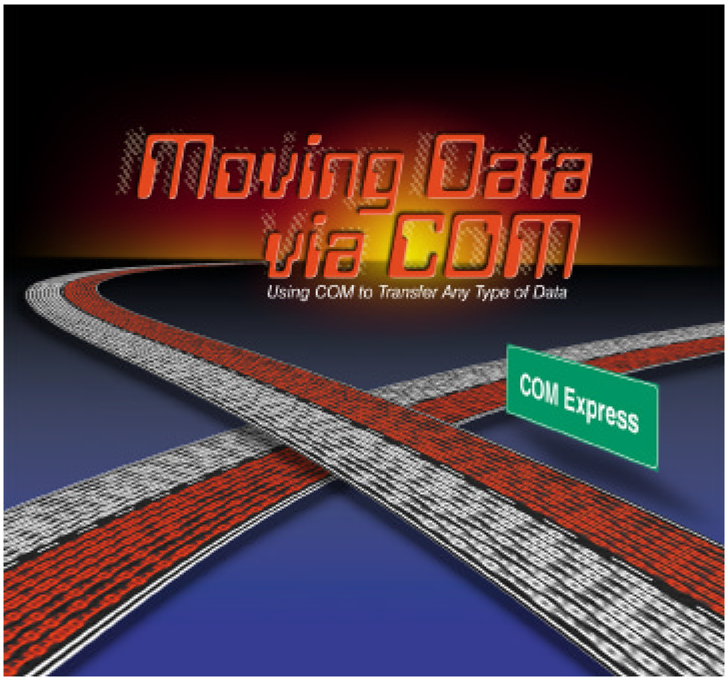
November 2000, Volume 6, Number 11



Cover Art By: *Arthur Dugoni*

Delphi TOOLS

New Products and Solutions

Tenon Announces iTools for Linux

Tenon Intersystems announced *iTools for Linux*, its suite of Internet tools for the Linux operating system designed to simplify Apache configuration and maintenance. iTools extends and enhances Linux's networking performance, efficiency, ease-of-use, and functionality with a family of tools essential to commercial content delivery and e-commerce.

iTools complements and extends the open source software that is included with Linux, and makes it easier for Webmasters to set up and support sophisticated network servers. iTools, based on open-source implementations of Apache, DNS, FTP, and sendmail, is created and maintained by software developers worldwide. Using it as a point-of-departure, iTools extends the underlying architecture with a point-and-click interface and a set of new features. iTools includes a fully customizable, anywhere, anytime WEBmail server, a two-tiered-certificate-savvy SSL encryption engine to support e-commerce, a caching engine with proxy support, a search engine, and FastCGI and mod_perl support. All tools are supported using a point-and-click, browser-based administration tool, enabling system administrators and Webmasters to manage their Web servers remotely using any Internet browser.

iTools runs on Linux PPC 1999/2000, Terra Soft's Yellow Dog Linux Champion Server 1.1/1.2, and Red Hat Linux 6.1 for Intel.

Tenon Intersystems
Price: US$199 (introductory price).
Phone: (800) 662-2410
Web Site: http://www.tenon.com

Paradigma Announces Valentina COM

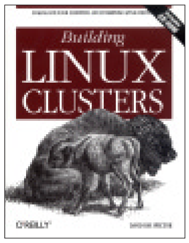
Paradigma Software announced *Valentina COM*, the implementation of the Valentina object relational database for users of COM-compliant development environments. With Valentina COM, developers that use COM-compliant development environments, such as Delphi and Macromedia Authorware, can embed Valentina COM into their applications.

Valentina implements the object-relational (OR) data model (a theoretical development of Paradigma), which is an extension of the traditional relational data model. The OR model embeds the relational model as an exact subset. Everything that works in a relational database management system (RDBMS) must work in Valentina, so you can smoothly switch from the familiar RDBMS model to the Valentina model.

Valentina databases support generic BLOb fields as well as two special BLObs: BLOb-TEXT and BLOb-Picture fields. You can automatically utilize JPEG compression to maximize storage of graphics and create cross-platform graphics databases suitable for use on the Internet.

Paradigma Software
Price: US$199.95
Phone: (503) 520-0191
Web Site: http://www.paradigmasoft.com

Building Linux Clusters
David HM Spector
O'Reilly & Associates



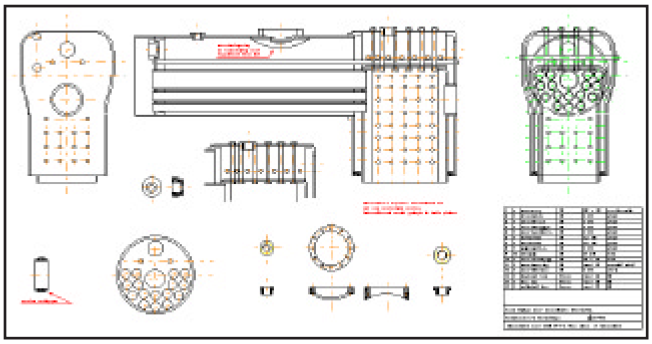
ISBN: 1-56592-625-0
Price: US$44.95
(332 pages, CD-ROM)
Web Site: http://www.oreilly.com

ProWorks Releases Flipper CAD Control 2.5

ProWorks LLC announced the release of *Flipper CAD Control 2.5*, the newest version of the company's ActiveX drawing control (.DLL) that adds a two-dimensional drawing canvas to any program that supports ActiveX/COM technology.

An assortment of shape types can be drawn and manipulated, including line, rectangle, ellipse, polyline, polygon, freehand, spline, arc, pie slice, chord, text, bitmap, and more. Shapes can be combined into a group object, which is useful for creating more complex objects.

The drawing canvas is made up of one or more layers, which are transparently stacked on top of each other. Each layer can have unique pen and brush settings. In addition, developers can modify layer settings to limit the amount of interaction the user has with shapes. Shape display, moving, and resizing can all be turned on or off.



Flipper CAD Control can be embedded in a client Web page and can load CAD drawings and DXF files from a server across the Internet. Included with this version is a signed .CAB file, allowing client machines to download a run-time version of the control for use on a Web page. The control also integrates into ASP pages. Flipper CAD Controls' integration with run-time dialog boxes provides a GUI for editing layer and shape properties.

A variety of mouse pointers for various actions within the control are provided. Custom mouse pointers can be loaded into the control as well.

ProWorks LLC
Price: US$349
Phone: (541) 752-9885
Web Site: http://www.proworks.com

InstallShield Unveils InstallTuner for Windows Installer

InstallShield Software Corp. unveiled *InstallTuner for Windows Installer*, the company's latest advancement in software customization for network professionals deploying Microsoft's Windows Installer applications.

InstallTuner is specifically designed for network administrators who need to modify Windows Installer applications (.msi) for customized deployment to end users using a simple GUI-driven, administrator-focused environment.

Key features include pre-validation of the original MSI package, which validates that .msi packages comply with Microsoft's Windows 2000 logo criteria before customization; setup organization, which allows administrators to set product properties such as the destination variable or the destination directory, set application features, and decide how features will be presented to the end user in the Custom Setup dialog box; target system configuration, which provides the ability to add additional files and modify existing registry information to an .msi-based installation; application configuration, which provides the ability to add or modify properties that affect the application setup, as well as specify properties for the Windows 2000 Add/Remove Programs applet in the Control Panel; post-validation, which allows validation of the Windows Installer package in conjunction with any modifications made via InstallTuner transforms; and the table viewer, which opens Installation Development Environment tables for Windows Installer packages and displays both standard tables and custom tables.

InstallShield Software Corp.
Price: Not available.
Phone: (800) 374-4353
Web Site: http://www.installshield.com

VideoSoft Announces VSVIEW 7.0

VideoSoft announced the release of *VSVIEW 7.0*, the latest version of its Visual Basic printer engine replacement. VSVIEW 7.0 allows developers to add print preview features to Windows-based applications.

VSVIEW 7.0 includes full export support to RTF and HTML while retaining all paragraph and font formatting. The HTML export filter also allows for paged HTML output to generate a series of hyperlinked pages.

Other new features of VSVIEW 7.0 include URL Download, which allows users to download documents on a Web page directly into a Web browser for printing on a local or network printer; expanded table support, which allows the creation of tables with rows that span page breaks, row and column spanning (cell merging), vertical text in table cells, and custom borders on a per-row/per-column basis; improved document navigation; new printer options, which allow improved scaling, cropping, and alignment; and retained styles.

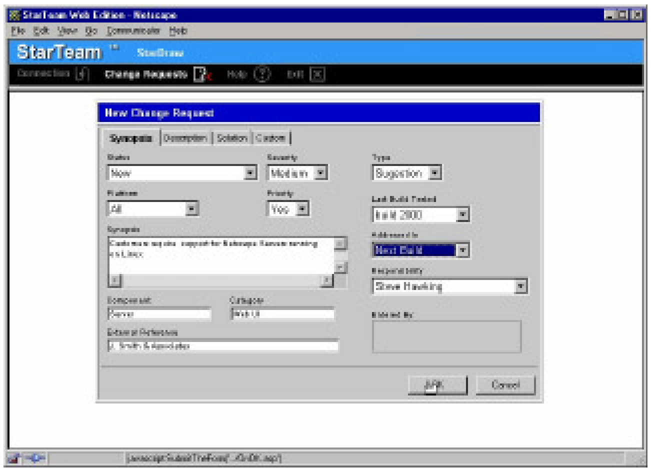
VideoSoft
Price: US$299
Phone: (888) ACTIVEX or (510) 595-2400
Web Site: http://www.videosoft.com

Starbase Announces StarTeam Web Edition

Starbase Corp. announced *StarTeam Web Edition*, which enables direct access to software and Web development projects from anywhere in the world through a secure Web browser. Web Edition expands StarTeam's solutions to new areas of the business enterprise, delivering productivity gains, and improving and accelerating project release schedules.

StarTeam Web Edition enables team members throughout the enterprise to collaborate during the entire development process, while allowing more users greater access to project data without increasing administration overhead. Technical and non-technical contributors, as well as partners, suppliers, and customers, can deliver feedback and suggestions into the same system used by the development team.

Web Edition's Internet-based architecture provides secure and efficient access to project information across local area networks, wide area networks, and the Internet. Web Edition utilizes industry-standard encryption technology from RSA Security, Inc. StarTeam Web Edition provides five levels of encryption, permitting users to select varying degrees of balance between security and performance. StarTeam Web Edition leverages Active Server Pages technology.



Status: New
Severity: Medium
Type: Suggestion
Platform: All
Priority: Yes
Last Build Tested: Build 2000
Synopsis: Customer requires support for Netscape Server running on Linux
Addressed In: Next Build
Responsibility: Steve Hawking
Component: Server
Category: Web UI
External Reference: J. Smith & Associates
Entered By:

javascript:SubmitTheForm('../CrOK.asp')

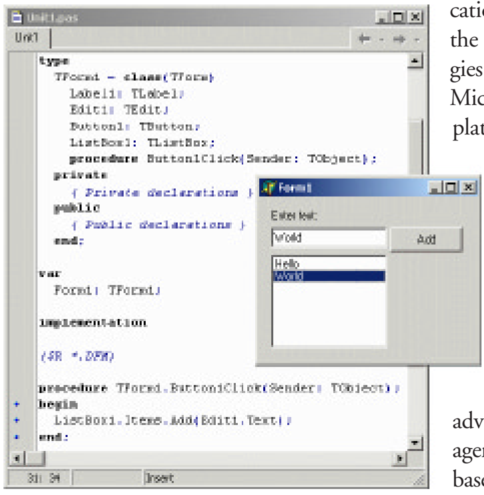
Starbase Corp.
Price: Contact Starbase for pricing.
Phone: (888) 782-7700
Web Site: http://www.starbase.com

Delphi TOOLS

New Products and Solutions

Marotz Offers ASP Express

Marotz, Inc. announced it is offering *ASP Express*, free of charge. ASP Express is the company's set of components and tools intended to help Delphi developers create data-aware Internet applications. It relies on the latest technologies supported on the Microsoft Windows platform: HTML, XML/XSL, MTS/COM+, IIS, and ASP.



```
type
  TForm1 = class(TForm)
    Label1: TLabel;
    Edit1: TEdit;
    Button1: TButton;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add(Edit1.Text);
end;
```

ASP Express includes more than 30 Delphi components for advanced Web development; advanced data management technology based on XML; a drag-and-drop Web form designer; property editors, wizards, and support utilities; standard COM+ components; an HTML-rendering engine based on XML/XSL transformations; a standard security system; standard Cascading Style Sheets; standard ASP templates; advanced Web session management; integration with MTS/COM+ run-time services; integration with IIS and ASP; and sample applications.

ASP Express applications are accessible from any fourth-generation browser.

Marotz, Inc.
Price: Free (without source code).
Phone: (619) 669-3100
Web Site: http://asp-express.com

devSoft Ships IP*Works! SSL V4

devSoft Inc. announced it is shipping *IP*Works! SSL V4*, a new addition to its IP*Works! Internet Toolkit. IP*Works! SSL introduces SSL and Digital Certificates to the IP*Works! components, providing secure Web browsing, secure client, secure server, secure mail, digital certificate management capabilities, and more. This is a comprehensive suite of royalty-free, SSL-enabled components.

IP*Works! SSL V4 consists of four editions: Delphi, C++Builder, C++, and ActiveX. Each edition includes 11 components for secure Internet connectivity: HTTPS, SMTPS, POPS, IMAPS, FTPS, LDAPS, NNTPS, TelnetS, IPPortS, IPDaemonS, and CertMgr. They consist of security-enabled versions of the corresponding components in the IP*Works! Internet Toolkit. The Delphi and C++Builder Editions include native Delphi and C++Builder VCLs fully integrated with the Delphi and C++Builder IDEs.

devSoft Inc.
Price: US$995
Phone: (919) 544-7770
Web Site: http://www.dev-soft.com

Mele Systems/Youseful.com Introduces YOUSEFUL 5.1

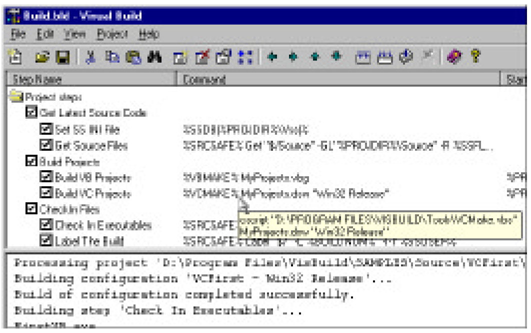
Mele Systems, LLC (**Youseful.com**) unveiled *YOUSEFUL 5.1*, the newest edition of its Delphi-integrated software deployment package.

YOUSEFUL 5.1 features include drag-and-drop for files; file type associations; the ability to create Windows 2000 hard links during install; ADO via mdac_typ.exe; a faster and more reliable Registry Browser; the ability to hit the Delete key (instead of just the popup) in the file lists to have all the selected install files deleted; an added *RunParameters* property to *TCustomInstallFile* for the file if it is an .EXE to run after install; Windows NT and 2000 special paths; the ability to password groups in the install; and more.

Mele Systems, LLC
Price: US$159
Phone: (631) 300-5214
Web Site: http://www.youseful.com

Kinook Releases Visual Build 2.0

Kinook Software released *version 2.0 of Visual Build*, a build management application for Windows and Web developers. Visual Build automates common tasks that must be performed when building software applications. It can retrieve the latest source code, initialize registry settings, register components, build source code projects, check in executables, create installation files, and more. It executes the steps that must be performed over and over again, while ensuring that all steps have been successfully completed. If any step fails, Visual Build pinpoints the error and continues from the point of failure after the problem has been corrected.



```
Project steps
  Get Latest Source Code
    Set SS INI File        %SSDIR%PROJDIR%\Vss%
    Get Source Files       %SRCSAFE% Get "$/Source" -GL"%PROJDIR%\Source" -R %SSFL...
  Build Projects
    Build VB Projects      %VBMAKE% MyProjects.vbg
    Build VC Projects      %VCMAKE% MyProjects.dsw "Win32 Release"
  Check In Files
    Check In Executables   %SRCSAFE%
    Label The Build        %SRCSAFE%

cscript "D:\PROGRAM FILES\VISBUILD\Tools\VCMake.vbs" MyProjects.dsw "Win32 Release"
```

```
Processing project 'D:\Program Files\VisBuild\SAMPLES\Source\VCFirst\
Building configuration 'VCFirst - Win32 Release'...
Build of configuration completed successfully.
Building step 'Check In Executables'...
```

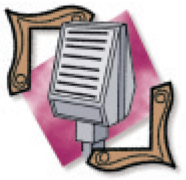
Visual Build provides a Windows interface, using tooltips extensively. It offers built-in support for Borland development tools, and can be integrated with virtually any third-party or home-grown tool.

Kinook Software
Price: US$79.95 per seat (up to five seats); volume discounts are available.
Phone: (719) 599-0442
Web Site: http://www.kinook.com

News

L I N E

November 2000



Inprise/Borland CEO Unveils Macintosh Support

Scotts Valley, CA — Inprise/Borland interim president and chief executive officer Dale Fuller announced the company will provide its JBuilder Java development environment for Apple Computer's new Macintosh operating system.

The announcement was one of several supporting the company's cross-platform approach to create a Web-based infrastructure for the delivery of applications across the Internet. Project Kylix, a high-performance native rapid application development tool to bring new and existing applications to the Linux platform, and the forthcoming integration of Borland's Delphi into the Inprise Application Server, were also demonstrated.

Mr Fuller said, "JBuilder will provide full support for Apple's new user interface, Aqua, while allowing Java developers on the Apple platform to build cross-platform pure Java applications. This is the first Inprise/Borland product to support the Macintosh platform."

To learn more, visit Inprise/Borland at http://www.borland.com/, the community site at http://community.borland.com/, or call the company at (800) 632-2864.

Inprise/Borland Introduces InterBase 6.0

Scotts Valley, CA — Inprise/Borland announced the availability of the source code for InterBase 6.0, its cross-platform, standard query language (SQL) relational database management system. Binary formats for the Linux, Windows, and Solaris operating systems are also available for download, free of charge.

InterBase 6.0 has been released under a variant on the Mozilla Public License V1.1. Developers using InterBase under this license can modify the code or develop applications without being required to open source them. The open source license applies to all platforms.

In addition to being the first open source release of InterBase, version 6.0 introduces a number of new features, including new data types (long integer, date, time, and datetime), extended SQL 92 compliance, an open interface for defining new national character sets, and performance and security enhancements.

Copies of the source code and binary versions for Linux, Windows, and Solaris are available at the Inprise/Borland Web site at http://www.inprise.com/interbase/.

Inprise/Borland's InterBase Ships with Cobalt Networks' Server Appliance

Inprise/Borland announced that InterBase 6.0 is shipping with Cobalt Networks' RaQ 4r, its fourth-generation server appliance for Internet and application service providers. InterBase 6.0, the open source version of Inprise/Borland's high-performance, structured query language (SQL) database, enables Cobalt RaQ 4r users to develop and deploy business-critical, Internet-based applications.

RaQ 4r is a server appliance with integrated development tools, applications, and a database. As part of RaQ 4r's full suite of Internet and application services, InterBase 6.0 will help facilitate the deployment of applications and layered services over the Web.

For more information on Cobalt Networks, visit http://www.cobalt.com.

Inprise/Borland Offers Fast Path to Creating E-commerce Sites

San Diego, CA — Inprise/Borland announced its e-Commerce Framework Solution, an integrated software and consulting solution that allows small- to mid-sized businesses worldwide to deploy open and scalable e-commerce sites, using Inprise/Borland's technology.

Written in Java and deployed in an *n*-tier topology, the e-Commerce application is managed by Inprise Application Server 4.0 and can be customized for multilingual, multi-currency implementation in weeks rather than months. The CORBA architecture delivers scalable, high-availability solutions that automatically load balance, both locally and over a wide-area network. The e-Commerce Framework Solution, which is priced based on client needs, is available worldwide.

e-Commerce Framework Solution offers the Firewall Broker to manage security; Partition Service for handling multiple clients from a single hosting facility; Language Service for automatically loading correct languages for clients; Session Service for unique user session identification; Cart Service for transient storage of Web site shopping state information; and much more.

The Inprise/Borland Canadian Web site http://www.borland.ca, which was built using the e-Commerce Framework, has been processing transactions since March 2000. The Canadian site was developed using JBuilder and VisiBroker for Java 3.4, is deployed using the Inprise Application Server 4.0, and is managed using Inprise AppCenter 1.6.

To learn more, visit Inprise/Borland at http://www.borland.com/, the community site at http://community.borland.com/, or call the company at (800) 632-2864.

Inprise/Borland Announces Results of Its 2000 Annual Meeting

Scotts Valley, CA — Inprise/Borland announced the results of its 2000 Annual Meeting of Stockholders, held in July at its headquarters in Scotts Valley, CA.

Dale L. Fuller, 41, and William K. Hooper, 45, were re-elected to the Inprise/Borland Board of Directors by more than 95 percent of the votes cast, and will serve three-year terms, expiring at the 2003 Annual Meeting of Stockholders. Mr Fuller has been a director of Inprise/Borland since April 1999 and has served as interim chief executive officer and president of Inprise/Borland since that time. Mr Hooper, president of the Woodside Hotels and Resorts Group Services Corp. and Monterey Plaza Hotel Corp., has been a director of Inprise/Borland since May 1999.

The stockholders also overwhelmingly approved an amendment to Inprise/Borland's 1997 Stock Option Plan to increase the number of shares of common stock that can be issued under that plan by 2,000,000 shares, and an amendment to Inprise/Borland's 1999 Employee Stock Purchase Plan to reserve for issuance pursuant to such plan an additional 500,000 shares of common stock.

