Google XML Document Format Style Guide

Version 1.0

Introduction

This document provides a set of guidelines for general use when designing new XML document formats (and to some extent XML documents as well; see Section 11). Document formats usually include both formal parts (DTDs, schemas) and parts expressed in normative English prose.

These guidelines apply to new designs, and are not intended to force retroactive changes in existing designs. When participating in the creation of public or private document format designs, the guidelines may be helpful but should not control the group consensus.

This guide is meant for the design of XML that is to be generated and consumed by machines rather than human beings. Its rules are *not applicable* to formats such as XHTML (which should be formatted as much like HTML as possible) or ODF which are meant to express rich text. A document that includes embedded content in XHTML or some other rich-text format, but also contains purely machine-interpretable portions, SHOULD follow this style guide for the machine-interpretable portions. It also does not affect XML document formats that are created by translations from proto buffers or through some other type of format.

Brief rationales have been added to most of the guidelines. They are maintained in the same document in hopes that they won't get out of date, but they are not considered normative.

The terms MUST, MUST NOT, SHOULD, SHOULD NOT, and MAY are used in this document in the sense of RFC 2119.

1. To design or not to design, that is the question

1. Attempt to reuse existing XML formats whenever possible, especially those which allow extensions. Creating an entirely new format should be done only with care and consideration; read Tim Bray's warnings first. Try to get wide review of your format, from outside your organization as well, if possible. [*Rationale:* New document formats have a cost: they must be reviewed, documented, and learned by users.]

2. If you are reusing or extending an existing format, make *sensible* use of the prescribed elements and attributes, especially any that are required. Don't completely repurpose them, but do try to see how they might be used in creative ways if the vanilla semantics aren't suitable. As a last resort when an element or attribute is required by the format but is not appropriate for your use case, use some fixed string as its value. [*Rationale:* Markup reuse is good, markup abuse is bad.]

3. When extending formats, use the implicit style of the existing format, even if it contradicts this guide. [*Rationale:* Consistency.]

2. Schemas

1. Document formats SHOULD be expressed using a schema language. [*Rationale:* Clarity and machine-checkability.]

2. The schema language SHOULD be RELAX NG compact syntax. Embedded Schematron rules MAY be added to the schema for additional fine control. [*Rationale:* RELAX NG is the most flexible schema language, with very few arbitrary restrictions on designs. The compact syntax is quite easy to read and learn, and can be converted one-to-one to and from the XML syntax when necessary. Schematron handles arbitrary cross-element and cross-attribute constraints nicely.]

3. Schemas SHOULD use the "Salami Slice" style (one rule per ele-

ment). Schemas MAY use the “Russian Doll” style (schema resembles document) if they are short and simple. The “Venetian Blind” style (one rule per element type) is unsuited to RELAX NG and SHOULD NOT be used.

4. Regular expressions SHOULD be provided to assist in validating complex values.

5. DTDs and/or W3C XML Schemas MAY be provided for compatibility with existing products, tools, or users. [*Rationale:* We can’t change the world all at once.]

3. Namespaces

1. Element names MUST be in a namespace, except when extending pre-existing document types that do not use namespaces. A default namespace SHOULD be used. [*Rationale:* Namespace-free documents are obsolete; every set of names should be in some namespace. Using a default namespace improves readability.]

2. Attribute names SHOULD NOT be in a namespace unless they are drawn from a foreign document type or are meant to be used in foreign document types. [*Rationale:* Attribute names in a namespace must always have a prefix, which is annoying to type and hard to read.]

3. Namespace names are HTTP URIs. Namespace names SHOULD take the form https://example.com/*whatever*/*year,* where *whatever* is a unique value based on the name of the document type, and *year* is the year the namespace was created. There may be additional URI-path parts before the *year.* [*Rationale:* Existing convention. Providing the year allows for the possible recycling of code names.]

4. Namespaces MUST NOT be changed unless the semantics of particular elements or attributes has changed in drastically incompatible ways. [*Ra-*

tionale: Changing the namespace requires changing all client code.]

5. Namespace prefixes SHOULD be short (but not so short that they are likely to be conflict with another project). Single-letter prefixes MUST NOT be used. Prefixes SHOULD contain only lower-case ASCII letters. [*Rationale:* Ease of typing and absence of encoding compatibility problems.]

