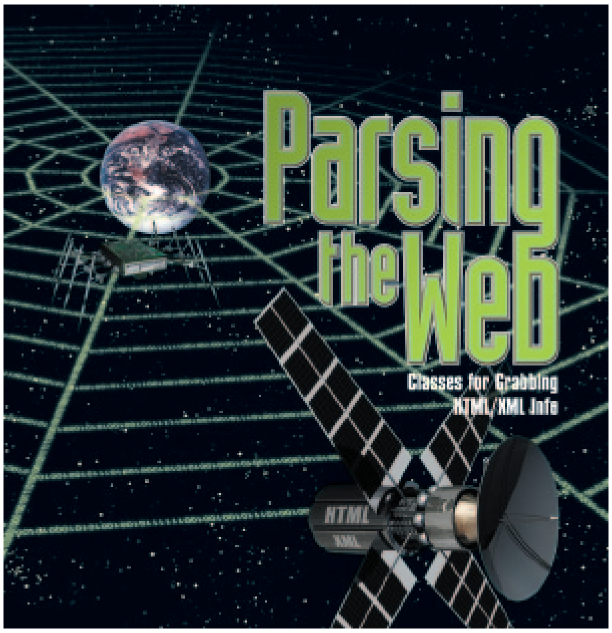
May 2000, Volume 6, Number 5

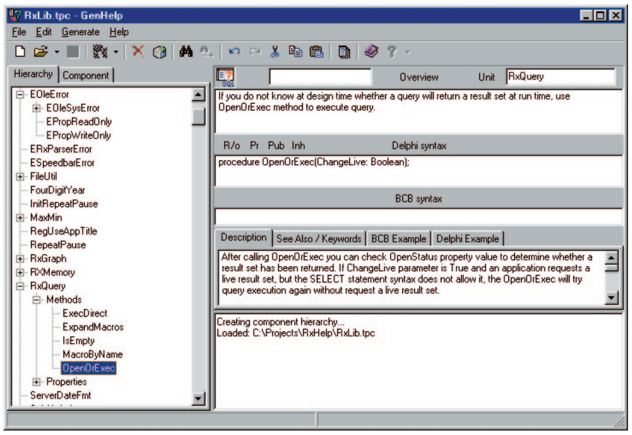


Cover Art By: *Arthur Dugoni*

FraserSoft Announces GenHelp

FraserSoft announced the latest version of *GenHelp*, the company's component Help file generator. GenHelp can import Pascal and C++ header files to produce the basis for component Help files. Help files created by GenHelp adhere to the look and feel of Borland's Help files and integrate fully into the IDEs of Delphi and C++Builder.

GenHelp employs a full graphical interface to speed up the creation of Help files.



```
EOleError
  EOleSysError
  EPropReadOnly
  EPropWriteOnly
ERxParserError
ESpeedbarError
FileUtil
FourDigitYear
InitRepeatPause
MaxMin
RegUseAppTitle
RepeatPause
RxGraph
RXMemory
RxQuery
  Methods
    ExecDirect
    ExpandMacros
    IsEmpty
    MacroByName
    OpenOrExec
  Properties
ServerDateFmt
```

If you do not know at design time whether a query will return a result set at run time, use OpenOrExec method to execute query.

```
procedure OpenOrExec(ChangeLive: Boolean);
```

After calling OpenOrExec you can check OpenStatus property value to determine whether a result set has been returned. If ChangeLive parameter is True and an application requests a live result set, but the SELECT statement syntax does not allow it, the OpenOrExec will try query execution again without request a live result set.

```
Creating component hierarchy...
Loaded: C:\Projects\RxHelp\RxLib.tpc
```

FraserSoft
Price: US$50
E-Mail: pete.fraser@frasersoft.clara.net
Web Site: http://www.frasersoft.clara.net/genhelp

Extended Systems Releases RPM Server and Crystal Reports Driver for Advantage Database Server

Extended Systems, Inc. released *RPM Server* (Remote Procedure Middleware) and *Crystal Reports Driver* for Advantage Database Server.

RPM Server is a middle-tier software solution that allows application developers to move intensive database processing off the client application and onto the database server. RPM Server complements Advantage Database Server, the company's SQL client/server software, by acting as a workhorse for thin clients. The combination provides an alternative to traditional two-tier database applications, where client applications perform much of the application's custom database activity. RPM Server offers a way to deploy and maintain code at a single point, providing greater stability to the application and increased efficiency for end users.

The distributed architecture of RPM Server is designed to allow future enhancements that will provide connected middle-tier processing for mobile device platforms, such as Palm OS and Microsoft Windows CE. This technology provides the framework for developers to expand their business applications to include the handheld computer market. Rather than building traditional two-tier client/server database applications, developers can build centralized business rules and processes. RPM Server enables developers using Delphi to create true middleware applications.

Crystal Reports Driver for Advantage Database Server provides developers with a native interface for Seagate Software's reporting tool (works with Crystal Reports Driver version 6 and version 7MR1).

The Advantage Crystal Reports Driver utilizes the Advantage StreamlineSQL engine to provide access to FoxPro DBFs and Advantage Database tables. With the release of this new driver, Extended Systems now offers ODBC-free access to the Advantage Database Server through Crystal Reports and moves all SQL processing to the server.

This is a free tool available to all Advantage developers and requires Advantage Database Server 5.6 (version 2.6 client).

Extended Systems, Inc.
Price: Call for pricing.
Phone: (800) 235-7576 x5030
Web Site: http://www.AdvantageDatabase.com

SkyLine Tools Imaging Releases Doc-to-Net 6.0

SkyLine Tools Imaging announced *Doc-to-Net 6.0*, its Internet document displaying application and image conversion tool. With Doc-to-Net, users can send documents over the Internet without downloading a plug-in.

This CGI application transforms a scanned .tiff to a .png, .gif, or .jpeg, anti-aliases it "on the fly," and streams it through the browser. Added to this application are zoom, pan, and scroll features, as well as rotating and inverting controls. Doc-to-Net offers scaling (resizing), has been engineered to perform at high speed, and is royalty free.

Features include a COM object for ASP pages that will return the number of pages in a multi-page .tiff; the ability to recognize and return sizes of varied size pages within a multi-page .tiff; fewer DLLs required; the ability to specify the size of an image; the UNC path name feature, allowing files on different computers within a system to be accessed by the same server; and password protection.

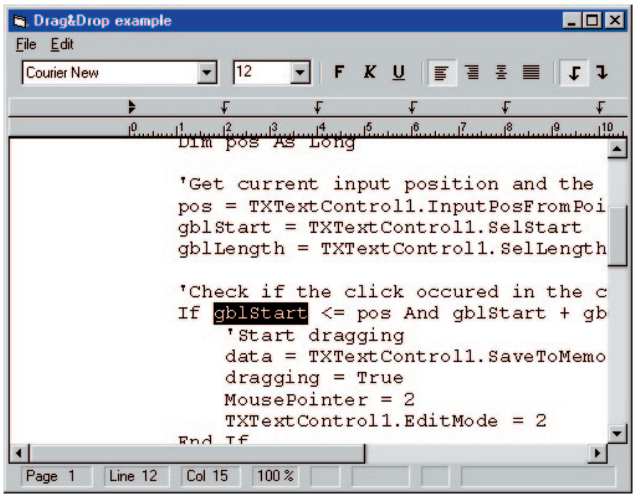
For photographic images, Doc-to-Net supports .bmp, .pcx, and .tga, and presents them as .jpeg, .png, or .gif files. Image correction tools offered for documents are also provided for images. The Doc-to-Net software is Windows-based and is recommended for use with Windows NT and Internet Information Server.

Developers who wish to utilize Doc-to-Net as a part of an independent programming application can leverage it as an add-on to SkyLine Tools' Corporate Suite.

SkyLine Tools Imaging
Price: US$599
Phone: (800) 404-3832
Web Site: http://www.imagelib.com

The Imaging Source Ships TX Text Control 7

The Imaging Source is shipping *TX Text Control 7*, its word processing software, in reusable component form. TX Text Control is a 32-bit programming component that includes a low-level API and an OCX that allows developers to add to their applications sophisticated text formatting and display capabilities typically seen in large word processing programs. TX Text Control 7 is royalty free; programs created with TX Text Control can be shipped to an unlimited number of customers without any additional costs.



```
Dim pos As Long

'Get current input position and the
pos = TXTextControl1.InputPosFromPoi
gblStart = TXTextControl1.SelStart
gblLength = TXTextControl1.SelLength

'Check if the click occured in the c
If gblStart <= pos And gblStart + gb
    'Start dragging
    data = TXTextControl1.SaveToMemo
    dragging = True
    MousePointer = 2
    TXTextControl1.EditMode = 2
End If
```

The Imaging Source
Price: From US$439 for TX Text Control Standard.
Phone: (877) 898-2875
Web Site: http://www.textcontrol.com

Blink*inc* Announces DeltaPatch 1.2

Blink*inc* announced *DeltaPatch 1.2*, a multi-platform change distribution system that makes it possible to create a small change or "patch" file to update databases, documents, and programs.

Instead of replacing an entire database after modifying a handful of records, or replacing a master document after changing only a few words, companies can e-mail a patch file of the changes. Programmers can send a small patch file of the changes to their software instead of redistributing the entire system every time they make a minor update. Remote sales people can download changes to customer and stock databases more frequently and in less time.

DeltaPatch also offers a solution to the problem of remotely updating entry-level computer users by creating self-applying patches that can automatically locate and update virtually any type of file. Patch files will only update existing users and the Apply program may be freely distributed, which makes it possible to safely post patch files on the Web for download. DeltaPatch allows companies to distribute product improvements, beta-test software, and bug fixes.

DeltaPatch's Build program uses an intelligent algorithm to compare old versions of files or directories to their new versions and create a patch file of the differences. A wizard guides users through a point-and-click process, and there is no limit to the size or number of files that can be compared. The Apply program uses the patch to update the destination system to the latest version under Windows NT/95/98/3.1 or DOS. The Apply program can back up all files as they are updated, so that application of a patch can be reversed.

Blink*inc*
Price: US$299
Phone: (804) 784-2087
Web Site: http://www.blinkinc.com

SkyLine Tools Announces ImageLib Combo@TheEdge 5.0

SkyLine Tools Imaging announced *ImageLib Combo@TheEdge 5.0*, a programming tool that provides 18 VCL components that enable the adding of images, image processing, and multimedia to applications in Windows 3.1/95/NT.

Museum-quality images can be scanned, rotated, flipped, sized, zoomed into, and converted to .jpeg, .png, .tiff, .gif, .pcx, .bmp, .ico, .wmf, .cms, and .scm. This package includes 16- and 32-bit versions. A VCL/DLL with source code is available for Delphi developers.

Combo@TheEdge features TWAIN support, a zoom tool, flipping of images, text over image, vertical and horizontal scrolling text messages, a thumbnail manager, and a video frame grabber.

Combo@theEdge provides 25 royalty-free flexible image correction and manipulation tools. Included are image cropping, scaling, and rotating, as well as image manipulation and special effects, including mosaic, wave, ripple, and a fisheye polar effect. Programmers can grayscale an image or add and reduce colors, brighten, sharpen, and increase the contrast of an image.

Multimedia formats supported are .avi, .mov, .rmi, .mid, and .wav with BLOb support. Scrolling text and vertical credit messages can be put into an application, and colors, fonts, and font sizes can be manipulated.

Combo@TheEdge enables the cutting, copying, and pasting of images to and from the Windows Clipboard and enables all image formats with 1-, 8-, 16- and 24-bit dithering. Optional RAD tool bars are available for image manipulation.

(Note: Unisys may require a royalty for applications developed using gif and tiff LZW.)

SkyLine Tools Imaging
Price: US$199
Phone: (800) 404-3832
Web Site: http://www.imagelib.com

New Wave Software Offers SPI 2.5

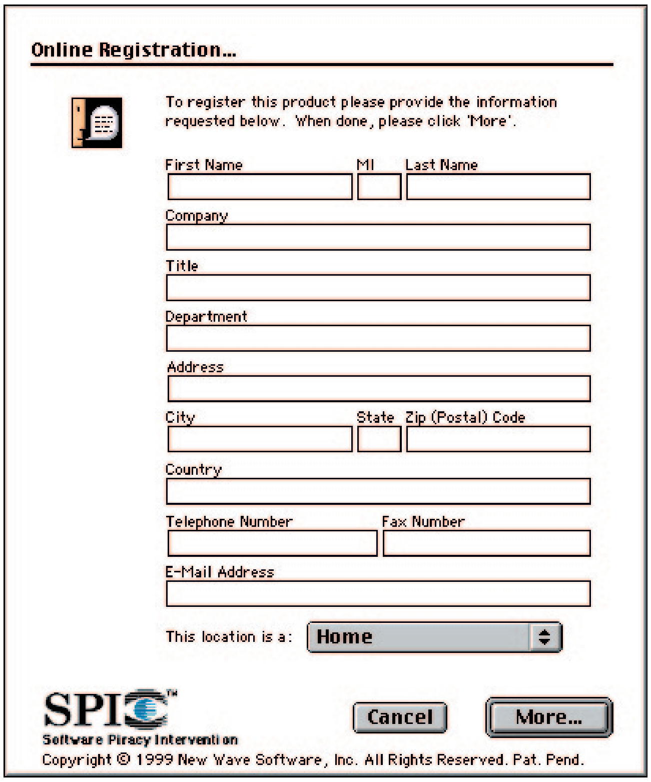
New Wave Software, Inc. announced *SPI 2.5*. SPI (Software Piracy Intervention) adds a protective layer to software products that allows only those end users who legitimately purchased the software to install it. No matter how many copies of a protected program get distributed illegally, only the valid purchaser can install the product.

SPI includes features such as passive and active encryption protection. SPI-protected products can be distributed over the Internet, on CD-ROM, or on diskette. Another feature of SPI is that it can remotely disable stolen prepackaged software; one phone call to New Wave can disable one, two, or thousands of stolen software products, rendering them useless.

The version 2.5 upgrade adds Locking General Purpose Registers, an Uninstaller feature, and built-in e-commerce. The Locking feature adds strength to the six internal General Purpose Registers by allowing the developer the option of setting them to Read Only, Write Only, Increment Only, Decrement Only, or Read/Write by clicking a radio button.

The Uninstaller feature allows the end user to move an SPI-protected product to another computer. The end user runs the SPI Installer on the computer where the product was first installed. SPI disables the original installation and resets the license, allowing the end user to install the protected product to another computer.

The built-in e-commerce feature allows vendors to sell products online without the use of "shopping cart" software; customers can simply download a product and select "Purchase" when they run the SPI Installer. A dialog box appears where the end user enters their credit card information, which is encrypted and transferred to CyberCash or Authorize.Net for real-time authorization. Upon authorization, the SPI-protected product is installed and ready to run.


Online Registration...

To register this product please provide the information requested below. When done, please click 'More'.

First Name	MI	Last Name
Company		
Title		
Department		
Address		
City	State	Zip (Postal) Code
Country		
Telephone Number		Fax Number
E-Mail Address		

This location is a: Home

Copyright © 1999 New Wave Software, Inc. All Rights Reserved. Pat. Pend.

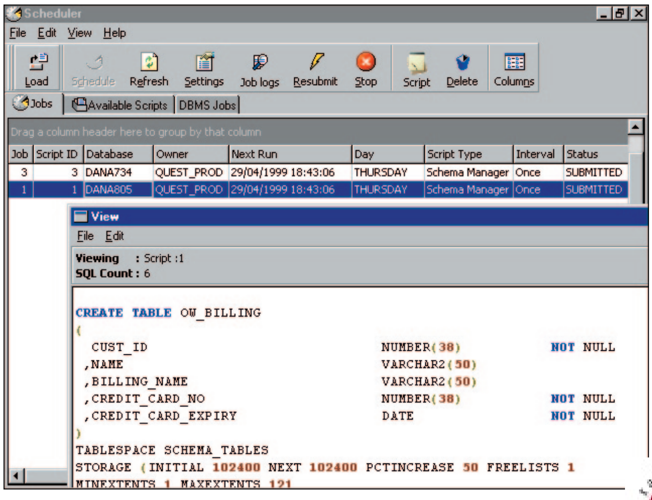
New Wave Software, Inc.
Price: Call or visit Web site for pricing information.
Phone: (800) 920-9283
Web Site: http://www.nwspi.com

Quest Announces Schema Manager 3.0 and Data Manager 3.0

Quest Software, Inc. announced *Schema Manager 3.0* and *Data Manager 3.0*. Schema Manager is a comprehensive solution that allows users to create, track, and deploy schema changes throughout an application lifecycle. A new feature quickly compares multiple schemas and database objects for large applications and ERP systems, such as Oracle applications, eliminating guesswork when applying patches.

Data Manager allows users to accurately move and subset data between databases for testing and reporting. Important new features include support for SQLoader, an import/export method available for data migrations, and support for SQL Server, enabling DBAs to migrate Oracle data to SQL Server for simplified data manipulation.

Quest Software, Inc.
Price: Call for pricing.
Phone: (949) 754-8000
Web Site: http://www.quest.com



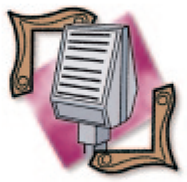
Job	Script ID	Database	Owner	Next Run	Day	Script Type	Interval	Status
3	3	DANA734	QUEST_PROD	29/04/1999 18:43:06	THURSDAY	Schema Manager	Once	SUBMITTED
1	1	DANA805	QUEST_PROD	29/04/1999 18:43:06	THURSDAY	Schema Manager	Once	SUBMITTED

Viewing : Script :1
SQL Count : 6

```
CREATE TABLE OW_BILLING
(
  CUST_ID                                    NUMBER(38)          NOT NULL
 ,NAME                                       VARCHAR2(50)
 ,BILLING_NAME                               VARCHAR2(50)
 ,CREDIT_CARD_NO                             NUMBER(38)          NOT NULL
 ,CREDIT_CARD_EXPIRY                         DATE                NOT NULL
)
TABLESPACE SCHEMA_TABLES
STORAGE (INITIAL 102400 NEXT 102400 PCTINCREASE 50 FREELISTS 1
MINEXTENTS 1 MAXEXTENTS 121
```

News LINE

May 2000



Informant Communications Group Launches ComputerBookstore.com

Elk Grove, CA — Informant Communications Group, Inc. (ICG) announced the opening of ComputerBookstore.com, a full-service technical bookstore catering to IT professionals. ComputerBookstore.com features a wide variety of computing, gaming, certification, training, science, programming, and business books from all the major book publishers, including IDG, Macmillan, SYBEX, Osborne/-McGraw-Hill, Coriolis Group, Wrox Press, Warner, Microsoft Press, John Wiley, Addison-Wesley, Prentice Hall, and Random House. In addition, ComputerBookstore.com will also make it easier to locate many hard-to-find titles from smaller, independent book publishers, such as Wordware, Manning, and Informant Press.

"Computer professionals and novices alike are looking for a comprehensive, competitively-priced source for IT-related books and training materials," said Mitchell Koulouris, president and CEO of Informant Communications Group. "ComputerBookstore.com is the perfect resource for experts in information technology, programming, database development, and related technologies and industries."

In addition to the vast selection of books, training materials, videos, and documentation, ComputerBookstore.com guarantees a savings of up to 41percent off the suggested retail price for featured on-sale items. ComputerBookstore.com offers a focused shopping experience that places the emphasis on technical books and training materials without the noise and distraction of non-related items. "Shoppers on the World Wide Web have become increasingly selective about the places they shop," said Koulouris. "ComputerBookstore.com is the best place to shop on the Web for technical books and related materials whether you're a CIO at a major corporation or a beginner just getting up to speed on your home computer."

For more information, visit http://www.ComputerBookstore.com.

Corel and Inprise Merger to Create Linux Powerhouse

New York, NY — Corel Corp. and Inprise/Borland Corp. announced they've entered into a definitive merger agreement. Upon completion of the merger, the combined organization, called Corel, will be a Linux powerhouse, offering a single source for end-to-end solutions, featuring a range of productivity applications, development tools, and professional services for all major platforms. The valuation for the entire transaction is approximately US$2.44 billion.

In 1999, the two companies had total revenues of US$418 million and currently have over US$200 million in cash. The merger will be accounted for as a purchase transaction under Canadian GAAP and is expected to be accretive to Corel's cash earnings per share before the amortization of goodwill.

