Delphi Developer's Guide to XML

Second Edition

Keith Wood

Cataloging-in-Publication Data

Wood, Keith, 1961-
Delphi Developer's Guide to XML, Second Edition / by Keith Wood.
ISBN 1-59109-862-9 (pbk.)
1. XML (Document markup language). 2. Delphi (Computer file). 3. Computer software—Development. I. Title

kbwood@iprimus.com.au

Published by BookSurge
5341 Dorchester Road Suite 16
North Charleston, SC, 29418

ISBN 1-59109-862-9
10 9 8 7 6 5 4 3 2 1

Contents

Preface

This book is designed as an introduction to XML and an examination of how XML can be used in conjunction with Delphi.

What is in the Book?

The **Appendices** include additional material from the first edition of this book. Firstly there is a discussion of the CUESoft DOM from the CUEXml package. Next follows a customizable mass mail-out program, based around the Microsoft DOM, that uses XML for its configuration file and for the message template. Finally there is a customized Windows client program for a particular class of XML documents, based on SAX for Pascal, with a description of how to automatically invoke it for appropriate content downloaded from the Internet.

Conventions

The main text of the book is set in a proportional font (like this), while terms introduced for the first time appear in *italics*, as do emphasized items. Code samples, references to Delphi classes and methods, the names of directories and files, and entered text are presented in a `fixed font`. The names of menu items and other UI controls appear in a sans-serif font.

Throughout the book various items are marked so as to bring them to your attention. The notations used and their meanings are described below:

NOTE
A note is information of probable interest regarding the surrounding text.

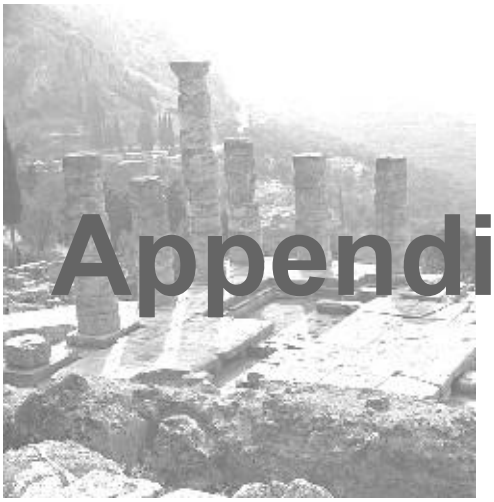
TIP
A tip is something to make your development with XML easier.

WARNING
A warning is something that you need to be aware of regarding the preceding text.

Code Downloads

The code samples used in this book are available on the accompanying Web site: http://home.iprimus.com.au/kbwood/DelphiXML. Downloads are arranged by chapter, with the addition of a single package that includes all the code.

This Web site also provides links to the various specifications for XML and its related technologies. Links to Delphi resources for XML also appear.



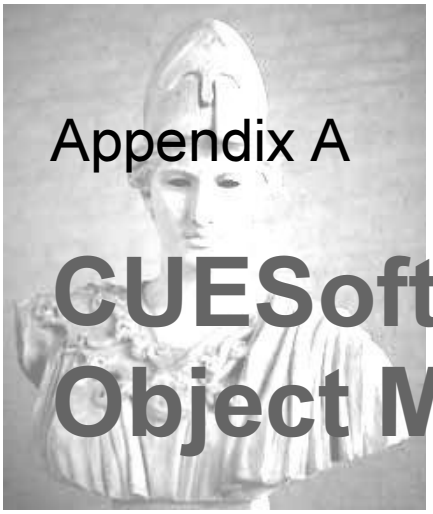
Appendices

These appendices include chapters from the first edition of this book.

Appendix A covers the CUESoft Document Object Model. Although this DOM is no longer commercially available, it is still in use, and forms the basis for TurboPower's XML Partner suite.

Appendix B presents a program that performs mass electronic mail-outs by merging database information with a message template and sending these out using SMTP. XML is used to specify the configuration file as well as the message template, which allows for the embedding of values from database fields within the text. The template also includes the actual SQL query used to obtain the information in the first place. For processing the XML, Microsoft's XML parser is used.

Appendix C demonstrates how to process XML documents into a customized client written in Delphi. The movie-watcher documents are used as the example, and are shown in a custom GUI with appropriate navigation links between the sections. A SAX-compliant parser is used to process the XML, demonstrating an implementation of the `IContentHandler` interface. As a bonus you see how to set up your browser to automatically open the movie-watcher documents in the new client when they are downloaded.



Appendix A

CUESoft's Document Object Model

CUESoft has also implemented the DOM specification under Windows, this time as a set of native Delphi objects. The advantage of having native objects is that the parser and DOM can be compiled directly into your program, with no need to worry about mismatched DLLs.

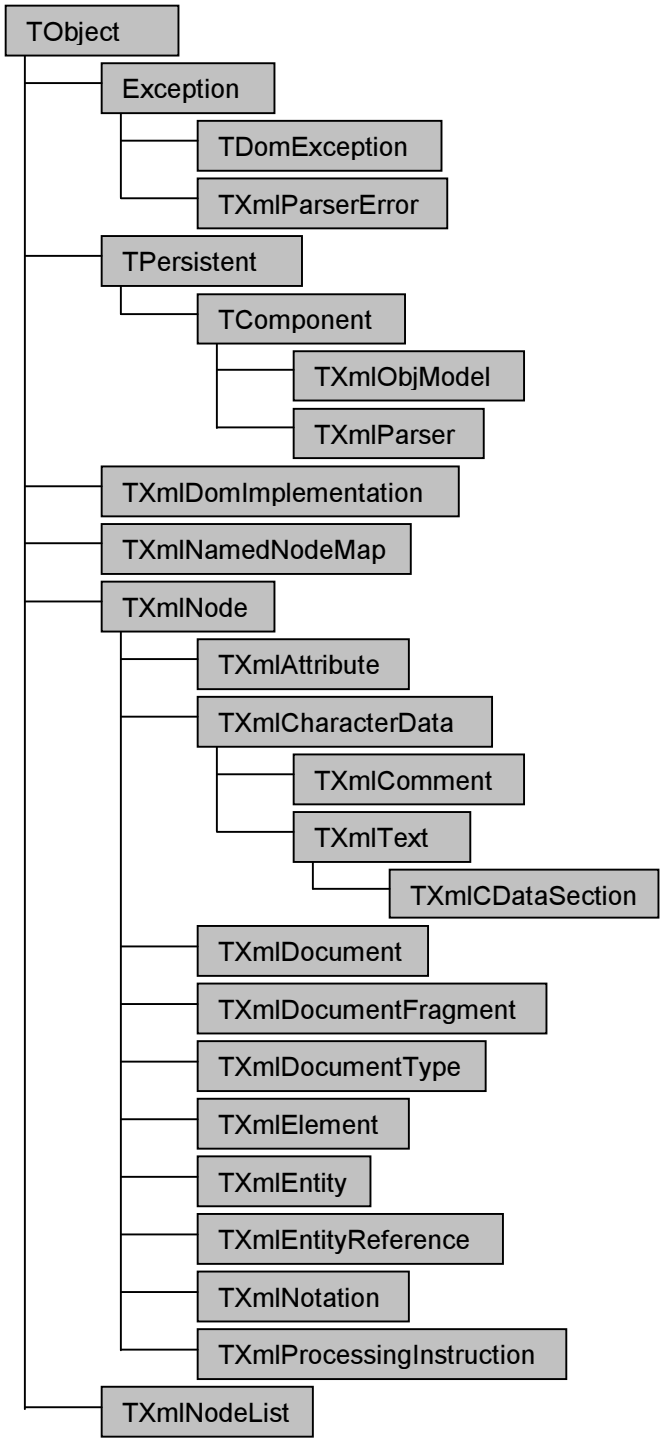
The W3C DOM interfaces are implemented as classes in the CUEXml Delphi package, with the class hierarchy shown in Figure A-1. CUESoft follows the DOM Level 1 specification very closely, although they also have several extensions for increased functionality and usability. They do have some support for namespaces, but handle only string values, not the expected `WideStrings`.

The CUESoft DOM was a commercial product that was acquired by TurboPower. If you have the source you can compile it into any 32-bit version of Delphi. Otherwise, there are prepackaged libraries available for Delphi versions 3 through 5. In general, all you have to do to install the package is as follows:

1. Unpack the files from CUESoft into an appropriate directory.
2. Select Component | Install Packages… from the Delphi menu.
3. Click the Add button.
4. Change the file type to Package collection (*.dpc), and browse to the directory where you unpacked the files.
5. Select `cuexml2_`*n*`.dpc` (where *n* is your version of Delphi) and click OK.
6. Click Finish on the package installation dialog and OK on the package dialog. The two components in the package appear on the CUESoft tab in the component palette.

Each of the classes is described in further detail below. Differences from the DOM specification are noted as they are encountered. Unless otherwise noted, all these classes appear in the `XmlObjModel` unit.

Figure A-1: The CUESoft DOM class hierarchy.



TDOMException Exception

General errors that occur during DOM processing within the CUESoft package are notified as `TDOMException`s (see Listing A-1). Following the DOM specification, it adds only a single numeric code to denote which type of problem arose.

Listing A-1: The TDOMException exception.

```
TDOMException = class(Exception)
public
  constructor CreateCode(oCode: EExceptionCode);
  property Code: EExceptionCode read FCode;
end;
```

The properties and methods of a `TDOMException` object are described below:

`constructor CreateCode(oCode: EExceptionCode);`
: This constructor generates a new exception passing in the type of error encountered. Typically you would not create these exceptions yourself, but would react to those raised by the DOM during its processing.

`property Code: EExceptionCode read FCode;`
: This read-only property indicates what type of problem arose. Its value is one of those shown in Table A-1. As you can see, these follow the DOM specification.

Table A-1: CUEXml error codes.

Error Code	Meaning
ecNone	No error
ecIndexSizeErr	An index or size is negative or greater than allowed.
ecWStringSizeErr	The text does not fit into a WideString.
ecHierarchyRequestErr	A node is inserted somewhere it does not belong.
ecWrongDocumentErr	A node from another document is used.
ecInvalidNameErr	An invalid name is used (usually containing an illegal character).
ecNoDataAllowedErr	Data is specified for a node that does not support data.
ecNoModificationAllowedErr	Attempting to modify a read-only node.
ecNotFoundErr	The specified node cannot be found in this context.
ecNotSupportedErr	The action specified for the object is not supported under this implementation.
ecInuseAttributeErr	An attribute already belonging to one element is being added to another.
ecInvalidParamErr	An invalid parameter is passed to a method.

TXmlParserError Exception

Errors arising from the parsing of a document are indicated through a `TXmlParserError` exception (see Listing A-2). These include further details about the reason for and position of the error. This class appears in the `XmlParser` unit.

NOTE
There is no equivalent of this exception in the W3C DOM specification.

Listing A-2: The TXmlParserError exception.

```
TXmlParserError = class(Exception)
public
  constructor CreateParseError(oFilePos, oLine, oLinePos: Integer;
    oUrl, oReason: string);
  property Reason: string read FReason;
  property Line: Integer read FLine;
  property LinePos: Integer read FLinePos;
  property FilePos: Integer read FFilePos;
  property Url: string read FUrl;
end;
```

A `TXmlParserError` object's properties and methods are listed below:

`constructor CreateParseError(oFilePos, oLine, oLinePos: Integer; oUrl, oReason: string);`
Build a new exception during the parse process with this constructor. The parameters set all the properties for this exception. Generally the parser itself raises these errors, and you only need to respond to them.

`property Reason: string read FReason;`
Retrieve a text description of the problem through this read-only property.

`property Line: Integer read FLine;`
This read-only property returns the line number in the XML document where the error was detected.

`property LinePos: Integer read FLinePos;`
The character position within that line is given by this read-only property.

`property FilePos: Integer read FFilePos;`
Find the offset into the XML document as a whole with this read-only property.

`property Url: string read FUrl;`
This read-only property returns the source name of the XML document in error.

TXmlNode Class

All nodes within the DOM structure derive from a common class that provides the basic functionality used by most of them. The `TXmlNode` class (shown in Listing A-3) embodies this in the CUESoft package.

Listing A-3: The TXmlNode declaration.

```
TXmlNode = class
protected
  FNodeId: Integer;
  FNodeName: string;
  FNodeType: Integer;
  FNodeValue: string;
  function GetNodeStringType: string;
public
  constructor Create;
  destructor Destroy; override;
  property Attributes: TXmlNamedNodeMap read FAttributes;
  property BaseName: string read GetBaseName;
  property ChildNodes: TXmlNodeList read FChildNodes;
  property FirstChild: TXmlNode read GetFirstChild;
  property LastChild: TXmlNode read GetLastChild;
  property LevelCode: string read GetLevelCode;
  property Namespace: string read GetNamespace;
  property NextSibling: TXmlNode read GetNextSibling;
  property NodeId: Integer read FNodeId write FNodeId;
  property NodeName: string read FNodeName write FNodeName;
  property NodeStringType: string read GetNodeStringType;
  property NodeType: Integer read FNodeType;
  property NodeValue: string read FNodeValue write FNodeValue;
  property OwnerDocument: TXmlDocument read GetOwnerDocument;
  property ParentNode: TXmlNode read FParentNode;
  property Prefix: string read GetPrefix;
  property PreviousSibling: TXmlNode read GetPreviousSibling;
  property Text: string read GetText;
  property XmlDocument: string read GetXmlDocument;
  procedure AddRef;
  procedure AppendChild(oNewChild: TXmlNode);
  function CloneNode(bDeep: Boolean = True): TXmlNode;
  procedure ForceOwnerDocument(oNode: TXmlNode);
  function GetChildNodesByNodeType(wType: Integer): TXmlNodeList;
  function GetNodesByNodeType(wType: Integer): TXmlNodeList;
  function HasAttributes: Boolean;
  function HasChildNodes: Boolean;
  procedure InsertBefore(oNewChild, oRefChild: TXmlNode);
  function IsAfter(oNode: TXmlNode): Boolean;
  procedure Release;
  procedure RemoveAll;
  function RemoveChild(oRefChild: TXmlNode): TXmlNode;
  function ReplaceChild(oNewChild, oRefChild: TXmlNode): TXmlNode;
end;
```

Using functionality from the basic node when it is not applicable results in an exception being thrown – for example, attempting to add child nodes to a text node. Simple properties return an empty string or `nil` if they do not apply to the current node type.

The properties and methods of a `TXmlNode` object are detailed below:

`constructor Create;`

Do not create `TXmlNodes` directly. They are only used within the DOM hierarchy as one of this class' subclasses.

`property Attributes: TXmlNamedNodeMap read FAttributes;`

Access the attributes of a node with this read-only property. It returns a named node map containing `TXmlAttribute` objects. Although it is defined on all nodes, only element nodes can contain attributes. All other types return `nil`.

`property BaseName: string read GetBaseName;`

Retrieve the local part of the node's name – the part after any namespace prefix – through this read-only property.

NOTE
In the W3C DOM specification, the local part of the node's name is given by the `localName` attribute.

`property ChildNodes: TXmlNodeList read FChildNodes;`

Moving down through the document tree uses this read-only property. It returns a "live" list of ordered nodes, meaning that any changes to the nodes in the list immediately update the main structure, and vice versa. If a node has no children, this property still returns a valid list, but that list has no entries in it.

`property FirstChild: TXmlNode read GetFirstChild;`

This convenience property returns the first entry in the `ChildNodes` list or `nil` if there are no children.

`property LastChild: TXmlNode read GetLastChild;`

Similarly, this property returns the last entry in the `ChildNodes` list, or `nil` if none.

`property LevelCode: string read GetLevelCode;`

This read-only property returns the node's location within the DOM hierarchy as a sequence of numbers separated by periods. Each number represents the position of the node's ancestors within their parent's list of children (although counting here starts at one). For example, `4.1.2` is the node at the second position in the node at the first position in the node at the fourth position in the document.

NOTE
The `LevelCode` property is not part of the W3C DOM specification.

`property Namespace: string read GetNamespace;`

The namespace descriptor for the node comes from this read-only property. It is blank if no namespace applies to the node. Namespaces are declared through "`xmlns`" prefixed attributes.

NOTE
In the W3C DOM specification, the namespace for the node is given by the `namespaceURI` attribute.

`property NextSibling: TXmlNode read GetNextSibling;`

Another convenience property, this one retrieves the node after the current one in its parent's list of children. Again, a `nil` is returned if there is no following node.

`property NodeId: Integer read FNodeId write FNodeId;`

Use this property to define your own ID for each node, separate from any that may be defined in the document itself.

NOTE
The `NodeId` property is not part of the W3C DOM specification.

`property NodeName: string read FNodeName write FNodeName;`

The name of the node is given by this property. For some nodes this is a predefined value. See Table A-2 for the meaning of this property based on the node's type.

Table A-2: Node name and value by node type.

Node Type	Node Name	Node Value
ELEMENT_NODE	Name of element	"" (Empty string)
ATTRIBUTE_NODE	Name of attribute	Attribute value
TEXT_NODE	"#text"	Content of text
CDATA_SECTION_NODE	"#cdata-section"	Content of CDATA section
ENTITY_REFERENCE_NODE	Name of entity	""
ENTITY_NODE	Name of entity	""
PROCESSING_INSTRUCTION_NODE	Target of instruction	Content excluding target
COMMENT_NODE	"#comment"	Content of comment
DOCUMENT_NODE	"#document"	""
DOCUMENT_TYPE_NODE	Name of document type	""
DOCUMENT_FRAGMENT_NODE	"#document-fragment"	""
NOTATION_NODE	Name of notation	""

`property NodeStringType: string read GetNodeStringType;`

This read-only property retrieves the node's type as a string value. It returns the node types from Table A-3 as text.

NOTE
The `NodeStringType` property is an extension to the W3C DOM specification.

`property NodeType: Integer read FNodeType;`

This read-only property identifies the type of node represented by this object, allowing it to be safely cast to that type to access its additional abilities. The value is one of those shown in Table A-3, alongside the corresponding subclass.

`property NodeValue: string read FNodeValue write FNodeValue;`

Retrieve or set the text value of the node through this property. Many node types do not use this property, as shown in Table A-2.

Table A-3: Node types.

