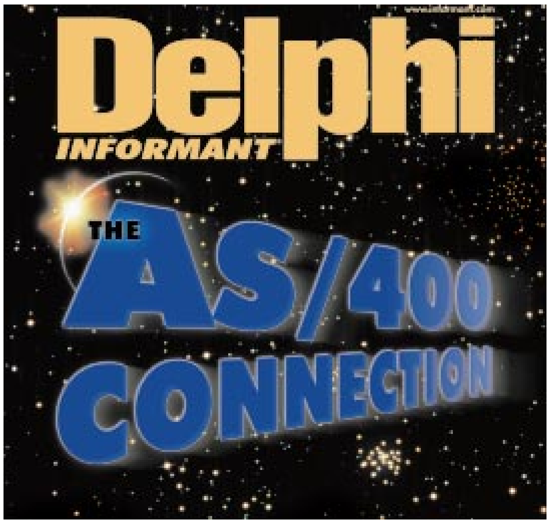
January 1998, Volume 4, Number 1

The AS/400 Connection

Four Methods to "Get There" from Delphi



Cover Art By: *Tom McKeith*

By Zack Urlocker

The New Game

In the seven years that I've been at Borland, the company has gone through more strategies than Internet Explorer has security patches. But there *have* been a few constants: great technologies, great products, and great people. What's been missing is an overall business strategy. Sure, we've had technology vision aplenty, but the technical visions weren't always rooted in the needs of customers. Lets face it, Borland was never known for great management, marketing, and sales. And each one of those is an essential ingredient in running a business, whether it's a technology-driven company or not.

One of the things I focused on in helping define Delphi was the need to talk to customers. As a result, we got the right set of features in, worked some significant compiler magic, and created a tremendously successful product. Some folks say that Delphi saved Borland; I prefer to think of Delphi as helping to define Borland. What Delphi helped define was a process of understanding what customers need, and then implementing it.

Even before the introduction of Delphi 2, it became clear we would have to deliver higher levels of scalability to customers.

Scalability means the ability to deal with more users, more transactions, and higher volumes of data. Delphi had such great performance and productivity that it was being used to create many more business-critical applications than we'd ever expected. Banks, insurance companies, retailers, shipping companies, brokerage houses, telecommunications companies, government departments, and IT shops in every sector of business imaginable were starting to use Delphi for all kinds of business-critical applications. These were not just Windows-client applications, but full-blown client/server applications that would impact hundreds — even thousands — of users.

As fast as Delphi native code was (and even faster in Delphi 2), we knew that fundamental architectural changes would be necessary to achieve the next level of scalability.

Borland acquired some potent technology in 1996; the Entera intelligent middleware from Open Environment Corporation. Entera middleware enabled developers to build systems that could scale to tens of thousands of users.

In fact, because Entera was so open, there were already Delphi customers using it to build large-scale, mission-critical applications. In the development of Delphi 3, and now Delphi Enterprise, we ensured we could create new, more scalable, multi-tier applications.

Why did Borland tackle multi-tier? The same reason we built Delphi: Because customers needed it. At the same time, we kept up a wicked pace of innovation in Delphi 3, adding far more features than any previous version. I found it hard to believe that we could "out ActiveX" Microsoft *and* develop the MIDAS multi-tier technology, but this was one heckuva team. They came through with those major innovations and tons more features — Decision Cube, Code Insight, and many others that I never expected to get. I'm still dazzled by the team that made all this happen on schedule.

Did Borland set out to be a multi-tier company, or a middleware vendor? Hardly. We just continued in the tradition of what made Delphi successful. We listened to customers and addressed their needs in a "Delphi way" that made things easy without applying limits.

So is Borland a tools company or a middleware company? You might as well ask whether Borland is a Windows company or an Internet company. There's no single binary answer. Borland is a company focused on making development easier, no matter the underlying technology.

In the past, middleware has been one of those difficult-to-grasp concepts; but now with the widespread use of the Internet, and companies building their own corporate intranets, developers and IT managers are beginning to understand the need for scalability. After all, what's the sense of building applications for the Internet, if performance is going to hit the wall after a few thousand users?

Borland has expanded its product line to ensure that customers can create these new, complex, Internet and intranet applications. Rather than take a strictly technological viewpoint and call them multi-tier, distributed blah-blah-blahs, or come up with some hokey acronym, we tried to look at it from the customer's point of view. That's why we talk about using Borland tools and intelligent middleware to build Information Nets.

In the months preceding the Borland Developers Conference in July, I spent a lot of time with our CEO, Del Yocam, talking to customers, partners, analysts, and gurus, to understand the problems they faced, and to help us develop the strategy.

Since the conference, I've spent even more time meeting with customers, rolling out the strategy. Case in point: last week was a six-day, six-city tour of Munich, Frankfurt, Moscow, Paris, London, and Amsterdam. We met with over 1,000 customers and partners, and over 100 press, and presented the Information Net strategy more than a dozen times. Apart from the frequent-flyer miles and an 11:00 pm site-seeing tour of Red Square, the highlight of the trip is that people understand the strategy. Developers get it. IT Managers get it. CEOs get it. That's quite a change from the days of having to explain why polymorphism, inheritance, and encapsulation mattered.

Borland is continuing to make a difference in developers' lives. We're focused on more than just Delphi, because you need many different tools and technologies to build Information Network applications. We've introduced a family of products, with recent additions such as JBuilder and IntraBuilder for Web development, and C++Builder as the most advanced C++ environment, bar none. And we'll continue to develop new kinds of technologies to help customers build the new generation of Information Network applications.

Delphi is core to our strategy. We're continuing to look at ways to push the Windows-development front with support for the latest Microsoft technologies like MTS, ATL, IE4, COM+, and whatever other three-letter acronyms come out in 1998.

We're also making sure our tools work together with increased interoperability across clients and servers, so you can combine, say, Delphi client applications with midddle-tier Java code. You'll see an exciting range of products coming from Borland that enable you to develop beyond the boundaries of traditional Windows applications.

With the power of today's tools and the connectivity of the Internet, I can't think of a more exciting time to be a developer.

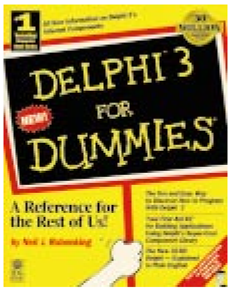
Zack Urlocker is Vice President of Product Management at Borland International. He has been with Borland since before anyone had even heard of the Internet. Out of politeness, his name still appears on the "credits" screen for Delphi.

table of CONTENTS

Delphi
TOOLS

New Products and Solutions

Delphi 3 for Dummies
Neil J. Rubenking
IDG Books Worldwide



ISBN: 0-7645-0179-8
Price: US$24.99 (428 pages)
Phone: (800) 762-2974 or (415) 655-3000

StarBase Adds Delphi Integration to StarTeam and Versions 2.0

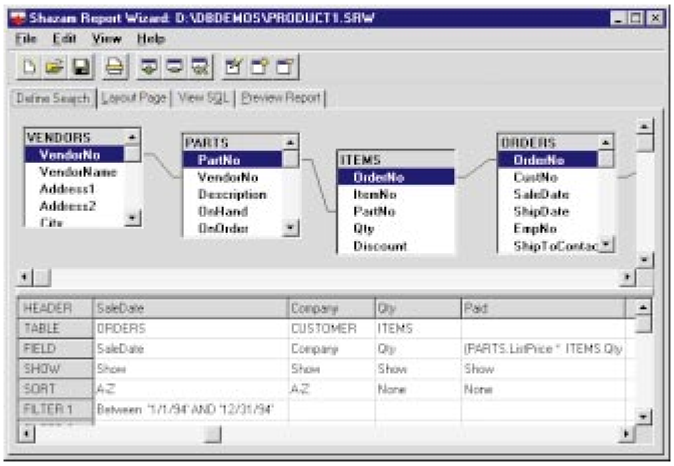
StarBase Corp. has announced Delphi 2 and 3 integration with the release of *StarTeam 2.1* and *Versions 2.0* software configuration management (SCM) products.

Versions 2.0 provides revision control, visual differencing of files, build and milestone management, audit logs, security, and an advanced project repository for development groups. Versions 2.0 also offers a direct upgrade path to the more advanced StarTeam products that add team collaboration, workflow, virtual team facilities with Internet/intranet connectivity, project status reports, and more sophisticated SCM facilities.

Developers can download an evaluation copy, and StarBase users can download an upgrade from the StarBase Web site.

StarBase Corp.
Price: StarTeam 2.1, US$199 per seat (volume pricing available); Versions 2.0, US$99 per seat.
Phone: (888) STAR-700 or (714) 442-4500
Web Site: http://www.starbase.com

Shazam Report Wizard 3.03 Ships



HEADER	SaleDate	Company	Qty	Paid
TABLE	ORDERS	CUSTOMER	ITEMS	
FIELD	SaleDate	Company	Qty	(PARTS.ListPrice * ITEMS.Qty
SHOW	Show	Show	Show	Show
SORT	A-Z	A-Z	None	None
FILTER 1	Between '1/1/94' AND '12/31/94'			

ShazamWare Solutions, Inc. has released *Shazam Report Wizard 3.03*, a native reporting component for Delphi and C++Builder that integrates query-by-example (QBE) and a visual report designer, allowing end users to create ad hoc reports at run time.

The new compression technology doesn't require DLLs, and is said to allow Shazam Report Wizard (SRW) to save reports that are over 85 percent smaller than similar reports created from other Delphi-based reporting components. The resulting files are ready for immediate e-mail or Internet distribution without manual compression.

SRW 3.03 also introduces the new *TSrwView* component that automatically decompresses reports before viewing. These reports are interchangeable among *TSrwView* programs compiled with all versions of Delphi and C++Builder. SRW 3.03 also supports password protection and report-file encryption.

ShazamWare Solutions, Inc.
Price: US$169
Phone: ZAC Catalogs, (800) 463-3574 or (813) 298-1181; Programmer's Paradise, (800) 445-7899.
Web Site: http://www.-shazamware.com

Insession Introduces TransFuse

Insession Inc. has unveiled *TransFuse*, a solution that integrates Web and client/server applications with venerable host transaction processing systems.

TransFuse performs the functionality of transaction processing (TP) monitor access and makes it available to application developers who use Delphi, C++, Visual Basic, Java, PowerBuilder, ActiveX, Lotus Script, CORBA IDL, and others.

On the back end, TransFuse supports IBM's CICS and IMS, Tandem's Pathway, and BEA's Tuxedo, as well as IBM's MQSeries messaging middleware. On the front end, its API supports Microsoft's DCOM and the Object Management Group's CORBA.

For large mainframe data centers where more than one TP monitor is operating, TransFuse also binds application components together into a single transaction that can span multiple TP monitors, or multiple passes to the same TP monitor.

In an *n*-tier architecture, TransFuse works with object request brokers (ORBs) to provide scalability. Insession supports ORBs from Sun Microsystems, Iona Technologies, Visigenic Software, and Expersoft Technologies.

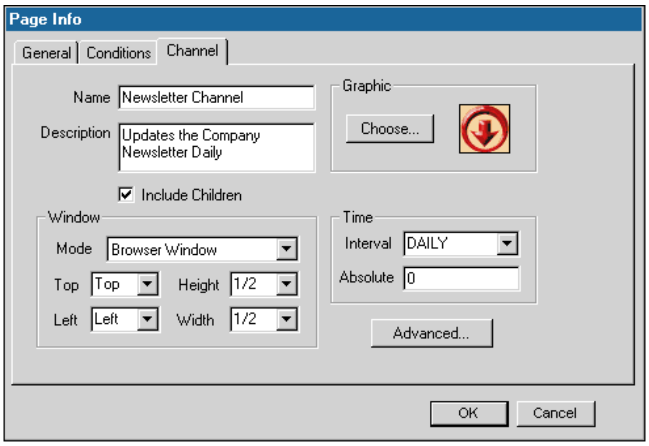
Insession Inc.
Price: US$5,000 for 10 concurrent users.
Phone: (800) 414-7759 or (303) 440-3300
Web Site: http://www.insession.com

Pictorius Announces iNet Developer 3.0 for Windows 95/NT

Pictorius Inc. announced version 3.0 of *Pictorius iNet Developer*, the company's intranet development tool. This update includes support for Microsoft's Component Object Model (COM), Design Time Controls (DTC), Data Binding, and Scriptlets.

By supporting COM, Pictorius iNet Developer 3.0 extends the power of iNet Developer sites using Delphi, Visual J++, Visual Basic, or Visual C++. Users can choose any programming language that supports the creation of COM objects, rather than the integrated programming environment.

DTC support enables users to add a DTC to their Web pages, and have it work with the iNet Developer site. Pictorius iNet Developer 3.0 includes a sample DTC called the Spline Path Control, which lets users define a curved path on a page and select an item (an image or text) to animate along that path each time the page is hit.


Page Info

General | Conditions | Channel

Name: Newsletter Channel
Description: Updates the Company Newsletter Daily
Graphic: Choose...
[✓] Include Children

Window
Mode: Browser Window
Top: Top — Height: 1/2
Left: Left — Width: 1/2

Time
Interval: DAILY
Absolute: 0

Advanced...

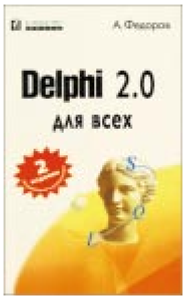
OK | Cancel

Support for data binding gives users the ability to download groups of database records to their browsers, and work with them there without returning to the server. Pictorius iNet Developer 3.0 also includes a client-side caching feature that works with Microsoft Internet Explorer 4.0 and Netscape Communicator, regardless of whether the Web server supports data binding.

Support for Scriptlets enables users to make use of pre-packaged code objects. Scriptlets, written in dynamic HTML code, are wrapped in a re-usable object, and can be called by any COM programming language.

