

# XML Usage Patterns

Eric Johnson <eric@tibco.com>

## Abstract

The flexible, but structured nature of XML documents leads to a variety of ways to implement XML effectively into your organization. XML is more than just a new way to transfer data - its true power is its ubiquity to transfer data, store configuration data, implement declarative or functional programming languages or describe data formats. This talk delves into the implementation strategies for XML that you may not have recognized, and some of the tools and design patterns that will speed your implementation of many different programming concepts.

## 1. Introduction

### 1.1. Why Define Usage Patterns

The question is, why define XML usage patterns, and why use them? The patterns described below define some common ways of employing XML and XML technologies. If nothing else, these usage patterns will help establish a common vocabulary, often a stumbling block for people new to a particular technology. And for people who are already familiar with the technologies involved, it gives them leverage to quickly communicate their ideas to others, without having to rehash details. Instead of spending time discussing what you mean by the "Declarative Language" usage pattern, you may simply refer to it.

Once we start establishing various patterns for using XML, we can also talk about the characteristics of various solutions, and hopefully, the optimal characteristics. Just as Design Patterns [[Design Patterns](#)] (Gamma, Helm, Johnson, Vlissides) establishes a basis for quickly understanding the tradeoffs of a particular design when writing software in C++ or Java, by defining usage patterns, we can more rapidly comprehend the trade-offs of a particular solution. If you suspect that adopting XML will help your solution, usage patterns can help you quickly understand where some incredible gains can be accomplished rapidly. In short, the usage patterns can help

you to make strategic solutions about where to add or augment your XML use, without having to understand all of the tactical development issues that might arise.

I cannot begin to think that I might identify all the usage patterns there might be, or even to explain them all in a way that makes sense in every situation. What I hope I can do, though, is give you a firm basis for moving forward with these ideas, and others, as you figure them out.

Briefly then, I hope to help you gain:

- A common vocabulary
- A way of identifying potential benefits to a software product or solution
- A basis for making strategic solutions for certain classes of problems
- A foundation for establishing new, useful patterns of your own

## 1.2. What Does it Mean to Use XML?

What does it mean to use XML in a software solution? This seems like a simple question, so it is a good place to start. Unfortunately, a little poking and prodding reveals a slightly more troubling reality. It turns out that you need to start asking questions about whether you are using Simple API for XML version 1.0 ([SAX 1](#)) or Simple API for XML version 2.0 ([SAX 2](#)), Document Object Model Level 1 ([DOM 1](#)) or Document Object Model Level 2 ([DOM 2](#)), raw strings, Java Document Object Model ([JDOM](#)), or some other tree oriented structure for data, that, when you look at it just right, and maybe squint a little bit (metaphorically, of course), it looks a lot like one of the items listed above.

The problems seem to compound themselves. You might soon start asking questions about what type of schema description you want to use. Do you want to use Document Type Definition ([DTD](#)), XML Schema, BizTalk, XML Data Reduced ([XDR](#)), Schema for Object-Oriented XML ([SOX](#)) or make up your own variation of an XML schema definition? Then once you have picked your schema variation, you might be confronted with some other questions, such as which industry standard mark-up languages might you need to conform to and manipulate, do you need to use XML

Stylesheet Language Transformation ([XSLT](#)) to transform your documents from one form to another, and do you need to use schema adjuncts?

The questions do not seem so simple now, and switching portions of your product to use an XML based solution does not seem like such a clear-cut benefit any more once you are faced with these questions. The difficulty stems, in part, from our understandable fascination with this technology on the one hand, and on the other, ignoring what is perhaps the single most important dictate of software development: "Keep it Simple". Any extra complexity you introduce will potentially cost you time and money, particularly if XML is a new technology for you, in which case the potential for mistakes is a lot higher. In the end, this discussion of XML usage patterns will help to restore some simplicity to a situation that seems to have turned into a long list of questions about how to adopt XML in your products. The particular tools that you adopt are less important if you have a clear understanding of the overall principles.

