COPIA LITERAL

Extracto del libro “GAMBAS, programación visual con Software Libre”, de la editorial EDIT LIN EDITORIAL S.L., cuyos autores son Daniel Campos Fernández y José Luis Redrejo.

7. XML

Hoy en día es común oír hablar de XML como la solución a todos los problemas de gestión informática en la empresa. Es cierto, desde luego, que se encuentra sobrevalorado, posiblemente debido a las campañas de marketing realizadas por grandes compañías de software.

No obstante, también es cierto que XML presenta algunas ventajas importantes a la hora de intercambiar datos entre diferentes sistemas. Pero ¿qué es XML? Pues en realidad no es nada nuevo ni revolucionario, se trata simple y llanamente de una definición de formato para cualquier tipo de documentos.

Cuando necesitamos escribir un fichero de configuración, un documento con registros extraídos de una base de datos, comunicar datos a un dispositivo, un formato para un fichero de texto o una hoja de cálculo, siempre se le plantea el mismo

problema al programador: la necesidad de un formato, documentado en el caso ideal, que defina el modo en que esa información se inserta en un fichero o en un flujo de datos a través de una red, con el fin de que los programas origen y destino de esos datos puedan escribir o leer cada parte del documento correctamente. Muchas veces escribir un fichero con un determinado formato es una tarea casi trivial, pero el proceso inverso, la lectura, suele ser engorroso, dado que hay que comprobar errores, trocear cadenas, comprobar la validez de cada fragmento, etc.

Por otra parte, cada aplicación ha desarrollado hasta hace poco sus propios formatos para el almacenamiento y lectura de la información que maneja, haciendo muy difícil eliminar un elemento de una cadena de programas para reemplazarlo por una aplicación nueva. Esto tiene una importancia grande cuando una empresa se plantea el paso de sistemas propietarios a los que se encuentran encadenados, hacia otros libres donde obtienen ventajas de precio, y capacidad de elección de proveedores y servicios más justos por la inversión realizada. Los formatos propietarios atan literalmente al cliente a los servicios y deseos de una empresa proveedora.

XML proporciona ayuda en todos los aspectos comentados: es un estándar que define cómo se han de insertar los datos y los campos en un fichero o flujo. Además, las herramientas de gestión de documentos XML proporcionan las funciones necesarias para leer, escribir y verificar los datos embebidos en dichos documentos. Por último, se trata de un estándar accesible a todos los programadores y casas de software, lo que proporciona libertad para manejar y comprender el contenido de los datos.

El formato de un documento XML es similar a uno escrito con HTML, no obstante hay diferencias sustanciales. La primera, y más importante, es que XML es un formato de carácter general, pensado para trabajar con cualquier tipo de datos, mientras que HTML se encuentra limitado al diseño de páginas web. La segunda es que, a diferencia de HTML, donde hay etiquetas, como la de párrafo nuevo (*<p>*) que se suelen dejar abiertas, en XML absolutamente todas las etiquetas deben estar anidadas, y cada etiqueta abierta debe cerrarse. Por otra parte, XML es sensible a mayúsculas, es decir, una sección que comienza con la etiqueta *<p>* no es igual a otra que comienza con la etiqueta *<P>*.

El aspecto de un fichero XML simple, puede ser éste:

```
<?xml version="1.0"?>
<datos>
 <usuario>
  <nombre>Eric Smith</nombre>
  <socio>113</socio>
 </usuario>
</datos>
```

Los documentos XML siempre comienzan, en la actualidad, por la cadena **<?xml version="1.0"?>**, en la cual se especifica que a continuación vienen datos con formato XML. Aunque esta etiqueta puede ser obviada, siempre es conveniente añadirla, ya que contiene información importante. En este caso se especifica la versión de XML que se está utilizando, la **1.0**, y que es la única que se emplea en la actualidad. Si en el futuro una nueva especificación internacional de XML ampliara o modificara XML, con una versión 2.0, por ejemplo, disponer de esa etiqueta en un documento almacenado hace meses o años garantizará que los programas sigan sabiendo cómo interpretar lo que contiene un documento. Por otra parte, los documentos XML, por defecto, emplean la codificación de caracteres UTF-8. Si por cualquier razón hemos de tratar con documentos XML que empleen otra codificación, también encontraremos esa información en la etiqueta inicial:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

A continuación, llega el cuerpo del documento XML. Hay una etiqueta inicial que da nombre al documento, en este caso simplemente **datos**. Después, en este documento vienen los datos de los usuarios de una asociación: el nombre y el número de asociado. Todo ello encerrado entre etiquetas abiertas y cerradas, de modo que el documento queda ordenado en secciones, con varios niveles de anidación.