Pictorius Inc.
Price: The commercial version is US$1,495, including the site and page editor, the application server, and the integrated programming environment. A pack of editors is available for US$1,495. Pictorius iNet Developer 2.0 customers can receive a free upgrade. A demonstration version is available for download from the Pictorius Web site.
Phone: (800) 927-4847
Web Site: http://www.pictorius.com

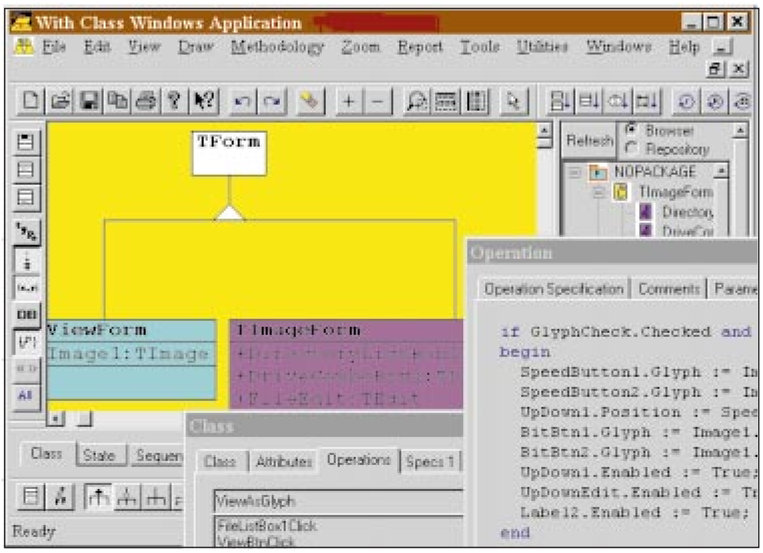
Delphi 2.0 for Everyone, 2nd Edition
Alex Fedorov
ComputerPress Publishing, Moscow



ISBN: 5-89959-029-7
(464 pages, Cyrillic)

MicroGOLD Ships WithClass 97

MicroGOLD Software, Inc. has released *WithClass 97*, a tool that allows developers to reverse-engineer Delphi, Java, C++, relational databases, IDL, and Visual Basic code into graphical designs. This new edition provides support for designing, drawing, and documenting for Unified Modeling Language (UML) 1.0.


With Class Windows Application
File Edit View Draw Methodology Zoom Report Tools Utilities Windows Help
TForm
ViewForm
Image1:TImage
TImageForm
Refresh
Browser
Repository
NOPACKAGE
TImageForm
Operation
Operation Specification | Comments | Parame
Class
Class | State | Sequen
Class | Attributes | Operations | Specs 1
ViewAsGlyph
FileListBox1Click
ViewBtnClick
Ready

```
if GlyphCheck.Checked and
begin
  SpeedButton1.Glyph := In
  SpeedButton2.Glyph := In
  UpDown1.Position := Spee
  BitBtn1.Glyph := Image1.
  BitBtn2.Glyph := Image1.
  UpDown1.Enabled := True;
  UpDownEdit.Enabled := Tr
  Label2.Enabled := True;
end
```

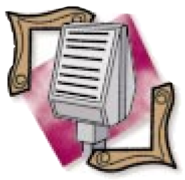
WithClass integrates software configuration management from PVCS, and adds a class repository using Microsoft Access. The tool also provides various methods of presenting final documentation, such as cutting and embedding OLE diagrams with the unique scripting language, using OLE Automation, or integrating with Seagate Crystal Reports.

WithClass 97 includes support for UML, Booch, Rumbaugh, Shlaer/Mellor, and Coad-Yourdon methodologies. The interface includes tabbed dialog boxes, an integrated toolbar, and a simultaneous-tree class browser. Code samples in Delphi, Visual J++, Visual C++, and Visual Basic are provided for generating codes and reports. In addition, support for PowerBuilder, SQL, Smalltalk, Ada, and Eiffel is included.

MicroGOLD Software, Inc.
Price: US$895
Phone: (908) 668-4779
Web Site: http://www.microgold.com

News LINE

January 1998



Point-of-Sale Module Added to Accounting for Delphi

Columbus, OH — ColumbuSoft has added a new point-of-sale module to Accounting for Delphi, a series of general accounting modules sold with optimized source code for Borland's Delphi development platform. The new module includes support for an unlimited number of registers per store, completely mouseless operation, and support for scanners, cash drawers, and receipt printers. In addition, the module contains utility functions that can be used in other applications, such as a reusable report printer with an integrated on-screen preview.

Accounting for Delphi provides general accounting that integrates with custom or vertical market applications. It includes General Ledger, Accounts Payable, Accounts Receivable, Order Entry, Inventory/-Purchasing, Fixed Assets, Payroll, Job Costing, Contact Management, and Point Of Sale modules. Each can be used alone or in combination with others. Purchasers receive a license that allows them to resell their compiled applications an unlimited number of times, with no additional royalties or fees.

For more information, visit the ColumbuSoft Web site at http://www.columbusoft.com.

Radiant Systems Adopts Delphi Client/Server Suite

Scotts Valley, CA — Borland announced that Radiant Systems, a provider of integrated technology solutions, has adopted Delphi Client/Server Suite 3.0. Radiant Systems builds and develops touch screens for companies such as Conoco, Shell Oil, Sony Theaters, Regal Cinemas, and Boston Market Restaurants.

Radiant Systems hopes to utilize the Suite's one-step ActiveX control feature, and support for COM, DCOM, and Decision Cube, to create customized information-network applications, including consumer-activated ordering systems, point-of-sale systems, back-office management systems, and headquarters-management systems. Radiant also plans to use Delphi in future data-warehousing projects.

Learning Tree Offers Object-Oriented Courses

Learning Tree International offers three courses designed to enhance your knowledge of object-oriented technology and applications.

Introduction to Object Technology is a four-day course that builds a foundation in the practices of object-oriented development.

Object-Oriented Analysis and Design offers a practical knowledge of object-oriented analysis and design methods using the UML modeling language.

Hands-On Delphi: Object-Oriented Applications Development is a four-day course that provides practical Delphi skills to develop stand-alone and client/server applications.

For information, call Learning Tree International at (800) THE-TREE, or visit http://www.-learningtree.com.

Borland Announces Visual dBASE 7 for Windows 95 & NT

Scotts Valley, CA — Borland announced Visual dBASE 7 for Windows 95 and Windows NT, a new version of its database-development tool. This version features new two-way visual-design tools, object-oriented component support, a 32-bit compiler, and enhanced connectivity.

Visual dBASE 7offers a new two-way integrated report writer for building and delivering customized database reports, the Project Explorer for managing application files, the Visual SQL Builder for graphically creating database queries, and support for ActiveX controls. Visual dBASE 7 also includes new data modules and object containers for managing and maintaining relationships between components, database tables, and objects and support for Microsoft FoxPro and Access data. The 32-bit Borland Database Engine has native enhancements for dBASE, Paradox, Oracle, Sybase, Informix, DB2, Microsoft SQL Server, InterBase and ODBC data sources.

In addition, Visual dBASE 7 includes Web wizards and DeltaPoint Web tools for publishing dynamic data over the Internet, and InstallShield Express for creating professional installation programs.

For more information, call (800) 457-9527, or visit the Visual dBASE Web site at http://www.borland.com/-VdBase/.

Borland Introduces High-End Internet Solutions Program

Washington, DC — Borland launched a new Internet Solutions program that creates platform-independent information networks by bridging client/server database systems and Internet platforms. The new program is based on Borland's JBuilder, a new line of visual development tools for Java, and IntraBuilder Client/-Server, used for database Web integration. The Internet solutions program combines the power of the Java and JavaScript programming languages with component-based visual tools, high-performance database connectivity, an HTML database-reporting engine, and one-year maintenance and high-end technical-support contracts.

The Borland Internet Solutions program is available from Borland in single- and multiple-developer packages, at prices starting at US$3,995. For more information, call Borland's Direct Corporate Sales department at (408) 431-1064.

By Bradley Mac Donald

The AS/400 Connection

Four Methods to "Get There" from Delphi

One of the AS/400's biggest drawbacks has been its proprietary image. In the past, an IBM development tool or 16-bit ODBC driver had to be used to develop against an AS/400 database. The resulting application was generally not portable to other database servers. Thus, the AS/400 was not considered an open platform for client/server development.

However, much has changed over the last few years, and the AS/400 is being looked at more seriously as a database server. More and more vendors, including Borland, are making their software and tools compatible with the AS/400.

This article considers various ways to connect Delphi to the AS/400 — each with its advantages and disadvantages. The trick is choosing the correct method for your shop. To help determine which method is best for you, I will discuss the pros and cons of the four main products (see Figure 1).

This review will compare the infrastructure required to support each method, ease of setup, how well each accesses the AS/400 files/programs, and the portability of developed applications. The performance of the four alternatives will not be compared in detail; from my general observation, however, the alternatives that use native-style access provide better performance.

File Access: Native versus SQL

The AS/400 native file system "feels" more like a Paradox database than a typical relational database. Like Paradox, each table is a separate, internally described file that's manipulated in a record based manner. These files are traditionally accessed using IBM's Advanced Program-to-Program Communication (APPC) protocol over Systems Network Architecture (SNA). This is the environment most AS/400 programmers are familiar with, and this article will refer to it as *native access*.

The base AS/400 operating system also ships with support for SQL Data Manipulation Language (DML) statements (e.g. SELECT, INSERT, DELETE, etc.). This means that every AS/400 can support SQL queries from a PC client machine. This enables a programmer to treat the AS/400 file system as a typical relational database. This environment will be referred to as *SQL access*.

The main difference to note between SQL and native access is the portability of applications. When you start using native access methods, whether to call a program or to access data, you may start coding to the AS/400, instead of a generic standard that could be ported to another database server. While porting between database servers is never simple, using native access will further complicate things.

Running AS/400 Programs from Delphi

Whether you're changing an AS/400 green-screen interface of an existing application into a Windows GUI, or creating a new application from scratch, there will be batch processes that need to be run on the AS/400. In general, processes such as large reports and other CPU/IO-intensive operations are best kept on the server. The issue is how to run programs on the AS/400 server from the Windows-based client.

Both Delphi/400 and Light Lib/400 use APPC to issue a Remote Procedure Call (RPC) to execute AS/400 programs. Delphi/400 uses a straightforward RPC, while Light Lib/400 uses the Delphi stored-procedure object to encapsulate the RPC call. If you're using SQL to access the AS/400, you must use stored procedures to run AS/400 programs. In most databases, a stored procedure is created using a SQL scripting language. On the AS/400, however, a stored procedure points to any high-level language program on the AS/400 (e.g. COBOL, RPG, Pascal, C, CL, etc.). It's relatively easy to submit a job on the AS/400 from a Delphi program, using either tool.

As with file access, using native methods to call an AS/400 program will make it more difficult to port your application to a different database server than will using SQL stored procedures. However, because stored procedures are so different

Product	Company	Pros	Cons
ODBC-Client Access/400 (V3R1M0)	IBM	Applications are portable. Doesn't require Delphi C/S edition. Only method that allows full use of SQL Explorer, data migration utility, and other BDE alias-related tools. Support for an AS/400 library list-like structure. Very easy to set up and configure.	Not a native driver; slower than the native access methods. Requires Client Access/400 to be installed on the workstation. No support for data areas and data queues without using the rest of Client Access/400.
SQL Links 3.5 (DB2 Driver)	Borland SQL Links	Applications are portable. Deals with AS/400 as if it's a DB2 relational database.	No native AS/400 support. No support for calling AS/400 programs due to DDCS stored procedure issue. Annoying property of DDCS to use the user's USERID to prefix table names in SQL queries. No support for data areas and data queues. Doesn't work with SQL Explorer. Difficult to set up and configure.
Light Lib/400 (version 2.0)	Luxent Software	Strong AS/400 native support. Applications are relatively portable due to their AS/400 driver for the BDE. Doesn't require Delphi C/S edition. Easy to set up and configure.	No support for data areas and data queues. Doesn't work with data migration utility.
Delphi/400 (Delphi 2)	Borland and TCIS	Very strong AS/400 native support, including data queues and data areas. Easy to set up and configure (excluding SNA gateway).	Applications are not as easily ported to other databases as with the other methods. Doesn't use the BDE nor its alias structure; this means the SQL Explorer, data-migration utility, and other tools that require BDE aliases do not work with this method.

Figure 1: An overview of the four connection methods; each has its strengths and weaknesses.

on the AS/400 compared to other database servers, you must be careful how you use them, to maintain some possibility of porting them to other database servers.

To set up stored procedures on the AS/400, you must have the full SQL product installed on your AS/400. This provides the support for the SQL Data Definition Language (DDL) statements (e.g. CREATE TABLE, CREATE INDEX, CREATE PROCEDURE, etc.). This functionality is available to anyone running OS/400 V3R1M0 with appropriate PTF levels, or a more recent version of the operating system.

ODBC

Since ODBC has gone 32-bit, some of the older issues with the 16-bit drivers and with performance have been reduced to the point where 32-bit ODBC should be considered a possible alternative. It's still not as fast as native access, but may be acceptable due to its portability. For this article, I used the 32-bit ODBC driver that ships with Client Access/400 for Windows 95/NT. One of the nice things about this driver is that it can run over TCP/IP directly to the AS/400 (it will also work with SNA), so that you don't need any protocol other than TCP/IP (which comes with Windows 95/NT) on your workstation. However, you do need the Client Access/400 product to be installed on your workstation.

Infrastructure and setup. The ODBC driver is available as soon as Client Access/400 is installed on the client machine. It's a simple matter of setting up the ODBC data source, and configuring the driver and alias in the BDE. It takes two to three minutes to set everything up (after Client Access/400 is installed), making it the easiest setup of the four products.

Access to AS/400 files and programs. The ODBC driver provides complete access to AS/400 files, and allows full use of the standard Delphi Table and Query components. In the ODBC driver, it even allows you to choose which libraries to access (almost like an AS/400 library list) — a very useful feature. Because every library on the AS/400 can be treated as a stand-alone database, each table must be qualified with the library name, so you know which file in which library you are referring to. By specifying the library(s) to check in the ODBC driver itself, you don't need to qualify the table name. This option makes porting an application from test to production as simple as changing the library list in the ODBC driver.

