Capítulo 1

Tipos de datos y su intercambio

Para que dos entidades puedan intercambiar información es necesario que tengan un acuerdo previo. Las especificaciones de formatos son el acuerdo desde el punto de vista de los datos.

Guía de lectura

Este capítulo tiene un carácter teórico. En la sección 1.1 definimos la noción de metadato y sus tipos posteriormente, en la sección 1.2 veremos las diferentes formas de representar los datos y los formatos estándar. En la sección 1.3 repasamos someramente el lenguaje de marcas XML junto con algunas técnicas para su validación. En la sección 1.4 se describen métodos para leer archivos XML y por último, en la sección 1.5 veremos dos alternativas al XML: JSON y YAML.

1.1. Metadatos

Un metadato es información descriptiva acerca de otros datos. Los metadatos son transparentes para el usuario, pero constituyen la infraestructura de la información visible. El estándar ISO/IEC 11179 define el concepto de metadato. Como concepto, son previos a la informática.

Los metadatos están fuertemente estructurados, lo que se refleja directamente en el código que los debe leer o escribir, así como en el diseño de las estructuras de datos o clases que los representan.

En general los metadatos van al lado de los datos que describen, por ejemplo los CRCs de las tramas del nivel de enlace o las direcciones de los DNS o las cabeceras de los correos electrónicos, pero a veces están separados, por ejemplo, en el sistema de archivos de UNIX, la tabla de inodos está en un lugar del disco aparte de los datos que referencia.

1.1.1. Tipos de metadatos

En función del foco de interés existe más de una clasificación, pero la que parece ser más aceptada es esta:

1. **Administrativos**: Permiten gestionar los datos para controlar el acceso y el uso. Ejemplos: los bits `drwxrwxrwx` de cada archivo Unix, la extensión, que informa del tipo de documento (PDF, BMP, etc) o la fecha de recepción de un correo.

2. **Estructurales**: Describen las relaciones entre datos complejos y su estructura interna. Ejemplos: índices de bases de datos o archivos referenciados en una página web.

3. **Descriptivos**: Dan información pormenorizada de cada recurso de información. Permiten a los usuarios búscar y recuperar información. Ejemplo: la descripción de las etiquetas de un documento HTML.

1.1.2. Utilización de metadatos

En un proyecto los metadatos se deben definir en la especificación y se debe tener en cuenta cómo se van a grabar y recuperar. El nivel de detalle de la definición debe ser al menos equiparable al de los datos que referencian. Y por último, se deben tener muy en cuenta los estándares de metadatos que ya estén definidos en el dominio sobre el que se trabaje para, de este modo, tener la capacidad de intercambiar datos con otros sistemas y no reinventar la rueda.

1.2. Representación y codificación de datos

Cuando se habla de codificación de datos se suele dar una clasificación condicionada por el tema de estudio, por ejemplo, en programación se habla de las

alternativas para representar datos numéricos y estructuras de datos; en comunicaciones se habla de señales digitales y analógicas y cómo se puede transportar información digital o analógica en cada una de ellas. En general, siempre se habla de la conversión de unos sistemas de codificación en otros.

En nuestro caso, el foco de interés va a estar en la web y formatos estándar, intentando no descender a una descripción excesivamente detallada. El objetivo es dar una descripción de las opciones para codificar los distintos tipos de información indicando ventajas e inconvenientes.

En 1.2.1 describiremos las distintas alternativas para representar texto plano; en 1.2.4 veremos las definiciones de diferentes tipos de formatos; en 1.2.2 describiremos los principales formatos utilizados por procesadores de texto, hojas de cálculo y presentaciones.

1.2.1. Texto plano

Es difícil creer que algo aparentemente tan sencillo como la codificación de caracteres no haya tenido una solución universal desde un principio, veremos los motivos con un pequeño recorrido histórico por los sistemas de codificación de caracteres.

ASCII: La primera versión, de 1963, tuvo su origen en una evolución de los códigos usados en telegrafía. Tiene 7 bits para representar información, mas un bit de paridad. De los $2^7 = 128$ posibles símbolos, la versión actual utiliza 32 para códigos de control no imprimibles (y obsoletos hoy en día) y 95 imprimibles. Los códigos de control son metadatos para dispositivos externos, como impresoras o cintas magnéticas. Es interesante destacar que los símbolos están en orden alfabético, las minúsculas y mayúsculas se diferencian en un bit y los caracteres numéricos empiezan a partir del 48, es decir el carácter 0 es 48, el 1 es 49, etc, lo que simplifica su conversión a BCD, tomando sólo los últimos 4 bits de los caracteres cuyos primeros 4 bits sean 0011. Presenta sin embargo algunos problemas serios: codifica pocos símbolos, que además están pensados sólo para el inglés (no contiene ñ, ç, etc) y, el único alfabeto soportado es el latino.

Codepages: ASCII es compacto, pero desperdicia un bit en la paridad, que además no es necesario, porque todos los sistemas de almacenamiento ya incor-

poran redundancia para detectar errores. Así que para ampliar el exiguo conjunto de caracteres y dar cabida a otros idiomas muchas empresas, cada una según su criterio e intereses, desarrollaron sus propias soluciones. IBM creó una serie de tablas de símbolos, distintas para cada país, conocidas como CP-*Número*, por ejemplo el famoso CP-437, mal llamado ASCII extendido; por su parte, la organización ISO sacó los ISO/IEC 8859-*x* y Microsoft los Windows-*xxxx*. La ventaja de los codepages es que la mayor parte de los idiomas tienen cubierto su alfabeto con sus acentos y símbolos propios, además, son compatibles hacia atrás, es decir, los códigos imprimibles en el primitivo ASCII de 7 bits tienen la misma codificación numérica en los codepages de 8 bits. El inconveniente es la falta de compatibilidad entre distintos codepages para los caracteres extra (del 128 al 255): un mismo carácter se puede interpretar como un símbolo u otro si no se sabe el codepage con el que ha sido codificado. Por otra parte, cualquier sistema de codificación de 8 bits ($2^8 = 256$) no puede cubrir la gran cantidad de ideogramas chinos, japoneses o coreanos.

