This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/jw-09-2005/jw-0905-xml.html

OO, XML, and GML: Are angle brackets a flexible modeling material?

Using XML comfortably among objects

By Milan Trnini&#263;, JavaWorld.com, 09/05/05

XML's emergence did not initially make our lives easier—at best, they did not change much. We quickly started writing our many data formats using angle brackets, which looked neat, but did not make much of a difference. But, over the years, many new XML-based specifications with disparate and often unique grammars have surfaced. How these grammars, and XML in general, fit into the world of object-oriented technology, at first, did not seem completely clear.

Essentially, we have one more model-mapping problem on our hands. This article looks at XML from the model perspective and tries to answer questions like: What is the model of XML? Does it have one at all? How do we use it while preserving the benefits of object-oriented programming and/or other models (such as relational or entity-relationship). Do the object-oriented model and the XML model fit well together?

To help answer these questions, a few example model rules from Geography Markup Language (GML) are described.

XML model

XML has a very simple model. Actually, simplicity was a key requirement in the development of XML. Several of the explicitly stated development goals in the XML specification directly support this quality.

The specification also states that XML shall be generic, i.e., that "XML shall support a wide variety of applications." Simplicity and generality complement each other: by extending a simple model, you can apply it to many different domains.

With XML, the initial learning curve is extremely short to the extent that many deem XML as trivial. However, using XML in real-world applications beyond recording simple data structures (e.g., user preferences or configuration values) and effectively exploiting XML's real power takes some significant time to master. XML's simplicity can often be deceiving.

Proof (although a tautology) of XML's simplicity is that, even with just a few rules taken from the specification, you can quickly produce "correct" XML content. For example, using just a single rule describing how elements are created and nested, you can create an XML file that encodes user preferences for a text editor application. An example of such a file is shown in Listing 1:

Listing 1. Simple XML content

```
 <Preferences>
   <AutoSave>5</AutoSave>
   <RecentList>
      <File>C:\documents\TheArticle.txt</File>
      <File>C:\documents\MyNotes.txt</File>
   </RecentList>
      …
</Preferences>
```

The XML specification also introduces document schema—the description of the document structure and/or types. This brings in some complexity, but again, not initially. You are free to use any existing schema description language: document type definition (DTD), Resource Description Framework (RDF), XML Schema, or any other. The DTD schema for the example above is shown below:

Listing 2. A simple DTD definition

```
 <!ELEMENT Preferences (AutoSave, RecentList)>
<!ELEMENT AutoSave (#PCDATA)>
<!ELEMENT RecentList (File*)>
<!ELEMENT File (#PCDATA)>
```

Obviously, a simple document structure is not all there is to XML. As already mentioned, XML is highly extensible. While the basic structure of XML is simple, it does allow the creation of models of arbitrary complexity. And this is exactly what has happened: XML has served as the basis for many different grammars created within or outside the W3C (World Wide Web Consortium). Usually, all of these grammars together are called "XML technology."

XML in itself does not imply any particular semantics, but only a simple, hierarchical structure. Imagining the possibilities of creating derived models and mastering a sufficient set necessary for your application is the difficulty of XML. Except for basic rules from the specification, XML does not prescribe any guidelines as to how the derived models should be created—they are created, literally, in an open space. And XML is generic enough to support encoding of an extremely broad, virtually unlimited range of models: data encodings, data processing rules and constraints, programming languages, and many others.

Creating an XML derived model

So, how do you create a model using XML? First you must capture your domain model. This activity results in a vocabulary of terms representing entities in the domain and their relationships, which together implies certain semantic information. For example, you may want to encode a domain of spatial features and their properties (e.g., a building feature with a height property and a road with number of lanes). You can (and probably will) use some of the modeling technologies in the process as well and thus also introduce a technological domain. Therefore, you would use terms like entities and relationships, objects with properties and methods, relational tables, or others.

