DTD Design Guidelines

10 April 2001

This version:
DTD_Design_Guidelines-1_1
Previous version:
DTD_Design_Guidelines-v01.00
Editors:
Paul Kiel (paul@xmlhelpline.com)
Kim Bartkus (kim@hr-xml.org)
Authors:
Paul Kiel (paul@xmlhelpline.com)
Chuck Allen (chucka@hr-xml.org)
Contributors:
Technical Steering Committee (TSC) (tsc@lists.hr-xml.org)

Abstract

Document Type Definitions (DTDs) are a major aspect of XML work within the Consortium. They are a document syntax that allows for the automated processing and error checking of an XML document via a parser. To aid in the development of DTDs, the Technical Steering Committee has fostered the development of a standard set of instructions on how to design them

Status of this Document

This document is available for general use by members of the HR-XML consortium.

Neither HR-XML nor its members shall be responsible for any loss resulting from any use of this document or the specifications herein

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Table of Contents

1 Scope and Purpose

1.1 Definitions

DTD:

A Document Type Definition or “DTD” defines the elements, attributes, entities, and notations that are valid within a conforming XML document.

XML is a language for creating markup languages, such as those being developed by the HR-XML Consortium. Within XML Version 1.0, a DTD is the tool for creating and describing a markup language. Once a DTD is created, documents can be compared to it programmatically to confirm whether they conform to the constraints specified within the DTD. This is called *validation*. If an XML document follows the rules specified in the DTD, then the document is said to be *valid*. XML documents that don't follow the rules in the DTD are *invalid.*

Schema:

A schema serves the same purpose as a DTD, but a schema provides a more powerful and flexible means of defining markup languages.

A schema is written using XML syntax. This is not true of DTDs, which use a syntax specified for Standard Generalized Markup Language (SGML), a precursor technology of which XML Version 1.0 is a subset. Schemas support data typing and constructs that allow one element to derive properties from another element type. Several groups and companies, including Microsoft and Commerce One, developed their own schema specifications in advance of the World Wide Web Consortium’s (W3C) issuance of its XML Schema Definition Language (“XSDL” or “XSD”) recommendation. The term “schema” is sometimes used loosely in a way that encompasses DTDs as well as the various schema formats. However, as used in this document, “schema” refers to the W3C’s XML Schema Definition Language (“XSDL” or “XSD”) recommendation. For further information, see http://www.w3c.org.

1.2 Scope

This document provides design guidelines and conventions for DTDs developed by HR-XML workgroups. For XML Schema guidance and conventions see the separate document titled “XML Schemas – Best Practices”.

1.3 Purpose

HR-XML’s DTD design guidelines and conventions are intended to ensure that DTDs developed by HR-XML workgroups are:

- Easy to use and intelligible to the various end-user constituencies – software developers, HR professionals, database administrators, etc.
- Consistent in design across HR-XML workgroups
- Easy to maintain

- Developed in a manner that provides a predictable development path towards XML Schema and namespaces

1.4 Relationship between examples and work group activities

While examples in this document occasionally draw upon working drafts of schemas being developed within the HR-XML Consortium, these schemas were often edited for space or simplicity and should not be construed as representing valid instances or schema fragments resulting from actual work group activities.

2 Naming Conventions

2.1 Use CamelCase for Element and Attribute Names

Element names that contain multiple words, should have each word capitalized. This is known as "camel case" notation. Other special delimiting characters, such as '_', '-', and '.' should be avoided.

2.2 Upper and Lower CamelCase

Follow these conventions for the use of upper and lower CamelCase:

Elements:

- All element names are Upper CamelCase – e.g. PersonName
- Abbreviations are Upper CamelCase – e.g. IntlCode, TelNumber
- Acronyms are Upper Case, with the remainder of the element in Upper CamelCase – e.g. UL, TTDNumber, EEOStatement

Attributes:

- All attribute names are Lower CamelCase – e.g. familyNamePrefix
- Abbreviations are Lower CamelCase – e.g. typeSrc, idRef, secType
- Acronyms are Lower Case, with the remainder of the attribute as Lower CamelCase – e.g. poBox, typeFAQResults

2.3 DTD File Naming Conventions

DTD filenames also should follow the CamelCase convention, followed by: a dash, and major and minor version number ("-1_0"). For instance, a DTD written to support a benefits enrollment process might be named:

BenefitsEnrollment-1_0.dtd.

