ON THE USE OF XML IN THE BLAISE ENVIRONMENT

JELKE BETHLEHEM & LON HOFMAN, STATISTICS NETHERLANDS

1. Introduction

XML is the Extensible Markup Language. It was developed in 1998 to cope with a number of restrictions of HTML, the language to design Internet sites. Initially, XML may have seemed like it was all just typical software industry hype. It would be the future of all data manipulation and data transmission, and therefore be the answer to the ultimate question of life, the universe and everything.

After a few years now, it has become clear that XML is no hype. It turned out that XML is much more than a potential successor of HTML. It is used as a tool for platform independent data exchange. Moreover, XML has turned out to be very useful for describing data in a very structured way. Therefore, it is playing an increasingly important role in the world of data and meta data. Various developers in all parts of the world are working on statistical meta data systems based on XML. These developments hold as promise of more standardisation of, or at least easier interaction between these meta data systems.

This paper explores a potential role for XML in the Blaise System. It concentrates on the meta data aspects by showing how the current Cameleon tool could be replaced by a new one based on XML. Section 2 gives a general introduction into XML. It also shows how information in an XML file can be manipulated using tools like XSL (Extended Style sheet Language). Section 3 describes how XML could play a role in the Blaise environment. It describes how Cameleon can generate XML files containing question meta data. Once such an XML file is available, there are several approaches of manipulating the meta data. Examples show how XSL can be used to generate an SPSS setup and questionnaire documentation in HTML format. It is also made clear that new meta data tools can be developed in fairly simple way. This is illustrated by means of a simple example of question documentation browser.

This paper is **not** the blueprint for the future of Blaise. It is only meant to trigger some discussion about possible future developments. And it shows that XML may be an important player in this field.

2. About XML

2.1. What is XML?

XML is a markup language. A markup language defines markup rules. Markup refers to anything included in a document that adds a special meaning to it or provides extra information about it. A markup language defines a set of rules that declare what markup constitutes, and exactly what the markup means. Three types of markup can be distinguished:

- *Stylistic markup* indicates how the document is to be presented. For example, it can instruct to present text in boldface or Italics, to use a specific font, etc.
- *Structural markup* gives information about the structure of a document. For example, it can instruct a piece of text to be handled as a section heading, or a paragraph.
- *Semantic markup* gives information about the content of the data. An example is a markup that declares a piece of text to be programmer's comment that is not to be printed or displayed.

XML is just one example of a markup language. Other examples are SGML and HTML. SGML (Standardized Generalized Markup Language) was developed in the late 1960's. Its purpose was to describe markup languages, by allowing the author to provide formal definitions for each of the elements and attributes of his markup language. Therefore, SGML is a meta-language. It is a language for describing languages. At the time, SGML was one of the competing meta-languages. However, it was its offspring HTML, which caused it to become more popular than the others.

SGML is a very powerful meta-language. The price paid for this power is complexity. The language has many features that are rarely used. It is difficult to interpret a SGML document without the definition of the markup language used. This definition is kept in a *Document Type Definition* (DTD). The DTD contains all language rules. The DTD has to be sent with, or included in, the SGML document so that custom created elements can be understood. Markup languages created by SGML are called *SGML applications*.

HTML is the language used for making web sites. It was originally an SGML application. It describes how information is to be prepared for the World Wide Web. HTML is just a set of SGML rules. In the case of HTML, the DTD is stored somewhere in the browser (e.g. Netscape Navigator or Internet Explorer). As a language, HTML is only a fraction of the size of SGML, and it is easy to learn. This quickly made it very popular.

HTML uses *tags* to markup a document. Examples of these tags are:

```
<B>This text will be displayed in boldface</B>
<H1>This text is a primary heading</H1>
<P>Treat the text between these tags as a paragraph</P>
```

Although HTML is now used at a very wide scale, the language has its limitations. Two of the most important ones are:

- HTML has a fixed set of tags. It is not possible to create new tags that can be interpreted properly by others.

- HTML focuses on presentation. The tags are used to describe how information is to be displayed by the Internet browser software. The tags do not carry any information about the meaning the text they enclose.

SGML does not have the drawbacks of HTML. However, SGML is very complex. The major players in the web browser market have made it clear that they have no intention of fully supporting SGML in their browsers. This caused moves to be made to create a simplified version of SGML capable of marking up documents according to their content. This development was supported by the World Wide Web Consortium (W3C). The result was XML (Extensible Markup Language). Like SGML, XML is also a meta-language. It is not a fixed format language like HTML. Users can create their own tags, and these tags can describe content.

HTML is mainly used to describe how information is to be displayed. XML is different. It is used to describe the structure and meaning of a document. Formatting of XML documents is described in a separate document, called a *style sheet*. Because XML describes structure and meaning, an XML document can be seen as a data file, on which processing instructions can be carried out. The data can be used and re-used across different computer platforms and in different applications. Therefore XML is gaining popularity as a data storage format.

Instead of using style sheets, there are also different ways of formatting information contained in an XML document. Microsoft offers (as part of Internet Explorer version 5) a DLL file containing an interface for parsing XML documents. This DLL can be used in a Delphi, C++, or Visual Basic environment to develop dedicated applications.

Using style sheets to format the information has a number of advantages. First, different style sheets can be used on the same XML document to present the same information in different ways. Second, style sheets offer the possibility to format XML documents from different sources in the same way. Therefore, it promotes standardisation. And third, style sheets allow for easy transformation of one XML document into another.

XML has also raised interest in the statistical community, in particular in the area of meta-data. An example is the ADDSIA project, financed by the Fourth Framework of the European Union, see Bi and Murtagh (1998). XML is used in this project to described the meta-data for the statistics on economic indicators. Another example is the DDI, the Data Documentation Initiative. This is an initiative of the international social science community of researchers and archivists to develop a standard for a code book describing variables in social survey data files, see the web site *http://www.icpsr.umich.edu/DDI/codebook.html*. Another application of XML as the TADEQ project, see Bethlehem (1999) or visit the project web site *http://neon.vb.cbs.nl/rsm/tadeq*. TADEQ is a tool to document electronic questionnaires, like generated by the Blaise System. It uses XML to store the execution tree of an interview.

2.2. Tags and attributes

To develop an XML application, three or more components are required: The Document Type definition (DTD), the actual XML file containing the information to be processed, and one or more style sheets (or a dedicated software tool to parse an XML document). These components are described in some more detail using an example of a Blaise questionnaire documentation. The documentation describes the questions in a Blaise data model. The data model has a name and a descriptive text. Each question has a name and a text. Furthermore, the position of the question in the data file is documented. For each different question type (open, closed, numeric), the relevant information is recorded. The XML file for this example could look like displayed in figure 2.2.1.