In addition, the stockholders ratified the selection of PricewaterhouseCoopers LLP to continue to serve as Inprise/Borland's independent auditors for the fiscal year ending December 31, 2000.

table of CONTENTS

OP TECH

COM / Data Exchange / MIDAS / Delphi 5

By Bill Todd

Moving Data via COM

Using COM to Transfer Any Type of Data

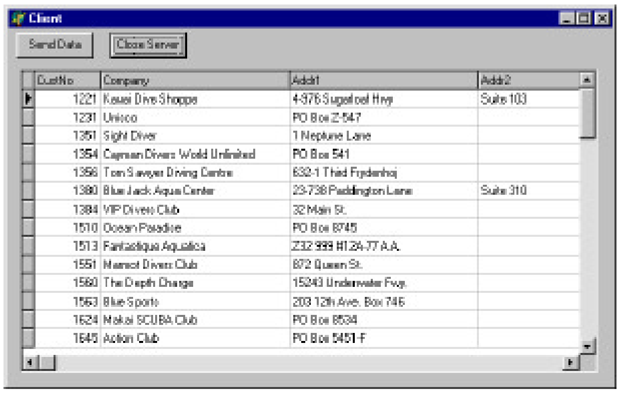
What do you do when you need to move data — data not stored in a database table — between a COM server and a COM client? Simply stuff it in a variant, and pass it as a parameter. You can pass anything, from integer arrays to large binary files, this way.

I'm not talking about using a MIDAS server and client, but any COM server and client. Although the techniques in this article will work with a MIDAS server and client using the *IAppServer* interface, they will work equally well between any COM server and client, using any interface that you can add methods to.

Passing Tabular Data

If you need to pass tabular data, the easiest thing to do is put it in a ClientDataSet, and pass that. This is demonstrated in the PassData sample application that accompanies this article (see end of article for download details). This application consists of a COM server and a COM client.

The client's main form, shown in Figure 1, contains a Database, Query, DataSetProvider, ClientDataSet, and DataSource connected to the DBGrid to display the data in the DBDEMOS sample customer table. The following is the **Send Data** button's *OnClick* event handler:



CustNo	Company	Addr1	Addr2
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
1231	Unisco	PO Box Z-547	
1351	Sight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310
1384	VIP Divers Club	32 Main St.	
1510	Ocean Paradise	PO Box 8745	
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	
1551	Marmot Divers Club	872 Queen St.	
1560	The Depth Charge	15243 Underwater Fwy.	
1563	Blue Sports	203 12th Ave. Box 746	
1624	Makai SCUBA Club	PO Box 8534	
1645	Action Club	PO Box 5451-F	

Figure 1: The COM client's main form.

```
procedure TMainForm.SendBtnClick(Sender:
          TObject);
begin
  PassDataServer := CoPassData.Create;
  PassDataServer.PassData(CustCds.Data);
end;
```

The client application uses the server's type library interface unit so it can connect to the server by calling the *Create* method of the server's CoClass, and assign the interface reference to the variable *PassDataServer*. *PassDataServer* is declared as a private member variable of the form, with a type of *IPassData*. *IPassData* is the interface implemented by the COM server. The second statement calls the *PassData* method of the *IPassData* interface, and passes the ClientDataSet's *Data* property as a parameter.

The following is the server's *PassData* method:

```
procedure TPassData.PassData(CdsData:
          OleVariant);
begin
  with MainForm.CustCds do begin
    Data := CdsData;
    Open;
  end; // with
end;
```

This method takes a single parameter of type OleVariant that's used to pass the ClientDataSet's *Data* property from the client to the server. The server application's main form contains a ClientDataSet, DataSource, and a DBGrid. The code assigns the *CdsData* parameter to the ClientDataSet's *Data* property and opens the ClientDataSet, causing the data that was passed from the client to appear in the grid on the server's form.

Note that the ClientDataSet in the server isn't connected to a remote server or provider in this example, but it could be.

Passing Flat-file Data

One of the neat things about MIDAS is that the data the MIDAS server sends to the client can come from anywhere. It doesn't have to be stored in a database table. One of the techniques in the PassOther sample application (also available for download; see end of article for details) supplies data to the MIDAS client from a comma-delimited ASCII file.

The easiest way to do this is to drop a ClientDataSet and DataSetProvider on the server's remote data module. Then use the Object Inspector to edit the ClientDataSet's *FieldDefs* property and add the field definitions you need for your data. Next, write a *BeforeGetRecords* event handler for the DataSetProvider that gets the data, in this case from the ASCII file, and loads it into the ClientDataSet. The DataSetProvider then gets the data from the ClientDataSet and sends it to the client application in the normal way. Figure 2 shows the *BeforeGetRecords* event handler.

```
procedure TPassOther.TextProvBeforeGetRecords(
  Sender: TObject; var OwnerData: OleVariant);
var
  AFile:     TextFile;
  FieldVals: TStringList;
  Rec:       string;
begin
  FieldVals := TStringList.Create;
  try
    with TextCds do begin
      { If the ClientDataSet is active, empty it; otherwise
        create it using the FieldDefs entered at design
        time. Calling CreateDataSet creates the in-memory
        dataset and opens the ClientDataSet. }
      if Active then
        EmptyDataSet
      else
        CreateDataSet;
      { Open the ASCII file. }
      AssignFile(AFile, OwnerData);
      Reset(AFile);
      { Loop through the ASCII file. Read each record and
        assign it to the CommaText property of the
        TStringList FieldVals. This parses the record and
        assigns each field to a string in the StringList.
        Insert a new record in the ClientDataSet and
        assign the StringList elements to the fields. }
      while not System.EOF(AFile) do begin
        Readln(AFile, Rec);
        FieldVals.Clear;
        FieldVals.CommaText := Rec;
        Insert;
        FieldByName('Name').AsString := FieldVals[0];
        FieldByName('Date').AsDateTime :=
          StrToDate(FieldVals[1]);
        FieldByName('Unit').AsString := FieldVals[2];
        Post;
      end; // while
      System.CloseFile(AFile);
      { Be sure to reposition the ClientDataSet to the
        first record, so the DataSetProvider will start
        with the first record when building its data
        packet to send to the client. }
      First;
    end; // with
  finally
    FieldVals.Free;
  end; // try
end;
```

Figure 2: The server's *BeforeGetRecords* event handler.

The *BeforeGetRecords* event handler starts by creating a StringList, named *FieldVals*, that's used to parse the records from the comma-delimited ASCII file. Next, it checks to see if the ClientDataSet is active, and if so, empties it. If not, it calls *CreateDataSet*, which creates the in-memory dataset using the *FieldDefs* supplied at design time, and opens the ClientDataSet.

The *AssignFile* and *Reset* statements open the ASCII file. Notice that the name of the file in the call to *AssignFile* is the *OwnerData* parameter passed to the event handler. *OwnerData* is provided so the client can pass any information it wants to the server by setting the value of the *OwnerData* parameter in the client application ClientDataSet's *BeforeGetRecords* event. Because *OwnerData* is a variant, you can pass any type of data, including a variant array of variants. This gives you the ability to pass as many values of any type as you wish.

The **while** loop reads a record from the text file into the string variable *Rec*, clears the StringList, and assigns *Rec* to the StringList's *CommaText* property. The demonstration application uses the following text, which is in a standard comma-delimited ASCII file named Text.txt:

```
"Sherman T. Potter","1/23/1901","MASH 4077"
"B. J. Hunnicut","4/19/29","MASH 4077"
"B. F. Pierce","6/6/1928","MASH 4077"
"Margaret Houlihan","8/8/1930","MASH 4077"
```

When you assign a string to *CommaText*, it's parsed on any commas or spaces not enclosed in quotation marks, and each substring is assigned to an element of the StringList. Next, the procedure inserts a new record into the ClientDataSet, and assigns the values from the StringList to the fields in the new record. Finally, the new record is posted. Once the end of the text file is reached, a call to *CloseFile* closes the ASCII file.

A call to the *First* method moves the ClientDataSet's cursor to the first record. This is critical because the DataSetProvider will start with the current record when it builds the data packet to send to the client. If you leave the ClientDataSet positioned on the last record, the last record is the only one that will be sent to the MIDAS client. Finally (literally **finally**), a call to the StringList's *Free* method destroys it.

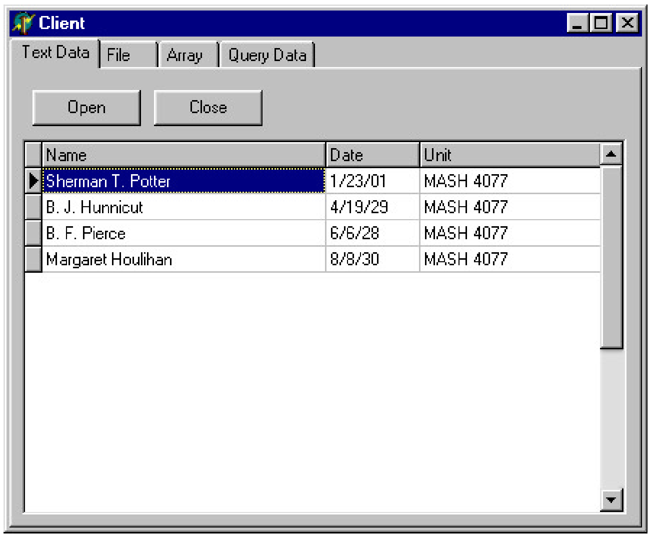
On the client side, things are even easier. When you open the ClientDataSet in the MIDAS client application, its *BeforeGetRecords* event fires. The code for the client's *BeforeGetRecords* event handler follows:

```
procedure TMainDm.TextCdsBeforeGetRecords(Sender: TObject;
  var OwnerData: OleVariant);
begin
  { Assign the file name to OwnerData which is passed to
    the MIDAS client automatically. }
  OwnerData :=
    ExtractFilePath(Application.ExeName) + 'text.txt';
end;
```

The only thing that happens here is that the name of the text file is assigned to the *OwnerData* parameter. *OwnerData* is automatically sent to the MIDAS server, where, as you've seen, it appears as a parameter to the DataSetProvider's *BeforeGetRecords* event. The result is shown in Figure 3.

Sending a File You Don't Want to Display

Using ClientDataSet is great for data you want to display on a form. Suppose, however, that you need to send a file from a COM server



Name	Date	Unit
Sherman T. Potter	1/23/01	MASH 4077
B. J. Hunnicut	4/19/29	MASH 4077
B. F. Pierce	6/6/28	MASH 4077
Margaret Houlihan	8/8/30	MASH 4077

Figure 3: The text-file data as it appears in the demonstration client at run time.

to its client, but you don't want the file's contents displayed in a ClientDataSet.

It's easy — even if you need to send a file that's too large to fit in memory. The File tab of the sample application, which contains Button and Memo components, demonstrates this. Figure 4 shows the code from the **Copy File** button's *OnClick* event handler. This procedure begins by declaring a constant, *ArraySize*, that determines the size of the variant array used to transfer the file from the COM server to the client.

This sample program displays the blocks of data read from the server in the Memo component on the form (see Figure 5). In an application where you are transferring a large amount of data, and storing it in memory or writing it to a file, you would use a much larger array (e.g. 4KB or 16KB) to transfer more data on each call to the server.

The first statement creates a variant array, *VData*, with a lower bound of zero and an upper bound of `ArraySize-1`, making it the same size as *ArraySize*. The array is of type varByte, so it can hold anything. Because we want to put the data into the Memo component, the string of bytes returned from the server must be put into a string variable, in this case *S*. The call to *SetLength* sets the size of *S* to the size of the array. Next, the DCOMConnection component is opened, and the Memo is emptied.

Transferring the file is accomplished by three custom methods, which are added to the server application's *IAppServer* interface using the Type Library editor. The first, *OpenFile*, takes a single parameter, the name of the file to be transferred. The **while** loop calls the second *IAppServer* method, *GetFileData*. *GetFileData* passes the variant array, *VData*, as a **var** parameter, and returns the number of bytes read from the file. This will be the size of the array for every block except the last one, which may contain fewer bytes if the file size is not an even multiple of the block size. If the number of bytes returned by a call to *GetFileData* is zero, the end of the file has been reached and the **while** loop is exited.

The next step is to put the bytes returned in the array into the string variable, *S*, and add the string to the Memo component. To access the data in the variant array faster, the array is locked by the call to

```
procedure TMainForm.CopyFileBtnClick(Sender: TObject);
const
  ArraySize = 20;
var
  VData:     Variant;
  PData:     PByteArray;
  S:         string;
  ByteCount: Integer;
begin
  with MainDm.Conn do begin
    { Create the variant array of bytes that will hold the
      data read from the text file by the server
      application. }
    VData := VarArrayCreate([0, ArraySize - 1], varByte);
    { Allocate the string variable S to hold the number of
      bytes returned in the variant array. }
    SetLength(S, ArraySize);
    { Connect to the MIDAS server and empty the memo
      component. }
    if not Connected then
      Open;
    Memo.Lines.Clear;
    { Call the server's OpenFile method. This creates the
      TFileStream on the server that is used to read the
      file. The name of the file to read is passed as a
      parameter. }
    AppServer.OpenFile(ExtractFilePath(
      Application.ExeName) + 'text.txt');
    { Read data from the server until the entire file has
      been read. }
    while True do begin
      { Read a block of data from the server. GetFileData
        returns the number of bytes read. The parameter is
        a variant array of bytes passed by reference. }
      ByteCount := AppServer.GetFileData(VData);
      { If the number of bytes read is zero, the end of the
        file has been reached. }
      if ByteCount = 0 then
        Break;
      { Lock the variant array and get a pointer to the
        array values. }
      PData := VarArrayLock(VData);
      try
        { The read that reaches the end of the file may
          return fewer bytes than requested. If so, resize
          the string variable to hold the number of bytes
          actually read. }
        if ByteCount < ArraySize then
          SetLength(S, ByteCount);
        { Move the data from the variant array to the
          string variable. }
        Move(PData^, S[1], ByteCount);
      finally
        VarArrayUnlock(VData);
      end; // try
      Memo.Lines.Add(S);
    end; // while
    AppServer.CloseFile;
  end; // with
end;
```

Figure 4: The **Copy File** button's *OnClick* event handler.

VarArrayLock(VData), which returns a pointer to the actual data array in the variant. The pointer is assigned to the variable *PData*, which is declared as type PByteArray. PByteArray is declared in the System unit as a pointer to an array of type Byte.

The data is moved from the array to the string variable by calling:

```
Move(PData^, S[1], ByteCount)
```

The *Move* procedure copies a specified number of bytes from one location in memory to another. The first parameter is the source location, the second parameter is the destination, and the third parameter is the number of bytes to copy.

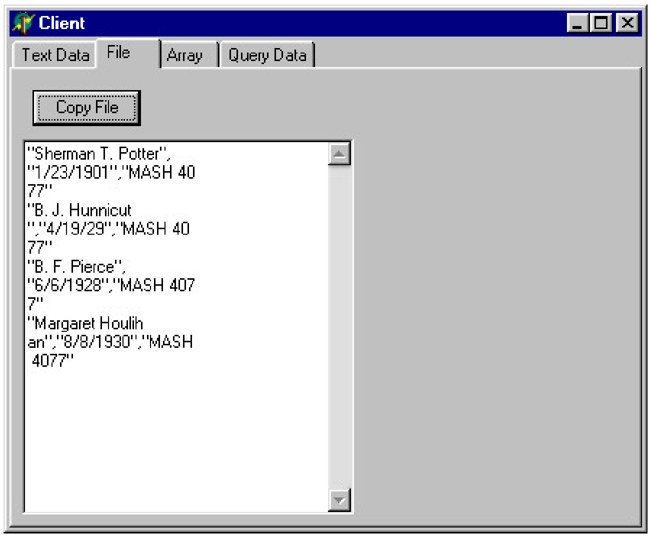
The ^ at the end of the pointer variable *PData* dereferences the pointer. In other words, *PData^* means "the location in memory which *PData* points to." Note that *Move* performs no error checking of any kind, so be careful to use the correct parameters. Strange things will happen at run time if you overwrite the wrong area of memory. In addition, *Move* does not perform any type checking. You can move any bit pattern into a string or any other kind of variable. Once the data has been moved from the array to the string, the variant array is unlocked and the string is added to the Memo. Once the entire file has been copied, a call to the third custom method of *IAppServer*, *CloseFile*, closes the file on the server.

On the server side, the methods *OpenFile*, *GetFileData*, and *CloseFile* were added to the *IAppServer* interface using the Type Library editor. The following shows the code from the remote data module's unit for the *OpenFile* method:

```
procedure TPassOther.OpenFile(FileName: OleVariant);
begin
  { Create the TFileStream object in read mode. Allow other
    applications to read the text file, but not to write
    to it. }
  Fs := TFileStream.Create(FileName,
                           fmOpenRead or fmShareDenyWrite);
end;
```

OpenFile contains a single statement, which creates a *FileStream* object for the file passed as a parameter to the method. The file is opened in read mode and is shared for reading, but no writing is allowed. The *FileStream* is assigned to the variable *Fs*, which is a private member variable of the remote data module.

Figure 6 shows the *GetFileData* method. This method expects a single **var** parameter, which is the variant array of bytes passed by the client. *GetFileData* locks the variant array for fast access, then assigns the pointer returned by *VarArrayLock* to the local variable *PData*. Next, it calls the *FileStream.Read* method, passing the address *PData* points to as the location to store the data. It also passes *VarArrayHighBound(Data, 1) + 1* as the number of bytes to read, so the number of bytes read is always equal to the size of the array. The number of bytes read is assigned to *Result*, and returned by the function. Finally, a call to *VarArrayUnlock* unlocks the variant array.



```
"Sherman T. Potter",
"1/23/1901","MASH 40
77"
"B. J. Hunnicut
","4/19/29","MASH 40
77"
"B. F. Pierce",
"6/6/1928","MASH 407
7"
"Margaret Houlih
an","8/8/1930","MASH
 4077"
```

Figure 5: The File tab of the sample application.

The following shows the *CloseFile* method, which frees the *FileStream* object and sets its instance variable to **nil**:

```
procedure TPassOther.CloseFile;
begin
  if Assigned(Fs) then begin
    Fs.Free;
    Fs := nil;
  end;
end;
```

The *OnDestroy* event handler for the remote data module also frees the *FileStream* if *Fs* is not **nil**, just in case the client program doesn't call *CloseFile*.

Although this example uses a MIDAS client and server, you can use exactly the same technique to transfer a file from a COM server to its client.

Sending Arrays or Other Memory Structures

You can also send an array, a Pascal record, or any other data structure that exists in memory, by stuffing it into a variant array of bytes. Figure 7 shows the *GetArray* and *LoadVariantArray* methods of the sample MIDAS server. *GetArray* declares a 10-element integer array and loads it with the numbers 1 through 10. The client application passes a variant, *VData*, as a **var** parameter. *GetArray* calls *LoadVariantArray* and passes it three parameters.

The first parameter value, *@IntArray*, is the memory address of the *IntArray* array. The "at" sign is the "Address of" operator in Pascal, so you can read *@IntArray* as "the address of *IntArray*." This provides *LoadVariantArray* with a pointer to the location in memory where the integer array values are stored. The second parameter value, *SizeOf(IntArray)*, passes the size of *IntArray* in bytes. The third and final parameter is the variant variable into which the integer array will be loaded.

LoadVariantArray begins by calling *VarArrayCreate*, which creates a variant array of bytes that's the same size as the integer array to be returned. Next, the variant array is locked, the integer array is moved into it, and the variant array is unlocked. Notice that the first parameter of *LoadVariantArray*, *PData*, is of type Pointer. Using the generic Pointer data type means that you can pass the

```
{ Reads a block of data from the TFileStream, Fs, into the
  parameter Data, which is a variant array of bytes.
  Returns the number of bytes read into the array. }
function TPassOther.GetFileData(var Data: OleVariant):
  Integer;
var
  PData: PByteArray;
begin
  { Lock the variant array and get a pointer to the array
    of bytes. This makes access to the variant array much
    faster. }
  PData := VarArrayLock(Data);
  try
    { Read data from the TFileStream. The number of bytes
      to read is the high bound of the variant array,
      plus one (because the array is zero-based). This
      function returns the number of bytes read. }
    Result :=
      Fs.Read(PData^, VarArrayHighBound(Data, 1) + 1);
  finally
    VarArrayUnlock(Data);
  end; // try
end;
```

Figure 6: The *GetFileData* method.

```
procedure TPassOther.GetArray(var VData: OleVariant);
var
  IntArray: array[1..10] of Integer;
  I:     Integer;
  PData: PByteArray;
begin
  { Put some numbers in the array. }
  for I := 1 to 10 do
    IntArray[I] := I;
  { Load the integer array into the variant array. }
  LoadVariantArray(@IntArray, SizeOf(IntArray), VData);
end;

procedure TPassOther.LoadVariantArray(PData: Pointer;
  NumBytes: Integer; var VData: OleVariant);
var
  PVData: PByteArray;
begin
  { Create the variant array of bytes. Set the upper bound
    to the size of the array, minus one, because the array
    is zero-based. }
  VData := VarArrayCreate([0, NumBytes - 1], varByte);
  { Lock the variant array for faster access. Then copy the
    array to the variant array, and unlock the variant
    array. }
  PVData := VarArrayLock(Vdata);
  try
    { Move the bytes at the location in memory that PData
      points to into the location in memory that PVData
      points to. PData points to the integer array and
      PVData points to the variant array of bytes. }
    Move(PData^, PVData^, NumBytes);
  finally
    VarArrayUnlock(VData);
  end; // try
end;
```

Figure 7: The *GetArray* and *LoadVariantArray* methods.

```
procedure TMainForm.CopyArrayBtnClick(Sender: TObject);
var
  IntArray: array[1..10] of Integer;
  VData: Variant;
  I:     Integer;
begin
  { Connect to the server application. }
  if not MainDm.Conn.Connected then
    MainDm.Conn.Open;
  { Call the server's GetArray method and pass a variant
    parameter. }
  MainDm.Conn.AppServer.GetArray(VData);
  { Get the data out of the variant array. }
  UnloadVariantArray(VData, @IntArray, SizeOf(IntArray));
  { Display the array values in the memo. }
  for I := 1 to 10 do
    ArrayMemo.Lines.Add(IntToStr(IntArray[I]));
end;

procedure TMainForm.UnloadVariantArray(
  var VData: OleVariant; PData: Pointer;
  NumBytes: Integer);
var
  PVData: PByteArray;
begin
  { Lock the variant array, copy the data to the array, and
    unlock the variant array. }
  PVData := VarArrayLock(VData);
  try
    { Move the data in memory that PVData points to (the
      variant array data), to the location in memory that
      PData points to (the integer array). }
    Move(PVData^, PData^, NumBytes);
  finally
    VarArrayUnlock(VData);
  end; // try
end;
```

Figure 8: The *CopyArrayBtnClick* and *UnloadVariantArray* methods.

address of any type of variable, array, Pascal record, or any other memory structure to this method. This makes *LoadVariantArray* completely generic. It can be used to load anything stored in memory into a variant array.