See also section on "Versioning."

2.4 Use Meaningful Element and Attribute Names

Choose element names that reflect the business language you are modeling. They should be meaningful and easy to understand. Abbreviations within names should be avoided, unless the element name would otherwise be excessively long. When using abbreviations, the most commonly accepted and clearest version should be used. Keep in mind international audiences, so avoid Americanized abbreviations.

<Shp2> Don't do this </Shp2>

<Dptmnt> Don't do this </Dptmnt>

<Dept> Write it this way </Dept>

2.5 Don't Use Private Encodings

When practical, use distinct elements to separate parts of data. If you have a data item that has internal structure, separating the parts with tags enables an XML parser to do the work of distinguishing components for you, which saves you the cost of writing a custom parser. Additionally, you will have made the structure publicly readable and thereby available to standard tools such as XSL style sheets and query languages. If something has more than one part, it is likely that it will gain still more parts in the future, and using tags up-front means easier extension later.

3 Elements and Attributes

HR-XML will strive toward clear and consistent distinctions in the use of elements and attributes.

3.1 Using Elements Versus Attributes

3.1.1 Use Elements to Represent Data Content

The choice of whether to represent semantic components as elements or attributes is made on a case-by-case basis. However, the general rule is that elements should be used to represent objects and data content. Attributes should be used to represent element metadata.
One rule to apply is to see if the data item could exist on its own as an inherent piece of information. If the information is a distinct item, but dependent upon another, make it a child element. If it merely refines a larger item, then use an attribute.

3.1.2 Elements/Attributes: Other Considerations

In deciding whether a semantic component should be represented as an element or attribute, the following key differences should be considered:

XML 1.0 rules to bear in mind

- Elements can contain only other elements or data, while attributes can contain only data. Thus, if a semantic component has a complex structure or could potentially contain other elements, the component should be modeled as an element.

- Elements can be declared so XML 1.0 conforming parsers preserve the order of elements. You cannot similarly control the order of attributes. Some parsers may preserve the order of attributes, but this is not a requirement under XML 1.0.
- Attribute type declarations allow for the setting of default and fixed values for attributes. XML 1.0 does not include the same type of default or fixed values for elements.
- Processors that use the SAX interface return attributes as a set with the start of an element. In contrast, child elements are returned one-by-one.
- An element can have only one attribute of type "ID." (And of course attributes of type "ID" must have unique values within the entire document.)
- Elements and attributes must begin with a letter. (i.e. <!ELEMENT 7DaysAWeek (#PCDATA)> is invalid. Also, <!ATTLIST SomeElement 12MonthsAYear (January | February | …) #IMPLIED> is equally invalid.)

Design Rules of Thumb

- If the potential values of data are known, then an enumerated list of attribute values will provide enforcement and error checking. (<!ATTLIST Days weekend (Friday | Saturday | Sunday) #IMPLIED>)
- If an item may need to occur more than once in a certain context, use an element. Attributes are not repeatable within the same element. (i.e. <Element anAttribute="first time is fine" anAttribute="second reference to the same attribute name is invalid">)

3.2 Elements

3.2.1 Strategies for Common Patterns : Use Names that indicate classes

It is important to note that there are other standards organizations that are advocating similar types of semantics in their naming conventions. In particular, **ebXML** has been working on "Representation Types" that are similar to the class words discussed here. This may present future opportunities to coordinate with other standards organizations.

A data element name is usually composed of between one and three component words. Limiting names to this length can keep the naming conventions consistent as well as minimizes the performance drain from excessively long element names.

Each component is classified as a **prime word**, **class word**, or **modifier word**.

First, a data name *may* contain a **prime word** designator. Prime words identify business information objects to which the data element (or attribute) belongs. For example, <Insurance>. Generic elements such as <Address> may not need an industry-specific prime word, especially if it will be used in many contexts.

Second, elements and attributes *may* be classified into common groupings using **class words**. Classes can better organize and provide meaningful names for elements.

Good Examples:

<!ATTLIST SomeElement expenseRatio CDATA #IMPLIED>

Here the class word "Ratio" is used to indicate the class of information.

<!ELEMENT DropDeadDate (#PCDATA)>

If the data element matched, this could also be DropDeadDay or DropDeadDays.

