October 1996, Volume 2, Number 10

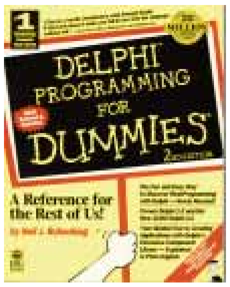
Cover Art By: *Tom McKeith*

OLE Automation

Controlling One Application with Another

New Delphi Book

Delphi Programming for Dummies 2nd Edition
Neil J. Rubenking
IDG Books



ISBN: 1-56884-621-5
Price: US$24.99 (418 pages)
Phone: (800) 762-2974

NuMega Technologies Supports Delphi 2 with BoundsChecker 4.0

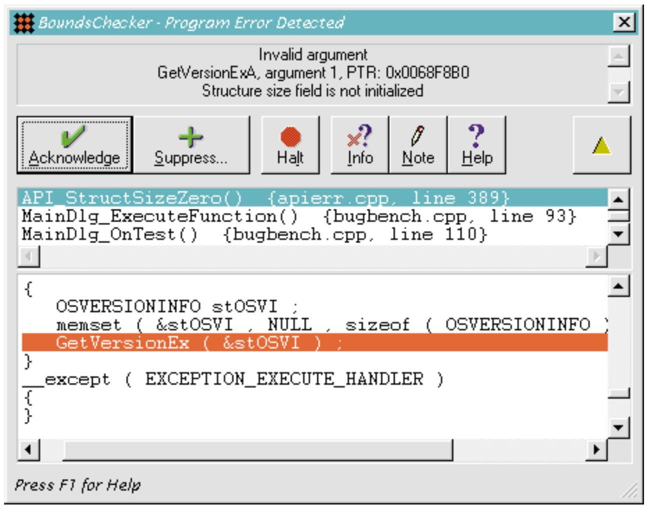
NuMega Technologies, Inc., of Nashua, NH has released ***BoundsChecker 4.0***, with enhancements to BoundsChecker's error detection solutions.

Version 4.0 supports Delphi 2 and ApiGen, a utility for the validation of user-extensible API calls.

The newest version of BoundsChecker supports ActiveX, and Smart Debugging (background error detection).

BoundsChecker is the only error detection tool that can evaluate Internet APIs, interfaces, and controls, including URLMON and HLINK, while validating hundreds of DLL functions. This includes the Internet's WinSock API.

BoundsChecker's new OLECheck features detect



```
Invalid argument
GetVersionExA, argument 1, PTR: 0x0068F8B0
Structure size field is not initialized
```

```
API_StructSizeZero()  {apierr.cpp, line 389}
MainDlg_ExecuteFunction()  {bugbench.cpp, line 93}
MainDlg_OnTest()  {bugbench.cpp, line 110}
```

```
{
   OSVERSIONINFO stOSVI ;
   memset ( &stOSVI , NULL , sizeof ( OSVERSIONINFO )
   GetVersionEx ( &stOSVI ) ;
}
__except ( EXCEPTION_EXECUTE_HANDLER )
{
}
```

OLE interface leaks and invalid parameters, and return codes for over 70 OLE interfaces.

BoundsChecker can also identify more than 85 errors in eight categories by type, stack trace, and exact location in source code.

Price: BoundsChecker Standard Edition, US$299.
Contact: NuMega Technologies, Inc., 9 Townsend West, Nashua, NH 03063
Phone: (800) 468-6342 or (603) 889-2386
Fax: (603) 889-1135 or (603) 889-2386
E-Mail: Internet: info@numega.com
Web Site: http://www.numega.com

Mercury Interactive and Borland Announce Client/Server Testing Solution

Mercury Interactive Corp. of Sunnyvale, CA, and Borland have announced a new client/server testing solution that integrates WinRunner 4.0 and LoadRunner 4.0 testing tools with Delphi Client/Server Suite 2.

The new Delphi extensions built into WinRunner 4.0, a GUI testing tool, and LoadRunner 4.0, a client/server and Web Load Testing tool, will integrate a Visual Testing environment that organizes automated testing facilities and makes them available to Delphi testers. The Delphi extensions also include automated record, replay, and verification for Delphi GUI object testing.

The integration of Delphi and WinRunner 4.0 includes: Visual Testing, enabling users to visually record and replay test scenarios using a point-and-click method of selecting Delphi objects; and Verification Testing, ensuring Delphi testers know when the application is performing as expected.

It also has SMARTest (Script Mapping for Adaptable and Reusable Test technology) to handle application changes automatically. SMARTest maintains Delphi object specific data independently of individual scripts. Its architecture separates variable object data from scripts, ensuring the same scripts can be reused even as the application changes during development.

The integration of Delphi and LoadRunner 4.0 includes: Client Load Testing, simulating multiple Delphi client users across the network; Server Load Testing, providing a way to stress a server's capacity by simulating the activity generated by multiple Delphi applications on a single PC; and Web Load Testing, for stressing a Web server capacity by recreating HTTP traffic through multi-tasking Web virtual users.

Delphi extensions for WinRunner and LoadRunner will be available this month for Windows 95 and Windows NT. Mercury Interactive will offer the Delphi extensions to existing customers free of charge.

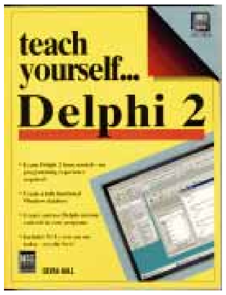
Price: LoadRunner 4.0 Standard edition (Windows 3.1), US$14,990; Professional edition (Windows 3.1, Windows 95, and NT), starts at US$40,000; Professional edition (UNIX), starts at US$50,000. WinRunner 4.0 Standard edition (Windows 3.1),US$2,850; Professional edition (Windows 3.1, Windows 95, and NT), US$3,995.
Contact: Mercury Interactive, 470 Potrero Ave., Sunnyvale, CA 94086
Phone: (800) 759-3302 or (408) 523-9900
Fax: (408) 523-9911
Web Site: http://www.merc-int.com

Delphi
TOOLS

New Products and Solutions

New Delphi Book

teach yourself... Delphi 2
Devra Hall
MIS Press



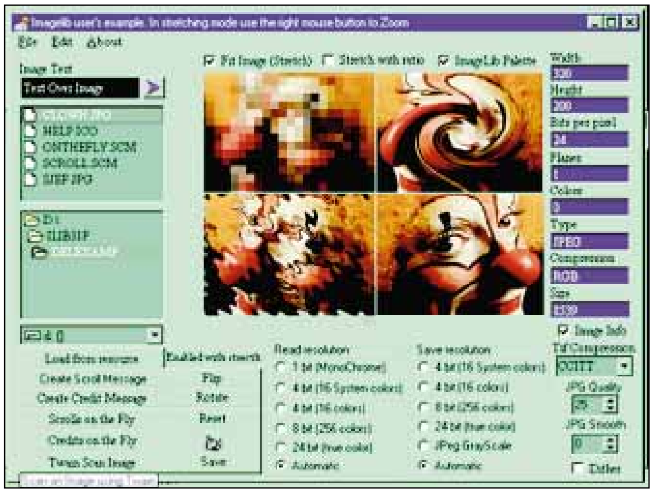
ISBN: 1-55828-457-5
Price: US$29.95
(322 pages, Disk)
Phone: (800) 488-5233

SkyLine Tools Introduces ImageLib@theEdge for Delphi

In North Hollywood, CA, **SkyLine Tools** announced the debut of ***ImageLib-@theEdge***. It sells as a stand-alone component for manipulating images in Delphi, or as an add-on to current ImageLib packages. These include ImageLib Portfolio 3.1, ImageLib '95, and ImageLib Combo.

theEdge produces imaging effects including mosaic, hole punch, page curl, whirlpool, and wave.

Additionally, theEdge features color and reduction properties; gamma correction; contrast, brightening, sharpening features; and can scale, crop, and rotate images by degree. It also features five new transitions (wipes and fades) that provide effects for presentations.



A free trial version is available from Skyline's Web site.

Price: theEdge stand-alone version US$129; ImageLibCombo@theEdge US$299.

Contact: SkyLine Tools, 11956 Riverside Dr. Suite 107, North Hollywood, CA 91607
Phone: (818) 766-4561
Fax: (818) 766-9027
E-Mail: Internet: 72130.353@compuserve.com
Web Site: http://www.imageLib.com

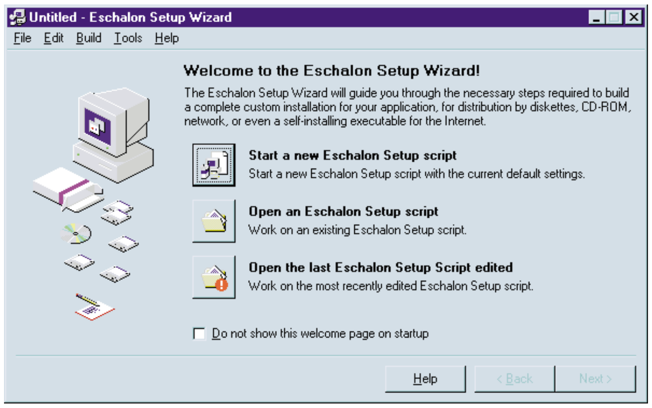
Eschalon Development Releases Power Libraries for Delphi 2

Eschalon Development Inc. of Coquitlam, BC, Canada has released ***Eschalon Power Libraries***. Developed specifically for Delphi 2, and compatible with Windows 95 and Windows NT, Eschalon Power Libraries provides over 450 functions for Delphi 2 applications.

Included in the product are functions for string manipulation, string parsing and tokens, file and disk access, date and time, hashing/checksums/CRC, MD5 message digest, sorting, searching, timing, and debugging.

Eschalon Power Libraries feature streams for memory mapped files, classes for containers (integer, string, indexed, variables, etc.), section and log files, and registry access. They also have asynchronous/synchronous execution of programs, as well as a "sendkeys" feature, and allow developers to use the custom structured file class for unique file formats.

Price: US$149.95
Contact: Eschalon Development Inc., 24-2979 Panorama Drive, Coquitlam, BC, Canada V3E 2W8
Phone: (604) 945-3198
Fax: (604) 945-7602
E-Mail: Internet: info@eschalon.com
Web Site: http://www.eschalon.com



Welcome to the Eschalon Setup Wizard!

The Eschalon Setup Wizard will guide you through the necessary steps required to build a complete custom installation for your application, for distribution by diskettes, CD-ROM, network, or even a self-installing executable for the Internet.

Start a new Eschalon Setup script
Start a new Eschalon Setup script with the current default settings.

Open an Eschalon Setup script
Work on an existing Eschalon Setup script.

Open the last Eschalon Setup Script edited
Work on the most recently edited Eschalon Setup script.

☐ Do not show this welcome page on startup

Open Window Releases OWShare

Open Window of Colorado Springs, CO has released ***OWShare,*** a source code package that provides screens and support code to turn program functionality into a shareware product.

OWShare is available in Delphi and Visual Basic versions that support ASP-compliant time and usage limiting, branding, escalating reminder screens, site licenses, evaluation period extensions, and other shareware features.

With OWShare, a developer can initialize the global variables appropriate for a project, edit the supplied forms with Delphi's built-in visual editors, and then "drop in" his or her program.

OWShare coincides with a revision of the requirements for shareware programs released by members of the Association of Shareware Professionals.

A demonstration program is available on the WINUTIL CompuServe Forum (OWSHR.ZIP), and

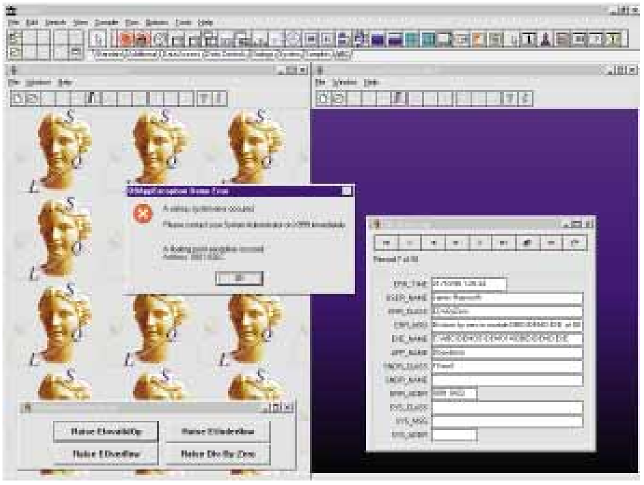
"Open Window Releases OWShare" continued on page 4

Objective Software Supports Delphi 2 with Product Updates

Objective Software Technology of Canberra, ACT Australia has announced Delphi 2 updates for their *ABC for Delphi* and *TRANSFORM* products. Both products are now available for Delphi 1 and 2.

ABC for Delphi is a library of 30 visual components including exception handling, user interface, and data components.

TRANSFORM is a Delphi component that allows generation of aggregate components from Delphi forms. It's suited for prototyping and iterative development of complex components. Patch updates for current users are available on the Informant CompuServe forum (GO ICGFORUM).



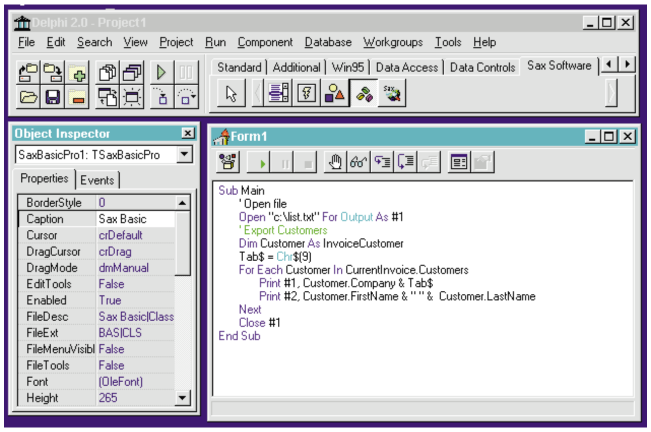
Price: ABC, US$89 or US$179 (with source). TRANSFORM, US$125.
Contact: In the US: ZAC Catalog, 1090 Kapp Drive, Clearwater, FL 34625
Phone: (800) 463-3574 or (813) 298-1181
Fax: (813) 461-5808
E-mail: Internet: sales@zac-catalog.com
Web site: http://www.zaccatalog.com
Contact: Outside the US: Objective Software Technology Pty Ltd., PO Box E138 Kingston, ACT 2604 Australia
Phone: (+61) 6-273-2100
Fax: (+61) 6-273-2190
E-mail: Internet: 100035.2441@-compuserve.com
Web Site: http://ourworld.com-puserve.com/homepages/objsoft or http://www.obsof.com

Sax Ships Basic Engine Pro

Sax Software, of Eugene, OR is now shipping the latest version of *Sax Basic Engine Pro 3.0*. An ActiveX control, Sax Basic Engine Pro integrates with Delphi, Visual C++, and Visual Basic applications, and supports events, classes, and multiple modules.

Sax Basic Engine Pro offers syntax highlighting, and includes an integrated editor and debugger that can call a stack, set a variable watch, or single-step through code. It also allows users to manipulate events by treating everything as an object. Whenever an object receives an action, an event is fired.

Sax Basic Engine Pro also works with multiple modules in macro code. Users can create complex macros and customize common dialog boxes.


Delphi 2.0 - Project1
File Edit Search View Project Run Component Database Workgroups Tools Help
Standard Additional Win95 Data Access Data Controls Sax Software
Object Inspector
Form1
Properties Events

SaxBasicPro1: TSaxBasicPro

BorderStyle	0
Caption	Sax Basic
Cursor	crDefault
DragCursor	crDrag
DragMode	dmManual
EditTools	False
Enabled	True
FileDesc	Sax Basic|Class
FileExt	BAS|CLS
FileMenuVisibl	False
FileTools	False
Font	(OleFont)
Height	265

```
Sub Main
    ' Open file
    Open "c:\list.txt" For Output As #1
    ' Export Customers
    Dim Customer As InvoiceCustomer
    Tab$ = Chr$(9)
    For Each Customer In CurrentInvoice.Customers
        Print #1, Customer.Company & Tab$
        Print #2, Customer.FirstName & " " &  Customer.LastName
    Next
    Close #1
End Sub
```

Price: US$495; requires no royalties or run-time fees, and ships with a 30-day money-back guarantee.
Contact: Sax Software, 950 Patterson St., Eugene, OR 97401
Phone: (800) 645-3729 or (541) 344-2235
Fax: (541) 344-2459
E-Mail: Internet: info@saxsoft.com
Web Site: http://www.saxsoft.com

Open Window Releases OWShare (cont.)

America Online in the Development Forum (OWSHR_10.ZIP).

In addition to the online Help, support is available via CompuServe, America Online, Internet, or US Mail. OWShare requires Windows 3.1 or Windows 95, and Delphi.

Price: US$59.95
Contact: Open Window, PO Box 49746, Colorado Springs, CO 80949-9746
Phone: (800) 531-0403 or (719) 531-0403
Fax: (719) 531-0403
E-Mail: Internet: 75236.3243-@compuserve.com
CIS Forum: GO WINUTIL
Web Site: http://www.openwindow.com

CD Cleaner

303 Products, Inc. of Palo Cedro, CA is shipping 303 Sonic Blast. This anti-static agent repels dust, fingerprints, and smudging on CDs. Discs treated with this biodegradable formula are protected for up to one year.

Each Kit contains a spray bottle and an applicator pad, and treats approximately 300 CDs.

Price: US$12.95
Contact: 303 Products, Inc., PO Box 966, Palo Cedro, CA 96073
Phone: (916) 549-5617
Fax: (916) 549-5577
Web Site: http://www.303-products.com

table of CONTENTS

News LINE

October 1996

ROI Systems Chooses Delphi

ROI Systems, Inc., an Enterprise Resource Planning (ERP) software supplier, has selected Delphi to create the interface for its next generation of integrated business systems.

ROI Systems is changing its GUI presentation, instead of updating their ASCII screens with screen scraper products that give a GUI appearance, but still require multiple steps to move from one function to another to complete a transaction. ROI's new client interface will in some cases combine 15 separate functions on one screen.

For more information, call: (800) 544-7849, e-mail: sales@roisysinc.com, or visit ROI's Web site at http://www.roisysinc.com.

EDS Chooses Delphi to Develop Case Tracking System

Scotts Valley, CA — Electronic Data Systems Corp. (EDS) has selected Delphi to develop a client/server-based Case Activity Tracking System (CATS) for the National Labor Relations Board (NLRB). The NLRB will use CATS to manage over 40,000 new legal cases annually, as well as to report statistics.

The NLRB, under the Defense Enterprise Integration Services contract, selected EDS to help convert a mainframe case tracking system to a client/server architecture, and develop a new prototype system for legal case tracking.

EDS used Delphi and the Borland RAD Pack to create preliminary CATS screens currently being demonstrated to NLRB staffers around the country for design review and user input. The CATS system is scheduled to begin full deployment in 1997.

For more information visit EDS' Web site at http://www.eds.com.

Borland Presents Golden Gate Internet & Intranet Strategy

Anaheim, CA — In a keynote presentation to more than 1,500 developers at the 7th annual Borland Developers Conference, Borland presented its overall Internet strategy and demonstrated new technologies, including products from the planned acquisition of Open Environment Corp.

Referred to as the Golden Gate initiative, Borland plans to merge both phases of its Internet plans: release Internet-enabled versions of its existing products; and then add Intranet solutions for the workgroup market segment.

With its Golden Gate strategy, Borland hopes to assist users building Internet-based applications by integrating existing investments in client/server architectures with the emerging Internet technologies. The strategy includes combining object-oriented development tools, Delphi, Latte, IntraBuilder, Borland C++, as well as Open Environment's Entera technology and related Internet products.

For more information, contact Borland at (408) 431-1064 or visit Borland Online at http://www.borland.com.

VCL Contest Winners Announced

Scotts Valley, CA — SAMS Publishing and Borland Press have announced the winners of *Delphi Informant*'s VCL Contest. Due to the extended deadline and delay in posting results, SAMS and Borland are awarding an extra book to each of the winners. *Database Developer's Guide with Delphi 2* will be awarded in addition to *Delphi Developer's Guide, Second Edition* as part of the VCL contest prize package. Top winners will also receive US$100 and a six-month subscription to *Delphi Informant.*

VCL entries were judged by representatives from Borland International, SAMS Publishing, and Informant Communications Group, Inc. for originality and functionality. Entries were received from around the world in each of the five categories.

The winner in the Internet/Communications category was Jower Garcia Toppin of Venezuela.

"VCL Contest Winners Announced" continued on page 6

Borland Ships Trial, Learning, and Low-Price Editions of Delphi

Scotts Valley, CA — Borland has released three new versions of Delphi, including a trial edition that users can download free from Borland Online; a "Learn to Program with Delphi" package for students and beginning programmers; and an easy-to-use, low-price version of Delphi 2 for developers to evaluate.

The "Learn to Program with Delphi" package includes the 16-bit Delphi 1 software, *Teach Yourself Delphi in 21 Days*, Computer-Based Training (an online curriculum), and new sample applications with source code.

The new special version of Delphi 2 has been re-designed with features making it more accessible to new developers. In addition, Borland has included new functionality to show how Delphi can be an add-on tool for Visual Basic and C++ developers. This version of Delphi 2 includes *Teach Yourself Delphi 2 in 21 Days*, an online Visual Basic-to-Delphi command reference, and an online Delphi reference for C++ developers.

The Delphi trial version is available from Borland Online at http://www.borland.com/delphi20/ or on CD (shipping and handling charges additional).

The "Learn to Program with Delphi" package and the low-price version of Delphi 2 are available now. "Learn to Program with Delphi" is US$49.95; the modified of Delphi 2 is US$99.95.

Borland Announces Availability of its Latest Version of InterBase

Scotts Valley, CA — Borland International Inc. has announced version 4.2 of InterBase. This version of InterBase has improved performance and resource usage, due to an enhanced version of the SuperServer Architecture.

InterBase 4.2 features ODBC 2.5 drivers for Windows 95 and Windows NT, and thread-safe 32-bit client libraries. It is also compatible across the Windows and UNIX environments maintaining one consistent API, database format, and SQL language. This consistency across platforms enables developers to write a server application once, and deploy it to either Windows or UNIX operating systems.

By adhering to industry standards such as SQL 92, and offering 32-bit ODBC drivers for Windows 95 and Windows NT, users can integrate third-party software products. In addition, Borland partnered with Visigenic Software earlier this year to develop the InterBase ODBC drivers.

The InterBase SuperServer Architecture offers thread-safe client and multi-threaded server. Its unique versioning engine ensures data availability for both transaction processing and decision support style applications.

InterBase 4.2 also includes performance enhancements for large multi-user systems; 32-bit GUI tools for interactive SQL, server, and license management; 32-bit ODBC drivers for Windows 95 and Windows NT; license manager expert to ease user license management; identical code base and feature set across Windows 95 and Windows NT platforms. It is also certified and optimized for Microsoft NT 4.0, and tuned for use with Borland's forthcoming Java JDBC driver for InterBase and InterClient.

