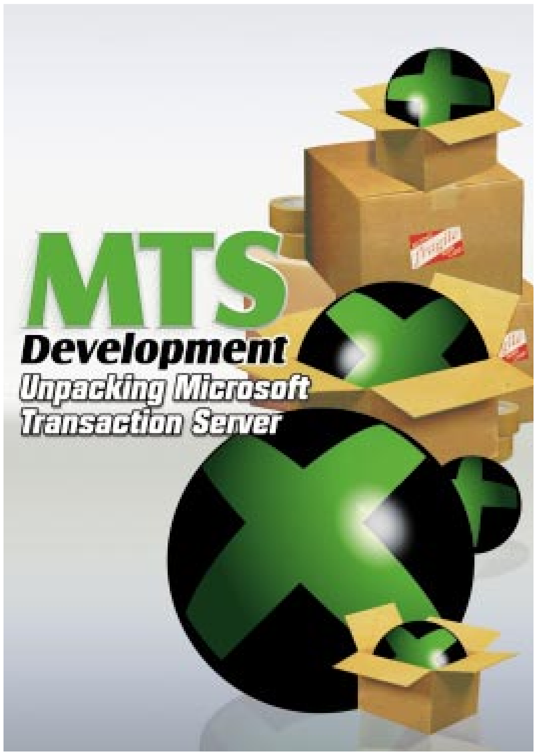
December 1998, Volume 4, Number 12



Cover Art By: *Darryl Dennis*

Delphi
T O O L S

New Products and Solutions

DT Software Releases Version 5.1 of dtSearch Text Retrieval Engine

DT Software, Inc. introduced version 5.1 of its ***dtSearch Text Retrieval Engine*** for the PC, LAN, and Internet/intranet. This edition makes it easier for Windows 95/98/NT developers to link the 32-bit dtSearch Engine to a range of enterprise data.

The dtSearch Text Retrieval Engine provides easy indexing and searching of any data accessible using ActiveX interfaces. The ActiveX support allows indexing of SQL databases using Active Data Objects, Data Access Objects, and Remote Data Objects. The API can also be used to index message stores using Collaboration Data Objects and Web sites using the Internet Client SDK.

The new version adds Active Server Pages (ASP) support, which enables developers to create server-side ASP scripts to add customized text searching features to an Internet or intranet site. Developers can use Delphi, ASP, Visual Basic, VBScript, or C/C++ to control the dtSearch Engine and add a custom user interface. The dtSearch Engine provides sample code in these languages, including source code to a fully functional, ISAPI-based Internet search engine.

The dtSearch Engine has unlimited capacity and includes built-in support for parsing word processor, database, spreadsheet, HTML, ZIP, PDF, and other file types. It also offers over two dozen search options, including relevancy-ranked natural language, thesaurus, fuzzy, and/or/not, proximity, phrase, wildcard, phonic, field, numeric range, and variable term weighting.

DT Software, Inc.
Price: From US$999
Phone: (800) 483-4637 or (703) 413-3670
Web Site: http://www.dtsearch.com

Dalco Announces dbOvernet

Dalco Technologies announced the release of dbOvernet, a MIDAS-alternative toolset that enables the rapid development of multi-tier applications using Delphi approaches and drop-and-go component technology. dbOvernet provides six controls for producing thin, client-side and plug-in, server-side applications.

Open extensibility enables the deployment of server-side applications against any database or data source, and the client-side dbOClientDataset component handles all communication and data delivery between the server and client data-aware controls.

Using TCP/IP socket-enabled methods, dbOvernet deploys applications over all transports, including LAN, WAN, and intranet, and provides real-time, two-way data communication over the Internet.

For more information, call Dalco Technologies at (250) 769-3951, or visit their Web site at http://www.dbovernet.com.

Raize Software Announces CodeSite 1.1

Raize Software Solutions, Inc. announced ***CodeSite 1.1***, an advanced debugging tool that provides several enhancements and Delphi 4 and C++Builder 3 support.

The CodeSite Object, used to send messages to the CodeSite Viewer, defines three new methods and one new property.

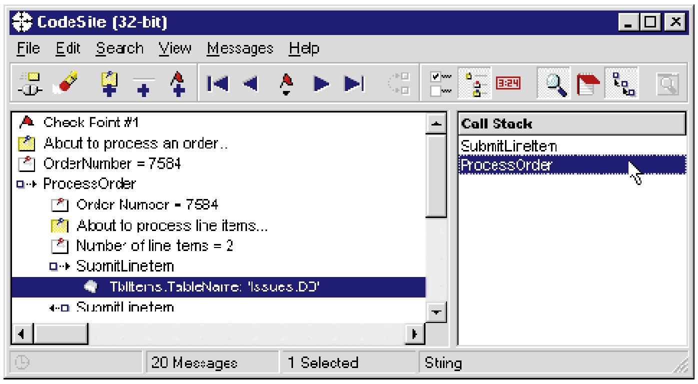
The first method is *SaveLogFile*, which allows a developer to instruct the CodeSite Viewer to save a log file containing all messages currently displayed in the viewer. The *SendDateTime* and *SendDateTimeEx* methods are used to send *TDateTime* values to the viewer, while the *DateTimeFormat* property is used to control how the values appear in the viewer.

The CodeSite Viewer has an option on the **Edit** menu that allows a user to reset the indent level. All panes in the view implement thumb tracking when scrolling. Cut, copy, and paste operations work correctly in the popup edit window displayed when editing an item in the message list.

Also, the user interface of the 16-bit version of the CodeSite Viewer running under Windows 3.x has been cleaned up.

The CodeSite Message Expert, which provides a point-and-click interface for constructing CodeSite messages when using CodeRush, remembers the last set of selected options and restores these options the next time the expert is displayed. The settings are also maintained across Delphi sessions.

Raize Software Solutions, Inc.
Price: US$79.95; free upgrade for users of version 1.0.
Phone: (630) 717-7217
Web Site: http://www.raize.com



```
Check Point #1
About to process an order...
OrderNumber = 7584
ProcessOrder
    Order Number = 7584
    About to process line items...
    Number of line tems = 2
    SubmitLineItem
        TblItems.TableName: 'Issues.DB'
    SubmitLineItem
```

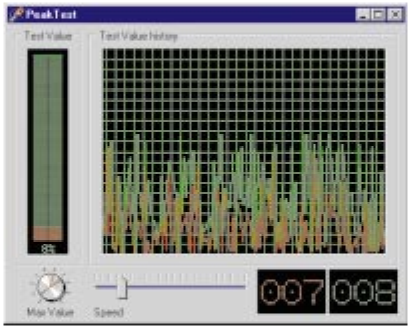
```
SubmitLineItem
ProcessOrder
```

```
20 Messages    1 Selected    String
```

ZieglerSoft Announces ZieglerCollection one Version 1.40

ZieglerSoft announced the availability of ***ZieglerCollection one Version 1.40***, which supports Delphi 1 through 4 and C++Builder 1 and 3.

ZieglerCollection one 1.40 includes 60 components and a collection of functions and routines, including *TzMinMax*, *TzBigLabel*, *Tz3Dlabel*, *TzAngleLabel*, *TzTabListBox*, *TzBitmap*, *TzAnimated*, *TzBackground*, *TzSegmentClock*, *TzGauge*, *TzSlideBar*, *TzFrame*, *TzDivider*, *TzMovePanel*, *TzTitleBar*, *TzHint*, *TzShowApp*, *TzVerSpilt*, *TzHorSplit*, *TzMouseSpot*, *TzCalc*, *TzShapeBtn*, *TzColorBtn*, *TzGradBtn*, *TzBitColBtn*, *TzIconColBtn*, and others.



ZieglerSoft
Price: US$52 (with full source code).
Phone: +45 9811 3772
Web Site: http://www.zieglersoft.dk

UnitOOPS Announces OLE Drag and Drop Components

UnitOOPS Software announced the UnitOOPS OLE Drag and Drop Components, four VCL components that allow a Delphi application to be the source or target of inter-application drag-and-drop of text and images (bitmaps or metafiles).

The components provide Delphi developers access to common inter-application (OLE) drag-and-drop capabilities with little user code. Also, they trigger events that can be handled to get more control over the drag-and-drop process. Plain text, file lists, rich text, Microsoft HTML format, URL links, bitmaps, DIBs, and metafiles can be accepted, encapsulated in Delphi-style objects (e.g. graphical content is always in a *TPicture*, text is always in a Delphi string or *TStringList*).

Applications using the components can run in Windows 95/98 and Windows NT 4 and higher.

For more information, call UnitOOPS at (203) 891-8333, or visit their Web site at http://www.pobox.com/~unitoops.

CNS Introduces The InterCom System

CNS International announced the release of *The InterCom System*.

The system is a tool intended for developers creating multi-user network applications.

This includes database and Internet applications, games, and groupware.

The InterCom System consists of a client control, used by the application, and the InterCom server.

The system is based on a Publisher-Subscriber architecture, where clients can subscribe to "events" that the developer defines, and publish data to those events.

When data is published to an event, all clients subscribed to that event are automatically notified. Clients can also send messages directly to other clients, which can be located anywhere on a LAN or WAN.

All communication is handled directly by the server.

Both network bandwidth and server usage can be kept low, as The InterCom System can utilize IP Multicasting technology.

The product can be used from various programming environments, including Delphi, Visual Basic, and Visual C++.

CNS International
Price: Not available at press time.
Phone: (31) 30 2802822
Web Site: http://www.cns.nl

Objective Releases Version 4 of ABC for Delphi

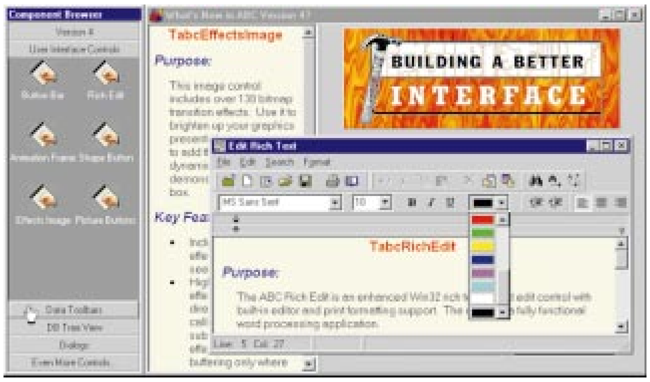
Objective Software Technology Pty Ltd. announced the release of ***ABC for Delphi Version 4***, a comprehensive set of data-navigation, presentation, and exception-handling components.

ABC for Delphi Version 4 includes user interface controls, such as Button Bar, Animation Frame, Effects Image, Shape Button, Picture Speed Button, and Rich Edit; floating toolbar components; and database controls, such as DB Tree View, DB Rich Edit, and Hint and Help Manager components.

In addition, ABC for Delphi Version 4 offers professional dialog boxes, including Splash Screen and Welcome Tips.

ABC for Delphi Version 4 supports Delphi 1 through 4 and C++Builder 1 and 3.

Objective Software Technology Pty Ltd.
Price: US$149; upgrade from version 3, US$69; ships on CD-ROM with 100MB of source code, sample programs, and run-time images for all versions of Delphi and C++Builder.
Phone: +61 2 9955 3397
Web Site: http://www.obsof.com



Delphi TOOLS

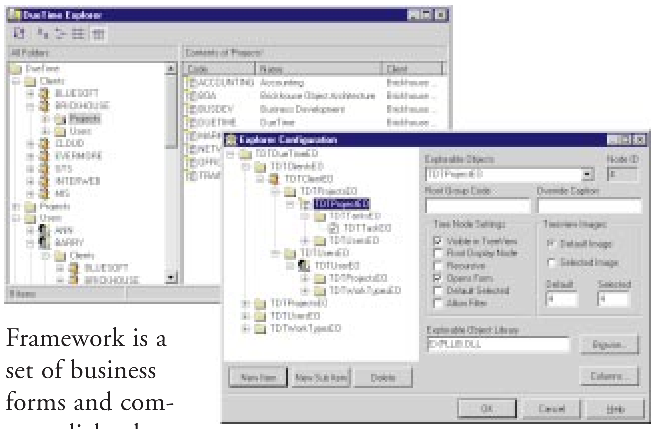
New Products and Solutions

Brickhouse Launches Brickhouse Object Architecture

Brickhouse Data Systems, Inc. launched its *Brickhouse Object Architecture* (BOA) technology to provide object-oriented development teams with an application development framework for deploying scalable *n*-tier business systems.

BOA structures applications so that the business objects are physically separated from the front-end application. Developers can select tools and deliver business applications more efficiently, with BOA wrapping everything together.

The Business Object Framework includes a Business Object Class Library, a Business Object Factory that acts as an object broker to interact with key system components, and a Business Object Wizard to help create business objects and link them to database tables.



Code	Name	Client
ACCOUNTING	Accounting	Brickhouse ...
BOA	Brickhouse Object Architecture	Brickhouse ...
BUSDEV	Business Development	Brickhouse ...
DUETIME	DueTime	Brickhouse ...

The Business Forms Framework is a set of business forms and common dialog boxes.

Visual Business Components are "business-object aware" versions of data controls, allowing presentation of business object data using a standard Windows user interface.

The Business Explorer is modeled around the Microsoft Windows Explorer, allowing users to explore business objects, their relationships, and functionality.

BOA is available for use with DCOM and is being extended to support technologies such as CORBA, Microsoft Transaction Server, and IBM MQSeries. While BOA includes a foundation database model, business objects can be connected to existing production databases.

Brickhouse Data Systems, Inc.
Price: From US$50,000; component pricing and additional project services are available.
Phone: (732) 764-4100
Web Site: http://www.brickhouse.com

Kinetic Announces CrackerJax

Kinetic Software Development Inc. announced the CrackerJax for Delphi suite. CrackerJax is a Delphi IDE extension that provides developers with a method of altering the layout of their Delphi source code.

CrackerJax features include over 200 formatting options; the ability to store and use multiple option configurations; the ability to hook directly into the Delphi IDE; fast execution; support for all 32-bit Delphi installations; and a built-in interface that allows the developer to make enhancement requests or report problems encountered via e-mail.

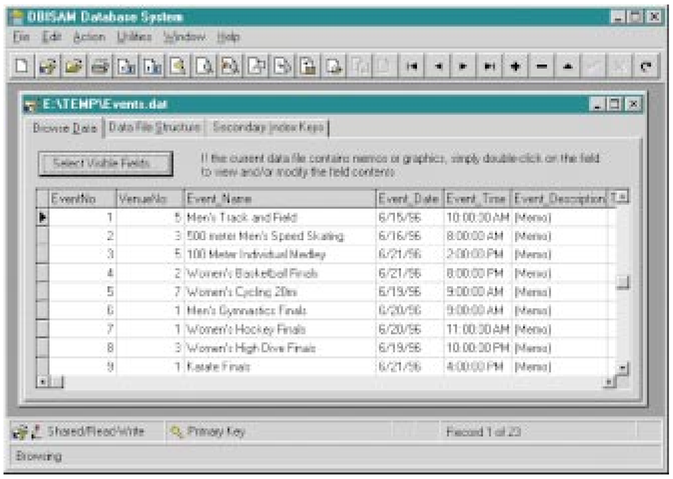
For more information, call (888) 893-7100 or (423) 899-8980, or visit the Kinetic Web site at http://www.kineticsoftware.com.

Elevate Software Announces DBISAM Database System for Delphi

Elevate Software announced the first release of *DBISAM Database System* for Delphi (all versions), a proprietary database system designed to merge the best features of local database formats available for Delphi. DBISAM is targeted at Delphi developers writing applications for single-user and multi-user use with heavy distribution requirements (such as shareware or downloadable software), or for small, in-house installations on a LAN.

DBISAM offers transparent single-user and multi-user usage; built-in repair facilities; a utility for transferring data from Paradox, dBASE, and FoxPro formats; a utility for browsing, restructuring, updating, and searching data files; complete BLOB support, including configurable block sizes; transactions; primary and secondary indexes; complete filter support; in-memory data files with support for streaming; partial index key searches and ranges; ranges with accurate record counts; and numerous other features.

DBISAM compiles into an application's EXE file and has a footprint of around 200KB.



If the current data file contains memos or graphics, simply double-click on the field to view and/or modify the field contents

EventNo	VenueNo	Event_Name	Event_Date	Event_Time	Event_Description
1	5	Men's Track and Field	6/15/96	10:00:00 AM	(Memo)
2	3	500 meter Men's Speed Skating	6/16/96	8:00:00 AM	(Memo)
3	5	100 Meter Individual Medley	6/21/96	2:00:00 PM	(Memo)
4	2	Women's Basketball Finals	6/21/96	8:00:00 PM	(Memo)
5	7	Women's Cycling 20m	6/19/96	9:00:00 AM	(Memo)
6	1	Men's Gymnastics Finals	6/20/96	9:00:00 AM	(Memo)
7	1	Women's Hockey Finals	6/20/96	11:00:00 AM	(Memo)
8	3	Women's High Dive Finals	6/19/96	10:00:00 PM	(Memo)
9	1	Karate Finals	6/21/96	4:00:00 PM	(Memo)

Elevate Software
Price: US$199
Phone: (800) 699-6395
Web Site: http://www.elevatesoft.com

News LINE

December 1998

Raize Software Acquires VisualPROS

Raize Software Solutions, Inc. announced that it has purchased all rights to the VisualPROS component set from Shoreline Software for an undisclosed amount. The set includes 10 native VCL controls that make it easier for developers using Borland Delphi to create professional applications. Raize will integrate the components into version 2.0 of its Raize Components product.

For more information about Raize Components, visit the Raize Software Solutions Web site at http://www.raize.com.

For more information on Shoreline Software, call (800) 261-9198 or (860) 870-5707, or visit http://www.shoresoft.com.

Web Broker Available for Delphi 4

Scotts Valley, CA — Web Broker technology, which enables Delphi 4 developers to deliver high-speed database applications over the Web, is available for customers of select Delphi 4 products.

With Delphi 4 Web Broker, developers can create high-speed, high-throughput, Web-delivered, HTML-based data applications. WebServer applications are Web server extensions that bind directly to ISAPI and NSAPI, Web server interfaces from Microsoft and Netscape, respectively.

WebBridge allows developers to program to a common API for NSAPI and ISAPI. This flexibility protects a developer's code base as the competing Internet standards evolve, allowing the developer to concentrate on implementing business solutions regardless of the back-end Web server.

WebModules centralize the dispatching of Web client requests, the responses for the request, and the creation of HTML content.

WebModules visually control multiple requests coming into a Web sub-site, resulting in full client/server functionality over the Web.

WebDispatcher simplifies the task of handling Web server application event handling. WebDispatcher works with existing query and table components to produce CGI and HTML applications. Four additional components produce Web content for queries, tables, CGI applications, and simple HTML.

The Web Broker technology is included in Delphi 4 Client/Server Suite. Delphi 4 Professional customers can purchase Web Broker for US$199.95. Web Broker does not work with Delphi 4 Standard.

Philips Uses VisiBroker to Make MIRACLE

San Francisco, CA — Philips Medical Systems, Care Flow Net, Inc., and Baptist Health developed MIRACLE (Medical Information Retrieval Application for Clinical Enhancement) for the Baptist Health Systems of South Florida, a not-for-profit healthcare organization composed of five hospitals, multiple outpatient clinics, and physicians' offices. Designed to streamline patient services, the MIRACLE is being built around Inprise Corp.'s VisiBroker CORBA object request broker technology. MIRACLE provides caregivers tightly controlled access to medical data anywhere at any time.

The main challenge of the Baptist Health Systems project was integrating the wide array of data sources, data types, and complex workflow patterns inherent in the healthcare industry, without requiring a single entity to have the broad domain knowledge needed to manage this variety of clinical data from all sources.

MIRACLE allows healthcare workers to store and access relevant information quickly. In addition, hospital IT departments save time and resources by not having to constantly correct the messaging and replication errors common with previous, non-CORBA based solutions.

Inprise Announces Additional Stock Buy-Back Program

Scotts Valley, CA — Inprise Corp. announced that its board of directors has authorized an additional stock buy-back program. The board's resolution authorizes Inprise to repurchase up to 10 percent of the company's outstanding shares of common stock on a fully diluted basis, or approximately 5,900,000 shares, inclusive of the 1,000,000 share repurchase completed on August 10, 1998.

Inprise Announces DCE-CORBA Bridge

New York, NY — Inprise Corp. announced the Inprise DCE-CORBA Bridge, a secure integration solution that allows corporations to Web-enable their existing DCE-based applications for electronic commerce.

The Inprise DCE-CORBA Bridge provides a smooth interface between CORBA clients and DCE servers. It leverages the distributed, open-architecture, and standard protocols of the Internet to build multi-tier, client/server intranet and extranet applications that can operate in heterogeneous infrastructures, using multiple hardware platforms and languages.

Global organizations that have based business-critical enterprise applications on Inprise's DCE and CORBA solutions include Bank of America, Barclaycard, Charles Schwab & Co., Credit Suisse, Daiwa Securities Ltd., First National Bank of Chicago, First Union Bank, Merrill Lynch, NationsBank, Sanford Bernstein & Co., State Street Bank, and T. Rowe Price.

table of CONTENTS

MTS / Delphi 4

By Paul M. Fairhurst

MTS Development

Part I: Unpacking Microsoft Transaction Server

The Client/Server edition of Delphi 4 has brought Delphi firmly into the distributed computing arena. We've had no shortage of tools, techniques, and models over the decades for developing applications: thin client, fat client, event-driven, object-oriented, and now distributed components. All strove for the "Holy Grail" of computing — 100 percent code reuse, 100 percent fault tolerance, an easy application upgrade path when things change, and maximum use of precious resources. Whether the distributed computing model will succeed is anybody's guess. However, we're getting closer to the way complex systems such as the human body function: small, dedicated components working in cooperation to serve a greater purpose. Copying nature can't be too bad a thing, considering how long it's had to get things right!

In this series of articles, I'm going to explore how Delphi empowers you with the ability to create distributed applications using Microsoft Transaction Server (MTS), and how it has again made implementing Microsoft technology easier than with any of Microsoft's development languages.

The Problem

The problem with Information Technology nowadays is its complexity. Our minds can only deal with a limited amount of complexity. To remedy this, we break large problems into small chunks that we can more easily understand. Software development has traditionally broken applications into modules, classes, and DLLs. However, the close implementation dependencies that still exist have left us unable to de-couple parts of an application, making upgrading a difficult task.

The solution to this is the software component. Each component performs a specific task, and access to this functionality is through well-specified interfaces. Because an interface can't change once it's specified, the dependency between a component and its client is reduced. Furthermore, any component can be replaced by another with a different implementation, as long as it supports the same set of interfaces. This allows the developer to replace parts of an application with newer functionality, guaranteeing that all component clients will operate correctly, which would lead to an easier upgrade path for an application.

Once you've de-coupled components from their clients, it doesn't take a genius to take the next step of moving these components onto powerful, fault-tolerant servers that can support hundreds or thousands of client requests. Furthermore, components can be placed near the data sources they'll act upon, reducing network traffic and lead times. Reliability is also increased because more than one server can support a component. If one server fails, another takes over, and the client application isn't any worse off.