Node Type	Implementing Class
ELEMENT_NODE	TXmlElement
ATTRIBUTE_NODE	TXmlAttribute
TEXT_NODE	TXmlText
CDATA_SECTION_NODE	TXmlCDataSection
ENTITY_REFERENCE_NODE	TXmlEntityReference
ENTITY_NODE	TXmlEntity
PROCESSING_INSTRUCTION_NODE	TXmlProcessingInstruction
COMMENT_NODE	TXmlComment
DOCUMENT_NODE	TXmlDocument
DOCUMENT_TYPE_NODE	TXmlDocumentType
DOCUMENT_FRAGMENT_NODE	TXmlDocumentFragment
NOTATION_NODE	TXmlNotation

```
property OwnerDocument: TXmlDocument read
    GetOwnerDocument;
```

All nodes contain a reference to the document that created them, which is available through this read-only property. For document nodes this returns `nil`.

```
property ParentNode: TXmlNode read FParentNode;
```

Once placed into a DOM structure, this read-only property lets you reach the parent of the node. The parent is `nil` for attribute, document, and document fragment nodes, as well as for other nodes that have not yet been added to the tree.

```
property Prefix: string read GetPrefix;
```

This read-only property returns the namespace prefix – the part up to the colon (:) – from the node's name, or an empty string if there is no prefix.

```
property PreviousSibling: TXmlNode read
    GetPreviousSibling;
```

This property retrieves the node before this one in its parent's list. `nil` is returned if there is no previous node.

```
property Text: string read GetText;
```

Retrieve all the text from this node and its descendants concatenated together via this read-only property.

NOTE
Both the `Text` and `XmlDocument` properties are extensions to the W3C DOM specification.

```
property XmlDocument: string read GetXmlDocument;
```

Extract the XML fragment that corresponds to this node and all of its descendants from this read-only property.

`procedure AddRef;`

Add a reference count to this node. Use `Release` to decrement the count. This method is automatically called when the node is created, and again when it is added to the tree.

NOTE
The `AddRef` method is not part of the W3C DOM specification.

`procedure AppendChild(oNewChild: TXmlNode);`

Adds the specified node to the end of this node's list of children. If the supplied node is already in the structure, it is first removed. Adding a document fragment node adds all of its children instead.

`function CloneNode(bDeep: Boolean = True): TXmlNode;`

Create a copy of the node through this method, including any attributes and their values. Attributes resulting from default values in the DTD are also duplicated. If the `bDeep` parameter is `False`, the process stops there. If it is `True`, all the descendants of this node are also cloned under the copy. The new duplicate has no parent until it is placed back into the DOM hierarchy.

`procedure ForceOwnerDocument(oNode: TXmlNode);`

Set the `OwnerDocument` property for the supplied node and all its descendants to be the same as the current node. This lets you transfer nodes from one document to another.

NOTE
The `ForceOwnerDocument` method is not part of the W3C DOM specification, but is akin to the `importNode` method.

`function GetChildNodesByNodeType(wType: Integer): TXmlNodeList;`

Retrieve a node list containing all the immediate child nodes of a given type. The types are specified using the values shown in Table A-3.

NOTE
Neither of the `GetChildNodesByNodeType` or `GetNodesByNodeType` methods are part of the W3C DOM specification, although they function somewhat like a `NodeIterator`.

`function GetNodesByNodeType(wType: Integer): TXmlNodeList;`

Similarly, this method returns a list of all descendants of the specified type.

`function HasAttributes: Boolean;`

This method returns `True` when there are entries in the attributes list and `False` when there are none.

`function HasChildNodes: Boolean;`

A convenience function, this returns `True` when there are child nodes in the list and `False` when it is empty.

`procedure InsertBefore(oNewChild, oRefChild: TXmlNode);`

Place the new node immediately before the specified reference node within this node's list of children. If the reference node is `nil`, the new node is placed at the end of the list. A new node already in the tree is first removed. Inserting a document fragment node adds all of its children instead.

`function IsAfter(oNode: TXmlNode): Boolean;`

This function returns `True` if the current node appears after the given node in a pre-order traversal of the hierarchy, and `False` if it does not. For example, a node is after its parent and any previous sibling, but it is before any next sibling and any child nodes.

NOTE
The `IsAfter` method is not part of the W3C DOM specification.

`procedure Release;`

Decrement the reference count for this node. When the count reaches zero, the object is destroyed. Be sure to call this method once you are finished with the node after adding it to the tree.

NOTE
The `Release` method is not part of the W3C DOM specification.

`procedure RemoveAll;`

Delete all child nodes from the list and destroy the node objects.

NOTE
The `RemoveAll` method is not part of the W3C DOM specification.

`function RemoveChild(oRefChild: TXmlNode): TXmlNode;`

Removes the specified node from this node's list of children. A reference to that node is returned. The old node should be released once the method is finished.

`function ReplaceChild(oNewChild, oRefChild: TXmlNode): TXmlNode;`

Remove the specified reference node and insert the new node in its place. The function returns a pointer to the node that is removed.

NOTE
Although the CUESoft package does not explicitly support DOM Level 2, it does include several properties dealing with namespaces. Missing from the DOM Level 2 specification are the `normalize` and `isSupported` methods. `normalize` does appear in the `TXmlElement` class in CUESoft's package, while `isSupported` is duplicated by the `HasFeature` method of the `TXmlDomImplementation` class.

TXmlNodeList Class

The `TXmlNodeList` class (see Listing A-4) encapsulates an ordered collection of nodes. It is the object returned by the `ChildNodes` property of a node, as well as by the various `GetNode*` methods. Items within the list are accessed sequentially by their position.

Listing A-4: The TXmlNodeList declaration.

```
TXmlNodeList = class
public
  constructor Create;
  destructor Destroy; override;
  property Length: Integer read GetLength;
  property XmlDocument: string read GetXmlDocument;
  procedure Add(oNode: TXmlNode);
  procedure Delete(wIndex: Integer);
  procedure Empty;
  function Exchange(wSrc, wDest: Integer): Boolean;
  function IndexOf(oNode: TXmlNode): Integer;
  procedure Insert(wIndex: Integer; oNode: TXmlNode);
  function Item(wIndex: Integer): TXmlNode;
  function Move(wSrc, wDest: Integer): Boolean;
  procedure Replace(wIndex: Integer; oNode: TXmlNode);
  procedure Sort(sAttribute: string = ''; wOrder: Integer = 0);
end;
```

The `TXmlNodeList` object's properties and methods are shown below:

`constructor Create;`
: Lists are automatically created for you as the result of a query, or through a node's `ChildNodes` property.

`property Length: Integer read GetLength;`
: Find the number of entries in the list through this read-only property. Access the individual items with indexes in the range zero to `Length - 1`.

NOTE
All the properties and methods except for `Length` and `Item` are extensions to the DOM Level 2 specification. The specification intentionally left out methods for manipulating the node list, other than reading items out.

`property XmlDocument: string read GetXmlDocument;`
: This read-only property returns all the items in the list as a formatted XML fragment. It is not well-formed XML unless there is a single element type node in the list.

`procedure Add(oNode: TXmlNode);`
: Add the given node to the end of the list.

`procedure Delete(wIndex: Integer);`
: Removes the indicated node from the list.

`procedure Empty;`
: Deletes all the nodes from the list.

`function Exchange(wSrc, wDest: Integer): Boolean;`
: Swaps the positions of two entries in the list, given their locations. A `True` value returns if the exchange succeeds, and a `False` returns otherwise.

`function IndexOf(oNode: TXmlNode): Integer;`
: Finds the position of the specified node within the list. A `-1` value is returned if the node cannot be found.

`procedure Insert(wIndex: Integer; oNode: TXmlNode);`
: Places the specified node at the given position in the list.

`function Item(wIndex: Integer): TXmlNode;`
: Access each entry in the list with this function, giving the item's position within the list. If the index value is out of range, the function returns `nil`.

`function Move(wSrc, wDest: Integer): Boolean;`
: Moves an item in the list from its source position to its new destination location. The function returns `True` if the move succeeds, and `False` otherwise.

`procedure Replace(wIndex: Integer; oNode: TXmlNode);`
: Removes the item currently at the nominated index and puts the new node in its place.

`procedure Sort(sAttribute: string = ''; wOrder: Integer = 0);`
: Order the nodes in the list with this method. If an attribute name is supplied, the nodes sort by the value of that attribute. If the attribute name is left blank, the nodes appear in order of their text content. Use the last parameter to sort in ascending (`0`, the default) or descending (`1`) order.

  If the node list is the `ChildNodes` of an element, then sorting physically reorders the actual nodes within the DOM. For other lists, only that list is sorted, without affecting the DOM hierarchy.

TXmlNamedNodeMap Class

The `TXmlNamedNodeMap` class (see Listing A-5) also manages a list of nodes, but provides primary access to them via their names. Although you can also retrieve items by their position, this is merely a convenience and does not imply any particular ordering of the nodes.

Listing A-5: The TXmlNamedNodeMap declaration.

```
TXmlNamedNodeMap = class
public
  constructor Create;
  destructor Destroy; override;
  property Length: Integer read GetLength;
  procedure Add(oNode: TXmlNode);
  procedure Delete(wIndex: Integer);
  procedure Empty;
  function GetNamedItem(sName: string): TXmlNode;
  function IndexOf(oNode: TXmlNode): Integer;
```

```
  procedure Insert(wIndex: Integer; oNode: TXmlNode);
  function Item(wIndex: Integer): TXmlNode;
  function RemoveNamedNode(sName: string): TXmlNode;
  procedure Replace(wIndex: Integer; oNode: TXmlNode);
  function SetNamedItem(oNode: TXmlNode): TXmlNode;
end;
```

The properties and methods of the `TXmlNamedNodeMap` object are described below:

`constructor Create;`

As for node lists, these node maps are automatically created for you. The `Attributes` property of the `TXmlNode` class and the `Entities` and `Notations` properties of the `TXmlDocumentType` class all return node maps containing their respective node types.

`property Length: Integer read GetLength;`

Return the number of entries in the map through this read-only property.

`procedure Add(oNode: TXmlNode);`

Add the specified node to the list.

NOTE

The `Add`, `Delete`, and `Empty` methods are extensions to the W3C DOM specification.

`procedure Delete(wIndex: Integer);`

Remove the node at the given position from the list. An out of range index is ignored.

`procedure Empty;`

Remove all the nodes from the list.

`function GetNamedItem(sName: string): TXmlNode;`

Retrieves the node from the mapping that has the given name. A `nil` is returned if no node matches this name. The resulting node can be cast to its appropriate subclass to access its specific abilities.

`function IndexOf(oNode: TXmlNode): Integer;`

Return the position of the given node in the list. If the node is not found, the function returns `-1`.

NOTE

The `IndexOf` and `Insert` methods are extensions to the W3C DOM specification.

`procedure Insert(wIndex: Integer; oNode: TXmlNode);`

Place the new node at a particular position within the list. If the index is out of range, an error occurs.

`function Item(wIndex: Integer): TXmlNode;`

Access the entries in the list via their position. If the supplied index is out of range, a `nil` is returned.

`function RemoveNamedNode(sName: string): TXmlNode;`

Find the node in the mapping with the given name and remove it from the list. A reference to that node is returned. If no matching node is found, return a `nil` instead.

`procedure Replace(wIndex: Integer; oNode: TXmlNode);`

Delete the node currently in the specified position and insert the new node in its place. An error is raised if the index is out of range.

NOTE
The `Replace` method is an extension to the W3C DOM specification.

`function SetNamedItem(oNode: TXmlNode): TXmlNode;`

Adds the given node to the mapping, using its `NodeName` as the index. If an entry already exists with that name, the new node replaces it and a reference to the deleted node is returned. Otherwise, the return value is `nil`.

NOTE
Missing from the DOM specification are the namespace-aware versions of the `Get/Set/RemoveNamedItem` methods above.

TXmlElement Class

Most of the nodes in the DOM will be `TXmlElement` objects (as shown in Listing A-6). These represent the elements from the XML document, and typically have attributes and child nodes attached to them.

Listing A-6: The TXmlElement declaration.

```
TXmlElement = class(TXmlNode)
public
  constructor Create;
  destructor Destroy; override;
  property ElementText: string read GetElementText;
  property FullEndTag: Boolean read FFullEndTag write FFullEndTag;
  property IgnoreEndTag: Boolean read FIgnoreEndTag
    write FIgnoreEndTag;
  property TagName: string read FNodeName write FNodeName;
  function CreateChildCDataSection(sText: string): TXmlCDataSection;
  function CreateChildElement(sElem: string): TXmlElement;
  function CreateChildText(sText: string): TXmlText;
  function FindElement(sName: string): TXmlElement;
  function GetAttribute(sName: string): string;
  function GetAttributeNode(sName: string): TXmlAttribute;
  function GetChildElementsByTagName(sName: string): TXmlNodeList;
  function GetElementsByTagName(sName: string): TXmlNodeList;
  function GetElementsByTagNameWithAttribute(
    sName, sAttr, sValue: string): TXmlNodeList;
  function MatchExpression(sTerm: string): TXmlNodeList;
  procedure Normalize(bAddSpace: Boolean = False);
  procedure RemoveAttribute(sName: string);
  function RemoveAttributeNode(oOldAttr: TXmlAttribute): TXmlAttribute;
  function SelectNodes(sQuery: string): TXmlNodeList;
  function SelectSingleNode(sQuery: string): TXmlElement;
  procedure SetAttribute(sName, sValue: string);
  function SetAttributeNode(oNewAttr: TXmlAttribute): TXmlAttribute;
end;
```

The `TXmlElement` object's properties and methods are listed below:

`constructor Create;`

Element nodes should not be created directly. Instead, use the `CreateElement` method on the document object or the `CreateChildElement` method described later.

`property ElementText: string read GetElementText;`

This read-only property returns the value of the single text node child of this element. If there is no single text child, it returns an empty string.

NOTE
The `ElementText`, `FullEndTag`, and `IgnoreEndTag` properties are not part of the DOM Level 2 specification.

`property FullEndTag: Boolean read FFullEndTag write FFullEndTag;`

Set this property to `True` to force the output of a full closing tag when generating XML. When `False` (the default), an element that has no children uses the shorthand syntax available in XML (placing a slash at the end of the opening tag). This property can be used to maintain compatibility with some existing applications (specifically HTML).

`property IgnoreEndTag: Boolean read FIgnoreEndTag write FIgnoreEndTag;`

Setting this property to `True` causes the end tag to be omitted entirely if the element has no children. By default, it is `False`, which always generates an end tag. Again, this property is intended for use with generating HTML, but should not be used in any true XML document.

`property TagName: string read FNodeName write FNodeName;`

Set or retrieve the name of the element through this property. It maps directly onto the inherited `NodeName` property.

`function CreateChildCDataSection(sText: string): TXmlCDataSection;`

This function creates a new CDATASection node and appends it to the element, returning a reference to the new node. You can achieve the same thing through the `CreateCDATASection` method on the document object, followed by an `AppendChild` call on this node.

NOTE
The `CreateChildCDataSection`, `CreateChildElement`, and `CreateChild-Text` methods are not part of the DOM Level 2 specification.

`function CreateChildElement(sElem: string): TXmlElement;`

Similarly, this function adds a newly created element node to the current element, and returns a pointer to it.

`function CreateChildText(sText: string): TXmlText;`

Lastly, you can easily create and add a child text node with this method. Again, you receive a reference to the new node as the return value.

`function FindElement(sName: string): TXmlElement;`

Find the first descendant element node with the given tag name through this method. The subtree is searched in a pre-order traversal. If no matching node is found, a `nil` is returned.

NOTE
The `FindElement` method is not part of the DOM Level 2 specification.

`function GetAttribute(sName: string): string;`

Although you could use the `Attributes` property to deal with an element's attributes, there are several convenience methods to assist you. This one returns the string value of the named attribute, or an empty string if it cannot be found.

`function GetAttributeNode(sName: string): TXmlAttribute;`

Access the entire attribute node by name with this method. If the attribute cannot be found, it returns `nil`.

`function GetChildElementsByTagName(sName: string): TXmlNodeList;`

Similar to the `GetElementsByTagName` method, this one only searches the immediate children of the element.

NOTE
The `GetChildElementsByTagName` method is not part of the DOM Level 2 specification.

`function GetElementsByTagName(sName: string): TXmlNodeList;`

Obtain a list of all the elements with a given name that are descendants of this node with this function. Use a name of "`*`" to get all elements in the subtree. The entries in the list appear in the same order as a pre-order traversal of the subtree.

`function GetElementsByTagNameWithAttribute(sName, sAttr, sValue: string): TXmlNodeList;`

Another variation on the `GetElementsByTagName` method, this one looks through all descendants, returning those elements that have the given name and also an attribute with the specified name and value.

NOTE
The `GetElementsByTagNameWithAttribute` method is not part of the DOM Level 2 specification, nor is the `MatchExpression` method below.

`function MatchExpression(sTerm: string): TXmlNodeList;`

This method searches the descendants of the element for nodes that match the given expression, and returns those found as a list. Their order in the list matches their order in a pre-order traversal of the hierarchy.

`procedure Normalize(bAddSpace: Boolean = False);`

Combine adjacent text nodes in the entire subtree beneath this element. Setting the `bAddSpace` parameter to `True` causes an extra space character to be placed between the contents of text nodes that are

concatenated. Doing so is not standard DOM functionality. However, the parameter has a default value of `False` and can safely be omitted.

NOTE
In the DOM Level 2 specification, the `Normalize` functionality has moved to the `Node` interface.

```
procedure RemoveAttribute(sName: string);
```
Remove the attribute with the given name using this method. Nothing happens if a matching node is not found.

```
function RemoveAttributeNode(oOldAttr: TXmlAttribute):
    TXmlAttribute;
```
Remove the specified attribute from the element's list through this method. A reference to that node is returned. If the given node is not an attribute of the element, nothing happens.

```
function SelectNodes(sQuery: string): TXmlNodeList;
```
Retrieves a list of all the nodes that match the given XPath expression (see Chapter 5). The current node acts as the starting point for relative references. An empty list is returned if no matching nodes are found.

NOTE
The `SelectNodes` and `SelectSingleNode` methods are not part of the DOM Level 2 specification.

```
function SelectSingleNode(sQuery: string): TXmlElement;
```
This method acts like the previous one, but returns only the first element found, or `nil` if there are none.

```
procedure SetAttribute(sName, sValue: string);
```
Set the value of an attribute with this method. Any existing attribute with the same name has its contents overwritten by the new value. The value supplied is not parsed at all.

```
function SetAttributeNode(oNewAttr: TXmlAttribute):
    TXmlAttribute;
```
Use this method to add attributes that have internal structure beyond a simple string value. Build your attribute node and attach its children before calling this method. The new node replaces any existing attribute with the same name, in which case a reference to the deleted node is returned. Otherwise, it returns `nil`.

NOTE
Missing abilities from the DOM specification include the namespace-aware versions of the methods above. Also, the `hasAttribute` and `hasAttributeNS` methods are not implemented, although the `IndexOf` method of the `Attributes` node map provides similar information.

TXmlAttribute Class

Attributes are attached to elements and are available through the `Attributes` property on the `TXmlElement` nodes. Other than appearing in these lists, they do not form a part of the normal DOM hierarchy. They have no parent and no siblings, so the corresponding properties return `nil`. Their CUESoft definition is shown in Listing A-7.

