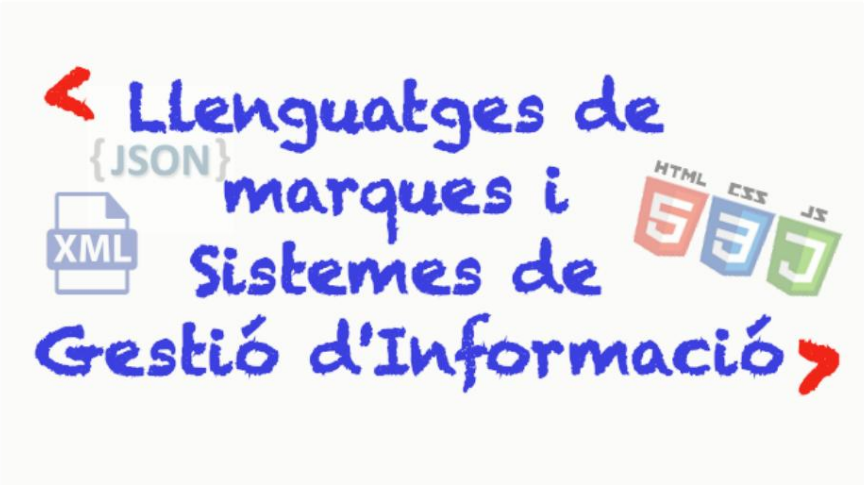
UNIDAD 5. DEFINICIÓN DE ESQUEMAS I VOCABULARIOS EN LENGUAJES DE MARCAS



Contenido

1. Introducción

XML permite crear los lenguajes de marcas que queramos, cualquiera que sea el campo de actuación. Éste es el punto fuerte del XML sobre otros lenguajes de marcas: se adapta a lo que se quiera representar sin importar la complejidad que pueda tener.

Pero los datos de los documentos XML normalmente tendrán que ser procesados por un programa de ordenador, y los programas de ordenador no dan tanta libertad como el XML. Éstos no son tan buenos interpretando y entendiendo información para la que no han sido programados para procesar.

Supongamos que se ha desarrollado un programa para representar imágenes a partir de las etiquetas <dibujo>, <rectángulo>, <círculo>. Desde un punto de vista de la libertad que deja el XML no habrá ningún problema para crear un archivo XML como éste:

```
<dibujo>
        <rectángulo>12,10,14,10</rectángulo>
        <linia>13,13,25,25</linia>
</dibujo>
```

El documento es perfectamente correcto desde un punto de vista del XML pero el programa no sabrá qué hacer con la etiqueta <línea> porque nadie lo ha programado. Es por este motivo que los programas normalmente se diseñan para procesar sólo tipos concretos de XML.

Por tanto, una de las cosas que deberá hacer un programa es comprobar que los datos del documento XML sean correctos. Dado que esta tarea es muy compleja se han definido sistemas para comprobar que el documento XML entrado contiene las etiquetas que debe contener y que están colocadas tal y como hace falta. El proceso de realizar estas comprobaciones se llama validación.

1.1. Validación de documentos

El proceso de comprobar que determinados archivos XML siguen un determinado vocabulario se denomina validación y los documentos XML que siguen las reglas del vocabulario se denominan documentos válidos. Hay que tener clara la diferencia entre lo que es un documento bien formado y un documento válido.

Un documento está bien formado cuando sigue las reglas de XML.

Un documento es válido si sigue las normas del vocabulario asociado.

Un documento puede estar bien formado pero no válido, y en cambio si es válido seguro que está bien formado.

La validación es el proceso de comprobar que un documento cumple con las reglas del vocabulario en todos sus aspectos.

La validación permite asegurar que la información está exactamente en la forma en que debe estar. Esto es especialmente importante cuando se comparte información entre sistemas, puesto que validar el documento es asegurarse de que realmente la estructura de la información que se está pasando es la correcta. Si se quiere hacer que dos programas situados en ordenadores diferentes colaboren es necesario que la información que se pasan uno a otro siga la estructura prefijada para enviarse mensajes.

Para forzar una determinada estructura en un documento hace falta alguna forma de definir aspectos como:

- ¿En qué orden deben ir las etiquetas
- ¿Cuáles son correctas y cuáles no
- ¿En qué orden deben aparecer
- Qué atributos se pueden poner
- Qué contenido puede haber

El XML hace esto mediante lenguajes de definición de vocabularios o lenguajes de esquemas.

Los lenguajes de definición de esquemas surgen por la necesidad de que los documentos XML sean procesados por programas (para extraer información, transformarlos en otras cosas, etc), ya que los programas no son tan flexibles como el XML y necesitan procesar datos con estructuras de documento cerradas.

2. DTD

El DTD (document type definitions) es un lenguaje de definición de esquemas que ya existía antes de la aparición de XML (se utilizaba en SGML). Al estar pensado para funcionar con SGML se podía utilizar en muchos de los lenguajes de marcas que se han basado en ellos, como XML o HTML.

Cuando se definió el XML se aprovechó para realizar una versión simplificada de DTD que fuera el lenguaje de especificación de esquemas original. Una ventaja asociada era que utilizar una versión de DTD permitiría mantener la compatibilidad con SGML, y por tanto se referenciaron las DTD en la especificación de XML (http://www.w3.org/TR/REC-xml/).

El principal objetivo de las DTD es proveer un mecanismo para validar las estructuras de los documentos XML y determinar si el documento es válido o no. Pero esta no será la única ventaja que nos aportarán las DTD, sino que también podremos utilizarlas para compartir información entre organizaciones, ya que si alguien tiene nuestra DTD nos puede enviar información en nuestro formato y con el programa que hemos hecho la podremos
procesar.

Durante mucho tiempo las DTD fueron el sistema de definir vocabularios más usados en XML pero actualmente han sido superados por XML Schema. Sin embargo todavía es muy usado, sobre todo porque es mucho más sencillo.

2.1. Asociar una DTD a un documento XML

Para poder validar un documento XML es necesario especificarle cuál es el documento de esquemas que se utilizará.
Si el lenguaje que se utiliza es DTD existen varias maneras de especificarlo pero en general siempre implicará añadir una declaración DOCTYPE dentro del documento XML para validar.

Lo habitual es añadir la referencia a la DTD dentro del documento XML mediante la etiqueta especial <!DOCTYPE. Debido a que la declaración XML es opcional pero si la tiene que ser la primera cosa que aparezca
en un documento XML, la etiqueta DOCTYPE deberá ir siempre detrás de él.

Por tanto, sería incorrecto hacer:

```
<!DOCTYPE ...        >
<?xml version="1.0"?>
```

Pero, por otra parte, la etiqueta no puede aparecer en ningún otro lugar que no sea justo a continuación de la declaración XML, y por tanto tampoco se puede incluir la etiqueta DOCTYPE una vez comenzado el documento.

```
<?xml version="1.0"?>
<documento>
<!DOCTYPE ...>
</documento>
```

Ni tampoco al final:

```
<?xml version="1.0" ?>
<documento>
</documento>
<!DOCTYPE ... >
```

La etiqueta DOCTYPE siempre debe ir o bien en la primera línea si no hay declaración XML, o justo detrás:

```
<?xml version="1.0" ?>
<!DOCTYPE ...      >
<documento>
</documento>
```

Hay dos formas de incorporar DTD en un documento XML:

- Declaración interna
- Declaración externa

2.1.1. Declaración DTD interna

En las declaraciones DTD internas, las reglas de la DTD están incorporadas en el documento XML.

Ejemplo:

```
<?xml version="1.0"?>
<!DOCTYPE clase [
        <!ELEMENT clase (profesor, alumnos)>
        <!ELEMENT profesor (#PCDATA)>
        <!ELEMENTO alumnos (nombre*)>
        <!ELEMENT nombre (#PCDATA)>
]>
<clase>
        <profesor>Marcel Puig</profesor>
        <alumnos>
                <nom>Frededic Pi</nom>
        </alumnos>
</clase>
```

2.1.2. Declaración DTD externa

En lugar de especificar la DTD en el mismo documento se considera una práctica mucho mejor definirla en un archivo aparte.