In short, the distributed model is better for clients, servers, and developers. The days of monolithic applications holding on to precious database connections and downloading large amounts of data over company networks are numbered.

The Solution

So where does Delphi fit into this model? On the PC, you have two main players in the software component and services arena: the Common Object Request Brokerage Architecture (CORBA), and the Component Object Model (COM). CORBA is an open specification for software components developed by the Object Modeling Group (OMG). It's operating-system independent, and offers no "out-of-the box" library code to help you. COM is similarly a specification for software components, but is also an implementation that's currently geared toward the Windows family of operating systems, and thus has operating system support built in. It's a binary specification, which makes it language-independent and capable of operating cross-platform.

Delphi has supported COM since version 3, although it was somewhat limited in its threading support. Delphi 4 has remedied this, and now includes comprehensive support for developing component-based applications with COM, which is the basis for other technologies, such as OLE, ActiveX, and Automation.

Now, COM allows transparent access to COM components running on other machines via Distributed COM (DCOM). By transparent, I mean that changes to the client applications of the COM component aren't necessary when the component is moved from the local machine to another machine on the network.

Distributed computing with DCOM is powerful stuff, but it brings with it a couple of inherent problems. First, Microsoft NT Server can't effectively host tens of COM objects being accessed by hundreds of clients. If each client created a component on the server, which then opened a connection to a database server, the burden on the server would be too great. (Not to mention the issue of database server licenses!) This is a scalability problem, and COM components don't naturally scale.

The second problem has to do with transactions. If you have business components spread over many machines, and you instruct every component to save its work, what happens if one fails? You need all the work to complete, or not at all; in short, you need a transaction. To support this, you would have to construct every component to be transaction-aware, not to mention get involved with distributed transactions across many machines. As a developer, you have more important functionality to implement than this!

Microsoft soon realized these problems and set about creating an infrastructure for COM components so you could write them as if you were writing a component for one client, while not having to worry about transactions. Any resources, such as database connections your components use, would be pooled and recycled for use by other components, freeing them up and instantly providing scalability to COM components running on NT Server. In other words, it would provide the "plumbing" for your components, freeing you from many of the responsibilities of writing distributed applications and moving the complexity to where it belongs — the operating system.

Microsoft succeeded in creating this infrastructure, and it has become so important to NT Server that many other operating system services now require it in order to function. They named it Microsoft Transaction Server (MTS).

The MTS Philosophy

MTS was originally sold as an additional product for NT Server. Microsoft quickly realized its importance in enabling NT Server to scale its way into enterprise markets, so it's now available for free as part of the NT 4.0 Option Pack. This pack ships with NT Server 4.0, and is freely downloadable from Microsoft's Web site. Be warned, though: It's around 87MB in size. MTS can also be hosted by NT Workstation and Windows 95 running DCOM. These versions are only 27MB in size. Alternatively, you can order the NT 4.0 Option Pack CD-ROM from Microsoft.

To allow NT scale to support many simultaneous clients, components running under MTS need to be written to support the MTS way of doing things. This means "surgical strike" components. A component needs to get in, do the job, and get out as quickly as possible. It should hold no resources once it's completed its work. Obviously, there are circumstances when it isn't possible to do this, or when it's actually more effective to hold on to resources between client calls. We will look at such cases in a later article. For now, we'll take a look at how MTS objects operate before moving on to writing our first MTS component.

Anatomy of an MTS Object

MTS controls and manages components by activating when they're needed by clients, and deactivating them when they're not. This is called Just In Time (JIT) activation and As Soon

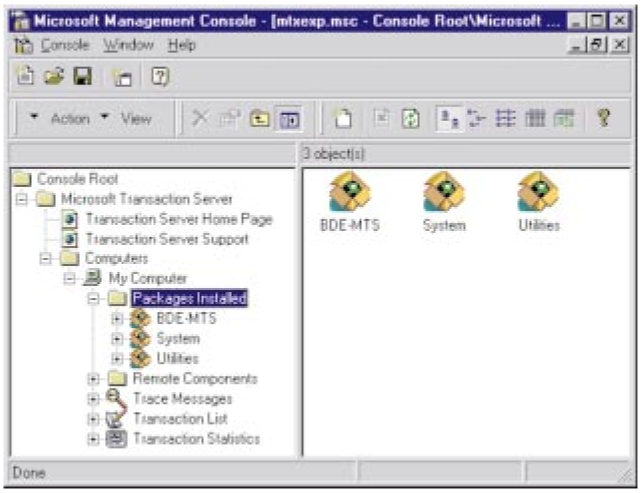


Figure 1: Displaying installed MTS packages in the MTS Explorer (AKA Microsoft Management Console).

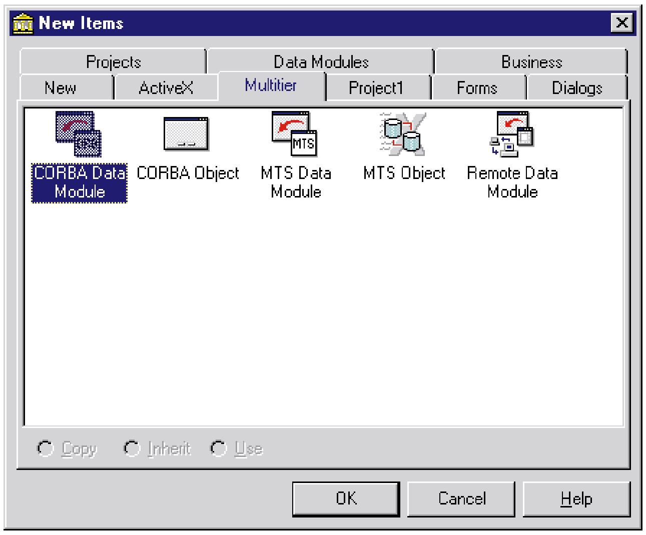


Figure 2: The MTS Object icon appears on the Multitier tab of Delphi's New Items dialog box.

As Possible (ASAP) deactivation.

When a client requests the use of a component, MTS usually won't create an instance of the component. Only when the client actually calls a method (or property) of the component does MTS instantiate it. What the component does when the call is finished determines the scalability of the component and what MTS does with it.

If the component has finished its work, or if it can't complete its work, it notifies MTS, which then deactivates it. Deactivation isn't the same as destruction. Deactivation means the component loses all information held about the client. On the next call into the component by any client, it's completely initialized. MTS may keep the component in a deactivated state, or destroy it.

Unless you specifically request otherwise, the component will be deactivated by MTS when a call into it has finished. When the client (or a new client) requests its services again, it's quickly reactivated, which is much quicker than creating it from scratch. When a component is reactivated, it knows nothing about the last client to access it, and doesn't maintain internal state between multiple interactions with a client. Such a component is said to be "stateless." A stateless component is a "surgical strike" component and scales better than a stateful component.

A "stateful" component holds internal state between interactions, and MTS can't deactivate it when a method call on the component ends until you specifically say it can. The component will remain active in server memory and hold on to potentially valuable resources (e.g. database connections) between method calls. This is obviously less efficient, but may be desirable if, for instance, the component is holding a connection to a resource that takes a long time to connect with.

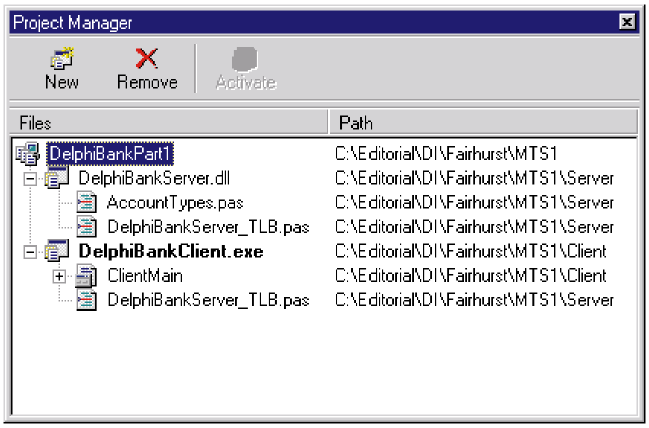
Another important point is that components must be in-process DLLs. You don't need to worry about this, though; Delphi's wizards take care of it for you. It's important, however, if you're considering moving existing COM component functionality to MTS. That's enough theory for now. Let's get on with some practice.

MTS Installation

I'll assume you've installed MTS on a machine. To keep things simple for now, you need to have Delphi 4 installed on the same machine. This means the clients we create will call the MTS components locally, rather than over a DCOM network connection, which would add another level of complexity that we don't want right now. It also means we can use Delphi's friendly automatic installation of your components into MTS, which saves us some grunt work. Ideally, you should be running Windows NT because it has a more sophisticated security capability (which we'll use in a later article). Delphi should be installed after MTS has been installed. If this isn't the case, then you need to take some additional steps, as detailed in section 9 of Delphi 4's Readme.txt file.

Fire up the MTS Explorer, and open the \My Computer\ Packages Installed folder to see all the packages installed in MTS (see Figure 1). A package is a set of components that performs related functions. All components in a package run in the same MTS server process, which isolates faults to the package level. Security credentials are checked when a client calls into a component in a package, but not when a component calls another component in the same package. This means that components in a package "trust" each other, and this is said to be a "trust boundary."

Depending on the options you selected during the MTS installation, you may see Microsoft's "Sample Bank" and "Tic-Tac-Toe" packages in the Explorer.



Files	Path
DelphiBankPart1	C:\Editorial\DI\Fairhurst\MTS1
DelphiBankServer.dll	C:\Editorial\DI\Fairhurst\MTS1\Server
AccountTypes.pas	C:\Editorial\DI\Fairhurst\MTS1\Server
DelphiBankServer_TLB.pas	C:\Editorial\DI\Fairhurst\MTS1\Server
DelphiBankClient.exe	C:\Editorial\DI\Fairhurst\MTS1\Client
ClientMain	C:\Editorial\DI\Fairhurst\MTS1\Client
DelphiBankServer_TLB.pas	C:\Editorial\DI\Fairhurst\MTS1\Server

Figure 3: The Delphi 4 Project Manager displaying the demonstration projects for this article.

The MTS *Programmer's Guide* makes use of these components in its examples, and you may find them handy to test your installation of MTS with the sample client applications that ship with MTS. You'll also see the "System" package, which is the core of MTS. We'll visit this in a later article when we talk about security.

The Component

When you select **File | New** in Delphi, you're presented with a multi-tabbed dialog box that gives access to application wizards. To develop an MTS component in Delphi, you simply create a new ActiveX Library (from the ActiveX tab), then add new MTS components with the MTS Object icon on the Multitier tab (see Figure 2). Each component will reside in the same DLL, but they can be added to MTS independently if you wish.

The example projects in this article show the basics of MTS component development. Throughout this series of articles, we'll build components and client applications with a common theme. I've chosen the banking theme. I know it isn't original, but it serves to illustrate many of the concepts you need to know. Besides, almost everyone has a bank account, and can relate to money quite easily. I could use widgets and gadgets, but I prefer to keep things simple and intuitive.

We'll create a fictitious company called "DelphiBank" and develop components and a client application that accesses and creates information about accounts, account types, customers, and customer transactions. We will implement security on our components to prevent unauthorized access to the bank's records, and use transactions to ensure integrity in our database.

There are two projects accompanying this article (see Figure 3). DelphiBankServer contains the MTS server DLL (DelphiBank), which currently contains only one component: AccountTypes. The other project, DelphiBankClient, is the client that will request services from our component and display the results on a form. The AccountTypes component performs operations on the various types of accounts that "DelphiBank" operates. It's rudimentary, with no database access, but it demonstrates some important concepts. (These projects are available for download; see end of article for details.)

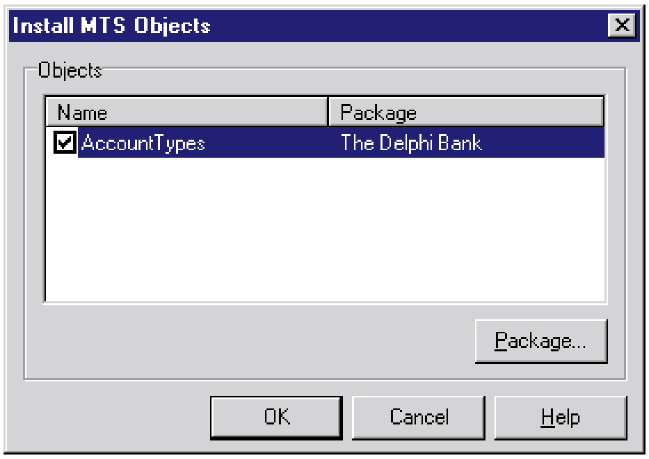
```
IAccountTypes = interface(IDispatch)
  ['{ 95752307-14E5-11D2-A49F-00A0C929E2FF }']
  function GetAccountType(AcctTypeID: Integer;
    var AccountName: WideString; var InterestRate: Single;
    var DefMinBalance: Single; var ResultStr: WideString):
    Integer; safecall;
  function AddAccountType(var AcctTypeID: Integer;
    const AccountName: WideString; InterestRate: Single;
    DefMinBalance: Single; var ResultStr: WideString):
    Integer; safecall;
  end;
```

Figure 4: The *IAccountTypes* interface, which is implemented by the AccountTypes component.

```
function TAccountTypes.GetAccountType(AcctTypeID: Integer;
  var AccountName: WideString;
  var InterestRate, DefMinBalance: Single;
  var ResultStr: WideString): Integer;
const
  cMethodName: string = 'AccountTypes.GetAccountType';
begin
  Result := cATSuccess;

  try
    case AcctTypeID of
      1 :
        begin { Current Account Type. }
          AccountName   := 'Current Account';
          InterestRate  := 0.1;
          DefMinBalance := -100;
        end;
      2 :
        begin { Gold Account Type. }
          AccountName   := 'Gold Account';
          InterestRate  := 1.2;
          DefMinBalance := 1000;
        end;
      3 :
        begin { Savings Account Type. }
          AccountName   := 'Savings Account';
          InterestRate  := 4.9;
          DefMinBalance := 500;
        end;
      4 :
        begin { Savings Plus Account Type. }
          AccountName   := 'Savings Plus Account';
          InterestRate  := 5.9;
          DefMinBalance := 5000;
        end;
      else
        raise Exception.Create('Unknown account type.');
    end;

    ResultStr := cMethodName + ' - successful';
    SetComplete; { We're done. }
  except
    on E: Exception do begin
      { Ooops. }
      Result  := cATFailure;
      ResultStr :=
        cMethodName + ' exception - ' + E.Message;
      SetAbort;  { Cannot continue. }
    end;
  end;
end;
```

Figure 5: The *GetAccountType* function.

Figure 4 shows the *IAccountTypes* interface, which is implemented by the AccountTypes component. This was generated



Name	Package
☑ AccountTypes	The Delphi Bank

Figure 6: Installing objects into MTS from Delphi.

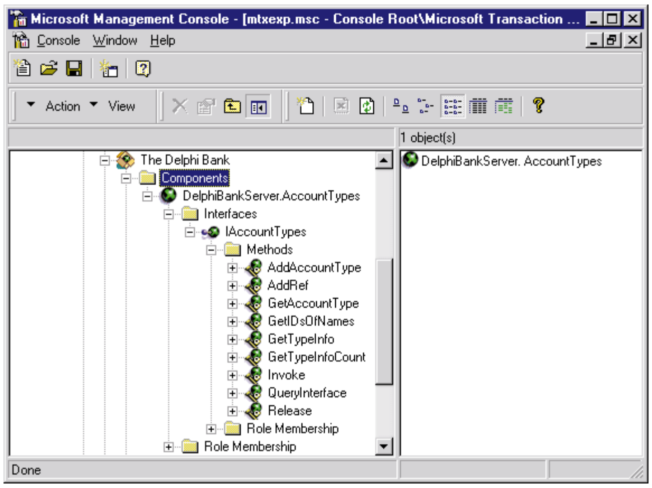


Figure 7: The MTS Explorer with the DelphiBankServer package installed.

by Delphi's Type Library Editor in DelphiBankServer_TLB. You can see that we have two functions that read and add an account type. Currently, we'll only implement reading because we have no database in which to store information, and we don't want to make our component stateful. Both functions return an integer to indicate success or failure (zero for success), and a result string we can use to provide an English description of what occurred (useful for debugging purposes).

Figure 5 shows the *GetAccountType* function found in AccountTypes. Normally, the account information would be looked up in a database, but here, we hard code it. The most interesting thing about this piece of code are the calls to *SetComplete* and *SetAbort*. These functions inform MTS that the component's work has succeeded or failed. *SetComplete* is called when the component has successfully finished its work to let MTS know its work can be committed. On the other hand, *SetAbort* informs MTS that its work can't be committed, and that the transaction in progress — if any — can't succeed. Either way, MTS will deactivate the component upon exiting this method.

Where have these functions come from? Every object hosted by MTS has an associated "object context." The object context provides information such as whether the object is executing within a transaction and, if so, the identity of the transaction.

I'll discuss the *object context* more thoroughly next month when I discuss transactions. I will mention, though, that you can access it with *GetObjectContext* from Delphi's Mtx module, which returns an *IObjectContext* interface. This contains (among other things) *SetComplete* and *SetAbort*. For now, let's install the component into MTS.

Installing the Component

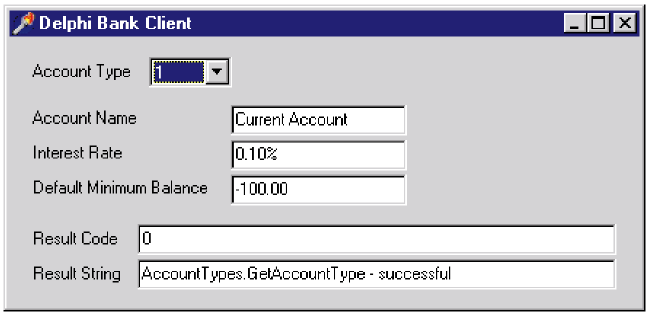
Open the DelphiBankServer project in Delphi and build it; open the project group, then select DelphiBankServer in the project manager and click the Activate speed button to make this the current project; then build it. Now select **Install MTS Objects** from the **Run** menu (as I mentioned earlier, Delphi must be installed on the same machine as MTS for this to work). Delphi presents a list of components to install into MTS. We only have one (AccountTypes), and because we haven't installed it before, Delphi doesn't know which package to install it into (all components must reside in a package).

Click the check box next to AccountTypes and a dialog box will appear. You need to select the Into New Package page and type in a name for the package. We'll call it `The Delphi Bank`. When you click **OK**, you should see the screen shown in Figure 6. Click **OK** again, and Delphi will do its magic and install your component into MTS via OLE Automation.

If you fire up the MTS Explorer (refreshing the "Packages Installed" if necessary by right-clicking and selecting **Refresh**), you should see The Delphi Bank package. If you open it and open the **Components** directory, you should see the DelphiBankServer.AccountTypes component (see Figure 7). Have a look at the properties of the package and component by right-clicking on them and selecting **Properties**. In particular, go to the Advanced tab of the Properties page for The Delphi Bank package. You'll see the **Server Shutdown Process** time is set to **3** minutes. While developing components, this is best set to `0`. If you don't do this, you'll have problems building the component in Delphi because MTS will keep the DLL locked for three minutes every time a client calls it before eventually releasing the server DLL. Delphi can't compile a new DLL while it's locked by MTS.

The Client

I want to make an important distinction about clients in MTS. In MTS, a client is an MTS component using another MTS component. An application running outside of the MTS environment that calls MTS components is called a *base client*. The second project accompanying this article (DelphiBankClient) is a base client — a Delphi application that imports the type library for our component. Build it in Delphi, and run it on the machine where MTS is installed. If you opened the project group, select DelphiBankClient in the project manager, click the Activate speed button to make this the current project, then run it.



Account Type: 1
Account Name: Current Account
Interest Rate: 0.10%
Default Minimum Balance: -100.00
Result Code: 0
Result String: AccountTypes.GetAccountType - successful

Figure 8: The client application in action.

While it's running, you can select an account type with the combo box displayed. When you do, a call will be placed to the AccountTypes component, which will then run inside MTS. You should now see something like Figure 8. You can select the four supported account types and one invalid one. The first time you select an account type, there will be a delay while MTS creates the component and the result comes back to the client.

```
procedure TFrmDelphiBankClient.CbAccountTypeChange(
  Sender: TObject);
const
  cUnknownStr : string = '?';
var
  AccountName   : WideString;
  InterestRate  : Single;
  DefMinBalance : Single;
  ResultStr     : WideString;
  ResultCode    : Integer;
begin
  { Query the AccountTypes component about this account. }
  if not Assigned(AnAccountTypes) then
    { Class function from DelphiBankServer_TLB. }
    AnAccountTypes := CoAccountTypes.Create;

  { If we got here then the call succeeded. }
  try
    ResultCode := AnAccountTypes.GetAccountType(
      CbAccountType.ItemIndex+1, AccountName,
      InterestRate, DefMinBalance, ResultStr);

    if ResultCode = 0 then
      begin
        { ...Display results... }
      end
    else
      raise Exception.Create(ResultStr);  { Failure. }
  except
    on E: Exception do begin
      { ... }
    end;
  end;
end;

procedure TFrmDelphiBankClient.FormDestroy(
  Sender: TObject);
begin
  { Release the AccountTypes component. Normally this would
    be done when work has finished on the component, not
    here. However, I placed it here so you could see the
    component being used in MTS and released when the
    client closes. }
  AnAccountTypes := nil;
end;
```

Figure 9: Code from the base client that responds to the combo box of account types being changed.

Open the MTS Explorer and select the \Components folder under The Delphi Bank package. Now select **View | Status View** from the toolbar. Make it so you can see the window while you're selecting an account type. You'll notice the DelphiBankServer.AccountTypes object is listed and that under the **Objects** column is a 1, signifying that one instance of this object is being accessed by a base client. (If you run more than one copy of the base client application, you'll see this number rise accordingly.) Now select the \Transaction Statistics folder, and watch how transactions are being committed and aborted with the calls to *SetComplete* and *SetAbort*.

Figure 9 shows a snippet of code from the base client that responds to the combo box of account types being changed. First, if the object hasn't been created already, it's done so with the *CoAccountTypes.Create* class function that Delphi creates in the type library module. If this succeeds, we get an *IAccountTypes* interface back on the object with which we can make calls. We make a call to *GetAccountType*, check the result code, and display the results accordingly. Presto! We have a normal Delphi application using the services of an MTS component.

One final point before I wrap up this article: Try commenting out the call to *SetComplete* and looking at the "Status View" when you run the base client. You should see a difference. The **Activated** column will show 1, meaning that the object hasn't been deactivated and has remained active. This would happen if you had a stateful component as well. Only when you close the base client and the object is released in *TFrmDelphiBankClient.FormDestroy* will the object deactivate and be released by MTS.

Conclusion

Make no mistake about it, MTS is an important piece of technology for Windows developers, and Delphi has great support for it. I've covered a lot of theory in this article, which I believe is necessary for you to understand the mechanics of developing for MTS.

