# DEFINICIÓN DE ESQUEMAS MEDIANTE DTD

EXTRACTO DEL LIBRO:

Lenguajes de marcas y sistemas de gestión de información

Editorial Garceta

## 4.7. Validación de documentos XML con DTD

Un DTD (Document Type Definition - Definición de Tipo de Documento) es una descripción de la estructura de un documento XML. Se especifica qué elementos tienen que aparecer, en qué orden, cuáles son optativos, cuáles obligatorios, qué atributos tienen los elementos, etc.

Es un mecanismo de validación de documentos que existía antes de la aparición de XML. Se usaba para validar documentos SGML (Standard Generalizad Markup Language - Lenguaje de Marcas eStándar Generalizado), que es el precursor de todos los lenguajes de marcas actuales. Cuando apareció XML, se integró en su especificación como modelo de validación gramatical.

En el momento de la publicación de este libro, la técnica de validación con DTDs ha quedado algo obsoleta, si bien se usó de manera intensiva en los años 1990 y principios de los 2000.

Actualmente, se usan más otras técnicas como los esquemas XML, también conocidos por las siglas XSD (XML Schema Document - Documento de Esquema XML). Sin embargo, muchos documentos han quedado validados únicamente mediante DTDs.

Ejemplo:

DTD estricto y DTD transicional para HTML 4.0l

- http://www.w3.org/TR/html4/sgml/dtd.html
- http://www.w3.org/TR/html4/sgml/loosedtd.html

## Generación automática de DTDS

Cuando los documentos XML contienen muchos datos, los DTD que los validan pueden ser muy extensos y su escritura tediosa. Para facilitar este proceso, existen herramientas de generación automática de DTDs a partir de un documento XML, proceso conocido como inferencia. Se detallará su uso más adelante.

Estas herramientas realizan deducciones "gruesas" de las reglas gramaticales, que el desarrollador debe después afinar para que se ajusten a los requisitos semánticos.

· Generadores on-ine:

http://www.hitsw.com/xml _ utilites/default.html

· Generadores instalables:
  - o Trang (http://www.thaiopensource.com/download): es un programa desarrollado en Java, ejecutable desde la línea de comandos, que genera un DTD a partir de un XML dado.

· Editores XML que generan un DTD a partir de un documento XML. Prácticamente todos los citados anteriormente:
  - o XMLSpy de Altova
  - o <oXygen/>, de Syncro SoR
  - o XMLPad Pro Edition, de WMHelp

## 4.7.1. Estructura de un DTD. Elementos

Al igual que ya sucedía con las hojas de estilo en cascada (CSS), el DTD puede aparecer integrado en el propio documento XML, en un archivo independiente (habitualmente con extensión . dtd) e incluso puede tener una parte interna y otra externa. La opción del documento externo permitirá reutilizar el mismo DTD para distintos documentos XML, facilitando las posibles modificaciones de una manera centralizada.

Siempre que se quiera declarar un DTD, se hará al comienzo del documento XML. Las reglas que lo constituyen son las que podrán aparecer a continuación de la declaración (dentro del propio documento XML) o en un archivo independiente.

Declaración del DTD:

<!DOCTYPE>

Es la instrucción donde se indica qué DTD validará el XML. Aparece al comienzo del documento XML. El primer dato que aparece es el nombre del elemento raíz del documento XML.

En función del tipo de DTD la sintaxis varia. Las características que definen el tipo son:

- Ubicación: dónde se localizan las reglas del DTD.
  - Interno: las reglas aparecen en el propio documento XML.
  - Externo: las reglas aparecen en un archivo independiente.
  - Mixto: mezcla de los anteriores, las reglas aparecen en ambos lugares. Las reglas internas tienen prioridad sobre las externas.

- Carácter: si es un DTD para uso privado o público.
  c) Para uso privado: se identifica por la palabra SYSTEM.
  o Para uso público: se identifica por la palabra PUBLIC. Debe ir acompañado del FPI (Formal Public Identifier - Identificador Público Formal), una etiqueta que identifica al DTD de manera "universal".

Las distintas combinaciones son:

| Sintaxis | Tipo de DTD |
|---|---|
| < !DOCTYPE elemento_ raíz [reglasl > | Interno (luego privado) |
| < !DOCTYPE elemento raíz SYSTEM url> | Externo y privado |
| < !DOCTYPE elemento _ raíz SYSTEM URL [reglasl > | Mixto y privado |
| < !DOCTYPE elemento_ raíz PUBLIC FPI url> | Externo y público |
| <!DOCTYPE elemento_ raíz PUBLIC FPI URL [reglasl > | Mixto y público |

Tabla 4.1: Diferentes sintaxis para doctype

La combinación interno y público no tiene sentido, ya que el hecho de ser público fuerza que el DTD esté en un documento independiente.

Atributos:

- Nombre del elemento raíz del documento XML: aparece justo a continuación de la palabra DOCTYPE.

- Declaración de privacidad/publicidad: aparece a continuación del nombre del elemento raíz e indica si el DTD es de uso público (PUBLIC) o de uso interno de la organización que lo desarrolla (SYSTEM).

- Identificador: sólo existe cuando el DTD es PUBLIC e indica el FPl por el que se conoce el DTD.

Estructura del FPI:

Está compuesto de 4 campos separados por el carácter //.

· Primer campo: noma formal o no formal

- Si el DTD no ha sido aprobado por una norma formal, como por ejemplo un DTD escrito por uno mismo, se escribe un signo menos (-).
- Si ha sido aprobado por un organismo no oficial. se escribe un signo más (+)-
- Si ha sido aprobado por un organismo oficial, se escribe directamente una referencia al estándar.

· Segundo campo: nombre del organismo responsable del estándar.

' Tercer campo: tipo del documento que se describe, suele incluir un número de versión.

· Cuarto campo: idioma del DTD.

Ejemplo:

Se va a declarar un DTD respecto a un documento XML cuyo elemento raíz se llama html y que es de carácter público.

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
" http: / /www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" >

O Al aparecer un signo menos (-) significa que el DTD no ha sido aprobado por una noma formal.

O El nombre del organismo responsable del DTD es el W3C.

O El tipo de documento es XHTML Transicional, en su versión 1.0.

O El idioma del DTD es el inglés (EN).

- url: sólo existe cuando el DTD (en parte o en su totalidad) se encuentra declarado en un archivo externo, del que da su ubicación. Como ya se ha comentado, puede darse el caso que un DTD esté definido en parte en un archivo externo y en parte en el documento XML.

Ejemplo:

Se declara el tipo de un documento XML cuyo elemento raíz se llama <planetas >. Se trata de un DTD de uso privado (SYSTEM) y la ubicación de las reglas del DTD es externa, en un archivo llamado planetas . dtd que se encuentra en el mismo directorio que el archivo XML.

<!DOCTYPE planetas SYSTEM "planetas.dtd">

Ejemplo:

Un DTD mixto y privado tendrá sus reglas repartidas entre la cabecera del documento XML y un archivo externo.

El documento XML, charlas.xml, será:

```
<?xml version="l.0" encoding="iso-8859-1" standalone="no"?>
<IDOCTYPE charlas SYSTEM "charlas.dtd" [
   <!ELEMENT charlas (charla)+>
   <IELEMENT charla (nombre, lugar, despedida)>
   <!ENTITY despedidalnglesa "Thank you, good byel">
   <!ENTITY despedidaFrancesa "Merci, au revoir!">
l>
<charlas>
   P P 7
</charlas>
```

El archivo de reglas externo, charlas.dtd, será:

```
< !ELEMENT nombre (#PCDATA) >
< !ELEMENT lugar (#PCDATA) >
< !ELEMENT despedida (#PCDATA) >
```

Nótese que el archivo externo no incluye la declaración del tipo de documento (DOCTYPE), puesto que sólo aparece en la cabecera del documento XML.

Conviene resaltar también que el atributo standalone de la instrucción de procesamiento < ?xml . . . ? > tiene el valor no, lo que significa que para el correcto procesado del documento necesitamos del uso de otros documentos externos (en este caso, de charlas.dtd).

**Componentes del DTD:**

Hay cuatro tipos posibles de componentes que se pueden declarar en un DTD:

- Elemento
- Atributo
- Entidad
- Notación

**<!ELEMENT>**

Es una declaración de tipo de elemento. Indica la existencia de un elemento en el documento XML.

Sintaxis general: < !ELEMENT nombre elemento modelo _ contenido>

El nombre del elemento tiene que ser el mismo que el correspondiente del documento XML, sin los signos de menor (<) y mayor (>) propios de las etiquetas.

El modelo de contenido puede indicar:

- Una regla, en cuyo caso será:
  ' any (cualquier cosa): se puede utilizar al construir el DTD para dejar la descripción de un elemento como válida en cualquier caso, eliminando cualquier comprobación sintáctica. Es un comodín que no debe aparecer en el DTD definitivo.

· empty (elemento vacío): describe un elemento sin descendientes.

Ejemplo:

La descripción del elemento <br / > de HTML será:

< !ELEMENT br EMPTY>

- Datos (caracteres), sean textuales, numéricos o cualquier otro formato que no contenga marcas (etiquetas). Se describe como #PCDATA (Parsed Character Data - Datos de Caracteres Procesados) y debe aparecer entre paréntesis.

Ejemplo:

Una regla de un DTD puede ser:

< !ELEMENT titulo (#PCDATA) >

Que se corresponde con un elemento llamado <titulo> con contenido textual.

<titulo>Lenguajes de marcas</titulo>

- Elementos descendientes. Su descripción debe aparecer entre paréntesis y se basa en las siguientes reglas:

Cardinalidad de los elementos:

Para indicar el número de veces que puede aparecer un elemento o una secuencia de elementos existen ciertos símbolos:

| Símbolo | Significado |
|---|---|
| ? | El elemento (o secuencia de elementos) puede aparecer 0 o 1 vez |
| * | El elemento (o secuencia de elementos) puede aparecer de 0 a N veces |
| + | El elemento (o secuencia de elementos) puede aparecer de 1 a N veces |

Secuencias de elementos:

En las secuencias de elementos se utilizan símbolos para indicar el orden en que un elemento debe aparecer, o bien si aparece como alternativa a otro:

| Símbolo | Significado |
|---|---|
| A, B | El elemento B aparecerá a continuación del elemento A |
| A I B | Aparecerá el elemento A o el B, pero no ambos |

Se pueden combinar el uso de los símbolos de cardinalidad de los elementos con los de la secuencias de elementos.

Ejemplo:

En un correo electrónico, se podría describir el elemento raíz <email> como una secuencia de elementos <para>, <cc> (optativo), <cco> (optativo), <asunto> y <cuerpo>. Las reglas que representan esto serán:

```
<!ELEMENT email (para, cc?, cco?. asunto, cuerpo)>
<!ELEMENT para (#PCDATA)>
<!ELEMENT CC (#PCDATA)>
```

Ejemplo:

Un contrato tiene una lista de cláusulas. Cada una de las cláusulas está compuesta de varios epígrafes y sus desarrollos asociados, y concluyen por un único epílogo:

```
< IELEMENT clausulas (clausula) +>
< !ELEMENT clausula ( (epígrafe, desarrollo) +, epilogo) >
```

- Contenido mixto, mezcla de texto más elementos descendientes.

  Ejemplo:

  Se quiere describir un elemento <párrafo> que simule el párrafo de un editor de textos, de forma que pueda contener texto sin formato, texto en <negrita> (rodeado de esta etiqueta) o texto en <cursiva> (rodeado de esta etiqueta). Estas dos últimas etiquetas podrán a su vez contener, bien texto sin formato, bien la otra etiqueta. Por tanto, <párrafo>, <negrita> y <cursiva> son elementos de contenido mixto. Las reglas que describen esto serán:

```
< !ELEMENT parrafo (negrita I cursiva I #PCDATA) * >
< !ELEMENT negrita (cursiva I #PCDATA) * >
< !ELEMENT cursiva (negrita I # PCDATA) * >
```

  * NOTA: No se puede usar el cuantificador + con un contenido mixto, por eso se ha usado el *

  Un documento XML válido con respecto a las reglas del anterior DTD:

```
<parrafo>
    Aquí un <cursiva>tema <negrita>importante</negrita>< /cursiva>
< /parrafo>
```

## Actividad 4.4:

Se quiere que el elemento <grupoSanguineo> tenga como único elemento descendiente a uno solo de los cuatro siguientes A, B, AB ó O. Indica cuál de las siguientes es una declaración correcta del citado elemento.

a. < !ELEMENT grupoSanguineo (A ? B ? AB ? O) >

b. < !ELEMENT grupoSanguineo (A , B , AB , O) >

c. < !ELEMENT grupoSanguineo (A I B I AB I O) >

d. < !ELEMENT grupoSanguineo (A + B + AB + O) >

Actividad 4.5:

¿Cuál de las siguientes afirmaciones es correcta respecto a la declaración del elemento?

a. < !ELEMENT contenido (alfa I beta*) >

Tanto el elemento alfa como el elemento beta pueden aparecer 0 o más veces como descendientes del elemento contenido.

b. < !ELEMENT contenido (alfa , beta) >

Tanto el elemento alfa como el elemento beta pueden aparecer una vez como descendientes del elemento contenido, sea cual sea el orden.

c . < !ELEMENT contenido (alfa I beta) >

El elemento alfa y el elemento beta deben aparecer una vez cada uno como descendientes del elemento contenido.

d.<!ELEMENT contenido (alfa , beta*) >

El elemento alfa debe aparecer una vez y a continuación el elemento beta debe aparecer 0 o más veces, ambos como descendientes del elemento contenido.

* NOTA: Los documenios HTML no necesitan estar bien formados para poderse visualizar en los navegadores. Por eso, en los DTD que validan estos documentos se usa una determinada nomenclatura para indicar que un elemento pueda ser opcionalmente abierto o cerrado. Así, si aparecen dos guiones después del nombre del elemento tanto la etiqueta inicial como la final son obligatorias. Un guion seguido por la letra "O" indica que puede omitirse la etiqueta final. Un par de letras "O" indican que tanto la etiqueta inicial como la final pueden omitirse.

### <!ATTLIST>

Es una declaración de tipo de atributo. Indica la existencia de atributos de un elemento en el documento XML. En general se utiliza un solo attlist para declarar todos los atributos de un elemento (aunque se podría usar un attlist para cada atributo).