XML permite que las etiquetas dispongan de *atributos*. Por ejemplo, podemos prever que nuestro fichero de datos contemple un número de versión para futuras

ampliaciones, y que en cada usuario se añada el año en que entró a formar parte de la asociación. Los atributos tienen la forma:

```
definicion="valor"
```

Es decir, se indicará el nombre del atributo, un signo de igual y, encerrado entre comillas, el valor de ese atributo. El fichero podría ser ahora así:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<datos version="2.4">
 <usuario>
  <nombre inicio="2005">Eric Smith</nombre>
  <socio>113</socio>
 </usuario>
</datos>
```


Una etiqueta que se abra y cierre sin información en su interior, por ejemplo * * para designar un párrafo en un documento XHTML, puede simplificarse escribiendo la etiqueta de apertura con la barra al final del nombre de la etiqueta: * *.

Estos son los elementos más básicos de XML, pero más allá existe una amplia gama de conceptos que quedan fuera del alcance de este manual, y que aportan un gran número de posibilidades a la gestión de documentos XML: existen etiquetas especiales para comentarios, un lenguaje para validación de datos, DTD, hojas de estilo con formato XSL, espacios de nombres, etc. Un buen punto de inicio para aprender XML es la web **http://www.w3schools.com/xml/default.asp**, donde se encuentran tutoriales al respecto, así como enlaces a otras fuentes de información.

7.1 Escritura con XmlWriter

La clase *XmlWriter*, contenida dentro del componente *gb.xml*, aporta un modo sencillo para escribir un documento XML. Supongamos que deseamos almacenar en

un programa los datos relativos a varias conexiones telefónicas a Internet. Nos interesará el nombre de la conexión, si está disponible en todo el territorio nacional o está limitada a una localidad, el teléfono y las DNS, primaria y secundaria, aportadas por el proveedor.

Plantearemos un documento en el cual tendremos una sección *conexion* para cada conexión, que tendrá un atributo *nombre* y otro *local*, este último con dos valores: 1, si es local, o 0, si es nacional. Dentro de la sección *conexion* habrá un campo para almacenar el número de teléfono, así como otro para los DNS primario y secundario, los cuales tendrán un atributo que determina si es el *primario*.

Siempre es conveniente emplear nombres de etiquetas y atributos que no contengan acentos o caracteres especiales de cada idioma (como ñ o ç), para evitar problemas si otros programadores que deban retocar el programa o el fichero tienen teclados distintos o emplean *parsers* de XML pobres que ignoren algunos aspectos de la codificación.

Éste será el aspecto del fichero XML del ejemplo:

```
<?xml version="1.0"?>
<conexiones version="1.0">
 <conexion id="castuonet" local="0">
  <telefono>1199212321</telefono>
  <dns primario="1">127.0.0.2</dns>
  <dns primario="0">127.0.0.3</dns>
 </conexion>
 <conexion id="limbonet" local="0">
  <telefono>229943484</telefono>
  <dns primario="1">127.0.10.10</dns>
  <dns primario="0">127.0.20.42</dns>
 </conexion>
</conexiones>
```

Antes de continuar, observemos el formato. Está separado en varias líneas, o retornos de carro, y existe una indentación para indicar los distintos niveles de anidación. Esto no es en absoluto necesario, y los *parsers* de XML ignoran los espacios en blanco y tabulaciones, así como los retornos de carro. Esto es sólo una ayuda para quien tenga que consultar o modificar el fichero desde un editor de texto común. Sólo tiene, por tanto, sentido a nivel humano, y un fichero XML correcto podría estar contenido en una sóla línea de texto. Si estamos acostumbrados a programar en C o C++, sabremos que con este lenguaje ocurre lo mismo: las diferentes líneas y tabulaciones dan orden y legibilidad al programa, pero el mismo podría escribirse en una sola línea. En C, las llaves y puntos y comas son significativas, no así los espacios o retornos de carro. En XML lo significativo son las aperturas y cierres de etiquetas.

A continuación vamos a crear un programa de consola llamado **EscribeXML**, y dentro de él un único módulo **modMain** con una función *Main*.

El programa tendrá una referencia al componente **gb.xml**.

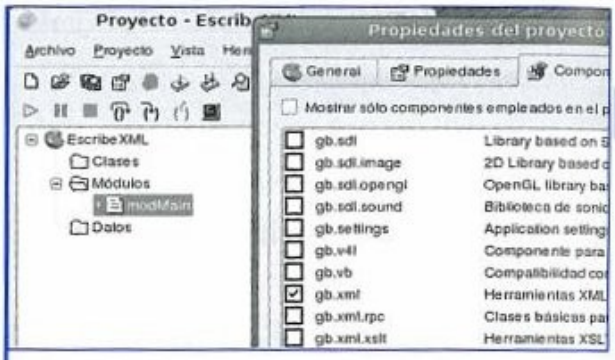

Figura 1. Proyecto **EscribeXML**.

Definimos y creamos un objeto **XmlWriter**. Tras esto, lo abrimos con **Open** para comenzar la escritura. Este método acepta tres parámetros: el primero, obligatorio, es el nombre del fichero a escribir, que puede ser una ruta dentro del sistema de archivos o una cadena en blanco, en cuyo caso se escribirá en memoria, y luego podremos obtenerlo como una cadena para, por ejemplo, enviarlo por la red a un equipo remoto. El segundo es opcional, e indica que deseamos que tenga indentación (como se comentó antes, para aumentar su legibilidad), o bien si se escribirá en una sola línea, para ahorrar espacio, a cambio de hacerlo ilegible desde un editor no especializado. El tercero, también opcional, sirve para, si se desea, especificar la codificación, diferente de UTF-8.