Figure 2.2.1. An example of an XML file

```
<?xml version="1.0"?>
<!DOCTYPE DataModel SYSTEM "blaise.dtd">
<!-- ?xml-stylesheet type="text/xsl" href="blaise1.xsl" ?-->
<DataModel>
  <ModelName>Commut</ModelName>
  <ModelText>The Commuting Survey</ModelText>
  <Question Qname="Name">
    <Qtext>What is your name?</Qtext>
    <FilePos First="1" Last="20"/>
    <Open Length="20"/>
  </Question>
  <Question Qname="Sex">
    <Qtext>What is your sex?</Qtext>
    <FilePos First="21" Last="21"/>
    <Closed Max="1">
      <Item Code="1" Name="Male" Label="Male"/>
      <Item Code="2" Name="Female" Label="Female"/>
    </Closed>
  </Question>
  <Question Qname="Age">
    <Qtext>What is your age (in years)?</Qtext>
    <FilePos First="22" Last="24"/>
    <Numeric Min="0" Max="120" Dec="0"/>
  </Question>
  <Question Qname="MarStat">
    <Qtext>What is your marital status?</Qtext>
    <FilePos First="25" Last="25"/>
    <Closed Max="1">
      <Item Code="1" Name="NeverMar" Label="Never married"/>
      <Item Code="2" Name="Married" Label="Married"/>
      <Item Code="3" Name="Divorced" Label="Divorced"/>
      <Item Code="4" Name="Widowed" Label="Widowed"/>
    </Closed>
  </Question>
</DataModel>
```

Information is enclosed in *tags*. Tags always come in pairs in XML. There is an opening tag and a closing tag. The name of the closing tag is equal to the opening tag, but with a slash added to it. See for example the tags <ModelName> and </ModelName> in the example above. There is an exception to this rule. If a tag pair never encloses any text, one tag may be used, and the tag name should end with a slash. Examples are <Open> and <Numeric> in the example above.
Tags can have so-called *attributes*. These are parameter values included in the opening tag. In the example above, the <Question> tag has one attribute Qname, and <FilePos> tag has two attributes First and Last.
Attributes are a different way of attaching specific values to an element. There is no simple rule about whether information should be specified as the value of an attribute or as text between an opening and closing tag. It is up to the developer of the XML application whether to use tags or attributes for defining structure. For short parameter values, attributes might be preferred. It improves readability of the XML document. Attributes also allow for defining default values of parameters. And if the data consists of nested elements, there is no other way than to used nested tags with information.
The tags <DataModel> and </Datamodel> mark the begin and end of the data model definition. The data model description can be found between the tags <ModelText> and </Modeltext>. This example contains the documentation of four questions. A question definition starts with <Question> and ends with </Question>. The first question is a closed question. The name of the question (LastWeek) can be found as the value of the attribute Qname of the <Question> tag. The <FilePos> tag contains the position of the values of this question in the data file. For closed questions, there is a <Closed> tag, and it contains a arbitrary number of <Item> tags. Each <Item> tag describes a possible answer.
The second question in the example is an open question. Its definition contains an <open> tag with the attribute Length to indicate the maximum length of the answer.
The fourth question is a numeric question. The <numeric> tag contains attributes for the lower and upper bound, and for the number of decimals.

2.3. Document Type Definitions

XML can be used to define two types of documents. They are called well-formed documents and valid documents. A document is called *well- formed* if it satisfies three conditions:

- The document contains at least one element. An *element* is a pair of tags (open and closing tag) enclosing some information;

- The document must contain a *root element*. This is a unique open and closing tag that surrounds the whole document;
- All other elements in the document must be *nested*. There may be no overlap between elements.

The XML-example in the previous subsection satisfies these three conditions.
An XML document is called *valid* if it not only is a well-formed document, but it also has a Document Type Definition (DTD) the document conforms to. This means the document can only contain elements defined by the DTD, and also these elements can only be used in the order defined by the DTD.
It is possible to use XML documents without a DTD, but having a DTD has some advantages. One is that it allows for a document to be parsed and checked, so that errors in the XML file can be detected.
A DTD is a pre-defined structure against which instances of documents can be checked. The DTD was introduced in version 1.0 of XML. Later versions of XML also suggest other ways of defining structure. On is called XML Schema. Here we focus on the use of the DTD. Figure 2.3.1 contains an example of how the DTD for the Blaise question documentation application could look like. Note that this DTD is far from complete because not all the basic field types are covered.

Figure 2.3.1. The Data Type Definition for Blaise

```
<!-- Blaise Data Type Definition - version 1.0 -->

<!ELEMENT DataModel (ModelName, ModelText?, Question+) >
<!ELEMENT ModelName (#PCDATA) >
<!ELEMENT ModelText (#PCDATA) >

<!ELEMENT Question (Qtext, FilePos, (Open | Closed | Numeric)) >
<!ATTLIST Question Qname CDATA #REQUIRED >

<!ELEMENT Qtext (#PCDATA) >

<!ELEMENT FilePos EMPTY >
<!ATTLIST FilePos First CDATA #REQUIRED >
<!ATTLIST FilePos Last CDATA #REQUIRED >

<!ELEMENT Open EMPTY >
<!ATTLIST Open Length CDATA #REQUIRED >

<!ELEMENT Closed (Item+) >
<!ATTLIST Closed Max CDATA #REQUIRED >

<!ELEMENT Item EMPTY >
<!ATTLIST Item Code CDATA #REQUIRED >
<!ATTLIST Item Name CDATA #REQUIRED >
<!ATTLIST Item Label CDATA #IMPLIED >

<!ELEMENT Numeric EMPTY >
<!ATTLIST Numeric Min CDATA #REQUIRED >
<!ATTLIST Numeric Max CDATA #REQUIRED >
<!ATTLIST Numeric Dec CDATA #REQUIRED >
```

The DTD defines for each element what can be put between the corresponding brackets. For example, between the <DataModel> tags one must put one <ModelName> element, followed by an optional <ModelText> element, and one or more <Question> elements. A "+" means that an element must be included one or more times. The "|" between open and closed and between closed and numeric means that a choice must be made between these three elements. The reserved word #PCDATA indicates parseble character data, i.e. plain text. So between <ModelText> tags only character data can appear, and no other elements.
The attribute parameters are described by <!ATTLIST>. The two attribute parameters code and name of <Item> must always be specified, and the parameter Label is optional.

2.4. Style sheets

The Document Type Definition defines the structure of the data, but not how they are to be displayed. This is handled by so-called *style sheets*. A style sheet is a set of rules that declare how a document should be displayed. Style sheets have a number of advantages:

- Since structure declarations are separated from style declarations, the readability of the documents is improved;
- Using different style sheets, it is possible to the display the same document in different ways;
- Using the same style sheet for different documents, it is possible to change the appearance of all these documents by simply changing one style sheet.

