already in use for other development needs, such as Java technology, databases, or the Web.

**Automatic derivation**

As discussed in the previous column, a model is a simplified description of a system that can assist in calculations and predications. In the context of this article, the system is always an XML vocabulary.

Figure 1 illustrates the modeling cycle as a continuum of models. The first models are drawn on the whiteboard (or on a sheet of paper) and tend to be informal. At this stage, the goal is to give all participants (users, developers, designers) a chance to express themselves freely.
The next step is to draw a UML model (or several models if the vocabulary is complex). The UML model is more refined and formal, but it remains synthetic and readable because it is intended primarily as a communication device between the team members.

The last model is the XML schema, which is the most precise of them all. Its goal is to allow the parser to validate XML documents against the vocabulary definition so it can forego readability in favor of precision.

The major difference between all these models is their goal: from informal communication to precise, formal validation by the parser. The difference is not in the nature of the models (simplified description of an XML vocabulary), but in the level of assistance each model provides.

If you think of a continuum of models, from the least precise to the most formal, it makes sense to look into automatic derivation -- the process of generating one model automatically from an earlier model. Obviously, automatic derivation works well only if the two models are equally descriptive, which sort of conflicts with the idea of some models being more descriptive than others. Addressing the different levels of description in the models will be the topic of the next column; here, I will focus on derivation.

**XML Metadata Interchange (XMI)**

You will recall from the previous installment that I implemented automatic derivation through XMI and XSLT. Assuming that you are already familiar with XML schema (if you're not, see Related topics), I will introduce XML Metadata Interchange (XMI) in this section.

**Vocabularies and compatibility**

XMI is a sophisticated specification (version 1.2 is over 400 pages), so, in this article, I will limit myself to the bare minimum description needed for automatic derivation.

XMI does not specify an XML vocabulary, but rather an algorithm that generates vocabularies for metamodels. In other words, XMI does not define Class, Attribute, Association, or other tags as you would expect. Instead, XMI specifies how to create tags for concepts in a metamodel. I know that’s a lot of models to work with, but bear with me -- it will become clearer in a moment.

Therefore XMI is not so much a vocabulary as a framework. Unfortunately, this means that no two tools interpret this framework in the same way. Differences also exist between different versions of the same tool: Rational Rose originally supported XMI through an add-on developed by Unisys.

---

**Figure 1. A continuum of models**
The latest versions of Rational XDE have built-in support for XMI, but it's a slightly different variant. The differences are not necessarily significant, but they may cause incompatibilities. In practice, it makes sense to target your stylesheets to the one or two tools that are used in your community and not worry about the rest.

In this article, rather than adopting one specific version of XMI, I will stick with the examples published by the OMG. Although no tool is directly compatible with the samples, this is good middle ground. Adapting them to your tool of choice will not be difficult.

The XMI header

Although it mostly specifies an algorithm, XMI also defines a few tags and attributes. You will need the following:

- **XMI** is always the root element. It must have an `xmi.version` attribute (valid versions are 1.0, 1.1, 1.2, and 2.0).
- **XMI.header** is a placeholder for information on the model. Its most important children are `XMI.documentation` and `XMI.metamodel`.
- **XMI.documentation** holds end-user information as these children elements (whose names are self-explanatory):
  - `XMI.owner`
  - `XMI.contact`
  - `XMI.longDescription`
  - `XMI.shortDescription`
  - `XMI.exporter`
  - `XMI.exporterVersion`
  - `XMI.exporterID`
  - `XMI.notice`
- **XMI.metamodel** records the metamodel to which the XMI algorithm has been applied -- in this case, the UML metamodel (XMI works with other metamodels such as Metaobject Facility, MOF, also published by the OMG).
- **XMI.content** contains the actual model.
- `xmi.id` and `xmi.idref` are attributes for encoding links: `xmi.id` is an element identifier that must be unique; `xmi.idref` is a reference to an element by its identifier.

