# Constructing a Document Type Definition (DTD) for XML

NEW MEXICO TECH
Computer Center www.nmt.edu/tcc

John W. Shipman

2013-08-24 12:16

### Abstract

Describes the Document Type Definition notation for describing the schema of an SGML or XML document type.

This publication is available in Web form[1] and also as a PDF document[2]. Please forward any comments to **tcc-doc@nmt.edu**.

## Table of Contents

## 1. What is a DTD?

The purpose of a Document Type Definition or DTD is to define the structure of a document encoded in XML (eXtended Markup Language).

For introductory material about XML, see the XML help page[3].

[1] http://www.nmt.edu/tcc/help/pubs/dtd/
[2] http://www.nmt.edu/tcc/help/pubs/dtd/dtd.pdf
[3] http://www.nmt.edu/tcc/help/xml/

It is possible to build and use files containing XML tags without ever defining what tags are legal. However, if you want to insure that files conform to a known structure, writing a DTD is the preferred method.

Two definitions:

- A *well-formed* file is one that obeys the general XML rules for tags: tags must be properly nested, opening and closing tags must be balanced, and empty tags must end with `'/>'`.
- A *valid* file is not only well-formed, but it must also conform to a publicly available DTD that specifies which tags it uses, what attributes those tags can contain, and which tags can occur inside which other tags, among other properties.

The advantage of a valid file is that its contents are more predictable for applications that want to process or present that file. The DTD insures that only certain tags can be used in certain places.

## 1.1. Definitions

We need to review some terminology before proceeding:

- A proper *XML name* must start with a letter or underbar (`_`), with the rest letters, underbars, digits, or hyphen (`-`).
- A *tag* is one of the XML constructs used to mark up documents. All tags start with a less-than symbol (`<`) and end with a greater-than symbol (`>`).
- An *element* is a section of an XML document that acts as a unit. It may be either *empty element*, or it may have *content*.
- An *empty element* consists of a single tag of the form

  ```
      <gi.../>
  ```

  Where *`gi`* is the tag type (or “generic identifier”), and the tag may include attributes. Note the slash before the closing “`>`”; this signifies an empty tag.
- An *opening tag* begins a section of an XML document that ends with the corresponding *closing tag*. An opening tag has this form:

  ```
      <gi...>
  ```

  where *`gi`* is the tag type (or “generic identifier”), and the tag may include attributes. A closing tag has the form:

  ```
      </gi>
  ```

- The *content* is everything between the opening tag and its corresponding closing tag. The content may be other elements or just plain text.

# 2. Where does a DTD live?

A Document Type Definition lives in a file whose name ends with `.dtd`. Inside this file are items that define tags (elements), attributes, and other beasties to be named later.

There are two ways to connect a DTD with the content of an XML file:

- *External DTD*: You can put the DTD in a separate file from the XML file, and refer to it using a `<!DOCTYPE ...>` line at the beginning of the XML file. The advantage of this method is that many XML files can all refer to the same DTD.
- *Internal DTD*: You can put the DTD inside the `<!DOCTYPE ...>` declaration at the front of the XML file. The advantage of an internal DTD is that the file can be validated by itself, without reference to any external files.

## 2.1. Linking an XML file to an external DTD

If your XML file is supposed to conform to an external DTD, place a declaration of this form at the beginning of the XML file:

```
<?xml version="1.0"?>
<!DOCTYPE root-name SYSTEM "dtd-name.dtd">
```

where `root-name` is the name of the root (highest-level) element of the document, and `dtd-name.dtd` is the name of the file containing the DTD.

## 2.2. Including the DTD inside your XML file

To include the DTD in an XML file, the file should start like this:

```
<?xml version='1.0'?>
<!DOCTYPE root-name [
    dtd-declarations ...
]>
```

Here's an example of a complete XML file with an internal DTD that defines two element types: a root element `<park>` and a second-level element `<trail>`. We'll explain the pieces of the DTD later on.

```
<?xml version="1.0"?>
<!DOCTYPE park [
  <!ELEMENT park (trail*)>
  <!ATTLIST park
      name    CDATA   #IMPLIED>
  <!ELEMENT trail (#PCDATA)>
  <!ATTLIST trail
      dist    CDATA   #REQUIRED
      climb   CDATA   #REQUIRED>
]>
<park name='Lincoln Natural Forest'>
  <trail dist='3400' climb='medium'>Canyon Trail</trail>
  <trail climb='easy' dist='1200'>Pickle Madden Trail</trail>
</park>
```

# 3. Types of DTD declarations

The DTD can contain several different types of declarations:

- *Element* declarations let you specify what kinds of tags can be used, and what (if anything) can appear inside the contents of the element.