Expressing these entities and relationships with XML is the next step. Your initial decisions relate to the use of the XML model itself. For example, you can often use both attributes and child elements to encode some information about a domain entity. So what should you do? Do you distinguish between attributes and elements at all and, if so, what exactly is the distinction? An example of such a distinction can be found in Scalable Vector Graphic (SVG) grammar, where elements resemble methods and the attributes resemble method arguments. This SVG path element:

```
 <svg:path stroke-width="0.5" d="M10,10L20,20L30,30"/>
```

can be easily seen as a call to a method defined as:

```
 public void path(float stroke-width, String d);
```

You may also decide that certain rules govern the use of elements alone and that not all elements in your XML content are the same. A simple rule may be that a root element differs from all the others in that it only serves as the container.

You probably need to establish more rules in your model—rules on how XML entities are used to express different types of objects and their relationships that exist in the domain. The complexity of the model is completely open. But note the trade-off between a too-simple model that moves complexity elsewhere (into the handling code) and a too-complex one, which is hard to maintain.

For example, say you want to create a route-finding application that calculates the shortest route between two points. The application operates on the data encoded in XML. If you encode only geometries and coordinates of roads and streets in the data, then the application must perform some fairly complex calculations for each client request: it will have to read the coordinates, infer the connectivity between the roads, apply the directional rules, and calculate the distance. On the other hand, a more complex XML model may include the topology information, in which case, the application doesn't create the topology network, but only calculates the distances.

Whatever the model, certain semantic information must be carefully and purposely captured, designed, and encoded. This step is extremely important, and, although it's a usual design step, it is often ignored. Improper design of semantics information yields, in worst cases, situations in which the resulting XML content exposes clearly and unambiguously to the developer only the parent-child and element-attribute relations—the XML model itself. The domain semantics slide completely into the code handling the content, i.e., the domain model becomes hard-coded. This reduces the software's maintainability, extensibility, and reusability.

Document type definition
The reason DTD grammar for defining XML content has been quickly surpassed by other definition grammars is precisely because it can capture only the simplest, structural relations between entities.

The domain model expressed in XML can take various forms, and setting any kind of firm rules for design and encoding is really impossible. Instead, let's look at an interesting example—the model of Geography Markup Language.

GML model

Geography Markup Language, an open specification of the Open Geospatial Consortium, is an XML encoding of geospatial information. Its purpose is manifold: transport encoding, data storage encoding, and geospatial Web modeling and implementation language. The GML model is represented by a set of mandatory rules and a more relaxed set of guidelines. Both are described in the GML specification.

Geospatial Web (geo-Web)

The term geospatial Web is related to expressing various relations between geospatial entities (such as features, geometries, and topologies) in a distributed manner, using the Internet as the medium and distributed information systems as system components, similar to the HTTP/HTML Web we use today, but with a content-wise focus on sharing of geospatial data.

The GML model is based on entity-relationship (E-R) and object-oriented (OO) concepts, and its type system reflects that fact. Thus, the GML model states that there are objects: features, such as `Road` or `House`; geometries, such as `Point` or `Polygon`; and topologies like `Node` or `Edge`; and others. There are also properties of objects, e.g., an integer `numberOfLanes` property of a `Road`, or a geometric `extentOf` property of a `House`. Every property has a value, which is again an object. This forms a triplet *object-property-value* model, which maps nicely to both *entity-relationship-entity* (E-R) and *object-field-value* (OO) models. It should be noted that GML initially employed the existing RDF model that uses the terminology *subject-predicate-object*. GML was initially defined using RDF as the schema language.

Further, GML encodes the real-world properties of real-world objects (such as geometry of a `Road`) as elements. On the other hand, properties pertaining to the information systems that process the information are encoded as attributes (e.g., a system identifier or an HTTP reference to a related object in the system).

To encode an object-property-value triplet, such as `House-extentOf-Polygon` (the *polygon* is the *extent* of the *house*), you use the elements in such a way that a child element of an object is always its property, while the child element of a property is its value. This further implies that an object element can have any number of child elements (an object can have an arbitrary number of qualifying properties), while the *property element* can have only a single child element. (**Note:**
The GML model states that a property can have a single value only, which sounds natural. The exception to this rule is collection types. Please consult the GML specification for further explanations.) Thus, not all the elements in the GML content have the same significance and qualification. The encoding of the triplet above in GML is shown below:

Listing 3. GML encoding of an object-property-value example

```
 <House>
   <extentOf>
      <Polygon>
         …
         coordinates and other content of geometry
         …
      </Polygon>
   </extentOf>
</House>
```

To improve the readability, the *`lowerCamelCase`* naming convention is used for properties and *`UpperCamelCase`* is used for objects.