Currently, there are several style sheet approaches with respect to XML, the two most important being CSS (*Cascading Style Sheets*) and XSL (*Extensible Style sheet Language*). The specifications of these two languages are by no means fixed yet. They are still under development. CSS has limited possibilities. It is only possible to affect the layout of the document text (font, bold, italic, underline, etc). XSL has more possibilities for influencing layout and adding other text elements.
Figure 2.4.1 contains an example of an XSL style sheet that works with the Blaise question documentation example.
Note that the XSL style sheet is an XML application in itself. The XSL instructions in the example generate an HTML page. All tags starting with xsl: denote special XSL instructions. For example, <xsl:for-each select="//Question" > searches the XML file for each occurence of the <Question> tag. And the instruction <xsl:value-of select="Qtext"/> displays the text between the <Qtext> tags.

Figure 2.4.1. An XSL style sheet

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/">
<HTML>
<BODY>
<H1><xsl:value-of select="DataModel/ModelName"/></H1>
<H2><xsl:value-of select="DataModel/ModelText"/></H2>

<xsl:for-each select="//Question" >
<TABLE BORDER="1" WIDTH="80%">
<TR><TD><B><xsl:value-of select="@Qname" /></B></TD></TR>
<TR><TD><xsl:value-of select="Qtext" /></TD></TR>
<xsl:choose>
<xsl:when match="Question[Closed]">
<TR><TD>Closed question</TD></TR>
</xsl:when>
<xsl:when match="Question[Open]">
<TR><TD>Open question</TD></TR>
</xsl:when>
<xsl:when match="Question[Numeric]">
<TR><TD>Numeric question</TD></TR>
</xsl:when>
</xsl:choose>
</TABLE>
<P/>
</xsl:for-each>
</BODY>
</HTML>
</xsl:template>
</xsl:stylesheet>
```

The file contains a combination of XML tags and HTML tags, and this makes it a proper XML document. However, note that HTML is somewhat sloppy in the use of tags, and XML is stricter. XML requires a closing tag to be present for each opening tag. Therefore, for each <TD> tag there must be a </TD> tag, which is not required in HTML. Also, empty tags, like <P> must have terminating slash. This is the reason why figure 2.4.1 contains <P/> instead of <P>.
When the XML file is loaded in the browser (Internet Explorer 5) using this style sheet, an HTML page like in figure 2.4.2 is displayed.
Note that this is only a very simple HTML page. Much more useful and effective HTML output could be generated by making use of all the interesting functions that HTML has. One example could be the inclusion of hyperlinks to offer the possibility to jump to other parts of the documentation, or the other documentation.
There are many books on the market, on more and more are published each day. For this paper, Morrison et al. (2000) and Boumphrey et al. (1998). The XML support implemented in Internet Explorer 5 is well documented in Homer (1999).

Figure 2.4.2. An HTML page generated by an XSL style sheet

Commut

The Commuting Survey

Name		FirstPos: 1	LastPos: 20
What is your name?			
Open question			Len: 20

Sex		FirstPos: 21	LastPos: 21
What is your sex?			
Closed question			
1	Male	Male	
2	Female	Female	

Age		FirstPos: 22	LastPos: 24
What is your age (in years)?			
Numeric question	Min: 0	Max: 120	Dec: 0

MarStat		FirstPos: 25	LastPos: 25
What is your marital status?			
Closed question			
1	NeverMar	Never married	
2	Married	Married	
3	Divorced	Divorced	
4	Widowed	Widowed	

3. Use of XML in Blaise

3.1. Languages for meta-data

Blaise was originally designed as a system for capturing interview data. In later versions it developed into an integrated survey processing system. Vital for Blaise is the Blaise language to document meta data in the data model. Blaise also uses a few other languages to manipulate data and meta data. One example is the language used to define Manipula setups. It resembles the Blaise language, but is not completely identical to it. Another example is the language used in the tool Cameleon. Cameleon takes Blaise meta data as input, and produces a description of this meta data in another format. A typical example is a setup file for the statistical package SPSS.

Cameleon meta data transformations are based on instruction files, or so-called Cameleon Scripts. Cameleon uses a dedicated language for defining scripts. An example of such a script can be found in appendix A. This is a script to transform Blaise meta data into an XML file.

Taking into account the growing popularity of XML, one can think of two approaches to consider XML as a means to manipulate Blaise meta data. One is to completely replace the Cameleon Script Language. This may be an option for the long run. Another approach is to offer a Cameleon Script that translates Blaise meta data into an XML file. Then the user has at his disposal all new possibilities for using XML without losing the current functionality of Cameleon. This paper explores the second approach.

3.2. A Cameleon Script for XML

Appendix A contains a Cameleon Script that translates a Blaise meta data file into an XML file. This script is somewhat simplified because it does not cover everything one may encounter in a Blaise data model. Still, it serves its purpose by showing how not too complex Blaise data models can be translated in a fairly straightforward way.

We illustrate our approach by using a very simple Blaise data model. It only has four questions of three different types. The data model is presented in figure 3.2.1.

When the meta data file of this data model is processed by the Cameleon XML Script an XML file is generated. The content of this file is displayed in figure 3.2.2. Note that this figure contains the same information as figure 2.2.1.

Once an XML file is available, it can be used to transform Blaise meta data into a format which can be read by other software. In the following subsections we describe various approaches. Only two transformations are considered: From Blaise into HTML question documentation, and from Blaise into an SPSS setup. Subsection 3.3 shows how to use XSL style sheets from within an HTML environment, and subsection 3.4 describes how the same thing can be accomplished using a simple Visual Basic tool. Subsection 3.5 shows how the XML parser available in Internet Explorer 5 can be included in a dedicated software tool for processing XML information without using style sheets.

Figure 3.2.1. The Blaise data model

```
DATAMODEL Commut "The Commuting Survey"

FIELDS
  Name    "What is your name?": STRING[20]
  Sex     "What is your sex?": (Male, Female)
  Age     "What is your age (in years)?": 0..120
  MarStat "What is your marital status?":
          (NeverMar  "Never married",
           Married   "Married",
           Divorced  "Divorced",
           Widowed   "Widowed")
 RULES
   Name Sex Age MarStat