Listing A-7: The TXmlAttribute declaration.

```
TXmlAttribute = class(TXmlNode)
public
  constructor Create; override;
  destructor Destroy; override;
  property Name: string read FNodeName write FNodeName;
  property Specified: Boolean read FSpecified write FSpecified;
  property Value: string read GetNodeValue write SetNodeValue;
  function CloneNode(bDeep: Boolean = True): TXmlNode; override;
end;
```

The properties and methods of the `TXmlAttribute` object are discussed below:

`constructor Create;`
: As for elements, use the `CreateAttribute` factory method on the document object instead of creating attributes yourself. You can also instantiate them through the `SetAttribute` method of an element object.

`property Name: string read FNodeName write FNodeName;`
: Retrieve or set the name of the attribute through this property. It maps directly onto the inherited `NodeName` property.

`property Specified: Boolean read FSpecified write FSpecified;`
: This property returns `True` if the value for the attribute came from the body of the XML document itself or was set through the `Value` property, and `False` if the value derives from a default specified for this attribute in the DTD.

`property Value: string read GetNodeValue write SetNodeValue;`
: Read or write the string value of the attribute with this property. The inherited `NodeValue` property has the same effect. Setting this value causes any children of the attribute to be discarded and to be replaced with just the supplied text. The value is not parsed at all, so any embedded entity references are ignored.

TXmlCharacterData Class

The `TXmlCharacterData` class (see Listing A-8) is the basis of all textual nodes within the DOM. It supplies common functionality for the various subclasses. The base class itself does not appear in the hierarchy.

Listing A-8: The TXmlCharacterData declaration.

```
TXmlCharacterData = class(TXmlNode)
public
  property Data: string read FNodeValue write FNodeValue;
  property Length: Integer read GetLength;
  procedure AppendData(sData: string);
  procedure DeleteData(wOffset, wCount: Integer);
  procedure InsertData(wOffset: Integer; sData: string);
  procedure ReplaceData(wOffset, wCount: Integer; sData: string);
  function SubStringData(wOffset, wCount: Integer): string;
end;
```

The `TXmlCharacterData` object's properties and methods are listed below. As for the other DOM implementations, all offsets start at zero.

`property Data: string read FNodeValue write FNodeValue;`
: Retrieve or set the text content of the node through this property.

`property Length: Integer read GetLength;`
: Find the number of characters in the `Data` property, which may be zero.

`procedure AppendData(sData: string);`
: Add the supplied text to the end of the existing value. Retrieve the combined text from the `Data` property.

`procedure DeleteData(wOffset, wCount: Integer);`
: Remove the text starting from the given offset, for the given number of characters.

`procedure InsertData(wOffset: Integer; sData: string);`
: Insert the supplied text into any existing value at the specified offset.

`procedure ReplaceData(wOffset, wCount: Integer; sData: string);`
: Delete the substring starting at the nominated offset and extending for the given number of characters, and then replace it with the supplied text.

`function SubStringData(wOffset, wCount: Integer): string;`
: Extract the section of text from the specified offset, for the given number of characters.

TXmlText Class

Inheriting from the base character data node, the `TXmlText` class (as shown in Listing A-9) holds the actual content of the XML document within the DOM. When a document is first loaded, some other node type separates all text nodes from each other; contiguous sections of text in the document are placed into a single text node. This state is restored by the `Normalize` method of the element object.

Listing A-9: The TXmlText declaration.

```
TXmlText = class(TXmlCharacterData)
public
  constructor Create;
  function SplitText(wOffset: Integer): TXmlText;
  function CloneNode(bDeep: Boolean = True): TXmlNode; override;
end;
```

The properties and methods of the `TXmlText` object are described below:

`constructor Create;`

Generate text nodes through the `CreateTextNode` method on the document object, or the `CreateChildText` method on an element. Do not construct text nodes directly.

`function SplitText(wOffset: Integer): TXmlText;`

Create a new text node containing all the text from the current node past the specified offset, and return a reference to that node. The current text node has that text deleted. The new node becomes the immediately following sibling of the original node.

TXmlCDataSection Class

Textual content containing characters that would normally be treated as markup can be flagged as just straight text through CDATA sections. Within the DOM these appear as `TXmlCDataSection` objects (as shown in Listing A-10). This class inherits all the abilities of a normal text node and simply serves as an indicator of its data's origin.

Listing A-10: The TXmlCDataSection declaration.

```
TXmlCDataSection = class(TXmlText)
public
  constructor Create; override;
  function CloneNode(bDeep: Boolean = True): TXmlNode; override;
end;
```

The `TXmlCDataSection` object's methods are shown below:

`constructor Create;`

Do not construct CDATA section nodes directly. Use the `CreateCDataSection` method on the document object or the `CreateChildCDataSection` method of an element instead.

TXmlComment Class

Comments usually contain additional, non-essential information about a document. Within the DOM they appear as `TXmlComment` objects (see Listing A-11). Another text-based node type, all of its abilities are inherited.

Listing A-11: The TXmlComment declaration.

```
TXmlComment = class(TXmlCharacterData)
public
  constructor Create; override;
  function CloneNode(bDeep: Boolean = True): TXmlNode; override;
end;
```

The methods of the `TXmlComment` object are discussed below:

`constructor Create;`

Build comments with the `CreateComment` method of the document object. Do not create them directly with this constructor.

TXmlProcessingInstruction Class

Processing instructions are designed to carry information through the document for use by applications reading those documents. The `TXmlProcessingInstruction` class (shown in Listing A-12) lets you access their contents.

Listing A-12: The TXmlProcessingInstruction declaration.

```
TXmlProcessingInstruction = class(TXmlNode)
public
  constructor Create; override;
  property Data: string read FNodeValue write FNodeValue;
  property Target: string read FNodeName write FNodeName;
  function CloneNode(bDeep: Boolean = True): TXmlNode; override;
end;
```

A `TXmlProcessingInstruction` object's properties and methods are listed below:

`constructor Create;`
: Use the document object's `CreateProcessingInstruction` method to instantiate these nodes, rather than this constructor.

`property Data: string read FNodeValue write FNodeValue;`
: The remainder of the tag's content appears in this property, from the first non-white space character following the target through to the character immediately before the closing "`?>`".

`property Target: string read FNodeName write FNodeName;`
: Retrieve or set the target application for the instruction with this property.

TXmlDocumentType Class

The `TXmlDocumentType` class (see Listing A-13) encapsulates the declaration of the document type for a document. It appears as the `DocType` property of the document, although this may be `nil`. Within it are references to the entities and notations defined within the document.

Listing A-13: The TXmlDocumentType declaration.

```
TXmlDocumentType = class(TXmlNode)
public
  constructor Create; override;
  destructor Destroy; override;
  property Entities: TXmlNamedNodeMap read FEntities;
  property Name: string read FNodeName write FNodeName;
  property Notations: TXmlNamedNodeMap read FNotations;
  function CloneNode(bDeep: Boolean = True): TXmlNode; override;
end;
```

The properties and methods of the `TXmlDocumentType` object are shown below:

`constructor Create;`
: Normally, a document type node is automatically created as a document is loaded. Even if you did create one of these nodes, you cannot attach it to a document since its `DocType` property is read-only.

`property Entities: TXmlNamedNodeMap read FEntities;`
: Obtain access to a list of the external entities defined within the document through this read-only property. This does not include internal entities, which are automatically expanded, nor parameter entities. Each item in the list is a `TXmlEntity` object.

`property Name: string read FNodeName write FNodeName;`
: Retrieve the name of the document type from this property. This corresponds to the name of the single top-level element in the document.

`property Notations: TXmlNamedNodeMap read FNotations;`
: Access the notations defined in the document's DTD with this read-only property. Items in the list are all `TXmlNotation` objects.

TXmlNotation Class

Notations describe the format of unparsed entities, of attributes, and of target applications for processing instructions. They are represented by the `TXmlNotation` class (see Listing A-14) in this DOM and are retrieved from the `Notations` property of the document type node.

Listing A-14: The TXmlNotation declaration.

```
TXmlNotation = class(TXmlNode)
public
  constructor Create; override;
  property PublicId: string read FPublicId write FPublicId;
  property SystemId: string read FSystemId write FSystemId;
  function CloneNode(bDeep: Boolean = True): TXmlNode; override;
end;
```

The `TXmlNotation` object's properties and methods are listed below:

`constructor Create;`
: Use the document object's `CreateNotation` method to build new notation nodes.

`property NodeName: string read FNodeName write FNodeName;`
: The name of the notation is found in this inherited property.

`property PublicId: string read FPublicId write FPublicId;`
: Retrieve the public identifier for this notation from this property, or an empty string if none is specified.

`property SystemId: string read FSystemId write FSystemId;`
: This property provides the system identifier for the notation, or an empty string if none is supplied.

TXmlEntity Class

The `TXmlEntity` class (see Listing A-15) supplies the definitions of external entities read from the document's DTD. Access them via the `Entities` property of the document type node. No parameter or internal entities appear in this list since these are automatically expanded and their value included in the DOM. Only the definition of the entity is modeled, not the declaration itself.

Listing A-15: The TXmlEntity declaration.

```
TXmlEntity = class(TXmlNode)
public
  constructor Create; override;
  property NotationName: string read FNodeName write FNodeName;
  property PublicId: string read FPublicId write FPublicId;
  property SystemId: string read FSystemId write FSystemId;
  function CloneNode(bDeep: Boolean = True): TXmlNode; override;
end;
```

The `TXmlEntity` object's properties and methods are discussed below:

`constructor Create;`
: Entity nodes are automatically created when a document is first loaded. They cannot be added to a document type node thereafter.

`property NodeName: string read FNodeName write FNodeName;`
: This inherited property provides the name of the entity.

WARNING
Unfortunately, the CUESoft DOM returns the name of the entity's notation through the `NodeName` property, rather than the name of the entity itself. There is no way to retrieve the entity's name unless you go to the underlying parser and its `OnEntityDecl` event.

`property NotationName: string read FNodeName write FNodeName;`
: Unparsed entities return the name of their notation type through this property. For parsed entities, it returns an empty string.

NOTE
Although the `NotationName` property is mapped onto the node name field, it does return the correct value. However, the node name field should hold the name of the entity itself.

`property PublicId: string read FPublicId write FPublicId;`
: Retrieve or set the public identifier for the entity from this property. If no public identifier is specified, an empty string results.

`property SystemId: string read FSystemId write FSystemId;`
: This property reads or writes the system identifier for the entity. Again, it returns an empty string if no system identifier is available.

TXmlEntityReference Class

References to parsed entities are placed into the DOM with the `TXmlEntityReference` class (as shown in Listing A-16). The children of this reference duplicate those of the named entity node (if available).

Listing A-16: The TXmlEntityReference declaration.

```
TXmlEntityReference = class(TXmlNode)
public
  constructor Create; override;
  function CloneNode(bDeep: Boolean = True): TXmlNode; override;
end;
```

NOTE
The CUESoft parser always expands entity references within the body of the document. So, when you load in a document, no entity reference nodes appear within the DOM, only their corresponding entity's subtree. Also, the contents of entities declared in external subsets may not be available.

The properties and methods of the `TXmlEntityReference` object are described below:

`constructor Create;`
: As before, do not build these objects directly. Instead, use the `CreateEntityReference` method of the document object.

`property NodeName: string read FNodeName write FNodeName;`
: This inherited property provides the name of the referenced entity.

TXmlDocumentFragment Class

A document fragment never appears in the main DOM structure. Its purpose is to manage subtrees of nodes outside of the document itself, allowing them to be constructed or extracted before adding them back into the hierarchy. The `TXmlDocumentFragment` class (see Listing A-17) provides this functionality. It derives from the basic node without adding any new abilities.

Listing A-17: The TXmlDocumentFragment declaration.

```
TXmlDocumentFragment = class(TXmlNode)
public
  constructor Create; override;
  function CloneNode(bDeep: Boolean = True): TXmlNode; override;
end;
```

When a document fragment is added to the main DOM, it is not inserted itself. Instead, all of its child nodes are placed into the hierarchy in its place.

The methods of a `TXmlDocumentFragment` object are shown below:

`constructor Create;`
: Build document fragment nodes with the `CreateDocumentFragment` method of the document object.

TXmlDocument Class

The primary access to the DOM is via the document object, as represented by the `TXmlDocument` class (shown in Listing A-18). Another important function of this class is to create new nodes to add to the DOM. Using the factory methods provided here ensures that the nodes are compatible with the document and each other.

Listing A-18: The TXmlDocument declaration.

```
TXmlDocument = class (TXmlNode)
public
  constructor Create; override;
  destructor Destroy; override;
  property ActualCDATA: Boolean read FActualCDATA write FActualCDATA;
  property DocType: TXmlDocumentType read FDocType;
  property DocumentElement: TXmlElement read GetDocumentElement;
  property DomImplementation: TXmlDomImplementation
    read FDomImplementation;
  property FormattedOutput: Boolean read FFormattedOutput
    write FFormattedOutput;
  property IdAttribute: string read FIdAttribute write FIdAttribute;
  property IgnoreCase: Boolean read FIgnoreCase write FIgnoreCase;
  function CloneNode(bDeep: Boolean = True): TXmlNode; override;
  function CreateAttribute(sName: string): TXmlAttribute;
  function CreateComment(sData: string = ''): TXmlComment;
  function CreateCDataSection(sData: string = ''): TXmlCDataSection;
  function CreateDocumentFragment: TXmlDocumentFragment;
  function CreateElement(sTagName: string): TXmlElement;
  function CreateEntityReference(sName: string): TXmlEntityReference;
  function CreateProcessingInstruction(sTarget: string;
    sData: string = ''): TXmlProcessingInstruction;
  function CreateTextNode(sData: string = ''): TXmlText;
  function GetElementsByTagName(sName: string): TXmlNodeList;
  procedure RemoveAll;
end;
```

The `TXmlDocument` object's properties and methods are discussed below:

`constructor Create;`

Documents are created as the result of loading an XML document through the `LoadDataSource` or `LoadMemory` methods of the `TXmlObjModel` class (described later). An empty document node exists initially in the object model class that can be used to generate a new document. All access should be through the `Document` property of the object model class.

`property ActualCDATA: Boolean read FActualCDATA write FActualCDATA;`

Set this property to `True` to output CDATA sections within the DOM as plain text instead of surrounding them with the normal CDATA tags. Leave it as `False` (the default) to use the CDATA syntax.

NOTE
The `ActualCData` property is not part of the W3C DOM specification.

`property DocType: TXmlDocumentType read FDocType;`

If a DTD exists for a loaded XML document, this read-only property returns the corresponding `TXmlDocumentType` node. If no DTD is specified, and for HTML documents, it returns `nil`.

TIP

You cannot create a document type declaration for a new document in memory since this field property is read-only.

`property DocumentElement: TXmlElement read GetDocumentElement;`

Retrieve the single, top-level element in the document with this read-only property. You can also reach it via the `ChildNodes` property of the document, but this property is more convenient.

`property DomImplementation: TxmlDomImplementation read FDomImplementation;`

Access the DOM implementation for this document through this read-only property.

NOTE

Since `implementation` is a reserved word in Delphi, this W3C DOM attribute is renamed `DomImplementation` in the CUESoft package.

`property FormattedOutput: Boolean read FFormattedOutput write FFormattedOutput;`

When `True`, this property causes the XML generated by the DOM to be formatted for readability. This involves adding line feeds and indentation surrounding the elements and text. When `False` (the default), the output appears as a single string with no breaks.

NOTE

The `FormattedOutput`, `IdAttribute`, and `IgnoreCase` properties are not part of the W3C DOM specification.

`property IdAttribute: string read FIdAttribute write FIdAttribute;`

Specify a default attribute to be used as the elements' IDs when querying with XSL and XQL (XML Query Language) expressions.

`property IgnoreCase: Boolean read FIgnoreCase write FIgnoreCase;`

This property controls matching through the `GetElementsByTagName` and `SelectNodes` methods. If set to `True`, matches are case-insensitive, whereas setting it to `False` (the default) enforces matching on case.

`function CloneNode(bDeep: Boolean = True): TXmlNode; override;`

Copy the document node and, if `bDeep` is `True`, all of its children to create a new document.

`function CreateAttribute(sName: string): TXmlAttribute;`

Build a new `TXmlAttribute` node using this method, by passing in the name of the new attribute. The resulting node still needs to be added to an element to become part of the DOM. Use the element's `SetAttributeNode` method.

`function CreateComment(sData: string = ''): TXmlComment;`

Generate a new `TXmlComment` node with the supplied text through this method. Add the new node to an existing one as one of its children.

```
function CreateCDataSection(sData: string = ''):
    TXmlCDataSection;
```

This method produces a new `TXmlCDataSection` node for adding to the DOM. Specify the text content of the node when it is called. You can also use the `CreateChildCDataSection` method of an element.

`function CreateDocumentFragment: TXmlDocumentFragment;`

Obtain a new `TXmlDocumentFragment` node with this method. Document fragments are not added to the main DOM hierarchy, but are used instead to manage nodes outside of that structure.

`function CreateElement(sTagName: string): TXmlElement;`

A new `TXmlElement` node is created by this method, passing in the element's name. Add it to the DOM as a child of another node. If placed as the child of the document node itself, it also becomes the value of the `DocumentElement` property. A new child element is automatically added with the `CreateChildElement` method of an element node.

```
function CreateEntityReference(sName: string):
    TXmlEntityReference;
```

Build a new `TXmlEntityReference` node using this method. Specify the name of the entity to be inserted, and add the new node to the DOM at the required position.

```
function CreateProcessingInstruction(sTarget: string;
    sData: string = ''): XmlProcessingInstruction;
```

Generate a new `TXmlProcessingInstruction` node via this method, passing in the name of the target application and its command. Again, add the new node to the DOM structure as the child of an existing node.

`function CreateTextNode(sData: string = ''): TXmlText;`

This method produces a new `TXmlText` node, with the specified content, for adding to the DOM. Alternately, you can use the `CreateChildText` method of an element to quickly add text to an element.

```
function GetElementsByTagName(sName: string):
    TXmlNodeList;
```

Find all the elements that are descendants of the document and that have the given name. Use a name of "`*`" to retrieve all nodes. The nodes appear in the order of a pre-order traversal through the document tree. If no matching nodes are found, an empty list is returned.

`procedure RemoveAll;`

Completely empty the document of all its children with this method.

NOTE
The `RemoveAll` method is not part of the W3C DOM specification.

NOTE
Missing from the W3C DOM Level 2 specification are the `importNode` method (whose functionality can be duplicated through the `ForceOwnerDocument` method of the `TXmlNode` class), the `getElementById` method, and the namespace-aware versions of the `CreateElement`, `CreateAttribute`, and `GetElementsByTagName` methods.

TXmlDomImplementation Class

The `TXmlDomImplementation` class (see Listing A-19) provides functions outside of any document. You access its abilities through the `DOMImplementation` property of a document.

