Xpriori Technical Paper

XML Information Modeling

Maximizing the Usefulness of Information

by Chris Brandin

Release 1.1

Xpriori, LLC
2864 S. Circle Dr.
Ste. 401
Colorado Springs, CO 80906

www.xpriori.com

INTRODUCTION

Is XML documents or data? Should XML be managed with a database or a document management system? Should XML be managed at all, or is it simply a data interchange standard? These are among the most common questions people ask about XML when they are trying to get a grip on what XML is. These questions get at what we *do* with XML, not what it *is*. Another question: Is a computer a word processor or a video game? Neither? Both? Actually, a computer is more. Word processors are computers, as are video games; but it doesn't work the other way around. We cannot define a computer in terms of a single application, or even a multitude of applications for that matter. Similarly, we cannot restrict our definition of XML in terms of its uses. There is much to be gained by seeing XML for what it is, rather than focusing on one functional use for it at a time.

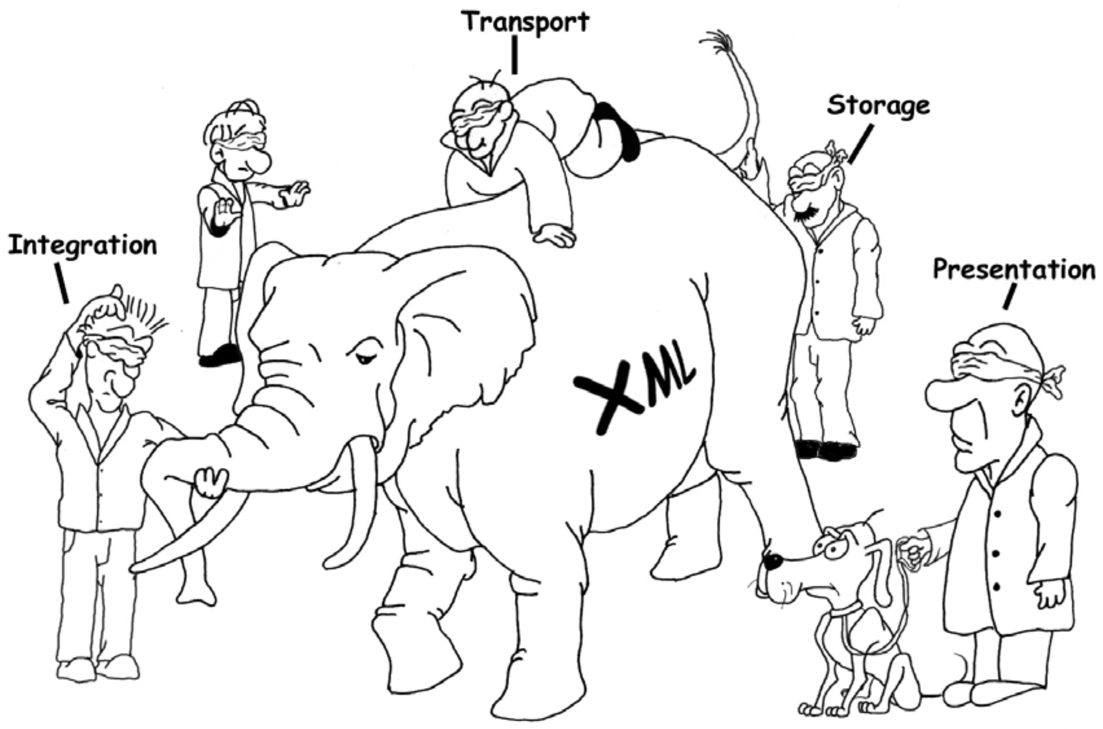


Illustration by: James Direen

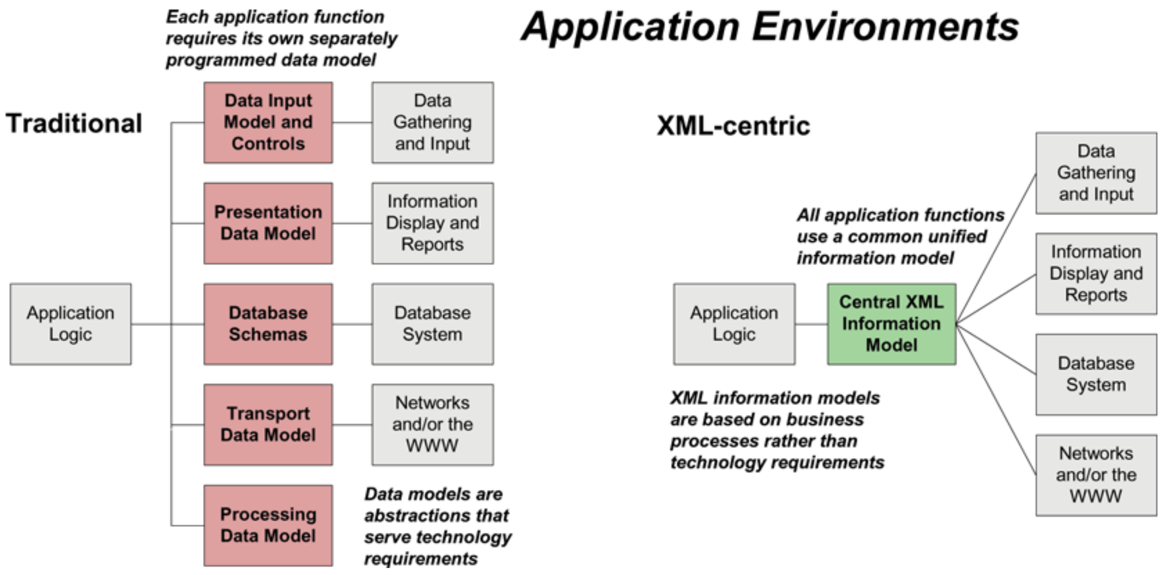
XML is a standard for expressing information in a complete form. It contains not only data, but also context and attributes. XML is, in other words, "informationally" complete. That is why it is useful for so many different things. XML plays an important role in unifying information from disparate applications. It can likewise play a role in unifying disparate functions within applications. Does this mean that we can use XML for every aspect of what an application does – gather, present, store, manage, transport, transform, and process information? In a word – *YES*. There are a number of important advantages in doing this, among them:

- XML is intuitive; it's easy to understand for technical and non-technical users alike.
- XML is simple and extensible; so we can build applications faster, and change them much faster.
- XML is standardized and ubiquitous; which means that applications can be built to be compatible with almost everything.
- XML is flexible and heterogeneous, so we can eliminate the rather excruciating exercise of trying to fit information in predefined containers.
- XML expresses complete information; so we can build unified and universal information models behind applications, instead of a series of redundant but different data models serving individual functions.

- XML encompasses standards (such as XML Schema) for providing robust and flexible controls for maintaining information integrity.