ENDMODEL
```

Figure 3.2.2. The XML file

```
<?xml version="1.0"?>
<!DOCTYPE DataModel SYSTEM "blaise.dtd">
<!-- ?xml-stylesheet type="text/xsl" href="blaise1.xsl" ?-->
<DataModel>
  <ModelName>Commut</ModelName>
  <ModelText>The Commuting Survey</ModelText>
  <Question Qname="Name">
    <Qtext>What is your name?</Qtext>
    <FilePos First="1" Last="20"/>
    <Open Length="20"/>
  </Question>
  <Question Qname="Sex">
    <Qtext>What is your sex?</Qtext>
    <FilePos First="21" Last="21"/>
    <Closed Max="1">
      <Item Code="1" Name="Male" Label="Male"/>
      <Item Code="2" Name="Female" Label="Female"/>
    </Closed>
  </Question>
  <Question Qname="Age">
    <Qtext>What is your age (in years)?</Qtext>
    <FilePos First="22" Last="24"/>
    <Numeric Min="0" Max="120" Dec="0"/>
  </Question>
  <Question Qname="MarStat">
    <Qtext>What is your marital status?</Qtext>
    <FilePos First="25" Last="25"/>
    <Closed Max="1">
      <Item Code="1" Name="NeverMar" Label="Never married"/>
      <Item Code="2" Name="Married" Label="Married"/>
      <Item Code="3" Name="Divorced" Label="Divorced"/>
      <Item Code="4" Name="Widowed" Label="Widowed"/>
    </Closed>
  </Question>
</DataModel>
```

3.3. Using style sheets in an HTML environment

An XML application has a tree structure. The tags represent the nodes of the tree. Tags that are nested in other tags represent branches of the tree. In principle, an XSL style sheet does nothing more than transforming this tree structure into another tree structure. The XSL instructions describe which nodes are transformed and how they are transformed.

An example of how XSL can transform XML into HTML was already given in section 2, see figure 2.4.1. The resulting HTML file for the Blaise example can be found in figure 2.4.2.

The transformation into an SPSS setup is somewhat more complex. The XSL file for this is given in figure 3.3.1. To create the setup file, three runs have to be made through the XML file: one to create the DATALIST section, one to create the VAR LABELS section, and one to create the VALUE LABELS section. The <xsl:for-each> instruction implements these loops. It locates each occurrence of the <Question> tag. In the first loop, the values of the attribute Qname of the <Question> tag and the values of the attributes First and Last of the <FilePos> tag are retrieved. In the second loop, the values of the attribute Qname of the <Question> tag and of the <Qtext> tag are retrieved.

The third loop is somewhat more complex. Only information on closed questions is retrieved. This is accomplished by searching for the pattern '//Question[Closed]' . Note that there is another loop nested within the loop. With the instruction <xsl:for-each select="Closed/Item" > all <Item> tags are located within each <Closed> tag, and from the <Item> tags the values of the Code and Label attributes are retrieved.

Note that the last item in the item loop gets a different treatment. This is necessary because for the last value in the VALUE LABELS labels list different output must be generated. This is accomplished by using the tags <xsl:when match="Item[end()]"> and <xsl:otherwise>.

Because blanks in XSL files are ignored, a simple JavaScript function is used to include blanks in the output. The <xsl:script> instruction is used to define this function, and the <xsl:eval> instruction makes it possible to execute it.

Figure 3.3.1. A style sheet for an SPSS setup

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:script language="javascript">
function B(N)
{ var T="                    "
  return T.substring(0, N)
}
</xsl:script>
<xsl:template match="/">

TITLE '<xsl:value-of select="DataModel/ModelText"/>'.
DATALIST FILE='<xsl:value-of select="DataModel/ModelName"/>.asc' /
<xsl:for-each select="//Question" >
<xsl:eval language="javascript">B(3)</xsl:eval>
<xsl:value-of select="@Qname" />
<xsl:eval language="javascript">B(1)</xsl:eval>
<xsl:value-of select="FilePos/@First" />-
<xsl:value-of select="FilePos/@Last" />
</xsl:for-each>
<xsl:eval language="javascript">B(3)</xsl:eval>.

VAR LABELS
<xsl:for-each select="//Question" >
<xsl:eval language="javascript">B(3)</xsl:eval>
<xsl:value-of select="@Qname" />
<xsl:eval language="javascript">B(1)</xsl:eval>'
<xsl:value-of select="Qtext" />'
</xsl:for-each>
<xsl:eval language="javascript">B(3)</xsl:eval>.

VALUE LABELS
<xsl:for-each select="//Question[Closed]" >
<xsl:eval language="javascript">B(3)</xsl:eval>
<xsl:value-of select="@Qname" />
<xsl:for-each select="Closed/Item" >
<xsl:choose>
<xsl:when match="Item[end()]">
<xsl:eval language="javascript">B(6)</xsl:eval>
<xsl:value-of select="@Code" />
<xsl:eval language="javascript">B(1)</xsl:eval>'
<xsl:value-of select="@Label" />'/
</xsl:when>
<xsl:otherwise>
```

```
<xsl:eval language="javascript">B(6)</xsl:eval>
<xsl:value-of select="@Code" />
<xsl:eval language="javascript">B(1)</xsl:eval>'
<xsl:value-of select="@Label" />'
</xsl:otherwise>
</xsl:choose>
</xsl:for-each>
</xsl:for-each>
<xsl:eval language="javascript">B(3)</xsl:eval>.

SAVE /OUTFILE '<xsl:value-of select="DataModel/ModelName"/>.sav'.
</xsl:template>
</xsl:stylesheet>
```

The result of applying the XSL style sheet to the XML file of the Blaise example can be found in figure 3.3.2.

Figure 3.3.2. SPSS setup generated by the XSL style sheet

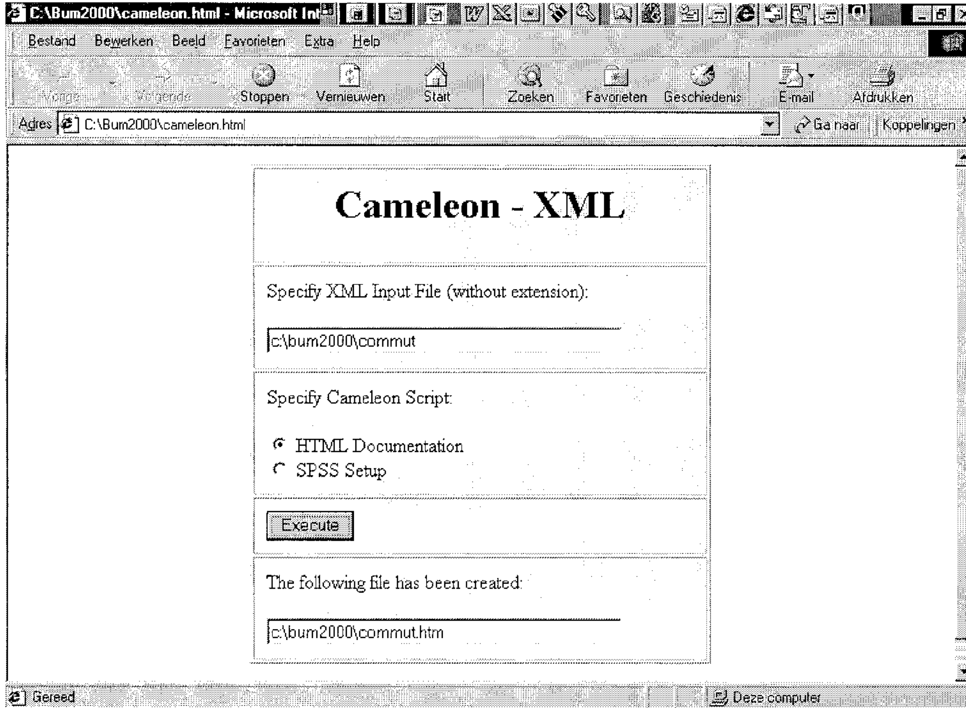
```
TITLE 'The Commuting Survey'.

DATALIST FILE='Commut.asc' /
  Name 1-20
  Sex 21-21
  Age 22-24
  MarStat 25-25
  .

VAR LABELS
  Name 'What is your name?'
  Sex 'What is your sex?'
  Age 'What is your age (in years)?'
  MarStat 'What is your marital status?'
  .

VALUE LABELS
  Sex
    1 'Male'
    2 'Female'/
  MarStat
    1 'Never married'
    2 'Married'
    3 'Divorced'
    4 'Widowed'/
  .

SAVE /OUTFILE 'Commut.sav'.
```