Figure 8 shows the *OnClick* event handler for the **Copy Array** button in the PassOther sample application, and the *UnloadVariantArray* method that is called by the button's *OnClick* event handler. This method connects to the MIDAS server by calling the *Open* method of the DCOMConnection component. It then calls the *GetArray* method of the server, passing a variant variable as its parameter.

Next, the *OnClick* event handler calls *UnloadVariantArray*, passing it three parameters. The first, *VData*, is the variant array that contains the values. The second, *@IntArray*, passes a pointer to the location in memory where you want to place the bytes from the variant array. In this case, the second parameter is the address of the integer array *IntArray*. The third parameter value is the size of *IntArray* in bytes.

UnloadVariantArray locks the variant array, and obtains a pointer to its data. It then moves the data from the variant array to the memory location pointed to by *PData*, the address of the *IntArray* array, in this case. Finally, the variant array is unlocked and the integers are displayed in the Memo component on the form (again, see Figure 5).

Conclusion

The ability to pass a variant array as a parameter to a COM method call lets you pass any kind of data between a COM server and its client. While the examples in this article used a text file and a variant array, the same methods will work for any kind of file or data stored in memory. The methods used to transfer the text file will work without change for any type of file, including binary files, such as image files. The same is true of the code that transferred the integer array. It will work equally well with a Pascal record, a multi-dimensional array of doubles, or anything else stored in memory. In short, you can put anything into a variant and transfer it to another application using COM. Δ

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD in INFORM\00\NOV\DI200011BT.

Bill Todd is president of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is co-author of four database programming books, author of more than 60 articles, a Contributing Editor to *Delphi Informant Magazine*, and a member of Team Borland, providing technical support on the Borland Internet newsgroups. He is a frequent speaker at Borland Developer Conferences in the US and Europe. Bill is also a nationally known trainer and has taught Delphi programming classes across the country and overseas. He is currently a speaker on the Delphi Development Seminars Kylix World Tour. Bill can be reached at bill@dbginc.com. For more information on the Kylix World Tour, visit http://www.DelphiDevelopmentSeminars.com.

By Jeremy Merrill

Polymorphic Programming

Three Ways to Break the VCL's "Protected Barrier"

Polymorphism is, in essence, the ability to treat different objects the same way. When objects have something in common, we should be able to write code that makes use of their similarities, i.e. we should be able to write polymorphic code. Unfortunately, we often run into barriers that prevent us from using common forms of polymorphism.

This is the case when using many of the classes in Delphi's Visual Component Library (VCL). When using VCL components, we often find ourselves writing non-polymorphic code to access similar functionality. For example, you may have written code such as that shown in Figure 1.

With all the power of polymorphic programming at our fingertips, why do we have to write such ugly code? *TEdit*, *TMaskEdit*, *TMemo*, and *TRichEdit* all descend from the *TCustomEdit* class. In addition to being non-polymorphic, the code in Figure 1 can only be used on these four, specific descendants of *TCustomEdit*; everything else is excluded. The *ReadOnly* property and the *PasteFromClipboard* method are both defined in the *TCustomEdit* class. If we can call the *PasteFromClipboard* method from the *TCustomEdit* level, why can't we read the *ReadOnly* property the same way? In an ideal, polymorphic world, we should be able to write something like the code in Figure 2.

```
procedure TForm1.mnuPasteClick(Sender: TObject);
var
  CanPaste: Boolean;
  Ctrl: TWinControl;
begin
  Ctrl := ActiveControl;
  if (Assigned(Ctrl) and
      Clipboard.HasFormat(CF_TEXT)) then
    begin
      if (Ctrl is TEdit) then
        CanPaste := (not TEdit(Ctrl).ReadOnly)
      else if (Ctrl is TMaskEdit) then
        CanPaste := (not TMaskEdit(Ctrl).ReadOnly)
      else if (Ctrl is TMemo) then
        CanPaste := (not TMemo(Ctrl).ReadOnly)
      else if (Ctrl is TRichEdit) then
        CanPaste := (not TRichEdit(Ctrl).ReadOnly)
      else
        CanPaste := False;
      if (CanPaste) then
        TCustomEdit(Ctrl).PasteFromClipboard;
    end;
end;
```

Figure 1: This code functions, but it's ugly because it cannot use polymorphism.

The only problem with the code in Figure 2 is that it generates a compile error: "Undeclared identifier: 'ReadOnly'." The reason is that *TCustomEdit* defines *ReadOnly* as a protected property. While this is done for a good reason — so descendant classes can choose which properties to expose — it gets in the way of writing polymorphic code. We can call the *PasteFromClipboard* method from the *TCustomEdit* level because it's defined as public. It's only when we typecast the active control as a class, where *ReadOnly* has been promoted to a higher visibility, that we gain access to it.

To use the power of polymorphism in this example, we need a way to break the protected barrier.

Breaking the Protected Barrier

Record-type definitions in Object Pascal can map the same bytes in memory to multiple variables. This is done using a special form of the **case** statement, as shown in the *TRect* type declaration (see Figure 3).

If we declare variable *R* as a *TRect*, *R.Left* refers to the same integer, occupying the same bytes in memory as *R.TopLeft.x*; and *R.Bottom* refers to the same integer as *T.BottomRight.y*. Essentially, we're defining two memory maps for the same bytes in memory.

Using a similar technique, we can use different memory maps for the same object. Every object is, at its lowest level, nothing more than a collection of bytes. An object's class contains all the executable code, but it also defines the layout of the bytes that make up the object's data. Descendant classes simply stack new data elements, or fields, on top of the fields defined in inherited classes. Figure 4 illustrates this relationship.

Notice in Figure 4 that the bottom four memory segments are identical for all four classes. Because the memory locations of data remain the same in all descendant classes, we can remap an object that exposes a protected property to one that exposes a public or published visibility. This allows us to write the polymorphic code shown in Figure 5.

Looking at this structure, you might not want to use the *TExposedCustomEdit* class. You could, for example, use *TRichEdit* instead, or any other *TCustomEdit* descendant. While this will work for reading the *ReadOnly* property, it also entails the possibility of memory corruption and access violation problems. An example of this would be typecasting a *TEdit* component as a *TRichEdit*, and referencing or writing to the *Lines* property, which is defined in *TRichEdit*, but not in *TEdit*. The compiler wouldn't catch such a violation, whereas using a class such as *TExposedCustomEdit* allows you to prevent invalid class mappings.

I struggled with writing this article for quite some time, because this technique raises some questions that I wasn't sure how to answer. After contacting Delphi Chief Architect, Chuck Jazdzewski, however, I'm able to provide a better explanation of why this works. I also received answers to the following questions:

Question: Is this technique subject to breaking in future versions of Delphi?

Answer: Not likely. I can't promise anything, but a change that would break your code would certainly break a lot of our own code. We will, however, make it clear if a version of Delphi breaks this, because a memory layout change that would break this would be significant.

Question: Are there other uses, besides exposing protected properties, for which this kind of typecasting would be reliable?

Answer: Actually, it works for anything in the object scope that is protected — fields, methods, etc.

I haven't tried to use this technique on protected methods. I've used it on a number of occasions to reference protected properties, and have never had any problems with it. While the main use has been to allow for polymorphic code, I've also used this technique to avoid having to register a new component with Delphi, simply to access a protected property.

Using Run-time Type Information

The technique just described doesn't solve all our problems, however. Imagine we're writing an application that sets certain visual attributes based on user preferences. To write a method that changes a control from flat to 3D, you would have to write something like this:

```
procedure TForm1.SetFlat(Ctrl: TControl; AFlat: Boolean);
begin
  if Ctrl is TSpeedButton then
    TSpeedButton(Ctrl).Flat := AFlat
  else if Ctrl is TCheckListBox then
    TCheckListBox(Ctrl).Flat := AFlat
  else if Ctrl is TToolBar then
    TToolBar(Ctrl).Flat := AFlat
end;
```

While this appears to be the same problem described previously, it's significantly different. The previous technique requires a property to be defined in a common ancestral class, but the *Flat* property is defined separately in each of the listed classes. About the only things these properties have in common are their name and type. How can we access them polymorphically without having to write class-specific code? One solution is to use the run-time type information (RTTI) stored for all published properties and methods. This is the technique used by Delphi's Object Inspector. Note that this technique might not be appropriate for some situations, because it's more processor-intensive than class-specific code.

Accessing RTTI previously required us to write a small library of routines that would call different routines in the TypInfo unit. With

```
procedure TForm1.mnuPasteClick(Sender: TObject);
var
  Ctrl: TWinControl;
begin
  Ctrl := ActiveControl;
  if (Assigned(Ctrl) and
      Clipboard.HasFormat(CF_TEXT)) then
    if (Ctrl is TCustomEdit) and
       (not TCustomEdit(Ctrl).ReadOnly) then
      TCustomEdit(Ctrl).PasteFromClipboard;
end;
```

Figure 2: Ideal, polymorphic code (which doesn't compile).

```
TPoint = record
    x: Longint;
    y: Longint;
  end;

TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
  end;
```

Figure 3: The *TRect* type declaration.

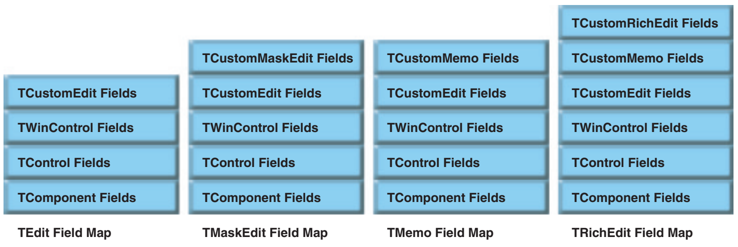


Figure 4: Memory map of three objects descending from *TCustomEdit*.

```
type
  TExposedCustomEdit = class(TCustomEdit)
  public
    property ReadOnly;
  end;

...

procedure TForm1.mnuPasteClick(Sender: TObject);
var
  Ctrl: TWinControl;
begin
  Ctrl := ActiveControl;
  if (Assigned(Ctrl) and
      Clipboard.HasFormat(CF_TEXT)) then
    if (Ctrl is TCustomEdit) and
       (not TExposedCustomEdit(Ctrl).ReadOnly) then
      TCustomEdit(Ctrl).PasteFromClipboard;
end;
```

Figure 5: Remapping an object with a protected property (*ReadOnly*), to one that has public visibility.

Delphi 5, however, we can now access any published property or method by calling a single procedure or function. Some of the routines in the TypInfo unit that allow us to do this are shown in Figure 6.

To use this technique in solving our *Flat* property problem, we would first use the *IsPublishedProp* function to see if the control had a *Flat* property, then use the *SetOrdProp* routine to change its value. If we don't use *IsPublishedProp*, then *SetOrdProp* will result in an access violation on any object that doesn't have a *Flat* property. We use *SetOrdProp* for a Boolean property, because the Boolean type is simply another enumerated type, and all enumerated types are simply ordinal (or numeric) types. To convert an enumerated type to an ordinal value, we use Pascal's *Ord* function. Translating all this information into usable code, we add TypInfo to the **uses** list, and write the following:

```
procedure TForm1.SetFlat(Ctrl: TControl; AFlat: Boolean);
begin
  if IsPublishedProp(Ctrl, 'Flat') then
    SetOrdProp(Ctrl, 'Flat', Ord(AFlat));
end;
```

Another possible use of this technique, which could potentially be more powerful than standard polymorphic techniques, is to allow the property name parameter passed to routines in the TypInfo unit to be a variable. There are generic *GetPropValue* and *SetPropValue* routines that allow you to read and write property values, regardless of type. These capabilities could be combined to take polymorphism to the extreme, at least as far as published properties and methods are concerned.

While an entire article could be devoted to the intricacies of the TypInfo unit (see Bill Todd's article "RTTI Gets Easier" in the November, 1999 issue of *Delphi Informant Magazine*), I've tried to illustrate here that, with Delphi 5, you no longer have to know any of these details to access RTTI.

Using Windows Messaging

Here's another wrinkle: Sometimes we need to reference common functionality that's declared in properties with different names. Take, for example, the following method:

```
procedure TForm1.ChangeText(Ctrl: TControl; Txt: string);
begin
  if (Ctrl is TLabel) then
    TLabel(Ctrl).Caption := Txt
  else if (Ctrl is TEdit) then
    TEdit(Ctrl).Text := Txt
  else if (Ctrl is TButton) then
    TButton(Ctrl).Caption := Txt
  else if (Ctrl is TMemo) then
    TMemo(Ctrl).Lines.Text := Txt;
end;
```

This routine could have been expanded to include numerous other component class types; this is a common problem in the VCL. Both of the techniques detailed previously could be used to reach a partial solution to this problem. However, finding a more complete solution without creating several new classes requires something more.

To find that solution, we need to first recognize that many of the components in Delphi are simply wrappers for Windows messages, and it is Windows itself that handles most of the components. If we were to create our own class that had a *Text* property, we would create an instance variable, called *FText*, of type **string**, and our *Text* property would store its value in *FText*. With many of the component

```
function IsPublishedProp(Instance: TObject;
  const PropName: string): Boolean;
function GetOrdProp(Instance: TObject;
  const PropName: string): Longint;
procedure SetOrdProp(Instance: TObject;
  const PropName: string; Value: Longint);
function GetStrProp(Instance: TObject;
  const PropName: string): string;
procedure SetStrProp(Instance: TObject;
  const PropName: string; const Value: string);
function GetFloatProp(Instance: TObject;
  const PropName: string): Extended;
procedure SetFloatProp(Instance: TObject;
  const PropName: string; Value: Extended);
function GetMethodProp(Instance: TObject;
  const PropName: string): TMethod;
procedure SetMethodProp(Instance: TObject;
  const PropName: string; const Value: TMethod);
```

Figure 6: These TypInfo unit routines allow us to use RTTI more easily in Delphi 5.

properties in the VCL, however, there is no instance variable holding a text or caption value. The data is actually stored in Windows, and the *GetText* and *SetText* routines of a *Text* property send Windows messages to retrieve or store the data.

Knowing this allows us to change a property value polymorphically, by talking directly to Windows instead of executing Delphi code. The difficulty with this approach is finding the correct messages to send. A thorough understanding of how Windows works will help, but it's not essential. In fact, the only solution is to dig into the VCL source code itself, because knowing how Windows works doesn't tell you anything about how Delphi works.

Digging deep into the VCL reveals that most components change *Text* or *Caption* properties with the WM_SETTEXT message, often followed by the Delphi custom CM_TEXTCHANGED message. WM_SETTEXT tells Windows what the text change is, and CM_TEXTCHANGED is used internally by Delphi to update various aspects of the component, e.g. forcing a repaint of the component to show the changed text. With this knowledge, we can change our routine to handle all controls polymorphically, regardless of class:

```
procedure TForm1.ChangeText(Ctrl: TControl; Txt: string);
begin
  Ctrl.Perform(WM_SETTEXT, 0, Longint(PChar(Txt)));
  Ctrl.Perform(CM_TEXTCHANGED, 0, 0);
end;
```

One of the nice things about this technique is that you don't have to worry about a component not understanding the message. Windows messages are ignored by components that don't know what to do with them. This technique is not without its share of problems, however. Unexpected bugs can occur when necessary Delphi source code is bypassed. Also, components may not respond to messages, or they may respond in undesirable ways, forcing us to write class-specific code for those cases. However, this technique gets us closer to a true polymorphic solution. For those uncomfortable with bypassing all that Delphi code, a combined approach using all three solutions presented in this article could be used.

Conclusion

Polymorphic programming is a powerful technique that can be used in many ways. When designing our own classes, we should make extended use of class hierarchies, interfaces, and custom Windows messages. When dealing with existing classes, however, especially those in the VCL, we often run into barriers to polymorphic programming, and end up writing class-specific code that isn't extensible to classes outside those specifically targeted.

This article has demonstrated three methods for bypassing some of those barriers:

- Breaking into the protected level of a class to access properties that are protected in the ancestral class allows us to make use of protected-but-common code in a parent class.
- Using RTTI allows us to polymorphically access properties with the same name, even when their implementation is not defined in a common ancestral class.
- When all else fails, digging deep into the VCL source code can sometimes provide us with the information we need to bypass Delphi entirely, and make polymorphic calls to Windows. ∆

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD in INFORM\00\NOV\DI200011JM.

Jeremy Merrill is an EDS contractor in a partnership contract with the Veteran's Health Administration. He is a member of the VA's Computerized Patient Record System development team, located in the Salt Lake City Chief Information Officer's Field Office.

By Keith Wood

XSL Transformations

Using XSLT to Format XML Documents

In previous issues, we've been introduced to XML, the Extensible Markup Language, and have seen how it can be used to transfer data from one platform or application to another. As noted before, XML was designed to describe data, not to present data (for which HTML is typically used). However, the designers of XML also provided a means of specifying the formatting for an XML document — the Extensible Stylesheet Language (XSL).

This article is an introduction to the XSL Transformations (XSLT) language, its syntax, and semantics. I included an example program that allows us to select an XML document and a corresponding XSL stylesheet, and combine the two to produce HTML. (The sample program is available for download; see end of article for details.) For examples throughout this piece, I'll refer to the movie-watcher XML document described in "Generating XML," an article that appeared in the February 2000 issue of *Delphi Informant Magazine*.

XSL Transformations

XSLT is a language that describes how to manipulate the nodes within an XML document, adding appropriate wrappings to create an output document in some other format. This output is often in HTML for viewing within a browser. However, XSLT can also be used to produce straight text, .rtf, and even other XML documents.

XSL itself is a separate specification that defines a formatting language for use in a presentation, and is outside the scope of this article. XSLT can be used to create an XSL-formatted document, which must then be given to a rendering engine to produce the final display.

A transformation stylesheet is itself an XML document. All the processing is encoded within elements that have a tag starting with <xsl:>, while the remaining elements are output to the final document. The main element of the stylesheet is <xsl:stylesheet>. This requires a version attribute, currently "1.0", and a namespace definition for the <xsl:> elements:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http:// www.w3.org/TR/WD-xsl">
  ...
</xsl:stylesheet>
```

XSLT is designed to be extensible, allowing for the inclusion of any set of formatting instructions. Each set must be identified by its own namespace and must have a reference to its definitions. The namespaces are specified within the <xsl:stylesheet> element as additional <xmlns:> attributes. In the stylesheet body, the namespaces are followed by a colon (:) to denote their scope.

Template and Patterns

The XSLT stylesheet uses a system of templates to match portions of the XML document, and to specify how they're manipulated. At least one template is required to initiate the process, matching with the XML document as a whole:

```
<xsl:template match="/">
  formatting commands for this document as
  a whole
</xsl:template>
```

Subsequent templates may be set up to match on only parts of the XML document hierarchy. To determine which elements a particular template targets, we use the match attribute with a pattern specification. Elements and attributes are identified by name, with attributes being prefixed by an "at" symbol (@). A slash (/) separates elements at different levels, and the asterisk (*) is a wildcard that matches anything.

Conditions may appear with brackets ([ ]) following the element to which they apply. Various function-like names identify particular nodes, either by their type or unique ID attribute, or as the current node of interest. Unless a pattern starts with a slash, making it an absolute reference, all patterns are within the context of the current node. Some sample patterns are shown in Figure 1.

These pattern specifications are used throughout a transformation stylesheet to identify nodes or attributes for subsequent processing. For example, in the <xsl:sort> element, they determine the criteria for ordering nodes. In the <xsl:for-each> element, they select the subset of nodes to process within a loop.

Within the main template, we can invoke other templates through the use of the <xsl:apply-templates> element. When used without a select attribute, this tag applies to all the child nodes of the current node. Specifying a pattern within the select attribute causes only the matching nodes to be processed through their templates. For example, the following affects all the movie elements within the movie-watcher/movies hierarchy:

```
<xsl:apply-templates select="movie-watcher/movies/movie"/>
```

XSLT has a number of built-in templates that provide basic functionality. One of these continues the recursive application of templates when no specific match is found. Another automatically copies all text and attribute nodes straight across to the output. One more matches processing instructions and comments but does nothing with them.

Generally there is only one template for each node type within the XML document. By using named templates, however, it's possible to process the same node in different ways at different times. The name attribute on the <template> tag identifies the template. After this, the <xsl:call-template> element is used to invoke it by specifying the same name.

An alternative method is to use modes. Again, a template can have a mode attribute specified as part of its declaration. If this

Pattern	Matches
/	The root node of the XML document.
*	Any element.
.	The current node.
movie	Any movie element.
director|star	Any director or any star element.
movie/name	Any name element with a movie element parent.
movies//name	Any name element with a movies element ancestor.
text	Any text node.
node	Any node other than the root node or an attribute node.
id("SW1")	The element with "SW1" as its unique ID.
context	The current node (contains the expression).
star[1]	Any star element that is the first child of its parent.
star[last=1]	Any star element that is the only child of its parent.
@url	Any url attribute within an element.
@*	Any attribute within an element.
movie[@rating="PG"]	Any movie element that has a rating attribute of "PG."
/movie-watcher/cinemas/cinema[@id=context/@cinema-id]/name	The name of the cinema element whose ID value is equal to the cinema-ID attribute of the current node (presumably a screening).
id(@cinema-id)/name	The same as above: the name of the element (presumably a cinema) whose ID is equal to the current cinema-ID attribute.

Figure 1: Some sample XSLT patterns.

same mode is supplied in the <apply-templates> tag, then only the applicable template is used. The following example displays only the names of each section within the table of contents:

```
<xsl:apply-templates select="section" mode="contents"/>

<xsl:template match="section" mode="contents">
  <H1><xsl:value-of select="name"/></H1>
</xsl:template>
```

Text Content

Text from the stylesheet is generally copied to the resulting document as is. Nodes that contain only white space are stripped from the document during processing. To retain these text nodes, we can enclose them with an <xsl:text> element.

To include the content of an element or attribute in the text stream, we can use the <xsl:value-of> element. This element's select attribute is a pattern that determines what is written out. Use the "." pattern to retrieve the contents of the current node:

```
<xsl:value-of select="@rating"/>
<xsl:value-of select="length"/> mins
```

Building Document Structure

Any elements in the template that don't belong to the XSLT namespace or to an extension namespace are copied directly across to the output document. In this way, it's easy to create HTML pages using XSLT simply by including them within a template. However, there are times when the tags or their attributes need to be more dynamic. XSLT provides the <xsl:element> and <xsl:attribute> elements for just these purposes.

To create an output element with a computed name, we use the <xsl:element> tag, and set its name attribute. Enclose the name calculation within braces ({ }) to denote it as an expression:

```
<xsl:element name="H{@level}">
  Element contents
</xsl:element>
```

Similarly, attributes can have computed names or values through the use of the <xsl:attribute> tag. This tag must appear within the bounds of the element to which it refers. The following example creates a named anchor, using the ID attribute of the current node:

```
<A>
  <xsl:attribute name="NAME">
    <xsl:value-of select="@id"/>
  </xsl:attribute>
  Anchor contents
</A>
```

Attribute values may also be created directly within the element using the expression technique we just discussed:

```
<A NAME="{@id}">
  Anchor contents
</A>
```

Processing instructions and comments are created within the output document in a similar manner using the <xsl:processing-

instruction> and <xsl:comment> tags. These elements can't be transferred directly from the XSL stylesheet because it's an XML document, and would, therefore, interpret or ignore these as part of its own processing:

```
<xsl:processing-instruction name="xml">
version="1.0" encoding="ISO-8859-1"
</xsl:processing-instruction>
<xsl:comment>Comment within the output document
</xsl:comment>
```

To transfer an existing node from the source XML to the output document, use the <xsl:copy> tag. This doesn't transfer the attributes or the child elements of this node — these must be copied manually.

Loops

The <xsl:for-each> tag provides a looping mechanism within XSLT. It applies its contents to each node selected by the expression in its select attribute:

```
<xsl:for-each select="starring/star">
  <xsl:value-of select="."/><BR/>
</xsl:for-each>
```

Sorting of the selected nodes is achieved through the <xsl:sort> element, which is placed as a child of the looping element. Its select attribute specifies the values to be used for the ordering. Multiple sorting tags allow for primary and secondary sort keys to be supplied. Additional attributes are used to determine ascending or descending sorts, the language to be used, and the data type (text, numeric, or other). Sorting tags can also be used with the <xsl:apply-templates> tag:

```
<xsl:for-each select="movie">
  <xsl:sort select="name"/>
  Movie content
</xsl:for-each>
```

The XSLT engine within Internet Explorer seems to disallow these sorting tags, preferring instead an <order-by> attribute, with the same pattern as the sorting tag:

```
<xsl:for-each select="movie" order-by="name"/>
  Movie content
</xsl:for-each>
```

Conditional Processing

XSL also includes two ways of making decisions within the template. The first is the <xsl:if> tag, which provides a simple "if" test around its contents. We use the test attribute to supply the expression to be evaluated. If this expression refers to an element or an attribute, this item's presence is being tested and the contents will be applied if the item exists. Otherwise the expression must evaluate to a true or false value. The following adds an <HREF> attribute to the output with the value of the source node's <url> attribute as its target, but only if the latter exists:

```
<xsl:if test="@url">
  <xsl:attribute name="HREF"><xsl:value-of select="@url"/>
  </xsl:attribute>
</xsl:if>
```

For an if-then-else or case statement type of test, we need to use the <xsl:choose> tag. This tag has no attributes, but it does contain a number of <xsl:when> tags and an optional <xsl:otherwise> tag. Each <when> tag acts like the <if> tag, specifying an expression to be evaluated in its <test> attribute. There may be several <when> tags within the <choose>, each testing a different condition. The <otherwise> tag is added to process any nodes that didn't get caught by one of the <when> tags (similar to the **else** clause in a Pascal **case** statement). The code shown in Figure 2 tests for the existence of a <logo-url> attribute on the current node, and inserts an IMG element if it's found. If not, the name is added within a header element.

```
<xsl:choose>
  <xsl:when test="@logo-url">
    <IMG>
      <xsl:attribute name="SRC">
        <xsl:value-of select="@logo-url"/></xsl:attribute>
      <xsl:attribute name="ALT">
        <xsl:value-of select="name"/></xsl:attribute>
    </IMG>
  </xsl:when>
  <xsl:otherwise>
    <H3><xsl:value-of select="name"/></H3>
  </xsl:otherwise>
</xsl:choose>
```

Figure 2: Using the <xsl:choose> tag.

XSLT Sample

To bring all of these pieces together, take a look at the XSLT stylesheet in Listing One (beginning on page 19). This transforms the XML data for a movie into HTML suitable for display on the Web. We can see the HTML tags embedded within the XSL processing. Note that the <IMG> tag in HTML doesn't have a closing tag, nor does it have the XML shorthand for closing, i.e. a trailing slash. However, within this stylesheet document, which is XML, the closing tag must be present. The correct handling of this is done by the XSLT engine.

The use of constructed links within the HTML generation provides easy navigation between the parts of the document. Here we connect movie descriptions with the lists of screenings where they're being shown. The <id> and <idref> attributes of the <movie-watcher> elements are needed to allow these connections to be made.

Applying Transformations

The Microsoft XML parser provides a *TransformNode* method on each node within the DOM (Document Object Model). This takes another node as a parameter, being a reference to the root node for a DOM corresponding to the XSL document (remember that these are XML documents, as well). Returned from this processing is a string value that contains the results of the transformation. If we prefer the output as another DOM for further processing, we can use the *TransformNodeToObject* method instead.

To demonstrate how this works, we create an application that asks for an XML document and an XSLT document, before applying one to the other and displaying the output. We assume the result is an HTML page, and so present it in a Web browser component. For each specified file, we create a DOM for it, and then call the *TransformNode* method on the XML root, passing the XSL root as its argument (see Figure 3).

We then save the results to a temporary file and redirect the browser to it. The original files are also loaded into memos for viewing. See Figure 4 for the outcome of applying this transformation to a movie-watcher document.

The accompanying application has one more feature. It displays the XML DOM in a tree view, allowing us to select a particular node (see Figure 5). This node is the one to which the transformation is applied. By loading it in a stylesheet based on a collection of templates, we can transform an individual node, rather than the entire document. Try this with the supplied template-based stylesheet, movie-watcher-t.xsl, and select one of the movie or cinema nodes before applying the transformation.

Conclusion

The power of XML lies in its hierarchical structure and simple encoding scheme. It allows arbitrary data to be described and transferred between applications and platforms. The separation between content and presentation is an inherent part of the XML idea. XSL Transformations are designed to bridge that divide, converting XML documents into a variety of output formats. This article serves as an introduction to XSLT syntax and semantics, and shows how to use its abilities with Delphi. Δ

References

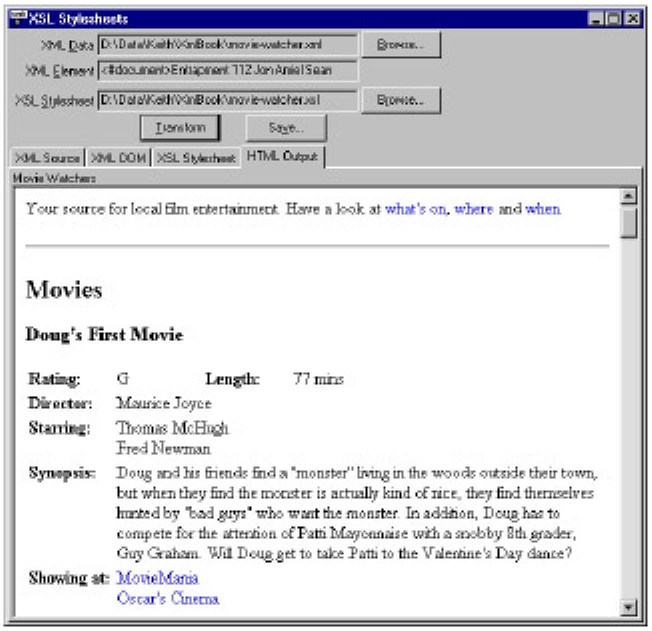
- XSLT Specification: http://www.w3.org/TR/xslt.
- XML Specification: http://www.w3.org/TR/REC-xml.
- Microsoft XML parser: http://msdn.microsoft.com/xml/default.asp.

```
{ Apply the stylesheet to the data and see the results. }
var
  iddXML: IXMLDOMDocument;
  iddXSL: IXMLDOMDocument;
  hRes: HResult;
  stmOut: TFileStream;
  stmString: TStringStream;
  sOutput: string;
begin
  memXML.Lines.Clear;
  memXSL.Lines.Clear;
  try
    { Instantiate the DOMs. }
    hRes := CoCreateInstance(CLASS_DOMDocument, nil,
      CLSCTX_INPROC_SERVER, IID_IXMLDOMDocument, iddXML);
    if hRes <> S_OK then
      raise Exception.Create(sNoDOM);
    hRes := CoCreateInstance(CLASS_DOMDocument, nil,
      CLSCTX_INPROC_SERVER, IID_IXMLDOMDocument, iddXSL);
    if hRes <> S_OK then
      raise Exception.Create(sNoDOM);

    { Load the XML data. }
    memXML.Lines.LoadFromFile(edtXML.Text);
    iddXML.Load(edtXML.Text);
    { Load the XSL stylesheet. }
    memXSL.Lines.LoadFromFile(edtXSL.Text);
    iddXSL.Load(edtXSL.Text);

    { Combine the two and display the results. }
    sOutput := iddXML.TransformNode(iddXSL);
    stmOut := TFileStream.Create(sXSLOutput, fmCreate);
    stmString := TStringStream.Create(sOutput);
    try
      stmOut.CopyFrom(stmString, 0);
    finally
      stmOut.Free;
      stmString.Free;
    end;
    { Load into the browser. }
    brsResults.Navigate(sXSLOutput);
  finally
    { Release the DOMs. }
    iddXML := nil;
    iddXSL := nil;
  end;
end;
```

Figure 3: Applying an XSL Transformation to an XML document and producing HTML.

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD in INFORM\00\NOV\DI200011KW.

Keith Wood is an analyst/programmer with CCSC, based in Atlanta. He started using Borland's products with Turbo Pascal on a CP/M machine. Often working with Delphi, he has enjoyed exploring it since it first appeared. You can reach him via e-mail at kbwood@compuserve.com.



```
XML Data       D:\Data\Keith\XmlBook\movie-watcher.xml
XML Element    <#document>Entrapment 112 Jon Amiel Sean
XSL Stylesheet D:\Data\Keith\XmlBook\movie-watcher.xsl
```

Movie Watchers

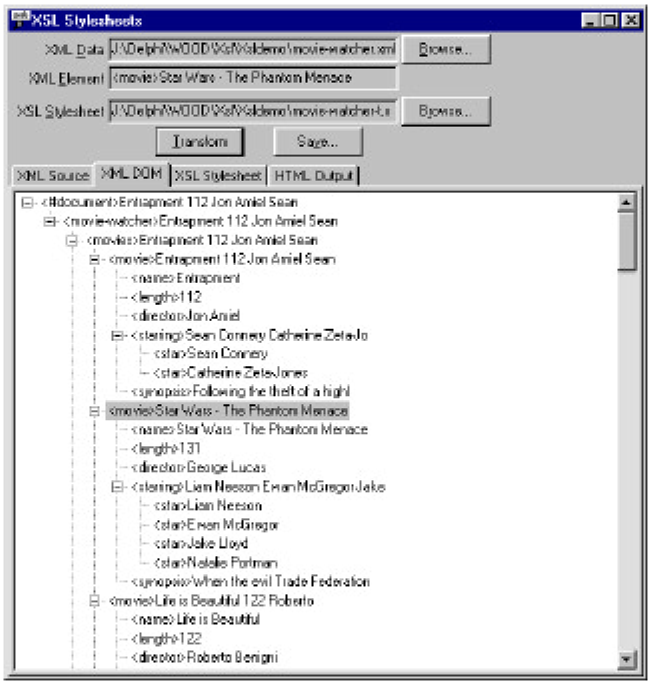
Your source for local film entertainment. Have a look at what's on, where and when.

Movies

Doug's First Movie

Rating:	G	Length:	77 mins
Director:	Maurice Joyce		
Starring:	Thomas McHugh Fred Newman		
Synopsis:	Doug and his friends find a "monster" living in the woods outside their town, but when they find the monster is actually kind of nice, they find themselves hunted by "bad guys" who want the monster. In addition, Doug has to compete for the attention of Patti Mayonnaise with a snobby 8th grader, Guy Graham. Will Doug get to take Patti to the Valentine's Day dance?		
Showing at:	MovieMania Oscar's Cinema		

Figure 4: The sample application in action; displaying the transformation.



```
XML Data       J:\Delphi\WOOD\Xsl\Xsldemo\movie-watcher.xml
XML Element    <movie>Star Wars - The Phantom Menace
XSL Stylesheet J:\Delphi\WOOD\Xsl\Xsldemo\movie-watcher-t.x

⊟ <#document>Entrapment 112 Jon Amiel Sean
  ⊟ <movie-watcher>Entrapment 112 Jon Amiel Sean
    ⊟ <movies>Entrapment 112 Jon Amiel Sean
      ⊟ <movie>Entrapment 112 Jon Amiel Sean
        ├ <name>Entrapment
        ├ <length>112
        ├ <director>Jon Amiel
        ⊟ <starring>Sean Connery Catherine Zeta-Jo
        │ ├ <star>Sean Connery
        │ └ <star>Catherine Zeta-Jones
        └ <synopsis>Following the theft of a highl
      ⊟ <movie>Star Wars - The Phantom Menace
        ├ <name>Star Wars - The Phantom Menace
        ├ <length>131
        ├ <director>George Lucas
        ⊟ <starring>Liam Neeson Ewan McGregor Jake
        │ ├ <star>Liam Neeson
        │ ├ <star>Ewan McGregor
        │ ├ <star>Jake Lloyd
        │ └ <star>Natalie Portman
        └ <synopsis>When the evil Trade Federation
      ⊟ <movie>Life is Beautiful 122 Roberto
        ├ <name>Life is Beautiful
        ├ <length>122
        ├ <director>Roberto Benigni
```

Figure 5: The sample application in action; the XML DOM in a tree view.

Begin Listing One — XSLT stylesheet

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- HTML style sheet for movie-watcher XML -->
<!-- Written by Keith Wood, 4 June 1999 -->
<xsl:stylesheet version="1.0"
  xmlns:xsl="http:// www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
  <HTML>
    <HEAD>
      <TITLE>Movie Watchers</TITLE>
    </HEAD>
    <BODY>
      <H1><A NAME="top">Welcome to Movie Watchers</A></H1>
      <P>Your source for local film entertainment.
        Have a look at <A HREF="#movies">what's on</A>,
        <A HREF="#cinemas">where</A> and
        <A HREF="#screenings">when</A>.</P>
      <HR/>
      <H2><A NAME="movies">Movies</A></H2>
      <xsl:for-each select="movie-watcher/movies/movie"
        order-by="name">
        <!-- Provide link target and optional web link -->
        <A>
          <xsl:attribute name="NAME">
            <xsl:value-of select="@id"/></xsl:attribute>
          <xsl:if test="@url">
            <xsl:attribute name="HREF">
              <xsl:value-of select="@url"/></xsl:attribute>
          </xsl:if>
          <xsl:choose>
            <xsl:when test="@logo-url">
              <IMG>
                <xsl:attribute name="SRC">
                  <xsl:value-of select="@logo-url"/>
                  </xsl:attribute>
                <xsl:attribute name="ALT">
                  <xsl:value-of select="name"/>
                  </xsl:attribute>
              </IMG>
            </xsl:when>
            <xsl:otherwise>
              <H3><xsl:value-of select="name"/></H3>
            </xsl:otherwise>
          </xsl:choose>
        </A>
        <TABLE BORDER="0" WIDTH="100%">
          <TR>
            <TH ALIGN="LEFT" VALIGN="TOP"
              WIDTH="15%">Rating:</TH>
            <TD WIDTH="15%">
              <xsl:value-of select="@rating"/></TD>
            <TH ALIGN="LEFT" VALIGN="TOP"
              WIDTH="15%">Length:</TH>
            <TD><xsl:value-of select="length"/> mins</TD>
          </TR>
          <TR>
            <TH ALIGN="LEFT" VALIGN="TOP">Director:</TH>
            <TD COLSPAN="3">
              <xsl:value-of select="director"/></TD>
          </TR>
          <TR>
            <TH ALIGN="LEFT" VALIGN="TOP">Starring:</TH>
            <TD COLSPAN="3">
              <xsl:for-each select="starring/star">
                <xsl:value-of select="."/><BR/>
              </xsl:for-each>
            </TD>
          </TR>
          <TR>
            <TH ALIGN="LEFT" VALIGN="TOP">Synopsis:</TH>
            <TD COLSPAN="3">
              <xsl:value-of select="synopsis"/></TD>
          </TR>
          <TR>
            <TH ALIGN="LEFT" VALIGN="TOP">Showing at:</TH>
            <TD COLSPAN="3">
              <xsl:for-each select="/movie-watcher/
                screenings/screening[
                @movie-id=context()/@id]">
                <A>
                  <xsl:attribute name="HREF">
                    #<xsl:value-of select="@movie-id"/>-
                    <xsl:value-of select="@cinema-id"/>
                    </xsl:attribute>
                  <xsl:value-of select=
                    "id(@cinema-id)/name"/>
                </A><BR/>
              </xsl:for-each>
            </TD>
          </TR>
        </TABLE>
      </xsl:for-each>
      <P>Back to <A HREF="#top">the top</A>.</P>
        ...
      <HR/>
      <P>Movie Watcher data supplied by
        <A HREF="mailto:kbwood@compuserve.com">Keith Wood
        </A>.</P>
    </BODY>
  </HTML>
</xsl:template>
</xsl:stylesheet>
```

End Listing One

By Alex Fedorov and Natalia Elmanova

A Practical Guide to ADO Extensions

Part II: ADO Multidimensional

Last month, we provided an overview of two ADO extensions: ADO Extension for DDL and Security (ADOX), and Jet and Replication Objects (JRO). This month, we complete this two-part series by discussing the third ADO extension, ADO Multidimensional (ADO MD), which is used to access multidimensional data stores.

Such data stores are often used in data warehousing and Online Analytical Processing (OLAP). So we'll begin our tour of ADO MD with a brief introduction to OLAP, data warehousing, and multidimensional data storage in general.

OLAP and Data Warehousing in Brief

Data analysis and decision support are critical tasks for many database applications. As a rule, an implementation of such analysis is based on OLAP, a popular technology for multidimensional business analysis. The concept of OLAP was described in 1993 by Dr E.F. Codd, the well-known database researcher and inventor of the relational database model. OLAP support is now implemented in many different DBMSes and tools. If you're familiar with Delphi Decision Cube components, you've already seen a simple OLAP implementation; the Decision Cube application is a primitive OLAP tool.

Let's look at what data the OLAP storage usually contains. Imagine a trading company that registers its data in a database that contains an Invoices view with the details of invoices. Let's suppose that querying this view results in the dataset shown in Figure 1.

Let's assume we want to know the sum of payments for all German customers. The SQL query for this is:

```
SELECT SUM(Payments)
  FROM Sales
 WHERE Country = 'Germany'
```

Of course, we could replace Germany with Austria, UK, or another country name. As a result, we'll receive a one-dimensional set of summaries — one for each country (see Figure 2).

Now let's make this query more complex. For example, we can ask for the sum of payments for all German invoices for vegetables:

```
SELECT SUM(Payment)
  FROM Invoices
 WHERE Country = 'Germany'
   AND ProductCategory = 'Vegetables'
```

If we examine all the possible combinations of product category and country name, we'll receive a two-dimensional table of summaries (see Figure 3).

Date	Product Category	Product Subcategory	Product Name	Country	City	Sales Person	Payment
01.01.99	Vegetables	Canned Vegetables	Canned Tomatoes	Germany	Berlin	Nicolas Wilson	$1,280
01.01.99	Vegetables	Fresh Vegetables	Dried Mushrooms	UK	London	Daniel Adams	$514
01.02.99	Dairy	Cheese	Cheddar Cheese	Germany	Frankfurt	Nicolas Wilson	$723
01.02.99	Dairy	Cheese	Gorilla Cheese Spread	Austria	Vienna	Nicolas Wilson	$330
01.03.99	Vegetables	Canned Vegetables	Canned Tomatoes	UK	London	Daniel Adams	$439
...	...	...	...	...	...	...	...

Figure 1: A dataset from a database query.

Germany	Austria	UK	USA	...
$2,003	$330	$953	$5,321	

Figure 2: A one-dimensional set of summaries.

	Vegetables	Dairy	Drinks	...
Germany	$1,280	$723	$239	...
UK	$514	$0	$732	...
Austria	$0	$330	$0	...
...	...	...	...	...

Figure 3: A two-dimensional table of summaries.

Such tables are also known as cross tables (or crosstabs), or pivot tables. The first dimension of the table is Country, and the second is ProductCategory.

Let's modify our query again to ask for the sum of payments for all German orders for vegetables sold by salesperson Nicolas Wilson:

```
SELECT SUM(Payments)
  FROM Sales
 WHERE Country = 'Germany'
   AND ProductCategory = 'Vegetables'
   AND SalesPerson = 'Nicolas Wilson'
```

If we examine all possible combinations of ProductCategory, Country, and SalesPerson, we'll receive a three-dimensional set of summaries that can be represented as a "cube" of data. We can continue to make the query more complex (adding dimensions to our data set), and receive appropriate summaries. This is what a Decision Cube application does: It calculates such summaries and stores them in its memory cache.

There can be several sets of summaries, e.g. the number of orders, average prices, etc. Such summaries are sometimes called measures.