The InterBase 4.2 family of products includes: Local InterBase (a single user version of the server), US$249.95; InterBase Server for Windows 95 (a multi-user server for up to four concurrent users), US$599.95; and InterBase Server for Windows NT (a departmental-to-enterprise server including five user licenses), US$850.

For more information visit Borland Online at http://www.borland.com.

Borland Announces Resignation of CFO

Borland has announced the resignation of David Mullin, vice-president and chief financial officer. Mullin will to continue his duties until mid-September. Borland is currently seeking a replacement for Mullin.

VCL Contest Winners Announced (cont.)

Toppin submitted Mail eXtension, a visual component adding e-mail features to Delphi applications, supporting Lotus Notes, cc:Mail, and Microsoft Mail and Exchange.

Lance Leverich of Kansas submitted the winning entry in the Database category. His *TMailLabel* component is used to define and print mailing labels, allowing developers to set the LabelDefinition to provide a pre-defined label layout. Using the various link fields, developers can set the interface to the database(s) for retrieval of mailing address information.

Jan Dekkers of SkyLine Tools in CA submitted ImageLib, the winning entry in the Multimedia category. ImageLib 3.1, from Skyline Tools, is a software development tool allowing programmers to implement BMP, CMS, GIF, ICO, JPG, PCX, PNG, SCM, THB, TIF, and WMF images into an application. In addition, AVI, MOV, MID, WAV, and RMI formats can be implemented to or from a file or database BLOB field.

The winning entry in the Interface category was received from Peter Andrew Van Lonkhoyzen of South Africa. Van Lonkhoyzen's *TDocPanel* is derived from *TPanel* and acts as a dockable toolbar with customizable behavior. The Dockbar package contains both 16- and 32-bit versions of the control.

Kenneth Clubb of Maryland submitted the winning entry in the Other category, with *TConnections*, a special *TPanel* that shows master-detail relationships by drawing lines between *TTables* and *TDataSources*.

Clubb was also the Grand Prize Winner for his Master-Detail Explorer, which provides a view of master-detail information for programmers and users. Each level of the tree expands into detailed information when selected, displaying up to eight levels of detail.

In addition to the prizes for winning the Other category, Clubb will receive a US$1,000, a copy of *Delphi Informant Works '96* and Delphi Client/Server, and an additional copy of each of the Delphi books from Borland Press for winning the Grand Prize.

A list of winners is available on SAMS' Web site (http://www.mcp.com/sams), Borland's Web site (http://www.borland.com), SAMS' CompuServe forum (GO SAMS), and Borland's CompuServe forum (GO BORLAND).

table of CONTENTS

Delphi 2 / Object Pascal / OLE

By Cary Jensen, Ph.D.

OLE Automation

Controlling One Application with Another

OLE automation is a convention by which one application can control another. The controlling application is referred to as the *automation client*, and the one being controlled is referred to as the *automation server*. The 32-bit applications you build with Delphi 2 can be automation clients or servers, or both.

In general, there are two types of automation servers: *in-process* and *local*. In-process servers are DLL-based, and local servers are .EXE-based. Again, you can easily create both of these with Delphi 2.

OLE automation servers can make properties and methods available to the automation client. Provided the automation server developer designs the server appropriately, the automation client can exert detailed control over the server.

Using an Automation Server

The four steps to using an automation server are:

1) Declare a variant. A variant is a loosely-typed variable that you employ to access and control the automation server.
2) Open the automation server using the *CreateOLEObject* function.
3) Set the properties and/or call the methods of the server.
4) Release the server when you are done.

Declaring a Variant for Use with a Server

Delphi 2 introduces a new data type called a *variant*, a variable whose type does not have to be known at compile time. In most cases, variants are used to control automation servers. The following is an example of how you can declare a variable named *MyServer* to be of the type variant:

```
var
  MyServer: Variant;
```

Opening an Automation Server

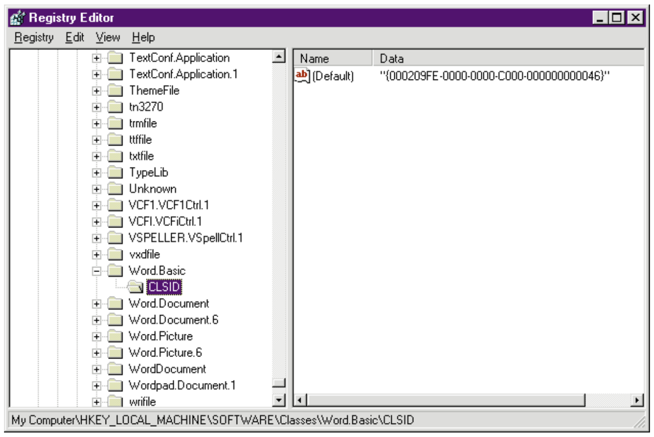
You open an automation server by calling *CreateOLEObject*. This function is declared in the OLEAuto unit. Therefore, you must include the OLEAuto unit in the **uses** clause of the unit from which you call *CreateOLEObject*.

CreateOLEObject requires a single string parameter containing the name of the automation server. This is the name the server registers in the HKEY_CLASSES_ROOT key of the Registry, and it associates the server with a CLSID (class ID), a value assigned to the server by Windows when the server is registered.

All automation servers have a unique name, as well as a unique CLSID. The server name is case-insensitive. Also, Delphi does not require you to reference the CLSID when using an automation server.

Figure 1 shows Registry Editor, a Windows 95 application that allows you to view and edit the Registry. Notice that the server name for WINWORD.EXE is `Word.Basic`. Consequently, to open Microsoft Word for Windows (Word) as a local automation server with the variable *MyServer*, you would execute the following statement:

```
MyServer := CreateOLEObject('word.basic');
```



```
TextConf.Application
TextConf.Application.1
ThemeFile
tn3270
trmfile
ttffile
txtfile
TypeLib
Unknown
VCF1.VCF1Ctrl.1
VCFI.VCFiCtrl.1
VSPELLER.VSpellCtrl.1
vxdfile
Word.Basic
  CLSID
Word.Document
Word.Document.6
Word.Picture
Word.Picture.6
WordDocument
Wordpad.Document.1
wrifile
```

Name	Data
(Default)	"{000209FE-0000-0000-C000-000000000046}"

My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Word.Basic\CLSID

Figure 1: Viewing the Registry with Registry Editor.

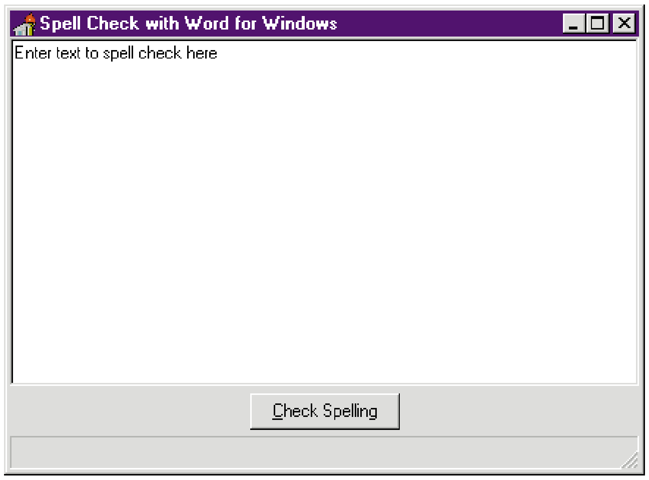
Controlling an Automation Server

After you have opened an automation server, you can control it by setting values to its properties, or calling its methods. Delphi does not provide you with the tools necessary to discover a particular server's properties and methods. (You could write such a tool using API calls, but that is outside the scope of this discussion.)

Normally, you will refer to the documentation provided by the server's developer to learn what properties and methods are available, and how to use them.

Word publishes a method for creating a new document. This method, *FileNew*, takes no parameters. Therefore, once *FileNew* is opened, you can instruct Word to create a new document by including the following statement in your code:

```
MyServer.FileNew;
```



Enter text to spell check here

Figure 2: The SPELL project Memo object.

Releasing the Server

There are two ways to release an automation server. One way is to permit the variant to go out of scope. For example, if you declare a variant local to a procedure, and then exit the procedure, the server opened with the variant will close (assuming that some other process, such as an application, is not also using the server).

The second way to close an automation server is to assign the constant *UnAssigned* to the variant. To use the same example, for instance, to release Word as a server, you would make the following assignment:

```
MyServer := UnAssigned;
```

Using an Automation Server Example

The project SPELL.DPR (see Figure 2) displays a simple Memo object. When you click the button labeled **Check Spelling**, the project opens Word as a local automation server, copies the memo to the Windows Clipboard, and then uses a series of method calls to instruct Word to spell check the document.

When the spell check is complete, additional server method calls are used to select the text and copy it back into the Clipboard, from which it is pasted back into the Memo object. Figure 3 is associated with the button's *OnClick* event handler.

SPELL.DPR is a simple example of using a local server. However, SPELL.DPR is provided for demonstration purposes only. If you attempt to use this example and Word is already running, an exception will be raised when the code attempts to close the server.

If you need to add spell checking to your Delphi applications, you are much better off using one of the third-party spell checkers available, instead of a large and sometimes ill-behaved server like Word. Delphi 2, for example, includes the VCSpeller OCX that you can use to spell check text without the problems associated with using Word.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  WinWord: Variant;
begin
  StatusBar1.SimpleText := 'Spell checking...';
  Application.ProcessMessages;
  // Open the OLE link to WinWord.
  WinWord := CreateOLEObject('word.basic');
  // Select all of the memo text.
  Memo1.SelectAll;
  // Copy the selected text to the Clipboard.
  Memo1.CopyToClipboard;
  // From WinWord, create a new file.
  WinWord.FileNew;
  // Paste Clipboard contents into the new file.
  WinWord.EditPaste;
try
  try
    try
      // Initiate spell checking.
      WinWord.ToolsSpelling;
    except
      // WinWord creates a message indicating that the
      // spell check is complete. This generates an
      // exception. Ignore this exception.
    end;
    // Select the spell checked text.
    WinWord.EditSelectAll;
    // Copy it to the Clipboard.
    WinWord.EditCopy;
  finally
    // Close WinWord.
    WinWord.Cancel;
    WinWord := UnAssigned;
  end; // Try-finally
except
  Exit;
  StatusBar1.SimpleText := 'Could not spell check'
end; // Try-except
// Bring the spell checked text back.
Memo1.PasteFromClipboard;
StatusBar1.SimpleText := 'Spell checking complete'
end;
```

Figure 3: The *OnClick* event handler for the **Check Spelling** button.

Creating an Automation Server

Delphi 2 makes it easy to create automation servers from your existing applications and DLLs. This allows you to expose some or all of your existing application's features to automation clients.

The four steps to convert an existing application into an automation server are:

1) Add a unit to your project that derives a new class from the *TAutoObject* class.
2) Within this derived class, declare any methods you want to expose to automation clients within the **automated** section.
3) Within this derived class, declare any properties you want to expose to automation clients within the **automated** section. These properties must use access methods to read and write the properties.
4) Implement the class and access methods.

This process is demonstrated with the project AUTOEX.DPR, an automated version of the local table packing utility described in the article "BDE Basics" [see the March 1996 *Delphi Informant*].

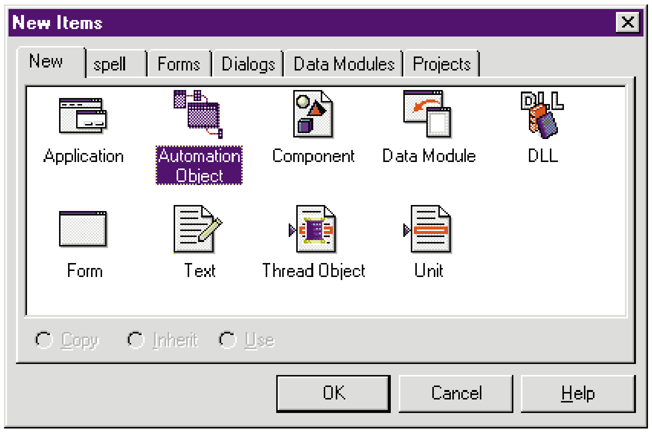
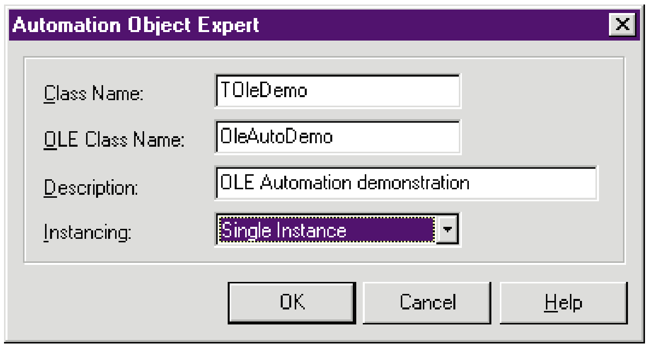


Figure 4: The Repository with the Automation Object selected.



Class Name: TOleDemo
OLE Class Name: OleAutoDemo
Description: OLE Automation demonstration
Instancing: Single Instance

Figure 5: You define the class and program name of your automation server using the Automation Object Expert.

Adding a *TAutoObject* Unit

You add a *TAutoObject* unit to your project by selecting **File | New**. Select the Automation Object from the Object Repository (see Figure 4). Delphi then displays the Automation Object Expert (see Figure 5).

In **Class Name**, enter a name for your derived class. Set **OLE Class Name** to the complete name that you want to register as the automation server in the Registry. Delphi will automatically complete this for you, based on your chosen class name, but you can change it if you wish.

In **Description**, provide a description of the automation object. This description will also be stored in the Registry. Finally, set the **Instancing**. Normally, applications use a `Single Instance` value, and DLLs use `Multiple Instance`.

Once you accept the Automation Object Expert, it will create and add a new unit to your project. Figure 6 is an example of how this unit will appear. (For more information on registering new applications, see the sidebar "Registration Note" on page 11.)

In this generated code, there are two critical elements to note. First, the expert has inserted a registration procedure that adds the server to the Registry. This procedure is called from the unit's **initialization** section. After you initially run this project, do not modify this code.

```
unit Unit1;

interface

uses
  OleAuto;

type
  TOleDemo = class(TAutoObject)
  private
    { Private declarations }
  automated
    { Automated declarations }
  end;

procedure RegisterOleDemo;
const
  AutoClassInfo: TAutoClassInfo = (
    AutoClass: TOleDemo;
    ProgID: 'OleAutoDemo';
    ClassID: '{3EE99F30-F141-11CF-82A3-444553540000}';
    Description: '';
    Instancing: acSingleInstance);
begin
  Automation.RegisterClass(AutoClassInfo);
end;

initialization
  RegisterOleDemo;
end.
```

Figure 6: The Automation Object Expert creates a new unit and adds it to your project.

Second, a new clause, **automated**, appears in the **type** declaration. You use **automated** to surface properties and methods to automation clients.

Declaring Server Methods

You declare methods for the server by adding them to the **automated** clause of the **type** declaration. For example, to add a method that can be called by the client, and have this method pack a table, you can modify the **type** declaration as follows:

```
type
  TDelphiOLEServer = class(TAutoObject)
  private
    { Private declarations }
  automated
    { Automated declarations }
    procedure PackTab; virtual;
 end;
```

These methods can be declared virtual, but not dynamic. Furthermore, they must use the register calling convention, which is the default. [Virtual and dynamic methods are described and compared in Ray Lischner's article "Virtual or Dynamic" in the May 1996 *Delphi Informant*.]

If you want, you can include the **dispid** directive, followed by an integer. This will register the method using the specified OLE Automation dispatch ID. If you do not include **dispid**, the compiler will automatically assign one.

Declaring Server Properties

You declare properties that you want to expose to your automation clients by declaring them within the **automated** clause. While this process is similar to how you declare properties in objects, in general, there are several restrictions:

- Automated properties cannot use direct access. You must declare and use read and write access methods for each property you declare.
- Only certain property types are permitted. These are *Byte*, *Currency*, *Double*, *Integer*, *LongInt*, *Single*, *SmallInt*, *string*, *TDateTime*, *Variant*, and *WordBool*.
- You cannot use the **index**, **stored**, **default**, or **nodefault** specifiers in automated property declarations.

To allow the automation client to define which table to pack using a property, you can add a *TabName* property to the automation server. The following is an example of how the **type** declaration may appear after doing so:

```
type
  TOleDemo = class(TAutoObject)
  private
    { Private declarations }
    FTabName: string;
  protected
    function GetTabName: string;
    procedure SetTabName(Value: string);
  automated
    { Automated declarations }
    procedure PackTab; virtual;
    property TabName: string
      read GetTabName write SetTabName;
  end;
```

Implementing Automation Methods

You implement automation methods the same way that you implement methods for any type of object. The following is how the *GetTabName*, *SetTabName*, and *PackTab* methods might appear:

```
function TOLEDemo.GetTabName: string;
begin
  Result := FTabName;
end;
procedure TOLEDemo.SetTabName(Value: string);
begin
  if FTabName <> Value then
    FTabName := Value;
end;
procedure TOLEDemo.PackTab;
var
  Tab: PChar;
begin
  GetMem(Tab,144);
  try
    StrPCopy(Tab,FTabName);
    Form1.PackTable(Form1,Tab);
  finally
    FreeMem(Tab,144);
  end;
end;
```

The *GetTabName*, *SetTabName*, and *PackTab* methods appear as they would in any object to which you are adding methods. However, the *PackTab* method deserves special attention.

PackTab is called by the automation client to produce an action by the server. From within *PackTab* you can set properties of objects within the server application, call object methods, and execute functions and procedures.

Registration Note

I've created OLE Automation servers from my existing applications for some time. However, since the release of Delphi 2.01, I could not add an automation server unit to an existing application and have it successfully register with the Registry. I have only been able to register new applications.

The only way I can successfully register an existing application is by performing the following steps. I begin by creating a new, blank application. After saving this application to a directory, I add the automation unit, and again save the application. I then run the application to register it.

Once the automation server is registered, I remove the default *Form1* from this application, and add each unit from my existing application that I want to convert into an automation server. When I am done, I have a new application with all the units and forms from my existing application, in addition to the OLE automation unit. Furthermore, this application is a registered automation server.

I want to point out that this may not actually be a problem with Delphi. Instead, it may be because of the large number of example applications I create while writing training materials. I may have damaged my Registry, or somehow affected Delphi in a way that prevents it from performing correctly. I haven't tried re-installing Delphi 2.01, but doing so may clear up this problem. If you can add an automation unit to an existing application, you do not need to worry about the above steps.

— *Cary Jensen*

In this example, the *PackTab* method calls the method *Form1.PackTable*, passing to it the name of the table stored in the *TabName* property.

Calling Your Automation Server

The automation server created in the preceding example demonstrates how easily you can convert an existing application into an automation server.

Using your new server from an automation client is even easier. Simply use the technique described earlier in the section "Using an Automation Server."

This is demonstrated by the code in the project CALLAUTO.DPR. The following shows how to open the OLEAutoDemo automation server, set the *TabName* property of the server, and then call the *PackTab* method, all from a button's *OnClick* event handler:

```
implementation

uses OleAuto;

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
var
  PackServer: Variant;
begin
  PackServer := CreateOLEObject('OLEAutoDemo');
  PackServer.TabName := Edit1.Text;
  PackServer.PackTab;
  PackServer := UnAssigned;
end;
```

Conclusion

Delphi makes it easy to convert any of your Delphi 2 applications into automation servers. By doing so, you enable other programs, even those not written in Delphi, to leverage the features of your existing applications. ∆

The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\OCT\DI9610CJ.

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, including *Delphi In Depth* [Osborne/McGraw-Hill, 1996]. He is also Contributing Editor of *Delphi Informant*. You can reach Jensen Data Systems at (713) 359-3311, or via CompuServe at 76307,1533.

Delphi 1 / Delphi 2 / Object Pascal

By Ray Lischner

Classy DLLs

Exporting a Class and Its Methods from a Delphi DLL

Dynamic link libraries (DLLs) are essential to Windows programming. Fortunately, Delphi has excellent support for creating and using DLLs, making it easy to employ a DLL to export a function or procedure. Moreover, with just a little more work, you can export a class and its methods.

This article describes how to export a class from a DLL, import a class in an application, and use objects that cross module (application and DLL) boundaries.

There are two facets of mixing classes, objects, and DLLs:
- the static, compile-time nature of exporting and importing classes, and
- the dynamic, run-time nature of using objects and Run-Time Type Information (RTTI).

The general principles for dealing with these aspects of DLLs are the same for Delphi 1 and 2, but the syntax can be different. This article addresses the problems and solutions for both versions.

Review of DLLs

To understand how to use classes with DLLs, you must first understand how DLLs work *without* classes. Earlier this year, Delphi DLL basics were covered in a series of articles by Andrew Wozniewicz [see "DLLs" Parts I through IV in the March through June 1996 issues of *Delphi Informant*.] Here's a quick review, in case you missed that series or just want a refresher.

A DLL can be implemented in most programming languages. In Delphi, DLLs are created using the **library** keyword in the project's .DPR file, instead of the **program** keyword (used for creating an application). At run time, any application or DLL can load another DLL, calling its functions and procedures. What makes DLLs so attractive is that Windows keeps a single copy of the DLL's code in memory, no matter how many applications load the same DLL.

For example, a spelling checker implemented in a DLL can be used by many applications, such as a word processor, spreadsheet, or e-mail tool. If all three applications use the same spelling checker, a single copy of the spelling checker's code is loaded in memory, not three. And with a little more work, the spelling checker can share its data as well, using a single copy of the dictionary, saving even more memory.

A DLL can also be effective when used by a single application. In this case, the benefit of using a DLL is not the code sharing, but the ability to load the DLL only when it's needed. For example, a large application might be divided into multiple DLLs, each implementing a key area of functionality. When the function is needed, the application loads the DLL into memory. When the function is no longer required, the application unloads the DLL, freeing its memory. This can reduce the total amount of memory required by the application.

As a more specific example, a word processor application might use a separate DLL

```
library WPDoc;

procedure ReadWP(const Filename: string;
                 Doc: TDocument); export;
begin
  ...
end;

procedure WriteWP(const Filename: string;
                  Doc: TDocument); export;
begin
  ...
end;

exports
  ReadWP  name 'READDOC',
  WriteWP name 'WRITEDOC';
end.
```

Figure 1: Exporting routines from a Delphi DLL.

for each file format it can read or write. Without DLLs, the application would consume memory for read and write routines for file formats the user never encounters. With each file format in its own DLL, the application loads only the DLL of the file format in use.

For an application to use a DLL, the DLL must export one or more routines. The DLL contains a table of all the routines it exports. Each exported routine has a unique index, and can also have a name that can be different from its procedure name. For example, the word processor might require the file format DLL to have two routines. The routine with index 1 has the exported name of "READDOC", and the routine with index 2 has the name "WRITEDOC". A DLL that reads and writes WordPerfect files might use the name ReadWP internally, but export the routine under the name "READDOC". The **exports** statement specifies which routines are exported, with their export names and indexes (see Figure 1).

The word processor application imports routines from the DLL so it can call the routines as though they were part of the application. When calling a routine from a DLL, the application typically uses an **external** declaration which includes the name and arguments of the routine, as well as the DLL name and the exported index or name (see Figure 2). Thus, the application never knows the internal routine names used by the DLL; it only sees the exported interface.