Next month, we'll look at database connectivity in components. We'll see how the new BDE supports MTS transactions that allow us to roll back database changes, and see how it supports the pooling of database connections for improved scalability. Δ

The projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\DEC\DI9812PF.

Paul M. Fairhurst is a First Class Computer Science graduate of Sheffield University, England and freelance consultant/developer specializing in client/server and multi-tier database development. He is currently developing information systems for BBC Television and Radio in London. You can contact him at paul@c-s-c.demon.co.uk.

Delphi 3, 4 / Plug-ins

By Marc Evans

Delphi Plug-Ins

Creating, Debugging, and Using Application Extensions

Have you ever used Adobe Photoshop? If so, you're familiar with the concept of plug-ins. For the uninitiated, a plug-in is simply a piece of code that is supplied externally to the application (for instance, in a DLL). The distinction between a plug-in and a normal DLL is the ability of the plug-in to extend the capabilities of the parent application. For example, Photoshop is, by itself, not capable of a great deal of image manipulation. The addition of plug-ins gives it the capability to produce blurs, smears, and all manner of strange effects, none of which are present in the parent application.

This is fine for a graphics program, but why go through the effort of producing a business application that supports plug-ins? Suppose, for example, that your application produces some reports. You know the customer is going to keep asking for updates or new reports to be added. You could use an external report generator such as ReportSmith — an inelegant solution, requiring extra files to be distributed and extra training for the users. You could also use QuickReport, but this leads to a version control nightmare if you have to rebuild the application each time a font changes.

You *could* use it, however, as long as you build the report into a plug-in. Want a new report? No problem; simply install a DLL and the application will see it the next time it's started. Another example could be an application that processes data from an external device, such as a bar code scanner. You want to give the end users some choice, so you need to support half a dozen devices. By simply writing each device interface routine as a plug-in, you have ultimate flexibility without changing your parent application.

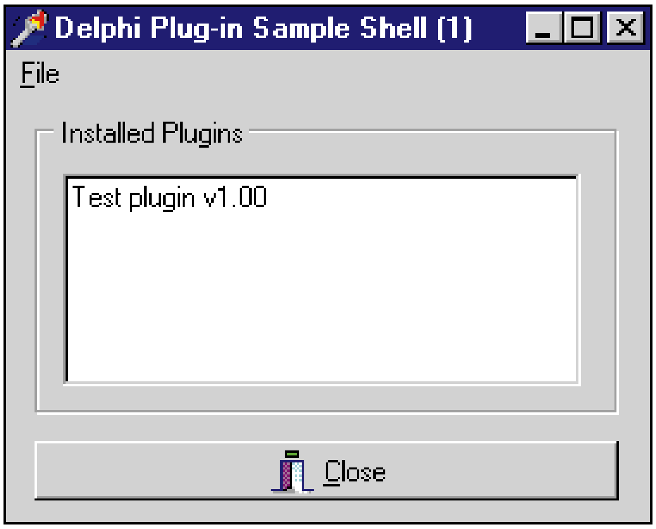


Figure 1: The plug-in test shell.

Getting Started

It's important to know what type of functionality your application will need to expand upon before writing any code. This is because the plug-ins interact with the parent application using a specific interface, which you will define according to your needs. In this article, we will build three plug-ins that illustrate some of the ways a plug-in can interact with a parent application.

We'll construct our plug-ins as DLLs. Before we do this, however, we must build a shell application to load and test them. Figure 1 shows our test application with the first plug-in loaded. This first plug-in doesn't do very much. In fact, all it does is return a string describing itself.

```
{ Iterate the application directory looking
for plug-in files. }
procedure TfrmMain.LoadPlugins;
var
  sr:    TSearchRec;
  path:  string;
  Found: Integer;
begin
  path := ExtractFilePath(Application.Exename);
  try
    Found := FindFirst(path + cPLUGIN_MASK, 0, sr);
    while Found = 0 do begin
      LoadPlugin(sr);
      Found := FindNext(sr);
    end;
  finally
    FindClose(sr);
  end;
end;
```

Figure 2: Finding plug-ins.

Nevertheless, it illustrates an important point — the application will run whether the plug-in is present or not. If the plug-in is not there, it will not be reported in the list of installed plug-ins, but the application will continue to function normally.

The only things that differentiate our plug-in shell from a normal application are the addition of the Sharemem unit to the **uses** clause of the project source, and the code to load the plug-in files. The Sharemem unit is required for any application that passes string parameters between itself and a child DLL. It's an interface to Delphimm.dll (supplied with Delphi). To test this shell, the Delphimm.dll file will have to be on the path, or copied into the application directory from the \Delphi 3\Bin directory. This DLL will also have to be distributed with the final application.

The plug-ins are loaded into this test shell via the *LoadPlugins* procedure, called in the *FormCreate* event of the main window and displayed in Figure 2. This procedure iterates the application directory using the *FindFirst* and *FindNext* functions, looking for plug-in files. When a file is found, it's loaded using the *LoadPlugin* procedure listed in Figure 3.

The *LoadPlugin* procedure demonstrates the heart of the plug-in mechanism. First, the plug-in is written as a DLL. Then, it is dynamically loaded using the *LoadLibrary* API function. Once the DLL has been loaded, we need a way of accessing the procedures and functions it contains. The API call, *GetProcAddress*, provides this mechanism, returning a pointer to the requested routine. In this simple demonstration, the only routine contained in the plug-in is a procedure named *DescribePlugin*, specified in the constant cPLUGIN_DESCRIBE. (The case of the procedure name is important; the name passed into *GetProcAddress* must exactly match the name of the routine in the DLL.) If the requested routine was not found in the DLL, *GetProcAddress* returns **nil**, allowing the return value to be tested using the assigned function.

```
{ Load the specified plug-in DLL. }
procedure TfrmMain.LoadPlugin(sr: TSearchRec);
var
  Description:  string;
  LibHandle:    Integer;
  DescribeProc: TPluginDescribe;
begin
  LibHandle := LoadLibrary(Pchar(sr.Name));
  if LibHandle <> 0 then
  begin
    DescribeProc := GetProcAddress(LibHandle,
                                   cPLUGIN_DESCRIBE);
    if Assigned(DescribeProc) then
      begin
        DescribeProc(Description);
        memPlugins.Lines.Add(Description);
      end
    else
      begin
        MessageDlg('File "' + sr.Name +
        '" is not a valid plug-in.',
          mtInformation, [mbOK], 0);
      end;
  end
  else
  begin
   MessageDlg('An error occurred loading the plug-in "' +
     sr.Name + '".', mtError, [mbOK], 0);
end;
```

Figure 3: Loading plug-ins.

To store a pointer to a function in a useful manner, it's necessary to create a special type for the variable that's used. Notice that the return value of *GetProcAddress* is stored in a variable, *DescribeProc*, which is of type *TPluginDescribe*. *TPluginDescribe* is defined as:

```
type
  TPluginDescribe = procedure(var Desc: string); stdcall;
```

The **stdcall** directive is used because the procedure resides in a DLL, which was built using the standard calling convention for all the exported routines. This procedure takes a single **var** parameter, which will contain a description of the plug-in when the procedure returns.

To call the procedure we've just found, simply use the name of the variable holding the address as the procedure name, followed by any parameters. Using our example, the statement:

```
DescribeProc(Description)
```

will call the descriptive procedure found in the plug-in and fill the *Description* variable with a string describing the plug-in functionality.

Building the Plug-in

Now that we've created a parent application, it's time to build the plug-in we want to load. The plug-in files will be standard Delphi DLLs, so start a new DLL project from the Delphi IDE and save it. Because the exported plug-in function will use a string parameter, place the Sharemem unit as the first unit in the project's **uses** clause. The listing in Figure 4 shows the project source for our sample plug-in.

It's worth noting that although the plug-in is a .DLL file, it doesn't need the .DLL extension. In fact, there's a good reason for changing it: When the parent application looks for files to load, it can use a specific file mask. If the extension is left as .DLL, the parent will have to try to load every .DLL in the project directory, looking for valid plug-ins. By changing the extension to something else (our samples use *.plg), you can be reasonably sure the application is only going to be loading relevant files. The compiler directive *$E* will accomplish this change, or the extension can be set from the Application page on the Project Options dialog box.

The code for the first sample plug-in is simple. Figure 5 shows the code contained in a new unit. Note that the *DescribePlugin* prototype is identical to that of the type *TPluginDescribe* in the shell application, with the addition of the **export** keyword to specify that the procedure will be exported. The name of the exported procedure also appears in the **exports** section of the main project source (shown in Figure 4).

Before this plug-in can be tested, it has to be copied into the main application directory. The easiest way of doing this is to create all the plug-ins in subdirectories of the main directory, and specify ..\ as the output path for the project. (The Directories/Conditionals page of the Project Options dialog box will allow this to be changed.)

Debugging

This is a good time to introduce one of the nicer features of Delphi 3: The ability to debug DLLs from within the IDE. An application is specified as the **Host application** in the Run Parameters dialog box in the DLL project. This is the path to the application that will call the DLL. (In our case, this is the path to the test shell we've just created). Then you can set breakpoints in the DLL code and press F9 to run as you would with a regular application. Delphi will execute the specified host application and — assuming the DLL is compiled with debug information — drop you into the debugger at your breakpoint inside the DLL code.

Extending the Parent

This simple plug-in is fine, but it doesn't do anything useful. The second example will rectify this. The aim of this plug-in is to add an item to the main menu of the parent application. This menu item, when clicked, will execute some code inside the plug-in. Figure 6 shows a revised version of the shell application with both plug-ins loaded. In this version of the shell, a new menu item, named **Plug-in**, has been added. The plug-in will attach a menu item to this at run time.

To accomplish this, we must first define a second interface into the plug-in DLL. The existing DLL only exports one procedure, *DescribePlugin*. The second plug-in will introduce a procedure named *InitPlugin*. Before the procedure can be seen by the main application, however, the *LoadPlugin* procedure must be modified to take account of it.

The code in Figure 7 illustrates the revised procedure. As you can see, after the first *GetProcAddress* call to find the description procedure, another call to *GetProcAddress* has been added.

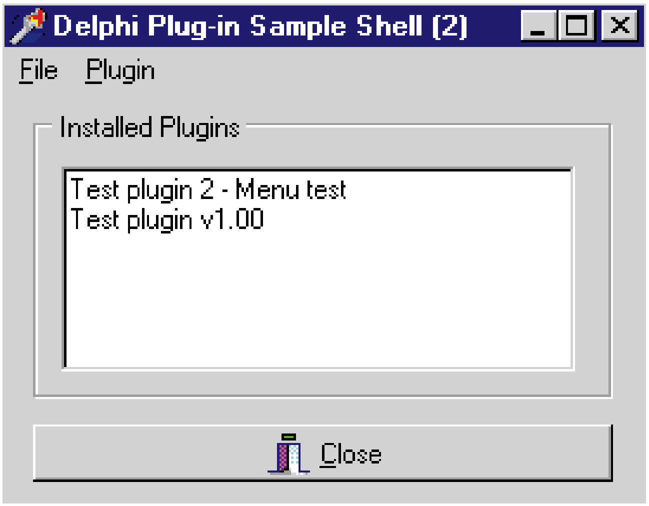
This time, we're looking for the constant cPLUGIN_INIT, which is defined as:

```
uses
  Sharemem, SysUtils, Classes,
  main in 'main.pas';

{$E plg.}

exports
  DescribePlugin;

begin

end.
```

Figure 4: The sample plug-in project source.

```
unit main;

interface

  procedure DescribePlugin(var Desc: string);
    export; stdcall;

implementation

procedure DescribePlugin(var Desc: string);
begin
  Desc := 'Test plugin v1.00';
end;

end.
```

Figure 5: The main unit for the sample plug-in.



Test plugin 2 - Menu test
Test plugin v1.00

Figure 6: A revised version of the shell application with both plug-ins loaded.

```
const
  cPLUGIN_INIT = 'InitPlugin';
```

The return value is stored in a variable of type *TPluginInit*, which is defined as:

```
type
  TPluginInit = procedure(ParentMenu: TMainMenu); stdcall;
```

When the *InitPlugin* procedure is executed, the main menu of the parent application is passed into it as a parameter. The procedure can then modify this menu as it pleases. Because all the return values of *GetProcAddress* are tested with *assigned*, this new version of the *LoadPlugin* procedure will still load the first plug-in, which did not contain the *InitPlugin* procedure. All that will happen is that the first call to find the *DescribePlugin* procedure will pass, and the second call to find *InitPlugin* will silently fail.

Now that the new interface has been defined, it's possible to write the code for the new *InitPlugin* procedure. As before, the bulk of the code for the new plug-in resides in a separate unit. Figure 8 shows the modified main.pas containing the *InitPlugin* procedure. The first change from the original plug-in is obviously the addition of the *InitPlugin* procedure. As before, the prototype is added to the top of the unit with the **export** keyword, and the procedure name is added to the **exports** clause of the project source. This procedure creates a new menu item using the *NewItem* function, which returns a *TMenuItem* object. The new menu item is added to the application menu by the statement:

```
ParentMenu.Items[1].Add(I);
```

Items[1] on the test shell main menu is the item Plug-in, so this statement adds a new menu item to the **Plugin** menu named **Plug-in Test**.

To handle the response to this new menu item, *NewItem* can take, as its fifth parameter, a procedure of type *TNotifyEvent*, which is called when the menu item is clicked. Unfortunately, procedures of this type are, by definition, methods of an object, and we have no objects in our plug-in. If we try to use

```
procedure TfrmMain.LoadPlugin(sr: TSearchRec);
var
  Description:  string;
  LibHandle:    Integer;
  DescribeProc: TPluginDescribe;
  InitProc:     TPluginInit;
begin
  LibHandle := LoadLibrary(Pchar(sr.Name));
  if LibHandle <> 0 then
  begin
    // Find DescribePlugin.
    DescribeProc := GetProcAddress(LibHandle,
                                   cPLUGIN_DESCRIBE);
    if Assigned(DescribeProc) then
    begin
      // Call DescribePlugin.
      DescribeProc(Description);
      memPlugins.Lines.Add(Description);
      // Find InitPlugin.
      InitProc := GetProcAddress(LibHandle, cPLUGIN_INIT);
      if Assigned(InitProc) then
      begin
        // Call InitPlugin.
        InitProc(mnuMain);
      end;
    end
    else
    begin
      MessageDlg('File "' + sr.Name +
        '" is not a valid plugin.',
        mtInformation, [mbOK], 0);
    end;
  end
  else
  begin
    MessageDlg('An error occurred loading the plugin "' +
      sr.Name + '".', mtInformation, [mbOK], 0);
  end;
end;
```

Figure 7: The revised *LoadPlugin* procedure.

```
unit main;

interface

uses Dialogs, Menus;

type
  THolder = class
  public
    procedure ClickHandler(Sender: TObject);
  end;

  procedure DescribePlugin(var Desc: string);
    export; stdcall;
  procedure InitPlugin(ParentMenu: TMainMenu);
    export; stdcall;

var
  Holder: THolder;

implementation

procedure DescribePlugin(var Desc: string);
begin
  Desc := 'Test plugin 2 - Menu test';
end;

procedure InitPlugin(ParentMenu: TMainMenu);
var
  i: TMenuItem;
begin
  // Create new menu item.
  i := NewItem('Plugin &Test', scNone, False, True,
               Holder.ClickHandler, 0, 'mnuTest');
  ParentMenu.Items[1].Add(i);
end;

procedure THolder.ClickHandler;
begin
  ShowMessage('Clicked!');
end;

initialization
  Holder := THolder.Create;

finalization
  Holder.Free;

end.
```

Figure 8: The code for the second plug-in.

a normal pointer to a function, the Delphi compiler will complain, so the only solution is to create an object to hold the menu click handler. This is the purpose of the *THolder* class. It has only one method: a procedure named *ClickHandler*. A global variable, named *Holder*, is defined as type *THolder* in the **var** section of this revised main.pas, and it's created in the initialization phase of the unit. Now that we have an object, we can use its method (*Holder.ClickHandler*) as a parameter to the *NewItem* function.

After all that, the *ClickHandler* procedure does nothing more than display a dialog box with the message "Clicked!" Not very interesting perhaps, but again, it proves a point. The plug-in DLL has successfully modified the main menu of the parent application, proving its new functionality. And, like the first example, the application will execute whether this plug-in is present or not.

Because we created an object to hold our menu click handler, it must be freed when the plug-in is no longer needed. The **finalization** section of the revised unit will take care of this. The **finalization** section is the opposite of the **initialization** section, and it's guaranteed to run when the application terminates if a previous **initialization** section was encountered.

Placing the statement:

```
Holder.Free
```

in the **finalization** section ensures that the *Holder* object will be properly disposed of.

It's easy to see that the although this plug-in simply modifies the main menu of the shell application, it could easily manipulate any other object that was passed into the *InitPlugin* procedure. The plug-in could open its own dialog boxes, add items to list boxes and tree views, or draw to a canvas if needed.

Event-driven Plug-ins

The techniques outlined so far can produce a reasonably versatile method of extending an application. By adding new menus, forms, and dialog boxes, it's possible to write entirely new functions without having to change the parent application in any way. There is a limitation however: It's a one-sided mechanism. The system, as described, relies on the user to initiate the plug-in code by means of a menu click or similar action. Once that code is running, it relies on another user action to stop it, i.e. to close any forms that the plug-in might have opened. One possible way of overcoming this limitation is to make the plug-ins respond to actions in the parent application — in effect, to simulate the event-driven programming model that works so well within Delphi.

With the final sample plug-in, we are going to create a mechanism whereby the plug-ins can respond to events generated within the parent application. In general terms,

```
{ Trap for WM_GETMINMAXINFO. Calls plugin routine
  on every message. }
procedure TfrmMain.MinMaxInfo(var msg: TMessage);
var
  m: PMinMaxInfo;  // Defined in Windows.pas.
  i: Integer;
begin
  m := pointer(msg.Lparam);
  for i := 0 to lstMinMax.count -1 do begin
    TResizeProc(lstMinMax[i])(m.ptMinTrackSize.x,
                              m.ptMinTrackSize.y);
  end;
end;
```

Figure 9: Message handler for WM_GETMINMAXINFO.

this is accomplished by deciding what events to trigger and creating a *TList* object in the parent application for each event. Each *TList* is then passed into the initialization procedure of the plug-ins. If a plug-in is interested in acting on an event, it adds the address of a responsible function to the relevant *TList*. The parent application iterates the lists of function pointers at the appropriate moment, calling each function in turn. In this way, it's possible for multiple plug-ins to act on the same event.

The events generated by the application will depend completely on the intended functionality of the program. For example, a TCP/IP network application might want to notify plug-ins of data arriving via the *TClientSocket* object *OnRead* event, while a graphics application might be more interested in palette changes.

To illustrate the concept of event-driven plug-in responses, we'll build a plug-in that limits the minimum size of the main window. This is a somewhat contrived example, as obviously it's far simpler to build this routine into the application. However, it has the advantage of being simple to code and easy to understand, which is the requirement for this article.

Obviously, the first thing to decide is which events to generate. In this case, the answer is simple: To limit the size of an application window, it's necessary to trap and modify the Windows message WM_GETMINMAXINFO. Therefore, to create a plug-in that does this, we must trap the message and call the plug-in routine inside the message handler. This will be the event we create.

We then need to create a *TList* to handle this event. This is handled in the **initialization** section of the main form, where the object, *lstMinMax*, is created. Next, a message handler is created to trap the Windows message WM_GETMINMAXINFO. The code in Figure 9 shows this message handler.

The *LoadPlugin* procedure of the shell application must be modified again to call the initialization routine. This new initialization function takes our *TList* as a parameter and appends to it the address of a function that modifies the message parameters. Figure 10 shows the final version of the *LoadPlugin* procedure, which performs the initializa-

```
{ Load the specified plugin DLL. }
procedure TfrmMain.LoadPlugin(sr: TSearchRec);
var
  Description:  string;
  LibHandle:    Integer;
  DescribeProc: TPluginDescribe;
  InitProc:     TPluginInit;
  InitEvents:   TInitPluginEvents;
begin
  LibHandle := LoadLibrary(Pchar(sr.Name));
  if LibHandle <> 0 then
  begin
    // Find DescribePlugin.
    DescribeProc := GetProcAddress(LibHandle,
                                   cPLUGIN_DESCRIBE);
    if Assigned(DescribeProc) then
    begin
      // Call DescribePlugin.
      DescribeProc(Description);
      memPlugins.Lines.Add(Description);
      // Find InitPlugin.
      InitProc := GetProcAddress(LibHandle, cPLUGIN_INIT);
      if Assigned(InitProc) then
      begin
        // Call InitPlugin.
        InitProc(mnuMain);
      end;
      // Find InitPluginEvents for the 3rd plugin.
      InitEvents := GetProcAddress(LibHandle,
                                   cPLUGIN_INITEVENTS);
      if Assigned(InitEvents) then
      begin
        // Call InitPlugin.
        InitEvents(lstMinMax);
      end;
    end
    else
    begin
      MessageDlg('File "' + sr.Name +
        '" is not a valid plugin.',
        mtInformation, [mbOK], 0);
    end;
  end
  else
  begin
    MessageDlg('An error occurred loading the plugin "' +
      sr.Name + '".', mtInformation, [mbOK], 0);
  end;
end;
```

Figure 10: The final version of *LoadPlugin*.

tions for all the plug-ins constructed so far.

The final step in this process is to create the plug-in itself. As in the previous examples, the plug-in incorporates a description procedure to identify itself. It also carries an initialization routine, which, in this case, simply accepts a *TList* as a parameter. Finally, it includes a non-exported routine named *AlterMinTrackSize*, which modifies the values passed into it. Figure 11 shows the complete code for the final plug-in.

The *InitPluginEvents* procedure is the initialization routine for this plug-in. It takes a *TList* as a parameter. This *TList* is the list created in the parent application to hold the addresses of relevant functions. The statement:

```
lstResize.Add(@AlterMinTrackSize);
```

```
unit main;

interface

uses Dialogs, Menus, classes;

  procedure DescribePlugin(var Desc: string);
    export; stdcall;
  procedure InitPluginEvents(lstResize: TList);
    export; stdcall;
  procedure AlterMinTrackSize(var x, y: Integer); stdcall;

implementation

procedure DescribePlugin(var Desc: string);
begin
  Desc := 'Test plugin 3 - MinMax';
end;

procedure InitPluginEvents(lstResize: TList);
begin
  lstResize.Add(@AlterMinTrackSize);
end;

procedure AlterMinTrackSize(var x, y: Integer);
begin
  x := 270;
  y := 220;
end;

end.
```

Figure 11: The code for the final plug-in.

adds the address of the *AlterMinTrackSize* function to this list. Note the declaration of this function: It's declared as type **stdcall** to match the other procedures, but there is no **export** directive. As the function is being accessed directly via its address, there is no need to export it from the DLL in the usual way.