Para realizar una declaración externa también se utiliza la etiqueta DOCTYPE pero el formato es ligeramente diferente y prevé dos posibilidades:

- DTD privada • DTD pública

DTD privadas

Las definiciones de DTD privadas son las más corrientes, ya que, aunque se define el vocabulario como privado, nada limita que el archivo se pueda compartir o se publique a través de Internet.

La definición privada queda de la siguiente forma:

```
<!DOCTYPE elemento-raíz SYSTEM "URI">
```

Donde URI (uniform resource identifier) es una cadena de caracteres que se utiliza para hacer referencia única a un recurso local, dentro de una red o Internet.

DTD públicas

Las definiciones PUBLIC están reservadas para DTD que estén definidas por organismos de estandarización, ya sean oficiales o no. En la definición de una DTD pública se añade un campo extra que hace de identificador del organismo.

La definición pública queda de la siguiente forma:

<!DOCTYPE elemento-raíz PUBLIC "identificador-público" "URI">

2.1.3. Definición de esquemas con DTD

La parte más importante de un sistema de definición de vocabularios es definir cómo se hace para determinar el orden en el que los elementos pueden aparecer y qué contenido pueden tener, qué atributos pueden tener las etiquetas, etc.

Esto, en DTD, se hace definiendo una serie de reglas por medio de un sistema de etiquetas predefinido. A diferencia de lo que ocurre en XML, al hacer una definición en DTD las etiquetas están definidas. Para definir elementos y atributos se utilizarán etiquetas concretas.

Elementos

<!ELEMENT nombre (contenido)>

Contenidos genéricos

Valor	Significado
AÑO	El contenido del elemento puede ser cualquier cosa.
EMPTY	El elemento no tiene contenido.
#PCDATA	El contenido de la etiqueta puede ser datos.

Los elementos que estén definidos con AÑO pueden contener cualquier cosa en su interior. Tanto etiquetas, como datos, o incluso una mezcla de ambas cosas.

Si se define persona de esta forma:

```
<!ELEMENT persona AÑO>
```

validará tanto elementos que contengan datos:

```
<persona>Frederic Pi</persona>
```

como elementos que contengan otros elementos:

```
<persona>
    <nom>Frederic</nom>
    <apellido>Pi</apellido>
</persona>
```

Los contenidos definidos con EMPTY están pensados para las etiquetas que no tendrán ningún contenido en su interior. Por tanto, el elemento <profesor> del ejemplo anterior se definiría de esta manera:

```
<!ELEMENT profesor EMPTY>
```

Esta definición sirve tanto para los elementos que se definen utilizando el sistema de una sola etiqueta…

```
<profesor/>
```

como para quienes definen las etiquetas de apertura y cierre.

```
<profesor></profesor>
```

El contenido genérico #PCDATA (parser character data) seguramente es el más usado para marcar que una etiqueta sólo tiene datos en su contenido.

Si partimos del siguiente ejemplo:

```
<persona>
    <nom>Marcel</nom>
    <apellido>Puigdevall</nom>
</persona>
```

Se puede utilizar #PCDATA para definir el contenido de los elementos <nombre> y <apellido> porque en su interior sólo tienen datos.

```
<!ELEMENT nombre (#PCDATA)>
<!ELEMENT apellido (#PCDATA)>
```

Pero no se puede utilizar, en cambio, para definir el elemento <persona>, porque en su interior no tiene sólo datos sino que tiene los elementos <nombre> y <apellido>.

Contenido de elementos

Una de las situaciones habituales en un documento XML es que un elemento dentro de su contenido tenga otros.

Tenemos varias posibilidades para definir el contenido de elementos dentro de una DTD:

- Secuencias de elementos
- Alternativas de elementos
- Modificadores

Si miramos el siguiente ejemplo veremos que tenemos un elemento <persona> que contiene dos elementos, <nombre> y <apellido>.

```
<persona>
    <nom>Pere</nom>
    <apellido>Martinez</apellido>
</persona>
```

Por tanto, el elemento <persona> es un elemento que tiene como contenido una secuencia de otros elementos. En una DTD se define esta situación definiendo explícitamente cuáles son los hijos del elemento, separándolos por comas.

```
<!ELEMENT persona (nombre,apellido)>
```

Nunca debe olvidarse que las secuencias tienen un orden explícito y, por tanto, sólo validarán si el orden en el que se definen los elementos es idéntico al orden en el que aparecerán en el documento XML.

Por tanto, si tomamos como referencia el siguiente documento:

```
<comida>
    <comida>Arroz</comida>
    <cena>Cena</cena>
</comida>
```

Si el elemento <comida> se define con la secuencia (<comida>,<cena>), éste validará, ya que los elementos que contiene están en el mismo orden que en el documento.

```
<!ELEMENT comida (almuerzo, cena)>
```

Pero no validaría, en cambio, si definiéramos la secuencia al revés (<cena>,<almuerzo>) porque el procesador esperará a que llegue primero una <cena> y se encontrará un <almuerzo>.

```
<!ELEMENT comida (cena,comida)
```

A veces, al crear documentos XML, sucede que en función del contenido que haya en el documento puede que tengan que aparecer unas etiquetas u otras.

Si diseñáramos un XML para definir las fichas de personal de una empresa podríamos tener una etiqueta <personal> y después definir el cargo con otra etiqueta. El presidente podría definirse así:

```
<personal>
    <president>Josep Maria Flaviol</president>
</personal>
```

Y uno de los trabajadores así:

```
<personal>
    <trabajador>Pere Vila</trabajador>
</personal>
```

Podemos hacer que una DTD valide ambos casos utilizando el operador de alternativa |.

El operador alternativa | permite que el procesador pueda elegir entre una de las opciones que se le ofrecen para validar un documento XML.

Por tanto, podemos validar los dos documentos XML anteriores definiendo <personal> de esta manera:

```
<!ELEMENTO personal (trabajador|presidente)>
```

No existe ninguna limitación definida para poner alternativas. Por tanto, podemos poner tantas como nos hagan falta.

```
<!ELEMENTO personal (trabajador|presidente|informático|gerente)>
```

También se permite mezclar la definición con EMPTY para indicar que el valor puede aparecer o no:

```
<!ELEMENTO alumno (delegado|EMPTY)>
```

Combinar alternativas con PCDATA es más complejo. Véase el apartado “Problemas al mezclar etiquetas y #PCDATA”.

Nada impide mezclar las secuencias y alternativas de elementos a la hora de definir un elemento con DTD. Por tanto, se pueden hacer las combinaciones que hagan falta entre secuencias y alternativas.

Si tenemos un XML que define la etiqueta <círculo> a partir de sus coordenadas del centro y del radio o diámetro podemos definir el elemento combinando las secuencias con una alternativa. La etiqueta <círculo> contendrá siempre en su interior las etiquetas <x>,<y> y después puede contener también <radio> o <diámetro>:

```
<!ELEMENTO círculo (x,y,(radio|diámetro))>
```

Combinar secuencias y alternativas permite saltarse las restricciones de orden de las secuencias. El siguiente ejemplo nos permite definir a una persona a partir de nombre y apellido en cualquier orden:

```
<!ELEMENT persona ((apellido,nombre)|(nombre,apellido))>
```

En los casos en que las etiquetas se repitan un número indeterminado en ocasiones será imposible especificarlas todas en la definición del elemento. Los modificadores sirven para especificar cuántas instancias de los elementos hijos puede haber en un elemento

Modificador	Significado
?	Indica que el elemento tanto puede que esté como no.
+	Se utiliza para indicar que el elemento debe salir una vez o más.
*	Indica que puede que el elemento esté repetido un número indeterminado a veces, o bueno no estar allí.

