April 1998, Volume 4, Number 4

Cover Art By: *Doug Smith*

File Notification

VB or Delphi: Which Does COM Automation Best?

ON THE COVER

FEATURES

REVIEWS

DEPARTMENTS

Delphi TOOLS

New Products and Solutions

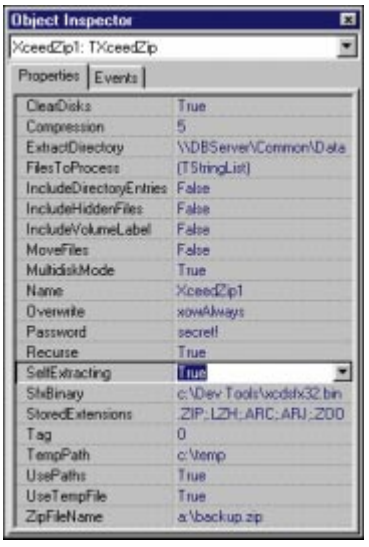
Xceed Releases Zip Compression Library and Self-Extractor Module

Xceed Software, Inc. announced the release of two products, the *Xceed Zip Compression Library 3.0* and the *Xceed Zip Self-Extractor Module 1.0*.

The new version of the Xceed Zip Compression Library provides enhanced support for 16- and 32-bit Windows development environments and features memory compression, global status reports, file renaming, and creation of self-extracting .ZIP files. Also, a Borland C++Builder component is added to the VBX, OCX, and Delphi VCLs, along with sample applications for Microsoft Word 97, Microsoft Access 97, and C++Builder.

The Xceed Zip Self-Extractor Module is an add-on module for the Xceed Zip Compression Library that allows developers to create customized self-extracting .ZIP files from within an application. The module enables developers to customize the title, introduction message, all text and button captions, default extract path, and default overwrite behavior. Other features include the ability to span disks, a real-time status window, and a prompt for decryption passwords.

Xceed Software, Inc.
Price: US$199.95 each; US$299.95 for both, if purchased together by April 30, 1998. All purchases include free technical support and a 60-day money-back guarantee.
Phone: (800) 865-2626 or (514) 442-2626
Web Site: http://www.xceedsoft.com



Pythoness Releases PSetting

Pythoness Software has announced *PSetting*, persistent storage components for real-world applications.

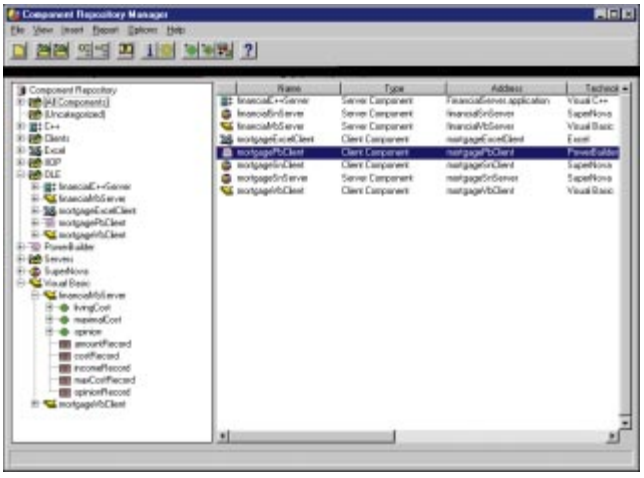
PSetting provides basic features such as automatic form position storage, MRU lists, and component properties. In addition, PSetting offers a storage mechanism, allowing a developer to save generic settings. PSetting also enables developers to specify where to store information, using the Windows registry, a TStream, an IStorage (OLE structured storage file), or any programmer-defined location. PSetting allows instant correlation of related settings between forms, so that only a line or two of code is required to update all forms.

PSetting is available for Delphi 2 and 3 and includes source code.

Pythoness Software
Price: US$69
Phone: (208) 359-1540
Web Site: http://www.pythoness.com

SuperNova Releases SuperNova/Visual Concepts

SuperNova, Inc. announced *SuperNova/Visual Concepts*, a component assembly environment that allows organizations to combine components written in any development language and deploy component-based applications across virtually any platform.



SuperNova/Visual Concepts is based on SuperNova's core virtual machine technology, which enables the product to incorporate components developed for more than 20 hardware/OS platforms and more than 25 commercial databases. Components can be written in any development language that supports ActiveX, OLE/COM, or CORBA/IIOP, including Delphi, Visual Basic, Java, C, C++, PowerBuilder, or SuperNova/Application Developer.

The SuperNova/Visual Concepts environment includes a graphical component repository that allows developers to identify components by language, platform, or business functionality; a drag-and-drop application-modeling environment; automatic generation of interfaces between components; a component deployment manager; and a component execution broker.

SuperNova, Inc.
Price: From US$19,995
Phone: (732) 248-6667
Web Site: http://www.supernova.com

table of CONTENTS

Delphi TOOLS

New Products and Solutions

TurboPower Announces Async Professional 2.5

TurboPower Software Co. announced *Async Professional 2.5*, the company's communications and faxing toolkit for developers using any version of Borland Delphi and C++Builder. Version 2.5 adds support for DTMF tone detection, .WAV file record and playback, and fax detect and hand-off.

With Async Professional 2.5, developers can create complex voice mail applications, automated help systems, fax-back systems, and other business applications.

The new version also includes AT-compatible ISDN support for high-speed communications, support for the RS-485 communications standard, user-defined data triggers, a redistributable fax cover page editor, enhanced Caller ID support, and revised and expanded documentation.

Async Professional ships with full source code (written in Delphi) and requires no royalties.

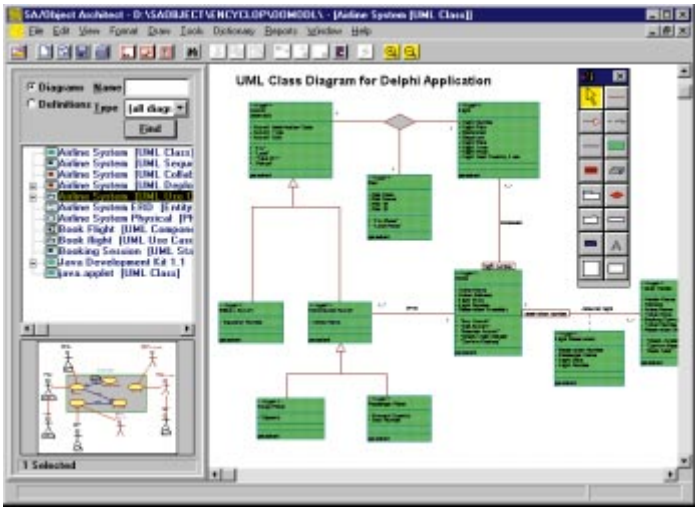
TurboPower Software Co.
Price: US$279; registered owners of Async Pro 2.x can upgrade for US$79; owners of Async Pro 1.x, or Async Pro Pascal Edition can upgrade for US$129.
Phone: (800) 333-4160
Web Site: http://www.turbopower.com

Popkin Releases SA/Object Architect 4

Popkin Software & Systems, Inc. has upgraded its System Architect family of repository-based client/server applications modeling tools with *SA/Object Architect 4*, offering a shared repository, customizable features, and scalability.

The SA/Object Architect 4 includes automatic generation of logical data models from object-oriented class models (and the reverse), so legacy systems can be migrated to the OO method. SA/Object Architect 4 generates C++, Smalltalk, Java, CORBA IDL, and HTML. It also allows point-and-click generation of Delphi forms, Visual Basic, and PowerBuilder data windows from the data model.

In addition, SA/Object Architect 4 has two-way links to Persistence Software and Magic Software, as well as two-way interfaces to the Microsoft Repository and Microsoft Visual Modeler.



Popkin Software & Systems, Inc.
Price: First license for stand-alone version, US$2,195; network version (minimum of two copies), US$2,395; discounts are available for multiple licenses; annual support for stand-alone and network versions, US$439 and US$479, respectively.
Phone: (800) 732-5227 or (212) 571-3434
Web Site: http://www.popkin.com

News LINE

April 1998

Borland Appoints Zack Urlocker Vice President of Marketing

Scotts Valley, CA — Borland announced the promotion of Zack Urlocker to Vice President of Marketing. Urlocker, who was previously Vice President of Product Management, is now responsible for Borland's worldwide marketing activities, including corporate marketing and corporate communications. He reports to Borland Chairman and CEO Delbert W. Yocam.

Urlocker has been with Borland for seven years. Prior to heading product management, he served as director of Delphi Product Management where he was involved in the development and introduction of Delphi. Prior to becoming director of Delphi Product Management, Urlocker served as a product manager in Borland's languages group. Before joining Borland, Urlocker was the manager of developer relations at The Whitewater Group.

Borland also announced the promotion of Mark de Visser to Vice President of Marketing Communications. De Visser, who will now report to Urlocker, is responsible for advertising, collateral, creative services, and electronic marketing.

Borland and Cayenne Software Announce Support for Delphi Client/Server Suite

Scotts Valley, CA — Borland announced that its Delphi Client/Server Suite is supported by Cayenne Software's ObjectTeam 7, a component modeling tool that is compliant with version 1.0 of the Object Management Group's Unified Modeling Language (OMG UML). ObjectTeam 7 integrates with Borland's Delphi Client/Server Suite by generating Delphi 2 and Delphi 3 code, supporting the reverse engineering and generation of large-scale, multi-tier Delphi applications.

ObjectTeam 7 allows teams of systems analysts and designers to build object-oriented applications. It includes editors for OMG UML 1.0 Use Case, Collaboration, Class, Sequence, and State diagrams. Support for OMG UML 1.0 allows organizations to standardize a common modeling language for specifying, designing, and documenting large and complex systems.

BSC Polska to Become Master Distributor of Borland Products in Poland

Frankfurt, Germany — Borland EMM (Eastern Europe - Mediterranean - Middle East) announced that BSC Polska is now Borland's master distributor in Poland.

BSC Polska was founded in November 1996 to provide professional technical services for Borland users. Its activities include support, training, development, and consulting. BSC Polska is supported by Borland's European Technical Team, directly associated with Borland Research & Development.

With this step, Borland confirms its commitment to its customers in Poland. BSC Polska will use the services of the current Borland distribution network represented by ABC Data, MSP, and Softpoint.

For additional information on Borland's presence in Poland, and BSC services, developers, customers, and corporation, visit http://www.bsc.-com.pl.

Borland Ships Translation Tools for Delphi

Scotts Valley, CA — Borland announced new versions of two tools for translating Delphi applications into international languages: the Delphi 3 Language Pack and the Delphi Translation Suite 3.0. These tools provide a way to localize Delphi applications into Danish, Dutch, English, French, German, Italian, Portuguese, Spanish, and Swedish.

The Delphi 3 Language Pack includes already-translated system messages for Delphi's VCLs and the BDE (Borland Database Engine), as well as translated message templates for Delphi forms, dialogs boxes, and menus. It also includes the Delphi Language Manager, which allows developers to switch between languages.

The Delphi Translation Suite 3.0 allows ISVs, VARs, and system integrators to deliver larger-scale applications internationally with a thorough translation. The suite can help translate new or existing applications, and includes all the functionality of the Delphi 3 Language Pack, plus a suite of development tools to automate and test the translation process from a single code base.

Delphi 3 Language Pack is available for US$199.95. Delphi Translation Suite 3.0 is available for US$2,499.95. For more information, or to order, call (800) 233-2444.

table of CONTENTS

By Bruce McKinney

File Notification

VB or Delphi: Which Does COM Automation Best?

Try this experiment: Run Windows Explorer in one window; then go to a command-line session or another instance of Windows Explorer and delete a file in the directory shown by the original Explorer. Watch what happens: There's a short pause, and then the file disappears from the first Explorer window.

How does Windows Explorer know about a file changed by another program? The same way your programs can know: by using the Windows API functions *FindFirstChangeNotification*, *FindNextChangeNotification*, and *FindCloseChangeNotification*. This article presents an Automation object that watches for changes in the file system using these API functions and broadcasts the details to interested clients. (Just to make sure we're talking the same language, a Delphi Automation object is what some environments call an ActiveX EXE server.)

We'll be discussing several parts of this problem. The easy part is using the file notification functions. The difficult part is making the connection between clients and server using COM Automation. Finally, I'll have a little bit to say about the unusual origins of my Notify server, which was originally written in Visual Basic (VB).

From VB to Delphi

Without giving a complete résumé, I am the author of an advanced VB book called *Hardcore Visual Basic* [Microsoft Press, 1997]. Without getting into a lot of criticism of a competing product, I was disappointed in VB 5 and weary after writing a book full of hacks to work around limitations of the language. Without getting into my personal history, my first compiler was Turbo Pascal 1.0 — a revolutionary product that changed the world and started me on the road to becoming a professional programmer.

Given those facts, experimenting with Delphi seemed like a natural next step. When I tried it a few months ago, the language felt like a long-lost friend, but a friend who has been through several wars, famines, and alien visitations since our last meeting. There were times when I felt daunted and confused, but not intimidated enough to avoid a foolish mistake: I volunteered to write an article about translating a program from a language I knew intimately to a language I didn't know nearly as well as I thought.

This article is the result. It focuses on COM, Automation, and file notification, but we'll take a few side trips to look at the technical and philosophical differences between the VB and Delphi implementations of COM. I'll give you an early hint. We're not going to find that Delphi is good and VB is bad, or vice versa. They just have very different COM philosophies.

The Client Side of File Notification

The File Notification Server (Notify.exe) encapsulates file change notification in an Automation server that you can use from any program. You can see how it works in the sample program (available for download; see end of article for details), Test Notify (TNotify.exe). The sample simply deletes, renames, creates, and saves files in the \Temp directory. The Notify server detects these changes and sends events reporting them back to the client.

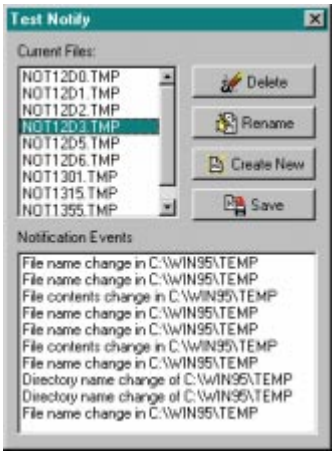


Figure 1: The sample application at run time.

Figure 1 shows the sample with a list of files at the top and a list of file events at the bottom. The buttons to allow changing the files in the test program are a testing convenience. Any changes made to files in the \Temp directory will show up in the event list. You can test this by modifying files with Windows Explorer while the sample is running. You may also want to run multiple copies of the sample to test whether the server can handle multiple clients. If you're really ambitious, you could write a similar sample client for the Delphi server in VB, or you could write a Delphi client for the VB version provided with my book.

In our first view of the problem, we're going to think of the server as a magical black box. By looking at how the client connects, we can see what the server does. Later, we'll see how it works.

After the Notify server has been run at least once, its type library is listed in the system registry. To use its classes and interfaces, we would normally read their definitions using the **Import Type Library** command from the **Project** menu. This would bring up a dialog box where we would select the Notify server from a list of registered type libraries. The result would be the creation of a new unit called Notify_TLB located somewhere deep in the \Program Files\Borland directory. But we're not going to follow the normal procedure, which is based on the assumption that the client developer may not have access to the source code of the server. In this case, we are developing both the client and the server, and part of the process of building the type library for the server is to create a unit called Notify_TLB in the current directory. Since this unit will be identical to the one we would create by importing the type library, we'll use it instead (so that the code supplied with the article can be complete).

We'll see how the file is created shortly. For now, we're interested in the contents of the Notify_TLB unit. The key definitions are a class representing the Notify server, and an interface that will be used to create notification events in the client. The class appears as *TNotify* in the server, but it is imported as a separate interface *INotify* and CoClass *CoNotify*. The interface looks like this:

```
INotify = interface(IDispatch)
  ['{ 64624092-5B32-11D1-8193-000000000000 }']
  function Connect(const notifier: INotifier; const
    sDir: WideString; iMode: Integer; fSubTree: WordBool):
    Integer; safecall;
  procedure Disconnect(hNotify: Integer); safecall;
end;
```

The CoClass provides *Create* and *CreateRemote* methods that you use to hook up the interface. The first thing the *FormCreate* procedure of the test form does is to hook up the *notify* variable (type *INotify*) using the CoClass:

```
notify : INotify;
...
notify := CoNotify.Create;
```

This is the first big difference between Delphi and VB. In VB the class used in the client is exactly the same as the class defined in the server. You have to add the type library of the server to your project, but then you simply declare an object variable of the class type:

```
Private notify As CNotify
...
Set notify = New CNotify
```

You don't have to initialize the object with CoClass and use it with an interface. Of course, VB is actually doing the same thing under the surface, but it hides the details. VB can do this because its primary purpose as a language is to wrap COM. Delphi is a general purpose language that supports COM as an optional feature. That's why you'll generally find that if COM doesn't support a feature (such as constructors or inheritance), VB doesn't support it either. Delphi classes aren't COM classes unless you specify the necessary COM support. VB classes are always COM classes, whether you need COM or not. With VB you can only do the most important COM operations that VB supports directly. With Delphi, you can do any COM operation, no matter how obscure. Of course, you can make Delphi COM operations easier by using standard COM classes and interfaces, and by using the Type Library Editor and other automated features.