Listing A-19: The TXmlDomImplementation declaration.

```
TXmlDomImplementation = class
public
  function HasFeature(sFeature, sVersion: string): Boolean;
end;
```

The methods of the `TXmlDomImplementation` object are listed below:

`function HasFeature(sFeature, sVersion: string): Boolean;`
: Determine whether this DOM implementation supports certain features with this method. Given a particular feature name and required version, it returns `True` if that functionality is available and `False` otherwise. The version parameter may be left blank to match on any supported version. This implementation currently recognizes the features "`XML`" and "`HTML`" (case-insensitive), and version "`1.0`" of each.

TXmlObjModel Component

Since the DOM Level 1 specification, which is the level supported by this implementation, defines no way of creating a document, it is left to the designers to provide this functionality. In the CUESoft package, the `TXmlObjModel` component (shown in Listing A-20) performs this necessary task. Consequently, this entire class is an extension to the W3C DOM specification (at least at Level 1).

Listing A-20: The TXmlObjModel declaration.

```
TPreserveSpaceEvent = procedure(oOwner: TObject;
  sElementName: string; var bPreserve: Boolean) of object;
TResolveEntityEvent = function (oOwner: TObject;
  sName, sPublicId, sSystemId: string): string of object;

TXmlObjModel = class(TComponent)
protected
  function GetErrorCount: Integer;
  function GetOnPreserveSpace: TPreserveSpaceEvent;
  procedure SetOnPreserveSpace(PreserveSpace: TPreserveSpaceEvent);
```

```
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  property Document:  read FDocument;
  property ErrorCount: Integer read GetErrorCount;
  property Errors: TStringList read FErrors;
  property XmlDocument: string read GetXmlDocument;
  procedure ClearDocument;
  function GetErrorMsg(wIdx: Integer): String;
  function LoadDataSource(sSource: String): Boolean;
  function LoadMemory(cpMem: PChar): Boolean;
  function SaveToFile(sFile: string): Boolean;
published
  property FormattedOutput: Boolean read GetFormattedOutput
    write SetFormattedOutput;
  property IdAttribute: string read GetIdAttribute
    write SetIdAttribute;
  property IgnoreCase: Boolean read GetIgnoreCase write SetIgnoreCase;
  property NormalizeData: Boolean read FNormalizeData
    write FNormalizeData
  property OnPreserveSpace: TPreserveSpaceEvent read GetOnPreserveSpace
    write SetOnPreserveSpace;
  property OnResolveEntity: TResolveEntityEvent read FOnResolveEntity
    write SetOnResolveEntity;
  property Password: string read GetPassword write SetPassword;
  property RaiseErrors: Boolean read FRaiseErrors write FRaiseErrors;
  property UserName: string read GetUserName write SetUserName;
end;
```

Since this class derives from `TComponent`, it can appear on the component palette and be dropped onto a form when required. Then set its properties and load the required document in code. Alternately, you can instantiate a copy entirely in code.

A `TXmlObjModel` component's properties and methods are listed below:

`constructor Create(AOwner: TComponent); override;`
: If you drag the component from the palette, you do not have to create an instance yourself. Otherwise, use this constructor to generate an object model for your use.

`destructor Destroy; override;`
: If you create the object model yourself, remember to free it up when you are finished. Objects are automatically destroyed when you drop the component onto the form from the component palette.

`property Document:  read FDocument;`
: This read-only property provides access to the document in memory and all its abilities. You should only use the document through this mechanism.

`property ErrorCount: Integer read GetErrorCount;`
: Find the number of errors that occurred during a parse through this read-only property.

`property Errors: TStringList read FErrors;`
: Retrieve a list of the errors from a parse with this read-only property.

`property FormattedOutput: Boolean read GetFormattedOutput write SetFormattedOutput;`
: Duplicating the same property on the document object, this property controls the formatting of any XML generated from the DOM. When

`True`, indentation and line breaks are added to make the text more legible. When `False` (the default), the text is just one long string.

`property IdAttribute: string read GetIdAttribute write SetIdAttribute;`
Also replicating a property on the document object, this one determines what attribute is treated as the ID attribute for searches within the hierarchy.

`property IgnoreCase: Boolean read GetIgnoreCase write SetIgnoreCase;`
Another property copied from the document object. When `True`, this property causes case to be ignored in matches using XSL and XQL queries. When `False` (the default), case is used in determining a match.

`property NormalizeData: Boolean read FNormalizeData write FNormalizeData`
Setting this property to `True` results in extra white space being stripped from character data in the parse process. Otherwise, all text data is sent through as is (the default).

`property OnPreserveSpace: TPreserveSpaceEvent read GetOnPreserveSpace write SetOnPreserveSpace;`
This event triggers once for each element encountered in the parse process. It supplies the name of that element and the current space preservation setting, based on the `NormalizeData` property and any `xml:space` attributes. An attached event handler may alter the preservation flag.

`property OnResolveEntity: TResolveEntityEvent read FOnResolveEntity write SetOnResolveEntity;`
External references can be resolved through this event. It passes across the name of the entity, along with its public and system identifiers. Using these you can adjust the actual path to the resource and send it back to the parser as the result of the handler function.

`property Password: string read GetPassword write SetPassword;`
When reading an XML file from an FTP site, this property establishes the password used to gain access to that site.

`property RaiseErrors: Boolean read FRaiseErrors write FRaiseErrors;`
Set this property to `True` to have the parser pass `TXmlParserError` exceptions through to the application. Otherwise, they are trapped by this component (the default).

`property UserName: string read GetUserName write SetUserName;`
Complementing the `Password` property, this one sets the user ID for retrieving documents from FTP sites. If not set, "`anonymous`" is used.

`property XmlDocument: string read GetXmlDocument;`
Generate an XML document from the DOM in memory with this read-only property.

`procedure ClearDocument;`
: Delete the entire DOM with this method. A new document can then be constructed.

`function GetErrorMsg(wIdx: Integer): String;`
: Retrieve individual error messages from the parse process with this method. Duplicating the abilities of the `Error` property, the index ranges from zero to `ErrorCount - 1`.

`function LoadDataSource(sSource: String): Boolean;`
: The heart of the process, this method invokes the parser on the specified document. Files are identified either as local filenames, or as HTTP or FTP URLs. A return value of `True` results if the document is successfully loaded and `False` is returned if problems are encountered. In the latter case, check the `Errors` property for the reason(s).

`function LoadMemory(cpMem: PChar): Boolean;`
: Similar to the previous method, this one parses a document held in memory at the supplied location. Again, it returns `True` if successful and `False` if not.

`function SaveToFile(sFile: string): Boolean;`
: Having created your DOM in memory, use this method to write it to a file. The document type declaration is not included in the document, although the remainder is well-formed XML. You can specify either a local filename or an FTP site to write to. The function returns `True` if it succeeded and `False` if a problem arose.

TXmlParser Component

The CUESoft package relies on a built-in parser to process XML documents into the DOM structure. CUESoft's parser is non-validating, although it does check for well-formed documents. You can access the parser yourself and use it to do your own processing by registering event handlers with it. The `TXmlParser` component (see Listing A-21) can also dwell on the component palette, making it easy to incorporate into your project. This class appears in the `XmlParser` unit.

Listing A-21: The TXmlParser declaration.

```
TAttributeEvent = procedure (oOwner: TObject;
  sName, sValue: string; bSpecified: Boolean) of object;
TDocTypeDeclEvent = procedure (oOwner: TObject;
  sDecl, sId0, sId1: string) of object;
TEntityDeclEvent = procedure (oOwner: TObject;
  sEntityName, sPublicId, sSystemId, sNotationName: string) of object;
TNonXMLEntityEvent = procedure (oOwner: TObject;
  sEntityName, sPublicId, sSystemId, sNotationName: string) of object;
TNotationDeclEvent = procedure (oOwner: TObject;
  sNotationName, sPublicId, sSystemId: string) of object;
TPreserveSpaceEvent = procedure (oOwner: TObject;
  sElementName: string; var bPreserve: Boolean) of object;
TProcessInstrEvent = procedure (oOwner: TObject;
  sName, sValue: string) of object;
TResolveEntityEvent = function (oOwner: TObject;
  sName, sPublicId, sSystemId: string): string of object;
TValueEvent = procedure (oOwner: TObject; sValue: string) of object;
```

```
TXmlParser = class(TComponent)
protected
  property OnIgnorableWhitespace: TValueEvent
    read FOnIgnorableWhitespace write FOnIgnorableWhitespace;
public
  constructor Create(oOwner: TComponent);
  destructor Destroy; override;
  property ErrorCount: Integer read GetErrorCount;
  property Errors: TStringList read FErrors;
  function GetErrorMsg(wIdx: Integer): string;
  function ParseDataSource(sSource: string): Boolean;
  function ParseMemory(cpMem: PChar): Boolean;
published
  property NormalizeData: Boolean read FNormalizeData
    write FNormalizeData;
  property OnAttribute: TAttributeEvent read FOnAttribute
    write FOnAttribute;
  property OnCDATASection: TValueEvent read FOnCDATASection
    write FOnCDATASection;
  property OnCharData: TValueEvent read FOnCharData write FOnCharData;
  property OnComment: TValueEvent read FOnComment write FOnComment;
  property OnDocTypeDecl: TDocTypeDeclEvent read FOnDocTypeDecl
    write FOnDocTypeDecl;
  property OnEndDocument: TNotifyEvent read FOnEndDocument
    write FOnEndDocument;
  property OnEndElement: TValueEvent read FOnEndElement
    write FOnEndElement;
  property OnEntityDecl: TEntityDeclEvent read FOnEntityDecl
    write FOnEntityDecl;
  property OnNonXMLEntity: TNonXMLEntityEvent read FOnNonXMLEntity
    write FOnNonXMLEntity;
  property OnNotationDecl: TNotationDeclEvent read FOnNotationDecl
    write FOnNotationDecl;
  property OnPreserveSpace: TPreserveSpaceEvent read FOnPreserveSpace
    write FOnPreserveSpace;
  property OnProcessingInstruction: TProcessInstrEvent
    read FOnProcessingInstruction write FOnProcessingInstruction;
  property OnResolveEntity: TResolveEntityEvent read FOnResolveEntity
    write FOnResolveEntity;
  property OnStartDocument: TNotifyEvent read FOnStartDocument
    write FOnStartDocument;
  property OnStartElement: TValueEvent read FOnStartElement
    write FOnStartElement;
  property Password: string read FPassword write FPassword;
  property RaiseErrors: Boolean read FRaiseErrors write FRaiseErrors;
  property UserName: string read FUserName write FUserName;
end;
```

TIP

To see CUESoft's parser in action, look at SAX for Pascal described in Chapter 14. Included is a SAX wrapper using the CUESoft offering.

The properties and methods of a `TXmlParser` component are shown below (most of which correspond directly with those in the `TXmlObjModel` class):

`constructor Create(oOwner: TComponent);`

For easiest use, drag-and-drop one of these components from the palette, then set its properties at design time. Otherwise, use this constructor to build a parser in code for your use.

`destructor Destroy; override;`

If you create the parser yourself, do not forget to release its resources when finished.

`property ErrorCount: Integer read GetErrorCount;`

Find the number of errors from the parse process with this read-only property.

`property Errors: TStringList read FErrors;`

Retrieve all the reasons for errors during the parse through this read-only property.

`property NormalizeData: Boolean read FNormalizeData write FNormalizeData;`

Strip out extra white space from the document when this property is set to `True`. Otherwise, all text is passed through unchanged to the `OnCharData` event (the default). CDATA sections are not affected by this property.

`property OnAttribute: TAttributeEvent read FOnAttribute write FOnAttribute;`

Respond to attributes encountered in the document through this event, which fires before the `OnStartElement` event for their containing element. The attribute name and value, and a flag indicating the origin of that value, are passed to the event handler.

`property OnCDATASection: TValueEvent read FOnCDATASection write FOnCDATASection;`

CDATA sections from the document trigger this event, which receives the entire contents of that section.

`property OnCharData: TValueEvent read FOnCharData write FOnCharData;`

Normal textual content causes this event to fire. Each contiguous section of text appears in one event through the supplied parameter.

`property OnComment: TValueEvent read FOnComment write FOnComment;`

The entire content of a comment from the document is available within a handler attached to this event.

`property OnDocTypeDecl: TDocTypeDeclEvent read FOnDocTypeDecl write FOnDocTypeDecl;`

Encountering the document type declaration in the document causes this event to trigger. The name of the document type, and its public and system identifiers are passed across to the event handler. Note that unparsed entities and notations declared in the DTD are notified in events that occur before this one.

`property OnEndDocument: TNotifyEvent read FOnEndDocument write FOnEndDocument;`

Once the entire document has been processed, this event fires. Use this event to complete your processing and to release any resources no longer required.

`property OnEndElement: TValueEvent read FOnEndElement write FOnEndElement;`

Receive notification of the end tag for an element through this event. The name of the element is supplied. All the content of that element appears as events between this one and its corresponding `OnStartElement`.

```
property OnEntityDecl: TEntityDeclEvent read
    FOnEntityDecl write FOnEntityDecl;
```
Unparsed entity declarations within the document type declaration trigger this event. Save the entity's name, public and system identifiers, and notation name from the parameters passed in. These events occur before the `OnDocTypeDecl` event to which they apply.

```
property OnIgnorableWhitespace: TValueEvent read
    FOnIgnorableWhitespace write FOnIgnorableWhitespace;
```
White space outside of normal text content is notified through this event. However, the fact that it can be ignored is only available if the document is validated against a DTD. Hence, this event is not currently available and appears as a protected property on the parser.

```
property OnNonXMLEntity: TNonXMLEntityEvent read
    FOnNonXMLEntity write FOnNonXMLEntity;
```
This event is triggered when a non-XML entity is encountered in the document. The callback lets you respond to this occurrence and perhaps provide some level of support for the entity within your application.

```
property OnNotationDecl: TNotationDeclEvent read
    FOnNotationDecl write FOnNotationDecl;
```
The notations used by entities and processing instructions trigger this event. Save the name, public, and system identifiers for later use. These events arrive before the event for the document type declaration to which they belong.

```
property OnPreserveSpace: TPreserveSpaceEvent read
    FOnPreserveSpace write FOnPreserveSpace;
```
Fired for each element encountered, this event lets you override the preservation flag setting. Check the element name and current setting, and update the flag if required.

```
property OnProcessingInstruction: TProcessInstrEvent read
    FOnProcessingInstruction write
    FOnProcessingInstruction;
```
Each processing instruction found in the document triggers this event. The target application and the actual command are supplied as parameters.

```
property OnResolveEntity: TResolveEntityEvent read
    FOnResolveEntity write FOnResolveEntity;
```
You can perform resolution for external entities through this event. Given the entity's name and its public and system identifiers, you should return the name of the actual resource to reference.

```
property OnStartDocument: TNotifyEvent read
    FOnStartDocument write FOnStartDocument;
```
Fired once at the start of the parse process, use this event to initialize your application in preparation for a new document.

```
property OnStartElement: TValueEvent read FOnStartElement
    write FOnStartElement;
```
The opening tag for each element triggers this event, supplying the name of the element encountered. Recall that the attributes for that element

have already appeared in `OnAttribute` events prior to their containing element.

`property Password: string read FPassword write FPassword;`
Set this property to supply a password when accessing documents at FTP sites.

`property RaiseErrors: Boolean read FRaiseErrors write FRaiseErrors;`
When set to `True`, this property causes parse errors (`TXmlParserError` exceptions) to be sent directly to the application. Otherwise, they are trapped internally and end the parse process in error (the default).

`property UserName: string read FUserName write FUserName;`
For accessing FTP sites, specify a user ID to give with this property. If not set, it defaults to "`anonymous`".

`function GetErrorMsg(wIdx: Integer): string;`
Retrieve individual error messages through this function. The index ranges from zero to `ErrorCount - 1`.

`function ParseDataSource(sSource: string): Boolean;`
Retrieve the document specified and parse its contents, invoking the appropriate events as necessary. The source specification may be either a local filename, or an HTTP or FTP URL. A `True` results if the parse succeeds and a `False` if it fails. Check the `Errors` property in the latter case for the reason(s) it failed.

`function ParseMemory(cpMem: PChar): Boolean;`
Similarly, this method parses a document in memory, returning `True` on success and `False` on failure.

Loading the CUESoft DOM

For comparison purposes, you can build the same XML viewer from Chapter 9, but using the CUESoft DOM. The `TXmlObjModel` class is the main entry point into the package. Since this is a Delphi component you can drag it from the component palette, or create it in code, as shown in Listing A-22. Do not forget to free it after use.

Listing A-22: Loading the document.

```
{ Load an XML document }
procedure TfrmXMLViewer.LoadDoc(Filename: string);
var
  XMLDOM: TXmlObjModel;
begin
  pgcDetails.ActivePage := tshDocument;
  { Initialize document-wide details for display }
  InitDocumentDetails;
  { Load the source document }
  memSource.Lines.LoadFromFile(Filename);
  dlgOpen.Filename := Filename;
  { Instantiate the DOM }
  XMLDOM := TXmlObjModel.Create(nil);
  trvXML.Items.BeginUpdate;
```

```
  try
    { Suppress white space? }
    XMLDOM.NormalizeData := mniSuppressWhitespace.Checked;
    { Parse the document }
    if not XMLDOM.LoadDataSource(Filename) then
      raise Exception.Create(
        Format(NoLoadError, [XMLDOM.Errors.Text]));

    edtSystemId.Text := Filename;
    { Add the structure to the tree view }
    AddElementToTree(XMLDOM.Document, nil);
    trvXML.Items[0].Expand(False);
  finally
    trvXML.Items.EndUpdate;
    { Release the DOM }
    XMLDOM.Free;
  end;
end;
```

An item on the menu in the viewer lets you suppress text nodes that contain only white space. This value is transferred directly to the `NormalizeData` property of the DOM. Calling the `LoadDataSource` method on the object model class then loads and parses the specified document, returning a `False` value if it fails. In that case you can raise an exception with the list of problems from the `Errors` property. Otherwise, pass the newly created document, accessed through the `Document` property, to the routine that builds up the tree view on the page.

Like the previous example, the construction of the tree view relies on recursive calls to the `AddElementToTree` routine (see Listing A-23). Initially the nodes can be treated in a generic manner to extract a meaningful display value for them. Thereafter, the node type determines what additional information is required and how to retrieve it. Each type is cast to its appropriate subclass before accessing its attributes.