Por tanto, si queremos especificar que una persona puede ser identificada por medio del nombre y uno o más apellidos podríamos definir el elemento <persona> de esta manera:

```
<!ELEMENT persona (nombre,apellido+)>
```

Dado que esta expresión indica que apellido debe salir una vez, podrá validar tanto este ejemplo:

```
<persona>
    <nom>Joan</nom>
    <apellido>Puig</apellido>
</persona>
```

como este otro:

```
<persona>
    <nom>Joan</nom>
    <apellido>Puig</apellido>
    <apellido>Garcia</apellido>
</persona>
```

Es evidente que ésta no es la mejor manera de definir lo que queríamos, ya que así definido también nos aceptará personas con 3 apellidos, 4 apellidos, o más.

Por tanto, ¿el modificador ideal para forzar que sólo pueda haber uno o dos apellidos sería ?, utilizándolo de la siguiente manera:

```
<!ELEMENT persona (nombre, apellido, apellido?)>
```

El modificador * aplicado en el mismo ejemplo nos permitiría aceptar que alguna persona no tuviera apellidos o que tenga un número indeterminado:

```
<!ELEMENT persona (nombre, apellido*)>
```

Los modificadores aplicados detrás de un paréntesis implican aplicarlos a todos los elementos de dentro de los paréntesis:

```
<!ELEMENTE escritor ((libro,fecha)*|artículos+) >
```

Contenido mezclado

El contenido mezclado se pensó para poder definir contenidos que contengan texto que se va intercalando con otros elementos, como por ejemplo:

```
<carta>Querido <empresa>Ferros Puig</empresa>:

  SR. <director>Manel Garcia</director>, le envío esta carta para comunicarle que le hemos enviado su pedido <comando>145</comando> a la dirección que nos proporcionó.


    Atentamente, <empresa>Ferreteria, SA</empresa>

</carta>
```

El contenido mezclado se define definiendo el tipo #PCDATA (siempre en primer lugar) y después se añaden los elementos con la ayuda del operador de alternativa, |, y termina todo el grupo con el modificador *.

```
<!ELEMENT carta (#PCDATA|empresa|director|pedido)*>
```

Se debe tener en cuenta que este contenido sólo sirve para controlar que los elementos puedan estar ahí pero que no hay ninguna manera de controlar en qué orden aparecerán los diferentes elementos.

Por tanto, podemos definir una DTD que valide el ejemplo presentado al principio de este apartado de la siguiente manera:

```
<!ELEMENT carta (#PCDATA|empresa|director )*>

<!ELEMENT empresa (#PCDATA)>

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

Atributos en DTDs

En las DTD deben especificarse cuáles son los atributos que se utilizarán en cada una de las etiquetas explícitamente. La declaración de atributos se realiza con la etiqueta especial ATTLIST.

Los dos primeros valores de la definición del atributo son el nombre de la etiqueta y el nombre del atributo, puesto que al definir un atributo siempre se especifica a qué etiqueta pertenece. Una de las críticas que se han hecho a las DTD ha sido que no se pueden realizar atributos genéricos. Si alguien desea definir un atributo que sea compartido por todos los elementos de su vocabulario debe ir especificando el atributo para todos y cada uno de los elementos.

Para definir un atributo llamado nombre que pertenezca al elemento <persona> podemos hacerlo de esta manera:

```
<!ATTLIST persona nombre CDATA #IMPLIED>
```

Especificar múltiples atributos

Si un elemento necesita varios atributos deberemos especificar todos los atributos en varias líneas ATTLIST.
Por ejemplo, definimos los atributos nombre y apellidos del elemento <persona> de este modo:

```
<!ATTLIST persona nombre CDATA #REQUIRED>

<!ATTLIST persona apellido CDATA #REQUIRED>
```

O bien se puede hacer la definición de ambos atributos con una sola referencia ATTLIST:

```
<!ATTLIST persona nombre        CDATA #REQUIRED

                                apellido CDATA #REQUIRED>
```

Las dos declaraciones anteriores nos permitirían validar los atributos del elemento <persona> de este ejemplo:

```
<persona nombre="Frederic" apellido="Pi" />
```

Atributos de ATTLIST

Los elementos ATTLIST aparte de tipos de datos también pueden tener atributos que permiten definir características sobre los atributos.

Atributo	Significado
#IMPLIED	El atributo es opcional. Los elementos pueden tenerlo o no.
#REQUIRED	El atributo es obligatorio. El elemento debe tenerlo definido o no validará.
#FIXED	Se utiliza para definir atributos que tienen valores constantes e inmutables. Se debe especificar, ya que es permanente.
#DEFAULT	Permite especificar valores por defecto en los atributos.

Por tanto, si se define un atributo de esta manera:

```
<!ATTLIST equipo posicion ID #REQUIRED>
```

se está obligando a que cuando se defina el elemento <equipo> en un documento XML se especifique obligatoriamente el atributo posicion y que además su valor no se repita en el documento, ya que es de tipo ID.

Sin embargo, definiendo el atributo DNI de <persona> con #IMPLIED se permitirá que al crear el documento XML el atributo DNI pueda estar o no.

```
<!ATTLIST persona dni NMTOKEN #IMPLIED>
```

Al definir un atributo como #FIXED o como #DEFAULT debe especificarse el valor. Este valor se especifica a continuación del atributo y debe ir entre comillas:

```
<!ATTLIST documento versión CDATA #FIXED "1.0">

<!ATTLIST documento codificacion NMTOKEN #DEFAULT "UTF-8">
```

Los atributos de tipo #FIXED deben tener el valor especificado en la definición y éste no se puede cambiar, mientras que los valores definidos con #DEFAULT sí pueden ser cambiados.

2.2. Limitaciones DTD

Una de las críticas que se ha realizado al uso de DTD para definir lenguajes XML es que no está basado en XML.
La DTD no sigue la forma de definir los documentos de XML y, por tanto, para utilizarlo, es necesario aprender un nuevo lenguaje. Si la DTD estuviera basada en XML no debería conocerse de qué manera se deben definir los elementos, marcar las repeticiones, los elementos vacíos, etc., ya que se haría con elementos.

Sin embargo, el hecho de que DTD no sea un lenguaje XML es un problema menor si se tienen en cuenta las otras limitaciones que tiene:

- No comprueba los tipos
- Presenta problemas al mezclar etiquetas y #PCDATA
- Sólo acepta expresiones deterministas

La DTD no comprueba los tipos

Uno de los problemas más importantes que nos encontraremos a la hora de usar DTD para definir nuestro vocabulario es que no tiene ninguna forma de comprobar los tipos de datos que contienen los elementos. A menudo los nombres de los elementos ya determinan que el contenido que habrá será de un tipo determinado (un número, una cadena de caracteres, una fecha, etc.) pero la DTD no deja que se le especifique qué tipo de datos se quiere poner.

Por tanto, si alguien rellena con cualquier cosa un elemento que se llame <día>, el documento será válido a pesar de que el contenido no sea una fecha:

```
<día>chocolate</día>
```

Al no poder comprobar el tipo del contenido, una limitación importante añadida es que no existe ninguna manera de poner restricciones. Por ejemplo, no podemos definir que queremos que una fecha esté entre los años 1900 y 2012.

Problemas al mezclar etiquetas y #PCDATA

Una limitación más compleja de ver es que no se pueden mezclar etiquetas y #PCDATA en expresiones si el resultado no es lo que se conoce como contenido mezclado.

Un ejemplo de ello consistiría en realizar una definición de un ejercicio como un enunciado de texto, pero que pueda haber varios apartados que también contengan texto.