```
procedure ReadDocument(const Filename: string;
  Doc: TDocument); external 'WPDOC' name 'READDOC';
procedure WriteDocument(const Filename: string;
  Doc: TDocument); external 'WPDOC' name 'WRITEDOC';
```

Figure 2: Importing a routine from a DLL.

Another benefit of this approach is that separate DLLs can implement the same interface, that is, the same exported indexes and names. After the word processor has loaded the proper DLL that corresponds to the desired file format, the rest of the application's code doesn't need to know which

```
var
  DllInst: THandle;
  ReadDoc: procedure(const Filename: string;
                     Doc: TDocument);
begin
  { Load the DLL. DllInst = 0 for an error. }
  DllInst := LoadLibrary('WPDOC.DLL');
  { Import the procedure. Test Assigned(ReadDocument)
    to check for errors. }
  if DllInst <> 0 then
    begin
      @ReadDocument := GetProcAddress(DllInst,'READDOC');
      ReadDocument(Filename, Document);
      { Unload the DLL when you are finished. }
      FreeLibrary(DllInst);
    end;
  ...
```

Figure 3: Calling a DLL routine dynamically.

DLL is loaded. The application calls the interface procedures according to their exported names. This makes it easier to add additional file formats: just add DLLs.

An **external** declaration, however, ties the application to a particular DLL. In this case, the word processor needs to load the file format routines from different DLLs at different times, depending on the user's file format choice. Instead of using an **external** declaration, the application can use the Windows API routine, *LoadLibrary*, to load a specific DLL, and then call *GetProcAddress* to locate a particular routine. Figure 3 illustrates how the word processor can load and call the READDOC routine. The path to the DLL is given by the variable *DllPath*, which contains the name of the DLL the application wants to read the bitmap (e.g. WPDOC.DLL).

Delphi 1 vs. Delphi 2

In the examples so far, an exported procedure has been declared with the **export** directive. An **export** directive informs Windows that the routine is being called across a module boundary, i.e. from an application to a DLL, from a DLL to an application, or from one DLL to another.

In Delphi 1, the **export** directive is required because Windows 3.1 uses a segmented memory architecture. When calling a routine across a module boundary, Windows must ensure that it is using the correct data segment. The **export** directive tells the compiler to generate special code at the start of a routine to establish the correct data segment. Code is also added to the end of the routine, restoring the data segment register to its previous value.

The **export** directive is not required in Delphi 2. The application and DLL share a flat memory architecture, so there are no memory segments to worry about. For purposes of backward compatibility, the Delphi 2 compiler ignores the **export** directive.

Instead of **export**, consider using the **stdcall** directive in Delphi 2. One of the Delphi 2 compiler's new features is its faster, register-based calling convention. When you call a function or procedure, Delphi doesn't push the first three arguments onto the stack. Rather, it loads them into registers, resulting in faster code execution. However, this calling convention is only compatible with Delphi.

The **stdcall** directive tells the compiler not to use Delphi's register calling convention, but to use the Windows standard calling convention, which pushes all arguments onto the stack in a standard order. All Windows API routines use the **stdcall** calling convention. To ensure maximum flexibility, use **stdcall** for all routines exported from a DLL. The application and DLL must declare the same calling convention, so remember to use **stdcall** when importing a routine as well.

The **stdcall** directive *is not* recognized by the Delphi 1 compiler, which poses a challenge when writing a DLL for Delphi 1 and 2. The only way to write a simple DLL source file for both Delphi 1 and 2 is by using conditional compilation (i.e. `$ifdef` and `$endif` statements). The example in Figure 4 demonstrates how the pre-defined symbol, `WIN32`, distinguishes between Delphi 1 (`WIN32` is not defined) and Delphi 2 (`WIN32` is defined).

```
procedure ReadWP(const Filename: string; Doc: TDocument);
  export; {$ifdef WIN32} stdcall; {$endif}
```

Figure 4: Exporting a routine for Delphi 1 and Delphi 2.

Exporting a Class

As we've seen, Delphi makes it easy to use normal functions and procedures stored in a DLL. Classes and objects, however, work slightly differently. This section describes how to export a class from a DLL, and import a class into an application.

In Delphi 1, the first step is to declare a class' methods with the **export** directive. Use the **export** or **stdcall** directive in the class declaration. (Again, when using Delphi 2, you might want to use the **stdcall** directive, to assure the DLL can be called from applications written in other languages.) It isn't necessary to repeat the directive in the method implementation, or for derived classes that override the method.

If you are using conditional compilation to declare the class for Delphi 1 and 2, you don't need to repeat the **export** or **stdcall** directive. The conditional compilation is kept in one location. You can usually keep derived classes and method implementation portable, without conditional compilation.

Figure 5 shows the declaration for the *TISpellChecker* class in the SpellChk unit. This class defines the interface of a rudimentary spelling checker. An application calls the *OpenDictionary* method to load a spelling dictionary. To load multiple dictionaries, the application calls multiple instances of *OpenDictionary*. The *ClearDictionary* method empties the entire dictionary, and the *CheckSpelling* method checks the spelling of a word. If the word is misspelled, the *CheckSpelling* method obtains a list of suggested corrections. These methods are exported from a DLL for use in an application, such as a word processor.

```
unit SpellChk;
...
type
  { Called the suggestion callback for
    suggested corrections. }
  TSuggestProc = procedure(Word: PChar) of object;
    {$ifdef WIN32} stdcall; {$endif}
  TISpellChecker = class
  public
    function OpenDictionary(Path: PChar): Boolean;
      virtual; export;
      {$ifdef WIN32} stdcall; {$endif} abstract;
    procedure ClearDictionary;
      virtual; export;
      {$ifdef WIN32} stdcall; {$endif} abstract;
    function CheckSpelling(Word: PChar;
      Suggest: TSuggestProc): Boolean; virtual; export;
      {$ifdef WIN32} stdcall; {$endif} abstract;
  end;
```

Figure 5: Simple spelling checker interface class.

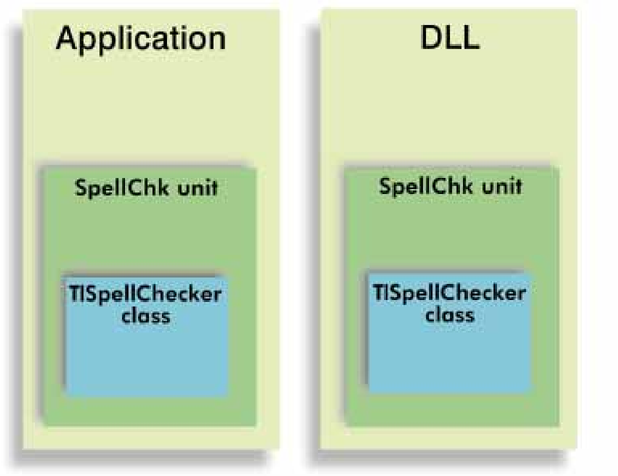


Figure 6: Using the same unit in an application and a DLL.

Note that every method is declared as **virtual** and **abstract**. An abstract method has no implementation, so a derived class must override and implement every abstract method. This must be done for every method you want to export from a DLL. To understand why, consider how classes and units work in Delphi.

The Interface Class

A class must be implemented in the same unit that it is declared. If the *TISpellChecker* class was implemented in the SpellChk unit, then the application would be linked with the full implementation of the spelling checker (see Figure 6). However, the point is to implement the spelling checker in a separate DLL. So, the application needs a way to access the class declaration for *TISpellChecker* without linking its methods. The solution is to declare an *interface class*.

An interface class, such as *TISpellChecker*, is a class with abstract methods. Because an abstract method has no implementation, the SpellChk unit doesn't define any methods for *TISpellChecker*. Both the application and DLL can link with the SpellChk unit to ensure they are using the same class dec-

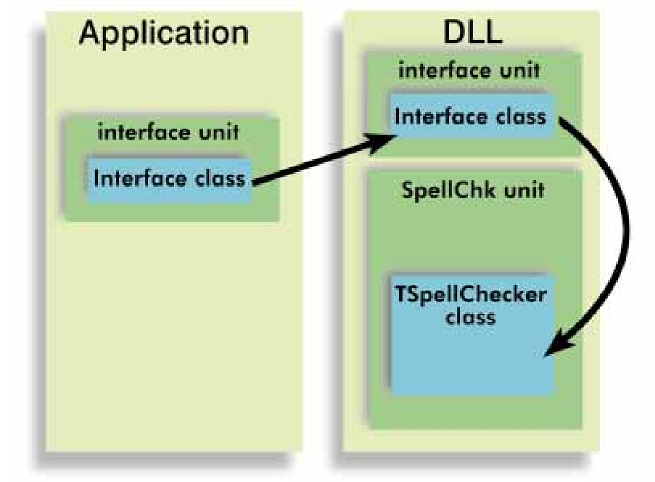


Figure 7: Sharing an interface between an application and a DLL.

laration for the *TISpellChecker* class. The DLL implements a derived class, *TSpellChecker*, overriding all the abstract methods of *TISpellChecker*. Figure 7 illustrates the DLL implementing the spelling checker, where the application and DLL share the abstract interface class.

The derived class (e.g. *TSpellChecker*) is often called a *concrete* class because it provides a concrete implementation of an abstract interface (see Figure 8). Remember: do not repeat the **export** or **stdcall** directives in a derived class.

With the *TISpellChecker* and *TSpellChecker* classes defined, the next step is to create a *TSpellChecker* object in the DLL, and return it to the application. In other situations, you might create an object in the application, and pass it to the DLL as a parameter to a function or procedure. In this case, the DLL exports the *InitSpellChecker* function, which creates and returns a *TSpellChecker* object (see Figure 9). Note that the type of *InitSpellChecker* is the abstract class, *TISpellChecker*. The application isn't aware of the concrete class, *TSpellChecker*, only the abstract class, *TISpellChecker*.

How Exported Methods Work

An application can call the exported method of the DLL because every class has a table of virtual method pointers. This Virtual Method Table (VMT) contains a pointer to the code for every virtual method declared by a class and its base classes. As such, the VMT for the interface class, *TISpellChecker*, contains a pointer for each virtual method declared by the class. When the DLL creates an instance of the concrete class, *TSpellChecker*, it also creates a VMT that points to all exported methods of the class.

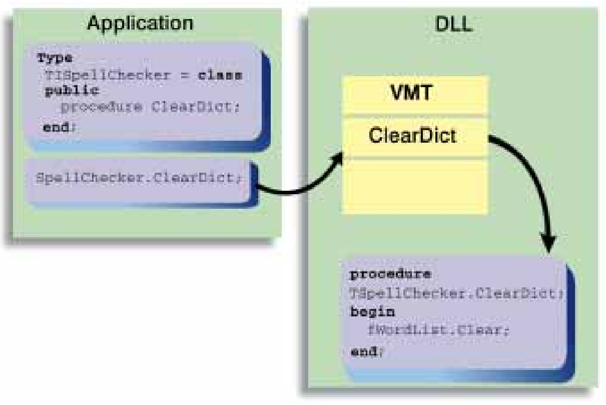
The VMT for *TSpellChecker* originates in the DLL, so its method pointers all point to the methods in the DLL. When the application calls any virtual methods, it knows that it must get the method pointer from the VMT. The application is unaware of whether the VMT comes from the application or DLL. Figure 10 illustrates how an application calls a virtual method.

```
type
  TSpellChecker = class(TISpellChecker)
    ...
  public
    function OpenDictionary(Path: PChar): Boolean;
      override;
    procedure ClearDictionary; override;
    function CheckSpelling(Word: PChar;
      Suggest: TSuggestProc): Boolean; override;
  end;
```

Figure 8: Concrete implementation of the interface spelling checker class.

```
function InitSpellChecker: TISpellChecker; export;
  {$ifdef WIN32} stdcall; {$endif}
begin
  try
    Result := TSpellChecker.Create;
  except
    Result := nil;
  end;
end;

exports InitSpellChecker;
```

Figure 9: *InitSpellChecker* creates and returns a spelling checker object.



```
Type
 TISpellChecker = class
 public
   procedure ClearDict;
 end;
```

```
SpellChecker.ClearDict;
```

```
procedure
TSpellChecker.ClearDict;
begin
  fWordList.Clear;
end;
```

Figure 10: Calling a virtual method.

You can think of the VMT as a means of exporting methods which replaces the table of indexes and names. In the case of the spelling checker, the only routine listed in the **exports** statement is *InitSpellingChecker*. All other exported routines are methods.

Memory Management

Not all the methods of an interface class need to be exported. Sometimes it's useful to declare a static method to take advantage of the fact that such a method is implemented in the application, and isn't called across a module boundary.

The most notable example is the *Free* method, which frees the memory for an object. This method is declared static by the *TObject* class. Every class inherits from *TObject*, therefore every class inherits the *Free* method, which calls the object's destructor, *Destroy*. *Destroy* is virtual, so the application gets the method pointer from the VMT, meaning the DLL's destructor is called from the application.

```
type
  TISpellChecker = class
  public
    procedure Free;
    procedure Release; virtual; export;
      {$ifdef WIN32} stdcall; {$endif}
    ...
  end;

{ Replace TObject.Free to call the exported
  Release method. }
procedure TISpellChecker.Free;
begin
  if Self <> nil then
    Release;
end;

{ Call the destructor from the DLL. }
procedure TISpellChecker.Release;
begin
  Destroy;
end;
```

Figure 11: Exporting the *Release* method for memory management.

The problem in Delphi 1 is that the destructor is not declared with the **export** directive. In Delphi 2, the destructor is not declared with the **stdcall** directive. If you create the object in the DLL and free it from the application, you can run into serious trouble. The solution is to declare an exported method, say, *Release*, that calls the destructor (see Figure 11).

Because *Release* is exported, it can be safely called across a module boundary. Once in the context of the DLL, it's safe to call the destructor. The interface class must then redeclare the *Free* method to call *Release* instead of *Destroy*. With this simple change to the interface class, you can create an object in the DLL and free it in the application, or create it in the application and free it in the DLL. (Note: The *Release* method discussed here is unrelated to the native *TForm Release* method.)

Run-Time Type Information

Using the VMT to export a method makes it easy to export a class in Delphi, but it also introduces complications. If you try to use the **is** or **as** operator, or the *InheritsFrom* method, you'll run into problems. Let's look at what's happening.

A class' VMT is part of its RTTI. The RTTI is a set of tables that describes a class. These tables include the VMT: a table for dynamic methods and message handlers; information about the published properties, methods, and fields; the name of the class; a pointer to the parent class; and other information. Figure 12 shows the organization of a class' RTTI.

Delphi uses the parent class pointer to implement the **is** and **as** operators, as well as the *InheritsFrom* method. Compiling a class reference as a pointer to the class' VMT, Delphi compares VMT pointers to test whether one class is derived from another.

For example, the **is** operator compares an object's VMT pointer to the test class' VMT. If they aren't the same, Delphi

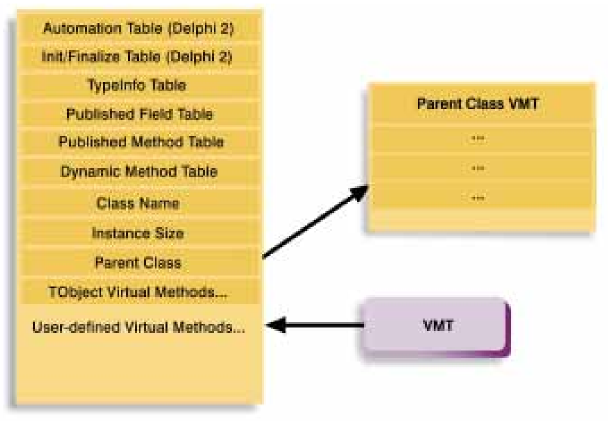


Figure 12: Run-Time Type Information.

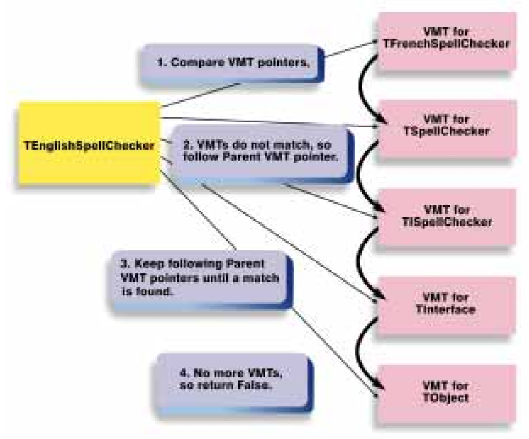

1. Compare VMT pointers.
TEnglishSpellChecker
2. VMTs do not match, so follow Parent VMT pointer.
3. Keep following Parent VMT pointers until a match is found.
4. No more VMTs, so return False.
VMT for TFrenchSpellChecker
VMT for TSpellChecker
VMT for TISpellChecker
VMT for TInterface
VMT for TObject

Figure 13: How the Object Pascal **is** operator works.

follows the base class pointers in the object's VMT, until it finds a match or runs out of base classes to compare — until it reaches *TObject*, which has no base class. If Delphi finds a match for the VMT, then the **is** operator returns *True*, otherwise it returns *False* (see Figure 13).

The problem is that this scheme doesn't work with an object that crosses a module boundary (i.e. is passed as a parameter to a DLL, or returned by a function in a DLL). This is because each module has its own copy of the RTTI for its classes. Delphi compares VMT pointers for exact equality, which can't happen if the VMTs being compared are from different modules (see Figure 14).

Unfortunately, no simple solution exists for this problem. Delphi must assume the classes are different since it has no way of knowing otherwise. Just because two classes have the same name doesn't mean they are identical. You can implement a DLL that accidentally uses the class name, *TISpellChecker*, but with completely different methods. Therefore, Delphi can't determine if two classes are identical based solely on the class name.

The solution is to build any type checking and type casting required by your class into the class itself. For instance, you might expand the spelling checker interface

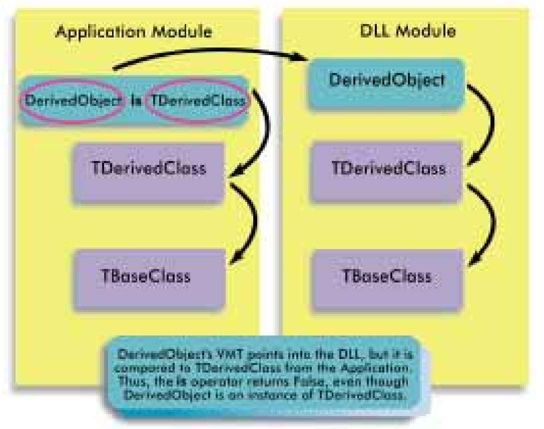

Application Module
DLL Module
DerivedObject is TDerivedClass
DerivedObject
TDerivedClass
TDerivedClass
TBaseClass
TBaseClass
DerivedObject's VMT points into the DLL, but it is compared to TDerivedClass from the Application. Thus, the is operator returns False, even though DerivedObject is an instance of TDerivedClass.

Figure 14: Objects and RTTI crossing a module boundary.

to include different classes for different languages. Each class implements the rules for plurals, verb conjugations, and so on. Every class uses the same interface, but with different implementations. To provide a common interface for the word processor, all classes inherit from the same base class, *TISpellChecker*. To test whether a spelling checker object is an instance of *TEnglishSpellChecker*, the *TISpellChecker* class declares virtual methods that return this information.

Figure 15 shows the new class declarations and virtual methods that can be used instead of the **is** operator. To test whether a spelling checker object is an instance of *TEnglishSpellChecker*, call the *IsEnglishSpellChecker* method, which is implemented by each concrete spelling checker class.