<!ELEMENT ErrorDescription (#PCDATA)>

"Description" indicates the meaning of this element, as opposed to an error code or an error handling instruction.

<!ATTLIST SomeElement jobSeekerId Id #IMPLIED>

<!ATTLIST SomeElement insuranceIndicator (covered | uncovered) #IMPLIED>

Bad Examples:

<!ELEMENT TheCode (#PCDATA)>

The class word "Code" does not make this element name more comprehensible.

<!ELEMENT ListOfAllVacanciesInTheDatabase (#PCDATA)>

If all elements were given such long names, the impact on performance with large documents would begin to show.

<!ELEMENT Item (#PCDATA)>

This element is unclear.

Common class names include: Cost, Date, Days, Day, Description, Id, IdRef, Indicator, Name, Number, Rate, and Ratio.

Lastly, in addition to a prime and class word designator, a data name *may* contain one or more **modifier words** or adjectives that further describe the data element.

An example that illustrates the use of all three types of component names is <JobPostingDate>. "Job" is a business object specific to this domain, "Posting" further describes the term "Job", and "Date" is a class name indicating the type of data element.

3.2.2 Special Element Content Models

Guidance on the use of the following content models:

3.2.2.1 Mixed

Mixed content models are elements that contain data (PCDATA) as well as sub elements. These content models should be avoided. Processing this data by applying stylesheets, for example, produces inconsistent results across parser implementations.

Avoid:

<Book>The HR-XML Story

<Author>Chuck Allen</Author>

</Book>

Good:

<Book>

<Title> The HR-XML Story </Title>

<Author>Chuck Allen</Author>

</Book>

3.2.2.2 EMPTY

While unusual, there are instances when EMPTY content models are appropriate. For example, when you want an unknown number of occurrences of an enumerated list.

<!ELEMENT ListOfDaysIncluded (SelectedDays)*>

<!ELEMENT SelectedDays EMPTY>

<!ATTLIST SelectedDays oneDay (Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday) #IMPLIED>

<ListOfDaysIncluded>

<SelectedDays oneDay=”Monday”/>

<SelectedDays oneDay=”Wednesday”/>

<SelectedDays oneDay=”Friday”/>

</ListOfDaysIncluded>

In this instance, the content of <SelectedDays> element is not needed, but the attribute repeated in each element provides multiple occurrences of enumerated content.

3.2.2.3 ANY

Content models of type ANY are not endorsed for any type of use and are highly discouraged.

Using content type ANY can result in the dilution of the core standard. This design is sometimes used to include vendor-specific extensions of the core DTD. This would have the effect of creating validation, processing, or style sheet problems, which would weaken the core DTD as an industry standard.

Also, the use of ANY is sometimes used for including future engineering into a design. If one needs to extend the DTD for future engineering, say for a process that is not currently designed but is known to exist, then refer to the section on “Extensibility.” By extending the DTD without the use of “ANY”, future implementation of a new process can be engineered into the specification, thus minimizing the impact on existing functionality.

3.2.2.4 A Note on Recursion

A recursive data content model is where an element contains itself as a descendent. For example:

```
<!ELEMENT RootElement  (ChildElement , AnotherChild)>

<!ELEMENT ChildElement (RootElement)>

        <!--here ChildElement calls its parent as a child -->

<!ELEMENT AnotherChild (AnotherChild)>

        <!--here AnotherChild calls itself directly -->
```

The use of a recursive content model should be done with caution. This structure could lead to system problems with creating or processing a document such as with applying style sheets.

An alternate way to model this type of relationship is to use ID and IDREF. For example,

```
<!ELEMENT RootElement  (ChildElement , AnotherChild)>

<!ATTLIST RootElement  id  ID #REQUIRED>

<!ELEMENT ChildElement (#PCDATA)>

<!ATTLIST ChildElement  rootIdRef  IDREF #REQUIRED>

        <!--here ChildElement points to its parent  -->

<!ELEMENT AnotherChild (#PCDATA)>

<!ATTLIST AnotherChild

                id  ID  #REQUIRED

                anotherIdRef  IDREF #REQUIRED>

    <!--here AnotherChild points to a sibling -->
```

Occasionally the correct modeling of business objects requires recursion. Here is one example, where an unstructured form of resume enables nested sections. This, in essence, provides structure for data that may not conform to structures modeled elsewhere in the DTD.