Connecting to the Server

The *notify* variable represents the EXE server that will use Win32 file notification API functions to watch for any changes to files or directories. The client has to tell the server which directories to watch and what to watch for by calling the *Connect* and *Disconnect* methods. This code, in the *FormCreate* method, disconnects the old directory (if there is one) and connects a new one:

```
with notify do begin
  // Watch for directory changes.
  hNotifyDir := Connect(notifier, sTemp,
                  FILE_NOTIFY_CHANGE_DIR_NAME, False);
  // Watch for name changes (delete, rename, create).
  hNotifyFile := Connect(notifier, sTemp,
                   FILE_NOTIFY_CHANGE_FILE_NAME, False);
  // Watch modifications of file contents.
  hNotifyChange := Connect(notifier, sTemp,
                     FILE_NOTIFY_CHANGE_LAST_WRITE, False);
end;
```

The first parameter of the *Connect* method is an *INotifier* variable (which I'll get to in a minute). The second parameter is the directory to be watched, in this case the \Temp directory. You can check the code to see how I got it from the *GetEnvironmentVariable* API function. The third parameter tells what kind of changes to look for. The FILE_NOTIFY... constants are defined in the Windows

unit as bitfields. You could combine several of them with the plus operator, but I have made three different kinds of connections so that when the notifications come back, I'll be able to tell what kind of event happened. The last Boolean parameter indicates whether to check child directories. Windows 95 doesn't support True, so you should supply False unless your program will run only on Windows NT. The *Connect* method returns a notification handle, which you must save to pass to the *Disconnect* method.

The first connection looks for changes to directories, including any that have been created, removed, or renamed. The next connection looks for any files whose names have changed in the current directory. Deleting a file or creating a new file obviously changes the file name. The third connection looks for files that have been modified. For example, if you modify a text file in the \Temp directory with Notepad, you'll get a notification. The notifications just tell you that a change has occurred. They don't say what file or directory changed. In most programs you can just update your view of the directory, but if you really need to know exactly what file or directory changed, you can usually figure it out by checking file dates. Windows NT supports identifying files changes with the *ReadDirectoryChanges* API, but that's beyond the scope of this article.

Getting Events from the Server

Now that we've told the server what we want it to watch, how will we get back the information? The obvious answer is through events. When the server receives a notification from the operating system, it should create an event to report the results. Clients hooked up to the server should receive those events. The problem is that the normal COM event mechanism doesn't work automatically in clients of Automation objects the way it does in clients of ActiveX controls. You can teach your clients how to receive standard events using Delphi's *TConnectionPoint* class, but I won't get into that. For this particular problem I prefer a simpler method of receiving events: interface callbacks.

An interface defined by the server provides a common method of communication for the clients. The clients get the interface from the type library and implement it to do whatever client-specific tasks they want. They pass an interface pointer to the server, which uses it to call back methods in the client. The server doesn't know or care how the interface is implemented, just that it provides a standard way of passing back data. Here's the *INotifier* interface defined by the Notify server:

```
INotifier = interface(IDispatch)
  ['{ 64624094-5B32-11D1-8193-000000000000 }']
  procedure Change(const sDir: WideString; iMode: Integer;
                   fSubTree: WordBool); safecall;
end;
```

The interface has only one method, *Change*. It is designed to pass back some of the same data passed in the *Connection* method so that a client with several connections can determine which one is coming back.

In Delphi you implement an interface by deriving a new class from it. The new class should also inherit from one of the library classes that knows how to handle interfaces. The most common of these is *TInterfacedObject*, but when dealing with Automation interfaces (those derived from *IDispatch*) you usually have to use a descendant called *TAutoIntfObject*. So the class that we will implement looks like this:

```
TNotifier = class(TAutoIntfObject, INotifier)
  procedure Change(const sDir: WideString; iMode: Integer;
                   fSubTree: WordBool); safecall;
end;
```

The *Change* procedure can be implemented to do whatever the client needs to do. The sample program simply sends a string describing the data to a listbox:

```
procedure TNotifier.Change(const sDir: WideString;
   iMode: Integer; SubTree: WordBool); safecall;
var
  s : string;
begin
  case iMode of
    FILE_NOTIFY_CHANGE_DIR_NAME:
      s := 'Directory name change of ' + sDir;
    FILE_NOTIFY_CHANGE_FILE_NAME:
      s := 'File name change in ' + sDir;
    FILE_NOTIFY_CHANGE_LAST_WRITE:
      s := 'File contents change in ' + sDir;
  end;
  TestNotify.lstMessage.Items.Add(s);
end;
```

For this implementation to be called by the server, an *INotifier* variable must be declared, constructed, and passed as the first parameter of the *Connect* method (as we saw earlier). The construction step is the difficult one, due to negligent documentation of the *TAutoIntfObject.Create* constructor. The Help file shows the following signature, but fails to mention what the two parameters mean:

```
constructor Create(const TypeLib: ITypeLib;
                   const DispIntf: TGUID);
```

After many hours of research, I finally got the answers from programmers on one of the Borland newsgroups (thanks Reggie Chen and Darren Clark). The *TypeLib* parameter must be an *ITypeLib* variable generated by the *LoadRegTypeLib* API function based on the library GUID from the imported type library. The *DispIntf* parameter must be the interface being created:

```
hr := LoadRegTypeLib(LIBID_Notify, 1, 0, 0, typelib);
notifier := TNotifier.Create(typelib, INotifier);
// Handle any construction failure.
```

Although the signature for the constructor indicates the *DispIntf* parameter is a *TGUID*, Delphi actually accepts the interface name and automatically converts it to a GUID.

This is a complicated mess by VB standards. Implementing an interface is as simple as declaring it with the **Implements** keyword:

```
Implements INotifier
```

VB automatically creates empty procedures for the methods and properties defined by the interface:

```
Private Sub IFileNotifier_Change(sDir As String, _
  IMode As Long, fSubTree As Boolean)
  ' Implementation code goes here.
End Sub
```

You just fill in the blanks. So from the client side, there's not much doubt that VB has a simpler, more intuitive syntax than Delphi. The winner isn't as clear on the server side.

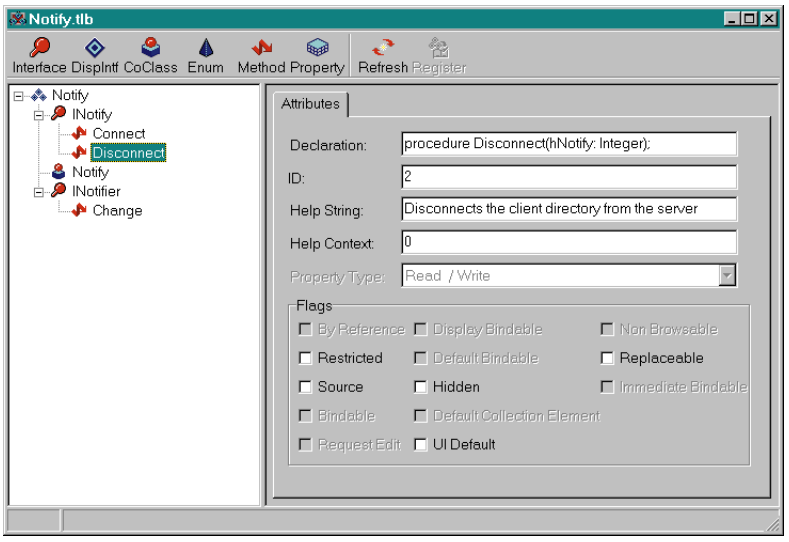


Figure 2: Delphi 3's Type Library Editor.

The Server Side of File Notification

The File Notification Server is a slow EXE server, not a fast DLL server. Why? Because a DLL server would be too efficient. Performance isn't really an issue with file notification. The more important issue is making sure the search for notifications doesn't slow down normal operation of the clients. The best way to make sure it doesn't is to put notification detection in a separate thread, and the easiest way to get it into a separate thread is to put it in a separate EXE file.

This was the only reasonable option in the original VB version, but Delphi opens up more possibilities, including separate threads to handle file notification inside a DLL server. Still, I'm not sure that would really be a better system. With a DLL you'd have a separate copy of the server in memory for each client. With an EXE, you have only one program in memory. Furthermore, writing multi-threaded ActiveX servers requires a knowledge of threading models far beyond the scope of this article.

When writing an Automation object in VB, you can just plunge in and start writing code. In Delphi, it's better to follow a sequence of steps that lets wizards write a code shell for you. As a bonus, this process forces you to think carefully about the design.

The first step is to create an empty application by selecting **New Application** from the **File** menu. Delphi creates the application shell, including a startup form you can discard, using the Project Manager. This server will have no user interface, so it has no need for a form. Next choose **New** from the **File** menu, press the ActiveX tab on the resulting dialog box, and select **Automation Object** from the list of application types. This starts the Automation Object Wizard, which asks you for a class name and an instancing mode for the object. Name the class `Notify`; the wizard will create *TNotify*, *INotify*, *CoNotify*, and other variations where appropriate. Accept the default instancing mode, multiple, to allow more than one client to connect to the same server. At this point I saved all the files, naming the project file `Notify` (so the server will be named Notify.exe) and the Automation unit `NotifySvr`.

Into the Type Library Editor

The next step is to open the Type Library Editor and start defining the classes and interfaces the Automation object will export. You've already seen a client's side view of the resulting class and interface. Figure 2 shows the same thing in the Type Library Editor. It takes the form of a hierarchy with the library at the top level. The second level has interfaces, dispinterfaces, CoClasses, and enums (we'll ignore enums and dispinterfaces). The third level has methods and properties inside the interfaces.

When you first start the Type Library Editor, the Automation Object Wizard has already created the *Notify* CoClass and the *INotify* interface. You just need to add the *Connect* and *Disconnect* methods to *INotify* by clicking the **Method** button and typing in the correct method names. The Type Library Editor isn't the smartest wizard you'll ever encounter. It will always give you procedures with no arguments when you press the **Method** button. You must add the parameters and change procedures to functions where appropriate. If I had designed the Type Library Editor, it would have a fourth level where GUI doodads would help you add valid parameters and return types. Delphi's editor allows you to type any definition, but then generates errors if your entry doesn't follow the Automation rules.

The Automation Object Wizard knows about the *TNotify* class, but it has no idea about the *INotifier* interface. You must add it in the Type Library Editor by pressing the **Interface** button, naming the interface, and adding the *Change* method.

If you click on various parts of the type library hierarchy, you'll see that the Type Library Editor has checkboxes for flags and fields for text and numeric data. I accepted the defaults for all these values except the **Help String** fields. It's important to add descriptive help strings because many users will examine your Automation objects in object browsers. Of course, you should also provide a help file with complete documentation. In fact, a really smart Type Library Editor would have additional fields where you could type in help topics instead of help contexts. It would automatically generate the help file at the same time as the type library. Delphi doesn't have these features, and that's my excuse for not supplying a help file with the Notify server.

Once you have entered all the desired values in the Type Library Editor, press the **Refresh** button to generate all the

necessary code in the Notify and Notify_TLB units. Don't mess with the contents of the Notify_TLB file. This is essentially the same file that the Import Type Library command will create in client programs, so you don't want it to change. The Type Library Editor will also create the *TNotify* class in the NotifySvr Unit. The contents of this class are the same as the *INotify* class shown earlier, but the class declaration is interesting:

```
TNotify = class(TAutoObject, INotify)
```

The class is inherited from *TAutoObject*, which implements *IDispatch* and a bunch of other COM stuff that you don't really want to know about — except that it's this implementation that needs the *INotifyDisp* dispinterface, the CoClass constructors, and other details that you will see if you study the contents of the Notify_TLB unit.

Implementation at Last

At last we've finished the description of what the Notify server looks like and can get down to how it works. Here's the short version. The *Connect* method stores all the data passed to it in arrays. The *Disconnect* method removes the array data for a specified stored connection. Meanwhile, the main dispatch loop periodically calls the system to check for file changes in the watched directories. When it receives a notification, it looks up the corresponding array data and uses the found interface object to pass the rest of the data back to the client.

Like most algorithms, this one is built on its data structures. The data for each connection is stored in the following record:

```
TConnection = record
  sDir : WideString;
  iMode : Integer;
  fSubTree : WordBool;
  notifier : INotifier;
end;
```

Notice this is exactly the same data passed by the *Connect* method. We'll need an array to store these records, and a parallel array to store the handles for notification objects:

```
aconNotify : array[0..cLastNotify] of TConnection;
ahNotify: array[0..cLastNotify] of Integer =
            ( -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
              -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
              -1, -1, -1, -1, -1, -1, -1, -1, -1, -1);
```

You might wonder why I didn't add *hNotify* as a field of the *TConnection* record and store all the data in a single array. Two reasons. We have to initialize all those handles to -1, and it's easier to do in a separate array. But more importantly, the API function that examines those handles needs them all lined up in an array with nothing in between. By the way, don't take the ability to initialize arrays for granted. This simple pleasure is denied to VB programmers.

I use arrays with a fixed size because it's easier, but you could probably modify the program to use dynamic arrays with enough extra code. I don't foresee the Notify server handling more than 28 connection requests at a time, but if you can, it's easy to increase the size of the arrays by changing the cLastNotify constant.

Figure 3 shows how the *Connect* method initializes the data for each notification request. It looks for a blank slot in the array. When it finds one, it initializes the slot with a file notification handle obtained by calling the *FindFirstChangeNotification* API function. If everything goes well, *Connect* stores the handle returned by the API function and the data passed in as parameters in the appropriate arrays. I'll let you look through the code supplied on the Informant Web site to see how *Disconnect* undoes the work done by *Connect*.

There are additional points I'd like to make about the connect code. First, notice the double cast on the directory string passed to *FindFirstChangeNotification*. You may have noticed that the directory string, *sDir*, is a WideString rather than a **string**, and the *fSubTree* flag is a WordBool instead of a Boolean. This causes some inconvenience, but **string** and Boolean are not among the official recognized Automation types. Blame it on VB, which was the source of most of the Automation standards. *Connect* receives a WideString, but *FindFirstChangeNotification* (like most API string functions) expects a PChar. You can't cast directly from a WideString to a PChar, but you can cast from a WideString to a **string**, and from a **string** to a PChar.

```
function TNotify.Connect(const notifier: INotifier;
  const sDir: WideString; iMode: Integer;
  fSubTree: WordBool): Integer;
var
  i, h, e : Integer;
begin
  Result := -1;         // Assume fail.
  fConnected := True;  // At least one connection.
  // Find blank handle space.
  for i := 0 to cLastNotify do
    if ahNotify[i] = -1 then
      begin
        // Set up notification.
        h := FindFirstChangeNotification(
               PChar(String(sDir)), fSubTree, iMode);
        if h = -1 then
          begin
            e := GetLastError;
            if e <> ERROR_NOT_SUPPORTED then
              // Notification not supported on remote disks.
              RaiseError(errRemoteNotSupported)
            else
              // Unknown error.
              RaiseError(errInvalidArgument);
          end;
        // Store information.
        ahNotify[i] := h;
        aconNotify[i].notifier := notifier;
        aconNotify[i].sDir := sDir;
        aconNotify[i].iMode := iMode;
        aconNotify[i].fSubTree := fSubTree;
        Result := h;
        Exit;
      end;
  // If we didn't exit, we're out of array space.
  RaiseError(errTooManyNotifications);
end;
```

Figure 3: The *Connect* method.

The second point is that exceptions in Automation servers must be raised through the *EOleException* class. These exceptions are raised in the server, but received in the client. We don't want an invalid argument from one client bringing down the server and thus terminating a valid connection with another client. Here's the *RaiseError* procedure I use to generate errors in the required format:

```
procedure RaiseError(iErr : Integer);
begin
  raise EOleException.Create(aerr[iErr], 5550 + iErr,
                             'Notify, '', 0);
end;
```

I have constants for each of the errors that the Notify server can raise. These constants are indexes into an array of error messages. My *RaiseError* procedure uses the constants and the array to generate an error message and an error number that I hope won't conflict with errors from other COM components being used by the client. If there is a conflict, the server name passed in the third parameter can be used to distinguish the error codes. The fourth and fifth parameters are the *HelpFile* and *HelpContext*, which I didn't have the decency to create for this server.

Waiting for Events, but Where?

The important point about the *Connect* method is that *FindFirstChangeNotification* asks the Windows kernel to watch for certain file events. All we need to do is wait for the event to happen. But wait where? Who will the kernel notify when it gets a file event? It can't wait in the *Connect* event (which must return to the client), but where else can it go? I had a lot of trouble with this in VB, so I wasn't surprised when I encountered similar difficulties in Delphi. Once I figured it out, however, the Delphi solution proved to be much cleaner than my VB hacks.

Remember that earlier we deleted the form automatically added by Delphi because the server has no user interface. That leaves the following code in the project file:

```
begin
  Application.Initialize;
  Application.Run;
end.
```

Application.Run starts the program's main window message loop, which dispatches messages to the program's windows. But this program has no windows. It won't receive many messages. It doesn't need a message loop. What it needs instead is a loop to wait for notifications. So I replaced *Application.Run* with my own loop:

```
begin
  Application.Initialize;
  NotifySvr.WaitForNotify;
end.
```

This is one trick you'll never get away with in VB. Since VB wouldn't let me mess with its message loop, I had to trick it by starting a Windows timer that immediately killed itself and launched my message loop. A language that tries to anticipate your every need is great as long as it guesses correctly, but when it guesses wrong, you're in trouble. One of Delphi's strengths is that while it does guess about what you want, it lets you override the default if it guesses wrong.