The metamodel

The UML **metamodel** is a model that describes the UML language -- specifically, it describes classes, attributes, associations, packages, collaborations, use cases, actors, messages, states, and all the other concepts in the UML language. For coherence, the metamodel is written in UML.

The prefix "meta" indicates that the metamodel describes a model of a model. Likewise, XML is a metalanguage because it's a language that describes languages.

The UML metamodel is published in the UML specification. More specifically, XMI uses the "UML Model Interchange" described in chapter 5 of the UML specification (see Related topics).
Be warned that the UML metamodel is rather large and intimidating. I can only give you a flavor for it in this article. Figure 2 is an excerpt from the metamodel that describes the class, one of the central concepts in class diagrams.

**Figure 2. the metamodel for a class**

In the metamodel, the class concept is modeled as the metaclass **Class** which inherits from the abstract metaclass **Classifier**. Classifier is the parent for Class, Interface, and Datatype (the latter two are not represented in Figure 2). The inheritance chain continues to: **GeneralizableElement**, which represents all concepts that can be generalized (inherited from); **ModelElement**, which represents all abstractions in the model (such as namespace, constraints, and class); and finally **Element**, the topmost metaclass. Each of these metaclasses has attributes from which Class inherits.

**XML variations**

Few vendors document their XMI variant. The workaround is to create a small model and export it. Open the file in a text editor and review it.

The XMI elements and attributes (**XMI.header**, **XMI.content**, or **xml:id**) serve as a roadmap through the file.

Look for the main elements from the metamodel (such as Class, Attribute, Association) and see how they are mapped into XML. It helps if you have an excerpt from the metamodel handy.

The differences are mostly cosmetic: It seems no two applications use the same namespace. Some applications encode metamodel attributes as XML elements while others use XML attributes (as in **Listing 1**). In practice, it is very easy to recognize the difference when you compare to the metamodel.
A composition exists between Classifier and Feature, which is the parent of StructuralFeature. Attribute is derived from StructuralFeature.

Confused by the metamodel? Try to forget it's a metamodel, try to forget it's about UML, and look at it as an ordinary model. Figure 2 is simply pointing out the concept of Class, which is a highly specialized element that is related to interface and data type (through its inheritance from Classifier). Class has a name, visibility, and many more attributes. Finally, there is an association between Class and Attribute.

So Figure 2 formally expresses that a class has a name, visibility, and other properties, and that it may have attributes. Indeed, Figure 2 is the definition of a UML class. If you find this confusing, it's probably because the definition itself is written in UML!

I have intentionally simplified Figure 1 to ignore namespace, constraint, stereotype, inheritance, and many other aspects of what makes a class a class. Trust me, they are included in the complete UML metamodel but they are not useful for this article.

Why bother with the metamodel? Because when you feed it to the XMI algorithm, you get an XML vocabulary for UML. As an example, Listing 1 is an XMI representation of Figure 3 (using the variation of XMI illustrated in the specification -- see above):

**Figure 3. A UML model for an address**

<table>
<thead>
<tr>
<th>Address</th>
<th>name</th>
<th>street</th>
<th>zip</th>
<th>region</th>
<th>city</th>
<th>country</th>
</tr>
</thead>
</table>

Listing 1. The address exported to XMI

```xml
<?xml version="1.0"?>
<XMI xmlns:UML="org.omg/UML/1.4">
  <XMI.header>
    <XMI.exporter>ananas.org stylesheet</XMI.exporter>
  </XMI.header>
  <XMI.metamodel xmi.name="UML" xmi.version="1.4"/>
  <UML:Model xmi.id="M.1" name="address" visibility="public" isSpecification="false" isRoot="false" isLeaf="false" isAbstract="false">
    <UML:Namespace.ownedElement>
      <UML:Class xmi.id="C.1" name="address" visibility="public" isSpecification="false" namespace="M.1" isRoot="true" isLeaf="true" isAbstract="false" isActive="false">
        <UML:Classifier.feature>
          <UML:Attribute xmi.id="A.1" name="name" visibility="private" isSpecification="false" ownerScope="instance"/>
          <UML:Attribute xmi.id="A.2" name="street" visibility="private" isSpecification="false" ownerScope="instance"/>
          <UML:Attribute xmi.id="A.3" name="zip" visibility="private" isSpecification="false" ownerScope="instance"/>
          <UML:Attribute xmi.id="A.4" name="region" visibility="private" isSpecification="false" ownerScope="instance"/>
        </UML:Classifier.feature>
      </UML:Class>
    </UML:Namespace.ownedElement>
  </UML:Model>
</XMI>
```
Notice how the XML elements and attributes in Listing 1 match the classes and attributes in Figure 2. You've now come full circle: The XMI document is a direct representation of the UML metamodel because the UML metamodel is a description of UML itself.