So, the sequence of events is as follows:
1) As the application initializes, a *TList* is created.
2) This list is passed to the plug-in initialization procedure, *InitPluginEvents*, on startup.
3) The plug-in procedure adds the address of a procedure to the list.
4) The Windows message WM_GETMINMAXINFO, which is generated each time a window is resized, is trapped by our application.
5) This message is handled by our message handler *TfrmMain.MinMaxInfo*, displayed in Figure 10.
6) The message handler iterates the list and calls the functions it references, passing the current *X* and *Y* minimum window sizes as parameters. Note that the *TList* class just stores pointers; so to do anything useful with the values held, we must cast the pointer into the required type — in this case, *TResizeProc*:

```
TResizeProc = procedure (var x, y: Integer); stdcall;
```

7) The plug-in procedure, *AlterMinTrackSize* (which is pointed to by the list), takes the *X* and *Y* values as **var** parameters and modifies them.
8) Control returns to the message handler in the parent application, which continues with the new values for the minimum window size.

9) The *TList* is freed in the **finalization** section of the main code when the application quits.

Conclusion

When using this architecture, it's probably a good idea to make use of the packages feature that Delphi provides. Under normal circumstances, I'm not a big fan of separate run-time modules, but when you consider that any Delphi DLL containing more than a trivial amount of code will be over 200KB, it starts to make sense.

Hopefully, this article has been of some use, if only to get you thinking about application design and how it could be made more flexible. I know that I could have saved myself some work on modifications if I had used some of these techniques in previous applications.

However, I do not present plug-ins as a universal solution. There are clearly some situations where the added complexity is unjustified or the application simply doesn't lend itself to being broken into extensible units. There are other ways of achieving the same effect. Delphi itself provides an interface for writing modules that integrate into the IDE, which is much more object-oriented (some would say "cleaner"), than the one I have outlined, and I am sure it's possible to imitate this for your own use. It's also possible to load Delphi packages at run time. Explore the possibilities. ∆

[The techniques discussed in this article apply to Delphi 4 as well. In fact, Delphi 4 adds a projects feature that makes the development of this kind of application-plus-DLL arrangement easier to develop.]

The projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\DEC\DI9812ME.

Marc Evans has been developing Bespoke Software using Delphi versions 1, 2, and 3 for the past three years. Marc currently works as an Analyst/Programmer, developing systems using client/server databases and data-capture devices.

By Rod Stephens

As the Crow Flies

Determining the Shortest Path through a Network

At the end of a long, hard day of hunt-and-peck at the keyboard, you probably don't want to waste a minute getting home. If you've worked at the same place for more than a few months, you probably know the absolute quickest path from your office to your home. You know exactly which streets to follow and what turns to take to get home in the minimum possible time.

This article explains how you can find shortest paths more generally through any network. You can use the algorithms described here to find the shortest route from your office to the airport, the best way to route e-mail through a computer network, or the shortest way to run telephone wires through existing conduits.

The Big Picture

The shortest-path problem is intuitively easy to understand. Given a network (such as the street network shown in Figure 1), a start position, and an end position, the goal is to find a path through the network that minimizes the total cost of the path.

A network consists of a collection of nodes connected by links. In a street network, nodes correspond to intersections and links correspond to the street segments that connect intersections. Each link has a cost that represents whatever quantity the program needs to minimize along the path. For a street network, the cost might be the link's length, or the average amount of time it takes to drive over that link. For a telephone network, the cost might be the amount of signal loss caused by the wires in the link.

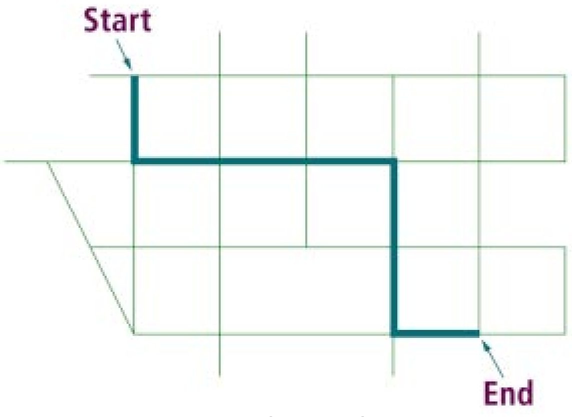


Figure 1: A simple network.

You can represent a network in Delphi using a *TNode* class to represent nodes, and a *TLink* class to represent links. The *TNode* class defines a node's X and Y coordinates. It includes a *TList* object to hold a list of the links leaving the node.

The *TLink* class includes references to the two nodes the link connects and the link's cost:

```
TNode = class(TObject)
  public
    x, y  : Integer;       // The node's location.
    links : TList;         // Links to neighbors.
end;

TLink = class(TObject)
  public
    node1, node2 : TNode;   // The endpoints.
    cost         : Integer; // The link's cost.
end;
```

For example, you could use the code shown in Figure 2 to create two nodes connected by a link with cost equal to the distance between the nodes.

```
var
  start_node, end_node : TNode;
  the_link             : TLink;
  dx, dy               : Integer;
begin
  // Create the start node.
  start_node := TNode.Create;
  start_node.x := 100;
  start_node.y := 150;

  // Create the end node.
  end_node := TNode.Create;
  end_node.x := 200;
  end_node.y := 300;

  // Create the link.
  the_link := TLink.Create;
  the_link.node1 := start_node;
  the_link.node2 := end_node;

  // Set the link's cost.
  dx := start_node.x - end_node.x;
  dy := start_node.y - end_node.y;
  the_link.cost := Round(Sqrt(dx * dx + dy * dy));

  // Add the link to the nodes' lists of links.
  start_node.links.Add(the_link);
  end_node.links.Add(the_link);
```

Figure 2: Creating two nodes connected by a link.

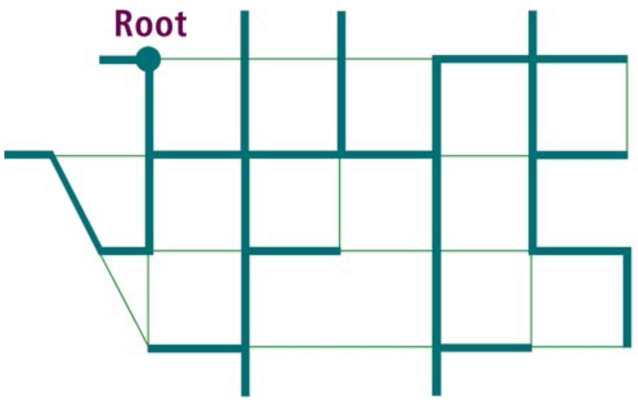


Figure 3: A shortest-path tree.

By calculating shortest paths from a particular node to every other node in the network, you can create a shortest-path tree, such as the one shown in Figure 3. The bold links show the shortest paths from any node in the network back to the tree's root node.

The two algorithms (described later) both build shortest-path trees rooted at a start node. These algorithms are quite fast. In fact, it's usually faster to build an entire shortest-path tree than it is to try to find a single shortest path between two specific nodes.

Most shortest-path algorithms fall into two categories: *label setting* and *label correcting*. Both build a shortest-path tree one branch at a time. The difference between the two lies in how the algorithms select the next link to add to the tree. Label-setting algorithms always add a link that belongs in the final shortest-path tree. Label-correcting

```
type
  TListStatus = (statNotInList, statNowInList,
                 statWasInList);
  TLink = class;           // Forward declaration.
  TNode = class(TObject)
    public
      x, y      : Integer; // The node's location.
      links     : TList;   // Links to neighbors.

      from_link : TLink;   // Path tree link.
      best_cost : Integer; // Best cost so far.
      status    : TListStatus;

      constructor Create;
      destructor Destroy; override;
  end;

  TLink = class(TObject)
    public
      node1, node2 : TNode;   // The endpoints.
      cost         : Integer; // The link's cost.
      in_tree      : Boolean; // In the path tree?
  end;
```

Figure 4: Enhanced *TNode* and *TLink* classes used by the example program PathS.

algorithms sometimes make mistakes; they may occasionally add a link that doesn't belong in the shortest-path tree. In that case, the algorithm will later correct itself by removing the incorrect link and replacing it with a different one.

Label Setting

The label-setting algorithm described here builds a candidate list to keep track of nodes that may be ready to join the shortest-path tree. The algorithm repeatedly removes a node from this list and adds it to the growing tree. When it adds a node to the shortest-path tree, the algorithm keeps track of the link that leads to that node in the tree. It also keeps track of the distance from the tree's root to each node, and whether the node is currently in the candidate list.

To track these items, the program uses the enhanced *TNode* and *TLink* classes shown in Figure 4. In the *TNode* class, *from_link* is the link that connects the node to the tree, *best_cost* is the current shortest distance from the root node to the node, and *status* tells whether the node is currently in the shortest-path tree.

In the *TLink* class, the *in_path* variable indicates whether the link is in the shortest path between two selected nodes. This value isn't needed for the shortest-path calculation, but it makes displaying the shortest path easier.

The label-setting algorithm begins by initializing the *best_cost* value for each node in the network to a very large value (32,767), and by setting each node's status to *statNotInList*. It then places the selected root node in the candidate list, and sets the root node's *best_cost* value to 0.

Next, while the candidate list is non-empty, the program searches it to find the next node to add to the shortest-path

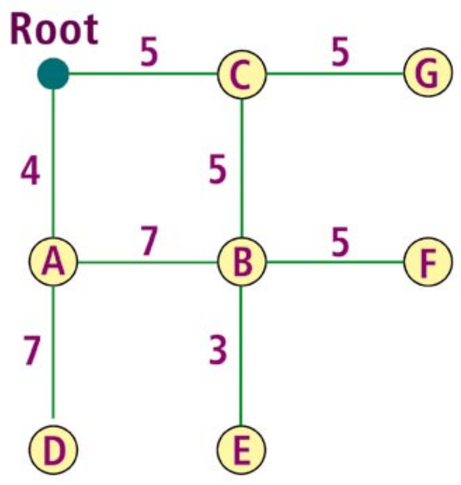


Figure 5: A small network.

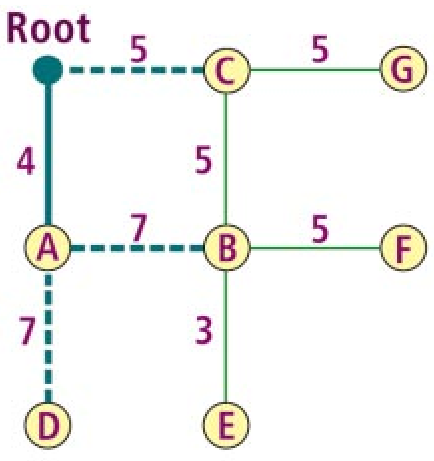


Figure 6: A partial shortest-path tree.

tree. It selects the node that has the smallest *best_cost* value. Once it has found the node with the smallest cost, the algorithm removes it from the candidate list and sets its status to *statWasInList*. At this point, the node is permanently in the shortest-path tree, and its *best_cost* and *from_link* fields have their final values. The algorithm does not consider this node again.

The program then examines the links leaving the node that was just added to the tree and the neighbor nodes at the other end of those links. For each neighbor that has not already been removed from the candidate list, the algorithm calculates the cost from the root to the node plus the cost of the link to the neighbor. This gives the total length to the neighbor via the shortest path through the node.

If the new total is less than the neighbor's current *best_cost* value, the algorithm updates *best_cost*. It stores a reference to the link in the neighbor's *from_link* to indicate that the neighbor may be connected to the tree using this link. Finally, if the neighbor is not already in the candidate list, the algorithm adds it.

Setting an Example

For a concrete example, consider the small network in Figure 5. Letters indicate nodes, and numbers indicate link costs. Initially, the candidate list contains only the root node and has a *best_cost* value of 0. Because it is the only item in the candidate list, the root node has the smallest *best_cost* value. The algorithm removes the root from the list and considers its neighbor nodes A and C. The *best_cost* value to the root node is 0. That value, plus the cost of the link from the root to node A, is 0 + 4 = 4. This is less than node A's current *best_cost* value of 32,767, so the program changes A's *best_cost* value to 4 and its *from_link* value to indicate the link between the root and node A. It then adds node A to the candidate list.

The algorithm then considers the other neighbor of the root: node C. Using steps similar to the ones it used to examine node A, the program sets node C's *best_cost* to 5 and adds it to the candidate list. The program is done examining the root node's neighbors.

Next, the program searches the candidate list for the node with the smallest *best_node* value. Currently, nodes A and C are in the list, and node A has the smaller *best_cost* value of 4. The algorithm removes node A from the list and examines its neighbors. At this point, node A has its final *best_cost* and *from_link* values, and will not be considered again.

When it examines node A's neighbors, the algorithm adds nodes B and D to the candidate list, both with *best_cost* values of 11. Figure 6 shows the network at this point. The bold link from the root to node A represents a link that will be in the final shortest-path tree. The dashed links connecting nodes C, B, and D represent the current *from_link* values for the nodes in the candidate list.

The algorithm searches the candidate list again and finds that node C has the smallest *best_cost* value at 5. It removes node C from the candidate list and considers its neighbors B and G. The distance to node B via node C is node C's *best_value* plus the cost of the link from node C to node B; that is, 5 + 5 = 10. This is lower than node B's current *best_cost* value of 11, so the algorithm updates B's *best_cost* and *from_link* values. Because node B is already in the candidate list, the algorithm does not need to add it again.

The algorithm considers node C's other neighbor, node G, and adds it to the candidate list. This gives the result shown in Figure 7. The algorithm continues removing the node from the candidate list with the smallest *best_cost* value until the list is empty. The result is the shortest-path tree shown in Figure 8.

Delphi Details

Listing One (on page 24) shows Delphi code that finds the shortest-path tree rooted at the node *StartNode*. To make working with all the nodes easier, the program keeps references to all nodes in the *Nodes TList* object. It keeps references to all the links in the *Links TList* object.

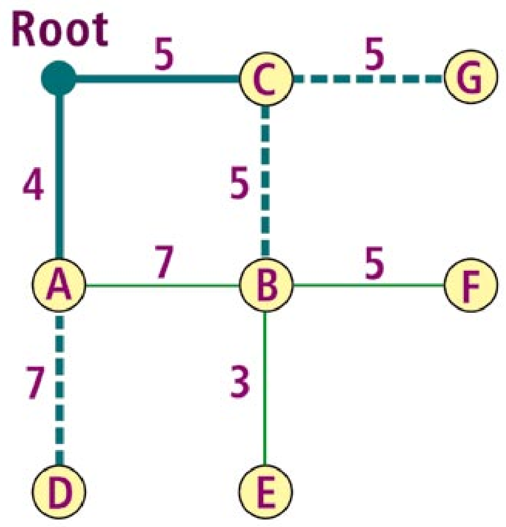


Figure 7: Adding to the shortest-path tree.

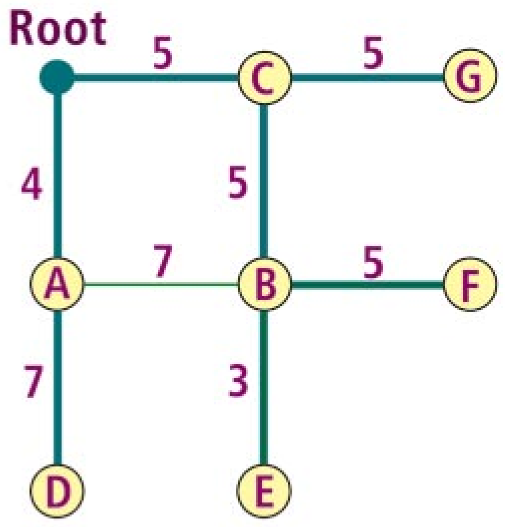


Figure 8: A final shortest-path tree.

Having seen the example, you should have little trouble following the Delphi code. The procedure initializes each node's *best_cost* value to INFINITY (32,767). It places *StartNode* in the candidate list. Then, as long as the candidate list is non-empty, it removes the node with the smallest *best_cost* value and processes it.

When the *FindPathTree* procedure finishes, each node's *from_link* value indicates the link that connects it to the shortest- path tree. A program can use these fields to trace a path backward from an end node to the start node. The code shown in Figure 9 does just that. It first sets each link's *in_path* value to False. It then traces the path backward from *EndNode* to *StartNode*, setting *in_path* to True for each link along the way. The PathS program uses the *in_path* values to determine which links to draw in bold.

The PathS project (shown in Figure 10) uses this code to display shortest paths in a network. Click on a node with the left mouse button to select a start node. At that point, the program finds the shortest-path tree rooted at the node. Then, click the right mouse button on an end node. The program uses the shortest-path tree to find and display the shortest path between the nodes. (PathS, and all other projects described in this article, are available for download; see end of article for details.)

```
// Find the shortest path from StartNode to EndNode.
procedure TPathForm.FindPath;
var
  link_num : Integer;
  node     : TNode;
  link     : TLink;
begin
  // Clear the previous data.
  for link_num := 0 to Links.Count - 1 do begin
    link := Links.Items[link_num];
    link.in_path := False;
  end;

  // Do no mode if StartNode = nil or EndNode = nil.
  if ((StartNode = nil) or (EndNode = nil)) then Exit;

  // Trace the path from EndNode back to StartNode
  // marking the links' in_path fields.
  TotalCost := 0;
  node := EndNode;
  while node <> StartNode do begin
    link := node.from_link;
    link.in_path := True;
    if link.node1 = node then
      node := link.node2
    else
      node := link.node1;
    TotalCost := TotalCost + link.cost;
  end;
end;
```

Figure 9: Tracing a path backward from *EndNode* to *StartNode*.

Label Correcting

The label-setting algorithm spends a lot of time searching through its candidate list for the node with the smallest *best_cost* value. That node's *best_cost* and *from_links* have reached their final values, so the algorithm adds the node to the final shortest-path tree.

Label-correcting algorithms take a different approach. Rather than searching the candidate list for the node with the smallest *best_cost*, the label-correcting algorithm just takes any node from the list. The algorithm adds the node to the shortest-path tree and considers its neighbors, just as a label-setting algorithm does.

Later, as it continues to add nodes to the shortest path tree, the algorithm may discover it can reduce the *best_cost* value of a node that's already in the tree. In that case, it updates the node's *best_cost* and *from_link* values and places it back in the candidate list. That allows the program to correct any other incorrect paths it may have made using the node's previous *best_cost* value.

Listing Two, on page 24, shows Delphi source code that finds a shortest-path tree using a label-correcting algorithm. The PathC program, shown in Figure 11, calculates shortest paths using a label-correcting algorithm. Aside from its choice of algorithm, the example program PathC is very similar to the example program PathS.

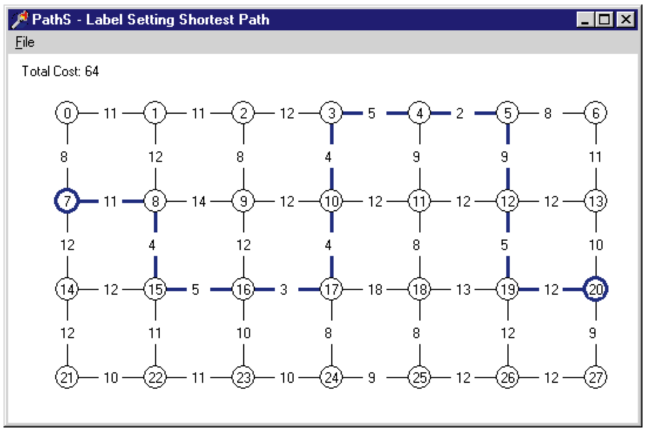


Figure 10: The example program PathS displaying a shortest path.

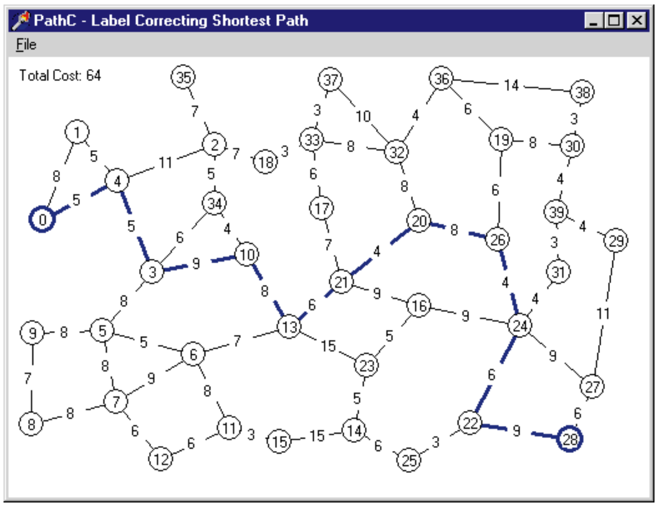


Figure 11: The example program PathC.

Label-correcting and label-setting algorithms are comparable in speed. One big difference between the two is that label-correcting algorithms cannot handle networks with negative cost cycles. If several links have negative costs and the program can find a circular path through those links, a label-correcting algorithm will follow the loop again and again, reducing the total cost of the path it is following.

On the other hand, a label-setting algorithm never reconsiders a node once it has been removed from the candidate list. It won't necessarily produce meaningful results for negative-length cycles, but it won't get stuck in an infinite loop either. Of course, for most physical networks, such as street networks, negative link costs don't mean much anyway, so you may never need to use this kind of network.

Conclusion

Label-correcting algorithms save time by not searching for the next node to remove from the shortest-path tree. They lose time when they are forced to put a node back in the candidate list and process it again.

Some hybrid algorithms try to reduce the amount of time needed to select a node from the candidate list while improving the chances of making a good choice. One variation adds nodes that have previously been on the candidate list to the beginning of the list, and it adds others at the end. The idea is to revisit previously examined nodes quickly, before long false paths are built using incorrect *best_cost* values. This modification produces a modest improvement without greatly complicating the algorithm.

Using one of these algorithms, you can build sophisticated network applications in Delphi. You can build programs that find paths through street networks, route packages through transshipment networks, or plan cable installations. You may even find an improvement on that all-important shortest path from your office to your home. ∆

The projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\DEC\DI9812RS.

Rod's just-released book, *Ready-to-Run Delphi 3.0 Algorithms* [John Wiley & Sons, 1998], explains dozens of algorithms like the ones described in this column. Rod has also written several other books, including *Custom Controls Library* [John Wiley & Sons, 1998] and *Visual Basic Graphics Programming* [John Wiley & Sons, 1997]. Learn more about his books at http://www.vb-helper.com, or contact him at RodStephens@vb-helper.com.