Sintaxis general:<!ATTLIST nombre_elemento
nombre_atributo tipo_atributo caracter
nombre_atributo tipo_atributo caracter
>

El nombre del atributo tiene que ser un nombre XML válido.

El carácter del atributo puede ser:

- un valor textual entre comillas, que representa un valor por defecto para el atributo.
- #IMPLIED, el atributo es de carácter opcional y no se le asigna ningún valor por defecto.

- #REQUIRED, el atributo es de carácter obligatorio, pero no se le asigna un valor por defecto.

- #FIXED, el atributo es de carácter obligatorio y se le asigna un valor por defecto que además es el único valor que puede tener el atributo.

Los posibles tipos de atributo son:

- cdata: caracteres que no contienen etiquetas

- entity: el nombre de una entidad (que debe declararse en el DTD)

- entities: una lista de nombres de entidades (que deben declararse en el DTD), separadas por espacios

- Enumerado: una lista de valores de entre los cuales, el atributo debe tomar uno

  Ejemplo:

  Se quiere definir una regla que valide la existencia de un elemento <semaforo>, de contenido vacío, con un único atributo color cuyos posibles valores sean rojo, naranja y verde. El valor por defecto será verde.

```
< !ELEMENT semaforo EMPTY>
< !ATTLIST semaforo
                color (rojo    |   naranja  |  amarillo)    "verde">
```

- id: un identificador único. Se usa para identificar elementos, es decir, caracterizarlos de manera única. Por ello, dos elementos no pueden tener el mismo valor en atributos de tipo ID. Además, un elemento puede tener a lo sumo un atributo de tipo ID. El valor asignado a un atributo de este tipo debe ser un nombre XML válido.

- idref: representa el valor de un atributo ID de otro elemento, es decir, para que sea válido, debe existir otro elemento en el documento XML que tenga un atributo de tipo ID y cuyo valor sea el mismo que el del atributo de tipo IDREF del primer elemento.

  Ejemplo:

  Se quiere representar un elemento <empleado> que tenga dos atributos, idEmpleado e idEmpleadoJefe. El primero será de tipo id y carácter obligatorio, y el segundo de tipo idref y carácter optativo.

```
< !ELEMENT semaforo (nombre, apellido) >
, < !ATTLIST empleado
                idEmpleado ID #REQUIRED
                idEmpleadoJefe IDREF #IMPLIED>
```

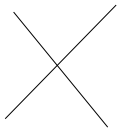
  Aquí tenemos un fragmento de XML válido con respecto a las reglas recién definidas. Cada elemento <empleado> tiene un atributo idEmpleado, con un valor válido (nombre XML válido) y el segundo elemento <empleado> tiene también un atributo idEmplejefe, cuyo valor ha de ser el mismo que el del atributo de tipo ID de otro

elemento existente en el documento. En este caso, este atributo idEmplejefe del segundo <empleado> vale igual que el atributo idEmpleado del primer <empleado>;

```
<empleados>
   <empleado idEmpleado="e _ 111">. . ,</empleado>
f
   <empleado idEmpleado="e_ 222" idEmpleadojefe="e _ 111">
      ...
</empleado>
 </empleados>
```

Un fragmento de XML no válido con respecto a las reglas recién definidas seria aquél en el cual el atributo idEmpleadoJefe del primer <empleado> tenga un valor que no exista para ningún atributo de tipo ID de otro elemento del documento:

```
<empleados>
   <empleado idEmpleado= "e _ 111" idEmpleadoJefe="e _ 333 " >
   ...
   </empleado>
   <empleado idEmpleado= "e _ 222 " idEmpleadoJefe="e _ 111" >
   ...
</empleado>
 </empleados>
```



- idrefs: representa múltiples IDS de otros elementos, separados por espacios.
- nmtoken: cualquier nombre sin espacios en blanco en su interior. Los espacios en blanco anteriores o posteriores se ignorarán.

Ejemplo:

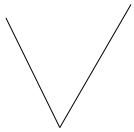
Se quiere declarar un atributo de tipo nmtoken de carácter obligatorio.

En el siguiente DTD, la declaración del elemento <rio> y su atributo país es:

```
< !ELEMENT río (nombre) >
< !ATTLIST río país nmtoken #REQUIRED>
```

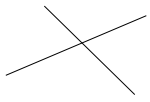
En el siguiente fragmento XML, el valor del atributo país es válido con respecto a la regla anteriormente definida:

```
<rio pais="EEUU">
   <nombre>Misisipi</nombre>
</rio>
```



En el siguiente documento XML, el valor del atributo país no es válido con respecto a la regla anteriormente definida por contener espacios en su interior:

```
<rio pais="Estados Unidos">
   <nombre>Misisipi</nombre>
</rio>
```



- nmtokens: una lista de nombres, sin espacios en blanco en su interior (los espacios en blanco anteriores o posteriores se ignorarán), separados por espacios.

- notation: un nombre de notación, que debe estar declarada en el DTD.

<!ENTITY>

Este es un elemento avanzado en el diseño de DTDS. Es una declaración de tipo de entidad. Hay diferentes tipos de entidades y, en función del tipo, la sintaxis varia:

- Referencia a entidades generales (internas o externas). Constante
- Referencia a entidades parámetro (internas o externas). Fragmentos de código (funciones?)
- Entidades no procesadas (unparsed). Tipos binarios

· Referencia a entidades generales, se utilizarán dentro del documento XML.

Sintaxis general: < !ENTITY nombre _ entidad definición _ entidad>

Ejemplo:

En primer lugar, se declara una entidad en el DTD

< !ENTITY rsa "República Sudafricana">

A continuación, se usa en el XML anteponiendo al nombre de la entidad el carácter ampersand (&) y a continuación un carácter punto y coma (;). El programa analizador del documento realizará la sustitución.

```
<país>
  <nombre>&rsa; </nombre>
  ...
<país>
```

Ejemplo:

En el DTD de HTML se declaran diversas entidades para referenciar caracteres con acentos diversos, como é para referenciar el carácter é.

· Referencia entidades generales externas, ubicadas en otros archivos:

Sintaxis general: < !ENTITY nombre _ entidad tipo _ uso url _ archivo>

Siendo el tipo de uso privado (SYSTEM) o público (PUBLIC).

Ejemplo:

Se dispone de un archivo de texto, autores.txt, que contiene el siguiente texto plano "Juan Manuel y José Ramón".

Se crea un documento XML que hace referencia a ese archivo de texto, en forma de entidad externa. Al visualizar el documento, la referencia a la entidad general externa se sustituirá por el texto contenido en el archivo.

```
, <?xml version="1.0"?>
<!DOCTYPE escritores [
    <!ELEMENT escritores (#PCDATA)>
    <!ENTITY autores SYSTEM "autores.txt">
]>
<escritores>&autores;</escritores>
```

- Referencia a entidades parámetro, que se usarán en el propio DTD y funcionan cuando la definición de las reglas del DTD se realiza en un archivo externo.

Sintaxis general: < !ENTITY % nombre _ entidad definición _ entidad>

Ejemplo:

Se declara una entidad parámetro dimensiones y se referencia dentro del propio DTD.

```
< !ENTITY % dimensiones "alto CDATA #IMPLIED ancho CDATA #IMPLIED
profundo CDATA #IMPLIED" >
< !ELEMENT objeto (nombre) >
< !ATTLIST objeto
                codigo ID #REQUIRED
                %dimensiones ; >    %nombre_entidad;
```

Este código es equivalente a haber escrito:

```
< !ELEMENT objeto (nombre) >
<!attlist objeto
                nombre CDATA #REQUIRED
                alto CDATA #IMPLIED
                ancho CDATA #IMPLIED
                profundo CDATA #IMPLIED>
```

- Referencia entidades parámetro externas, ubicadas en otros archivos:

Sintaxis general: < !ENTITY % nombre _ entidad tipo _ uso fpi url _ archivo>

Siendo el tipo de uso privado (SYSTEM) o público (PUBLIC). Si es PUBLIC, es necesario definir el FPI que, recordemos, es el nombre por el cual se identifica públicamente un determinado elemento (sea un DOCTYPE, un ELEMENT o una ENTITY), en este caso la entidad.

- Entidades no procesadas, referencian a datos que no deben ser procesados por el analizador XML, sino por la aplicación que lo use (como una imagen).

Sintaxis general: < !ENTITY % nombre _ entidad tipo _ uso fpi valor _ entidad NDATA tipo>

Ejemplo:

Se declara una notación (se comentará más adelante) de nombre JPG para el tipo **MIME** (**M**ultipurpose **I**nternet **M**ail **E**xtensions - Extensiones Multipropósito de Correo de Internet). Se declara una entidad no procesada de nombre mediterráneo, asociada al archivo de

imagen mediterraneo.jpg. Por último, se declara un elemento <rnar>, que cuenta con atributo imagen que es del tipo ENTITY recién declarado.

```
< !NOTATION JPG SYSTEM "image/jpeg">
< !ENTITY mediterraneo SYSTEM "mediterraneo. jpg" NDATA JPG>
< !ELEMENT mar . . . >
  <!ATTLIST mar
   imagen ENTITY #IMPLIED>
```

Y en el XML, el valor del atributo imagen del elemento <mar> es la entidad no procesada declarada en el DTD:

```
<mares>
   <mar imagen= "mediterraneo" >
      <nombre >Mediterráneo< /nombre>
   </mar>

</mares>
```

Se usan comillas dobles

NOTA: En ocasiones se utiliza otra tecnología, llamada XLink, en sustitución de las entidades no procesadas.

Ejemplo:

Una extensión del ejemplo anterior sería el permitir incluir múltiples imágenes como valor del atributo imagen del elemento <mar>.

```
< !NOTATION JPG SYSTEM " image/ jpeg" >
< !ENTITY mediterraneo1 SYSTEM "mediterraneo1. jpg" NDATA JPG>
< !ENTITY mediterraneo2 SYSTEM "mediterraneo2 . jpg" NDATA JPG>
< !ELEMENT mar . . . > !
< !ATTLIST mar
          imagen ENTITIES #IMPLIED>
```

Y en el documento XML, el valor del atributo imagen del elemento <mar> son las dos entidades no procesadas declaradas en el DTD:

```
<mares>
   <mar iinagen= "mediterraneo1 mediterraneo2 " >
      <nombre>Mediterráneo< /nombre>
   </mar>

< /mares>
```

### <!NOTATION>

Este es un elemento avanzado en el diseño de DTDs. Es una declaración del tipo de atributo NOTATION. Una notación se usa para especificar un formato de datos que no sea XML. Se usa con frecuencia para describir tipos MIME, como image/gif o image/jpg.

Se utiliza para indicar un tipo de atributo al que se le permite usar un valor que haya sido declarado como notación en el DTD.

Sintaxis general de la notación:

< !NOTATION nombre _ notación SYSTEM "identificador _ externo" >

Sintaxis general del atributo que la usa:

```
< !ATTLIST   nombre elemento
             nombre atributo   NOTATION valor defecto>
```

Ejemplo:

Se declaran tres notaciones que corresponden a otros tantos tipos MIME de imágenes (gif, jpg y png). También se declara un elemento <mar> y sus atributos imagen y formato _ imagen, este último referenciando a las notaciones recién creadas. Por último, se declara una entidad no procesada que se asocia a un archivo de imagen.

```
< !NOTATION GIF SYSTEM " image/gif " >
< !NOTATION JPG SYSTEM "image/ jpeg" >
< !NOTATION PNG SYSTEM " image/png" >
< !ELEMENT mar . . . >
< !ATTLIST mar
             imagen ENTITY #IMPLIED
             formato _ imagen NOTATION (GIF I JPG I PNG) #IMPLIED>
< !ENTITY mediterráneo SYSTEM "mediterraneo. jpg" "
```

Y en el documento XML, el valor del atributo imagen del elemento <mar> es la entidad no procesada declarada en el DTD, y el valor del atributo formato _ imagen el de una de sus alternativas válidas, la notación jPG:

```
<mares>
  <mar imagen="mediterraneo" formato_ímagen="JPG">
    <nombre>Mediterráneo</nombre>
  </mar>
    . . .
</mares>
```

### Secciones condicionales

Permiten incluir o excluir reglas en un DTD en función de condiciones. Sólo se pueden ubicar en DTDS externos. Su uso tiene sentido al combinarlas con referencias a entidades parámetro. Las secciones condicionales son IGNORE e INCLUDE, teniendo la primera precedencia sobre la segunda.

Ejemplo:

Supónganse dos estructuras diferentes para un mensaje:

- una sencilla que incluye emisor, receptor y contenido
- otra extendida que incluye los datos de la estructura sencilla más el título y el número de palabras.

Se quiere diseñar un DTD que, en función del valor de una entidad parámetro, incluya una estructura de mensaje o la otra. Por defecto se incluirá la larga.