With the *WaitForNotify* procedure, we've finally gotten to the heart of the Notify server. Figure 4 shows the code. It's just a loop that starts with a call to *WaitForMultipleObjects*, which waits for what Windows calls an object. A Win32 object can be a process, a thread, a mutex, an event, a semaphore, console input, or the one we're interested in, a change notification. Whatever you're waiting on, you must put its handles in a contiguous array and pass the number of objects, followed by the address of the first object. You must also indicate whether you want to wait until all objects have returned or wait only for the first one. In this case, you pass False to wait for the first file notification object. Finally, you pass the timeout period, 200 milliseconds.

When I say that *WaitForMultipleObjects* waits, I mean that literally. As soon as the server hits *WaitForMultipleObjects*, it stops dead. The server is no longer running. All the other programs in the system get all the cycles, and you get nothing. On most iterations through the loop, no notification will come through and the wait will time out. In this case, *Application.ProcessMessages* will be called to handle any waiting messages. When a file notification does come through, *WaitForMultipleObjects* returns its index into the handle array. The **else if** block that tests for valid indexes handles the notifications by using the stored client interface object to call the client's *Change* method. Notice that this call is protected in a **try** block so that one bad client can't bring the

```
procedure WaitForNotify;
var
  f : LongBool;
  iStat : Integer;
begin
  repeat
    //  Wait 100 milliseconds for notification.
    iStat := WaitForMultipleObjects(Count, @ahNotify,
                                    False, 200);
    if iStat = WAIT_TIMEOUT then
      // Timed out with nothing happening.
      Application.ProcessMessages
    else if (iStat >= 0) and (iStat <= Count) then
      begin
        // Ignore errors from client; that's their problem.
        try
          // Call client object with information.
          with aconNotify[iStat] do
            notifier.Change(sDir, iMode, fSubTree);
        finally
          // Wait for next notification.
          f := FindNextChangeNotification(ahNotify[iStat]);
          Assert(f, 'FindNextChangeNotification failed');
        end;
      end
    else if iStat = WAIT_FAILED then
      // Indicates no notification requests.
      Application.ProcessMessages;
  // Done when server is connected and object count is 0.
  until fConnected and (ComServer.ObjectCount = 0);
end;
```

Figure 4: The *WaitForNotify* procedure.

server down and stop other clients. After the callback, *FindNextChangeNotification* is called to wait for the next notification. Incidentally, *Count* is a function that counts the valid handles in the array by looping until it finds a handle with value -1.

The final issue is how you get out of the notification loop. You don't want to loop forever, thus keeping the server running, even if there are no notification connections. You want to keep track of how many client objects there are, and when the count reaches zero, terminate the loop (and thus the server). But the count is always zero when the server first starts, and you don't want to terminate then. So you watch a flag that is False when the server first starts, but becomes True as soon as the first connection is made.

But how do you keep track of the objects? The solution of testing the *ComServer.ObjectCount* property looks easy in the source, but you're not seeing how long it took me to figure that out. In the VB version, I kept a reference count of objects by incrementing a count when an object was created and decrementing it when the object was destroyed. VB classes have Class_Initialize and Class_Terminate events where you can count objects. Delphi classes have no such mechanism, and the obvious solution of giving your classes constructors and destructors doesn't work because Automation classes are derived from *TAutoObject*, which has a static constructor. Fortunately the *TComServer* class and its automatically-created global variable, *ComServer*, do the reference counting for you (and provide several other important services).

One last issue. Why would you care whether the server terminated when its object count is zero before the first connection? Normally, the server is started automatically when the first client connects to it. At that point, it has one connection, so it won't terminate on startup. The only time it would is if you started it from a command line or from Windows Explorer — something you wouldn't have any reason to do. It turns out there is one important situation when you need to start the server first: during debugging.

Because the Delphi documentation doesn't mention how to debug Automation objects, I'll explain it here. You have to run two copies of Delphi. Put the server in one and a client in the other. Run the server first. If you run the client first, a server will be loaded, but it won't be under the debugger and you won't be able to step through it. If you run the server first and then the client, the client code that activates the server will automatically switch to the server copy of Delphi when you hit a breakpoint in the server code.

Which Does Automation Best?

There's no clear winner when you compare Automation and other COM tasks in VB and Delphi. VB makes it easier to create and use simple Automation objects. It does this by hiding the irrelevant details. But once you move to more complex Automation problems, you may find that those details aren't as irrelevant as they first seemed; you may want to see and control them. If you're clever enough, you can usually hack your way around VB limitations, but in Delphi you can usually solve the problems without arcane hacks.

This dichotomy continues as you go further in advanced programming. For example, VB has automatic multi-threading models that can assign each new object created to a different thread. Delphi doesn't have anything comparable, but it gives you a fighting chance to design your own multi-threading schemes where you have complete control over what happens in each thread. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\APR\DI9804BK.

Bruce McKinney is the author of *Hardcore Visual Basic* [Microsoft Press, 1997]. He is currently learning new languages, experimenting with new technology, and trying to recover from a long career at Microsoft. You can reach him at brucem@pobox.com or visit his Web site at http://www.pobox.com/HardcoreVB.

By Dan Miser

ClientDataset

MIDAS on the Cheap

Delphi 3 has brought the concept of distributed computing to every programmer's desktop. With MIDAS, a developer can quickly and easily build, test, and deploy a multi-tier application. However, not every problem requires a multi-tier solution. Furthermore, MIDAS comes with a retail price tag of $5,000 per server for deployment. While this price is much cheaper than any other alternative on the market, it may still be cost-prohibitive for smaller programming shops. This article will explore how to take advantage of Borland's multi-tier technologies in 2-tier applications.

Delphi 3 VCL Enhancements

A new component, ClientDataset, was created for Delphi 3. This component descends directly from *TDataset*, and as such, is a database-independent component. Furthermore, all the data of a ClientDataset is stored in memory, and therefore is very fast.

Another modification to Delphi 3 is that all DBDataset components have a *Provider* property. This property is of type *IProvider*, which is a COM interface. The *IProvider* interface models a standard producer/consumer relationship. The producer is a DBDataset component, while the consumer will typically be the ClientDataset. In this way, data can flow to and from producer and consumer with minimal intervention on the programmer's part.

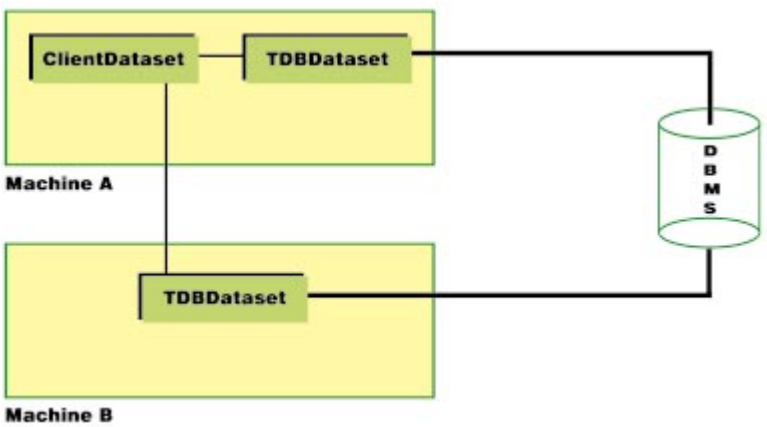


Figure 1: Diagram of 3-tier (client, server app, DBMS) vs. 2-tier (client, DBMS).

Figure 1 shows the communication between all the components related to database development in an *n*-tier application. Machine A contains all the components necessary to communicate directly with the DBMS, and as such, is categorized as a 2-tier application. If you introduce Machine B to communicate to the DBMS on Machine A's behalf, you have a 3-tier application. You are only using MIDAS if the ClientDataset gets its data from a DBDataset on a separate machine. Another way to check if you need a MIDAS license would be if you still need to deploy and install the BDE to make your client application run. If you need the BDE on your client machine, you probably don't need MIDAS.

Let's walk through a high-level overview of ClientDataset use in a 3-tier application:

- Create a server application.
- On a Remote DataModule, export the *IProvider* interfaces of the DBDataset components.

- Create a client application.
- Place a RemoteServer component on the client, attaching to the server application.
- Place a ClientDataset component on the client. Attach the *RemoteServer* property to the RemoteServer. Assign the *ProviderName* property to the provider exported from the server application.

Figure 2: Diagram of Provider interaction.

The act of setting the *ProviderName* property also sets the *Provider* property. Remember that the *Provider* property is of type *IProvider*, and the *IProvider* interface controls the flow of data (see Figure 2). Therefore, the ClientDataset does not point directly to the database table, but rather, points to the *Provider* property of the DBDataset assigned to the ClientDataset.

By contrast, using the ClientDataset component in a 2-tier application does not require you to create a server application from which you will receive data. However, since the ClientDataset must get its data from somewhere, we will link to a DBDataset on the same tier. By doing this, you will not be required to purchase a MIDAS license, but you will get the added benefits of using ClientDataset technology. You don't set the *RemoteServer* or *ProviderName* properties of the ClientDataset; those properties are only used when accessing the component in a 3-tier application.

We've seen how to assign the data to the ClientDataset in a 3-tier application. How do we accomplish this in a 2-tier application? There are four ways to accomplish this:

- Run-time assignment of *IProvider*
- Run-time assignment of data
- Design-time assignment of data
- Design-time assignment of *IProvider*

Assigning *IProvider*

At run time, you can assign the *Provider* property in code. This can be as simple as the following statement, found in *FormCreate*:

```
ClientDataset1.Provider := Table1.Provider;
```

A very important point to remember is that if you use this method of *IProvider* assignment, you must add the unit BdeProv to the **uses** clause. If you don't, you will receive the error message "No provider available" when running the application.

We can also assign the data directly from the DBDataset to the ClientDataset at run time with the following statement:

```
ClientDataset1.Data := Table1.Provider.Data;
```

Delphi can also bind a ClientDataset to a DBDataset at design time by selecting the **Assign Local Data** command from the context menu of the ClientDataset component. Then, you specify with which DBDataset component this ClientDataset should communicate, and the data is brought to the ClientDataset. A word of caution: If you were to save the file in this state and compare the size of the DFM file to the size before executing this command, you would notice an increase in the DFM size. This is because Delphi has stored all the physical table data associated with the DBDataset in the DFM. Delphi will only stream this data to the DFM if the ClientDataset is *Active*. You can also trim this space by executing the **Clear Data** command on the ClientDataset context menu.

Lastly, you can use the component provided here to tie the *Provider* properties together at design time. (In the Delphi 3.02 update, functionality was added to the ClientDataset that would allow the design-time assignment of *IProvider* if you leave the *RemoteServer* property blank. The component presented here can still be used for those with Delphi 3.0 or Delphi 3.01). This component publishes a *DataProvider* property where you can assign a component that exposes the *IProvider* interface, such as Table, Query, and Provider. When you set this property, a link between the *Provider* properties of the ClientDataset and the specified component will be created.

By using this component, you will have full access to the table to which the ClientDataset is indirectly connected. This means that you can add fields from the table and create calculated fields. Note that the *IProvider* interface will not send calculated fields across from the DBDataset to the ClientDataset. If you want calculated fields for a ClientDataset, you must create them on the ClientDataset.

The big difference between using DBDataset components and ClientDataset is that when you are using ClientDataset, you are using the *IProvider* interface to broker your requests for data to the underlying DBDataset component. This means that you will be manipulating the properties, methods, events, and fields of the ClientDataset component, not the DBDataset component. Think of the DBDataset component as a separate application that can't be manipulated directly by you with code.

Advantages of Using ClientDataset in 2-tier Applications

Using the ClientDataset component will dramatically reduce the network traffic in several instances:

- Reading an entire table
- Static lookup tables
- Sorting a table

In addition, you can control the number of records retrieved at one time via the *PacketRecords* property, just as in a multi-tier application.

Briefcase Model

ClientDataset has the ability to read and write its contents to local files. This is accomplished by using the *LoadFromFile* and *SaveToFile* methods. These methods are very powerful; in addition to storing the metadata associated with a table, they also store the data and the change log for that table. This means you can retrieve data from the database, edit the data, and save the data to a local CDS file. Later, when you are fully disconnected from the database, you can load the data from that CDS file and still have the ability to undo changes using the standard ClientDataset methods. Another great use for this is lookup tables.

Lookup

Typically, lookup tables are relatively small, and rarely change. If they rarely change, they don't need to take up bandwidth to send this static data across the network every time a client application starts. Instead, we can save the data locally in ClientDataset format. If you implement this method with dynamic tables, however, you need to implement some mechanism to let your application know when the lookup table has changed on the database server. This way, your application can download the latest version of the lookup table into the local cache.

Since the data is stored in a component derived from *TDataset*, we can use this component in a lookup capacity. For example, using a DBLookupComboBox component requires a *DataSource* and a *ListSource*. Until now, this *ListSource* needed to attach itself to the database server. This would tie up precious resources, and require more network traffic. With the ClientDataset method, we can store the data locally, and let the user look up the data from the data stored on the client. See the sample project Lookup.dpr in this month's .ZIP file (available for download; see end of article for details) for an example of how this can be put to use.

Indexing ClientDataset

If you want to sort the result set in ClientDataset, you can use the *IndexFieldNames* property, just as you would with the Table and Query components. In addition, the *AddIndex* and *DeleteIndex* methods are supplied to give you complete control over indexing of a ClientDataset. For example, using these methods, you can control whether an index is ascending or descending.

Since the ClientDataset uses the data stored on the local machine, there will be no need to ask the database server to re-run a query to sort on a different field. The benefits of this method are many: reduced network traffic, incredibly fast sort times, and the ability to sort on calculated fields.

To take advantage of calculated field sorting, you must specify the *FieldKind* of the calculated field as *fkInternalCalc*. However, you should only specify that a field is internally calculated if you plan to filter or sort on it, because marking this field as internally calculated will cause the ClientDataset to store the field in memory just like a regular field. If you don't need the added capabilities for this calculated field, continue to identify this field as a calculated field, and the values will be derived only when necessary.

You can define which type of calculated field this is at design time or at run time. At design time, you can add a new calculated field for the ClientDataset just as you have always done with DBDataset components. Invoke the Fields Editor by double-clicking on the ClientDataset; then right-click to display its context menu, and select **New field**. When the New Field dialog box appears, you can select either **Calculated** or **InternalCalc** to set the *TField.FieldKind*. You can also change the field type at design time by using the Object Inspector to change the value of the *FieldKind* property from *fkCalculated* to *fkInternalCalc*. Finally, to modify this attribute at run time, simply assign the property the value of *fkInternalCalc* in code after you have created the corresponding *TField*. Failure to set this property correctly will result in a "Field out of Range" error when you try to sort on the field. See the example CDSIndex project for a demonstration of this technique.

Cached Updates

All the preceding uses of ClientDataset are geared to mimic the use of local, or in-memory, tables. The final example presented here will show how to use the ClientDataset to greatly enhance cached update logic. According to Chuck Jazdzewski, the principal architect of Delphi, ClientDataset will be the official way to handle cached updates in the future.

Cached updates were introduced in Delphi 2 and gave developers another way to present and edit data in a multi-user application. Using cached updates reduced network traffic, but introduced the problem of data concurrency when trying to post the data stored on the client machine back to the database server. (For more information on using cached updates, see Cary Jensen's articles in the May, June, and July 1997 issues of *Delphi Informant*.)

The implementation of cached updates did have some problems, however. Some of the shortcomings of the CachedUpdate model are:

1) Using master/detail queries, you cannot cache detail records from different master records.
2) Inserting records in detail tables is not possible without changes to the VCL.
3) When using joined tables, you must use multiple *TUpdateSQL* and *OnUpdateRecord* events.

Delphi 3.01 corrected some of the problems associated with numbers 1 and 2 above; however, there is still one major limitation to using cached updates. Due to the way cached updates are implemented, you must apply the updates any time you move from a master record. This effectively means that your transactions and updates must occur on one batch of master/detail records. This may suit your needs, and, if it does, you can use the code written by Mark Edington of Borland (see Figure 3; attach the code to the *BeforeClose* event of the detail table).

```
procedure TForm1.DetailBeforeClose(DataSet: TDataSet);
begin
  if Master.UpdatesPending or Detail.UpdatesPending then
    if Master.UpdateStatus = usInserted then
      Database1.ApplyUpdates([Master, Detail])
    else
      Database1.ApplyUpdates([Detail, Master])
end;
```

Figure 3: Automatic ApplyUpdates when using CachedUpdates.

```
procedure TForm1.btnApplyClick(Sender: TObject);
var
  MasterVar, DetailVar: OleVariant;
begin
  cdsMaster.CheckBrowseMode;
  cdsDetail.CheckBrowseMode;

  { Setup the variant with the changes (or NULL if
    there are none). }
  if cdsMaster.ChangeCount > 0 then
    MasterVar := cdsMaster.Delta
  else
    MasterVar := NULL;
  if cdsDetail.ChangeCount > 0 then
    DetailVar := cdsDetail.Delta
  else
    DetailVar := NULL;

  { Wrap updates in a transaction; if any step creates an
    error, raise an exception and Rollback the transaction.
    This would normally be done on the middle-tier, i.e.
    MIDASConnection.AppServer.ApplyUpdates(
      DetailVar, MasterVar); }
  Database.StartTransaction;
  try
    ApplyDelta(cdsMaster, MasterVar);
    ApplyDelta(cdsDetail, DetailVar);
    Database.Commit;
  except
    Database.Rollback
  end;

  { If previous step resulted in errors, reconcile
    error datapackets. }
  if not VarIsNull(DetailVar) then
    cdsDetail.Reconcile(DetailVar)
  else if not VarIsNull(MasterVar) then
    cdsMaster.Reconcile(MasterVar)
  else
    begin
      cdsDetail.Reconcile(DetailVar);
      cdsMaster.Reconcile(MasterVar);
      cdsDetail.Refresh;
      cdsMaster.Refresh;
    end;
end;
```

Figure 4: ApplyUpdates when using ClientDataset.