To access AS/400 programs, you must set up SQL stored procedures on the AS/400. Once the stored procedures are set up, calling the AS/400 programs is easy, as is passing the parameters to them; it's all done using Delphi's standard StoredProc component. Because the ODBC driver reviewed here is part of the Client Access/400 product, there is also support for RPC calls. This is useful to AS/400 shops that don't have the full SQL product on their machine; they could use the ODBC driver for database access, and use the Client Access/400 RPC support to call AS/400 programs.

Portability. The portability of an application written using the ODBC driver is very good. It's probably the best of the four products in this category. The only thing that might restrict portability of an application is the use of stored procedures that are so AS/400-specific that they would be difficult (or impossible) to reproduce on other types of database servers.

SQL Links 3.5 (DB2 Driver)

This option makes use of Borland's SQL Links 3.5 DB2 driver to access the AS/400 as if it were a DB2 database. Since it deals with the AS/400 as a relational database, this option holds the promise of creating applications that can be easily ported to other types of database servers. Due to IBM's Distributed Database Connection Services (DDCS) for NT products, however, there are various problems with this scenario. Hopefully, IBM will solve them in the new release.

Infrastructure and setup. To use this option, you must have Microsoft's SNA server (or some other SNA gateway) and IBM's DDCS for NT. IBM is working on a TCP/IP solution for DDCS that will eliminate the need for an SNA server from the configuration. This may be available by the time this article is published. Because of the extra setup to get the SNA and DDCS components up and running, this method is by far the most difficult to set up.

Access to AS/400 files and programs. All access to the AS/400 file system is done via the DB2/400 SQL engine. SQL Links 3.5 allows full use of the standard Delphi Table and Query components. There is one feature of this product that's troublesome: When you set up a Query-component connection in Delphi, DDCS tries to prefix the table name in the SELECT statement with the USERID of the user. This is done because the database is connected to the AS/400 as a whole. You then have to specify in which library you want to look for the table. In other database systems, each AS/400 library would actually be a separate database, so you wouldn't have to qualify the table name. Forcing the qualifier to be the USERID makes using the product a little unwieldy. It would be nice if IBM would allow the developer to assign the default qualifier, or provide a library list feature, such as the one in the ODBC driver. To access AS/400 programs, stored procedures have to be used. However, the DDCS (version 2.3) product has a problem that doesn't allow the use of stored procedures on the AS/400.

Portability. Like ODBC, this option offers excellent portability. Because everything is accessed using standard SQL, porting to another database should be fairly straightforward. Again, the only thing that might restrict portability of an application is the use of stored procedures specific to AS/400.

Light Lib/400

Light Lib/400, a product from Luxent Software (http://www.luxent.com), offers a unique solution, providing "native" Delphi access to the AS/400 while maintaining a good degree of portability.

Infrastructure and setup. Light Lib/400 can connect to the AS/400 via an SNA gateway or TCP/IP. It provides a new BDE driver that handles the APPC communication with the AS/400. This provides a native connection to the AS/400, while allowing full use of the BDE alias structure. It also allows you to specify the library on the AS/400, where it will look for the files in the BDE driver. This means that porting your application from test to production can be as simple as changing the BDE alias setup.

The installation process installs software on both the PC and the AS/400 directly from the PC workstation. It requires a USERID with QSECOFR privileges in order to install the software on the AS/400; it's a fairly easy installation process.

Access to AS/400 files and programs. Light Lib/400 provides access to AS/400 tables, and full use of the standard Delphi Table and Query components, so porting to other database servers is straightforward. It uses an RPC call, encapsulated inside a Delphi StoredProc component to execute programs on the AS/400. It is reasonably easy to set up, and only involves Delphi code. There is no setup on the AS/400 for calling AS/400 programs, as there is for using stored procedures with ODBC. Although the product offers a native APPC link to the AS/400 files and programs, it doesn't support data areas or data queues.

Portability. Light Lib/400 offers a good degree of portability for applications. Its combination of native access and use of a BDE driver should encourage AS/400 shops to evaluate this product.

Delphi/400

Delphi/400 is Borland's offering for connecting to the AS/400 in a native manner. It's based on a component suite from TCIS of France, and is the only product that provides access to AS/400 data areas and data queues. The version I evaluated was for Delphi 2. Hopefully, Delphi/400 for Delphi 3 will be shipping by the time this article is printed. The new version will support a TCP/IP-only connection, and provide a driver for use with the BDE.

Infrastructure and setup. Delphi/400 requires an SNA gateway, such as Microsoft's SNA Server, to connect to the AS/400. This product, unlike the other three, doesn't use the BDE to communicate with the AS/400. It uses APPC direct, bypassing the BDE and its aliases. The installation is relatively straightforward, but requires the use of an AS/400 USERID with QSECOFR privileges to install the AS/400 portion. As with the Light Lib product, both the workstation and AS/400 programs are installed from the workstation setup program.

Access to AS/400 files and programs. Delphi/400 provides the most support for native access to the AS/400 database. It has the ability to access files in their native mode (i.e. record-based access to Physical and Logical files), or as Delphi tables and queries. It is the only product that provides access to AS/400 data areas and data queues. To execute programs on the AS/400, it uses RPC calls over its APPC connection.

Portability. Portability is the biggest issue with this product. To attach to the AS/400, you must use its custom Table400 and Query400 components. These components, which replace Delphi's Table and Query components, use direct APPC communication to the AS/400, instead of the BDE or BDE aliases, to retrieve the data. This means that when you port this application to another database server, you'll have to replace the Table400 and Query400 components with Table and Query components. While this is not a huge problem (if you keep all your Table400 and Query400 components in one place), it means you have to change code in your application.

A benefit of this architecture is that you could inherit from the connection object, and add your own functionality. This inheritance feature is not available if a BDE driver is used to communicate to the AS/400, as in the case of Light Lib/400. If you use the RPC function to call AS/400 programs, it will mean changing your application's code to port it to another database server.

Conclusion

The product that offers the best native connectivity to the AS/400 is definitely Delphi/400. It offers not only direct connection to the native file system, data queues, data areas and programs, but also their own Table and Query components that are customized to the AS/400. If porting your applications to another database server is not a concern at your shop, then you should definitely take a look at this product. The new version of Delphi/400, with its support for TCP/IP and a BDE driver, should eliminate some of my concerns regarding portability of applications written using this product.

ODBC offers the best portability of the four methods, but the slowest access. The SQL Links product also promises excellent portability; however, the problems with IBM's DDCS for NT should prevent most AS/400 shops from considering it.

Luxent's Light Lib product is interesting in that it has a native APPC connection to the AS/400, yet maintains a good degree of portability. Because it still uses all the standard Delphi database connection objects and the BDE alias structure, porting to another platform should be straightforward. However, support for data areas and data queues — which would allow integration with existing AS/400 programs that use these constructs — is missing from the product. Overall, it is the blend of native access and portability that should encourage AS/400 shops to look at this product.

All these products may have new releases out by the time this article is published. Each vendor seems to be working hard to provide the best connection from Delphi to the AS/400. For

AS/400 shops using or considering Delphi, this competition and drive for improvement is good news. ∆

Bradley Mac Donald is a Technical Analyst at the Liquor Distribution Branch of British Columbia, where he is responsible for supporting Delphi and AS/400 developers. He can be reached at GBMACDON@BCSC02.GOV.BC.CA or Bradley_MacDonald@BC.Sympatico.CA.

Delphi 3 Client/Server Edition

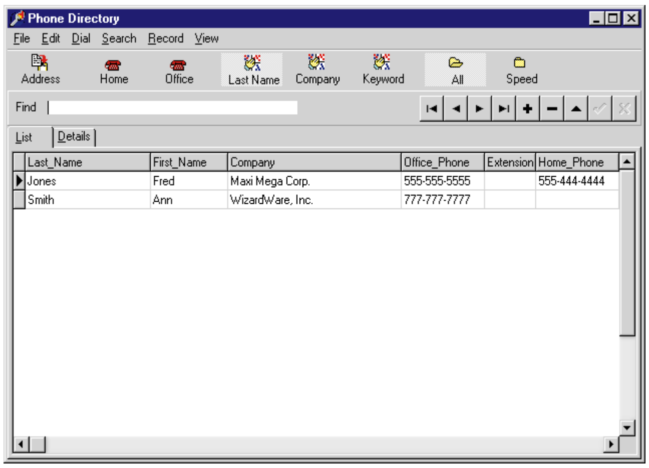
By Bill Todd

Single-Tier Database Apps

Putting the ClientDataSet Component to Work

With the Delphi 3 Client/Server Suite, you can build database applications that don't use the Borland Database Engine (BDE). And you do it with the same technology you use to build briefcase-model applications. Chapter 7 of the *Delphi 3 Developer's Guide* calls this a single-tier database application. Unfortunately, like every other aspect of the MIDAS technology in Delphi 3 Client/Server, the discussion of single-tier database applications is incomplete.

Single-tier applications use the new ClientDataSet component as their only dataset. In a multi-tier application, ClientDataSet gets its data from the middle-tier (server) application. However, ClientDataSet also has the ability to save its data to, and load its data from, a flat file. Unlike the other Delphi dataset components, ClientDataSet holds all its data in memory. While this may sound like a severe limitation, consider that 50,000 100-byte records consume only 5MB of memory. For a single-tier or briefcase-model application, this means you will probably want to limit the total number of records in all ClientDataSet components that are open simultaneously to something in the 10,000 to 50,000 range, depending on the record size.



Last_Name	First_Name	Company	Office_Phone	Extension	Home_Phone
Jones	Fred	Maxi Mega Corp.	555-555-5555		555-444-4444
Smith	Ann	WizardWare, Inc.	777-777-7777		

Figure 1: The Phone program's main form.

Because ClientDataSet uses a flat file, it doesn't require the Borland Database Engine (BDE). The only supporting file that ClientDataSet requires is DBCLIENT.DLL, which is only 150KB. This makes distributing and installing single-tier applications very easy.

To show you a simple, practical single-tier application in action, this article describes a phone-book application. The main form is shown in Figure 1. In addition to the ability to add, delete, and edit records, this program includes the following features:

- a tabbed notebook to provide a list and detail (single record) view.
- the ability to sort the directory by person's name, company name, or keyword.
- an incremental search on the current index.
- the ability to copy the name and address to the Clipboard, formatted for pasting into a letter.
- the ability to dial the home or office phone number with the click of a button.
- the ability to search for a phone number to help reconcile your phone bill.

The main form contains a single ClientDataSet component, a DataSource component, and the necessary data-aware controls to display the data. The DataSource is connected to the ClientDataSet that will be setting its *DataSet* property, just as you would if you were using a Table or Query component as the dataset.

```
procedure TPhoneForm.FormCreate(Sender: TObject);
begin
  { Start out on the List page. }
  PageControl1.ActivePage := ListSheet;

  { Get the path to the data file. }
  if ParamCount > 0 then
    begin
      { Get path to the data file from the command line. }
      DataFilePath := ParamStr(1);
      if DataFilePath[Length(DataFilePath)] <> '\' then
        DataFilePath := DataFilePath + '\';
    end
  else
    begin
      { The phone directory data file is in the same
        directory as the .EXE file. }
      DataFilePath := ExtractFilePath(Application.ExeName);
    end;

  DataFilePath := DataFilePath + DataFileName;

  { Load the data file if it exists; otherwise create it. }
  if FileExists(DataFilePath) then
    PhoneSet.LoadFromFile(DataFilePath)
  else
    CreatePhoneTable;

  SetDisplayWidths;

  { Build the indices since they aren't saved
    with the file. }
  CreateIndices;

  { Create the dialer. }
  TapiDialer := TTapiDialer.Create(Self);

end;
```

Figure 2: The main form's *OnCreate* event handler.

Building the Database

To create a single-tier application, you have to create your data tables in code. ClientDataSet doesn't provide any tools for interactive database creation, because it was primarily intended for the client side of a multi-tier application in which the data is supplied by the server. Figure 2 shows the main form's *OnCreate* event handler.

This code starts by ensuring that the List page of the page control is active at startup. Next it checks for command-line parameters. If a command-line parameter is found, it's assumed to be the path to the data file. If not, then the dataset is assumed to be in the same directory as the program's .EXE file. The *FileExists* function is used to determine if a dataset already exists. If a dataset is found, it's loaded by calling the *TClientDataSet.LoadFromFile* method. If not, the dataset is created by calling the custom method *CreatePhoneTable*, shown in Figure 3. Next, a call to the custom *SetDisplayWidths* method sets the *DisplayWidth* and *EditMask* properties of the dataset's field objects. Because saving the data in a ClientDataSet doesn't save the indices, they need to be recreated each time the data is loaded, by calling the custom method, *CreateIndices*. The last line of the *OnCreate* event handler creates an instance of the TapiDialer component used to dial phone numbers.