```
' Gambas module file

PUBLIC SUB Main()

  DIM Xml AS XmlWriter

  Xml = NEW XmlWriter
  Xml.Open("", TRUE)
```

Indicar una codificación diferente sirve para que ésta quede reflejada al inicio del documento, pero no realizará por nosotros la conversión de las cadenas que escribamos, tarea que habremos de hacer con funciones como *Conv$*, si procede.

Escribimos la primera sección. Hemos de introducir una etiqueta de apertura, tarea que se realiza con el método StartElement:

```
Xml.StartElement("conexiones")
```

Además dispone de un atributo "version" con valor "1.0":

```
Xml.Attribute("version","1.0")
```

En este caso también podemos unir las dos instrucciones anteriores en una sola: StartElement acepta un parámetro opcional, consistente en una matriz de cadenas que sean pares de valores atributo-valor del atributo:

```
Xml.StartElement("conexiones", ["version", "1.0"])
```

Dentro de esta sección tenemos cada una de las conexiones definidas. Comenzamos por la primera que, al igual que antes, es una etiqueta de apertura, con un nombre y dos atributos, en este caso:

```
Xml.StartElement("conexion", ["id", "castuonet",
"local", "0"])
```

Dentro de esta región, disponemos de una nueva apertura de etiqueta para el teléfono:

```
Xml.StartElement("telefono")
```

Contiene un texto que comprende el número de teléfono:

```
Xml.Text("1199212321")
```

Tras este paso cerramos la etiqueta:

```
Xml.EndElement()
```


No obstante, podemos resumir las tres instrucciones anteriores en una sola, válida para estos elementos simples que tan sólo contienen un texto dentro (o nada) y luego se cierran.

```
Xml.Element("telefono", "1199212321")
```

Para los DNS iniciamos una etiqueta "dns" con un atributo:

```
Xml.StartElement("dns", ["primario", "1"])
```

Incluimos el texto con la IP:

```
Xml.Text("127.0.0.2")
```

Y finalizamos la sección:

```
Xml.EndElement()
```

Lo mismo para el segundo DNS:

```
Xml.StartElement("dns", ["primario", "0"])
Xml.Text("127.0.0.3")
Xml.EndElement()
```

Finalmente, cerramos la etiqueta "**conexion**" que contiene los elementos anteriores relativos a esa conexión:

```
Xml.EndElement()
```

Procedemos de igual manera para la segunda conexión, tal y como podemos observar en el siguiente código:

```
Xml.StartElement("conexion", ["id", "limbonet", "local",
"1"])
Xml.Element("telefono", "229943484")
Xml.StartElement("dns", ["primario", "1"])
Xml.Text("127.0.10.10")
Xml.EndElement()
Xml.StartElement("dns", ["primario", "0"])
Xml.Text("127.0.20.42")
Xml.EndElement()
Xml.EndElement()
```

Finalmente cerramos el documento para que se escriba. Si hubiésemos especificado un nombre de fichero, sería ahora cuando se volcaría al fichero al haberlo hecho, en nuestro caso, en memoria.

La llamada a **EndDocument** nos devuelve una cadena con el documento XML, que mostramos por la consola:

```
PRINT Xml.EndDocument()
```

El programa completo queda de la siguiente manera:

```
' Gambas module file

PUBLIC SUB Main()

  DIM Xml AS XmlWriter

  Xml = NEW XmlWriter

  Xml.Open("", TRUE)

  Xml.StartElement("conexiones", ["version", "1.0"])

  Xml.StartElement("conexion", ["id", "castuonet",
  "local", "0"])
  Xml.Element("telefono", "1199212321")
  Xml.StartElement("dns", ["primario", "1"])
  Xml.Text("127.0.0.2")
  Xml.EndElement()
  Xml.StartElement("dns", ["primario", "0"])
  Xml.Text("127.0.0.3")
  Xml.EndElement()
  Xml.EndElement()

  Xml.StartElement("conexion", ["id", "limbonet",
  "local", "1"])
  Xml.Element("telefono", "229943484")
  Xml.StartElement("dns", ["primario", "1"])
  Xml.Text("127.0.10.10")
  Xml.EndElement()
  Xml.StartElement("dns", ["primario", "0"])
  Xml.Text("127.0.20.42")
```

```
    Xml.EndElement()
    Xml.EndElement()

    PRINT Xml.EndDocument()

END
```

l ejecutarlo, veremos el documento por la consola. Lo modificamos para que se scriba en un fichero en lugar de en la consola, lo cual nos servirá para leerlo a con-nuación en un ejemplo de lectura de fichero XML. Para ello modificaremos el méto-o **Open**:

```
Xml.Open(User.Home & "/conexiones.xml", TRUE)
```

eliminaremos la instrucción *PRINT* al final:

```
Xml.EndDocument()
```

n este caso, la llamada al método **EndDocument** volcará el documento en un fiche-o dentro de nuestra carpeta personal, llamado **conexiones.xml**.

El fichero abierto con *Open* no se vuelca realmente al disco duro hasta la llamada a *EndDocument*. A su vez, el método *EndDocument* se encarga de cerrar todas las etiquetas que hubieran quedado abiertas, para garantizar la coherencia XML de éste.

exploramos la clase *XmlWriter*, observaremos que, además de los métodos mentados, aporta otros para escribir comentarios y elementos correspondien-al lenguaje DTD, entre otras características. También dispone de los métodos áticos *Base64* y *BinHex*, que convierten una cadena a las codificaciones con estos mbres. Dichas codificaciones permiten introducir datos binarios dentro de un hero XML.

7. 2 Lectura con XmlReader

Modelos de lectura

Al menos existen tres métodos para leer los contenidos de un fichero XML:

1. Un método se basa en leer el fichero de principio a fin. El lector va generando eventos conforme se entra y sale de los distintos nodos del documento, y los gestores de eventos escritos por el programador van recibiendo la información. Esta forma de trabajo aún no se ha implementado en el componente *gb.xml*, aunque se prevé su inclusión en futuras versiones.

2. Otro método consiste en cargar el documento completo en memoria, para luego navegar por él, con lo cual se obtiene gran flexibilidad a costa de un consumo considerable de recursos del sistema. Este método está parcialmente implementado en Gambas a través de la clase *XmlDocument*, si bien su finalización no está prevista hasta futuras versiones, y no se recomienda su empleo para lectura de ficheros XML. No obstante, esta clase ya se emplea para transformaciones XSLT, como veremos más adelante en este capítulo.

3. Por último, el método que consume menos recursos y aporta bastante simplicidad de aprendizaje y uso, es disponer de un cursor, que sólo se mueve hacia adelante, de nodo en nodo, y que en cada momento podemos emplear para conocer el contenido y tipo de cada nodo. Si hemos trabajado con la plataforma .NET(TM) o Mono(TM), nos será familiar la clase *XmlReader* y sus derivadas, como *XmlTextReader*, que trabajan de la misma manera. Este modo de trabajo se encuentra perfectamente soportado en el componente *gb.xml* a través de la clase *XmlReader*.

Planteamiento inicial

En el caso de Gambas, la lectura de documentos la realizaremos utilizando objetos de la clase *XmlReader*.

El código de lectura resultará más complejo, al tener en cuenta varios aspectos:

- Rechazar cada fichero no válido: puede haber ficheros con formato no XML o que contienen datos sin sentido para nuestra aplicación.

- Ignorar datos no conocidos: es posible que un documento contenga datos que no nos interesan, pero se han añadido al fichero por otra aplicación en previsión de futuros usos (puede haber, por ejemplo, una etiqueta *<tarifa>* dentro de cada conexión). También una etiqueta conocida puede contener atributos desconocidos.

- Orden desconocido: en un fichero XML no es relevante el orden en que se escriben los nodos hijos de un nodo, es decir, que estos dos ejemplos deberían ser dados por válidos:

```
<conexion>
    <dns>...
   <telefono>...
   <dns>..
</conexion>

<conexion>
   <telefono>....
   <dns>...
   <dns>...
</conexion>
```

Si la aplicación espera encontrar el nodo **telefono** antes del nodo **dns**, fallará al tratar el primer fichero, que, sin embargo, contiene la misma información.

- Ignorar etiquetas sin interés para nuestra aplicación: XML, como indicamos brevemente al principio, prevé la posibilidad de añadir comentarios (similares a los comentarios de cualquier programa, sin uso para éste pero que aumentan la legibilidad), nodos DTD, etc. Habremos de pasar sobre estos nodos ignorándolos y sin presuponer si existen o no.

Cuantas más posibilidades añadamos a nuestro código de salvar lo desconocido, más flexible haremos nuestro lector XML para permitir la lectura de datos provenientes, tal vez, de programas escritos por varios programadores con los que no tenemos contacto, o que tienen pensamientos algo diferentes acerca del contenido del fichero.

Un ejemplo de lectura

Crearemos un proyecto gráfico llamado **LeeXML**, con un formulario **FMain**, que tendrá la propiedad *Arrangement* con el valor *Fill*, y un único control *TreeView* en su interior llamado **Arbol**. El programa contendrá una referencia al componente **gb.xml**.

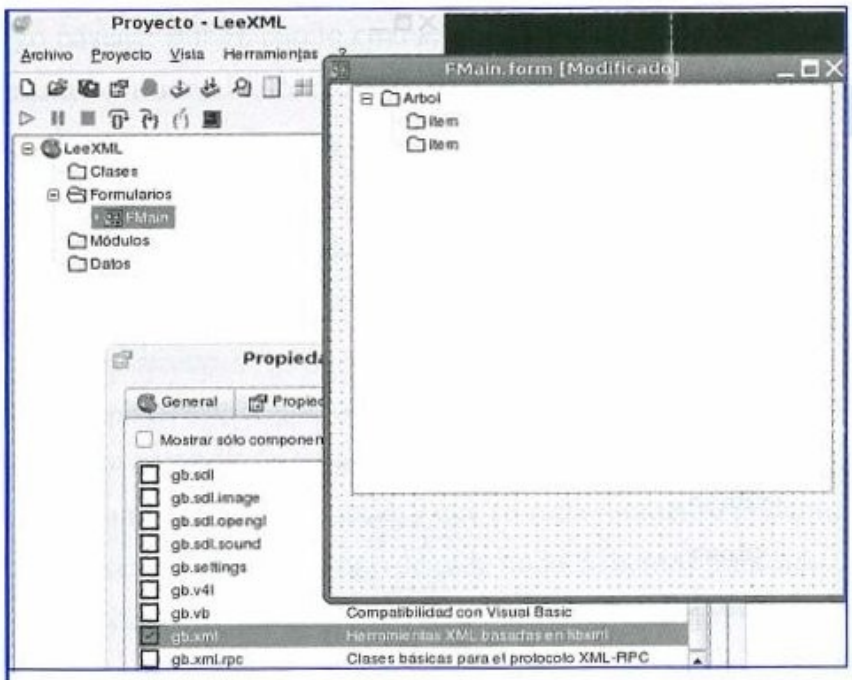

Figura 2. Proyecto **LeeXML**.

En la apertura del formulario leeremos el fichero XML. El método **Form_Open** quedará así:

- En primer lugar definimos el objeto **XmlReader**, lo creamos y tratamos de abrir el fichero XML. Si el fichero no existe, o no atiende a este formato, se generará un error en ese punto.

```
PUBLIC SUB Form_Open()

  DIM Xml AS XmlReader
  Xml = NEW XmlReader
  TRY Xml.Open(User.Home & "/conexiones.xml")
  IF ERROR THEN
    Message.Error("Fallo al abrir el fichero indicado")
    RETURN
  END IF
```

- Entramos en un bucle en el que leemos cada nodo avanzando por el contenido del fichero. Nos interesa encontrar el primero de tipo **Element** y que su nombre sea **conexiones**. De no ser así, el fichero no contendría datos de interés y lo rechazaríamos. Pero si es correcto, llamaremos a una función **RellenaArbol**, donde trataremos estos datos.

```
  DO WHILE TRUE

    IF Xml.Node.Type = XmlReaderNodetype.Element THEN

      IF Xml.Node.Name = "conexiones" THEN

        RellenaArbol(Xml)

      ELSE

        Message.Error("El documento no contiene datos
        de conexiones")
        Xml.Close()

      END IF

    END IF
```

Por cada iteración del bucle, empleamos el método **Read**, que sitúa el puntero interno en el siguiente nodo del fichero XML. En este proceso, puede darse un error si el puntero llega a una zona donde el fichero está corrupto, es decir, que no cumple la norma XML y, por tanto, no puede ser leído.

```
TRY Xml.Read()
IF ERROR THEN
  Message.Error("Formato XML no válido")
  RETURN
END IF
```

Si llegamos al final del fichero (tras el último nodo), terminamos el bucle. Esta circunstancia se puede conocer porque la propiedad **Eof** del objeto *XmlReader* toma el valor *TRUE*.

```
IF Xml.Eof THEN BREAK

LOOP
```


Tras la lectura del fichero, cerramos el objeto *XmlReader*.

```
TRY Xml.Close()

END
```

Vamos ahora a implementar el procedimiento **RellenaArbol**. Entramos de nuevo en el bucle y, en primer lugar, leemos el siguiente nodo para situarnos dentro de *conexiones*.

En este caso habremos de seguir leyendo por cada uno de los elementos *conexion* que existen dentro de la etiqueta principal *conexiones*, y salir de la función cuando encontremos el final de esta etiqueta:

```
PUBLIC SUB RellenaArbol(Xml AS XmlReader)

  DO WHILE TRUE

    TRY Xml.Read()
    IF ERROR THEN RETURN
```

Si encontramos un nodo de tipo **Element** que se llame **conexion**, llamaremos a una función llamada **RellenaItem** para tratarlo. Pero si su nombre es desconocido para nosotros, lo ignoraremos, saltando todo su contenido para llegar al siguiente nodo del mismo nivel, con el método Next.

```
    IF Xml.Node.Type = XmlReaderNodeType.Element THEN

      IF Xml.Node.Name = "conexion" THEN

        RellenaItem(Xml)

      ELSE

        TRY Xml.Next()
        IF ERROR THEN BREAK

      END IF

    ELSE

      IF Xml.Node.Name = "conexiones"
         IF Xml.Node.Type = XmlReaderNodeType.
         EndElement THEN BREAK
      END IF

    END IF
```

```
LOOP

END
```

RellenaItem es el procedimiento más complejo, en el cual leeremos el contenido de cada conexión exitente.