By contrast, all the changes made to the data are stored locally on the client machine — even across different master records. Remember that a key benefit of ClientDataset is that it will allow us to delay the processing and reconciliation of the data until absolutely necessary. To reconcile the data back to the database, we need to write our own ApplyUpdates logic (see Figure 4). This isn't as simple as most tasks in Delphi, but it does give you full flexible control over the update process.

Applying updates in a multi-tier application is usually triggered by a call to *ClientDataset.ApplyUpdates*. This method sends the information needed to update the database to its Provider on the middle tier, where the Provider will then write the changes to the database. All of this is accomplished within a transaction, and is done without programmer intervention. To accomplish the same thing in a 2-tier application, you must understand what Delphi is doing for you when you make that call to *ClientDataset.ApplyUpdates*.

Any changes you make to ClientDataset data are stored in the *Delta* property. *Delta* contains all the information that will eventually be written to the database. This is what Delphi passes to the Provider in the multi-tier scenario above. Since our Provider exists on the same tier as the ClientDataset, we can call *ClientDataset.Provider.ApplyUpdates*. Remember to wrap these calls in a transaction so you can write all the changes as one unit. After applying the updates, a call to *Reconcile* will finish clearing the cache for this ClientDataset.

Note that there were some inconsistencies in using this method, depending on what database back-end you were using. For instance, native Paradox access yielded sporadic results in the testing of the MDCDS sample application. Switching to the ODBC drivers for Paradox faired a little better, but still produced some anomalies during the update process. However, the sample worked flawlessly with Microsoft SQL Server and Sybase SQL Anywhere back-ends.

We could extend this example further, and take advantage of the briefcase model previously mentioned. This would allow our users to be completely disconnected from the database server, make changes to the data, and apply the changes and reconcile any errors back to the database at a later date, when they can be physically connected to the network.

Lastly, if you find yourself missing the functionality of the UpdateSQL component, you can find a version that is compatible with ClientDataset in the \Demos\Midas\Usqlprov directory of the Delphi 3.01 update. The UpdateSQLProvider component expands on the functionality of the Provider component by providing an *OnUpdateRecord* event. With this event, the developer can have record-by-record control of the update process, which is useful for implementing business rules. This component is also necessary for performing updates using stored procedures. In addition, it also expands on the functionality of the UpdateSQL component by generating SQL code at run time. This is necessary if you are working with data that contains NULL values that need to be updated.

Deployment

When using the ClientDataset component, you have to deploy two additional files: DBClient.DLL and STDVCL32.DLL. DBClient implements the interfaces that drive ClientDataset, while STDVCL32 is a type library for Delphi's standard VCL. COM uses the Windows registry to read and write information about its components. Since these files are COM-based, they need to be registered.

During the installation of your application, these files should be copied to the \Windows\System directory and registered by setting the appropriate option to "Register an OCX." However, if your installation program doesn't allow automatic registration, you can use Regsvr32.exe, or Borland's Tregsvr.exe (in the \Bin directory), to register these files externally. One last point: The VCL automatically tries to register these libraries if they are present, but not registered.

Conclusion

This article has demonstrated the advantages of using ClientDataset architecture in a 2-tier application. The importance of becoming acquainted with these tools cannot be understated. Borland's commitment to this technology shows you can take advantage of these controls today, while giving your application a head start in transitioning to a 3-tier model in the future. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\APR\DI9804DM.

Dan Miser is a Design Architect for Stratagem, a consulting company in Milwaukee. He has been a Borland Certified Client/Server Developer since 1996, and is a frequent contributor to *Delphi Informant*. You can contact him at http://www.iinet.com/users/dmiser.

By Gary Warren King

Setting Limits: Part I

Enforcing Minimum and Maximum Form Size

You want to let the users of your application resize your forms. But they keep making them too small or too big, then blaming you because things "don't look right." If you switch the *FormStyle* to *fsDialog* or *fsSingle*, they complain the application "isn't flexible enough" because they can't resize the form!

What to do? Can we create resizable forms that won't let themselves get too small (or too big)? Yes, we can. As you'll see, we can even do it in a nicely-packaged component. This two-part article will take the basic Windows API solution (i.e. using WM_GETMINMAXINFO) and package it several ways. This first installment will:

- Show the general idea by creating a form that gives us the control we want;
- discuss how to extend this solution using form inheritance in Delphi 1, 2, and 3; and
- discuss what's wrong with inheritance, and look for something better.

The second installment will continue the discussion, and show how we can create a component that adds the desired resizing control to any Delphi form by using composition instead of inheritance.

Getting the Message

Windows accomplishes most things sending messages between the windows and processes running on your computer. When you use the keyboard or mouse, you generate messages that are received and processed by your application's event loop. Delphi hides most of the low-level details very nicely, so you don't usually need to worry about them. When you resize a form, it generates the messages shown in Figure 1 — more or less. Windows 3.1 and Windows 95 messages differ slightly, and there are some added complexities that we'll gloss over for now.

Message	Explanation
WM_NCLBUTTONDOWN	To resize, click the left button on the form's border.
WM_SYSCOMMAND	Windows converts this click into a system-command message (as if you had chosen the **Size** command from the **Control** menu).
WM_GETMINMAXINFO	Windows asks the form for minimum and maximum size information.
WM_MOUSEMOVE	You move the mouse.
WM_SIZING	Windows tells your form that its size is changing.
WM_LBUTTONUP	You release the mouse button.
WM_SIZE	Windows tells your form its new size.
WM_WINDOWPOSCHANGING	Windows tells your form that its position (size, in this case) is changing.
WM_WINDOWPOSCHANGED	Windows tells your form that its position (size, in this case) has changed.

Figure 1: Windows messages.

The meaning of most of these messages is pretty clear just from their names and the context. If we look up each message in the Windows Help file, we can delve into the details. For example, WM_GETMINMAXINFO's Help topic reads:

> *The WM_GETMINMAXINFO message is sent to a window when the size or position of the window is about to change. An application can use this message to override the window's default maximized size and position, or its default minimum or maximum tracking size.*

The maximum tracking size is the largest window size that can be produced by using the borders to size the window. The minimum tracking size is the smallest window size that can be produced by using the borders to size the window.

This is exactly the message we need to control the minimum and maximum size of the form. If our form could intercept this message and alter it, then we would gain complete control over just how big and how small it could appear.

Catching the Message

Delphi conveniently hides almost all the Windows messaging arcana behind its pretty face: You know it's happening somewhere; but, because the standard VCL controls do what you want most of the time, you seldom need to care exactly where. Fortunately, Delphi makes it ridiculously easy to catch particular messages and handle them in your own way. It's this combination of ease and power that makes Delphi such a wonderful development tool.

To catch and handle the WM_GETMINMAXINFO message in our form, we need to create a message-handling method. Start with a new project, and add the contents of Figure 2 to *TForm1*'s class interface. The *wmGetMinMaxInfo* procedure is our new message handler. The **message** keyword tells the Delphi compiler that this method should be called if the form receives the WM_GETMINMAXINFO message from Windows. In this case, the compiler handles all the hard work of correctly interpreting and interfacing with the Windows messaging system; all we need to do is write the method that responds to WM_GETMINMAXINFO. Here's one possibility:

```
procedure TForm1.wmGetMinMaxInfo(var msg: TMessage);
begin
  with TWMGetMinMaxInfo(msg).MinMaxInfo^ do begin
    { Don't let this form get smaller than (200, 100) or
      bigger than (400, 300). }
    ptMinTrackSize := point(200, 100);
    ptMaxTrackSize := point(400, 300);
  end;
end;
```

Let's see what's going on here. Each Windows message carries extra data in its *wParam* and *lParam* fields. The meaning of this data varies depending on the needs of the particular message. The WM_GETMINMAXINFO message uses its *lParam* to contain a pointer to a *TMinMaxInfo* structure, which is defined (in Wintypes.pas or Windows.pas) as:

```
PMinMaxInfo = ^TMinMaxInfo;
TMinMaxInfo = packed record
  ptReserved: TPoint;
  ptMaxSize: TPoint;
  ptMaxPosition: TPoint;
  ptMinTrackSize: TPoint;
  ptMaxTrackSize: TPoint;
end;
```

Notice that Delphi also defines a pointer type (*PMinMaxInfo*) to point to this structure. *TMinMaxInfo* holds five points determined by (*X*, *Y*) pairs: four to hold the various minimum and maximum sizes, and one reserved for future Microsoft schemes.

```
type
  TForm1 = class(TForm)
  protected
    { Protected declarations }
    procedure wmGetMinMaxInfo(var msg: TMessage);
      message WM_GETMINMAXINFO;
  end;
```

Figure 2: Handling the WM_GETMINMAXINFO message.

To make life easier for the Delphi programmer, Borland provides specialized message types for most of the Windows messages, including WM_GETMINMAXINFO. Here is *TWMGetMinMaxInfo* (from Messages.pas):

```
TWMGetMinMaxInfo = record
  Msg: Cardinal;
  Unused: Integer;
  MinMaxInfo: PMinMaxInfo;
  Result: Longint;
end;
```

This means that the somewhat cryptic **with** statement:

```
with TWMGetMinMaxInfo(msg).MinMaxInfo^ do begin
```

accomplishes four things:

1) It takes the incoming message;
2) casts it as a *TWMGetMinMaxInfo*;
3) pulls out the *MinMaxInfo* field (which is a pointer to a *TMinMaxInfo*); and
4) de-references the pointer so we can use and modify its contents.

If you compile and run this simple application, you'll see that you're still able to resize the form, but only within the limits we specified in the *wmGetMinMaxInfo* method. This message-handler method works fine in all versions of Delphi. The ability to conveniently catch any Windows message is powerful, but has its limitations as well. For example, if we want to constrain the sizes of many of the forms in our application, this method would have us typing the same information over and over again. There must be a better way!

Inherit the Form

One better solution (though, as we'll see in the second installment, not the best solution) is to put our special message handling into its own form, then inherit this functionality wherever we need it. Though handled differently, form inheritance is a viable option in all versions of Delphi. Before we see how it works, however, lets improve the interface of our existing min/max form. We want it to be easy to modify the size constraints of the form (ideally both at design and run time). A better way to expose the minimum and maximum tracking sizes is to make them properties of the form. Figure 3 shows an abridged version of *TMinMaxForm*.

We've given the form new properties (*MinTracking* and *MaxTracking*) for the height and width of the sizes we want to constrain. These properties are stored in *TPoint* structures, and accessed by the various *Get* and *Set* methods of the class. The *Set* methods have the side effect of calling *UpdateSize*:

```
TMinMaxForm = class(TForm)
    procedure FormShow(Sender: TObject);
  private
    FMinTrackSize: TPoint;
    FMaxTrackSize: TPoint;
    function  GetMaxTrackSizeX: integer;
    function  GetMaxTrackSizeY: integer;
    ...
    procedure SetMinTrackSize(const p: TPoint);
    procedure SetMaxTrackSize(const p: TPoint);
    procedure SetMaxTrackSizeX(const x: Integer);
    procedure SetMaxTrackSizeY(const y: Integer);
    ...
    procedure UpdateSize;
    procedure wmGetMinMaxInfo(var msg: TMessage);
      message WM_GETMINMAXINFO;
  published
    property ResizeMaxHeight: Integer
      read GetMaxTrackSizeY write SetMaxTrackSizeY
      default 0;
    property ResizeMaxWidth: Integer
      read GetMaxTrackSizeX write SetMaxTrackSizeX
      default 0;
    ...
end;
```

Figure 3: Converting tracking sizes to form properties.

```
procedure TMinMaxForm.SetMinTrackSize(const p: TPoint);

begin
  if (p.x = FMinTrackSize.x) and
     (p.y = FMinTrackSize.y)) then
    Exit;
  FMinTrackSize := p;
  UpdateSize;
end;
```

UpdateSize makes sure any changes to the size constraints are immediately reflected in the size of the form. If, for example, you tell the form that it can't be any wider than 400 pixels — and it's currently sized at 500 pixels — you want it to immediately resize itself, taking the new constraint into account. *UpdateSize* does this by calling the *MoveWindow* Windows API command, using the current size and position of the form as its parameters. Calling *MoveWindow* this way doesn't actually move the form, but

```
procedure TMinMaxForm.wmGetMinMaxInfo(var msg: TMessage);
var
  P : TPoint;
begin
  with TWMGetMinMaxInfo(msg).MinMaxInfo^ do begin
    if not ((FMinTrackSize.X = 0) and
            (FMinTrackSize.Y = 0)) then
      begin
        P := FMinTrackSize;
        if P.X = 0 then P.X := Screen.Width;
        if P.Y = 0 then P.Y := Screen.Height;
        ptMinTrackSize := P;
      end;
    if not ((FMaxTrackSize.X = 0) and
            (FMaxTrackSize.Y = 0)) then
      begin
        P := FMaxTrackSize;
        if P.X = 0 then P.X := Screen.Width;
        if P.Y = 0 then P.Y := Screen.Height;
        ptMaxTrackSize := P;
      end;
  end;
end;
```

Figure 4: Implementing the *wmGetMinMaxInfo* method.

generates WM_SIZE messages that eventually result in WM_GETMINMAXINFO messages, which cause the new constraints to be used:

```
procedure TMinMaxForm.UpdateSize;
begin
  { Make sure any min/max settings are activated by calling
    the MoveWindow API command. Although we call it with
    parameters that match the current size of the window,
    it will send a WM_GETMINMAXINFO message. }
  MoveWindow(Handle, Left, Top, Width, Height, True);
end;
```

Finally, we implement the *wmGetMinMaxInfo* method (see Figure 4). We first check to see if we've defined a size constraint, by determining if either value is non-zero. We then need to add special checks for constraints that set only one of the properties — so that we don't try to tell Windows that, for example, the minimum height is zero. These improvements will make this form much easier to use when we deal with inheritance.

Form Inheritance in Delphi 1

Now that we have a form from which to inherit, it's an easy matter to do the inheriting. Delphi 1 got a bad rap because it doesn't support visual form inheritance; but it's important to remember that it *does* support regular form inheritance. You just can't see what you're doing! To try this out, create a blank project in Delphi 1, add the unit name of the form we created to the **uses** clause, and modify the type definition of the form (see Figure 5). You can then add size constraints to *TForm1*:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ResizeMaxHeight := 200;
  ResizeMinWidth := 400;
  ResizeMinHeight := 100;
end;
```

If you compile and run this new project, you'll see that although it wasn't a visual process, *TForm1* does inherit from *TMinMaxForm*. Delphi 2 and 3 make inheritance even easier with the Object Repository.

Form Inheritance in Delphi 2 and 3

Delphi 2 and 3 greatly enhance form inheritance by making it visual, and supporting it directly within the Delphi IDE. You can easily extend the Delphi environment by adding your own forms (and projects, classes, etc.) to the Object Repository. Once added, these forms become the starting point for any new forms you create.

To place a form in the repository, just open it from within the IDE, and right-click to produce its SpeedMenu. Select the

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
    Graphics, Controls, Forms, Dialogs, fmminmax;

type
  TForm1 = class(TMinMaxForm)
```

Figure 5: A test for *TForm1*.

Add to Repository command, and you'll see the dialog box shown in Figure 6. (I've already filled it in.) Once you've added something to the repository, you get it out again by choosing the **File | New** command from within the Delphi IDE (in version 2 and newer). The Delphi 2 New Items dialog box is shown in Figure 7. As you can see, Delphi gives you three options when taking things out of the Repository. You can:

- create a new item as a copy of the Repository item,
- create a new item that inherits from the item in the Repository, or
- simply use the Repository item directly.

Copying is useful when you want to base your new work on an existing solution, but don't want changes in the existing solution to alter the new item. Using the item directly generally isn't a good idea unless you need to modify an item in the Repository. When you use an item, you're creating a direct link to the Repository. Any changes made there will affect your new work. Inheritance provides the full power of object-oriented class inheritance: Your new item will have all the properties and methods of the item it inherits from. This can be a great way of placing common functionality or look-and-feel in one form, then building the rest of your application on that form. If you change the base form, all your child forms will immediately inherit the new look and behavior.