As Figure 3 shows, creating the dataset is a simple process. The ClientDataSet component is named *PhoneSet*, and a call

```
procedure TPhoneForm.CreatePhoneTable;
begin

  with PhoneSet do begin
    FieldDefs.Add('Last_Name', ftString, 24, False);
    FieldDefs.Add('First_Name', ftString, 16, False);
    FieldDefs.Add('Company', ftString, 35, False);
    FieldDefs.Add('Office_Phone', ftString, 20, False);
    FieldDefs.Add('Extension', ftString, 8, False);
    FieldDefs.Add('Home_Phone', ftString, 20, False);

    FieldDefs.Add('Prefix', ftString, 4, False);
    FieldDefs.Add('Title', ftString, 35, False);
    FieldDefs.Add('Address_1', ftString, 35, False);
    FieldDefs.Add('Address_2', ftString, 35, False);
    FieldDefs.Add('City', ftString, 20, False);
    FieldDefs.Add('State', ftString, 2, False);
    FieldDefs.Add('Zip', ftString, 20, False);
    FieldDefs.Add('Country', ftString, 20, False);
    FieldDefs.Add('Fax', ftString, 20, False);
    FieldDefs.Add('Spouse', ftString, 16, False);
    FieldDefs.Add('Type', ftString, 1, False);
    FieldDefs.Add('Keyword', ftString, 35, False);
    FieldDefs.Add('Speed_Dial', ftString, 1, False);
    FieldDefs.Add('EMail', ftString, 32, False);
    FieldDefs.Add('URL', ftString, 64, False);
    FieldDefs.Add('Note', ftMemo, 35, False);
    CreateDataSet;
  end;

end;
```

Figure 3: The *CreatePhoneTable* method.

```
procedure TPhoneForm.SetDisplayWidths;
begin

  with PhoneSet do begin
    FieldByName('Last_Name').DisplayWidth := 20;
    FieldByName('First_Name').DisplayWidth := 12;
    FieldByName('Company').DisplayWidth := 28;
    FieldByName('Office_Phone').DisplayWidth := 13;
    FieldByName('Extension').DisplayWidth := 6;
    FieldByName('Home_Phone').DisplayWidth := 13;
    { Set the EditMasks. }
    FieldByName('Type').EditMask := '>l;1;_';
    FieldByName('Speed_Dial').EditMask := '>l;1;_';
  end;

end;
```

Figure 4: The *SetDisplayWidths* method.

is made to its *FieldDefs.Add* method for each field in the dataset. Once all the fields have been defined, a call to the *TClientDataSet.CreateDataSet* method creates the dataset.

Figure 4 shows the code for the *SetDisplayWidths* method, which sets the *DisplayWidth* property of each field, and the *EditMask* property of two fields. This code was not included in the *CreatePhoneTable* method, because these properties must be reset each time the data is loaded from disk, not just when the dataset is initially created.

Figure 5 shows the custom *CreateIndices* method, which creates the required indices when the dataset is first created, or when it's loaded from disk. The *TClientDataSet.AddIndex* method is called once for each index created. The final statement makes the "Name" index the active index. Users can change the active index at any time by clicking the **Last Name**, **Company**, or **Keyword** toolbar buttons, or by selecting the corresponding choice from the **View** menu.

```
procedure TPhoneForm.CreateIndices;
begin

  with PhoneSet do begin
    AddIndex('Name', 'Last_Name;First_Name',
             [ixCaseInsensitive]);
    AddIndex('Company', 'Company;Last_Name',
             [ixCaseInsensitive]);
    AddIndex('Keyword', 'Keyword;Company',
             [ixCaseInsensitive]);
    AddIndex('ByHomePhone', 'Home_Phone', []);
    AddIndex('ByOfficePhone', 'Office_Phone', []);
    AddIndex('ByFax', 'Fax', []);
    IndexName := 'Name';
  end;

end;
```

Figure 5: The *CreateIndices* method.

Saving Data

Since ClientDataSet holds its data in memory, the data must be saved to disk when the user exits the program. In addition, an option on the **File** menu lets the user save the dataset at any time during a session, to prevent data loss in case of a system crash. Both the main form's *OnClose* event handler and the **File | Save Changes** menu choice's *OnClick* event handler call the *SaveChanges* method shown in Figure 6. Since there's no reason to save the dataset if the user hasn't made changes, this method checks the *PhoneSet*'s *ChangeCount* property to see if it's greater than zero. The main form's *OnClose* event handler also calls the TapiDialer component's *CloseDialer* method, and frees the TapiDialer.

The call to the dataset's *MergeChangeLog* method requires some explanation. As a user adds, deletes, and changes records, the changes aren't made to the dataset (which is held in the *TClientDataSet.Data* property), but instead are written to the *TClientDataSet.Delta* property. Although the changes are stored separately, the user always sees the latest version of the data on the screen, and so will you if you access the data in code. However, before saving the data to disk, you must call *MergeChangeLog* to merge the changes into the dataset. Finally, the call to the *TClientDataSet.SaveToFile* method writes the dataset to disk.

An undo plethora. One side benefit of storing the changes separately is that you have enhanced "undo" capability. The **Edit** menu contains three choices: **Undo Last Change**, **Undo All Changes To Record**, and **Undo All Changes**. **Undo Last Change** makes the following call:

```
PhoneSet.UndoLastChange(True)
```

The *UndoLastChange* method undoes the last change that was made, regardless of which record was changed. Setting the parameter to *True* moves the cursor to the record that was last changed. If the parameter is *False,* the change will be undone, but the current record will not change.

The **Undo All Changes To Record** menu choice calls *PhoneSet.RevertRecord*, which undoes all changes to the current record. Note that the changes will be undone whether or not the record has been posted. Calling *PhoneSet.CancelUpdates* undoes all changes to all records to implement the **Undo All Changes** menu choice.

All the code and techniques discussed so far in this article are equally applicable to multi-tier applications that implement the briefcase model. The only difference is that you don't have to create the dataset in code. Instead the data is retrieved from the middle-tier server application. You can then save the data to disk, using the *TClientDataSet.SaveToFile* method, allow the user to disconnect from the network and at a later time start the application, load the dataset from disk, and make changes.

Although the Phone program uses a single ClientDataSet component, you can use as many as you need, and link them using the *MasterSource* and *MasterFields* properties just as you would link Table components. Note that the *TClientDataSet* class also provides the same suite of events as other Delphi dataset components.

Dialing Phone Numbers

Although this article is primarily about using the ClientDataSet component, it's worth spending a few minutes looking at the phone-dialer component contained in the TapiDial unit, shown in Figure 7.

This simple component descends from *TComponent,* and adds two methods, two properties, and two fields.

The two methods are *Execute*, used to dial a number, and *CloseDialer*, used to close the Windows phone dialer.

The two properties are *PhoneNumber* and *PartyToCall*. *PartyToCall* contains the string that's displayed in the dialer dialog boxes when the call is placed. The two fields, *FPhoneNumber* and *FPartyToCall*, hold the values of the properties. The **type** block also declares a custom exception, *ETapiDialError*, that's used in the *Execute* method.

This component avoids the complexity of accessing your modem through the serial port to dial the phone number, by using the Microsoft TAPI telephony API to tell the Windows phone-dialer application to dial the number. The code in the *Execute* method begins by calling the Windows *LoadLibrary* function to load the TAPI32.DLL. If the call to *LoadLibrary* fails, the method raises the *ETapiDialError* exception.

Next comes a call to the Windows API function, *GetProcAddress*, to get the address of the *TapiRequestMakeCall* function in TAPI32.DLL, and assign it to the procedure variable *TapiRequestMakeCall* (which is declared in the **var** block at the beginning of this method).

Watch where you point. If you haven't worked with procedure variables, the syntax of the following statement may be confusing:

```
@TapiRequestMakeCall :=
  GetProcAddress(TapiLibrary, 'tapiRequestMakeCall');
```

```
procedure TPhoneForm.SaveChanges;
begin

  with PhoneSet do begin
    if ChangeCount > 0 then
      begin
        MergeChangeLog;
        SaveToFile(DataFilePath);
      end;
  end;

end;
```

Figure 6: The *SaveChanges* method.

The @ operator is normally used to designate the address of a variable (i.e. it returns a pointer to the variable). For example, the following code assigns the address of the variable *SomeNumber* to the pointer *SomeNumPtr*:

```
var
  SomeNumber: Integer;
  SomeNumPtr: ^Integer;
begin
  SomeNumPtr := @SomeNumber;
end;
```

In the case of a procedure variable, such as *TapiRequestMakeCall*, however, there is a problem. A procedure variable is a pointer to a procedure or function. In Pascal, when a procedure variable appears in code, the compiler interprets it as a call to the procedure whose address the variable contains.

So the statement:

```
TapiRequestMakeCall :=
  GetProcAddress(TapiLibrary, 'tapiRequestMakeCall');
```

would be interpreted as a call to the procedure that *TapiRequestMakeCall* points to, not as an assignment of the address returned by *GetProcAddress* to the variable.

To resolve this problem, you must precede the procedure variable with the @ operator when your intent is to access the address the variable contains, instead of calling the procedure at that address.

Once the address of the *TapiRequestMakeCall* function in the DLL is obtained, the code checks the length of the phone

```
unit TapiDial;

interface

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

type
  ETapiDialError = class(Exception);
  TTapiDialer = class(TComponent)
  private
    { Private declarations  }
    FPhoneNumber: string;
    FPartyToCall: string;
  protected
    { Protected declarations }
  public
    { Public declarations }
    procedure Execute;
    procedure CloseDialer;
  published
    { Published declarations }
    property PhoneNumber: string read  FPhoneNumber
                                 write FPhoneNumber;
    property PartyToCall: string read  FPartyToCall
                                 write FPartyToCall;
  end;

procedure Register;

implementation

procedure TTapiDialer.Execute;
var
  NumberToDial: string;
  TapiResult: LongInt;
  TapiLibrary: THandle;
  TapiRequestMakeCall:
    function (lpszDestAddress, lpszAppName,
              lpszCalledParty, lpszComment: PChar):
              LongInt; stdcall;
begin
  if FPhoneNumber = '' then
    raise ETapiDialError.Create(
             'The phone number is blank.');

  { Load the TAPI dll. }
  TapiLibrary := LoadLibrary('TAPI32.DLL');

  if TapiLibrary = 0 then
    raise ETapiDialError.Create(
            'Error loading TAPI32.DLL library.');

  { Get the address of the dll function. }
  @TapiRequestMakeCall :=
    GetProcAddress(TapiLibrary, 'tapiRequestMakeCall');

  { If the number includes an area code prepend a one. }
  if Length(FPhoneNumber) > 8 then
    NumberToDial := '1' + FPhoneNumber
  else
    NumberToDial := FPhoneNumber;

  { Dial the number. }
  TapiResult :=
    TapiRequestMakeCall(PChar(NumberToDial),'',
                        PChar(FPartyToCall),'');
  FreeLibrary(TapiLibrary);

end;

procedure TTapiDialer.CloseDialer;
{ Close the Phone Dialer if it is open. }
var
  Dialer: THandle;
begin
  Dialer := FindWindow(nil, 'Phone Dialer');

  if Dialer <> 0 then
    PostMessage(Dialer, WM_QUIT, 0, 0);

end;

procedure Register;
begin
  RegisterComponents('Samples', [TTapiDialer]);
end;

end.
```

Figure 7: The TapiDial unit.

number. If it includes an area code, a "1" is added to the beginning of the string, and *TapiRequestMakeCall* is called to dial the number. *TapiRequestMakeCall* dials the number by loading the Windows phone-dialer application, and passing the number to it.

The *CloseDialer* method closes the Windows phone dialer. It calls the Windows API function *FindWindow* to get the window handle of the phone dialer's window, then calls *PostMessage* to send a WM_QUIT message to the dialer.

Conclusion

The ClientDataSet component gives you a lot more than just multi-tier application capability. You can bring a new dimension to your development projects by building simple database applications that don't require the BDE, and briefcase-model programs that let users take their data on the road — and edit it on the way. ∆

The project referenced in this article is available on the Delphi Informant Works CD located in INFORM\98\JAN\DI9801BT.

Bill Todd is President of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is co-author of four database programming books, including *Delphi: A Developer's Guide* [M&T Books, 1995] and *Creating Paradox for Windows Applications* [New Riders Publication, 1994]; a contributing editor of *Delphi Informant*; a member of Team Borland providing technical support on the Borland newsgroups; and has been a speaker at every Borland Developers Conference. He can be reached at Bill_Todd@compuserve.com or (602) 802-0178.

OP TECH

Delphi 2 / Delphi 3

By Keith Wood

The Property Explorer Expert

Using the *ToolServices* Object to Build a Development Aid

Delphi provides numerous ways to make building applications easier. Its components can be customized and integrated into the environment, and templates and code-generating experts can be created and added to the Object Repository for reuse. We can also add applications to its **Tools** menu for easy access to supporting programs.

Delphi's experts can generate individual forms or entire projects. Delphi also allows you to create experts that provide additional functionality within its environment, and makes them available through the menu structure (the Database Form Expert is one example). In this article, we'll build an expert that locates properties of objects with a specified name or value in the current project.

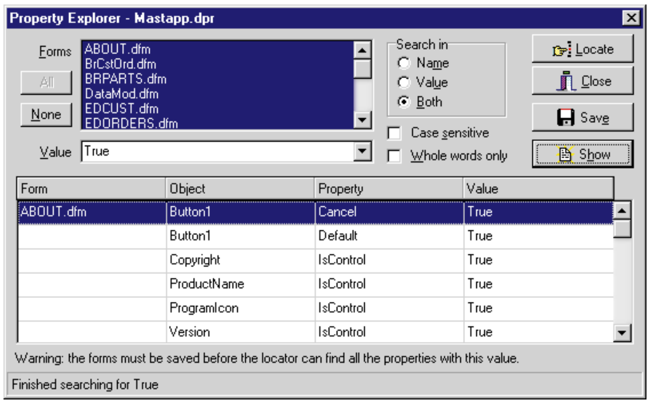
Property Values

Have you ever wondered what event calls a given method in your form, or what list boxes contain a given phrase? Although Delphi provides easy access from an object to its properties and values, it doesn't make it easy to do the reverse. The property values are hidden in the .DFM file, which are normally stored in a binary format that isn't very readable. These files can be loaded into the Code Editor and viewed as text, but this approach may not be practical for a medium to large project — hence the need for a utility that shows what object properties have a given value.

We have three objectives in our example:

- search one or more of the forms in the current project,
- enter the property name or value to be located, and
- set various matching options.

After performing the search, the utility displays a list of the properties of objects that have this name or are set to this value, and the names of the forms in which to find them. Figure 1 shows the expert in action.



Property Explorer - Mastapp.dpr

Forms: ABOUT.dfm, BrCstOrd.dfm, BRPARTS.dfm, DataMod.dfm, EDCUST.dfm, EDORDERS.dfm

Value: True

Search in: Name, Value, [✓ Both]

Case sensitive: ☐
Whole words only: ☐

Form	Object	Property	Value
ABOUT.dfm	Button1	Cancel	True
	Button1	Default	True
	Copyright	IsControl	True
	ProductName	IsControl	True
	ProgramIcon	IsControl	True
	Version	IsControl	True

Warning: the forms must be saved before the locator can find all the properties with this value.