```
<ejercicio>

    Lea el texto "Validación de documentos XML" y responda a las siguientes preguntas:


    <apartado numero="1">¿Qué quieren decir las siglas XML?</apartado>

    <apartado numero="2">¿Qué es un DTD?</apartado>

</ejercicio>
```

Lo sencillo sería mezclar un #PCDATA y la etiqueta <apartado>, pero es incorrecto:

```
<!ELEMENTE ejercicio (#PCDATA | apartado*)>
```

Además, no se permite que se hagan declaraciones duplicadas de elementos, o sea que tampoco podemos arreglarlo con:

```
<!ELEMENTE ejercicio(#PCDATA)>

<!ELEMENTE ejercicio(apartado*)>
```

La única forma de combinarlo sería utilizar la fórmula del contenido mezclado:

```
<!ELEMENTE ejercicio(#PCDATA|apartado)*>
```

Sólo acepta expresiones deterministas

Otra de las limitaciones de las DTD es que obliga a que las expresiones deban ser siempre deterministas.

Si miramos un documento DTD:

<!ELEMENT clase(profesor|alumnos)>

<!ELEMENT profesor (nombre,apellidos)>

<!ELEMENTO alumnos (alumno*) >

<!ELEMENTO alumno (nombre,apellido) >

<!ELEMENT nombre (#PCDATA)>

<!ELEMENT apellido (#PCDATA)>

Cuando se analiza un archivo XML se define cuál es la raíz de la DTD. Si consideramos que la raíz es el elemento <clase>, el proceso de validación empezará en la primera línea que nos dice que para que el documento sea válido después de la raíz debe haber un elemento <profesor> o bien uno elemento <alumnos>. Si lo que llega es un <profesor> el procedimiento de validación pasará a evaluar la segunda y si llega un <alumno> pasará a la tercera. Si seguimos con la evaluación veremos que realmente el validador siempre que se encuentre con una alternativa acabará yendo a una sola expresión. Esto se debe a que la DTD analizada es determinista.

Lo mismo podemos hacerlo con una expresión más compleja que contenga diferentes elementos dentro de la alternativa. El validador debe poder elegir, al leer el primer elemento, con cuál de las alternativas debe quedarse. Si me llega un elemento <nombre> me quedo con la alternativa de la izquierda, nombre, apellidos, y si me llega un <alias> me quedo con la de la derecha, alias, nombre, apellidos.

<!ELEMENT persona(nombre,apellido|alias,nombre,apellido)>

Si en algún momento alguien crea un documento que no sea determinista, la validación fallará. Esto ocurrirá si en algún momento el validador se encuentra que no puede decidir cuál es la alternativa correcta.

<!ELEMENTO terrestre(persona, hombre | persona, mujer)>

Cuando desde el elemento <terrestre> nos llegue un elemento <persona> el procesador no tendrá ninguna forma de determinar si estamos haciendo referencia al elemento <persona> de la expresión de la derecha persona,hombre o el izquierda persona,mujer, y por tanto fallará porque tiene dos opciones posibles. Es una expresión no determinista.

A menudo las expresiones no deterministas las podemos expresar de otro modo, de modo que se conviertan en deterministas. El ejemplo anterior podía haberse escrito de forma determinista para que al llegar un elemento <persona> no haya dudas.

<!ELEMENTO terrestre(persona,(hombre|mujer))>

Es bastante que siempre aparezca un elemento <persona> y después estará la alternativa entre un <hombre> o un <mujer> que es determinista.

Las expresiones no deterministas son más corrientes de lo que parece, ya que los modificadores pueden provocarlas y puede que no lo parezcan. Si se quisiera escribir una expresión que determine un libro a un libro a partir del autor o título en cualquier orden:

<!ELEMENTE libro(¿autor?, ¿título?|título?,autor?)>

Pero la expresión anterior es incorrecta, ya que si el validador se encuentra un <autor> ¿no sabe si es el autor del lado derecho de la condición autor?, ¿título? o bien es el del lado izquierdo que no tiene título, ¿título?, ¿autor?.

La expresión determinista idéntica a la anterior sería:

<!ELEMENTE libro(autor,titlo?|titulo,autor?|EMPTY)>

2.3. Ejemplo de creación de una DTD

Las DTD se siguen utilizando porque son sencillas de crear pero hay que recordar siempre las limitaciones que tienen, y tener presente que no siempre se adaptarán perfectamente a lo que se quiere hacer.

Enunciado

Una empresa ha preparado una tienda de Internet que genera los pedidos de los clientes en un formato XML que se envía al programa de gestión de forma automática.

Los XML que se generan contienen datos del cliente y del pedido realizado:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE comando SYSTEM "pedido.dtd">
<pedido numero="26" día="2011-12-01" >
    <cliente código="20">
        <nom>Frederic</nom>
        <apellido>Garcia</apellido>
    </client>
    <artículos>
        <artículo>
            <descripción>Yoda Mimobot USB Flash Drive 8GB</descripción>
            <cantidad>5</cantidad>
            <precio>38.99</precio>
        </artículo>
        <artículo>
            <descripción>Darth Vader Half Helmet Case for iPhone</descripción>
            <cantidad>2</cantidad>
            <precio>29.95</precio>
        </artículo>
    </artículos>
    <total valor="254,85"/>
</comando>
```

Antes de pasarlo al programa han decidido que para tener mayor seguridad se dará un paso previo que consistirá en validar que el documento. Por eso necesitan que se genere la DTD.

Resolución

Habrá que hacer dos cosas:

- Asociar las reglas al archivo XML. • Crear un archivo con las reglas, que se llamará “comando.dtd”.

Asociar el archivo XML

En la segunda línea del documento se especifica la regla DOCTYPE para asociar el archivo con la DTD.

```
<?xml version="1.0"?>
<!DOCTYPE albarán SYSTEM "pedido.dtd">
```

Crear las reglas

No hay una forma única de crear una DTD pero normalmente seguir un sistema puede ayudarle a evitar errores.
El sistema que se utilizará para resolver el sistema consiste en empezar por la raíz e ir definiendo las hojas por niveles.

La raíz del documento es el elemento <pedido>, que tiene tres hijos:

```
<!ELEMENT pedido (cliente, artículos, total)>
```

También tiene dos atributos, número y día, que sólo pueden tener valores sin espacios y, por tanto, serán NMTOKEN. Son datos obligatorios por motivos fiscales.

```
<!ATTLIST pedido número NMTOKEN #REQUIRED>
<!ATTLIST pedido día NMTOKEN #REQUIRED>
```

Una vez definido el primer nivel podemos pasar a definir a los demás. Por un lado el elemento <cliente>, que tiene un atributo para los clientes existentes y no tendrá para los nuevos:

```
<!ELEMENT cliente (nombre, apellido)>
<!ATTLIST cliente código NMTOKEN #IMPLIED >
```

Por otro lado el elemento <total>, que está vacío pero tiene un atributo:

```
<!ELEMENT total EMPTY>
<!ATTLIST total valor CDATA #REQUIRED >
```

También el elemento <artículos>, que contendrá una lista de los artículos que ha comprado el cliente. Como no tendría sentido realizar un pedido sin artículos el modificador que se utilizará será +.

```
<!ELEMENT artículos (artículo+)>
```

Por su parte, desarrollar <artículo> tampoco traerá problemas:

```
<!ELEMENT artículo (descripción, cantidad, precio)>
```

Por último sólo quedan los elementos que contienen datos:

```
<!ELEMENT nombre (#PCDATA)>
<!ELEMENT apellido (#PCDATA)>
<!ELEMENT descripción (#PCDATA)>
<!ELEMENT cantidad (#PCDATA)>
<!ELEMENT precio (#PCDATA)>
```