Example instance:

```
<FreeFormResume>
    <ResumeSection>
        <SectionTitle>Skills</SectionTitle>
        <SecBody>
            <!-- here, ResumeSection is a descendent of itself describing
                    the “Management” part of the Skills section -->
            <ResumeSection>
                <SectionTitle>Management</SectionTitle>
                <SecBody>
                    <P>Supervised 8 FTE.</P>
                </SecBody>
            </ResumeSection>
            <!-- here, ResumeSection is again a descendent of itself describing
                    the “Information Technology” part of the Skills section -->
            <ResumeSection>
                <SectionTitle>Information Technology</SectionTitle>
                <SecBody>
                    <P>Programmed in C++ and Java.</P>
                </SecBody>
            </ResumeSection>
        </SecBody>
    </ResumeSection>
</FreeFormResume>
```

4 Building Modular DTDs

The objectives of this modularization proposal are two fold:

1. To provide a solution that will support the reuse of data models during the initial release of HR-XML DTDs.
2. To provide a smooth transition to Schema and namespace support once the Schema recommendation has been finalized and appropriate tools are available in the open market.

4.1 Start with Element and Attributes

The use of modular DTDs is analogous to good object oriented design. While there may be some front loading of effort in order to accommodate potential reuse, it ultimately reduces work further down in the workflow. In addition, it provides for easier maintenance and modification.

The first way to build modular DTDs is to design elements and attributes properly. Discussed in section “Elements and Attributes”, the decisions made about using these data structures affect the reusability of the design.

4.2 The use of entities (global content models)

Another way to create modular DTDs is with parameter entities. Two methods for using these are in reusable global content models and in creating modular components. First, global content models can be parameterized for easy referencing and updating.

<!ENTITY % att.content “yes | no”>

<!ELEMENT AnElement (#PCDATA)>

<!ATTLIST AnElement someBoolean (%att.content;) #IMPLIED>

If this content model were used often, then the use of entities would make updating easier. If the “att.content” value were to change, it could be changed globally, perhaps to:

<!ENTITY % att.content “yes | maybe | no”>

It is also important to note that internal entities are not applicable in XML Schemas. So these will need to be expanded prior to converting to Schemas. Many good DTD design tools will do this for you.

4.3 The use of entities (reusable components)

To more fully enable cross-process objects, the use of external parameter entities can be a powerful tool. This allows the creation of modules that are aggregated into usable business processes. Just as in object oriented design, if one breaks a process into logical components, we can serialize them into reusable components with the help of external entities. The best examples of what to modularize would be the most universally used objects in an industry. Names, addresses, contact information, along with their associated unique identifiers are commonly made into modules for aggregating into various processes.

Not only will modular use of entities promote object-oriented design, they can also aid in the future migration to Schemas. These modules can be used in creating complex data typed elements. Schemas use complex data types as one method of making reusable objects.

<!ENTITY % contact.info.mod SYSTEM “hr-contact.mod”>

% contact.info.mod;

<!ELEMENT Client (ContactInfo)*>

<!ELEMENT Advertiser (ContactInfo)*>

<!ELEMENT JobSeeker (ContactInfo)*>

In this case, the <ContactInfo> element, and its subordinates, is defined in an external module referred to as “contact.info.mod” and with the literal filename of “hr-contact.mod.” If the “hr-contact.mod” module is updated, say with new international name support, then one need only to update the module and all elements that need it can use it.

4.4 The down side of modularizing

One drawback to using entities is that it can be hard to work with and visualize during the building process.

Perhaps the biggest potential drawback of modularizing is that it requires more open content models. If an object is going to be used in many contexts, it must not be too restrictive in any one of them. Elements and attributes will tend to be optional instead of required, as certain aspects may not be applicable in all processes. In this fashion, the potential for insufficient or inaccurate data can be possible. The result is that some of the logic and completeness checking is left to the implementation rather than the parser. Overall, the use of modularized components provides a powerful tool for cross-process objects, and usually outweighs the drawbacks.

4.5 Final Production

While the power of modularization is clear, its implementation can be problematic, with so many modules to manage. So the final serialization of a process should consolidate the modules into a single deliverable, thus leveraging the use of modularization while not creating implementation problems. In short, build modules, but deploy a single DTD.