Begin Listing One — Label Setting

```
// Find the shortest path tree.
procedure TPathForm.FindPathTree;
var
  node_num, link_num    : Integer;
  best_node, best_cost  : Integer;
  new_cost              : Integer;
  node, to_node         : TNode;
  link                  : TLink;
  candidates            : TList;
begin
  // Do nothing if StartNode = nil.
  if StartNode = nil then Exit;

  // Clear previous path tree data.
  for node_num := 0 to Nodes.Count - 1 do begin
    node := Nodes.Items[node_num];
    node.from_link := nil;
    node.best_cost := INFINITY;
    node.status := statNotInList;
  end;

  // Create the candidate list and add the root to it.
  candidates := TList.Create;
  candidates.Add(StartNode);
  StartNode.best_cost := 0;
  StartNode.status := statNowInList;

  // While the candidate list is not empty, process it.
  while candidates.Count > 0 do begin
    // Find the node with the smallest cost.
    node := candidates.Items[0];
    best_cost := node.best_cost;
    best_node := 0;
    for node_num := 1 to candidates.Count - 1 do begin
      node := candidates.Items[node_num];
      if node.best_cost < best_cost then begin
        best_cost := node.best_cost;
        best_node := node_num;
      end;
    end;

    // Remove this node from the candidate list.
    node := candidates.Items[best_node];
    candidates.Remove(node);
    node.status := statWasInList;

    // Add the node's neighbors to the candidate list.
    for link_num := 0 to node.links.Count - 1 do begin
      // Get the neighbor node.
      link := node.links.Items[link_num];
      if link.node1 = node then
        to_node := link.node2
      else
        to_node := link.node1;

      // See if node had been on the candidate list before.
      if to_node.status <> statWasInList then begin
        // See if we can improve its best_cost.
        new_cost := node.best_cost + link.cost;
        if new_cost < to_node.best_cost then begin
          // This is an improvement. Update the
          // neighbor and add it to the list.
          to_node.best_cost := new_cost;
          to_node.from_link := link;
          if to_node.status = statNotInList then begin
            to_node.status := statNowInList;
            candidates.Add(to_node);
          end;
        end;
      end;
    end; // End examining the node's neighbors.
  end; // Repeat until candidate list is empty.

  // Free the candidate list.
  candidates.Free;
end;
```

End Listing One

Begin Listing Two — Label Correcting

```
// Find the shortest path tree.
procedure TPathForm.FindPathTree;
var
  node_num, link_num : Integer;
  new_cost           : Integer;
  node, to_node      : TNode;
  link               : TLink;
  candidates         : TList;
begin
  // Do nothing if StartNode = nil.
  if StartNode = nil then Exit;

  // Clear previous path tree data.
  for node_num := 0 to Nodes.Count - 1 do begin
    node := Nodes.Items[node_num];
    node.from_link := nil;
    node.best_cost := INFINITY;
    node.status := statNotInList;
  end;

  // Create the candidate list and add the root to it.
  candidates := TList.Create;
  candidates.Add(StartNode);
  StartNode.best_cost := 0;
  StartNode.status := statNowInList;

  // While the candidate list is not empty, process it.
  while candidates.Count > 0 do begin
    // Remove the first item from the candidate list.
    node := candidates.Items[0];
    candidates.Remove(node);
    node.status := statWasInList;

    // Add the node's neighbors to the candidate list.
    for link_num := 0 to node.links.Count - 1 do begin
      // Get the neighbor node.
      link := node.links.Items[link_num];
      if link.node1 = node then
        to_node := link.node2
      else
        to_node := link.node1;

      // See if we can improve the path
      // to the neighbor.
      new_cost := node.best_cost + link.cost;
      if new_cost < to_node.best_cost then begin
        // This is an improvement. Update the
        // neighbor and add it to the list.
        to_node.best_cost := new_cost;
        to_node.from_link := link;
        // Add the node to the candidate list.
        if to_node.status <> statNowInList then begin
          candidates.Add(to_node);
          to_node.status := statNowInList;
        end;
      end;
    end; // End examining the node's neighbors.
  end; // Repeat until candidate list is empty.

  // Free the candidate list.
  candidates.Free;
end;
```

End Listing Two

By Cary Jensen, Ph.D.

Delphi Database Development

Part IV: Data Modules

Over the past few months, this series has taken an in-depth look at the foundations of Delphi database development. Topics have included an overview of the Borland Database Engine (BDE), as well as how to use and configure the basic BDEDataSet components: Table, Query, and StoredProc. This month's "DBNavigator" examines the uses and misuse of data modules.

The Data Module Defined

A data module is a form-like container that first appeared in Delphi 2. Unlike a form, a data module is never visible to the user. Instead, its sole purpose is to hold one or more components that can be shared by other parts of your program. One of the common uses of a data module is to hold DataSets (including the *TClientDataSet*, which is available in the Delphi 3 and 4 Client/Server Suites), permitting two or more forms within the application to share the properties and methods defined for those DataSets.

The alternative to using a data module is to place a different set of data-set components on each form in your application. While there is certainly nothing fundamentally wrong with this approach, it means that every form contains data-set components that must be individually configured. If two or more forms need to display the same data or event handlers (for providing client-side data validation, for example), placing those DataSets on a single data module shared by the two or more forms provides easier development and maintenance.

But DataSets aren't the only components that can be used with data modules. In fact, a data module can hold any component that doesn't descend from *TControl*. This includes MainMenu, PopupMenu, OLEContainer, IBEventAlerter, Timer, and any component on the Data Access and Dialogs pages of the Component palette, just to name a few.

Just because a data module can hold a certain component, however, doesn't mean it's necessarily a good idea to always place that type of component on a data module. For example, imagine that you place a MainMenu on a data module. Any event handlers for that menu would naturally be placed on that data module as well. As a result, any reference to the object variable *Self* from within those event handlers would refer to the data module, and not the form from which a particular menu item was selected. Such an arrangement would require complex code that would probably be difficult to maintain, thereby canceling any advantage afforded by the data module.

Using a Data Module

Delphi makes it easy to use a data module and to share it between multiple forms. The following steps demonstrate how to use a single data module to share a Table between two forms:

1) Create a new project.
2) Add the following components to the main form: From the Data Controls page of the Component palette, add one

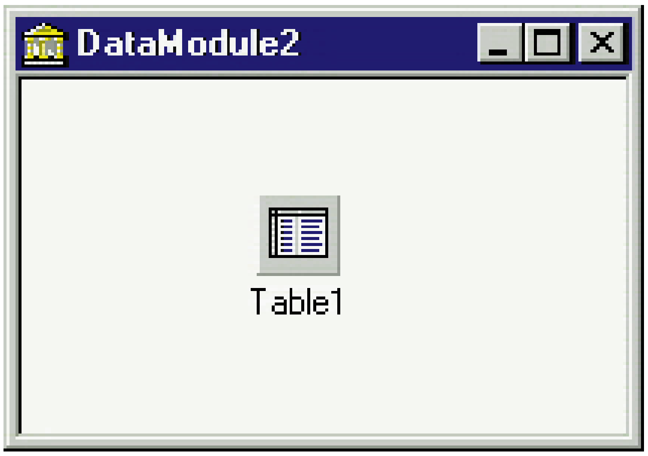


Figure 1: A data module containing a single Table component that can be shared by multiple forms.

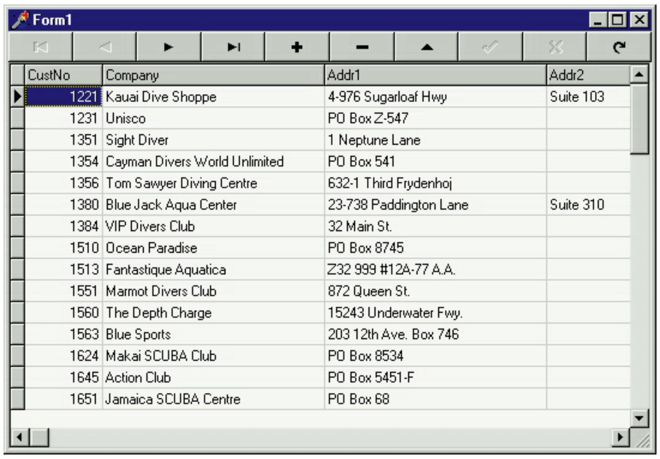
DBNavigator and one DBGrid, and from the Data Access page, add one DataSource. Set the *Align* property of the DBNavigator to *alTop*, and the *Align* property of the DBGrid to *alClient*. Next, set the *DataSource* properties of the DBNavigator and DBGrid to *DataSource1*.

3) Next, add a data module. Select **File | New**, then double-click the Data Module Wizard in the Object Repository. (In Delphi 2 and 3, you can also select **File | New Data Module**.)
4) Add a single Table to the data module. Set the Table's *DatabaseName* property to `DBDEMOS`, and its *TableName* property to `CUSTOMER.DB`.
5) Double-click the data module to add an *OnCreate* event handler. Add the following code to this event handler:

```
procedure TDataModule2.DataModule2Create(
  Sender: TObject);
begin
  Table1.Open;
end;
```

The data module should now look like that shown in Figure 1.

6) The only step remaining is to associate the DataSet on the data module with the data source on the main form.



CustNo	Company	Addr1	Addr2
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
1231	Unisco	PO Box Z-547	
1351	Sight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310
1384	VIP Divers Club	32 Main St.	
1510	Ocean Paradise	PO Box 8745	
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	
1551	Marmot Divers Club	872 Queen St.	
1560	The Depth Charge	15243 Underwater Fwy.	
1563	Blue Sports	203 12th Ave. Box 746	
1624	Makai SCUBA Club	PO Box 8534	
1645	Action Club	PO Box 5451-F	
1651	Jamaica SCUBA Centre	PO Box 68	

Figure 2: This form displays data defined by a Table component contained in a data module.

Doing this requires that the unit defining the main form use the unit that defines the data module. The easiest way to achieve this is to select *Form1*, then select **File | Use Unit** to display the Use Unit dialog box. Select *Unit2*. In response, Delphi adds a **uses** clause specifying *Unit2* to the implementation section of *Unit1*. Once you're using *Unit2* from *Unit1*, select *DataSource1*, and set its *DataSet* property to *DataModule2.Table1*. The project is complete. Press F9 to run this project. An example of the main form is shown in Figure 2.

As you learned earlier, any component from the Data Access page of the Component palette can be placed onto a data module. You might then ask, didn't we place the DataSource component on the data module, as well? The answer is we could have; the project would have worked. However, unless you have a particularly good reason to do otherwise, it's generally best to place DataSource components onto individual forms. Doing so permits you to convert a form from using one data module to another by simply changing the *DataSet* property of the DataSource. If the DataSource component appears on the data module, switching a form from using one data module to another requires that the *DataSource* property for every data-aware control on the form be changed. Depending on the number of data-aware controls, this can be a major task.

Sharing Data Using a Data Module

While the preceding example demonstrates how to create and use a data module, the DataSet on that data module was used by a single form. While there's nothing inherently wrong with doing this, the real power of the data module is realized when two or more forms share one or more DataSets appearing on the data module. The following steps expand on the preceding example, demonstrating how multiple forms can share a DataSet on a data module:

1) With the project we just created open in Delphi, add another form to it by selecting **File | New Form**. By default, the newly added form is named *Form3*.
2) We now want to add one or more data-aware controls to *Form3*. Begin by adding a DBNavigator to *Form3*. Set the *Align* property of this DBNavigator to *alTop*.
3) You could continue adding data-aware components in this fashion, but because the next data-aware controls you want to add are DBEdit controls, there's an easier way to do this. Select **DataModule2** and right-click **Table1** to display the Fields Editor. In Delphi 4, press Ctrl F to add all fields from **Table1** to the Fields Editor. Using Delphi 3, right-click the Fields Editor (or press the **Add** button with Delphi 2), then select **OK** to choose all fields from the Add Fields dialog box.
4) In the next steps, you're going to drag and drop several fields from the Fields Editor onto *Form3*. Make sure that both *Form3* and the Fields Editor are visible. Then, select the **CustNo**, **Company**, **City**, **State**, and **Zip** fields in the Fields Editor. Do this by holding down Ctrl while clicking each of these fields (see Figure 3).
5) Now, drag the selected fields from the Fields Editor, and drop them onto *Form3*. It doesn't matter from which of

your selected fields you begin the drag operation — all selected fields will be dragged.

6) Because *Form3* does not yet use *Unit2* (the unit that defines the data module), you're first presented with a dialog box asking you to confirm that you want to use *Unit2* from *Form3*. Select **Yes**. Once you do, Delphi creates one data-aware control and one label for each of the fields you dropped. It also adds a DataSource. Furthermore, the properties of each of the data-aware controls are set to use the newly-created data source, and their *DataField* properties are set to the appropriate field in *Table1*. In addition, the *DataSet* property of the newly-created data source is set to *DataModule2.Table1*. All you need to do is set the *DataSource* property of the DBNavigator to *DataSource1*, then resize *Form3* so it better accommodates the newly-placed fields. An example of how *Form3* might now look is shown in Figure 4.

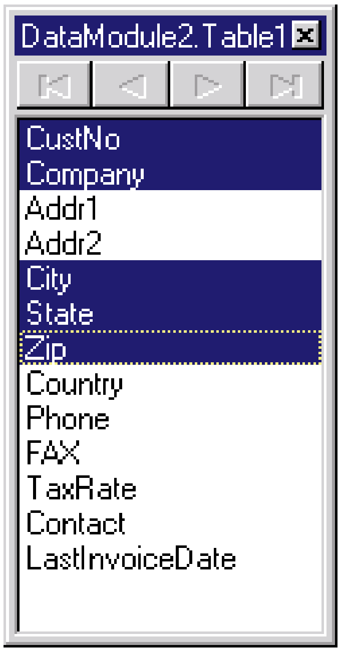


Figure 3: Fields can be selected within the Fields Editor, then dropped onto a form to automatically create data-aware controls and their associated labels.

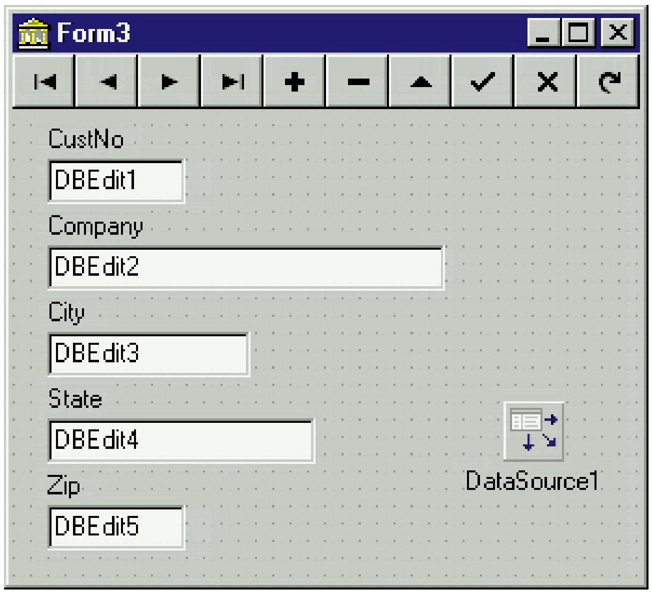


Figure 4: The controls placed on this form were dragged and dropped from the Fields Editor for DataModule2.Table1.

7) To complete this project, add a menu to *Form1*, and create an event handler that will display *Form3*. To do this, return to *Form1* and place a MainMenu component on it.

8) Double-click the main menu to display the Menu Designer. With the first menu item selected in the Menu Designer, enter the value `Show Single Record` as the *Caption* property. With this menu item still selected, select the Events tab of the Object Inspector and double-click the *OnClick* event to create its event handler. Enter the following code into this event handler:

```
procedure TForm1.ViewSingleRecord1Click(
  Sender: TObject);
begin
  with TForm3.Create(Self) do Show;
end;
```

9) Because the code in the preceding step references *TForm3*, which is defined in *Unit3*, you must add *Unit3* to *Form1*'s **uses** clauses. With *Form1* selected, choose **File | Use Unit**, and select **Unit3**, as you did earlier in this example.

10) The preceding code also assumes that *Form3* won't be auto-created. Because it's auto-created by default, select **Project | Options**, select **Form3** in the **Auto-create forms** list, and click the right-arrow button to move it into the **Available forms** list (see Figure 5).

11) Finally, to ensure the release of *Form3* when the user closes it, select **Form3**; then with **Form3** selected in the Object Inspector, go to the Events page and add the following *OnClose* event handler:

```
procedure TForm3.FormClose(Sender: TObject);
begin
  Action := caFree;
end;
```

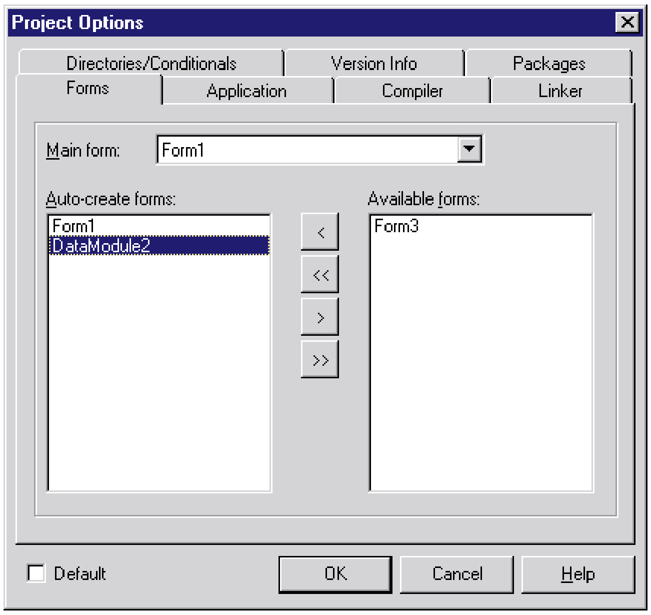
This code causes the instance of *Form3* that's being closed to be destroyed. Failure to add this code results in every instance of *Form3* that is created remaining in memory, albeit invisible to the user after being closed, until the application terminates.

12) Your project is complete. Save the project and run it. With *Form1* displayed, select **View Single Record**. Now, using the DBNavigator on either of the displayed forms, navigate to another record. Both forms display the same current record (see Figure 6). This occurs because both use one single-Table component on the data module.

Should You Always Use a Data Module?

With apologies to Dennis Miller, I don't want to get on a rant here, but I can't believe it's still necessary to answer the question about whether you should always put your DataSet components in a data module. However, at the Inprise 98 conference in Denver, I actually heard an Inprise employee state in front of a large audience that you should always put DataSets into data modules. This is understandable only in that it's likely that this person doesn't build real-world database applications for a living. Nonetheless, this wrong-thinking must be addressed.

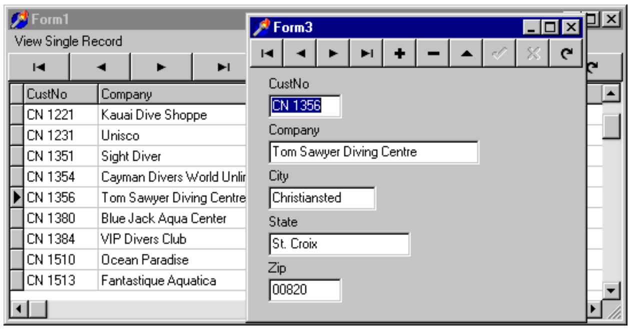
The answer is: No, you don't always put DataSets on data modules. Yes, data modules are great. Yes, they provide you with a



Main form: Form1

Auto-create forms:	Available forms:
Form1	Form3
DataModule2	

Figure 5: Use the Project Options dialog box to remove a form or data module from the **Auto-create forms** list.



	CustNo	Company
	CN 1221	Kauai Dive Shoppe
	CN 1231	Unisco
	CN 1351	Sight Diver
	CN 1354	Cayman Divers World Unlir
▶	CN 1356	Tom Sawyer Diving Centre
	CN 1380	Blue Jack Aqua Center
	CN 1384	VIP Divers Club
	CN 1510	Ocean Paradise
	CN 1513	Fantastique Aquatica

CustNo: CN 1356
Company: Tom Sawyer Diving Centre
City: Christiansted
State: St. Croix
Zip: 00820

Figure 6: The multi-record view and the single record view are synchronized to the same record, because both use the same Table component on the data module.

single repository for configurations and event handlers that can be shared. However, they aren't appropriate in every situation.

When Should You Use a Data Module?

Data modules are a perfect solution for those situations where two or more forms, or other similar containers, need to share a common set of components. The project example built earlier in this article is a good example of that. Because *Form1* and *Form3* needed to share a common view of *Table1*, including any ranges, filters, sort orders, calculated fields, and so on, the data module provided an easy and effective means for this. This sharing isn't limited to single Tables either; there's no reason why two or more forms can't share a multitude of DataSets, dialogs, timers, and the like, placed on one or more data modules.

In fact, there are a number of situations where you must use a data module. For example, if you're using the MIDAS technology found in Delphi 3 and Delphi 4 Client/Server Suites, you must place your BDEDataSet components, as well as any provider components that you need, onto a remote data module. Remote data modules are special data modules that implement certain interfaces necessary for the cross-application communication that's required by MIDAS.

Another example where you're required to use a data module can be found with the Web Broker components. These components, also available in Delphi 3 and Delphi 4 Client/Server Suites, as well as available separately from Inprise, make use of a Web module (a *TDataModule* descendant). You define the actions to which your Web server extension can respond using the Web module. (If you use a WebDispatch component, a Web module isn't necessary.)

When Should You Avoid Data Modules?

What really bugs me about people saying you should always use a data module is that there are situations in which you shouldn't use a data module. And there are other situations where a data module can be used, but doing so unnecessarily complicates your applications.

The general rule of thumb is that you don't use a data module when there should be no sharing of a DataSet. The classic example of this is when you're creating reports that use Delphi DataSets for their data. These DataSets should never be shared. The reason for this is that VCL-based reporting tools must navigate a DataSet in order to print data. Imagine what can happen if one of these reports uses a DataSet on a data module, and that same DataSet is used by data-aware controls on a form. The user is viewing the form, then prints the DataSet. The next thing the user sees is their form scrolling frantically as the report navigates the DataSet. Not only can this be confusing to the user, it causes a catastrophic loss of performance for the report, as the data-aware controls in the user interface must be repainted for each record.

And this presents a best-case scenario for data-module sharing with reports. Imagine what happens to the report if the user is currently editing a record, and that record contains errors that prevent the cursor from leaving it. Imagine what would happen if a user prints two reports simultaneously, and the reports share a data module. They would be fighting for the control of the cursor, but the user may never know this. Clearly, reports should not share a DataSet.

While reports provide a clear example where DataSet sharing is unacceptable, there are two other situations where data modules are generally a bad idea. The first is when you have one form that uses a completely unique view of data. That view may involve either a table that's never viewed from any other form, or a table that makes use of a range, filter, or sort order used nowhere else in the application.

A second instance where data modules should typically be avoided is when you're writing multiple-instance forms. A multiple-instance form is one where more than one copy can be displayed simultaneously. Of course, part of such a design is that each instance displays a different record or set of records, different sort order, or some similar difference. Obviously, such forms can't share a single DataSet. The easiest way to design a multiple instance form is to add the DataSet or DataSets directly on the form. This ensures that each instance of the form has its own DataSet or sets, meaning that each form has its own cursor(s) and view(s) of the data.

For these last two examples, it could be argued that a data module could still be used. For unique view forms, a data module can be used, just not shared. Likewise, with multiple-instance forms, each instance of the form can be responsible for creating its own instance of a data module. However, using a data module in these cases unnecessarily complicates your application. Why use two containers — a form and a data module — when one will suffice? Because the primary benefit of the data module is simplicity, it's absurd to use a data module when it increases complexity.