El DTD, que ha de ser externo, tendrá la siguiente estructura:

```
<!-- Mensaje corto -->
<! [IGNORE[
    <!ELEMENT mensaje (emisor, receptor, contenido)>
]]>

<!-- Mensaje largo -->
<! [INCLUDE[
    <!ELEMENT mensaje (titulo, emisor, receptor, contenido, palabras)>
    <!ELEMENT titulo (#PCDATA)>
    <!ELEMENT palabras (#PCDATA)>
]]>

<!-- Declaración de elementos y atributos comunes -->
<!ELEMENT emisor (#PCDATA)>
<!ELEMENT receptor (#PCDATA)>
<!ELEMENT contenido (#PCDATA)>
```

Al añadir las referencias a entidad parámetro, el DTD quedaría:

```
<!ENTITY %corto "IGNORE">
<!ENTITY %largo "INCLUDE">
```

En el dtd ponemos largo como predeterminado

```
<!-- Mensaje corto -->
<![%corto[
    <!ELEMENT mensaje (emisor, receptor, contenido)>
]]>

<!-- Mensaje largo - ->
<![%largo[
    <!ELEMENT mensaje (titulo, emisor, receptor, contenido, palabras)>
    <!ELEMENT titulo (#PCDATA)>
    <!ELEMENT palabras (#PCDATA)>
]]>
<!-- Declaración de elementos y atributos comunes -->
<!ELEMENT emisor (#PCDATA)>
<!ELEMENT receptor (#PCDATA)>
<!ELEMENT contenido (#PCDATA)>
```

Para cambiar el tipo de mensaje que se va a incluir, basta con modificar la asociación de valores de %corto a INCLUDE y de %largo a IGNORE, de la forma:

```
<!ENTITY %corto "INCLUDE">

<!ENTITY %largo "IGNORE">
```

Se podrían dejar la declaración de las entidades en la DTD interna al documento XML, donde se determinaría qué tipo de mensaje se quiere incluir de manera especifica para ese documento:

```
<?xml version="1.0"?>
<!DOCTYPE mensaje SYSTEM "mensaje.dtd" [
    <!ENTITY %corto "INCLUDE">
    <!ENTITY %largo "IGNORE">
]>
```

En el XML decidimos usar la definición no predeterminada %corto

```
<mensaje>
...
</mensaje>
```

### 4.7.2. Poniendo todo junto

Se quiere construir un DTD que valide el siguiente documento XML, persona.xml:

```
<?xml version="1.0" encoding= " iso- 8859-1" ?>
< !DOCTYPE persona SYSTEM "persona.dtd"] >
<persona dni="12345678-L" estadoCivil="Casado">
   <nombre>María Pilar</nombre>
   <apellido>Sánchez</apellido>
   <edad>60</edad>
   <enActivo/>
< /persona>
```

El esquema XML asociado se quiere que cumpla ciertas restricciones semánticas:

- El atributo dni es un identificador obligatorio.
- El estado civil puede ser: Soltero, Casado o Divorciado. Por defecto es Soltero.
- El elemento <enActivo> es optativo.

El DTD que cumple con esto, ubicado en el archivo persona.dtd, es:

```
<!ELEMENT persona (nombre, apellido, edad, enActivo?)>
<!ATTLIST persona dni ID #REQUIRED
          estadoCivil (Soltero I Casado I Divorciado) "Soltero">
<!ELEMENT nombre (#PCDATA)>
<!ELEMENT apellido (#PCDATA)>
<!ELEMENT edad (#PCDATA)>
<!ELEMENT enActivo (#PCDATA) EMPTY>
```

## Herramientas de generación automática de DTDS

Va a haber muchos documentos XML que puedan ser válidos por un mismo DTD. Imagínese simplemente un DTD que valide documentos XML que simulen un correo electrónico básico, es decir, que contengan elementos <para>, <cc> (optativo), <cco> (optativo), <asunto> (optativo) y <cuerpo>.

Cualquier correo que cumpla con estas reglas será un documento válido con respecto a este DTD. Por ejemplo, igualmente válido seria un documento XML que contenga el elemento <cc> que el que no lo contenga, puesto que es optativo.

La cuestión aquí es que se va a tratar de inferir, a partir de un documento XML (que es un ejemplo concreto válido respecto a un DTD), el DTD genérico que lo valide.

Por ello, lo que se generará es un DTD que valide exactamente el XML que lo generó, pero si las reglas de negocio imponen restricciones adicionales, la herramienta de generación automática no será capaz de deducirlas, lo que nos obligará a añadirlas manualmente al DTD generado.

Entiéndase por **regla de negocio** a una restricción o característica propia del ámbito en el que se trabaja. Por ejemplo, una regla de negocio podría ser que un pago se haga en euros o dólares.

Ejemplo:

Se quiere generar un DTD a partir de un XML en el que hay un elemento <distancia>, el cual posee un atributo unidad, y se quiere que el valor de dicho atributo pueda ser centímetro, metro o kilómetro.

No hay manera de indicárselo al programa de generación automática. Éste sólo podrá suponer que existe un atributo unidad, perteneciente al elemento distancia, que contiene texto. Nada más. La enumeración de los posibles valores y, si se diera el caso, la existencia de uno de ellos por defecto, tendrá que indicarse "a mano" sobre el DTD generado.

A pesar de todo, son herramientas prácticas desde el punto de vista que realizan el trabajo mecánico inicial y, sobre el DTD que generan, resulta más cómodo introducir los ajustes necesarios para cumplir las restricciones semánticas (reglas de negocio).

ⓘ NOTA: Para lograr una inferencia lo más precisa posible, se recomienda escribir un documento XML lo más completo posible, sin omitir elementos opcionales, ni atributos, etc.

## Limitaciones de los DTD

Algunas limitaciones de los DTD son:

1. Un DTD no es un documento XML, luego no se puede verificar si está bien formado.
2. No se pueden fijar restricciones sobre los valores de elementos y atributos, como su tipo de datos, su tamaño, etc.
3. No soporta espacios de nombres.
4. Sólo se puede enumerar los valores de atributos, no de elementos.
5. Sólo se puede dar un valor por defecto para atributos, no para elementos.
6. Existe un control limitado sobre las cardinalidades de los elementos, es decir, concretar el número de veces que pueden aparecer.

# Resumen DTD

## Qué es DTD

- **DTD** (***D**ocument **T**ype **D**efinition*).
- Sirve para definir la estructura de un documento SGML o XML, permitiendo su validación.
- Un documento XML es válido (*valid*) cuando, además de estar bien formado, no incumple ninguna de las normas establecidas en su estructura.

## Declaración de tipo de documento

- Una DTD se puede escribir tanto interna como externamente a un archivo XML.
- En ambos casos hay que escribir una definición **DOCTYPE** (*Document Type Declaration*, Declaración de Tipo de Documento) para asociar el documento XML a la DTD. Asimismo, un archivo XML se puede asociar simultáneamente a una DTD interna y externa.
- Sintaxis DTD interna:
  **`<!DOCTYPE elemento-raíz [ declaraciones ]>`**
- Sintaxis DTD externa privada:
  **`<!DOCTYPE elemento-raíz SYSTEM "URI">`**
- Sintaxis DTD externa pública:
  **`<!DOCTYPE elemento-raíz PUBLIC "identificador-público" "URI">`**
- Sintaxis DTD interna y externa:
  **`<!DOCTYPE elemento-raíz SYSTEM "URI" [ declaraciones ]>`**
  **`<!DOCTYPE elemento-raíz PUBLIC "identificador-público" "URI" [ declaraciones ]>`**

## Estructura de un documento XML

- Un documento XML será válido si –además de no tener errores de sintaxis– cumple lo indicado en las declaraciones de elementos, atributos, entidades y notaciones, de la DTD a la que esté asociado.

## Declaración de elementos

- Sintaxis:
  **`<!ELEMENT nombre-del-elemento tipo-de-contenido>`**
- En el **`tipo-de-contenido`** se especifica el contenido permitido en el elemento, pudiendo ser:
  Texto, **`(#PCDATA)`**.
  Otros elementos (hijos).
  Estar vacío, **`EMPTY`**.
  Cualquier cosa (texto, etiquetas, otros elementos...), **`ANY`**.
- Un elemento vacío puede tener atributos.
- Un elemento (padre) puede ser declarado para contener a otro u otros elementos (hijos). En la sintaxis, los hijos –también llamados sucesores– tienen que escribirse entre paréntesis " `()` " y separados por comas " `,` ".
- Los elementos (hijos) de un elemento (padre), deben escribirse en el mismo orden en el que han sido declarados en la DTD.
- Operadores de cardinalidad en DTD:
  **`?`** (interrogación): 0-1
  **`*`** (asterisco): 0-n
  **`+`** (signo más): 1-n
- Los elementos declarados en una DTD sobre los que no actúe ningún operador de cardinalidad, tendrán que aparecer obligatoriamente una única vez, en el o los documentos XML a los que se asocie.
- En la DTD asociada a un documento XML, se pueden declarar elementos que contengan elementos opcionales. Para ello, se utiliza el *operador de elección*, representado por una barra vertical (`|`).
- Al utilizar el operador de elección (`|`) en una DTD, si una de las opciones es **`#PCDATA`**, esta debe escribirse en primer lugar.

## Declaración de atributos

- Sintaxis:
  **`<!ATTLIST nombre-del-elemento nombre-del-atributo tipo-de-atributo valor-del-atributo>`**

## Tipos de declaración de atributos

- **`valor`** entre comillas dobles (`"`) o simples (`'`).
- **`#REQUIRED`**
- **`#IMPLIED`**
- **`#FIXED valor`** entre comillas dobles (`"`) o simples (`'`).

**Tipos de atributos**

- **CDATA**, **Enumerado**, **ID**, **IDREF**, **IDREFS**, **NMTOKEN**, **NMTOKENS**, **NOTATION**, **ENTITY**, **ENTITIES**, **Especiales**

**Declaración de entidades**

- En una DTD se pueden declarar entidades generales y paramétricas (de parámetro).
- Las entidades generales pueden utilizarse en el cuerpo de un documento XML y en su DTD. Sin embargo, las entidades paramétricas solo pueden utilizarse dentro de la DTD.
- Sintaxis entidad general interna analizable:
  **<!ENTITY nombre-de-la-entidad "valor-de-la-entidad">**
- Sintaxis entidad general externa analizable privada:
  **<!ENTITY nombre-de-la-entidad SYSTEM "URI">**
- Sintaxis entidad general externa analizable públic:
- **<!ENTITY nombre-de-la-entidad PUBLIC "identificador-público" "URI">**
- Sintaxis entidad general externa no analizable privada:
  **<!ENTITY nombre-de-la-entidad SYSTEM "URI" NDATA notación>**
- Sintaxis entidad general externa no analizable pública:
  **<!ENTITY nombre-de-la-entidad PUBLIC "identificador-público" "URI" NDATA notación>**
- Sintaxis entidad paramétrica interna analizable:
  **<!ENTITY % nombre-de-la-entidad "valor-de-la-entidad">**
- Las entidades paramétricas tienen que declararse antes de ser referenciadas.
- Las entidades paramétricas pueden declararse en DTD internas o externas. Sin embargo, no pueden referenciarse desde una DTD interna.
- Sintaxis entidad paramétrica externa analizable privada:
  **<!ENTITY % nombre-de-la-entidad SYSTEM "URI">**
  **%nombre-de-la-entidad;**
- Sintaxis entidad paramétrica externa analizable pública:
  **<!ENTITY % nombre-de-la-entidad PUBLIC "identificador-público" "URI">**
  **%nombre-de-la-entidad;**
- Una entidad se puede usar dentro de otra.
- La referencia circular o recursiva de entidades no es correcta.

**Declaración de notaciones**

- Sintaxis notación privada:
  **<!NOTATION nombre-de-la-notación SYSTEM "identificador-del-sistema">**
- Sintaxis notación pública:
  **<!NOTATION nombre-de-la-notación PUBLIC "identificador-público">**
  **<!NOTATION nombre-de-la-notación PUBLIC "identificador-público" "identificador-del-sistema">**
- En una DTD, pueden existir elementos con atributos cuyo valor sea el nombre de una notación.

**Secciones condicionales**

- Sintaxis:
  **<![ IGNORE [ declaraciones ]]>**
  **<![ INCLUDE [ declaraciones ]]>**

**Llamadas a entidades**

- Generales: &entidad; (en el xml)
- Paramétricas: %entidad; (en el DTD)

**Comentarios**

- En una DTD asociada a un documento XML, se pueden escribir comentarios entre los caracteres "**<!--**" y "**-->**".

# Capítulo 4

# XML Schema

## 4.1. Introducción

Cada tipo de documento es apropiado para modelar cierto tipo de información, por ejemplo la estructura de una carta (compuesta de remitente, destinatario y contenido) es distinta de la estructura de un libro (compuesta de autor, titulo, capítulos, secciones, apéndices). El esquema de un documento lo determinan los elementos que lo componen y la forma en que dichos elementos se encuentran estructurados u organizados.

Un segundo componente de la familia XML corresponde a un lenguaje que nos permite especificar y validar la estructura o esquema que debe tener un tipo especifico de documento XML. Los DTDs y los XML Schemas son dos lenguajes que nos permiten describir esquemas de documentos. En este curso revisaremos algunos conceptos claves para el diseño de esquemas de documentos XML y nos concentraremos en el uso de XML Schema.

**Describiendo la estructura** Para describir la estructura de un documento debemos tener en cuenta ciertos aspectos:

- Estudiar el dominio de la Aplicación (ver estándares ya definidos).
- Definir el propósito y audiencia para los cuales se modela el documento.
- Definir partes del documento que son claramente identificables y cuando nivel de detalle es importante modelar.
- Identificar piezas o elementos repetidos que puedan ser agrupados o reutilizados (modularidad).

- Verificar si la estructura puede ser generalizada a otras instancias de documentos. Considerar futuras ampliaciones (extensibilidad).
- Tener claros los conceptos de contenido, estructura y presentación.
- La descripción final debe identificar claramente: estructuras componente-nivel; la naturaleza del contenido; reglas de orden y ocurrencia.

**Schema de un documento:** Un Esquema define la estructura de un tipo de documento, y es utilizado para validar o chequear que una instancia del documento cumple dicha estructura.

Para el caso de XML existen varios lenguajes que permiten definir esquemas que permitan validar documentos XML. Originalmente se utilizaron los DTDs (Document Type Definition), pero actualmente se utilizan esquemas XML (W3C XML Schema).