El archivo resultante será:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT pedido (cliente, artículos, total) >
<!ATTLIST pedido número NMTOKEN #REQUIRED >
<!ATTLIST pedido día NMTOKEN #REQUIRED>
<!ELEMENTO cliente (nombre,apellido) >
<!ATTLIST cliente código NMTOKEN #IMPLIED >
<!ELEMENT total EMPTY>
<!ATTLIST total valor CDATA #REQUIRED >
<!ELEMENT artículos (artículo+)>
<!ELEMENT artículo (descripción, cantidad, precio)>
<!ELEMENT nombre (#PCDATA)>
<!ELEMENT apellido (#PCDATA)>
<!ELEMENT descripción (#PCDATA)>
<!ELEMENT cantidad (#PCDATA)>
<!ELEMENT precio (#PCDATA)>
```

3. XML Schema Definition Language

En la especificación XML se hace referencia a las DTD como método para definir vocabularios XML, pero las DTD tienen una serie de limitaciones que llevaron al W3C a definir una nueva especificación. Esta especificación recibió el nombre de W3C XML Schema Definition Language (que popularmente se llama XML Schema o XSD), y se creó para sustituir la DTD como método de definición de vocabularios para documentos XML.

Se puede encontrar la especificación más reciente en www.w3.org/XML/Schema.

El éxito de XSD ha sido muy grande y actualmente se utiliza para otras tareas aparte de simplemente validar XML. También se utiliza en otras tecnologías XML como XQuery, servicios web, etc.

Las características más importantes que aporta XSD son:

- Está escrito en XML y, por tanto, no es necesario aprender un lenguaje nuevo para definir esquemas XML.
- Tiene su propio sistema de datos, por lo que se podrá comprobar el contenido de los elementos.
- Soporta espacios de nombres para permitir mezclar distintos vocabularios.
- Permite ser reutilizado y sigue los modelos de programación como herencia de objetos y sustitución de tipos.

3.1. Asociar un esquema a un archivo XML

A diferencia de lo que ocurre en otros lenguajes de definición –como por ejemplo en las DTD, en las que la asociación debe especificarse en el documento XML–, para validar un XML con un XSD no es necesario modificar el archivo XML. De todas formas, también es posible hacerlo definiendo el espacio de nombres.

Para asociar un documento XML a un documento de esquema es necesario definir el espacio de nombres con el atributo xmlns, y por medio de uno de los atributos del lenguaje definir cuál es el archivo de esquema:

```
<liga xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="liga.xsd">
```

Se pueden definir las referencias al archivo de esquema de dos formas:

Atributo	Significado
noNamespaceSchemaLocation	Se utiliza cuando no se utilizarán espacios de nombres en el documento.
schemaLocation	Se emplea cuando se utilizan explícitamente los nombres de los espacios de nombres en las etiquetas.

3.2. Definir un archivo de esquema

XSD está basado en XML y, por tanto, debe cumplir con las reglas de XML:

- Aunque no es obligatorio normalmente siempre se comienza el archivo con la declaración XML.
- Sólo existe un elemento raíz, que en este caso es <schema>.

Dado que no se está generando un documento XML libre sino que se está utilizando un vocabulario concreto y conocido para poder utilizar los elementos XML siempre se deberá especificar el espacio de nombres de XSD: ”http: //www.w3.org/2001/XMLSchema”.

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

En la declaración de este espacio de nombres se está definiendo que xs es el sobrenombre para hacer referencia a todas las etiquetas de ese espacio de nombres. Los alias para XSD normalmente son xs o xsd, pero en realidad no importa cuál se utilice siempre que se utilice el mismo en todo el documento.

Etiquetas de XSD

XSD define muchas etiquetas y no se verán todas aquí. Puede encontrar todas las etiquetas posibles en la especificación www.w3.org/TR/xmlschema11-1 .

La etiqueta <schema> puede tener distintos atributos.

Algunos atributos de la etiqueta <schema>:

Atributo	Significado
attributeFormDefault	Puede ser qualified si hacemos que sea obligatorio poner el espacio de nombres delante de los atributos o unqualified si no lo hacemos. Por defecto se utiliza unqualified.
elementFormDefault	Sirve para definir si es necesario el espacio de nombres delante del nombre. Puede tomar los valores qualified y unqualified.
version	Permite definir qué versión del documento de esquemas estamos definiendo (no es la versión de XML Schemas).

A partir de la etiqueta raíz ya se pueden empezar a definir las etiquetas del vocabulario que se desea crear.

Definición de elementos

Los elementos se definen utilizando la etiqueta <elemento>, y con atributos se especifica al menos el nombre y opcionalmente el tipo de datos que contendrá el elemento.

XSD divide los elementos en dos grandes grupos basándose en los datos que contienen:

- Elementos con contenido de tipo simple: son elementos sin atributos que sólo contienen datos.
- Elementos con contenido de tipo complejo: son elementos que pueden tener atributos, no tener contenido o contener elementos.

A partir de la definición puede verse que casi siempre habrá algún tipo complejo, ya que la raíz normalmente contendrá otros elementos.

Elementos con contenido de tipo simple

Se consideran elementos con contenido de tipo simple aquéllos que no contienen otros elementos ni tienen atributos.

La versión 1.1 de XSD define una cincuentena de tipos de datos diferentes, que se pueden encontrar en su definición (www.w3.org/TR/xmlschema11-2). Entre los más usados destacan:

Tipo	Datos que se pueden almacenar
string	Cadenas de caracteres
decimal	Valores numéricos
boolean	Sólo puede contener 'true' o 'false' o (1 o 0)
date	Fechas en forma (AAAA-MM-DD)
anyURI	Referencias a sitios (URL, caminos de disco…)
base64binary	Datos binarios codificados en base64
integer	Números enteros

A partir de los tipos básicos, el estándar crea otros con el objetivo de tener tipos de datos que se puedan adaptar mejor a los objetivos de quien diseña el esquema. En este sentido aparecen los tipos llamados positiveInteger, nonNegativeInteger, gYearMonth, unsignedInt…

Los tipos de datos permiten restringir los valores que contendrán las etiquetas XML. Por ejemplo, si se parte de la siguiente definición:

```
<xs:elemento name="posición" type="xs:integer"/>
</xs:schema>
```

Sólo se conseguirá validar un elemento si su contenido es un número entero. Por ejemplo, el siguiente ejemplo no validará:

```
<posicion>Primer</posicio>
```

A continuación se pueden ver ejemplos de definiciones de elementos y valores que validan.

Etiqueta	Ejemplo
<xs:elemento name=“día” type=“xs:date” />	<día>2011-09-15</dia>
<xs:elemento name=“altura” type=“xs:integer”/>	<altura>220</altura>
<xs:elemento name=“nombre” type=“xs:string”/>	<nom>Pere Puig</nom>
<xs:elemento name=“tamaño” type=“xs:float”/>	<tamaño>1.7E2</tamaño>
<xs:elemento name=“sitio” type=“xs:anyURI”/>	<sitio>http://www.ioc.cat</sitio>

Cuando se define una etiqueta en XSD, se está definiendo que la etiqueta deberá salir en el documento XML una vez. Es bastante habitual que haya etiquetas que se repitan un número determinado en ocasiones.
En XSD esto se ha simplificado por medio de unos atributos de la etiqueta <elemento> que determinan la cardinalidad de los elementos:

- minOccurs: permite definir cuántas veces debe salir un elemento como mínimo. Un valor de '0' indica que el elemento puede no salir.
- maxOccurs: sirve para definir cuántas veces como máximo puede salir un elemento. unbounded implica que no existe límite en las veces que puede salir.

Usando los atributos se puede definir que el elemento <nombre> debe salir una vez y el elemento <apellido> un máximo de dos veces.

```
<xs:elemento name="nombre" />

<xs:elemento name="apellido" maxOccurs="2"/>
```

También se pueden dar valores a los elementos con los atributos fixed, default y nullable.

El atributo fixed permite definir un valor obligatorio para un elemento:

```
<xs:elemento name="centro" type="xs:string" fixed="IOC"/>
```

De modo que sólo se podrá definir el contenido con el valor especificado (o sin nada):

```
<centro />

<centro>IOC</centro>
```

Pero nunca un valor distinto al especificado:

```
<centro>Instituto Cendrasos</centro>
```

A diferencia de fixed, default asigna un valor por defecto pero deja que sea cambiado en el contenido de la etiqueta.

```
<xsi:elemento name="centro" type="xs:string" default="IOC" />
```

La definición permitiría validar con los tres casos siguientes:

```
<centro />

<centro>IOC</centro>

<centro>Instituto Cendrasos</centro>
```

El atributo nullable se utiliza para decir si se permiten contenidos nulos. Por tanto, sólo puede tomar los valores yes o no.

Tipos simples personales

Puesto que a veces puede interesar definir valores para los elementos que no deben coincidir necesariamente con los estándares, XSD permite definir tipos de datos personales. Por ejemplo, si se desea un valor numérico pero que no acepte todos los valores sino un subconjunto de los enteros.

Para definir tipos simples personales no se pone el tipo en el elemento y se define un hijo <simpleType>.