Upon completion of the merger, Inprise/Borland will operate as a wholly-owned subsidiary of Corel. Dr Michael Cowpland will remain as president, CEO, and a director of the corporation. Dale Fuller, Inprise/Borland's interim president and CEO, will be appointed as chairman of Corel's board of directors. The operations of the combined entity will be headquartered in Ottawa, with the Inprise/Borland operations remaining in its current Silicon Valley locations. The combined businesses will have a presence in over 100 countries.

Under the terms of the agreement, Inprise/Borland shareholders will receive 0.747 Corel common shares for each share of Inprise/Borland common stock held. As a result of the merger, Corel expects to issue approximately 53.7 million common shares in the aggregate, in exchange for Inprise/Borland's outstanding shares.

Based on the closing price of US$20.00 per share of Corel as of February 4, 2000, this represents a value of US$14.94 per share of Inprise/Borland, giving a US$2.44 billion valuation for the entire transaction, on a fully diluted basis. Upon closing of the transaction, Inprise/Borland shareholders will own approximately 44 percent of Corel, with the balance being held by Corel's current shareholders. The boards of directors of both companies have approved the transaction.

The merger will offer training, education, and migration paths so customers can fully exploit the power of Linux-based Internet solutions. Both companies will also continue to provide support for open standards, ensuring compatibility across Linux, Windows, and Solaris platforms and applications.

Corel's work on the Linux operating system grew out of its earlier efforts developing software for the UNIX operating system. With the release of WordPerfect 8 for Linux in December 1998, Corel established itself as a software developer for the open-source operating system. Corel also developed the first Linux operating system (OS) built specifically for the desktop.

Inprise/Borland provides tools to create enterprise applications for the Linux operating system. Most recently, Inprise/Borland announced a free download of JBuilder 3 Foundation, a pure Java development environment for Linux; Kylix, planned to be one of the first rapid application development (RAD) tools for the Linux platform, scheduled to be available in mid-2000; and a free download of the Linux Just-In-Time (JIT) compiler.

The merger is subject to certain customary conditions, including shareholder approval from Inprise/Borland and Corel, compliance with the Hart-Scott-Rodino Antitrust Improvements Act, and certain other regulatory filings and approvals. The transaction is expected to close in the late spring and is expected to be tax-free to Inprise/Borland shareholders.

For more information, visit http://www.inprise.com or http://www.corel.com.

Delphi 5 Update Pack Available

An Update Pack for Delphi 5 is now available from the Inprise/-Borland Web site at http://www.inprise.com/devsupport/delphi/downloads/index.html#Delphi 5. Customers have other options to get the Delphi 5 Update Pack. US customers can obtain the update CD by calling (800) 457-9527. US customers can also order the CD online at http://shop.borland.com/shop/proddtl/0,1070,1-523,00.html. The Delphi 5 Update Pack CD will be shipped for US$6.95 (shipping charge) beginning at the end of March. The update CD contains all updates, including the ADOExpress Update and the most recent BDE update, BDE 5.1.1.

To order the Delphi 5 Update Pack CD outside the US, please contact the Inprise office that serves your country.

table of CONTENTS

By Richard Phillips

Parsing the Web

Three Classes for Grabbing HTML/XML Information

I recently bought one of the new digital satellite dishes and ran across an interesting challenge — figuring out just what was on, and when. DirectTV provided a Web site with a basic search engine, but the Web-based search was slow and very narrowly focused. As a typical programmer, I knew I could provide a better, more powerful UI, if I could just figure out how their Web search engine worked.

A quick scan of the HTML source pointed out a relatively simple search form that I could easily duplicate, but the HTML results came back in a mildly complicated HTML table. Brute force code would have been simple enough to construct to parse through the table, but I'd been looking for a reason to build a more general parser, so off I went. If I'd known just how lax the HTML rules are, and just how many hacks there are, I'd have just stuck with the brute force method and saved myself a lot of agony, but since I'm here now ...

The Basics

To put together a parser for HTML, an understanding of the rules is required. HTML originated as an SGML-like syntax, and over time has grown to fit more closely within the confines of SGML. These days the syntax is described within an SGML Data Type Definition (DTD), bringing it into a reasonably well-understood and managed domain. Given that SGML now establishes the underpinnings of HTML, the parser should apply the SGML syntax rules as a starting point. This also allows for the consideration of some simple extensions that allow parsing of XML.

Therefore, the parser is built to work on SGML in general, with specific handlers for exceptions and extensions that occur in HTML and XML. The rules for SGML are straightforward, and provide five basic constructs that we care about:

- elements,
- attributes,
- comments,
- SGML directives, and
- "everything else".

Elements are the facet of the SGML content with which we are most concerned, and around which the parser is established. Elements have a start tag, content, and an end tag, for example:

```
<TITLE>HTML Parsing</TITLE>
```

where `TITLE` is considered to be the "name" of the tag. Element names are case-insensitive. So we start with the following parsing rules:

- Element start and end tags are surrounded by `<` and `>` characters.
- Element end tags are denoted by the `/` character immediately following a `<`.
- Element content is surrounded by start and end tags.

HTML Extensions

In HTML, we immediately note that there are exceptions to these rules. For some elements — most notably <P> — the end tags may be omitted, even though the element may have contents. This offers perhaps the most annoying challenge of the HTML parsing rule set, because there are several methods by which the element may be terminated.

To start with, we note another syntax rule from SGML: elements may not span. That is, if an element's start tag is contained within another element's start and end tags, its end tag must also appear there. Put simply, if we encounter an end tag, all omitted end tags are considered "closed" back up to the matching start tag. Also, by observation (I couldn't find a formal rule for this in the HTML specification), virtually all elements with optional end tags close when they encounter another of themselves, <LI> and <OPTION> being fine examples. Further, the HTML reference material does indicate that <P> elements that omit their end tags are terminated by "block elements." The HTML DTD must be consulted to determine which elements are considered block elements. Unfortunately, all of this prevents us from using a general rule, and requires that we become concerned with the HTML DTD.

A quick consideration of the DTD is therefore in order. The DTD calls out which elements must have end tags, and which may omit (or are forbidden to have) end tags. For example, the DTD fragment for <P> is:

```
<!ELEMENT P - O (%inline;)* >
```

The important thing to note here is `- O`. The `-` indicates the start tag is required, and `O` means the end tag may be omitted. Compare this to the fragment for <BR>:

```
<!ELEMENT BR - O EMPTY >
```

where `- O EMPTY` indicates that the start tag is required. Since the element is `EMPTY`, however, the optional end tag is now expressly forbidden. In the fragment for <P>, the `(%inline;)*` shows the legal contents of the P element. In this case, `%inline;` refers to a list of elements defined earlier in the DTD. Notably missing from the `%inline;` list is <P> itself. A perusal of the other ambiguous elements reinforces the observation that in general, an element may not immediately contain itself (although this ambitiously general rule is certainly not guaranteed to remain valid for future releases of the HTML DTD). In the same way that `%inline;` is defined, so there exists a list name `%block;`, which contains the list of block elements.

This leads to another set of parsing rules:

- <P> with an omitted end tag is terminated when a block element is encountered.
- Elements are terminated when another element of the same name is encountered.
- Elements are terminated if a parent's end tag is encountered; no spans are allowed.

Attributes

Attributes represent the various properties of elements. By definition, attributes appear in name/value pairs within the start tag of the element. For example, in:

```
<BODY bgcolor="#FFFFFF">
```

the `BODY` element has an attribute name `bgcolor`, and the attribute has a value of `#FFFFFF`. Double quotation marks or single quotation marks are required to delimit the value, unless the value contains only letters, digits, hyphens, and periods. If the value contains single quotation marks, it should be delimited with double quotation marks, and vice versa. Attribute names are case-insensitive. Also worth noting is that not all attributes have a value. For instance, the `NOWRAP` attribute of the <TD> element.

- Attributes appear within an element's start tag
- Attributes are delimited by a space character (ASCII 32)
- Attribute values are delimited by `"`or `'`

Comments

SGML also provides that its content may include comments. Comments are of the form:

```
<!-- This is a comment -->
```

The `<!` is a markup declaration open delimiter, and indicates that an SGML directive is to follow. Comments are specifically denoted by `--` following the open delimiter (white space is not allowed between the `<!` and the `--`, but is allowed between the closing `--` and >). Further, a comment may contain < and > characters. Comments may not include other comments.

- `<!--` indicates a comment; `-->` terminates a comment.
- < and > are ignored while parsing a comment.

The remaining SGML directives are denoted by the beginning markup declaration open delimiter `<!`. To further complicate things, comments may exist within the directives delimited by `--`.

- `<!` denotes an SGML directive (if it's not a comment)
- Comments within the directives are delimited by `--`

Lastly, we consider what remains. Content of elements not contained within a start tag, end tag, or comment is considered by the parser to be PCData (parsed character data).

- Store text not included in element start/end tags or comments as PCData.

XML Extensions

As mentioned before, XML is also derived from SGML. While HTML is basically a DTD described within and using SGML, XML is a subset of SGML capable both of representing data and containing other DTDs of its own. XML also demonstrates that those working on the standards in the programming community actually do learn from the mistakes of those that went before them. For instance, the rules of containment are much more formal in XML than they are in HTML, making parsing a great deal simpler. This means that while a DTD may be included in an XML document for syntax checking purposes, it isn't necessarily required for the actual parsing of the XML content, as it is for HTML.

Knowing this, we can add two more rules and provide DTD-less XML parsing as well. For one, empty elements in HTML are simply called out as such in the DTD with their end tags forbidden (<BR> for example). If an element in XML is to be empty (that is, it will have no content), its start tag may have a `/` just before the closing `>`, indicating that no content and no end tag will follow. Additionally, XML directives may appear with the `?` character rather than `!`.

- Empty elements in XML may be terminated by a `/` just before the `>` in the start tag, e.g. `<partno/>`.
- Additional directives appear using `?`, instead of the `!` character.

Everything Else

Any items encountered in the content that are not contained in element start or end tags, comments, or DTD items are considered by the parser to be PCData. The content of elements fits this bill, as do carriage returns and line feeds encountered by the parser. This leads to the final parsing rule:

```
<HTML>
<HEAD>
<TITLE>Example HTML</TITLE>
</HEAD>
<BODY>
<!-- Insert non-sensical comment here -->
<H1>Example HTML</H1>
Plain old text right out there in the middle of the document.
<P>Text contained within in paragraph</P>
<B>Unordered List</B>
<UL>
<LI>Item #1
<LI>Item #2
<LI>Item #3
</UL>
</BODY>
</HTML>
```

Figure 1: Sample HTML.

- Any content located outside of start/end tags, comments, or DTD items is PCData.

Additional Considerations

It is also worth noting that syntax errors and occurrences of "browser-tolerated" HTML inconsistencies are frequently encountered, and as such, should not raise exceptions except in extreme cases. Instead, a warning should be noted and parsing should continue if possible.

The Parser

The goal of parsing the SGML content is to place the data it represents into a form more readily accessible to other components. Because SGML calls out a hierarchical structure, a hierarchy is probably the most accurate way to store the parsed content. With that in mind, the parser is built from two primary classes, and a third supporting class:

- First and foremost is the *THTMLParser* class. Its *Parse* method accepts the content to be parsed, and places the processed results in the *Tree* property.
- Next is the *TTagNode* class in which the parsed results are contained. This class is a hierarchical storage container with *Parent* pointing to the *TTagNode* that contains the current node, and *Children* containing a list of children immediately owned by the current node.
- *TTagNodeList* is provided as a list container for a collection of *TTagNode* objects, typically produced by a call to the *GetTags* method of the *TTagNode* class.

A Simple Example

Consider the sample HTML shown in Figure 1. The parser would produce from the HTML a hierarchy that can be visualized as in Figure 2. Each of the boxes in the tree represents a *TTagNode* instance.

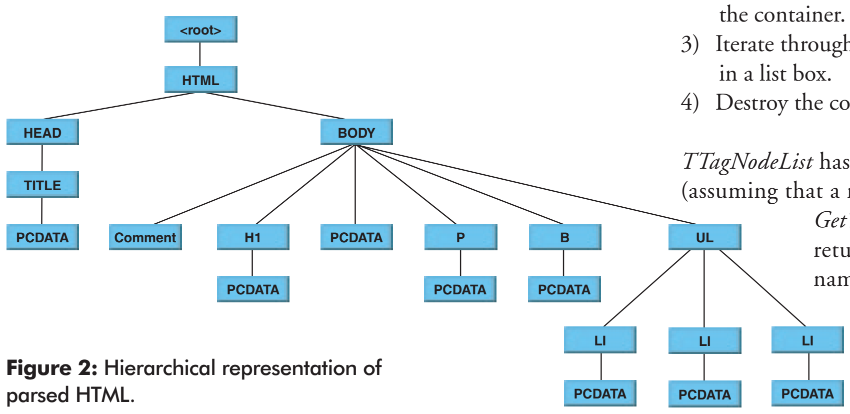


Figure 2: Hierarchical representation of parsed HTML.

Node Contents	*NodeType*	Caption
HTML elements	*nteElement*	Element name
Text	*ntePCData*	
Comments	*nteComment*	!
SGML/XML/DTD directives	*nteDTDItem*	! or ? and directive

Figure 3: *TTagNode* node types.

```
procedure Button1OnClick(Sender: TObject);
var
  Elements : TTagNodeList;
  Counter : Integer;
begin
  Elements := TTagNodeList.Create;
  HTMLParser1.Tree.GetTags('LI',Elements);
  for Counter := 0 to Elements.Count - 1 do
    ListBox1.Items.Add(Elements[Counter].GetPCData);
  Elements.Free;
end;
```

Figure 4: Using *GetTags*.

Each node has a *NodeType* property that indicates what type of node it is. All node types except *ntePCData* also have text in the *Caption* property that provides more information about the node's contents. See the table in Figure 3 for details.

For example, the HTML node in Figure 2 has a *NodeType* of *nteElement*, while the comment tag is of type *nteComment*, and the PCData nodes are of type *ntePCData*.

The content or text for each HTML element is contained in a node in its *Children* list. For example, the TITLE node in Figure 2 has a PCData node whose *Text* property contains "Example HTML". The *GetPCData* method of a *TTagNode* returns the PCData text for all children of the node for which it's called. Note, this method is recursive and will return the PCData text for all nodes in the tree beneath the node upon which it's called.

Retrieving Elements

The *GetTags* method of a *TTagNode* will return a list of all children that match an element name. If `'*'` or `''` is specified as the element name, then all children will be returned. Note that this method is recursive. The result list is a *TTagNodeList*.

The code in Figure 4 illustrates how the *GetTags* method is used to collect a list of all <LI> elements in the HTML from Figure 1 and insert their contents into a list box. The process is as follows:

1) Create a container for the results, i.e. the *Elements* list.
2) Call the *GetTags* method, passing the desired element name and the container.
3) Iterate through the container, placing the text for each element in a list box.
4) Destroy the container.

TTagNodeList has two methods that offer another approach (assuming that a result set has already been acquired): *GetTagCount* and *FindTagByIndex*. *GetTagCount* returns a count of the occurrences of an element name. *FindTagByIndex* returns the index within the list of the specified occurrence of an element name. For instance, the statement:

```
ShowMessage(Elements.FindTagByIndex('li',1).GetPCData);
```

were it included in Figure 4, would display the text for the second occurrence of the <LI> element in the *Elements* container. This can prove exceptionally useful for locating a specific tag from target HTML content. For example, if the third <TABLE> in the HTML contained the desired data, the following code would make quick work of locating the root <TABLE> node:

```
HTMLParser1.GetTags('*',Elements);
Node := Elements.FindTagByIndex('table',2);
if Assigned(Node) then
  begin
    // Perform processing on <TABLE> node.
  end;
```

Working with Results as a Hierarchy

The procedure in Figure 5 provides yet another method for accessing the contents of the *Tree* (albeit a more brute force approach). In this example, the HTML content is assumed to be fairly straightforward: <TABLE> elements contain <TR> elements, which contain <TH> and <TD> elements. Given this reasonably accurate assumption, the code will walk the children of a <TABLE> node, and for each <TR> node found will walk its children looking for occurrences of either <TH> or <TR>, and add their text (contained in PCData nodes) to a *TStrings* container, e.g. the *Lines* property of a *TMemo* control.

As an illustration of just how difficult HTML processing can be, the following caveats apply to the code provided in Figure 5 and would have to be handled to provide a robust solution:

- Subtables are not handled. That is, <TABLE> elements encountered within a <TD> element are ignored.
- Row and column spanning is not handled.
- <TBODY>, <THEAD> and a host of other table elements are not considered (although admittedly they are rare).

Working with Results as a List

The procedure in Figure 6 demonstrates working with the parsed results as a list. The goal here is to retrieve a list of all comments, links, meta tags, and images from the document, with the <TITLE> thrown in for good measure. The code does this in several simple steps:

1) Parse the HTML.
2) Create a container for the list of matching nodes from the tree.
3) Call the *GetTags* method passing `'*'`, and the container (`'*'` indicates that all items in the tree should be returned).
4) Iterate through the container collecting matches and place their contents in the StringList.
5) Destroy the container.

The important thing to understand here is that the *TTagNodeList* class is just a list of pointers to nodes from the *Tree*. This is quite beneficial in that once a desired node is located in the list, it may be used as if it had been acquired by traversing the tree. For example, when the **case** statement in Figure 6 encounters a TITLE element,

```
// Return TableNode's contents in a TStrings container.
procedure GetTable(TableNode: TTagNode; Lines : TStrings);
var
  RowCtr,
  DataCtr : Integer;
  Node,
  RowNode : TTagNode;
  TempStr : string;
begin
  Lines.Clear;
  if CompareText(TableNode.Caption,'table') = 0 then begin
    for RowCtr := 0 to TableNode.ChildCount - 1 do begin
      RowNode := TableNode.Children[RowCtr];
      if CompareText(RowNode.Caption,'tr') = 0 then begin
        TempStr := '';
        for DataCtr := 0 to RowNode.ChildCount - 1 do begin
          Node := RowNode.Children[DataCtr];
          if CompareText(Node.Caption,'td') = 0 then
            TempStr := TempStr + Node.GetPCData + #9
          else
            if CompareText(Node.Caption,'th') = 0 then
              TempStr := TempStr + Node.GetPCData + #9;
        end;
        TempStr := Trim(TempStr);
        if TempStr <> '' then
          Lines.Add(TempStr);
      end;
    end;
  end;
end;
```

Figure 5: Working with results as a hierarchy.

```
procedure TForm1.Parse(HTML: string; Lines: TStrings);
const
  cKnownTags = '|title|img  |a    |meta |!    ';
  cTITLE = 0 ;
  cIMG = 1;
  cA = 2;
  cMETA = 3;
  cComment = 4;
var
  Index,
  Counter : Integer;
  TempStr : string;
  Nodes : TTagNodeList;
begin
  HTMLParser1.Parse(HTML);
  Nodes := TTagNodeList.Create;
  // Retrieve all nodes.
  HTMLParser1.Tree.GetTags('*',Nodes);
  for Counter := 0 to Nodes.Count - 1 do begin
    TempStr := '|' + LowerCase(Nodes[Counter].Caption);
    // Index of element name.
    Index := Pos(TempStr,cKnownTags);
    if Index > 0 then begin
      Index := Index div 6;
      case Index of
        cTITLE :
          Lines.Add('Title=' +
            HTMLDecode(Nodes[Counter].GetPCData));
        cIMG :
          begin
            TempStr := Nodes[Counter].Params.Values['src'];
            if TempStr <> '' then
              Lines.Add(
                Nodes[Counter].Params.Values['src']);
          end;
        cA :
          begin
            TempStr :=
              Nodes[Counter].Params.Values['href'];
            if TempStr <> '' then
              Lines.Add(TempStr + '=' +
                HTMLDecode(Nodes[Counter].GetPCData));
          end;
        cMETA :
          with Nodes[Counter].Params do
            Lines.Add(Values['name'] + '=' +
              Values['content']);
        cComment :
          Lines.Add('[Comment] ' +
            HTMLDecode(Nodes[Counter].Text));
      end;  { case Index }
    end;  { if Index > 0 }
  end;  { for Counter := 0 to Nodes.Count - 1 }
  Nodes.Free;
end;
```

Figure 6: Working with results as a list.

its contents are retrieved by making a call to the TITLE node's *GetPCData* method (which depends on the parsed tree structure behaving as it appears to in Figure 2). Note that the PCData often contains encoded items such as `&gt;` and `&lt;` (< and > respectively). *HTMLDecode* is provided for handling most simple cases, but doesn't handle all cases (notably non-US character encoding).