**Importancia de los esquemas** Un documento XML no necesita ir acompañado de un esquema, pero existen algunas razones por las cuales es importante su definición:

- La definición de un esquema permite establecer un contrato entre productores y consumidores de documentos XML. Este contrato especifica la estructura de los documentos XML que comparten.
- Permite a los productores de documentos XML asegurarse del contenido que ellos están proporcionando.
- Permite a los consumidores de documentos XML chequear lo que reciben de los productores y así proteger sus aplicaciones.
- Un esquema provee de un interfaz garantizada.
- Simplifican la tarea de los desarrolladores de aplicaciones para validación de documentos, dejando las tareas de detección y análisis de errores a un validador de propósito general.
- Los esquemas son un rica fuente de metadatos. Un esquema contiene información sobre los datos en los documentos instancia, por ejemplo tipos de datos, rango de valores, relaciones entre datos, etc).

instancia02.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE libreria [
  <!ENTITY amp "&#38;"> <!-- Entidad-->
  <!ELEMENT libreria (libro*, revista*)>
  <!ELEMENT libro (titulo, autor+, precio?)>
  <!ELEMENT revista (titulo, numero, precio?)>
  <!ELEMENT titulo (#PCDATA)>
  <!ELEMENT autor (#PCDATA)>
  <!ELEMENT numero (#PCDATA)>
  <!ELEMENT precio (#PCDATA)>
  <!ATTLIST precio
    moneda CDATA #REQUIRED
>
]>
<libreria>
 <libro>
  <titulo>El Hobbit</titulo>
  <autor>J.R.R. Tolkien</autor>
  <precio moneda="USdolar">15</precio>
 </libro>
 <revista>
  <titulo>Cine &amp; Arte</titulo>
  <numero/>
  <precio moneda="peso">3500</precio>
 </revista>
</libreria>
```

**Documento XML válido** Inicialmente vimos que un documento XML estaba bien formado si respetaba la sintaxis XML. Adicionalmente, si el documento respeta la reglas de estructura y tipos definidas por un esquema decimos que es un "documento XML válido"'.

**Document Type Definition (DTD):** Uno de los primeros lenguajes para declarar esquemas de documentos XML fueron los DTDs. Las declaraciones de tipos de documentos (DTDs) nos permiten declarar reglas para instancias de documentos, como se muestra en el ejemplo *instancia02.xml*.

### Limitaciones de los DTDs

- La Sintaxis no es XML (difíciles de manipular).
- No soportan espacios de nombres (ver sección 4.4).

- No permiten especificar tipos de datos (por ejemplo: enteros, flotantes, fechas, etc.).
- No hay soporte para declaraciones sensibles al contexto ya que todos los elementos se definen a nivel de documento.
- Soporte limitado para referencias cruzadas.
- No es posible formar claves a partir de varios atributos o de elementos.
- No son extensibles. Una vez definido, no es posible añadir nuevos vocabularios a un DTD.

## 4.2. XML Schema

XML Schema (esquema XML) [3] es un lenguaje de definición de esquemas para documentos XML propuesto por la W3C. Un esquema XML especifica los tipos de datos y como los datos XML estarán organizados en un documento instancia.

El archivo *esquema03.xsd* ilustra la declaración de un esquema XML para el documento *instancia03.xml*.

esquema03.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="libreria">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="libro" type="CTlibro" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="CTlibro">
   <xsd:sequence>
    <xsd:element name="titulo" type="xsd:string"/>
    <xsd:element name="autor" type="xsd:string"/>
    <xsd:element name="editorial" type="xsd:string" minOccurs="0"/>
   </xsd:sequence>
   <xsd:attribute name="codigo" type="xsd:string" use="required"/>
   <xsd:attribute name="stock" type="xsd:integer" use="optional"/>
  </xsd:complexType>
</xsd:schema>
```

instancia03.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<libreria xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="esquema03.xsd">
 <libro codigo="34674" stock="5">
  <titulo>El Hobbit</titulo>
        <autor>J.R.Tolkien</autor>
        <editorial>Minotauro</editorial>
 </libro>
 <libro codigo="83467">
  <titulo>El arte de la Guerra</titulo>
  <autor>Sun-Tzu</autor>
 </libro>
</libreria>
```

**Referenciar el esquema XML en un documento instancia:** La declaración `xsi:noNamespaceSchemaLocation="esquema03.xsd"` permite especificar la localización del esquema XML del documento instancia. Para poder utilizar el atributo `xsi:noNamespaceSchemaLocation` necesitamos declarar el prefijo `xsi`. Esto se hace con la declaración:
`xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`,
la cuál identifica el espacio de nombres para documentos instancia XML (en la sección 4.4 veremos el uso de espacios de nombres).

### 4.2.1. Características de XML schema

- Sintaxis XML y soporte para Espacios de Nombres.
- Mayor expresividad: restricciones numéricas, de estructura, etc.
- Tipos de datos: buena cantidad de tipos de datos predefinidos; adicionalmente soporta la creación de tipos de datos definidos por el usuario.
- Soporta construcción modular de esquemas.
- Extensibilidad: Inclusión y Redefinición de esquemas; reutilización y herencia de tipos de datos.
- Soporta Documentación.

### 4.2.2. Ventajas de XML Schemas

- Define tipos de datos primitivos mejorados y soporta la definición de tipos de datos personalizados.
- Tienen la misma sintaxis de los documentos instancia XML.
- Varias opciones para la definición de contenido de un elemento.
- Soporta la definición de múltiples elementos con el mismo nombre pero con diferente contenido.
- Permite definir elementos de contenido nulo.
- Soporta la definición de elementos sustituibles.
- Permite definir elementos únicos, claves y referencias a claves.
- Presenta mecanismos de orientación a objetos.

## 4.3. Sintaxis de XML Schema

En esta sección estudiaremos la sintaxis definida por la especificación de XML Schema [4, 5, 6].

### 4.3.1. Estructura Básica

**El elemento** `Schema`

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.libreria.org"
            xmlns="http://www.libreria.org"
            elementFormDefault="qualified"
            attributeFormDefault="qualified">
 ...
</xsd:schema>
```

- El elemento `schema` es el elemento contenedor de todas las declaraciones de elementos, atributos y tipos de datos del esquema XML.
- Los atributos `targetNamespace` y `xmlns` permiten definir espacios de nombres globales (esto será discutido en la sección 4.4).

- Todo esquema debe hacer referencia al espacio de nombres de XML Schema (`xmlns:xsd=‘‘http://www.w3.org/2001/XMLSchema"`).
  Este espacio de nombres contiene elementos para la definición del esquema XML (ej. xsd:schema, xsd:element, etc.) y tipos de datos definidos por XML Schema (ej. xsd:integer, xsd:string, etc.).

- Los atributos `elementFormDefault` y `attributeFormDefault` permiten establecer si ciertos elementos o atributos deben ser calificados en el documento instancia (esto será discutido en la sección 4.4).

**Anotaciones** El elemento `xsd:annotation` es usado para documentar el schema y su contenido no afecta la validación del mismo. Este elemento puede ir antes o después de algún componente global y solo al inicio de componentes no globales.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:annotation>
  <xsd:documentation xml:lang="es">
   Esquema XML para datos de una Libreria
  </xsd:documentation>
  <xsd:appinfo>
   <archivo>esquema03.xsd</archivo>
  </xsd:appinfo>
 </xsd:annotation>
 ...
</xsd:schema>
```

El elemento `xsd:annotation` contiene dos subelementos:

- `xsd:documentation` se usa para describir un comentario para personas. Puede tener 2 atributos: `source` que contiene un URL de un archivo con información complementaria; y `xml:lang` que especifica el lenguaje de la documentación.

- `appinfo` se usa para proporcionar comentarios a programas. Su contenido es algún texto XML bien formado y puede contener el atributo `source`.

**Declaraciones globales y locales:** XML schema permite definir la estructura de un documento declarando elementos simples, elementos complejos, tipos simples, tipos complejos y atributos. Estos pueden ser declarados para uso global o local:

- Los elementos, tipos o atributos *globales* son creados por declaraciones que aparecen como hijos del elemento `xsd:schema`.
- Los elementos, tipos o atributos *locales* son definidos dentro de otros elementos distintos de `xsd:schema`.

Los elementos y atributos globales son reutilizados en cualquier parte del esquema usando el atributo `ref`. Los tipos globales son reutilizados usando el atributo `type`. El ejemplo (*esquema04.xsd*) presenta definiciones globales y locales.

esquema04.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="libreria">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="libro" type="CTlibro" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <!-- tipo complejo global -->
 <xsd:complexType name="CTlibro">
  <xsd:sequence>
   <xsd:element ref="titulo"/>
   <xsd:element name="autor" type="xsd:string"/>
   <xsd:element name="editorial" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="codigo" type="STcodigo" use="required"/>
  <xsd:attribute ref="stock" use="optional"/>
 </xsd:complexType>
 <!-- elemento global -->
 <xsd:element name="titulo" type="xsd:string"/>
 <!-- atributo global -->
 <xsd:attribute name="stock" type="xsd:integer"/>
 <!-- tipo simple global -->
 <xsd:simpleType name="STcodigo">
  <xsd:restriction base="xsd:string">
   <xsd:length value="5"/>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:schema>
```

### 4.3.2. Declaración de Tipos

**Tipos Simples** : son elementos que representan datos atómicos, por consiguiente no pueden contener elementos ni atributos. Existen 2 clases de tipos simples:

- Predefinidos o built-in: Definidos en la especificación de XML Schema (Primitivos, Derivados predefinidos). La figura A.1 presenta la Jerarquía de tipos de XML schema. Las tablas A.1 y A.2 describen los tipos primitivos y derivados predefinidos.
- Definidos por el usuario: son los tipos simples definidos a partir de tipos built-in. Mas adelante veremos como son declarados.

**Tipos Complejos** : Son tipos que tienen contenido complejo en el sentido que pueden contener subelementos y atributos. La definición de su contenido lo constituyen la declaración de los subelementos seguido de las declaraciones de atributos.

El orden de los atributos no es relevante en el documento instancia, sin embargo existen distintas formas de estructurar los subelementos de un tipo complejo (*esquema05.xsd*):

- Alternativa: `xsd:choice` define un conjunto de elementos alternativos (or-exclusiva), por lo tanto el elemento puede contener solo uno de los subelementos.

```
<xsd:complexType name="CTpublicacion">
 <xsd:choice>
  <xsd:element name="libro" type="CTlibro"/>
  <xsd:element name="revista" type="CTrevista"/>
 </xsd:choice>
</xsd:complexType>
```

- Secuencia ordenada: `xsd:sequence` define un conjunto de elementos que deben ir ordenados.

```
<xsd:complexType name="CTlibro">
 <xsd:sequence>
  <xsd:element name="titulo" type="xsd:string"/>
  <xsd:element name="autor" type="xsd:string"/>
  <xsd:element name="editorial" type="xsd:string" minOccurs="0"/>
 </xsd:sequence>
 <xsd:attribute name="codigo" type="xsd:string" use="required"/>
 <xsd:attribute name="stock" type="xsd:integer" use="optional"/>
</xsd:complexType>
```

- Secuencia no ordenada: `xsd:all` define un conjunto de subelementos que pueden ir en cualquier orden.

```
<xsd:complexType name="CTrevista">
 <xsd:all>
  <xsd:element name="titulo" type="xsd:string"/>
  <xsd:element name="numero" type="xsd:string"/>
 </xsd:all>
 <xsd:attribute name="codigo" type="xsd:string" use="required"/>
 <xsd:attribute name="stock" type="xsd:integer" use="optional"/>
</xsd:complexType>
```

  Observaciones:
  - Los elementos `xsd:choice` y `xsd:sequence` pueden contener a su vez otras declaraciones `xsd:sequence` y `xsd:choice`, pero no `xsd:all`.
  - El elemento `xsd:all` solo puede contener declaraciones de elementos, no puede contener elementos *xsd:sequence, xsd:choice, xsd:all.*
  - Los elementos declarados dentro de `xsd:all` deben tener los atributos `maxOccurs` y `minOccurs` con valores de "0"ó "1".

- Contenido mixto: al declarar el atributo `mixed="true"` definimos un elemento cuyo contenido consiste en una mezcla de texto y subelementos (*esquema06.xsd*).

```
<xsd:element name="carta">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="cuerpo">
    <xsd:complexType mixed="true">
     <xsd:sequence>
      <xsd:element name="enfatizar" type="xsd:string"/>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

  Esta definición permite el siguiente contenido:

```
<cuerpo>
 Estimados clientes.
 Esta carta es para desearles
 una <enfatizar>feliz navidad</enfatizar>.
</cuerpo>
```

### 4.3.3. Restricciones de ocurrencia para elementos y atributos

Existen una serie de atributos que permiten restringir la ocurrencia y los posibles valores que tomarán los elementos y atributos (ver tabla 4.1). Estos atributos llamados *facetas* son:

- `maxOccurs`: numero máximo de veces que un elemento puede aparecer.
- `minOccurs`: numero mínimo de veces que un elemento puede aparecer.
- `use`: define si un atributo será opcional("optional"), requerido("required"') o prohibido ("prohibited").
- `fixed`: define un valor fijo para un elemento simple, complejo o un atributo. Si no se incluye, se utiliza el valor fijo. Si se incluye, debe coincidir con el valor definido.
- `default`: define un valor por defecto para un elemento (de contenido simple) o atributo.

| faceta/valor | Elementos | Atributos | Default |
|---|---|---|---|
| `minOccurs` | entero positivo o "unbounded" | – | "1" |
| `maxOccurs` | entero positivo o "unbounded" | – | "1" |
| `use` | –<br>–<br>– | "required"<br>"optional"<br>"prohibited" | "optional" |
| `fixed` | algún valor | algún valor | – |
| `default` | algún valor | algún valor | – |

Cuadro 4.1: Restricciones de ocurrencia ("–"significa no aplicable)

Observaciones:
- El uso de los atributos `fixed` y `default` es excluyente.
- El valor por defecto (default) de un atributo solo tiene sentido si el atributo es opcional.
- Los valores por defecto de los atributos se aplican cuando éstos no están presentes, y los valores por defecto de los elementos se aplican cuando estos están vacíos.
- Los atributos `minOccurs`, `maxOccurs` y `use`, no pueden aparecer en la declaración de elementos o tipos globales.
- Al usar los atributos `fixed` o `default` en un elemento de contenido complejo, será necesario el uso de entidades predefinidas para representar símbolos no permitidos como contenido (Ver tabla 3.1).

### 4.3.4. Elementos y atributos

**Declaración de elementos** La instrucción `xsd:element` permite declarar un elemento, ya sea éste de tipo simple o complejo.

Existen 2 formas de declarar un elemento (*esquema07.xsd*):

- Haciendo referencia a un tipo simple (C1), a un tipo complejo (C2) o a un elemento definido globalmente (C3).

```
(C1) <xsd:element name="autor" type="xsd:string"/>
(C2) <xsd:element name="libro" type="CTlibro" maxOccurs="unbounded"/>
(C3) <xsd:element ref="titulo"/>
```

  Observación: Si al definir un elemento no se especifica el atributo `type`, el tipo asignado por defecto es `xsd:anyType`, el cual permite que el elemento contenga cualquier cosa.

- Definiendo "inline"el contenido del elemento, ya sea como un tipo simple (C4) o un tipo complejo (C5).