As long as XML was used as a container for data it was sufficient to consider only syntax when building documents. To do more, we must consider grammar and style as well. Obviously, proper syntax is necessary for XML to be usable at all. Good grammar insures that once XML information has been created, it can be subsequently interpreted without an inordinate need for specific (and redundant) domain knowledge on the part of application program components. Good style insures good application performance, particularly when it comes to storing, retrieving and managing information.

Most application programs encompass the same basic functions - input, presentation, communication, processing, and information management. Although the same information underlies each of these functions, different data models have traditionally been employed to accommodate application components accomplishing these tasks. Even with XML, remodeling the same information in different ways for each component of an application is inefficient and yields programs that are buggy and difficult to maintain and change. A better way is to create a central, unified model for information that can adequately accommodate all functions of an application.


Application Environments
Traditional
Each application function requires its own separately programmed data model
Application Logic
Data Input Model and Controls
Presentation Data Model
Database Schemas
Transport Data Model
Processing Data Model
Data Gathering and Input
Information Display and Reports
Database System
Networks and/or the WWW
Data models are abstractions that serve technology requirements
XML-centric
All application functions use a common unified information model
Application Logic
Central XML Information Model
XML information models are based on business processes rather than technology requirements
Data Gathering and Input
Information Display and Reports
Database System
Networks and/or the WWW

In order to create XML information models that can do more than accommodate single functions we have to look at XML in a different way – as an information domain rather than simply a transport standard. For simple data transport applications we can use XML any way we like, as long as it is syntactically correct. To do more, XML models must be built that are complete, provable, and unambiguous. Creating robust universal information models in XML is easier that it may seem at first. The first step is to understand the basic patterns inherent to information expressed in XML.