```
<xs:elemento name="persona">
    <xs:simpleType>
        ...
    </xs:simpleType>
</xs:elemento>
```

Dentro de simpleType se especifica cuál es la modificación que se desea realizar. Lo más corriente es que las modificaciones sea hechas con list, union, restricción o extension.

A pesar de que se pueden definir listas de valores no se recomienda su uso. La mayoría de los expertos creen que es mejor definir los valores de la lista utilizando repeticiones de etiquetas.

Utilizar list permite definir que un elemento puede contener listas de valores. Por tanto, para especificar que un elemento <partidos> puede contener una lista de fechas se definiría:

```
<xs:elemento name="partidos">
  <xs:simpleType>
      <xs:list itemType="xs:date"/>
  </xs:simpleType>
</xs:elemento>
```

La etiqueta validaría con algo como:

```
<partidos> 2011-01-07 2011-01-15 2011-01-21</partits>
```

Los elementos simpleType también pueden definirse con un nombre fuera de los elementos y posteriormente usarlos como tipo de datos personal.

```
<xs:simpleType name="días">
    <xs:list itemType="xs:date"/>
</xs:simpleType>
  <xs:elemento name="partidos" type="días"/>
```

Usando los tipos personalizados con nombre se pueden crear modificaciones de tipo union. Los modificadores union sirven para que se puedan mezclar tipos diferentes en el contenido de un elemento.

La definición del elemento <precio> hará que el elemento pueda ser de tipo valor o tipo símbolo.

```
<xs:elemento name="precios">
    <xs:sipleType>
        <xs:union memberTypes="valor símbolo"/>
    </xs:simpleType>
</xs:elemento>
```

Con esto le podríamos asignar valores como estos:

```
<precios>25 €</precios>
```

Pero sin duda el modificador más interesante es el que permite definir restricciones a los tipos base. Con el modificador restricción se pueden crear tipos de datos en los que sólo se acepten algunos valores, que los datos cumplen una determinada condición, etc.

El elemento <nacimiento> sólo podrá tener valores enteros entre 1850 y 2011 si se define de esta forma:

```
<xs:simpleType name="año_nacimiento">
    <xs:restricion base="xs:integer">
        <xs:maxInclusive value="2011"/>
        <xs:minInclusive value="1850"/>
    </xs:restriction>
</xs:simpleType>
    <xs:elemento name="nacimiento" type="año_nacimiento"/>
```

Atributos que permiten definir restricciones en XSD

Elementos	Resultado
maxInclusive / maxExclusive	Se utiliza para definir el valor numérico máximo que puede tomar un elemento.
minInclusive / minExclusive	Permite definir el valor mínimo del valor de un elemento.
length	Con lenght restringimos la longitud que puede tener un elemento de texto. Podemos utilizar <xs:minLength> y <xs:maxLenght> para ser más precisos.
enumeración	Sólo permite que el elemento tenga alguno de los valores especificados en las distintas líneas <enumeration>.
totalDigidos	Permite definir el número de dígitos de un valor numérico.
fractionDigidos	Sirve para especificar el número de decimales que puede tener un valor numérico.
pattern	Permite definir una expresión regular a la que el valor del elemento se ha de adaptarse para poder ser válido.

Por ejemplo, el valor del elemento <respuesta> sólo podrá tener uno de los tres valores “A”, “B” o “C” si se define de este modo:

```
<xs:elemento name="respuesta">
    <xs:simpleType>
        <xs:enumeration value="A"/>
        <xs:enumeration value="B"/>
        <xs:enumeration value="C"/>
    </xs:simpleType>
</xs:elemento>
```

Una de las restricciones más interesantes son las definidas por el atributo patterno, que permite definir restricciones a partir de expresiones regulares. Como norma general tenemos que si se especifica un carácter en el patrón ese carácter debe salir en el contenido.

Definición de expresiones regulares

Símbolo	Equivalencia
.	Cualquier carácter
\d	Cualquier dígito
\D	Cualquier carácter no dígito
\s	Caracteres no imprimibles: espacios, tabuladores, saltos de línea…
\S	Cualquier carácter imprimible
x*	El delantero * debe salir 0 o más veces
x+	El delantero de + debe salir 1 o más veces
x?	El delantero de ? puede salir o no
[abc]	Tiene que haber algún carácter de los de dentro
[0-9]	Debe haber un valor entre los dos especificados, con estos incluidos
x{5}	Tiene que haber 5 veces lo que haya delante de los corchetes
x{5,}	Tiene que haber 5 o más veces el de delante
x{5,8}	Debe haber entre 5 y 8 veces el de delante

Utilizando este sistema se pueden definir tipos de datos muy personalizados. Por ejemplo, podemos definir que un dato debe tener la forma de un DNI (8 cifras, un guión y una letra mayúscula) con esta expresión:

```
<xs:simpleType name="dni">
    <xs:restriction base="xs:string">
        <xs:pattern value="[0-9]{8}-[AZ]"/>
    </xs:restriction>
</xs:simpleType>
```

Aparte de las restricciones también existe el elemento extension, que sirve para añadir características extra a los tipos. No tiene sentido que aparezcan en elementos de tipo simple pero se puede utilizar, por ejemplo, para añadir atributos a los tipos simples (lo que los convierte en tipos complejos).

Elementos con contenido tipo complejo

Los elementos con contenido de tipo complejo son aquellos que tienen atributos, contienen otros elementos o carecen de contenido.

Los elementos con contenido complejo han recibido muchas críticas porque se consideran demasiado complicados, pero deben utilizarse porque en todos los archivos de esquema normalmente habrá un tipo complejo: la raíz del documento.

Se considera que existen cuatro grandes grupos de contenidos de tipo complejo:

- Los que en su contenido sólo tienen datos. Por tanto, son como los de tipos simples pero con atributos.
- Los elementos que en su contenido sólo tienen elementos. • Los elementos vacíos. • Los
- elementos con contenido mezclado.

Los elementos de tipo complejo se definen especificando que el tipo de datos del elemento es <xs:complexType>.

```
<xs:elemento name="clase">
    <xs:complexType>
        ....
    </xs:complexType>
</xs:elemento>
```

Al igual que con los tipos simples, se pueden definir tipos complejos con nombre para reutilizarlos como tipos personalizados.

```
<xs:complexType name="curso">
    ...
</xs:complexType>

<xs:elemento clase type="curso"/>
```

Atributos

Una característica básica de XSD es que sólo los elementos de tipo complejo pueden tener atributos. En esencia no existen diferencias entre definir un elemento o un atributo, ya que se hace de la misma manera pero utilizando la etiqueta attribute.

Los tipos de datos son los mismos y, por tanto, pueden tener tipos básicos como en el ejemplo siguiente:

```
<xs:attribute name="número" type="xs:integer" />
```

Se pueden poner restricciones al igual que en los elementos. En este ejemplo, el atributo año no puede tener valores superiores a 2011 si se define de esta manera:

```
<xs:attribute name="año">
    <xs:simpleType>
        <xs:restriction base="xs:integer">
            <xs:maxInclusive value="2011"/>
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>
```

Si no se especifica lo contrario, los atributos son siempre opcionales.

La etiqueta <attribute> tiene una serie de atributos que permiten definir características extras sobre los atributos

Atributos más importantes de xs:attribute

Atributo	Uso
use	Permite especificar si el atributo es obligatorio (required), opcional (optional) o no se puede utilizar (prohibited).
default	Define un valor predeterminado.
fixed	Se utiliza para definir valores obligatorios para los atributos.
form	Permite definir si el atributo debe ir con el sobrenombre del espacio de nombres (qualified) o no (unqualified).

Por ejemplo, el atributo año deberá especificarse obligatoriamente si se define de la siguiente forma:

```
<xs:attribute name="año" type="xs:integer" use="required" />
```

'simpleContento'

Si el elemento contiene sólo texto, el contenido de complexType será un simpleContento. El simpleContento permite definir restricciones o extensiones a elementos que sólo tienen datos como contenido.

La diferencia más importante es que en ese caso se pueden definir atributos en el elemento. Los atributos se añaden definiendo una extensión al tipo utilizado en el elemento.

En este ejemplo el elemento <Mida> tiene contenido de tipo entero y define dos atributos, longitud y anchura, que también son enteros.