```
          (C4)                          (C5)
<xsd:element name="editorial">  |  <xsd:element name="libreria">
 <xsd:simpleType>               |   <xsd:complexType>
  ...                           |    ...
 </xsd:simpleType>              |   </xsd:compleType>
</xsd:element>                  |  </xsd:element>
```

**Declaración de atributos:** La instrucción `xsd:attribute` permite la declaración de atributos. Un atributo puede ser global al declararse como hijo del elemento `xsd:schema`, o puede ser ser local al declararse como contenido de cualquier otro elemento. Un atributo puede ser declarado de 2 maneras:

- Haciendo referencia a un tipo simple predefinido (C1), a un tipo simple personalizado (C2) o a un atributo definido globalmente (C3) .

```
(C1) <xsd:attribute name="issn" type="xsd:string"/>
(C2) <xsd:attribute name="stock" type="STstock"/>
(C3) <xsd:attribute ref="isbn"/>
```

- Definición *"inline"* del valor del atributo.

```
<xsd:attribute name="codigo" use="required">
 <xsd:simpleType>
  ...
 </xsd:simpleType>
</xsd:attribute>
```

**Agrupar Declaraciones:** Con la finalidad de organizar un esquema XML, podemos usar las instrucciones `xsd:group` y `xsd:attributeGroup` para agrupar declaraciones de elementos y atributos respectivamente. Para poder referenciar un grupo, este debe ser declarado globalmente (*esquema08.xsd*).

```
<xsd:complexType name="CTlibro">
 <xsd:sequence>
  <xsd:group ref="elementosPublicacion"/>
  <xsd:element name="editorial" type="xsd:string" minOccurs="0"/>
 </xsd:sequence>
 <xsd:attributeGroup ref="atributosPublicacion"/>
</xsd:complexType>

<xsd:group name="elementosPublicacion">
 <xsd:sequence>
  <xsd:element name="titulo" type="xsd:string"/>
  <xsd:element name="autor" type="xsd:string"/>
 </xsd:sequence>
</xsd:group>

<xsd:attributeGroup name="atributosPublicacion">
 <xsd:attribute name="codigo" type="xsd:string" use="required"/>
 <xsd:attribute name="stock" type="xsd:integer" use="optional"/>
</xsd:attributeGroup>
```

### 4.3.5. Derivación de Tipos

La Derivación de tipos consiste en crear *tipos derivados* desde tipos ya existentes llamados tipos base. XML esquema soporta la creación de tipos derivados simples y tipos derivados complejos.

**Tipos simples Derivados (Tipos simples personalizados):** Un nuevo tipo simple derivado puede crearse restringiendo el valor de otro tipo llamado tipo base, el cual puede ser un tipo simple primitivo (Ej. xsd:string) u otro tipo simple derivado. El nuevo tipo simple es creado usando el elemento `xsd:restriction` y especificando valores para una o mas facetas opcionales, las cuales dependen del tipo que será extendido:
- Facetas del *tipo string*: length, minLength, maxLength, pattern, enumeration, whitespace.
- Facetas del *tipo integer*: totalDigits, maxInclusive, maxExclusive, minInclusive, minExclusive, enumeration, whitespace, pattern.
Nota. La faceta *pattern* permite el uso de expresiones regulares. La tabla A.3 presenta ejemplos de expresiones regulares.

Algunos ejemplos de tipos simples derivados (*esquema07.xsd*):

- Aplicando facetas `xsd:minInclusive` y `xsd:maxInclusive`. El valor predefinido para ambas facetas es "1".

```
<xsd:simpleType name="STstock">
 <xsd:restriction base="xsd:integer">
  <xsd:minInclusive value="2"/>
  <xsd:maxExclusive value="10"/>
 </xsd:restriction>
</xsd:simpleType>
```

- Aplicando faceta `xsd:enumeration`.

```
<xsd:element name="editorial">
 <xsd:simpleType>
  <xsd:restriction base="xsd:string">
   <xsd:enumeration value="Minotauro"/>
   <xsd:enumeration value="O'Reilly"/>
   <xsd:enumeration value="McGraw Hill"/>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
```

- Aplicando faceta `xsd:length`

```
<xsd:attribute name="codigo" use="required">
 <xsd:simpleType>
  <xsd:restriction base="xsd:string">
   <xsd:length value="5"/>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:attribute>
```

- Aplicando faceta `xsd:pattern`.

```
<xsd:simpleType name="STrut">
 <xsd:restriction base="xsd:string">
  <xsd:pattern value="\d{8}-(\d{1}|k)"/>
 </xsd:restriction>
</xsd:simpleType>
```

Adicionalmente existen dos instrucciones que nos ayudan a definir el contenido de un tipo simple:

- `xsd:union` permite definir un tipo simple como la unión de 2 o mas tipos simples.

```
<xsd:simpleType name="STnota1">
 <xsd:restriction base="xsd:float">
  <xsd:minInclusive value="0" />
  <xsd:maxInclusive value="7" />
 </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="STnota2">
 <xsd:restriction base="xsd:string">
  <xsd:enumeration value="NSP" />
 </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="STnota">
 <xsd:union memberTypes="STnota1 STnota2"/>
</xsd:simpleType>
```

- `xsd:list` permite definir que el contenido de un tipo simple es una lista de valores separados por espacios.

```
<xsd:simpleType name="STlistaCodigos">
 <xsd:list itemType="xsd:positiveInteger"/>
</xsd:simpleType>
...
<xsd:element name="codigos" type="STlistaCodigos"/>
```

  La definición anterior permitiría un elemento de la forma:
  <codigos>637 746 348 829</codigos>

  Observaciones:
  - No se pueden crear listas de listas, ni listas de tipos complejos
  - En el documento instancia los elementos de la lista deben ir separados por espacios en blanco.
  - Las facetas disponibles para `xsd:list` son: length, minLength, maxLength, enumeration, pattern.

**Tipos complejos Derivados:** Existen 2 formas de crear tipos complejos derivados, por extensión y por restricción de tipos (*esquema09.xsd*).

- Derivar por extensión: extiende el tipo complejo base agregando mas elementos o atributos. Este tipo de derivación permite aplicar el concepto de polimorfismo.

```
<xsd:complexType name="CTpublicacion">
 <xsd:sequence>
  <xsd:element name="titulo" type="xsd:string"/>
  <xsd:element name="autor" type="xsd:string" maxOccurs="unbounded"/>
 </xsd:sequence>
 <xsd:attribute name="codigo" type="xsd:string" use="required"/>
 <xsd:attribute name="stock" type="xsd:integer" use="optional"/>
</xsd:complexType>

<xsd:complexType name="CTlibro">
 <xsd:complexContent>
  <xsd:extension base="CTpublicacion">
   <xsd:sequence>
    <xsd:element name="editorial" type="xsd:string" minOccurs="0"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Un documento instancia permitirá los siguientes elementos:

```
<publicacion codigo="42345">
 <titulo>Comunidades Virtuales</titulo>
 <autor>J.H.Gray</autor>
 <autor>S.Kling</autor>
</publicacion>

<libro codigo="34674" stock="5">
 <titulo>El Hobbit</titulo>
 <autor>J.R.Tolkien</autor>
 <editorial>Minotauro</editorial>
</libro>
```

- Derivar por restricción: define un tipo el cual es subconjunto del tipo base. Existen 2 maneras:

  - Redefinir el tipo base para restringir el contenido eliminando o agregando elementos. Siguiendo el ejemplo anterior, podemos definir un elemento revista como una publicación que no tiene autores.

```
<xsd:complexType name= "CTrevista">
 <xsd:complexContent>
  <xsd:restriction base="CTpublicacion">
   <xsd:sequence>
    <xsd:element name="titulo" type="xsd:string"/>
    <xsd:element name="numero" type="xsd:string"/>
   </xsd:sequence>
  </xsd:restriction>
 </xsd:complexContent>
</xsd:complexType>
```

    Un documento instancia permitirá el siguiente elemento:

```
<revista codigo="38475">
 <titulo>PC-Magazine</titulo>
 <numero>Enero2006</numero>
</revista>
```

  - Redefinir el tipo base para tener un número más restringido de ocurrencias o valores. Esto se logra modificando los valores para `minOccurs` y `maxOccurs` para los subelementos y `use` para los atributos. En el ejemplo inicial, podemos podemos definir un tipo de publicación que tenga un solo autor.

```
<xsd:complexType name="CTpublicacionPersonal">
 <xsd:complexContent>
  <xsd:restriction base="CTpublicacion">
   <xsd:sequence>
    <xsd:element name="titulo" type="xsd:string"/>
    <xsd:element name="autor" type="xsd:string" maxOccurs="1"/>
   </xsd:sequence>
  </xsd:restriction>
 </xsd:complexContent>
</xsd:complexType>
```

    Un documento instancia permitirá el siguiente elemento:

```
<publicacionPersonal codigo="43434">
 <titulo>Remembranzas de guerra</titulo>
 <autor>A. Scholl</autor>
</publicacionPersonal>
```

**Tipos complejos especiales:**

- Definir un tipo complejo con atributos y contenido simple

```
<xsd:element name="precioA">
 <xsd:complexType>
  <xsd:simpleContent>
   <xsd:extension base="xsd:decimal">
    <xsd:attribute name="moneda" type="xsd:string"/>
   </xsd:extension>
  </xsd:simpleContent>
 </xsd:complexType>
</xsd:element>
```

Un documento instancia permitirá el siguiente elemento:

```
<precioA moneda="peso">5000</precioA>
```

- Definir un tipo complejo con atributos y contenido vacío

```
<xsd:element name="precioB">
 <xsd:complexType>
  <xsd:complexContent>
   <xsd:restriction base="xsd:anyType">
    <xsd:attribute name="moneda" type="xsd:string" use="required"/>
    <xsd:attribute name="valor" type="xsd:decimal" use="required"/>
   </xsd:restriction>
  </xsd:complexContent>
 </xsd:complexType>
</xsd:element>
```

Un documento instancia permitirá el siguiente elemento:

```
<precioB moneda="peso" valor="5000"/>
```

**Controlar la derivación de tipos complejos:** Al definir un elemento complejo, podemos evitar su posible derivación definiendo el atributo `final` con alguno de los siguientes valores:
- "restriction": El elemento no permite derivaciones por restricción.
- "extension": El elemento no permite derivaciones por extensión.
- "#all"El elemento no permite derivaciones ni por extensión ni por restricción.

```
<xsd:complexType name="..." final="restriction | extension | #all" >
```

## 4.4. Espacios de nombres (XML Namespaces)

### 4.4.1. Introducción

Un *Uniform Resource Identifier (URI)* [7] o Identificador Uniforme de Recursos es una cadena compacta de caracteres que permite identificar un recurso abstracto o físico (servicio, página, documento, dirección de correo electrónico, enciclopedia, etc).

Normalmente un URI consta de dos partes, el identificador del método de acceso o protocolo al recurso (ej. http:, mailto:, ftp) y el nombre del recurso (ej. "//www.libreria.org"). Pueden identificarse dos tipos de URI:

- URL (Uniform Resource Locator - Localizador Uniforme de Recursos) [8]: permite identificar recursos mediante una representación de su mecanismo primario de acceso (ej. localización en la red), en vez de identificar el recurso por su nombre o por otro atributo del recurso.

  `http://www.libreria.org/libros/elhobbit.xml`

- URN (Uniform Resource Name - Nombre Uniforme de Recursos) [9]: permite etiquetar persistentemente un recurso con un identificador aún cuando el recurso desaparezca o no este disponible. El recurso es identificado independientemente de su localización.

  `urn:libreria:libros:elhobbit:xml`

### 4.4.2. Objetivo de los Espacios de Nombres

Supongamos que tenemos los siguientes fragmentos XML; Un primer fragmento contiene datos de Chile, el segundo representa una inversión y el tercero junta los datos anteriores.

```
<!-- fragmento 1 -->
<pais nombre="Chile">
  <capital>Santiago</capital>
</pais>

<!-- fragmento 2 -->
<inversion>
  <capital>$7000</capital>
</inversion>
```

```
<!-- fragmento 3 -->
<inversiones>
 <pais nombre="Chile">
  <capital>Santiago</capital>
  <capital>$1200</capital>
 </pais>
 ...
</inversiones>
```

Vemos que el significado del elemento *"capital"* depende del contexto en el cual se utiliza. Este problema se conoce como *Homonimia* (mismo nombre con diferentes propósitos) y puede causar muchos problemas cuando queremos reutilizar o integrar datos.

El problema de homonimia, puede solucionarse asociando a cada elemento un URI que indica cual es el contexto (o espacio de nombres) al que pertenece. Por ejemplo, podríamos definir dos espacios de nombres, uno para identificar datos sobre geografía (`"http://www.geografia.org/terminos#"`) y otro para datos sobre economía (`"http://www.economia.org/terminos#"`). De esta manera los fragmentos iniciales se representarían como:

```
<http://www.geografia.org/terminos#pais nombre="Chile">
 <http://www.geografia.org/terminos#capital>
  Santiago
 </http://www.geografia.org/terminos#capital>
</http://www.geografia.org/terminos#pais>

<http://www.economia.org/terminos#inversion>
 <http://www.economia.org/terminos#capital>
  $7000
 </http://www.economia.org/terminos#capital>
</http://www.economia.org/terminos#inversion>

<inversiones>
 <http://www.geografia.org/terminos#pais nombre="Chile">
  <http://www.geografia.org/terminos#capital>
   Santiago
  </http://www.geografia.org/terminos#capital>
  <http://www.economia.org/terminos#capital>$1200</capital>
 </http://www.geografia.org/terminos#pais>
 ...
</inversiones>
```

Esta solución genera un nuevo problema, al agregar a cada elemento el URI al que pertenece sobrecargamos el documento con información repetida y perdemos legibilidad.