The next idea we'll explore is the hierarchical structure of dimensions. For example, if one of the dimensions is a date or date/time field, we can obtain summaries for different years, as well as quarters, months, days, etc. (This is the only type of hierarchy supported by Decision Cube.) We can also decide to compare summaries for similar time periods (e.g. for all Wednesdays, or for January 1999 and January 2000), or define another type of hierarchy (e.g. Country/State/City, Product Category/Product, or Subcategory/Product Name).

Rather than keeping summaries in memory, a more progressive idea is to store calculated summaries in a database. This is the core idea of data warehousing that is OLAP-based, i.e. the process of collecting and sifting data from different information systems, and making the resulting information available to end users for analysis and reporting. Data warehouses are used to describe stores of collected and summarized data. As a rule, from the "logical" point of view, such storages have non-relational data structures. Most server DBMS vendors provide server-side OLAP tools for creating and using multidimensional storages for data warehousing. This method of data analysis is preferable to using client-side tools such as Decision Cube components, because it doesn't require transferring the original data to the client application, or recalculating summaries every time they are necessary.

In this article, we'll use the Microsoft SQL Server OLAP Extensions to illustrate how to use the ADO Multidimensional objects. At the time of this writing, the OLE DB Provider for OLAP Services that comes with Microsoft SQL Server OLAP Extensions and with Microsoft Office is the only available OLE DB provider that allows access to multidimensional data. With this provider, we can read data stored in Microsoft SQL Server multidimensional databases, or in local .cub files that can be created with Microsoft Excel 2000. However, we can expect OLE DB providers for other OLAP servers to appear in the near future.

A discussion of creating multidimensional stores is outside the scope of this article. Doing this programmatically requires using the SQL DSO (Decision Support Objects) library.

To illustrate how ADO MD works, we'll use the FoodMart sample multidimensional database that comes with Microsoft SQL Server 7.0 OLAP Extensions. It's also possible to use any local .cub file created with Microsoft Excel 2000, in which case, of course, you must have Excel 2000.

ADO MD Objects

We'll begin with the ADO MD object model that's available to Delphi applications. The ADO MD object model has two main branches (see Figure 4). The first branch is used to access metadata of multidimensional databases. The second is used when we need to retrieve the data stored in this database by querying OLAP cubes.

Objects for Retrieving Metadata

The first of the ADO MD objects for retrieving metadata of multidimensional databases, the *Catalog* object, represents the particular multidimensional data store. Such stores contain zero, one, or more cubes, and, therefore, the *Catalog* object contains the *CubeDefs* collection. Any element in this collection is a *CubeDef* object that represents a particular cube in storage. The name of a cube is the value of the *Name* property of an appropriate *CubeDef* object. In the FoodMart sample database, there are three cubes: Sales, Warehouse, and Warehouse and Sales.

The multidimensional cubes can contain — you guessed it — several dimensions. Correspondingly, any of the *CubeDef* objects can contain the *Dimensions* collection, which contains *Dimension* objects. Each *Dimension* object represents a particular dimension of the cube, with the name of the dimension stored in its *Name* property. For example, the Sales cube in the FoodMart

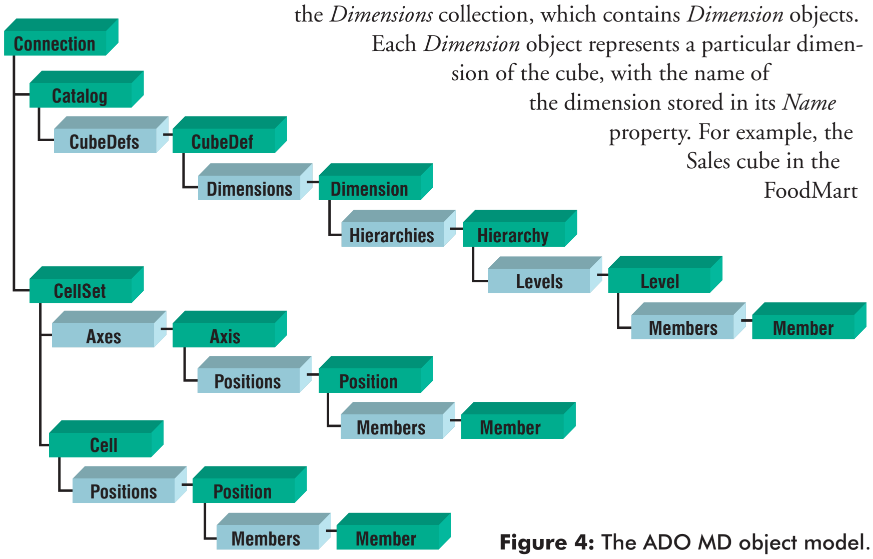


Figure 4: The ADO MD object model.

database contains several dimensions: Store, Time, Product, Promotion Media, Promotions, Customers, and so on.

As we have discussed before, a dimension's data can be hierarchical. Correspondingly, the *Dimension* object contains the *Hierarchies* collection, which theoretically can contain one or more *Hierarchy* objects. In practice, however, there's only one hierarchy for each dimension (at least, for Microsoft SQL Server OLAP Extensions), so this collection will consist of a single item.

The dimension hierarchy can contain one or more levels. Therefore, the *Hierarchy* object contains the *Levels* collection of *Level* objects. For example, the hierarchy of the Store dimension in the Sales cube contains four levels, with the *Name* property values of the corresponding *Level* objects being Store Country, Store State, Store City, and Store Name.

Any level of a hierarchy can contain one or more members that can be source database field values, or values obtained by calculation, i.e. grouping. Therefore, the *Level* object contains the *Members* collection of *Member* objects. For example, the Store Country level of the Store dimension hierarchy contains three members: Canada, Mexico, and the US. The Store State level of the same hierarchy contains 10 members: CA, OR, WA, and several states in Mexico and Canada. And the Store City contains members that are cities in those states.

All of these objects have a wide list of properties and methods. You can find a detailed description of them in the MSDN Library.

Objects for Retrieving Data

The second branch of the ADO MD object model is used for retrieving the "cube" slices (usually they're two-dimensional pivot tables similar to those shown at the beginning of this article). For obtaining such slices, we need to query the cube. Because a multidimensional database isn't relational, we can't use standard SQL to query it. Instead, we need to describe the rows and columns, and which summaries we used in the slice. To do this, we need to use Multidimensional Expressions (MDX), the SQL extension for querying OLAP cubes. A typical multidimensional expression looks like this:

```
SELECT axis_specification ON COLUMNS,
       axis_specification ON ROWS
  FROM cube_name
 WHERE slicer_specification
```

where *axis_specification* describes the horizontal or vertical axis. For example, [Store].[Store Country].MEMBERS is the list of countries, and [Store].[Mexico].CHILDREN is the list of the states in Mexico. The *slicer_specification* is the name of the summary (measure) that will be used to create a pivot table. For example, the following MDX query:

```
SELECT [Time].[Quarter].MEMBERS ON COLUMNS,
       {[Store].[USA].CHILDREN,
        [Store].[Canada].CHILDREN} ON ROWS
  FROM Sales
 WHERE [Measures].[Profit]
```

will result in a pivot table that contains the quarterly profit for all states of the US and Canada. You can find more details of the MDX syntax and its keywords in the Platform SDK.

In the ADO MD object model, the pivot table that results from the MDX query is represented by a *CellSet* object. This object provides array-like access to the *Cell* objects that represent particular cells in the pivot table. In addition, this object contains the *Axes* collection of *Axis* objects (there are two objects in this collection). The *Cell* object and the *Axis* object have the *Positions* collection of *Position* objects that represent positions along the axis. The *Position* object has a *Members* collection that represents a particular data value in the axis.

Now that we're familiar with the ADO MD objects, we can use them to create a simple OLAP manager.

Creating a Simple OLAP Manager

Let's use the ADO MD objects with Delphi. We'll create an application that can:

- display cube metadata in a tree view;
- copy the selected names of tree view nodes and MDX keywords from a predefined list into a Memo component where an MDX query is edited; and
- execute an MDX query and copy the results to the client dataset to present in a DBGrid component.

To perform this task, we'll create a new project and place several components on its main form: a ToolBar with eight buttons, and TreeView, ListBox, DBGrid, ClientDataSet, DataSource, and Memo components. Then, we'll set the *DataSource* property of the *DBGrid1* component to `DataSource1`, and the *DataSet* property of the *DataSource1* component to `ClientDataSet1`. The *ListBox1* component should be filled with MDX keywords, such as CHILDREN, MEMBERS, DESCENDANTS, etc. (The finished application is shown in Figure 6. It's available for download; see end of article for details.)

We need to refer to the ADO MD Type Library contained in the MSADOMD.DLL file, because ADO MD isn't supported by Delphi 5 at the component level. To do this, select **Project | Import Type Library**, and pick Microsoft ActiveX Data Objects (Multi-dimensional) 1.0 Library from the list of available type libraries. Please note that if you've already imported the ADOX type library and haven't renamed the Delphi class for the ADOX *Catalog* object, the Delphi *TCatalog* class is already declared. To avoid conflict with this declaration, just rename *TCatalog* to *TADOMDCatalog*. It's also desirable to uncheck the **Generate Component Wrapper** checkbox, because we only need to create a .pas file to access ADO MD objects. Click the **Create Unit** button. This will generate the ADOMD_TLB.PAS file, which is the interface unit to the ADO MD type library. Now we need to include a reference to this file in our application's **uses** clause, as well as references to the ComObj and ADODB units.

Now we need to connect to the multidimensional database. The connection string that will be used for this should include the OLE DB Provider name (in our case, it will be OLE DB Provider for OLAP Services, and you need to be sure it's installed), the computer name, and the database name:

```
DS := 'Provider=MSOLAP.1;Data Source=localhost; ' +
      'Initial Catalog=FoodMart';
```

Again, you can also connect to the local cubes stored in Excel .cub files. In this case, the connection string should be similar to this:

```
DS :=
  'Provider=MSOLAP.1;Data Source=C:\Data\Cubes\NW1.cub';
```

We'll place the code responsible for connecting to the multidimensional database in the *OnClick* event handler for one of the buttons.

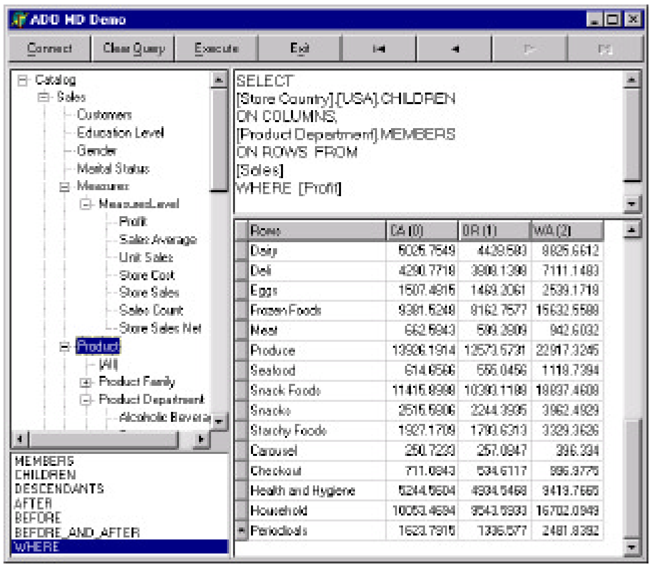
This code, along with the procedure that fills the *TreeView1* component with the names of cubes, is shown in Figure 5. Here we connect to the database, create a *Catalog* object, and iterate through its *CubeDefs* collection to retrieve their names, i.e. the value of the *Name* property for each *CubeDef* object.

The real processing of cube metadata is implemented in the *OnMouseDown* event handler of the *TreeView1* component shown in Listing One. Here we define the type of node just clicked (a node for a cube, dimension, level, or member) using its *Level* property, and whether its child nodes are already downloaded from the database. If they haven't been downloaded (i.e. the *Count* property of the node is zero), we download them and create child nodes using the respective *Name* property of the ADO MD object. If there's nothing to download, we copy the node name to the current cursor position of the *Memo1* component.

The code for copying the MDX keywords to *Memo1* is also shown in Listing One. Now we have a tool for browsing cube metadata,

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DS := 'Provider=MSOLAP.1;Data Source=localhost;' +
        'Initial Catalog=FoodMart';
  FillTreeView(DS);
end;

procedure TForm1.FillTreeView(DataSource: WideString);
var
  I : Integer;
begin
  // Create a new Catalog object.
  Catalog1 := CoCatalog.Create;
  TreeView1.Items.Clear;
  RootNode := TreeView1.Items.Add(nil, 'Catalog');
  // Connect to the multidimensional database.
  Catalog1._Set_ActiveConnection(OleVariant(DataSource));
  // Iterate through cubes and retrieve their names.
  for I := 0 to Catalog1.CubeDefs.Count-1 do begin
    CubeDef1 := Catalog1.CubeDefs[I] as CubeDef;
    CubeDefNode :=
      TreeView1.Items.AddChild(RootNode, CubeDef1.Name);
  end;
end;
```

Figure 5: Connecting to the multidimensional database.



```
SELECT
[Store Country].[USA].CHILDREN
ON COLUMNS,
[Product Department].MEMBERS
ON ROWS FROM
[Sales]
WHERE [Profit]
```

Rows	CA (0)	OR (1)	WA (2)
Dairy	5026.7549	4429.583	9825.6612
Deli	4290.7719	3909.1398	7111.1483
Eggs	1507.4915	1469.2061	2539.1719
Frozen Foods	9381.5249	9162.7577	15632.5589
Meat	662.5943	599.2909	942.6032
Produce	13926.1914	12573.5731	22917.3246
Seafood	614.6566	565.0456	1118.7394
Snack Foods	11415.8989	10390.1189	19037.4609
Snacks	2515.6906	2244.3935	3962.4929
Starchy Foods	1927.1709	1790.6313	3329.3626
Carousel	250.7229	257.0847	396.334
Checkout	711.0843	534.6117	996.3775
Health and Hygiene	5244.5604	4934.5468	9419.7665
Household	10053.4694	9543.5933	16702.0949
Periodicals	1623.7915	1336.577	2481.8392

Figure 6: The sample application: a simple OLAP manager.

and creating the text of MDX queries by clicking on tree nodes or keyword list items.

The next step in creating our OLAP manager is to execute the MDX query entered in the *Memo1* component, and fill the client dataset with the results. This functionality is implemented in the *CDSFill* procedure shown in Listing Two (on page 24). Here we create a *CellSet* object and call its *Open* method. If the MDX query is valid, we create an empty client dataset and fill its field names with the *Caption* property values of the first items in the *Members* collections of all items in the *Position* collection of the first *Axis*.

The field names in datasets should be unique. However, in real multidimensional databases, the *Caption* property values of *Members* can be duplicated. For example, in the Year/Month hierarchy, the caption for January 1999 and January 2000 members can both be "January". There are many ways to avoid duplicate field names. We've selected the simplest, i.e. add a unique number contained in the *Ordinal* property of the *Member* object.

After the field names of the client dataset are defined, we iterate through the *CellSet* rows. For each row, we place the *Caption* property value of the first item in the *Members* collection of the respective item in the *Position* collection of the second *Axis*. Then we place the values of the appropriate *Cell* objects available through the *Item* collection of the *CellSet* object. As a result, we'll receive a *DBGrid1* component filled with a data slice (see Figure 6).

We have created a simple OLAP manager using ADO MD. This is just a simple example to illustrate the possibilities of ADO MD. They can be extended of course — for example, by adding charting possibilities, or a more sophisticated MDX query generator.

Conclusion

In this series, we've discussed the ADO extensions: ADO Extension for DDL and Security (ADOX), Jet and Replication Objects (JRO), and ADO Multidimensional (ADO MD). They allow us to provide our users with database applications with complex and useful functionality that is unavailable with ADO Express components alone.

We are thankful to Dan Miser for his advice while creating the code used in this example. ∆

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD in INFORM\00\NOV\DI200011AF.

Alex Fedorov is an executive editor for *ComputerPress* magazine published in Moscow. He's one of the co-authors of *Professional Active Server Pages 2.0* [Wrox Press, 1998] and *ASP 2.0 Programmer's Reference* [Wrox Press, 1999]. Natalia Elmanova, Ph.D., is an Associate Professor of the Sechenov's Moscow Medical Academy and a freelance Delphi/C++Builder programmer, trainer, and consultant. She was a speaker at the 10th Annual Inprise/Borland Conference. Natalia and Alex are authors of *Advanced Delphi Developer's Guide to ADO* [Wordware Publishing, 2000], and several programming books written in Russian. You can visit their Web site at http://d5ado.homepage.com.

Begin Listing One — *TreeView1MouseDown*

```
procedure TForm1.TreeView1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  HitTest   : THitTests;
  CurrNode  : TTreeNode;
  I         : Integer;
  NodeName  : string;
  AddString : string;
begin
  HitTest := TreeView1.GetHitTestInfoAt(X,Y);
  // Check if one of the tree nodes is clicked.
  if (htOnItem in HitTest) then begin
    CurrNode := TreeView1.GetNodeAt(X, Y);
    // If the node can have child nodes, but they aren't
    // added yet, we will add them.
    if ((CurrNode.Count=0) and (CurrNode.Level<4)) then
      case CurrNode.Level of
        1: // This node represents a cube.
          begin
            CubeDef1 :=
              Catalog1.CubeDefs.Get_Item(CurrNode.Text);
            // Iterate through dimensions of the cube
            // and retrieve their names.
            for I := 0 to CubeDef1.Dimensions.Count-1 do
              begin
                Dimension1 :=
                  CubeDef1.Dimensions[I] as Dimension;
                DimNode := TreeView1.Items.AddChild(
                  CurrNode, Dimension1.Name);
              end;
          end;
        2: // This node represents a dimension.
          begin
            CubeDef1 := Catalog1.CubeDefs.Get_Item(
              CurrNode.Parent.Text);
            Dimension1 := CubeDef1.Dimensions.Get_Item(
              CurrNode.Text);
            // Iterate through levels of hierarchy of
            // this dimension.
            for I := 0 to Dimension1.Hierarchies[0].
                                    Levels.Count-1 do begin
              Level1 := Dimension1.Hierarchies[0].
                          Levels[i] as Level;
              LevelNode := TreeView1.Items.AddChild(
                             CurrNode, Level1.Name);
            end;
          end;
        3: // This node represents a level.
          begin
            CubeDef1 := Catalog1.CubeDefs.Get_Item(
                          CurrNode.Parent.Parent.Text);
            Dimension1 := CubeDef1.Dimensions.Get_Item(
                            CurrNode.Parent.Text);
            Level1 := Dimension1.Hierarchies[0].Levels.
                        Get_Item(CurrNode.Text);
            // Iterate through members of this level.
            for I := 0 to Level1.Members.Count-1 do begin
              Member1 := Level1.Members[I] as Member;
              MemberNode := TreeView1.Items.AddChild(
                              CurrNode, Member1.Name);
            end;
          end;
      end  // case
    else
      // If the node already has child nodes, or there is
      // nothing to download, we will copy the node name to
      // the MDX query editor.
      // If this isn't the root node...
      if Currnode.Level > 0 then begin
        CurrNode := TreeView1.GetNodeAt(X, Y);
        NodeName := CurrNode.Text;
        // Copy the node name formatted according to the
        // MDX syntax to the MDX query editor.
        if ((CurrNode.Level=1) or
            (CurrNode.Parent.Parent.Text='Measures')) then
          AddString := '[' + NodeName + ']'
        else
          AddString := '[' + NodeName  + '].';
        Memo1.SetSelTextBuf(PChar(AddString));
      end;
  end;
end;

procedure TForm1.ListBox1Click(Sender: TObject);
var
  AddString : string;
begin
  // Add an MDX keyword from the Listbox to the MDX
  // query editor.
  AddString := Listbox1.Items[Listbox1.ItemIndex] + ' ';
  Memo1.SetSelTextBuf(PChar(AddString));
end;
```

End Listing One

Begin Listing Two — *CDSFill*

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  CDSFill(DS);
end;

procedure TForm1.CDSFill(DataSource: WideString);
var
  I, J : Integer;
  V    : OleVariant;
begin
  // Create a new CellSet object.
  CellSet1 := CoCellSet.Create;
  try
    // Execute an MDX query in memo and open a cellset.
    CellSet1.Open(Memo1.Text,DataSource);
    with ClientDataSet1 do begin
      Close;
      with FieldDefs do begin
        // Clear all field definitions in a client dataset.
        Clear;
        // Add new field definitions.
        // Here is the first one for row names.
        with AddFieldDef do begin
          Name := 'Rows';
          DataType := ftString;
        end;
        // Iterate through the Positions collection
        // of the first axis.
        for I := 1 to
                 CellSet1.Axes[0].Positions.Count do begin
          with AddFieldDef do begin
            // The field value of the source database will
            // be the column name. In datasets, the column
            // names should be unique, so we will add the
            // unique number containing in the Ordinal
            // property of the Position object to the
            // field value.
            Name := CellSet1.Axes[0].Positions[I-1].
              Members[0].Caption + '(' + IntToStr(
              CellSet1.Axes[0].Positions[I-1].Ordinal)+')';
            DataType := ftFloat;
          end;
        end;
      end;
      // Create and open the client dataset.
      CreateDataSet;
      Open;
      // Add rows to the client dataset.
      for J := 1 to
                 CellSet1.Axes[1].Positions.Count do begin
        // Add a row.
        Append;
        // Add the row name using the Position collection
        // item of the second axis.
        Fields[0].Value := CellSet1.Axes[1].Positions[J-1].
                             Members[0].Caption;
        // Iterate through the cells in the row.
        for I := 1 to
                  CellSet1.Axes[0].Positions.Count do begin
          // Create an array of cell coordinates.
          V := VarArrayCreate([0,1], varVariant);
          V[0] := I-1;
          V[1] := J-1;
          // If the cell in the cellset is not empty...
          if CellSet1.Item[PSafeArray(TVarData(V).VArray)].
                                    FormattedValue <> '' then
            // ...the field value is set to the cell data.
            Fields[I].Value := Cellset1.Item[PSafeArray(
                                   TVarData(V).VArray)].Value
          else
            // Otherwise, the field value is set to zero.
            ClientDataSet1.Fields[I].Value := 0;
        end;
      end;
      // Close the cellset and free resources.
      CellSet1.Close;
      CellSet1 := nil;
    end;
  except
    ShowMessage('Invalid MDX Query');
  end;
end;
```

End Listing Two

By Nikolai Sklobovsky

Children of Threadmare

Sharing Progress with Users Intelligently

In the article "Waking from Threadmare," which appeared in the June 2000 issue of *Delphi Informant Magazine*, we discussed different problems a novice programmer might encounter when dealing with multiple threads. We also outlined a few strategies for solving those problems. Our main weapon in the fight with the *TThread* demon was a powerful component named *TActivityProcessLog*, and its lightweight sidekick, *TCustomActivityLogClient*. Our sample application provided the diligent reader with answers to the natural how's and why's. Due to obvious article space limitations, however, we could only scratch the surface of its possible uses.