Finished searching for True

Figure 1: The property explorer expert.

Expert Basics

Delphi experts are shared libraries (DLLs). Each expert must register its presence when the DLL is loaded. Thereafter, Delphi extracts information about most experts when displaying the Object Repository or **Help** menu. Once selected, these experts are invoked through their *Execute* methods, allowing them to perform a specific task.

All experts are classes derived from *TIExpert*, which defines the interface that must be implemented for the expert to interact with the Delphi environment. *TIExpert* has a number of methods that must be overridden to supply basic information about the expert: its name, style, state, and a unique ID string. The style of an expert determines how it appears in the Delphi IDE and how it's invoked. Four possible styles exist:

- *esForm* for an expert that usually resides on the Forms tab in the Object Repository and generates code for a single form;
- *esProject* for an expert that produces code for an entire project and is usually called from the Projects tab in the Object Repository;
- *esStandard* for an expert that is invoked from the **Help** menu; and
- *esAddIn* for experts that handle their own interactions with Delphi. Add-in experts are different from the others in that they're not invoked through their *Execute* method. (We'll save that topic for another article.)

Additional information depends on the type of expert being developed. Form and project experts require a longer descriptive text and an icon to represent it in the Object Repository. A standard expert requires the text displayed on the menu to be supplied. Finally, all these experts must override the *Execute* method to actually perform the work.

Working from the example experts available in Delphi, we see that only one function must be exported from the DLL. This is *InitExpert*, which registers the expert with Delphi. The code for this routine can be copied directly from the example (see the end of this article for download details), needing only the registration statement to be changed:

```
{ Register the expert. }
RegisterProc(TPropExplorerExpert.Create);
```

ToolServices

To make interactions with the IDE possible, Delphi defines a *TIToolServices* class and creates a *ToolServices* object. A reference to this object is available to the expert when it's invoked. The public face of this class appears in the ToolIntf.pas file in \Source\Toolsapi.

The *ToolServices* object allows us to open, save, and close forms and projects, as well as create new modules within a project. It returns the number and names of the forms and units in a project, and allows us to examine the Component palette. It also provides a mechanism for reporting exceptions to the IDE.

```
{ Create new expert dialog and find current file. }
constructor TfrmPropExplorerExpert.Create(
  AOwner: TComponent);
var
  i: Integer;
  sProjectPath, sFilePath, sFileName: string;
  slsModified: TStringList;
  ediEditor: TIEditorInterface;
  fmiForm: TIFormInterface;
begin

  inherited Create(AOwner);
  { Display project name.  }
  Caption := Caption + ' - ' +
             ExtractFileName(ToolServices.GetProjectName);
  { Align results grid columns.  }

  for i := 0 to hdrResults.Sections.Count - 1 do
    stgResults.ColWidths[i] :=
      hdrResults.SectionWidth[i] - 1;

  { Create list for modified files. }
  slsModified := TStringList.Create;
  try
    sProjectPath := ExtractFilePath(
                      ToolServices.GetProjectName);

    { Load form names into listbox. }
    for i := 0 to ToolServices.GetFormCount - 1 do begin
      sFilePath := ExtractFilePath(
                     ToolServices.GetFormName(i));
      if sFilePath = sProjectPath then
        sFilePath := ''
      else
        sFilePath := ' in ' + sFilePath;

      lbxForms.Items.AddObject(ExtractFileName(
        ToolServices.GetFormName(i)) + sFilePath,
        TString.Create(ToolServices.GetFormName(i)));
      sFileName :=
        ChangeFileExt(ToolServices.GetFormName(i),'.pas');

      { Check if modified. }
      if ToolServices.IsFileOpen(sFileName) then
        with ToolServices.GetModuleInterface(sFileName) do
          try
            ediEditor := GetEditorInterface;
            fmiForm := GetFormInterface;
            if ediEditor.BufferModified or
               fmiForm.FormModified then
              slsModified.Add(sFileName);
          finally
            fmiForm.Free;
            ediEditor.Free;
            Free;
          end;
    end;
    { Ask for action if modified files haven't been saved. }
    if slsModified.Count > 0 then
      case MessageDlg(
             'Some files in this project have'#13#10 +
             'been modified but not yet saved.'#13#10 +
             'Save these files?',
             mtConfirmation, mbYesNoCancel, 0) of
        mrYes:
          for i := 0 to slsModified.Count - 1 do
            ToolServices.SaveFile(slsModified[i]);
        mrNo:     { Ignore. };
        mrCancel: Abort;
      end;
  finally
    slsModified.Free;
  end;

  { Highlight current form. }
  i := lbxForms.Items.IndexOf(ChangeFileExt(
    ExtractFileName(ToolServices.GetCurrentFile),'.dfm'));

  if i > -1 then
    lbxForms.Selected[i] := True
  else if lbxForms.Items.Count = 1 then
    lbxForms.Selected[0] := True;

end;
```

Figure 2: The new constructor for the expert form.

```
type
  { Encapsulate a string in an object. }
  TString = class
  private
    FValue: string;
  public
    constructor Create(sValue: string);
    property Value: string read FValue write FValue;
  end;

{ Create a new object with an embedded string. }
constructor TString.Create(sValue: string);
begin
  inherited Create;
  FValue := sValue;
end;
```

Figure 3: A wrapper class for a string.

In this utility, we use the *ToolServices* object to list the forms in the current project and identify those (if any) being modified. The list simplifies the selection of forms to be searched — keeping in mind that the current form can be automatically selected as soon as the expert appears. This allows us to search the current form with minimal effort.

Initialization

In the initialization of the expert's form, we want to retrieve this list and place the values into a list box. To do this, we need access to the *ToolServices* object from within the form. This is easily accomplished by including the ExptIntf unit in the **uses** clause of our unit. We also need to include the ToolIntf unit to provide the definition of the class.

The initialization occurs in the *Create* method of our form. First, call the *Create* method from the parent class, passing the supplied parameter through to this call, allowing the owner of the new form to be established. Thereafter, use the *ToolServices* object to obtain the number of forms in the project and their names, and to identify those currently being modified. The names of the forms are added to the list box, which has already been set to allow for multiple selection of entries. Because of this, highlight the current form with the *Select* property rather than the *ItemIndex* property. Figure 2 shows the new constructor for the expert form.

To keep the list of form names legible, we only display the name of the file, unless the form doesn't reside in the project directory (in which case we want to display the full path name). The path is easily extracted from the form name by using the *ExtractFileName* procedure, but we want to keep the full path name to supply to the scanning process later. Delphi allows you to associate objects with given strings in all string lists, such as the one inside a list box. Unfortunately, the value that we want to store is a string, which isn't an object.

To work around this, create a smaller wrapper class to encapsulate a string value (see Figure 3). It is derived directly from *TObject* (by default) and has a single property, *Value*, which is a string. To simplify its use, we have a customized constructor that takes the value of the string to be contained. This can then be accessed by referring to the property. The use of this class can be seen in Figure 2. We created an instance of the wrapper class

```
object Form1: TForm1
  Left = 200
  Top = 99
  AutoScroll = False
  Width = 243
  Height = 147
  Caption = 'TabSet Drag/Drop Demo'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = [fsBold]
  PixelsPerInch = 96
  TextHeight = 13
  object TabSet1: TTabSet
    Left = 0
    Top = 99
    Width = 235
    Height = 21
    Align = alBottom
    Font.Color = clWindowText
    Font.Height = -11
    Font.Name = 'MS Sans Serif'
    Font.Style = []
    Tabs.Strings = (
      'First'
      'Second'
      'Third'
      'Fourth'
      'Fifth')
    TabIndex = 0
    TabOrder = 0
    OnDragDrop = TabSet1DragDrop
    OnDragOver = TabSet1DragOver
    OnMouseDown = TabSet1MouseDown
  end
end
```

Figure 4: An example of the text representation of a .DFM file.

with the appropriate value, in this case the full name of the form, and added it with the short filename to the list.

As we'll see later, we will read the properties from the .DFM files on the disk. This means if the forms have been updated but not yet saved, we're reading outdated values. To overcome this, we can use the *ToolServices* object to check and save any forms.

For this, we need to create a module interface (*TIModuleInterface*) for each form. From this, we can retrieve interfaces to the Code Editor for the unit (*TIEditorInterface*) and to the form itself (*TIFormInterface*). The *BufferModified* and *FormModified* properties of these objects show if the unit or form has been altered and should be saved. We keep a reference to any that have changed and ask the user to confirm saving them. If they agree, use the *SaveFile* method of the *ToolServices* object to complete the process. Results can still be obtained even when the latest versions are not saved, but obviously will not reflect the most recent values.

Reading the Properties

As mentioned earlier, the values of the properties set in the Object Inspector are held in the .DFM file in a binary format. These are the values we're interested in, but we need an easier way of accessing them.

Remember that we can view a text representation of the .DFM file in the Code Editor (see Figure 4). Each object, including the form, is identified by the keyword **object**, followed by its name and type. Next appears a list of the properties with values different from the defaults set for that component. These appear as the property name, an equals sign (=), and the property's value. String values are enclosed in quotes. Lists of strings are denoted by enclosing the list in parentheses and placing each entry on a separate line. Events also appear as properties with the value being the name of the method to invoke. Objects contained in another object appear inside the latter's definition. This can continue to an arbitrary depth. When an object has been fully defined, it's delimited with an **end** keyword.

The .DFM file on the disk is in a binary format, but we want the text representation to manipulate. To get this, use the *ObjectResourceToText* procedure. This takes two streams as parameters, reading the binary format from the first, and writing the text version to the second. Opening an input stream on the .DFM file is simply a matter of creating a *TFileStream* object and specifying the name of the file to read. To avoid creating a temporary file for the output stream, we can generate it directly in memory by creating a *TMemoryStream* object. After calling the conversion procedure, reset the output stream to the beginning to initiate processing it:

```
{ Create streams for processing. }
stmForm := TFileStream.Create(TString(
  lbxForms.Items.Objects[iForm]).Value, fmOpenRead);
stmMemory := TMemoryStream.Create;
{ Translate binary resource file to text. }
ObjectResourceToText(stmForm, stmMemory);
{ Move back to beginning of text format. }
stmMemory.Position := 0;
```

Streams into Tokens

For each of the form files selected by the user, we must open it and convert it into the text version in a memory stream. This stream then provides a sequence of bytes for us to read. To make this flow more useful, break it into lines and divide the lines into individual tokens.

```
{ Build up a text line from the stream.
  Return True when end of the stream encountered. }
function EndOfStream(stm: TStream): Boolean;
var
  c: Char;
begin
  sLine := '';  { Clear current line. }
  try
    { Exception raised at end of stream. }
    while True do begin
      { Read next character from stream. }
      stm.ReadBuffer(c, 1);
      if c = cCR then  { End of line. }
        begin
          stm.ReadBuffer(c, 1);  { Skip line feed. }
          Result := False;  { More to come. }
          Exit;
        end
      else
        sLine := sLine + c;  { Add character to line. }
    end;
  except
    on EReadError do  { End of stream encountered. }
      Result := True;
  end;
end;
```

Figure 5: Reading a line from a stream.

The lines in the stream are delimited by a carriage return and line feed combination. If this were a straight text file, we could use the Pascal *Readln* procedure and *EOF* function to process each line. Because memory streams don't provide this functionality, however, we must add it ourselves.

```
{ Break line into tokens separated by white space }
procedure GetTokens;
var
  iPos, iLen: Integer;
  sToken, sLiteral: string;
  cDelimiters: set of Char;
begin
  slsTokens.Clear;
  iLen := Length(sLine);
  iPos := 1;
  repeat
    { Skip whitespace separators }
    while (iPos <= iLen) and
          (sLine[iPos] in cWhiteSpace) do
      Inc(iPos);
    if iPos > iLen then  { End of line ? }
      Break;

    sToken := '';
    { Quote delimited; look for matching quote }
    if sLine[iPos] = '''' then
      cDelimiters := ['''']
    { String starting with an embedded literal }
    else if sLine[iPos] = '#' then
      cDelimiters := ['''']
    else
      cDelimiters := cWhiteSpace;
    repeat
      repeat
        { Check for embedded literal in string;
          stored as #n }
        if ('''' in cDelimiters) and
           (sLine[iPos] = '#') then
          begin
            sLiteral := '';
            Inc(iPos);
            while (iPos <= iLen) and
                  (sLine[iPos] in ['0'..'9']) do begin
              sLiteral := sLiteral + sLine[iPos];
              Inc(iPos);
            end;
            sToken := sToken + Chr(StrToInt(sLiteral));
          end;

      until (iPos > iLen) or
            not ('''' in cDelimiters) or
            (sLine[iPos] <> '#');
      { Collect characters in token }
      if '''' in cDelimiters then  { Skip opening quote }
        Inc(iPos);
      while (iPos <= iLen) and
            not (sLine[iPos] in cDelimiters) do begin
        sToken := sToken + sLine[iPos];
        Inc(iPos);
      end;
      if '''' in cDelimiters then  { Skip closing quote }
        Inc(iPos);

    until (iPos > iLen) or
          not ('''' in cDelimiters) or
          (sLine[iPos] <> '#');
    { And add to list }
    slsTokens.Add(sToken);

  until iPos > iLen;

end;
```

Figure 6: Extracting tokens from a line.

```
{ Check the tokens for meaning. }
procedure ProcessTokens;
var
  i: Integer;
begin
  if slsTokens.Count > 0 then
    begin
      { Add new object. }
      if (slsTokens[0] = 'object') or
         (slsTokens[0] = 'inherited') then
        begin
          slsObjects.Add(Copy(slsTokens[1], 1,
            Length(slsTokens[1]) - 1));  { Remove colon. }
          Exit;
        end;
      { Remove current object. }
      if (slsTokens[0] = 'end') and not bInCollection then
        begin
          slsObjects.Delete(slsObjects.Count - 1);
          Exit;
        end;

      { Check inside list. }
      if bInList then
        begin
          sValue := sValue + slsTokens[0];
          { Matched - record where it was found. }
          if ((ragSearchIn.ItemIndex in iValues) and
             Matched(slsTokens[0], sMatchValue)) or
             ((ragSearchIn.ItemIndex in iProperties) and
             Matched(sProperty, sMatchValue)) then
            bFound := True;
          if slsTokens.Count > 1 then  { End of list. }
             begin
               bInList := False;
               if bFound then
                 slsCalls.Add(lbxForms.Items[iForm]  + cSep +
                   slsObjects[slsObjects.Count - 1]  + cSep +
                   sProperty + cSep + sValue + cSep +
                   TString(
                     lbxForms.Items.Objects[iForm]).Value);
               sProperty := '';
             end;
        end;

      { Check for end of collection. }
      if bInCollection and (slsTokens[0] = 'end>') then
        bInCollection := False;

      { Check normal property. }
      if slsTokens.Count > 2 then
        begin
          sValue := '';
          for i := 2 to slsTokens.Count - 1 do
            sValue := sValue + slsTokens[i];
          { Matched - record where it was found. }
          if ((ragSearchIn.ItemIndex in iValues) and
             Matched(sValue, sMatchValue)) or
             ((ragSearchIn.ItemIndex in iProperties) and
             Matched(slsTokens[0], sMatchValue)) then
            slsCalls.Add(lbxForms.Items[iForm] + cSep +
              slsObjects[slsObjects.Count - 1] + cSep +
              slsTokens[0] + cSep + sValue + cSep +
              TString(lbxForms.Items.Objects[iForm]).Value);
          { Start of list. }
          if slsTokens[2] = '(' then
            begin
              bInList := True;
              sProperty := slsTokens[0];
              bFound := False;
              sValue := '';
            end;
          { Start of collection. }
          if slsTokens[2] = '<' then
            bInCollection := True;
        end;
    end;
end;
```

Figure 7: Checking property values.