It is rather simple to implement this kind of conversion tool in an HTML environment. Appendix E contains an example of how this can be done. The main part of the work is done by a JavaScript function Transform(). The following fragment contains the core of this function.

```
InFile = document.Form1.InputFile.value + ".xml"
XmlFile = new ActiveXObject('microsoft.XMLDOM')
XmlFile.load(InFile)
XslFile = new ActiveXObject('microsoft.XMLDOM')
XslFile.load(ScrFile)
Output = xmlFile.transformNode(xslFile)
Fso = new ActiveXObject("Scripting.FileSystemObject")
OutFile = fso.CreateTextFile(OutFile, true)
OutFile.Write(output)
OutFile.Close()
```

The first three lines read the XML file. The first line assigns the name of the file as specified by the user on the HTML page to the variable inFile. The second line creates an new XML object. Note that this only works in a browser supporting XML, like Internet Explorer 5. And in the third line the XML file is loaded into the XML object.
The second two lines create another XML object. This time it is used to load the XSL style sheet file as specified in the variable ScrFile. Line six does the transformation. It creates a new XML object Output by transforming the original object and using the XSL instructions. The final lines of this JavaScript fragment see to it that the transformed object is written to file.
When the HTML file of appendix E is loaded into the browser Internet Explorer 5, it will look like in figure 3.3.

Figure 3.3.3. The HTML version of Cameleon



This simple prototype only offers two transformation possibilities: HTML question documentation and an SPSS setup. Of course, it can easily be expanded to include more types of transformations.

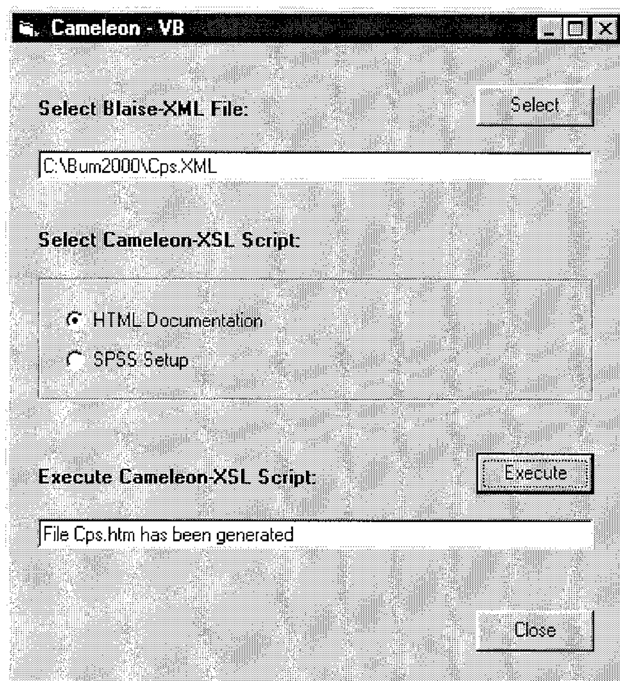
3.4. Using style sheets in a Visual Basic Environment

Implementing a tool like Cameleon in an HTML environment has its limitations. If this is a concern, one can consider implementing it in a real programming language environment. This is illustrated by means of a simple Visual Basic program. The complete code of this program can be found in appendix F. The XML support offered by Internet Explorer 5 is available in a DLL file (MSXML.DLL). This file can be referenced to from within Visual Basic (or other programming languages, like Delphi or C++).
The statement that does most of the work of the VB version of Cameleon is:

```
outDoc = xmlDoc.transformNode(xslDoc)
```

It transforms the XML object in the variable xmlDoc into a new XML object outDoc using the XSL instructions in the XML object xslDoc. The rest of the program code is concerned with the user interface. When this program is run, it looks like in figure 3.4.1.

Figure 3.4.1. The Visual Basic version of Cameleon



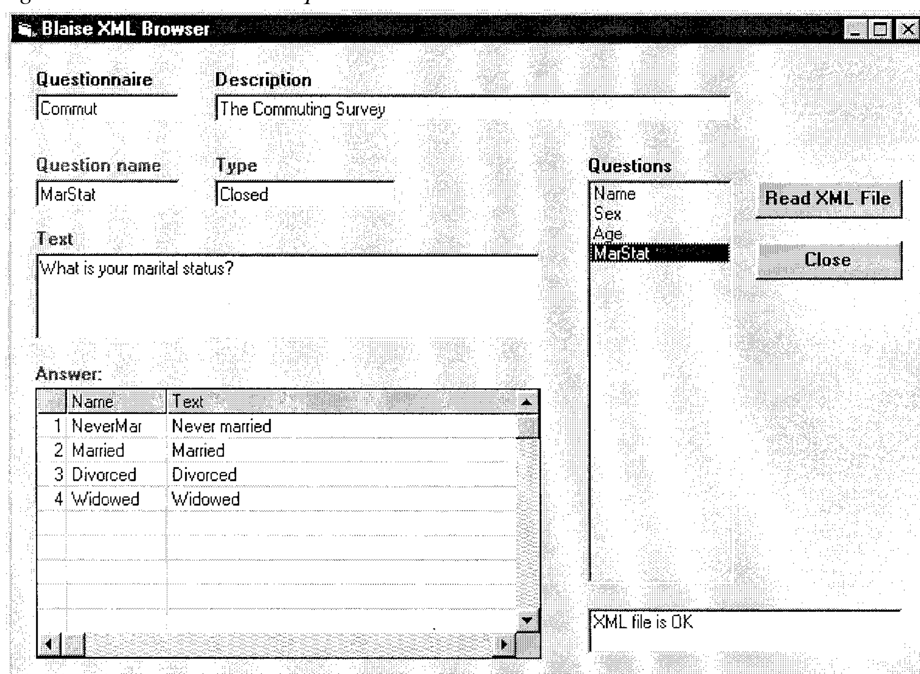
Also here it should be noted this is just a very simple example of a Visual Basic application. It can be extended at wish.