Para evitar la información redundante de la solución anterior, se propuso definir un *alias* para cada URI utilizado en el documento, el cual podrá ser usado como un *prefijo* al utilizar los elementos de un espacio de nombres.

```
<!--
Prefijos:
geo =  http://www.geografia.org/terminos#
eco = http://www.economia.org/terminos#
-->
<inversiones>
 <geo:pais nombre="Chile">
  <geo:capital>Santiago</geo:capital>
  <eco:capital>$1200</eco:capital>
 </geo:pais>
 ...
</inversiones>
```

En resumen, los espacios de nombres nos permiten solucionar el problema de homonimia y son ampliamente utilizados para combinar vocabularios ya que facilitan la incorporación de elementos no previstos inicialmente

### 4.4.3. Definiendo y utilizando espacios de nombres

**Definiendo espacios de nombres en un esquema XML** .
El elemento *schema* define los siguientes atributos relacionados a espacios de nombres (ver el ejemplo *esquema10.xsd*):

- `targetNamespace` (espacio de nombres de destino) define el espacio de nombres del esquema. El targetNamespace asocia los elementos definidos en el esquema con un espacio de nombres.
- `xmlns` define un espacio de nombres por defecto (default Namespace). Este será usado para identificar elementos que no tienen asignado un prefijo dentro del esquema. En el ejemplo el namespace por defecto es idéntico al targetNamespace, pero puede referenciar otro namespace.
- `xmlns:alias` define un prefijo "alias"que permitirá referenciar a un namespace en el ámbito del esquema (namespace global). De esta manera se pueden definir distintos alias a namespaces, por ejemplo `xmlns:xsd` asigna el alias *"xsd"* al namespace de XML Schema.
- Los atributos `elementFormDefault` y `attributeFormDefault` especifican si los elementos o atributos globales definidos en el esquema deben ser explícitamente calificados ("qualified") o no calificados ("unqualified") en un documento instancia (Ver mas adelante).

esquema10.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://www.libreria.org"
            xmlns="http://www.libreria.org"
            xmlns:lib="http://www.libreria.org"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="unqualified"
            attributeFormDefault="unqualified">
 <xsd:element name="libreria">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="libro" type="CTlibro" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:complexType name="CTlibro">
  <xsd:sequence>
   <xsd:element ref="titulo"/>
   <xsd:element name="autor" type="xsd:string"/>
   <xsd:element name="editorial" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="codigo" type="xsd:string" use="required"/>
  <xsd:attribute name="stock" type="xsd:integer" use="optional"/>
 </xsd:complexType>
 <xsd:element name="titulo" type="xsd:string"/>
</xsd:schema>
```

A excepción de los esquemas sin espacio de nombres, un esquema XML define al menos dos espacios de nombres, el *targetNamespace* y el *XML Schema namespace* (`xmlns:xsd="http://www.w3.org/2001/XMLSchema"`).

**Espacios de nombres en instancias XML** .
Si el esquema no define un espacio de nombres, se hace uso del atributo `xsi:noNamespaceSchemaLocation` para referenciarlo.

```
<?xml version="1.0" encoding="UTF-8"?>
<libreria
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="esquema04.xsd">
  ...
  ...
</libreria>
```

Cuando un esquema define un espacio de nombres usando el atributo *targetNamespace*, sus documentos instancia deberán hacer referencia a este esquema usando el atributo `xsi:schemaLocation` y declarar los prefijos de espacios de nombres que sean necesarios. Observar el ejemplo *instancia10.xml*.

instancia10.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<lib:libreria
  xmlns:lib="http://www.libreria.org"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.libreria.org esquema10.xsd">
 <libro codigo="34674" stock="5">
  <lib:titulo>El Hobbit</lib:titulo>
  <autor>J.R.Tolkien</autor>
  <editorial>Minotauro</editorial>
 </libro>
 <libro codigo="83467">
  <lib:titulo>El arte de la Guerra</lib:titulo>
  <autor>Sun-Tzu</autor>
 </libro>
</lib:libreria>
```

**Calificar o no calificar elementos en los documentos instancia** .
Si un elemento o atributo especifica su espacio de nombres usando un prefijo (por ejemplo `<lib:titulo>`), o usando el namespace por defecto, decimos que es un elemento o atributo calificado.

Los atributos `elementFormDefault`(forma de elemento por defecto) y `attributeFormDefault`(forma de atributo por defecto) permiten especificar si los elementos o atributos deberán ir calificados o no en un documento instancia:

- Si su valor es 'unqualified"(valor por defecto) *solo* los elementos o atributos declarados globalmente en el esquema deben ir calificados en un documento instancia. En contraste, los elementos y atributos definidos localmente no deben ir calificados.
- Si su valor es "qualified", todos los elementos o atributos usados en un documento instancia deben ir calificados.

*Observaciones:*
- Los elementos y atributos definidos globalmente en el esquema deberán ir necesariamente calificados en un documento instancia, sin importar el valor de los atributos `elementFormDefault` o `attributeFormDefault`.
- El uso del default namespace como medio para calificar no es aplicable para *attributeFormDefault="qualified"*.
- La calificación puede especificarse de forma separada para cada declaración local utilizando el atributo *form*.
- Si un esquema no tiene un namespace asociado, los atributos `elementFormDefault` y `attributeFormDefault` no afectan los documentos instancia.
- Los ejemplos *esquema10a, esquema10b, esquema10c, esquema10d* presentan las posibles situaciones de calificar y no calificar.

## 4.5. Conceptos Avanzados

### 4.5.1. Elementos Nulos

El atributo `nillable="true"` permite especificar que un elemento puede aceptar un valor nulo, sin implicar que el elemento este vacio. Por ejemplo considere la siguiente declaración (*esquema14.xsd*):

```
<xsd:element name="persona">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="nombre" type="xsd:string"/>
   <xsd:element name="primerApellido" type="xsd:string"/>
   <xsd:element name="segundoApellido" type="xsd:string" nillable="true"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

Un documento instancia permitirá el siguiente elemento:

```
<persona>
 <nombre>Luis</nombre>
 <primerApellido>Rojas</primerApellido>
 <segundoApellido xsi:nil="true"/>
</persona>
```

Observaciones:
- El atributo `xsi:nil` se define como parte del espacio de nombres para instancias del esquema XML.
- Un elemento con atributo `xsi:nil="true"` puede no tener contenido pero puede tener atributos.

### 4.5.2. Elementos `any`, `anyType` y `anyAttribute`

**Elementos con cualquier contenido** El tipo de dato *xsd:anyType* no restringe el contenido de un elemento de modo alguno, permitiendo cualquier mezcla de caracteres excepto elementos.

```
<xsd:element name="cualquier_cosa" type="xsd:anyType">
```

*anyType* es el tipo usado por defecto cuando no se especifica alguno. Por lo tanto la declaración anterior es equivalente a :

```
<xsd:element name="cualquier_cosa"/>
```

**Permitiendo cualquier elemento o atributo** .
El elemento `xsd:any` permite que en un documento instancia contenga elementos no definidos por el esquema.
De manera similar, el elemento `xsd:anyAttribute` permite que un elemento contenga cualquier atributo no definido por el esquema.

El siguiente ejemplo (*esquema15.xsd*), muestra la definición del elemento `persona`, el cual permite agregar elementos y/o atributos no definidos en el esquema como `id`, `edad`, `direccion` y `telefono`.

```
<xsd:element name="persona">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="nombre" type="xsd:string"/>
   <xsd:element name="apellidos" type="xsd:string"/>
   <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:anyAttribute/>
 </xsd:complexType>
</xsd:element>
```

Un documento instancia (*instancia15.xml*) permitirá el siguiente elemento:

```
<persona id="427" edad="26">
 <nombre>Luis</nombre>
 <apellidos>Rojas</apellidos>
 <direccion>Romero 2385</direccion>
 <telefono>89428282</telefono>
</persona>
```

*Nota:* Un documento instancia cuyo esquema utiliza `xsd:any` o `xsd:anyAttribute` se denomina *extensible*.

El atributo `processContents` especifica como se evaluará el contenido del elemento o atributo, pudiendo tomar uno de los siguientes valores:
- `"strict"` el contenido debe estar declarado en el esquema (default).
- `"skip"` el contenido solo debe ser bien formado.
- `"lax"`, aceptar cualquier contenido y validar cuando sea posible.

### 4.5.3. Grupos de sustitución

Los grupos de sustitución permiten que los elementos sean sustituidos por otros elementos. Se definen declarando un elemento denominado "head"(cabecera) y luego declarando otros elementos sustituibles usando el atributo `substitutionGroup`.
El siguiente ejemplo (*esquema18.xsd*) declara el grupo de sustitución `grupoTrabajo`:

```
<xsd:element name="grupoTrabajo">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element ref="integrante" minOccurs="3" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>

<!-- declaramos un elemento que actuará como "head" -->
<xsd:element name="integrante" type="xsd:string"/>
<!-- asignamos elementos al grupo de sustitucion-->
<xsd:element name="jefe" substitutionGroup="integrante"
             type="xsd:string"/>
<xsd:element name="analista" substitutionGroup="integrante"
             type="xsd:string"/>
<xsd:element name="programador" substitutionGroup="integrante"
             type="xsd:string"/>
```

Un instancia válida sería (*instancia18.xml*):

```
<grupoTrabajo>
 <integrante>Jorge</integrante>
 <jefe>Adrian</jefe>
 <analista>Luis</analista>
 <programador>Andres</programador>
</grupoTrabajo>
```

*Observaciones:*
- Los elementos del *substitutionGroup* pueden ser utilizados en cualquier lugar que pudiésemos utilizar el elemento cabecera.

- El elemento cabecera así como los elementos del *substitutionGroup* deben ser declarados globalmente.
- Cuando se crean grupos de sustitución a nivel de tipos, se debe considerar que el tipo de cada elemento del substitutionGroup debe ser el mismo o un derivado del tipo del elemento cabecera.

### 4.5.4. Elementos y tipos abstractos

Los tipos o elementos abstractos sirven como plantilla para crear tipos derivados. Un elemento o tipo se declara como *abstracto* definiendo el atributo `abstract="true"`.

**Elementos Abstractos:** son elementos que pueden ser reemplazados por algún elemento perteneciente a un grupo de sustitución. Un elemento abstracto no puede usarse en un documento instancia.

El siguiente ejemplo (*esquema16.xsd*) declara el elemento abstracto `integrante`:

```
<!-- declaramos un elemento que actuara como cabecera -->
<xsd:element name="integrante" type="xsd:string" abstract="true"/>

<!-- asignamos elementos al grupo de sustitucion-->
<xsd:element name="jefe" substitutionGroup="integrante"
             type="xsd:string"/>
<xsd:element name="analista" substitutionGroup="integrante"
             type="xsd:string"/>
<xsd:element name="programador" substitutionGroup="integrante"
             type="xsd:string"/>
<xsd:element name="grupoTrabajo">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element ref="integrante" minOccurs="2" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

Un instancia válida sería (*instancia16.xml*):

```
<grupoTrabajo>
 <jefe>Adrian</jefe>
 <analista>Luis</analista>
 <programador>Andres</programador>
</grupoTrabajo>
```

**Tipos Abstractos:** se implementan definiendo un tipo abstracto y uno o mas tipos derivados. En un documento instancia se puede usar el elemento abstracto pero requiere el atributo `xsi:type=‘‘..."` para referenciar el tipo derivado que implementará.
El siguiente ejemplo (*esquema17.xsd*) declara el tipo abstracto `CTpublicacion`:

```
<!-- Elemento que implementa el tipo abstracto -->
<xsd:element name="publicacion" type="CTpublicacion"/>
<!-- declaracion del tipo abstracto -->
<xsd:complexType name="CTpublicacion" abstract="true">
 <xsd:sequence>
  <xsd:element name="titulo" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
<!-- declaracion del tipo derivado CTlibro -->
<xsd:complexType name="CTlibro">
 <xsd:complexContent>
  <xsd:extension base="CTpublicacion">
   <xsd:sequence>
    <xsd:element name="autor" type="xsd:string"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<!-- declaracion del tipo derivado CTrevista -->
<xsd:complexType name="CTrevista">
 <xsd:complexContent>
  <xsd:extension base="CTpublicacion">
   <xsd:sequence>
    <xsd:element name="numero" type="xsd:string"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

La declaración anterior permite el siguiente contenido (*instancia17.xml*):

```
<publicacion xsi:type="CTlibro">
 <titulo>El Hobbit</titulo>
 <autor>J.R.Tolkien</autor>
</publicacion>
<publicacion xsi:type="CTrevista">
 <titulo>PC-Magazine</titulo>
 <numero>Enero2006</numero>
</publicacion>
```

### 4.5.5. Unicidad y declaración de Claves

**Unicidad:** el elemento `xsd:unique` permite indicar que el valor de un atributo o elemento debe ser único dentro de un cierto contexto.

El siguiente ejemplo (*esquema19.xsd*) declara el atributo `rut` como único para elementos `vendedor` dentro del contexto del elemento `vendedores`:

```
<xsd:element name="vendedores">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="vendedor" type="CTvendedor" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:unique name="Urut">
  <xsd:selector xpath="vendedor"/>
  <xsd:field xpath="@rut"/>
 </xsd:unique>
</xsd:element>
<xsd:complexType name="CTvendedor">
 <xsd:sequence>
  <xsd:element name="nombre" type="xsd:string"/>
 </xsd:sequence>
 <xsd:attribute name="rut" type="xsd:integer"/>
</xsd:complexType>
```

Un instancia válida sería (*instancia19.xml*):

```
<vendedores>
 <vendedor rut="100">
  <nombre>Juan</nombre>
 </vendedor>
 <vendedor rut="101">
  <nombre>pedro</nombre>
 </vendedor>
</vendedores>
```

Observaciones:
- El elemento `xsd:selector` define el elemento que deseamos definir como "único".
- El elemento `xsd:field` define el atributo o subelemento que será usado para identificar al elemento.
- Es posible crear combinaciones de campos que deben ser únicos agregando elementos `xsd:field` para identificar los valores involucrados.

**Claves:** Una clave es un valor o conjunto de valores que identifican unívocamente un elemento. El elemento `xsd:key` permite definir claves haciendo referencia a un atributo, un elemento e incluso se permiten claves compuestas por una combinación de elementos y atributos. El valor de una clave debe ser único y no puede ser igualado a "nil".

**Referencias a Claves:** Cuando el valor de un atributo o elemento tiene el valor (referencia) de una clave, este elemento o atributo se denomina *clave externa*. El elemento `xsd:keyref` permite declarar clave externas. El declarar un elemento o atributo como clave externa no implica que su valor debe ser único, pero significa que debe existir como clave en algún lugar de la instancia.

Por ejemplo, dado el siguiente documento instancia (*instancia20.xml*):

```
<libreria>
 <catalogo>
  <producto codigo="100">
   <nombre>PC Magazine</nombre>
   <precio>400</precio>
  </producto>
  <producto codigo="101">
   <nombre>PC World</nombre>
   <precio>200</precio>
  </producto>
 </catalogo>
 <ventas>
  <venta>
   <codigoProducto>101</codigoProducto>
   <cantidad>5</cantidad>
  </venta>
 </ventas>
</libreria>
```

El siguiente esquema (*esquema20.xsd*), define que el atributo `codigo` es la clave de cada elemento `producto`, y que el atributo `codigoProducto` de un elemento `venta` es una clave externa que hace referencia a un código de producto. Ambas restricciones deben ser respetadas dentro del contexto del elemento `libreria`.