```
PUBLIC SUB RellenaItem(Xml AS XmlReader)

  DIM Limite AS Integer
  DIM sNodo AS String
  DIM sLocal AS String
  DIM sPrim AS String
```

En primer lugar vamos a recoger los datos de los atributos de la etiqueta *conexion*. Para ello, hemos de iterar con la propiedad **Attributes** del nodo. A lo largo del proceso de iteración iniciado con FOR EACH, el puntero interno del lector XML pasará por cada uno de los atributos del nodo, que a su vez son nodos, cuyo nombre y valor es el nombre y valor del atributo. Finalizado el proceso de iteración, el puntero vuelve al nodo sobre el que estábamos situados, y con los datos recabados, añadimos en nuestro control *TreeView*, un nodo al efecto para su representación gráfica.

```
FOR EACH Xml.Node.Attributes

  IF Xml.Node.Name = "id" THEN sNodo = Xml.Node.Value
  IF Xml.Node.Name = "local" THEN sLocal = Xml.Node.
  Value

NEXT

IF sNodo <> "" AND sLocal <> "" THEN

  IF sNodo = "0" THEN
```

```
      TRY Arbol.Add(sNodo, sNodo & " (local)")
    ELSE
      TRY Arbol.Add(sNodo, sNodo & " (nacional)")
    END IF

  END IF
```

Pasamos ahora al interior del nodo *conexion* para extraer información de sus nodos *hijo*, es decir, las DNS y el número de teléfono. Buscaremos nodos de tipo **Element** y en función de su nombre actuaremos de un modo u otro.

```
TRY Xml.Read()
  IF ERROR THEN RETURN

  DO WHILE TRUE

    IF Xml.Node.Type = XmlReaderNodeType.Element THEN

      SELECT CASE Xml.Node.Name
```

Para el caso del teléfono, pasaremos del nodo actual (la etiqueta **teléfono**), al siguiente nodo que contendrá el texto con el número de teléfono, (podríamos, no obstante, mejorar el algoritmo contemplando la posibilidad de encontrar algo distinto a un nodo de texto, cosa que no haremos aquí por no complicar más el código, que siempre puede resultar complejo al principio).

```
        CASE "telefono"
          TRY Xml.Read()
          TRY Arbol.Add(sNodo & "-tel", "tel: " &
          Xml.Node.Value, NULL, sNodo)
```

Si el nodo es de tipo "**dns**", tendremos que comprobar el valor del atributo que indica, si es DNS primario o no, y luego leer el texto que contiene la IP:

```
CASE "dns"
   sPrim = "0"
   FOR EACH Xml.Node.Attributes
     IF Xml.Node.Name = "primario" THEN sPrim =
     Xml.Node.Value
   NEXT
   TRY Xml.Read()
   IF sPrim = "0" THEN
     TRY Arbol.Add(sNodo & "-dns2", "-dns2" &
     Xml.Node.Value, NULL, sNodo)
   ELSE
     TRY Arbol.Add(sNodo & "-dns1", "-dns1" &
     Xml.Node.Value, NULL, sNodo)
   END IF

 END SELECT
```


Una vez leído el nodo, pasamos al siguiente y continuamos leyendo hasta encontrar uno de tipo **EndElement**, donde sabremos que hemos encontrado el final del nodo *conexion*.

```
TRY Xml.Next()
IF ERROR THEN BREAK

 LOOP

ELSE

  IF Xml.Node.Type = XmlReaderNodeType.EndElement
  THEN BREAK


END IF
```

```
    TRY Xml.Read()
    IF ERROR THEN BREAK
  LOOP

END
```

e es el código completo expuesto:

```
' Gambas class file


PUBLIC SUB RellenaItem(Xml AS XmlReader)

  DIM Limite AS Integer
  DIM sNodo AS String
  DIM sLocal AS String
  DIM sPrim AS String

  FOR EACH Xml.Node.Attributes

    IF Xml.Node.Name = "id" THEN sNodo = Xml.Node.Value
    IF Xml.Node.Name = "local" THEN sLocal = Xml.Node.
    Value

  NEXT

  IF sNodo <> "" AND sLocal <> "" THEN

    IF sNodo = "0" THEN
      TRY Arbol.Add(sNodo, sNodo & " (local)")
    ELSE
      TRY Arbol.Add(sNodo, sNodo & " (nacional)")
    END IF
```

```
END IF

TRY Xml.Read()
IF ERROR THEN RETURN

DO WHILE TRUE

  IF Xml.Node.Type = XmlReaderNodeType.Element THEN

    SELECT CASE Xml.Node.Name

      CASE "telefono"
        TRY Xml.Read()
        TRY Arbol.Add(sNodo & "-tel", "tel: " & Xml.
        Node.Value, NULL, sNodo)