You need not feel like you must have answers to even some of the questions listed above to use XML well. One important concept to keep in mind as you are working with XML is that there is no "correct" way to use it. XML is valuable precisely because it is a transparent way to encapsulate information, rather than a vendor specific, database specific, language specific, platform specific solution that you might already be using. So feel free to "cheat" - avoid defining "schema" documents if you do not need them, if all you want to do is extract some information from an XML document, perhaps a simple XPath expression with a [DOM 2](#) will do the trick.

### **1.3. Patterns**

Jumping right in, then, here is a quick description of some of the usage patterns.

- Not using XML - when the economies of making information readily accessible are too expensive.
- Configuration pattern - describes a simple way to define structured data while avoiding the need to develop custom editors for your structured data.
- Declarative Language pattern - use XML as a basis for inventing and implementing a simple programming "language".

- Data Transmission pattern - describes using XML to transfer data among various sources.
- Envelope pattern - wrap up data that you need to send that is not XML in an XML container.
- Structured Storage pattern - store your data in XML, thereby making it easier to identify the contents of your data.
- Interface Definition pattern - use a schema description as a remote invocation interface definition.

## 2. When XML Does Not Make Sense

In *Blown To Bits* (Evans, Wurster) [[Blown To Bits](#)], the authors describes how the changing economics of the information economy can completely undermine the validity of a business model. This will tend to happen when a core profit center of a business lies in the value of the information it has. Once information has been turned into XML, it is readily accessible to anyone who thinks to open the file.

In the context of a discussion about XML, consider Intuit's QuickBooks software, and its associated "Tax Table Update Service". Every year, Intuit will send you the latest US tax information, provided by them for a subscription fee. Now imagine for a moment that the information that they deliver arrived in XML syntax. Before too long, either some competitor would provide a similar service for a much smaller fee, simply by generating an equivalent XML file, or federal state and local governments would provide the information for free in hopes of increasing compliance. The result would be that Intuit would undermine the profitability of the QuickBooks software line, as it would no longer be collecting income from subscriptions that virtually every one of their customers now has.

When you use XML, the information inside the file is readily accessible to anyone who tries to access it. To retrieve the data manually, someone need only look. To retrieve it programmatically they need only figure out the correct XPath expression. For this reason, the environment in which the XML is employed should define the value proposition, rather than the contents of the XML data. Avoid using XML when

the information contained in the XML has the value.

### 3. Configuration Pattern

For most software projects, there is some small but crucial set of information that is necessary to get the project up and running. For example, you might need to specify the name of the remote machine name that you need to connect to, the name of the database that you are connecting to, or perhaps even the location of critical files on the hard disk. XML makes it easy to readily encapsulate this information.

For most simple configuration information, there is little obvious immediate gain here. In Windows, you could use an "INI" file, or the system registry, to store similar information. XML makes this an interesting proposition by having approximately the same programming complexity to access the data, yet it is also far more expandable in the long term. It is worth pointing out as well that it can be much more reliable to edit and inspect the data in XML format. As XML document editors such as XML Instance become more commonplace, you can provide an optional schema for your configuration data, and insure that as your customers modify the contents of the document within carefully defined limits. This is not something you will ever be able to accomplish with "INI" files or the Window registry.

To extract data from your configuration file, there are two straightforward, quick to implement solutions. The first is to simply read the entire XML contents into a [DOM 2](#), then parse individual XPath expressions to extract the data that you need when you need it. The second approach would be to use a [SAX 2](#) based parser, and upon encountering various tags, you would instantiate objects and set instance variables of your objects with the data that you might have. I will discuss the latter approach later when talking about the Data Transmission pattern.

If you have existing code to save your configuration information, by all means continue using that code. Once you start changing that code, or find yourself writing new code for configuration information, consider that using XML, as it delivers a language independent, platform independent, highly expandable, tightly constrained solution.