3.5. Parsing the XML tree with a dedicated program

The previous two sections used XSL style sheets to transform the meta data in the Blaise XML into a different format. This approach has the advantage that only XSL knowledge is required for a user to modify existing Cameleon Scripts or to create new ones. The XSL instruction language is reasonably powerful but has its limitations. For example, the result of an XSL transformation is always an XML tree. This prohibits, for example, the implementation of more interactive meta data applications.

There is a different approach possible that does not make use of XSL style sheets. As was mentioned in the previous subsection, it is easy in a programming environment to get access the XML parser that is part of Internet Explorer 5. This parser contains a lot of functions that allow a programmer to process the XML tree in a self-defined way. This approach is illustrated by describing a Visual Basic application that offers an interactive question documentation browser. The complete could of this application can be found in appendix G. When executed this Visual Basic program looks like in figure 3.5.1.

Figure 3.5.1. An interactive question browser



The XML parser sees each tag as an element of the XML tree. Furthermore, the text between an opening and closing tag is considered to be a child element of the tag element.
The XML parser makes available the routine getElementsByTagName to create a list of references to tags with a specified name. The elements in the list are denoted by Item(0), Item(1), etc. So, with the Visual Basic statement

```
Set e = xmlDoc.getElementByTagName("ModelName").Item(0)
```

a reference is obtained to the first occurence of the tag <ModelName>. And the subsequent statement

```
e.firstChild.nodeValue
```

retrieves the value of the first child of the tag <ModelName>, and this is the text between <ModelName> and </ModelName>.
As another example, the list of question names on the right-hand side of the screen in figure 3.5.1 is created with the statements

```
Qlist.Clear
Set eList = xmldoc.getElementsByTagName("Question")
n = eList.length
I = 0
Do While I < n
  Set e = eList.Item(I)
  Set ec = e.attributes.getNamedItem("Qname")
  QList.AddItem ec.firstChild.nodeValue
  I = I + 1
Loop
```

The object eList contains a list of references to all <Question> tags. This list is processes in a while-loop. The variable e points to the current <Question> tag, and ec to the Qname attribute. With ec.firstChild.nodeValue the attribute value (the question name is obtained).
The subroutine DispQuestion displays the characteristics of the current question. Note the somewhat complex way of obtaining the question type. With the statement

```
Set e = eList.Item(I)
```

e points to the current question tag. The first child of e is the <Qtext> tag, the second child is the <FilePos> tag, and the third child refers to the tag indicating the question type (either <Open>, <Closed> or <Numeric>. So, the statement

```
Set e = e.firstChild.nextSibling.nextSibling
```

makes e point to the tag indicating the question type. With e.nodeName the tag text (either Open, Closed or Numeric) is obtained.
Depending on the type of question, certain characteristics are displayed. Note that for a closed question, first a list of references to <Item> tags is built.
The XML parser offered by Internet Explorer has much more possibilities of processing XML file. For more details, see Homer (1999).

4. Conclusion

Support for XML is rapidly growing and more and more XML applications are developed, both inside and outside statistics. Even within the world of Official Statistics we see interesting new XML developments. A good example is the Data Documentation Initiative, which promotes documentation of survey data sets by means of XML. Traditionally, survey data set documentation is a cumbersome, boring and time-consuming activity. If computer assisted interviewing systems are able to automatically generate documentation in XML format, and data archives need documentation in XML format, then future for survey data documentation suddenly looks a lot brighter.
Some experts say XML promotes standardisation. This is only partly true, because everyone can develop its own XML application based on its own language. However, it is certainly true that it is not difficult to transform one XML application into another. XSL is the instrument that does just that. So one could definitely say that XML is the glue that ties everything together.
Is there a future for XML in Blaise? The examples in this paper show that it is not very difficult to add a Cameleon script to the Blaise system that generates XML based on a defined DTD. One step further would be to completely replace the Cameleon Script Language by XML combined with a set of XSL scripts or some other tool able of parsing the XML file.
A little bit more speculative is considering a future version of Blaise that stores its meta data itself in XML format. This may be not such a very wild idea, considering rumours that future versions of MS Word will store text in XML format, and that the next version of Mathtype, the equation editor of MS Word, will store mathematical expressions in XML.

References

Bethlehem, J.G. (1999): The Routing Structure of Questionnaires. Proceedings of the Third ASC International Conference, Association of Survey Computing, Chesham, United Kingdom, pp. 405-418.

Bi, Y. and Murtagh, F. (1998), The Roles of Statistical Metadata and XML in Structuring and Retrieving Statistical Information. Pre-proceedings of the International Seminar on New Techniques and Technologies for Statistics, Sorrento, Italy, pp. 73-78.

Boumphrey, F., Direnzo, O., Duckett, J, Graf, J., Hollander, D., Houle, P., Jenkins, T., Jones, P., Kingsley-Hughes, A., Kingsley-Hughes, K., McQueen, C., and Mohr, S. (1998): XML Applications. Wrox Press, Birmingham, UK.

Homer, A. (1999): XML IE5, Programmer's Reference. Wrox Press Ltd, Birmingham, UK.

Morrison, M., Boumphrey, F. and Brownell, D. (2000): XML Unleashed. Sams Publishing, Indianapolis, USA.