To create a new form based on the min/max form we just added to the Repository, select it (on the Forms tab), choose the **Inherit** radio button, and click **OK**. We can then add our size-setting code to the form's *OnCreate* event, and be ready to roll. Here's the code that Delphi generates when we click the *OnCreate* event handler in the object inspector, and add our settings to it:

```
procedure TMinMaxForm1.FormCreate(Sender: TObject);
begin
  inherited;
  ResizeMaxHeight := 200;
  ResizeMinWidth := 400;
  ResizeMinHeight := 100;
end;
```

Because Delphi knows this form inherits from a class other than *TForm*, it automatically adds the **inherited** keyword. If you compile this project and run it, you'll see it has exactly the functionality we want.

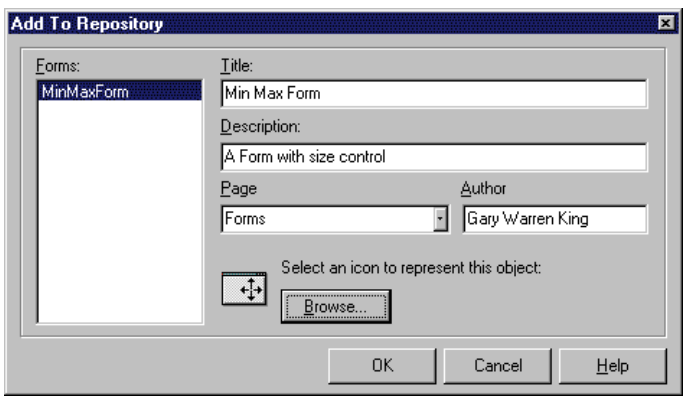


Figure 6: Delphi 2's Add To Repository dialog box.

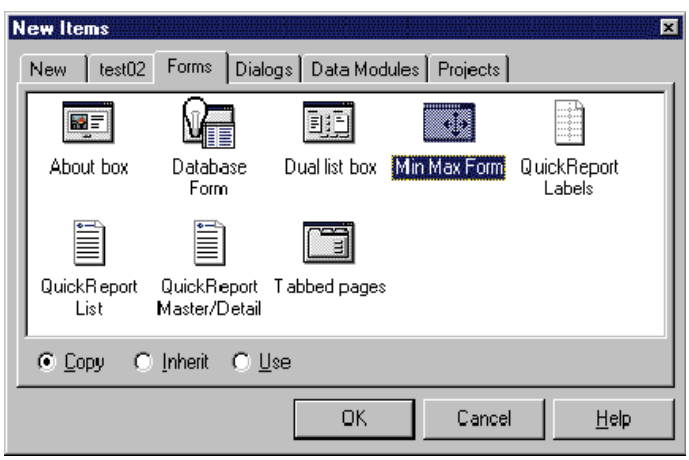


Figure 7: The Delphi 2 New Items dialog box.

What's Wrong Here?

On the surface, inheritance (visual or otherwise) provides a wonderful solution for us. When we want to create a form whose size is constrained, we can simply inherit from *frmMinMax*. This is certainly much nicer than our first solution (duplicating many lines of codes in each form), but still suffers from some knotty problems.

Design-time control versus run-time flexibility. Inheritance requires that we make our decisions at design time through the construction of our class hierarchy. Inheritance is something that works with classes, and classes exist only in our source code and compiler. When mixing and matching objects at run time, the constraints of doing everything with class-based inheritance can be overwhelming.

The multiplying-class problem. When we have only one special property or ability to add to a form, inheritance works wonderfully. But suppose we must add more special forms to our Repository (for example, a form that paints its title bar differently, or a form with rounded borders or a gradient background). Now suppose we want to use inheritance to mix and match between all these special abilities. With four to choose from, we'd need to create 15 classes to handle our needs: *TfrmMinMax*, *TfrmMinMaxSpecialTitle*, *TfrmMinMaxRoundBorder*, *TfrmMinMaxSpecialTitleRoundBorderGradient*, etc.

Clearly, this quickly becomes unmanageable! Multiple inheritance would provide one solution to this problem; but a better solution is to realize that inheritance is not the correct way to view this situation. Inheritance is applicable when the child class is a specialization of the parent class — when the differences between the child and parent are not simply differences in the data they contain, and when the child class extends the responsibilities of the parent instead of negating or completely changing them. These new abilities (*MinMax*, *SpecialTitle*, etc.) are just decorations to the existing form class. (You can find an excellent discussion of this sort of inheritance problem — and a prototypical solution — in the Decorator pattern in *Design Patterns: Elements of Reusable Software Solutions*

[Addison Wesley, 1994], by Erich Gamma, et al.) They are better modeled as collaborations that can be mixed and matched at design and run time, rather than as static class-inheritance structures. Ideally, we would like to build new classes that worked with a form to add the features we desire.

Conclusion

This article has shown how to add minimum and maximum size control to a Delphi form, first by adding code directly to the form class, then by using inheritance. Along the way, we've looked at Windows messaging, form inheritance (in Delphi 1), visual form inheritance (in Delphi 2 and 3), and the Object Repository. Finally, we observed that inheritance is not the best solution in every situation, and outlined the elements of a better solution. Part II will explore mix-and-match feature composition in detail, and build a component that can add size control instantly to any form, regardless of its class. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\APR\DI9804GK.

Gary King is the principal of DesignSystems (http://www.oz.net/dsig), makers of Interface Gizmos and other fine products for Delphi. He can be reached at gking@oz.net.

By Cary Jensen, Ph.D.

Interfaces

An Introduction

If you're a regular reader of *Delphi Informant*, you've had the opportunity to read a number of articles featuring the use of COM (Component Object Model) technologies in Delphi 3 applications. These technologies include OLE automation and ActiveX, among others. One reason for this extensive coverage is that Delphi 3 is an outstanding tool for creating COM applications.

But why is Delphi so good at this? The answer can be found in a new feature added to the Object Pascal language with the introduction of Delphi 3: the *interface*. A number of these COM-related articles have discussed interfaces, but most of them have done so only in passing. This month's "DBNavigator" will take a different tack. Instead of focusing on COM, let's consider the interfaces.

A couple of general comments are in order, however, before we get to the primary focus of this article. First, interfaces are abstract by nature. In this regard, I am reminded of some beginning programming classes that I taught to a group of non-programmers many years ago. One of the concepts I had to introduce was that of the variable. The thing is, once you grasp the concept, it seems perfectly natural and, in fact, obvious. But it's easy to forget that for people who haven't been introduced to variables via algebra, statistics, or some other similar field of study, the concept of a variable is a difficult one to grasp. This is also true with interfaces. If you have not already dealt with abstract classes or interfaces in either Java or Delphi (or other object-oriented languages), the concept may seem difficult. Fortunately, once you begin working with interfaces, they become quite natural.

The second point I want to make is that while interfaces have several useful applications, they are currently used only for COM by an overwhelming percentage of Delphi developers. Consequently, it may appear somewhat odd to consider interfaces without much discussion of COM. But interfaces do have other uses, and one of these, the polymorphic treatment of dissimilar objects, is discussed in this article. However, for a more in-depth look at the use of interfaces in COM-related applications, refer to those other articles that have appeared, and will continue to appear, in the pages of this magazine.

What Is an Interface?

An interface is a declaration of methods and properties, much like the methods and properties declared in a class. Unlike a class definition, however, an interface doesn't actually implement those methods, nor does it store the property values. In this respect, an interface is like a pure virtual, abstract class, where method declarations serve as placeholders to be overridden and implemented by descendent classes, giving those classes a common set of methods. In other words, interfaces provide a second mechanism, after inheritance, for polymorphism.

While an interface never implements the methods it declares, the methods are designed to be implemented by a class. When a class is declared, any interfaces it will implement are included in the class definition, in a comma-separated list following the name of the class from which the new class is descending. A class that includes an interface in its definition is said to "implement that interface."

If a class implements an interface, it's required to implement all the methods of that interface. These methods may be implemented either by declaring and implementing the interface methods, or by inheriting methods that provide the implementation of the methods defined in the interface.

When two different classes implement the same interface, by definition they share those common methods declared in that interface. Furthermore, instances of those two classes can be assigned to a variable declared to be of the **interface** type. In the absence of an interface, two objects that represent instances of two different classes are not assignment compatible unless they descend from a common ancestor. Furthermore, they are only assignment compatible with respect to that common ancestor class.

However, with interfaces, it's possible to declare a variable of the **interface** type, and assign to it an instance of any class that implements that interface. This is true even when the two classes are unrelated in an object-oriented sense (other than by their common *TObject* ancestor class). Once an instance of an object that implements an interface is assigned to a variable of that interface type, that variable can be used to call the methods of the interface. In other words, the methods of the interface can be called without regard to how those methods are implemented. This is the essence of polymorphism. Another way of looking at it is that interfaces provide for the polymorphic benefits of multiple inheritance, even though classes in Delphi can descend from only a single ancestor. (With languages that support multiple inheritance, such as C++, a new class can be declared that simultaneously descends from two or more existing classes.)

Declaring an Interface

An interface declaration looks similar to a class declaration. It must appear in a **type** block, and include the reserved word **interface**, as opposed to **class**. Like class declarations, interfaces cannot be declared within functions or procedures. Furthermore, unlike a class declaration, an interface cannot include instance variables. As a result, declarations of properties in interfaces must use accessor methods to get and set properties, as opposed to direct access (which requires instance variables).

The following is the declaration of an interface that declares three methods and a property. Two of the methods, *getMessText* and *setMessText,* are accessor methods (sometimes called "getter" and "setter" methods):

```
type
  IShowMess = interface(IUnknown)
    ['{ F8AA90A1-EA2D-11D0-82A8-444553540000 }']
    function ShowMessage: Boolean;
    function getMessText: string;
    procedure setMessText(Value: string);
    property MessText: string read getMessText
                              write setMessText;
  end;
```

This interface is named *IShowMess*, and it descends from the interface *IUnknown*. Interface names begin with an "I" by convention. *IUnknown* is similar to *TObject*, in that it is the highest-level interface, itself having no ancestor.

When you declare a new interface, you can specify from which interface it descends. However, forward declarations of interfaces are allowed. This is done simply by declaring the interface using the keyword **interface** without a descendant interface. Like forward class declarations, this is only necessary when you are declaring two (or more) interfaces whose declarations are mutually dependent.

The second line of this interface declaration contains a 128-bit binary number known as a GUID (Globally Unique IDentifier). Interfaces used in COM and related technologies require a GUID, but an interface that is not used in COM does not. *IUnknown* is the interface required by all COM objects.

You never generate a GUID on your own, since it is essential that the GUID for a particular interface be unique across all machines that exist or will exist. Instead, you should use the appropriate Windows API call to generate it. In Delphi, making this call manually is never necessary, since you can simply press Ctrl Shift G from within the editor to generate a valid GUID and insert it at the position of the cursor.

As mentioned earlier, all interfaces, by definition, descend from *IUnknown*. Since *IUnknown* specifies three functions, *QueryInterface*, *_AddRef*, and *_Release*, these methods must be implemented for any interface. It should also be noted that the *IUnknown* interface provides for one very specific function: the referencing of objects. Specifically, the implementation of *QueryInterface* is designed to instantiate an instance of a requested object and return a pointer to it. Furthermore, *_AddRef* and *_Release* are designed to perform reference counting. Taken together, these methods must implement the creation and destruction of requested objects.

IUnknown is special for another reason. Its definition is specified by COM. Furthermore, any component that implements the *IUnknown* interface, regardless of which language it is written in, is a COM server. It is due to this fact that we can say that Delphi has language-level support for COM.

The following is the Object Pascal declaration of *IUnknown*:

```
IUnknown = interface
    ['{ 00000000-0000-0000-C000-000000000046 }']
    function QueryInterface(const IID: TGUID;
      out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

As far as OLE automation is concerned, there is a second interface that is important. This interface is *IDispatch*. Any component that implements *IDispatch* is an OLE automation server, by definition. The following is the Object Pascal declaration of *IDispatch*:

```
IDispatch = interface(IUnknown)
    ['{ 00020400-0000-0000-C000-000000000046 }']
    function GetTypeInfoCount(out Count: Integer):
      Integer; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer;
      out TypeInfo): Integer; stdcall;
    function GetIDsOfNames(const IID: TGUID;
      Names: Pointer; NameCount, LocaleID: Integer;
      DispIDs: Pointer): Integer; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID;
      LocaleID: Integer; Flags: Word; var Params;
      VarResult, ExcepInfo, ArgErr: Pointer):
      Integer; stdcall;
  end;
```

The *IUnknown* and *IDispatch* interfaces are declared in Delphi's System unit.

Implementing Interfaces

The methods of an interface are implemented by an object that implements the interface. In other words, when an interface appears in the declaration of a class, the class must declare and implement all the methods defined in the interface (or interfaces when the class implements more than one interface). Furthermore, if an object implements an interface other than *IUnknown*, it must implement all methods declared in the interfaces from which the implemented interface descends. For example, if a class implements the *IDispatch* interface, the class must not only implement the four methods declared in *IDispatch*, it must also implement the three methods declared in *IUnknown*, since *IDispatch* descends from *IUnknown*. In fact, since every interface descends from *IUnknown*, any class that implements an interface must implement the methods of *IUnknown*, by definition.

The implementation of the methods declared in an interface may be inherited methods. In other words, if a new class is declared to descend from a class that implements the three methods declared in *IUnknown* (*QueryInterface*, *_AddRef*, and *_Release*), the implementation of *IUnknown* is satisfied. In fact, Delphi provides a large number of classes that implement the basic interface methods of *IUnknown*. For example, the *TInterfacedObject* class implements the *IUnknown* interface. As a result, any object that descends from *TInterfacedObject* is a COM server. Additional classes include *TAutoObject* and *TActiveXObject*. *TAutoObject* implements *IDispatch*, so any object that descends from *TAutoObject* is an OLE automation server. Likewise, *TActiveXControl* implements more than 10 additional interfaces required by the ActiveX specification.

It's also interesting to note that the *TComponent* class implements all seven methods defined by *IUnknown* and *IDispatch*. Consequently, you can declare any descendant of *TComponent* to implement *IDispatch* without having to manually implement these methods.

Consider the following **type** declarations, which include the declaration of one interface, *IShowMess*, and two new classes, both descending from *TInterfacedObject*:

```
type
  IShowMess = interface(IUnknown)
    ['{ F8AA90A1-EA2D-11D0-82A8-444553540000 }']
    function ShowMessage: Boolean;
    function getMessText: string;
    procedure setMessText(Value: string);
    property MessText: string read getMessText
                              write setMessText;
  end;
  TYesDefault = class(TInterfacedObject, IShowMess)
    FMessText: string;
    function ShowMessage: Boolean;
    function getMessText: string;
    procedure setMessText(value: string);
  end;
  TNoDefault = class(TInterfacedObject, IShowMess)
    FMessText: string;
    function ShowMessage: Boolean;
    function getMessText: string;
    procedure setMessText(value: string);
  end;
```

Since both of these objects implement *IShowMess*, it's necessary for them to also declare and implement the methods declared in the interface *IShowMess*. In this case, this includes the *ShowMessage* function, as well as the *getMessText* and *setMessText* accessor methods (methods used to read and write the property declared in the interface). Notice, while the interface declares the property, it doesn't need to be redeclared in the classes that implement the interface. When this approach is taken, the property belongs to the interface, but not to the class that implements the interface.

As mentioned earlier, when a class implements an interface, it must provide for the implementation of the methods declared in the interface. Since the *TYesDefault* and *TNoDefault* classes descend from *TInterfacedObject*, the implementation of the *IUnknown* methods are satisfied. However, both of these classes must implement the methods of *IShowMess*. This can be seen in Listing One (see page26), from the Intrface project. Notice that while the implementation of the accessor methods are identical for both classes, the *ShowMessage* method differs. Specifically, when an instance of the *TYesDefault* class displays a confirmation dialog box, the **OK** button is the default, while an instance of the *TNoDefault* class specifies that the **Cancel** button is the default.

On the main form of the Intrface project (see Figure 1), there are two buttons representing the two means by

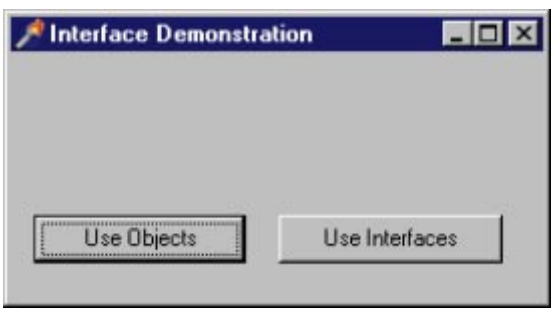


Figure 1: The main form of the Intrface project.

which the methods of *TYesDefault* and *TNoDefault* instances can be called. The first button, labeled **Use Objects**, demonstrates the way that these methods are typically called, using object references. The following is the *OnClick* event handler for this button:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Mes1: TYesDefault;
  Mes2: TNoDefault;
begin
  Mes1 := TYesDefault.Create;
  Mes2 := TNoDefault.Create;
  Mes1.setMessText('This is a TYesDefault');
  Mes1.ShowMessage;
  Mes2.setMessText('This is a TNoDefault');
  Mes2.ShowMessage;
  Mes1.Free;
  Mes2.Free;
end;
```

When this button is clicked, the objects are created, the *setMessText* method is called to assign a value to the *FMessText* field, the *ShowMessage* method is called to display the message, and then the objects are freed.