```
<xs:complexType name="Tamaño">
    <xs:simpleContento>
        <xs:extension base="xs:integer">
            <xs:atribute name="largo" type="xs:integer"/>
            <xs:atribute name="ancho" type="xs:integer"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
```

Contenido formado por elementos

Los elementos que contienen otros elementos también pueden ser definidos en XSD dentro de un <complexType> y pueden ser algunos de los siguientes elementos:

Etiqueta	Sirve para
secuencia	Especificar el contenido como una lista ordenada de elementos.
choice	Permite especificar elementos alternativos.
ajo	Definir el contenido como una lista desordenada de elementos.
complexContento	Extender o restringir el contenido complejo.

El elemento <sequence> es una de las formas con las que el lenguaje XSD permite que se especifiquen los elementos que deben formar parte del contenido de un elemento. Incluso en el caso en que sólo exista una sola etiqueta se puede definir como una secuencia.

Su condición más importante es que los elementos del documento XML para validar deben aparecer en el mismo orden en el que se definen en la secuencia.

```
<xs:elemento name="persona">
    <xs:complexContento>
        <xs:sequence>
            <xs:elemento name="nombre" type="xs:string"/>
            <xs:elemento name="apellido" type="xs:string" maxOccurs="2"/>
            <xs:elemento name="tipo" type="xs:string" />
        </xs:sequence>
    </xs:complexContent>
</xs:elemento>
```

En el ejemplo anterior se define que antes de la aparición de <tipo> pueden aparecer uno o dos apellidos.

```
<persona>
    <nom>Marcel</nom>
    <apellido>Puig</apellido>
    <apellido>Lozano</apellido>
    <tipo>Profesor</tipo>
</persona>
```

No validará ningún contenido si algún elemento no está exactamente en el mismo orden.

```
<persona>
    <tipo>Profesor</tipo>
    <apellido>Puig</apellido>
    <nom>Marcel</nom>
</persona>
```

Las secuencias pueden contener en su interior otras secuencias de elementos.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <xs:elemento name="persona">
        <xs:complexType>
            <xs:sequence>
                <xs:elemento name="nombre completo">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:elemento name="nombre" type="xs:string"/>
                            <xs:elemento name="apellido" type="xs:string" maxOccurs="2"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:elemento>
                <xs:elemento name="profesión" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:elemento>
</xs:schema>
```

El elemento <choice> sirve para hacer que se deba elegir una de las alternativas de las que se presentan dentro sede.

En este ejemplo el elemento persona podrá contener la etiqueta <nombre Apellidos> o bien <dni>, pero no ambas.

```
<xs:complexType nombre="persona">
    <xs:choice>
        <xs:elemento name="nombre Apellidos" type="xs:string"/>
        <xs:elemento name="dni" type="xs:string"/>
    </xs:choice>
```

Entre las alternativas puede haber secuencias u otros elementos <choice>. La siguiente definición es un ejemplo más elaborado que el anterior y permite que se pueda elegir entre los elementos <nombre> y <apellido> o <dni>.

```
<xs:choice>
    <xs:sequence>
        <xs:elemento name="nombre" type="xs:string"/>
        <xs:elemento name="apellido" type="xs:string" maxOccurs="2"/>
    </xs:sequence>
    <xs:elemento name="dni" type="xs:string"/>
</xs:choice>
```

La diferencia más importante entre el elemento <all> y <sequence> es el orden. El elemento <all> permite especificar una secuencia de elementos pero permite que se especifiquen en cualquier orden.

Por tanto, si definimos el elemento <persona> de la siguiente manera:

```
<xs:elemento name="persona">
    <xs:complexType>
        <xs:all>
            <xs:elemento name="nombre"/>
            <xs:elemento name="apellido"/>
        </xs:all>
    </xs:complexType>
</xs:elemento>
```

Nos servirá para validar tanto este documento:

```
<persona>
    <nom>Pere</nom>
    <apellido>Garcia</nom>
```

como éste:

```
<persona>
    <apellido>Garcia</nom>
    <nom>Pere</nom>
```

Pero siempre deben tenerse en cuenta las limitaciones de este elemento que no estaban presentes en las secuencias ordenadas:

- En su interior sólo puede haber elementos. No puede haber ni secuencias, ni alternativas.
- No se puede utilizar la cardinalidad en los elementos que contenga, ya que provocaría un problema de no determinismo.

Por tanto, el ejemplo siguiente no es correcto, ya que se pide que <apellido> pueda salir dos veces.

```
<xs:all>
    <xs:elemento name="nombre" type="xs:string"/>
    <xs:elemento name="apellido" maxOccurs="2" type="xs:string"/>
</xs:all>
```

Una posible forma de permitir que se puedan especificar el nombre y los dos apellidos en cualquier orden sería hacer lo siguiente:

```
<xs:complexType>
    <xs:choice>
        <xs:sequence>
            <xs:elemento name="nombre" type="xs:string"/>
            <xs:elemento name="apellido" type="xs:string" maxOccurs="2"/>
        </xs:sequence>
        <xs:sequence>
            <xs:elemento name="apellido" type="xs:string" maxOccurs="2"/>
            <xs:elemento name="nombre" type="xs:string"/>
        </xs:sequence>
    </xs:choice>
</xs:complexType>
```

'complejoContento'

La etiqueta complexContent permite definir extensiones o restricciones a un tipo complejo que contengan contenido mezclado o sólo elementos.

Esto hace que con una extensión se pueda ampliar un contenido complejo ya existente o restringir sus contenidos.

Por ejemplo, si ya hay definido un tipo de datos nombreCompleto en el que están los elementos <nombre> y <apellido> se puede reutilizar su definición para definir un nuevo tipo de datos, agenda, en el que se añada el dirección de correo electrónico.

```
<xs:complexType name="nombreCompleto">
   <xs:sequence>
        <xs:elemento name="nombre" type="xs:string"/>
        <xs:elemento name="apellido" type="xs:string" maxOccurs="2"/>
   </xs:sequence>
</xs:complexType>
     <xs:complexType name="agenda">
    <xs:complexContento>
        <xs:extension base="nombreCompleto">
            <xs:sequence>
                <xs:elemento name="email" type="xs:string" />
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

De esta forma se podrá definir un elemento de tipo agenda:

```
<xs:elemento name="persona" type="agenda"/>
```

que deberá tener los tres elementos <nombre>, <apellido>, y <email>:

```
<persona>
    <nom>Pere</nom>
    <apellido>Garcia</apellido>
    <email>pgarcia@ioc.cat</email>
</persona>
```

Elementos sin contenido

Para XSD, los elementos sin contenido son siempre de tipo complejo. En la definición simplemente no se especifica ningún contenido y tendremos un elemento vacío.

```
<xs:elemento name="delegado">
    <xs:complexType />
</xs:elemento>
```

La definición permite definir el elemento de esta forma:

```
<delegado />
```

Si el elemento necesita atributos simplemente se especifican dentro del complexType.

```
<xs:elemento name="delegado">
    <xs:complexType>
        <xs:attribute name="año" use="required" type="xs:gYear"/>
    </xs:complexType>
</xs:elemento>
```

Y ya se podrá definir el atributo en el elemento vacío:

```
<delegado año="2012"/>
```

Contenido mezclado

Los elementos con contenido mezclado son los elementos que tienen contenido tanto elementos como texto. Se pensó para incluir elementos en medio de un texto narrativo. En XSD el contenido mezclado se define poniendo el atributo mixed="true" en la definición del elemento <complexType>.