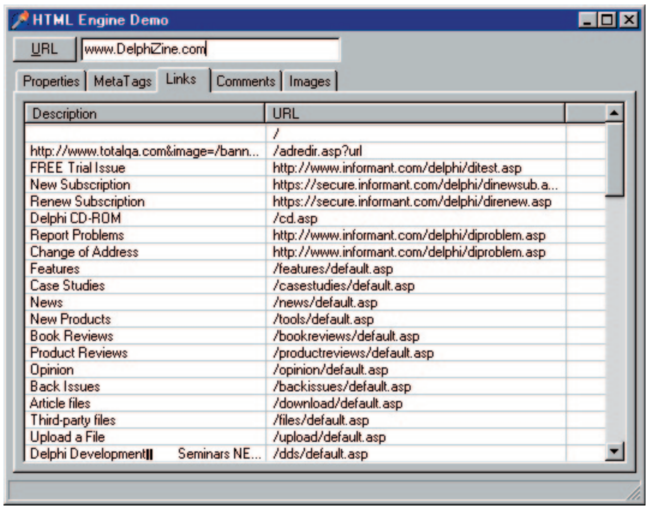
This example also demonstrates the use of the attributes from an element. When an <A> element is encountered, the HREF attribute is examined. If it exists, the <A> element is treated as a link to some other resource. If the HREF attribute were not specified, this might be an instance of <A> serving as an anchor instead of a link. For more details on how the *Params* property of the *TTagNode* behaves, see the Delphi help for the *Names* and *Values* properties of the *TStrings* class.

Searching the www.directv.com Program Guide

Applying the HTML parser to the original need turns out to be another simple — albeit involved — exercise (see Listing One beginning on page 11). First, an understanding of the CGI scripts that allow searching of the program guide is required. The search script is rather crude and accepts only three parameters: *timezone*, *category*, and *search text*. *timezone* is simply a number representing Eastern, Central, Mountain, or Pacific. *category* allows the search to span all programs, or to be narrowed to certain types of programs. The *search text* should be all, or a portion of, the desired program name. The search is specific to program names, and doesn't consider program descriptions.

The results of the search are returned as an HTML table including channel, date, time, duration, and program name. The program name is contained within a link to the program description, which will need to be retrieved as well. This is slightly complicated by the fact that the link provided is written using JavaScript, which we cannot simply call. However, the URL produced by the JavaScript function is easy to replicate, as it contains a program ID number that can be passed to another CGI script that returns the desired description.

The next step is to parse the HTML and process the results into a more useful format. *TStringTable* is provided as a simple container for just this purpose. The *TStringTable* offers a non-visual equivalent to *TStringGrid* with a few additional methods to make manipulating the data a bit easier. Once the HTML table has been processed and placed in the string table, a bit of house cleaning is required. For one, there are rows at the end of the HTML table that need to be ignored, as



Description	URL
	/
http://www.totalqa.com&image=/bann...	/adredir.asp?url
FREE Trial Issue	http://www.informant.com/delphi/ditest.asp
New Subscription	https://secure.informant.com/delphi/dinewsub.a...
Renew Subscription	https://secure.informant.com/delphi/direnew.asp
Delphi CD-ROM	/cd.asp
Report Problems	http://www.informant.com/delphi/diproblem.asp
Change of Address	http://www.informant.com/delphi/diproblem.asp
Features	/features/default.asp
Case Studies	/casestudies/default.asp
News	/news/default.asp
New Products	/tools/default.asp
Book Reviews	/bookreviews/default.asp
Product Reviews	/productreviews/default.asp
Opinion	/opinion/default.asp
Back Issues	/backissues/default.asp
Article files	/download/default.asp
Third-party files	/files/default.asp
Upload a File	/upload/default.asp
Delphi Development▌▌ Seminars NE...	/dds/default.asp

Figure 7: The HTML Engine Demo application.

they contain images, not program content. Also, the channel appears only in the first row of a set of programs that occur on that channel.

The contents can now be added to a ListView. Once that task is complete, the descriptions can be fetched using the program IDs to call the description CGI, and then added to the ListView.

Further Study

While these examples are not terribly glamorous, the parser can be applied to more meaningful problems. For instance, a friend of mine has put together an extremely handy application using the Pricewatch site (http://www.pricewatch.com) to monitor prices on PC hardware. Pricewatch offers a current snapshot of pricing of a particular piece of hardware from various vendors, usually sorted from least expensive to most. However, it doesn't allow for viewing of several different pieces of hardware at once, and it doesn't track the history of the price changes for the hardware. So, the application provides a way to build a list of hardware to be tracked, and then offers a simple trend analysis by gathering and saving off the price information on a regular basis. This provides a useful picture for the consumer of just how quickly the prices are moving downward on a particular item. If the pricing is in a steep downward curve, waiting to purchase might be wise. If the curve is flat, the time to purchase might be at hand.

The parser is used not only to retrieve the pricing data, but also to help deal with one of the more significant issues facing those attempting to interface with sites they do not control: unexpected changes in the target site's contents. In this case, a review of the <form> elements from the main page is performed to ensure that the query mechanism remains intact. As a further safeguard, the search results page is also examined to verify a match against the expected HTML format. If unexpected items are found in either case, processing cannot continue, but at least the user can be warned of the situation.

In a more interesting demonstration of the parser's abilities, it has been combined with a database to create a poor man's OODB (object-oriented database). XML is used to wrap the data, and is then stored in text fields in the database. When needed, the XML is retrieved and the parser used to extract the data. Without going in to detail, this is useful because the data stored in the database can carry semantic information with it (the XML elements) that provides information about the data's structure. In systems where the data structure is dynamic, this provides a simple way to avoid excessive database maintenance and further provides a clean, easily understood mechanism for the exchange of data between various applications and platforms. In the case mentioned here, a legacy defect (bug) tracking system hosted on a Solaris platform was wrapped in a Delphi-based UI.

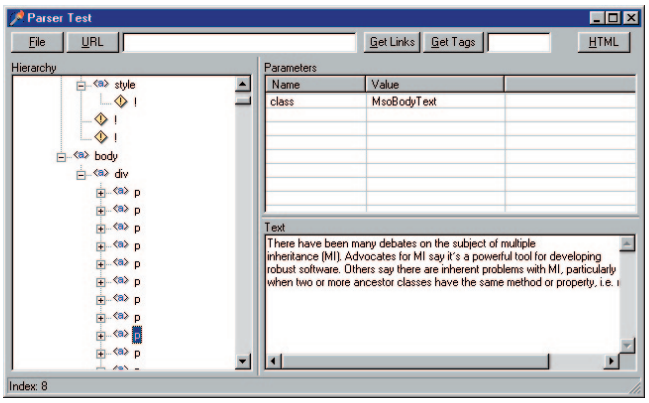
Additional Demonstration Applications

To further demonstrate the power of the classes presented in this article, two additional applications accompany this article. An HTML Engine Demo application (see Figure 7) displays a great deal of information about any selected URL, including meta tags, links, and images.

The Parser Test application parses any URL, or SGML/HTML/XML document, and displays the results in a TreeView (see Figure 8). It can also display links, selected tags, text, etc.

Room for Improvement

This parser builds a reasonable basis for parsing of HTML and XML, but offers significant room for further development.



Name	Value
class	MsoBodyText

Text

There have been many debates on the subject of multiple
inheritance (MI). Advocates for MI say it's a powerful tool for developing
robust software. Others say there are inherent problems with MI, particularly
when two or more ancestor classes have the same method or property, i.e. i

Figure 8: The Parser Test application.

Incorporation of a DTD processor (DTDs can be parsed with the existing parser, but no handling of the parsed contents is provided). This would provide two main benefits: more thorough parsing of elements based on a true understanding of their legal contents, and no need for hard coding a representation of the HTML DTD within the parser. Further, DTD-based XML parsing would then be possible.

A DOM container model to complement the *TTagNode* model. DOM represents a fairly well understood and commonly encountered model for representing the parsed contents of XML. While it doesn't suit all needs, it does provide a useful, standard way to communicate about the parsed elements.

XQL or other suitable extended model query mechanism. The *GetTags* method is reasonably sufficient, but for more exhaustive queries against XML contents, a more advanced mechanism is desired. For example, it would be extremely handy if *GetTags* could be passed `'order/partno'` to indicate that we're searching for all PARTNO elements that are immediately below an ORDER element.

Resources and Alternatives

HTML 4.0 Specification — http://www.w3.org/TR/REC-html40. This is the single most useful resource to those seeking HTML enlightenment. It is extremely detailed and well written.

HTML 4.0 Loose DTD — http://www.w3.org/TR/REC-html40/loose.dtd. A part of the HTML 4.0 specification, this offers the exact specification of just what the HTML rules are. It is upon this DTD (as opposed to the "strict" DTD) that the parser is designed to operate.

"XML: Creating Structures of Meaning," *Visual Developer*, Nov/Dec 1998, Vol. 9 No. 4. A quick survey of XML for the beginner. Syntax and use are explored here with an eye to bringing the novice on board.

"Using Internet Explorer's HTML Parser," *Dr. Dobb's Journal*, #302, August 1999 (http://www.ddj.com/articles/1999/9908/9908toc.htm). This article offers an examination of using the HTML parser that is available within Microsoft's Internet Explorer via COM interface. The source for the article is in C++, but it's not difficult to follow.

"XML from Delphi," *Delphi Informant Magazine*, July 1999, Vol. 5 No. 7. A beginner's explanation of XML, and an excellent example of using the XML parser included in Internet Explorer 4.0.

Further performance tuning. While some attention has been paid to this area, no extreme efforts to speed things up were applied. Most notably, Delphi's string routines are not considered to be as fast as those provided in some third-party string-handling collections (notably HyperString). Δ

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD in INFORM\00\MAY\DI200005RP.

Richard Phillips is a Development Manager for i2 Technologies working on their TradeMatrix technology integrating disparate datasources. He's been using Delphi since it was introduced and Borland Pascal since 1985. He can be reached at richardp@dallas.net.

Begin Listing One — Searching www.directv.com

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls, Forms,
  Dialogs, ComCtrls, StdCtrls, ExtCtrls, GetURL, HTMLMisc,
  HTMLParser, StringTable;

type
  TForm1 = class(TForm)
    WIGetURL1: TWIGetURL;
    StatusBar1: TStatusBar;
    Panel2: TPanel;
    pbSearch: TButton;
    ebSearchText: TEdit;
    Panel3: TPanel;
    ListView1: TListView;
    HTMLParser1: THTMLParser;
    procedure pbSearchClick(Sender: TObject);
    procedure WIGetURL1Status(Sender: TObject;
      Status: Integer; StatusInformation: Pointer;
      StatusInformationLength: Integer);
  private
    procedure GetTable(TableNode: TTagNode;
      Table : TStringTable);
    function GetDescription(Node: TTagNode): string;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

const
  cDirecTVSearchURL =
    'http:// 206.17.88.15/cgi-bin/pgm_search.cgi/';
  cDirecTVDescURL =
    'http:// 206.17.88.15/cgi-bin/pgm_desc.cgi/';

// Return TableNode's contents in Table
procedure TForm1.GetTable(TableNode: TTagNode;
  Table: TStringTable);
var
  RowCtr,
  DataCtr : Integer;
  Node,
  RowNode : TTagNode;
begin
  Table.Clear;
  if LowerCase(TableNode.Caption) = 'table' then begin
    for RowCtr := 0 to TableNode.ChildCount - 1 do begin
      RowNode := TableNode.Children[RowCtr];
      if LowerCase(RowNode.Caption) = 'tr' then begin
        Table.NewRow;
        for DataCtr := 0 to RowNode.ChildCount - 1 do begin
          Node := RowNode.Children[DataCtr];
          if LowerCase(Node.Caption) = 'td' then
            Table.AddColumnObject(Node.GetPCData,Node)
          else
            if LowerCase(Node.Caption) = 'th' then
              Table.AddHeader(Node.GetPCData);
        end;
        if Table.Row[Table.RowCount - 1].Count <= 0 then
          Table.DeleteRow(Table.RowCount - 1);
      end;
    end;
  end;
end;

function TForm1.GetDescription(Node : TTagNode) : string;
var
  TempStr : string;
begin
  Result := '';
  if Node.ChildCount > 0 then
    TempStr := Node.Children[0].Params.Values['href']
  else
    TempStr := '';

  if TempStr <> '' then begin
    // Parse out the description id
    Delete(TempStr,1,Pos('(',TempStr));
    Delete(TempStr,Pos(')',TempStr),Length(TempStr));
    WIGetURL1.URL := cDirecTVDescURL + TempStr;
    Screen.Cursor := crHourglass;
    Application.ProcessMessages;
    WIGetURL1.GetURL;
    Screen.Cursor := crDefault;
    StatusBar1.Panels.Items[0].Text := '';
    if WIGetURL1.Status = wiSuccess then begin
      TempStr := WIGetURL1.Text;
      // Use brute force to scrape out program description.
      if Pos('<BLOCKQUOTE>',TempStr) > 0 then begin
        Delete(TempStr,1,Pos('<BLOCKQUOTE>',TempStr) + 11);
        Delete(TempStr,Pos('</BLOCKQUOTE>',TempStr),
               Length(TempStr));
      end;
      Result := TempStr;
    end;
  end
end;

procedure TForm1.pbSearchClick(Sender: TObject);
const
  tzPacific = '0';      // Time zones.
  tzMountain = '1';
  tzCentral = '2';
  tzEastern = '3';
  cgMovies = '0';       // Categories.
  cgSports = '1';
  cgSpecials = '2';
  cgSeries = '3';
  cgNews = '4';
  cgShopping = '5';
  cgAllCategories = '-1';
var
  Cols,
  Rows : Integer;
  NewItem : TListItem;
  Node : TTagNode;
  Nodes : TTagNodeList;
  ResultTable : TStringTable;
  TempStr : string;
begin
  if ebSearchText.Text = '' then
    Exit;
  WIGetURL1.URL := cDirecTVSearchURL + tzCentral + '/' +
    cgAllCategories + '/' + urlEncode(ebSearchText.Text);
  Screen.Cursor := crHourglass;
  Application.ProcessMessages;
  WIGetURL1.GetURL;
  Screen.Cursor := crDefault;
  StatusBar1.Panels.Items[0].Text := '';
  if WIGetURL1.Status = wiSuccess then
    begin
      if Pos('No program titles that match',
             WIGetURL1.Text) > 0 then
        ShowMessage('No matches found')
      else
        begin
          // Attempt to parse HTML table we're looking for.
          HTMLParser1.Parse(WIGetURL1.Text);
          Nodes := TTagNodeList.Create;
          HTMLParser1.Tree.GetTags('table',Nodes);
          if Nodes.Count > 0 then
            begin
              ResultTable := TStringTable.Create;
              GetTable(Nodes[0],ResultTable);
              // Get rid of image tags at bottom of search
              // response (the 2nd column has no contents).
              with ResultTable do
                for Rows := RowCount - 1 downto 0 do
                  if Cells[1,Rows] = '' then
                    DeleteRow(Rows);
              // Ensure all cells are filled appropriately
              // (in the HTML table, a RowSpan attribute
              // allows the "Channel" to be displayed in
              // one cell for several programs).
              with ResultTable do
                for Rows := 0 to RowCount - 1 do
                  for Cols := 0 to ColCount - 1 do
                    if Cells[Cols,Rows] = '' then
                      if Rows > 0 then
                        Cells[Cols,Rows] :=
                          Cells[Cols,Rows - 1];
              // Add items to ListView (Program, Channel,
              // Data, Time).
              ListView1.Items.Clear;
              for Rows := 0 to
                    ResultTable.RowCount - 1 do begin
                NewItem := ListView1.Items.Add;
                NewItem.Caption :=
                  ResultTable.Cells[4,Rows];
                NewItem.SubItems.Add(
                  ResultTable.Cells[0,Rows]);
                NewItem.SubItems.Add(
                  ResultTable.Cells[1,Rows]);
                NewItem.SubItems.Add(
                  ResultTable.Cells[2,Rows]);
              end;
              // Retrieve program descriptions (program id
              // contained in 4th column's node).
              for Rows := 0 to
                    ResultTable.RowCount - 1 do begin
                // It's rude to whack the server :-)
                Sleep(500);
                Node :=
                  TTagNode(ResultTable.Objects[4,Rows]);
                TempStr := GetDescription(Node);
                ListView1.Items[Rows].
                  SubItems.Add(TempStr);
              end;
              ResultTable.Free;
            end  // if Nodes.Count > 0 ...
          else
            ShowMessage(
              'Error - Expected table...found none');
        end;  // else of Pos('No program titles that...
    end  // WIGetURL1.Status = wiSuccess...
  else
    ShowMessage('Unable to contact search server [' +
                WIGetURL1.ErrorMessage + ']');
end;

end.
```

End Listing One

By Ron Loewy

Delphi in the Office

Writing Office 2000 Add-ins in Delphi

I don't have a "personal productivity" profiler on my machine, but if I had to guess, I would say that I probably spend most of my time in front of a monitor using these programs: Delphi for development, an Internet browser/e-mail for communication, and Microsoft Office for productivity — more specifically, Word for document writing, Excel for spreadsheets, and the occasional PowerPoint presentation.

When Office 2000 was released, it immediately caught my attention. It offered a unified Visual Basic for Applications (VBA) environment for its applications, Web documents, and even an assistant that I find useful rather than annoying (yes, I am talking about the paper clip). The new feature that really interests me, however, is the unified COM-based add-in architecture. With this architecture, I'm able to write enhancements and remedies (using Delphi) to the many things that irritate me about Office applications.

The problem with writing COM add-ins in Delphi 3 or 4 was that I had to create my own wrappers around the published COM interfaces. Worse than that was the prospect of having to use connection points and event sinks to interface with events published by OLE Servers. Because a useful add-in needs to be notified of events in Office applications, I've never looked forward to the task. As if they were reading my mind, the good people at Inprise released Delphi 5, which offers the great /L+ switch to the type library import utility, giving developers the ability to create object wrappers around OLE Servers. I have no more excuses.

The Office 2000 Add-in Architecture

Office 2000 is available in many editions. One of these is the Developer edition, and it includes the ability to create add-ins in Visual Basic (VB). As nice as VBA is in the latest Office release, VB hides a lot of the stuff behind closed doors, and writing a VB add-in doesn't teach you how COM is used in the new add-in architecture.

On Microsoft's Web site, I located Knowledge Base article Q230689: Office 2000 COM Add-In Written in Visual C++ (http://support.microsoft.com/download/support/mslfiles/COMADDIN.EXE). This downloadable file describes the COM interfaces that take part in an add-in construction. Inspecting the C++ code also reveals how we can use Delphi to write Office 2000 add-ins.

An Office 2000 add-in is a COM automation object that implements the *IDTExtensibility2* interface. This interface is remarkably simple, which explains its use in many different applications, such as Word, Excel, PowerPoint, and Outlook. The add-in must implement all five functions defined in the interface:

1) *OnConnection*. This function is called when the application connects to the add-in. The add-in receives initialization information in this call — including the pointer to the application's object model entry point, the connection mode (e.g. was the add-in started manually on application startup or via the command line?), a pointer to the object that represents the add-in in the application's object model, and user-defined information.
2) *OnDisconnection*. This function is called when the application disconnects from the add-in. This is the place the add-in uses to clean the resources it allocated, and remove the user-interface elements it added to the application.
3) *OnStartupComplete*. This function is called only if the add-in was automatically started by the application. When this function is called, all the other add-ins have been loaded into memory. If your add-in needs to communicate with them, it can. I like to use this event to add the user-interface elements to the application.
4) *OnBeginShutdown*. This function is called when the application is getting ready to shut down and will continue to disconnect from the add-in. At this point, the add-in needs to stop accepting user input.
5) *OnAddInsUpdate*. This function is called when the list of registered add-ins is changed. My guess is that if your add-in depends on another add-in(s), this function might be of some interest to you; otherwise, you will usually leave it empty.