- *Attribute* declarations define what attributes you can use inside a given element.
- *Entity* declarations define chunks of fixed text that can be included elsewhere.
- *Notation* declarations define file types (like JPG and WAV files) so you can refer to non-XML files like image and sound files.

We'll discuss element declarations first, since they make up most or all of a typical DTD.

# 4. Element declarations

In a DTD, an element declaration defines one of the kinds of elements you can use, that is, one of the tag types.

All element declarations have this general form:

```
<!ELEMENT gi (content)>
```

where `gi` is the element name (also called the “generic identifier”) and the `content` describes what content (if any) can go inside the element. The generic identifier must follow the rules for XML names, above.

The `content` part describes the syntax of the element's content using a general notation with a number of different parts. The next few sections describe the items that can go into the content.

## 4.1. Declaring empty elements

If you don't want a certain element to have any content, that is, you want that element always to be represented by an empty tag (see above), use this element declaration:

```
<!ELEMENT gi (EMPTY)>
```

For example, if your DTD contains this declaration:

```
<!ELEMENT pagebreak (EMPTY)>
```

then an XML document conforming to this DTD could contain a tag that looks like:

```
<pagebreak/>
```

## 4.2. Elements with text content only

If you want an element to contain only text, declare it like this:

```
<!ELEMENT gi (#PCDATA)>
```

The `#PCDATA` part means “parsed character data.”

For example, suppose your DTD has this declaration:

```
<!ELEMENT remark (#PCDATA)>
```

Then a conforming XML document could have an element in it that looks like this:

```
<remark>There was something fishy about the butler.
  I think he was working for scale.</remark>
```

## 4.3. Elements with mixed content

In general, an element can have any mixture of text and other elements as children. You can specify exactly which elements can be children. If you like, you can even specify that the children must occur in a given order. You can also specify that the child elements are optional.

So, in the general form of the declaration `<!ELEMENT gi (content)>`, the *`content`* is an expression syntax—that is, it consists of operators and operands arranged in arbitrarily complex ways.

Let's start with some simple cases to show you the features of a content declaration, but keep in mind that these features can be used in combination.

The simplest case is when an element *`a`* has a single child element *`b`*:

```
    <!ELEMENT a (b)>
```

The above declaration in a DTD means that an element `<a>...</a>` must contain exactly one `<b>` element.

To specify that a child element can occur one or more times, append a plus sign (`+`) after the child element name. For example, to say that a `<squid>` element may contain one or more `<tentacle>` elements:

```
    <!ELEMENT squid (tentacle+)>
```

You can also specify that a child element can occur any number of times, or not at all. Append an asterisk (`*`), meaning “zero or more of the previous,” after the child element name:

```
    <!ELEMENT lizard (leg*)> <!-- some <lizard>s have no <leg>s -->
```

The question-mark suffix (`?`) means the child element is optional: it can occur zero or one time in the content of the element you're declaring. For example, suppose an `<oven>` element can either be empty or contain a `<pie>` element:

```
    <!ELEMENT oven (pie?)>
```

If you want a certain sequence of children, name the child elements in a comma-separated list. For example, suppose a `<memo>` element must contain exactly one `<from>` element, then one `<to>` element, one `<subject>`, and one `<message>` element:

```
    <!ELEMENT memo (from,to,subject,message)>
```

But you can use the `+`, `*`, and `?` operators in this declaration. For example, suppose that you want to require that a `<memo>` must have `<from>` and `<to>` elements, but the `<subject>` element is optional, and it can have zero or more `<message>` elements. You'd then declare it like this:

```
    <!ELEMENT memo (from,to,subject?,message*)>
```