The second button, labeled **Use Interfaces**, demonstrates how an interface variable can be used to call the *ShowMessage* methods of the *TYesDefault* and *TNoDefault* class instances. The following is the *OnClick* event handler for this button:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  IntVar: IShowMess;
begin
  IntVar := TYesDefault.Create;
  IntVar.MessText := 'This is a TYesDefault';
  IntVar.ShowMessage;
  IntVar := TNoDefault.Create;
  IntVar.MessText := 'This is a TNoDefault';
  IntVar.ShowMessage;
end;
```

In this case, the values returned by the call to the *TYesDefault* and *TNoDefault* constructors are assigned to the interface variable, which is a legal assignment due to both of these classes implementing the *IShowMess* interface. With the interface variable pointing to an interfaced object that implements the interface, the methods and properties of the interface can be accessed.

The key here is that the properties and methods of the interface can be called for any object that implements the interface, even when those objects that implement the interface do not inherit the properties and methods from a common ancestor. In this case, the *MessText* property and *ShowMessage* method are not inherited by *TYesDefault* and *TNoDefault* from *TInterfacedObject*. Instead, they are instantiated within each of these classes. Without the interface, they would be incompatible. It is the interface itself, which, by being implemented by these two, separates classes and guarantees cross-object compatibility. To put this another way, the interface provides for polymorphism in the absence of methods inherited from a common ancestor.

While the *TYesDefault* and *TNoDefault* classes both descend from a common ancestor, this was not necessary. This technique would have worked just as well if *TYesDefault* descended from *TInterfacedObject*, and *TNoDefault* descended from *TLabel* (recall that *TLabel* inherits the *IUnknown* methods from *TComponent*).

There is one additional characteristic of the interface variable example that I'm sure you've noticed. Specifically, unlike in the object reference example, there is no explicit call to free the objects referenced by the interface variable. In fact, since *Free* is not declared for the interface, attempting to call *Free* using the interface variable would result in a compiler error. But this does not mean that the objects referenced by the interface variable are not freed; indeed, they are. As you recall, *IUnknown* implements methods responsible for object creation, reference counting, and object destruction. The destruction of an object being referenced using an interface reference occurs automatically after the last reference to the object is released.

There are three conditions that cause the *_Release* method of an interfaced object to be called:
1) The interface variable goes out of scope.
2) The interface variable is assigned a different interfaced object.
3) The interface variable is set to **nil.**

When the last reference to the interface object is released, and the object's reference count decrements to zero, the object is automatically freed.

Interfaces and Method Resolution

As you've already learned, any object that implements an interface is required to implement the methods defined within that interface. This can pose a problem if the interface declares a method whose name is already in use by an object that needs to implement the interface. For example, imagine there is a class named *TMessageObject* that must implement the *IShowMess* interface, but has inherited the method *ShowMessage* from its ancestor. This is a problem if the inherited method is conceptually different from the interface method of the same name. In this case, it's necessary to implement a method corresponding to the interface method, and distinguish it from the inherited method.

Delphi provides for the mapping of interface methods onto implemented methods of a different name. For example, consider the following declaration of *TMessageObject*:

```
type
  TMessageObject = class(TParentMessageObject, IShowMess)
    FMessText: string;
    function IShowMess.ShowMessage := DisplayMessage;
    function GetMessText: string;
    procedure setMessText(value: string);
    function DisplayMessage: Boolean;
  end;
```

This declaration specifies that *TMessageObject* descends from *TParentMessageObject*, and implements *IShowMess*. Since *TMessageObject* already has a *ShowMessage* method, it

is necessary to map the *IShowMess.ShowMessage* method to a different method name, which in this case is *DisplayMessage*. When using method resolution to map an interface method onto a new method name, the interface method, and the one it's being mapped to, must have the same argument list and return type.

Given the preceding **type** declaration, if the *ShowMessage* method for an instance of *TMessageObject* is called using an object reference, the inherited *ShowMessage* method is executed. However, if an instance of *TMessageObject* is assigned to an *IShowMess* variable, and the *ShowMessage* method is called, the *DisplayMessage* method will execute.

Conclusion

Interfaces provide Delphi with language-level support for COM. However, interfaces aren't just for COM. Specifically, interfaces provide you with an additional source of polymorphism, providing for assignment compatibility across dissimilar objects. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\APR\DI9804CJ.

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 a Contributing Editor of *Delphi Informant*, and was a member of the Delphi Advisory Board for the 1997 Borland Developers Conference. For information concerning Jensen Data Systems' Delphi consulting and training services, visit the Jensen Data Systems Web site at http://idt.net/~jdsi. You can also reach Jensen Data Systems at (281) 359-3311, or via e-mail at cjensen@compuserve.com.

Begin Listing One

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  IShowMess = interface(IUnknown)
    ['{ F8AA90A1-EA2D-11D0-82A8-444553540000 }']
    function ShowMessage: Boolean;
    function getMessText: string;
    procedure setMessText(Value: string);
    property MessText: string read getMessText
                             write setMessText;
  end;
  TYesDefault = class(TInterfacedObject, IShowMess)
    FMessText: string;
    function ShowMessage: Boolean;
    function getMessText: string;
    procedure setMessText(value: string);
  end;
  TNoDefault = class(TInterfacedObject, IShowMess)
    FMessText: string;
    function ShowMessage: Boolean;
    function getMessText: string;
    procedure setMessText(value: string);
  end;
  TForm1 = class(TForm)
    Button2: TButton;
    Button1: TButton;
    procedure Button2Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

function TYesDefault.getMessText: string;
begin
  Result := FMessText;
end;

procedure TYesDefault.setMessText(value: string);
begin
  FMessText := Value;
end;

function TYesDefault.ShowMessage;
begin
if MessageBox(HInstance, 'Continue?', PChar(FMessText),
             MB_OKCANCEL+MB_DEFBUTTON1) <> IDOK then
  Result := False
else
  Result := True;
end;

function TNoDefault.getMessText: string;
begin
  Result := FMessText;
end;

procedure TNoDefault.setMessText(value: string);
begin
  FMessText := Value;
end;

function TNoDefault.ShowMessage;
begin
if MessageBox(HInstance, 'Continue?', PChar(FMessText),
             MB_OKCANCEL+MB_DEFBUTTON2) <> IDOK then
  Result := False
else
  Result := True;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Mes1: TYesDefault;
  Mes2: TNoDefault;
begin
  Mes1 := TYesDefault.Create;
  Mes2 := TNoDefault.Create;
  Mes1.setMessText('This is a TYesDefault');
  Mes1.ShowMessage;
  Mes2.setMessText('This is a TNoDefault');
  Mes2.ShowMessage;
  Mes1.Free;
  Mes2.Free;
end;

procedure TForm1.Button2Click(Sender: TObject);
```

```
var
  IntVar: IShowMess;
begin
  IntVar := TYesDefault.Create;
  IntVar.MessText := 'This is a TYesDefault';
  IntVar.ShowMessage;
  IntVar := TNoDefault.Create;
  IntVar.MessText := 'This is a TNoDefault';
  IntVar.ShowMessage;
end;

end.
```

End Listing One

By Rod Stephens

Rough around the Edges

Antialiasing in Delphi

If you look closely at a computer monitor, you can see the pixels that make up the lines and characters. Lines drawn at certain angles can appear particularly rough and jagged. This effect is called *aliasing*. Figure 1 shows a close-up of some aliased text. Notice how rough the left edges of the letters "A" and "l" are.

Aliasing occurs when data is sampled at a frequency that is insufficient to capture all of the data's important information. In this case, the monitor displays images at a resolution of around 96 pixels per inch. Your eye can resolve a few thousand points per inch; because your eye has such high resolution, you can see the places where the monitor draws lines and curves with jagged approximations.

Normally, when a computer draws a black line, it makes some pixels black and it leaves others unchanged, as shown in Figure 2. This produces a jagged, aliased line. You can smooth the rough edges using a technique called *antialiasing* (or *dejagging*). In antialiasing, pixels that intersect the line are drawn using a color that corresponds to the amount by which they are covered by the line. If a pixel is 50-percent covered, it's drawn 50-percent black; if the pixel is 25-percent covered, it's drawn 25-percent black. Figure 3 shows a close-up of a line drawn with antialiasing. Seen at a distance, this line appears smoother than the line in Figure 2. It also appears wider and slightly fuzzier.

Supersampling

The calculations that determine the amount of a pixel covered by a line, or other shape, are quite complicated. Fortunately, a much simpler technique called *supersampling* produces a similar result. Instead of drawing the image at the monitor's normal 96-or-so pixels per inch, you draw it at a higher resolution.

Figure 1: An example of aliasing.

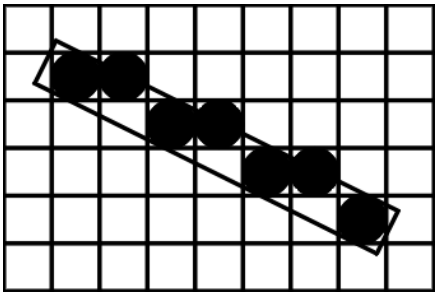


Figure 2: Drawing lines normally produces a jagged result.

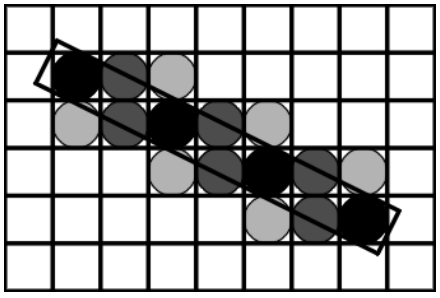


Figure 3: Drawing with antialiasing.

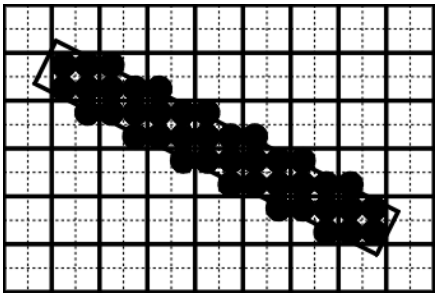


Figure 4: A supersampled line.

You then use the pixel values at the higher resolution to determine the pixel values at the monitor's normal resolution.

For example, if you supersample at twice the monitor's resolution, the supersampled image will be twice as many pixels wide and twice as many pixels tall as the output image. Each output pixel at the monitor's scale corresponds to a two-by-two rectangle of supersampled pixels.

Figure 4 shows the line from Figure 2 supersampled. Each pixel at the original scale has been broken into four smaller pixels at the enhanced scale. The dark circles show the pixels at the supersampled scale that are drawn to represent the line. For each output pixel, you now count the corresponding supersampled pixels that are colored. For example, if three of the four supersampled pixels representing a point are black, you make the output pixel 3/4 black.

One last trick makes all this easy in Delphi. Instead of trying to calculate the pixels to draw at a supersampled resolution, you can simply draw the line at a larger scale. Instead of calculating pixel values at twice the monitor's resolution, you simply draw the line at twice its normal size using Delphi's normal line drawing routines. The resulting image has twice the number of pixels vertically and horizontally as it would normally. You can then examine the colors of the pixels drawn by Delphi to determine the colors for the output pixels.

A Bit of Code

The AAliasP program (available for download; see end of article for details) uses these techniques to draw antialiased text. Enter a string in the text box and click the **Go** button. The program redraws the text at twice its original scale using a font twice as large as the original. It then uses the code shown in Figure 5 to reduce the enlarged image in the *big_bm* bitmap and produce an antialiased image at normal scale in the bitmap *out_bm*. (The helper procedure, *SeparateColor,* breaks a color into its red, green, and blue components. The *RGB* function does the opposite — it combines red, green, and blue values to produce a color value.)

Figure 6 shows the results of the AAliasP program. Notice how much smoother the antialiased text is than the original. Because antialiasing makes objects slightly fuzzier, the program works best when the font used is relatively large. Small fonts may be blurred until they are hard to read.

An Artist's Palette

AAliasP draws black text on a white background. Because it scales the original image by a factor of two vertically and horizontally, each output pixel corre-

```
for y := 0 to (big_bm.Height - 3) div 2 do begin
  for x := 0 to (big_bm.Width - 3) div 2 do begin
    // Compute the value of output pixel (x, y).
    totr := 0;
    totg := 0;
    totb := 0;
    for j := 0 to 1 do begin
      for i := 0 to 1 do begin
        SeparateColor(
          big_bm.Canvas.Pixels[2*x+i, 2*y+j], r, g, b);
        totr := totr + r;
        totg := totg + g;
        totb := totb + b;
      end;
    end;
    out_bm.Canvas.Pixels[x, y] :=
      RGB(totr div 4, totg div 4, totb div 4);
  end;
end;
```

Figure 5: The heart of the antialiasing code.

Figure 6: Antialiasing a string in the sample program AAliasP.

```
// Create a color palette including various combinations
// of yellow, white, black, and aqua.
procedure TAntiAliasForm.SetPalette(bm: TBitmap);
var
  r, g, b              : array [1..4] of Integer;
  totr, totg, totb     : Integer;
  clr, i1, i2, i3, i4  : Integer;
  pal                  : PLogPalette;
  hpal                 : HPALETTE;
begin
  pal := nil;
  try
    GetMem(pal, SizeOf(TLogPalette) + SizeOf(TPaletteEntry) * 255);
    pal.palVersion := $300;

    // Calculate RGB values for the colors.
    SeparateColor(clYellow, r[1], g[1], b[1]);
    SeparateColor(clWhite,  r[2], g[2], b[2]);
    SeparateColor(clBlack,  r[3], g[3], b[3]);
    SeparateColor(clAqua,   r[4], g[4], b[4]);

    // Calculate all combinations of the colors.
    clr := 0;
    for i1 := 0 to 4 do begin
      for i2 := 0 to 4 - i1 do begin
        for i3 := 0 to 4 - i1 - i2 do begin
          // Create the color entry.
          i4 := 4 - i1 - i2 - i3;
          totr := i1 * r[1] + i2 * r[2] +
                  i3 * r[3] + i4 * r[4];
          totg := i1 * g[1] + i2 * g[2] +
                  i3 * g[3] + i4 * g[4];
          totb := i1 * b[1] + i2 * b[2] +
                  i3 * b[3] + i4 * b[4];
          pal.palPalEntry[clr].peRed   :=
            Byte(Round(totr / 4));
          pal.palPalEntry[clr].peGreen :=
            Byte(Round(totg / 4));
          pal.palPalEntry[clr].peBlue  :=
            Byte(Round(totb / 4));
          clr := clr + 1;
        end;
      end;
    end;
    pal.palNumEntries := clr;

    hpal := CreatePalette(pal^);
    if hpal <> 0 then bm.Palette := hpal;
  finally
    FreeMem(pal);
  end;
end;
```

Figure 7: Creating a color palette for antialiasing.

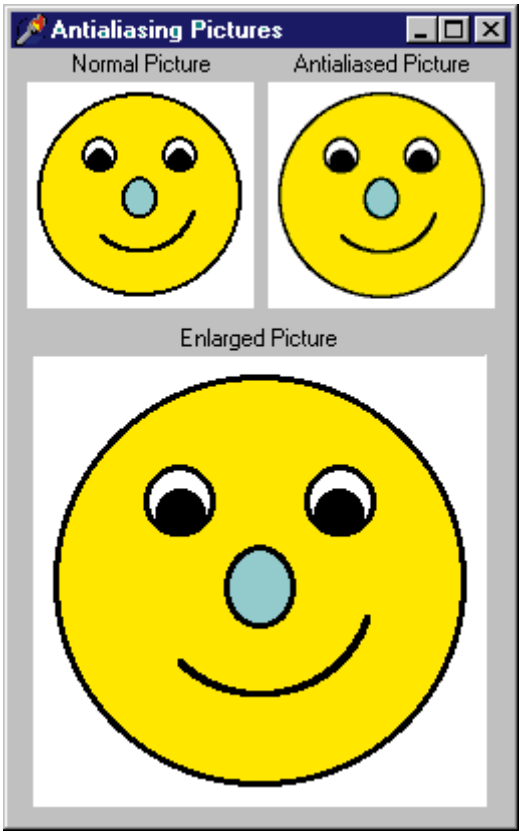


Figure 8: The sample program AAlias2P antialiasing a smiley face.

sponds to four supersampled pixels. This means the resulting pixels can have five different colors: white, 25-percent black, 50-percent black, 75-percent black, and black. The standard Windows colors include enough shades of gray that the result produced by AAliasP is reasonable.

If you add a few more colors to the image, however, the standard Windows colors may not be good enough. The shades of color needed to antialias the image may not be available. In that case you can create your own color palette that includes the colors you might need.

The *SetPalette* procedure shown in Figure 7 creates a color palette for use with a picture that contains black, white, yellow, and aqua. It defines colors for all of the 35 possible combinations of four pixels using these four colors. For example, it defines a color that is 25-percent black, 50-percent yellow, and 25-percent aqua. *SetPalette* then creates a color palette using these colors and assigns it to the bitmap it takes as a parameter. When the program generates an antialiased image in the bitmap, all of the colors it needs are available.