4. Names and enumerated values

Note: "Names" refers to the names of elements, attributes, and enumerated values.

1. All names MUST use lowerCamelCase. That is, they start with an initial lower-case letter, then each new word within the name starts with an initial capital letter. [*Rationale:* Adopting a single style provides consistency, which helps when referring to names since the capitalization is known and so does not have to be remembered. It matches Java style, and other languages can be dealt with using automated name conversion.]

2. Names MUST contain only ASCII letters and digits. [*Rationale:* Ease of typing and absence of encoding compatibility problems.]

3. Names SHOULD NOT exceed 25 characters. Longer names SHOULD be avoided by devising concise and informative names. If a name can only remain within this limit by becoming obscure, the limit SHOULD be ignored. [*Rationale:* Longer names are awkward to use and require additional bandwidth.]

4. Published standard abbreviations, if sufficiently well-known, MAY be employed in constructing names. Ad hoc abbreviations MUST NOT be used. Acronyms MUST be treated as words for camel-casing purposes: informationUri, not informationURI. [*Rationale:* An abbreviation that is well known to one community is often incomprehensible to others who

need to use the same document format (and who do understand the full name); treating an acronym as a word makes it easier to see where the word boundaries are.]

5. Elements

1. All elements MUST contain either nothing, character content, or child elements. Mixed content MUST NOT be used. [*Rationale:* Many XML data models don't handle mixed content properly, and its use makes the element order-dependent. As always, textual formats are not covered by this rule.]

2. XML elements that merely wrap repeating child elements SHOULD NOT be used. [*Rationale:* They are not used in Atom and add nothing.]

6. Attributes

1. Document formats MUST NOT depend on the order of attributes in a start-tag. [*Rationale:* Few XML parsers report the order, and it is not part of the XML Infoset.]

2. Elements SHOULD NOT be overloaded with too many attributes (no more than 10 as a rule of thumb). Instead, use child elements to encapsulate closely related attributes. [*Rationale:* This approach maintains the built-in extensibility that XML provides with elements, and is useful for providing forward compatibility as a specification evolves.]

3. Attributes MUST NOT be used to hold values in which line breaks are significant. [*Rationale:* Such line breaks are converted to spaces by conformant XML parsers.]

4. Document formats MUST allow either single or double quotation marks around attribute values. [*Rationale:* XML parsers don't report the difference.]

7. Values

1. Numeric values SHOULD be 32-bit signed integers, 64-bit signed integers, or 64-bit IEEE doubles, all expressed in base 10. These correspond to the XML Schema types xsd:int, xsd:long, and xsd:double respectively. If required in particular cases, xsd:integer (unlimited-precision integer) values MAY also be used. [*Rationale:* There are far too many numeric types in XML Schema: these provide a reasonable subset.]

2. Boolean values SHOULD NOT be used (use enumerations instead). If they must be used, they MUST be expressed as true or false, corresponding to a subset of the XML Schema type xsd:boolean. The alternative xsd:boolean values 1 and 0 MUST NOT be used. [*Rationale:* Boolean arguments are not extensible. The additional flexibility of allowing numeric values is not abstracted away by any parser.]

3. Dates should be represented using RFC 3339 format, a subset of both ISO 8601 format and XML Schema xsd:dateTime format. UTC times SHOULD be used rather than local times. [*Rationale:* There are far too many date formats and time zones, although it is recognized that sometimes local time preserves important information.]

4. Embedded syntax in character content and attribute values SHOULD NOT be used. Syntax in values means XML tools are largely useless. Syntaxes such as dates, URIs, and XPath expressions are exceptions. [*Rationale:* Users should be able to process XML documents using only an XML parser without requiring additional special-purpose parsers, which are easy to get wrong.]

5. Be careful with whitespace in values. XML parsers don't strip whitespace in elements, but do convert newlines to spaces in attributes. However, application frameworks may do more aggressive whitespace stripping. Your document format SHOULD give rules for whitespace stripping.

8. Key-value pairs

1. Simple key-value pairs SHOULD be represented with an empty element whose name represents the key, with the value attribute containing the value. Elements that have a value attribute MAY also have a unit attribute to specify the unit of a measured value. For physical measurements, the SI system SHOULD be used. [*Rationale:* Simplicity and design consistency. Keeping the value in an attribute hides it from the user, since displaying just the value without the key is not useful.]