**Presentation aspects**

A portion of the UML metamodel deals with the visual representation of concepts -- where to draw the concepts on screen. I don't process that information in my stylesheets for two reasons:

- It is not needed when deriving an XML schema from the UML model.
- It is extremely difficult to produce the visual output when deriving the UML model from an XML schema. A more reasonable option is to open the model in a modeling tool and take a few minutes to prepare the visual representation of the model on screen. The hardest work (getting the definitions right) has been done by the stylesheet.

**XSLT stylesheets**

Now that you have the key to reading XMI files, it's easy to map XMI tags to their XML schema equivalents. One possible mapping is:

- **UML:Model** becomes `xs:schema`; its target namespace is derived from the model name.
- **UML:Class** becomes a global XML element declaration (`xs:element`).
- **UML:Attribute** become a local XML element declaration (`xs:element`).

Listing 2 is an XSLT stylesheet that implements the mapping:

**Listing 2. XML schema derivation**

```xml
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:UML="org.omg/UML/1.4"
    exclude-result-prefixes="UML"
    version="1.0">
    <xsl:output indent="yes"/>

    <xsl:template match="XMI[@xmi.version='1.2']">
      <xsl:apply-templates select="XMI.content/UML:Model"/>
    </xsl:template>

    <xsl:message terminate="yes">Unknown XMI version</xsl:message>
</xsl:stylesheet>
```
Obviously, the stylesheet in Listing 2 is still very limited (and it does very limited error checking) because it only supports a small subset of the UML metamodel. It ignores packages, interfaces, associations, and more. You can enrich the stylesheet and support those concepts with a simple extension of the process I’ve shown you so far: Study the appropriate portion of the UML metamodel, define a mapping to XML schema, and implement it.

Vice versa

Listing 2 is very handy if you follow the normal modeling workflow: from the least detailed to the most detailed model. Frequently you will find that an XML schema already exists, and that it should serve as the starting point for your work. It would be tedious to recreate the UML model, so a stylesheet that implements the reverse mapping is handy. Listing 3 is an example:

Listing 3. Reverse derivation (from XML schema to UML)
Towards more comprehensive stylesheets

To say that the stylesheets I have introduced in this article are simplistic would be an understatement. They are less than 50 lines long and deal with a small subset of the UML metamodel. Real-world stylesheets recognize many more UML concepts, and typically weigh in at 500 lines or more. My goal in this installment has been to introduce the concepts behind automatic model derivation:

- Even the models (UML, XML schema) are represented as a data set; this special data set is called the metamodel.
- You can establish a mapping between the UML metamodel and an XML schema.
- You can implement the mapping in XSLT stylesheets.
- UML and XML schema are just different representations of the same reality; they differ because they serve different goals.

In this article, I have had to make simplifications. If you try to extend the stylesheets from Listings 2 and 3, you may encounter two problems:

- The UML model may not be specific enough (because it's a high-level view) to enable meaningful derivation of XML schema (low-level, detailed model).
- You may find more than one sensible mapping between the UML metamodel and XML schema.
Solving these two problems is the topic of my next two column installments.
Related topics

- Find hundreds more XML resources on the developerWorks XML zone, including previous installments of Benoît Marchal's Working XML column.
- Find out how you can become an IBM Certified Developer.