```
type
  TISpellChecker = class
  public
    function IsEnglishSpellChecker: Boolean;
      virtual; export;
      {$ifdef WIN32} stdcall; {$endif} abstract;
    function IsFrenchSpellChecker: Boolean;
      virtual; export;
      {$ifdef WIN32} stdcall; {$endif} abstract;
  end;

{ In the DLL ... }
type
  TEnglishSpellChecker = class(TISpellChecker)
  public
    function IsEnglishSpellChecker: Boolean; override;
    function IsFrenchSpellChecker: Boolean; override;
  end;

function TEnglishSpellChecker.IsEnglishSpellChecker:
  Boolean;
begin
  Result := True
end;

function TEnglishSpellChecker.IsFrenchSpellChecker: Boolean;
begin
  Result := False
end;
```

Figure 15: Virtual methods for testing the type of an exported object.

Version Control

Version control presents a similar problem. Imagine what happens if there is a new version of the application with a different interface for the *TISpellChecker* class. Suppose a problem arises when installing the new DLL. In this case, the VMT for the *TISpellChecker* class in the DLL doesn't match the VMT in the application.

As another example, let's say that adding support for multiple languages required the addition of several new methods. The application and DLL must have compatible VMTs. Otherwise, the application can easily call the wrong method (see Figure 16).

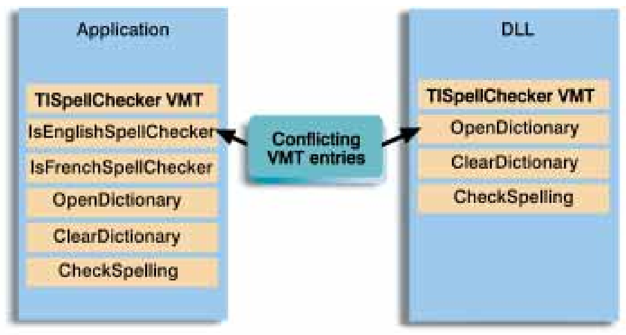


Figure 16: Conflicting VMTs because of different versions of *TISpellChecker*.

To reiterate, Delphi has no way of knowing if the two versions of *TISpellChecker* have identical interfaces by looking at the class names. It is your responsibility to ensure the application and DLL are using the same version of the *TISpellChecker* class.

To solve this problem, define a *GetVersion* method to return the current version of *TISpellChecker*. A constant, *ExpectedVersion*, records the version the application expects. When you change the interface of *TISpellChecker*, it is also necessary to change the expected version number. When the application obtains a *TISpellChecker* object from the DLL, it checks the version numbers, and if they don't agree, the application reports the appropriate error message (see Figure 17).

TInterface Class

To make your job a little easier, Delphi defines the *TInterface* class in the VirtIntf unit. This class declares the *Release* and *GetVersion* methods, as described in this article.

Although the *TInterface* class is undocumented, every version of Delphi 1 and 2 has the source code for the VirtIntf unit. However, each version's source file is in a different directory. Try looking in \DOC, \SOURCE\VCL, or \SOURCE\TOOLSAPI. Your interface class must imple-

```
const
  ExpectedVersion = 2;
type
  TISpellChecker = class
  public
    function GetVersion: Integer; virtual;
      export; {$ifdef WIN32} stdcall; {$endif} abstract;
    ...
  end;

function GetSpellChecker: TISpellChecker;
begin
  { Get the spelling checker object from the DLL. }
  Result := InitSpellChecker;
  { Check the version number. }
  if Result.GetVersion <> ExpectedVersion then
    raise Exception.Create(
      'Incorrect version of SPELCHEK.DLL');
end;
```

Figure 17: Implementing version control.

ment *ExpectedVersion* as a constant, so it is not part of the *TInterface* declaration.

Conclusion

DLLs can be extremely useful in Delphi programs. And as we've seen, you can even export a class from a DLL by following a few simple rules. You must first define an interface class, declaring all exported methods as virtual, abstract methods. In Delphi 1, you must supply the **export** directive; in Delphi 2, you might want to use **stdcall**. Derive the interface class from Delphi's *TInterface* class to automatically get version checking and memory management (i.e. so objects can be created in one module and freed in another). Type checking and type casting are not generally permissible with exported objects, but you can write additional methods to support this functionality, even if in a slightly restricted form. ∆

The demonstration project referenced in this article is available on the Delphi Informant Works CD located in INFORM\96\OCT\DI9610RL.

Ray Lischner is the founder and president of Tempest Software (http://www.tempest-sw.com), which specializes in object-oriented components and tools. Mr Lischner has been a software developer for over a decade, at large and small firms across the United States. He has recently finished *Secrets of Delphi 2*, a book that reveals undocumented aspects of Delphi, to be published by the Waite Group Press this autumn. You can reach Mr Lischner at delphi@tempest-sw.com.

Delphi 1 / Object Pascal

By Keith Wood

Saved by an Expert

An Expert to Generate a Screen Saver Application

Delphi 1 provides many ways for developers to reuse code with minimal effort. These include the components we drop on our forms and the other objects that Delphi defines. In addition, Delphi 1 provides us with the Gallery, a container of templates and experts for forms and projects.

Templates allow us to copy static code that we have previously developed, while experts allow us to interact with the code generation to customize it for our needs. In this article, we'll develop a Delphi expert to generate a project framework. To make this worthwhile, we need an application that's interesting, hopefully useful, and not too trivial. A screen saver seems to fit the bill nicely — it's a fairly simple application with code that is common (indeed required) across all screen savers, and features several extras that we can have our expert automatically include or exclude.

Screen Saver Basics

Screen savers serve two functions on a computer system:

1) They are used to protect the monitor from phosphor burn that can be caused by leaving an image on the screen too long. This is not really a problem anymore with newer monitors.
2) They can be used to hide and possibly protect information on the screen from other people when the system is unattended.

all right — maybe there's a third function:

3) Entertainment! For example, some screen savers play vignettes of a longer story, or show selections from collections of graphics or cartoons.

A screen saver is really just an ordinary program that has been renamed with an .SCR extension instead of the normal .EXE. It's expected to interact with the Windows Desktop through the Control Panel, and display its configuration or saver screen when requested. It should also define a title appearing in the list of available screen savers.

From the Command Line

The program determines which mode it should start in — configuration or display — based on command line parameters passed to it. For example, if a `/S` switch is present, the screen saver opens its display window to cover the entire screen and begins drawing on it. The screen saver must then monitor the system for any user activity, via the mouse or keyboard, which then causes it to close itself.

Conversely, if a switch isn't present, the program opens its configuration form. This allows the user to control the display screen's appearance. These parameters can be stored in an .INI file for use when the display screen appears. Typically the form features a **Test** button that the user can press to view the effects of the configuration selections.

Some screen savers allow users to specify a password to prevent unauthorized users from halting it and returning to the original programs. To implement this, two additional forms are required. One to allow the password to be changed, and another to ask for the password when the saver is running. An option on the configuration screen allows the user to enable or disable this feature, and has a button that allows the user to change the password.

The actual drawing of the display screen can show anything imaginable. One possibility is to manipulate the original screen image, causing it to dissolve, re-arrange, or be "eaten" by something. To do this, we need to capture an image of the screen just before the screen saver starts, and use the image as the starting point for our activities.

Sounds fairly simple doesn't it? We could just write one screen saver and then copy the code when we want to write another, adding or deleting the bits that we need in the new version.

The workable solution is to create a screen saver expert. We'll then install the expert into the Delphi environment, allowing us to quickly generate a generic screen saver and add code to customize it.

Forms: First Things First

The application should create the configuration and display forms automatically. In configuration mode the user will probably want to test the saver with the new parameters, requiring the display screen to be available. While in display mode, we must also create the configuration form to provide access to the user parameters.

Regardless of the mode we're in, the configuration form provides a single point for loading the parameters. Normally, each form that is auto-created is constructed in the code of the project source, with the main form as the first to appear. This form is made visible while the others are hidden. However, we want different forms to be the main form in each of the screen saver's modes.

To accommodate the different ways of invoking the screen saver we must modify the code in the project source. We want:

- the display form to be the main form when the program is invoked with a `/S` parameter, and
- the configuration form to be the main form at all other times.

Therefore, depending on the mode, we must alter the order of creation of these two forms.

The code in Figure 1 shows this form selection based on the mode. It also includes a check to ensure that only one copy of the screen saver program is running. The *hPrevInst* variable contains the handle of the previous instance of this program. If it's zero, then there is no previous instance and we can continue. If it's non-zero, the program takes no further action and terminates.

The screen saver's title is also encoded in the project source. The `{$D}` compiler directive is used to insert the name into the executable. It must appear with an identifier of `SCRNSAVE`

```
begin
  { Only one instance is allowed at a time }
  if hPrevInst = 0 then
    begin
      { Display }
      if (ParamCount > 0) and
         (UpperCase(ParamStr(1)) = '/S') then
        begin
          { fmDisplay needs to be the main form }
          Application.CreateForm(TfmDisplay, fmDisplay);
          Application.CreateForm(TfmConfiguration,
                                 fmConfiguration);
        end
      else
        { Configure }
        begin
          { fmConfiguration needs to be the main form }
          Application.CreateForm(TfmConfiguration,
                                 fmConfiguration);
          Application.CreateForm(TfmDisplay, fmDisplay);
        end;
      Application.Run;
    end;
end;
```

Figure 1: This project source code allows the screen saver to swap main forms based on a command-line parameter.

and can only be present in the project source itself — not one of the units. Only one of these entries can exist, per project.

Here's a screen saver name example:

```
{ Description that appears in drop-down list }
{$D SCRNSAVE Dropping Off}
```

The configuration form loads the screen saver parameters from their location — in our case from an .INI file — and displays them onscreen. These can be updated and saved back to the file for later use. The parameters themselves depend on the screen saver being implemented and cannot be automatically generated.

The form contains three standard buttons:
1) OK saves the parameters and closes the form,
2) Cancel closes the form without saving, and
3) Test makes the display form visible.

If the screen saver is password protected, the form also contains a check box controlling whether to ask for the password, and a button allowing the user to enter a password.

The display form is a borderless window that maximizes on creation and stays on top of all other windows. It contains a single Timer component that allows for the periodic updating of the display for the screen saver. This Timer is enabled when the form is shown, and disabled when it's hidden.

To allow the form to monitor all user activity, we create a customized message handler and direct all messages for the form to it using:

```
Application.OnMessage := DeactivateScreenSaver;
```

Within this message handler, we check if the message relates to a mouse movement, mouse click, or key press. If one of these is

```
{ Check for any user activity and stop when found }
procedure TfmDisplay.DeactivateScreenSaver(
  var msg: TMsg; var bHandled: Boolean);
var
  bDone: Boolean;
begin
  StopMonitoring;  { Don't monitor further messages }

  { Check for largish mouse movement }
  if msg.Message = WM_MOUSEMOVE then
    bHandled := (Abs(LOWORD(msg.lParam) - ptOrig.x) > 5) or
                (Abs(HIWORD(msg.lParam) - ptOrig.y) > 5)
  else
    { Or for key presses or mouse clicks }
    bHandled := (msg.Message = WM_KEYDOWN)      or
                (msg.Message = WM_SYSKEYDOWN)   or
                (msg.Message = WM_ACTIVATE)     or
                (msg.Message = WM_NCACTIVATE)   or
                (msg.Message = WM_ACTIVATEAPP)  or
                (msg.Message = WM_LBUTTONDOWN)  or
                (msg.Message = WM_RBUTTONDOWN)  or
                (msg.Message = WM_MBUTTONDOWN);

  bDone := bHandled;
  if bDone then
    begin
      { Check whether screen is password controlled }
      if fmConfiguration.cbxPassword.Checked then
        begin
          fmPassword := TfmPassword.Create(self);
          try
            if fmPassword.ShowModal = mrOK then
              bDone := (fmPassword.edPassword.Text =
                fmConfiguration.sPassword)
            else
              bDone := False;
          finally
            fmPassword.Free;
          end;
        end;

         if bDone then
        { Password OK or no password }
        Close;
    end;

  StartMonitoring;  { Monitor all messages }
end;
```

Figure 2: The message handler for the display screen.

detected, the user has returned and the screen saver is terminated. Figure 2 shows the complete message handler code.

The message type is available through the handler's *Message* property. Small mouse movements — less than five pixels — are ignored, as it's assumed that these are not intentional. The *bHandled* parameter is set to indicate whether this handler responded to the message sent. Note that the call to *StopMonitoring* at the beginning deactivates this message handler for the duration of its execution.

And the Password Is ...

If the screen saver is password protected, we must ask for the password when we detect the user's return. This is done by the *fmPassword* form that is created and displayed in response to any user activity. If the password entered matches that held in the configuration form, then the screen saver terminates as before. Otherwise the message handler is reactivated and the screen saver continues.

The change password and password request forms are straightforward. The former implements several checks to ensure that the password is only altered by an authorized person and that the new password is confirmed and different from the original. The latter simply asks the user to enter a password.

All the edit fields on these forms have their *PasswordChar* property set to *. This causes the characters entered to be displayed as this character, hiding the actual values. Additionally, the edit fields have their *MaxLength* property set to 8 to limit the password's length.

Note that in the password request form, all key strokes are previewed by the form, through setting its *KeyPreview* property to *True* and adding a method to the *OnKeyDown* event. This method discards the Alt Tab and Alt Esc sequences since we don't want the user to switch to another program while the screen saver runs.

Capturing the Image

To have the screen saver manipulate the screen image, we need to copy and draw it on the display form. Delphi provides no direct way of accessing the screen image, so we must resort to the appropriate Windows API calls to achieve this.

We call the *GetDC* function to obtain a handle to the screen image. Normally we would pass the handle of a window to *GetDC* and then receive a handle to the device context (usually encapsulated by Delphi's *TCanvas*) of its client area. If, instead, we pass a handle of zero, we receive a handle to the device context of the screen itself. This can then be copied to a bitmap for later use. Figure 3 shows the Object Pascal required to capture the screen image.

The timing of this activity is critical. We must capture the screen before the screen saver is made visible or we'll only get an image of ourselves. The place to capture the screen, therefore, is in the *OnCreate* event of the form. Unfortunately, at this stage the form itself has not yet been created and thus cannot be referenced. This means that we cannot transfer the screen image directly to the *Canvas* of the form.

For this reason, we create a bitmap to hold the image and copy it there instead. This is then transferred to the form the first time that its *OnPaint* event is invoked using the canvas' *Draw* method. This also means that the screen image is available throughout the screen saver's execution. Of course, both the device context and the bitmap must be freed (when possible) to return their resources to Windows.

Now that we have all the pieces necessary to build a generic screen saver, we can incorporate them into an expert to automate the building process.

Delphi Experts

Delphi experts are a mysterious breed. Online Help contains entries that describe how to use them, but it does not con-

```
{ Capture screen to bitmap. Must do this before
  the form shows! }
procedure TfmDisplay.FormCreate(Sender: TObject);
var
  h: HDC;
begin
  bFirst           := True;
  bmpScreen        := TBitmap.Create;
  bmpScreen.Height := Screen.Height;
  bmpScreen.Width  := Screen.Width;
  { Get handle to device context for the screen }
  h := GetDC(0);
  { And copy to internal bitmap }
  try
    BitBlt(bmpScreen.Canvas.Handle, 0, 0,
           Screen.Width, Screen.Height,
           h, 0, 0, SRCCOPY);
  finally
    { Must release device context back to Windows }
    ReleaseDC(Handle, h);
  end;
end;
```

Figure 3: Capturing the screen image for the display form.

tain information on how to develop a new one. Fortunately, the source code for the Application and Dialog experts is available in DELPHI\DEMOS\EXPERTS under the ExptDemo project. From these we can divine how they work and implement one of our own.

First, we find that an expert is implemented as a DLL with certain pre-defined entry points. So all we must do is follow this formula and generate our own expert in a DLL.

From examining the ExptDemo project, we see an expert is a class derived from *TIExpert*, which defines a number of methods that must be overridden. These return various pieces of information about the expert, including its name and description, an image to represent it in the Gallery, whether it's a form, project, or standard expert, and its current state.

An id string must also be defined. This consists of a two-part name: the first part typically identifies the vendor or author, and the latter identifies the expert itself. Together these should be unique within Windows. Finally, there is the *Execute* method that actually runs the expert when requested. Further comments on all of this are available in the ExptIntf unit (located in your DELPHI\DOC directory).

For the expert to be correctly linked into Delphi's IDE, it must be registered. This is achieved by the *InitExpert* function that is exported from the DLL. *InitExpert* is called by Delphi while the IDE loads. After some initialization, *InitExpert* makes calls to *RegisterProc* (it's passed in as a parameter) with an instance of each expert to register.

Copying this project and modifying the various pieces to implement our own expert results in the ScrSavEx project file that accompanies this article.

The application expert (the one we're copying) is actually implemented in the APP.PAS file. The **interface** procedure, the one invoked by the *Execute* method of the expert class, displays the expert's form and requests input from the user. Once this has been completed, it performs some initialization, generates the source and .DFM files required by the application, and opens the completed project, allowing the programmer to further customize it.

The form displayed to the user employs a notebook component to hold the different pages of information being requested. The user can navigate these pages using the **Next** and **Prev** buttons. An image on the form provides visual feedback on the purpose of the current page, and can be used to indicate the effects of various options on that page (see the Dialog Expert for an example of this).

On the final page, the **Next** button becomes a **Create** button. When it's pressed, **Create** generates the project. The user can also press the **Cancel** button to terminate the process at any time. To allow our expert to better integrate with Delphi, we should follow this layout and behavior.

Our expert uses only two pages. The first requests the screen saver's title and the interval for the Timer, and provides options to capture the screen image at startup and/or to password protect the screen saver. The second page requests the name of the project for this screen saver and the directory in which to store it.

After the user enters the appropriate details, the expert generates the requested code. First it needs to load the snippets of code that comprise the project. These are held in a string resource file in the example experts and are read into an array of PChars by the *InitCodeGeneration* procedure.

Resource files allow an application's various components to be separately maintained and easily altered (e.g. when changing to another language). The string resource can be changed using Borland's Resource Workshop, or by compiling resource source files using Borland's Resource Compiler. The example experts take the latter approach. This is invoked through the BUILD.BAT batch file.

Bitmaps that the experts can display are also stored in a resource file, and can be altered using Delphi's Image Editor (you can also use Borland's Resource Workshop to alter bitmaps). Both of these resources are included in the final project with the `{$R}` compiler directive in the project source file:

```
{ Bitmaps for the expert }
{$R SCRSAVBM.RES}

{ String resource for the expert }
{$R SCRSAVST.RES}
```

The string resource containing the code snippets must be located and made available for use within the program. This is done by the *LoadResource* and *FindResource* Windows API functions, which copy the named resource into memory and produce a handle to it. The resource is then locked to prevent it from being overwritten in memory, and a pointer to it is returned so that we can access it directly.

```
{ Generate the source for the project file,
  with the specified name }
function GenerateProjectSource(
  fmExpert: TfmScrSavExpert): TFileName;
var
  stmSourceFile: TFileStream;
begin
  { Create the full path and name for the project file }
  Result := fmExpert.edPath.Text;
  if (Result > EmptyStr) and
     not Result[Length(Result)] in [':', '\'] then
    Result := Result + '\';
  Result := Result + fmExpert.edName.Text + sProjectExt;

  { Check whether file can be created }
  CheckFileOverwrite(Result);

  { Create the file and write the code to it }
  stmSourceFile := TFileStream.Create(Result, fmCreate);
  try
    WriteSnippetFormat(stmSourceFile, csProjectBegin,
                       [fmExpert.edName.Text]);
    if fmExpert.cbxPassword.Checked then
      WriteSnippetFormat(stmSourceFile, csProjectPwd,
                         [LoadStr(iPasswordFile),
                          LoadStr(iChgPswdFile)]);
    WriteSnippetFormat(stmSourceFile, csProjectEnd,
                       [LoadStr(iDisplayFile),
                        LoadStr(iConfigFile),
                        fmExpert.edTitle.Text]);
  finally
    stmSourceFile.Free;
  end;
end;
```

Figure 4: The function to generate the project source for the screen saver and return the file name to the caller.

The code snippets consist of text blocks that are each terminated by a vertical bar (|). This delimiter is searched for within the resource, with each section being pointed to by the next position in the array. The delimiter is changed to a null character so the array elements appear to be PChars. Note that the **for** loop controlling this assignment uses the *Low* and *High* functions to avoid hard-coding the first and last values in the *TCodeSnippet* type.

The code for use in generating the basic screen saver can be taken from an actual screen saver project. Code from each of the units is combined into a single text file with vertical bars terminating the different sections, including the optional parts. During the generation process, placeholders act as substitutions for variable values. This is done using the *StrLFmt* function wherein string positions are marked by `%s` and integer values by `%d`.

Loaded with Code

With the code snippets loaded, the code can now be generated. First we produce the project source by constructing its name from the details (provided by the user) and then writing the necessary code from the code resource into that file, substituting values where appropriate (see Figure 4). Then each of the program units follows, generating the Object Pascal source and the corresponding .DFM file.

```
{ Generate the .DFM file for the screen display form }
procedure GenerateDisplayFormFile(
  fmExpert: TfmScrSavExpert);
var
  sFormName:   TFileName;
  stmFormFile: TFileStream;
  stmTextFile: TMemoryStream;
begin
  { Create the full path and name for the screen
    display form resource file }
  sFormName := GetFileName(fmExpert.edPath.Text,
                           iDisplayFile, sResourceExt);

  { Check whether file can be created }
  CheckFileOverwrite(sFormName);

  { Create text version of file and write object
    descriptions to it }
  stmTextFile := TMemoryStream.Create;
  try
    WriteSnippet(stmTextFile, csDisplayForm);
    if fmExpert.cbxCapture.Checked then
      WriteSnippet(stmTextFile, csDisplayFormCapt);
    WriteSnippetFormat(stmTextFile, csDisplayFormEnd,
                       [fmExpert.spnInterval.Value]);
    { Return to the beginning of the file for conversion }
    stmTextFile.Position := 0;
    { Create resource version of file and convert
      from text version }
    stmFormFile := TFileStream.Create(sFormName, fmCreate);
    try
      ObjectTextToResource(stmTextFile, stmFormFile);
    finally
      stmFormFile.Free;
    end;
  finally
    stmTextFile.Free;
  end;
end;
```

Figure 5: The procedure to generate the .DFM file for the display form.

To generate the .DFM files, we capitalize on the fact that they can be represented as a "straight" ASCII text file. To see this we can select **File | Open File** from the menu, change the file type to .DFM, and load any form file. Delphi then presents us with a list of all the objects comprising the form with those properties that do not have default values being set. Objects that are contained within another on the form appear within the definition of that object. This file can be altered as text and is then converted back into its normal binary format by Delphi.

We can do the same with the *ObjectTextToResource* procedure. It takes two parameters, being streams that connect to the text source for the file and another for the binary output. Figure 5 is the Object Pascal code that generates the .DFM file for the display form.

The code snippets to construct the .DFM files are included in the string resource as described earlier. Each is opened in its text format, and cut and pasted into the combined text field of all snippets.

For all files being generated, we check that they do not already exist, and ask permission to overwrite them if they do. If permission is refused, the expert terminates without completing the new project. Because we have locked the string resource into memory, we must remember to unlock it and free the space it was using at the end of the process. This is done in the *DoneCodeGeneration* procedure.

Open the Project

Finally, after all the code is generated in its appropriate format, we tell Delphi to open the new project with the *OpenProject* method of the *ToolServices* object. This loads in the nominated project and opens its main form for editing (in our case the display form, since this appears first in the project source). We also open the configuration form directly with *OpenFile* since both these forms are typically amended by the programmer (more information on the *ToolServices* object is available in the ToolIntf unit in the DELPHI\DOC directory). The programmer can now customize the project knowing that the pieces required for the screen saver to correctly interface with Windows are already in place.

Before it can be used, the expert must be compiled into a DLL. This assumes that the string resource has already been compiled from its own source files. However, searching through the IDE reveals no way of adding our expert to the Gallery. Templates can easily be added from the Gallery itself by clicking on the **Add** button. However, this is disabled when we move to the Experts tab.

Some further searching provides the answer. The experts are notified to Delphi through its initialization file, DELPHI.INI. DELPHI.INI has a section called Experts, with a list of the DLLs that contain them. To install another one, simply add its name to the list, for example:

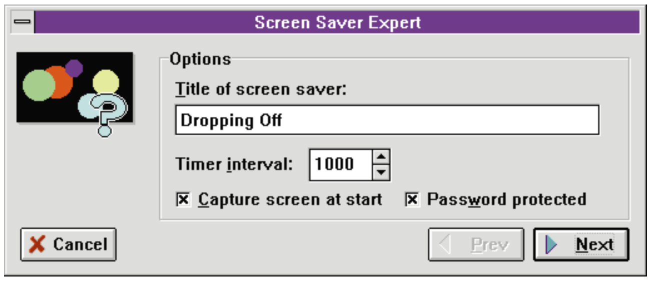
```
[Experts]
ExptDemo=C:\DELPHI\BIN\EXPTDEMO.DLL
ScrSavEx=C:\DELPHI\EXPERT\SCRSAVEX.DLL
```

Delphi must then be re-started to acknowledge these changes. Note that the expert must be unloaded — by removing its entry and re-starting Delphi — before it can be re-compiled following alterations.

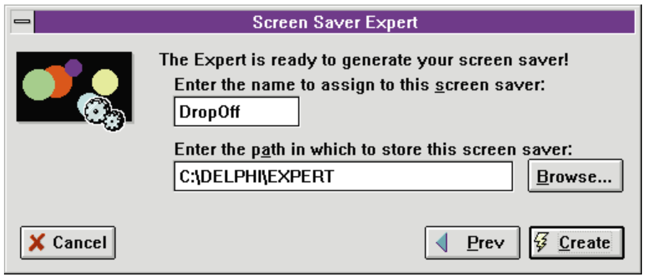
Generating the Screen Saver

Now we'll use the Screen Saver Expert to help us write our screen saver. It's named Dropping Off and appears to shave off sections of the screen and drop them down and off the screen.

From Delphi's menu, select **Options | Environment** and click on the Preferences Tab. In the **Gallery** group, make sure the **Use on New Project** option is checked. Click **OK** when you've verified this. Now, from Delphi's menu, select **File | New Project** to display the Project Gallery. Click on the Experts tab and double-click on the Screen Saver Expert. On the Options screen, check the **Capture screen at start** and **Password protected** check boxes. Leave the **Timer interval** at its default value and enter the title of the screen saver (see Figure 6).



Title of screen saver: Dropping Off
Timer interval: 1000
☒ Capture screen at start ☒ Password protected



The Expert is ready to generate your screen saver!
Enter the name to assign to this screen saver: DropOff
Enter the path in which to store this screen saver: C:\DELPHI\EXPERT

Figure 6 (Top): The Options screen from the Screen Saver Expert. **Figure 7 (Bottom):** The Create screen from the Screen Saver Expert.

Click the **Next** button to advance to the second page. Enter the name of the project, `DropOff`, and select a directory in which to store it (see Figure 7). Press the **Create** button and watch the code appear.

On the configuration form add two spin edit controls along with their labels. These control the size of the sections dropped and their speed down the screen. Set their properties appropriately: range 1 to 10, initial values 4 and 10 respectively. Change the labels to meaningful prompts and provide for accelerator keys if required. In the code, update the *LoadConfig* and *SaveConfig* procedures to read and write the values for these parameters. Figure 8 is the code for the resulting *LoadConfig* procedure.