Asia: El problema de algunos lenguajes asiáticos es que tienen miles de símbolos distintos. La solución inicial a ese problema fue *Double-Byte Character Set* (DBCS), que utiliza 8 bits para los símbolos más comunes y 16 para el resto. En DBCS es fácil recorrer una cadena hacia adelante, el problema es que al recorrerla hacia atrás puede ocurrir que no sea posible saber si un byte forma parte de un símbolo de dos bytes o de uno solo. Este problema se llama *ambigüedad de límites*.

Unicode Transformation Format (UTF): Las diferentes versiones de UTF son, hasta la fecha, el sistema más universal, pues dan soporte a todos los idiomas mayoritarios.

- UTF 16: Es un sistema de longitud variable. Está dividido en diecisiete planos de 2^{16} símbolos, que pueden tener una longitud de una o dos palabras de 16 bits. El *plano básico multilingüe* puede representar todos los símbolos de codepages anteriores y ocupa una sola palabra. Al ser su longitud mínima de 16 bits, no es compatible con ASCII.

- UTF 32: Tiene una longitud fija de 32 bits y permite representar 2^{32} símbolos, lo que es más que suficiente. El único problema de UTF-32 es su baja

eficiencia por la memoria que ocupa. Tampoco es compatible con ASCII.

- UTF 8: Es un sistema de longitud variable de entre uno y seis bytes. Los códigos de un byte son compatibles con ASCII y en los demás se han seguido varios criterios, como codificar con la menor longitud posible los símbolos más usuales o poner juntos los símbolos de un idioma en la medida de lo posible. Ocupa un poco más que los codepages porque los símbolos específicos se codifican con más de un byte, y también ocupa un poco más que DBCS, pero la diferencia, hablando siempre de promedios, es pequeña. Además, es posible recorrer una cadena hacia adelante y atrás sin problemas de ambigüedad de límites, e incluye suficientes símbolos para soportar todos los lenguajes. El último pequeño problema es que, al ser un código de longitud variable, para acceder al símbolo n-ésimo de una cadena hay que recorrerla desde el principio, lo que convierte a esta operación de complejidad $O(1)$ en una de complejidad $O(n)$. Debido a sus ventajas es el estándar usado en Linux. La mayor parte de los editores lo soportan.

1.2.2. Formatos en ofimática

Las aplicaciones típicas en ofimática son procesadores de texto, hojas de cálculo, programas para presentaciones, bases de datos sencillas y agendas. Un procesador de texto no sólo representa texto plano, también necesita códigos especiales para fijar márgenes, tipos de letra, elementos flotantes como tablas o imágenes, etc.

Los formatos propietarios son claramente mayoritarios, en concreto el formato *DOC* de *Microsoft Word* que ha sido más usado en su momento que todos los demás juntos. El formato seguido por este procesador de textos desde hace 2007 es *DOCX*, también de uso mayoritario en la actualidad. Por lo que respecta a las hojas de cálculo, el formato más usado es el *XLS* de *Microsoft Excel*, y posteriormente el *XLSX*. Para presentaciones el formato más usado fue en su momento el *PPT*, de *Microsoft Power Point*, seguido actualmente por el PPTX. Existen formatos abiertos, el principal es *Open Document Format* (ODF), soportado por *LibreOffice*, *OpenOffice* y otros, aunque comparativamente poco extendido. El formato *Portable Document Format* (PDF), creado por la empresa *Adobe* permite contener varios tipos de información: texto formateado, gráficos y enlaces.

Los formatos de texto siempre han partido de la base de un tamaño fijo de página, asumiendo que debían ser imprimibles. El desarrollo de formas distintas de presentación ha propiciado la aparición de formatos que se adaptan a distintos tamaños de página, como el ePub, mobi, djvu y otros, usados en lectores de libros electrónicos.

1.2.3. Multimedia

El término multimedia siempre ha sido confuso. En general se refiere a todo lo relacionado con imagen, sonido y vídeo.

Formatos de imagen

Las imágenes ocupan mucho espacio en comparación con otro tipo de información, por ejemplo, una sola fotografía puede ocupar más que todo un libro en texto plano. Debido a esto se han desarrollado distintos tipos de formatos que comprimen las imágenes de uno u otro modo.

Formatos sin compresión: Guardan la imagen original tal cual. La ventaja es que no hay que aplicar ningún algoritmo para dibujarlos y se pueden manipular fácilmente. El problema consiste en el gran tamaño que ocupan. Los formatos más relevantes[1] son estos dos:

- Formato crudo, en inglés *raw*: Es la información tal cual sale del *charge-coupled device* (CCD), que se podría traducir como dispositivo de carga acoplada, o más coloquialmente, sensor de imagen. No existe un único formato crudo porque existen muchos tipos de CCD y sus características difieren, aunque se intenta conseguir un estándar. Su ventaja principal es que da la imagen más perfecta que se puede obtener de una cámara. Los profesionales que trabajan con imágenes prefieren este formato porque, las sucesivas transformaciones que se aplican sobre una imagen al manipularla, degradan su calidad, y el formato crudo sufre menos este problema. Es el formato que más memoria ocupa (36 a 48 bits/pixel).

[1] Existen decenas de formatos de imagen. En este apartado y sucesivos sólo hablaremos por encima de los más significativos.

- *Bitmap Image File* (BMP): Cada imagen contiene una pequeña cabecera con metadatos y una matriz rectangular de pixels. Cada pixel se representa con entre 1 y 32 bits en función de si es monocromo o a color y de la profundidad de color (tiene varias versiones). Después del formato crudo es el que más ocupa.

Formatos vectoriales: Aquellas imágenes que consisten en diagramas tienen poca variedad de colores y formas. En lugar de guardar la información de cada pixel, uno por uno, se guarda la forma de reproducir el dibujo, que consiste en elementos gráficos: puntos, segmentos, líneas curvas (que pueden ser descritas por ecuaciones sencillas), texto y, opcionalmente, importar archivos gráficos de otros formatos, como BMP. Se suele especificar también, para cada elemento gráfico, varios atributos como su color, tipo de línea, terminaciones de los segmentos, fuente, etc.

Los formatos vectoriales tienen dos ventajas: la posibilidad de escalar (agrandar o reducir) la imagen sin perder calidad y el mínimo tamaño que ocupan.

Algunos formatos vectoriales relevantes son:

- *Scalable Vector Graphics* (SVG): Basado en XML y definido por el W3C. Se está convirtiendo en el estándar para páginas web debido a su poco peso.
- *Windows MetaFile* (WMF): Es un conjunto de llamadas a funciones de dibujo. Se le descubrió una vulnerabilidad que permitía ejecutar código de forma remota.
- *Enhanced MetaFile* (EMF): Es el sucesor de WMF con más funciones y la posibilidad de ser usado como lenguaje gráfico para impresoras.