2. If the number of possible keys is very large or unbounded, key-value pairs MAY be represented by a single generic element with key, value, and optional unit and scheme attributes (which serve to discriminate keys from different domains). In that case, also provide (not necessarily in the same document) a list of keys with human-readable explanations.

9. Binary data

Note: There are no hard and fast rules about whether binary data should be included as part of an XML document or not. If it's too large, it's probably better to link to it.

1. Binary data MUST NOT be included directly as-is in XML documents, but MUST be encoded using Base64 encoding. [*Rationale:* XML does not allow arbitrary binary bytes.]

2. The line breaks required by Base64 MAY be omitted. [*Rationale:* The line breaks are meant to keep plain text lines short, but XML is not really plain text.]

3. An attribute named xsi:type with value xs:base64Binary MAY be attached to this element to signal that the Base64 format is in use. [Rationale: Opaque blobs should have decoding instructions attached.]

10. Processing instructions

1. New processing instructions MUST NOT be created except in order to specify purely local processing conventions, and SHOULD be avoided altogether. Existing standardized processing instructions MAY be used. [*Rationale:* Processing instructions fit awkwardly into XML data models and can always be replaced by elements; they exist primarily to avoid breaking backward compatibility.]

11. Representation of XML document instances

Note: These points are only guidelines, as the format of program-created instances will often be outside the programmer's control (for example, when an XML serialization library is being used). *In no case* should XML parsers rely on these guidelines being followed. Use standard XML parsers, not hand-rolled hacks.

1. The character encoding used SHOULD be UTF-8. Exceptions should require extremely compelling circumstances. [*Rationale:* UTF-8 is universal and in common use.]

2. Namespaces SHOULD be declared in the root element of a document wherever possible. [*Rationale:* Clarity and consistency.]

3. The mapping of namespace URIs to prefixes SHOULD remain constant throughout the document, and SHOULD also be used in documentation of the design. [*Rationale:* Clarity and consistency.]

4. Well-known prefixes such as html: (for XHTML), dc: (for Dublin Core metadata), and xs: (for XML Schema) should be used for standard namespaces. [*Rationale:* Human readability.]

5. Redundant whitespace in a tag SHOULD NOT be used. Use one space before each attribute in a start-tag; if the start tag is too long, the space MAY be replaced by a newline. [*Rationale:* Consistency and conciseness.]

6. Empty elements MAY be expressed as empty tags or a start-tag immediately followed by an end-tag. No distinction should be made between these two formats by any application. [*Rationale:* They are not distinguished by XML parsers.]

7. Documents MAY be pretty-printed using 2-space indentation for child elements. Elements that contain character content SHOULD NOT be wrapped. Long start-tags MAY be broken using newlines (possibly with extra indentation) after any attribute value except the last. [*Rationale:* General compatibility with our style. Wrapping character content affects its value.]

8. Attribute values MAY be surrounded with either quotation marks or apostrophes. Specifications MUST NOT require or forbid the use of either form. &apos; and &quot; may be freely used to escape each type of quote. [*Rationale:* No XML parsers report the distinction.]

9. Comments MUST NOT be used to carry real data. Comments MAY be used to contain TODOs in hand-written XML. Comments SHOULD NOT be used at all in publicly transmitted documents. [*Rationale:* Comments are often discarded by parsers.]

10. If comments are nevertheless used, they SHOULD appear only in the document prolog or in elements that contain child elements. If pretty-printing is required, pretty-print comments like elements, but with line wrapping. Comments SHOULD NOT appear in elements that contain character content. [*Rationale:* Whitespace in and around comments improves readability, but embedding a comment in character content can lead to confusion about what whitespace is or is not in the content.]

11. Comments SHOULD have whitespace following <!– and preceding –>. [*Rationale:* Readability.]

12. CDATA sections MAY be used; they are equivalent to the use of &amp; and &lt;. Specifications MUST NOT require or forbid the use of CDATA sections. [*Rationale:* Few XML parsers report the distinction, and combinations of CDATA and text are often reported as single objects anyway.]

13. Entity references other than the XML standard entity references &amp;, &lt;, &gt;, &quot;, and &apos; MUST NOT be used. Character references MAY be used, but actual characters are preferred, unless the character encoding is not UTF-8. As usual, textual formats are exempt from this rule.