Listing A-23: Reading the nodes.

```
{ Add a TXMLElement to the tree view }
function AddElement(Parent: TTreeNode; Name: string;
  Element: TXMLElement): TTreeNode;
begin
  FList.Add(Element);
  Result := trvXML.Items.AddChildObject(Parent, Name, Element);
  with Result do
  begin
    ImageIndex    := Ord(Element.ElementType);
    SelectedIndex := ImageIndex;
  end;
end;

{ Add current element to the treeview and
  then recurse through children }
procedure AddElementToTree(Node: TXmlNode; TreeParent: TTreeNode);
var
  Index: Integer;
  DisplayName: string;
  NewNode: TTreeNode;
  Attribs: TStringList;
```

```
  { Extract an attribute value from a string }
  function GetPseudoAttr(const Name, Data: string): string;
  var
    PosStart, PosEnd: Integer;
  begin
    Result   := '';
    PosStart := Pos(Name, Data);
    if PosStart = 0 then
      Exit;

    PosStart := PosStart + Length(Name) + 1;
    PosEnd   := Pos(Data[PosStart],
      Copy(Data, PosStart + 1, Length(Data)));
    if PosEnd = 0 then
      Result := ''
    else
      Result := Copy(Data, PosStart + 1, PosEnd - 1);
  end;

begin
  { Generate name for display in the tree }
  if Node.NodeType in
    [TEXT_NODE, COMMENT_NODE, CDATA_SECTION_NODE] then
  begin
    if Length(Node.NodeValue) > 20 then
      DisplayName := Copy(Node.NodeValue, 1, 17) + '...'
    else
      DisplayName := Node.NodeValue;
  end
  else
    DisplayName := Node.NodeName;
  { Create storage for later display of node values }
  case Node.NodeType of
    ELEMENT_NODE:
      with Node as XmlObjModel.TXmlElement do
      begin
        Attribs := TStringList.Create;
        try
          if HasAttributes then
            for Index := 0 to Attributes.Length - 1 do
              with Attributes.Item(Index) do
                Attribs.Values[NodeName] := NodeValue;
          NewNode := AddElement(TreeParent, DisplayName,
            TXMLElement.Create(xtElement, NodeName,
            Namespace, BaseName, '', Attribs));
        finally
          Attribs.Free;
        end;
      end;
    TEXT_NODE:
      with Node as TXmlText do
        NewNode := AddElement(TreeParent, DisplayName,
          TXMLElement.Create(xtText, '', '', '', Data, nil));
    CDATA_SECTION_NODE:
      with Node as TXmlCDATASection do
        NewNode := AddElement(TreeParent, DisplayName,
          TXMLElement.Create(xtCData, '', '', '', Data, nil));
    ENTITY_REFERENCE_NODE:
      NewNode := AddElement(TreeParent, DisplayName,
        TXMLElement.Create(xtEntityRef, Node.NodeName,
        '', '', '', nil));
    PROCESSING_INSTRUCTION_NODE:
      with Node as TXmlProcessingInstruction do
      begin
        NewNode := AddElement(TreeParent, DisplayName,
          TXMLElement.Create(xtInstruction, Target,
          '', '', Data, nil));
```

```
        if UpperCase(Target) = XMLValue then
        begin
          { Special handling for the XML declaration }
          edtVersion.Text       := GetPseudoAttr(VersionAttr, Data);
          edtEncoding.Text      := GetPseudoAttr(EncodingAttr, Data);
          cbxStandAlone.Checked := (UpperCase(GetPseudoAttr(
            StandAloneAttr, Data)) = YesValue);
        end;
      end;
    COMMENT_NODE:
      with Node as TXmlComment do
        NewNode := AddElement(TreeParent, DisplayName,
         TXMLElement.Create(xtComment, '', '', '', Data, nil));
    DOCUMENT_NODE:
      with Node as TXmlDocument do
      begin
        NewNode := AddElement(TreeParent, XMLDocDesc,
          TXMLElement.Create(xtDocument, XMLDocDesc, '', '', '', nil));
        AddElementToTree(DocType, NewNode);
      end;
    DOCUMENT TYPE NODE:
      with Node as TXmlDocumentType do
      begin
        edtDocType.Text := Name;
        NewNode := AddElement(TreeParent, DTDDesc,
          TXMLElement.Create(xtEntityRef, DTDDesc, '', '', '', nil));
        for Index := 0 to Entities.Length - 1 do
          AddElementToTree(Entities.Item(Index), NewNode);
        for Index := 0 to Notations.Length - 1 do
          AddElementToTree(Notations.Item(Index), NewNode);
      end;
    ENTITY_NODE:
      with (Node as TXmlEntity), stgEntities do
        if NotationName <> '' then
        begin
          { Unparsed entity }
          if Cells[0, RowCount - 1] <> '' then
            RowCount := RowCount + 1;
          Cells[0, RowCount - 1] := NodeName;
          Cells[1, RowCount - 1] := PublicId;
          Cells[2, RowCount - 1] := SystemId;
          Cells[3, RowCount - 1] := NotationName;
        end
        else
          { Parsed entity }
          NewNode := AddElement(TreeParent, DisplayName,
            TXMLElement.Create(xtEntityRef, NodeName,
            '', '', '', nil));
    NOTATION_NODE:
      with (Node as TXmlNotation), stgNotations do
      begin
        if Cells[0, RowCount - 1] <> '' then
          RowCount := RowCount + 1;
        Cells[0, RowCount - 1] := NodeName;
        Cells[1, RowCount - 1] := PublicId;
        Cells[2, RowCount - 1] := SystemId;
      end;
  end;
  { And recurse through any children }
  if Node.HasChildNodes then
    for Index := 0 to Node.ChildNodes.Length - 1 do
      AddElementToTree(Node.ChildNodes.Item(Index), NewNode);
end;
```

Elements have their attributes converted into a string list before saving all the details in a `TXMLElement` object. Note that this is a local class defined in the viewer unit, and does not refer to the `TXmlElement` class of the CUESoft

package. The local definition replaces the external one, so all references to this class use the internal one. To access the original class, you must prefix it with the name of its unit, `XmlObjModel.TXmlElement`. The results of processing an element are seen in Figure A-2.

Figure A-2: Displaying an element.

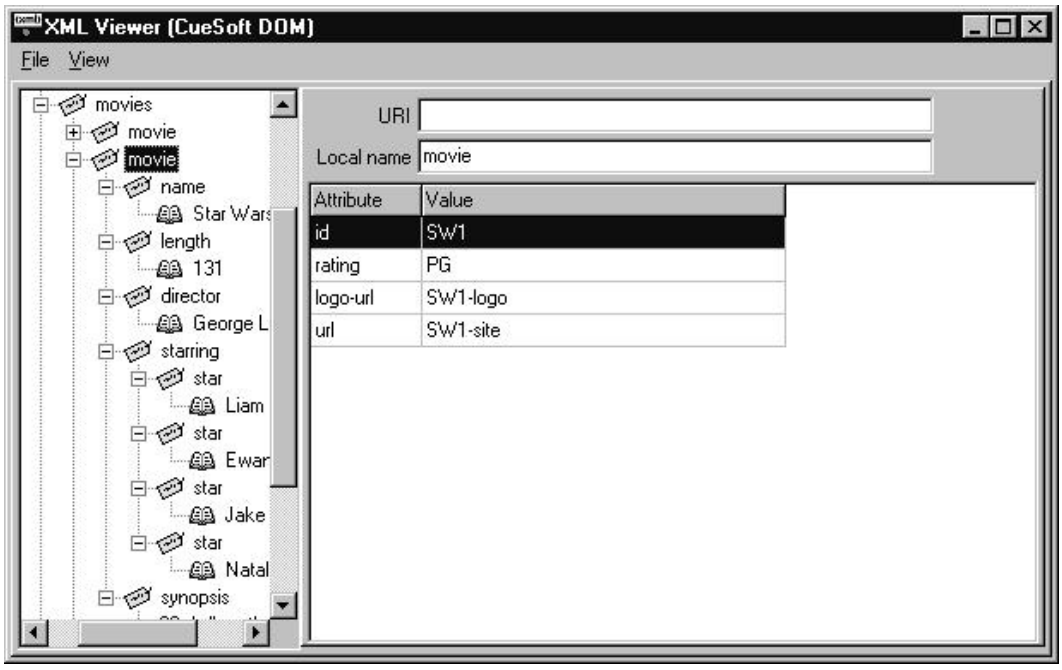
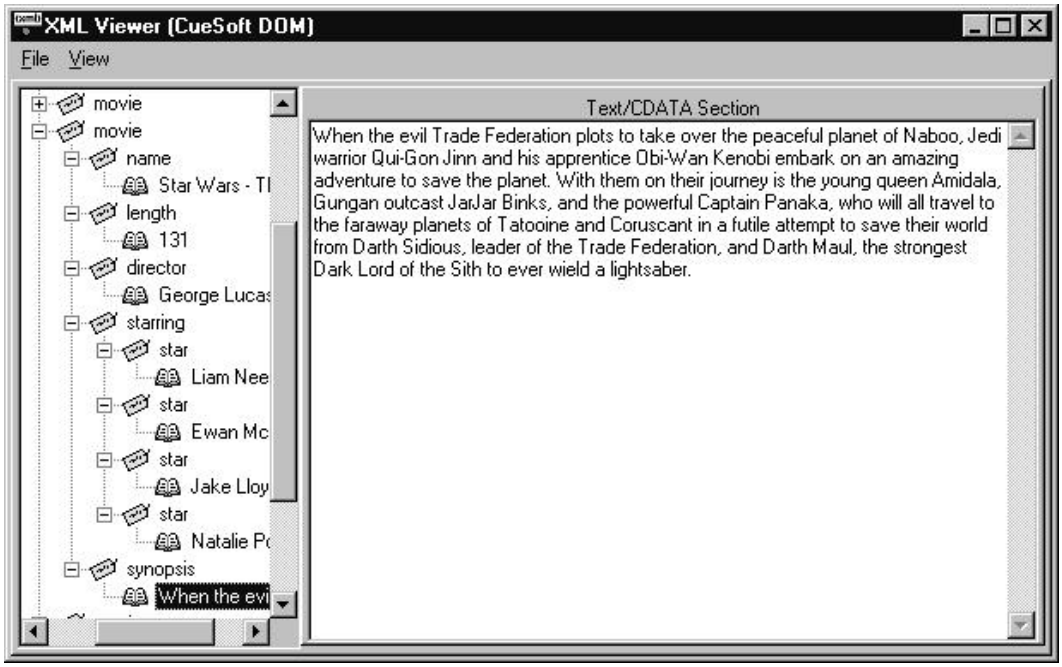


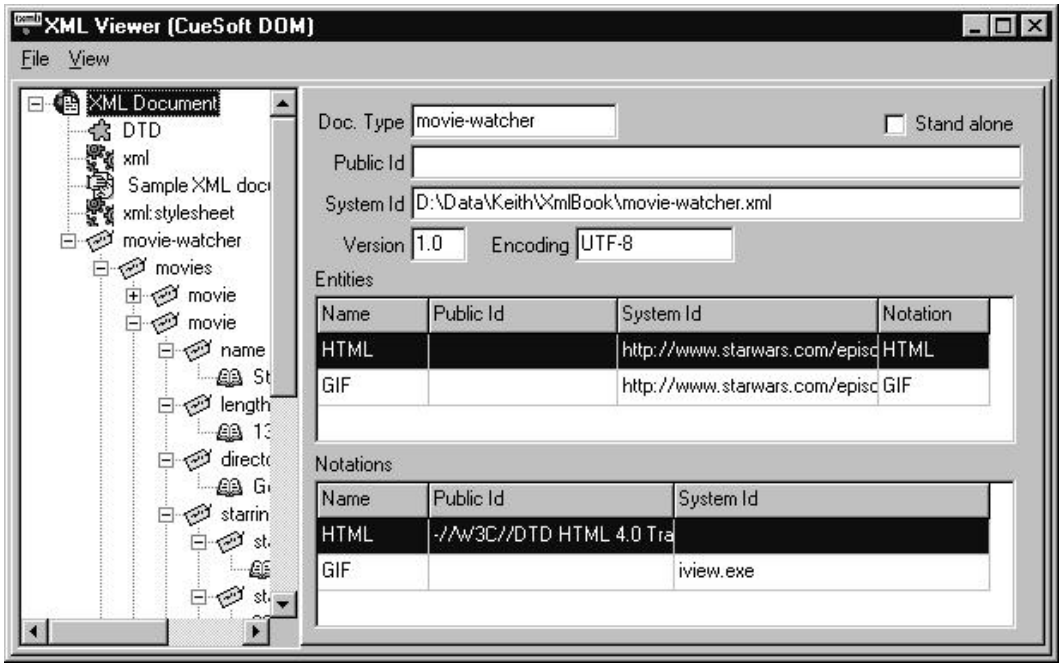
Figure A-3: Text content within the viewer.



Text type nodes, including CDATA sections and comments, simply copy their content into the corresponding field in the `TXmlElement` for later use. An example of these is shown in Figure A-3. Processing instructions follow a similar path, placing their command content in the data field of the storage object. A special case exists for the XML declaration whereby its pseudo-properties are extracted and transferred to particular fields on the document page of the viewer.

The rest of the information for the document page comes from the document type node, and its entity and notation properties. The latter are not actually children of the document type node in the CUESoft DOM, so you must step through them within their lists and manually invoke the next level of node processing. Thereafter, the notation and unparsed entity nodes get added to the grids on the document page. The document type node also supplies the name of the top-level element for the document. Figure A-4 shows all this information on the document page in the viewer.

Figure A-4: The document page in the viewer.



Entity references do not appear within the CUESoft DOM since it expands all such references during the parse process. Only the results of the expansion are passed along. Similarly, parsed entities do not appear within the document type node's list of entities.

Finally, each child of the current node is processed in turn through a recursive call. The newly created `TTreeNode` is passed along to provide the context for any additions to the view.

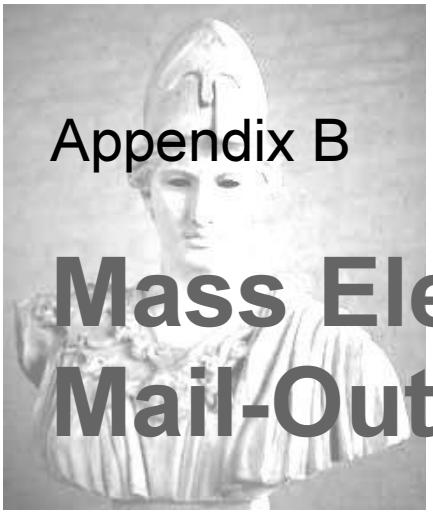
Summary

The CUESoft DOM implements the W3C DOM Level 1 specification very closely, and includes a few elements of the Level 2 specification. However, it

does not provide full support for namespaces, which limits its usefulness in some situations.

Having the DOM available as Delphi components and classes makes it very simple to use within your application. The initial steps can be performed without any coding by dragging the `TXmlObjModel` component from the palette onto your form, then setting its properties in the inspector. Once compiled, the parser and DOM become part of your executable, making it easier to distribute.

The parser in this package can be used on its own without building the associated DOM. Include the `XmlParser` unit in your project and create an instance of the `TXmlParser` component, or drag one from the component palette and drop it on your form. By registering event handlers with the parser, you can respond to the items within the XML document as they are encountered. See the SAX for Pascal discussion in Chapter 14 for an example of its use.



Appendix B

Mass Electronic Mail-Outs

The purpose of the mass-mailer program described in this appendix is to perform mass electronic mail-outs based on a document template. Fields within the template are merged with recipient data (extracted from a datasource) to customize the mailings. An additional objective is to make the application as modular as possible, allowing you to easily maintain different parts independently.

To protect the program against future technology changes, it relies on several standards, each of which is encapsulated in a class:

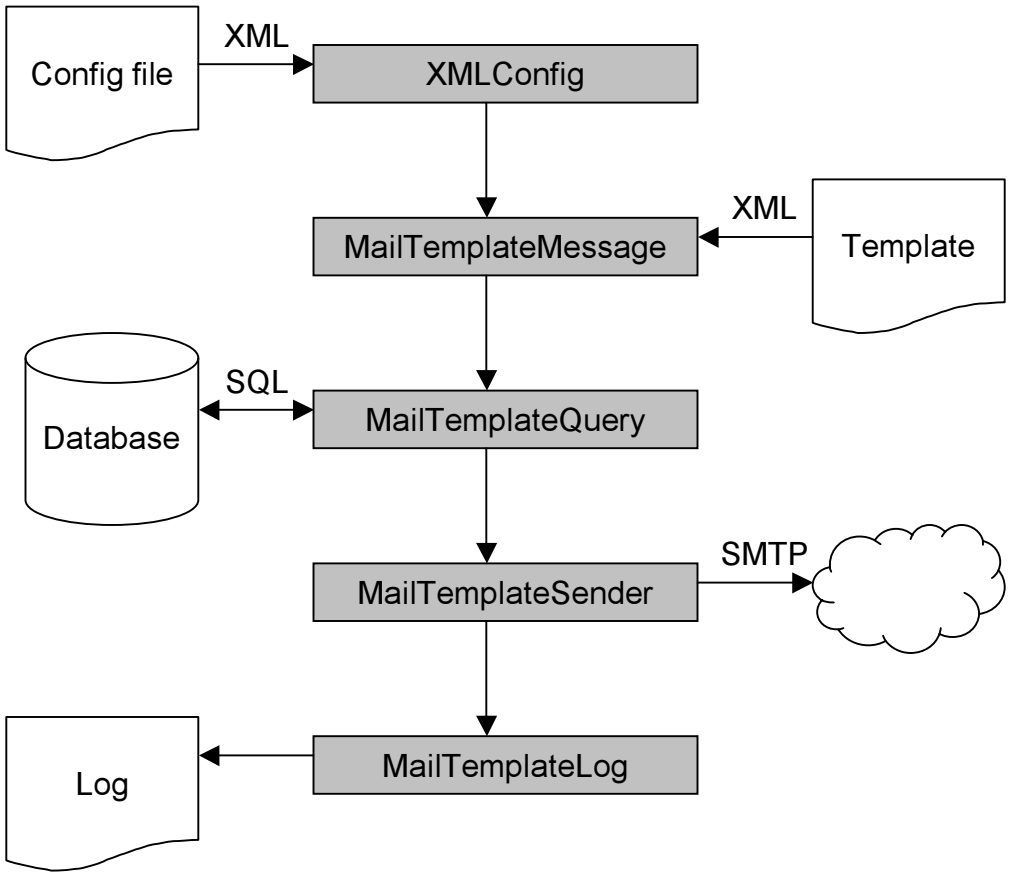
- XML is used for the configuration properties, as well as for the template containing the message to be sent out. The text-based format of XML allows these files to be easily maintained through normal text editors, in addition to specialized XML editors. Changes to the message template can be made without affecting the rest of the program, nor requiring a recompile.
- SQL is used to retrieve data from a datasource. Information from here determines where the e-mails are sent and the field values that can be included in the message. Access to the datasource uses a BDE alias. Together, this approach means that your data can reside in almost any format, since Delphi provides access to several databases natively and many others through ODBC and OLE DB. Using SQL gives you a common method for retrieving the data, freeing you from worrying about how it is actually held. If the data needs to be moved (to a different server and/or to a more powerful database), then all you need to do is update the BDE alias and the program still runs as before.
- The Simple Mail Transfer Protocol (SMTP) is used to communicate with the e-mail server for the dispatch of the messages.