The program AAlias2P uses the *SetPalette* procedure to give its output picture an appropriate color palette. It then draws a picture of a smiley face and generates an antialiased version. The program is shown in Figure 8. The antialiased picture is smoother than the original, though the effect may not be as striking as it is with antialiased text.

AAlias2P doesn't use all of the 35 colors defined by *SetPalette*. For example, in the original image, aqua is adjacent only to black, not white or yellow. This means none of the colors containing a mixture of aqua and white or aqua and yellow are actually needed. For this picture, it does no harm to define the colors anyway. If the picture used a few more colors, however, there could easily be more than 256 color combinations. In that case, the program would need to have been more selective about the colors it added to the color palette.

Conclusion

While antialiasing is too much work to be practical for every label and text box, you can use these techniques to remove the rough edges in a few special places. Using antialiased images on splash screens, About boxes, and large section titles can give your applications a smooth, polished look. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\APR\DI9804RS.

Rod Stephens is the author of several books, including *Visual Basic Graphics Programming* and *Visual Basic Algorithms,* both from John Wiley & Sons. He also writes algorithm columns in *Visual Basic Developer* and *Microsoft Office & Visual Basic for Applications Developer.* You can reach him at RodStephens@compuserve.com or see what else he's up to at http://www.vb-helper.com.

By Chris Austria

The Spirit of Nagano?

The 1998 *Delphi Informant* Readers Choice Awards

Maybe it was the anxiety of a long, arduous winter. Or maybe it was the spirit of the Winter Olympics fueling the competitive fires of Delphi vendors and developers. Whatever it was that stimulated the competition, we're thankful for it. It made for an exciting year filled with innovation, perseverance, and progress from people dedicated to delivering the best products to the market.

The all-powerful voice of the people has spoken and has chosen what it deems to be the best tools-of-the-trade. It's time to see how well your best-of-the-best choices match up with your peers'. Some who have followed previous *Delphi Informant* Readers Choice Awards may be in for a surprise; others may find themselves nodding their heads knowingly. First-timers will come to understand what all the fuss has been about.

Again, several changes were made to this year's ballot to accommodate ever-shifting technological trends. The Best Database Server and Best Version Control categories have been omitted, while a new category, Best Charting Component, has been added. Many of last year's participants hung on for another year; others were quietly ousted and replaced. New players managed to cram themselves into already jam-packed categories, forcing us to reshape some boundaries. However, one thing remained the same: the intensity of the competition.

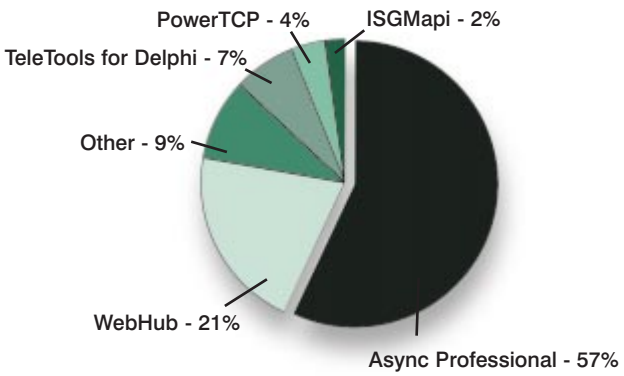
Having set the stage, here are the winners…

Best Connectivity Tool

Previously named Best Internet/Communications, this category had essentially the same participants as last year, with the exception of TeleTools for Delphi from ExceleTel, Inc. (which debuted this year in a respectable fourth place). The top two finishers, TurboPower's Async Professional and HREF Tools' WebHub, are this year's first repeat performers. WebHub gained significant ground this year, earning 21 percent of the votes (compared to last year's seven percent), and seems to be closing in on the popular Async Professional, which received 57 percent (compared to last year's 78 percent) to win the category.

But don't count on TurboPower to rest on their laurels. They've started shipping Async Pro 2.5, which adds business telephony features, such as .WAV file recording and playback, DTMF tone detection and generation, voice-to-fax handoff, and ISDN support to its powerful communications library.

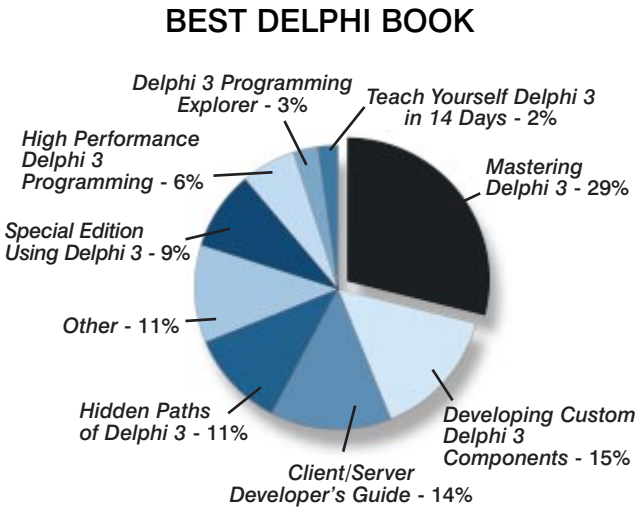
BEST CONNECTIVITY TOOL



Best Delphi Book

The book that kept the most developers up late was Marco Cantù's *Mastering Delphi 3* from SYBEX, collecting enough votes to claim 29 percent of this category. *Mastering Delphi 3* not only covers Delphi 3's powerful new features, it provides tips and techniques that allow developers to create innovative Delphi applications.

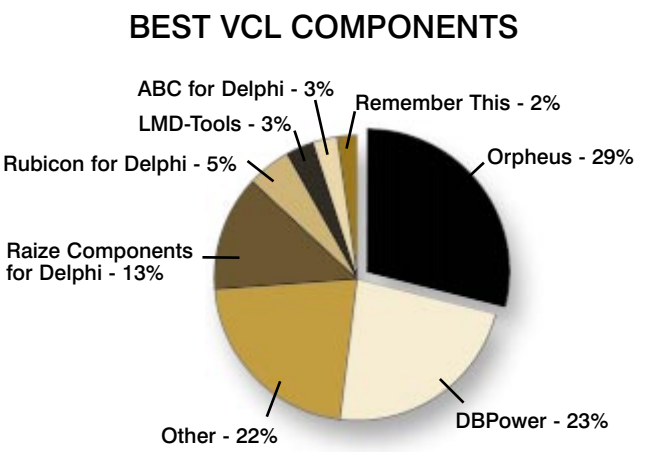
Also banking on the latest version of Delphi was Ray Konopka's *Developing Custom Delphi 3 Components* from The Coriolis Group. Coming in second, the "sequel" to last year's *Developing Custom Delphi Components* collected 15 percent of the votes.



Best VCL Component

This year's VCL category presented several new players. In the end, two contenders stood above the rest. Orpheus by TurboPower out-muscled DBPower from Luxent Software (which acquired DFL Software and Successware Intl. in June, 1997) by a narrow 5 percent. Orpheus, last year's runner-up, offers over 50 components to expand your library, whether you're using Delphi 1, 2, or 3. It adds capabilities such as incremental search fields, scrolling list boxes, and data-entry field editors for every native VCL data type.

Worthy of a second look is DBPower, the second-place winner and a new participant in this category. Amassing a solid 24 percent, DBPower offers over 35 DBMS controls for Delphi and C++Builder, including super grids, image buttons, and Quicken-style lookups.



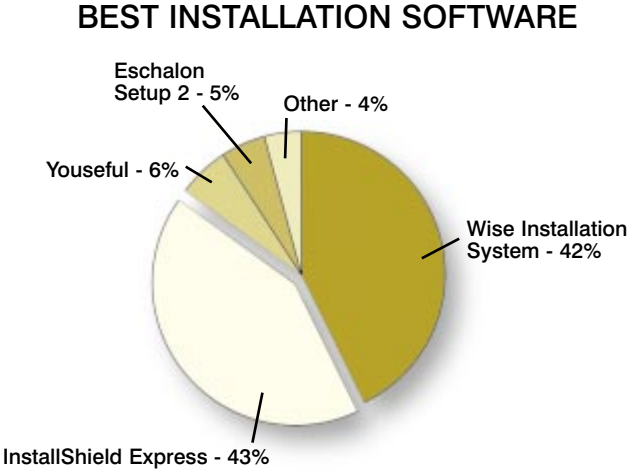
Best ActiveX

Previously named Best OCX, the Best ActiveX category has a familiar face in its winner's circle: Luxent Software. Luxent accumulated 60 percent of all ActiveX votes with its Light Lib Magic Menus. This component allows developers to create visual and dynamic user interfaces for their applications using background images, textures, bitmap menu items, and tool-button palettes.

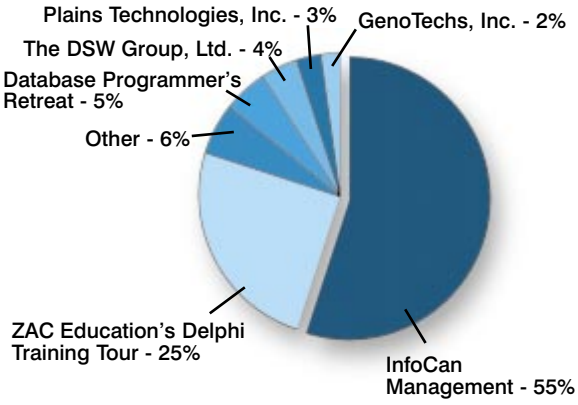
Second place went to last year's winner, OCX Expert from Apiary, Inc., and ActiveX Voice Tools from Speech Solutions, Inc. Each collected 8 percent of the votes.

Best Installation Software

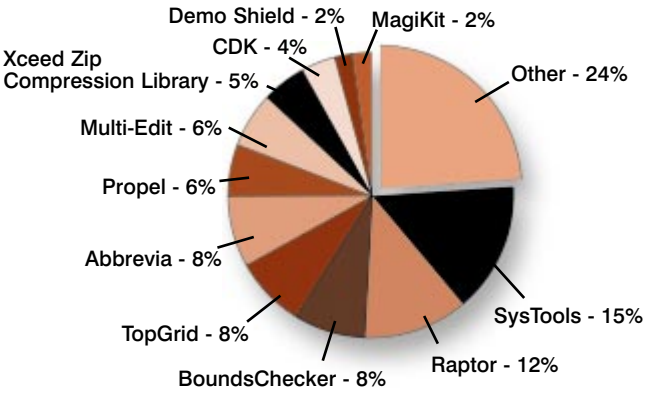
This one was a real nail biter. The top two contenders from last year, InstallShield Express from InstallShield Software and Wise Installation System from Great Lakes Business Solutions, made repeat performances this year. This time around, the gap between them was smaller — much smaller. InstallShield Express edged out Wise Installation System by a single percentile (InstallShield took 43 percent and Wise took 42 percent). It seems they've both established a strong foothold on the installation software market. We'll see what next year brings.



BEST TRAINING



BEST DELPHI ADD-IN



Best Training

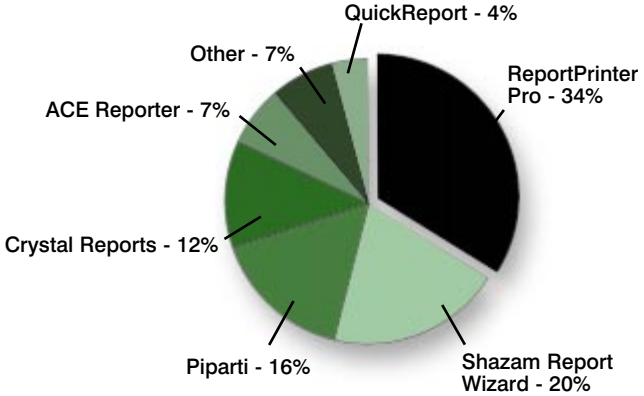
I think everyone learned a little something after last year's Best Training competition, especially InfoCan Management. Last year's runner-up, InfoCan, gained enough ground to not only pass ZAC Education's Delphi Training Tour, but it won by a significant margin, garnering 55 percent to ZAC Education's 25 percent. InfoCan is the only Canadian-based training company on the ballot, and offers a variety of Delphi-related training courses. InfoCan is also a Borland Connections SI/VAR & Training Member.

Best Reporting Tool

This category brings another repeat winner: ReportPrinter Pro from Nevrona Designs. Amassing 34 percent (14 points higher than the second place winner), ReportPrinter Pro stands tall after last year's intensely close race in which ReportPrinter won by a slim three percent. Offering 11 components with over 400 methods, properties, and events, and full 16- and 32-bit source code, ReportPrinter Pro proved too tough to topple.

Something to keep a watchful eye on, however, is the runner-up, Shazam Report Wizard from Shazamware Software Solutions. At 20 percent, compared to less than three last year, Shazam promises to be a worthy rival to ReportPrinter Pro in the next Readers Choice Awards.

BEST REPORTING TOOL



Best Delphi Add-In

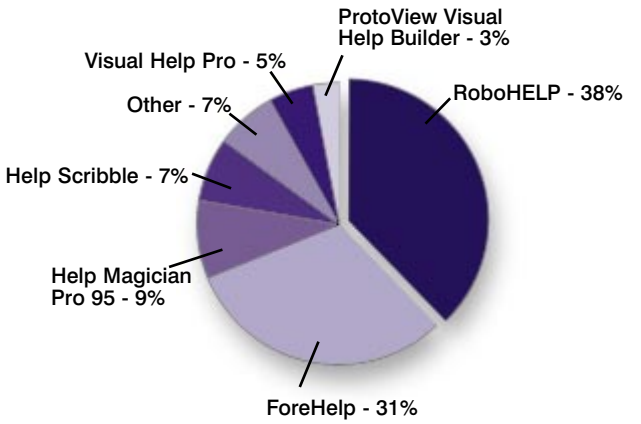
The largest category also added many new players to the game, but last year's top vote-getter, SysTools from TurboPower, retained its title of Best Delphi Add-In with 15 percent of the votes. With this award, TurboPower set a new record for having the most first-place products in the same year of competition (Async Pro and Orpheus were the first two). SysTools provides over 600 routines for string manipulation, date and time arithmetic, high-precision mathematics, and sorting.

Coming in a close second is Raptor from Eagle Software with 12 percent — only three points behind SysTools. Can this rookie player knock down the veteran next year, or will TurboPower add another add-in award to its trophy case?

Best Windows Help Authoring Tool

Last year, ForeFront Inc.'s ForeHelp took pole position with 40 percent, with RoboHELP from Blue Sky Software trailing closely at 37 percent. This year delivered another exciting finish. RoboHELP outscored ForeHelp 38 to 31 percent, establishing itself as the new champion of Windows Help Authoring Tools.

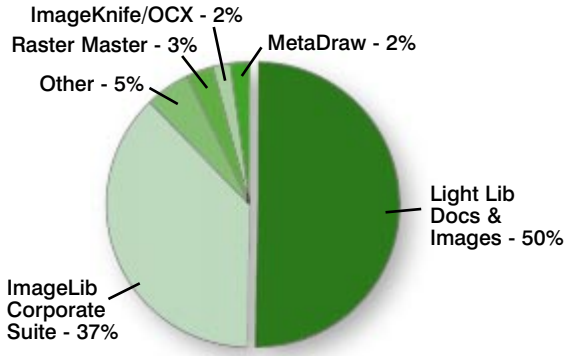
BEST WINDOWS HELP AUTHORING TOOL



Best Imaging Component

Picture this: Light Lib Docs & Images, from Luxent Software, gathers 51 percent to take first place as Best Imaging Component. Light Lib Docs & Images paints a pretty picture using support for most image formats and compression types, 24-bit color image display, an integrated image gallery, an image navigator, and a whole list of picture-perfect features.

BEST IMAGING COMPONENT



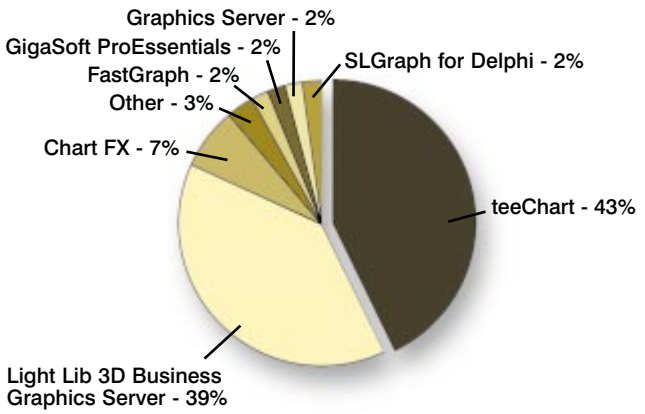
The popular ImageLib Corporate Suite, from Skyline Tools (last year's winner with the same score of 37 percent), took second place. Stay focused on this category for a great race.

Best Charting Component

Our charts showed us that a Charting Component category would be quite practical and well received, and we were right. The numerous responses to this brand new category yielded teeChart from teeMach SL as a winner with 43 percent. teeChart, packaged with Delphi 3, offers extensive and flexible charting capabilities and is available in a professional version that offers Candle, Volume, and Bar series types and extended function types, such as Moving Average and Relative Strength Index.

Right on teeChart's heels was Light Lib 3D Business Graphics Server from (no other than) Luxent Software, with a total of 39 percent.