5 Referencing Outside Taxonomies/Content

"Don't re-invent the wheel." Using and incorporating the existing taxonomies and codes developed by recognized standards bodies and by other authoritative third parties is almost always preferable to the development of substitute taxonomies or codes by HR-XML Consortium workgroups. One source for researching existing standards is the Diffuse Project website (http://www.diffuse.org).

Cost/benefit considerations generally prevent the Consortium from engaging in the development of large or complex taxonomies or "codes." The Consortium may not necessarily have the resources or expertise to support such projects.

The development of relatively small taxonomies or code sets may be necessary to effectively execute a given business process. Workgroups may want to define a taxonomy or set of codes where the universe of possible codes or descriptors is relatively small. For instance, the ability to concisely reference frequency of pay is probably necessary to execute the business processes targeted by the Consortium's workgroups. Defining a taxonomy to express pay frequency is a project the Consortium might want to undertake if a standard was not available from a standards body or other authoritative third-party. The universe of possible types of pay frequency is probably relatively small --- e.g, a taxonomy that is likely to consist of less than 100 or so descriptors or types.

Workgroups must research available standards thoroughly and they should weigh the costs and benefits of any proposal to define a taxonomy before beginning such a project. If there are questions or concerns about the merits of such a proposal, workgroups should seek advice from the Technical Steering Committee or Cross-Process Objects Workgroup as may be appropriate.

6 Extensibility

Despite the name “Extensible Markup Language”, DTDs are not easy to extend unless DTD developers leave openings for such extensions. Several sections in this document discuss ways to extend DTDs.

See “Considerations for Migrating to Namespaces”, as namespaces themselves are not in the XML 1.0 specification.

See “Building Modular DTDs” for guidance on extendable design techniques (including the use of parameter entities).

DTDs can also be extended outside the scope of the core DTD using entities. This technique in essence uses a “wrapper” to enclose the core DTD and extends it without violating any core components. This enables 100% compliance with the core standard while allowing extensibility. Here is an example.

Extended Job Seeker Example:

```
<?xml version = "1.0"?>
<!DOCTYPE MyInternalDoc [
<!ELEMENT MyInternalDoc (JobPositionSeeker , SomeSpecialInfo)>
<!-- Include the official SEP JobPositionSeeker dtd as an external entity so that I can validate the public portion of the document -->
<!ENTITY % JobPositionSeeker-v1.0.dtd SYSTEM "JobPositionSeeker-v1.0.dtd">
%JobPositionSeeker-v1.0.dtd;
<!-- This is the internal extension. My confidential opinions about this seeker -->
<!ELEMENT SomeSpecialInfo (MyOpinionOfThisPerson , WhatIThinkTheyAreWorth)>
<!ELEMENT MyOpinionOfThisPerson (#PCDATA)>
<!ELEMENT WhatIThinkTheyAreWorth (#PCDATA)>
]>
<MyInternalDoc>
    <JobPositionSeeker status = "active">

        <!--additional valid core DTD content -->

    </JobPositionSeeker>
    <SomeSpecialInfo>
        <MyOpinionOfThisPerson>What a bozo</MyOpinionOfThisPerson>
        <WhatIThinkTheyAreWorth>He should pay us to work here!</WhatIThinkTheyAreWorth>
    </SomeSpecialInfo>
</MyInternalDoc>
```

Extended Job Seeker Example (With Namespaces):