A final note about data modules is in order. By default, they're auto-created. If you always use data modules, and they're always auto-created, it's likely all your DataSets will be opened when you start your application. This can result in long application start-up times, and an unnecessary number of table-locking resources being used. I once saw an application that had an auto-created data module containing about 100 DataSets. As you can imagine, one reason the client asked me to look at this application in the first place was that they were unhappy with the load time.

The solution to problems caused by auto-created data modules is to remove them from the **Auto-create forms** list on the Project Options dialog box, just as we did in the example presented earlier in this article. Once you do this, however, you must take responsibility for creating your data modules on-the-fly, before displaying a form that makes use of the components on the data module. This can be complicated, however. If one data module can be used by two or more forms, each form must test for the pre-existence of the data module upon the form's creation. If the data module doesn't yet exist, it must be created. Releasing the data module, if this is desired, also requires more coding. Specifically, because one data module may be used by more than one form, it's not enough to simply free the data module when a form is closing. Instead, you must implement some form of reference counting for the data module, so that you release it only when the last form requiring it is being closed.

Conclusion

Data modules provide you with a container that can hold components that can be shared by multiple forms. While data modules can simplify the development and maintenance of your applications, however, they're not appropriate for every situation. Inappropriate use of data modules can lead to unwanted side effects, as well as increase the overall complexity of your applications. ∆

The projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\DEC\DI9812CJ.

Cary Jensen is president of Jensen Data Systems, Inc., a Houston-based database development company. He is co-author of 17 books, including *Oracle JDeveloper* [Oracle Press, 1998], *JBuilder Essentials* [Osborne/McGraw-Hill, 1998], and *Delphi in Depth* [Osborne/McGraw-Hill, 1996]. He is a Contributing Editor of *Delphi Informant*, and is an internationally-respected trainer of Delphi and Java. For information about Jensen Data Systems consulting or training services, visit http://idt.net/~jdsi or e-mail Cary at cjensen@compuserve.com.

Delphi 3, 4 / ADO / ASP / OLEDB / IIS

By Ron Loewy

Much ADO about the Web

Using ActiveX Data Objects from Delphi Applications

OLEDB is the new Microsoft standard that provides a standard interface to data stores. OLEDB recognizes that an increasing number of applications are being written using the three-tier model that differentiates between data providers, logic processors, and information clients. Unlike ODBC and other data standards of the past (like the BDE), OLEDB isn't limited to traditional data sources (relational desktop or SQL-based databases). OLEDB providers can be written for e-mail clients, proprietary applications, data stores, graphic applications, and more.

The OLEDB SDK ships with an ODBC data provider that allows every ODBC data source to be used by an OLEDB information client application. While the OLEDB SDK is a powerful collection of COM interfaces you can use to create data providers, logic processors, or information clients, it's a complex API and can't be used directly by scripting languages.

Microsoft recognized this problem and built a set of Automation objects that simplify the tasks of data retrieval and updating: the ActiveX Data Objects (ADO). It combines features from older Microsoft data access object hierarchies, such as DAO and RDO, and uses OLEDB to provide access to any data store that has an OLEDB data provider. ADO can be employed by any programming language that can use Automation objects, including Delphi and Visual Basic. ADO's most popular application so far is the Web server application market, where ADO is used in Microsoft's Active Server Pages (ASP) applications.

ASP applications are interpreted HTML pages that embed JScript or VBScript. These scripts, implemented using ActiveScript (see Tom Stickle's "ActiveX Scripting" article in the February, 1998 *Delphi Informant*), can inspect the parameters sent from the Web browser, access databases using ADO, write to the HTML stream, or activate Automation objects to perform additional tasks.

Because Delphi can create Automation objects for use in ASP applications, it makes sense for your Delphi application and ASP code to share the same database; it also makes sense to use ADO in your Delphi applications (just as you do in your ASP code).

The ADO Object Hierarchy

ADO provides a set of loosely hierarchical objects. These objects provide the functionality to connect to data sources, query and update record sets, and report errors. The objects are exposed as dual-interface COM objects, so you can either import the msado15.dll file to create a Pascal type library, or use Automation. I use Automation in this article. It's a little slower than calling COM objects with a vtable, but the code is the equivalent of the ADO code I use in the ASP samples.

The *Connection* Object

The ADO *Connection* object is used to establish an open connection to a data source and represents a unique session with the data source. In Delphi terms, think of a *Connection* object as a combination of the Database and Session components.

You open a *Connection* using the *Open* method, which receives a "connection string," i.e. a string that specifies the data source you connect to. Assuming you created a system-level ODBC data source (using the Control Panel's 32-bit ODBC applet) called MyDB, the following Delphi code snippet will open the *Connection* and store it in an *OleVariant* variable called *Conn*:

```
var
  Conn : OleVariant;
  CS   : string;

...

  Cs := 'DSN=MyDB';
  Conn := CreateOleObject('ADODB.Connection');
  Conn.Open(Cs);
```

The *Connection* object can also be used to execute data provider-specific commands using the *Execute* method, and manage transactions using *BeginTrans*, *CommitTrans*, and *RollbackTrans*.

The *Recordset* Object

The *Recordset* object is used to read, update, or clear a set of records (tabular data). You can think of a *Recordset* as a Delphi Table or Query component.

The *Open* method is used to populate the *Recordset* object from the data source. You provide this method with the open *Connection* object you retrieved earlier, the SQL statement that specifies the record selection, and parameters that specify locking and cursor options.

The following is a Delphi code snippet that selects all the records in the table MyTable:

```
var
  Rs  : OleVariant;
  SQL : string;
begin
  Rs:= CreateOleObject('ADODB.Recordset');
  SQL := 'SELECT * FROM [MyTable]';
  Rs.Open(SQL, Conn, 3, 3);
```

Navigating the *Recordset* object is done with the *Move*, *MoveFirst*, *MoveLast*, *MoveNext*, and *MovePrevious* methods.

The *AddNew* method is used to add a new record to the *Recordset*, and the *Delete* method is used to remove records from the *Recordset*.

The *Fields* property of the *Recordset* object is used to gain access to the fields of the current record (the record pointed to by the cursor). These can be used to inspect and update the values of the specific columns in the record.

```
procedure SetCurrentRecordFields(Rs: OleVariant;
  FieldNames, FieldValues: array of OleVariant);
var
  i     : Integer;
  Fld   : OleVariant;
  fName : string;
  fVal  : OleVariant;
begin
  for i := Low(FieldNames) to High(FieldNames) do begin
    fName := FieldNames[i];
    Fld := Rs.Fields[fName];
    fVal := FieldValues[i];
    Fld.Value := fVal;
  end;
  Rs.Update;
end;
```

Figure 1: Updating the current row in the recordset using the *Fields* collection.

The *Field* Object

A *Field* object represents a column value in the current row of a *Recordset* object. The *Field* object provides access to the column's *Name*, *Type*, *Attributes*, and the *Value* of the column in the current *Recordset* row. If the field represents a BLOb (Binary) column, the *AppendChunk* and *GetChunk* methods can be used to write or read the binary data.

The code in Figure 1 shows how to update the current row in the *Recordset* using the fields collection of *FieldObjects*. Here, *Rs* is an open *Recordset* object, and *FieldNames* and *FieldValues* are open arrays that hold the names of the columns that need to be updated and the new values that need to be assigned to them.

For example, if we have a *Recordset* object called *Customers* and we need to update a customer address based on a form *CustomerForm*, we could use the following call:

```
SetCurrentRecordFields(Customers, ['Customer Name',
  'Address 1', 'Address 2', 'City', 'Zip', 'Telephone'],
  [CustomerForm.Name.Text, CustomerForm.Address1.Text,
  CustomerForm.Address2.Text, CustomerForm.City.Text,
  CustomerForm.Zip.Text, CustomerForm.Tel.Text];
```

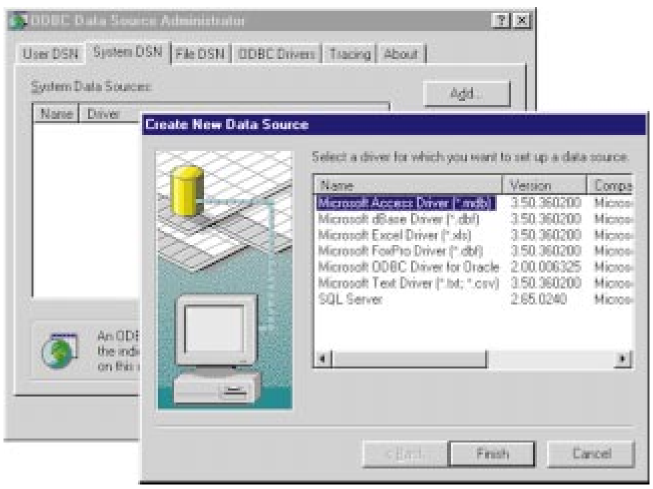
Additional ADO Objects

ADO includes additional objects that can be used in applications that access data. These include:

- The *Command* object, which allows you to activate a data source-specific command.
- The *Parameter* object, which can be used to pass parameters to commands executed with the *Command* object.
- The *Error* object, which can be used to analyze errors that happened during your database access or update operations.

Interfacing Delphi ADO Code with ASP

ASP is a great way to write Web applications. You can combine your page layout HTML with logic written in JavaScript or VBScript, and take advantage of Automation objects. When you need to perform complicated logic in your ASP application, scripting is less appealing.



Select a driver for which you want to set up a data source.

Name	Version	Compa
Microsoft Access Driver (*.mdb)	3.50.360200	Micros
Microsoft dBase Driver (*.dbf)	3.50.360200	Micros
Microsoft Excel Driver (*.xls)	3.50.360200	Micros
Microsoft FoxPro Driver (*.dbf)	3.50.360200	Micros
Microsoft ODBC Driver for Oracle	2.00.006325	Micros
Microsoft Text Driver (*.txt; *.csv)	3.50.360200	Micros
SQL Server	2.65.0240	Micros

Figure 2: Selecting the data source from the System DSN page of the 32-bit ODBC applet.

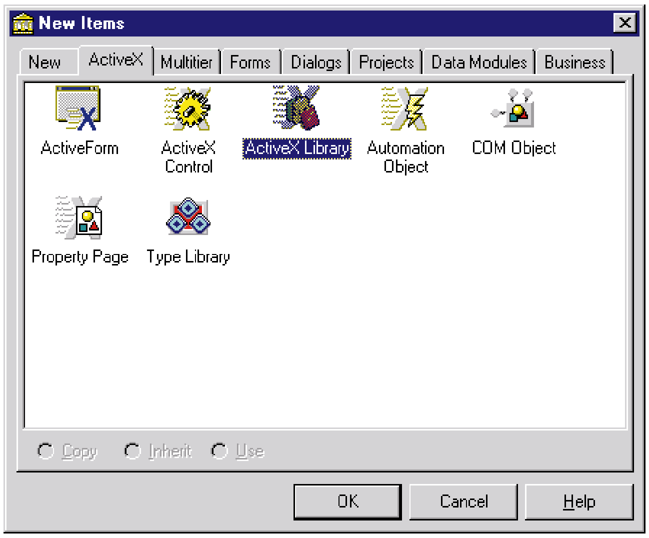


Figure 3: The ActiveX page of the New Items dialog box.

We can take advantage of Delphi's ability to create Automation objects and access the same databases ASP can (using ADO) to write the complex logic parts of the application in Delphi, while using ASP for the presentation and light-weight features.

Once you've created and compiled your Automation object in Delphi, you will use ASP's *Server.CreateObject* method to create an instance of the object and use it.

For this article, I created a simple customer database in Microsoft Access 97 and added it as a System DSN on my machine. The code archive for this article contains the pre-defined Access database. (The example Delphi project, ASP file, and Access database are available for download; see end of article for details.)

Creating a System DSN

To register the customer database as a System DSN, follow these steps:

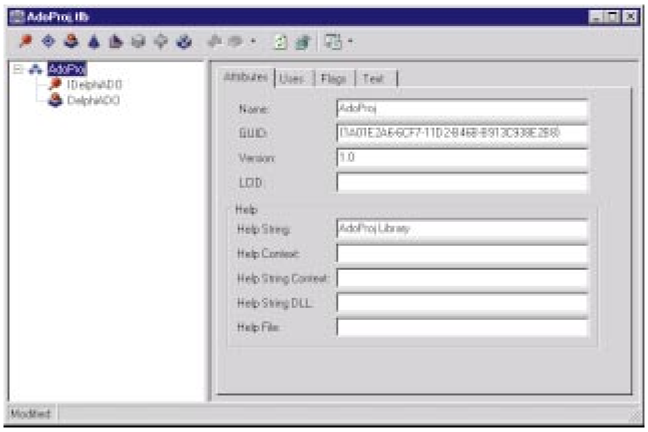
1) Go to Control Panel and activate the ODBC applet.
2) Select the System DSN tab and click the **Add** button (see Figure 2).
3) Choose the Access driver and click **Finish**.
4) At the ODBC setup dialog box, enter `DISample` as the **Data Source Name**.
5) Click the **Select** button and browse for the Customers.mdb file that accompanies this article.
6) Click **OK** twice to close the dialog boxes.

The System DSN, DISample, now points to the sample database we will use in this article.

Creating a Delphi Automation Object

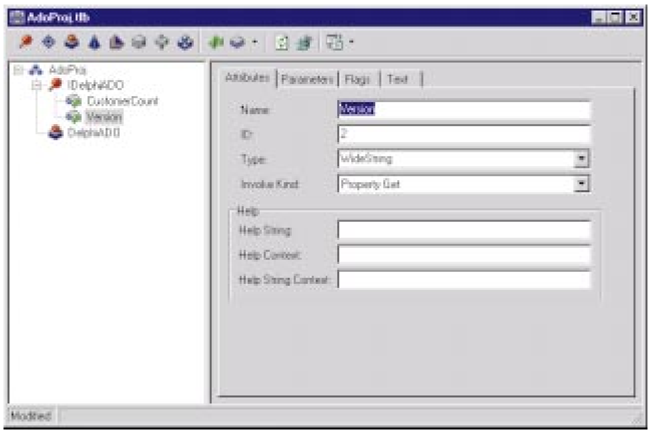
To create a Delphi Automation object, follow these steps:

1) Create a new ActiveX Library project by selecting **ActiveX Library** from the ActiveX page of the New Items dialog box (see Figure 3), and save it as AdoProj.dpr.
2) Create an Automation object by selecting **Automation Object** from the ActiveX page of the New Items dialog box.



Name: AdoProj
GUID: {1A01E2A6-6CF7-11D2-B468-E913C938E2B8}
Version: 1.0
LCID:
Help
Help String: AdoProj Library
Help Context:
Help String Context:
Help String DLL:
Help File:

Figure 4: The Delphi Type Library editor displaying the AdoProj.tlb file.



Name: Version
ID: 2
Type: WideString
Invoke Kind: Property Get
Help
Help String:
Help Context:
Help String Context:

Figure 5: The AdoProj type library with the *CustomerCount* and *Version* properties added.

3) The Automation Object Wizard will be displayed. Enter `DelphiADO` as the **Class Name** and click **OK**. The AdoProj type library (AdoProj.tlb) will be created and displayed in Delphi's Type Library editor (see Figure 4).
4) Close the Type Library editor and save the implementation unit as ADOObj.pas.

You're now ready to add the methods and properties of the object. Reopen the Type Library editor by selecting **View | Type Library**. Right-click on the *IDelphiADO* interface and select **New | Property** to create a read-only Integer property named *CustomerCount*. Delphi will automatically create get and put methods, so you will need to delete the put method. Repeat this step to create a WideString read-only property named *Version* (see Figure 5).

Click the Type Library editor's **Refresh Implementation** button and enter the functions' code in the ADOObj unit, as shown in Figure 6.

We can now compile the object. After compiling, register the object with the system by selecting **Run | Register ActiveX Server**.

```
function TDelphiADO.Get_CustomerCount: Integer;
var
  Conn : OleVariant;
  Rs   : OleVariant;
  SQL  : string;
begin
  Conn := CreateOleObject('ADODB.Connection');
  Conn.Open('DSN=DISample');
  Rs := CreateOleObject('ADODB.Recordset');
  SQL := 'SELECT * FROM [Customers]';
  Rs.Open(SQL, Conn, 3, 3);
  Result := Rs.RecordCount;
  Rs.Close;
end;

function TDelphiADO.Get_Version: WideString;
begin
  Result := 'Version 1.0';
end;
```

Figure 6: The *Get_CustomerCount* and *Get_Version* methods.

```
<%@ Language=VBScript%>
<HTML>
<HEAD>
<META NAME="GENERATOR"
  CONTENT="HyperAct eAuthor Help 3.0 Preview 5">
<META HTTP-EQUIV="Content-Type"
  CONTENT="text/html; CHARSET=iso-8859-1">
<TITLE>Document Title</TITLE>
</HEAD>
<BODY>
<%
  Set DelphiObj = Server.CreateObject("AdoProj.DelphiADO")
%>
<H3>This page provides information gathered from
 a Delphi Automation object</H3>
Object Version: <%= DelphiObj.Version %><BR>
<%
  Set Conn = Server.CreateObject("ADODB.Connection")
  Conn.open "DISample", "", ""
  Set Rs = Server.CreateObject("ADODB.Recordset")
  Rs.open "SELECT * FROM [Customers]", Conn, 3, 3
%>
Number of records in recordset <%= Rs.RecordCount %>
<BR>
<%
  Counter = DelphiObj.CustomerCount
%>
Number of records reported by Delphi object <%= Counter
%>
<BR>
End of Story
</BODY></HTML>
```

Figure 7: The ASP file, MyTest.asp.

Calling the Delphi Object from ASP

To test the object from ASP, you will need to create a virtual directory on your Internet Information Server (IIS) or Personal Web Server (PWS). Define a virtual directory named DIADO, and give it read and scripts permissions. The ASP file shown in Figure 7 needs to be placed in this directory.

You embed scripts in an ASP page between <% and %> tags. The ASP *Server* object is used to start external Automation objects with the *CreateObject* method. For our sample, the following statement will start our object and assign it to a VBScript variable:

```
<%
  Set DelphiObj = Server.CreateObject("AdoProj.DelphiADO")
%>
```

The object is identified by the project name, separated by a dot from the Delphi object name defined when you created the object in Delphi. We can now call the object using the following syntax:

```
Object Version: <%= DelphiObj.Version %><BR>
```

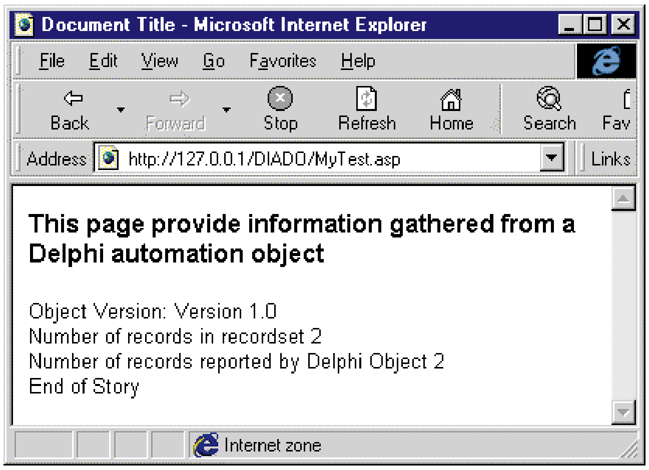
Calling the code that accesses the database via ADO is just as easy:

```
<% Set Counter = DelphiObj.CustomerCount %>
Number of records reported by Delphi Object <%= Counter %>
```

This obviously is a very simple use for Delphi accessing ADO from within an ASP application. It's so simple that we can check the results returned by our object with comparable code written in pure JScript:

```
<%
  Set Conn = Server.CreateObject("ADODB.Connection")
  Conn.open "DISample", "", ""
  Set Rs = Server.CreateObject("ADODB.Recordset")
  Rs.open "SELECT * FROM [Customers]", Conn, 3, 3
%>
Number of records in recordset <%= Rs.RecordCount %>
```

Figure 8 shows MyTest.asp at run time.



This page provide information gathered from a Delphi automation object

Object Version: Version 1.0
Number of records in recordset 2
Number of records reported by Delphi Object 2
End of Story

Figure 8: The results of the MyTest.asp page check.

Because you can use JScript to access database sources using ADO, it doesn't make sense to use a Delphi Automation object unless you need to take advantage of Delphi's superior power.

If you need to perform complex queries or calculations from an ASP application, and your project needs access to an ADO data source, a Delphi Automation object is a great way to go.

Conclusion

OLEDB and ADO are the new data access kids on the Microsoft block. While we might not have the easy-to-use, DB-aware controls that we are used to with Delphi BDE databases, ADO is here to stay, and in many cases, it makes sense to familiarize yourself with it.

ASP applications are gaining popularity on Microsoft's Web server platforms. The ability to use the same database object model from your scripting code, and your heavy duty Delphi code, can be yours with Delphi Automation objects that use ADO to access application databases. ∆

The projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\DEC\DI9812RL.

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

By *Warren Rachele*

SysTools 2

A Treasure Chest for Delphi Developers

Have you ever admired the toolbox lugged around by a handyman? You know, the huge wooden box that seems to hold just the right tool to solve any problem. From its inception, the SysTools library from TurboPower Software Co. has represented just that to Delphi developers. If your project needs a low-level routine to solve a programming problem, chances are, it's somewhere in there.

SysTools 2 is the latest release of TurboPower's library of system-level routines and components. This collection of units and classes differs from other offerings by TurboPower in that it covers wide-ranging areas of interest. Other products from the company focus on clearly delineated areas, such as communications or databases. SysTools provides developers with a collection of highly optimized tools that address the low-level functions needed to complete nearly every development effort. The functions save developers from the work of building their own library of tools, allowing them instead to focus their efforts on the functionality of their application.

Why Get It

The initial reaction of a programmer when reviewing the functions provided in the library is often "Why not create this myself?" The answer comes when you take a long look at the effort involved in designing and programming them for your own use. TurboPower provided an excellent example in response to the issue. Consider the tasks involved in changing an integer value to a string that is properly formatted with commas. The integer value must be converted to a string and the length determined. Based on the decimal position, the proper number of commas needs to be determined and inserted. Finally, if the value was below zero, a negative sign must be prepended to the string. Oh, and don't forget that Delphi supports a rich variety of string types, and your function should support all of them.