XML consists primarily of tags, attributes and data elements. Tags provide context; or in other words, tags describe what data elements in their scope are. Attributes provide information about, or indicate how to interpret data elements in their scope. Data elements represent data in the traditional sense. The structure of XML also provides information – about hierarchy, groupings, relationships, etc. It is possible to create meaningless XML. For example, one could create perfectly correct XML by taking the entire text from a telephone book and simply putting the starting tag "<Telephone_Book>" at the beginning and the ending tag "</Telephone_Book>" at the end. It would be perfectly correct XML, but not useful XML. In this case XML is being used solely as a container for a block of data, and provides no context for the information contained therein. At the other extreme we have XML where all information is expressed in a semantically meaningful way. For example, consider the following XML fragment:

```
<Telephone_Book_Listing format=text>
      <Name>
            <Last> Smith </Last>
            <First> Tracy </First>
      </Name>
<Telephone>
            <Area_Code> 719 </Area_Code>
            <Number> 555-1234 </Number>
      </Telephone>
</Telephone_Book_Listing>
```

The explicit patterns in the example above are:

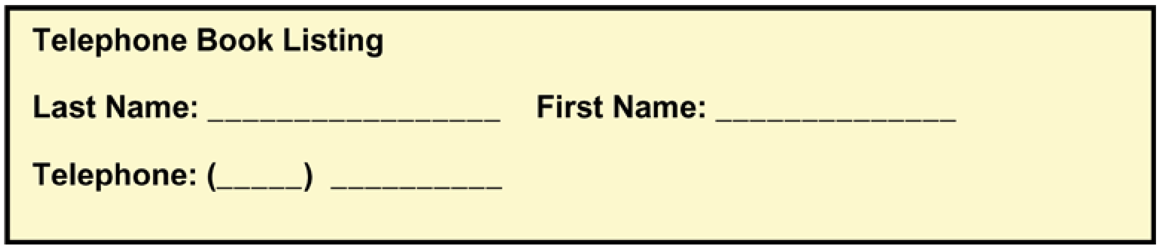
- This is a telephone phone book listing.
- All data elements are to be interpreted as text.
- The listing’s last name is “Smith”.
- The listing’s first name is “Tracy”.
- The telephone number’s area code is “719”.
- The telephone number is “555-1234”.

The implicit patterns in the example above are:
- “Smith” and “Tracy” belong to the instance group “Name” and are the last and first name respectively.
- “719” and “555-1234” belong to the instance group “Telephone” and are the area code and number respectively.
- Everything belongs to the instance group “Telephone_Book_Listing”.

Typically, most explicit patterns are converted to database columns with some or all of them being available as query terms. At least one XML information management system (NeoCore® XMS) automatically organizes itself around these natural XML patterns and indexes them all without the need for any database design whatsoever. Implicit patterns are used to determine groupings, relationships, and sometimes also convergence points for query set intersections.

Much like a Web browser can determine how to display HTML information based on presentation metadata embedded in the HTML, application components can determine how to treat and interpret XML information based on semantic metadata embedded in it. XML can contain any kind of metadata. This is where XML differs from HTML. Where HTML was targeted to a single function – the presentation of information, XML fulfills a more universal purpose – the complete characterization of information.