Sometimes you need to specify that there is a choice of children. The “or” operator (`|`) can be used to separate the choices. For example, suppose that a `<trophy>` element can have either a child named `<bowling>` or a child named `<tennis>`. Here's how you'd declare it:

```
    <!ELEMENT trophy (bowling|tennis)>
```

You can also apply the usual suffix operators to groups of elements. For example, suppose you have an element `<timerecord>` that starts with a required `<purpose>` element, followed by zero or more pairs of `<start-time>` and `<end-time>` records:

```
    <!ELEMENT timerecord (purpose,(start-time,end-time)*)>
```

Here's another more general example:

```
<!ELEMENT stock ((pig|chicken|cow)*)>
```

The above example says a `<stock>` element can contain any number of the three child elements, in any order.

Moreover, you can allow regular, untagged text to be mixed in with your specified child tags by placing `#PCDATA` at the start of a list of choices. For example, suppose a `<speech>` element can contain any mixture of regular text, and text tagged with the elements `<loud>` and `<soft>`:

```
<!ELEMENT speech ((#PCDATA|loud|soft)*)>
<!ELEMENT loud (#PCDATA)>
<!ELEMENT soft (#PCDATA)>
```

So, the `content` part of the element declaration can be arbitrarily complex. There are some ways #PCDATA cannot be used, and there are other uncommon features you may need; refer to the XML standard or a good book on the subject.

## 5. Attribute declarations

If an element is to have attributes, the names and possible values of those attributes must be declared in the DTD. Here is the general form:

```
<!ATTLIST ename {aname atype default} ...>
```

where *`ename`* is the name of the element for which you're defining attributes, *`aname`* is the name of one of that element's possible attributes, *`atype`* describes what values it can have, and *`default`* describes whether it has a default value. The last three items can be repeated inside an `<!ATTLIST...>` declaration, one group per attribute.

The *`atype`* part describing the attribute's type can have three kinds of values:

- The keyword `CDATA` means that the attribute can have any character string as a value.

  For example, suppose you want every `<play>` element to have a `title` attribute that can contain any text, and that attribute is required. Here is the complete attribute declaration:

  ```
  <!ATTLIST play title CDATA #REQUIRED>
  ```

- There are several *tokenized* attribute types, which are required to have a certain structure. See tokenized attributes below.

- You can provide a specific set of legal values for the attribute; see enumerated attributes below.

The last part of the declaration, *`default`*, specifies whether the attribute can be omitted, and what value it will have if omitted. This must be one of the following:

**`#REQUIRED`**
The attribute must always be supplied.

**`#IMPLIED`**
The attribute can be omitted, and the DTD does not provide a default value. Anyone reading this file may assume a default value, but that is not the DTD's problem.

**`"`*`value`*`"`**
The attribute can be omitted, and the default value is the quoted string that you provide.

**`#FIXED "`*`value`*`"`**

The attribute must be given and must have the given `"`*`value`*`"`.

## 5.1. Tokenized attributes

You can restrict an attribute to have only values with a certain structure. Here are the possible values of the *`atype`* part of the attribute declaration for such attributes:

**`ID`**

An `ID` attribute must be a unique identifier for that node. This allows other nodes to refer to it. The attribute value must also be a valid XML name (see above).

**`IDREF`**

An `IDREF` attribute is a reference to an `ID` attribute in a different node.

For example, suppose that in your DTD, there is a `<sailor>` element with an `ID`-type `nickname` attribute, and another element `<duty>` with an `IDREF`-type attribute called `sailor-nick`. Then if you have an element like this:

```
    <sailor nickname='Bluto'>...</sailor>
```

then this tag would refer to that element:

```
    <duty sailor-nick='Bluto'>...</duty>
```

**`IDREFS`**

The value of an `IDREFS` attribute must contain one or more `ID` references separated by spaces. Example:

```
    <roster sailor-nicks='Bluto Popeye Olive_Oyl'/>
```

**`ENTITY`**

Use this attribute type to refer to external, non-parsed entities. See the section on notations, below.

**`ENTITIES`**

Like `ENTITY`, but the attribute can be a list of one or more entity names separated by spaces.

**`NMTOKEN`**

The attribute value must be a name token, conforming to the rules for XML names (see above).