Formatos de compresión sin pérdida: No todas las imágenes son diagramas, las fotografías no pueden guardarse con formatos vectoriales, pero ocupan demasiado en formato crudo o BMP. La compresión sin pérdida es un intento de nadar y guardar la ropa, que consiste en aplicar un algoritmo de compresión que permite, al descomprimir posteriormente, restaurar la imagen original bit a bit. La tasa de compresión de estos algoritmos es modesta, y tienen el inconveniente del tiempo de proceso, sobre todo en la compresión.

Los más relevantes son:

- *Graphics Interchange Format* (GIF): Ocupa 8 bits/pixel. Es antiguo pero aun muy usado. Su tasa de compresión[2] esta entre 4:1 a 10:1. La compresión es sin pérdida para imágenes con número de colores <= 256.

- *Portable Network Graphics* (PNG): Está pensado para reemplazar al anterior. Permite más profundidad de color, hasta 48 bits, y monocromo de hasta 16 bits. Su tasa de compresión es entre un 10 % y un 30 % mejor que la de GIF.

Formatos de compresión con pérdida: Si una imagen no se va a examinar con detalle, se puede comprimir con un algoritmo que alcance tasas de compresión muy altas, aunque la imagen que luego se restaura no sea exactamente igual a la original.

El algoritmo típico de compresión con pérdida es *Joint Photographic Experts Group* (JPEG y JPEG 2000), que tiene una tasa de compresión que se puede ajustar, desde 10:1 a 1000:1; cuanto mayor sea la tasa de compresión, peor es la calidad de la imagen. La nueva versión, además de añadir nuevos algoritmos, permite también compresión sin pérdida. Las siglas son el nombre del comité que creó el estándar.

Conversión entre formatos de imagen: La necesidad de convertir un tipo de imagen en otra viene motivada por tres causas: almacenamiento, transmisión y manipulación. Los problemas de almacenamiento y transmisión son el mismo: las imágenes ocupan mucho. El problema de manipulación consiste en que no todos los programas están preparados para mostrar las imágenes en cualquier formato, con lo que hay que convertirlas a uno que entienda el programa visualizador. En cualquier caso, siempre se debe tener presente que con cada transformación puede haber pérdida de calidad, y si se aplican varias en cadena, esa pérdida se va acumulando.

Formatos de sonido

Al igual que con las imágenes, el sonido se puede guardar sin y con pérdida. Afortunadamente no hay tanta variedad de formatos como con las imágenes. Los

[2] La tasa de compresión la vamos a expresar de este modo: *X:1*, que significa que la imagen comprimida ocupa $\frac{1}{X}$ de lo que ocupa la original.

más importantes son estos:

- *Waveform Audio File Format* (WAV): Es el formato sin compresión. Ocupa mucho espacio y sólo es recomendable para edición.

- *MPEG-1 (o 2) Audio Layer III* (MP3): Es un formato de compresión con pérdida. El ratio se puede ajustar y aproximadamente es de 1:10. Es sin duda el más popular, lo que ha llevado a desarrollar otros formatos a partir de él: WMA de Microsoft y OGG usado por el software libre como Linux. Ninguno de ellos ha tenido mucho éxito.

- *Musical Instrument Digital Interface* (MIDI): Está basado en una aproximación similar a los formatos vectoriales para gráficos. En lugar de codificar sonido contiene instrucciones para sintetizarlo por software. Ocupa mucho menos que MP3 pero no es posible comprimir en formato MIDI un sonido grabado desde un micrófono, lo que ha limitado su uso.

Formatos de video

El video es una colección de imágenes y sonido, cada uno de ellos se trata por separado y se unen en lo que se llaman formatos contenedores. La compresión y descompresión se realiza mediantes *codecs* (acrónimo de codificador/decodificador), que se pueden definir como procesos que convierten una corriente de datos (*stream*) en otra. El *stream* resultante es de menor tamaño que los datos originales excepto si se van a utilizar para edición.

El *streaming* se puede traducir como "transmisión de flujo de datos", y consiste en un proceso de transmisión de información entre la máquina A y la máquina B en el que los datos se van consumiendo en B a medida que se reciben, reconstruyendo la información original. La información que se recibe es, generalmente, multimedia.

Al igual que en los anteriores apartados veremos una lista de los formatos de vídeo más importantes:

- *Audio Video Interleaved* (AVI): Es el más usado en la actualidad. Admite varios codecs, como DivX, XviD, CinePack, Digital Video, Intel Indeo, etc. Su variedad de codecs le permite múltiples ratios y calidades de compresión.

- *Moving Pictures Expert Group* (MPEG): También tiene varios codecs, como MPEG-1, 2 y 4 para vídeo y MP3 para sonido.
- *MOV*: Es el formato desarrollado por Apple. Tiene un codec propio. La calidad es alta y el tamaño de los archivos bajo, lo que le hace adecuado para ser usado en páginas web. Admite streaming.
- *WMV*: Es el formato de Microsoft, que utiliza el codec MPEG-4. Admite streaming.
- *RM*: Formato de Real Networks, tiene un codec propio y admite streaming.
- *FLV*: Es el formato de Adobe Flash. Los codecs son Sorenson Spark y On2 VP6. Necesita un reproductor de Flash y admite streaming.

1.2.4. Tipos de formatos desde el punto de vista legal

Los derechos de autor también se pueden aplicar a los formatos, y hay que ser muy consciente de ello al elegir en que formato se almacena la información, por lo que respecta al precio y al esfuerzo de mantenimiento a largo plazo.

A continuación introducimos algunas definiciones importantes:

Formato cerrado: Es aquel cuyas especificaciones no se han publicado.

Formato abierto: Es aquel cuyas especificaciones son públicas y cualquiera puede hacer software que lea o escriba en él. Suelen ser legibles con un editor de texto.

Formato propietario: Es aquel que está protegido por derechos de autor o patentes. Sólo el propietario de los derechos puede hacer software que lea o escriba en ese formato. Es común que los formatos propietarios también sean cerrados.