```
// IDTExtensibility2 methods
procedure OnConnection(const Application: IDispatch;
  ConnectMode: ext_ConnectMode; const AddInInst: IDispatch;
  var custom: PSafeArray); safecall;
procedure OnDisconnection(RemoveMode: ext_DisconnectMode;
  var custom: PSafeArray); safecall;
procedure OnAddInsUpdate(var custom: PSafeArray); safecall;
procedure OnStartupComplete(var custom: PSafeArray);
  safecall;
procedure OnBeginShutdown(var custom: PSafeArray);
  safecall;
```

Figure 1: The declarations of the *IDTExtensibility2* methods.

Interfaces, Type Libraries, and Constants

To create the add-in, you'll need to import some COM objects and type libraries into Delphi. I used Delphi 5's TlibImp.exe (installed in the /Bin sub-directory of the standard Delphi installation directory) to import everything. The new version of this utility supports the new /L+ flag, which creates an OLE Server Delphi wrapper around COM objects and automatically maps their properties and events for easy Delphi use.

The *IDTExtensibility2* interface that our add-in needs to implement is declared in the file MSADDNDR.DLL, located in the \Program Files\Common Files\Designer\ directory.

I used TLIBIMP /L+ \Program Files\Common Files\Designer\MSADDNDR.DLL from the Imports sub-directory of Delphi's root directory. The result is the file AddInDesignerObjects_TLB.pas (and AddInDesignerObjects_TLB.dcr). We will need to use this file in the **uses** clause of our project to gain access to the interface. For some reason, TLIBIMP renamed the interface *_IDTExtensibility2* (notice the underline prefix). Experience taught me to accept this as a necessary evil. (I would suggest not fighting TLIBIMP for the underlines that it loves to add at the start and end of interfaces, properties, methods, or constants).

The next step is deciding which Office application (or applications) we want to create the add-in for. This article will use Word 2000 as the sample. There are plenty of articles, books, and documentation resources about the different Office application object models. Naturally, when you create an add-in for Outlook, Excel, or any other Office application, you'll need to access the object model for the particular application and import its type libraries.

I imported Word's type library from the file MSWORD9.OLB in the \Program Files\Microsoft Office\Office directory. Similarly, if you want to create an add-in for Excel, you will import EXCEL9.OLB; for Access you need to import MSACC9.OLB; and for Outlook MSOUTL9.OLB. TLIBIMP automatically imports the type library of the shared Office components (MSO9.DLL). The result are the files Office_TLB.pas and Word_TLB.pas.

Note that Borland supplied imports of the Office 97 files like Word97.dcu. The unified add-ins architecture doesn't work in Office 97 applications, and we have to perform the new import to gain access to the latest object models.

A Basic Add-in

A basic add-in implements *IDTExtensibility2* and doesn't do anything that actually interfaces with the host application. It's obviously a useless add-in, but we must start with such an add-in before we write our specific functionality.

Add-ins can be implemented as either in-process or out-of-process COM servers. For this article, I created an in-process server (available for download; see end of article for details). In Delphi, select **File | New**. On the ActiveX tab, double-click the ActiveX Library icon. Save the file in your development directory (I named it DIWordAddIn). Now, double-click on the Automation Object icon on the ActiveX tab. I named the class `AddIn` and saved the implementation unit as AddInMain.pas. In AddInMain.pas, I added AddinDesignerObjects_TLB, Word_TLB, and Office_TLB to the **uses** statement. Then, I added *IDTExtensibility2* to the class definition as one of the interfaces that the object implements. The class definition now reads:

```
type
  TAddIn = class(TAutoObject, IAddIn, IDTExtensibility2)
...
```

To the **protected** section of the class definition, I added the declarations of the *IDTExtensibility2* methods (see Figure 1).

You can use Delphi's Ctrl Shift C to complete the class definition, and add the method implementation bodies in the unit source. To test the add-in, add the following code to the *OnConnection* method:

```
ShowMessage('Connected to ' + WordApp.Name);
```

Add this code to the *OnDisconnection* method:

```
ShowMessage('Bye Bye');
```

The basic add-in is now ready to be compiled and registered with Word.

Registering an Add-in with an Office Application

Like any other COM object, an add-in needs to be registered with the system. By selecting **Run | Register ActiveX Server**, you can register the object with the COM run-time manager. In addition to the standard COM registration, you need to register the COM object with the Office application for which it was created. To do so, you need to create a new key in the registry. The key's name should be:

```
HKEY_CURRENT_USER\Software\Microsoft\Office\<AppName>\
  Addins\<AddInProgID>
```

where `<AppName>` is the name of the application (Word in our case) and `<AddInProgID>` is the name of the automation object. The automation object in our case is called DIWordAddIn.AddIn (the name of the ActiveX library and the name of the class).

We need to create several values under the following key:

```
HKEY_CURRENT_USER\Software\Microsoft\Office\Word\Addins\
  DIWordAddIn.AddIn
```

A DWORD value called *LoadBehavior* determines how the add-in is loaded and used in the application. In our case, we would set it to 3 — a combination of *Connected* and *BootLoad*. A list of available value combinations is shown in the table in Figure 2. The add-in is then connected to the application, and starts when the application is opened.

Additional values can appear in the registry key, like a friendly name that will be displayed in the application's COM manager dialog box, and an indication of whether the add-in can be activated from the command line.

The Office 2000 User Interface

A common set of objects that represents the user interface elements is shared among all the Office applications. Menu bars, toolbars, common controls (e.g. toolbar buttons and combo boxes), and even the often-maligned Office assistant exist throughout the Office suite.

When you imported the Word type library earlier, the common Office object's type library was also imported, and the Office_TLB.pas was created. Unfortunately, when Delphi imports this type library, it doesn't create OLE Server wrapper objects. We will have to manually create a Delphi wrapper for the *CommandBarButton* object exposed by Office. This object represents a simple menu item or toolbar button in Office.

Like most of Microsoft's applications, an *Application* object represents an entry point to the object model. Office *Application* classes provide access to the *CommandBars* property. This is a collection of *CommandBar* objects: toolbars, floating toolbars, or menus. The Office object model allows you to create or update existing *CommandBars*. Office_TLB.pas contains the *ICommandBar* interface, which represents such an object.

A *CommandBar* object has a *Controls* collection, which is a collection of *CommandBarControl* items. A *CommandBarControl* is an item that appears in a command bar; a simple *CommandBarButton* is used to represent a simple toolbar button (or menu item), but you can also create *CommandBarCombo* controls (combo boxes), *CommandBarPopup* (drop-down menus), and *CommandBarActiveX* (everything that wraps an ActiveX control).

If you check Office_TLB.pas, you'll notice that, in addition to an *ICommandBarButton* interface, an *ICommandBarButtonEvents* interface is implemented. As you can imagine, if our add-in wants to add toolbar buttons and menu items, we'll need to create new *CommandBarButtons* and implement the *Click* event defined in the events interface.

Connecting to a connectable object is done via connection points and sinking events (see Binh Ly's two-part "COM Callbacks" series in the June and July 1998 issues of *Delphi Informant Magazine*). Call me lazy, but if I can avoid sinking events, I jump at the opportunity. Fortunately, inspecting the code that Delphi created for *TWordApplication* in Word_TLB.pas can give us a simple template to create a Delphi wrapper for every connectable object that does the event sinking automatically.

The file BtnSvr.pas is a module that contains such a manually-created wrapper. In addition to properties of a *CommandBarButton* that I surfaced as standard Delphi properties, the *InitServerData*, *InvokeEvent*, *Connect*, *ConnectTo*, and *Disconnect* methods were needed. If you inspect the code, you'll see that it's virtually the same code that Delphi created for *TWordApplication* simplified for the simple *CommandBarButton* events. We must define the server data in the *InitServerData* method. You can find the different interface GUIDs in Office_TLB.pas, define an internal interface (*Fintf*) to a *CommandBarButton* interface, and set the *InvokeEvent* method to activate a Delphi event based on the *DispID* defined in the events interface. Finally, the *Connect*, *ConnectTo*, and *Disconnect* methods set *Fintf* to the desired interface and sink its events.

The Delphi *Wrapper* class defined in BtnSvr.pas is called *TButtonServer*. It's derived from Delphi's *TOleServer* and uses its methods to perform event sinking, etc.

Connecting to the Application

Now we have a Delphi wrapper for a command bar button, and are ready to interface with Word in our add-in. We need to declare a *TWordApplication* field that will hold the reference to the Word application. We also need to define an interface pointer to the new toolbar (*CommandBar*) we'll create, as well as two fields using the new *TButtonServer* class to hold a toolbar button and menu item.

Value	Use
$0	*Disconnected* — not loaded.
$1	*Connected* — loaded.
$2	*BootLoad* — automatic application start.
$8	*DemandLoad* — load only on user request.
$16	*ConnectFirstTime* — load only once next application startup.

Figure 2: A list of value combinations for loading your add-in.

In the **private** section of the add-in class, add:

```
FWordApp : TWordApplication;
DICommandBar : CommandBar;
DIBtn : TButtonServer;
DIMenu : TButtonServer;
```

In the *OnConnection* method, store the application pointer using the following code:

```
var
  WA : Word_TLB._Application;
begin
  FWordApp := TWordApplication.Create(nil);
  WA := Application as Word_TLB._Application;
  WordApp.ConnectTo(WA);
```

TWordApplication is defined by Delphi, and the *ConnectTo* method is used to connect it to the interface passed by Word to our add-in. Because *TWordApplication* maps the interface events to Delphi events, we can now set event handlers using the standard Delphi syntax:

```
WordApp.OnEventX := EventXHandler;
```

For example, if we want to do something when the selection in Word changes, we could write an event handler for the *OnWindowSelectionChange* event.

Creating a New Toolbar, Button, and Menu Item in the Add-in

Before we add a new toolbar and button, we need to create an event handler for the button's *OnClick* event. For this sample, we'll create a very simple event handler:

```
procedure TAddIn.TestClick(const Ctrl: OleVariant;
  var CancelDefault: OleVariant);
begin
  ShowMessage('Ouch, this hurts!');
  CancelDefault := True;
end;
```

The *CancelDefault* parameter is used when you replace the functionality of a pre-built menu item or toolbar button. It's unnecessary in this specific case, because we're creating a new button with our add-in.

Because our add-in is registered to start on application startup, we can be sure the *OnStartupComplete* method will be called, and will use it for the creation of the add-in user interface elements. We'll define *BtnIntf* as a *CommandBarControl* interface (the parent interface of the *CommandBarButton* that we'll use for our button). Our first task is to determine if our toolbar has already been created (see Figure 3).

```
DICommandBar := nil;
for i := 1 to WordApp.CommandBars.Count do
  if (WordApp.CommandBars.Item[i].Name =
        'Delphi Informant') then
    DICommandBar := WordApp.CommandBars.Item[i];
// See if we already registered the command bar with Word.
if (not Assigned(DICommandBar)) then begin
  DICommandBar := WordApp.CommandBars.Add(
    'Delphi Informant',EmptyParam,EmptyParam,EmptyParam);
  DICommandBar.Set_Protection(msoBarNoCustomize)
end;
```

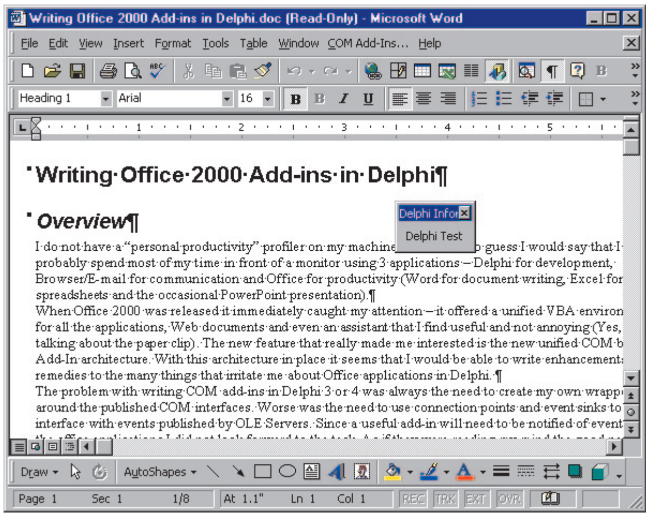
Figure 3: Determining whether our toolbar has already been created.

```
procedure TAddIn.MenuClick(const Ctrl: OleVariant;
  var CancelDefault: OleVariant);
var
  Sel : Word_TLB.Selection;
  Par : Word_TLB.Paragraph;
begin
  Sel := WordApp.ActiveWindow.Selection;
  if (Sel.Type_ in [wdSelectionNormal,
                    wdSelectionIP]) then begin
    Par := Sel.Paragraphs.Item(1);
    if (Par.Borders.OutsideLineStyle <
        wdLineStyleInset) then
      Par.Borders.OutsideLineStyle :=
        1 + Par.Borders.OutsideLineStyle
    else
      Par.Borders.OutsideLineStyle := wdLineStyleNone;
  end;
end;
```

Figure 4: Adding a new menu item by creating its *OnClick* event handler.

```
ToolsBar := WordApp.CommandBars['Tools'];
MenuIntf := ToolsBar.FindControl(EmptyParam, EmptyParam,
  'DIMenu', EmptyParam, EmptyParam);
if (not Assigned(MenuIntf)) then
  MenuIntf := ToolsBar.Controls.Add(msoControlButton,
    EmptyParam, EmptyParam, EmptyParam, EmptyParam);
DIMenu := TButtonServer.Create(nil);
DIMenu.ConnectTo(MenuIntf as _CommandBarButton);
DIMenu.Caption := 'Delp&hi Menu';
DIMenu.ShortcutText := '';
DIMenu.Tag := 'DIMenu';
DIMenu.Visible := True;
DIMenu.OnClick := MenuClick;
```

Figure 5: Connecting a *TButtonServer* wrapper to the *MenuClick* event handler.


Writing Office 2000 Add-ins in Delphi.doc (Read-Only) - Microsoft Word
File Edit View Insert Format Tools Table Window COM Add-Ins... Help
Heading 1
Arial
16
Delphi Infor
Delphi Test
Draw
AutoShapes
Page 1 Sec 1 1/8 At 1.1" Ln 1 Col 1 REC TRK EXT OVR

Writing·Office·2000·Add-ins·in·Delphi¶

Overview¶

I·do·not·have·a·"personal·productivity"·profiler·on·my·machine [illegible] guess·I·would·say·that·I·probably·spend·most·of·my·time·in·front·of·a·monitor·using·3·applications·—·Delphi·for·development,·Browser/E-mail·for·communication·and·Office·for·productivity·(Word·for·document·writing,·Excel·for spreadsheets·and·the·occasional·PowerPoint·presentation).¶

When·Office·2000·was·released·it·immediately·caught·my·attention·—·it·offered·a·unified·VBA·environ for·all·the·applications,·Web·documents·and·even·an·assistant·that·I·find·useful·and·not·annoying·(Yes, talking·about·the·paper·clip).·The·new·feature·that·really·made·me·interested·is·the·new·unified·COM·b Add-In·architecture.·With·this·architecture·in·place·it·seems·that·I·would·be·able·to·write·enhancement remedies·to·the·many·things·that·irritate·me·about·Office·applications·in·Delphi.·¶

The·problem·with·writing·COM·add-ins·in·Delphi·3·or·4·was·always·the·need·to·create·my·own·wrapp around·the·published·COM·interfaces.·Worse·was·the·need·to·use·connection·points·and·event·sinks·to interface·with·events·published·by·OLE·Servers.·Since·a·useful·add-in·will·need·to·be·notified·of·event

Figure 6: Word 2000 displaying the toolbar created by our add-in. Notice the first paragraph border created by the **Tools | Delphi** menu option, which our add-in added to Word.

Now we give our toolbar a unique name ("Delphi Informant") and check on startup if this toolbar already exists. If it does, we set the *DICommandBar* field to it; otherwise, we use the Word application's *CommandBars* property's *Add* method to create a new one. After the new toolbar is created, we set its protection to *msoBarNoCustomize*, which prevents users from adding or removing buttons from our toolbar.

At this point, *DICommandBar* points to a valid toolbar. We're now ready to get the interface pointer to our toolbar button (*CommandBarButton*) via the *Controls* collection of this interface. Because we set the protection of the toolbar to no customization, we know that if there is a control on the toolbar, it will be our button (we'll add just one button to the toolbar). If we can't find any controls on the toolbar, we'll add a new one:

```
if (DICommandBar.Controls.Count > 0 ) then
  BtnIntf := DICommandBar.Controls.Item[1]
else
  BtnIntf := DICommandBar.Controls.Add(msoControlButton,
    EmptyParam, EmptyParam, EmptyParam, EmptyParam);
```

Notice that the first item in a collection in Office is item #1, not item #0 as we are used to from the Windows API or Delphi's *TList* and *TStringList*. At this point, *BtnIntf* has the desired toolbar button interface, and we can create a *TButtonServer* wrapper around it:

```
DIBtn := TButtonServer.Create(nil);
DIBtn.ConnectTo(BtnIntf as _CommandBarButton);
DIBtn.Caption := 'Delphi Test';
DIBtn.Style := msoButtonCaption;
DIBtn.Visible := True;
DIBtn.OnClick := TestClick;
```

We'll use the *ConnectTo* method to connect to the toolbar button's events, and set the *OnClick* event to the *TestClick* event handler we wrote earlier. We'll finish by making sure that the toolbar is visible:

```
DICommandBar.Set_Visible(True);
```

TLIBIMP creates a read-only property for the *Visible* property of the *CommandBar* interface, but it creates a Set_*Visible* method that we use here. You'll see the same technique used in the property implementation of *TButtonServer*. (My guess is that this is a bug in TLIBIMP). Now we'll use a similar technique to add a new menu item. First we'll create the *OnClick* event handler for the menu item (see Figure 4). This event handler iterates the first paragraph of the selected text between the available border styles. Here we take advantage of the specific Word application capabilities.

In *OnStartupComplete*, we add the code in Figure 5 to obtain an interface pointer to the Tools menu, search it for the existence of our menu item, and, if it doesn't exist, add it. We'll later create a *TButtonServer* wrapper around it, and connect it to the *MenuClick* event handler (see Figure 5).

Notice the use of the *FindControl* method of the *CommandBar* interface to look for a specific control based on a unique tag assigned to it. If the control is found, it will be set to *MenuIntf*; otherwise *MenuIntf* won't be assigned, and we'll create the new control (menu item). Figure 6 shows the toolbar that we've created for Word with our add-in.