**`NMTOKENS`**

Like `NMTOKEN`, but the attribute value can contain one or more name tokens separated by spaces.

## 5.2. Enumerated attributes

You can specify that attributes must have one of a set of one or more values. Here is the general form of the *`atype`* part of the `<!ATTLIST...>` declaration:

```
    (value1|value2|...)
```

For example, suppose you want your `<vehicle>` element to have a `kind` attribute that must have a value of either `"car"`, `"truck"`, or `"boat"`:

```
    <!ATTLIST vehicle
        kind (car|truck|boat) #REQUIRED>
```

You can also supply a default value in quotes. For example:

```
    <!ATTLIST vehicle
        kind (car|truck|boat) "car">
```

# 6. Declaring and using entities

In a DTD, entities come in four flavors:

- A *general entity* is a chunk of text with a name attached, so you can use the entity as a sort of shorthand to get the related text substituted in its place.

  For example, suppose you are working on a new product called Project Giant-Slayer, but you know that the marketing department will change the name when it's released to the market. You could define the current product name as an entity named `&product;`, and use it everywhere in your product literature. Then, when the marketing department decides on the final name, you can change the declaration of the entity and the new name will magically appear in place of the old one in all your web pages and brochures.
- A *character entity* is one of the many standardized special characters that you can use when you need a character unavailable in your local character set.
- A *parameter entity* is like a general entity, but it can be used as shorthand for parts of a content declaration in an element declaration.
- A *binary* or *non-parsed* entity represents an external file that is not in XML format.

## 6.1. General entities

General entities have names of the form `&`*`name`*`;`, where the name follows the usual rules for XML names (above).

To declare a general entity, use a declaration of this general form in your DTD:

```
    <!ENTITY ename "text">
```

where *`ename`* is the name of the entity you are defining (without the initial `&` and final `;`), and *`text`* is the text you want substituted for that entity.

For example, to define an entity named `&cr;` with your copyright string, you might use a declaration like this:

```
    <!ENTITY cr "Copyright (C) 1763 Cotton Mather LLP">
```

## 6.2. Character entities

To use special characters in your document, you can use the form `&#`*`n`*`;` where *`n`* is the decimal number of the character you want. A table of these entities is online at http://www.w3.org/TR/html401/sgml/entities.html.

## 6.3. Parameter entities

The purpose of a parameter entity is to serve as a shorthand for some or all of the content part of an element declaration.

The general form is:

```
<!ENTITY % ename "text">
```

For example, suppose you have a lot of tags whose content model is `"#PCDATA|bold|ital)*"`. You could define an entity like this:

```
<!ENTITY bitext "(#PCDATA|bold|ital)*">
```

Then, to define an element `<excuse>` with that content:

```
<!ELEMENT excuse %bitext;>
```

### 6.4. Binary (non-parsed) entities

This last type of entity represents a file, like an image or sound file, that is not XML. To declare such an entity:

```
<!ENTITY ename SYSTEM "url" NDATA nname>
```

where *`ename`* is the name of the entity you are defining, *`url`* is the URL where the file can be found, and *`nname`* is the name of the notation that the file uses. See the section on notations below for an example.

## 7. Notation declarations

The purpose of a notation declaration is to define the format of some external non-XML file, such as a sound or image file, so you can refer to such files in your document.

The general form of a notation declaration can be either of these:

```
<!NOTATION nname PUBLIC std>
<!NOTATION nname SYSTEM url>
```

where *`nname`* is the name you are giving to the notation; *`std`* is the published name of a public notation, and *`url`* is a reference to a program that can render a file in the given notation.

There are four steps to connecting an attribute to a notation:

1. Declare the notation. Example:

   ```
   <!NOTATION jpeg PUBLIC "JPG 1.0">
   ```

2. Declare the entity. For example:

   ```
   <!ENTITY bogie-pic SYSTEM
       "http://stars.com/bogart.jpg" NDATA jpeg>
   ```

3. Declare the attribute as type `ENTITY`. For example:

   ```
   <!ATTLIST star-bio pin-shot ENTITY #REQUIRED>
   ```

4. Use the attribute:

   ```
   <star-bio pin-shot="bogie-pic">...</star-bio>
   ```