      CASE "dns"
          sPrim = "0"
          FOR EACH Xml.Node.Attributes
            IF Xml.Node.Name = "primario" THEN sPrim
            = Xml.Node.Value
          NEXT
          TRY Xml.Read()
          IF sPrim = "0" THEN
            TRY Arbol.Add(sNodo & "-dns2", "dns2: " &
            Xml.Node.Value, NULL, sNodo)
          ELSE
            TRY Arbol.Add(sNodo & "-dns1", "dns1: " &
            Xml.Node.Value, NULL, sNodo)
          END IF


    END SELECT
```

```
        TRY Xml.Next()
        IF ERROR THEN BREAK

      ELSE

        IF Xml.Node.Type = XmlReaderNodeType.EndElement
        THEN BREAK

      END IF

      TRY Xml.Read()
      IF ERROR THEN BREAK

    LOOP
END

PUBLIC SUB RellenaArbol(Xml AS XmlReader)

    DO WHILE TRUE

      TRY Xml.Read()
      IF ERROR THEN RETURN

      IF Xml.Node.Type = XmlReaderNodeType.Element THEN

        IF Xml.Node.Name = "conexion" THEN

          RellenaItem(Xml)

        ELSE

          TRY Xml.Next()
          IF ERROR THEN BREAK
```

```
      END IF

    ELSE

      IF Xml.Node.Name = "conexiones" AND Xml.Node.Type
      = XmlReaderNodeType.EndElement THEN
        BREAK
      END IF

    END IF

  LOOP

END

PUBLIC SUB Form_Open()

  DIM Xml AS XmlReader

  Xml = NEW XmlReader

  TRY Xml.Open(User.Home & "/conexiones.xml")
  IF ERROR THEN
    Message.Error("Fallo al abrir el fichero indicado")
    RETURN
  END IF


  DO WHILE TRUE

    IF Xml.Node.Type = XmlReaderNodetype.Element THEN

      IF Xml.Node.Name = "conexiones" THEN
```

```
        RellenaArbol(Xml)

      ELSE

        Message.Error("El documento no contiene datos
        de conexiones")
        Xml.Close()

      END IF

    END IF

    TRY Xml.Read()
    IF ERROR THEN
      Message.Error("Formato XML no válido")
      RETURN
    END IF

    IF Xml.Eof THEN BREAK

  LOOP

  TRY Xml.Close()

END
```

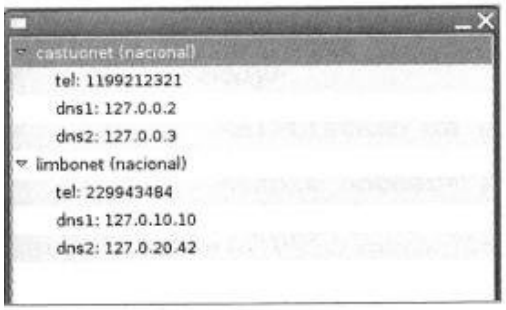

Figura 3. Resultado del fichero XML.

Este código dará lugar, al ejecutarse, a una visión en árbol de los datos contenidos en el fichero XML.

7. 3 XSLT

¿Qué es XSLT?

Acompañando a XML, XSLT permite realizar conversiones de formatos de documentos. Con XSLT se puede, por ejemplo, convertir datos de un documento XML en un documento HTML, o cosas más complejas como generar un documento PDF o StarWriter a partir de datos XML que nosotros hayamos diseñado.

Gracias a XSLT se puede separar de modo definitivo la información de su representación, por lo cual se emplea extensivamente en aplicaciones web, que pueden recibir datos de una base remota en formato XML y convertirlos, generalmente, a HTML para enviarlos al cliente con una representación agradable.

XSLT se basa en unos documentos, con formato XML, llamados plantillas, que contienen las instrucciones necesarias para convertir un determinado documento XML (con las etiquetas y atributos propios de dicho documento) en otro con distinto formato.

XSLT es extenso para tratarlo aquí en profundidad aunque, como siempre ocurre con los estándares abiertos y casi nunca con los formatos propietarios, podemos encontrar fuentes específicas de información adicional, por ejemplo en **http://www.w3schools.com.**

Una plantilla de ejemplo

Cada plantilla XSLT se refiere al contenido de un determinado documento XML. Supongamos un documento XML como éste, en el que se encuentra un listado de socios:

```
<?xml version="1.0"?>
<socios>
  <socio>
    <numero>1123</numero>
    <nombre>Juan L. Aguilar</nombre>
```

```
    <tipo>Honorario</tipo>
  </socio>
  <socio>
    <numero>2135</numero>
    <nombre>Salvador G. Tierra</nombre>
    <tipo>Regular</tipo>
  </socio>
  <socio>
    <numero>9654</numero>
    <nombre>Alberto N. Parra</nombre>
    <tipo>Regular</tipo>
  </socio>
</socios>
```

Las plantillas XSLT siempre comienzan con unos identificadores. El primero, de documento XML, ya lo conocemos, el segundo denota que lo que viene a continuación es un documento XSLT.