```
{ Load configuration parameters from .INI file }
procedure TfmConfiguration.LoadConfig;
var
  inifile : TInifile;
begin
  inifile := TInifile.Create(sConfigFile);
  try
    with inifile do begin
      spnSize.Value  :=
        ReadInteger(sIniSection,sIniSize,4);
      spnSpeed.Value :=
        ReadInteger(sIniSection, sIniSpeed, 10);
      cbxPassword.Checked :=
        ReadBool(sIniSection, sIniProtected, False);
      sPassword :=
        ReadString(sIniSection, sIniPassword, '');
    end;
  finally
    inifile.Free;
  end;
end;
```

Figure 8: The *LoadConfig* procedure after being amended. Note that all but two lines are generated by the expert.

```
{ Enable timer and start screen saver }
procedure TfmDisplay.FormShow(Sender: TObject);
begin
  iCurLine := Screen.Height;
  iCurPos  := iCurLine;
  tmrDraw.Interval :=
    500 - fmConfiguration.spnSpeed.Value * 45;
  tmrDraw.Enabled := True;
  StartMonitoring;
end;
```

Figure 9: The *FormShow* procedure after being amended. About half of this is generated by the expert.

```
{ Each timer tick draw the next dropped line }
procedure TfmDisplay.tmrDrawTimer(Sender: TObject);
var
  iSize: Integer;
begin
  iSize := fmConfiguration.spnSize.Value;
  with Canvas do begin
    { Drop the current line one position }
    CopyRect(Bounds(0, iCurPos + iSize,
            Screen.Width, iSize), Canvas,
            Bounds(0, iCurPos, Screen.Width, iSize));
    FillRect(Bounds(0, iCurPos, Screen.Width, iSize));
    Refresh;
    Inc(iCurPos, iSize);

    { If at the bottom, go to the next line to drop }
    if iCurPos > Screen.Height then
      begin
        Dec(iCurLine, iSize);
        { If at the top, start again }
        if iCurLine < 0 then
          begin
            iCurLine := Screen.Height;
            Draw(0, 0, bmpScreen);
          end;
        iCurPos := iCurLine;
      end;
  end;
end;
```

Figure 10: The *tmrDrawTimer* procedure moves the screen sections down and off the screen. The expert generates only the skeleton of this procedure.

In the code for the display form, we add the functionality described above: to drop sections of the screen image down and off the screen. To start, add two variables to the **private** section of the form. These record the current section being dropped and its position on the screen. Update the *FormShow* code to initialize these variables (see Figure 9). To alter the speed of the section's movement we set the Timer's *Interval* property from the configuration's speed parameter (a larger value produces a smaller interval). All this is done in the *FormShow* event so that it is re-initialized each time the screen is displayed. This may happen several times during configuration, with different parameters each time.

Finally, add code to the *Timer* event to actually draw the screen. At each tick of the Timer, we move the current section one spot further down the screen. When it disappears we start on the next section, and when they have all gone we start the entire process again (see Figure 10). We use the configuration's size parameter to determine how much of the screen to move at any one time. The *Brush* of the form's *Canvas* is initialized in the *FormCreate* method to be solid black, and is used in the *FillRect* routine to blank out the section that we have just moved.

After the project is compiled it can be incorporated into Windows along with the other screen savers. Copy the executable to the directory containing your screen savers (typically \WINDOWS\SYSTEM) and then rename it to have an extension of .SCR. Now open the Desktop from within the Control Panel and our new screen saver should appear in the list. Select it and press **Setup** to display the configuration form. Change the parameters if required and test it.

Conclusion

One of Delphi's key features is its ability to reuse code. Using or inheriting from components is one way to achieve this, as is the use of templates and experts. Templates provide static code, while experts allow generation of customized code from a standard base.

We have built a project expert — without too much difficulty — for a useful application, and have incorporated it into the Delphi environment. It ensures that required values are not omitted, and allows us to easily include or exclude sections of code as we see fit. It removes the repetitive part of the code generation and allows us to concentrate on the main purpose of the application. Along the way we've constructed a solid base for building custom screen savers in Delphi.

I'd like to thank Mark Johnson for his screen saver article in the July 1995 issue of the *Delphi Connection* (available on the Internet at http://www.pennant.com/delphi.html). The code generated by the screen saver expert is based on the code from this source. ∆

The demonstration files referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\OCT\DI9610KW.

Keith Wood is an analyst/programmer with CSC Australia, based in Canberra. He started using Borland's products with Turbo Pascal on a CP/M machine. You can reach him via e-mail at kwood@netinfo.com.au or by phone (Australia) 6 291 8070.

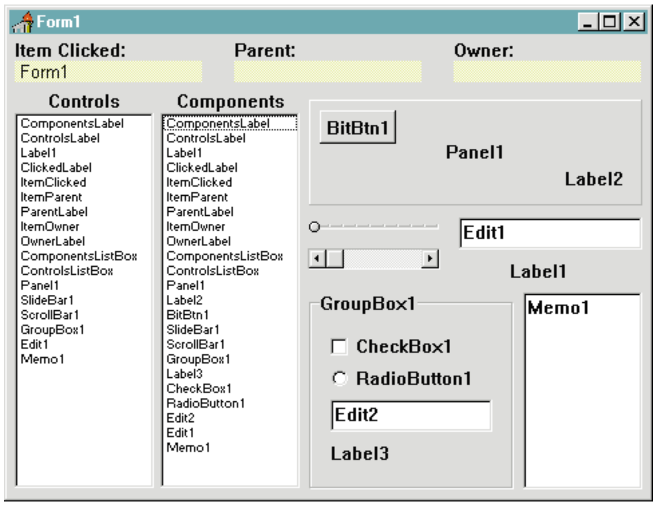
By Robert Vivrette

Parentage and Ownership

Understanding the Controls and Components Arrays

Like any programming language, Object Pascal has a few features that are obscure and not very well documented. One such feature is Delphi's use of the Controls and Components arrays. These arrays are internal list structures used to keep track of components and their relation to one another. This article clarifies their function and use in Delphi programs.

First, let's take a brief look at these arrays. The Components array is a *TList* structure that is a part of *TComponent* and all of its descendants. Similarly, the Controls array is a *TList* structure, and a part of *TWinControl* and its descendants. Therefore all descendants of *TComponent* will have a *Components* property, and all descendants of *TWinControl* will have a *Controls* property. Because most visual controls in the Delphi VCL are descendants of *TWinControl*, you'll find most have both properties.



Item Clicked: Form1
Parent:
Owner:

Controls:
ComponentsLabel
ControlsLabel
Label1
ClickedLabel
ItemClicked
ItemParent
ParentLabel
ItemOwner
OwnerLabel
ComponentsListBox
ControlsListBox
Panel1
SlideBar1
ScrollBar1
GroupBox1
Edit1
Memo1

Components:
ComponentsLabel
ControlsLabel
Label1
ClickedLabel
ItemClicked
ItemParent
ParentLabel
ItemOwner
OwnerLabel
ComponentsListBox
ControlsListBox
Panel1
Label2
BitBtn1
SlideBar1
ScrollBar1
GroupBox1
Label3
CheckBox1
RadioButton1
Edit2
Edit1
Memo1

Figure 1: This application illustrates the relationship between the Controls and Components arrays.

Show ...

As they say, a picture is worth a thousand words. Here then, is a simple application that illustrates the relationship between the Controls and Components arrays (see Figure 1).

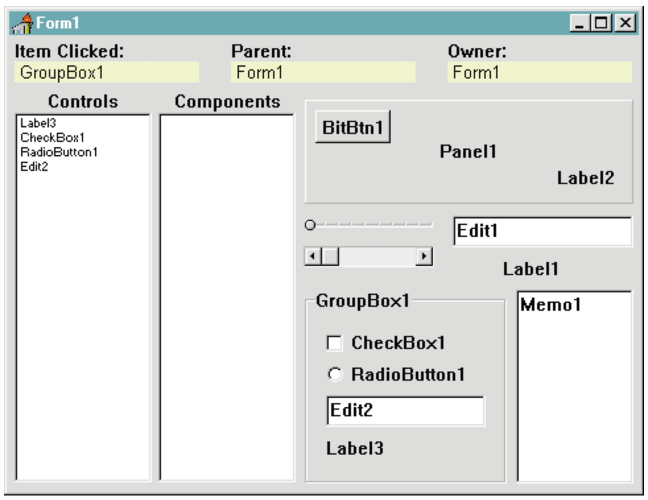
By clicking on an empty area of the main form, this sample application shows which items are in the form's Controls array and which ones are in its Components array. As you can see, the Components array holds all items that are physically on the form, while the Controls array holds only those items that are immediate children of the form.

If you look carefully, you'll notice the button and label that are on Panel1 are not listed in the Controls array. In addition, the Checkbox, RadioButton, EditBox, and Label in GroupBox1 are also not listed in the Controls array. This is because the form is not the parent of each of these items. Rather, their respective containers (Panel1 and GroupBox1) are.

Two concepts — *parentage* and *ownership* — might make things simpler to understand.

... and Tell

A component's parent is its holder, or container. For example, CheckBox1 is a child of


Form1

Item Clicked: GroupBox1
Parent: Form1
Owner: Form1

Controls:
Label3
CheckBox1
RadioButton1
Edit2

Components:

Figure 2: The result of clicking on the frame of GroupBox1.

GroupBox1; conversely, GroupBox1 is therefore the parent of CheckBox1.

One primary function of a component's parent is to define that component's coordinate system. If we were to examine CheckBox1's coordinates, we would see that its *Left* coordinate is 16 and its *Top* coordinate is 32. CheckBox1 is obviously not at 16,32 on the form; rather, these coordinates are relative to its parent, GroupBox1. A component's parent also controls other drawing-related steps, such as the stacking order, tab order, and clipping.

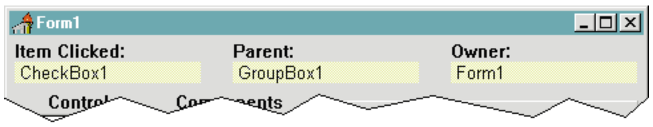
A component's *owner* is the object ultimately responsible for the component's creation and destruction. When a component is destroyed, all other components that list the first component as their owner will be destroyed as well. This simplifies the programmer's job — you don't need to concern yourself with the destruction of every component on a form.

Back to Controls and Components

How does this relate to the Controls and Components arrays? Basically, the Controls array manages *parentage* and the Components array manages *ownership*.

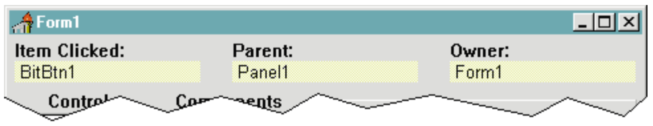
Let's use another example to illustrate. Look at what happens when we click on the frame of GroupBox1 (see Figure 2). As expected, the *Controls* property for GroupBox1 shows only its children. However, note that the Components array is empty. This is because Label3, CheckBox1, RadioButton1, and Edit2 are in the form's Components array, rather than in that of GroupBox1.

Note also the program indicates at the top that GroupBox1's Parent is Form1, and its Owner is also Form1. When we had clicked on just the form (in the previous example), GroupBox1 was in Form1's Controls *and* Components array. This is why Form1 is listed as GroupBox1's Parent (in Form1's Control array) and Owner (in Form1's Components array).


Form1

Item Clicked: CheckBox1
Parent: GroupBox1
Owner: Form1

Figure 3: Click on the CheckBox item within GroupBox1 to observe who is the Parent and Owner.


Form1

Item Clicked: BitBtn1
Parent: Panel1
Owner: Form1

Figure 4: Click on BitBtn1 to identify Panel1 as its Parent.

Now if we click on an item within GroupBox1 — say the CheckBox — observe who's the Parent and who's the Owner (see Figure 3). As you can see, GroupBox1 is CheckBox1's Parent, but the Owner is unchanged — it's still Form1. Again, look at it from the reverse direction: when we had clicked on GroupBox1, it listed CheckBox1 as one of its children (i.e. in its Controls array). Therefore, when we clicked on CheckBox1, it identified GroupBox1 as its Parent. Similarly, when we click on BitBtn1, it identifies Panel1 as its Parent (see Figure 4).

In both cases, Form1 is still responsible for the creation and destruction of these components. In fact, you'll find that all the components will list Form1 as their owner. Remember, all the components were listed in the Components array of Form1, which therefore identifies Form1 as the owner of all the components.

The Sample Program

The source code for this sample application is simple. Essentially, it consists of only two event handlers: *FillLists* and *Reset*. The *Reset* handler clears the two list boxes, and identifies the item clicked, its parent, and its owner. The *FillLists* handler is used only when Form1, Panel1, or GroupBox1 are clicked. It first calls the *Reset* handler, and then populates the right list box with the contents of the Components array and the left list box with the contents of the Controls array:

```
procedure TForm1.FillLists(Sender: TObject);
var
  I : Integer;
begin
  Reset(Sender);
  for I := 0 to (Sender as TComponent).ComponentCount -1 do
     ComponentsListBox.Items.Add(
       (Sender as TComponent).Components[I].Name);
  for I := 0 to (Sender as TWinControl).ControlCount -1 do
     ControlsListBox.Items.Add(
       (Sender as TWinControl).Controls[I].Name);
end;
```

The object passed in (Form1, Panel1, or GroupBox1) is first cast as a *TComponent*. Then the *ComponentCount* property determines how many items are in the Components array. Then, it's simply a matter of looping through the Components array as many times as there are items, extracting the name of each component, and adding it to the list box. The Controls array is treated in the same manner.

What Good Are They?

These arrays can be very useful in a program. Say you want to change some property on all components on a form (or perhaps just certain components like *TPanel*s for example). Using the Components array of the form is an excellent technique to accomplish this. Consider the following code:

```
procedure TForm1.FormClick(Sender: TObject);
var
  I : Integer;
begin
  for I := 0 to ComponentCount -1 do
    if Components[I] is TPanel then
      (Components[I] as TPanel).Color := clLime;
end;
```

With this code, when the mouse is clicked on the form, all Panels are changed to Lime Green. Not a particularly useful example, but it illustrates how you can easily filter through controls on a form and selectively make changes to them.

The Controls array can also achieve some interesting effects. You may want to move all controls within a Panel a relative amount, but don't want to disturb other controls outside the Panel. The following code accomplishes this:

```
procedure TForm1.Panel1Click(Sender: TObject);
var
  I : Integer;
begin
  for I := 0 to Panel1.ControlCount -1 do
    with (Panel1.Controls[I] as TControl) do
      Left := Left + 1;
end;
```

Every time the Panel is clicked, each of the controls within it are moved one pixel to the right.

Conclusion

The Controls and Components arrays can be very powerful, if you know how to tap their power. Unfortunately, the Delphi online Help files are a bit sparse in their discussion of these structures. Hopefully this article has cleared up some of the confusion, and will help you understand how to best access and manage these arrays in your Delphi programs. ∆

The demonstration project referenced in this article is available on the Delphi Informant Works CD located in INFORM\96\OCT\DI9610RV.

Robert Vivrette is Technical Editor for *Delphi Informant*. He has worked as a game designer and computer consultant, and has experience in a number of programming languages. He can be reached on CompuServe at 76416,1373.

Delphi C/S

Database Servers / InterBase / Delphi

By Bill Todd

Versioning: The InterBase Advantage

An Examination of Database Concurrency Models

As you rush headlong into the world of client/server computing, one of the first things you must do is select a database server. The architectures of database servers vary widely, and as a result, their behavior in a given situation also varies widely.

This means that to select the appropriate server for your Delphi database application you must understand two things: how data will be accessed and modified in your application, and how the server will behave in each data access or update situation.

In this article, we'll explore the issues that affect concurrent access to data, as well as how they may impact your application.

Locking Schemes

The oldest and most common method of controlling concurrent access to data by several users is *locking*. When a user locks an object in a database, he or she restricts other users' ability to access that object.

How much a lock affects concurrency depends on the lock's *granularity*. For example, a lock placed on an entire table will restrict other users' access to all the records in the table. Therefore, a table-level lock has very low granularity. A lock placed on a single page in a table limits access to all the records on that page. A page-level lock is more granular than a table-level lock. By contrast, a lock placed on a single row is very granular, and provides the minimum restriction to concurrent data access.

Most database servers support either row- or page-level locking. The problem with page-level locks is easy to understand by looking at an example. Suppose a page's size is 2KB (2048 bytes) and a row's size is 100 bytes. Thus, each page can hold 20 rows, and each time a page is locked, access is restricted to all 20 rows. With row-level locking only a single row would be locked, and other users could freely access other records on the page. Thus, row-level locking provides better concurrency.

Pessimistic Locking. If your background is in desktop databases (e.g. Paradox), you're probably familiar with *pessimistic locking*. This scheme is so named because it assumes there's a high probability that another user will try to modify the same object in the database you're changing. In a pessimistic locking environment, the object you want to change is locked before you begin changing it, and the object remains locked until your change is committed. The advantage of pessimistic locking is that you're guaranteed the ability to commit the changes you make.

Let's say you need to change a customer's address. Using pessimistic locking, you first lock the customer information at either the page or row level. You can then read the customer's record, change it, and be guaranteed that you can write your changes to the database. Once you commit your changes, your lock is released and others are free to change the customer's record. Locks can persist for a long time when pessimistic locking is used. For example, you could begin a change and then take a lunch break. After returning, you can then commit the change and release the lock.

Clearly, you want to use locks with high granularity if you're going to use pessimistic locking in a multi-user environment. If you must lock an entire page of customer records while changing a single row, then no other user can change any other customer record on that page. Row-level locks are best when pessimistic locking is used. This is because they impose the least restriction on access by other users. Page-level locks are much less satisfactory, because as long as they persist, they restrict access to many rows.

Optimistic Locking. The most common locking scheme found in database servers (e.g. Oracle, Sybase, SQL Server) is *optimistic locking*. The locking mechanism is optimistic in that it assumes it's unlikely another user will try to change the same row you're changing. An optimistic lock is not placed until you're done committing your changes.

To understand optimistic locking, consider two users — Fred and Ethel — who are trying to change a customer's record. First, Fred reads the record and begins to make changes. Next, Ethel reads the same record and begins to make changes. This is possible because, in the optimistic locking scheme, no lock is placed when a user reads a record and begins changing it.

Then Fred completes his changes and attempts to commit them. The database locks the record, commits the changes, and releases the lock. When Ethel tries to commit her changes, the software detects that the record has been changed since she'd read it. Ethel's change is rejected, and she must re-read the record and begin again.

Optimistic locking has a clear advantage because locks are only held for a brief period while the data is updated. This means that with an optimistic locking scheme, you can achieve adequate concurrency with less lock granularity. Therefore, databases that use optimistic locking may lock at the page level and not at the row level. Conversely, optimistic locking does not fare well in an environment where there's a high probability that two users will simultaneously try to update the same row.

From the database vendor's point of view, page-level locking is advantageous because fewer locks must be placed — particularly during batch operations that affect many rows. This means the resource requirements of the lock manager module in the database management system are lower, and this can help improve the performance of the database server. However, users are invariably the slowest part of any database application, so you'll usually get better overall performance in an environment where one user cannot block another.

Why You Should Care

Understanding how your database manages locks can be critically important. Consider an Orders table. New records are added continuously as new orders are received. Because the Order data does not include a field (or fields) that would form a natural primary key, you decide to use an artificially generated Order Number as a surrogate key. Order Numbers will be assigned sequentially as orders are received.

Because your application must frequently select groups of orders, you create a *clustered index* on the Order Number column. A clustered index provides superior performance when retrieving adjacent records. This is because the records are physically stored in key order within the database pages.

Unfortunately, this design will probably produce poor performance if the database uses page-level locking. Because sequential adjacent keys are being assigned and a clustered index is being used, each new record added will probably be placed on the same page as the preceding record. Because the database locks at the page level, two users cannot add new orders to the same page simultaneously. Each new order must wait until the page lock placed by the preceding order is released. In this case, you would get much better performance by randomly assigning the keys. This will reduce the chance that successive records will be added to the same page.

Transactions

Database servers also require the ability to group changes to the database into *transactions*. Transactions consist of one or more changes to one or more tables in the database that must be treated as a single unit. This is so that either all or none of the changes that comprise the transaction occur.

Transaction processing occurs in three steps: First, tell the database you want to begin a transaction. This informs the database that all changes — until further notice — are to be treated as a single unit. Next, the changes are made to the tables in the database. Finally, notify the database system that you want to either *commit* or *rollback* the transaction. If you commit the transaction, the changes become permanent. All the changes are "undone" with a rollback.

Transaction processing is vital to ensure the database's logical integrity. Let's say that Fred transfers $100 from his savings account to his checking account. This transaction would proceed as follows: start a new transaction, update the savings account balance to show a withdrawal of $100, update the checking account balance to reflect an increase of $100, and either commit or rollback the transaction.

Suppose the system crashes after step one, but before step three. Without transaction control, Fred would have lost $100. With transaction control, when the system is restarted, the database management system (DBMS) will automatically rollback any transactions not committed at the time of the system's crash. This guarantees that the database will be left in a consistent state.

You also need transaction control for read transactions that will read more than a single record. This is to ensure that the read returns a consistent view of the data. We'll discuss this requirement in more detail in the next section.

Transaction Isolation

Transaction isolation governs how simultaneously-executing transactions interact with each other. Many of today's database servers were originally designed to process short update transactions intermixed with single row reads.

The perfect example of this is an automated teller machine (ATM). An ATM reads the balance in a single account, or updates the balance in one or more accounts. In this environment, transactions are short, and reads involve a single row at a time, so transaction isolation is not a serious concern. However, many of today's database applications do not fit this model.

Short update transactions are still the norm. However, the advent of executive information systems has introduced long-running read transactions that span entire tables — sometimes entire databases.

Let's consider the following scenario. An executive requests the total value of the company's inventory by warehouse. While the query is scanning the inventory table, a user moves a pallet of platinum bars from warehouse A to warehouse B and commits the transaction. It's possible for the query to count the platinum in both warehouses, thus producing an erroneous inventory valuation report.
The question becomes, "Which updates should a read transaction see, and when should it see them?" This is what the transaction isolation level controls. There are three basic isolation levels:

- *Dirty Read* — This isolation level allows any record in the database to be read whether or not it has been committed.
- *Read Committed* — This level allows read transactions to see only those changes that were committed.
- *Repeatable Read* — A repeatable read allows the read transaction to immediately see a snapshot of the database when the transaction began. Neither committed nor uncommitted updates that occur after the read transaction starts will be seen.

Note that the *TransIsolation* property of Delphi's *TDatabase* component allows you to set all three of these isolation levels. However, this doesn't mean that your server supports the isolation level you have selected. In addition, if you're using an ODBC driver, the driver must also support the isolation level you set. Search on "Transactions | Transaction Isolation Levels" in the Delphi online Help to view a table showing what each of these isolation levels maps to on your server.

In the example above, you need a *repeatable read isolation* to ensure the accuracy of your inventory valuation report. The problem is the price you must pay to get repeatable read in a database with a locking architecture. With the locking model, the only way to ensure that data does not change during a long read transaction is to prevent any updates from occurring until the read transaction ends. In many situations, the effect on users of stopping all updates for the duration of a long read transaction is unacceptable.

Versioning

Versioning is another model for concurrency control. It overcomes the problems that locking model databases have when the environment consists of a mixture of update and long read transactions. This model is called the *versioning model.* To date, InterBase is the only DBMS to use the versioning model.

Let's reconsider the preceding example. The read transaction to produce the inventory valuation report begins. When the update transaction to move the pallet of platinum from warehouse A to warehouse B is committed, a new version of each updated record is created. However, the old versions still exist in the database.

In a versioning database, each transaction is assigned a sequential transaction number. In addition, the DBMS maintains an inventory of all active transactions. The transaction inventory pages show whether the transaction is active, committed, or rolled back.

When an update transaction commits, the DBMS checks if there are transactions with lower transaction numbers that are still active. If so, a new version of the record is created that contains the updated values. Each version also contains the transaction number of the transaction that created it.

When a read transaction begins, it retrieves the next transaction number and a copy of the transaction inventory pages that show the status of all uncommitted transactions. As a read transaction requests each row in a table, the DBMS checks if the transaction number for the latest version of the row is greater than the transaction number of the transaction that's requesting it. The software also checks if the transaction was committed when the read transaction started.

Let's say the transaction number of the row's latest version is greater than the requesting transaction's number; or, the transaction which created the latest version was active when the read transaction started. With either scenario, the DBMS looks back through the chain of prior versions. The software continues until it encounters a version with a transaction number that is less than the transaction number of the transaction that is trying to read the row, and whose transaction status was committed when the read transaction started.

When the DBMS finds the most recent version that meets these criteria, it returns that version. The result is repeatable read transaction isolation without preventing updates during the life of the read transaction.

Consider the following example of a row for which four versions exist:

```
Tran=100 (status=committed)
   Tran=80 (status=active when read started)
      Tran=60 (status=rolled back)
         Tran=40 (status=committed when read started)
```

Assume that a read transaction with transaction number 90 attempts to read this row. The read transaction will not see the version of the row created by transaction 100 because the update that created this version took place after transaction 90 began. Also, transaction 90 cannot read the version created by transaction 80, even though it has a lower transaction number. This is because transaction 80 isn't yet committed. Although the version for transaction 60 still exists on disk, transaction 60 has rolled back — and rolled back versions are always ignored. Therefore, the version that transaction 90 will read is the version created by transaction 40.

Note that in this example, transaction 80 is not allowed to commit. When transaction 80 attempts to commit, the DBMS will discover that transaction 100 has committed, and transaction 80 will be rolled back.

Advantages of Versioning

For a more complete understanding of how the locking and versioning models compare, you must examine two things: the types of concurrency conflicts that can occur in a multi-user database, and how each model behaves in each case.

The following examples assume that the locking model uses a shared read lock and an exclusive write lock to implement optimistic locking. Multiple users can place read locks, but no user can place a write lock if another user has either a read or write lock. If one user has a write lock, another user can neither read nor write the row. This is typical of databases that use locking architecture.

Consider the case where a husband and wife go to different ATMs at the same time to withdraw money from their joint checking account. Without concurrency control, the following sequence of events occurs:

- Fred reads the account's balance as $1,000.
- Ethel reads the account's balance as $1,000.
- Fred posts a $700 withdrawal.
- Ethel posts a $500 withdrawal.

At this point, the account balance is -$200 and the bank is not happy. This happened because without a concurrency control mechanism, Fred's update is lost as far as Ethel is concerned. She never sees the change in the account balance. However, under the locking model:

- Fred reads the account's balance, causing a read lock.
- Ethel reads the account's balance, also causing a read lock.
- Fred posts his withdrawal, attempting a write lock that fails because of Ethel's read lock.
- Ethel posts her withdrawal, attempting a write lock that fails because of Fred's read lock.

A deadlock now exists. Hopefully, the DBMS will detect the deadlock and rollback one of the transactions.

Under the versioning model, Fred reads the account's balance and Ethel reads the account's balance. Then, Fred posts his withdrawal, which causes a new version with a new balance to be written. When Ethel posts her withdrawal, it's rolled back when the newer version is detected.

A different problem occurs if a user does not commit a transaction. Let's say Fred withdraws money from the account and this updates the balance. Ethel reads the balance and Fred cancels the transaction before committing. Now Ethel has seen the wrong balance. In this case, a dependency exists between the two transactions. Ethel's transaction produces the correct results only if Fred's transaction commits. This illustrates the danger of reading uncommitted data.

Using locking, Fred reads the balance that places a read lock, and then commits a withdrawal that places a write lock during the update. Ethel reads the balance, which attempts a read lock, but must wait because of Fred's write lock. Fred cancels the transaction before committing. This rolls back and releases the write lock. Ethel can now read and get the correct balance.

Under versioning, Fred withdraws the money. This updates the balance and creates a new uncommitted version. At her machine, Ethel reads the balance, but it does not reflect Fred's uncommitted withdrawal. Fred rolls back, so the version showing the withdrawal is marked rolled back. This illustrates a performance advantage of versioning because Ethel does not have to wait to read the balance.

The following is a different example, but it's the same as our earlier scenario of moving platinum from one warehouse to another:

- Fred requests the total of all accounts.
- Ethel transfers money from savings to checking while Fred's transaction is running.
- Fred receives the wrong total. The analysis of the data is inconsistent because the data's state was not preserved throughout the life of the read transaction.

Under locking, Fred requests a total of all accounts, thereby placing a read lock. Ethel transfers money but cannot place a write lock to commit the transfer because of Fred's read lock. Ethel must wait until the read transaction finishes. Finally, Fred gets the right total and releases the read lock, and Ethel's transaction can proceed.

Under versioning, Fred requests the total. At her ATM, Ethel transfers money from savings to checking, resulting in new versions which Fred's transaction does not see. Fred gets the correct total and Ethel's update is not delayed.

Another variation of the repeatable read problem occurs if you must reread the data in the course of the transaction. For example:

- A query is started for all rows meeting certain criteria.
- Another user inserts a new row that meets the criteria.
- Repeat the query and you will get one additional row. The appearance of this "phantom row" is not consistent within the transaction.

With a database that uses the locking model, the only way to prevent this inconsistency is to read lock the whole table for the duration of the transaction. Thus the sequence of events is:

- Place a read lock on the table.
- Query for all records meeting certain criteria.
- Another user attempts to insert a record, but is blocked by the table-level read lock.

- Repeat the query and you'll get the same results because other users cannot commit changes.

Under versioning there's no problem, because the newly inserted record has a higher transaction number than the read transaction. Therefore, it's ignored on the second and subsequent reads that are part of the same transaction.

Disadvantages of Versioning

So far it looks as if the versioning model handles most concurrency conflicts better than the locking model. However, this is not always the case. In this example, Fred and Ethel are both told to make their salaries equal:

- Fred reads his salary.
- Ethel reads her salary.
- Fred sets Ethel's salary equal to his.
- Ethel sets Fred's salary equal to hers.

Under versioning, the result is that their salaries are simply swapped. Using locking, you can prevent this by locking both records. For example, both Fred and Ethel read their own salaries and place read locks. Fred sets Ethel's salary equal to his, but cannot commit because of Ethel's read lock. Likewise, Ethel sets Fred's salary equal to hers, but cannot commit because of Fred's read lock.

Once again, you have a deadlock that the database system should resolve by rolling back one transaction. Another solution using locking is to write lock the entire table. For example, Fred write locks the table and reads his salary. Ethel then tries to read her salary, but is blocked by Fred's table-level write lock. Fred sets Ethel's salary equal to his and releases the write lock. Ethel's transaction is now free to proceed.

Under versioning, Fred reads his salary and Ethel reads hers. Fred sets Ethel's salary equal to his and commits. Then Ethel sets Fred's salary equal to hers and commits. Once again the salaries are swapped, because versioning allows both transactions to process concurrently. The only way to solve this problem with the versioning model is as follows:

- Fred reads his salary.
- Ethel reads her salary.
- Fred sets Ethel's salary equal to his.
- Fred sets his salary to itself, creating a newer version.
- Ethel sets Fred's salary equal to hers, but it rolls back because a newer version exists.

Here the problem is solved by setting Fred's salary equal to itself. This forces the creation of a new record version for Fred's salary. Versioning architecture will not allow a change to be committed when a version of the record to be updated exists (which was created after the start of the current transaction). Therefore, Ethel's update rolls back.

Recovery

One very important issue in any database application is recovery time when the server crashes. No matter how robust your hardware and software and/or how reliable your electric power supply, there's always a possibility the server will fail. Both locking and versioning databases will recover automatically when the server is restarted. However, there's a significant difference in the recovery time.

Locking-model databases write each transaction to a log file. To recover after a crash, the DBMS must read the log file and rollback all the transactions that were active at the time of the crash by copying information from the log to the database.

A versioning database does not have a log file. The record versions in the database already provide all the information required to recover. No data needs to be copied from one place to another. Instead, when the DBMS comes back on line, it simply goes through the transaction inventory pages and changes the status of all active transactions to rolled back. At most this will take a few seconds, even on a large database or one with a large number of active transactions. Thus, crash recovery is another area where the versioning model excels.

Other Issues

At first it may appear that a versioning database has a significant disadvantage. This is because the multiple record versions will cause the database size to temporarily increase rapidly compared to a locking database. While this is true, don't forget that other databases also grow as their log files expand.

However, versioning databases will certainly grow rapidly if something is not done to control the proliferation of record versions. The DBMS performs some of the housekeeping for you automatically. Each time a record is accessed, the DBMS checks if any prior versions of that record are no longer needed. A version is obsolete if its transaction rolled back, or if there is a later committed version of the record and there are no active transactions with a transaction number less than the transaction number of the newer committed version. Versions that are obsolete are automatically deleted and the space they occupied in the database pages is reused.

Many rows in many databases are visited infrequently. To remove unnecessary versions of these rows, the database must be periodically "swept." A sweep operation visits every row in every table in the database and deletes outdated versions. You can run the sweep while the database is in use, but the sweep will impact performance while it's running.

InterBase, by default, will automatically start a sweep after 20,000 transactions. This isn't the best way to manage sweeping, because you have no control over when the sweep will start. In addition, the user who starts the transaction that triggers the sweep is locked until the sweep finishes. It's better to periodically start a sweep manually when database use is low.

Conclusion

Selecting the correct database for your application requires a clear understanding of the types of transactions the system must process. Many applications today require a mixture of multi-row read transactions and updates. In this environment, versioning has a clear advantage because it can process read and write transactions concurrently while still providing repeatable read to ensure accuracy.

Versioning also provides rapid crash recovery because there's no log file to process. When a versioning database restarts, it simply marks all open but uncommitted transactions as rolled back, and it's ready to go.

As stated earlier, InterBase is the only DBMS to use the versioning model. In addition to the advantages of the versioning model, InterBase has the smallest disk and memory footprint (it ships on two diskettes), is self-tuning, and runs on NetWare, Windows NT, and a wide variety of UNIX platforms. Therefore, InterBase is highly scalable. ∆

Bill Todd is President of The Database Group, Inc., a Phoenix area consulting and development company. He is co-author of *Delphi 2: A Developer's Guide* [M&T Books, 1996], *Creating Paradox for Windows Applications* [New Riders Publishing, 1994], and *Paradox for Windows Power Programming* [QUE, 1995]; a member of Team Borland; and a speaker at every Borland Developers Conference. He can be reached at (602) 802-0178, or on CompuServe at 71333,2146.

By Douglas Horn

Return to Sender

The Lowly *Sender* Parameter Can Make Applications Shine

The humble *Sender* parameter could be one of Delphi's most useful tools for modular, extensible programming. Although *Sender* is normally used to simply determine which object called a particular procedure, with some creativity, this parameter can be extended to allow robust modular programming with simple, reusable code.

Sender Basics

The *Sender* parameter is so ingrained in Delphi programming that it appears in practically every program procedure. Even the Form's *OnCreate* event — which has no sender — includes the `(Sender: TObject)` parameter in its definition.

Despite this, the *Sender* parameter is not necessary for a procedure to function. Removing *Sender* from a procedure that doesn't use it results in functional code, as long as the parameter is also removed from the procedure's **type** statement in the form object's **interface** section. Delphi will generate an "incompatible parameter" warning at compile time, but this can be ignored. In fact, removing unused *Sender* parameters results in slightly leaner code.

Not only can *Sender* be omitted from any procedure, but when used, it doesn't even have to be called *Sender*. This name is only used by convention. The common term makes understanding code simpler, but there's no reason *not* to name it *Caller*, *Origin*, or *Banana* if the developer prefers. So the *Sender* parameter doesn't have to be called *Sender*, and is, in fact, unnecessary. So what is it? And why is it important enough to add to practically every Delphi procedure?

Sender is a parameter passed from the component that called the event handler, and is a parameter of the type *TObject*. Because *TObject* is the ancestor of all components, *Sender* can accommodate any Delphi object that can trigger an event. *Sender* forms a two-way link between an object and an event handler; it tells the procedure what object triggered it to be called. In other words, if an event handler was a letter, the *Sender* parameter would be the return address.

Using *Sender*

The simplest way to use *Sender* is with an **if...then** or **case** statement that performs a certain action depending on what object triggered the procedure. Here's a typical example:

```
procedure Form1.ButtonClick(Sender: TObject);
begin
  if Sender is StopButton
    then Stop(Sender);
end;
```

This code examines the *Sender* parameter to see if it's the object, *StopButton*. If so, the event handler calls another procedure, *Stop*, passing the *Sender* parameter to that procedure. The *Stop* procedure calls another procedure and so on, passing the *Sender* parameter through the program code. Because each procedure is simply passing along the parameter it received, wherever the parameter is used, it will reflect the object that called the original event handler.

Not only can Delphi developers use *Sender* to identify the object that called a procedure, but they can also use it to access that object's properties. The form in Figure 1 uses a number of colored panels as a palette. When the user clicks on a panel, the large panel on the left changes to the color of the selected panel. This procedure does not need the name of the panel selected, only its *Color* property. Therefore, each panel in the palette can have its own color, but all share the following *OnClick* event handler:

```
procedure
TForm1.Panel1Click(Sender:
TObject);
begin
  Panel1.Color := TPanel(Sender).Color;
end;
```

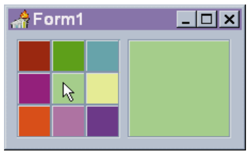


Figure 1: This mini-program sets the color of the panel at right to the color of the panel selected by the user.

Since *Sender* provides a "return address" to the calling object, that object's properties can be read and set as well. This way, the procedure can also change the panel's *Color* property without actually working with the panel object's name. For example:

```
TPanel(Sender).Color := clRed;
```

Any type of property can be accessed using this framework, provided it's a property belonging to the specified type. In the previous example, the procedure specifies that *Sender* is a *TPanel*, which therefore has a *Color* property. While *Sender* is declared as a *TObject* in the procedure header, the following line would return a compiler error:

```
TPanel1.Color := TObject(Sender).Color;
```

As *TPanel*'s ancestor type, *TObject* is perfectly valid in other respects; but it does not contain a *Color* property. Since *TObject* contains no properties, developers cannot use it to determine the properties of a wide range of objects. Thus, a program can't find the color of a *TPanel*, *TLabel*, or *TForm* all with the same `TObject(Sender).Color` code.

In fact, these three component types all contain the *Color* property. However, because they do not share this property in common ancestor classes, programmers cannot easily create one block of code to get the *Color* property from these various types of objects. The simplest solution a developer could use would be this:

```
if Sender is TPanel then
  Brush.Color := TPanel(Sender).Color;
if Sender is TLabel then
  Brush.Color := TLabel(Sender).Color;
if Sender is TForm then
  Brush.Color := TForm(Sender).Color;
```

Fortunately for developers, the Delphi designers added the *Tag* property. *Tag* is a little catch-all integer parameter that developers can use for whatever purpose they choose. What makes *Tag* especially useful is that it resides well up the inheritance tree in the *TComponent* class. Just two steps down the ladder from *TObject*, *TComponent* is the ancestor of all controls. This means that any of these objects will have the *Tag* property. Unlike the *Color* property cited earlier, the *Tag* property of any object can be accessed with a simple line of code:

```
TForm.Tag := TComponent(Sender).Tag;
```

Sender Impostors

As procedures pass the *Sender* parameter from one to another, it never changes. It always points back to the object that called the original event handler. This is true provided each procedure along the path passes the original parameter. However, you can also substitute another parameter for the original *Sender*, fooling subsequent procedures into acting on another object as if it were the true *Sender*.

This is the basis of our sample application, SENDER.DPR. It uses *Sender* and *Tag* so that Delphi can be "fooled" into having similar menu and button commands perform special actions on the buttons. Traditionally, this wouldn't require special *Sender* parameters — both buttons and menu items will share an event handler that can update the button as part of its functionality:

```
procedure Button1Click(Sender: TObject);
begin
  { Main functionality }
  Button1.Font.Style := [fsBold];
end;
```

However, when several components share a single event handler, the programmer cannot simply assume to act on a specific object. Modifying the code in question to read:

```
TButton(Sender).Font.Style := [fsBold];
```

is fine if the code is always called by a *TButton*. However, this makes it troublesome to call the same event handler from a corresponding menu item because the menu item causes an error each time it calls the event handler.

The Sample Application's Framework

The sample application provides the framework for a modular, easily-extendible application. It also demonstrates how to make the simple *Sender* parameter do a lot of high-powered work, and shows how to get around some of the limitations mentioned earlier.

Most applications contain menu items and buttons that perform the same action. Delphi makes it simple to do this by routing multiple events to a shared event handler. If multiple buttons and menu items all use the same event handler, it's simple to determine which control called the procedure. However, it becomes more difficult to determine, say, which menu item corresponds to the calling event if the control was actually a button. The user may obviously find that the **Word Wrap** button and the **Word Wrap** menu command are synonymous in functionality. However, to have Delphi understand this and update the menu item's *Checked* property — regardless of which control was the *Sender* — takes some special programming.

The sample application contains four SpeedButtons, as well as main and pop-up menu items that correspond to each (see Figure 2). The buttons and menu items perform similar functions, in this case, using a database to track the time spent on each of a number of tasks. Program users can configure any number of tasks, all of which call the same event handler. (For clarity, the sample application illustrates only the *Sender* functions.)

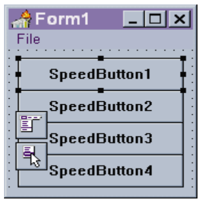


Figure 2: The layout of the sample application's components.

While a task is active, its SpeedButton remains depressed. Pressing a depressed button releases it and ends the task. Pressing a button while another is depressed releases the first and depresses the new button, causing the timesheet database to be simultaneously updated. The trick is to handle a number of MenuItems and SpeedButtons with one event handler. This technique enables you to allow for any number of tasks without writing a new handler for each.

Implementing this functionality requires one more ingredient: the *Components* property. Like *Tag*, *Components* is a property of *TComponent*, meaning that it's accessible to all components. *Components* is a property of every component, but in most cases it's only used on *TForms*. In this case, the *Components* property is an array of all components owned by the form.

Build It

To build the sample application, create a single form named *Form1*. Add four SpeedButtons (used here to easily allow two-state buttons), a Label, a MainMenu (with an item called *File1* and four sub-items), and a pop-up menu (also with four menu items). Set the properties as shown in Figure 3. Note that all the buttons and menu items have the same event handler, namely *SpeedButtonClick* (see Figure 4). This procedure is called from any of the SpeedButtons or menu items on the form (except MenuItem *File1*). It references two external procedures, *PushButton* and *ReleaseButton*.

SpeedButtonClick first determines which object sent the event. If it's a *TSpeedButton*, it passes that object to the *PushButton* or *ReleaseButton* procedure, depending on the button's state (up or down). These procedures allow convenient spots for messages to be handled. In the following example, the *Caption* of *Label1* is set to report the event that occurred:

```
procedure TForm1.PushButton(Sender: TObject);
begin
  Label1.Caption :=
    TComponent(Sender).Name + ' was clicked.';
end;

procedure TForm1.ReleaseButton(Sender: TObject);
begin
  Label1.Caption :=
    TComponent(Sender).Name + ' was released.';
end;
```

If the *Sender* of the event is not one of the buttons, we must determine which button to click. This is where the *Tag* property and the *Components* array property come in. As you noticed in Figure 3, *SpeedButton1* had a *Tag* value of 1. *MainItem1* and *PopupItem1* also have *Tag* values of 1. Therefore, if any of the menu items are selected, it's a simple matter of viewing their *Tag* values and transferring the event to the SpeedButton with the same *Tag*. This is handled in the *SpeedButtonClick* handler:

```
for A := 0 to ComponentCount-1 do
  if Components[A] is TSpeedButton then
    with Components[A] as TSpeedButton do
      if Tag = TComponent(Sender).Tag then
```

This code snippet scans the *Components* array and examines each component it finds. When a *TSpeedButton* is found, the code compares this object's *Tag* value with the *Tag* value of the object that originally fired the message. If the values are the same, then the code found the button that ultimately receives the event. Since the code has already determined that it's a *TSpeedButton*, this object's up/down state can now be toggled. Then, either the *PushButton* or *ReleaseButton* procedure is called with the selected SpeedButton passed as the parameter as follows:

SpeedButtons

Name/Caption	AllowAllUP	GroupIndex	*Tag*	*OnClick* event handler
SpeedButton1	True	1	1	SpeedButtonClick
SpeedButton2	True	1	2	SpeedButtonClick
SpeedButton3	True	1	3	SpeedButtonClick
SpeedButton4	True	1	4	SpeedButtonClick

MenuItems

Name/Caption	*Tag*	*OnClick* event handler
File1		0 --
MainItem1	1	SpeedButtonClick
MainItem2	2	SpeedButtonClick
MainItem3	3	SpeedButtonClick
MainItem4	4	SpeedButtonClick
PopupItem1	1	SpeedButtonClick
PopupItem2	2	SpeedButtonClick
PopupItem3	3	SpeedButtonClick
PopupItem4	4	SpeedButtonClick

Figure 3: Setting the values of properties for the sample application's SpeedButton and MenuItem components.