Appendix A. Cameleon Script for an XML file

```
[VAR T: STRING]
[OUTFILE:= '.\' + METAINFOFILENAME  + '.xml']
<?xml version="1.0"?>
<!DOCTYPE DataModel SYSTEM "blaise.dtd">
<!-- ?xml-stylesheet type="text/xsl" href="blaise1.xsl" ?-->
<DataModel>
  <ModelName>[DATAMODELNAME]</ModelName>
  <ModelText>[DATAMODELTITLE]</ModelText>
[FIELDSLOOP]
  [IF TYPE = STRING OR TYPE = INTEGER OR TYPE = REAL OR
      TYPE = ENUMERATED OR TYPE = SET THEN]
    [:2]<Question Qname="[FIELDNAME]">
    [&][IF FIELDLABEL = '' THEN]
       [T:= FIELDNAME][ELSE][T:= FIELDLABEL][ENDIF]
    [:4]<Qtext>[T]</Qtext>
    [:4]<FilePos First="[FIRSTPOSITION]" Last="[LASTPOSITION]"/>
    [IF TYPE = STRING THEN]
      [:4]<Open Length="[FIELDLENGTH]"/>
    [ELSEIF TYPE = INTEGER OR TYPE = REAL THEN]
      [:4]<Numeric Min="[LOWVALUE]" Max="[HIGHVALUE]"
              Dec="[NUMBEROFDECIMALS]"/>
    [ELSEIF (TYPE = ENUMERATED) OR (TYPE = SET) THEN]
      [IF TYPE = ENUMERATED THEN]
        [:4]<Closed Max="1">
      [ELSE]
        [:4]<Closed Max="[NUMBEROFCHOICES]">
      [ENDIF]
      [ANSWERLOOP]
        [&][IF ANSWERTEXT = '' THEN]
        [T:= ANSWERNAME][ELSE][T:= ANSWERTEXT][ENDIF]
        [:6]<Item Code="[ANSWERCODE]" Name="[ANSWERNAME]"
              Label="[T]"/>
      [ENDANSWERLOOP]
      [:4]</Closed>
    [ENDIF]
    [:2]</Question>
  [ENDIF]
[ENDFIELDSLOOP]
</DataModel>
```

Appendix B. The XML Data Type Definition for Blaise

```
<!-- Blaise Data Type Definition - version 1.0 -->

<!ELEMENT DataModel (ModelName, ModelText?, Question+) >

<!ELEMENT ModelName (#PCDATA) >

<!ELEMENT ModelText (#PCDATA) >

<!ELEMENT Question (Qtext, FilePos, (Open | Closed | Numeric)) >
<!ATTLIST Question Qname CDATA #REQUIRED >

<!ELEMENT Qtext (#PCDATA) >

<!ELEMENT FilePos EMPTY >
<!ATTLIST FilePos First CDATA #REQUIRED >
<!ATTLIST FilePos Last CDATA #REQUIRED >

<!ELEMENT Open EMPTY >
<!ATTLIST Open Length CDATA #REQUIRED >

<!ELEMENT Closed (Item+) >
<!ATTLIST Closed Max CDATA #REQUIRED >

<!ELEMENT Item EMPTY >
<!ATTLIST Item Code CDATA #REQUIRED >
<!ATTLIST Item Name CDATA #REQUIRED >
<!ATTLIST Item Label CDATA #IMPLIED >

<!ELEMENT Numeric EMPTY >
<!ATTLIST Numeric Min CDATA #REQUIRED >
<!ATTLIST Numeric Max CDATA #REQUIRED >
<!ATTLIST Numeric Dec CDATA #REQUIRED >
```

Appendix C. XSL style sheet for HTML question documentation

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/">
<HTML>
<BODY>
<H1><xsl:value-of select="DataModel/ModelName"/></H1>
<H2><xsl:value-of select="DataModel/ModelText"/></H2>

<xsl:for-each select="//Question" >
<TABLE BORDER="1" WIDTH="80%">
<TR><TD><B><xsl:value-of select="@Qname" /></B></TD></TR>
<TR><TD><xsl:value-of select="Qtext" /></TD></TR>
<xsl:choose>
<xsl:when match="Question[Closed]">
<TR><TD>Closed question</TD></TR>
</xsl:when>
<xsl:when match="Question[Open]">
<TR><TD>Open question</TD></TR>
</xsl:when>
<xsl:when match="Question[Numeric]">
<TR><TD>Numeric question</TD></TR>
</xsl:when>
</xsl:choose>
</TABLE>
<P/>
</xsl:for-each>
</BODY>
</HTML>
</xsl:template>
</xsl:stylesheet>
```

Appendix D. XSL style sheet for an SPSS setup

```
<?xml version="1.0"?>
 <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
 <xsl:script language="javascript">
 function B(N)
 { var T="                    "
   return T.substring(0, N)
 }
 </xsl:script>
 <xsl:template match="/">

 TITLE '<xsl:value-of select="DataModel/ModelText"/>'.
 DATALIST FILE='<xsl:value-of select="DataModel/ModelName"/>.asc' /
 <xsl:for-each select="//Question" >
 <xsl:eval language="javascript">B(3)</xsl:eval>
 <xsl:value-of select="@Qname" />
 <xsl:eval language="javascript">B(1)</xsl:eval>
 <xsl:value-of select="FilePos/@First" />-
 <xsl:value-of select="FilePos/@Last" />
 </xsl:for-each>
 <xsl:eval language="javascript">B(3)</xsl:eval>.

 VAR LABELS
 <xsl:for-each select="//Question" >
 <xsl:eval language="javascript">B(3)</xsl:eval>
 <xsl:value-of select="@Qname" />
 <xsl:eval language="javascript">B(1)</xsl:eval>'
 <xsl:value-of select="Qtext" />'
 </xsl:for-each>
 <xsl:eval language="javascript">B(3)</xsl:eval>.

 VALUE LABELS
 <xsl:for-each select="//Question[Closed]" >
 <xsl:eval language="javascript">B(3)</xsl:eval>
 <xsl:value-of select="@Qname" />
 <xsl:for-each select="Closed/Item" >
 <xsl:choose>
 <xsl:when match="Item[end()]">
 <xsl:eval language="javascript">B(6)</xsl:eval>
 <xsl:value-of select="@Code" />
 <xsl:eval language="javascript">B(1)</xsl:eval>'
 <xsl:value-of select="@Label" />'/
 </xsl:when>
 <xsl:otherwise>
 <xsl:eval language="javascript">B(6)</xsl:eval>
 <xsl:value-of select="@Code" />
 <xsl:eval language="javascript">B(1)</xsl:eval>'
 <xsl:value-of select="@Label" />'
 </xsl:otherwise>
 </xsl:choose>
 </xsl:for-each>
 </xsl:for-each>
 <xsl:eval language="javascript">B(3)</xsl:eval>.