Key to creating useful XML is to create semantically meaningful XML first. The easiest way to do this is to simply create XML representations that are easy to read and understand by humans which are what we did in the sample XML fragment. If one were to create a manual entry form for the XML fragment above it would probably look something like this:

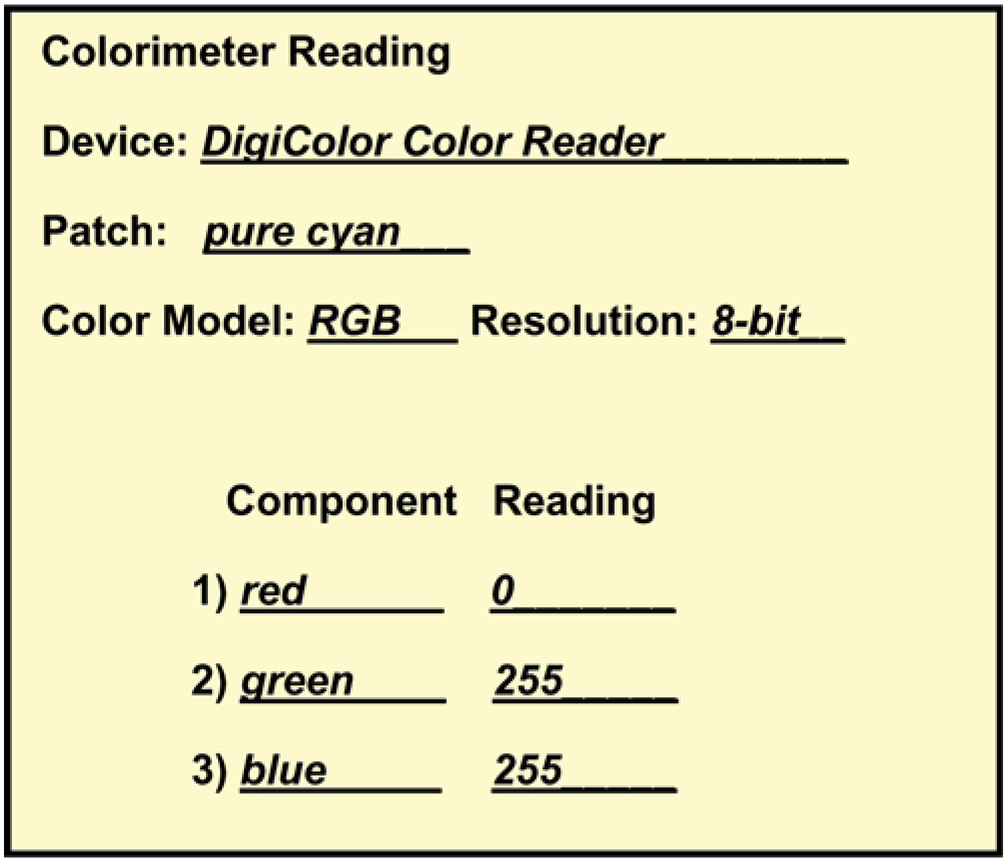

Telephone Book Listing

Last Name: _________________ **First Name:** ______________

Telephone: (_____) __________

Note that the XML fragment is a direct, and obvious, analog of how one would represent the information on a manual entry form. All we have done is represent the pre-printed parts as tags and the filled-in parts as data elements. The hierarchy we created is likewise obvious. This method of creating grammatically valid XML may seem so simple as to hardly be worth mentioning, but the fact is that grammatically valid XML is relatively rare. To illustrate why we will examine some common mistakes programmers make when creating XML.

For these examples we will use an application dealing with colorimeter readings. A colorimeter is a device that measures color using tri-stimulus readings using a number of color models. For example, a colorimeter can be used to measure computer monitor colors using red-green-blue components of light (this being the "RGB" color model). This particular colorimeter can provide readings in multiple resolutions. A manual form for transcribing colorimeter readings might look something like this after being filled-in:



Colorimeter Reading

Device: *DigiColor Color Reader*

Patch: *pure cyan*

Color Model: *RGB* **Resolution:** *8-bit*

	Component	Reading
1)	*red*	*0*
2)	*green*	*255*
3)	*blue*	*255*

One typical way to characterize this information in XML would be the following:

```
<colorimeter_reading>
      <device> DigiColor Color Reader </device>
      <patch> pure cyan </patch>
      <RGB resolution=8 red=0 green=255 blue=255 />
</colorimeter_reading>
```

Here, the information modeler has made a couple of optimizations in the interest of saving space. First, the “Color Model: RGB” field has been collapsed into the single tag “<RGB>”. This is a reasonable thing to do as the color model information arguably (and unambiguously) indicates what the readings are for, and therefore qualify as true context. Second, the readings themselves have been collapsed into three attributes – “red=0”, “green=255”, and “blue=255”. Expressing data elements as attributes is a common practice. Although syntactically correct, there are a number of problems with this approach:

- Attributes provide information about or information on how to interpret data elements in their scope. The readings are not attributes - they represent data from the colorimeter.
- The first attribute “resolution=8” is needed to correctly interpret the readings. Attributes apply to items in their scope, not each other. We have a mixture of attributes and data expressed as attributes with no indication of how they relate.\
- All four attributes apply to nothing because there is nothing in their scope.