Dependencia del proveedor: Es la consecuencia de los formatos propietarios. El fabricante es el único que puede mantener el software. Tiene dos problemas: la posible desaparición del formato y la falta de interoperabilidad, es decir, el propio fabricante no hace el software que pueda traducir desde su formato hasta otros, aunque si se suelen suministrar facilidades de importación. Por este motivo los formatos propietarios y cerrados pueden suponer un mercado cautivo. Es

frecuente que una empresa que controle la mayor parte del mercado use estos formatos como una barrera para la competencia, a veces incluso pueden crear formatos nuevos introduciendo cambios mínimos en estándares ya establecidos, de modo que sólo su propio software, de uso mayoritario, pueda leer ambas versiones y grabar sólo en la suya. Al cabo de cierto tiempo sólo se usa el formato de esa empresa.

1.3. XML

El ***eXtensible Markup Language*** **(XML)** es un lenguaje de marcas desarrollado por el *World Wide Web Consortium* (W3C). La característica importante de XML es que está diseñado para describir datos, separando el contenido de la presentación, a diferencia de HTML. Un archivo XML se almacena como texto plano legible por cualquier editor.

El XML es el formato base de la mayor parte de las tecnologías usadas por las aplicaciones web. No es el propósito de este libro describir el XML ya que hay libros dedicados sólo a este formato como [Par09] o [AG12], presentaremos no obstante un resumen mínimo para seguir el contenido del libro y dos nuevos formatos que podrían sustituir al XML.

1.3.1. Estructura de un documento XML

Las primeras líneas de un archivo XML se llaman *Prólogo* y son opcionales. El prólogo puede definir la versión del XML, la codificación del texto plano y la gramática con la que se va a validar el documento, o el lugar donde encontrarla.

El resto del documento es un árbol, que sólo tiene una raiz. Cada elemento puede contener atributos y otros elementos anidados en él. Las cadenas de caracteres van delimitadas por comillas, que pueden ser simples o dobles. No puede ocurrir que un elemento *B* se cierre después de que se cierre el elemento *A* que lo contiene, es decir, esto es válido: *<A>...<B>...</B>...</A>*, pero esto no: *<A>...<B>...</A>...</B>*.

Los identificadores tienen que cumplir ciertas normas en XML:

- Se distingue entre mayúsculas y minúsculas, lo que significa que los identificadores *identificador* e *Identificador* son distintos.

- Un identificador no puede empezar por un número o un signo de puntuación (exceptuando "_") o la secuencia de caracteres *xml*. Los caracteres que van después del primero sí pueden ser signos de puntuación.

- No puede contener espacios en blanco.

Character DATA (CDATA)

Es la parte de un documento XML que contiene una cadena de caracteres y en consecuencia no debe interpretarse por el parser como si fueran marcas XML.

Es posible que un elemento indique su contenido de este modo: <*elemento attr="valor del atributo"*>*este es el contenido* </*elemento*>. Cuando un elemento no tiene contenido, tenga o no atributos, se abre y se cierra en la misma línea: <*elemento*/> ó <*elemento atributo="valor del atributo"*/>. Tanto la cadena *este es el contenido* como *valor del atributo* son CDATA.

Parsed Character DATA (PCDATA)

Son datos que van a ser interpretados por el parser y que pueden tener una mezcla de elementos XML y cadenas de caracteres.

1.3.2. Document Type Definitions (DTDs)

Un DTD es un conjunto de metadatos sobre archivos XML que indican al parser la estructura del documento y una lista de los elementos y atributos permitidos, o lo que es lo mismo, una gramática. Los DTDs son opcionales y por tanto, un documento XML no tiene porque referenciar a ninguno, aunque exista.

Las ventajas de los DTDs son su sencillez, que son compactos y que pocas veces se hace necesario utilizar algo más complejo, aunque más descriptivo como un XML Squema.

Referencias a un DTD en el prólogo

Es posible incluir un DTD en el propio documento XML, lo que se llama referencia privada, véase el listado 1.1, aunque tiene más sentido que sean archivos separados, que es una referencia pública, véase el listado 1.2.

La referencia al DTD empieza con esta sintaxis: *<!DOCTYPE **elemento-raíz** ...>*. El *elemento-raiz* es el nombre del elemento que está en la raíz del documento XML.

```
<?xml version=``1.0'' encoding=``UTF-8''?>
<!-- DTD de ejemplo para ClassXML -->
<!DOCTYPE Class SYSTEM ``class.dtd''>
[ <!ELEMENT Class...
...]
<Class>
...
```

Figura 1.1: Referencia privada en XML.

```
<?xml version=``1.0'' encoding=``UTF-8''?>
<!DOCTYPE Class
      PUBLIC ``-//W3C//DTD XHTML 1.0 STRICT/EN''
      ``http://www..../class.dtd''>
<Class>
...
```

Figura 1.2: Referencia pública en XML.

Elementos

Un *ELEMENT* define una marca, que es equivalente a la expresión a su derecha. La sintaxis es: *<!ELEMENT **nombre-del-elemento definición-del-elemento***>. Lo importante es la definición del elemento.

Símbolos para expresar la múltiplicidad: En la especificación de una gramática se suelen utilizar símbolos especiales para la multiplicidad con la que aparecen determinados elementos, así, escribir *(elemento)?* significa que el elemento puede aparecer o no, *(elemento)*+ significa que el elemento aparece entre una y muchas veces y, por último, *(elemento)** significa que el elemento aparece entre cero y muchas veces.

Tipos de elementos:

- *EMPTY*: El elemento no puede contener a otros, aunque sí puede tener atributos. Ejemplo: <*!ELEMENT Elemento EMPTY*>. Esto significa que el elemento *Elemento* tiene que tener el cierre (/>) al final de la línea.
- *#PCDATA*: El elemento sólo contiene texto plano en su interior y no puede contener a otros elementos. Ejemplo: <*!ELEMENT text (#PCDATA)*>.
- *Elementos que sólo pueden contener un tipo de hijo*: La sintaxis es: <*!ELEMENT padre (hijo)*>. El elemento hijo puede tener un signo ?, + o * opcional para indicar su multiplicidad. Ejemplo: <*!ELEMENT Items ((Item)+)*> el elemento *Items* sólo puede tener como hijos a entre 1 y varios elementos *Item*.
- *Elementos que contienen una secuencia de elementos*: La sintaxis es: <*!ELEMENT padre (hijo-1, hijo-2,...,hijo-n)*>.
- *Elementos que contienen una opción o una secuencia de ellas*: La sintaxis es: <*!ELEMENT padre (hijo-opt-1 | hijo-opt-2 |...| hijo-opt-n)*>.
- *ANY*: Es el tipo más genérico, porque significa que el elemento no es vacío y puede contener cualquier tipo de información.