```
<xsd:element name="libreria">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="catalogo" type="CTcatalogo"/>
   <xsd:element name="ventas" type="CTVentas"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:key name="PKcodigoProducto">
  <xsd:selector xpath="catalogo/producto"/>
  <xsd:field xpath="@codigo"/>
 </xsd:key>
 <xsd:keyref name="FKcodigoProducto" refer="PKcodigoProducto">
  <xsd:selector xpath="ventas/venta"/>
  <xsd:field xpath="codigoProducto"/>
 </xsd:keyref>
</xsd:element>
<xsd:complexType name="CTcatalogo">
 <xsd:sequence>
  <xsd:element name="producto" maxOccurs="unbounded">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="nombre" type="xsd:string"/>
     <xsd:element name="precio" type="xsd:double"/>
    </xsd:sequence>
    <xsd:attribute name="codigo" type="xsd:string"/>
   </xsd:complexType>
  </xsd:element>
 </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="CTVentas">
 <xsd:sequence>
  <xsd:element name="venta" maxOccurs="unbounded">
   <xsd:complexType name="CTventa">
    <xsd:sequence>
     <xsd:element name="codigoProducto" type="xsd:string"/>
     <xsd:element name="cantidad" type="xsd:integer"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
 </xsd:sequence>
</xsd:complexType>
```

## 4.6. Reutilización de Esquemas

### 4.6.1. Inclusión de esquemas

El elemento `xsd:include` permite incluir definiciones y declaraciones desde otro esquema, el cual no debe definir un targetNamespace o este debe ser igual al targetNamespace del esquema principal.

Si un esquema incluido no define un targetNamespace, este adquiere el namespace del esquema que lo incluye. Esto se conoce como *efecto camaleón* y los componentes del esquema incluido se llaman *componentes camaleón*. Por ejemplo, el siguiente esquema (*esquema11a.xsd*).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="libro">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="titulo" type="xsd:string"/>
    <xsd:element name="autor" type="xsd:string"/>
    <xsd:element name="editorial" type="xsd:string" minOccurs="0"/>
   </xsd:sequence>
   <xsd:attribute name="codigo" type="xsd:string" use="required"/>
   <xsd:attribute name="stock" type="xsd:integer" use="optional"/>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

es incluido en el siguiente esquema principal (*esquema11.xsd*):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.libreria.org"
 xmlns:lib="http://www.libreria.org"
 elementFormDefault="qualified">
 <xsd:include schemaLocation="esquema11a.xsd"/>
 <xsd:element name="libreria">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="lib:libro" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

el cual permite validar la siguiente instancia (*instancia11.xml*):

```
<?xml version="1.0" encoding="UTF-8"?>
<libreria xmlns="http://www.libreria.org"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.libreria.org esquema11.xsd">
 <libro codigo="34674" stock="5">
  <titulo>El Hobbit</titulo>
  <autor>J.R.Tolkien</autor>
  <editorial>Minotauro</editorial>
 </libro>
 ...
</libreria>
```

### 4.6.2. Importación de esquemas

El elemento `xsd:import` permite importar componentes desde otro esquema con un targetNamespace distinto al del esquema importador.

Por ejemplo, el siguiente esquema (*esquema12a.xsd*).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.libros.org"
            xmlns:libros="http://www.libros.org">
 <xsd:element name="libro">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="titulo" type="xsd:string"/>
    <xsd:element name="autor" type="xsd:string"/>
    <xsd:element name="editorial" type="xsd:string" minOccurs="0"/>
   </xsd:sequence>
   <xsd:attribute name="codigo" type="xsd:string" use="required"/>
   <xsd:attribute name="stock" type="xsd:integer" use="optional"/>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

es importado en el siguiente esquema principal (*esquema12.xsd*):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.libreria.org"
            xmlns:libros="http://www.libros.org">
 <xsd:import schemaLocation="esquema12a.xsd" namespace="http://www.libros.org"/>
 <xsd:element name="libreria">
  <xsd:complexType>
   <xsd:sequence>
```

```
    <xsd:element ref="libros:libro" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

el cual permite validar la siguiente instancia (*instancia12.xml*)

```
<?xml version="1.0" encoding="UTF-8"?>
<lib:libreria xmlns:lib="http://www.libreria.org"
              xmlns:libros="http://www.libros.org"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://www.libreria.org esquema12.xsd">
 <libros:libro codigo="34674" stock="5">
  <titulo>El Hobbit</titulo>
  <autor>J.R.Tolkien</autor>
  <editorial>Minotauro</editorial>
 </libros:libro>
 ...
</lib:libreria>
```

Cuando `xsd:import` especifica el atributo `namespace`, este debe coincidir con el targetNamespace del esquema importado; los componentes importados mantienen su espacio de nombres. Cuando solo se especifica el atributo `schemaLocation`, el esquema a importar no debe definir un targetNamespace; los nuevos componentes no tendrán espacio de nombres.

### 4.6.3. Redefinición de esquemas

El elemento `xsd:redefine` es similar a `xsd:include` pero posibilita la redefinición de tipos complejos, tipos simples, grupos de elementos y grupos de atributos que se obtienen desde otros esquemas.

Por ejemplo, el tipo complejo "CTlibro"definido en el siguiente esquema (*esquema13a.xsd*).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:complexType name="CTlibro">
  <xsd:sequence>
   <xsd:element name="titulo" type="xsd:string"/>
   <xsd:element name="autor" type="xsd:string"/>
   <xsd:element name="editorial" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="codigo" type="xsd:string" use="required"/>
 </xsd:complexType>
</xsd:schema>
```

El esquema anterior es redefinido en el siguiente esquema (*esquema13.xsd*):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:redefine schemaLocation="esquema13a.xsd">
  <xsd:complexType name="CTlibro">
   <xsd:sequence>
    <xsd:element name="titulo" type="xsd:string"/>
    <xsd:element name="autor" type="xsd:string"/>
   </xsd:sequence>
   <xsd:attribute name="codigo" type="xsd:string" use="required"/>
   <xsd:attribute name="stock" type="xsd:integer" use="optional"/>
  </xsd:complexType>
 </xsd:redefine>
 <xsd:element name="libreria">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="libro" type="CTlibro" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

en el cual hemos eliminando el elemento "editorial", y a cambio hemos agregado el atributo "stock". Este ultimo esquema permite validar la siguiente instancia (*instancia13.xml*)

```
<?xml version="1.0" encoding="UTF-8"?>
<libreria xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.libreria.org esquema13.xsd">
 <libro codigo="34674" stock="5">
  <titulo>El Hobbit</titulo>
  <autor>J.R.Tolkien</autor>
 </libro>
 ...
</libreria>
```

Al igual que con *xsd:include*, los esquemas redefinidos deben tener el mismo targetNamespace que el esquema principal. El efecto Camaleón también es posible.

## 4.7. Desventajas de XML Schema

- No soporta entidades como un mecanismo para crear macros (los DTDs si soportan)
  <!ENTITY &texto; "Este texto se repite muchas veces">
- La especificación de restricciones es limitado. Ejemplo: Verificar que un valor total es correcto acorde a la suma de un conjunto de valores parciales.
- Sensibilidad al contexto limitada. Ejemplo: especificar que el contenido de un elemento depende del valor de un atributo.
- Tamaño de archivos XML Schema puede ser excesivo y complejo.
- La flexibilidad que presenta (existen varias opciones para declarar estructuras), hace necesaria la especificación de buenas practicas.
- Falta de Legibilidad en especificaciones complejas y poco modulares.

## 4.8. Buenas prácticas para el diseño de esquemas XML

### 4.8.1. Aspectos a considerar al diseñar esquemas XML

- *Modularidad:* Determinar como se dividirá la información a modelar (en términos de documentos y elementos). Esto influirá en la facilidad de lectura, escritura y procesamiento de los documentos.
- *Flexibilidad:* Diseñar esquemas flexibles en términos de la estructuración y división de los datos. Acá debe considerarse cuando modelar datos como elementos o atributos.
- *Extensibilidad:* Diseñar esquemas flexibles manejando adecuadamente los espacios de nombre y la reutilizacion de tipos simples y complejos.
- *Consistencia:* Evitar características incoherentes
- *Nivel de abstracción:* Buscar término medio en nivel de detalle

  ```
  <fecha>10 Marzo 2003</fecha>
  ```

  versus

  ```
  <fecha><día>10</dia><mes>Marzo</mes><año>2003</año></fecha>
  ```

### 4.8.2. Tipos complejos anónimos Vs. Tipos complejos con nombre

Considerar el siguiente ejemplo:

```
<!-- Tipo complejo anonimo -->
<xsd:element name="alumno">
 <xsd:complexType>
  ...
 </xsd:complexType>
</xsd:element>

<!-- Tipo complejo con nombre -->
<xsd:ComplexType name="TipoAlumno">
 ...
</xsd:ComplexType>
...
<xsd:element name="alumno" type="TipoAlumno"/>
```

- Si se busca legibilidad o el concepto no se reutilizara, entonces definir como tipo complejo anónimo.
- Si el concepto sera utilizado en distintas partes del esquema (o importado por otros esquemas), entonces definir como tipo complejo con nombre.
- Al definir tipos complejos anónimos se genera un anidamiento de definiciones conocido como *Diseño de Muñeca Rusa*.

### 4.8.3. ¿Atributo o SubElemento?

- *Atributos:* Información sobre el contenido (meta-información). Valores asociados con objetos sin identidad propia (ej. edad).
- *Subelementos:* Datos que se consideran contenido de otro elemento. valores con identidad propia (ej. fecha-nacimiento).

Por ejemplo, modelar `edad` como elemento o como atributo:

```
<persona>
 <nombre>Juan</nombre>
 <edad>23</edad>
</persona>
```

```
<persona edad="23">
 <nombre>Juan</nombre>
</persona>
```

# XML – Schema en dos patadas

Unai Estébanez
unai@unainet.net

## Motivación

Estos apuntes sobre Schema surgen de mi apremiante necesidad de validar una serie de documentos XML.

Tuve que programar un exportador de datos que transformaba datos de una base de datos a XML, estos XML debían mantenerse con un formato dado para que siguieran siendo compatibles con los que utilizaba una aplación ya existente.

El mayor problema con el que me encontré una vez finalizado el exportador fue el validar estos documentos (que no tenían en su diseño original ni un DTD ni un Schema). Encontré una solución interesante en los XML-Schemas, ahora bien, al comenzar a mirar por Internet sentí cierta frustración porque no me gustaban las explicaciones de la mayoría de los artículos, o bien eran muy "serias", rodeadas de una parafernalia indescifrable o bien eran demasiado simplones y repetían una y otra vez el mismo ejemplo estúpido.

Es por esto que he realizado estos apuntes con los siguientes objetivos:

- Explicar de forma sencilla que es Schema
- Dar un repaso breve a la elaboración de un Schema
- Crear un ejemplo para aplicar lo visto

### *Audiencia*

Este documento está dirigido a alguien que sabe lo que es un XML y que tiene un cierto bagaje en informática.

El perfil ideal es el de un programador que trabaja habitualmente con XML y nunca le han formado en este área o bien alguien que comienza a trabajar con XML por primera vez.

# ¿Que es XML-Schema?

Un esquema XML no es más que un documento XML que a su vez describe la estructura de un otro documento XML que es el que se quiere validar.

Los motivos para validar un XML son variados, por ejemplo:

- Asegurarnos de que la transmisión de datos entre dos puntos ha sido correcta en lineas con alta tasa de fallos.
- Asegurarnos de que una de las dos partes que se comunican mediante XML no tiene bugs y transmite documentos erróneos que pueden hacer fallar a la otra parte
- Asegurarnos de que las dos partes transmiten la información en el mismo formato, como por ejemplo las fechas.

En definitiva se trata de poder, dado un xml, decir si se sigue un acuerdo preestablecido(el esquema) o no.

Una vez escrito el esquema que describe el xml que queremos tratar bastará con enlazar cada xml que queramos validar al esquema y de esta forma se podrá detectar si este está o no bien formado.

# Enlazar un XML con un esquema

Fijémonos en el siguiente ejemplo:

```
<?xml version=”1.0” encoding=”UTF-8” standalone= “no”?>
<!DOCTYPE pedido >
<pedido xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
xsi:noNamespaceSchemaLocation=”u4.xsd” codigo=”A1234567” observaciones=”urgente”>
```

Ahora lo observaremos linea a linea:

`<?xml version=”1.0” encoding=”UTF-8” standalone= “no”?>`

Es la declaración normal de un XML, marcamos que necesita otro documento con standalone = “no”.

`<!DOCTYPE pedido >`

Tipo de documento en el que sólo incluimos el nombre del ejemplar

`<pedido xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance” xsi:noNamespaceSchemaLocation=”u4.xsd” codigo=”A1234567” observaciones=”urgente”>`

El ejemplar, aquí incluimos un namespace con prefijo xsi con las reglas de Schema y una vez que lo tenemos cargado lo usamos con `xsi:noNamespaceSchemaLocation=”u4.xsd”` para cargar nuestro esquema al que, en este caso, no le ponemos prefijo.
Finalmente agragmos los atributos que tuviera el ejemplar como en cualquier XML.

## Cabecera del xsd

Fijémonos en el siguiente ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Nuevamente lo observaremos linea a linea:
`<?xml version="1.0" encoding="UTF-8">` Es la declaración normal de un XML.