In the following example, namespaces are used to resolve the conflict between the element <Contact> defined in the DTD and the element <Contact> used as an extension outside the core DTD.

```
<?xml version = "1.0"?>
```

```
<!DOCTYPE MyInternalDoc [
<!ELEMENT MyInternalDoc (JobPositionSeeker , foo:SomeSpecialInfo)>
<!-- Include the official SEP JobPositionSeeker dtd as an external entity so that I can validate the
 public portion of the document -->
<!ENTITY % JobPositionSeeker-v1.0.dtd SYSTEM "JobPositionSeeker-v1.0.dtd">
%JobPositionSeeker-v1.0.dtd;
<!-- This is the internal extension.  My confidential opinions about this seeker -->
<!ELEMENT foo:SomeSpecialInfo (foo:MyOpinionOfThisPerson , foo:WhatIThinkTheyAreWorth ,
foo:Contact)>
<!ATTLIST foo:SomeSpecialInfo  xmlns:foo CDATA  #FIXED 'http://www.foo.org/' >
<!ELEMENT foo:MyOpinionOfThisPerson (#PCDATA)>
<!ATTLIST foo:MyOpinionOfThisPerson xmlns:foo CDATA  #FIXED 'http://www.foo.org/' >
<!ELEMENT foo:WhatIThinkTheyAreWorth (#PCDATA)>
<!ATTLIST foo:WhatIThinkTheyAreWorth xmlns:foo CDATA  #FIXED 'http://www.foo.org/' >
<!ELEMENT foo:Contact (#PCDATA)>
<!ATTLIST foo:Contact xmlns:foo CDATA  #FIXED 'http://www.foo.org/' >
]>
<MyInternalDoc>
    <JobPositionSeeker status = "active">
        <Contact>
            <PersonName>James Brown</PersonName>
            <PositionTitle>Managing Partner</PositionTitle>
            <VoiceNumber><TelNumber>800-555-1212</TelNumber></VoiceNumber>
        </Contact>

        <!--additional valid core DTD content -->

    </JobPositionSeeker>
    <foo:SomeSpecialInfo xmlns:foo = "http://www.foo.org/">
        <foo:MyOpinionOfThisPerson>What a bozo</foo:MyOpinionOfThisPerson>
        <foo:WhatIThinkTheyAreWorth>He should pay us to work
here!</foo:WhatIThinkTheyAreWorth>
        <foo:Contact>toughreviewer@meancompany.com</foo:Contact>
    </foo:SomeSpecialInfo>
</MyInternalDoc>
```

7 Considerations for Migrating to Namespaces

The Namespace issue is still open. However, the current thought is to use a single namespace for HR-XML Schemas. Of course namespaces are not supported in the XML 1.0 specification, and thus not supported in DTDs. However, there is a naming convention that can be used to indicate such information. The use of “xmlns” as an attribute with a FIXED value in the root element (or any element) is often used in this regard. Since some parsers or XSL processors may not permit the use of an attribute name beginning with “xml”, it is recommended that the attribute “hrxmlns” be used.

```
<!ATTLIST RootElement    hrxmlns  CDATA #FIXED "[tbd]">
```

The current working default namespace for HR-XML Schemas is:

http://www.hr-xml.org/schemas/…*[tbd]*

8 Versioning

Versioning is often an overlooked aspect of DTD development, but one which has important and potentially disastrous consequences down the road. It is important to keep track of which DTD is being used, so that validation is correctly applied. Documents validated against an outdated DTD could result in loss of functionality.

Each DTD will use a REQUIRED attribute named “version” in the root element. This attribute will be an enumerated type with a single value, which is the version number. For example,

<!ELEMENT RootElement (#PCDATA)>

<!ATTLIST RootElement version (01.00) #REQUIRED>

The only valid instance for this architecture is:

<RootElement version=”01.00”>

The reason for this approach is to require that implementers explicitly include a version number into the document instance. Had a default or FIXED value been included, this could be passively overlooked by implementation, or could create conflicts during upgrading (i.e. when version 01.10 deploys). The above structure ensures that implementers state the version number explicitly, and minimizes the potential for passive oversight or upgrading conflicts.

See also “DTD File Naming Conventions.”

9 Language

The language of an xml instance should be indicated with an attribute “contentLanguage” on the root element. For example:

DTD Declaration:

<!ELEMENT RootElement (#PCDATA)>

<!ATTLIST RootElement

contentLanguage CDATA "EN"

version (01.00) #REQUIRED

Document Instance:

<RootElement contentLanguage="EN" version="01.00"/>

In this example, the contentLanguage attribute has a default value of “EN”, which is the standard code for English. This attribute should follow the ISO 639 Standard Language Codes. A quick reference to these codes can be found at:
http://www.netstrider.com/tutorials/HTMLRef/standards/iso639.html.

(Tangentially, if country codes are used, ISO 3166 Standard Country Name Codes are the standard, as referenced at: http://www.netstrider.com/tutorials/HTMLRef/standards/iso3166.html.)