The need for such a model and the benefit it brings can be described in an example. Let's encode the polygon geometry of a radio tower feature depicted in Figure 1.

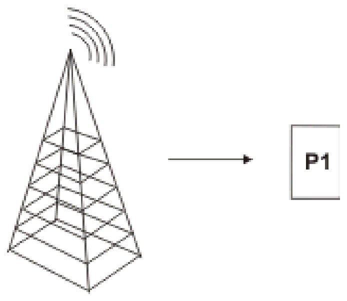

Figure 1. Radio tower and its polygon geometry

Creating a simple XML (non-GML) snippet that encodes this feature is easy:

Listing 4. Simple non-GML XML encoding of the RadioTower feature

```
 <RadioTower>
   <Polygon>
      …
      coordinates and other content of geometry
      …
   </Polygon>
</RadioTower>
```

Expanding the example, you may also decide to encode the geometry of the building adjacent to the tower, as symbolized in Figure 2:

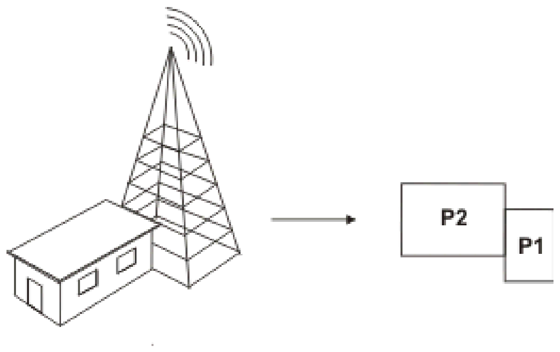

Figure 2. Geometries of the radio tower and the adjacent building

When you add another `Polygon`, you now obviously have to distinguish between the polygon representing the building and the one representing the tower. You should also encode in some way the role names of each of the polygons (like naming the fields in a class). In GML, you do that by using an intermediate element representing the property (that is, the relationship and role name). Proper GML encoding is shown in the following code fragment:

Listing 5. GML encoding of a feature with multiple geometries and geometric properties

```
 <RadioTower>
   <extentOfTower>
      <Polygon>
         …
         coordinates and other content of geometry
         …
      </Polygon>
   </extentOfTower>
   <extentOfBuilding>
      <Polygon>
         …
         coordinates and other content of geometry
         …
      </Polygon>
   </extentOfBuilding>
</RadioTower>
```

This XML content can be directly represented using entity-relationship and UML (Unified Modeling Language) diagrams as shown in Figure 3. In other words, those models and GML content map very nicely.

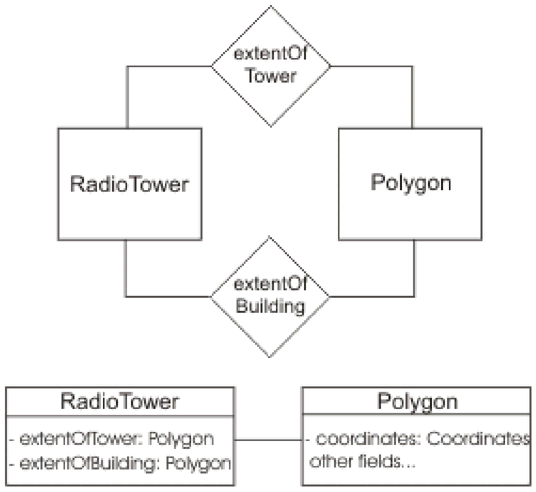

Figure 3. E-R and UML representations of the tower example

Besides modeling of associations, many other aspects can and usually should be addressed: the type hierarchy that must be designed, potential rules for extending the model further by the domain application developers, and others. The intention here is not to describe them, but only to use an example to emphasize the importance and the virtually endless flexibility of modeling.

Resolving issues

XML, as shown, does not coalesce with Java on its own. We must model and fit it nicely into the object-oriented world. However, even with dedicated effort, you must deal with issues that are in some way inherent in the context of this work. There are issues regarding the use of XML- or object-oriented-specific features, the model's extensibility, backwards compatibility, and performance—to name just a few—and these issues often depend on the domain itself. Some can be resolved at the model level and some on the technology level.