```
if (Assigned(DIBtn)) then begin
  DIBtn.Free;
  DIBtn := nil;
end;
if (Assigned(DIMenu)) then begin
  DIMenu.Free;
  DIMenu := nil;
end;
if (Assigned(DICommandBar)) then begin
  // This is an interface, not an object!
  DICommandBar.Delete;
  DICommandBar := nil;
end;
```

Figure 7: Cleaning the user-interface elements with the *OnBeginShutdown* method.

Cleaning Up

I used the *OnBeginShutdown* method to clean up the user interface elements we created (see Figure 7). Notice that we free the *TButtonServer* wrappers and the *CommandBar* interface.

Sharing Add-in Code among Office Applications

Because the add-in architecture is identical among Office applications, you can use the same physical OLE Server DLL to serve multiple applications. The trick is to determine which application is the one that activated the add-in, and use the appropriate object model. The easiest way to determine the application that activated the add-in is to assign the *Application IDispatch* pointer that's passed to the *OnConnection* method to an *OleVariant* variable, and use the *Name* property to determine what application activated the add-in:

```
var
  AppVar : OleVariant;
begin
  AppVar := Application;
  if (AppVar.Name = 'Outlook') then
    begin
      ...
    end
  else if (AppVar.Name = 'Microsoft Word') then
    begin
      ...
    end else ...
```

Conclusion

We've discussed the ease of creating add-ins for Office 2000 applications. The new COM-based add-in architecture makes it easy to share add-in code among Office 2000 applications. The same COM object can be used as an add-in for more than one application, and the same user-interface objects can be shared among Office applications. Also, if you create toolbars, menu items, or Office assistant Help in one application, the same code will work in other applications.

For more information about Office development and the creation of Office 2000 add-ins, consult the microsoft.public.officedev newsgroup hosted on msnews.microsoft.com. Δ

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\MAY\DI200005RL.

Ron Loewy is a software developer for HyperAct, Inc. He is the lead developer of eAuthor Help, HyperAct's HTML Help authoring tool. For more information about HyperAct and eAuthor Help, contact HyperAct at (515) 987-2910 or visit http://www.hyperact.com.

By Ron Gray

Dynamic Forms

Creating Forms and Controls at Run Time

One of Delphi's strongest features is how easy it is to create user interfaces using the VCL. Forms, components, and controls are created visually at design time, and the VCL generates the code to create and display them at run time. These "static" forms satisfy most UI requirements. When they don't, Delphi still makes it easy to create forms and controls dynamically at run time based on run-time criteria. This opens exciting possibilities for most applications.

Data-driven interfaces are the most common reason for using dynamic forms and controls. They can vary from simple mechanisms, such as dynamic dialog boxes, to refined processes that read menus, data entry forms, reports, and other settings from information stored on the disk. In other words, the data drives the application. As long as the properties are known, creating forms and controls is straightforward.

Creating Forms Dynamically

Use *TForm* to create most forms, such as a main window, dialog box, or MDI child. The following example creates an instance of the *TForm* component and calls the *Show* method:

```
procedure NewForm;
var
  frmForm: TForm
begin
  frmForm := TForm.Create(Application);
  frmForm.Show;
end;
```

This example creates and displays a form, but what will it look like? No properties are set, so the form is displayed using all the default values. It's not clear what type of window it is, how big it is, or where it appears on the screen. These are basic properties (listed in Figure 1) that should be set before displaying the form or adding components to it.

Property	Possible Values
BorderIcons	*biSystemMenu, biMinimize, biMaximize, biHelp*
BorderStyle	*bsNone, bsSingle, bsSizeable, bsDialog, bsToolWindow, bsSizeToolWin*
FormStyle	*fsNormal, fsMDIChild, fsMDIForm, fsStayOnTop*
Height	<integer>
Left	<integer>
Position	*poDesigned, poDefault, poDefaultPosOnly, poDefaultSizeOnly, poScreenCenter, poDesktopCenter*
Top	<integer>
Width	<integer>
WindowState	*WsNormal, wsMinimized, wsMaximized*

Figure 1: Properties of *TForm* that determine the type of window, its size, and position.

Using the properties listed in Figure 1, the following example displays a modal, stay-on-top dialog box in the center of the screen:

```
frmForm := TForm.Create(Application);
with frmForm do begin
  BorderStyle := bsDialog;
  FormStyle := fsStayOnTop;
  Height := 185;
  Width := 450 ;
  Position := poScreenCenter;
end;
frmForm.ShowModal;
```

This code is much better than the previous example because it clearly provides pertinent information about the form, and doesn't rely on guesswork about default properties. Of course, there are many other properties that can be set to further customize the form. As a general rule, all the properties a developer would set at design time must be set programmatically at run time. Properties that are rarely changed from their defaults should be set programmatically as well, to promote clarity and avoid surprises.

Parent and Owner

The blank form created previously needs some controls to give it some functionality. Before discussing how to create controls dynamically, however, it's important to note the difference between the *Parent* property declared in

TControl, and the *Owner* property declared in *TComponent*. The *Parent* is always a windowed control that visually contains the control or form. The *Owner* is passed as a parameter in the constructor and determines when the component is freed. The former property is used for display purposes, the latter for memory management. Both are important.

TControl.Parent

Many forms do not have a *Parent*. For example, forms that appear directly on the Windows desktop (like the dialog box in the previous example) have *Parent* set to **nil**. However, a form can be embedded in any other visual control by setting the *Parent* to that control. For example, the same form that can be shown as a stand-alone form on the desktop can also be embedded in a tab sheet by simply changing the *Parent* property to the tab sheet.

For controls, the *Parent* is usually the form on which it's displayed. But it could also be a panel, group box, or some control that is designed to contain another. When creating a new control, always assign a *Parent* property value for the new control.

TComponent.Owner

The *Owner* of a control or form determines when the memory used by the control can be freed. It's passed to the constructor when creating a new form or control. When a component is destroyed, all of the components owned by it are also destroyed. Usually, the form is the *Owner* of all the controls on it, so when the form is destroyed, all the controls are destroyed with it. Likewise, most stand-alone forms are owned by the *Application* object, so they can be freed when the application is terminated.

Creating Controls Dynamically

Creating controls dynamically is not much different from creating forms. Each new control must have an *Owner* (passed to the constructor and usually the form), and a *Parent* (usually the form, but can also be a panel, tab sheet, or other container control). Set additional properties to further customize the control.

The following example displays a button on the bottom right of the form:

```
btnButton := TButton.Create(frmForm);
with btnButton do begin
  Parent := frmForm;
  Height := 25;
  Width := 75;
  Left := frmForm.ClientWidth - 85;
  Top := frmForm.ClientHeight - 35;
  Caption := 'Hey now!';
  ModalResult := 1;
  Show;
end;
```

There are a few properties of interest here. First, note that the button's position is relative to the form. The dimensions of the form can change, but the button will still appear on the bottom right of the form. When positioning controls relative to the form, use the client area (*ClientHeight* and *ClientWidth*) rather than the actual form area (*Width* and *Height*), which includes the form's caption and other space that can't be used by the client. Panels and anchors are also useful for positioning controls.

Second, if the form is shown modally, the *ModalResult* property can be used to determine whether, and how, the button closes the form. Third, the control's *Parent* is the form, but it could have been a previously created panel or other containing control. For example, a panel could be created dynamically and anchored to the bottom of the form. The panel would then be assigned to the button's *Parent* so it will resize along with the panel. Finally, the *Show* method is only required if the button's *Parent* (the form in the previous example) is already visible. Otherwise, when the form is shown, it will cause all "child" controls to be shown as well.

A Simple Example

Using the techniques previously described, a custom function *MessageDlgEx* can be written to display a dialog box like Delphi's *MessageDlg* function, except that the button captions can be defined. The function is shown in Listing One beginning on page 21. (The function, and all other source discussed in this article, is available for download; see the end of this article for details.) This allows new options to be displayed rather than relying on the standard dialog box buttons, such as OK, Yes, No, and so on. For example, the following code prompts for a decision from the user:

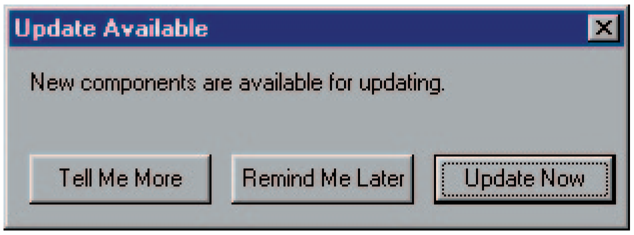
```
nResult := MessageDlgEx('Update Available',
  'New components are available for updating.',
  mtWarning, ['Update Now', 'Remind Me Later',
  'Tell Me More']);
```

Calling this function displays a modal dialog box like the one shown in Figure 2.

The function performs some simple math to determine the correct widths of the buttons and form, based on the widths of the button captions and message. Once the form and button widths are determined, the function simply loops through the array and creates a button for each item. The button's *ModalResult* is its one-based position in the array, as shown in Figure 3. The *Return* value is the ordinal position of the clicked button. Code can be written to respond to each button.

Working with Messages

Creating controls at run time is easy, but dynamically responding to their events is more difficult. Generally, actions and basic functionality must be coded in the application, and events and controls are simply


Update Available

New components are available for updating.

Figure 2: The *MessageDlgEx* function creates the form and buttons dynamically.

```
for ii := 0 to High(AButtons) do begin
  btnButton := TButton.Create(frmForm);
  btnButton.Height := 25;
  btnButton.Width := nButtonWidth;
  btnButton.Left :=
    frmForm.Width - ((ii + 1) * (nButtonWidth + 10 ));
  btnButton.Top := frmForm.ClientHeight - 35;
  btnButton.Caption := AButtons[ii];
  btnButton.ModalResult := ii + 1;
  btnButton.Parent := frmForm;
end;
```

Figure 3: Once the form and button widths are determined, the function simply loops through the array and creates a push button for each item.

linked to pre-defined actions. No matter how generic and dynamic a form is, it still needs a save procedure. The *MessageDlgEx* function mentioned previously dynamically responds to events by simply assigning each button's *ModalResult* property to its ordinal position in the array. How the event is then handled is outside the function's control.

With more complex data entry forms, buttons must be linked to specific actions, such as the **OK** button that calls a save routine, and the **Cancel** button that performs any rollback procedures. Assigning a specific action to a control requires code that responds to the control's events, or overrides the control's message handler to respond to messages. The easiest way is to respond to the control's events.

Delphi converts most Windows messages sent to the control to events. At design time, developers link the control's events to methods of the form. For example, the **OK** button, named *btnOK*, responds to the *OnClick* event by calling the *btnOKClick* method of the form. Event handlers can be assigned programmatically. However, the compiler expects a method of a class rather than a stand-alone procedure, so the method itself cannot be created programmatically. In other words, the **OK** button's *OnClick* event cannot simply call a generic *SaveChanges* function, but rather must call a method of the form. An easy way around this is to visually create a blank template form that is used dynamically at run time, and write generic event handlers that can be used by the various controls that are created programmatically.

For example, the *OnClick* event of an **OK** button must call the *SaveChanges* function. The form's code defines a generic *OnClick* event for all buttons. Within the method, specific actions can be programmed for each button. So, if the **OK** button generated the event, the *SaveChanges* function is called:

```
procedure TForm1.ButtonClick(Sender: TObject);
begin
  if TControl(Sender).Name = 'btnOK' then
    SaveChanges
  else
    ShowMessage('You did not click the OK button.');
end;
```

When the **OK** button is created, simply assign the *OnClick* handler as in the following code:

```
oButton := TButton.Create(Self);
with oButton do begin
  Name := 'btnOK';
  Height := 25;
  Width := 110 ;
  Left := 240 ;
  Top := 16;
  OnClick := ButtonClick;
  Caption := 'OK';
  Parent := Self;
end;
```

Other generic event handlers can be defined in a template form, such as *ListBoxClick*, *ListBoxSelect*, and others. By defining generic event handlers at the form level, it's easy to respond dynamically to most events.

Working with Data

One of the more compelling reasons for creating forms and controls dynamically is the ability to generate data entry screens at run time that are either designed by the user, or based on some other data-driven criteria. Creating data access components and data controls is no different than what has already been described. Of course, the data controls must be linked to specific fields using the various data access components, so the order of their creation and linking is important. A rule of thumb is to simply program in the same sequence that would normally be done visually. The basic steps to creating a simple data entry form are:

1) Create a *TTable* component and assign the *DatabaseName* and *TableName* properties.
2) Create a *TDataSource* component and set the *DataSet* property to the *TTable* object.
3) Create *TDBEdit* controls and set the *DataSource* property to the *TDataSource* object, and *DataField* property to the field name.
4) Set *TTable.Active* to True.
5) Provide **OK** and **Cancel** buttons, or a navigation bar for posting or canceling changes.

The code in Figure 4 can be called from a method of a blank form to create data access components and controls on the form for data entry. The example assumes there is an alias named DBDEMOS pointing to Delphi's sample database.

The code in Figure 4 creates the data entry screen shown in Figure 5.

```
var
  tblTable: TTable;
  dsDataSource: TDataSource;
  dbNavigator: TDBNavigator;
  oEdit: TDBEdit;
  oLabel: TLabel;
begin
  tblTable := TTable.Create(Self);
  tblTable.DatabaseName := 'DBDEMOS';
  tblTable.TableName := 'animals.dbf';

  dsDataSource := TDataSource.Create(Self);
  dsDataSource.Name := 'dsDataSource';
  dsDataSource.DataSet := tblTable;

  dbNavigator := TDBNavigator.Create(Self);
  dbNavigator.Parent := Self;
  dbNavigator.Left := 8;
  dbNavigator.Top := 12;
  dbNavigator.DataSource := dsDataSource;

  oLabel := TLabel.Create(Self);
  oLabel.Parent := Self;
  oLabel.Left := 8;
  oLabel.Top := 52;
  oLabel.Caption := '&Name:';

  oEdit := TDBEdit.Create(Self);
  oEdit.Parent := Self;
  oEdit.Left := 10 0 ;
  oEdit.Top := 52;
  oEdit.DataSource := dsDataSource;
  oEdit.DataField := 'NAME';

  tblTable.Active := True;
end;
```

Figure 4: Creating data access components and controls on a form.

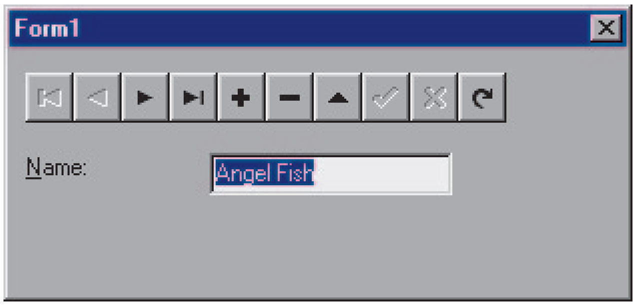


Figure 5: A dynamically created data entry form.

Of course, this is a simple data entry example, but the same techniques apply when creating more complex forms. For example, a self-configuring data entry form can simulate Delphi's Form Wizard at run time, automatically creating controls for each field in a selected table. This requires looping through *TDataSet.Fields*, creating a control and associated label for each field in the data set (see Figure 6).

Note that the *Caption* of the associated *TLabel* control is set to the field name. This is fine for small utility operations, but when using data-driven forms within an application, the application's framework should provide a way to save and restore the *DisplayName* of each field. This way, a more descriptive label is provided, rather than using the field name.

Data validation is straightforward as well. Values can be validated in the application before they are sent to the database server. For field-level validation, use *TField.EditMask* to restrict data that can be entered in the field, and *TField.OnValidate* to validate data before the record is updated. To reject the current value of the field, raise an exception in the event handler. For record-level validation, use *TDataSet.BeforePost*, which is called just before posting the record. Call *Abort* to cancel the *Post* operation.

Working with Other Components

Most components can be created dynamically with no problem. Menus, grids, OLE containers, ActiveX controls, and many others can all be created programmatically at run time. This means that just about any data entry form that can be created visually can also be created dynamically. Of course, the *Parent* and *Owner* properties are still very important. And each component has unique properties that must be set.

The *Controls* and *Components* Arrays

Managing controls created dynamically is somewhat different from managing controls created at design time. When created at design time, controls are usually given names that help define what they are. For example, it's obvious that a control named *btnOK* is the OK button. The developer can easily write code that specifically references this button. Dynamically created controls tend to be more generic — and should be. So it's not always possible to refer to controls specifically by name. Fortunately, components and controls are stored in arrays that can be accessed to help manage dynamically created controls.

```
for ii := 0 to Table1.FieldCount-1 do begin
  LabelCaption := Table1.Fields[ii].FieldName;
  DataType := Table1.Fields[ii].DataType;
  // ... create associated TLabel control using field name.
  if DataType = ftString then
    // Create a TDBEdit control.
  if DataType = ftMemo then
    // Create a TDBMemo control.
  // ... and so on.
end;
```

Figure 6: Looping through *TDataSet.Fields*.

```
var
  I: Integer;
  Temp: TComponent;
begin
  for I := ComponentCount - 1 downto 0 do begin
    Temp := Components[I];
    if not (Temp is TControl) then begin
      RemoveComponent(Temp);
      DataModule2.InsertComponent(Temp);
    end;
  end;
end;
```

Figure 7: Using the *Components* array.

The *Controls* Array

The *TWinControl.Controls* property returns an array of all controls for which the windowed control is the *Parent*. When the *Parent* is assigned to a control, Delphi automatically updates the *Controls* array of the previous *Parent* and new *Parent*. This is a read-only property, but it can be updated by simply changing the *Parent* of a control. Using this array it's quite easy to iterate through all the controls without having to refer to each of them by name. For example, the following code loops through the *Controls* array to disable all controls:

```
for ii = 0 to ControlCount - 1 do
  if Controls[ii] is TControl then
    Controls[ii].Enabled := False;
```

The *Components* Array

TComponent.Components provides access to all components owned by the component. The sample in Figure 7, taken from Delphi's Help file, moves any non-visual components on the form into a separate data module.

Conclusion

The ability to create forms and controls dynamically opens exciting possibilities for applications. Examples range from simple dialog boxes, such as the *MessageDlgEx* function described in this article, to elaborate data-driven menus and data entry forms.

Forms and controls have several key properties that must be set when they are created dynamically. The *Parent* and *Owner* properties are important for display and memory management purposes. Messages can be trapped by defining generic event handlers to an empty form, or by overriding the control's message handler. Finally, there are techniques, properties, and methods that help manage controls. ∆

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\MAY\DI200005RG.

Ron Gray is a software developer specializing in business database applications. He has written numerous articles using different languages and is the author of LookUp Manager, a collection of components for visually managing lookup codes and abbreviations in applications. He can be reached at rgray@compuserve.com.

Begin Listing One — The *MessageDlgEx* Function

```
function MessageDlgEx(const Caption, Msg: string;
  AType: TMsgDlgType; AButtons: array of string): Word;
var
  oForm: TForm;
  oLabel: TLabel;
  oButton: TButton;
  nButtonWidth: Integer;
  nAllButtonsWidth: Integer;
  nCtrlHeight: Integer;
  nMessageWidth: Integer;
  ii: Integer;
begin
  // Create the form.
  oForm := TForm.Create(Application);
  oForm.BorderStyle := bsDialog;
  oForm.BorderIcons := oForm.BorderIcons - [biSystemMenu];
  oForm.FormStyle := fsStayOnTop;
  oForm.Height := 185;
  oForm.Width := 450 ;
  oForm.Position := poScreenCenter;
  oForm.Caption := Caption;
  // Loop through buttons to determine the longest caption.
  nButtonWidth := 0 ;
  for ii := 0 to High(AButtons) do
    nButtonWidth := Max(nButtonWidth,
      oForm.Canvas.TextWidth(AButtons[ii]));
  // Add padding for the button's caption.
  nButtonWidth := nButtonWidth + 10 ;
  // Determine space required for all buttons.
  nAllButtonsWidth := nButtonWidth * (High(AButtons) + 1);
  // Each button has padding on each side.
  nAllButtonsWidth := nAllButtonsWidth +
    (10  * (High(AButtons) + 2));
  // The form has to be at least as wide as the buttons.
  if nAllButtonsWidth > oForm.Width then
    oForm.Width := nAllButtonsWidth;
  // Determine if the message can fit in the form's width,
  // or if it must be word wrapped.
  nCtrlHeight := oForm.Canvas.TextHeight('A') * 3;
  nMessageWidth := oForm.Canvas.TextWidth(Msg);
  // If the message can fit with just the width of the
  // buttons, adjust the form width.
  if nMessageWidth < nAllButtonsWidth then
    oForm.Width := nAllButtonsWidth;
  if nMessageWidth > oForm.ClientWidth then begin
    // Determine how many lines are required.
    nCtrlHeight := Trunc(nMessageWidth/oForm.ClientWidth);
    // Add 3 more lines as padding.
    nCtrlHeight := nCtrlHeight + 3;
    // Convert to pixels.
    nCtrlHeight :=
      nCtrlHeight * oForm.Canvas.TextHeight('A');
  end;
  // Adjust the form's height accomodating the message,
  // padding and the buttons.
  oForm.Height :=
    nCtrlHeight + (oForm.Canvas.TextHeight('A') * 4) + 22;
  // Create the message control.
  oLabel := TLabel.Create(oForm);
  oLabel.AutoSize := False;
  oLabel.Left := 10 ;
  oLabel.Top := 10 ;
  oLabel.Height := nCtrlHeight;
  oLabel.Width := oForm.ClientWidth - 20 ;
  oLabel.WordWrap := True;
  oLabel.Caption := Msg;
  oLabel.Parent := oForm;
  // Create the pushbuttons.
  for ii := 0 to High(AButtons) do begin
    oButton := TButton.Create(oForm);
    oButton.Height := 25;
    oButton.Width := nButtonWidth;
    oButton.Left :=
      oForm.Width - ((ii + 1) * (nButtonWidth + 10 ));
    oButton.Top := oForm.ClientHeight - 35;
    oButton.Caption := AButtons[ii];
    oButton.ModalResult := ii + 1;
    oButton.Parent := oForm;
  end;
  Result := oForm.ShowModal;
end;
```

End Listing One

By Simon Murrell

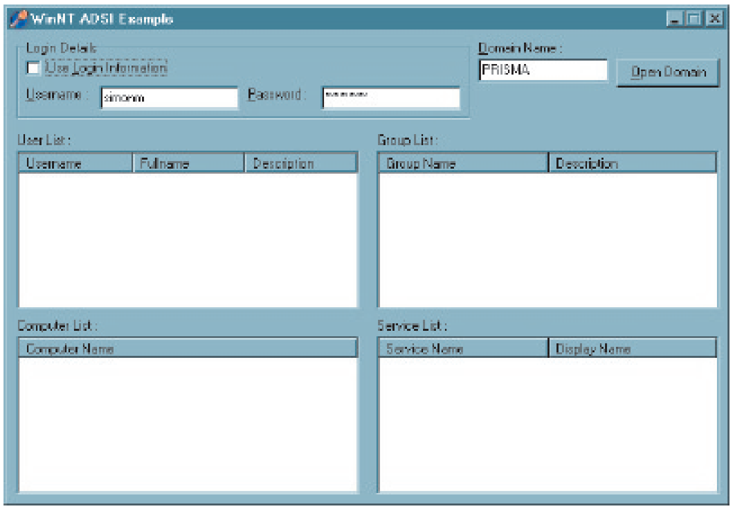
Active Directories

Using ADSI on Your Windows NT and Windows 2000 Systems

Active Directory is the directory service used in Windows NT 4.0 and Windows 2000. It's also the foundation of Windows 2000. To access the Active Directory Service, you need to use the API that Microsoft provides, named ADSI (Active Directory Service Interfaces). ADSI is a set of COM interfaces used to access the different directory services. Programmers can currently access four network directory structures using the providers supplied in ADSI: WinNT (Microsoft SAM database), LDAP (Lightweight Directory Access Protocol), NDS (NetWare Directory Service), and the NWCOMPAT (Novell NetWare 3.x).

ADSI now makes the Windows NT administrator's job easier. ADSI allows the administrator to perform common tasks, such as the addition of new users, managing printers, security settings, and controlling the NT domain. Because ADSI uses COM interfaces, well-known languages, such as Visual Basic, Visual C++, C++Builder, Delphi, or any other COM-enabled language, can use ADSI to write new client software. Well-known ISVs have used ADSI to make their software applications directory-enabled.

Active Directory runs on either Windows NT 4.0 or Windows 2000. Client applications can run on Windows 95, 98, NT 4.0, and 2000. To use ADSI, you need to install the ADSI COM interfaces. You can download the ADSI 2.5 SDK from the Microsoft ADSI Web site at http://www.microsoft.com/adsi. The SDK contains documentation, online help, and samples.



Username : simonm
Domain Name : PRISMA

Figure 1: Sample application.

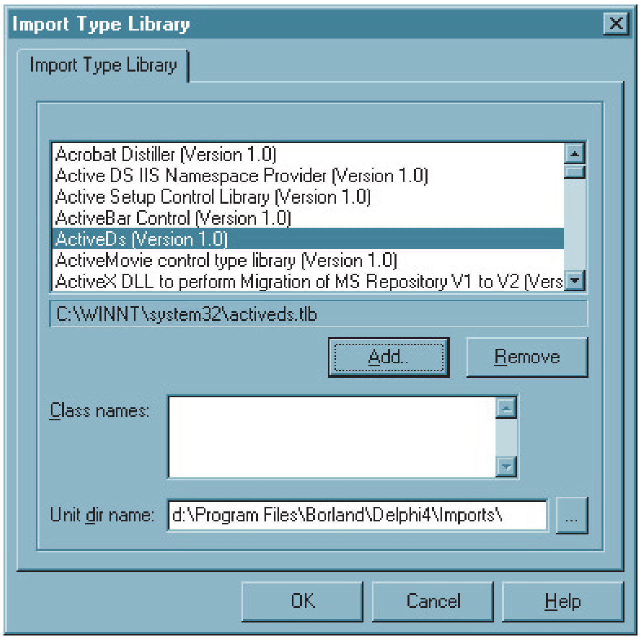
The only problem is that the samples and documentation are aimed toward Microsoft products, namely Visual Basic and Visual C++. This is one of the main reasons I'm writing this article: so there are examples for the Delphi community. The documentation is also mainly aimed at the WinNT provider — not the more common LDAP provider used in Microsoft Exchange and Microsoft Site Server.

The Demo Application

The sample application I wrote, shown in Figure 1, demonstrates how to use some of the functionality provided by the WinNT provider (this application is available for download; see end of article for details). The sample application is used to connect to a domain within your organization. Once connected to the domain, the sample application will list the NT users and groups found on the PDC, along with the computers participating in the domain. I'll also demonstrate how to view services from the NT computers found within the domain, along with adding and removing users and viewing users found within the NT groups.

Using ADSI to Control Windows NT/2000

The WinNT provider allows the developer to control and manipulate the Windows NT SAM database. You can use WinNT to gain access to resources, such as users, groups, computers, file shares, printer jobs, printer queues, and services. To use ADSI from a Delphi application, you need to import the Active Directory type library. To do this, select the **Project | Import Type Library** option from the menu, select **ActiveDs (Version 1.0)**, and click **OK** (see Figure 2). Delphi will create the type library declaration .pas file, and import it into your project.