`<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">`
Tcreamos un namespace con prefijo xs. http://www.w3.org/2001/XMLSchema es un URI no URL, es decir, no tiene porque apuntar a nada, pero por convenio lo hacemos así
Además el prefijo usado en este caso es xs, también por convenio, aunque otros autores usan xsd

## Conceptos básicos

El esquema no es más que un xml como otro cualquiera y por lo tanto debe estar formado como tales. Tags, abiertos y cerrados, comentarios, elementos, atributos, etc.

El nodo padre del esquema se llamará **<schema>.**

Una vez abierto este tag <schema> dentro se especificará que es lo que debe contener todo documento que siga este esquema.

```
<schema>
"declaración de elementos del documento" ← Esto es lo que vamos a tener que "picar"
</schema>
```

En el esquema todo lo que vamos a hacer es definir los **elementos** que contiene el xml que sigue el esquema.

Los **elementos** serán **simples** o **complejos.**

## Elementos simples

Los elementos simples son los que solamente contienen texto, no contiene otros elementos hijos ni tampoco atributos.

Eso si, el texto puede ser cualquier tipo de datos definido en schema (Integer,string, fecha-hora,booleano,etc) o cualquier tipo definido por nosotros, es posible incluso añadir restricciones (llamadas facetas) u obligar a que se cumpla algún patrón mediante expresiones regulares.

Ejemplo:

```
<elemento> solo contengo texto </elemento>
```

Para definir un elemento usaremos un tag llamado, como no, "element", por ejemplo:

```
<xs:element name="apellidos" type="xs:string"/>
<xs:element name="edad" type="xs:integer"/>
<xs:element name="nacimiento" type="xs:date"/>
```

Hemos visto en el ejemplo como definir tres elementos simples y hemos comentado que se pueden establecer restricciones sobre los datos, veamos como.

## *Restricciones sobre los datos*

Podemos incluir los siguientes tipos de restricciones:

- Sobre los valores
- Sobre un conjunto de valores
- Sobre los espacios en blanco
- Sobre la longitud
- Sobre patrones

## Sobre los valores

Veamos como limitar la edad de un individuo para que valga de 0 a 100 años.

```
<xs:element name="edad">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

## Sobre un conjunto de valores

Podemos obligar a que un valor se establezca dentro de unos límites:

```
<xs:element name="color">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Rojo"/>
      <xs:enumeration value="Verde"/>
      <xs:enumeration value="Azul"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Si reescribimos lo anterior de la siguiente forma, podremos reaprovechar la definición de color RGB:

```
<xs:element name="color" type="colorRGB"/>

<xs:simpleType name="colorRGB">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Rojo"/>
    <xs:enumeration value="Verde"/>
    <xs:enumeration value="Azul"/>
  </xs:restriction>
</xs:simpleType>
```

## Sobre los espacios en blanco

Podemos jugar con los espacios en blanco para **mantenerlos intocables,restringirlos o eliminarlos.**

El siguiente ejemplo los mantiene intocables, esto significa que retornos de carro, tabuladores y espacios en blanco no serán alterados dentro del campo de tipo dirección:

```
<xs:element name="direccion">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="preserve"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Este otro ejemplo, haremos que se sustituyan todos los tabuladores,retornos de carro(CR) y fines de linea (LF) por espacios en blanco:

```
<xs:element name="direccion">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="replace"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Colapsando es posible hacer que múltiples espacios en blancos, retornos de carro, tabuladores y demás se colapsen a un solo espacio en blanco.

```
<xs:element name="direccion">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="collapse"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

## Sobre la longitud

Podemos especificar un mínimo o máximo de longitud, veamos dos ejemplos:

```
<xs:element name="apellido">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:length value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

```
<xs:element name="apellido">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="8"/>
      <xs:maxLength value="255"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Vemos como es posible jugar con las anchuras de estos campos fijándolos a un tamaño concreto o indicando márgenes.

Existen más restricciones sobre tipos de datos en concreto como los numéricos, en cualquier manual de schema podréis encontrarlos.

### Sobre patrones

Es posible también controlar mediante expresiones regulares los contenidos de un campo, esta es una forma muy potente de asegurarnos el contenido deseado en cada campo:

```
<xs:element name="mayusculas">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="([A-Z])*"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

En este ejemplo vemos como obligamos a que el elemento "mayusculas" tenga un contenido en mayúsculas mediante una expresión regular.

## Atributos

Hemos comentado que los tipos simples no pueden tener, entre otras cosas, atributos. Ahora bien, los atributos, que serán utilizados en los tipos complejos, son declarados en si mismos como un tipo simple.

La forma general de declarar un atributo es como sigue:

```
<xs:attribute name="xxx" type="yyy"/>
```

Donde xxx es el nombre del atributo y yyy será un tipo de datos de los nativos de schema o de los definidos por nosotros, veamos como definir un atributo "Alias" para usarlo dentro de un elemento "nombre":

```
<xs:attribute name="alias" type="xs:string"/>
<!-- Ahora podemos usarlo dentro de un elemento nombre -->
<nombre alias="El cadenas">Iker</nombre>
```

## Elementos complejos

Un elemento complejo es aquel que contenga otro elemento y/o atributos.

Hay cuatro tipos de elementos complejos:

- Elementos que contienen solo otros elementos
- Elementos vacios
- Elementos que contienen solo texto
- Elementos que contienen otros elementos y texto

En estas cuatro categorías además se pueden incluir atributos en los elementos.

### *Elementos que contienen solo otros elementos*

Veamos como definir un tipo complejo (elemento que contiene solo otros elementos):

```
<xs:element name="alumno" type="persona"/>

<xs:complexType name="persona">
  <xs:sequence>
    <xs:element name="apellido" type="xs:string"/>
    <xs:element name="nombre" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Esto se traduce en poder utilizar elementos como:

```
<persona>
      <apellido>Estébanez</apellido>
      <nombre>Unai</nombre>
</persona>
```

Hemos usado en esta definición un **indicador** secuencia (sequence) que explicaremos más adelante.

**Se puede apreciar en el ejemplo como esto es similar a definir objetos en un lenguaje de programación. Además es posible "heredar"**, veamos como:

```
<xs:element name="destinatario" type="direcion"/>

<xs:complexType name="persona">
  <xs:sequence>
    <xs:element name="apellido" type="xs:string"/>
    <xs:element name="nombre" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="direccion">
  <xs:complexContent>
    <xs:extension base="persona">
      <xs:sequence>
        <xs:element name="calle" type="xs:string"/>
        <xs:element name="ciudad" type="xs:string"/>
        <xs:element name="pais" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

## *Elementos vacios*

Veamos como definirlos:

```
<xs:element name="nombre" type="nombre_t"/>

<xs:complexType name="nombre_t">
  <xs:attribute name="alias" type="xs:string"/>
</xs:complexType>
```

Esto nos permitiría definir elementos como estos:

<nombre alias="Capitan Morgan"/>

## *Elementos que contienen solo texto*

Es posible definir elemento que solo contengan texto, lo podemos hacer por ejemplo utilizando un tipo base de schema y ampliandolo:

```
<xs:element name="Pie">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:integer">
        <xs:attribute name="pais" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<pie pais="España">41</pie>
```

### *Elementos que contienen otros elementos y texto*

Supongamos que queremos validar XMLs con este formato:

```
<carta>
  Querido Señor:<nombre>Alex Conceiro</nombre>.
  Su pedido <pedido>1032</pedido>
  será entregado el <fechaEntrega>2001-07-13</fechaEntrega>.
</carta>
```

Definiriamos un schema como el siguiente,ojo al atributo mixed, es necesario:

```
<xs:element name="carta" type="tipoCarta"/>

<xs:complexType name="tipoCarta" mixed="true">
  <xs:sequence>
    <xs:element name="nombre" type="xs:string"/>
    <xs:element name="pedido" type="xs:positiveInteger"/>
    <xs:element name="fechaEntrega" type="xs:date"/>
  </xs:sequence>
</xs:complexType>
```

## Indicadores

Sirven para controlar como utilizar los elementos en los documentos

Hay siete tipos:

- Indicadores de orden
  - All
  - Choice
  - Sequence
- Indicadores de frecuencia de aparición
  - maxOccurs
  - minOccurs
- Indicadores de grupo
  - Grupos de elementos
  - Grupos de atributos

### *All*

El indicador **all** especifica que los elementos hijos pueden aparecer en cualquier orden y que aparecen como mucho solo una vez

```
<xs:element name="persona">
  <xs:complexType>
    <xs:all>
      <xs:element name="nombre" type="xs:string"/>
      <xs:element name="apellido" type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

## *Choice*

Indica que podran aparecer como elementos hijos uno u otro de los definidos, en este caso una persona puede ser trabajadora o miembro de un club por ejemplo:

```
<xs:element name="persona">
  <xs:complexType>
    <xs:choice>
      <xs:element name="empleado" type="empleado"/>
      <xs:element name="miembro" type="miembro"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

## *Sequence*

Se indica así en que orden exacto deben aparecer los hijos y que deben aparecer todos una vez.

```
<xs:element name="persona">
   <xs:complexType>
    <xs:sequence>
      <xs:element name="nombre" type="xs:string"/>
      <xs:element name="apellido" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

## *maxOccurs y minOccurs*

Indica respectivamente el máximo y mínimo número de veces que aparecerá un elemento, en este caso una persona con un nombre dado puede tener desde 0 hasta 10 nombres de hijos asociados.

```
<xs:element name="persona">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="nombre" type="xs:string"/>
      <xs:element name="nombreHijos" type="xs:string"
      maxOccurs="10" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

## *Indicadores de grupo y de atributo*

Sirven para agrupar elementos o atributos bajo una misma denominación:

```
<xs:group name="grupoPersona">
  <xs:sequence>
    <xs:element name="nombre" type="xs:string"/>
    <xs:element name="apellido" type="xs:string"/>
    <xs:element name="cumpleaños" type="xs:date"/>
  </xs:sequence>
</xs:group>
```

Ahora es posible incluir este grupo dentro de otra definición, observesé el atributo **href** para hacer referencia a una definición dada:

```
<xs:element name="persona" type="infoPersona"/>

<xs:complexType name="infoPersona">
  <xs:sequence>
    <xs:group ref="grupoPersona"/>
    <xs:element name="pais" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Podemos hacer algo similar pero agrupando atributos

```
<xs:attributeGroup name="grupoAttrPersona">
  <xs:attribute name="nombre" type="xs:string"/>
  <xs:attribute name="apellidos" type="xs:string"/>
  <xs:attribute name="cumpleaños" type="xs:date"/>
</xs:attributeGroup>

<xs:element name="persona">
  <xs:complexType>
    <xs:attributeGroup ref="grupoAttrPersona"/>
  </xs:complexType>
</xs:element>
```

De esta forma se pueden reaprovechar muchas definiciones.

## Namespaces

Los namespaces (espacios de nombres) son una forma de poder asociar los nombres de elementos que definimos a un nombre (el namespace) único para evitar duplicidad de nombres.

Me explicaré mejor, supongamos que definimos un grupo de atributos para definir tipos de dia (festivo,laboral,etc). Este grupo se define de la siguiente forma

```
<xs:attributeGroup name="DayType_attr">
      <xs:attribute name="Name" type="xs:string" use="required"/>
      <xs:attribute name="Abr" type="xs:string" use="required"/>
      <xs:attribute name="Type" type="xs:integer" use="required"/>
</xs:attributeGroup>
```

De esta forma podemos usar esta definición por ejemplo de la siguiente manera:

```
<DayType Name="Sabado" Abbr="SABA" Type="2"/>
```

Esto nos permitiría definir que el Sábado es de tipo “2” y nosotros en otra parte definiremos que significa que un dia sea de tipo “2”.

El caso es que supongamos que otra persona define en otro esquema un grupo de atributos y también le llama DayType_attr y nosotros incluimos ese esquema dentro del nuestro. ¿Como diferenciar cuando queremos usar uno u otro?, la solución son los namespaces.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.unainet.net"
xmlns:tns="http://www.unainet.net"
elementFormDefault="qualified">

...

</xs:schema>
```

Esto significa lo siguiente:

xmlns:xs=”http://www.w3.org/2001/XMLSchema"

Usamos un namespace xml (xmlsn), cuyo prefijo será “xs” y cuya nombre es el sigueinte URI “http://www.w3.org/2001/XMLSchema”.

Esto NO indica que haya un esquema en esa dirección ni nada por el estilo,simplemente se usan URIs para definir namespaces para asegurar de que no colisionan nombres de namespaces.

Ahora podemos usar los elementos del w3 consortium prefijandolos con “xs”, tal y como se puede ver en el ejemplo anterior.

Por otro lado tenemos el atributo **targetNamespace** que nos indica que los elementos definidos en este esquema pertenecerán a un namespace llamado “http://www.unainet.net” y ademas mediante el atributo **xmlns:tns="http://www.unainet.net"** indicamos que el

namesapce http://www.unainet.net se prefija con "tns".

Por último el atributo **elementFormDefault="qualified"** nos obliga a prefijar todos los elementos que usemos en el esquema.

## Reaprovechar definiciones usando includes

Supongamos que queremos reaprovechar definiciones de elementos comunes a varios esquemas para no reescribir una y otra vez estas.

Se puede hacer un esquema coumun y enlazar con este desde el resto de esquemas.

Esto se hace mediante un **include**.

Veamos un ejemplo en el que reaprovechamos un elemento que consiste en un tipo de datos carácter mayúsucla. Metemos esta definicion en un fichero common.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tlv="http://www.telvent.com"
targetNamespace="http://www.telvent.com" elementFormDefault="qualified">
<xs:simpleType name="SingleUpperCaseChar">
      <xs:restriction base="xs:string">
        <xs:pattern value="[A-Z]"/>
      </xs:restriction>
</xs:simpleType>
</xs:schema>
```

Ahora queremos reaprovecharla desde otro esquema, pongamos uno que se llama lin.xsd,fijaos como se utiliza el elemento **include**:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tlv="http://www.telvent.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" targetNamespace="http://www.telvent.com"
elementFormDefault="qualified" attributeFormDefault="unqualified">
      <xs:include schemaLocation="common.xsd"/>
</xs:schema>
```

Ahora es posible utilizar SingleUpperCaseChar y cualquier otra definición que hayamos hecho en common.xsd