```
<?xml version="1.0">
<xsl:stylesheet version="1.0" xmlns:xsl="http://
www.w3.org/1999/XSL/Transform">
```

Tras esto, se escribe el código en sí, en el cual generaremos una tabla HTML con los datos.

```
<xsl:template match="/">
  <html>
  <body>
    <h1>Listado de socios</h1>
    <table border="1">
      <tr>
        <th><b>Nro.</b></th>
        <th><b>Nombre</b></th>
```

```
            <th><b>Tipo</b></th>
          </tr>
          <xsl:for-each select="socios/socio">
          <tr>
            <td><xsl:value-of select="numero"/></td>
            <td><xsl:value-of select="nombre"/></td>
            <td><xsl:value-of select="tipo"/></td>
          </tr>
          </xsl:for-each>
        </table>
      </body>
      </html>
    </xsl:template>
    </xsl:stylesheet>
```


Como podemos observar, vamos embebiendo las etiquetas HTML; empleamos el iterador **for each** para tomar cada elemento del fichero XML; y situamos en cada punto de la tabla uno de los campos elegidos.

Transformando el documento con Gambas

Hasta aquí, lo que tenemos es un documento XML y una plantilla XSLT, pero ahora necesitamos un motor que realice la conversión. Para ello guardaremos el fichero con datos en nuestra carpeta personal como **socios.xml**, y la plantilla como **socios.xsl**. Creamos un nuevo proyecto de consola llamado **TransformaXSLT**, con un único módulo *modMain* y una referencia al componente *gb.xml.xslt*.

El código será tan simple como éste:

```
' Gambas module file

PUBLIC SUB Main()

  DIM Documento AS NEW XmlDocument
```

```
    DIM Plantilla AS NEW XmlDocument
    DIM Resultado AS XmlDocument

    Documento.Open(User.Home & "/socios.xml")
    Plantilla.Open(User.Home & "/socios.xsl")

    Resultado = XSLT.Transform(Documento, Plantilla)

    Resultado.Write(User.Home & "/socios.html")

END
```

›mo indicamos anteriormente, la clase **XmlDocument** carga y verifica un documento ML en memoria. En este caso cargamos dos documentos: el primero, llamado **›cumento**, contiene los datos de los socios; el segundo, **Plantilla**, es la hoja XSLT que dica cómo transformarla en HTML. La única clase que aporta el componente *gb.xml.xslt*, llamada **XSLT**, es estática y dispone de un único método **Transform**, al cual pasamos como parámetros el documento y la plantilla, y devuelve un documento nuevo con el formato indicado, en este caso una página web. Escribimos dicha página en un fichero en nuestra carpeta personal, y salimos. Si consultamos con el navegador la página obtenida, veremos un resultado como el que se muestra en la figura de la izquierda.

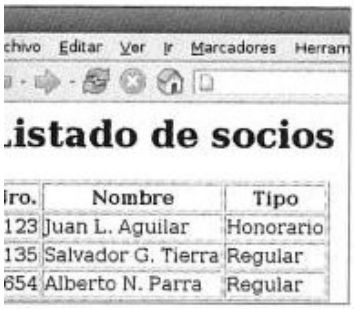

Figura 4. Resultado de la página obtenida.

7. 4 Acerca de XML-RPC

futura versión Gambas 2, dispondrá también de un componente XML-RPC. Estas las se refieren al uso de XML como sistema para comunicar dos procesos en dos iquinas diferentes. XML-RPC es un subconjunto de XML que define un lenguapara llamar a procesos remotos.

Un servidor XML-RPC acepta llamadas remotas a sus métodos. El proceso es muy similar a llamar a una función local dentro de un programa: el cliente llama a la función pasando unos parámetros (números enteros, cadenas, estructuras de datos...); y el servidor procesa la llamada y devuelve al cliente un resultado, que también será una cadena, número, fecha, etc.

El componente XML-RPC aportará varias facilidades para implementar estos procesos.

En el lado servidor dispondrá de un *parser*, en el cual definiremos los nombres de los métodos que se aceptarán, así como la correspondencia entre estos y otros locales del programa servidor. De este modo, tan sólo hay que pasar la petición del cliente al *parser*, el cual se encargará de verificar el número y tipo de los parámetros, y devolver un error al cliente, o llamar a la función local y devolver el resultado al cliente.

El servidor podrá funcionar en solitario, implementando un pequeño servicio web para atender las peticiones, o bien funcionar de modo controlado por la aplicación servidora, con el fin de implementar CGIs que se sirvan desde Apache, por ejemplo.

En la parte cliente es posible definir la URL del servidor y la forma de cada método. Igualmente podrá actuar en solitario, solicitando mediante una petición web la llamada al servidor. También se puede crear el documento XML de la petición, dejando a la aplicación el diseño del transporte y recepción de los datos con el servidor.