Now, it's time to take a closer look at the client side and see what perks are immediately available to us. Throughout this article we'll discuss this in greater detail, and learn how to fulfill various tasks that range from the simplest to the most sophisticated. Our examples will cover the following widely used areas:

- short-term process progress display, e.g. loading a document;
- advanced interruptible short-term process progress display, e.g. generating data and finding text/data;
- status bar meets progress bar, i.e. background processing;
- and, finally — the ultimate weapon — a progress dialog box component.

In contrast to the article in June's issue, which pursued mostly conceptual goals, this text takes a more practical approach. This means you will see a lot of "inline" code. Sample application code from this article and "Waking from Threadmare" are available for download; see end of article for details.

```
DisplayForm = class(TRunTimeForm)
private
  FPanel: TPanel;
  FTaskList: Pointer;
  FCursor: TCursor;
  FPumpOnUpdate: Boolean;
  FDoNotDecreaseAutoWidth: Boolean;
  FCancelled: Boolean;
  procedure FormKeyPress(Sender: TObject; var Key: Char);
protected
  procedure DoShow; override;
  procedure DoHide; override;
public
  constructor Create; reintroduce; overload;
  procedure SetMessage(const csMessage: string;
    bAutoSize: Boolean = True);
  property  PumpOnUpdate: Boolean
    read FPumpOnUpdate write FPumpOnUpdate;
  property  DoNotDecreaseAutoWidth: Boolean
    read FDoNotDecreaseAutoWidth
    write FDoNotDecreaseAutoWidth default True;
  property  Cancelled: Boolean
    read FCancelled write FCancelled;
end;

// Helper function - wrapper for a constructor.
function CreateDisplayForm(const csMessage: string;
  iWidth: Integer = -240; iHeight: Integer = 40):
  TDisplayForm;
```

Figure 1: The interface portion of *TDisplayForm*.

Auxiliary Tools

Before we can proceed with our main goal of catching thread/process feedback, we need to develop a simple visual tool for displaying it. What kind of tool do we need? Something simple that will allow us to display a line of text that will eventually be replaced with a new line, while disabling a user's ability to manipulate the application.

To achieve this, we're going to create a special form, conveniently named *TDisplayForm*. Its interface portion is shown in Figure 1.

Why are we deriving this form from some mysterious *TRunTimeForm*, rather than simply from *TForm*? The answer will become obvious if you're willing to experiment and dare to change the base class to *TForm*. The inherited *TCustomForm.Create* constructor will detect that our new form isn't a *TForm*, and will attempt to locate and load its resource part (originally stored in a .dfm file). As long as there's no existing RunTimeForm.dfm file, an exception will be raised, effectively preventing us from using anything that is not a *TForm*.

To overcome this, we create an heir to *TForm* and supply it with the overridden *Create* constructor. Its interface and implementation are straightforward (see Figure 2). In essence, it mimics the standard *TCustomForm* constructor, skipping only its resource-locating segment.

Now let's return to our Display form. All of its methods are shown in Listing One (on page 32). The constructor creates and initiates the necessary visible controls. The *SetMessage* and *FormKeyPress* procedures are also obvious. The *DoShow* and *DoHide* methods, however, require some comment. They use one of the very powerful Delphi commodities — the set of task list routines. This "sandwich" consists of two methods: *DisableTaskWindows* and *EnableTaskWindows*. The first call disables all of the application's high-level windows (except the one passed as a parameter) exactly as the *ShowModal* method does. The other one performs the reverse operation.

Although not obvious, the advantage of using this technique rather than the traditional *ShowModal* call is indisputable. *ShowModal* is a

```
interface
  ...
  // This form allows any descendant form to be created
  // without a DFM file.
  TRunTimeForm = class(TForm)
  public
    constructor Create(AOwner: TComponent); override;
  end;
  ...
implementation
...
constructor TRunTimeForm.Create(AOwner: TComponent);
begin
  GlobalNameSpace.BeginWrite;
  try
    inherited CreateNew(AOwner);
    // Resource-locating-loading part skipped.
  finally
    GlobalNameSpace.EndWrite;
  end;
end;
```

Figure 2: The interface and implementation of *TForm*.

```
procedure ScanDataSet(Table: TDataSet;
  bSingleThreaded: Boolean);
var
  frm: TDisplayForm;
begin
  frm := CreateDisplayForm('Scanning the table...');
  with Table do
    try
      frm.PumpOnUpdate := bSingleThreaded;
      First;
      while not Eof do begin
        if frm.Cancelled then
          Break;  // User hit [Esc].
        // We don't need to react on every record.
        if RecNo mod 1000 = 0 then
          frm.SetMessage(Format(
            'Records processed: %d out of %d',
            [RecNo, RecordCount]));
        // Do something with table here.
        Next;
      end;
    finally
      frm.Free;
    end;
end;
```

Figure 3: Capitalizing on our asynchronous *CreateDisplayForm*.

synchronous call, which forces you to move all of your business logic to the form being displayed. Our *CreateDisplayForm* call is asynchronous, allowing us to show the form and immediately proceed with our errands. Furthermore, the business code doesn't require a link to the UI element, and thus may reside wherever we choose. We'll see more real-life examples of its usage down the road. In the meantime, a simple, but helpful example is shown in Figure 3.

The Display form provides us with an adequate means for displaying one line of variable text in a modal way, and for intercepting the user's possible cancellation instructions. You can easily expand its capability by adding pictures or animations, or by allowing for more text. Even without multiple threads, this object is already valuable. And wait until you see how it works with our activity log component.

Displaying Loading Info

One of the typical tasks programmers encounter when dealing with a new middle-size project is a splash screen display. Traditionally, this task is handled in the manner shown in Figure 4. Although there's nothing wrong with this approach, sometimes only a few seconds are required to load an application. In this scenario, displaying an About screen with its sizable graphical content would be a waste of time and resources. Loading it may consume 50 or 100 percent of the otherwise relatively short loading time. Remember that your primary goal in this case is to make the loading time a little bit more reasonable, not 100 percent longer.

Another reason for not displaying an About screen is because, quite often, you're dealing with some kind of document processing and thus, you may need to display something "splashy" many times during the session, i.e. each time you change the document. During these changes, you definitely don't want your user to hit various buttons. Due to the data dependency, you also have no idea how long each load process might take. However, it would be unfair to leave the user face-to-face with only the hourglass cursor. This is where our Display form and activity log may come in handy.

Let's consider the following example. Our application will be processing a file of records with variants. Apart from the other important business information, each record holds a "type" field. Let's assume

```
begin  // My project.
  Application.Initialize;
  Application.Title := 'Children of Threadmare';
  with TfrmAbout.Create(Application) do
    try
      // Remove visual reminder of form's "closeability."
      btnOK.Hide;
      BorderIcons := [];
      // The only place we can say something, and prevent
      // user from doing anything to it.
      Caption := 'Loading, please wait...';
      Enabled := False;
      Show;
      // Inevitable evil; otherwise it won't be
      // properly drawn.
      Update;
      Application.CreateForm(TfrmMain, frmMain);
      // Some additional initialization code goes here.
      // You can use sleep(3*1000) to imitate it.
    finally
      Free;
    end;
  Application.Run;
end.
```

Figure 4: Traditional splash-screen implementation.

we're asked to provide a simple navigator to display all the record types in a narrow list. When a user clicks on a particular row, the record details must be displayed in the main part of the window. (We're not going to do this in the sample code, but we'll do everything else.)

To achieve this, we may need to read the entire file immediately upon opening it to collect all the record types. (Let's assume, for the sake of the overall task, that we have to do it for some important reason.) Because this step will take a relatively short, but unpredictable, length of time, it would be nice to provide some progress information.

We purposely use a highly simplified and somewhat old-fashioned approach to the data processing here. This article isn't devoted to I/O optimization or data structures design. The real code may use asynchronous I/O routines, caching, streaming, reading ahead, and all the other usual optimization techniques. Here, we're only concentrating on the visualization aspects. Our file-reading code looks relatively simple, and is shown in Listing Two (on page 32).

So far we have only a few log-related lines, namely its initialization and finalization in the *OpenDataFile* procedure. As long as we don't use multiple threads in this case, it's wise to take complete control and set the log's *AutoFinalize* property to False. Of course, the *UsePumping* property should be set to True; otherwise everything would appear frozen until the end of the process.

To fulfill our assignment, we only need to set up a few of the log's event handlers. At the initialization point, we will create the Display form, which will eventually be released in *OnFinalize*. Also, we'll collect some information during each *OnProcessUpdate* event, and display it at *OnDataUpdateFinish* (see Figure 5). As a result of our efforts, we finally get something like that shown in Figure 6.

```
procedure TfrmMain.Log1Initialize(
  Sender: TActivityProcessLog);
begin
  FForm := CreateDisplayForm('Working, please wait...');
end;

procedure TfrmMain.Log1Finalize(
  Sender: TActivityProcessLog);
begin
  FreeAndNil(FForm);
end;

procedure TfrmMain.Log1DataUpdateStart(
  Sender: TActivityProcessLog);
begin
  FMessage := '';
end;

procedure TfrmMain.Log1ProcessUpdate(
  Sender: TActivityProcess);
begin
  Sender.GetMessage(FMessage);
end;

procedure TfrmMain.Log1DataUpdateFinish(
  Sender: TActivityProcessLog);
begin
  if FAllowCancel and FForm.Cancelled then
    FLog1Cancel
  else
    if FMessage <> '' then
      FForm.SetMessage(FMessage);
end;
```

Figure 5: Code to set up a few of the log's event handlers.

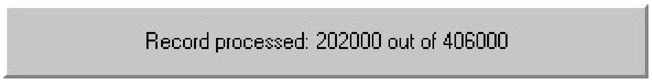

Record processed: 202000 out of 406000

Figure 6: The result of our efforts in Figure 5.

Generating Data

That was pretty good for a few lines of code, but can we get more from it? Sure thing! We now need to test our application somehow. Let's allow the user to generate a file to play with. And rather than displaying an additional entry dialog box for a projected file size, let's simply allow our user to cancel the generation process at any moment. All we would need to do is provide generation code, and bind it to a menu item. The result is shown in Figure 7.

Now, after we start the generating process, the user may hit Esc any time he or she decides the generated file size is big enough. No other

```
procedure TfrmMain.Generate1Click(Sender: TObject);
var
  i: Integer;
begin
  if SaveDiaFLog2Execute then begin
    FLog1Initialize;
    try
      FAllowCancel := True;
      i := GenerateDataFile(SaveDiaFLog2FileName, 1000000,
                            FLog1Client);
    finally
      FLog1Finalize;
    end;
    ShowMessage(Format('Records generated: %d', [i]));
  end;
end;

function GenerateDataFile(const csFileName: string;
  MaxRecCount: Integer;
  LogClient: TCustomActivityLogClient = nil): Integer;
var
  i, rectype: Integer;
  data: TDataRecord;
  datafile: TDataFile;
begin
  Result := 0;
  Assign(datafile, csFileName);
  Rewrite(datafile);
  try
    try
      LogClient.Start(csFileName, cbProgress);
      try
        for i := 0 to Pred(MaxRecCount) do begin
          if i mod 100 = 0 then
            LogClient.Report(i, MaxRecCount, Format(
              rsRecords_II, [i, MaxRecCount]));
          FillChar(data, SizeOf(data), #0);
          rectype := Random(Ord(High(TDataRecordType)));
          data.RecType := TDataRecordType(rectype);
          // You can fill additional fields here.
          Write(datafile, data);
          Inc(Result);
        end;
      finally
        LogClient.Finish;
      end;
    except
      // Nothing serious.
      on e: EActivityProcessLogAbort do ;
      else
        raise;
    end;
  finally
    Close(datafile);
  end;
end;
```

Figure 7: Testing our application.

changes are necessary; our log will automatically create the Display form and release it when it's no longer needed. We can add as many business processes as we need, and each time all we have to do is initialize the log and finalize it when the work is done.

Finding Text/Data

To make our application a little bit more useful, let's provide some simple search capabilities. First, let's add some randomly-generated phone data to our file:

```
case data.RecType of
  drtPhoneFax:
    begin
      data.Area := Format('%.3d', [Random(999)]);
      data.Phone := IntToStr(1000000 + Random(9999999));
    end;
end;
```

```
procedure TfrmMain.Log1Initialize(
  Sender: TActivityProcessLog);
begin
  // Nothing here.
end;

procedure TfrmMain.Log1DataUpdateStart(
  Sender: TActivityProcessLog);
const
  OneSecond = 1.0 / (24 * 60 * 60);
begin
  if (FForm = nil) and
     ((Now - Sender.TimeStart) > OneSecond) then
    FForm := CreateDisplayForm('Working, please wait...');
  FMessage := '';
end;

procedure TfrmMain.Log1DataUpdateFinish(
  Sender: TActivityProcessLog);
begin
  if FForm = nil then
    Exit;
  if FAllowCancel and FForm.Cancelled then
    FLog1.Cancel
  else
    if FMessage <> '' then
      FForm.SetMessage(FMessage);
end;
```

Figure 8: Our modified event handlers.

```
procedure TfrmMain.FormCreate(Sender: TObject);

  procedure SetupProgressBar;
  begin
    ProgressBar1.Hide;
    ProgressBar1.Parent := StatusBar1;
    ProgressBar1.Left := StatusBar1.Panels[0].Width + 3;
    ProgressBar1.Top := 3;
    ProgressBar1.Height := StatusBar1.ClientHeight - 3;
    ProgressBar1.Width :=
      StatusBar1.ClientWidth - ProgressBar1.Left - 20;
  end;

begin
  // Other initialization...
  SetupProgressBar;
end;

procedure TfrmMain.StatusBar1Resize(Sender: TObject);
begin
  ProgressBar1.Width :=
    StatusBar1.ClientWidth - ProgressBar1.Left - 20;
end;
```

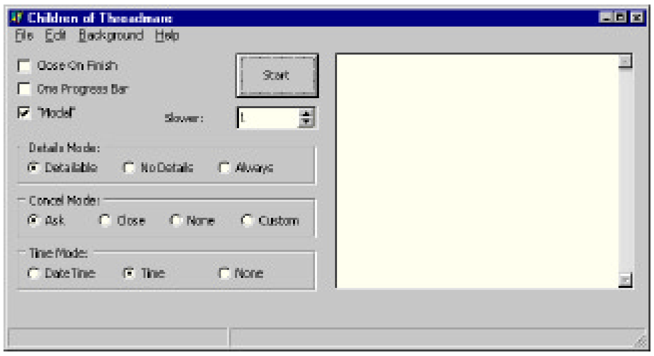
Figure 9: "Dropping" a progress bar onto a status bar at run time.

Dropping and setting up *FindDialog* isn't a problem. The problem is that sometimes we can find what we're looking for in no time at all, and other times we would need to scan the entire file and still not find the match. We certainly don't want to display our progress form if all we have to do is move a couple of records ahead. Otherwise, some visual feedback is definitely welcome.

This may sound like a relatively tricky problem, but it isn't. Let's use the same technique as before, but let's postpone our Display form creation for a second or so. It would only be worthwhile to display the form if we didn't find a match during that time. The modified version of our event handlers is shown in Figure 8. As you can easily see, the Display form is now only created if more than one second passed since the log initialization moment. Also, form presence is checked before its *SetMessage* method is called. Now it's time to add some threads to our application, and change its visual representation.

```
procedure TfrmMain.Log2Initialize(
  Sender: TActivityProcessLog);
begin
  StatusBar1.Panels[1].Bevel := pbNone;
  ProgressBar1.Position := ProgressBar1.Min;
  ProgressBar1.Show;
end;

procedure TfrmMain.Log2Finalize(
  Sender: TActivityProcessLog);
begin
  ProgressBar1.Hide;
  StatusBar1.Panels[1].Bevel := pbLowered;
  StatusBar1.Panels[0].Text := '';
end;

procedure TfrmMain.Log2ProcessUpdate(
  Sender: TActivityProcess);
begin
  with Sender do begin
    if Level = 1 then
      StatusBar1.Panels[0].Text := Format(
        'Steps completed: %d out of %d', [Position, Max])
    if Level = 2 then begin
      ProgressBar1.Max := Max;
      ProgressBar1.Position := Position;
    end;
  end;
end;

procedure TfrmMain.Start1Click(Sender: TObject);
begin
  FLog2.Initialize;
  TTestThread.Create(FLog2Client, 100, 100000, 100, 0);
end;
```

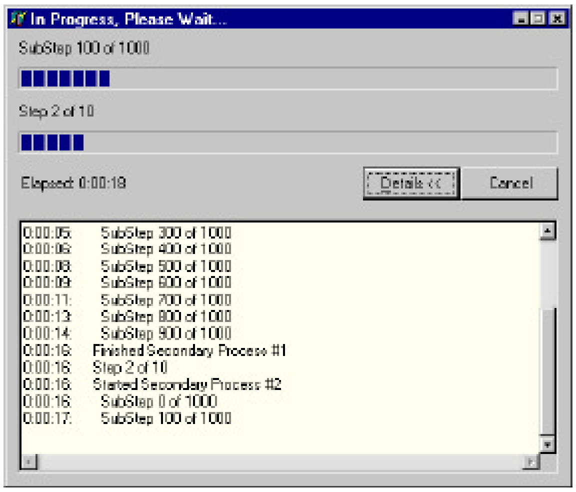
Figure 10: Event handlers for our new *FLog2* component.



☐ Close On Finish
☐ One Progress Bar
☑ "Modal"
Slower: 1

Details Mode: [✓ Detailable] No Details Always
Cancel Mode: [✓ Ask] Close None Custom
Time Mode: DateTime [✓ Time] None

Figure 11: The main form of our application.



```
SubStep 100 of 1000
Step 2 of 10
Elapsed: 0:00:18

0:00:05:    SubStep 300 of 1000
0:00:06:    SubStep 400 of 1000
0:00:08:    SubStep 500 of 1000
0:00:09:    SubStep 600 of 1000
0:00:11:    SubStep 700 of 1000
0:00:13:    SubStep 800 of 1000
0:00:14:    SubStep 900 of 1000
0:00:16:   Finished Secondary Process #1
0:00:16:   Step 2 of 10
0:00:16:   Started Secondary Process #2
0:00:16:    SubStep 0 of 1000
0:00:17:    SubStep 100 of 1000
```

Figure 12: The customizable Progress dialog box at run time.

Background Processing

Let's use our old friend from "Waking from Threadmare," which runs two nested loops in its *Execute* method. Because we're now dealing with multiple threads, let's add one more log component, and this time leave its *AutoFinalize* and *UsePumping* properties intact. We'll also use a progress bar as our progress indicator and a status bar as its container. We would have to add a few lines of code to ensure that the progress bar resides on the status bar. By default, you cannot drop a progress bar onto a status bar, so we're going to achieve it at run time, as shown in Figure 9. The only thing left to do is attach some code to our new *FLog2* component event handlers (see Figure 10).

In this case we don't call *Finalize* manually. Instead, we rely upon log's *AutoFinalize* feature, which guarantees it will be called in due time. Now you can launch the program, select **Background | Start**, and watch it run while you're doing something else.

Progress Dialog Box

Having had so much time invested in our log, it would be completely unwise not to spend a little bit more to create a component that can be used in all applications that require similar feedback from a business process. It's not even difficult at this point. We just need to provide some properties and handle them properly. The main form of our sample application provides access to almost all of these properties (see Figure 11). Hit the **Start** button and you'll get what you want: a fully customizable Progress dialog box in action (see Figure 12).

Let's take a closer look at the properties we can use to make our Progress dialog box suit a wide spectrum of different tasks. The public/published part of its interface is shown in Figure 13. Although some of the features are obvious and self-explanatory, others will definitely benefit from a few extra words. See the sidebar "Progressive Features" on page 31.

Conclusion

We have considered some typical ways to use *TActivityProcessLog*. It turns out to be very useful in conventional single-threaded, and — of course — multi-threaded environments. We also came out with a new multipurpose Progress dialog box component, and some other useful tools and techniques. Happy multi-threading!

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD in INFORM\00\NOV\DI200011NS.

```
TActivityProgressDialog = class(TComponent)
// Private and protected parts are omitted for brevity.
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  procedure Start;
  procedure Journal(const csText: string); overload;
  procedure Journal(strs: TStrings); overload;
  procedure Finish;
  // In case you don't use CloseOnFinish,
  // you may need this one.
  procedure CloseDialog;

  // Run-time properties.
  property Client: TCustomActivityLogClient read FClient;
  // WARNING: Under no circumstances change form/log
  // event handlers! The main purpose of these properties
  // is to provide easy read-only access to their
  // sub-properties without wrapping each one.
  property Form: TfrmActivityProgress read FForm;
  property Log: TActivityProcessLog read FLog;

published
  // These properties should set in inactive mode only.
  property FileName: string
    read GetFileName write SetFileName;
  property FileMode: TActivityDialogFiling
    read FFileMode write SetFileMode;
  property DetailsMode: TActivityDialogDetails
    read FDetailsMode write SetDetailsMode;
  property CancelMode: TActivityDialogCancel
    read FCancelMode write SetCancelMode;
  property TimeMode: TTimeReportMode
    read GetTimeMode write SetTimeMode;
  property UsePumping: Boolean
    read GetUsePumping write SetUsePumping;
  property Modal: Boolean
    read FModal write SetModal default True;
  property OneLevel: Boolean
    read FOneLevel write SetOneLevel;
  // These properties may be set in both active
  // and inactive modes.
  property CloseOnFinish: Boolean
    read FCloseOnFinish write SetCloseOnFinish;
  property Caption: string
    read GetCaption write SetCaption;
  property AutoScrollDetails: Boolean
    read FAutoScrollDetails write FAutoScrollDetails
    default True;
  // Event-handlers.
  property OnStart: TActivityDialogEvent
    read FOnStart write FOnStart;
  property OnFinish: TActivityDialogEvent
    read FOnFinish write FOnFinish;
  property OnClose: TActivityDialogEvent
    read FOnClose write FOnClose;
  property OnCancel: TActivityDialogQueryEvent
    read FOnCancel write FOnCancel;
  property OnTimer: TActivityDialogEvent
    read FOnTimer write FOnTimer;
end;
```

Figure 13: The public/published part of the Progress dialog box.