## 4. Declarative Language Pattern

If there were any particular usage pattern that I could claim as my favorite, this would be the one. In some ways, it is a logical extension of the configuration pattern above. Consider starting with very simple case, where a software program needs to know the name of a remote computer with which it is going to establish a connection. Suppose that the server is not responding. The developers may very well find themselves describing a list of fallback servers to try in turn. And then they might specify how long to try each of these servers before giving up. As the final step, they might realize that you need to display some kind of message to the administrator when the last of the fallback servers has failed to connect. Before long, perhaps without realizing it, they have written a carefully constrained, interpreted, declarative programming language. One of the notable characteristics of this solution is that your developers did not have to pay the penalty of figuring out how to parse the data, a huge improvement over the pre-XML days, where it would be necessary to use tools like "lex", "yacc", and "bison" to generate a fairly complicated chunk of code to parse a fairly simple file. In fact, the old style solution has a big enough ramp-up curve to prevent most developers from attempting it, leaving them to solve the solution in custom, one-off ways that would be expensive to maintain over time.

[XSLT](#) is perhaps the best example of a declarative XML based programming language, and is certainly the most widely used. Since it is likely familiar to most of you, I will note some of the attributes of that language that are likely to be useful as you consider implementing your own variations of such languages.

- XSL files either contain, or are contained in larger bodies of tree structured, arbitrarily complex data. When producing HTML output for example, the XSL can be fairly non-intrusive, allowing the XSL file to closely approximate the desired output.
- The language is declarative. Since XML is most definitively *not* a programming language per se, once you get into complex flow-of-control structures, such as while and for loops, switch statements, procedure calls, and possibly even objects, you are stretching the limits of what fits nicely in the XML syntax.

- There is a well-defined context in which the "language" is evaluated. In the case of XSL, there is an input file, an output stream, and an option program state that can be used in the process of performing the transformation. XML based languages are not general purpose.
- It is mostly insensitive to spacing. This may not seem important at first, but it is worth noting that [XSLT](#) has special constructs to guarantee that spaces are not ignored in specific contexts. Outside those contexts, extra white space can and will be ignored. As a programming language, ignoring extra spaces is essential, since software developers will generally require indentation and extra white space to insure clear thinking and code comprehension.

There are three general approaches to implementing an XML-based declarative language. The first approach involves loading the XML file, which I will now call a "script", into a [DOM 1](#), or [DOM 2](#) data structure. Then simply walk the child nodes. At each step evaluate the node to see what the name of the node is, look it up in a hash table to see which function should be called to perform the action, and if the node is one that contains other instructions, then walk the children of the current node in the same way. The second approach extends the first by turning each of the nodes into an instance of a class that performs the instruction, then executing those instructions later, thereby allowing for faster execution, and freeing the [DOM 2](#) structure once the conversion is done. The final approach, by far the most sophisticated, extends the second approach and adds dynamically determined relations among sets of instructions to compute the actual order of execution. This last approach is what [XSLT](#) must do.

An XML-based declarative language can be implemented fairly quickly, can be expanded quickly as needed, and introduce flexibility into your software projects where it does not already exist.

## 5. Data Transmission Pattern

This pattern has been a clear winner in today's e-commerce world. In a nutshell, put your data into XML, and then send it where it needs to go, instead of sending proprietary or binary formats. In general, neither you nor the recipient of your data will want to waste time determining if incorrect data is being sent, and with an

XML-based solution, you can simply examine the contents to find out. With a non-XML solution, there may be some amount of platform dependent, proprietary, 3rd party code that would need to be used to examine the validity of the data; and since it is proprietary, it would be more difficult to discover whether there were bugs in the validation code. Using XML sidesteps all of these issues, and isolates you from the choices that your suppliers and partners might have made, however well intentioned.

When transmitting data via XML, there are some questions to be asked, and some fairly simple answers to each of those questions. To wit:

- Are the data sources and data sinks trusted sources of data? More precisely, can you be guaranteed that the data being sent is in fact valid data? If the data source is trusted, then you will want to set up mechanisms for validating the XML content only when in debugging mode, so that in a high-performance environment you will get maximum throughput. If you cannot trust the data source, always validate.
- What do you use to validate the XML? There are numerous sources for XML validation, supporting various different schema dialects, including [DTD](#), XML Schema (XSD), [SOX](#), and [XDR](#). I strongly encourage you to find the strictest possible validation you can afford, and use that. As of this writing, XML Schema provides the most capabilities for reuse and data constraint, so you should adopt that if you can. Sources for validation tools include various open source projects, and the validation software provided by TIBCO Extensibility.
- What if your business partners cannot support XML Schema? Provide them with converted schemas that they can use for data creation and validation purposes. Tools such as TIBCO Extensibility's XML Authority and XML Console will help you speed through the conversion process.
- Should you use extended validation? Extended validation refers to issues such as having a partner send an order for a part, but they send an invalid part number. In this case, the part number may fall within the constraints that can be defined by your XML Schema based description, however, it simply doesn't exist in your parts database. Here, the answer is not a simple yes-or-no, but if you