The program works as follows (see Figure B-1):

1. You read in an XML configuration document that contains the various settings to use in the current run. This file details the SMTP attributes, the database connection, and the name of the file containing the message template to be used.
2. The actual query and the message template are read in from another XML document (as identified in the configuration parameters above).

3. You retrieve your list of recipients from the database, which you access using the BDE alias from the configuration file and the SQL query from the template.
4. For each record retrieved, merge the requested fields from the database into the message template.
5. The merged messages are sent out using SMTP, allowing you to talk to any number of mail servers. For testing purposes the messages are redirected to a log file where they can be reviewed.

Figure B-1: Data flow through the program.



Loading the Configuration Properties

The loading of the program properties is accomplished in a generic manner, retrieving them from an XML document. The hierarchy within the XML file determines the names of the properties, compiled from the full element path to the value (separated by periods (.) and ignoring the top-level element), with the property value coming from the actual text content. For example, the XML configuration file in Listing B-1 results in the accumulation of these properties and values:

```
smtp.host=mail.thingies.com
smtp.user=keith
smtp.from=kbwood@thingies.com
database.alias=mailtemp
settings.pauseTime=2000
settings.template=MailMessage.xml
settings.testing=Y
```

Listing B-1: Sample XML configuration file.

```
<?xml version="1.0"?>
<mailTemplate>
  <smtp>
    <host>mail.thingies.com</host>
    <port/>
    <user>keith</user>
    <from>kbwood@thingies.com</from>
  </smtp>
  <database>
    <alias>mailtemp</alias>
    <user/>
    <password/>
  </database>
  <settings>
    <pauseTime>2000</pauseTime>
    <template>MailMessage.xml</template>
    <testing>Y</testing>
  </settings>
</mailTemplate>
```

This layout takes advantage of the structure inherent in XML to group related property values. It also builds on the ability of XML documents to be processed when they are merely well-formed, without requiring conformance to a DTD. In this way, the technique is general enough that it could be reused elsewhere.

NOTE
Recall that a well-formed document simply follows the conventions of XML – only one top-level element, all elements have matching end tags in order, etc. If the document claims to follow a DTD and, in fact, does, then it is deemed to be valid. The DTD prescribes which elements and attributes may appear where within the document. In many cases, well-formed documents are sufficient for useful work.

To aid in its reuse, the functionality of the property loading is placed into its own unit, `XMLConfig`. Property names and values are placed into a string list using its `Values` property, providing a simple way to retrieve them by name later on. The `LoadPropertiesFromXML` procedure (see Listing B-2) takes the name of the file to load and a reference to a string list and fills the latter with the properties found. Just add this unit to another project to reuse its abilities.

Listing B-2: Configuration properties from XML.

```
{ Open the configuration file and then load the properties }
procedure LoadPropertiesFromXML(Filename: string; Props: TStrings);
var
  XMLDoc: IXMLDOMDocument;
  Index: Integer;

  { Recursively read XML document until text leaves are reached.
    Property name is the accumulated tags to this point
    (separated by periods).
```

```
    Property value is the actual text.
    Add these into a string list using its Values property. }
  procedure LoadSubProperties(Element: IXMLDOMNode;
    PropPrefix: string);
  var
    Index: Integer;
  begin
    with Element do
      if (NodeType = NODE_TEXT) or (NodeType = NODE_CDATA_SECTION) then
        Props.Values[Copy(PropPrefix, 2, Length(PropPrefix) -1)] :=
          NodeValue
      else
        for Index := 0 to ChildNodes.Length -1 do
          LoadSubProperties(ChildNodes[Index],
            PropPrefix + '.' + NodeName);
  end;

begin
  XMLDoc := CoDOMDocument.Create;
  Props.Clear;
  try
    XMLDoc.Load(Filename);
    { Read through each second level element and process them }
    with XMLDoc.DocumentElement do
      for Index := 0 to ChildNodes.Length -1 do
        LoadSubProperties(ChildNodes[Index], '');
  finally
    XMLDoc := nil;
  end;
end;
```

The routine itself creates an instance of the Microsoft XML parser, `IXMLDOMDocument`, and asks it to parse the specified document. It then steps through all the child nodes of the main document element and calls the internal procedure, `LoadSubProperties`, on each. This latter routine tests for text-type nodes and creates an entry in the properties list when one is found. The name for the property is built up from the names of the elements leading to the text node, which is achieved through recursive calls to this same routine for embedded child nodes.

TIP

Freeing up the DOM in the `finally` clause is not strictly necessary. Delphi automatically decrements the reference count for an interface and frees it when zero when its variable goes out of scope.

Mail Message Template

Once the configuration properties have been loaded, you can extract the name of the message template file and load that, too. This file is another XML document that holds the text of the message to be sent, along with the query used to retrieve the recipients and their details. A sample template is shown in Listing B-3.

Listing B-3: XML mail-out template.

```
<?xml version="1.0"?>
<template>
  <query emailfield="EmailAddress">select * from customer</query>
  <subject>Come visit your new Web site</subject>
  <message>Dear <field>FirstName</field>,
```

```
Our new Web site is up and running at http://www.thingies.com.
As <field>Position</field> of <field>Company</field> we think you
would find something of interest here.
Yours sincerely,
Keith Wood
  </message>
</template>
```

First, the SQL query is specified. The only database field that needs to be specifically identified for the application's use is the recipient's e-mail address, which is done through the `emailfield` attribute of the `query` element. Otherwise, the query can be as complex or as simple as necessary and can retrieve whatever fields it requires for use in the message itself. Recipients can easily be filtered out of the database as a whole for targeted mailings – just include an appropriate `where` clause in the query.

The subject of the e-mail appears in the `subject` element, with the body of the message being specified in the `message` element. Within the latter you can insert values from fields in the database by positioning `field` elements, containing the name of the field to display, at the appropriate points in the text. Any formatting of the field values can be done within the SQL query, so no additional processing should be necessary here.

Using XML means that the templates can be maintained by anyone with a text or XML editor. A minimal knowledge of SQL is required. To hide a complex query, a view could be constructed presenting the necessary values in a simple-to-use format. Having the database query in the document along with the message text ensures that the two remain synchronized. As a security measure, the logon details for the database are not included in the XML message template.

The XML document is loaded and accessed through the `TMail-TemplateMessage` class, which resides in its own unit, `MailTemplate-Message`. In its constructor the class creates an instance of the Microsoft XML parser, and requests this to load the specified XML file. Thereafter, you have access to the complete contents of the document. Setting the `preserveWhiteSpace` property to `True` ensures that your message appears in the e-mail the same way it does in the template. If this property was left at `False`, white space next to the field elements is lost.

```
{ Initialization }
constructor TMailTemplateMessage.Create(Filename: string);
begin
  inherited Create;
  FXMLDoc := CoDOMDocument.Create;
  FXMLDoc.preserveWhiteSpace := True;
  FXMLDoc.load(Filename);
end;
```

Two methods provide easy access to the elements and attributes within the document (see Listing B-4). `NodeValue` returns the text contained within the specified element, or an empty string if the element cannot be found. The routine assumes that there is only one of each type of element in the document, and that it contains only a single text node. `AttributeValue` returns the value of the named attribute of a given element. Again, a single occurrence of the element is assumed, and an empty string is returned if the attribute or node does not exist.

Listing B-4: Retrieving element and attribute values.

```
{ Return the value of the named attribute -
  assumes only one such node }
function TMailTemplateMessage.AttributeValue(
  NodeName, AttrName: string): string;
var
  Elements: IXMLDOMNodeList;
begin
  Elements := FXMLDoc.getElementsByTagName(NodeName);
  if Elements.length = 0 then
    Result := ''
  else
    Result :=
      Elements.item[0].attributes.getNamedItem(AttrName).text;
end;

{ Return the value of the named node -
  assumes only one such node and no children }
function TMailTemplateMessage.NodeValue(NodeName: string):
  string;
var
  Elements: IXMLDOMNodeList;
begin
  Elements := FXMLDoc.getElementsByTagName(NodeName);
  if Elements.length = 0 then
    Result := ''
  else
    Result := Elements.item[0].text;
end;
```

In both routines you use the `getElementsByTagName` method of the document to locate and return the required node. Actually, this routine returns a list of nodes, but you only expect a single result. This method saves you the process of searching through all the nodes yourself. From the list, it is easy to retrieve the node of interest and then its value or attribute.

The main activity involving the XML document is the processing of the message template and the substitution of field values in marked positions. The `ParseMessage` method provides this functionality (see Listing B-5), accepting a string list that contains the field mappings for the current record. The mappings are established and accessed using the `Values` property of a string list, which associates a text value with an identifying key.

Listing B-5: Performing the mail merge.

```
{ Parse the message tag and return its value }
function TMailTemplateMessage.ParseMessage(Fields: TStrings):
  string;
var
  Elements: IXMLDOMNodeList;
  FieldValue: string;
  Index: Integer;
begin
  Elements := FXMLDoc.getElementsByTagName(MessageTag);
  if Elements.length = 0 then
    raise EMailException.Create(NoMessage)
  Result := '';
  with Elements[0] do
    for Index := 0 to childNodes.length -1 do
      { Add text elements directly }
      if (childNodes[Index].nodeType = NODE_TEXT) or
          (childNodes[Index].nodeType = NODE_CDATA_SECTION)
          then
        Result := Result + childNodes[Index].text
      { For 'field' elements get the field value }
```

```
      else if (childNodes[Index].nodeType = NODE_ELEMENT) and
          (childNodes[Index].nodeName = FieldTag) then
      begin
        FieldValue := Fields.Values[childNodes[Index].text];
        if FieldValue = '' then
          { Error if no such field }
          raise EMailException.Create(
            Format(MissingField, [childNodes[Index].text]));
        if FieldValue = Empty then
          { Replace empty field notation with empty string }
          FieldValue := '';
        Result := Result + FieldValue;
      end;
end;
```

You locate the `message` element in the document (again using the `getElementsByTagName` method) and then step through each of its child elements, constructing the message text as you go. The children should only consist of text nodes, which are appended directly to the message, or `field` elements, for which you extract the field name and then append the value of that field from the mapping. Note that the `text` method of a node returns all the text contained within that node (at any level), so you do not have to traverse down to the actual text node and retrieve its value. An exception occurs if the field does not exist in the record (denoted by an empty string being returned from the mapping).

TIP

One special case exists when the field has an empty string value. If you were to try to place this directly in the field list, it would not save the entry (the list automatically returns an empty string for any key that does not have a value set). To let you recognize the difference between a field that does not exist at all, as opposed to one that has an empty value, you must substitute a flagging value for the missing one. This flag, the constant `Empty`, is checked for when the value is retrieved and is then reset to its empty value.

Database Access

In keeping with the modular approach, all the database access is contained within one unit, `MailTemplateQuery`, and managed through the `TMailTemplateQuery` class. An instance of the class is created and initialized by passing to it the configuration properties and the query to be executed (from the message template XML document).

From the configuration details it extracts the BDE alias and logon parameters. It then creates internal instances of a `TDatabase` and a `TQuery`, which are initialized from the passed-in values, before opening the query (see the code in Listing B-6).

Listing B-6: Initializing the query and extracting its field values.

```
{ Initialization—connect to database and open query }
constructor TMailTemplateQuery.Create(Props: TStrings;
  QuerySQL: string);
begin
  inherited Create;
  FFields   := TStringList.Create;
  FDatabase := TDatabase.Create(nil);
```

```
  with FDatabase do
  begin
    AliasName    := Props.Values[QueryAliasProp];
    DatabaseName := 'MailOut';
    LoginPrompt  := False;
    if Props.Values[QueryUserProp] <> '' then
      Params.Add('username=' + Props.Values[QueryUserProp]);
    if Props.Values[QueryPasswordProp] <> '' then
      Params.Add('password=' + Props.Values[QueryPasswordProp]);
    Connected := True;
  end;
  FQuery := TQuery.Create(nil);
  with FQuery do
  begin
    DatabaseName := FDatabase.DatabaseName;
    SQL.Text     := QuerySQL;
    AfterScroll  := QueryAfterScroll;
    Active       := True;
  end;
end;

{ Set up the list of fields and values }
procedure TMailTemplateQuery.QueryAfterScroll(DataSet: TDataSet);
var
  Index: Integer;
begin
  with FQuery do
    for Index := 0 to FieldCount -1 do
      if Fields[Index].DisplayText = '' then
        { If string value is empty then entry doesn't appear
          in the list, so replace it }
        FFields.Values[Fields[Index].FieldName] := Empty
      else
        FFields.Values[Fields[Index].FieldName] :=
          Fields[Index].DisplayText;
end;
```

Thereafter, the program interacts with the resulting data through the following attributes: the `NextRecord` method to step through each record in turn, the `EOF` property to determine when it has reached the end, and the `Fields` property to access the values from the current record. The field values are held in an associative format in a string list for use by the message substitution routine, as accessed through the `Values` property of the string list.

To place the field values into the list, you attach an event handler to the `AfterScroll` event of the query (see the code in Listing B-6). This handler is called whenever the current record changes, which is ideal for your purposes. You can cycle through each field returned by the query and place its name and value into the list. As mentioned earlier, special processing is required for fields with empty string values.

Drop It in the Post

Once you have constructed the mail message and merged in the fields from the database, you are ready to send it off. Again, make use of open standards by using an SMTP server to post the mail.

Wrap a `TNMSMTP` component in another object to provide a simple interface for the rest of the program (a *Façade* design pattern). One advantage of this approach is that you could come back later and replace the underlying

mail implementation without affecting the rest of the program. All you must do is retain the existing interface.

NOTE
As mentioned before, feel free to replace the `TNMSMTP` component with your favorite e-mail component. In Delphi 3, you can use the `TSMTP` component since the `TNMSMTP` one is not available.

The mailing object, `TMailTemplateSender` (from the `MailTemplate-Sender` unit), is passed the list of configuration properties upon its creation (see Listing B-7). From this list it extracts the ones it requires (the name and port of the SMTP host, and the user account to use) and initializes the SMTP component with them.

Listing B-7: Interfacing with the SMTP component.

```
{ Initialization }
constructor TMailTemplateSender.Create(Props: TStrings);
begin
  inherited Create;
  FSender := TNMSMTP.Create(nil);
  with FSender do
  begin
    Host := Props.Values[MailHostProp];
    try
      Port := StrToInt(Props.Values[MailPortProp]);
    except  { Ignore }
    end;
    UserId := Props.Values[MailUserProp];
    Connect;
  end;
end;

{ Send an e-mail }
procedure TMailTemplateSender.Send(
  FromEmail, ToEmail, Subject, Message: string);
begin
  with FSender.PostMessage do
  begin
    FromAddress    := FromEmail;
    ToAddress.Text := ToEmail;
    Subject        := Subject;
    Body.Text      := Message;
  end;
  FSender.SendMail;
end;
```

Thereafter, the only interaction with the mailer is to request that a completed message be sent. The `Send` method (also in Listing B-7) takes the sender's name and the recipient's e-mail addresses, along with the subject and body of the message as parameters. These are parceled up and sent out.

Logging and Testing

To keep an eye on what is happening within your application, generate a log file for each run. This log contains the parameters passed to the program and the recipients of the completed messages.

For testing purposes, the log file also captures the entire text of the message, as it would have been sent. This approach allows you to verify that

the merge process is working as expected before sending out your message. A flag in the configuration file determines whether or not you are in test mode.

To continue your goal of modularizing the program, you put the logging functionality into its own object in a separate unit, `MailTemplateLog`. The `TMailTemplateLog` object (see Listing B-8) automatically creates a timestamped log file based on the name of the application when it is itself created. Including the current time within the filename ensures that previous logs are not overwritten (although you must remember to purge the old log files at some stage). The log file is automatically closed when the wrapper object is destroyed. This class is another example of the *Façade* design pattern, hiding several more complex functions behind a simplified interface.

Listing B-8: Logging your actions.

```
{ Open the log file }
constructor TMailTemplateLog.Create;
var
  Filename: string;
begin
  inherited Create;
  Filename := ChangeFileExt(ExtractFileName(ParamStr(0)),
    FormatDateTime(LogFormat, Now) + LogExt);
  AssignFile(FLogFile, Filename);
  Rewrite(FLogFile);
end;

{ Close the log file }
destructor TMailTemplateLog.Destroy;
begin
  CloseFile(FLogFile);
  inherited Destroy;
end;

{ Write an error message }
procedure TMailTemplateLog.Error(Error: Exception);
begin
  Log(Error.Message);
end;

{ Write a log message }
procedure TMailTemplateLog.Log(Message: string);
begin
  Writeln(FLogFile, TimeStamp + Message);
  Flush(FLogFile);
end;

{ Write a testing message }
procedure TMailTemplateLog.LogTest(
  FromEmail, ToEmail, Subject, Message: string);
begin
  Writeln(FLogFile, TestOnly);
  Writeln(FLogFile, LogFrom + FromEmail);
  Writeln(FLogFile, LogTo + ToEmail);
  Writeln(FLogFile, LogSubject + Subject);
  Writeln(FLogFile, LogMessage + Message);
  { Ensure it gets written out }
  Flush(FLogFile);
end;

{ Return the current time }
function TMailTemplateLog.TimeStamp: string;
begin
  Result := FormatDateTime(TimeFormat, Now);
end;
```

You then have three methods for interacting with the log file: `Log`, `LogTest`, and `Error`. `Log` adds a simple timestamped message to the file. `LogTest` is a convenience method that records all the details for a message sent while in test mode. Finally, `Error` records any exceptions that are passed to it. All these methods flush the file buffer before they complete, ensuring that you are able to see all the relevant log messages.

All Together Now

Now that you have a set of objects, each performing its own specialized task with minimal interactions between them, you can pull them all together into a coherent whole.

The application has no user interface, so all of the main code appears in the `.dpr` unit (see Listing B-9), and is marked as being a console application with the `{$APPTYPE CONSOLE}` directive.