```
Acrobat Distiller (Version 1.0)
Active DS IIS Namespace Provider (Version 1.0)
Active Setup Control Library (Version 1.0)
ActiveBar Control (Version 1.0)
ActiveDs (Version 1.0)
ActiveMovie control type library (Version 1.0)
ActiveX DLL to perform Migration of MS Repository V1 to V2 (Vers
C:\WINNT\system32\activeds.tlb
```

```
d:\Program Files\Borland\Delphi4\Imports\
```

Figure 2: Importing the Active Directory type library.

```
// Procedure is used to bind to the ADSI WinNT directory.
procedure TMainFrm.actOpenWinNTExecute(Sender: TObject);
var
  UnknownObject: IUnknown;
  DomainPath: WideString;
  Domain: IADsContainer;
begin
  // Assign domain path.
  DomainPath := 'WinNT://' + ADSIDomainName.Text;
  // If login details are used then.
  if cbUseLogin.Checked then
    // Use login details and create domain object.
    OleCheck(AdsOpenObject(PWideChar(DomainPath),
      PWideChar(ADSIUsername.Text),
      PWideChar(ADSIPassword.Text), 0 , IID_IADsContainer,
      UnknownObject));
  else
    // Create domain object.
    OleCheck(ADsGetObject(PWideChar(DomainPath),
      IID_IADsContainer, UnknownObject));
  // Assign domain object.
  Domain := UnknownObject as IADsContainer;
  // Get Lists from Domain.
  GetDomainInformation(Domain);
end;
```

Figure 3: Binding to the ADSI object using the default security credentials or alternate credentials.

Binding to the WinNT Directory Service

Connecting to the WinNT directory service is simply a matter of finding the domain controller, and then binding to the required object. To bind to the WinNT provider in Delphi, you can use the *ADsGetObject* or *ADsOpenObject* function.

The *ADsGetObject* function is used to bind to an ADSI object by specifying the object's path, along with the interface identifier and object. The declaration of this function is:

```
function ADsGetObject(lpszPathName: PWideChar;
  const riid: TIID; out obj): HResult; stdcall;
  external 'activeds.dll';
```

The first parameter is the path name used to bind to the object in the underlying directory service. The second parameter is the interface identifier for a specified interface on this object; and the third parameter is the indirect pointer to the requested interface. By default, this function uses secure authentication, i.e. the function uses the security context of the current user.

The second function, *ADsOpenObject*, is used to bind to an ADSI object using an alternate security context by stipulating a username and password of the required user. The declaration of this function is:

```
function ADsOpenObject(lpszPathName: PWideChar;
  lpszUserName: PWideChar; lpszPassword: PWideChar;
  dwReserved: LongInt; const riid: TIID; out obj): HResult;
  stdcall; external 'activeds.dll';
```

The first parameter is the path name used to bind to the object in the underlying directory service. The second and third parameters are the username and password of the user whose security credentials you want to use. The fourth parameter is a reserved provider flag, which stipulates the authentication method you want to bind with. The fifth parameter is the interface identifier for a specified interface on this object. Finally, the sixth parameter is the indirect pointer to the requested interface.

Thus, the first function defaults to the logged user's credentials, and the second function allows the developer to specify security credentials to bind to the ADSI object. Figure 3 shows the procedure used to bind to the ADSI object, by using either the default security credentials or alternate credentials.

Let's step through the procedure to see exactly what's going on. First, three variables are declared. The first is the interface variable, which the binding functions are going to return from the specified object path:

```
UnknownObject: IUnknown;
```

Second is a WideString variable, which is used to generate an object path in the binding functions:

```
DomainPath: WideString;
```

Third is an *IADsContainer* variable, which will be assigned to the returned interface variable:

```
Domain: IADsContainer;
```

The *IADsContainer* variable is going to be used to retrieve all the users, groups, and computers from the ADSI object specified. Alternatively, you can declare this variable as *IADsDomain*, but I want to enumerate the children objects found within the domain. (Read the SDK to find out the exact differences between the *IADsContainer* and the *IADsDomain*.)

The following statement assigns the path for the object we want to retrieve. If your domain name is "PRISMA," for example, to retrieve the domain ADSI object you would need to assign the path as "WinNT://PRISMA":

```
// Assign domain path.
DomainPath := 'WinNT://' + ADSIDomainName.Text;
```

The next section of code uses an alternate security credential, or the default logged on security credential:

```
// If login details are used then.
if cbUseLogin.Checked then
  // Use login details and create domain object.
  OleCheck(AdsOpenObject(PWideChar(DomainPath),
    PWideChar(ADSIUsername.Text),
    PWideChar(ADSIPassword.Text), 0 , IID_IADsContainer,
    UnknownObject));
else
  // Create domain object.
  OleCheck(ADsGetObject(PWideChar(DomainPath),
    IID_IADsContainer, UnknownObject));
```

Then we assign the indirect pointer to the *IADsContainer*, so we can query the domain for children objects:

```
// Assign domain object.
Domain := UnknownObject as IADsContainer;
```

Finally, we pass the *IADsContainer* to the procedure that retrieves the child objects within the domain:

```
// Get Lists from Domain.
GetDomainInformation(Domain);
```

Searching the Domain's *IADsContainer*

To search through the domain for child objects, we use the *GetDomainInformation* procedure. We pass the *IADsContainer* that we retrieved from the binding procedure. The source code to search the *IADsContainer* is shown in Figure 4.

The following is a detailed explanation of the source code in Figure 4. We declare the following variable so we can enumerate the child objects with the container. We'll use this variable to search through the children variables in the container:

```
Enum: IEnumVariant;
```

The following is a temporary variable used to store the children retrieved from the container object:

```
ADsTempObj: OLEVariant;
```

The following is the interface variable of the children in the container object:

```
ADsObj: IADs;
```

We now assign the enumerator object of the container to the *Enum* variable, so we can begin searching the container:

```
// Assign enumerator object.
Enum := (Domain._NewEnum) as IEnumVariant;
```

Once we've assigned the variable, we begin searching through the enumerator variable, assigning each child object to the temporary *OLEVariant* object:

```
// Search through enumerator object.
while (Enum.Next(1, ADsTempObj, Value) = S_OK) do begin
```

This *OLEVariant* object reference is then assigned to the ADSI object:

```
// Assign temporary object.
ADsObj := IUnknown(ADsTempObj) as IADs;
```

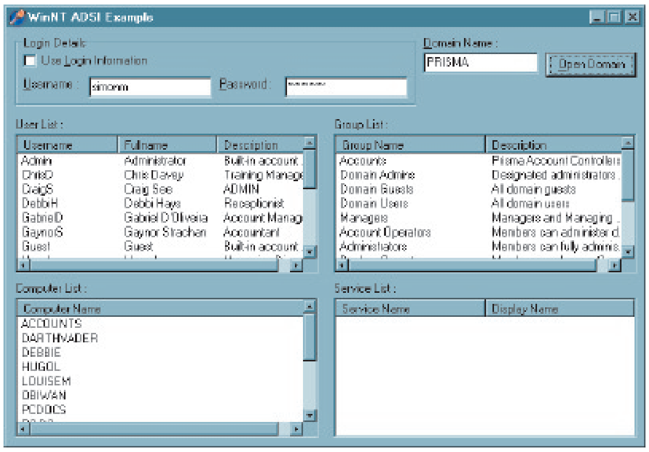
Once the ADSI object has been assigned, we check the child object's class. According to the type of class, we then pass the ADSI to the respective procedure to add the properties from the ADSI object to the respective ListView component on the form:

```
// If object is a user object then.
if AdsObj.Class_ = 'User' then
  AddUserToList(ADsObj);
// If object is a group object then.
if AdsObj.Class_ = 'Group' then
  AddGroupToList(ADsObj);
// If object is a computer object then.
if AdsObj.Class_ = 'Computer' then
  AddComputerToList(ADsObj);
```

Once you've assigned all the values from the child objects, your application should be filled with information, as in Figure 5.

```
// Procedure retrieves the domain information.
procedure TMainFrm.GetDomainInformation(
  Domain: IADsContainer);
var
  Enum: IEnumVariant;
  ADsTempObj: OLEVariant;
  ADsObj: IADs;
  Value: LongWord;
begin
  // Empty User, Group, and Computer lists.
  UserListView.Items.Clear;
  GroupListView.Items.Clear;
  ComputerListView.Items.Clear;
  // Assign enumerator object.
  Enum := (Domain._NewEnum) as IEnumVariant;
  // Search through enumerator object.
  while (Enum.Next(1, ADsTempObj, Value) = S_OK) do begin
    // Assign temporary object.
    ADsObj := IUnknown(ADsTempObj) as IADs;
    // If object is a user object then.
    if AdsObj.Class_ = 'User' then
      AddUserToList(ADsObj);
    // If object is a group object then.
    if AdsObj.Class_ = 'Group' then
      AddGroupToList(ADsObj);
    // If object is a computer object then.
    if AdsObj.Class_ = 'Computer' then
      AddComputerToList(ADsObj);
  end;
end;
```

Figure 4: Source code to search the *IADsContainer*.



Use Login Information [ ]
Username: simonm   Password: ********
Domain Name: PRISMA

User List:

Username	Fullname	Description
Admin	Administrator	Built-in account
ChrisD	Chris Davey	Training Manage
CraigS	Craig See	ADMIN
DebbiH	Debbi Hays	Receptionist
GabrielD	Gabriel D'Oliveira	Account Manag
GaynorS	Gaynor Strachan	Accountant
Guest	Guest	Built-in account

Group List:

Group Name	Description
Accounts	Prisma Account Controllers
Domain Admins	Designated administrators
Domain Guests	All domain guests
Domain Users	All domain users
Managers	Managers and Managing
Account Operators	Members can administer d
Administrators	Members can fully adminis

Computer List:

Computer Name
ACCOUNTS
DARTHVADER
DEBBIE
HUGOL
LOUISEM
OBIWAN
PCDOCS

Service List:

Service Name	Display Name

Figure 5: Your application should be filled with information.

Creating and Removing Users from Computers

With the WinNT provider, one can create and remove users from any computer within any domain. To create a user within the domain on a specified computer, you need to bind to the computer to which you want to add the user. Once you've bound to the ADSI container object of the required computer, you need to call the *Create* method. The *Create* method of the container object takes two arguments. One is the type of ADSI object you want to create; the other is the name describing the new ADSI object. The *Create* method then returns a reference to the new ADSI object created. Figure 6 is the source code that creates the user.

As usual, first the variables are declared. The first is the computer container variable to bind to. From this container, we'll add the new user to the domain:

```
ComputerObj: IADsContainer;
```

The following is a temporary interface variable, which is used to store the newly created ADSI object:

```
TempUserObj: IUnknown;
```

The following is the *IADsUser* variable, which we're going to assign from the newly created ADSI user object. It will enable the application to represent and manage the end-user account on the domain:

```
UserObj: IADsUser;
```

This WideString variable will be used to store the name of the PDC (Primary Domain Controller) computer of the domain. You can specify any other computer name, and the source code will add the user to the computer, but the source code will only return the users from the domain:

```
PDCName: WideString;
```

The following WideString variable is used to store the name of the new user one wishes to add:

```
NewUserName: WideString;
```

The following WideString variable is going to store the ADSI object path of the computer we're going to bind to, e.g. "WinNT://PDCCOMP,computer":

```
AdsPath: WideString;
```

In the following, we call the *InputBox* method to retrieve the names of the PDC computer and the new user we want to add:

```
// Retrieve information from user.
PDCName := InputBox('Create New User',
  'Please type in the name of the Domain's PDC : ', '');
NewUserName := InputBox('Create New User',
  'Please type in the user name : ', '');
```

Here, we assign the ADSI object path of the computer that we want to bind to:

```
// Assign AdsPath.
AdsPath := 'WinNT://' + PDCName + ',computer';
```

We then create the new *IADsContainer* object from the specified ADSI object path:

```
var
  ComputerObj: IADsContainer;
  TempUserObj: IUnknown;
  UserObj: IADsUser;
  PDCName: WideString;
  NewUserName: WideString;
  AdsPath: WideString;
begin
  // Retrieve information from user.
  PDCName := InputBox('Create New User',
    'Please type in the name of the Domain's PDC : ', '');
  NewUserName := InputBox('Create New User',
    'Please type in the user name : ', '');
  // Assign AdsPath.
  AdsPath := 'WinNT://' + PDCName + ',computer';
  // Create computer object.
  OleCheck(AdsGetObject(PWideChar(AdsPath),
    IID_IADsContainer, ComputerObj));
  // Create new user.
  TempUserObj := ComputerObj.Create('user', NewUserName);
  UserObj := TempUserObj as IADsUser;
  // Set information back to directory.
  UserObj.SetInfo;
  // Refresh list.
  actOpenWinNT.Execute;
```

Figure 6: Creating a new user.

```
// Create computer object.
OleCheck(AdsGetObject(PWideChar(AdsPath),
  IID_IADsContainer, ComputerObj));
```

We then call the *Create* method from the ADSI container object, which will create the new user and return the new user object back to the temporary ADSI object. We then assign the temporary interface object to the ADSI user object:

```
// Create new user.
TempUserObj := ComputerObj.Create('user', NewUserName);
UserObj := TempUserObj as IADsUser;
```

We then save the user information back to the WinNT directory:

```
// Set information back to directory.
UserObj.SetInfo;
```

Then we refresh the list to demonstrate the new user addition:

```
// Refresh list.
actOpenWinNT.Execute;
```

Removing a user from a computer is a similar task, except we don't need to create any user objects, and the user will call the *IADsContainer*'s *Delete* method (instead of the *Create* method), as shown in Figure 7.

The *Delete* method has two parameters: one is the type of ADSI object to delete; the second specifies the name of the user to delete:

```
// Create new user.
ComputerObj.Delete('user', UserName);
```

Viewing Users in Groups

With the WinNT provider, you can also control the NT groups by adding and removing users from the groups, and performing other group maintenance tasks. In the example application, I've decided to enumerate the group and view the users found within the groups in the specified domain. The code in Figure 8 binds to an *IADsGroup* object, and then uses an *IADsMember* object that is an enumeration object used to view and control the users within the *IADsGroup*. The code demonstrates listing the users found within different groups.

The first variable is *IADsGroup*, which will be used to represent and manage the NT group information from the directory. We'll use the *Members* property of the object, which is of type *IADsMembers*:

```
GroupObj: IADsGroup;
```

```
var
  ComputerObj: IADsContainer;
  PDCName: WideString;
  UserName: WideString;
  AdsPath: WideString;
begin
  // Retrieve information from user.
  PDCName := InputBox('Create New User',
    'Please type in the name of the Domain's PDC : ', '');
  UserName := InputBox('Create New User',
    'Please type in the user name to delete : ', '');
  if MessageDlg('Are you sure you want to delete user : ' +
     UserName + ' ?', mtConfirmation,
     [mbYes, mbNo], 0) = mrYes then
    begin
      // Assign AdsPath.
      AdsPath := 'WinNT://' + PDCName + ',computer';
      // Create computer object.
      OleCheck(AdsGetObject(PWideChar(AdsPath),
        IID_IADsContainer, ComputerObj));
      // Create new user.
      ComputerObj.Delete('user', UserName);
      // Refresh list.
      actOpenWinNT.Execute;
    end;
```

Figure 7: Removing a user from a computer.