Atributos

El comando *ATTLIST* permite definir los atributos que puede tener un *ELEMENT*, su tipo, si son obligatorios, valor por defecto, etc.

La sintaxis es: <*!ATTLIST elemento nombre-atributo tipo-atributo valores-atributo*>.

Valores de los atributos: Vamos a definir primero los valores de los atributos porque los usaremos para definir los tipos de atributos y porque son más sencillos.

- *Valor por defecto*: Cadena delimitada por comillas con el valor por defecto, véase el ejemplo 1.3.
- *#IMPLIED*: El atributo es opcional y si no aparece su valor es indefinido.

- *#REQUIRED*: El atributo siempre tiene que estar presente, aunque su valor sea una cadena vacía.
- *#FIXED*: Ademas de estar siempre presente, su valor está prefijado a un valor concreto, es la forma de definir una constante en un DTD.

Tipos de atributos: Se pueden agrupar en tres categorías: cadenas, tokens y enumerados.

1. **Cadenas**,
   - *CDATA*, como se ha dicho, es una cadena de caracteres que el parser no va a interpretar. Ejemplo:
     <!ATTLIST Method type CDATA "void" >.
     El tipo de un método es una cadena de caracteres, y si no se especifica, se asume que su valor es *void*.

```
1  <Method name="setVariable">
2       ...
3  <Method name="getVariable" type="Variable">
```

Figura 1.3: Aparición de un atributo *CDATA* con y sin valor por defecto.

2. **Tokens**.
   - *ID*: El valor es un identificador cuyo valor no se puede repetir en ningún lugar del documento. El valor por defecto sólo puede ser *#REQUIRED* o *#IMPLIED*. Ejemplo:
     <!ATTLIST Persona NIF ID #REQUIRED >.
   - *IDREF*: Sirve para referenciar el valor de un atributo del tipo *ID* definido previamente, véase el ejemplo 1.4.
   - *IDREFS*: Es una lista de *IDREF* separados por espacios.
   - *ENTITY*: El valor del atributo es una referencia a una entidad que puede ser interna, véase el ejemplo 1.6 o externa, véase el ejemplo 1.5; en este último caso no se analiza. Se puede utilizar, por

```
<?xml version=``1.0'' standalone=``yes''?>
<!DOCTYPE plantilla [
  <!ELEMENT plantilla (nombre)*>
  <!ELEMENT nombre (#PCDATA)>
  <!ATTLIST nombre NIF ID #REQUIRED>
  <!ATTLIST nombre jefe IDREF #IMPLIED>
]>
<plantilla>
  <nombre NIF=``12987322K''>Luis Delgado
  </nombre>
  <nombre NIF=``35876743I''
    jefe=``12987322K''>Manuel Navarro
  </nombre>
</plantilla>
```

Figura 1.4: Ejemplo de un DTD y un XML correcto con IDREF.

ejemplo, para referenciar una imagen dentro de un archivo de texto representado en un XML. Sintaxis: *<!ENTITY nombre-entidad "valor-entidad">*.

- *ENTITIES*: Al igual que con *IDREFS*, permite múltiples entidades separadas por espacios en blanco.
- *NMTOKEN*: El atributo es un conjunto de caracteres sin espacios. El nombre *NMTOKEN* es una abreviatura de *named token*. Es similar a *CDATA*.
- *NMTOKENS*: Permite varios *NMTOKEN* separados por espacios en blanco.

3. **Enumerados**

- *NOTATION*: El valor del atributo se ajusta a una notación. El primer carácter debe ser una letra, '_'ó ':', véase el ejemplo 1.7.
- Enumerado: El valor del atributo es un enumerado de una lista, véase el ejemplo 1.8.

```
<?xml version=``1.0'' standalone=``no''?>
<!DOCTYPE universidades [
  <!ELEMENT universidades (universidad)*>
  <!ELEMENT universidad EMPTY>
  <!ATTLIST universidad imagen ENTITY #REQUIRED>
  <!ENTITY uned SYSTEM
    ``http://portal.uned.es/NUEVOWEB/
      IMAGENES/logo_uned.gif''>
]>
<universidades>
  <universidad imagen=``uned''/>
<universidades>
```

Figura 1.5: DTD y un XML correcto con una entidad externa.

```
<!ENTITY pi-dir "/home/manuel/
   Asignaturas/ProgramacionIntegrativa/">
```

Figura 1.6: DTD y un XML correcto con una entidad interna.

```
<!ATTLIST imagen formato
   NOTATION (gif|jpeg|png) #REQUIRED>
```

Figura 1.7: Ejemplo de la etiqueta NOTATION.

```
<?xml version="1.0"?>
<!DOCTYPE UML [...
  <!ELEMENT ClaseRobustez (#PCDATA)>
  <!ATTLIST ClaseRobustez TipoClase
     (Entidad|Frontera|Control) #REQUIRED>
...]>
```

Figura 1.8: Ejemplo del uso de enumerados en DTDs.

1.3.3. Documentos esquema

Es un lenguaje para describir gramáticas de ficheros en XML. Su extensión es *.xsd*. Fue desarrollado por el W3C entre 1998 y 2001. Los esquemas son una aproximación posterior que permite superar algunas limitaciones de los DTDs. Ventajas de los esquemas:

- Usan la misma sintaxis que el XML.
- Se pueden definir tipos de datos.
- Son extensibles.

Cualquier especificación que se pueda hacer con un DTD se puede hacer con un esquema, pero no al revés. El único inconveniente de los esquemas es que son más complejos; por este motivo y porque cualquier DTD se puede traducir automáticamente a un esquema, si lo que se está haciendo es sencillo, se prefieren los DTDs, sobre todo por parte de aquellos que aprendieron a hacer DTDs antes de que existieran los esquemas.

La recomendación del W3C divide la documentación en tres partes:

1. XML Schema Parte 0 Primer: Introducción con ejemplos y explicaciones.
2. XML Schema Parte 1 Structures: Descripción de los componentes.
3. XML Schema Parte 2 Datatypes: Tipos de datos y restricciones.