can safely catch the mistake downstream of your validation, then you may simply let the request continue, only to be rejected later in your process. If you cannot recover from the problem later, catch it as soon as possible.

- Will you need to increase the bandwidth of your communications to support the use of XML? If this proves to be an issue in your environment, I recommend that you start with adding compression to your communications environment. Also keep in mind that you will not be adding substantial overhead for people using modems, as 56K modems already include compression software.

Having covered the above straightforward issues, you are left with two not-so-simple problems. The first problem is getting your native C++ or Java classes to produce XML that conforms to the agreed upon data exchange formats, and the second problem will be creating C++ and Java objects from XML that you receive.

If you only need to solve this problem for one or two Java or C++ classes, then you can readily use custom code to quickly and directly solve this problem. If you have more than one or two classes, though, a more general solution applies. The general solution looks something like this:

- To go to XML, provide a "Bridge" [[Design Patterns](#)] that converts your internal data structure into something that looks like the tree oriented structure of XML. In Java, this bridge could use reflection [[JDK2.0](#)], in C++ you can use some clever macros.
- If the resulting XML does not look exactly like what you want your results to look like, apply [XSLT](#) or some other transform mechanism to make it conform.

And getting data from XML, the following steps will get you back your data.

- To get from XML, you may need to transform the data to conform more closely to your data structures.
- Having done this, define the reverse Bridge that will go from the XML to your C++ or Java class.
- Now define a template (in XML!) that maps the incoming XML to the structure of

your C++ or Java class. Now you have the flexibility to change just the template as the format of the XML, or the structure of your internal data storage changes.

## 6. Envelope Pattern

A quick extension of the Data Transmission pattern is worth noting. As a compromise between your existing solution and an XML based solution, consider wrapping your proprietary format data in an "XML envelope". In the future, as you are able to migrate some or all of that data to XML format, you can simply expand your "envelope" to include more data that would otherwise be in binary format.

To encode your binary data inside of XML, a simple Base64 encoding will probably do the trick. Consider the following example:

```
<invoice>
  <customerId>35143</customerId>
<invoiceData>9324dfs234sdf8ft53423sflldfgAKDFITh34sdfs</invoiceData>
</invoice>
```

(the above binary data is completely arbitrary, as this is only intended as an example).

Encoded in this format, the binary data is just about 33% larger than the original data.

## 7. Structured Storage Pattern

The Structured Storage pattern is another pattern that follows as a logical extension of the Configuration pattern. If you look at your software from a slightly different perspective, the data that you collect and store as user documents is not much different from the configuration data that you might have to configure your system properly. The "user data" that you collect does tend to have two characteristics that lead me to define this as a separate usage pattern: namely that the "user data" changes much more frequently, and there is a *lot* more of it.

There are various reasons for storing your user data as XML, and several reasons for avoiding it. As it informs the discussion about how best to go about storing your

data as XML, I will briefly note the reasons to avoid using XML for storage purposes:

- As it will be easier for the user to move their data from one software package to another, you will not be able to lock your customers in to your existing solution (however temporarily) unless you provide more benefits than your competitors.
- XML is more verbose than binary formats and will take more space. Unfortunately, compression here is not a good option, as it obscures the data. If you have large quantity of data, this explosion of data may matter. Keep in mind that you might be able to adjust the schema for your data to eliminate some of this overhead.
- Load times will increase. A binary format for data storage is guaranteed to be faster.
- If your current storage mechanism uses a database, you will almost certainly want to continue using a database, as you are likely already leveraging the abilities of your relational or object oriented database in ways that the existing pure XML storage mechanisms do not quite enable.