10 Linking

While XML 1.0 does not support standards such as Xlink, there are ways to incorporate linking into a DTD. The simplest approach is to use the following:

```
<!ELEMENT Link (#PCDATA)>
<!ATTLIST Link URL CDATA #IMPLIED
                idRef CDATA #IMPLIED>
```

In this case, the content of the <Link> element is used for a description or label for the link. An external URL or internal idRef is provided in the attribute as indicated. This is the preferred method for adding link information.

When simple linking is insufficient, extending the DTD to include Xlink can be accomplished using attributes. An example of this design comes from the Scalable Vector Graphic (SVG) DTD adapted here:

```
<!ATTLIST SomeElement
        xmlns:xlink CDATA #FIXED 'http://www.w3.org/1999/xlink'
        xlink:href CDATA #REQUIRED
        xlink:type (simple|extended|locator|arc) 'simple'
        xlink:role CDATA #IMPLIED
        xlink:arcrole CDATA #IMPLIED
        xlink:title CDATA #IMPLIED
        xlink:show (embed) 'embed'
        xlink:actuate (onRequest|onLoad) 'onLoad'>
```

(The full SVG specification can be found at: http://www.w3.org/TR/SVG/.)

11 Basic Syntax Quick Reference

This section provides a brief reference to correct syntax of DTDs. It is not intended as an exhaustive listing, but as a quick reference tool for DTD designers.

Element and Attribute declarations are typically declared as:

```
<!ELEMENT SomeElement  (SomeSubElement )*>
<!ATTLIST SomeElement  someAttribute CDATA  #IMPLIED >
<!ELEMENT SomeSubElement  (#PCDATA )>
```

Occurrences are indicated with the following:

*	0 or more occurrences
+	1 or more occurrences
?	0 or 1 occurrence

| logical “or”

, follows (content after “,” must occur after it’s precedent in the document instance)

Data types:

The XML 1.0 specification permits only limited data typing. The allowed data types are:

Elements: PCDATA (parsed character data), (some grouping of sub elements), EMPTY, or ANY

Attributes: CDATA (character data), enumeration (a fixed list of valid values), NMTOKEN (name tokens), NMTOKENS, id, idref, idrefs, ENTITY, ENTITIES, and NOTATION.

Doctype references are as follows:

<!DOCTYPE RootElement SYSTEM “ *path* ”>

<!DOCTYPE RootElement [*declared internal subset*]>

Internal Parameter Entities are as follows:
<!ENTITY % att.content “yes | no”>

<!ELEMENT AnElement (#PCDATA)>

<!ATTLIST AnElement someBoolean (%att.content;) #IMPLIED>

External Parameter Entities are as follows:
<!ENTITY % commonmod SYSTEM “hr-comm.mod”>

%commonmod;

Processing Instructions:
<?xml-stylesheet type=”text/xsl” href=”someStylesheet.xsl”?>

Notations:

<!NOTATION jpeg PUBLIC "ISO/IEC 10918:1993//NOTATION Digital Compression and Coding of Continuous-tone Still Images (JPEG)//EN" >

Comments:

<!-- comments are done this way,

in fact this is a multi-line comment -->

12 References

Annotated XML 1.0

An excellent *annotated* version of the XML 1.0 specification. (See http://www.xml.com/axml/axml.html.)

DTD Tutorial

Miloslav Nic has a DTD syntax tutorial. (See http://www.zvon.org/xxl/DTDTutorial/General/book.html.) ZVON has several xml-related tutorials and quick references at http://www.zvon.org/.)

ebXML

See ebXML Core Components Dictionary Entry Naming Conventions at www.ebxml.org.

XML 1.0

The official specification. (See http://www.w3.org/TR/REC-xml.)

XML Syntax Quick Reference

Excellent XML/DTD syntax quick references and tutorials exit at the *Mulberry Technologies* website. (See http://www.mulberrytech.com/quickref/index.html.)

13 Description of Changes

10 April 2001: Updated minor change to **3.2.2.2 EMPTY** [content models]. Some general edits.

Updated **3.2.2.4 A Note on Recursion** [of content models] to illustrate when this may be appropriate.

Added **1.4 Relationship between examples and work group activities**, to accurately show that examples in this document are not to be perceived as valid results of work group activities.

27 March 2001: A minor update of "DTD Design Guidelines-v01.00." Example syntax in Section 9 "Language" was corrected.

22 January 2001: DTD Design Guidelines-v01.00 released.