12. Elements vs. Attributes

Note: There are no hard and fast rules for deciding when to use attributes and when to use elements. Here are some of the considerations that designers should take into account; no rationales are given.

12.1. General points:

1. Attributes are more restrictive than elements, and all designs have some elements, so an all-element design is simplest – which is not the same as best.
2. In a tree-style data model, elements are typically represented internally as nodes, which use more memory than the strings used to represent attributes. Sometimes the nodes are of different application-specific classes, which in many languages also takes up memory to represent the classes.
3. When streaming, elements are processed one at a time (possibly even piece by piece, depending on the XML parser you are using), whereas all the attributes of an element and their values are reported at once, which costs memory, particularly if some attribute values are very long.
4. Both element content and attribute values need to be escaped appropriately, so escaping should not be a consideration in the design.

5. In some programming languages and libraries, processing elements is easier; in others, processing attributes is easier. Beware of using ease of processing as a criterion. In particular, XSLT can handle either with equal facility.
6. If a piece of data should usually be shown to the user, consider using an element; if not, consider using an attribute. (This rule is often violated for one reason or another.)
7. If you are extending an existing schema, do things by analogy to how things are done in that schema.
8. Sensible schema languages, meaning RELAX NG and Schematron, treat elements and attributes symmetrically. Older and cruderschema languages such as DTDs and XML Schema, tend to have better support for elements.

12.2 Using elements

1. If something might appear more than once in a data model, use an element rather than introducing attributes with names like foo1, foo2, foo3 ….
2. Use elements to represent a piece of information that can be considered an independent object and when the information is related via a parent/child relationship to another piece of information.
3. Use elements when data incorporates strict typing or relationship rules.
4. If order matters between two pieces of data, use elements for them: attributes are inherently unordered.
5. If a piece of data has, or might have, its own substructure, use it in an element: getting substructure into an attribute is always messy. Similarly, if the data is a constituent part of some larger piece of data, put it in an element.
6. An exception to the previous rule: multiple whitespace-separated tokens can safely be put in an attribute. In principle, the separator can be anything, but schema-language validators are currently only able to handle whitespace, so it's best to stick with that.
7. If a piece of data extends across multiple lines, use an element: XML parsers will change newlines in attribute values into spaces.

8. If a piece of data is very large, use an element so that its content can be streamed.

9. If a piece of data is in a natural language, put it in an element so you can use the xml:lang attribute to label the language being used. Some kinds of natural-language text, like Japanese, often make use annotations that are conventionally represented using child elements; right-to-left languages like Hebrew and Arabic may similarly require child elements to manage bidirectionality properly.

12.3 Using attributes

1. If the data is a code from an enumeration, code list, or controlled vocabulary, put it in an attribute if possible. For example, language tags, currency codes, medical diagnostic codes, etc. are best handled as attributes.
2. If a piece of data is really metadata on some other piece of data (for example, representing a class or role that the main data serves, or specifying a method of processing it), put it in an attribute if possible.
3. In particular, if a piece of data is an ID for some other piece of data, or a reference to such an ID, put the identifying piece in an attribute. When it's an ID, use the name xml:id for the attribute.
4. Hypertext references are conventionally put in href attributes.
5. If a piece of data is applicable to an element and any descendant elements unless it is overridden in some of them, it is conventional to put it in an attribute. Well-known examples are xml:lang, xml:space, xml:base, and namespace declarations.
6. If terseness is really the *most* important thing, use attributes, but consider gzip compression instead – it works very well on documents with highly repetitive structures.

13. Parting words

Use common sense and *BE CONSISTENT*. Design for extensibility. You *are* gonna need it. [*Rationale:* Long and painful experience.]

When designing XML formats, take a few minutes to look at other formats and determine their style. The point of having style guidelines is so that people can concentrate on what you are saying, rather than on how you are saying it.

Break *ANY OR ALL* of these rules (yes, even the ones that say MUST) rather than create a crude, arbitrary, disgusting mess of a design if that's what following them slavishly would give you. In particular, random mixtures of attributes and child elements are hard to follow and hard to use, though it often makes good sense to use both when the data clearly fall into two different groups such as simple/complex or metadata/data.

Newbies always ask:

”Elements or attributes?

Which will serve me best?”

Those who know roar like lions;

Wise hackers smile like tigers.

–a tanka, or extended haiku

[TODO: if a registry of schemas is set up, add a link to it]