```
var
  GroupObj: IADsGroup;
  Members: IADsMembers;
  AdsPath: WideString;
  Enum: IEnumVariant;
  TempUserObj: OLEVariant;
  UserObj: IADsUser;
  TempListObj: TListItem;
  Value: LongWord;
begin
  // Clear List.
  GroupListView.Items.Clear;
  // Assign AdsPath.
  AdsPath := 'WinNT://' + MainFrm.ADSIDomainName.Text +
             '/' + GroupName;
  // Create group object.
  OLECheck(AdsGetObject(PWideChar(AdsPath), IID_IADsGroup,
    GroupObj));
  // Assign members.
  Members := GroupObj.Members;
  // Assign enumerator object.
  Enum := (Members._NewEnum) as IEnumVariant;
  // Search through enumerator object.
  while (Enum.Next(1, TempUserObj, Value) = S_OK) do
    try
      // Assign temporary object.
      UserObj := IUnknown(TempUserObj) as IADsUser;
      // Create new list item.
      TempListObj := GroupListView.Items.Add;
      // Assign property.
      TempListObj.Caption := UserObj.Name;
    except
      on E:Exception do
    end;
```

Figure 8: Listing the users found within different groups.

Members is an interface variable for managing the list of users or ADSI objects with the NT group:

```
Members: IADsMembers;
```

The following WideString variable is going to store the ADSI object path of the group we're binding to, e.g. "WinNT://PDCCOMP/*Domain* Administrators":

```
AdsPath: WideString;
```

We declare the following variable so we can enumerate the child objects found within the *IADsMembers* container. We'll use it to search through the child objects in the container:

```
Enum: IEnumVariant;
```

This variable is a temporary interface variable used to store each ADSI object found in the *Members* variable list:

```
TempUserObj: OLEVariant;
```

We'll use the *IADsUser* variable to retrieve the end-user information for child objects found in the NT group. It will enable the application to represent and manage the end-user account on the domain:

```
UserObj: IADsUser;
```

We assign the ADSI object path of the group we want to bind to:

```
// Assign AdsPath.
AdsPath := 'WinNT://' + MainFrm.ADSIDomainName.Text +
           '/' + GroupName;
```

We then create the new *IADsGroup* object from the specified ADSI object path:

```
// Create group object.
OLECheck(AdsGetObject(PWideChar(AdsPath),
  IID_IADsGroup, GroupObj));
```

Next, we retrieve the NT groups member list, and assign it to the enumerator variable:

```
// Assign members.
Members := GroupObj.Members;
// Assign enumerator object.
Enum := (Members._NewEnum) as IEnumVariant;
```

Then we search through the list of NT users, and assign each member to the temporary interface variable:

```
// Search through enumerator object.
while (Enum.Next(1, TempUserObj, Value) = S_OK) do
```

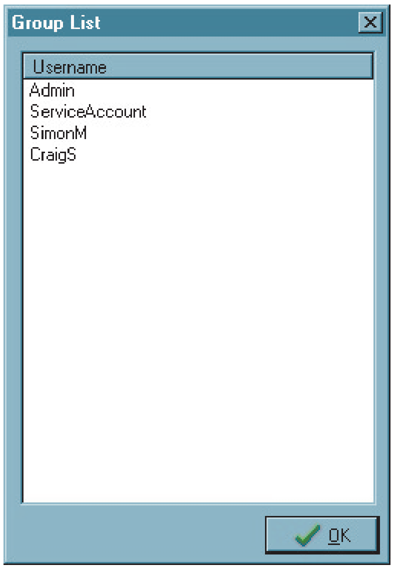
Next, we assign the temporary interface variable to the user object. Then we create the list item and assign the user's username to the list item:

```
// Assign temporary object.
UserObj := IUnknown(TempUserObj) as IADsUser;
// Create new list item.
TempListObj := GroupListView.Items.Add;
// Assign property.
TempListObj.Caption := UserObj.Name;
```

Once the search through the user list is complete, the result should look similar to Figure 9.

Controlling NT Services

The WinNT provider can also control NT services, NT servers, and workstations. The code in Figure 10 is similar to code used to retrieve the domain information. Instead of searching for users, groups, or computers, however, the code searches for NT services and assigns the services to the respective service list according to the computer selected in the computer list.



```
Username
Admin
ServiceAccount
SimonM
CraigS
```

Figure 9: Listing users from a group.

To retrieve information from NT services, we need to bind the ADSI object representing the service. ADSI provides the *IADsService* object, which allows us to maintain the information about the NT service running on its host computer. The code in Figure 11 binds to the respective NT service and displays the name and display name of the service.

First, an *IADsService* variable is declared to represent the NT service information from the directory:

```
ServiceObj: IADsService;
```

This WideString variable is going to store the ADSI object path of the NT service we're going to bind to, e.g. "WinNT://PDCCOMP/Messenger":

```
AdsPath: WideString;
```

We assign the ADSI object path of the service we want to bind to:

```
AdsPath := 'WinNT://' + ComputerName + '/' + ServiceName;
```

Then we create the new *IADsService* object from the specified ADSI object path:

```
// Assign Service Object.
OLECheck(ADsGetObject(PWideChar(AdsPath),
  IID_IADsService, ServiceObj));
```

We then display the properties retrieved from the respective NT service:

```
lblServiceName.Caption := 'Service Name : ' + ServiceName;
lblDisplayName.Caption := 'Service Display Name : ' +
  ServiceObj.Get_DisplayName;
```

To start and stop the NT service, you need to use the *IADsServiceOperations* ADSI object. You will bind to the NT service the same way as in the previous code, except that instead of returning an *IADsService* ADSI object, you must return an *IADsServiceOperations* ADSI object.

For example:

```
// Create Computer Object.
OleCheck(AdsGetObject(PWideChar(AdsPath),
  IID_IADsServiceOperations, Result));
```

Next, bind to the *IADsServiceOperations* ADSI object using the *GetServiceObj* function, which returns *IADsServiceOperations*:

```
// Assign service object.
ServiceObj := GetServiceObj;
```

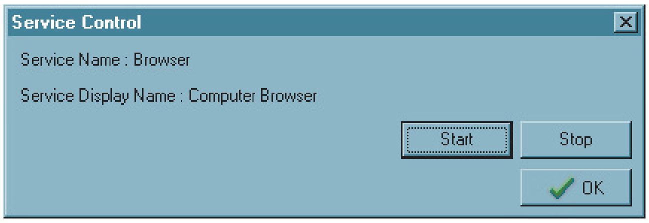
If the service is stopped, then call the *IADsServiceOperations Start* method, which starts the NT service:

```
var
  UnknownObject: IUnknown;
  Computer: IADsContainer;
  ComputerPath: WideString;
  Enum: IEnumVariant;
  AdsTempObj: OLEVariant;
  AdsObj: IADs;
  Value: LongWord;
begin
  if Item.Caption = '' then
    Exit;
  // Assign computer path.
  ComputerPath := 'WinNT://' + ADSIDomainName.Text +
                  '/' + Item.Caption;
  // Create computer object.
  OleCheck(ADsGetObject(PWideChar(ComputerPath),
    IID_IADsComputer, UnknownObject));
  // Assign computer object.
  Computer := UnknownObject as IADsContainer;
  // Remove items from list.
  ServiceListView.Items.Clear;
  // Assign enumerator object.
  Enum := (Computer._NewEnum) as IEnumVariant;
  // Search through enumerator object.
  while (Enum.Next(1, ADsTempObj, Value) = S_OK) do begin
    // Assign temporary object.
    ADsObj := IUnknown(ADsTempObj) as IADs;
    // If object is a service object then.
    if AdsObj.Class_ = 'Service' then
      AddServiceToList(ADsObj);
  end;
```

Figure 10: This code searches for NT services and assigns the services to the respective service list according to the computer selected in the computer list.

```
var
  ServiceObj: IADsService;
  AdsPath: WideString;
begin
  // Assign AdsPath.
  AdsPath := 'WinNT://' + ComputerName + '/' + ServiceName;
  // Assign Service Object.
  OLECheck(ADsGetObject(PWideChar(AdsPath),
    IID_IADsService, ServiceObj));
  // Assign labels.
  lblServiceName.Caption :=
    'Service Name : ' + ServiceName;
  lblDisplayName.Caption := 'Service Display Name : ' +
    ServiceObj.Get_DisplayName;
```

Figure 11: This code binds to the respective NT service and displays its name and display name.



Service Name : Browser

Service Display Name : Computer Browser

Figure 12: Controlling NT services.

```
// Start service.
if ServiceObj.Get_Status = 1 then
  ServiceObj.Start;
```

To stop the service, use the *Stop* method of the *IADsServiceOperations* ADSI object. First, as before, bind to the *IADsServiceOperations* ADSI object by using the *GetServiceObj* function found, which returns *IADsServiceOperations*:

```
// Assign service object.
ServiceObj := GetServiceObj;
```

If the service is running, call the *IADsServiceOperations Stop* method, to stop the NT service.

The dialog box in Figure 12 demonstrates the NT service functionality mentioned here:

```
// Start service.
if ServiceObj.Get_Status = 1 then
  ServiceObj.Start;
```

Conclusion

In this article, we explored the ways in which the ADSI directory services can be put to use. With the demonstration application included in this article, we were able to see functionality in the WinNT provider that makes life easier for developers and administrators.

ADSI is the way of the future for Windows 2000. It has done for directories what ADO did for databases. It's up to you to find ways of tapping into this resource and making it work to your benefit. ∆

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\MAY\DI200005SM.

Simon Murrell is currently the Senior Developer at Galdon Data, a well-known Microsoft Solutions Provider. He has used languages such as Delphi, VB, C++, Java, and JavaScript in his professional career. Simon can be reached at simonm@galdon.co.za.

By Eric Whipple

VisiBroker 3.3 for Delphi

Keeping Up with a CORBA World

I know what you're thinking: "Does the assembly method of development ever actually work, or is it just something I told the vice president to make him feel better about the value of distributed systems?" The assembly method, of course, refers to the creation of software by "gluing together" pre-made pieces of functionality to create a total development solution. By taking advantage of the flexibility and power of interface-based development, the CORBA (Common Object Request Broker Architecture) standard has brought us one step closer to this development dream.

The question so many Delphi developers struggle with is: "Can Delphi hack it in a CORBA world?" The answer is, "Yes." VisiBroker 3.3 for Delphi firmly establishes Delphi as a major tool for creating powerful, Windows-based CORBA clients.

In the Beginning ...

Before the arrival of VisiBroker 3.3, Delphi developers were already using Delphi to create CORBA clients and servers. Using the CORBA wizard to create a new CORBA object, and then using the Type Library editor to add a few methods and properties to a new interface, is, in fact, pretty easy. But in today's development community, a good CORBA client must be able to play well with others.

The biggest barrier to creating a CORBA solution with Delphi is that there's no simple way to create a Delphi client for a CORBA server written in Java or C++. If both the client and server are written in Delphi, you can simply add the *xxx*_TLB.pas file from the server object to the client project, and not worry about the details of CORBA communication. But what if the server wasn't written in Delphi? One of the fundamental advantages of CORBA is that it is language-independent. Until the release of VisiBroker 3.3, however, Delphi did not provide a way to create a CORBA client based on existing interfaces, written with other development tools. Instead, you were forced to manually code the client's communication with the ORB. And that can take you further into the guts of CORBA than you want to go.

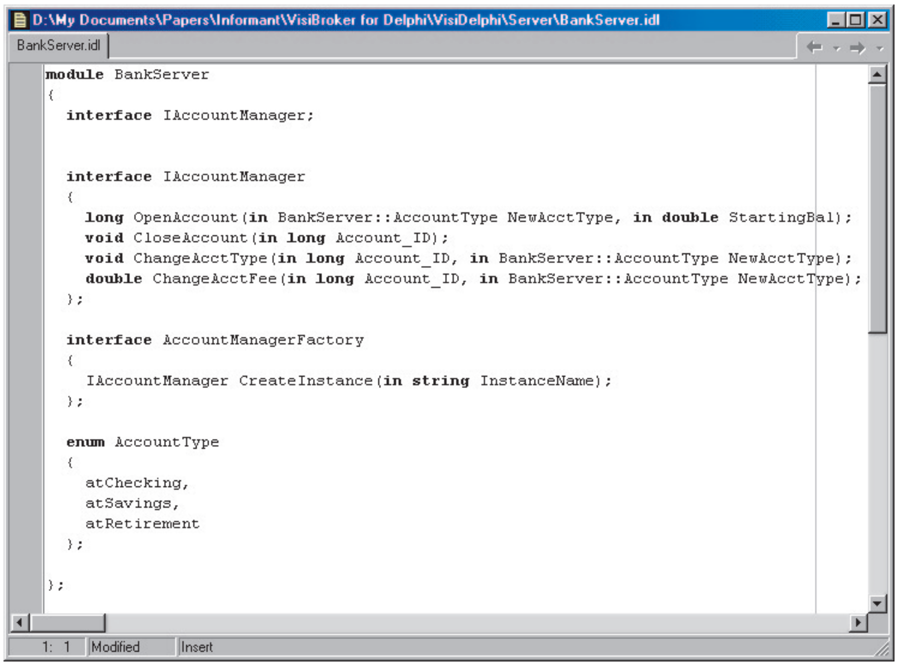
A Trip to the Bank

To see just how useful Delphi can be, let's look at a real-world example of VisiBroker. A major problem facing larger companies today is how to deal with corporate inertia. Strangely, Webster's doesn't define corporate inertia, but if it did, it would probably look something like this:

***cor.po.rate in.er.tia** \'kor-pë-rët in-'ër-shë\ n - A principle by which a corporation in a certain technological state will tend to remain in that state unless acted upon by an external force. The magnitude of the inertia is directly proportional to the mass of the corporation that it acts upon.*

As an example, imagine that you're the IT manager for a regional group of banks owned by First American Bank of Chicago. First American's main offices are located in Chicago, of course, but you're managing one of its recently acquired subsidiaries in the southeast. Your local branches are equipped with the latest in ATM hardware, which supports Windows-based applications. First American requires you to communicate with their CORBA servers, but has not developed latest-generation ATM client software because many of its older subsidiaries can't purchase or run the applications. It's your responsibility to create a quick and easy client application to connect to the First American server.

In the past, First American's corporate inertia would have caused a serious problem for your Delphi developers. Because Delphi had no way to create client code based on existing interface declarations, you would've been forced to develop your Delphi CORBA clients from scratch by hand. This includes the creation of marshalling and stub code for every implementation object, which is tedious to say the least. Even if your developers are intimately familiar with Delphi and CORBA, the cost of writing the application (in developer hours) alone could prohibit you from realistically attempting it (not to mention debugging it).

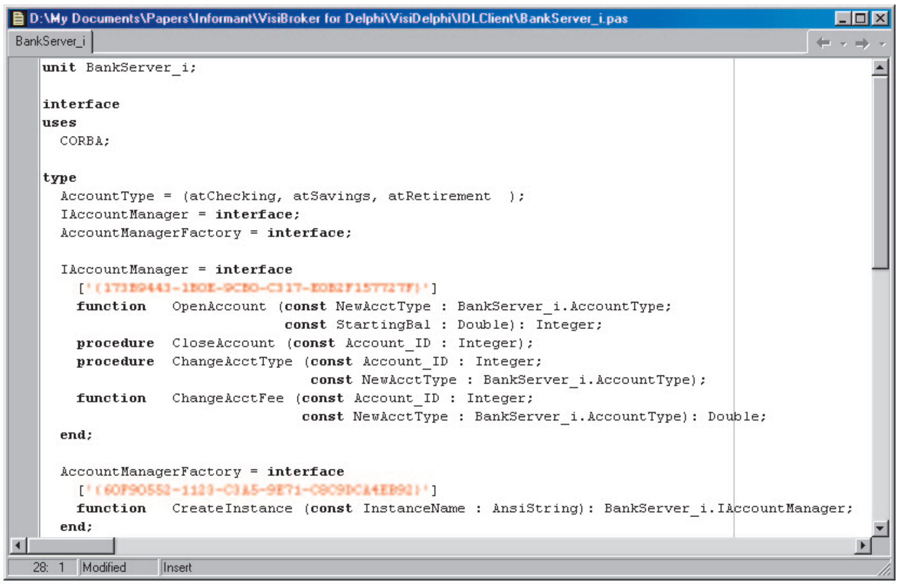


D:\My Documents\Papers\Informant\VisiBroker for Delphi\VisiDelphi\Server\BankServer.idl

```
module BankServer
{
  interface IAccountManager;


  interface IAccountManager
  {
    long OpenAccount(in BankServer::AccountType NewAcctType, in double StartingBal);
    void CloseAccount(in long Account_ID);
    void ChangeAcctType(in long Account_ID, in BankServer::AccountType NewAcctType);
    double ChangeAcctFee(in long Account_ID, in BankServer::AccountType NewAcctType);
  };

  interface AccountManagerFactory
  {
    IAccountManager CreateInstance(in string InstanceName);
  };

  enum AccountType
  {
    atChecking,
    atSavings,
    atRetirement
  };

};
```

1: 1 Modified Insert

Figure 1: The BankServer.Idl file.



D:\My Documents\Papers\Informant\VisiBroker for Delphi\VisiDelphi\IDLClient\BankServer_i.pas

```
unit BankServer_i;

interface
uses
  CORBA;

type
  AccountType = (atChecking, atSavings, atRetirement  );
  IAccountManager = interface;
  AccountManagerFactory = interface;

  IAccountManager = interface
    ['{173B9443-1B0E-9CB0-C317-E0B2F157727F}']
    function   OpenAccount (const NewAcctType : BankServer_i.AccountType;
                            const StartingBal : Double): Integer;
    procedure  CloseAccount (const Account_ID : Integer);
    procedure  ChangeAcctType (const Account_ID : Integer;
                               const NewAcctType : BankServer_i.AccountType);
    function   ChangeAcctFee (const Account_ID : Integer;
                              const NewAcctType : BankServer_i.AccountType): Double;
  end;

  AccountManagerFactory = interface
    ['{60F90552-1123-C3A5-9E71-C8C9DCA4EB92}']
    function   CreateInstance (const InstanceName : AnsiString): BankServer_i.IAccountManager;
  end;
```

28: 1 Modified Insert

Figure 2: The BankServer_i.pas file after some condensing.

A Better Way to CORBA

VisiBroker for Delphi allows you to automate the creation of CORBA clients for existing CORBA servers by creating client-side stubs based on interfaces and types defined in the server's IDL file(s). IDL (Interface Definition Language) is a generic language for specifying CORBA-compliant interfaces and complex data structures. It can be generated by almost any development tool that allows the creation of CORBA objects, e.g. Delphi. Because most IDEs can generate IDL, Delphi can be used to create clients for any CORBA object, no matter what language it's written in. In our bank example, VisiBroker for Delphi relieves you of the burden of First American's corporate inertia.

VisiBroker for Delphi uses an IDL2PAS compiler to create client-side Pascal CORBA code from any IDL file. In other words, IDL2PAS creates client-side stubs (based on interface and object definitions found in the IDL file) that allow a client application to communicate with any CORBA object that supports the listed interface. The key is that no other information (aside from the IDL file) is required from the server.

Inside VisiBroker 3.3

It's important to note that VisiBroker for Delphi is not a Delphi ORB (object request broker); that is, it's not written in Delphi. It uses the same ORB found in VisiBroker for C++. A special DLL, orbpas33.dll, provides a wrapper around the C++ library, orb_br.dll. The main reason for this approach is that the C++ ORB is already one of the leading CORBA ORBs out there, and it avoids the cost and timing issues involved in writing a Delphi ORB from scratch. (More information on this topic can be found in the VisiBroker for Delphi documentation.)

Probably the first thing you'll notice about VisiBroker for Delphi is that it's not installed as part of the Delphi environment. In fact, it's run from a DOS window. The IDL2PAS command can be used with a variety of flags to tweak its behavior. After installing the product, you'll find two new files in your bin directory: IDL2PAS.bat and IDL2PAS.jar. The batch file is simply used to run the jar file on whatever IDL file you supply. As you might have guessed, you need a Java Virtual Machine (JVM) to run IDL2PAS. Most Windows machines have at least one JVM installed, so this shouldn't be a problem.

Although the concept of IDL2PAS is fairly simple, the importance of this tool cannot be overstressed. Now if a pre-packaged Java CORBA server exists, all you need to write a customized client are the IDL files to the CORBA objects that you need to talk to. Because IDL is a generic language, it doesn't matter what language the server was written in. Let's see how this new tool affects our bank example.

We start by executing the IDL2PAS command on BankServer.Idl by opening a DOS window and typing the following:

```
IDL2PAS BankServer.Idl
```

The BankServer.Idl file is shown in Figure 1. IDL2PAS generates two new Pascal files: BankServer_i.pas and BankServer_c.pas. The BankServer_i.pas file has a specific purpose. Its job is to declare (in Pascal) all of the interfaces and any types defined in the IDL file. Although the file appears large, it's likely to contain more comments than code. Each line from the IDL file is identified with a comment in the resulting Pascal interface definition.

After some condensing, the BankServer_i.pas file looks like the code shown in Figure 2. Notice that our *IAccountManager* has magically become two interfaces. Not only is there a Pascal declaration for our initial interface, there is also a factory interface, named *AccountManagerFactory*.

Again, I know what you're thinking: Why isn't it named *IAccountManagerFactory?* The notion that all interfaces should begin with the letter "I" is really more of a COM standard. Because Delphi supports both COM and CORBA, it always puts the "I" prefix on the servers it creates, whether COM or CORBA. VisiBroker, of course, is a CORBA implementation and therefore does not use the same convention.

The *AccountManagerFactory* interface is very simple. The job of any factory object is to generate instances of another object (in this case, *IAccountManager*). Its declaration is shown in Figure 2. It contains a single method named *CreateInstance*, which returns a reference to an *IAccountManager* object.

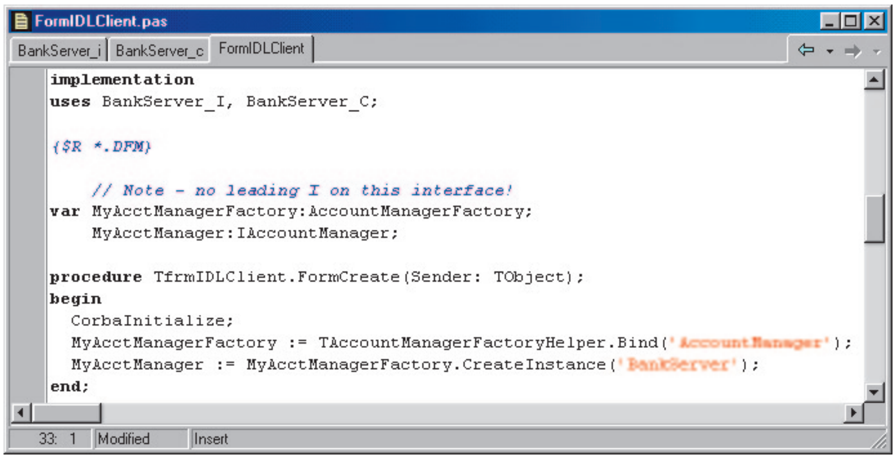
Looking again at Figure 1, we see that the IDL file contains an **enum** declaration named *AccountType*. This **enum** type will be turned into a Pascal type declaration. However, when Delphi creates IDL, it always puts the enumerations at the bottom of the file. If there are method declarations that pass parameters of that type, the IDL2PAS command will fail. This is because at the time the compiler tried to resolve the parameter type, the type had not yet been declared. The simple solution is to open the IDL file and move the **enum** declaration to the top.

The second file generated by IDL2PAS is the BankServer_c.pas file. This file holds all of the client-side stub code for communicating with CORBA objects on the server. This file is similar in nature to the implementation section of the traditional xxx_TLB.pas file. A big difference between the two, however, is the virtual absence of unnecessary COM structures, such as CoClasses. You will notice some COM-like items, such as references to the *QueryInterface* method, but according to documentation, this is only a thin layer:

"... the *TCorbaObject* class definition has a COM-style with a *queryinterface*, *addref* and *release* methods and a GUID. In reality, however, it only uses COM for automated garbage collection of the stub objects. Under the covers the *TCorbaObject* implementation substitutes CORBA-based calls in place of the operating system COM calls."

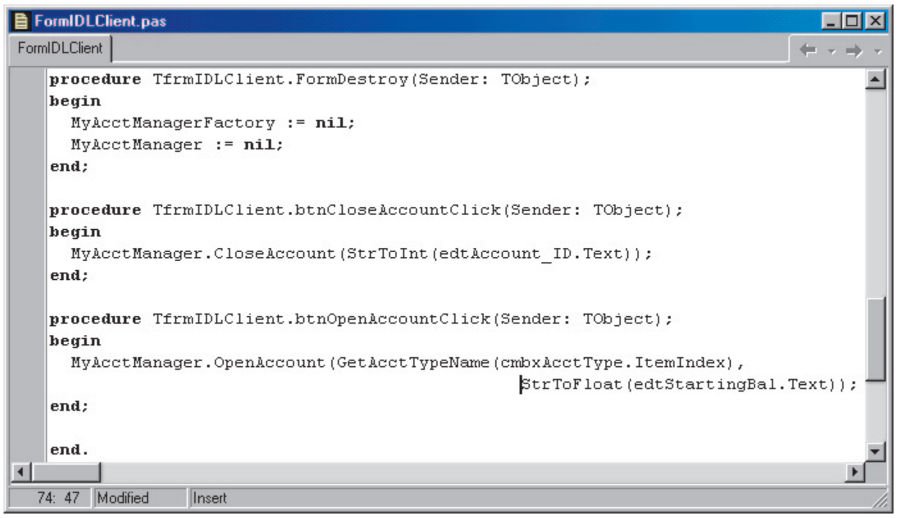
The *xxx*_c.pas file typically contains two types of classes: the helper class and the stub class. The helper class (found in the *xxx*_c.pas file) allows a client to make calls with non-primitive parameter types (structs, arrays, and other non-standard types defined by the developer) and allows for the typecasting of returned objects via the *Narrow* method. In addition, the helper class contains two versions of the all-important *Bind* method. This is what allows a client to create a connection to a CORBA server. More information on the helper class can be found in the VisiBroker for Delphi documentation. The stub class is used to marshall method calls to the ORB. You'll see very similar method calls for each member of your interface as the stub class creates, formats, and invokes requests and responses. It's unlikely that you'll need to directly edit the contents of this class.

VisiBroker for Delphi also contains a program named IDL2IR. The IDL2IR command populates an interface repository with



```
implementation
uses BankServer_I, BankServer_C;

{$R *.DFM}

    // Note - no leading I on this interface!
var MyAcctManagerFactory:AccountManagerFactory;
    MyAcctManager:IAccountManager;

procedure TfrmIDLClient.FormCreate(Sender: TObject);
begin
  CorbaInitialize;
  MyAcctManagerFactory := TAccountManagerFactoryHelper.Bind('AccountManager');
  MyAcctManager  := MyAcctManagerFactory.CreateInstance('BankServer');
end;
```

Figure 3: Bind to a factory object and request an *IAccountManager* object.



```
procedure TfrmIDLClient.FormDestroy(Sender: TObject);
begin
  MyAcctManagerFactory := nil;
  MyAcctManager := nil;
end;

procedure TfrmIDLClient.btnCloseAccountClick(Sender: TObject);
begin
  MyAcctManager.CloseAccount(StrToInt(edtAccount_ID.Text));
end;

procedure TfrmIDLClient.btnOpenAccountClick(Sender: TObject);
begin
  MyAcctManager.OpenAccount(GetAcctTypeName(cmbxAcctType.ItemIndex),
                                           StrToFloat(edtStartingBal.Text));
end;

end.
```

Figure 4: Then, we can begin making calls to the server's *IAccountManager* interface.

objects and other constructs contained in an IDL file. An interface repository contains information about the ORB and any objects with which it is currently communicating. This can be very useful for dynamic binding.

Building a Client

The downloadable code for this article (see end of article for details) contains a CORBA server and a traditional client, as well as an IdlClient application. Let's take a look at how to use the *xxx*_i.pas and *xxx*_c.pas files. The client application consists of a simple form that's used to call methods on the server. After running IDL2PAS on BankServer.Idl, the resulting BankServer_i.pas and BankServer_c.pas files are used by the main form of the client application (notice that we need nothing from the server once we have the IDL file). Once we've called *CorbaInitialize*, all that remains is to bind to a factory object and request an *IAccountManager* object (see Figure 3). Once that's done, we can begin making calls to the server's *IAccountManager* interface (see Figure 4).

Comparing the two client programs, we see there isn't much difference in terms of time or complexity. The difference, of course, is that if the server had not been written in Delphi, the IdlClient would still work as-is, while the standard client would have required a good bit of work.

The Dreaded "Known Issues" Clause

The first version of any tool always includes some "known issues." It would be misleading to say that VisiBroker 3.3 for Delphi is any different. Two of the most notable issues include the fact that VisiBroker for Delphi only helps on the client side; it cannot create server objects based on IDL files. Because of this, the technique of CORBA callbacks is not supported. According to the documentation, long-term goals include a more complete mapping of the C++ ORB, as well as support for CORBA callbacks and server-side code.

Conclusion

VisiBroker 3.3 for Delphi significantly strengthens Delphi's expanding capability to create flexible and global solutions in what is fast becoming a CORBA world. Its ability to integrate with prepackaged CORBA components in a language-independent environment will allow Delphi to help set the standards in a young and powerful market. Δ

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\MAY\DI200005EW.

Eric Whipple is a Delphi trainer and mentor for Pillar Technology Group, Inc. of Detroit, a full-service consulting, training, and mentoring firm specializing in project management and in the analysis, design, and development of distributed, enterprise systems (http://www.knowledgeable.com). Eric is a Delphi 4-certified developer and trainer, and can be reached at ewhipple@knowledgeable.com or by phone at (317) 915-9031.

Some Comments on Commenting

Do you comment your code? Commenting is one of those topics upon which you can hear conflicting opinions from various programming experts — pseudo, self-proclaimed, or legitimate. Some say you should always comment your code. Others say, basically, that commenting is for ninnies and newbies; it's unnecessary for "real" programmers, because you should be able to read the code and determine what it's doing.

In the classic book *Code Complete* by Steve McConnell (Microsoft Press, 1993), "Socrates" is quoted as saying, "I think that people who refuse to write comments 1) think their code is clearer than it could possibly be; 2) think that other programmers are far more interested in their code than they really are; 3) think other programmers are smarter than they really are; 4) are lazy; or 5) are afraid someone else might figure out how their code works."

After maintaining many other people's code on several different projects, I agree with Socrates and am firmly entrenched in the commenting-is-a-good-thing camp. But not just any kind of commenting. Many reluctant commenters end up producing worthless comments, thus completing a self-fulfilling-prophecy.

Those who object to commenting often say that: 1) they don't have time for commenting; or 2) all you need to do is read the code to see what it's doing.

Let me respond to these objections in turn. First, in regard to having no time for commenting ("I'm too busy doing the real work of coding"): If you don't have time for commenting your code at the time you write it, when will you have time for it? Using PDL (programming design language; see *Code Complete* for details and examples) to delineate what a section of code should do, and adjusting and/or enhancing it immediately following coding, when the logic (and the reason for it) is still fresh in your mind, is a coding "best practice." As in analysis and design, this work done up front will, in the long run, *save* you time.

As to comments being unnecessary because the code explains itself: The problem here is a failure to understand the true nature and purpose of commenting. It's true that low-level comments (at least, if the code has been written as cleanly and simply as possible) shouldn't be necessary. What comments should convey is a high-level view of the code. In layman's terms: what does it do? why? when is it called and by whom? and how does it fit in with the overall body of code?

Viewed in this light, commenting can be likened to news reporting: Be sure to convey who, what, why, where, when, and how. Done right, commenting is an integral part of the coder's responsibility (to the coder, maintenance programmers who will have to work on the code, and the coder's employer or client).

Not all comments are good comments, however. For example, this type of comment:

```
// Loop through the doohickeys.
for i := 0 to Doohickeys.Count-1 do
  sl.Add(Doohickey[i]);
```

is superfluous. It's obvious from looking at the code that it's looping through the doohickeys. Any programmer can see that.

Here, on the other hand, is a good example of commenting. The reader is provided with information that's neither transparent nor trivial:

```
function AddLeading0 s(ADate: string): string;
begin
  // If "m/", make it "mm/".
  if (Pos('/',ADate)=2) then begin
    ADate := '0 '+ADate;
    Result := ADate;
  end;
  // If now "mm/d/", make it "mm/dd/".
  if (ADate[5] ='/') then
    Result := Copy(ADate,1,3)+'0 '+Copy(ADate,4,4);
end;
```

To make things crystal clear, include in the comments examples of input and output. For example:

```
// Example: 991231 is received and converted to 12/31/99.
function ConvertFromYYMMDDToMMDDYY(
  ADateToConvert: string): string;
begin
  // DateToConvert should be length of 6 (YYMMDD).
  Assert(Length(DateToConvert)=6,
         SDatePassedOfUnexpectedLength);
  Result := Format('%s/%s/%s',[Copy(ADateToConvert, 3, 2),
                   Copy(ADateToConvert, 5, 2),
                   Copy(ADateToConvert, 1, 2)];
end;
```

Adding a history of modifications can also be beneficial. If a block of code breaks, consulting the revision history can help you determine what was changed and when. It's also a good idea to sometimes simply "comment out" replaced code, rather than deleting it, especially if a simpler version of the code was replaced by a more complicated (although perhaps more efficient) version.

Besides commenting procedures and functions, it is also helpful to add high-level comments to units (see Figure 1). In this way, a maintenance programmer (the original programmer, or someone else) can quickly determine the general purpose and functionality of the unit. Unit header comments can also follow the reportorial style of answering (where appropriate) who, what, why, when, where, and how: Who wrote it (so they know to whom they should direct questions); what is the purpose of the unit; why any "unusual" practices were followed, and/or what factors influenced the decision-making process regarding the architecture and

```
{ *************************************************************** }
{                                                                 }
{  BSaleObj (unit)                                                }
{  No variable/instance name. Aggregation used in dm (this        }
{  object is encapsulated in that one as a private field).        }
{                                                                 }
{  This is the custom report object for the "Billed Sales         }
{  Report by Sales Rep" and "Billed Sales Credit Report by        }
{  Sales Rep" report. These are two quite dissimilar reports,     }
{  but happen to be accessed from the same interface.             }
{                                                                 }
{  See NBFCWO 17                                                   }
{                                                                 }
{  "BILLED SALES REPORT BY SALES REP" portion:                    }
{  "Order Number" is extracted from Arledger.OrderNum and         }
{   ARLedger.VendorCode                                           }
{                                                                 }
{   "Bill-To" is extracted from Customer.Company based on         }
{   ARLedger.CustNum                                              }
{                                                                 }
{   "Margin" is calculated - (CustOrder - InvoicedCost)           }
{                                                                 }
{   "Profit %" is calculated -                                    }
{    (((CustOrder-Cost) / CustOrder)*10 0 )                         }
{                                                                 }
{   . . . additional notes left out for brevity                   }
{                                                                 }
{  Coded by Rupert "Ruprecht" Pupkin III                          }
{                                                                 }
{  Revision History:                                              }
{    0 5/27/1999   Began coding:                                   }
{    0 9/28/1999.  It has been decided that the "Cost X Factor"   }
{    and "Cost" columns are not needed. Rather than code them     }
{    out, it will be faster to simply suppress them from          }
{    printing. A new column "Sell Price"*, will take the place    }
{    (in the string list) of "Cost".                              }
{    * Cost * Quantity.                                           }
{                                                                 }
{ ************************************************************* }
```

Figure 1: An example of a unit comment.

implementation; what other units use this one; how the various methods are used, etc.

You can make it easier on yourself to add unit and method comment headers if you create templates for them. You can do this by following these steps:

1) Select **Tools | Editor Options**.
2) Select the Code Insight tab.
3) Select the **Add** button in the Code Templates section.
4) Provide a **Shortcut Name** and a **Description** (such as `Uhdr` and `Unit Header`).
5) Select the **OK** button.
6) Type in your boilerplate placeholders, for example:

```
{
 Name of unit

 Purpose of unit

 Anything unusual

 Coded by:

 Revision history:
    Began coding
 }
```

7) Select the **OK** button.
8) Repeat the process, this time creating a template for method headers.

You can now insert these templates in the code editor by pressing Ctrl J and selecting the desired template (I guess the J stands for "Just when you thought everything in Delphi made sense...).

Albert Einstein said that if you can't explain something to an eight year old, you don't really understand it yourself. Commenting your code lets you prove that you truly understand your own code. Δ

— Clay Shannon

Clay Shannon is a Delphi developer for eMake Corporation, located in Post Falls, Idaho. Having visited 49 states (all but Hawaii) and lived in seven, he and his family have settled in northern Idaho, near beautiful Coeur d' Alene Lake. He has been working (almost) exclusively with Delphi since its release, and is the author of the book Developer's Guide to Delphi Troubleshooting *(Wordware, 1999). You can reach Clay at clayshannon@usa.net.*

The Future of Computing: Preparing for Delphi for Linux (cont. from page 36)

writers I enjoy reading as much as Swan, so I recommend this book highly.

The last book is probably the most unusual. *Maximum Linux Security: A Hacker's Guide to Protecting Your Linux Server and Workstation* (SAMS, 1999) lists no author. What's going on here? Is the content so controversial that the author must disguise his or her identity? Perhaps. It's well known that some of the most successful security experts have had experience with hacking into seemingly secure networks. This work is particularly well researched, with a wealth of specific examples of security setups that failed.

The opening section provides basic information on Linux security, including the role of the system administrator. The second section concentrates on the Linux user and deals with password and code (virus) issues. The third section explains techniques that apply to networks, while the final and longest section covers the Internet. Because Linux is particularly popular in running network servers, this book may be the most useful to Delphi developers looking forward to Delphi for Linux. It's informative and entertaining; I recommend it highly.

Many developers have contributed to the growth of Linux as one of the most highly regarded operating systems. The Internet has been crucial to this development. Next month we'll conclude this three-part series with a look at the Linux Internet sites as we prepare for Delphi for Linux. Δ

— Alan C. Moore, Ph.D.

Alan Moore is a Professor of Music at Kentucky State University, specializing in music composition and music theory. He has been developing education-related applications with the Borland languages for more than 10 years. He has published a number of articles in various technical journals. Using Delphi, he specializes in writing custom components and implementing multimedia capabilities in applications, particularly sound and music. You can reach Alan on the Internet at acmdoc@aol.com.

The Future of Computing: Preparing for Delphi for Linux

Last month, we explored some of the issues related to the eagerly anticipated Delphi for Linux. This month, we'll continue that discussion and take a look at five books that might be helpful in preparing for this momentous event. First, I'd like to share a message I received from a reader regarding my January 2000 column on Delphi and Visual Basic.

"[In your article] I think you left out one of the most important points for using Delphi: it will be ported to Linux. Microsoft (MS) will never do this. Therefore as the world evolves away from MS's buggy software there will be no MS competition. Once Inprise ports [Delphi to Linux] there will be Corel [WordPerfect] Suite on the desktop with Paradox as a desktop database and Delphi as your enterprise solution. Where will MS be? Who knows and who cares. Some people say MS will never loose the desktop. Once consultants realize they can pocket 50% of the money that MS charges for their OS and still leave the client with a surplus it won't take long for the massive defection."

Interestingly, I received this the day after I submitted my previous column. Will these predictions come to pass? Who can say? However, it is clear that Linux will provide increasing competition for Microsoft Windows in the coming years. Windows and Linux do have one interesting thing in common: Most of the high-end programming for each is written in C. So in this column, I will discuss both the general books introducing Linux and books about programming for Linux using C.

In writing last month's column, I found one book particularly helpful in developing a general understanding of Linux: *Caldera OpenLinux 2.3 Unleashed* by David Skoll (SAMS, 1999). If you're looking for an excellent introduction to Linux, this is worth examining. The first part covers all the basic information about Linux, from the historical information I included in my previous column, to working with the Linux shell, to Linux text editors. It also includes an introduction to the X Windows system, Linux's graphical user interface. Many developers will be interested in using Linux on a server. For them, the second section dealing with System Administration will be of particular interest. This work covers all the basic tasks, including working with file systems, printing, setting up a TCP/IP network, and configuring a system, among many others.

However, *Linux Programming Unleashed* by Kurt Wall, et al. (SAMS, 1999) is written specifically for developers and deals with advanced programming topics such as input/output, memory management, and communication. Not surprisingly, the emphasis is on C/C++; however, there are sections on Java and other languages. The first section of the book provides directions for setting up a complete development environment, including using the Make tool to build large applications, writing software that can be automatically configured, establishing a version-control system, using programming macros (with Emacs), and so on. Although there is no CD-ROM, there are some code examples and many useful links to Linux programming sites. (The remaining books include CD-ROMS.)

If you want to get up to speed quickly using C for Linux, *SAMS Teach Yourself C for Linux Programming in 21 Days* by Erik de Castro Lopo, et al. (SAMS, 1999) is ideal. It assumes little or no knowledge of programming and starts from the basics. In the first week you learn about Linux and C, including the elements of a C program. There are separate chapters (days) teaching program control, syntax elements, and structure, such as expressions and statements. In the second week you explore more advanced aspects of the language, including pointers, strings, structures, scope, and more advanced program control techniques (including loops). The last week introduces practical examples, such as using libraries, working with memory and disks, and exploring several advanced topics. This book is particularly appropriate for the novice. The next book is better for intermediate-level programmers.

Tom Swan is probably one of today's best known computer programming writers, having written over thirty popular works. *Tom Swan's GNU C++ for Linux* (QUE, 1999) is an excellent introduction, providing detailed coverage of the essential topics, from setting up your system to working with the X Windowing environment. As with the previous title, this book begins with the basics. However, it provides more detailed information, accompanied by a plethora of useful tips.

After an introductory section, Swan provides the basics of C/C++ programming. Section three explores the details of object-oriented programming. Among the useful topics here are techniques for handling exceptions. Sections four and five introduce advanced C++ techniques and C++ class libraries, respectively. Section six introduces the X Windowing environment. Section seven includes five appendices that provide additional information. There are few

"The Future of Computing: Preparing for Delphi for Linux"
continued on page 35