Listing B-9: The completed mail-out processing.

```
var
  FromEmail, ToEmail, Subject, Message: string;
  QuerySQL, EmailField: string;
  Count: Integer;
  LogFile: TMailTemplateLog;
  Template: TMailTemplateMessage;
  Query: TMailTemplateQuery;
  Sender: TMailTemplateSender;
Begin
  Props    := TStringList.Create;
  LogFile  := nil;
  Template := nil;
  Query    := nil;
  Sender   := nil;
  Count    := 0;
  try
    try
      { Load the program properties }
      LoadMailProperties(Props);
      { Create and open the log file }
      LogFile    := TMailTemplateLog.Create;
      { Open the XML template document }
      Template   :=
        TMailTemplateMessage.Create(Props.Values[TemplateProp]);
      { Extract various parameters }
      FromEmail  := Props.Values[MailFromProp];
      QuerySQL   := Template.NodeValue(QueryTag);
      Subject    := Template.NodeValue(SubjectTag);
      EmailField := Template.AttributeValue(QueryTag, EmailAttr);
      { Query the database }
      Query      := TMailTemplateQuery.Create(Props, QuerySQL);
      { Create an interface to the e-mail system }
      if not Testing then
        Sender := TMailTemplateSender.Create(Props);
      { Log parameters }
      LogFile.Log(Started);
      LogFile.Log(LogFrom + FromEmail);
      LogFile.Log(LogTemplate + Props.Values[TemplateProp]);
      LogFile.Log(LogSubject + Subject);
      LogFile.Log(LogDatabase + Props.Values[QueryAliasProp]);
      LogFile.Log(LogQuery + QuerySQL);
      { Process each record from the query }
      while not Query.EOF do
      begin
```

```
        { Get the recipient }
        ToEmail := Query.Fields.Values[EmailField];
        { Perform the mail merge -
          XML document with query fields }
        Message := Template.ParseMessage(Query.Fields);
        { And output the results }
        if Testing then
          logFile.LogTest(FromEmail, ToEmail, Subject, Message)
        else
        begin
          Sender.Send(FromEmail, ToEmail, Subject, Message);
          LogFile.Log(Format(EmailSent, [ToEmail]));
          { Pause so as not to overwhelm the e-mail server }
          Sleep(PauseTime);
        end;
        Inc(Count);
        Query.NextRecord;
      end;
    except on Error: Exception do
      { Catch any errors and report them }
      LogFile.Error(Error);
    end;
  finally
    LogFile.Log(Format(Finished, [Count]));
    { Tidy up }
    Props.Free;
    LogFile.Free;
    Template.Free;
    Query.Free;
    Sender.Free;
  end;
end.
```

The steps in generating and sending the e-mail messages are as follows:

1. Check for any command-line parameters in `LoadMailProperties`, as these can be used to pass in the name of a configuration file to read instead of the default one. If no file is specified, the program looks for one with the same name as itself but with an `.xml` extension. From the selected file, the program properties are retrieved into a string list using the `LoadPropertiesFromXML` routine from the `XMLConfig` unit. This list is passed to the other objects for them to extract their necessary values.

2. Create a `TMailTemplateLog` object to record your current session and write initial settings to it.

3. Load the XML template file. Its name is retrieved from the configuration parameters and is passed to a `TMailTemplateMessage` object.

4. Extract the query to be executed from the template file and pass it, along with the configuration parameters, to a `TMailTemplateQuery` object.

5. Iterate through all the records returned from the query, performing the mail merge as you go.

6. Send each completed message to a `TMailTemplateSender` object to mail out, or write it to the log file if only testing. A pause is taken after each message is sent. This wait reduces the load on the mail server, and its length is configurable through the properties file.

7. Finalize the log file entries and free up all the objects. Your mail-out is complete. To run the example project, you need to set up the `mailtemp` database alias with the BDE to point to the supplied customer table.

NOTE
The code for this application appears within the `.dpr` file but not within a procedure or class method. All Pascal programs have their main code in the body of the main unit, between a `begin` and the final `end`. In a more typical Delphi program for Windows, you find that the `.dpr` contains code to initialize the application, create the opening forms, and then set it all going. You are free to add or alter the code that appears there, although most often the standard code is sufficient.

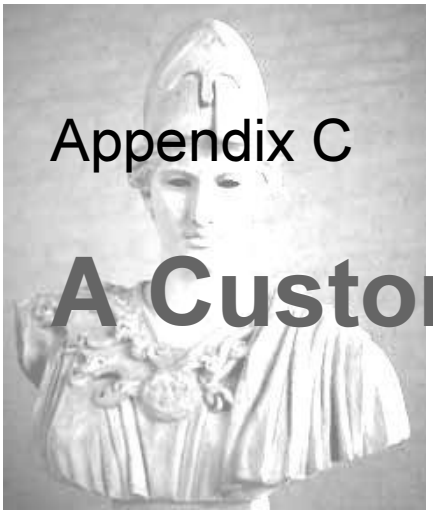
You can use the application as it stands for generating mass mailings from your database of contacts (but only with their permission of course). Just alter the configuration file for your database and server situation, then create the mail template with its embedded query, and away you go. Enhancements to the program could include an `attachment` element in the template XML document that causes the named file(s) to be sent out with each message. The rest is up to you.

Summary

Using open standards helps to protect your coding investment from future technology changes. This program works with any SQL database and with any SMTP server. Similarly, partitioning the application into several modules/objects, each of which has a well-defined and simple interface, allows you to more easily modify parts of the program with minimal effects on the remainder.

The application described here performs customized electronic mass mail-outs. It retrieves configuration information from an XML document, selects records from a database using SQL, merges fields from these records into a message format held in another XML document, and sends the completed message out into the world using a SMTP server.

Due to the use of XML for the configuration file and message template, these details can be easily altered without an in-depth knowledge of the program mechanics, and without requiring a recompilation.



Appendix C
A Customized Client

Since all XML documents follow the rules described in Chapter 2 and have a simple tree structure, it is easy to process them in a generic manner. Applications can display the tree, create new documents based on the DTD, or search through the data for specific values in particular fields. However, generic applications are not always the most user-friendly. You are forced to use the tree structure that XML defines, whereas related data may be better presented in some other format. Hence there is often a need for a customized client program, designed specifically to handle a particular document type (those based on one DTD). XML still provides an application-independent transfer mode, allowing the client to easily interoperate with a database serving up the data, or with another application that also knows about this XML type.

To illustrate how to load and process an XML document on the client side, you can use the movie-watcher format described previously. With Delphi you can produce a program that reads the document, transforms it into domain-specific objects, and then presents a UI to browse through them. Recall that the elements in this document are related to each other through `ID` and `IDREF` type attributes, which form the basis of the navigation you provide within the application.

The Client

Your client application extracts all the relevant details from the XML document and places them into three lists: movies, cinemas, and screenings. The main form then displays the details to the user and lets them browse the information. A tab control provides the main access to each of the three lists. As an item is selected from a list, its details are displayed on the right side of the form (see Figures C-1 through C-3).

Secondary navigation is provided by double-clicking on linking fields, such as on the list of cinemas on the movie page, or on the movie name on the screening page. In this way you can easily find a movie, select a session, and find out where the cinema is.

The XML document to load is specified as a command-line parameter to the application (this is necessary for later on). To access the file's name, use the `ParamStr` function:

```
LoadDocument(ParamStr(1), FMovies, FCinemas, FScreenings);
```

Figure C-1: Select a current movie.

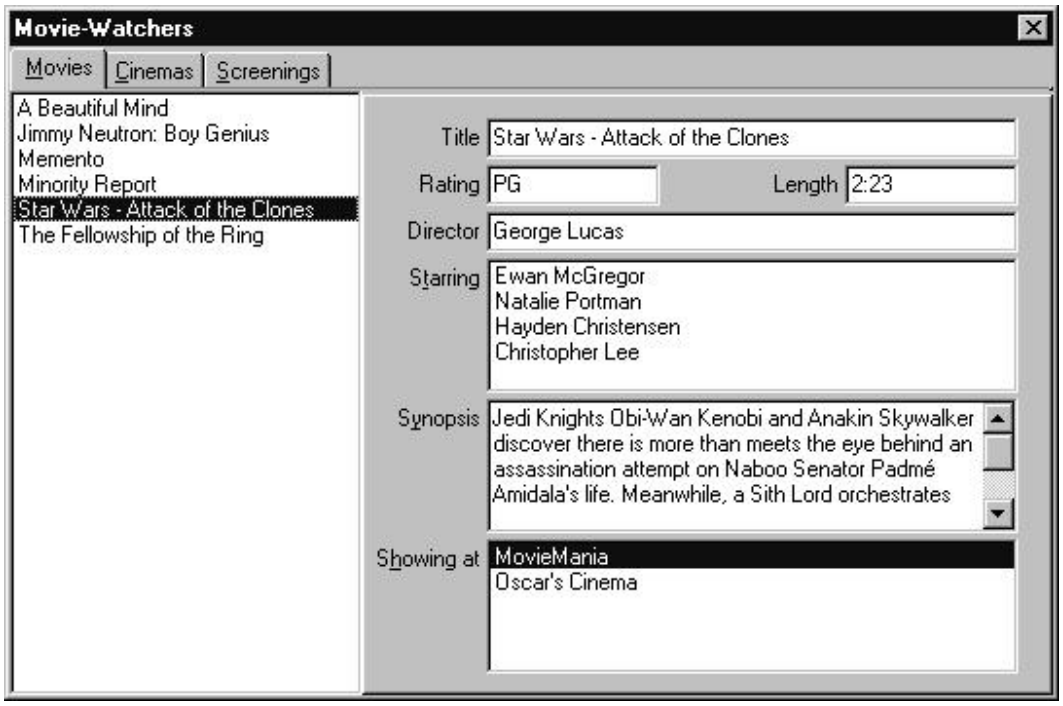


Figure C-2: Find a time when it is showing.

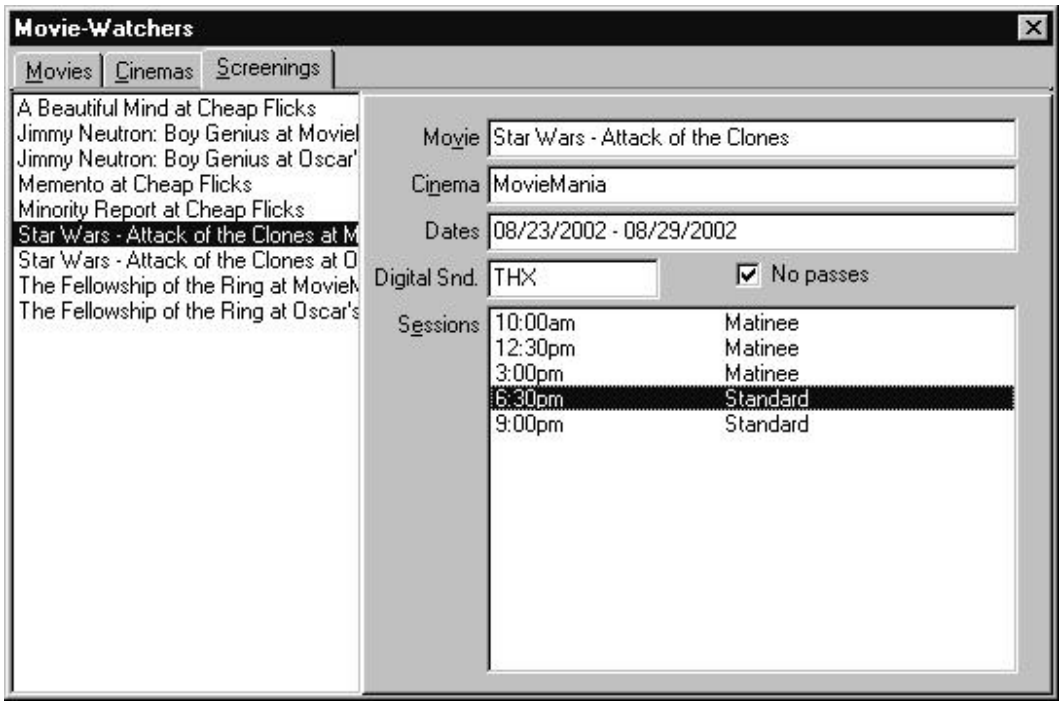
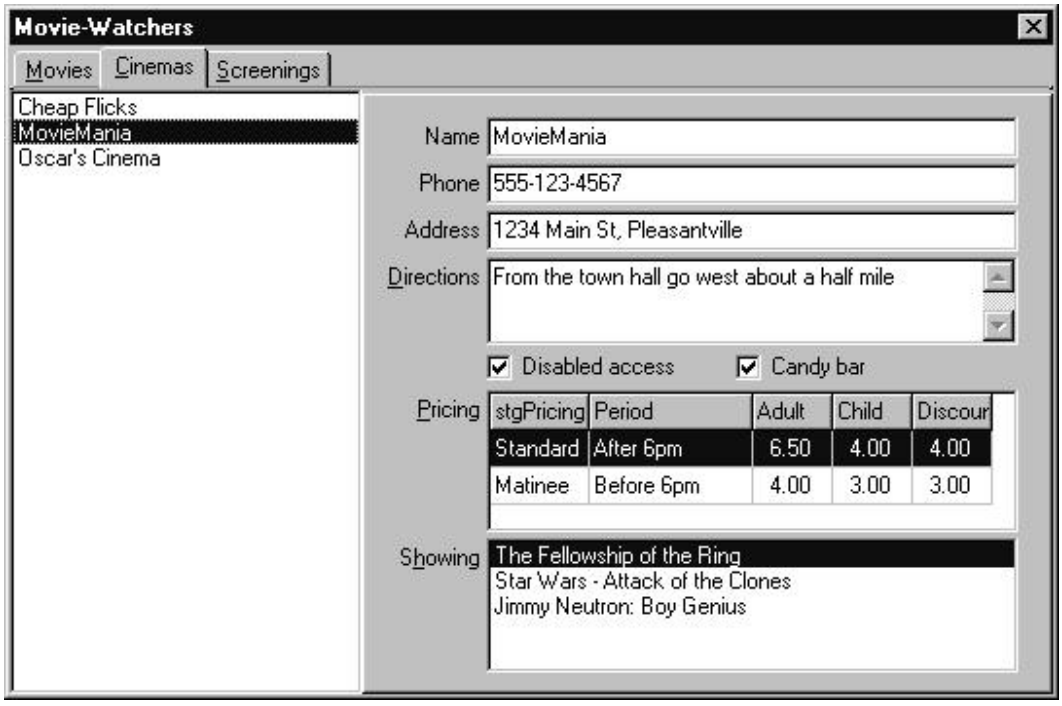


Figure C-3: See what the cinema has to offer.



You make use of the string list's ability to associate an object with each string throughout the program. Each list returned from the load procedure contains the object's display name as the string value, and adds a reference to an appropriate object in the corresponding `Objects` property. As the user selects different lists, copy their contents into the list box on the form (which sorts them automatically) along with the object references. Then, when more details are requested, you have immediate access to the necessary object and its attributes.

TIP

String lists are very useful in Delphi programming. They do much more than just manage a list of strings. Setting the `Sorted` property to `True` automatically orders the contents. Use the `Duplicates` property to control the handling of duplicate values in sorted lists. The `Values` property allows you to map from one string value to another, especially useful when dealing with `.ini` file style values. And finally, the `Objects` property lets you associate any object with a particular string value.

Information Hiding

To insulate the user interface from the source XML document, introduce a separate unit, `MWObjs`, which defines the classes corresponding to the objects extracted from the XML. Here you flatten out the XML tree structure, providing properties for sub-elements and attributes, and direct pointers to other objects rather than indirect ones through `ID` references. Compare the movie object in Listing C-1 with the XML structure shown in Listing 2-1.

Listing C-1: A movie object.

```
{ Details about a movie }
TMovie = class(TObject)
private
  FId: string;
  FName: string;
  FRating: TMovieRating;
  FLength: TDateTime;
  FDirector: string;
  FStars: TStringList;
  FSynopsis: string;
  function GetRatingText: string;
  procedure SetRatingText(RatingText: string);
public
  constructor Create(Id: string);
  destructor Destroy; override;
  property Id: string read FId write FId;
  property Name: string read FName write FName;
  property Rating: TMovieRating read FRating write FRating;
  property RatingText: string read GetRatingText
    write SetRatingText;
  property Length: TDateTime read FLength write FLength;
  property Director: string read FDirector write FDirector;
  property Stars: TStringList read FStars write FStars;
  property Synopsis: string read FSynopsis write FSynopsis;
end;
```

Although you could navigate through the XML tree itself and extract all the necessary details yourself, this approach makes it much easier for the application to deal with the information. You do not have to know about the structure of XML documents and what internal objects are used to represent them. Instead, you have real-world objects with familiar patterns of properties. Furthermore, having this extra layer means that you could, at some time in the future, load the data from another source or in some other way without having to change the user interface.

NOTE

This hiding of implementation details is one of the mainstays of object-oriented programming, known as *encapsulation*. By reducing the knowledge of one object or module required by another, you reduce their reliance on one another. This *decoupling* of the objects makes it easier to make changes in one place without adversely affecting another area. Using interfaces is another important way to enforce decoupling.

The `LoadDocument` procedure declared in this unit handles all the translation for you. Just pass it the name of the XML document and three lists to use in returning the data.

To create these movie-watcher objects, parse the source XML document to generate them. Using a SAX-compliant parser makes this an easy and maintainable task.

Parsing the XML Documents

As is usually the case in using SAX for XML processing, you need to write a content handler that knows about the expected document format, which means implementing the `IContentHandler` interface. Passing an instance of the handler to the SAX-compliant parser and supplying a document identifier

causes the parser to invoke the events in the handler as it reads the various parts of the document.

The simplest way to define a class that implements the document handler interface is to make use of the default handler supplied by SAX. The `TDefaultHandler` class (in the `SAXHelpers` unit) implements all of the standard SAX handler interfaces, supplying default behaviors for each method that generally do nothing. All these routines are declared as virtual, allowing you to easily replace them in a subclass through overriding.

This is exactly what you do with the movie-watcher content handler, as shown in Listing C-2. To generate the movie-watcher objects, only a few of the SAX events need to be dealt with. Here you see the benefit of using the default handler as a base. All the other SAX routines, which must be implemented to satisfy the requirements of the interface, are already defined, and do not interfere with your specific processing of the document.

Listing C-2: Declaring a movie-watcher document handler.

```
{ A SAX content handler that knows about movie-watcher documents }
TMWContentHandler = class(TDefaultHandler)
private
  FCinema: TCinema;
  FCinemas: TList;
  FMovie: TMovie;
  FMovies: TList;
  FPrice: TPrice;
  FScreening: TScreening;
  FScreenings: TList;
  FText: string;
public
  constructor Create;
  destructor Destroy; override;
  property Cinemas: TList read FCinemas;
  property Movies: TList read FMovies;
  property Screenings: TList read FScreenings;
  { IContentHandler }
  procedure Characters(const ch: SAXString); override;
  procedure EndElement(const uri, localName, qName: SAXString);
override;
  procedure StartElement(const uri, localName, qName: SAXString;
    const atts: IAttributes); override;
end;
```

As can be seen in the `LoadDocument` routine (see Listing C-3), an instance of the customized content handler is created, along with an instance of a SAX-compliant parser (in this case the default one). The `TMWContentHandler` class constructs three lists, corresponding to the string lists used in the client program, and fills them with the domain-specific objects it extracts from the document.