To begin, combine the effect of these two Pascal routines into a single function that returns *True* when the end of the stream has been encountered. If it returns *False*, it has the side effect of updating a variable with the next line retrieved from the stream (up to, but excluding, the carriage return/line feed). This function is used to control our loop for processing all the lines in each file (see Figure 5). It uses the *TStream.ReadBuffer* method to read one character at a time. These are concatenated into the current line until the end of the line is reached. This method raises an *EReadError* exception when it reaches the end of the input stream. The routine traps this exception and returns a *True* flag to the caller.

Once we have an individual line from the stream, divide it into its component parts (tokens) for easier processing. From the example in Figure 4, we can see that "white space" is used to separate the items of interest, and string values are enclosed in single quotes. The spacing before the first character varies depending on the containership of the objects and properties. Skip over any white space to extract the tokens, then read characters until you find more white space. Each token that is found in this way is added to a list of tokens for that line. It's easy to determine the total number of tokens found in that line and to refer to any one of them. Figure 6 shows the code that performs this.

If the token started with a quote, it's a string value and we must read until we find a matching quote — possibly bypassing spaces as we go. A complication arises if the string contains an embedded quote or null character. This appears as a character literal:

```
Caption = 'Keith'#39's form'
```

with quotes before and after to delimit the surrounding strings. This special case must be handled in the code.

Finding the Matches

As we encounter each new object, denoted by the first token for the line being the keyword **object**, we add it to a LIFO (last-in-first-out) stack of object names. The current object is at the top of the stack (the end of the list in our case). When we encounter the end of an object (i.e. the first token is the keyword **end**), remove the current object from the stack, returning it to its parent.

For each property, we compare its name (the first token), and/or value (the third token), with the value for which we are searching. If it matches, we record the names of the current form, the current object and its property, and the property's value. These are combined into another list of strings. This allows the results to be easily sorted, making

```
{ Open the selected form and focus on the component. }
procedure TfrmPropExplorerExpert.btnShowClick(
  Sender: TObject);
var
  i: Integer;
  sFileName: string;
  fmiForm: TIFormInterface;
  cpiComponent: TIComponentInterface;
begin
  with stgResults do begin
    i := Row;
    while (i >= 0) and (Cells[0, i] = '') do
      Dec(i);
    if i >= 0 then
      begin
        { Open file. }
        sFileName := ChangeFileExt(Cells[4, i], '.pas');
        ToolServices.OpenFile(sFileName);
        { Attach to the module. }
        with ToolServices.GetModuleInterface(sFileName) do
          try
            { Find the specified component... }
            fmiForm := GetFormInterface;
            cpiComponent :=
              fmiForm.FindComponent(Cells[1, Row]);
            if cpiComponent <> nil then
              { ...and focus on it. }
              cpiComponent.Focus;
          finally
            cpiComponent.Free;
            fmiForm.Free;
            Free;
          end;
        Close;
      end;
  end;
end;
```

Figure 8: Selecting a component from the list.

it simpler for the user to locate a particular object or property in the results.

A radio group on the form allows us to restrict the search to only the property names, their values, or both. Also, two checkboxes have been added to the expert to control the way matching occurs during processing. These allow the value entered to be matched exactly as typed, or to ignore differences in case, and/or to match the whole of the property value or just part of it.

If the property is a list of strings, then its initial property value is an opening parenthesis. Once we see this, the processing needs to change slightly. We must remember the name of the property from the current line (the first token) then check each entry in the list (now the first token in each line) until we find its end. The final list element has a closing parenthesis following its value (see Figure 7).

Additional complications arise when processing collections. These are delimited by angle brackets (< >), and consist of a list of nameless items — each with its properties and ending keyword. These keywords must be ignored to avoid closing off the parent object.

Displaying the Results

Once we've processed the form files, there's a string list that contains the properties with the required name or value. These have already been sorted by the string list and now we must step through each and extract the component parts into the string grid that displays the results to the user. If no matches are found, the grid displays this fact.

We can save the results of the search to a disk file using the **Save** button. This asks for the name of the file to save, then creates a four-column text report listing the form, object and property names, and the property value. Each column is restricted to 32 characters.

Alternatively, we can choose one of the objects and return to the form designer with it selected by pressing the **Show** button. To achieve this, we'll use the *ToolServices* object. First, make a call to its *OpenFile* method to ensure that the unit and form for this object are available within Delphi. Next, open interfaces to the module and the form itself, before locating the desired component on that form using the *FindComponent* method. Finally, select that component with the *Focus* method of the component interface (see Figure 8).

Implementation

Once the DLL has been created, our expert can be incorporated into the Delphi IDE by updating the Windows registry. (This is done using RegEdit.exe. Important note: Make no changes to the registry without first backing it up.) Under the key HKEY_CURRENT_USER\Software\Borland\Delphi\2.0\Experts (or 3.0 as appropriate), add a new string value for the DLL:

```
PropExpl C:\Program Files\Borland\Delphi 2.0\
  PropExpl\PropExpl.dll
```

Then it's just a matter of restarting Delphi; the expert should appear on the **Help** menu (as shown in Figure 9). Now you can open a project and start searching it for that lost property. As a demonstration, open the MASTAPP project in \Demos\Db\Mastapp. The project has multiple forms and hundreds of object properties available for searching. String values in the form file(s) have their surrounding quotes removed when they're extracted as tokens, so no quotes are necessary when entering a search value.

Uses for this utility include determining whether a particular method is being called by any event, and if so, which one(s). It can also check standards throughout a project, such as locating all *Font.Name* properties and checking their settings.

Remember, however, that the expert only finds values that have been saved to the form file on the disk. Properties that have default values set are usually not saved to this file.

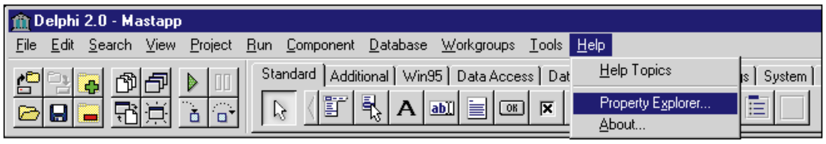

Delphi 2.0 - Mastapp
File Edit Search View Project Run Component Database Workgroups Tools Help
Standard Additional Win95 Data Access Dat
System
Help Topics
Property Explorer...
About...

Figure 9: Accessing the new expert from the Delphi 2 **Help** menu.

Furthermore, the expert cannot find values for properties that have been updated in the current session, but haven't been saved; a warning to this effect appears on the expert.

Conclusion

Delphi is one of the most customizable development environments running on Windows. We can add customized components and make them indistinguishable from those that come with Delphi. New form and project templates can be added to the Object Repository for later use, just as standard ones. Experts to generate code for either forms or projects can be written in Delphi then integrated into the Object Repository. External applications can be added to the Delphi menus to provide additional support.

Lastly, we can develop specialized tools, designed to work inside the Delphi IDE, and include them on the menus as well. The utility presented in this article uses the *ToolServices* object provided by Delphi to access some of its internal workings. It then interacts with the IDE to extract valuable information to help reduce our programming efforts. Δ

The project referenced in this article is available on the Delphi Informant Works CD located in INFORM\98\JAN\DI9801KW.

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

By Rod Stephens

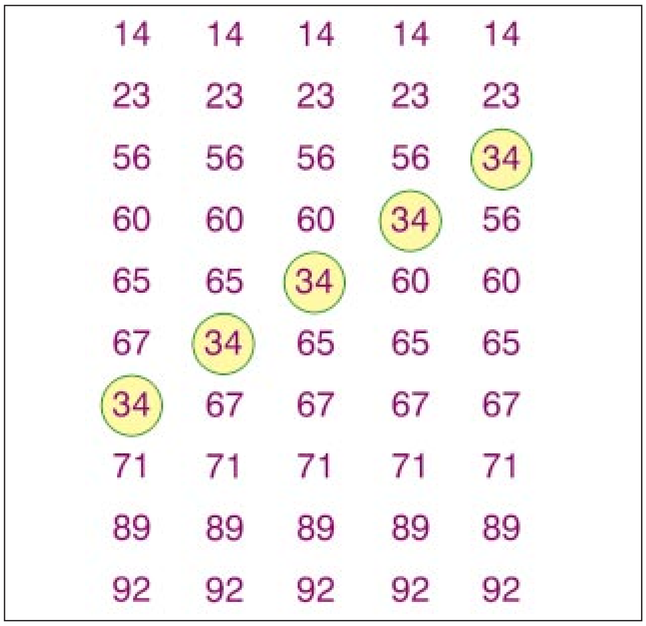
Sorts of All Types

Implementing Classic Sort Routines in Delphi

Sorting is one of the most heavily studied topics in algorithms — and for good reason. First, sorting is a common programming task. Allowing a user to sort and view information in different ways makes data more meaningful. Second, sorting algorithms demonstrate many important algorithmic techniques, such as binary subdivision and recursion. Studying these algorithms allows you to hone your programming skills in a relatively simple setting.

Finally, and perhaps most importantly, different sorting algorithms work differently under different circumstances, so no single algorithm is the best choice for all occasions. Understanding which algorithm works best with different sets of data is as important as understanding the algorithms themselves. Only by understanding a variety of algorithms can you pick the one that is right for a given task. This article describes four sorting algorithms that can handle a wide variety of sorting tasks:

- Bubblesort
- Selectionsort
- Quicksort
- Countingsort



14	14	14	14	14
23	23	23	23	23
56	56	56	56	34
60	60	60	34	56
65	65	34	60	60
67	34	65	65	65
34	67	67	67	67
71	71	71	71	71
89	89	89	89	89
92	92	92	92	92

Figure 1: The item "34" bubbles up to its correct position.

Bubblesort

Bubblesort is without a doubt the most misunderstood sorting algorithm ever invented. Some programmers always use Bubblesort — even though it's usually not the fastest algorithm. Others claim that Bubblesort is a bad algorithm that should never be used. As is often the case with such extreme positions, both are wrong.

Bubblesort is a highly specialized algorithm that is useful under only two sets of circumstances. First, if the list of items to be sorted is very short, Bubblesort is quite fast. Its method of arranging items by switching them around is very efficient if the list to be sorted contains fewer than roughly 10 items. Second, if the items to be sorted are already in mostly-sorted order, Bubblesort is extremely efficient. If most of the list is randomly arranged, however, Bubblesort is abysmally slow. For this reason, you must be careful before you use Bubblesort.

Bubblesort scans through a list looking for two adjacent items that are out of order. When it finds two such items, it swaps them. It repeats this process until all the items are in order.

Let's look at an example: In the list of numbers on the left of Figure 1, the item 34 is out of order. When the algorithm passes through the list, it finds that 34 and 67 are out of order, so it switches them. During the second pass, the algorithm sees that the items 65 and 34 are out of order, so it switches them. The algorithm continues "bubbling" the item 34 toward the top of the array until it reaches its correct position.

```
type
  ValueType = Longint;   // Type used in the arrays.
  IndexType = Longint;   // Type used to index arrays.

procedure Bubblesort(var List: array of ValueType;
  min, max: IndexType);
var
  last_swap, i, j: IndexType;
  tmp: ValueType;
begin
  // During this loop, min and max are the smallest and
  // largest indexes of items that might still be
  // out of order.

  // Repeat until we are done.
  while (min < max) do begin
    // Bubble up.
    last_swap := min - 1;
    // for i := min + 1 to max.
    i := min + 1;
    while (i <= max) do begin
      // Find a bubble.
      if (List[i - 1] > List[i]) then
        begin
          // See where to drop the bubble.
          tmp := List[i - 1];
          j := i;
          repeat
            List[j - 1] := List[j];
            j := j + 1;
            if (j > max) then
              Break;
          until (List[j] >= tmp);
          List[j - 1] := tmp;
          last_swap := j - 1;
          i := j + 1;
        end
      else
        i := i + 1;
    end;   // while (i <= max) do.
    // End bubbling up.

    // Update max.
    max := last_swap - 1;

    // Bubble down.
    last_swap := max + 1;
    // for i := max - 1 downto min.
    i := max - 1;
    while (i >= min) do begin
      // Find a bubble.
      if (List[i + 1] < List[i]) then
        begin
          // See where to drop the bubble.
          tmp := List[i + 1];
          j := i;
          repeat
            List[j + 1] := List[j];
            j := j - 1;
            if j < min then
              Break;
          until (List[j] <= tmp);
          List[j + 1] := tmp;
          last_swap := j + 1;
          i := j - 1;

        end
      else
        i := i - 1;
    end;   // while (i >= min) do.
    // End bubbling down.

    // Update min.
    min := last_swap + 1;
  end;   // while (min < max) do.
end;
```

Figure 2: Two-way Bubblesort.