Another common way this form might be modeled is:

```
<colorimeter_reading>
      <device> DigiColor Color Reader </device>
      <patch> pure cyan </patch>
      <color_model> RGB </color_model>
      <resolution> 8 </resolution>
      <band> red </band>
      <value> 0 </value>
      <band> green </band>
      <value> 255 </value>
      <band> blue </band>
      <value> 255 </value>
</colorimeter_reading>
```

Here, the entire contents of the form have been flattened into one level of hierarchy. There are several problems with this model:

- Two of the data elements, “RGB” and “8”, provide context that is necessary to interpret the readings correctly. They are not cast in a way that puts that the readings in their scope.
- Which readings values belong to which bands is ambiguous. Even though one could argue that the ordering implies grouping, a database may not know the difference and return false query results. For example, suppose you wanted to return any colorimeter reading that had a band of “green” and a value of “0”. According to this model, there is no unambiguous indication of which band belongs to which reading, and this XML fragment contains both terms; so a database could falsely return this fragment even though it actually has a green band reading of 255.
- This model is not extensible. If we wanted to include readings in another color model (CIE xyY, for example), there would not be a practical way to do so.

A third common modeling technique yields the following:

```
<colorimeter_reading>
      <field name=device> DigiColor Color Reader </field>
      <field name=patch> pure cyan </field>
      <field name=color_model> RGB </field>
      <field name=resolution> 8 </field>
      <field name=red> 0 </field>
```

```
            <field name=green> 255 </field>
            <field name=blue> 255 </field>
      </colorimeter_reading>
```

This is an example of a model that might be used by a forms generation program. It fulfills one objective – to accommodate the forms generation function; other application functions have not been considered. The shortcomings of this model include:

- Every tag (except the root) is the same – "field". This means that any query will result in a join operation involving nearly the entire contents of the database.
- Every name of every attribute is the same – "name". Attributes are not intended to indicate what data elements are, that is the function of tags. Essentially, this model is casting what should be tags as attributes.
- Because they are all the same, the tags and attribute names in this model serve no purpose. This model would have been better if the tags and attribute names had been eliminated and the attribute values had been used as tags.
- The color model and resolution fields are necessary to correctly interpret the readings, so they should be cast in a way that puts the readings in their scope.
- This model is flat, taking no advantage of hierarchy at all.

The most literal XML representation of the colorimeter readings form would be as follows:

```
      <colorimeter_reading>
            <device> DigiColor Color Reader </device>
            <patch> pure cyan </patch>
            <color_model> RGB </color_model>
            <resolution> 8-bit </resolution>
            <reading_1>
                  <component> red </component>
                  <reading> 0 </reading>
            </reading_1>
            <reading_2>
                  <component> green </component>
                  <reading> 255 </reading>
            </reading_2>
            <reading_3>
                  <component> blue </component>
                  <reading> 255 </reading>
            </reading_3>
      </colorimeter_reading>
```

Although this is grammatically correct, it is not optimal - nor is it particularly good style. This model represents another common practice – expressing context as data elements in name/value pairs rather than as tags. The weaknesses of this model include:

- Two of the information items, "color_model" and "resolution", do not mean anything by themselves. They actually represent metadata necessary to correctly interpret and understand the meaning of the readings themselves. Moreover, they are stand-alone (they have no scope), so their applicability is unspecified. The "patch" information item, on the other hand, is correctly cast a separate item because it is not needed to determine how to interpret the readings; rather it specifies what the readings are for.
- The "component" information items are not actually data. They represent the color bands for the readings - so they are really context. The major drawback to breaking information down into name/value pairs is that it can adversely affect performance because simple

queries against the information item result in join operations. This is often done to emulate the ability to support heterogeneous information in relational database management systems. If the information modeler finds himself doing a lot of this sort of thing, he may want to consider using an XML information management system that natively supports heterogeneity – performance and scalability will be much better.