Listing C-3: Loading the movie-watcher document.

```
{ Load XML document and process into string lists
  with references to the appropriate objects }
procedure LoadDocument(URI: string;
  MoviesList, CinemasList, ScreeningsList: TStringList);
var
  Index: Integer;
  XMLReader: IXMLReader;
  Handler: TMWContentHandler;
begin
  { Create the XML parser }
  Handler := TMWContentHandler.Create;
```

```
  try
    XMLReader               := GetSAXVendor.XMLReader;
    XMLReader.ContentHandler := Handler;
    { And parse the document }
    XMLReader.parse(URI);
    with Handler do
    begin
      { Are they all here? }
      if (Movies.Count = 0) or (Cinemas.Count = 0) or
          (Screenings.Count = 0) then
        raise Exception.Create(InvalidDocument + URI);
      { Step through the handler's lists and convert to output format }
      for Index := 0 to Movies.Count - 1  do
        MoviesList.AddObject(TMovie(Movies[Index]).Name, Movies[Index]);
      for Index := 0 to Cinemas.Count - 1 do
        CinemasList.AddObject(
          TCinema(Cinemas[Index]).Name, Cinemas[Index]);
      for Index := 0 to Screenings.Count - 1 do
        ScreeningsList.AddObject(Format(ScreeningDesc,
          [TScreening(Screenings[Index]).Movie.Name,
          TScreening(Screenings[Index]).Cinema.Name]),
          Screenings[Index]);
    end;
  finally
    if Handler.RefCount = 0 then
      Handler.Free;
  end;
end;
```

Once the parse process has completed, these lists are transferred to the ones supplied by the calling program. For each of the internal lists, you step through all the items and set the identifying string to an appropriate value. Movies and cinemas have their names entered, while screenings combine the names of their associated movie and cinema.

Constructing Model Objects

The first step in building the movie-watcher object model is performed by the document handler's constructor. Since the class is intended for a single use, the constructor creates the necessary lists.

Then, as each element is encountered and the handler is notified through the `StartElement` method, you prepare the model environment for later processing in the other event routines. For elements that correspond to objects within the internal model, create a new instance of them and add it to their appropriate list (see Listing C-4). References to the most recently constructed objects are held within the object for them to be accessed later.

Listing C-4: Preparing a new real-world object.

```
{ Create objects as necessary for document elements }
procedure TMWContentHandler.StartElement(
    const uri, localName, qName: SAXString; const atts: IAttributes);

  { Locate the movie with the given identifier }
  function FindMovie(Id: string): TMovie;
  var
    Index: Integer;
  begin
    Result := nil;
    for Index := 0 to FMovies.Count - 1 do
      if TMovie(FMovies[Index]).Id = Id then
      begin
```

```
        Result := TMovie(FMovies[Index]);
        Exit;
      end;
  end;

  { Locate the cinema with the given identifier }
  function FindCinema(Id: string): TCinema;
  var
    Index: Integer;
  begin
    Result := nil;
    for Index := 0 to FCinemas.Count - 1 do
      if TCinema(FCinemas[Index]).Id = Id then
      begin
        Result := TCinema(FCinemas[Index]);
        Exit;
      end;
  end;

  { Locate the pricing scheme with the given identifier }
  function FindPrice(PriceId: string): TPrice;
  var
    Index, Index2: Integer;
  begin
    Result := nil;
    for Index := 0 to FCinemas.Count - 1 do
      with TCinema(FCinemas[Index]) do
      begin
        Index2 := Pricing.IndexOf(PriceId);
        if Index2 > -1 then
        begin
          Result := TPrice(Pricing.Objects[Index2]);
          Exit;
        end;
      end;
  end;

begin
  if qName = MWMovie then
  begin
    FMovie            := TMovie.Create(atts.getValue(MWId));
    FMovie.RatingText := atts.getValue(MWRating);
    FMovies.Add(FMovie);
  end
  else if qName = MWCinema then
  begin
    FCinema := TCinema.Create(atts.getValue(MWId));
    FCinemas.Add(FCinema);
  end
  else if qName = MWPrices then
  begin
    FPrice := TPrice.Create(atts.getValue(MWId));
    FCinema.Pricing.AddObject(atts.getValue(MWId), FPrice);
  end
  else if qName = MWScreening then
  begin
    FScreening := TScreening.Create(FindMovie(atts.getValue(MWMovieId)),
      FindCinema(atts.getValue(MWCinemaId)));
    FScreenings.Add(FScreening);
  end
  else if qName = MWSession then
    FPrice := FindPrice(atts.getValue(MWPriceId));
end;
```

Movie objects are created with their ID and rating, as extracted from the attributes of the element, before being added to the list of movies. Similarly, cinema instances are constructed and added to the cinemas list. Pricing details

belong to a particular cinema, so price elements cause a new price object to be added to the current cinema's (`FCinema`) own list.

Screenings contain references to the movie and cinema linked together through `IDREF` attributes. These objects are located from their respective lists before being passed to the screening object's constructor. As before, the resulting object is added to its list. Individual sessions within a screening refer to their pricing structure via an attribute. The associated price object is located and saved for later.

Accumulating Content

Other elements appear as properties of the model objects, rather than as objects in their own right. Their content appears as text that is returned to the handler through the `Characters` event. However, this method is only invoked as the content is parsed, following the `StartElement`. Hence, these elements are dealt with in the `EndElement` event, once their content has been identified.

Within the text content event (shown in Listing C-5), you add the new text to any existing value and save it for later. It is possible for an element's content to be made up of several text nodes, perhaps coming from different embedded elements (such as the `emph` element in the `synopsis`), or through the use of entity references or CDATA sections.

Listing C-5: Accumulating text.

```
{ Accumulate text content }
procedure TMWContentHandler.Characters(const ch: SAXString);
begin
  FText := FText + ch;
end;
```

TIP
Some XML parsers automatically normalize text as they read it. In others, this behavior can be controlled through a property. The parser used here is fairly basic and simply returns all the text it finds, requiring the handler to do the operation itself.

Saving Properties

As described earlier, elements from the XML document that are present in the movie-watcher model as properties have their content built up within the `Characters` event. Once the end tag for those elements is encountered, you can transfer that accumulated text into the corresponding model object.

The `EndElement` routine (see Listing C-6) uses the element name to determine which object and property to set from the text. In the case of the `name` element, the element name is insufficient identification since it appears in the movie, cinema, and prices elements. For this reason, you need to check which object is currently being constructed (the non-`nil` one).

Listing C-6: Saving object model property values.

```
{ Save text content to appropriate property }
procedure TMWContentHandler.EndElement(
  const uri, localName, qName: SAXString);

  { Replace consecutive white space with one space }
  function Normalize(const Text: string): string;
  const
    Blanks = [#1..#32];
  var
    Index: Integer;
  begin
    Result := Text;
    if Length(Text) < 2 then
      Exit;
    for Index := Length(Result) downto 2 do
      if (Result[Index] in Blanks) and
          (Result[Index - 1] in Blanks) then
      begin
        Result[Index - 1] := ' ';
        Delete(Result, Index, 1);
      end;
  end;

  { Return the accumulated text and clear for next time }
  function ReadAndClearText: string;
  begin
    Result := Trim(Normalize(FText));
    FText  := '';
  end;

begin
  if qName = MWMovie then
    FMovie := nil
  else if qName = MWMovie then
    FCinema := nil
  else if qName = MWPrices then
    FPrice := nil
  else if qName = MWScreening then
    FScreening := nil
  else if qName = MWName then
  begin
    if Assigned(FMovie) then
      FMovie.Name := ReadAndClearText
    else if Assigned(FPrice) then
      FPrice.Name := ReadAndClearText
    else if Assigned(FCinema) then
      FCinema.Name := ReadAndClearText;
  end
  else if qName = MWLength then
    FMovie.Length := StrToInt(ReadAndClearText) / 24 / 60
  else if qName = MWDirector then
    FMovie.Director := ReadAndClearText
  else if qName = MWStar then
    FMovie.Stars.Add(ReadAndClearText)
  else if qName = MWSynopsis then
    FMovie.Synopsis := ReadAndClearText
  else if qName = MWPhone then
    FCinema.Phone := ReadAndClearText
  else if qName = MWAddress then
    FCinema.Address := ReadAndClearText
  else if qName = MWDirections then
    FCinema.Directions := ReadAndClearText
  else if qName = MWCandyBar then
    FCinema.CandyBar := True
  else if qName = MWDisabledAccess then
    FCinema.DisabledAccess := True
  else if qName = MWPeriod then
```

```
    FPrice.Period := ReadAndClearText
  else if qName = MWAdult then
    FPrice.Adult := StrToFloat(ReadAndClearText)
  else if qName = MWChild then
    FPrice.Child := StrToFloat(ReadAndClearText)
  else if qName = MWDiscount then
    FPrice.Discount := StrToFloat(ReadAndClearText)
  else if qName = MWStartDate then
    FScreening.StartDate := StrToDateTime(ReadAndClearText)
  else if qName = MWEndDate then
    FScreening.EndDate := StrToDateTime(ReadAndClearText)
  else if qName = MWNoPasses then
    FScreening.NoPasses := True
  else if qName = MWDigitalSound then
    FScreening.DigitalSound := ReadAndClearText
  else if qName = MWSession then
    FScreening.Showing.AddObject(ReadAndClearText, FPrice);
end;
```

The supplied text must be normalized before being used. This processing replaces consecutive occurrences of white space characters with a single space and trims white space from the start and end of the text. The `ReadAndClearText` function performs this activity, as well as clearing out the `FText` field so it is ready for accumulating text for the next node.

Properties that are not text values are converted as necessary, such as the ticket prices and the screening dates. Some elements provide information simply through their presence, like the disabled access and candy bar settings for a cinema. Here you set the corresponding Boolean property to `True` when they are encountered.

The objects that are being operated on were created in the appropriate `StartElement` method, and the saved references are used here.

NOTE

Elements that do not contribute to the object model structure, and do not have any text content can be ignored in the event handlers. Examples from the current documents include the `starring` element from the movies, and the `facilities` element from the cinemas. Although they are not used here, they are necessary when generating an HTML representation since they serve to group their sub-elements.

Client Processing

The returned lists are used within the client application for display and navigation purposes. Since they are string lists they can be assigned directly to the `Items` property of the list box on the left of the form. Setting the `Sorted` property of that list box automatically reorders the entries for display, retaining the association with the attached objects. The `tabNavigationChange` method of the form (see Listing C-7) is invoked when the user selects one of the tabs on the screen (and during the initial load). It performs the necessary assignment.

As items in the list are selected, it is easy to retrieve all the information to be displayed through the corresponding `Objects` entry. This technique is shown in the `lbxNavigationClick` routine (see Listing C-7).

Listing C-7: Display movie items.

```
{ Show selected details in listbox }
procedure TfrmMovieWatchers.tabNavigationChange(Sender: TObject);
begin
  with lbxNavigation do
  begin
    Items.BeginUpdate;
    Items.Clear;
    if tabNavigation.TabIndex = MoviesTab then
      Items := FMovies
    else if tabNavigation.TabIndex = CinemasTab then
      Items := FCinemas
    else if tabNavigation.TabIndex = ScreeningsTab then
      Items := FScreenings;
    Items.EndUpdate;
  end;
  lbxNavigation.ItemIndex := 0;
  lbxNavigationClick(lbxNavigation);
  ActiveControl           := lbxNavigation;
end;

{ Select an item to display its details }
procedure TfrmMovieWatchers.lbxNavigationClick(Sender: TObject);
begin
  with lbxNavigation do
  begin
    if ItemIndex < 0 then
      ItemIndex := 0;
    if tabNavigation.TabIndex = MoviesTab then
      ShowMovie(TMovie(Items.Objects[ItemIndex]))
    else if tabNavigation.TabIndex = CinemasTab then
      ShowCinema(TCinema(Items.Objects[ItemIndex]))
    else if tabNavigation.TabIndex = ScreeningsTab then
      ShowScreening(TScreening(Items.Objects[ItemIndex]));
  end;
end;

{ Display details for a movie }
procedure TfrmMovieWatchers.ShowMovie(Movie: TMovie);
var
  Index: Integer;
begin
  with Movie do
  begin
    edtTitle.Text          := Name;
    edtRating.Text         := MovieRatingText[Rating];
    edtLength.Text         := FormatDateTime(TimeFormat, Length);
    edtDirector.Text       := Director;
    lbxStars.Items         := Stars;
    memSynopsis.Lines.Text := Synopsis;
    { Show which cinemas it is playing at }
    with lbxCinemas.Items do
    begin
      BeginUpdate;
      Clear;
      for Index := 0 to FScreenings.Count -1 do
        if TScreening(FScreenings.Objects[Index]).Movie = Movie
        then
          AddObject(TScreening(
            FScreenings.Objects[Index]).Cinema.Name,
            FScreenings.Objects[Index]);
      if Count > 0 then
        lbxCinemas.ItemIndex := 0;
      EndUpdate;
    end;
  end;
  pgcDetails.ActivePage := tshMovie;
end;
```

From that object, you extract the details appropriate to its type and set them into the controls on the screen. The `ShowMovie` routine is shown in Listing C-7 as an example of the required processing. Using the power of string lists, combined with the domain-specific objects, makes displaying the details of the movies and their screenings fairly simple.

Other navigation comes from responding to user interactions with the client program. For example, double-clicking an entry in the list of cinemas showing a particular movie invokes the event handler shown in Listing C-8, which moves to the Screening page and locates the corresponding combination. For keyboard users, another event handler reacts to pressing the Enter key while on an entry in this list (reusing the functionality of the double-click routine).

Listing C-8: Additional navigation.

```
{ Go to the screening details for a movie }
procedure TfrmMovieWatchers.lbxCinemasDblClick(Sender: TObject);
begin
  ShowList(ScreeningsTab, Format(ScreeningDesc,
    [edtTitle.Text, lbxCinemas.Items[lbxCinemas.ItemIndex]]));
end;

{ Enter acts like a double-click }
procedure TfrmMovieWatchers.lbxCinemasKeyDown(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  if Key = VK_RETURN then
    lbxCinemasDblClick(lbxCinemas);
end;
```

To run the program, you must supply the name of the target XML document as a command line parameter. Running from within Delphi you specify this value through the Run | Parameters menu option.

Through the Browser

So far the application has been standalone. You supply it with the name of the file to load as a command-line parameter and it opens and displays that file. But one of the advantages of XML is its delivery across the Internet. To enable a downloaded file to trigger your client automatically, all you do is define a new file type for this class of documents.

To define this type in Windows you do the following:

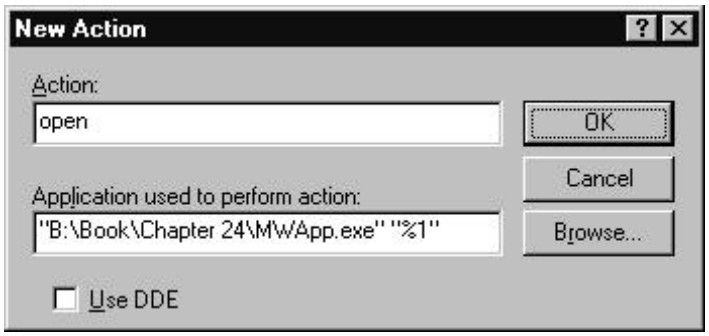
1. Open Windows Explorer, select View | Options, and select the File Types tab.
2. Examine the list of the registered file types and the associated programs that deal with them. Note that each has a list of file extensions that identify the type, the corresponding MIME type, and the name of the program that knows how to deal with them.
3. Add a new file type for the movie-watcher XML documents by pressing New Type.
4. Enter a description, "`Movie-Watcher`", the content (MIME) type, "`application/x-movie-watcher`", and the extension, "`.mwx`". The

MIME type, "`application/x-???`," indicates that the file is application specific.

5. Press New for a default action.

6. Enter its name, "`open`", and press Browse to search for your application. Follow the path and filename with the text "`%1`" to indicate that the name of the file being opened is passed to the program (hence the need for the processing of the command-line parameter earlier). Press OK to save the action (see Figure C-4).

Figure C-4: The open action for movie-watcher documents.



7. Change the associated icon if you wish. Set the other check box options if desired.

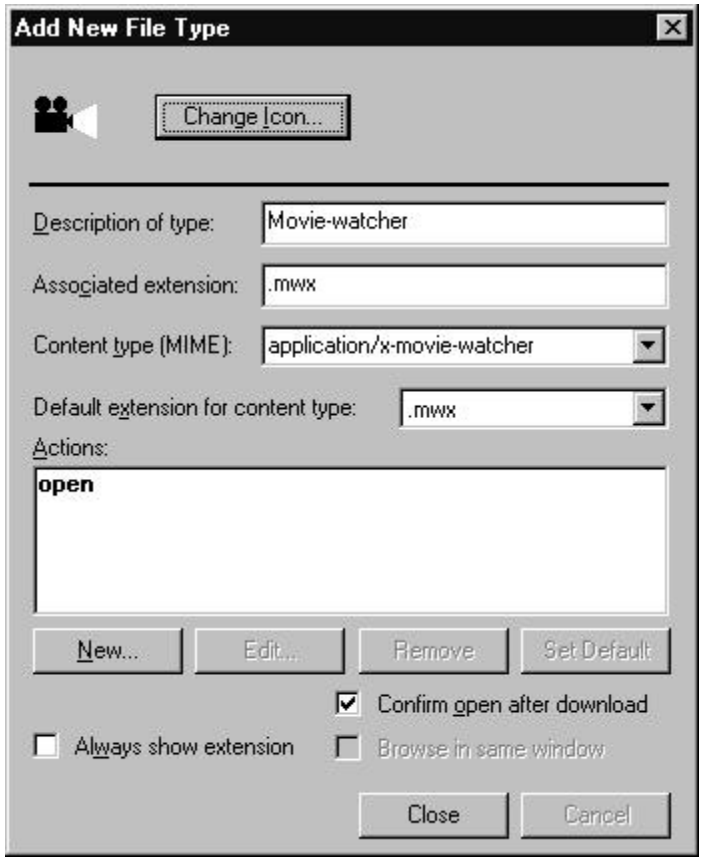
8. Save the results (see Figure C-5) by pressing the Close button.

Having defined the new type, you must rename the movie-watcher XML document to have an `.mwx` extension. Now whenever this file type is opened up within your browser, it loads directly into your application. A temporary file is created to hold the downloaded text, with the name of that file being passed to your client program as a command-line parameter.

TIP

You may need to set up your Web server to supply the correct MIME type for these documents. This process is dependent on the server that you are using, however, you need to associate the `application/x-movie-watcher` MIME type with the `.mwx` extension.

Figure C-5: A new file type for movie-watcher documents.



Summary

Although generic processors can handle XML in many useful ways, one of the advantages of using XML is that the information held within can also be sent to specialized applications and easily accessed. This allows for more user-friendly processing, as well as increased integrity and validations, without losing the benefits of XML in data interchange and legibility.

The application described here shows how you can write a client application in Delphi that receives and processes a particular class of XML documents. By defining a new file type in the registry specific to this type of XML document, you can have your Web browser automatically kick off the program whenever such a file is downloaded. Delivering data was never so easy.

Compare this SAX implementation of the custom viewer with the XML data binding version discussed in Chapter 22.

Index