```
procedure TForm1.SpeedButtonClick(Sender: TObject);
var
  A : Integer;
begin
  if Sender is TSpeedButton then
    if TSpeedButton(Sender).Down then
      PushButton(Sender)
    else
      ReleaseButton(Sender)
  else
    for A := 0 to ComponentCount-1 do
      if Components[A] is TSpeedButton then
        with Components[A] as TSpeedButton do
          if Tag = TComponent(Sender).Tag then
            begin
              Down := not Down;
              if Down then
                PushButton(Self.Components[A])
              else
                ReleaseButton(Self.Components[A]);
            end;
end;
```

Figure 4: Referencing *PushButton* and *ReleaseButton* with the *SpeedButtonClick* procedure.

```
begin
  Down := not Down;
  if Down then
    PushButton(Self.Components[A])
  else
    ReleaseButton(Self.Components[A]);
end;
```

This statement:

```
PushButton(Self.Components[A])
```

calls the procedure *PushButton*, passing along the parameter:

```
Self.Components[A]
```

The *PushButton* procedure interprets this *Self* statement as the original *Sender*. The *Self* portion is required because we want to look at the *Components* property of *Form1*. Since the code uses a **with** statement to simplify readability, we must ensure that the code is viewing the correct *Components* property. In this case, SpeedButtons also have a *Components* property and without the *Self* qualifier, it will erroneously look there instead.

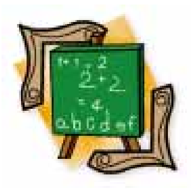
The *PushButton* procedure is where the timesheet database would be updated or other functions would be performed. Since the previous procedure handled the bulk of the work, *PushButton* simply performs the actions necessary when the button has been pressed. Remember that *SpeedButtonClick* did not send the actual *Sender* parameter, but sent a modified one that points to the *SpeedButton* regardless of how the button was selected. *SpeedButtonClick*'s other procedure call, *ReleaseButton*, is likewise called to handle whatever events should occur when the button is released.

Conclusion

Alone, the *Sender* parameter is a simple return address, allowing functions to trace which component triggered their event. But used in conjunction with a few other properties, such as *Tag* and *Components*, *Sender* becomes a powerful tool for tracing and filtering program flow. This ability to work around program limitations allows developers to create single, modular event handlers for components of all types, without the need for multiple **if..then** statements. This, of course, is just one way of extending *Sender*'s abilities. There are as many possibilities as there are programs to write. ∆

The demonstration program referenced in this article is available on the Delphi Informant Works CD located in INFORM\96\OCT\DI9610DH.

Douglas Horn is a free-lance writer and Contributing Editor to *Delphi Informant*. He can be reached via e-mail at horn@halcyon.com. Readers may browse a collection of his past articles at his Web site, http://www.halcyon.com/horn/default.htm.



Delphi / Object Pascal

By Dan Miser

The INISource Component

Creating INI-Aware Controls

Delphi's ability to interact with databases is second to none. However, there is a related feature that is not supplied in Delphi — the ability to interact seamlessly with .INI files. .INI files are a convenient way for a program to store user configuration settings that can affect almost any aspect of a program.

Delphi made access to the .INI file easy by wrapping almost all existing functionality of .INI files from the Windows API into an object named *TINIFile*. However, this still leaves a great deal of manual coding. The programmer must read each item from the .INI file and set the corresponding control with that value. When the user is done editing these controls, the programmer needs to save the value of each control back into the .INI file.

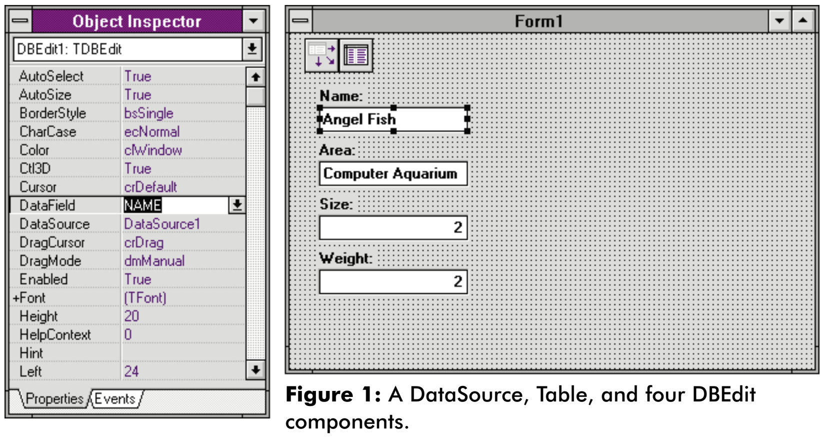
The entire process is very similar to the way Delphi's data-aware controls interface with databases. In that model, the *DataSource* and *DataField* properties of the data-aware control determine the value of the data-aware control. The programmer never needs to worry about reading data from, or writing data to, the database table again. By creating a set of .INI-aware controls to mimic the behavior of the data-aware controls, the programmer will be freed from the mundane task of reading and writing individual .INI file entries throughout the application.

Determining the Approach

The first step in deciding how to create the .INI-aware controls is to consider how Delphi's data-aware controls work. During Delphi's development, Borland adopted a standard in which each data-aware control is created as a descendant of its non-data-aware ancestor, and any database access needed would be added in every data-aware control. For example, the standard data-aware edit control, *TDBEdit*, descends from *TCustomMaskEdit*. It also has its own data-handling methods that interact with the database and various properties of the control. Therefore, the .INI-aware controls should also descend from their non-data-aware ancestors, and add methods to interact with the .INI file and control.

An example best reveals the integration of Delphi's data-aware controls. Figure 1 shows a DataSource, Table, and four DBEdit components. Note that the *TableName* property is assigned in only one place, the Table component. The DataSource component becomes linked to that Table via the *DataSet* property. This interaction allows the data-aware controls to reference a database table by assigning the appropriate DataSource component in their *DataSource* property.

This mechanism is used so data-aware controls can refer to either a Table or Query component. Since we only need to read and write from one type of data source (the .INI file), we can have the *TINISource* component contain the file name.



Object Inspector

DBEdit1: TDBEdit

AutoSelect	True
AutoSize	True
BorderStyle	bsSingle
CharCase	ecNormal
Color	clWindow
Ctl3D	True
Cursor	crDefault
DataField	NAME
DataSource	DataSource1
DragCursor	crDrag
DragMode	dmManual
Enabled	True
+Font	(TFont)
Height	20
HelpContext	0
Hint	
Left	24

Properties / Events

Form1

Name: Angel Fish

Area: Computer Aquarium

Size: 2

Weight: 2

Figure 1: A DataSource, Table, and four DBEdit components.

The INISource Component

The INISource component is the master controller of all .INI-aware controls. It will be used to associate the .INI file and .INI-aware controls. When the programmer assigns an .INI file name to the *FileName* property, a *TINIFile* object is created. There is an interesting side-effect of this creation: not only will the *TINIFile* open an existing .INI file, but if the file does not exist, it will create the file automatically. Therefore, the programmer does not need to do anything different if this is the first time the user has run the program, or if they have already saved settings from a previous run.

The INISource component will also serve as a centralized read/write mechanism for the .INI-aware controls. Using Borland's *TTable* object as a model, the *Post* and *Cancel* methods will be used to allow the programmer to easily save and restore all the .INI-aware controls that have been registered with a specific INISource component.

Once the INISource component is in place and an .INI file has been chosen, the .INI-aware controls need a way to refer to that existing component. This is easily accomplished in Delphi by specifying a property of type *TINISource* in the **published** section of the component. Delphi will search the form for all components of type *TINISource* and place them in the list of the Object Inspector automatically.

Of course, no component that references another component is complete without a *Notification* method. This method is called every time a component is inserted into or deleted from a form. Thus, the programmer can take special action if the component that is about to be deleted is referenced in the current component. This method is also necessary to ensure the *INISource* property references a valid INISource component at all times, thereby avoiding a nasty GPF. Figure 2 shows the *TINIEdit.Notification* method.

Each .INI-aware control will have several properties that will describe the appropriate .INI file entry for that control. For example, once the INISource component is set for a *TINIEdit*, the *INISection* property will have a list of currently defined section names for the .INI file that the INISource component has defined. Lastly, the *INIDefault* property can be set to a value appropriate for its type, in case the user has not yet saved any values to the .INI file.

Into the INI

Displaying existing Section and Keyword names of an .INI file will be handled by writing property editors to show all the possibilities available to the programmer at any time. This is particularly helpful to complete the total concept of visual programming. By providing this functionality, the programmer does not need to remember each and every section name for a given .INI file, or every keyword that exists inside the section that was just selected.

Another benefit of this visual programming metaphor is the elimination of typing errors by allowing the programmer to

```
procedure TIniEdit.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (FIniLink <> nil) and
     (AComponent = IniSource) then
    IniSource := nil;
end;
```

Figure 2: The *TINIEdit.Notification* method.

```
procedure TIniKeywordProperty.GetValueList(List: TStrings);
var
  Instance     : TComponent; IniSourceInfo  : PPropInfo;
  SectionInfo  : PPropInfo;    IniSource     : TIniSource;
  Section      : string;
begin
  Instance      := GetComponent(0);
  IniSourceInfo  :=
    TypInfo.GetPropInfo(Instance.ClassInfo,'IniSource');
  if (IniSourceInfo <> nil) and
     (IniSourceInfo^.PropType^.Kind = tkClass) then
  begin
    IniSource :=
      TObject(GetOrdProp(Instance,IniSourceInfo))
        as TIniSource;
    SectionInfo :=
      TypInfo.GetPropInfo(Instance.ClassInfo,'IniSection');
    Section := GetStrProp(Instance, SectionInfo);
    if (IniSource <> nil) and (SectionInfo <> nil) and
       (Section <> '') then
      IniSource.IniFile.ReadSection(Section,List);
  end;
end;
```

Figure 3: The *TINIKeywordProperty* property editor.

select from a list of valid possibilities. Of course, the user can always type in a value that does not exist to create a new section or keyword.

The property editors rely on a great deal of run-time type information (RTTI) in order to communicate directly with other components on the form. An example of this communication can be found in the *TINIKeywordProperty* property editor (see Figure 3). This method needs to set the list of possible keywords that exist in an .INI file. The call to *GetComponent* retrieves the component that triggered this property editor. This is necessary to identify the component throughout the rest of the method. After the component has been identified, its other properties can also be accessed in the same fashion. (The VCL source code has several property editors that are excellent learning aides. For more information, see the file \DELPHI\LIB\DBREG.PAS.)

When the programmer specifies an .INI file name for the INISource component, all section names of that .INI file need to be identified. Any lines that are enclosed in brackets can be identified as section names, and therefore need to be inserted into an awaiting *TStringList* variable called *SectionNameList*.

Since both the *Section* and *Keyword* properties are nothing more than a list of possible entries, the *paValueList* is the correct value to assign in the *GetAttributes* method. To assign the

```
procedure TIniEdit.WriteFile(Sender : TObject);
begin
  if (IniSource <> nil)  and  (IniSource.Active) and
     (FIniSection <> '') and (FIniKeyword <> '') then
    IniSource.IniFile.WriteString(
      FIniSection, FIniKeyword, Text)
end;
...
procedure TIniEdit.ReadFile(Sender : TObject);
begin
  if (IniSource <> nil)  and (IniSource.Active)  and
     (FIniSection <> '') and (FIniKeyword <> '') then
    Text := IniSource.IniFile.ReadString(
              FIniSection, FIniKeyword, FIniDefault)
  else
    Text := Name;
end;
...
constructor TIniEdit.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);
  FIniLink             := TIniLink.Create;
  FIniLink.Control     := Self;
  FIniLink.OnReadFile  := ReadFile;
  FIniLink.OnWriteFile := WriteFile;
end;
```

Figure 4: *TINIEdit* routines.

list of possible values, the method *GetValues* will be called. It is here that the abstract method *GetValueList* needs to assign the appropriate items to the list.

Reading and Writing

The work of reading and writing the .INI file is left to the non-visual component, *TINILink*. This component acts as intermediary between the *TINISource* component and the .INI-aware control to which it belongs. Setting up the *TINILink* to communicate properly requires two steps: creating the *TINILink* in the .INI-aware control, and adding the newly created *TINILink* component to the list of INILinks contained in the appropriate *TINISource* component.

In addition to these prerequisites, each .INI-aware control is also responsible for providing methods to read from, and write to, the .INI file. By assigning these methods to the event-handlers of the *TINILink* component, each .INI-aware control is able to respond as it should when a request to read or write the .INI file occurs (see Figure 4).

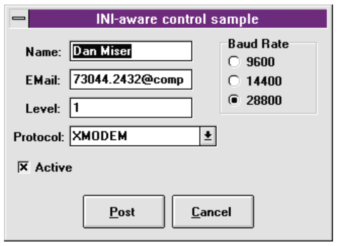
All this setup is necessary to allow the INISource component to communicate back to all of the links that have registered with it. For example, when the user first set the *INISource.Active* property to *True*, the INISource component needs to notify all .INI-aware controls that they can read the .INI file and display the appropriate value. This architecture will also allow design-time viewing of the data as it exists at that moment (see Figure 5).

```
procedure TIniSource.NotifyIniLinks(Event : TIniEvent);
var
  i : Integer;
begin
  for i := 0 to FIniLinks.Count-1 do
    TIniLink(FIniLinks[i]).IniEvent(Event);
end;
```

Figure 5: The *Active* procedure calls NotifyLinks, which in turn calls the *ReadFile* method of the .INI-aware controls.

Conclusion

Now that the components are built, it's time to see them in action. The sample program provided will allow the user to modify certain settings in his or her WIN.INI file. There is a button on the form to save the user's changes to the .INI file, as well as a button to revert all the changes to the current .INI file's state. A complete .INI file editor in two lines of code! Notice the effect of setting the *INISource.Active* property both at design time and run time (see Figure 6).



Name: Dan Miser
EMail: 73044.2432@comp
Level: 1
Protocol: XMODEM
[✓ Active]
Baud Rate: 9600, 14400, [✓ 28800]

Figure 6: The example program in action.

The reader is left with many different opportunities to extend the scope of these components. Some examples would be:

- Add other non-basic components to behave in a similar fashion (e.g. Orpheus controls).
- Allow access to the registration files of Delphi 2.
- Allow different properties to be saved to the .INI file.

Visual programming and component-based design are the new paradigms in programming, and Delphi supports those models superbly. The beauty of Delphi's extensibility is evident when even a small feature can be automated, allowing the programmer to focus on more difficult problems. The components developed here relieve the programmer from the tedium of ever dealing with .INI files again. ∆

Dan Miser is a contract programmer who has been writing Windows database programs since 1991. He is also a Borland Certified Delphi Client/Server Developer. You can contact him at 73044.2432@compuserve.com.

By *James Callan*

With Class 3.0

MicroGOLD Software Develops OOA & OOD Tool

If you are familiar with Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD), but don't want to spend a fortune for CASE support, then With Class 3.0 from MicroGOLD Software, Inc. is worth a look. With Class is the affordable CASE tool many developers have been waiting for, supporting Delphi 1 and 2. It lacks the polish of a US$1,200 CASE tool, but at one-fourth the price (US$295), you might be willing to forego some sheen.

Restraining RADicals

According to industry accolades and product awards, most developers agree Delphi is leading in Rapid Application Development (RAD). Unfortunately, prototype systems developed with RAD are frequently deployed as production systems, lacking the endurance and elegance found in their well-engineered counterparts. The difference is proper analysis and design.

When designing larger applications, programmers must comply with some unifying methodology. Formal OOA and OOD methodologies enable programmers and analysts to visualize object-oriented systems by providing graphical notations for capturing the relationships between classes and the dynamic interactions between run-time objects. Properly employed, OOA and OOD techniques help refine these RADicals into well-engineered production applications.

Although many academicians and consultants have proposed different graphical notations on requirement analysis, the notations proposed by Rumbaugh, Coad-Yourdon, Booch, and Shlaer-Mellor are generally used. There are exceptions (e.g. Jacobson's method in the Telecom industry), but not many. Developers and companies often standardize on their favorites. I prefer the Booch method, yet many CASE tool authors find the Booch *blobs* difficult to automate. Every method requires the drawing of diagrams to visualize, communicate, and document designs. Just as the pencil begot the eraser, these methodologies have spawned CASE tools.

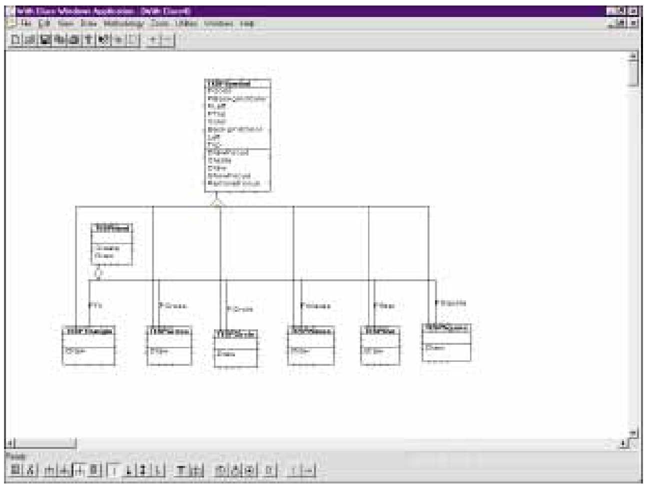


Figure 1: A class diagram produced from Delphi code using With Class.

Casing CASE Tools

Accelerating the drawing process is the first step in automating a methodology. Typically, graphical drawing tools such as Visio quickly capture the graphical diagrams. Soon designers will want to capture the semantics between class and object relationships, and diagrams augmented by detailed attribute (property) and class definitions. Ultimately, the goal is to document a system's requirements, including all the decisions made until deployment. These are worthy goals, and CASE tools can help.

Although they serve many additional purposes, these CASE packages are essentially smart drawing tools; capturing designs for

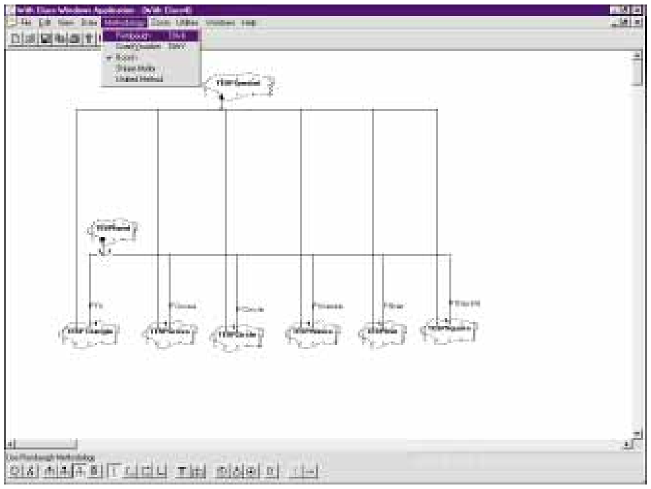


Figure 2: A Booch methodology class diagram.

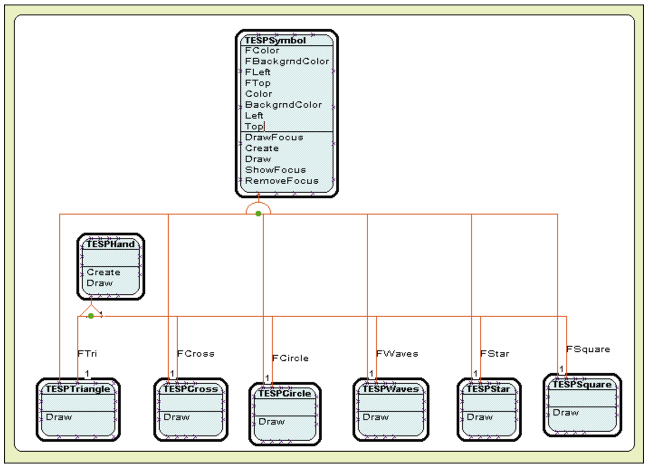
communication and documentation are their primary use. All CASE tools permit users to drag and drop graphical shapes, connect the shapes with lines, and adorn the lines with special symbols. CASE tools also provide dialog boxes for recording semantic and design information about classes, objects, properties, methods, events, exceptions, and such. After accepting the information, the tools assist in analyzing designs for completeness, consistency, and integrity. With most of the tools, both the diagrams and semantic information can be printed and distributed to team members. Some tools even support design versioning. Additionally, many tools generate class definitions, include files (used in C++), and complete source code from the information captured in their CASE repositories. Some tools help perform *what-if* impact analysis on the labor and costs required for system changes. Since most CASE tool buyers are existing developers who have outgrown one-person projects, many tools reverse-engineer existing code. Reverse engineering automatically creates diagrams and specifications from existing source code or database tables. "Marketeers" often bill such products as "full-cycle" CASE tools.

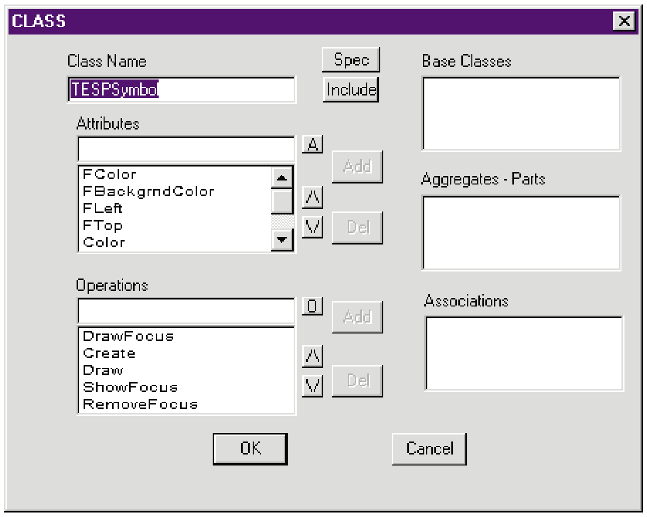
With Class

Recently, MicroGOLD released version 3.0 of their With Class product to the Delphi community. Since many developers have been waiting for an affordable CASE tool that supports both versions of Delphi, we arranged a test drive.

With Class includes a 100-page user manual, two diskettes, and an online tutorial and code generation primary. For an additional US$35, a larger printed tutorial is available. (I recommend the larger tutorial to learn all the features.)

With Class creates class, state, and object interaction diagrams that use Rumbaugh, Coad-Yourdon, Booch, and Shlaer-Mellor notations. It collects information on system, class, attribute (property), operation (method), state, transition, and object interactions. With Class generates source and SQL code, CASE reports from the With Class repository (forward engineering), and renders class diagrams from existing source code (reverse engineering). Through its scripting mechanism, With Class also enables the crafting of custom reports and code generators.





CLASS

Class Name: TESPSymbol

Attributes:
FColor
FBackgrndColor
FLeft
FTop
Color

Operations:
DrawFocus
Create
Draw
ShowFocus
RemoveFocus

Base Classes

Aggregates - Parts

Associations

Spec | Include | Add | Del | OK | Cancel

Figure 3 (Top): A customized class diagram with color added.
Figure 4 (Bottom): The CLASS dialog box.

(With Class includes scripts to support C++, Eiffel, Ada, Object Pascal, Smalltalk, Visual Basic, SQL, and Java.)

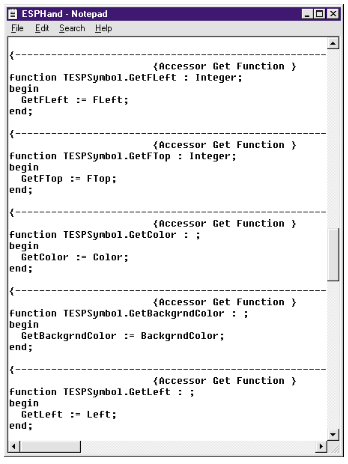
The Five-Minute Miracle

My initial five-minute test was simple: install With Class, point it at existing Delphi code, and run it. No studying manuals, no perusing online Help, and no peeking at the Tip of the Day. I wanted to test the reverse engineering as a beginner, so I took a quick dive in.

With Class passed my five-minute smoke test with flying colors. From my Delphi code, it produced an accurate class diagram without a single problem. After a minute of dragging things around, I had the diagram shown in Figure 1.

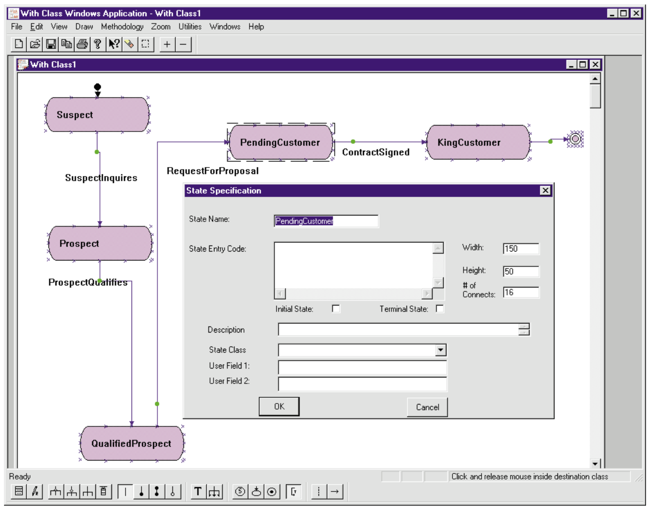
Since OOD methodologies are largely equivalent, With Class switches between them easily. After selecting the Booch methodology from With Class' main menu, a class diagram was produced in the Booch blobs (see Figure 2).

Other With Class options help you customize class diagrams and add color (see Figure 3). Double-clicking the *TESPSymbol* class displays the CLASS dialog box (Figure 4). Add another menu click, and With Class regenerates a class definition for *TESPSymbol*. A closer look at the generated source code revealed that With Class had generated the formal *Get* and *Set* methods that aren't in the original (see Figure 5). You can also customize the coding style. With Class' Code Generation dialog box allows you to change languages and port applications (e.g. from Visual Basic to Delphi). For example, prototype applications in Delphi will use the reverse engineering facility to generate Java code.