There is no one “perfect” XML information model for this application. By applying a few simple techniques, however, a model can be built that is unambiguous, performs well, and will serve all components of the application program. First we examine each information item in order to determine what it really is – data, context, or attribute. Going down the list of items in the original colorimeter reading form, the following interpretations and actions would certainly be reasonable:

- “Colorimeter Reading” is the name of the form, so using this item as the root name is appropriate.
- “Device: DigiColor Color Reader” represents a stand-alone information item about the specific equipment used to gather the readings, so it will be represented as a stand-alone tag/data element pair.
- “Patch: pure cyan” represents a stand-alone information item about the entire dataset so it will be represented as a stand-alone tag/data element pair.
- “Color Model: RGB” specifies a context for all the readings. It could be cast as an attribute or as a tag. We will cast it as an attribute with the readings in its scope. The rational behind this has to do with how we might process colorimeter readings. Suppose that there were records that used other color models, CMY or CIE xyY for example. The color model attribute could be used to call up the appropriate parsers and interpreters to process the readings appropriately.
- “Resolution: 8-bit” directly applies to the interpretation of the readings below it, so it will be cast as an attribute encompassing all three readings.
- The “Component” items are arguably redundant. One could cast these as tags, but in the interest of efficiency we will eliminate them.
- The readings themselves each consist of two parts – a color band component, and the reading itself. We will make the color band component s tags, as they are context for the readings. The readings will be cast as data elements.

Converting all this into an XML fragment yields the following:

```
<colorimeter_reading>
      <device> DigiColor Color Reader </device>
      <patch> pure cyan </patch>
      <readings color_model=RGB resolution=8>
            <red> 0 </red>
            <green> 255 </green>
            <blue> 255 </blue>
      </color_model_RGB>
</colorimeter_reading>
```

This model fulfills all the requirements of good grammar and good style, yet remains relatively terse:

- All tags truly represent context and have the proper scope.
- There are no meaningless or redundant tags.
- All data elements truly represent data. No data elements are necessary for the proper interpretation of other data elements.
- Attributes are cast such that they directly apply to the interpretation of all data elements in their scope and no others.

- There are no tag/data element pairs masqueraded as attributes.

- The model is extensible. If we wanted to add readings in the CIE xyY model, for example, we could do so by adding a new readings section with the attribute "`color_model=CIE_xyY`" to the XML fragment.

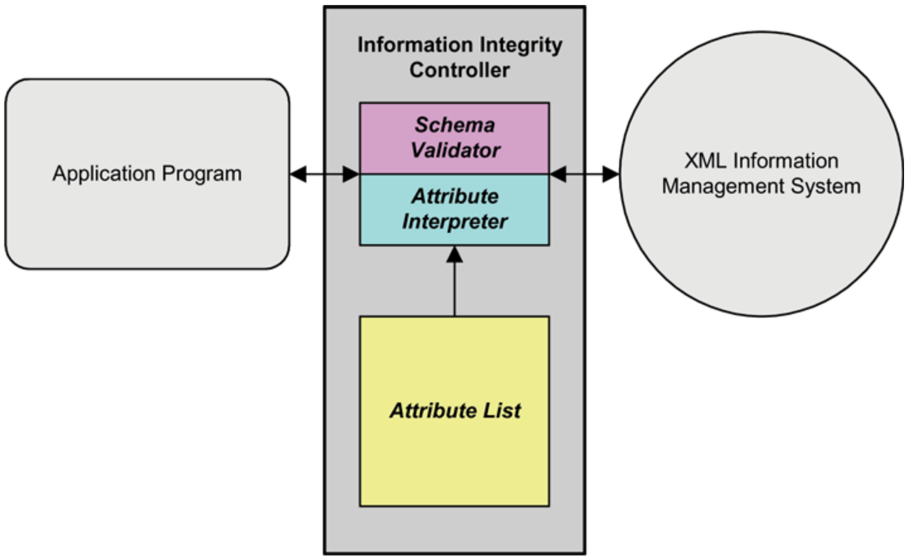
So far, we have discussed creating semantically valid XML and how it applies to how application components interpret information. The next step involves how applications can be architected to leverage XML as a central information model in an unambiguous and provable environment. To accomplish this, XML must be used in a somewhat more stringent way than many application developers are accustomed to. From the discussions above we can establish a series of rules for modeling information in XML:

- Data elements should be used to express only data, and not metadata.
- Tags should indicate what data elements in their scope are, not how to interpret them.
- Tags should apply directly to all data elements in their scope.
- Tag hierarchies should clearly group information items belonging together.
- Tag hierarchies should clearly separate information items not directly related to each other.
- Attributes should indicate how to interpret, or provide information about information items in their scope.
- Attributes should not be critical to interpreting what information items in their scope are.
- Attributes should apply directly to all information items in their scope, and no others.
- Attributes should be understood by all application components that will encounter them.

The last rule listed above is very important and bears some explanation. XML is very flexible; in fact too flexible the way it is often used. The one practice most responsible for making XML information models difficult to prove and control is the abuse of attributes.

Developers regularly use tag/data element pairs and attributes interchangeably in the interest of avoiding data-bloat (an unnecessary practice because even trivial compression techniques can eliminate that problem). This brings an unfortunate ambiguity to information expressed in XML. Attributes are intended to provide information about, or how to interpret items in their scope. If an application program encounters a tag that it does not understand, it will ignore everything in its scope – including all attributes. If an application program recognizes all the tags leading up to a data element, for example, it knows what the data element is; but if an attribute is encountered which is not understood, the application program may or may not know how to interpret the data element – and this situation is ambiguous.

In order to enforce information integrity controls we need a provable, unambiguous mechanism to do so. Attributes and XML schema definitions provide suitable mechanisms to do so - as long as an exception is thrown whenever an attribute that is not understood is encountered. To control XML information we need to use a combination of attribute interpretation and schema validation. This gives us an arbitrary and fine-grained degree of control unachievable with traditional databases. This should be done once, with a centrally controlled mechanism. To accomplish we need to architect enterprise applications in a new way. Traditionally, applications programs serve as user interfaces and database systems manage and control data. In the XML world, a three-component architecture should be employed: the user facing application, the XML information management system, and an information integrity controller (schema validator and attribute interpreter).



The information integrity controller can be implemented in a number of ways: as a server-side extension, as a standard application program component, or as a web service. The trick is to have a common information integrity controller for each category of application. In order to have applications interact with XML information in a consistent and provable manner, the following model can be adopted:

- If all tags leading up to a data element are understood, the application can assume it can correctly determine *what* that data element represents.
- If a tag leading up to a data element is not understood, the application must assume it does not have a complete context for that data element.
- An application component may ignore information containing a tag that is not understood.
- If all attributes having a data element in their scope are understood, the application component can assume it can correctly interpret that data element.
- An application must not assume it can correctly interpret any data elements that are in the scope of an attribute that is not understood, or violate it. It is up to the information integrity controller module (whether is a separate module or an integral part of the application) to indicate that such an attribute has been encountered and to which information items it applies. Such exceptions should be treated as catastrophic in so far as the information items in question are concerned.
- An application must be able to interpret attributes and their contents unless there is a specific rule stipulating that an attribute can be ignored (a font attribute, for example, may be ignored by application components not responsible for presentation).
- Schema validation violations should be treated as potentially catastrophic information integrity failures.

Following these XML information modeling guidelines is not substantially more difficult than building semantically weak XML, and the payback can be considerable. Not only can XML be used to integrate disparate data sources, but it can be used as a unified means to express the information underlying functional components of application programs as well. This results in better application programs that can be built faster and maintained with less effort. One thing is inevitable: as information technology progresses into things like the semantic Web and more and more computer systems need to be integrated, information expressed in XML will have to become increasingly semantically valid in order for us to be able to keep up. XML already contains all the elements necessary to do this – we just have to be a little more thoughtful about the ways we use it.

# # #