Nikolai Sklobovsky is a senior system analyst for Retail Technologies International, house of RetailPro (http://www.retailpro.com), one of the world's best POS systems, where he developed a sophisticated, yet easy-to-use, DSS (Decision Support System) for OLAP analysis of merchant data. He has over 10 years of experience in applied mathematics and teaching at the university level, as well as over 10 years of experience in IT. You can contact Nik at delphi@sklobovsky.com or at his Web site at http://www.sklobovsky.com.

Progressive Features

However important the *OnStart*, *OnFinish*, *OnClose*, *OnCancel*, and *OnTimer* event handlers are, you'll find that it's possible to keep them all unassigned and still have a very useful progress dialog box component. While their use is highly application-specific, it's altogether possible to provide some rules of thumb for each.

— *Nikolai Sklobovsky*

Feature	Explanation
procedure *Start*	One of the more important methods. Always call it when you want the dialog box to be shown. It initializes the internal log component and displays the dialog box.
procedure *Finish*	The opposite of its companion *Start*, you may never call this method. All it does is call the internal log's *Finalize* method. As long as the dialog box keeps *AutoFinalize* equal to True (unless you change it to False for some better reason), the need for *Finalize*, and, *Finish* is negligible. This method exists mostly for symmetry/aesthetic purposes.
procedure *Journal*	These two overloaded methods allow you to "log" a bunch of lines of text to the dialog box's journal in a single call. They're only wrappers to the relevant internal log's methods.
procedure *CloseDialog*	As it says in the inline comment, you would only need to call this manually when you have the *CloseOnFinish* property set to False.
property *Client*	This important property is a shortcut to the dialog box's internal log's client. This gives you an entry point to all of the goodies we learned to use here.
property *Form* property *Log*	Although these properties basically provide full access to their targets, it's important that you never modify these objects' event handlers, or other important properties. Rule of thumb: It's safe to use them in read-only mode, and you should think thrice before changing anything you're not sure of.
property *FileName* property *FileMode*	These two properties serve as safe wrappers to the log's filing properties and help you deal with the log file.
property *DetailsMode* (*addUser*, *addNone*, or *addAlways*)	This property allows you to specify the desired details level. The first (default) option allows the end user to show or hide the journal with the help of the **Details** button. The other two hide this controlling button and freeze the dialog box in either detail-less or detailed mode.
property *CancelMode* (*adcAsk*, *adcDontAsk*, *adcNone*, or *adcCustom*)	Similar to *DetailsMode*, this property provides you with a one-touch way of configuring the dialog box's behavior, versus a possible canceling action. The first two options are pretty obvious. The third one eliminates the possibility of closing this dialog box for the end-user. The fourth option provides programmers with ultimate control, but in this case, you would have to handle the *OnCancel* event yourself.
property *TimeMode* property *UsePumping*	These two properties should be familiar to the log's user. They are, in essence, write-safe wrappers for the internal log's similar properties.
property *Modal*	This important property provides you with modal-like dialog box behavior (Task List routines are used again).
property *OneLevel*	By default, the dialog box would display two progress bars. If you have or need only one level of nesting, you may want to set this property to True.
property *CloseOnFinish*	This property controls auto closing. If you don't care about letting the end user study the journal details before closing the dialog box, simply go ahead and set this property to True. You'll save your user one click.
property *Caption*	This is basically a *Form.Caption*. It's just a little bit smarter, and displays some default text, even if you set it to an empty string.
property *AutoScrollDetails*	Whether you want the dialog box's journal to automatically scroll down on acquiring new data, or you want it to stand still, this property gives you a fair chance to create the proper effect. If you study the code, you'll see that its scrolling part is smart enough to not scroll if a user decided to investigate some part of the journal. If it detects any user activity, it suspends scrolling for 10 seconds and resumes it only if the user didn't do anything to the journal during this time.
property *OnStart*	Fired after the dialog box is shown.
property *OnFinish*	Fired when the dialog box's log has been finalized and its internal timer has been stopped. It means no more new data is expected as long as all of the business processes cease to run.
property *OnClose*	Similar to *OnFinish*, but it's fired a little bit later, namely when a dialog box has been closed. If you have *CloseOnFinish* set to True, you would probably use only one of these two events.
property *OnCancel*	May only be fired if you set the *CancelMode* property to *adcCustom*. In this case, you'll be given a chance to consult with the other parts of your application or with your end user, and then modify the *Continue* parameter as you decide.
property *OnTimer*	Provides you with a decent way to do something else. Normally, you would leave this event handler unassigned, but in case you do some system tray programming, it may come in handy.

Figure A: Specific features of the Progress dialog box.

Begin Listing One — *TDisplayForm*

```
constructor TDisplayForm.Create;
begin
  inherited Create(Application);
  Position := poScreenCenter;
  BorderStyle := bsNone;
  FPanel := TPanel.Create(Self);
  FPanel.Parent := Self;
  FPanel.BevelWidth := 2;
  FPanel.Align := alClient;
  FDoNotDecreaseAutoWidth := True;
  KeyPreview := True;
  OnKeyPress := FormKeyPress;
end;

procedure TDisplayForm.SetMessage(const csMessage: string;
  bAutoSize: Boolean);
const
  csDelta = 'WWWWWWWWWW';
var
  w, wDelta: Integer;
begin
  FPanel.Caption := csMessage;
  if bAutoSize then begin
    w := Canvas.TextWidth(csMessage);
    wDelta := Canvas.TextWidth(csDelta);
    w := ((w + wDelta + pred(wDelta)) div wDelta) * wDelta;
    if (w > Width) or not FDoNotDecreaseAutoWidth then
      SetBounds(0 + ((Screen.Width - w) div 2),
                0 + ((Screen.Height - Height) div 2),
                w, Height);
  end;
  if FPumpOnUpdate then
    Application.ProcessMessages
  else
    Repaint;
end;

procedure TDisplayForm.DoShow;
begin
  inherited DoShow;
  FTaskList := DisableTaskWindows(Handle);
  FCursor := Screen.Cursor;
  Screen.Cursor := crHourGlass;
end;

procedure TDisplayForm.DoHide;
begin
  EnableTaskWindows(FTaskList);  // Works with nil okay.
  Screen.Cursor := FCursor;
  inherited DoHide;
end;

procedure TDisplayForm.FormKeyPress(Sender: TObject;
  var Key: Char);
begin
  if Key = #27 then  // User hit [Esc] key.
    Cancelled := True;
end;

function CreateDisplayForm(const csMessage: string;
  iWidth, iHeight: Integer): TDisplayForm;
begin
  Result := TDisplayForm.Create;
  Result.Height := iHeight;
  if iWidth > 0 then
    Result.Width := iWidth;
  Result.SetMessage(csMessage, iWidth <= 0);
  Result.Show;
  Result.Update;
end;
```

End Listing One

Begin Listing Two — *TDataRecordType*

```
  TDataRecordType = (drtName, drtAddr1, drtAddr2,
                     drtCityStateZip, drtPhoneFax);
  TDataRecordTypeArray = array of TDataRecordType;

  TDataRecord = record
  case RecType: TDataRecordType of
    drtName:
      (First, Last: string[16]);
    drtAddr1:
      (Street: string[30]; Apt: Integer);
    drtAddr2:
      (Addr2: string[30]);
    drtCityStateZip:
      (City: string[20]; State: string[2]; ZIP: string[5]);
    drtPhoneFax:
      (Area: string[3]; Phone: string[7];
       Ext: string[4]; Fax: string[7]);
  end;
  PTDataRecord = ^TDataRecord;

  TDataFile = file of TDataRecord;
  private  // Part of the form.
    FForm: TDisplayForm;
    FMessage: string;
    FTypes: TDataRecordTypeArray;
    FAllowCancel: Boolean;
    procedure OpenDataFile(const csFileName: string);

resourcestring
  rsRecords_II = 'Record processed: %d out of %d';

procedure TfrmMain.Open1Click(Sender: TObject);
begin
  if OpenDiaFLog2Execute then
    OpenDataFile(OpenDiaFLog2FileName)
end;

procedure TfrmMain.OpenDataFile(const csFileName: string);
begin
  FLog1Initialize;
  try
    FAllowCancel := False;
    FTypes := ReadDataTypes(csFileName, FLog1Client);
  finally
    FLog1Finalize;
  end;
end;

function ReadDataTypes(const csFileName: string;
  LogClient: TCustomActivityLogClient = nil):
  TDataRecordTypeArray;
var
  i, iCount: Integer;
  data: TDataRecord;
  datafile: TDataFile;
  x: TDataRecordTypeArray;
begin
  Assign(datafile, csFileName);
  Reset(datafile);
  iCount := FileSize(datafile);
  LogClient.Start(csFileName, cbProgress);
  SetLength(x, iCount);
  try
    for i := 0 to pred(iCount) do begin
      if i mod 100 = 0 then
        LogClient.Report(i, iCount,
          Format(rsRecords_II, [i, iCount]));
      Read(datafile, data);
      x[i] := data.RecType;
    end;
    Close(datafile);
    Result := x;
  finally
    LogClient.Finish;
  end;
end;
```

End Listing Two

By Bill Todd

VMware 2.0

Multiplatform Testing on a Single Machine

Would you like to develop and test your applications using Delphi for Windows and Delphi for Linux at the same time on the same computer? With VMware 2.0 from VMware, Inc., it's easy. VMware takes advantage of the virtual machine architecture of Intel Pentium II (and higher) chips to let you run multiple operating systems simultaneously.

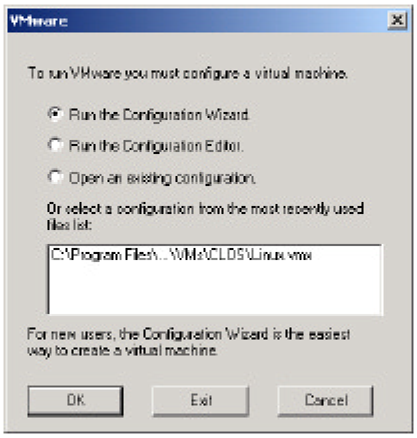
You can install VMware on a computer running NT 4 Server or Workstation, or Windows 2000 Professional, Server, or Advanced Server. You can also install VMware on a machine running Linux. Once VMware is installed, you can create as many virtual machines as you wish, and install a different operating system on each machine.

Creating a Virtual Machine

Upon starting VMware, the dialog box in Figure 1 appears. This dialog box allows you to choose to create a new virtual machine using the Configuration Wizard, open the Configuration Editor, open an existing configuration, or choose a configuration from the most-recently-used list. The best way to create a virtual machine is with the wizard.

When you create a new virtual machine with the wizard, the first step is to choose the operating system for the new virtual machine from the list shown in Figure 2. Next, the wizard asks for the path to the directory that will host the new virtual machine. You can place this directory on a local hard drive, a removable hard disk, or a network file server.

The third step requires you to specify how to create a disk drive for the new virtual machine. Although you can use an unused disk partition, the best choice is to create a virtual disk drive. A virtual drive is a file in the directory you specified in the preceding step. You can specify a maximum size for the virtual disk of up to 2GB. It's best to use the 2GB maximum because you can't change the maximum size after the disk has been created. Although the virtual disk appears to be its maximum size to the virtual machine's operating system, the file that hosts the virtual disk starts small and grows as files are added to the virtual disk. If you later delete files from the virtual disk, you can shrink it to recover the space on the host operating system's hard drive. Although the maximum size of a virtual disk is limited, you can add more disks any time after creating the virtual machine.


VMware

To run VMware you must configure a virtual machine.

(●) Run the Configuration Wizard.
( ) Run the Configuration Editor.
( ) Open an existing configuration.

Or select a configuration from the most recently used files list:

C:\Program Files\...\VMs\CLDS\Linux.vmx

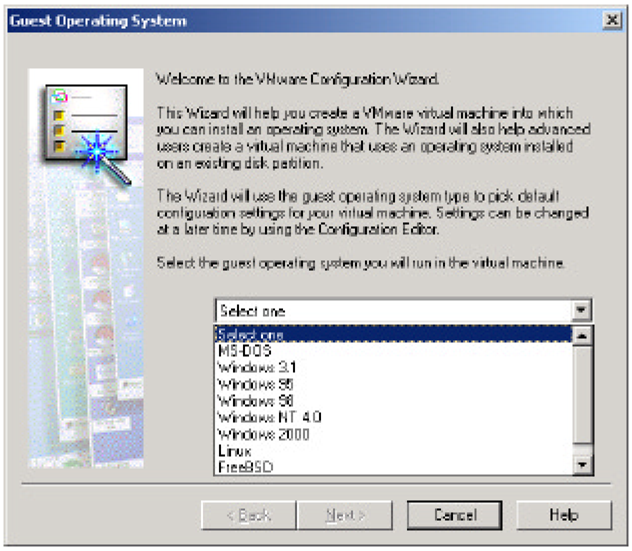
For new users, the Configuration Wizard is the easiest way to create a virtual machine.

Figure 1: The VMware startup dialog box.

The wizard continues by asking if you want the virtual machine to start with the computer's floppy drive and CD-ROM connected to the virtual machine, and what kind of networking you want to use, if any. You can choose host-only networking if your computer is not on a network and you want the guest and host operating systems to be networked. If you're on a LAN, choose bridged networking to make both the host and guest operating systems available on the network.

After you've clicked the wizard's **Finish** button, you'll find yourself in the virtual machine's window, as shown in Figure 3. At this point, it's a good idea to open the Configuration Editor, as shown in Figure 4, and make any necessary changes to the virtual machine before you install the operating system. You'll probably want to increase the amount of memory because VMware's defaults are quite conservative. When I created a virtual machine for Linux, the default memory size was 48KB, when in fact 64KB is a more realistic number to get reasonable performance. You may need more, depending on the applications you will run. While the VMware literature says that the minimum memory you must have installed on your PC is 96MB, that is likely not enough. For example, if you're going to run Windows 2000 Professional and Linux, you should plan on at least 96MB for Windows and 64MB for Linux, so a system with 192MB or more would be a good choice.

Also, the wizard configures only those devices you need to install and boot an operating system. While the wizard defaults to automatically connect your CD-ROM and floppy drives to the virtual machine, it does not automatically connect your parallel port, serial ports, or sound card. If you want these devices connected when your virtual machine starts, you'll need to change their settings in the Configuration Editor.



Guest Operating System

Welcome to the VMware Configuration Wizard.

This Wizard will help you create a VMware virtual machine into which you can install an operating system. The Wizard will also help advanced users create a virtual machine that uses an operating system installed on an existing disk partition.

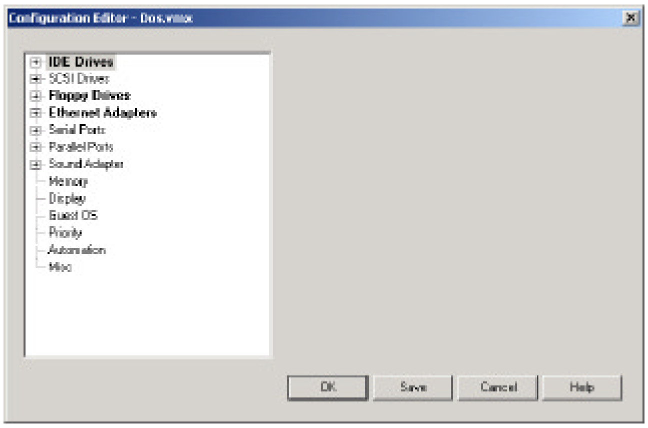
The Wizard will use the guest operating system type to pick default configuration settings for your virtual machine. Settings can be changed at a later time by using the Configuration Editor.

Select the guest operating system you will run in the virtual machine.

Select one
Select one
MS-DOS
Windows 3.1
Windows 95
Windows 98
Windows NT 4.0
Windows 2000
Linux
FreeBSD

Figure 2: Choosing an operating system for the new virtual machine.

Figure 3: After completing the wizard, you end up in the virtual machine's window.



Configuration Editor - Dos.vmx

IDE Drives
SCSI Drives
Floppy Drives
Ethernet Adapters
Serial Ports
Parallel Ports
Sound Adapter
Memory
Display
Guest OS
Priority
Automation
Misc

Figure 4: The Configuration Editor.

Installing the Operating System

Before you install the operating system on your virtual machine, you may want to take a trip to VMware's Web site. In my first attempt to install Corel Linux, I had some problems. After visiting the Web site and searching for "Corel," I immediately found step-by-step instructions for installing Corel Linux. With those in hand, my second attempt went smoothly.

Installing the operating system on a virtual machine is no different than installing it on a physical computer: Put the floppy disk or CD-ROM in the drive, and click the **Power On** button on the VMware toolbar to start the virtual machine's boot process. Although the computer I tested on was not set up to boot from its CD-ROM drive, I was pleased to discover that the virtual machine would boot from the CD-ROM so I didn't need a boot disk to install Linux.

Running a Virtual Machine

You can run a virtual machine in a window or switch to full screen mode. Because I was running the virtual machine at the maximum resolution of my monitor, I found full screen mode much easier to work in. Running in a window meant that part of my Linux desktop was not visible, and I found it annoying to use the scroll bars to bring the hidden part into view. You can switch from full screen to window mode quickly by pressing Ctrl Alt Esc. Figure 5 shows the Corel Linux desktop running in a window.

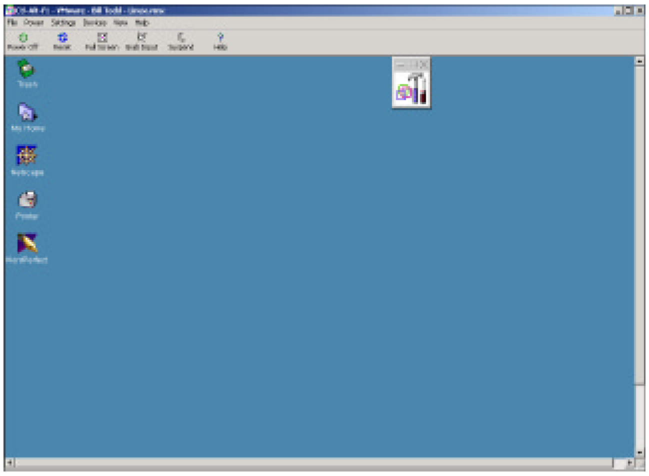


Figure 5: The Corel Linux desktop running in a window.

Informant Fact File

If you develop, test, or provide support for multiple operating systems, this is the ideal way to do it. VMware 2.0 is easy to install, configure, and use. It simply works.

VMware, Inc.
3145 Porter Drive, Bldg. F
Palo Alto, CA 94304

Phone: (650) 475-5000
Web Site: http://www.vmware.com
Price: VMware 2.0 for Windows NT and Windows 2000, or VMware for Linux, US$299 (prices are for electronic distribution); US$329 for packaged distribution (includes SuSE Linux 6.3 and TurboLinux 6.0).

Because the guest and host operating systems run simultaneously, VMware lets you change the priority of the guest operating system any time you wish. You may also choose to give the guest operating system a higher priority when it has focus, and a lower one when it does not. Another great feature is the ability to suspend a virtual machine to disk or memory at any time. You can suspend to disk in the middle of a compile, shut down VMware, turn off the computer, and later restart your virtual machine exactly where you left it.

VMware also gives you impressive control over what happens on the virtual machine's disks. You can choose non-persistent mode, where changes to files are discarded at the end of the session; undoable mode, where you can choose to save or discard any changes at the end of the session; or persistent mode, where the disk behaves normally and all changes are permanent.

Conclusion

VMware is cheaper than multiple computers and more convenient and flexible than dual boot. If you have to develop, test, or provide support under multiple operating systems, this is the ideal way to do it. It's easy to install, configure, and use. And once you have it set up, it simply works. ∆

Bill Todd is president of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is co-author of four database programming books, author of more than 60 articles, a Contributing Editor to *Delphi Informant Magazine*, and a member of Team Borland, providing technical support on the Borland Internet newsgroups. He is a frequent speaker at Borland Developer Conferences in the US and Europe. Bill is also a nationally known trainer and has taught Delphi programming classes across the country and overseas. He is currently a speaker on the Delphi Development Seminars Kylix World Tour. Bill can be reached at bill@dbginc.com. For more information on the Kylix World Tour, visit http://www.DelphiDevelopmentSeminars.com.

table of CONTENTS

By Warren Rachele

UIL Security System 2.0

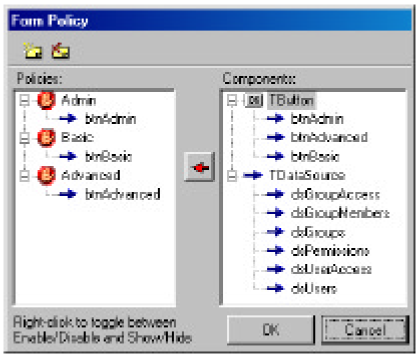
Delphi Components Provide Readymade Security

System security is ubiquitous in the current era of networked computing. Most access control is governed through the user's login to either the local area network or inter-networked servers. For many applications, this top-level security is more than sufficient, but occasionally, you'll need component-level security within an application.

For example, users with differing security levels within an application may be able to open existing files, but not have the ability to create new ones. Higher levels may be able to create files, but not delete them. To implement this variety of access control, the application must maintain a database of user logins and their associated component permissions.