```
{-----------------------------------------------------------
                        {Accessor Get Function }
function TESPSymbol.GetFLeft : Integer;
begin
  GetFLeft := FLeft;
end;

{-----------------------------------------------------------
                        {Accessor Get Function }
function TESPSymbol.GetFTop : Integer;
begin
  GetFTop := FTop;
end;

{-----------------------------------------------------------
                        {Accessor Get Function }
function TESPSymbol.GetColor : ;
begin
  GetColor := Color;
end;

{-----------------------------------------------------------
                        {Accessor Get Function }
function TESPSymbol.GetBackgrndColor : ;
begin
  GetBackgrndColor := BackgrndColor;
end;

{-----------------------------------------------------------
                        {Accessor Get Function }
function TESPSymbol.GetLeft : ;
begin
  GetLeft := Left;
end;
```

Figure 5: Source code generated by With Class.

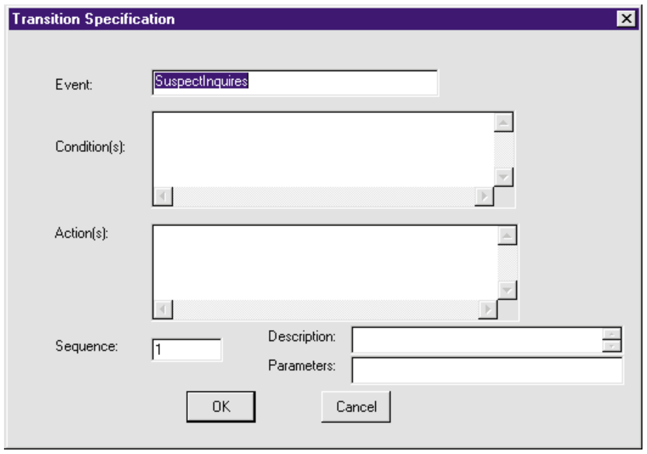


State Name: PendingCustomer
State Entry Code:
Width: 150
Height: 50
# of Connects: 16
Description
State Class
User Field 1:
User Field 2:

Figure 6: The simple state-transition diagram.

Modeling Dynamics

After you've mastered modeling static class diagrams, it's time to model system dynamics. In OOD, this means state-transition diagrams and object interaction diagrams, and With Class supports both. Figure 6 displays a simple state-transition diagram of the sales with the State Specification dialog box. Figure 7 shows the Transition Specification dialog box that displays when you double- click a transition line.



Event: SuspectInquires
Condition(s):
Action(s):
Sequence: 1
Description:
Parameters:

Figure 7: The Transition Specification dialog box that displays when you double-click a transition line.

Object interaction diagrams display the message passing (method invocation) that occurs dynamically between objects. The interaction diagram in Figure 8 illustrates a general menu option dispatcher executing the *change color* use case.

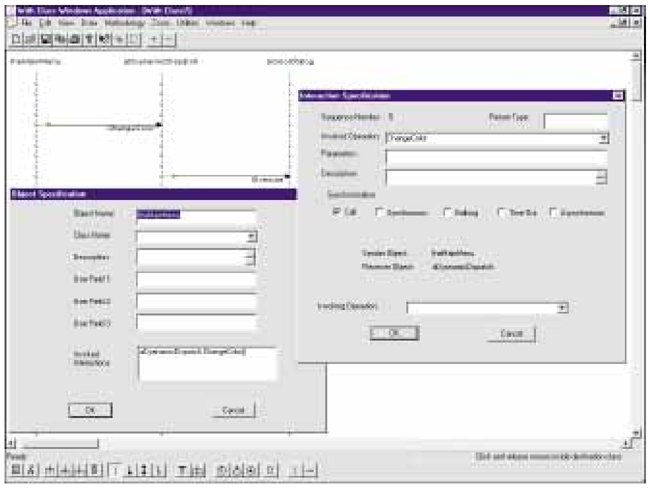


Figure 8: An interaction diagram that illustrates a general menu option dispatcher.

Open Architecture

A CASE tool's value increases proportionally to its usefulness for your project. Different projects and teams require varying degrees of design documentation; one size never fits all. Realizing this, MicroGOLD created a CASE framework to support popular methodologies, and opened its architecture. MicroGOLD achieved this in two ways:

- MicroGOLD created an OLE server. Using With Class' OLE services, you can embed With Class CASE diagrams in compound documents. With Class supports OLE2 in place activation of CASE diagrams embedded in other documents.
- With Class creates a parameter for each item that users enter, and uses them in a simple text-replacement merge tool to generate reports and code. Parametric programming is the general term for this technique. As shown in Figure 9, With Class calls its rendition *Scripting*.

By creating custom scripts, With Class collects design information, and through custom scripts, formats the information as you like. You can also export CASE information via scripts to word processors, databases, spreadsheets, or custom applications and tools.

Horrors in Heaven

With Class has an impressive array of basic CASE features, but the tool is not without flaws. Beginners beware. With Class' documentation is written in an academic style that isn't well-adapted for Delphi (With Class was originally written for C++ environments). You'll find the publication standards are below other similarly-priced software packages. However, a more extensive tutorial that most developers will find helpful can be purchased to straighten the learning curve. The good news is that the manuals are being redone. In the meantime,

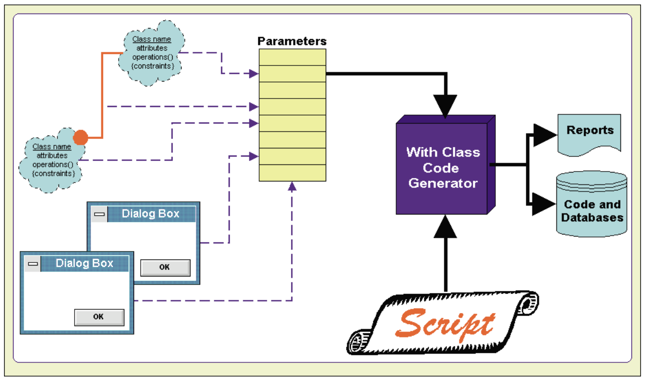


Figure 9 (Bottom): A diagram mapping Parametric programming.

prepare to labor to become acclimated with these CASE tools. You may also notice additional irritations in With Class. First, the drawing engine does not double buffer, so you get the pixellation effect as the images constantly redraw. It's fast, but you may go blind on a large project. This method of drawing causes occasional *ghost* images to appear, especially in Booch diagrams. Also, the multi-select feature might make you scream. It is used to cut and paste common design elements, and move designs on the canvas. It is neither intuitive, nor easy to use. Reworks to the user interface are also expected in future versions.

When generating code in With Class, be aware that its parse engine doesn't perform a merge generation. It either generates new files or overwrites existing files. Everything must be in the CASE tool and scripts. If not, you must create additional scripts and run a second pass to merge in any code that was manually added.

Conclusion

For US$295 (the 16-bit version retails for US$195) With Class ships with some surprising features, but the tool has flaws. Without better documentation, plan to spend a week or two being unproductive. To perform real work, you must anticipate and overlook With Class' user interface quirks. If you can ignore the irritations and learn With Class' scripting language, you'll produce some amazingly well-documented and carefully-engineered object-oriented applications. ∆

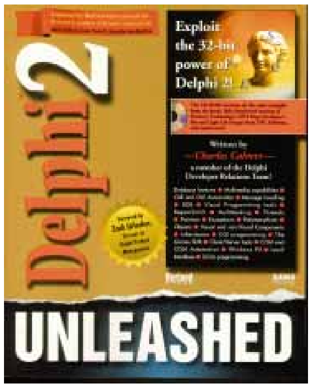
INFORMANT FACT FILE

For its price, With Class ships with some surprising features, and is capable of producing well-documented and carefully-engineered object-oriented applications. However, the documentation is poor, and the interface is quirky.

MicroGold Software, Inc.
76 Linda Lane,
Edison, NJ 08820
Phone: (908) 668-4779
Fax: (908) 668-4386
E-mail: 71543.1172@-compuserve.com
Web Site: http://www.microgold.com
Price: With Class 3.0 Professional, US$295; With Class 3.0 Enterprise, US$395.

James Callan, an 18-year computing veteran and former consulting director for Oracle Corp., is currently president of Gordian Solutions, Inc., an information technology consulting provider in Cary, NC. A frequent writer and speaker on information technology and client/server computing, James specializes in product design. He can be reached at (919) 460-0555, or 102533.2247@compuserve.com.

Delphi 2 Unleashed Aimed at Advanced Developers



It is not uncommon for a book published on one version of software to be updated and reprinted for a subsequent version of that software. Unless you take a look, you'll probably assume *Delphi 2 Unleashed* by Charles Calvert [SAMS Publishing, 1996] is merely an updated version of *Delphi Unleashed* [SAMS Publishing, 1995]. Upon inspection, however, you'll discover *Delphi 2 Unleashed* shares little with its highly successful predecessor.

Make no mistake, *Delphi 2 Unleashed* is a major new book on Delphi 2. But be forewarned, this book won't be much help if you're only working in Delphi 1. If you're not using Delphi 2, you'll be better served with a book that exclusively covers Delphi 1, or even one that covers both products.

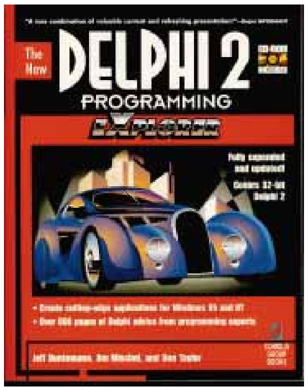
If you want to learn about Delphi 2, however, this book requires serious consideration. The fundamental reason is its extensive coverage of the Win32 API. If you're primarily a Windows 3.x developer, this material will give you a detailed, in-depth look at this powerful new environment. While you could consider a book strictly on Windows 95 or Windows NT programming, *Delphi 2 Unleashed* excels because it discusses these topics with respect to Delphi 2. And Charles Calvert is in a good position to discuss these issues. In addition to being the author of *Delphi Unleashed*, he is the author of *Teach Yourself Windows 95 Programming in 21 Days* [SAMS Publishing, 1995].

But the Win32 API coverage is just the beginning. This book also includes detailed discussions of the new Delphi 2 features, as well as a large section on game and Internet development. Other sections include discussions about Delphi's data types, creating and using databases, Delphi's object model, as well as OLE Automation and the Component Object Model.

In addition, *Delphi 2 Unleashed* ships with a CD loaded with code examples, demoware, shareware, and freeware. The CD also contains Microsoft Word for Windows .DOC files of the text of 15 chapters dropped from the original edition. However, these chapters are formatted for typesetting, and aren't nearly as easy to read as printed material. Consequently, if you are a Delphi 1 developer, you should get a copy of *Delphi Unleashed.*

"*Delphi 2 Unleashed Aimed at Advanced Developers*"
continued on page 47

The New Delphi 2 Programming EXplorer


The New
DELPHI 2
PROGRAMMING
EXPLORER
Fully expanded and updated!
Covers 32-bit Delphi 2
Jeff Duntemann, Jim Mischel, and Don Taylor

The New Delphi 2 Programming EXplorer by Jeff Duntemann, Jim Mischel, and Don Taylor [Coriolis Group Books, 1996] is a revised and expanded edition of *Delphi Programming EXplorer* [Coriolis Group Books, 1995]. The first edition was an excellent introduction to Delphi programming. In this edition, the authors have added 200 pages, covering SQL database programming, custom Delphi components, Windows messages, and Delphi exception handling. They have done a good job, and the book adheres to the standards of the first edition.

I reviewed the original edition twice in *Delphi Informant*: in August 1995 upon its initial release, and October 1995 when it shipped with a CD-ROM of Delphi-related material. In those reviews, I stated the authors had done a great job introducing Delphi to beginning programmers. The book was divided into three sections: the basic mechanics of using Delphi; object-oriented Windows programming with Delphi; and database application development with Delphi. The final section was presented in an atypical, yet refreshing manner: Taylor cast it as a detective story starring Ace Breakpoint, a hard-boiled private eye struggling to turn into a sensitive, new-age software consultant. This was a great way to cover a lot of ground quickly, in a manner that kept readers engaged.

The authors retained this successful formula in the second edition. The first three quarters of the book are essentially unaltered, although the authors added small sections describing features of Delphi 2 that differ significantly from Delphi 1. They also re-shot the screen captures to show the new look of Delphi 2 running under Windows 95, and corrected minor errors in the text and program listings.

"*The New Delphi 2 Programming EXplorer*"
continued on page 47

Delphi 2 Unleashed Aimed at Advanced Developers (cont.)

Delphi 2 Unleashed is aimed strictly at advanced developers because of its detailed discussions of many high-end topics. By comparison, *Delphi Unleashed* was an excellent book for beginning/intermediate Delphi developers, while still containing enough detail to satisfy advanced developers.

I must admit, as much as I like this book, there are several aspects I don't care for. The first is its size. The original edition, as well as several of its competitors, were big books. But they were nothing compared to *Delphi 2 Unleashed*. This book is huge and, I think, unwieldy. I simply cannot take this book on the road. At over five pounds, and a thickness exceeding two inches, it seems more like a weapon than reading material.

This book is also expensive. While you can easily get it for less than its US$59.99 cover price, it remains the most expensive mass-market computer book I've seen. Why isn't this two books? Let's get real. How many readers want to lug around a 1,400-page book when they're interested in only a few of the topics?

Finally, there is the writing style. If you enjoyed Calvert's chatty, conversational style in *Delphi Unleashed*, you won't be let down here. Some readers, however, will find this casual style of writing distracting and, given the size of this book, unnecessary.

But don't misunderstand me. *Delphi 2 Unleashed* makes an outstanding addition to the library of Delphi 2 developers, particularly those relatively new to the Win32 API. Although this book is big and expensive, it deserves your serious consideration. And don't be fooled by the title — this book may be for you, even if you own the earlier edition.

— *Cary Jensen, Ph.D.*

Delphi 2 Unleashed
by Charles Calvert, SAMS Publishing, 201 West 103rd Street, Indianapolis, IN 46290, (800) 428-5331.

ISBN: 0-672-30858-4
Price: US$59.99
(1,400 pages, CD-ROM)

The New Delphi 2 Programming EXplorer (cont.)

The new material starts in Part 3 (Chapter 15), where Mischel introduces SQL database programming concepts. In two well-written chapters, Mischel and Duntemann cover the basics of using SQL and Delphi. The first demonstrates managing data using SQL, the Borland Database Engine, and the *TQuery*, *TDBGrid*, and *TDataSource* components. For non-database Delphi programmers, this is a treasure-trove of well-illustrated basic concepts. The next chapter treats SQL as a database programming API, and provides in-depth coverage on how to use SQL with Delphi. It covers enough material to learn how to create a real working SQL database project. This section also contains a tutorial on creating custom Delphi components by deriving them from existing component classes. Mischel uses personal anecdotes to keep the reader interested in this moderately advanced subject.

The fourth part of *The New Delphi 2 Programming EXplorer* contains the complete text of Ace Breakpoint's Database Adventure. This description of the design and implementation of a complete database package covers eight chapters, and more than 200 pages. It is also an entertaining story; Ace is a tough guy who experiences some anxious moments as he tries to win a consulting contract — without losing his girlfriend.

The original Ace Breakpoint story was a refreshing way to learn a serious subject. I was delighted to see Taylor added two new Breakpoint mini-adventures to this edition. I won't reveal the outcome between Ace and his girlfriend, but I will say that the new adventures delve into some of the more advanced areas of Windows and Delphi. In Chapter 27, Ace explores the Windows messaging system. He uses it to create Delphi forms that are "aware" of each other; they can broadcast messages and commands to each other. While interesting, I think this chapter needs a more concrete example project; it works, but is unrelated to a practical application.

Chapter 28 does not suffer from such problems. It clearly shows how to work with one of the most useful parts of Delphi — the exception mechanism for handling errors. Using exceptions, you can trap any kind of error condition, from dividing by zero to typing non-numeric characters in the edit field of a financial application. However, because many older languages weren't designed with error trapping, it's an unfamiliar concept for many programmers. Adding error handling in languages such as C is messy and boring, so developers often exclude it from their program's first version. Or worse, they do an incomplete job, and their error handling fails. It's easy to *not* have good error handling in such languages; but with Delphi, it's easy to build it in, and this chapter demonstrates how.

I used to recommend the first edition of Duntemann, Mischel, and Taylor's book to anybody looking to get into Delphi programming. I don't do that anymore. Now I recommend the second edition. It's a fine upgrade to an already excellent work.

— *Tim Feldman*

The New Delphi 2 Programming EXplorer
by Jeff Duntemann, Jim Mischel, and Don Taylor, Coriolis Group Books, 7339 E. Acoma Drive, Suite 7, Scottsdale, AZ 85260, (800) 410-0192 or (602) 483-0192. Web Site: http://www.coriolis.com.

ISBN: 1-883577-72-1
Price: US$44.99
(806 pages, CD-ROM)

Borland's Annual Harbinger

The annual Borland Developers Conference (BDC) tends to be a harbinger of the company's standing each year. In 1994, a rather undisciplined keynote address by former CEO Phillipe Kahn typified the scatterbrained nature of the Borland of the early 1990s. 1995 was highlighted by a return to respectability for the company, along with unbridled enthusiasm for the newly-released Delphi. This year's BDC, held in Anaheim in July, showcased Borland's emerging technology in areas where the software industry is rapidly moving — Internet/intranet and multi-tier architectures. To sum it up, I'll discuss four main impressions I have of BDC '96.

Borland is well positioned for the emerging Internet marketplace. Its new Web-related products — Latté and IntraBuilder — drew a great deal of interest and enthusiasm at the conference. If this duo ships promptly, Borland can take a technological lead in the Internet tools market. Additionally, Borland is working closely with Sun and Netscape on several Web technology fronts. These relationships have produced tangible results for Borland, such as Sun's adoption of Borland's BAJA specification for the Java Beans Initiative, as well as Netscape's licensing of the Just-In-Time compiler for Navigator 3.0. Ironically, while firmly entrenched in the Sun/Netscape camp, Borland may have more to gain if Microsoft succeeds in the Web wars, due to their Windows orientation with Delphi, IntraBuilder, and other products.

Delphi looks well poised to tackle the next generation of applications. After working with Delphi 2, my immediate impression was "Where could Borland possibly take this product?" Besides minor enhancements, how much more do you really need in a desktop development environment? My horizons were broadened when I heard where they are taking the next version, called Delphi 97. Extending beyond the desktop and two-tier client/server models, Delphi 97 aims to bring multi-tier application servers, COM (the Component Object Model), and Internet deployment into the mainstream. You'll learn more about Delphi 97 next month, but suffice to say that as application architectures become more complex, Delphi will look to encapsulate such technology into its environment.

The technology gap between software vendors and customers appears to be growing. By the nature of the business, software vendors will always be ahead of their customers on the technology curve. This year, the chasm between Borland, Netscape, and Microsoft and their customers seems larger than ever. You have probably heard that an Internet year is equal to three calendar months because of the rapid pace of technology innovation in that market. While vendors race for the leading edge, many customers are still dealing with issues that are far more mundane. For example, in one session I led on migrating to Delphi 2, the majority of developers in the audience were still building 16-bit applications. Borland's current online survey of whether to develop a new 16-bit version of Delphi is a sign of the company's realization that the market is not moving as fast as they'd like.

Delphi is maturing in the marketplace. The task of establishing itself in a market ruled by Visual Basic and PowerBuilder has been daunting, but Delphi is making definite inroads. Not only was this the largest BDC ever, of the 2,500 participants, 75 percent of them attended Delphi sessions. Another sign of a product's acceptance is employment opportunities. While newspapers continue to be dominated by PowerBuilder advertisements, it was encouraging to see many more Delphi job postings than at BDC '95, including a representative of a Fortune 100 firm who intended to hire several hundred Delphi developers during the conference. ∆

— Richard Wagner

Visit the "File | New" home page at http://www.acadians.com/filenew/filenew.htm. In addition to downloading past articles, you can get the latest tips and information from the world of software development.

Richard Wagner is Contributing Editor to Delphi Informant *and Chief Technology Officer of Acadia Software in the Boston, MA area. He welcomes your comments at rwagner@acadians.com, or on the File | New home page at http://www.acadians.com/filenew/filenew.htm.*