Sólo daremos una breve introducción, correspondiente a la parte 0.

En el listado 1.9 tenemos un ejemplo mínimo para definir una clase.

Al igual que en los DTDs, tenemos que especificar cómo se va a validar el documento. Hacemos referencia al espacio de nombres en la línea 2 del listado 1.10.

Elementos: Para especificar un elemento utilizamos la etiqueta *xsd:element*. Existen dos tipos de elementos: simples y complejos. Un elemento *simple* sólo puede contener texto, no atributos ni otros elementos. La sintaxis de los elemento simples es <*xsd:element name=“nombre” type=“tipo”*>. En la tabla 1.1 están recogidos los tipos de datos que se suelen utilizar. La sintaxis para es-

```
<?xml version="1.0"?>
<clase nombre="ProbNet"
   xmlns="x-schema:claseSchema.xml">
  <atributos>
    <atributo nombre="probNodes"
      tipo="ArrayList<ProbNode>" alcance="privado"/>
  </atributos>
  <metodos>
    <metodo nombre="setPotential"
       alcance="publico">
      <parametros>
        <parametro nombre="potential" tipo="Potential"/>
      </parametros>
    </metodo>
    <metodo nombre="getPotentials"
       alcance="publico" tipo="List<Potential>">
      <parametros>
        <parametro nombre="variable" tipo="Variable"/>
      </parametros>
    </metodo>
  </metodos>
  <herencia>
    <padre>Network</padre>
  </herencia>
  <comentario>
Un Potential contiene una lista ordenada de Variables.
Cada ProbNode esta asociado a una Variable X y
contiene los Potenciales cuya primera Variable es X.
  </comentario>
</clase>
```

Figura 1.9: XML que define la clase *ProbNet*.

```
<xsd:schema
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="clase">
    <xsd:complexType>
      <xsd:sequence>
           <xsd:element name="to" type="xsd:string"/>
              <xsd:element name="from"
                 type="xsd:string"/>
              <xsd:element name="heading"
                 type="xsd:string"/>
              <xsd:element name="body"
                 type="xsd:string"/>
        </xsd:sequence>
       </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figura 1.10: Schema correspondiente a 1.9

Tabla 1.1: Tipos de datos simples.

Tipo
xsd:string
xsd:decimal
xsd:integer
xsd:boolean
xsd:date
xsd:time

pecificar el valor por defecto de un elemento es: <*xsd:element name="type" type="xsd:string" default="void"*/>.

Los elementos complejos pueden contener atributos.

Atributos: Son muy parecidos a los elementos simples. Su sintaxis es <*xsd:attribute name="nombre del atributo" type="tipo del atributo"*/>. Los tipos son los mismos que los de los elementos. Un atributo, por defecto es opcional. Para indicar lo contrario se utiliza *use* en la declaración, por ejemplo: <*xsd:attribute name="nombre" type="xsd:string" use="required"*/>.

Los atributos también pueden tener valores por defecto, añadiendo la palabra *default* a la sintaxis anterior del mismo modo que con los elementos.

Facetas: Son la forma de restringir los valores que puede tomar elementos y atributos. Una faceta puede ser:

- Un rango, véase el ejemplo del listado 1.12.
- Un conjunto de valores, véase el ejemplo del listado 1.11.
- Un conjunto de caracteres.
- Longitud de una cadena.
- ...

Elementos complejos: Al igual que en los DTDs, existe una forma de indicar que los elementos pueden contener a otros elementos, véase el listado 1.13.

Reutilización de sintaxis: En ocasiones hay elementos que tienen la misma sintáxis que otros, la etiqueta *type* es una forma de re-utilizar el código escrito para otro elemento, véase el listado 1.14.

Una forma más avanzada de reutilización, similar a la herencia, permite ampliar un elemento complejo usando un elemento definido anteriormente, véase el listado 1.15.

```
<xsd:element name="tipos-alcance-java">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="public"/>
      <xsd:enumeration value="private"/>
      <xsd:enumeration value="protected"/>
      <xsd:enumeration value="package"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Figura 1.11: Faceta para indicar el tipo de alcance de un atributo o método en Java.

```
<xsd:element name="letra-dni">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[A-Za-z]"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Figura 1.12: Faceta que indica que el valor debe ser una letra.

```
<xsd:element name="class">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="atributos" type="xsd:string"/>
      <xsd:element name="metodos" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Figura 1.13: Declaración de un esquema con un elemento complejo que contiene a otros.

Tabla 1.2: Restricciones aplicables a las facetas de los esquemas.

Tipos de restricciones	
enumeration	lista de valores.
fractionDigits	Número de cifras defimales.
length	Número de caracteres.
maxInclusive, maxExclusive	Máximo valor de un rango, incluido y no incluido.
minInclusive, minExclusive	Máximo valor de un rango, incluido y no incluido.
maxLength, minLength	Número máximo y mínimo de caracteres permitidos.
totalDigits	Idem para dígitos.
pattern	Secuencia de caracteres permitida.
whitespace	Caracteres en blanco.

```
<xsd:element name=``alcance-atributo''
   type=``alcance''/>
<xsd:element name=``alcance-metodo''
   type=``alcance''/>

<xsd:complexType name=``alcance''>
  <xsd:sequence>
    <xsd:element name=``private''
       type=``xsd:string''/>
    <xsd:element name=``protected''
       type=``xsd:string''/>
    <xsd:element name=``public''
       type=``xsd:string''/>
    <xsd:element name=``package''
       type=``xsd:string''/>
  </xsd:sequence>
</xsd:complexType>
```

Figura 1.14: Reutilización de sintaxis 1.

```
<xsd:element name="asociacion"
   type="asociacion-completa"/>

<xsd:complexType name="extremos-asociacion">
  <xsd:sequence>
    <xsd:element name="nombre-clase-1"
       type="xsd:string"/>
    <xsd:element name="nombre-clase-2"
       type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="asociacion-completa">
  <xsd:complexContent>
    <xsd:extension base="extremos-asociacion">
      <xsd:sequence>
        <xsd:element name="rol-1" type="xsd:string"/>
        <xsd:element name="rol-2" type="xsd:string"/>
        <xsd:element name="nombre" type="xsd:string"/>
        <xsd:element name="card-1" type="xsd:string"/>
        <xsd:element name="card-2" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figura 1.15: Reutilización de sintaxis 2.

Atributos: En el listado 1.16 se puede ver un ejemplo de un elemento vacío con atributos, aunque es más habitual que un elemento además de atributos contenga información propia.