SysTools provides a ready-made solution to this and numerous other processes similar to it. The equivalent process in SysTools would be:

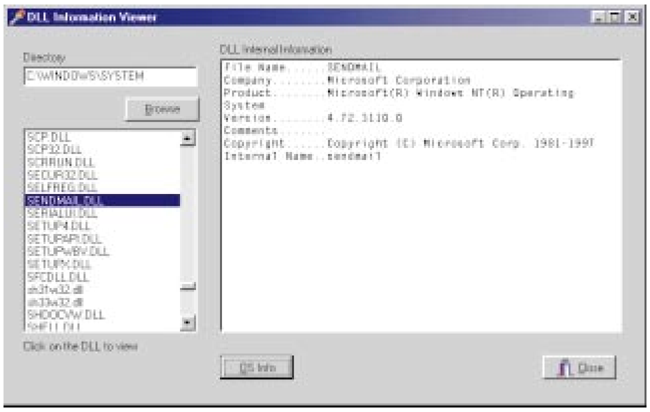
```
StringVariable := Commaize(intVariable);
```

In addition to providing useful utility functions, SysTools provides component interface elements to simplify calling some of the commonly used, but complex, Windows API calls. Using the library functions speeds and simplifies the development process. The functions in SysTools are written in units specifically focused on one area of development; only the units needed to support the function you wish to use need to be included in your program. This prevents the software bloat associated with unneeded features.

The first version of SysTools garnered much praise and was rewarded with industry recognition and awards. The TurboPower developers are not known for resting on their laurels, and went back to work to provide additional functionality to the library. Version 2 finds that completely new units and visual components have been added alongside the improvements made to the original units.

Win32 Shell Tools

The first of the new libraries to be examined is the Win32 GUI routines. These system-



C:\WINDOWS\SYSTEM

SCP.DLL
SCP32.DLL
SCRRUN.DLL
SECUR32.DLL
SELFREG.DLL
SENDMAIL.DLL
SERIALUI.DLL
SETUP4.DLL
SETUPAPI.DLL
SETUPWBV.DLL
SETUPX.DLL
SFCDLL.DLL
sh31w32.dll
sh33w32.dll
SHDOCVW.DLL
SHELL.DLL

```
File Name......SENDMAIL
Company........Microsoft Corporation
Product........Microsoft(R) Windows NT(R) Operating
System
Version........4.72.3110.0
Comments.......
Copyright......Copyright (C) Microsoft Corp. 1981-1997
Internal Name..sendmail
```

Figure 1: The DLL Information Viewer.

level components encapsulate many of the common Win32 API shell routines, hiding the messy details and simplifying their use. The StBrowser component wraps the Windows shell browser. Property changes allow the programmer to determine the amount of detail shown in the browser window, from directries-only to including printers and other desktop items. To rapidly provide the user with information about Windows, the StShellAbout component wraps the Windows shell About dialog box. The user will see the shell About dialog box, which provides version and memory information. The programmer can perform some minimal modification to the dialog box to personalize it for the particular application in which it appears.

The DLL Information Viewer (see Figure 1) puts these components to work alongside the StVersionInfo component, which provides access to the version information stored inside EXE and DLL files. The **Browse** button executes the StBrowser component, allowing the user to select the file's directory. Upon selecting the file in the file listbox, the internal information is gathered by methods of the StVersionInfo component and displayed in the memo component.

Three components wrap the file operations exposed through the API. StFormat works only with removable media disk drives, and displays the Format Drive dialog box on execution. The dialog box allows the user to adjust the format properties before starting the process. StFileOperation is a wrapper for Windows file operations, such as copy, rename, move, and delete. All confirmations are handled automatically, and the component works with individual files or entire directories. The user must provide the source file names and a destination directory, even if it doesn't exist. The operation to be performed is defined internally and is carried out with the appropriate confirmations. The user is treated to the "paper storm" animation during the process.

The StDropFiles component is an interface for adding drag-and-drop capabilities to your application. Adding this component to your application enables you to identify a component that becomes a drop destination. The StDropFiles component populates a string list with the names and paths of the files that were dropped, exposing this list through the *Files* property.

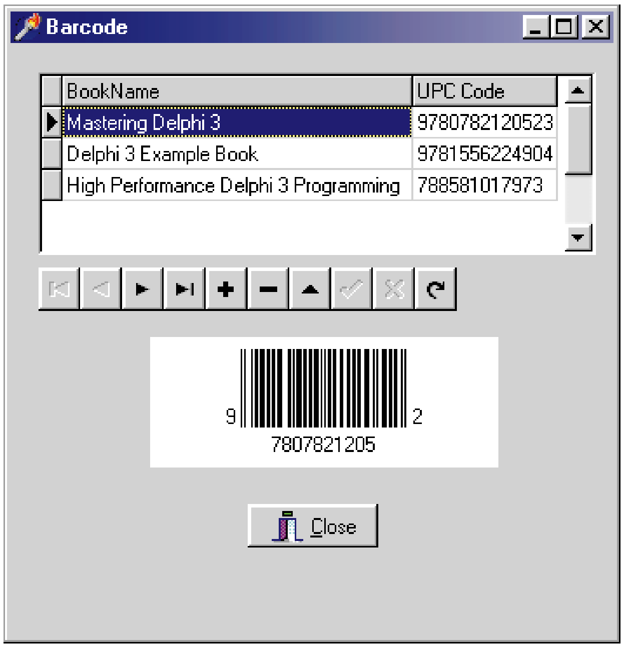
The StShortcut component allows your application to create Windows shortcuts in numerous possible locations. These include the **Desktop**, **Start**, or **Programs** menus, or any one of several other locations in the Windows system. At a minimum, the developer need only set the target file name and a destination for the shortcut, and call the *CreateShortcut* method to use the component. Other options are available, including a method for resolving shortcuts by returning the name, working directory, etc.

The last Win32 component is the StTrayIcon. This tool simplifies the process of creating programs that can reside in the tray portion of the Taskbar. Depending on the combination of property choices the developer selects, a variety of actions can be set for such actions as minimization and a click on the **Close** button.

Another visual component, apart from the Win32 calls, is the Bar Code Builder. It generates, displays, and prints bar codes in numerous industry-standard formats. The component comes in two forms: a static model that handles a single code, and a data-aware version that can be linked to a DataSource component. A demonstration program using the data-aware version, StDBBarCode, is shown in Figure 2. The linkage of the bar code to the data field UPC code is automatic and transparent; as the DBNavigator is used to step through the records in the table, the bar code automatically reflects the changes in the field, refreshing the component display with each change.

Real Business Financial/Statistics Library

New to version 2 is a pair of units that support business finance and statistics functions. The functions do not replace



BookName	UPC Code
Mastering Delphi 3	9780782120523
Delphi 3 Example Book	9781556224904
High Performance Delphi 3 Programming	788581017973

Figure 2: A demonstration program using StDBBarCode.

the Math unit included with Delphi; rather, they offer additional functionality for your programs. The TurboPower designers made an excellent implementation decision by basing them on similar functions in Microsoft Excel, a program that many computer users are familiar with. The excellent manual makes the most of this similarity by noting the Excel function that is closest in functionality to the SysTools function in the documentation.

The StFin unit provides a wide-ranging collection of finance-related routines. The unit contains tools for working with interest calculations, depreciation, the time value of money, various bond-related calculations, and others. These functions again point to the value of this package in terms of the time and effort saved through their use. Think about all the values necessary for calculating accrued interest. You would need the issue date, date of maturity, annual rate, a par value, and the interest payment convention to be used in the calculation. Each of these elements must be correctly typed and verified for reasonableness before use. For example, the maturity date cannot be less than the issue, and the interest rate cannot be zero or less. This relatively simple calculation could easily consume a great deal of your time. SysTools handles all data verification in their highly optimized code, allowing you to concentrate on producing results.

The StStat unit supplements the Delphi Math unit with 45 additional statistical functions. They cover three general categories of statistics: general statistical measures, linear regression modeling, and probability distribution. Again, unless you are intimately familiar with the mathematics behind the statistical calculations, it would be easy to become mired in the creation of these tools, especially from an accuracy standpoint. In addition, because statistical functions work with large bodies of data, creating your own functions would require huge amounts of data and memory management.

Closely related to the finance and statistics functions is StExpr, a visual component that implements a drop-in Expression Editor. This unit also supports a vast array of mathematical functions in 16- and 32-bit versions. The expression grammar acceptable to the parser closely follows Pascal rules with regard to precedence and structure, so adding this functionality into a program should be relatively easy. While true of all TurboPower products, this component helps point to a unique advantage of the libraries created by this company. Because the source code for all the libraries ships with the package and is clearly commented, the programmer would be able to quickly implement special-use functions by extending this unit. The same cannot be said of binary libraries.

CRC Routines

Cyclic Redundancy Check (CRC) routines give programmers a tool for detecting errors in arbitrary blocks of data or files. The need for verification of data increases each day, with data streams passing through conditions that cannot guarantee the integrity of the data from one point to another. SysTools contains a unit — StCRC — that provides programmers with multiple CRC routines — two 16-bit, two 32-bit, and the Internet Checksum. Whether called in primitive form (low-level) or with a single parameter passed (which then calls the primitive functions), the return value of the CRC allows the programmer to implement a verification routine based on that value.

Internet Data Conversion Kit

Another unit new to version 2 is the Internet Data Conversion kit. The StMime unit encapsulates classes that perform conversion functions on common data formats used to send binary data across the Internet. The unit handles the most common formats: Raw, Quoted-Printable, UUEncode, Base64, and BinHex. If additional formats are required for your program, the unit is extensible.

Updated Libraries

Nearly every unit in the SysTools library received upgrades on some level. The rich set of string functions remains attractive to programmers of all levels. Delphi supports three flavors of strings: the standard Pascal-length byte string, the default ANSI string, and the Wide string. The SysTools library supports all three, providing the identical function for each, changing the parameters and return types as appropriate.

The Date and Time unit is equally powerful, addressing the mathematics of time and date differences, as well as providing compact storage of the data. The unit gives equally powerful conversion and formatting routines to the programmer. SysTools also continues to include the BCD (Binary Coded Decimal) unit for high-performance, floating-point mathematics. This method of data storage provides accuracy to 18 significant digits.

The StUtils unit is a set of operating system utilities that provide access to Windows 95/98, DOS, Windows 3.1, and Windows NT. The routines in this unit supplement the tools provided with Delphi in the SysUtils unit and focus on lower-level tasks. Coupled with the operating system utilities is StText, a unit that brings the functions associated with files viewed as a collection of bytes to text files. For example, the routines allow for determining the size of a text file, or returning the file pointer position. A separate class wraps the API calls necessary to work with INI files or the registry.

Container classes are routines that support the methods and data that support programming constructs such as linked lists, queues, binary trees, and hash tables. SysTools gives the developer ready-made structures that can be quickly added to any application. The SysTools implementations can handle data structures as large as the user's hard drive, or the core memory contained in the machine. New to version 2 is the priority queue class. A priority queue is a data structure that allows the program to add new items to the structure and remove the highest valued item. The

INFORMANT FACT FILE

The first version of SysTools garnered much praise and recognition, and this latest release only improves and extends the practicality of version 1. SysTools provides a collection of highly optimized tools that address the low-level functions needed to complete most development efforts. The functions save you the work of building your own library of tools, allowing you to focus your efforts on the functionality of your application.

TurboPower Software Co.
PO Box 49009
Colorado Springs, CO 80949-9009

Phone: (800) 333-4160
Fax: (719) 260-7151
E-Mail: order@turbopower.com
Web Site: http://www.turbopower.com
Price: US$199

structure is commonly used in queuing systems where highest priority items are given a higher value. When the system is looking for tasks to process, it pulls the highest value from the structure. SysTools implements the structure as a double-ended heap (deap), giving additional functionality by drawing the priority value from the top or bottom of the heap.

To simplify the use of these complex structures and the functions pointers needed to support such large data collections, SysTools includes a set of non-visual container class components. Dropping the components into your application brings with it all the supporting code necessary with a minimal increase in overall code size. The components also provide simplified functional access through the methods included with each container. A separate class, but one that is functionally closely linked to the container classes, is also included. This is a high-performance sort engine capable of sorting up to 2 billion elements, which can be anything from discrete values, to records or arrays.

Finally, the SysTools library provides a class that supports astronomical routines. To understand their implementation, it's assumed that you are familiar with concepts such as astronomical coordinate systems and Universal Time.

The Product

SysTools 2 ships on a CD-ROM (along with other TurboPower products). The accompanying 606-page printed manual is comprehensive and well written. It provides examples where needed, including full programs demonstrating the use of non-intuitive classes. Several example programs that also demonstrate the component or class use are installed. TurboPower also allows you to perform evaluation installations of their other products from the CD-ROM that work only within the development environment. This gives you the opportunity to evaluate other products at your leisure.

The SysTools package contains complete source for all classes in the library. The code is very well written and documented, thus providing the additional benefit of being an excellent learning tool. The package is royalty-free; you only pay for the library one time. All versions of Delphi, as well as C++Builder, are supported in a single package, so there's no need to buy separate 16- and 32-bit libraries.

Conclusion

As with all TurboPower products, the SysTools library is well executed and implemented. The wide range of functionality is sure to provide the answer to many programming situations you'll face. The functions are rock-solid, gracefully handling all exceptions caused during testing. Using the routines supplied by the package will save the typical programmer immense amounts of development that can be directly applied to the high-level functionality of the program. Regardless of their focus, Delphi developers should have this library in their toolbox. ∆

Warren Rachele is Chief Architect of The Hunter Group, an Evergreen, CO software development company specializing in database-management software. The company has served its customers since 1987. Warren also teaches programming, hardware architecture, and database management at the college level. He can be reached by e-mail at wrachele@earthlink.net, or by telephone at (303) 674-8095.

table of CONTENTS

By Bill Todd

InfoPower 4.0

Woll2Woll's Must-have Add-in Gets Better

InfoPower 4.0 adds a host of new features to an already outstanding set of Delphi database tools. For example, two new calendar controls, wwDBDateTimePicker and wwDBMonthCalendar, resemble their non-data-aware ancestors from the Win32 page of the Delphi Component palette, but feature many additional capabilities.

The wwDBDateTimePicker component can be embedded in the InfoPower wwDBGrid component (see Figure 1), and is also used for date fields in the wwRecordView and wwFilterDialog components. It can also be used without a DataSource and allows entry of date and time in a single control. When entering a date, the cursor automatically advances when enough characters are entered; today's date can be entered by tapping the space bar. You can display dates in your choice of formats using the *DisplayFormat* property, or the system long or short date formats.

The wwDBDateTimePicker component also includes an *Epoch* property, which controls how dates entered with a two-digit year will be treated. The default *Epoch* value is 1950, so all dates with a year less than 50 will be assigned to the twenty-first century. There are also many options for controlling the information displayed in the calendar. For example, the calendar in Figure 1 has the first day of the week set to Thursday, and the week number displayed to the left of each week. To control which days are shown in bold, simply create an event handler for the *OnCalcBoldDay* event. Setting the *MultiSelect* property to True lets users select a range of dates using the keyboard or mouse. The wwDBMonthCalendar component lets you drop a calendar on a form, so the entire calendar is always in view. It includes all the features of wwDBDateTimePicker, plus the ability to show multiple months simultaneously. Figure 2 shows wwDBMonthCalendar on a form displaying six months with a range of dates selected.

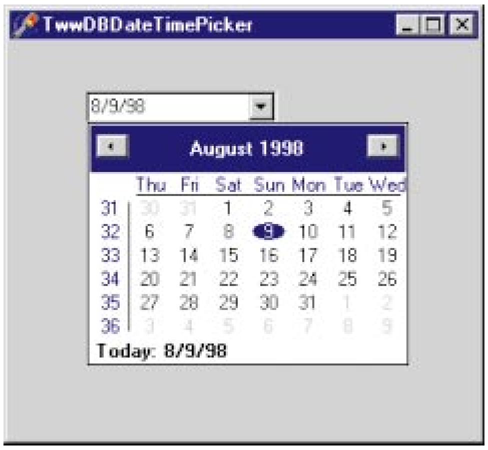


Figure 1: The InfoPower wwDBDateTimePicker component with week number displayed.

wwDBRichEdit Enhancements

The wwDBRichEdit control, introduced in version 3.0, has also been enhanced with new features, including bitmap and OLE support. Bitmaps and OLE objects can now be embedded into your rich text documents (RTFs) and saved in your database. URLs are also supported in two ways. Any URL in the document will appear underlined, and double-clicking the URL will start your Web browser and open the link.

There is also a new pop-up paragraph ruler to quickly set margins and indentations in your RTF documents. Multi-level undo and redo has also been added to let you easily undo or redo a series of actions. User-definable speed buttons let you add custom functions, such as saving the document to a file or calling a spelling checker. Also, InfoPower's search and filter components will now work on the contents of RTF fields.

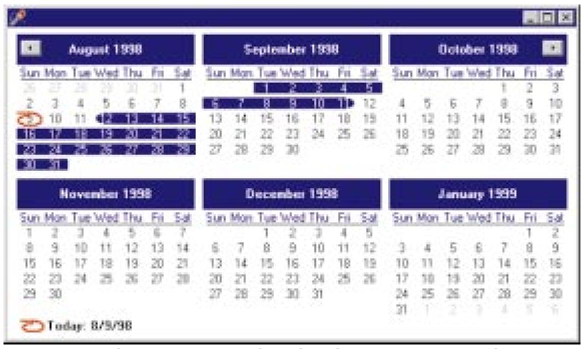
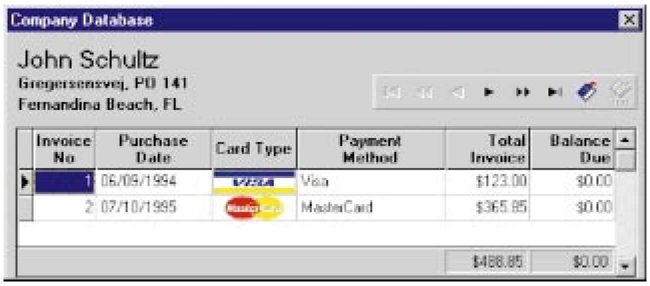


Figure 2: The wwDBMonthCalendar component showing six months and a selected date range.



Company Database

John Schultz
Gregersensvej, PO 141
Fernandina Beach, FL

Invoice No	Purchase Date	Card Type	Payment Method	Total Invoice	Balance Due
1	06/08/1994		Visa	$123.00	$0.00
2	07/10/1995		MasterCard	$365.85	$0.00
				$488.85	$0.00

Figure 3: The InfoPower wwDBGrid component with bitmaps and footer cells.

The wwDBRichEdit component includes an outstanding RTF word processor. Users simply right-click the rich edit component and choose **Edit** to display a full WordPad-like RTF word processor. Users are able to: set font styles and sizes, italics, colors, paragraph alignment, margins, indents and spacing, any number of tabs and their positions, and bold and underline; create bulleted lists; and embed graphics or OLE objects. Printer-support options let users set the paper size, orientation, and printer margins.

New Grid Features

One of the jewels of InfoPower since version 1.0 has been its powerful data-aware grid component, wwDBGrid, which features many enhancements in version 4. You can now display a cell in the grid as a normal edit box, checkbox, combo box, spin edit, date time picker, bitmap, lookup combo box, or a custom edit box. Footer cells can be added to the bottom of the grid to provide a convenient place to display column summary information. Footer cells do not scroll with the grid, so the information they contain is always visible. Because you control which columns have footer cells, they only appear for those columns that have summary information to display.

The ImageList component is now supported as a source for bitmaps to display in column headings or in grid cells. Perhaps the most important new features for developers are the additional events; *OnCellChanged*, *OnRowChanged*, *OnDrawTitleCell*, and *OnDrawFooterCell* make it easier than ever to customize the appearance and behavior of the grid. Figure 3 shows an InfoPower grid that displays bitmaps in cells and footer cells to show column totals.

InfoPower's grid continues to support many other great features not found in the standard Delphi grid, such as scalable row heights with word-wrap within cells, rich text display within the grid, and InfoPower's picture masks to control editing within a cell. The column headers can serve as buttons and will depress when clicked. By creating an *OnTitleButtonClicked* event handler, you can perform any function you wish when the user clicks a column title. You also have complete control over the appearance of column titles. You can justify the text left, right, or center, set the number of lines, and control the colors.

The wwDBGrid component also supports smart key mapping. By changing a single property, you can cause the grid to treat Enter↵ as Tab↹. Properties also let you control whether Ctrl Delete deletes a record, and Ins inserts a record. The grid also lets you edit calculated and lookup fields with just a few lines of code. Another particularly handy feature is the ability to embed a speed button at the top of the grid's indicator column. This is particularly useful for calling the InfoPower wwRecordViewDialog component.

The wwRecordViewDialog component automatically generates a form that contains appropriate field objects to view a single record in a dataset. It's particularly useful in conjuntion with a grid to give users an easy way to display and edit all the fields of the current record. For example, using the InfoPower grid, Lookup combo box, and Record View dialog box, it's easy to build a generic lookup table editing form that allows the user to select a lookup table from the combo box, view, and edit it in the grid, or click a button and pop up the Record View dialog box showing the current record.

New in version 4 is the wwRecordViewPanel component. It provides the same features as wwRecordViewDialog, but can be dropped onto your form to add dynamic generation of single record editing controls.

A Better Navigator

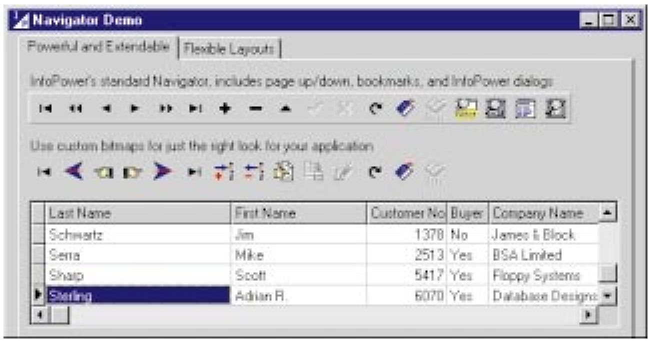
Another great new component in InfoPower 4.0 is defined by wwDBNavigator. The InfoPower navigator supports adding buttons and custom icons to the standard DBNavigator set. It even includes icons for common functions, such as setting a bookmark, going to a bookmark, InfoPower dialog boxes, and page up/page down. Figure 4 shows one of the forms from the InfoPower demonstration program with all these buttons and icons added to the navigator.

INFORMANT FACT FILE

InfoPower 4.0 adds a host of features to an already outstanding set of Delphi database tools. InfoPower components make any database application — desktop, client/server, or multi-tier — more powerful and easier to develop. InfoPower 4.0 Professional supports Delphi 3 and 4. The professional version also includes C++Builder 3 support, complete source code for all InfoPower components, and InfoPower 3 for compatibility with Delphi 1 and 2.

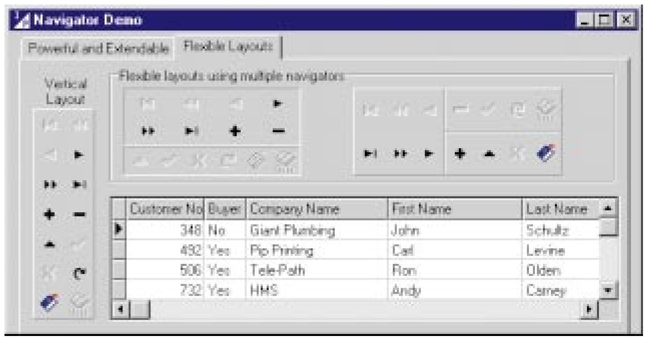
Woll2Woll Software
2217 Rhone Drive
Livermore, CA 94550

Voice: (925) 371-1663
Fax: (925) 371-1664
Web Site: http://www.woll2woll.com
E-Mail: sales@woll2woll.com
Price: InfoPower 4.0 Professional, US$299; upgrade, US$129; InfoPower 4.0 Standard for Delphi 3 and 4, US$199; upgrade, US$99.