 SAVE /OUTFILE '<xsl:value-of select="DataModel/ModelName"/>.sav'.
 </xsl:template>
 </xsl:stylesheet>
```

Appendix E. An HTML version of Cameleon

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="Javascript">
function Transform()
{ if (document.Form1.Script[0].checked)
  { ScrFile = "html.xsl"
    OutFile = document.Form1.InputFile.value + ".htm"
  }
  else
  { ScrFile = "spss.xsl"
    OutFile = document.Form1.InputFile.value + ".sps"
  }
  InFile = document.Form1.InputFile.value + ".xml"
  xmlFile = new ActiveXObject('microsoft.XMLDOM')
  xmlFile.load(InFile)
  xslFile = new ActiveXObject('microsoft.XMLDOM')
  xslFile.load(ScrFile)
  output = xmlFile.transformNode(xslFile)
  fso = new ActiveXObject("Scripting.FileSystemObject")
  outFile = fso.CreateTextFile(OutFile, true)
  outFile.Write(output)
  outFile.Close()
  document.Form1.Result.value = OutFile
}
</SCRIPT>
</HEAD>

<BODY>
<P>
<FORM Name="Form1">
<CENTER>
<TABLE BORDER="1" WIDTH="50%" Cellpadding="10">
  <TR><TD><H1>Cameleon - XML</H1></TD></TR>
  <TR><TD>
      Specify XML Input File (without extension):<P>
      <INPUT Type="text" Name="InputFile" Size="40">
      </TD>
  </TR>
  <TR><TD>
      Specify Cameleon Script:<P>
      <INPUT Type="radio" Name="Script" Value="HTML" Checked>
      HTML Documentation<BR>
      <INPUT Type="radio" Name="Script" Value="SPSS">
      SPSS Setup<BR>
      </TD>
  </TR>
  <TR><TD>
      <INPUT Type="button" Name="Execute" Value="Execute"
       onClick="Transform()">
      </TD>
  </TR>
  <TR><TD>
      The following file has been created:<P>
      <INPUT Type="text" Name="Result" Size="40">
      </TD>
  </TR>
</TABLE>
</CENTER>
</FORM>
<P>
</BODY>
</HTML>
```

Appendix F. Visual Basic version of Cameleon

```
Private Sub EndButton_Click()
  End
End Sub

Private Sub ExecButton_Click()
Dim OutFil As String
If xmlOpen.filename = "" Then
  Exit Sub
End If
Select Case ScriptNum
  Case 0
    xslDoc.Load ("html.xsl")
    OutFil = ShortFileName + ".htm"
  Case 1
    xslDoc.Load ("spss.xsl")
    OutFil = ShortFileName + ".sps"
End Select
outDoc = xmlDoc.transformNode(xslDoc)
Open OutFil For Output As #1
Print #1, outDoc
Close #1
Result.Text = "File " + OutFil + " has been generated"
End Sub

Private Sub Form_Load()
  Scripts(0).Value = True
  ScriptNum = 0
  Set xmlDoc = New DOMDocument
  Set xslDoc = New DOMDocument
End Sub

Private Sub Scripts_Click(Index As Integer)
  ScriptNum = Index
End Sub

Private Sub SelButton_Click()
  xmlOpen.filename = ""
  xmlOpen.ShowOpen
  If xmlOpen.filename = "" Then
     End
  End If
  xmlDoc.Load (xmlOpen.filename)
  If xmlDoc.parseError.errorCode <> 0 Then
    Result.Text = xmlDoc.parseError.reason
    Exit Sub
  End If
  Source.Text = xmlOpen.filename
  ShortFileName = xmlOpen.FileTitle
  ShortFileName = Left(ShortFileName, Len(ShortFileName) - 4)
End Sub
```

Appendix G. Interactive Question Browser in Visual Basic

```
Private Sub DispQuestion(I)
  Dim S, T As String
  Dim J, K As Integer
  Set e = eList.Item(I)
  Set ec = e.attributes.getNamedItem("Qname")
  Qname.Text = ec.firstChild.nodeValue
  Set ec = e.getElementsByTagName("Qtext").Item(0)
  QText.Text = ec.firstChild.nodeValue
  Set e = e.firstChild.nextSibling.nextSibling
  T = e.nodeName
  QType.Text = T
  Select Case T
  Case "Open"
    Grid.Visible = False
    Qanswer.Visible = True
    Set ec = e.attributes.getNamedItem("Length")
    S = "Text of atmost " + ec.firstChild.nodeValue + " characters"
    Qanswer.Text = S
  Case "Numeric"
    Grid.Visible = False
    Qanswer.Visible = True
    S = "Number between "
    Set ec = e.attributes.getNamedItem("Min")
    S = S + ec.firstChild.nodeValue + " and "
    Set ec = e.attributes.getNamedItem("Max")
    S = S + ec.firstChild.nodeValue
    Qanswer = S
  Case "Closed"
    Qanswer.Visible = False
    Grid.Visible = True
    Set iList = e.getElementsByTagName("Item")
    For K = 0 To iList.length - 1
      Grid.Row = K + 1
      Set ec = iList.Item(K)
      T = ec.attributes.getNamedItem("Code").firstChild.nodeValue
      Grid.Col = 0
      Grid.Text = T
      T = ec.attributes.getNamedItem("Name").firstChild.nodeValue
      Grid.Col = 1
      Grid.Text = T
      T = ec.attributes.getNamedItem("Label").firstChild.nodeValue
      Grid.Col = 2
      Grid.Text = T
    Next K
    For K = iList.length To 19
      Grid.Row = K + 1
      Grid.Col = 0
      Grid.Text = ""
      Grid.Col = 1
      Grid.Text = ""
      Grid.Col = 2
      Grid.Text = ""
    Next K
  End Select
End Sub

Private Sub Disp(Text As String)
  Result.AddItem Text
End Sub

Private Sub CloseButton_Click()
  End
End Sub
```

```
Private Sub Command1_Click()
    Dim I, p, n As Integer
    Set xmldoc = New DOMDocument
    Result.Clear

    xmlOpen.filename = ""
    xmlOpen.ShowOpen
    If xmlOpen.filename = "" Then
      Exit Sub
    End If

    xmldoc.Load (xmlOpen.filename)
    If xmldoc.parseError.errorCode <> 0 Then
      Result.AddItem (xmldoc.parseError.reason)
      Exit Sub
    Else
      Result.AddItem ("XML file is OK")
    End If

    QList.Clear

    Set e = xmldoc.getElementsByTagName("ModelName").Item(0)
    MName.Text = e.firstChild.nodeValue

    Set e = xmldoc.getElementsByTagName("ModelText").Item(0)
    MText.Text = e.firstChild.nodeValue

    Set eList = xmldoc.getElementsByTagName("Question")
    n = eList.length
    I = 0
    Do While I < n
      Set e = eList.Item(I)
      Set ec = e.attributes.getNamedItem("Qname")
      QList.AddItem ec.firstChild.nodeValue
      I = I + 1
    Loop
    I = 0
    DispQuestion (0)
    QList.ListIndex = 0
    QList.SetFocus
End Sub

Private Sub Form_Load()
  Grid.Rows = 21
  Grid.ColWidth(0) = 300
  Grid.ColWidth(1) = 1000
  Grid.ColWidth(2) = 8000
  Grid.Row = 0
  Grid.Col = 1
  Grid.Text = "Name"
  Grid.Col = 2
  Grid.Text = "Text"
  Grid.Visible = True
  Qanswer.Visible = True
End Sub

Private Sub QList_Click()
    DispQuestion (QList.ListIndex)
End Sub
```