```
<xsd:element name="persona">
  <xsd:complexType>
    <xsd:attribute name="edad"
      type="xsd:positiveInteger"/>
  </xsd:complexType>
</xsd:element>
```

Figura 1.16: Definición de atributos en un esquema.

1.3.4. Extensible Stylesheet Language (XSL)

Es un conjunto de tres lenguajes que definen cómo se tiene que presentar una información contenida en un archivo XML. Los tres lenguajes son:

- *Extensible Stylesheet Language Transformations* (XSLT): Define cómo transformar un documento XML en otro tipo de documento.
- *eXtensible Stylesheet Language Formatting Objects* (XSL-FO): Define la presentación en pantalla o papel de los datos de un archivo XML. Un programa llamado *procesador de XSL-FO* es el que obtiene el documento final en distintos formatos; el más usual es el PDF.
- *XML Path Language* (XPath): Es una sintaxis no XML para referirse a fragmentos de un archivo XML.

1.4. Lectura de documentos XML

El gran éxito del XML ha traído consigo una gran cantidad de herramientas para editar, diseñar o *parsear*[3] un fichero XML.

[3]Entenderemos por *parsear*, leer un archivo XML desde un programa. Usaremos la palabra *procesar*, en este contexto, como un sinónimo de parsear.

Las características deseables en un editor XML son el coloreado de la sintaxis, la posibilidad de desplegar/colapsar ramas del árbol y que tenga capacidades de autocompletar texto usando una gramática (DTD, Esquema u otro). Por ejemplo, los editores TDTD o EZDTD validan un XML con una gramática en DTD.

El diseño de un formato XML supone diseñar una gramática, y es un tema complejo. Merece la pena comprar una herramienta comercial porque las soluciones que ofrece el software libre son un tanto incompletas.

La lectura de un XML desde un programa se hace con APIs. Hay dos tipos: basadas en eventos, que son generados por los elementos y atributos del documento que se está leyendo, y basadas en DOM, que permite acceder a cualquier parte del documento en cualquier momento.

1.4.1. Simple API for XML (SAX)

SAX es un API del primer tipo. Al principio, SAX fue una API para leer documentos XML en Java, pero con el paso del tiempo se ha convertido en un estándar para leer documentos XML. Actualmente existen versiones para otros lenguajes.

SAX es capaz de detectar el comienzo y fin de elementos, gestionar espacios de nombres y comprobar que el documento está bien formado.

La lectura que hace SAX del documento es secuencial. Esto tiene la ventaja de la eficiencia, al ser rápido y necesitar menos memoria por no almacenar la estructura del árbol. La contrapartida es que no permite volver atrás, a diferencia de DOM. Como SAX está basado en eventos, el lector tiene que saber de algún modo el lugar del documento en el que está, lo que puede ser complejo.

Streaming API for XML (StAX)

StAX define un “cursor” que lee la información a medida que el analizador lo va pidiendo. La diferencia con SAX es que no está basado en eventos, lo que simplifica los parsers, aunque la lectura sigue siendo secuencial.

1.4.2. Document Object Model (DOM)

DOM es un estándar del W3C que define el modo en que un parser debe mostrar la información que lee de un archivo XML. DOM representa el XML en

un árbol de objetos en memoria, al que se accede con un interfaz. Es más sencillo que SAX o StAX aunque consume más recursos.

1.5. Alternativas al XML

Aunque el XML es un estándar que ha alcanzado un bien merecido éxito, tiene algunos problemas: la sintaxis es algo redundante, la construcción de parsers no es tan trivial como parece (o como podría ser), no soporta un gran conjunto de tipos de datos primitivos, por lo que los nuevos hay que definirlos con un esquema y por último, un fichero de texto plano en XML es difícil de leer por humanos.

1.5.1. JavaScript Object Notation (JSON)

Es un formato basado en un subconjunto del lenguaje JavaScript, aunque es independiente de este lenguaje. Al igual que XML es texto plano. Las principales ventajas sobre XML son que es más fácil construir parsers y ocupa algo menos. Por otro lado no tiene espacios de nombres, no es extensible, no tiene validador y está soportado por prácticamente todos los lenguajes de programación actuales. Al ser JSON un subconjunto de JavaScript, éste permite analizarlo con la función *eval()*, que permite transformar cadenas de texto a objetos.

Tipos de datos

En JSON se pueden definir dos tipos de estructuras de datos básicas: una colección de pares (clave, valor) y una lista ordenada de variables. Estas estructuras pueden combinarse entre ellas por anidación.

Tipos de datos en JSON:

- *string*: Conjunto de caracteres unicode (por defecto UTF-8) delimitados por comillas. Las comillas, caracteres de control y los símbolos poco habituales de unicode también se pueden representar.

- *number*: Utiliza la misma sintaxis que en los lenguajes de programación C o Java, excluyendo las representaciones octal y hexadecimal.

- *value*: Puede ser uno cualquiera de estos tipos: *string, number, object, array, true, false* y *null*.

- *object*: Es un conjunto no ordenado de pares (clave, valor), siendo *clave* un *string*. El conjunto está delimitado por llaves y los elementos se separan por comas. Ejemplo: { “Jorgito”:1 , “Juanjito”:2, “Jaimito”:3 }.

- *array*: Es un conjunto ordenado de valores. El conjunto está delimitado por corchetes y, como en los objetos, los elementos se separan por comas, véase el ejemplo 1.17.

```
[
  {
    "Nombre":"Manuel",
    "Hobby":"Starcraft II",
    "Software preferido":["Linux", "LaTeX"]
  },
  {
    "Nombre":"Brigitte",
    "Hobby":"Senderismo",
    "Software preferido":["Windows", "Word"]
  }
]
```

Figura 1.17: Array de objects en JSON.

1.5.2. YAML

Es un lenguaje para serializar objetos. Las siglas son un acrónimo de *YAML Ain’t Another Markup Language*, que se traduce como *YAML no es otro lenguaje de marcado*. El lenguaje es aun más legible por humanos que JSON, también usa caracteres unicode (UTF-8 o UTF-16) y su estructura se define con caracteres en blanco, sin permitir tabuladores.

Tipos de datos

Al igual que con JSON, vamos a definir por encima el modo de definir estructuras de datos sencillas. En el listado 1.18 podemos ver un ejemplo mínimo.

- El principio de documento se especifica con tres guiones: ---; el fin de documento con tres puntos: ...
- Listas: Cada miembro va encabezado por un guión o bien toda la lista entre corchetes y con sus miembros separados por comas.
- Conjuntos con elementos (clave, valor): Cada línea los dos elementos con esta sintaxis *clave: valor*, sin olvidar el espacio entre los dos puntos y el valor. Otra opción es entre llaves y separados por comas.
- Una lista asociativa empieza con el signo de interrogación.
- Comentarios: Tienen el caracter # al principio de cada línea.
- Es posible etiquetar un nodo con el signo de exclamación y una cadena.