This algorithm is faster if it alternates upward and downward passes through the list. During downward passes like those illustrated in Figure 1, the item 34 can move only one position upward. In this example, the algorithm requires four passes through the list to move the item to its correct position.

On the other hand, an upward pass through the list would be able to move the item many positions. It would see that 34 and 67 were out of position, and switch them. During the same pass, it would then notice that 34 and 65 were out of position so it would switch them, too. It would then switch 34 with 60 and 56 — all in one pass.

During each set of upward and downward passes, at least one new item will reach its final position. If the list is initially mostly sorted, the algorithm will need only a few passes through the list to reposition the items that are out of order.

Figure 2 shows the Delphi source code for the two-way Bubblesort algorithm. This code, and all of the code described in this article, is demonstrated by the Sort program, and is available for download (see end of article for details). You can use the program to test the algorithms on random lists of various sizes. The algorithm code is contained separately in the unit SortAlgs.pas, so you can easily add it to your projects.

Selectionsort

Like Bubblesort, Selectionsort is efficient for very small lists. It also has the advantage of being extremely simple, so it's easy to implement, debug, and maintain.

The Selectionsort algorithm begins by searching the list for the smallest item. It then swaps that item with the item at the front of the list. It then searches the remaining items for the next smallest item, and swaps it into the list's second position. This algorithm continues searching the shrinking list of unsorted items, picking out the smallest and swapping it to the end of the sorted section at the front of the list. When it has swapped every item into its final position, the algorithm stops. The Delphi source code for Selectionsort is shown in Figure 3.

```
procedure Selectionsort(var List : array of ValueType;
  min, max : IndexType);
var
  i, j, best_j: IndexType;
  best_value: ValueType;
begin
  for i := min to max - 1 do begin
    best_value := List[i];
    best_j := i;
    for j := i + 1 to max do begin
      if (List[j] < best_value) then
        begin
          best_value := List[j];
          best_j := j;
        end;
    end;  // for j := i + 1 to max do.
    List[best_j] := List[i];
    List[i] := best_value;
  end;  // for i := min to max - 1 do.
end;
```

Figure 3: Selectionsort.

Quicksort

The Quicksort algorithm uses a recursive divide-and-conquer technique. As long as the list to be sorted has at least two items, this algorithm splits the list in two. It then recursively calls itself to sort the two smaller sublists. To split a list, Quicksort picks an item from the list to use as a dividing point. It moves all the items that are smaller than the dividing item to the beginning of the list; it moves the other items to the end of the list. It then calls itself recursively to sort the two sublists.

How the Quicksort algorithm selects the dividing item is critical, and there are several possible approaches. The simplest is to use the first item in the list. If the items are initially arranged randomly, there is a good chance the first item will belong near the middle of the list. When the algorithm uses that item to divide the list, the two smaller lists will have roughly equal size. This makes the sublists grow smaller very quickly, so it provides good performance.

On the other hand, if the list is initially sorted, this method produces terrible results. When the algorithm uses the first item to divide the list, it moves all the other items into the second of the sublists (because the dividing item is the smallest item present). The second sublist will contain all the other items, so the algorithm makes little progress. The algorithm will not only be slow, it also will enter a very deep chain of recursion, possibly exhausting the system resources.

A much better method for selecting a dividing element is to pick one randomly. Then, no matter how the items are initially arranged, the item will probably not be too near the smallest or largest. When the algorithm divides the list, the two smaller lists will be significantly smaller than the original, so the algorithm will make reasonable progress.

Quicksort suffers from one more special situation: If the list of items contains many duplicates, the algorithm cannot always separate the list into two smaller lists of roughly equal size. For example, if the list holds 100,000 items with values between 1 and 10, the algorithm will quickly reduce the problem to sorting smaller lists containing thousands of identical values. Quicksort handles these trivial lists very inefficiently. The algorithm can be modified to handle this special case, but most programmers just don't use Quicksort in such circumstances. This situation is uncommon and, in any case, is better handled by the Countingsort algorithm described in the next section.

In practice, Quicksort is extremely fast, so it's the sorting algorithm of choice for most programmers. The Delphi source code for the Quicksort algorithm is shown in Figure 4.

Countingsort

It can be shown that the fastest possible sorting algorithms that use comparisons must take on the order of *N * log(N)* steps to sort a list of *N* items. In fact, Bubblesort and Selectionsort take on the order of *N2* steps. Quicksort needs only *N * log(N)* steps in most cases, though it can use *N2* steps in the worst case.

Countingsort doesn't use comparisons, so it isn't restricted by the *N * log(N)* limit, and, under the right circumstances, is much faster than the others. On the other hand, Countingsort works only under very specific circumstances. First, the items being sorted must be integers; it's difficult to sort strings using Countingsort. Second, the items' values must range over a rel-

```
procedure Quicksort(var List : array of ValueType;
  min, max : IndexType);
var
  med_value: ValueType;
  hi, lo, i: IndexType;
begin
  // If the list has <= 1 element, it's sorted.
  if (min >= max) then
    Exit;

  // Pick a dividing item randomly.
  i := min + Trunc(Random(max - min + 1));
  med_value := List[i];

  // Swap it to the front so we can find it easily.
  List[i] := List[min];

  // Move the items smaller than this into the left
  // half of the list. Move the others into the right.
  lo := min;
  hi := max;
  while (True) do begin
    // Look down from hi for a value < med_value.
    while (List[hi] >= med_value) do begin
      hi := hi - 1;
      if (hi <= lo) then
        Break;
    end;
    if (hi <= lo) then
      begin
        // We're done separating the items.
        List[lo] := med_value;
        Break;
      end;

    // Swap the lo and hi values.
    List[lo] := List[hi];

    // Look up from lo for a value >= med_value.
    lo := lo + 1;
    while (List[lo] < med_value) do begin
      lo := lo + 1;
      if (lo >= hi) then
        Break;
    end;
    if (lo >= hi) then
      begin
        // We're done separating the items.
        lo := hi;
        List[hi] := med_value;
        Break;
      end;

    // Swap the lo and hi values.
    List[hi] := List[lo];
  end;   // while (True) do.

  // Sort the two sublists.
  Quicksort(List, min, lo - 1);
  Quicksort(List, lo + 1, max);
end;
```

Figure 4: Quicksort.

```
procedure Countingsort(var List,
  SortedList: array of ValueType; min, max: IndexType;
  min_value, max_value : ValueType);
var
  i, j, next_index : IndexType;
  count_index : ValueType;
  counts : PCountArray;
begin
  // Create the Counts array.
  GetMem(counts,
         (max_value - min_value + 1) * SizeOf(IndexType));

  // Initialize the counts to zero.
  for i := 0 to max_value - min_value do
    counts[i] := 0;

  // Count the items.
  for i := min to max do begin
    count_index := List[i] - min_value;
    counts[count_index] := counts[count_index] + 1;
  end;

  // Place the items in the sorted array.
  next_index := min;
  for i := min_value to max_value do begin
    for j := 1 to counts[i - min_value] do begin
      SortedList[next_index] := i;
      next_index := next_index + 1;
    end;
  end;

  // Free the memory allocated for the counts array.
  FreeMem(counts);
end;
```

Figure 5: Countingsort.

atively limited set of values. If values range from 1 to 1,000, Countingsort will give good results. If values range from 1 to two billion, Countingsort will be slow; and most computers do not have enough memory to make it work at all.

Countingsort begins by creating a temporary array with bounds that cover the range of items in the list. If the items range in value from 1 to 1,000, the algorithm creates an array with bounds between 1 and 1,000.

Because Delphi doesn't allow a program to dynamically resize arrays, the example program declares the *counts* variable to be a pointer to an array containing 100 million entries. It then uses *GetMem* to allocate space for the *counts* array, as shown in the following code fragment:

```
type
  TCountArray =  array[0..100000000] of IndexType;
  PCountArray = ^TCountArray;

  // Code omitted...

counts : PCountArray;

GetMem(counts,
       (max_value - min_value + 1) * SizeOf(IndexType));
```

The algorithm then examines each item in the list, and increments the *counts* array entry for the item's value. For instance, if an item has the value 12, the algorithm adds one to *counts*(12). When this stage is finished, *counts(i)* holds the number of items in the list that have the value *i*.

The program can then read out the sorted list. It gives the first *counts*(*min_value*) items the value *min_value*. It gives the next *counts*(*min_value* + 1) items the value *min_value* + 1. It continues through the list assigning values until it has completely filled the sorted list.

To sort *N* items that span a range of *M* values, Countingsort uses roughly *2 * N + M* steps. First, it uses *M* steps to initialize the *counts* array entries to zero. Next, it uses *N* steps looking through the list to count the item values. It then uses *N* more steps to fill in the sorted list entries.

If *N* is large and *M* is small, *2 * N + M* is much smaller than *N * log(N)*. For example, suppose a list contains 1 million numbers with values between 1 and 1,000. Then *N * log(N)* is roughly 20 million, while *2 * N + M* is only about 2 million.

Lists like this are particularly troublesome for Quicksort because they contain many duplicate values. In one test on a 166 MHz Pentium, Quicksort took 15 seconds to sort this list, while Countingsort took only 0.15 seconds — one hundredth as long.

Figure 5 shows the Delphi source code for Countingsort.

Conclusion

Every sorting algorithm has its strengths and weaknesses. Bubblesort is fast for short lists that are almost sorted, but slow for others. Selectionsort is easy to program and fast for small lists, but is painfully slow for large lists. Quicksort is fast most of the time, but has trouble if the list contains many duplicate values. Finally, Countingsort only works with integers that span a small range of values, but for those lists it is unbeatable.

Here's a checklist that can help you determine the correct algorithm for your situation:

- If the list is more than 99 percent sorted, use Bubblesort.
- If the list is very small, use Bubblesort or Selectionsort.
- If the list contains integers with a small range of values, use Countingsort.
- In all other cases, use Quicksort.

Choose your algorithm wisely, and one of these four will provide excellent performance in almost any circumstance. ∆

The project referenced in this article is available on the Delphi Informant Works CD located in INFORM\98\JAN\DI9801RS.

Rod Stephens is the author of several books, including *Visual Basic Algorithms* [John Wiley & Sons, Inc., 1996]. He writes an algorithm column in *Visual Basic Developer*; some of the material presented here has appeared there in Visual Basic form. You can reach him on CompuServe at 102124,33 or RodStephens@compuserve.com.

By Cary Jensen, Ph.D.

Hitting the Highlights

Center Stage for Data-Entry Controls

Highlighting is a technique for drawing attention to one or more controls through the use of distinctive colors. There are two primary reasons for using highlighting in your Delphi applications: The first is to make it obvious to users which control has focus, i.e. which one is ready to receive user input; the second is to draw attention to one or more records, based on the data they contain.

This month's "DBNavigator" demonstrates highlighting techniques. It begins by considering single-field controls, which display data from a single field of a single record. Then it moves to the two multi-record controls: DBGrid and DBCtrlGrid.

Data-Based, Single-Field Highlighting

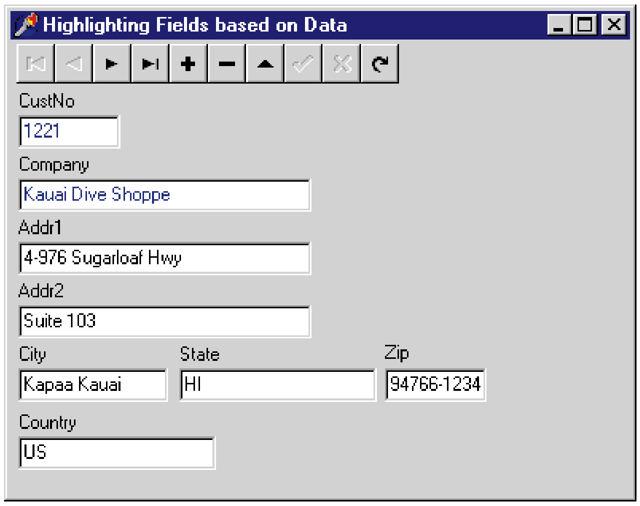
Single-field controls such as Edit and DBEdit display a single value. You can use several techniques to highlight data in a single-field control, based on the data it displays. The most common is to use the *OnDataChange* event handler for a DataSource component. This event handler executes when a DataSet is first opened, as well as each time a new record becomes the current record. From this event handler, your code can evaluate the contents of the record, then set properties of one or more single-field controls.

```
procedure TForm1.DataSource1DataChange(Sender: TObject;
                                       Field: TField);
begin
  if Table1Country.Value = 'US' then
    begin
      DBEdit1.Font.Color := clBlue;
      DBEdit2.Font.Color := clBlue;
    end
  else
    begin
      DBEdit1.Font.Color := clBlack;
      DBEdit2.Font.Color := clBlack;
    end;
end;
```

Figure 1: The HILITE project.