The UIL Security System, from Unlimited Intelligence Ltd., adds easy-to-use and powerful end-user security to your Delphi programs. Integrating the UIL components into a Delphi program lets you control access to components on the basis of a user login. While programming these features isn't difficult, it's time-consuming. When you compare hand-programming the security you need with the elegance of dropping in the UIL components and setting a few properties, this system will win every time.

UIL Security System's four components integrate with the Delphi code base. The core component is *TuilSecurityManager*, a required control through which the other controls connect to the security database. The security manager tracks who is logged in and the access permissions assigned to them. The six tables that provide the data-management element of the security system are also bound to the program through this component and their associated data source controls.



Right-click to toggle between Enable/Disable and Show/Hide

Figure 1: The Form Policy dialog box.

The *TuilFormPolicy* component maintains all of the permission policies a form contains, and the action to be taken on a call to the policy. The action is determined by the association of components with a named policy. The user gains access to the components by having the appropriate permission policy assigned to his/her login. *TuilFormPolicy* is connected directly to a security manager component, and most of its functionality is used at design time.

The other pair of components are visual controls that provide a user interface into the security functions. *TuilLoginDlg* is a dialog box that hooks directly to the security manager. It provides a simple method for collecting the user name and password, and feeds them into the UIL system. *TuilSecurityDlg* is used to simplify access to the security settings (policies) for the end user at run time.

Installation

Installing the UIL Security System is strictly a manual affair. The components and supporting tables are provided in the form of a zip archive file, which is unzipped into the folder that will hold the units, etc. This may be a product-specific folder or a Delphi folder defined in the library path. Before adding the components to the VCL, you must compile a version-specific run-time package. In Delphi 5, open and compile uSec2050.dpk. (The product provides files for Delphi 3, 4, and 5.) Once this step has been completed, the design-time package is installed, adding the components to the VCL on a new "UIL" tab.

While it's not mentioned in the documentation, you must ensure that the units for the components can be found on Delphi's library path. If not, your projects will fail at compilation time. If you installed the files in a folder outside of the main Delphi folders, be sure to add the folder to the library path, found under the **Tools | Environment Options** menu of the Delphi IDE.

Using UIL Security System

Putting the pieces together within a Delphi project is a simple matter once you understand the system. The process is started by dropping a *TuilSecurityManager* component onto a form. Notice in the Property Inspector that the properties are in three "bindaries." These will reference the tables added in the next step.

When using the BDE by default, along with the native Data Access components, all the security data is maintained in a set of Paradox tables, although the

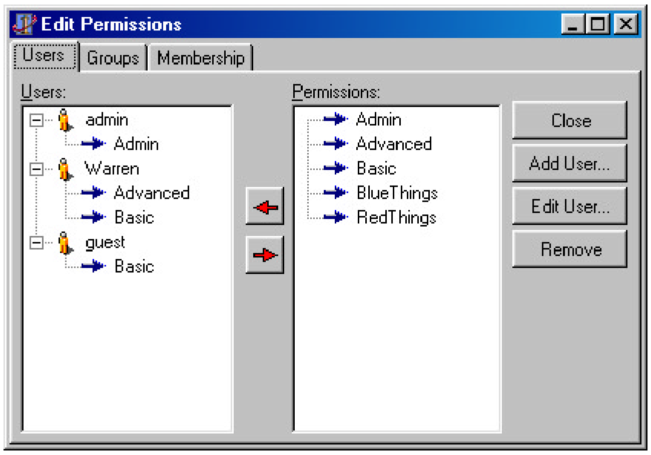


Figure 2: The Edit Permissions dialog box.

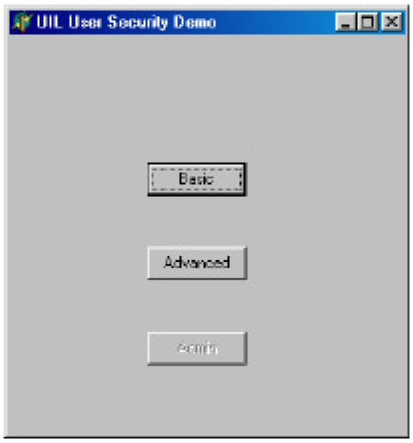


Figure 3: Logging in as Warren.

UIL Security System can use tables managed by other database systems supported by Delphi. The Paradox relations are supplied with the product, but the structures aren't documented. Be sure to distribute a clean set of tables with your project, so make a backup before you experiment.

Next, add six Table and six DataSource components to the form. The database alias for each of the Table components will be the folder in which you installed the UIL database tables. Each of the Table components will be associated with a table in the database. Each of the DataSource components will be associated with a Table. Add these table associations to the Bindary properties of *TuilSecurityManager*.

A step that isn't mentioned in the documentation is setting the master-detail relationships between the tables. This is critical to the proper operation of the system. To demonstrate the use of the product, I lined up three buttons on the form, captioning them **Basic**, **Advanced**, and **Admin**. Finally, a *TuilFormPolicy* component was added to the form.

This last component is used to set the security policies you want to apply to users. Figure 1 shows the Form Policy dialog box, through which you'll create the policies and then group components within them. This is accessible by double-clicking on the *TuilFormPolicy* component. I've added three policies that will encompass all of these controls, and give users various degrees of access based on their experience. For example, after establishing the Basic policy, I've added *btnBasic* to it. UIL Security will correlate this policy to the user login, and allow access to only those components allowed under the policy.

To enable login verification, I've added *TuilLoginDlg* to the project and tied it to the *OnCreate* method of the form. This will verify user login before access is allowed. Also added is the last component, *TuilSecurityDlg*. When this component is executed, the dialog box shown in Figure 2 is presented. Here the user's logins are associated with the policies created earlier. This dialog box can also be accessed at design time by double-clicking on the *TuilSecurityManager* component.

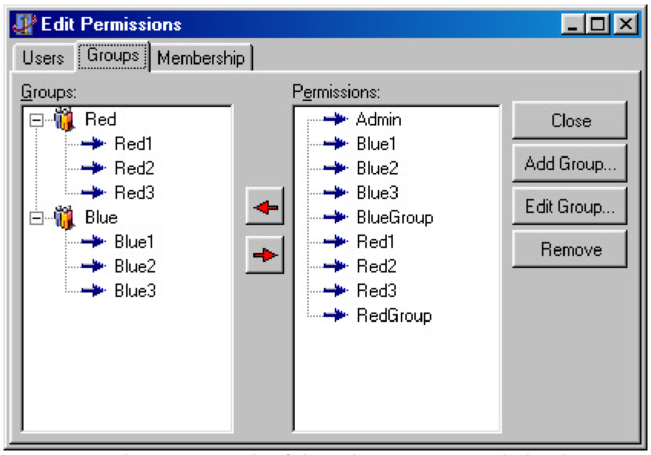


Figure 4: The Groups tab of the Edit Permissions dialog box.

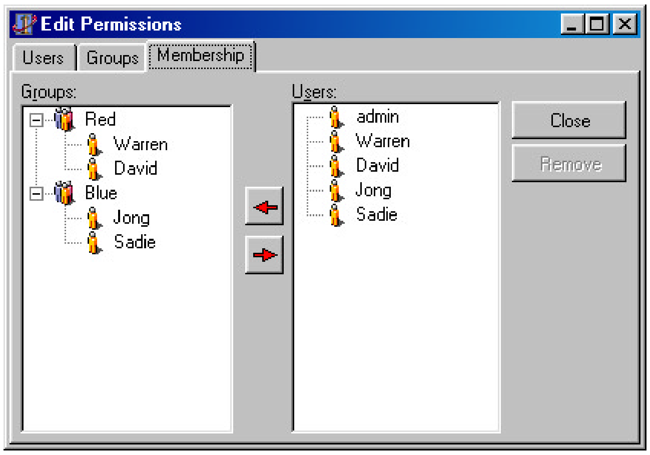


Figure 5: The Membership tab of the Edit Permissions dialog box.

Figure 3 shows the results of logging in as Warren. Warren has Basic and Advanced access, which lets him use the *btnBasic* and *btnAdvanced* components. The design for assignment of the permissions has a subtle twist. In reviewing the samples and documentation, it appeared that the controls to which you would want to grant access through a permission should be grouped under that permission; for example, if you wanted to grant Warren access to both the **Basic** and **Advanced** buttons. My inclination was to group these controls under the Advanced permission. But this is not how the product is designed, and it would lead to inconsistent results. The proper way to grant these permissions is to provide Warren with both Basic and Advanced permission, which grants access to the appropriate controls. The author of the product was very patient and helpful in making this clear to me.

Groups

Using groups to manage users is the only way to go. Rather than laboriously adding individual permissions to single users as they are added to the database, UIL Security System lets you assign permissions to a named group, and then add or remove users to and from those groups. Figure 4 shows the Edit Permissions dialog box with the Groups tab selected. Two groups have been created — Red and Blue — and access to the appropriate buttons has been assigned to the group. The "Red" buttons have all been assigned to the Red group. This gives all members of the Red group access to the group's permissions.

The Membership tab of the same dialog box is shown in Figure 5. You add users to the groups through this dialog box. The Red group now includes users Warren and David. Using the group features does not

Informant Fact File

UIL Security System 2.0 is a niche component set that provides a convenient way of implementing component-level security in Delphi programs. Although poorly documented, the convenience of this well-designed drop-in security solution will allow you to focus on other aspects of your projects.

Unlimited Intelligence Ltd.
#9-552 Church Street
Toronto, ON, M4Y 2E3
Canada

E-Mail: info@uil.net
Web Site: http://www.uil.net
Price: US$199 for product and source.

preclude you from using the individual user access control to refine your security. In this example, the Admin user is assigned sole access to the *btnAdmin* component.

Documentation

Documentation is the weak link. There's a README.TXT file, which provides installation and product notes, and a standard Windows Help file. Otherwise, there is no documentation. Within the Help file, there is a brief description of each of the components and the properties and methods of each, but few examples. The Help file provides a short tutorial, but many key requirements are missing. The product's developer is helpful in resolving most issues through his Web site. He also offers usage instructions when the design isn't clear. This will work on a limited scale of distribution, but it will cause problems if the product becomes popular. The developer should focus on this before adding features.

Conclusion

UIL Security System 2.0 is a niche product that provides a convenient way of implementing component-level security in your Delphi programs. Its four components are well constructed and designed, and work without problem once all of the appropriate settings are discovered. The convenience of this drop-in solution allows you to focus on other aspects of your project. ∆

Warren Rachele is Chief Architect of The Hunter Group, an Evergreen, CO software-development company specializing in database-management software. The company has served its customers since 1987. Warren also teaches programming, hardware architecture, and database management at the college level. He can be reached by e-mail at wrachele@earthlink.net.

Code Analyst

There are many tools available to analyze your code for potential memory leaks, etc. But what about analyzing the writer of the code? This may be a bit too complex to automate, but it can be done "manually" as I'm about to prove. Your code — what you write and how you write it — reveals much about your inner self. It's an indicator of habits and personality traits, at least as much so as handwriting.

Let's analyze some code samples I've picked up over the years (never mind what this says about me). Where necessary, user-defined class names and file names have been altered to protect the disturbed.

Our first example:

```
procedure TkeuDBMemo.DoEnter;
begin
 inherited DoEnter;
 enterCount := Lines.Count;
 enterStringList := TStringList.Create;
 try
  if enterCount > 0 then
   enterStringList.AddStrings (Lines);
 except
  enterStringList.Free;
  enterStringList := nil;
  raise;
 end;
end;
```

This programmer is self-conscious, introverted, even sneaky. Look at the indention. One measly space! He (although it is reportedly not true that all wackos are male, for sake of convenience I will refer to the patients ... er ... coders as male) wants to attract as little attention as possible. He probably speaks in a barely audible voice, walks with mincing steps, and is startled by rodents. And what happens to the StringList if there's no exception? It isn't freed! He must be into bondage as well. This person is high strung, lives alone (and likes it), and drives a Ford Pinto.

On to our next victim ... I mean subject:

```
function ReadBtjIniFile(
  Section, Identifier : String) : String;
var
  IniResult,pSect,pIdent : PChar;
begin
  IniResult := StrAlloc(256);
  pSect := StrAlloc(Length(Section) + 1);
  pIdent := StrAlloc(Length(Identifier) + 1);
  StrPCopy(pSect,Section);
  StrPCopy(pIdent,Identifier);
  GetPrivateProfileString(
    pSect,pIdent,'',IniResult,255,'.\BTJSOFT.INI');
  if (StrLen(IniResult) = 0) then
    GetPrivateProfileString(
      pSect,pIdent,'',IniResult,255,'..\BTJSOFT.INI');
  Result := StrPas(IniResult);
  StrDispose(pIdent);
  StrDispose(pSect);
  StrDispose(IniResult);
end;
```

This is sad — a clear indication of a troubled psyche. The man is resting on his laurels. He's not completely lacking in programming knowledge, but has not bothered to learn Delphi's implementation of Object Pascal. Instead of using the *TRegIniFile* class, PChars are used with dynamic memory allocation. Worse — and similar to the first example — are the potential resource leaks. Look at all those StrAllocs without **try**..**finally** blocks. This coder is haughty, "I'll do it my way. I don't need no stinking *TRegIniFile*"; lazy, "This worked for me in the 70s. Why should I change now?"; and reckless, "What are the chances of a problem between memory allocation and deallocation?" This person is grossly overweight, sports an unkempt beard, eats pizza and Oreos exclusively, and drives a Chevy Citation.

Here's another:

```
...
end;
function TFomrMaintainanceRunWizard.DoImportSQL(
  Page: TWizardPage; Step: TScriptStep): Integer;

var
SourceFields: TStringList;
DestFields: TStringList;
SourceMasterFields: TStringList;
DestMasterFields : TStringList;
Query: TQuery;

function ExecuteUpdateQuery : Boolean;
var i : Integer;
begin
{ code removed }
end;

var DestinationQuery: TQuery;
var DataBase: TDataBase;
var i: Integer;
var ResultLabel: TLMDLabel;
var RecordsRead: Integer;

begin
...
```

This person should be institutionalized. He leaves no space between methods, yet adds a space between the function header and variable declarations. He sandwiches a nested function between the local variable declarations, and fails to indent the nested function. Not only that, the **var** keyword is explicitly stated for every variable declared. This is not only unnecessary, because these declarations resemble reference parameters passed to the function, but they could cause confusion for programmers forced to maintain this stuff. To beat a dead horse, he's also careless (notice the typo of "form" in the class name) and can't spell, e.g. "Maintainance." You would hate to see his bedroom: Socks are draped over the bedpost and underwear hangs from the lamp. He bangs Enter ↵ like a madman when his program hangs, and drives a "kit car" with a lawnmower engine.

We'll end on a positive note:

```
class procedure TAboutForm.Cre8Yourself;
begin
  with TAboutForm.Create(nil) do
    try
      ShowModal;
    finally
      Free;
    end;
end;
```

This guy knows his stuff; the code is tight and elegant. He uses a class procedure and a **with** statement — during the creation of the form, to boot. On the other hand, naming the procedure *Cre8Yourself* is a little on the cute side. Nevertheless, this person is well adjusted, exemplary in every way, humble, has a good sense of humor, and drives a Plymouth PT Cruiser.

The doctor is in. Fork over a code sample and lie back on the couch. Δ

— *Clay Shannon*

Clay Shannon is an independent Delphi consultant based in northern Idaho. He is available for Delphi consulting work in the greater Spokane/Coeur d'Alene areas, remote development (no job too small!), and short-term or part-time assignments in other locales. Clay is a certified Delphi 5 developer, and is the author of *Developer's Guide to Delphi Troubleshooting* [Wordware, 1999]. You can reach him at BClayShannon@aol.com.

Delphi Book Wrap-up 2000

Perhaps you expected "Delphi 5 Book Wrap-up?" This year, however, we've witnessed a continuing trend in Delphi titles: Most have abandoned the notion of identifying themselves with a particular version of Delphi. The exception, of course, is the latest editions of the two perennial favorites I reviewed earlier this year. I will begin this column by taking another look at those classics. I will then discuss five new works of great merit that cover the spectrum of Delphi topics and levels. Finally, I will suggest an "Essential Delphi Library of Advanced Works" with books that go back as far as Delphi 2.

The Classics. I'm sure that most readers can guess the titles of the two Delphi classics. If not, let me give you a couple of hints: Which two Delphi books have had editions for almost every new version of Delphi? Which two books consistently top the annual reader's poll? Of course, I'm talking about *Mastering Delphi 5* by Marco Cantù [SYBEX, ISBN: 0-7821-2565-4] and *Delphi 5 Developer's Guide* by Steve Teixeira and Xavier Pacheco [SAMS, ISBN: 0-672-31781-8]. Because I reviewed both of these outstanding works this year, I will limit my remarks here to a brief summary of each, and a comparison.

Of the two, I feel that Cantù's work is better suited for the less-experienced developer. The author carefully explains many basic principles and techniques. Beginning with a tour of Delphi's IDE, providing an introduction to Object Pascal, and exposing the VCL, he provides a solid foundation for working in Delphi. He goes on to expose more advanced topics, including a wonderful exposition of one of my favorites — dynamic-link libraries.

Steve Teixeira and Xavier Pacheco take a similar approach, beginning with an introduction to basic issues like the Object Pascal language. The authors go beyond Delphi fundamentals, however, with a chapter on "Application Frameworks and Design Concepts" — an excellent introduction to good programming practice. Both books cover all essential topics, including working with components and database programming. *Developer's Guide* emphasizes the latter topic, devoting the final third of the book to database and client/server programming. Both works also cover a fair amount of advanced topics, including component writing and technologies like ActiveX.

The Class of 1999-2000. There's no author whose works are more eagerly anticipated than Ray Lischner's, so it's no wonder I've included his two previous works in the "Essential Delphi Library." His latest, *Delphi In a Nutshell* [O'Reilly, ISBN: 1-56592-659-5], is no less than an in-depth exploration of the very heart of Delphi — Object Pascal — with all of the powerful extensions that make Delphi the Cadillac of programming languages. At its heart is a lengthy chapter that includes every type, variable, function, procedure, etc. that is part of Delphi's underlying language and system units. I predict this book will become an essential reference for every Delphi developer who adds it to his or her library.

The Tomes of Delphi: Win32 Database Developer's Guide by Warren Rachele [Wordware, ISBN: 1-55622-663-2] is one of the few Delphi books devoted exclusively to this essential discipline. What sets this work apart from other works that deal with database topics is its in-depth discussion of the Borland Database Engine (BDE). In an early chapter, Rachele outlines the basic architecture of the BDE, and later provides an overview of the functions. The author also covers the essential topics of SQL, data-access and data-aware components, and database tasks, including preparing reports and printing.

At the other end of the spectrum of specialized Delphi books is John Ayres' new work, *Delphi Graphics and Game Programming Exposed! with DirectX* [Wordware, ISBN: 1-55622-637-3]. Appropriately, Ayres begins with an introduction to game programming in general, followed by an exposition of techniques. After that, the real fun begins. Using Erik Unger's Project JEDI DirectX header conversion as a foundation, he treats the reader to a first-rate explanation on using this vital technology in game programming.

The next book was something of an afterthought for me. I wanted to examine all of the new Delphi books, and having seen (but not read) a review of it in *Delphi Informant Magazine*, I requested a copy. I expected to spend a couple of hours skimming it to assess its style and content. What I didn't expect was spending an entire evening and a good portion of the next morning completely captivated by Eric Harmon's *Delphi COM Programming* [Macmillan Technical Publishing, ISBN: 1-57870-221-6]. Harmon has a wonderful writing style, using just the right amount of repetition to expose this fundamental technology. He carefully explains the similarities and subtle differences between interfaces and objects, shows how to build and implement interfaces, and provides an example of multiple interfaces within a single class — all within the first 50 pages! Having established a solid understanding of interfaces, he goes on to discuss different kinds of COM servers, creating ActiveX controls, and other related technologies, with a plethora of excellent code examples and wonderful tips.

The final book in the new crop is a bit unusual for me. It's a book for beginners. Maybe I'm an elitist, but I'll certainly admit to an enjoyment of going off the beaten path. Nevertheless, I like *Discover Delphi* [Addison-Wesley, ISBN: 0-201-34286-3] by

Shirley Williams and Sue Walmsley. In fact, if I ever teach a course for beginning programmers with Delphi, this will be my text. It assumes very little background besides a familiarity with basic computing. It presents all of the basic topics cogently, with sufficient detail and code examples to meet the needs of novice programmers.

Oldies but Goodies: An Essential Delphi Library of Advanced Works. Many of the books I'll discuss in this section are out-of-print and difficult to find. Nevertheless, I would recommend adding any or all of them to your library. If you write experts, or work in other ways with Delphi internals (e.g. RTTI), then you need both of Ray Lischner's early books, *Secrets of Delphi 2* [Waite Group, ISBN: 1-57169-026-3] and *Hidden Paths of Delphi 3* [Informant Press, ISBN: 0-9657366-0-1]. The first deals with a variety of advanced topics, and the second concentrates on the Open Tools API. If you write components, Ray Konopka's seminal treatise on this discipline, *Developing Custom Delphi 3 Components* [Coriolis, ISBN: 1-88357-747-0] is the fundamental book you must include in your library. Also be sure to keep an eye open for Danny Thorpe's *Delphi Component Design* [Addison-Wesley, ISBN: 0-201-46136-6], an insightful work by one of Delphi's best-known architects.

If you're working with Delphi at the API level, you should own each of the first two volumes of the *Tomes of Delphi 3* series by John Ayres, et al.: *Win32 Core API* [Wordware, ISBN: 1-55622-556-3] and *Win32 Graphical API* [Wordware, ISBN: 1-55622-610-1]. If you're looking for an advanced work that covers a multitude of topics from memory management to hacking the Delphi environment, take a look at *Delphi Developer's Handbook* [SYBEX, ISBN: 0-7821-1987-5] by Marco Cantù, Tim Gooch, and John Lam.

I have reviewed most of these books, so be sure to check your old issues of *Delphi Informant Magazine*. Of course, all of this magazine's book reviews are available at http://www.DelphiZine.com. Until next time ... 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.