```
<xs:elemento name="carta">
    <xs:complexType mixed="true">
        <xs:sequence>
            <xs:elemento name="nombre" type="xs:string"/>
            <xs:elemento name="día" type="xs:gDay"/>
        </xs:sequence>
    </xs:complexType>
</xs:elemento>
```

Esto nos permitiría validar un contenido como éste:

```
<carta>Querido señor <nombre>Pere<nombre>:
Le envío esta carta para recordarle que hemos quedado por
encontrarnos el día <día>12</dia>
</carta>
```

3.3. Ejemplo de creación de un XSD

Se pueden crear definiciones de vocabularios XSD a partir de la idea de lo que queremos que contengan los datos o bien a partir de un archivo XML de muestra.

Enunciado

En un centro en el que sólo se imparten los ciclos formativos de SMX y ASIX, cuando deben entrar las notas a los alumnos lo hacen creando un archivo XML como el siguiente:

```
<?xml version="1.0" ?>
<clase modul="3" nommodul="Programación básica">
    <curso numero="1" especialidad="ASIX">
        <profesor>
            <nom>Marcel</nom>
            <apellido>Puig</apellido>
        </profesor>
        <alumnos>
            <alumno>
                <nom>Filomeno</nom>
                <apellido>Garcia</apellido>
                <nota>5</nota>
            </alumno>
            <alumno delegado="si">
                <nom>Frederic</nom>
                <apellido>Pi</apellido>
                <nota>3</nota>
            </alumno>
            <alumno>
                <nombre>Manel</nombre>
                <apellido>Puigdevall</apellido>
                <nota>8</nota>
            </alumno>
        </alumnos>
    </curs>
</clase>
```

En la secretaría necesitan que se genere la definición del vocabulario en XSD para comprobar que los archivos que reciben son correctos.

Resolución

A la hora de diseñar un esquema desde un XML siempre se debe partir de un archivo XML que tenga todas las opciones que pueden salir en cada elemento, o el resultado no será válido para todos los archivos.

Una de las cosas que deben tenerse en cuenta a la hora de hacer un ejercicio como éste es que no hay una manera única de hacerlo. Se puede hacer con tipos personalizados, sin tipos personalizados, a veces se puede resolver combinando unos elementos pero también con otros, etc. Por tanto, la solución que se ofrecerá aquí es sólo una de las posibles soluciones.

Análisis

La primera fase normalmente consiste en analizar el archivo para determinar si existen partes que pueden ser susceptibles de formar un tipo de datos. En el ejercicio se puede ver que tanto para el profesor como para los alumnos los datos que contienen son similares (los alumnos añaden la nota), y por tanto se puede crear un tipo de datos que hará más simple el diseño final.

Por tanto, en la raíz podemos crear el tipo complejo persona, que contendrá el nombre y el apellido.

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

Como los alumnos son iguales pero con un elemento extra se puede aprovechar el tipo persona extendiéndolo.

```
<xs:complexType name="estudiante">
    <xs:complexContento>
        <xs:extension base="persona">
            <xs:sequence>
                <xs:elemento name="nota">
                    ...
                </xs:elemento>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Las notas no pueden tener todos los valores (sólo de 1 a 10), es decir, que se puede añadir una restricción al tipo entero.

```
<xs:elemento name="nota">
    <xs:simpleType>
        <xs:restriction base="xs:integer">
            <xs:maxInclusive value="10"/>
            <xs:minInclusive value="1"/>
        </xs:restriction>
    </xs:simpleType>
</xs:elemento>
```

Desarrollo

La raíz siempre será de tipo complejo porque contiene atributos y tres elementos, por lo que se puede empezar la declaración como una secuencia de elementos.

```
<xs:elemento name="clase">
    <xs:complexType>
        <xs:sequence>
            <xs:elemento name="curso">
                ...
            </xs:elemento>
            <xs:elemento name="profesor" type="persona"/>
            <xs:elemento name="alumnos">
                ...
            </xs:elemento>
        </xs:sequence>
    </xs:complexType>
</xs:elemento>
```

Se han dejado los elementos <curso> y <alumnos> para desarrollarlos, mientras que el elemento <profesor> ya puede cerrarse porque se ha definido el tipo persona.

El elemento <curso> no tiene contenido y, por tanto, será de tipo complejo. Además, deben definirse dos atributos.

```
<xs:elemento name="curso">
    <xs:complexType>
        <xs:attribute name="numero" use="required">
            ...
        </xs:attribute>
        <xs:attribute name="especialidad" use="required">
            ...
        </xs:attribute>
    </xs:complexType>
</xs:elemento>
```

Los atributos deben desarrollarse porque no pueden tener todos los valores:

• número sólo puede ser “1” o “2”. •
especialidad será “SMX” o “ASIX”.

La forma más adecuada para ello será restringir los valores con una enumeración.

```
<xs:attribute name="numero" use="required">
    <xs:simpleType>
        <xs:restriction base="xs:integer">
            <xs:enumeration value="1"/>
            <xs:enumeration value="2"/>
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>
<xs:attribute name="especialidad" use="required">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="ASIX"/>
            <xs:enumeration value="SMX"/>
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>
```

Ya sólo queda por desarrollar <alumnos>, que tiene una repetición de elementos <alumno>, del que ya hay un tipo.

```
<xs:elemento name="alumno">
    <xs:complexType>
        <xs:sequence>
            <xs:elemento name="alumno" type="estudiante"
                        maxOccurs="unbounded" minOccurs="1"/>
        </xs:sequence>
    </xs:complexType>
</xs:elemento>
```

El resultado final será:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:complexType name="persona">
    <xs:sequence>
        <xs:elemento name="nombre" type="xs:string"/>
        <xs:elemento name="apellido" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="estudiante">
    <xs:complexContento>
        <xs:extension base="persona">
            <xs:sequence>
                <xs:elemento name="nota">
                    <xs:simpleType>
                        <xs:restriction base="xs:integer">
                            <xs:maxInclusive value="10"/>
                            <xs:minInclusive value="1"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:elemento>
            </xs:sequence>
            <xs:attribute name="delegado" use="optional">
                <xs:simpleType>
```

```
                                        <xs:restriction base="xs:string">
                                            <xs:enumeration value="si"/>
                                        </xs:restriction>
                                    </xs:simpleType>
                                </xs:attribute>
                            </xs:extension>
                        </xs:complexContent>
                    </xs:complexType>
                    <xs:elemento name="clase">
                        <xs:complexType>
                            <xs:sequence>
                                <xs:elemento name="curso">
                                    <xs:complexType>
                                        <xs:sequence>
                                            <xs:elemento name="profesor" type="persona"/>
                                            <xs:elemento name="alumnos">
                                                <xs:complexType>
                                                    <xs:sequence>
                                                        <xs:elemento                    name="alumno"
type="estudiante" maxOccurs="unbounded" />
                                                    </xs:sequence>
                                                </xs:complexType>
                                            </xs:elemento>
                                        </xs:sequence>
                                        <xs:attribute name="numero" use="required">
                                            <xs:simpleType>
                                                <xs:restriction base="xs:integer">
                                                    <xs:enumeration value="1"/>
                                                    <xs:enumeration value="2"/>
                                                </xs:restriction>
                                            </xs:simpleType>
                                        </xs:attribute>
                                        <xs:attribute name="especialidad" use="required">
                                            <xs:simpleType>
```

```
                                        <xs:restriction base="xs:string">
                                            <xs:enumeration value="ASIX"/>
                                            <xs:enumeration value="SMX"/>
                                        </xs:restriction>
                                    </xs:simpleType>
                                </xs:attribute>
                            </xs:complexType>
                        </xs:elemento>
                    </xs:sequence>
                    <xs:attribute name="modulo" use="required" type="xs:integer" />
                    <xs:attribute name="nomodul" use="required" type="xs:string" />
                </xs:complexType>
            </xs:elemento>
            </xs:schema>
```

4. Enlaces de interés

DTDs en w3school: https://www.w3schools.com/xml/xml_dtd_intro.asp

5. Bibliografía

Sala, Javier. (2023) Lenguajes de marcas y sistemas de gestión de información.

Instituto Abierto de Cataluña