For example, XML Schema's derivation of types by restriction is not known in the object-oriented world. What can you do about it? You cannot map it directly, so you must include some rules in your model. The simplest rule can completely ignore this capability, which means to forbid it. Or you can have more flexible rules, where you allow the capability while defining the exact meaning of it and specifying the constraints on its use.

Another ordinary question is how do you implement the XML-expressed model in an object-oriented language? Do you use an existing API (e.g., DOM)? Do you create domain objects? If you use DOM, you get generality (e.g., `parent.getChildByName()`); if you use domain objects, you get focus (e.g., `house.getExtent()`). I use this rule of thumb: If your domain model is *closed*
and will not be extended by some more specific application schema (e.g., SVG), you can safely create and use domain objects. If the model is *open*
and meant to be extended (like GML), you should use a generic API like DOM. In both cases, a combination of DOM and domain objects can be used.

Complexity suspects

After spending some time and effort in learning a number of XML technologies (that is, models expressed in XML), you can easily arrive to the point where you don't label XML as "trivial" anymore, but instead accuse it of being extremely complex. Arguments for this stance often are:

1. Verbose in terms of number of elements/attributes.
2. Verbose in terms of naming (long human-readable names).
3. Too deep of a hierarchy. Unlimited nesting of elements is possible.
4. To understand data, XML Schema must be parsed and analyzed.
5. Schemas tend to be on the Internet—so there is possible need for complex schema management (as if data management is not enough).

It is important, however, to carefully examine these statements and issues in the light of your requirements and the purpose of XML. The analysis can easily reveal that the actual suspects are elsewhere:

1. **Verbose in terms of number of elements and attributes:** The **domain** is probably extensive. The domain of XML, i.e., the applicability of XML, is, as mentioned, extremely broad. Extensible grammars built on XML, such as GML, can also be broad—geospatial domain covers many specific subdomains like transportation, hydrography, or others. The broad applicability implies that the number of entities in the domain vocabulary is certainly not small.
2. **Verbose in terms of naming:** This is **true of XML** in general, but names can be easily coded/compressed for machine use and decompressed/decoded for human use.
3. **Too deep of a hierarchy:** This can simply mean that you have encoded **complex domain relationships**. So whatever complexity is introduced from the domain will reflect in the XML encoding and no more than that. I mentioned the trade-off between the two extremes earlier in this article.
4. **To understand data, XML Schema must be parsed and analyzed:** Knowledge about the **data schema** must exist somewhere—this was the case 30 years ago as it is today. Only before, much of the knowledge was captured in the handling code.
5. **Distributed schemas and data:**
   XML and related grammars are not distributed in themselves. The fact that the data is described and encoded using XML does not necessarily mean that you must have an Internet connection to process it. It is **the systems** that are distributed and built for distributed processing.

Essentially, XML and related grammars are a consequence and technology media for realizing domain requirements and established goals (distribution, automation, integration, interoperability, etc.). In other words, we have gotten exactly what we wanted.

Conclusion

XML has a well-defined model that specifies a simple hierarchical structure and relations. It is not object-oriented. Nor does it conflict with object-oriented programming. In the strict sense, XML and object-oriented models cannot be compared—they are completely different abstractions. XML is the basis, an alphabet that can be used to express concepts like object-oriented ones or pretty much any other model of a higher abstraction level, and in that sense, XML can fit nicely in the object-oriented world. And how do you, while using XML, successfully preserve the benefits that the object-oriented paradigm gives us? Model XML to fit the programming concepts you are using. Or, if that isn't at all possible, define precise mappings between XML grammar and your program and use a mediator component to do the transformation. But avoid, as much as you can, stretching and squeezing your processing code to fit XML—Java and other object-oriented languages were not made for that.

Author Bio

Milan Trninić is currently a product architect at Galdos Systems. He has been working in software development in various roles for 10 years. He has been developing Java and Web service applications for the past 6 years. He is one of the contributing authors of Geography Markup Language and the book Geography Mark-Up Language: Foundation for the Geo-Web, and has taught GML and XML courses on various occasions.