Navigator Demo
Powerful and Extendable
Flexible Layouts
InfoPower's standard Navigator, includes page up/down, bookmarks, and InfoPower dialogs
Use custom bitmaps for just the right look for your application

Last Name	First Name	Customer No	Buyer	Company Name
Schwartz	Jim	1378	No	James & Block
Serra	Mike	2513	Yes	BSA Limited
Sharp	Scott	5417	Yes	Floppy Systems
Sterling	Adrian R.	6070	Yes	Database Designs

Figure 4: The InfoPower wwDBNavigator component with added buttons and icons.



Customer No	Buyer	Company Name	First Name	Last Name
348	No	Giant Plumbing	John	Schultz
492	Yes	Pip Printing	Carl	Levine
506	Yes	Tele-Path	Ron	Olden
732	Yes	HMS	Andy	Carney

Figure 5: Various wwDBNavigator layouts.

Adding buttons to the navigator not only lets you access other InfoPower features, but call your own code as well.

You can also arrange the navigator to suit your form geometry by displaying it horizontally, vertically, or with multiple rows. Figure 5 shows several possible navigator layouts. Adding buttons to the navigator is a snap using the InfoPower collection editor; select the navigator's *Buttons* property in the Object Inspector, click its ellipsis button and the collection editor appears. You can add and delete buttons, and select individual ones to set their properties in the Object Inspector. The collection editor also lets you rearrange the buttons within the navigator.

Finding Data

InfoPower provides a complete suite of controls for finding data, starting with the wwFilterDialog component (see Figure 6). wwFilterDialog works with tables, queries, and ClientDataSet events to let users filter their view of data. When filtering a query, you also have the option to let the database server perform the filter instead of fetching all the data and performing the filter on the user's workstations. The filter can encompass any number of columns and supports matching on single values or a range of values.

Filters on columns can be logically connected using AND or OR. When searching for a single value in a column, you can specify an exact match, starts with, or search for the value anywhere in the columns text. You can also control if the match is case-sensitive, and you can give the user the ability to logically invert the search and see all the records that don't match the specified criteria. Users can even filter on lookup fields.

Other search components include wwLocateDialog, wwSearchDialog, and wwIncrementalSearch. wwLocateDialog lets a user easily search for a value in a column using exact match, starts with, or is contained in options. The user can also control the case sensitivity of the search and use wild cards in the search string. The Find First and Find Next buttons let a user easily step through the records that match the search criteria.

The wwIncrementalSearch component looks like an edit box, but provides incremental searching on a single field in a dataset. As the user types each succeeding letter, the dataset is dynamically repositioned to the first records that match the characters currently typed. The wwSearchDialog component also provides incremental searching, but in a dialog box that displays the dataset being searched in a grid. The user can choose which field to search, and the dialog box can contain a user-defined button that lets you add custom features.

Another related component is wwKeyCombo. This control has been enhanced in InfoPower 4.0 to allow users to incrementally search on, or sort by, any column in a ClientDataSet.

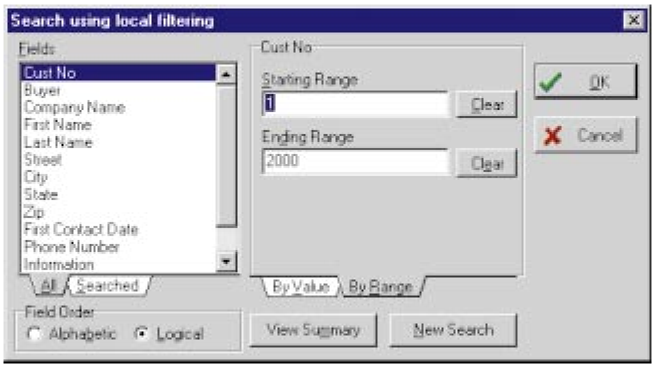
Other Features

InfoPower also includes the wwTable, wwQuery, wwQBE, and wwClientDataSet components, which add powerful filtering capabilities. The wwQBE component is particularly handy for applications that use local Paradox or dBASE tables, since QBE queries frequently execute faster than SQL queries.

All InfoPower components also support InfoPower's picture masks. Picture masks are what Delphi's edit masks should have been — a very powerful template system for controlling what a user can enter into a field and how it's displayed.

InfoPower didn't forget the international developer. The wwIntl component contains all the strings used by the other components and provides an easy, centralized location to translate the InfoPower component suite into another language.

InfoPower has always been known for its excellent documentation, and version 4 continues that tradition. In addition to complete online help, version 4 comes with a 258-page



Starting Range: 1
Ending Range: 2000
Field Order: Alphabetic [✓ Logical]

Figure 6: The wwFilterDialog component in action.

manual. All the properties, methods, and events are clearly described, and code samples are provided where necessary.

Conclusion

If I could only have one Delphi add-in, it would be InfoPower. The InfoPower components make any database application more powerful and easier to develop — whether it uses local tables, or is a client/server or multi-tier application. I haven't seen any other tool set that allows you to give users more features with less effort. With version 4, InfoPower retains its well-deserved position as the "must have" Delphi add-in. ∆

Bill Todd is president of The Database Group, Inc., a database consulting and development firm based near Phoenix. A Contributing Editor to *Delphi Informant,* he is also co-author of four database programming books, the author of over 60 articles, and a member of Team Borland, providing technical support on the Inprise Internet newsgroups. Bill is a frequent speaker at Inprise conferences in the US and Europe. He is also a nationally known trainer and has taught Paradox and Delphi programming classes across the country and overseas. He was an instructor on the 1995, 1996, and 1997 Borland/Softbite Delphi World Tours. He can be reached at bill@dbginc.com or (602) 802-0178.

SAMS Teach Yourself Borland Delphi 4 in 21 Days

SAMS Teach Yourself Borland Delphi 4 in 21 Days applies the same concept as the rest of SAMS' "Teach Yourself" series. A set of daily lessons will transform you into a <insert tool> programmer in *x* number of days. From a pure language standpoint, this might be a reasonable expectation, but Delphi isn't just an Object Pascal implementation. It's a complete visual development environment, with scores of components, a complete database, etc. It's just not fair to expect a reader to gain even a reasonable level of competence with such a product after 21 daily lessons.

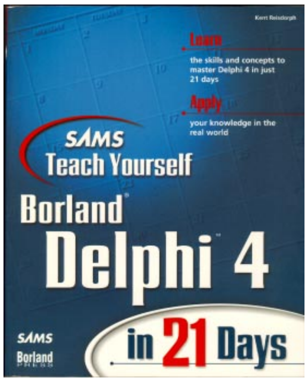
The most positive aspect of this book is that it provides a thorough survey of the Delphi development environment. The author, Kent Reisdorph, passes up no opportunity to show what Delphi can do, providing coverage to such wide ranging topics as an introduction to the IDE, graphics and multimedia programming, COM and ActiveX controls, and a compare-and-contrast piece on Delphi and C++Builder. This book is an excellent choice for the programmer who is trying to determine if Delphi is the appropriate tool for a project. There is just enough material here to give one an adequate taste of the capabilities of the tool.

There are some excellent chapters, with the best concentrated in the week three segment. Day 19, for example, focuses on "Creating and Using DLLs," and the author does an excellent job of explaining these executables to the reader. He defines the term and explains why it should be considered over other options. The topic is demonstrated with a simple, yet effective project that gives the reader a good look at the process of loading and calling a function from within a DLL. He also does a good job of explaining the differences in requirements for using static vs. dynamic loading.

Another excellent chapter delves into COM and ActiveX, and would make a good stand-alone article on the topic. Reisdorph's writing brings the reader along slowly to a point at which, using the Delphi Wizards and IDE tools, the developer has a fundamental understanding of these objects. Two basic demonstration projects are included with the chapter.

The third gem in the final week's section deals with component creation. While experienced developers know this is no trivial topic, the author shines by making a complex subject accessible. A simple control is derived that points out most of the basic pieces necessary to build a component. The reader may still not be prepared to develop a library of commercial class components, but some ideas for expanding the stock VCL components should come across.

A project called Scratch Pad, a simple editor, was started in the week one section. This project is used throughout the book to demonstrate different techniques used in building a complete application. So many books push the reader from one demonstration project to the next. By sticking with a single project that grows to include newly developed skills, the reader gets a much better sense of the program construction practice.


Kent Reisdorph
Learn
the skills and concepts to master Delphi 4 in just 21 days
Apply
your knowledge in the real world
SAMS
Teach Yourself
Borland
Delphi 4
in 21 Days
SAMS
Borland

Two chapters demonstrate the strength of Delphi quite well. The first is a chapter that delves into the debugging facilities of the IDE, a topic that usually receives short shrift. The author guides us through the standard break-point debugging process and points out the useful features of ToolTip Expression Evaluation. In addition, he provides some hints for resolving common application errors, some of which may have driven the developer to the debugger in the first place.

Another outstanding entry is the chapter dealing with the variety of tools and options provided with Delphi. This chapter covers such wide-ranging topics as using WinSight and the Image Editor. It also discusses the

Windows Messaging System, the Package Collection Editor, and various Delphi environment options. This might seem like a lot of ground to cover in a single chapter, but the author carefully sticks to high-level descriptions of the tools. There is just enough here to make the developer aware of the tool's availability and its usage, while in-depth explanations are left to the product documentation.

Unfortunately, the reader may never arrive at these chapters, having given up trying to wade through the initial pages of the book. In the first 126 pages of this 918-page book, the author attempts to cover the entire breadth of Object Pascal, control structures, classes, and object-oriented programming. Entire books have been devoted to these topics, and this presentation just doesn't do them justice. Rather than build a project that adds incrementally with each new concept, these topics are presented in snippet form, making it difficult to associate the items with one another.

These opening chapters are also rife with forward referencing. Reisdorph references a topic, usually more advanced than what belongs in the current context, then states that it will be covered in later chapters. This makes for a frustrating read and, at the pace of the chapters, many readers will give up. The presentation order of the material could also use a bit of the editor's pen. A beginning programmer, this book's intended audience, should not be introduced to things such as pointers or the heap until they have a better concept of the program execution process. The first couple of chapters leave the reader with the impression of a book that is haphazard and uneven. They are rewarded for persevering and getting to the later chapters, but some may surrender.

Overall, *SAMS Teach Yourself Borland Delphi 4 in 21 Days* is a good introductory text to the Delphi development tool. Version 4 features are well covered and a programmer spending some time going through the materials and reviewing the excellent quiz and exercise sections provided at the end of each chapter will come away with a good understanding of the capabilities of the environment.

— *Warren Rachele*

SAMS Teach Yourself Borland Delphi 4 in 21 Days by Kent Reisdorph, SAMS Publishing, 201 West 103rd St., Indianapolis, IN 46290, http://www.samspublishing.com.

ISBN: 0-672-31286-7
Price: US$39.99 (918 pages)

Delphi 4 Developer's Guide

I'm sure many of you have read one of the earlier editions of this popular guide, either for Delphi 1 or Delphi 2 (no Delphi 3 edition was published). One of the factors that puts this book in a special category is the background of the authors, Steve Teixeira and Xavier Pacheco. Both have worked for Inprise (Teixeira still does as a Research and Development engineer); they are thus able to bring the insightful perspective of the "insider" to this work. This quality is particularly evident in the plethora of valuable tips. As we'll see, however, the value of this book goes much further than its helpful suggestions and exposure of hidden traps.

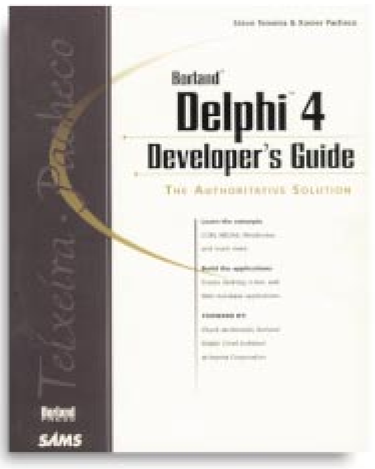
Developer's Guide is divided into five parts. The first, "Essentials for Rapid Development," is an introduction to Windows programming with Delphi 4. I found the chapter on Object Pascal particularly valuable. For developers moving to Delphi from Visual Basic, C++, or another language, it provides an excellent introduction to Delphi's underlying language. More importantly, for experienced Delphi developers it provides a valuable overview of the extensions to the language introduced in Delphi 4: the new types, dynamic arrays, and function overloading. The chapter on "Application Frameworks and Design Concepts" gives novice Windows programmers a succinct overview of essential techniques and conventions.

Part Two, "Advanced Techniques," includes chapters on graphics programming, printing, working with files, and MDI applications, all in Adobe Acrobat format on the CD-ROM that accompanies the book. The obligatory chapters on dynamic link libraries and multithreading are among the best I have seen — and I've seen some excellent ones. The tips and insider information in these chapters will be of value, even to advanced Delphi developers. In this part, I was particularly impressed with the chapters entitled "Hard-Core Techniques" and "Snooping System Information," which broached such topics as using hook functions, incorporating assembler code, and obtaining system information.

Part Three provides an overview of the Visual Component Library and covers many component-writing topics, including writing component editors, working with packages, and using the *TCollection* class in building a component. I've been waiting for someone to tackle this latter topic in this way for a long time. Bravo! If you are interested in working with OLE, COM, or ActiveX, you should find the chapter "COM and ActiveX" to be an excellent introduction. This is followed by a 70-page demonstration showing how to build an ActiveX control.

Part Four, which deals with database programming, follows the pattern established in the earlier sections, going well beyond the usual topics. While it includes the expected topics of working with *TTable*, *TQuery*, and *TStoredProc*, it includes exciting excursions into client/server programming and Internet database issues. The chapter on "Extending Database VCL" is particularly exciting. Here, the authors provide advice on working with the BDE, issues related to Paradox and dBASE tables, and writing data-aware controls. Overall, I haven't seen a better introduction to Delphi database development in a general Delphi work.

Part Five, "Rapid Database Application Development," goes further into the realm of Delphi database development. Among other things, it provides a solid model of building a client/server application and a fascinating desktop application. The latter, a "Bug Reporting Tool," while not necessarily including



all the functionality you might want (the authors point this out), provides an excellent example of how to build a database application that can be deployed on the Web with a minimum of problems.

These days I am more interested in specialized Delphi books rather than general encyclopedias. This book, however, is an exception. Its more than 1,000 pages of text (with 500 more in additional chapters on the CD-ROM) provide a wealth of information from two leading Delphi experts. If any of the topics I have outlined above are on your list of "new Delphi directions," I suggest you give this wonderful treatise serious consideration. *Delphi 4 Developer's Guide* should become an important part of many Delphi libraries.

— *Alan C. Moore, Ph.D.*

Delphi 4 Developer's Guide by Steve Teixeira and Xavier Pacheco, SAMS, 201 West 103rd Street, Indianapolis, IN 46290, http://www.samspublishing.com.

ISBN: 0-672-31284-0
Price: US$59.99
(1,185 pages, CD-ROM)

What Does Vendor Certification Really Mean?

That's the question being asked more and more as the marketplace becomes inundated with vendor certification programs. CNE (Certified Novell Engineer), MCSD (Microsoft Certified Systems Developer), SCJP (Sun Certified Java Programmer), OCP (Oracle Certified Professional), SCP (Sybase Certified Professional), and BCD (Borland Certified Developer) are just a few of the acronyms professional programmers can add to the end of their names.

But what do they mean? A cynic believes certification is just another profit center for vendors. The naïf accepts vendor certification as a reliable indicator that certified persons excel at the subject for which they are certified. Realistically, the answer lies somewhere in between.

As someone who's gone down the certification road several times, I believe you need to examine your motivations for being certified. If it's simply for the money, you will be sorely disappointed; the time spent pursuing certification is rarely fully compensated. And depending on what certification you want, there may be a considerable number of people already certified in that subject area.

So why go through with it? You may want to consider certification as a way to keep pace with your technological peers. I suppose that's a fancy way to say "If everyone jumped off a bridge, would you?" In this case, jumping off the bridge isn't self-destructive; it's something you do just to keep up.

Becoming certified means you have a certain amount of product knowledge. It *does not* necessarily make you a better programmer. It *does* demonstrate motivation, initiative, and perseverance — all good traits for a programmer to possess. Once you've achieved certification, you can congratulate yourself on a job well done — many people fall by the wayside during the certification process.

Applicants must have ... However, certification is not the sole barometer of your desirability to an employer. As an employer, it's probably best to view a certification as a bonus attribute of a candidate, rather than as a necessity. For one, many exam questions are poorly written. Microsoft especially is infamous for questions like this:

Given the following solution/code, how well does it work?
A) Perfect
B) Pretty good
C) Somewhat good
D) Not so good
E) It doesn't work

Questions such as these are difficult to answer "correctly."

There are other reasons why certification isn't a reliable indicator of a programmer's skills. Some people simply don't take exams well. Also, I don't know of a single certification program where you're "on your own," i.e. without the ability to consult manuals, examples, existing code, or other people while taking the test. Therefore, some exams only test a person's ability to gather facts quickly and accurately. Which isn't a bad characteristic for a programmer to have. Perhaps that's why the certification programs are designed that way (although I doubt it). In today's fast-paced world of evolving technology, it's much better to be able to adapt and know how and where to find answers.

Quality software development requires more than product knowledge, however. Design, planning, and implementation acumen are also important, and most certification exams do not test for such skills. For example, none of the tests I've seen pose the following scenario: "Here's a specification. Please write an application that meets these requirements in a timely manner. For extra credit, design the application to allow easy maintenance and maximum flexibility for enhancements."

It's not enough. Being certified is rarely enough; other factors, such as college degrees, and — especially — experience must enter into the equation when evaluating a candidate. Some of the best developers have never been certified; they're too busy delivering applications.

All this said, I think certification is valuable. The Delphi certification — Inprise currently offers the "Delphi 3 Client/Server Suite Certification Exam" — has been a historically difficult exam, but that makes the accomplishment that much more worthwhile and valuable. You can find out more about the Inprise certification options for Delphi and JBuilder at http://www.inprise.com/programs/certify. ∆

— Dan Miser

Dan Miser is a consultant who lives in Milwaukee with his wife and daughter. He is active on the newsgroups, where he serves as a member of TeamB. He has been a Borland Certified Delphi Developer since 1996 and recently obtained his MCSD certification. Dan is a frequent contributor to Delphi Informant. *You can contact him at http://www.execpc.com/~dmiser.*

Delphi 3 Book Wrap-Up

Now that Delphi 4 has shipped, it's time to review the outstanding collection of Delphi 3 books. You might be thinking, "With Delphi 4 now available, shouldn't I wait for the new crop of Delphi 4 books?" First, much of the information in the earlier books is relevant to Delphi 4. Plus, you might find some real bargains.

How should we group these works? Some are general, covering a wide variety of topics. These fall into two categories: entry-level and advanced. Others are references that aren't meant to be read cover to cover. Finally, there are focussed works of interest to particular developers.

Let's start with the general books. The entry-level works differ from the more advanced in one key respect: They describe the Delphi IDE and introduce the VCL. Two of the best in this category are *Mastering Delphi 3* [SYBEX, 1997] by Marco Cantù and *Special Edition Using Delphi 3* [QUE, 1997] by Todd Miller, et al. Cantù's *Mastering* is still the best introduction to Delphi. *Using Delphi 3* also has a great deal of merit, with excellent chapters on using threads and working with dynamic link libraries. Both cover essential topics, and provide excellent introductions to working with databases in Delphi.

Three general, advanced books worth considering are *Delphi Developer's Handbook* [SYBEX, 1998] by Marco Cantù, Tim Gooch, and John F. Lam; *High Performance Delphi 3 Programming* — the new incarnation of the *KickAss* series — [Coriolis Group Books, 1997] by Don Taylor, et al.; and *Collaborative Computing with Delphi 3* [Wordware Publishing, 1998] by James Callan. While the latter stresses working with databases and within client/server environments, the other two cover a wide range of topics. If you're a regular reader of *Delphi Informant*, you're no doubt familiar with the fine work that John Penman has done in exploring Winsock. *High Performance* includes some of his early work in this field. My favorite is *Delphi Developer's Handbook*. It explores topics that are seldom broached, and engages in some righteous hacking of the Delphi environment.

Among the specialized Delphi offerings, Ray Konopka's work on writing components has earned a place among Delphi classics. Its successor, *Developing Custom Delphi 3 Components* [Coriolis Group Books, 1997], with its thoughtful treatment of packages and new Delphi 3 features, continues to be the best introduction of this topic. Ray Lischner's *Hidden Paths of Delphi 3* [Informant Press, 1997] is another essential work. I know of no other work (with the possible exception of Lischner's other volume, *Secrets of Delphi 2* [Waite Group Press, 1996]) that digs deeper into Delphi's innards, exposing myriad undocumented features. Finally, *Learn Graphics File Programming with Delphi 3* [Wordware Publishing, 1998] by Derek Benner, explores the graphics file formats available in Windows. Beginning with an excellent introduction to graphics programming and the .BMP format, it explores .TGA, .PCX, .GIF, and several other file formats. With the expanding number of Delphi developers and the sophisticated knowledge base of the established Delphi community, I anticipate an even larger crop of advanced and specialized works in the coming year.

We were also treated to some excellent Delphi 3 references. Of these, *The Tomes of Delphi 3: Win32 Core API* and *The Tomes of Delphi 3: Win32 Graphical API* [Wordware Publishing, 1998] by John Ayers, et al. deserve special mention. The former, which covers the Windows API in great depth, is a milestone in Delphi publishing. The latter, which deals with the Graphical Device Interface, is equally as thorough. I recommend these volumes for any developer working extensively with the Windows API. *Nathan Wallace's Delphi 3 Example Book* [Wordware Publishing, 1998] is another valuable reference. While the *Tomes* pair covers the Windows API, Wallace's volume covers the Delphi VCL in similar detail. All three include excellent code examples.

Before closing, I want to revisit two Delphi 2 classics: Lischner's *Secrets of Delphi 2* and Neil Rubenking's *Delphi Programming Problem Solver* [IDG Books Worldwide, 1997]. *Secrets* continues to rank among my all-time favorites, covering certain topics in greater depth and clarity than I've found anywhere else. Rubenking's work is insightful, and its wealth of useful tips is impressive. Both continue to be quoted and recommended on Internet discussion groups. And both provide useful information on the differences between 16- and 32-bit Delphi programming.

I've not included every Delphi 3 book. Some I would not recommend, and there are probably one or two I haven't seen. Most of these have been reviewed in *Delphi Informant*; refer to those reviews for more information. If you're about to shop for Delphi 2 and 3 book bargains, I hope these summaries will be useful.

— Alan C. Moore, Ph.D.

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

Table of Contents