The benefits of storing your data as XML, however, can be nothing short of astonishing for both the development of your project and for your customers:

- Since it is a text file, all sorts of existing text file differencing utilities can be applied. During development, this will enable automated and manual testing to quickly identify data errors during a save process.
- As your file format changes over time, rather than writing custom C++ or Java code to convert the file, you can write [XSLT](#) files that will perform the conversion.
- Source control tools prefer text-based files; this will allow your customers to more closely follow changes to the contents of documents.
- With the additional work of writing a few [XSLT](#) files, your data can quickly be transformed to and from compatible products, increasing the flexibility of your product for your users.

- You can have the same file format for any hardware platform.

The engineering behind converting your data structures to read and write XML for storage was roughly outlined under the Data Transmission pattern above.

## 8. Interface Definition Pattern

In building an application that works across a network, it is necessary to define the remote method function signatures. The traditional, pre-XML solution to this problem is to use an "Interface Definition Language" (IDL) that declares all of the parameters to a function, and the result of a function. With XML, you can take a new approach, and use a schema to define the function parameters and results.

Using a schema to define the parameters to a remote function makes a great deal of sense. Consider a function "setNameAndAge" that sets a person's name and age. If you follow the Data Transmission pattern, and turn this into XML, the XML might look something like this:

```
<setNameAndAge>
  <personId>456</personId>
  <name>Any Name</name>
  <age>32</age>
</setNameAndAge>
```

You can, of course, define a schema document that corresponds to the above XML document that also tightly constrains the personId, name, and age fields to reasonable values. In a sense, simply by defining the schema, you have defined the parameters to the function "setNameAndAge". In this relatively trivial example, note that if this function returned the users previously defined age, we have not defined how the remote application would return this data. That problem could be solved by something like the following result XML:

```
<setNameAndAgeResult>
  <personId>435</personId>
  <previousAge>31</previousAge>
</setNameAndAgeResult>
```

By defining the schema for both the "setNameAndAge" and the

"setNameAndAgeResult" elements, you have completely described the remote method invocation.

This example shows that the schema for the XML can be treated as an abstract definition of how to invoke remote functions. If nothing else, this is useful because you can take any existing schema editing tool, with a nice, easy to use interface, and define how your remote communications work. It is worth noting that there are various standards, such as Simple Object Access Protocol ([SOAP](#)) that define similar approaches to defining remote function invocations. The point of this pattern is to define the use of schemas to define a remote interface - whatever particular protocol you choose to use, and whether or not you choose to use XML.

## 9. Conclusion

There are many reasons to adopt XML based solutions in your software projects. Hopefully this document has given you some tools to understand various ways that you can quickly leverage XML to improve the quality and utility of your projects. May you have much success in your implementations.

## Bibliography

[Blown To Bits] *Blown To Bits*, Philip Evans, Thomas S. Wurster, Harvard Business School Press, Boston Massachusetts, 2000.

[Design Patterns] *Design Patterns*, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[JDK2.0] *Java 2 Platform Specification*, <http://java.sun.com/j2se/1.3/docs.html>, Sun Microsystems, Inc.

## Glossary

DOM 1

Document Object Model Level 1

DOM 2	Document Object Model Level 2
DTD	Document Type Definition
JDOM	Java Document Object Model
SAX 1	Simple API for XML version 1.0
SAX 2	Simple API for XML version 2.0
SOAP	Simple Object Access Protocol
SOX	Schema for Object-Oriented XML
XDR	XML Data Reduced
XSLT	XML Stylesheet Language Transformation

## Biography

### Eric Johnson

Senior Architect  
TIBCO Extensibility Inc.  
Chapel Hill  
USA  
Email: [eric@tibco.com](mailto:eric@tibco.com)

*Eric Johnson* - Eric Johnson is product development leader at TIBCO for XML technologies' enterprise class solutions. He also founded Visionscape Technologies, Inc., a software development consulting firm specializing in Windows and Macintosh C++ consulting. Before becoming an entrepreneur he worked for Berkeley Systems Inc., as project leader, and as a developer at Slate Corporation, an early developer in pen-based computing.