```
1 --- # Personas y hobbies
2 - Nombre: Manuel
3 Hobby: Starcraft II
4 Software preferido: [Linux, LaTeX]
5 - Nombre: Brigitte
6 Hobby: Senderismo
7 Software preferido: [Windows, Word]
```

Figura 1.18: Lista de listas asociativas en YAML.

Lecturas recomendadas

En [Ang10] se analiza como describir recursos electrónicos usando metadatos.

En [SG02] se describe en detalle el lenguaje XML, los espacios de nombres, los DTDs y los esquemas, XPath, XSL, SAX y otros temas de interés.

Recursos y tutoriales en la web:

- XML: `http://www.w3schools.com/xml/`
- DTDs: `http://www.w3schools.com/dtd/`
- Schemas: `http://www.w3schools.com/schema/` y `http://www.xml.com/pub/a/2000/11/29/schemas/part1.html`
- jDom: `http://www.jdom.org/`

1.6. Autoevaluación

1.6.1. Preguntas

Este tema es muy teórico, en consecuencia las preguntas de autoevaluación son sobre todo referencias a distintas partes del texto, aunque en lo posible, se ha procurado hacer preguntas no evidentes.

1. ¿Qué es un metadato?
2. Tenemos un paquete del nivel de red formado por:

   a) Dirección de destino.

   b) Dirección de origen.

   c) Tiempo de vida en milisegundos.

   d) Puerto TCP.

   e) Datos de usuario.

   ¿Qué tipo de metadatos son cada uno de los cinco tipos de datos enumerados?
3. ¿Qué es la dependencia del proveedor?
4. ¿Cómo se puede hacer uso de un formato propietario sin conocer sus especificaciones?
5. En general ¿qué problemas tiene la conversión de un tipo de formato en otro?
6. ¿Qué diferencia existe entre CDATA y PCDATA?
7. Proponga un formato en XML para representar un archivo de configuración de un programa que está hecho en base a componentes. Cada componente puede tener o no sub-componentes y cada componente guarda información del tipo (nombre, valor).
8. Diseñe un DTD para el ejercicio anterior. Suponer que existe la restricción de que el nombre de un componente o sub-componente no se puede repetir.

1.6.2. Respuestas

1. Véase la sección 1.1.

2. Las direcciones de origen y destino son metadatos administrativos, porque sirven a los nodos de la red para encaminar los paquetes por la línea adecuada, o sea, gestionarlos. El mismo razonamiento se puede aplicar para el puerto TCP y el tiempo de vida en milisegundos, porque sirve para descartar o no el paquete, que es parte de la gestión. Los datos del usuario no son metadatos.

3. Véase la sección 1.2.4.

4. Haciendo ingeniería inversa, lo que significa estudiar, a nivel de bit. archivos generados en el formato propietario e ir generando todos los casos particulares de las estructuras de datos que se vayan descubriendo. Supone mucho trabajo y sólo tiene éxito al 100 % en los formatos más simples.

5. En ocasiones puede no ser reversible, por ejemplo, convertir una imagen desde un formato vectorial a un mapa de bits. Otro problema puede ser la pérdida de información, por ejemplo, desde un mapa de bits a un jpg con pérdida.

6. En las secciones 1.3.1 y 1.3.1 se han definido ambos términos. También se puede ver como la distinción entre metadatos y datos en los ficheros XML. La diferencia consisten en que en un caso la información se procesa por el parser y en el otro no.

7. En el listado 1.19 se representa esquemáticamente cómo sería un archivo de configuración.

8. El listado 1.20 es una posible solución.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuracion>
   <Componente nombre = "componente-nombre-1">
      <Variable nombre="variable-1"
           type="String" valor="valor1"/>
      <Variable nombre="variable-2"
           type="Numerico" valor=1/>
   </Componente>
   <Componente nombre = "component-nombre-2">
      <Variable nombre="variable-3"
           type="String" value="valor2"/>
      <Componente nombre = "sub-componente-2.1">
         <Variable nombre="variable-3.1"
          type="String" value="valor2.1"/>
      </Componente>
   </Componente>
</Configuracion>
```

Figura 1.19: Archivo de configuración en XML.

```
<?xml version=``1.0'' standalone=``no''?>
<!DOCTYPE configuracion [
  <!ELEMENT configuracion (componente)*>
  <!ELEMENT componente (variable,componente)*>
  <!ATTLIST componente nombre CDATA ID #REQUIRED>
  <!ATTLIST variable nombre CDATA #REQUIRED>
  <!ATTLIST variable tipo CDATA #REQUIRED>
  <!ATTLIST variable valor CDATA #REQUIRED>
]>
```

Figura 1.20: DTD del archivo de configuración.

1.7. Cuestiones abiertas

1. Suponga que tiene un entorno de programación que permite guardar el código fuente en varios formatos. Discuta ventajas e inconvenientes del ASCII, UTF-8, 16 y 32. ¿Se decantaría por alguno para guardar el código fuente o es indiferente?

2. Siguiendo la definición de *Dependencia del proveedor* en 1.2.4. Cite algún ejemplo en el que haya ocurrido este fenómeno.

3. Entre los formatos gráficos se pueden hacer conversiones. Algunas veces un archivo en el formato A se puede convertir al formato B pero el proceso inverso o no es posible o se pierde calidad. Proponga una representación, que puede ser en forma de grafo (nodos, enlaces, ...), o de clases de equivalencia, o algo mixto, de los formatos propuestos en el texto desde el punto de vista de la conversión de un formato a otro, representando, por ejemplo en los enlaces, si es un grafo, la problemática de la pérdida de información. Puede ampliar el número de formatos gráficos propuestos en el texto.