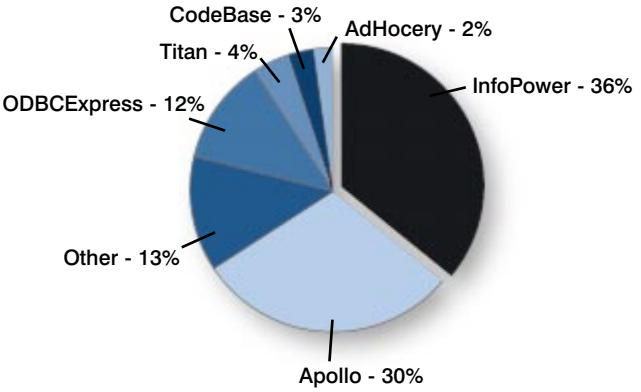
BEST CHARTING COMPONENT



Best Database Tool

In its second year, this category has been expanded to include several new products, making the fight to the top that much more challenging. The first to reach the peak this year was InfoPower from Woll2Woll Software. No stranger to winning — it was last year's Product of the Year — InfoPower flexed its database muscles to take 36 percent of the votes, passing Apollo from Luxent Software, which won last year with a whopping 72 percent. But Apollo had plenty of thrust left in its jets, finishing with 24 percent this year.

BEST DATABASE TOOL

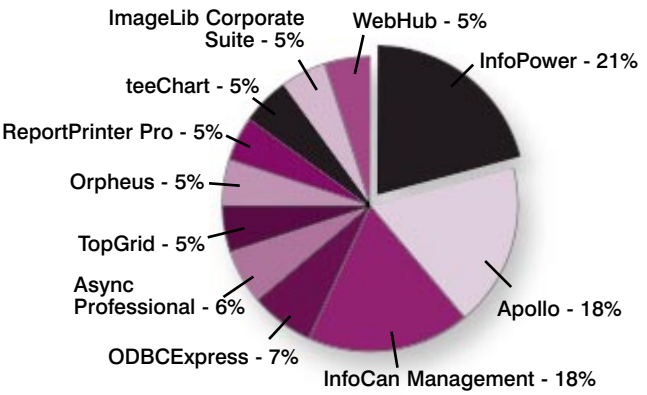


Product of the Year

There was a tremendous response to the Product of the Year (we tallied approximately 150 products), but there can only be one first-place winner. For the third time in three years, InfoPower from Woll2Woll Software came out on top, garnering 21 percent of the votes for this category.

Although InfoPower is well deserving of this award, an 18 percent showing by Luxent's Apollo, as well as InfoCan Management, certainly merits recognition. Only three percent behind InfoPower, Apollo and InfoCan gained significant ground this past year. (Note: The accompanying chart reflects the percentage points earned by the top 10 finalists only. A pie chart including all entries would not have effectively rendered the relationship among the winners.)

PRODUCT OF THE YEAR



Thank You All

This edition of the Readers Choice Awards, while being one of surprises and upsets, also had its fair share of repeat performers — products and companies that, after establishing themselves in last year's awards, made their presence known again this year. Instead of resting with the knowledge that they have accomplished their goals, they endeavored to go further than ever before, formulating new goals and strategies, then executing them to perfection. In the end, we all benefit; the vendors bask in the glory of being the best at what they do; developers can choose the best tools to make better applications; and consequently, the end users benefit from having powerful and innovative products that help make their everyday lives more productive. It's a great operation.

Somewhere in the middle of it all, we at *Delphi Informant* are constantly gathering and translating the information that you, our readers, need to know to make your Delphi-related decisions a little easier. We thank you all for voting and making your voices heard. Congratulations to all the winners and to all who reached their goals. We look forward to seeing you next year. Δ

Chris Austria, Products Editor at Informant Communications Group, can be reached via e-mail at caustria@informant.com.

Contacting the Winners

Best Connectivity Tool
Async Professional
TurboPower Software
Phone: (800) 333-4160 or (719) 260-9136
Web Site: http://www.tpower.com

Best Delphi Book
Mastering Delphi 3
Marco Cantù
SYBEX
Phone: (510) 523-8233
Web Site: http://www.sybex.com

Best VCL Component
Orpheus
TurboPower Software
Phone: (800) 333-4160 or (719) 260-9136
Web Site: http://www.tpower.com

Best ActiveX
Light Lib Magic Menus
Luxent Software
Phone: (909) 699-9657
Web Site: http://www.luxent.com

Best Installation Software
InstallShield Express
InstallShield Corp.
Phone: (800) 374-4353 or (847) 240-9111
Web Site: http://www.installshield.-com

Best Training
InfoCan Management
Phone: (888) INFOCAN or (604) 736-5888
Web Site: http://www.infocan.com

Best Reporting Tool
ReportPrinter Pro
Nevrona Designs
Phone: (888) 776-4765 or (602) 491-5492
Web Site: http://www.nevrona.-com

Best Delphi Add-In
SysTools for Delphi
TurboPower Software
Phone: (800) 333-4160 or (719) 260-9136
Web Site: http://www.tpower.com

Best Windows Help Authoring Tool
RoboHELP
Blue Sky Software
Phone: (800) 793-0364 or (619) 459-6365
Web Site: http://www.blue-sky.-com

Best Imaging Component
Light Lib Docs & Images
Luxent Software
Phone: (909) 699-9657
Web Site: http://www.luxent.com

Best Charting Component
teeChart
teeMach, SL
Phone: 34 72 59 71 61
Web Site: http://www.teemach.-com

Best Database Tool
InfoPower
Woll2Woll Software
Phone: (510) 371-1663
Web Site: http://www.woll2woll.-com

Product of the Year
InfoPower
Woll2Woll Software
Phone: (510) 371-1663
Web Site: http://www.woll2woll.-com

The Tomes of Delphi 3: Win32 Core API

There's been a buzz about this book on the Internet since last Fall. Finally, as 1997 came to an end, the first volume of this eagerly anticipated series by John Ayres, et al. was released: *The Tomes of Delphi 3: Win32 Core API*.

Tomes is nothing less than a milestone in Delphi publishing. No longer will we feel like second-class citizens of the Windows development community. No longer will we look at our C/C++ brethren with envy while secretly studying Waite "Bibles" to learn basic, low-level Windows API techniques. No longer will we need to translate everything in those Windows references from C to Pascal. Let's see what's included in this first volume.

After an overview of the Windows API, the authors delve into an important, but seldom broached topic: qualifying for the Windows 95 logo. This exposition is comprehensive and very well done, explaining what Delphi provides, what additional resources you might need, and where to find more information.

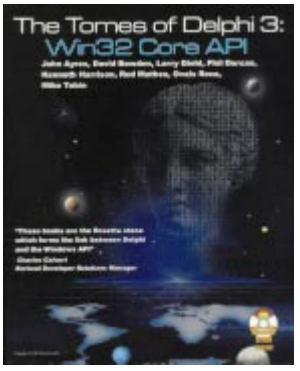
Next, there are three lengthy chapters on topics concerning windows: creating them, sending messages to/from them, and getting information about them. All of the basic API functions needed to create and manage windows are covered, including *CreateWindow*, *DestroyWindow*, *DefWindowProc*, *DispatchMessage*, *SendMessageCallback*, *TranslateMessage*, and a multitude of Windows information functions.

However, you may be asking, "What can I do with this information that I can't already do with Delphi alone?" Let's see.

The *CreateWindowEx* function, that is described in some detail in the first of these chapters, gives you the ability to create a variety of window types, including ones which can accept files dragged from other applications, floating toolbars, and windows with various border styles. Besides the basic message-handling functions, the chapter on messages includes detailed information about the message process itself. It covers a number of interesting topics, such as hooks, which allow you to intercept and handle messages. There's also a discussion and example of how to create and register your own messages.

Of these three chapters, "Window Information Functions" is the lengthiest, covering a plethora of such functions. For example, you learn how to use *EnumWindows* to traverse all current top-level windows, and *FindWindowEx* to get the handle of a specific window. There is also a discussion of the various functions that allow you to manipulate windows, including *SetClassLong* and *SetFocus*.

Chapter 6 explains the Windows multi-processing, thread-related functions. While Delphi, with its *TThread* class, encapsulates and simplifies a good deal of the basic multitasking functionality, this chapter gives you the information you need to go further, by working with critical sections, semaphores, and mutexes. Next, there are two chapters covering dynamic link libraries, INI and registry files, and memory management. As before, the treatment is detailed and very easy to follow. Developers who find Delphi's memory management tools insufficient to meet their needs will be delighted with the new possibilities expounded in the chapter on "Memory Management Functions."



The remaining chapters explore familiar topics such as the Clipboard, input devices, file I/O, and system information. Developers writing international applications will find the tools and techniques

"The Tomes of Delphi 3" continued on page 38

The Tomes of Delphi 3 (cont. from page 37)

in the "String and Atom Functions" chapter useful. Basic string-manipulation functions and others which provide access to Windows' internationalization information are included.

Toward the end of the book, there's a rather lengthy chapter on "System Information Functions," functions useful for ascertaining and/or changing a multitude of Windows settings on a particular machine, including accessibility features, date and time settings, user information, and hardware information. The book concludes with a couple of short chapters on timer and error functions.

The Tomes of Delphi 3: Win32 Core API is very well organized and clearly written. It includes many code examples, all of which are included on its accompanying CD-ROM. Each chapter begins with a concise description of the relevant topics, a description of each major function, and tables describing the various function parameters and return values. This book will be indispensable for any developer who plans to make use of Windows API functions or develop custom components that rely on those functions. It also provides an excellent road map of Delphi's Visual Component Library source code, which — of course — itself uses many of these functions. While some of this information has appeared in other Delphi books, this is the first time it has all been gathered in one place and organized into a highly usable and comprehensive reference. I recommend it highly.

— *Alan C. Moore, Ph.D.*

The Tomes of Delphi 3: Win32 Core API by John Ayres, David Bowden, Larry Diehl, Phil Dorcas, Kenneth Harrison, Rod Mathes, Ovais Reza, and Mike Tobin, Wordware Publishing, Inc., 2320 Los Rios Boulevard, Plano, TX 75074, (972) 423-0090, http://www.wordware.com.

ISBN: 1-55622-556-3
Price: US$54.95
(788 pages, CD-ROM)

The Upgrade Game

Quick show of hands. How many of you upgraded to Delphi 3.01? Okay. How about Delphi 3.02? BDE 4.51? How about the various service packs for your operating system?

Software companies are working at a feverish pace to develop and release new versions of their products as soon as humanly possible. Maintenance releases and patches are being produced with even greater regularity. With each new version you can count on several new features — and several new bugs. For end users, it's no fun to play the upgrade game; sometimes upgrading one part of a software package introduces problems in another part.

It's even worse for developers. To successfully deploy quality applications today, you need to stay on top of every .001 release, and monitor how it interacts with every other part of your application. The amount of changes from Redmond alone can cripple a budding software company trying to support all the possible permutations of software configuration on an end user's machine.

Why, then, should you play this game? Developing software using outdated tools and libraries puts you at a distinct disadvantage in obtaining support. Most developers tend to update their development software when a new version comes out, so don't go to the newsgroup expecting to get the same level of help for Delphi 2 and Delphi 3. It's not going to happen.

When was the last time you were stuck maintaining old code? It probably wasn't the most glamorous position, and it certainly wasn't fun. Programmers want to build. They want to evolve the creation they started into something aesthetically pleasing. Maintaining dated applications doesn't fit the bill.

We can look at the reason software updates occur. This may help explain why it's beneficial to play the upgrade game. New versions are built to provide increased functionality. What was difficult — or impossible — to accomplish with the previous version, is made easier in the new one. This is why it's extremely important to keep your software current; subtle — and not-so-subtle — flaws are worked out in subsequent versions.

This is especially true if you are on the leading edge of technology, or using some of the new features of Delphi 3. For example, multi-tier development with MIDAS, One-step ActiveX, and WebModules were all introduced in Delphi 3. Each maintenance release has corrected problems in these areas. If you don't upgrade, you'll be stuck trying to find workarounds for problems that are already fixed, i.e. re-inventing the wheel.

The upgrade to Delphi 3.01 was welcome for many reasons. Changes were made, bugs were fixed, and documentation was updated on such a grand scale that Borland decided to make the upgrade a full install, as opposed to a patch. Even at US$15 for an upgrade, it's a good deal. Some of the major changes and additions in Delphi 3.01 include:

- revised documentation, including additions of example code
- *TMidasConnection* to allow multi-tier development with TCP/IP and OLEnterprise (Client/Server Suite only)
- updated ISAPITER.DLL to accommodate Netscape Server 3
- addition of Socket components to the Professional version
- a CAB file to easily deploy, install, and configure the BDE over the Internet

There are many more updated features; check http://www.borland.com/-delphi/del3update.html for details. However, even this short list shows that Borland has been listening to its customers. They're striving to make the product better with each release. They're even introducing new features in a .01 release! Borland has also increased the frequency of maintenance updates. No longer do you have to wait for Borland to release the next version of Delphi to get a bug fix.

Now, if you *really* want to be up-to-date, Borland released Delphi 3.02 in December 1997 (again, see http://www.borland.com/delphi/-del3update.html for details). You can download this patch from the Web site, but you need to be using Delphi 3.01 to apply it. You can also find the latest version of the BDE, version 4.51, at http://www.borland.com/devsupport/-bde/bdeupdate.html. Δ

— Dan Miser

Dan Miser is a Design Architect for Stratagem, a consulting company in Milwaukee. He has been a Borland Certified Client/Server Developer since 1996, and is a frequent contributor to Delphi Informant. *You can contact him at http://www.iinet.com/users/dmiser.*

Best Vendor Web Sites

In his July, 1997 "File | New" column, Richard Wagner discussed several independent Delphi Web sites, most of which featured Delphi components, resources, and information. This month, I will vary that theme a bit, and discuss seven sites maintained by third-party Delphi producers.

The Veterans. While I won't be discussing Borland's site specifically, I will use its features-detailed product information, technical papers, file download area, support facilities, and links as benchmarks for assessing the other sites. Besides Borland, there's one other company that's certainly a veteran. That company, TurboPower, has been supporting Borland products nearly as long as the latter has been in existence. You'd expect them to have their Web act together, and they certainly do. Granted, you won't find a lot of general Delphi information or free stuff on their site. However, you will find excellent information on their many products. With their newsgroups, weekly product tips, and a wish list for customer input on product upgrades, they provide many avenues for customer communication on their site.

Now, we'll shine the spotlight on two vendors whose sites go beyond just support of their products.

Eagle Software and Raize Software Solutions. I've spent a lot of time at the Eagle Software site, following with interest the beta development of Raptor and CDK (Component Development Kit) 3. But there's much more. I'm particularly impressed with Mark Miller's outstanding papers, particularly the one on "Good Class Design." Another outstanding programmer and writer, Ray Konopka, is the chief architect of Raize Software Solutions. The Raize site provides ample information and support for the Raize components, and also contains an invaluable archive of articles Konopka has written for *Visual Developer* magazine and its predecessor, *PC Techniques*.

Outstanding Component Producers. HREF Tools Corp. (maker of the well-known WebHub components) and Shoreline Software (maker of Web Solution Builder) are two vendors who produce Delphi Web components. HREF's site is large and very well organized. While their educational material is geared toward potential or existing WebHub customers, you can learn a great deal about building a Web site here. On the other hand, Shoreline's site demonstrates the creative use of multimedia with its use of music and animation.

Speaking of multimedia, Skyline Tools is one of the leaders in multimedia Delphi add-ons, producing the award-winning ImageLib components. Their site is attractively laid out and easy to navigate. Nevrona Designs' — producer of three Delphi add-ons: AdHocery, Propel, and the award-winning ReportPrinter Pro — is another attractive site. As with some of the Eagle Software products, you can trace the development of ReportPrinter Pro on their site. Be sure to download their multimedia Propel demo and some of the free features produced with Propel. (Watch for my review of this product in an upcoming issue.) It's worth the visit.

Rating the Vendor Sites. How we rate vendor sites depends on what we're looking for. If we're most interested in product information and customer support, TurboPower is hard to beat. On the other hand, if we're looking for general programming information or free components, it's a close contest between Eagle Software and Raize Software Solutions. In terms of innovation, HREF's "portfolio" of links to sites built with WebHub is a strong marketing tool for that product. And Nevrona has some refreshing programming humor that I really enjoyed. My summary of the sites' features is given in the table below. I recommend visiting all of them. As I plan to return to this theme again, please send me your favorite Delphi sites at acmdoc@aol.com. Δ

— Alan C. Moore, Ph.D.

All URLs begin with http://www.	Downloads	Technical papers	Free components, services, and add-ons	Online support	Delphi links
Eagle Software eaglesoftware.com	Excellent	Excellent	Excellent	Very Good	Fair
HREF Tools Corp. webhub.com	Excellent	Very Good	None	Very Good	Fair
Nevrona Designs nevrona.com	Excellent	Good	Good	Good	Excellent
Raize Software Solutions raize.com	Excellent	Very Good	Excellent	Good	Excellent
Shoreline Software shoresoft.com	Excellent	Good	Very Good	Very Good	Very Good
Skyline Tools imagelib.com	Excellent	None	None	Good	Good
TurboPower turbopower.com	Excellent	Good	None	Excellent	None

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