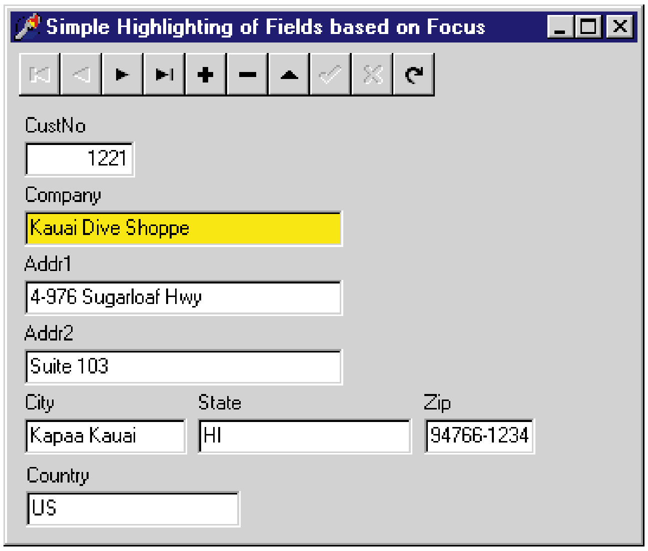
The HILITE project's main form contains a DBNavigator component, along with a series of Label and corresponding DBEdit components. Each DBEdit is associated with a different field from the Customer.db table (from the DBDEMOS alias). The code in Figure 1, which is attached to the *OnDataChange* event handler for *DataSource1*, displays the contents of the CustNo and Company fields in blue whenever the current record represents a US customer, and in black otherwise.

Table1Country is an instantiated *TStringField* that points to the Country column of the Customer table. This code compares the contents of the Country field to the string "US", and if found, sets the *Font.Color* property of the DBEdits that display the customer number and company name to *clBlue*. If the field does not contain the value US, the font color



CustNo: 1221
Company: Kauai Dive Shoppe
Addr1: 4-976 Sugarloaf Hwy
Addr2: Suite 103
City: Kapaa Kauai
State: HI
Zip: 94766-1234
Country: US

Figure 2: The HILITE project displays the contents of US customers' CustNo and Company fields in blue, and all others in black.



CustNo: 1221
Company: Kauai Dive Shoppe
Addr1: 4-976 Sugarloaf Hwy
Addr2: Suite 103
City: Kapaa Kauai
State: HI
Zip: 94766-1234
Country: US

Figure 3: When a field has focus, it's displayed in yellow; otherwise, white is the rule.

of these DBEdits is set to *clBlack*. Figure 2 shows how this form looks when a US customer record is displayed.

Highlighting Focus

If highlighting single-field controls based on data is easy, then highlighting them based on focus isn't much more difficult. However, instead of writing one event handler, you must use two: *OnEnter* and *OnExit*. From within the *OnEnter* event handler, you set a single-field control to its highlighted color; from within *OnExit*, you return the control to its normal color.

The problem with this approach is that you must attach this code to every single-field control on your form. For example, if your form contains eight DBEdits, you would need to attach code to both event handlers for all eight. This is not to say that you must write 16 event handlers. Instead, you could write a single *OnEnter* and a single *OnExit*, and use the same two for each of your DBEdits. Doing so requires that you use the *Sender* parameter of these event handlers to determine which DBEdit to affect, for example:

```
procedure TForm1.DBEdit1Enter(Sender: TObject);
begin
  TDBEdit(Sender).Color := clYellow;
end;

procedure TForm1.DBEdit1Exit(Sender: TObject);
begin
  TDBEdit(Sender).Color := clWhite;
end;
```

This technique is demonstrated in the project named HILITE1(see Figure 3), which was simple in that each of the single-field controls are DBEdits. Because all the fields were displayed using DBEdits, the *Sender* parameter could be cast as a DBEdit to control its color.

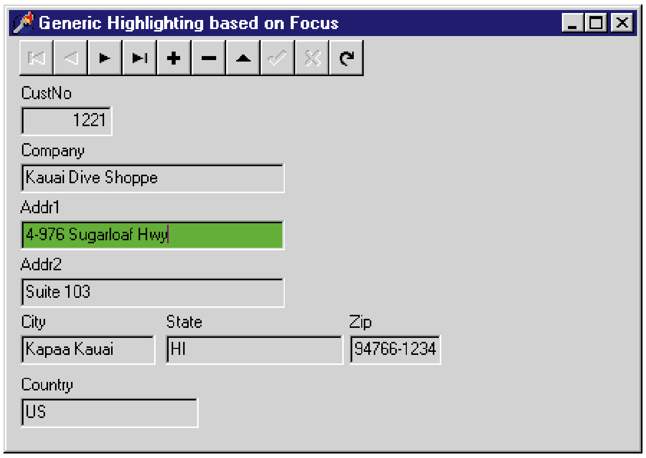
When more than one type of single-field control is used on a form, the technique demonstrated in HILITE1 cannot be used. This is because the *Color* property of a DBEdit is published in the *TDBEdit* class, but is protected in the *TCustomEdit* class, the direct ancestor of *TDBEdit*. Protected properties cannot be accessed at run time. Consequently, to write a generic event handler that will highlight a variety of single-field controls, it's necessary to test for class membership using the **is** operator, then cast *Sender* to the appropriate class before accessing its *Color* property. For example, the code in Figure 4 demonstrates how the *OnEnter* event handler may look.

```
procedure THighlight.Edit1Enter(Sender: TObject);
begin
  if Sender is TEdit then
    TEdit(Sender).Color := clYellow
  else if Sender is TMemo then
    TMemo(Sender).Color := clYellow
  else if Sender is TDBMemo then
    TDBMemo(Sender).Color := clYellow
  else if Sender is TMaskEdit then
    TMaskEdit(Sender).Color := clYellow
  else if Sender is TDBEdit then
    TDBEdit(Sender).Color := clYellow;
end;
```

Figure 4: The *OnEnter* event handler.

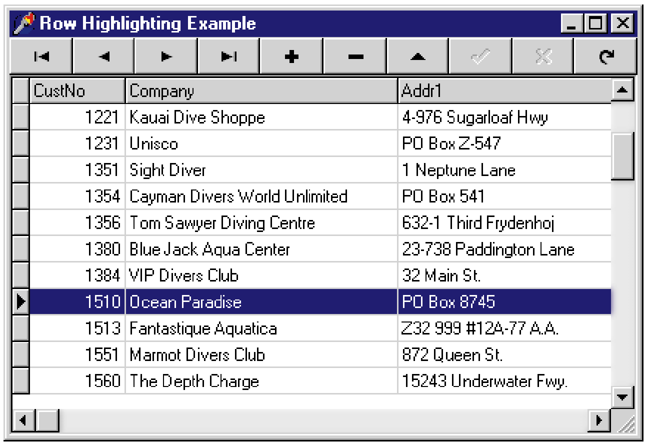
While this might seem like a lot of code to write — especially because you would need to create a similar *OnExit* event handler — there is a solution that permits you to write this code once, then reuse it many times. You can write a generic *OnEnter* and *OnExit* event handler, like the one shown previously. Then, for the form, write a generic *OnCreate* event handler that iterates through all components on the form at run time, and assigns these event handlers to the corresponding event properties of all single-field controls. Such a form can then be saved to the Object Repository. Whenever you want to create a new form that includes single-field, focused-based highlighting, you can use the template in the repository.

Listing One (beginning on page 32) contains the complete code for such a form. This code is associated with a project named HILITE2. This form includes two public fields of the type *TColor*, used to store the highlight color and the normal color. It also declares two event handlers designed to be assigned to the *OnEnter* and *OnExit* event properties. Both are generic, capable of highlighting the most common



CustNo		
1221		
Company		
Kauai Dive Shoppe		
Addr1		
4-976 Sugarloaf Hwy		
Addr2		
Suite 103		
City	State	Zip
Kapaa Kauai	HI	94766-1234
Country		
US		

Figure 5: This form, created from a template, contains generic field-highlighting code.



CustNo	Company	Addr1
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy
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
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.

Figure 6: Setting the *dgRowSelect* flag of the *Options* property causes the DBGrid to highlight the current row.

single-field controls. Finally, an *OnCreate* event handler initializes the colors used for highlighting, then iterates through all components on the form, using a **for** loop. Within this loop, any single-field control encountered is assigned the corresponding event handlers for its *OnEnter* and *OnExit* properties; then the control is initialized to the normal (unhighlighted) color.

If you create this form (or download it, see end of article for details), you can add it to your Object Repository. Then, whenever you want a form to show single-record, focused-based highlighting, you can select this form from the repository, and add your single-field controls. The form shown in Figure 5 was created from a template containing the code in Listing One.

Highlighting Multi-Record Controls

Delphi provides two multi-record controls. The DBGrid is available in all versions of Delphi, while the DBCtrlGrid is available in Delphi 2 and Delphi 3. These controls permit you to display two or more records from a DataSet simultaneously. Consequently, they are useful in a number of database applications.

Highlighting focus in DBGrids. The simplest type of highlighting in a DBGrid is focus-based. This requires no additional code. Instead, you merely set the *dgRowSelect* and/or the *dgAlwaysShowSelection* properties, which are subproperties of the *Options* property. When *dgRowSelect* is set to *True*, the current row is highlighted. When *dgAlwaysShowSelection* is set to *True*, the current row is highlighted, even when the DBGrid doesn't have focus. Figure 6 shows a DBGrid where *dgRowSelect* and *dgAlwaysShowSelection* are set to *True*.

Data-based DBGrid highlighting. In Delphi 1, you control the coloring of individual cells in a DBGrid by attaching code to the *OnDrawDataCell* event property of the DBGrid. This event handler, which is called as each cell in a DBGrid is being painted, is passed four parameters: the DBGrid being painted, the rectangular region being painted, the field associated with the current cell, and the drawing state of the grid.

From within this event handler, you can easily evaluate the data being displayed in the DBGrid, then change the DBGrid's canvas properties to control the color of the cell. During the painting of a particular cell, the cursor of the *DataSet* indicated by the DBGrid's *DataSource* property is associated not with the current record, but with the record being painted. For example, if you want to highlight only those records in the Customer.db table associated with a given country, you can evaluate the Country field of the table, changing the color of the DBGrid's canvas when a record for the country in question is being painted.

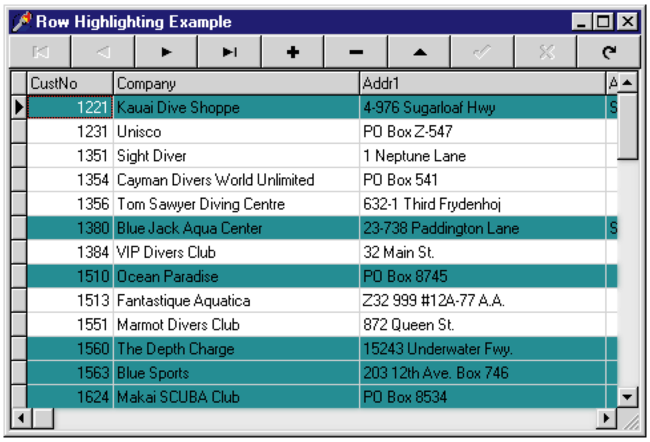
If you assign code to the *OnDrawDataCell* event handler, a cell isn't painted unless you make an explicit call to the DBGrid's *DefaultDrawDataCell* method. This must be called whether or not you made changes to the DBGrid's canvas properties. When calling *DefaultDrawDataCell*, you pass to it the rectangle being painted, the field being painted, and the DBGrid's drawing state. Fortunately, these are parameters of *OnDrawDataCell*, and can be passed without modification.

The following code from the HILITE3 project demonstrates this technique. From within this event handler, the Country field of a Table component is tested against the value "US":

```
procedure TForm1.DBGrid1DrawDataCell(Sender: TObject;
  const Rect: TRect; Field: TField; State: TGridDrawState);
begin
  if Table1.FieldByName('Country').AsString = 'US' then
    DBGrid1.Canvas.Brush.Color := clTeal;
  DBGrid1.DefaultDrawDataCell(Rect,Field,State);
end;
```

The effects are shown in Figure 7. What's particularly interesting about this technique is that it's not necessary to revert the DBGrid's canvas to its original colors. From within this event handler, changes made to the DBGrid's canvas are transitory, and revert to their original values upon exit.

This technique can be extended easily. For example, you can paint specific columns by testing which field is being painted, ignoring the contents of the current record. On the other hand, if you want to paint only certain columns for records containing certain values, you can test both the record's contents and the field being painted.


Row Highlighting Example

	CustNo	Company	Addr1	A
▶	1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	S
	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	S
	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	

Figure 7: The HILITE3 project demonstrates the use of an *OnDrawDataCell* event handler.

While *OnDrawDataCell* is also available in Delphi 2 and 3, these later versions of Delphi provide a more-appropriate event handler to use when painting a DBGrid: *OnDrawColumnCell*, which is similar to *OnDrawDataCell*. You must also call *DefaultDrawColumnCell* from within an *OnDrawColumnCell* event handler to paint the contents of the cell.

The primary difference between *OnDrawDataCell* and *OnDrawColumnCell* is the parameters passed to these methods. *OnDrawColumnCell* is passed more parameters, which can be used to inspect the characteristics of the cell being drawn, and exert greater control over the display of data in the DBGrid. For example, using the *Column* parameter passed to *OnDrawColumnCell*, it's possible to control the column properties of the cell being painted.

While controlling column properties is beyond the scope of this article, the following example demonstrates the same effect as shown in the preceding example — with the exception that the *OnDrawColumnCell* event handler is used instead of *OnDrawDataCell*:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
begin
  if Table1.FieldByName('Country').AsString = 'US' then
    DBGrid1.Canvas.Brush.Color := clTeal;
  DBGrid1.DefaultDrawColumnCell(Rect,DataCol,Column,State);
end;
```

Focus Highlighting of the DBCtrlGrid

The DBCtrlGrid is a free-form, multi-record control introduced in Delphi 2. While the display capabilities of this control are somewhat limited, it's a welcome addition to Delphi's repertoire of data-aware controls.

A DBCtrlGrid doesn't have cells like those contained in a DBGrid. Instead, it has panels on which certain other data-aware controls, such as DBEdits and DBLookupComboBoxes can be displayed. In addition, the only DBCtrlGrid-provided event handler that relates to the painting of the control is *OnPaintPanel*.

At first glance, you might be tempted to set the DBCtrlGrid's canvas properties from within the *OnPaintPanel* event handler, to control the color of individual panels. Doing so, however, is ineffective. Instead, you must explicitly draw onto the panel yourself. This can easily be done using the canvas' *Rectangle* method. (Alternatively, you could draw a bitmap for an even more impressive effect.)

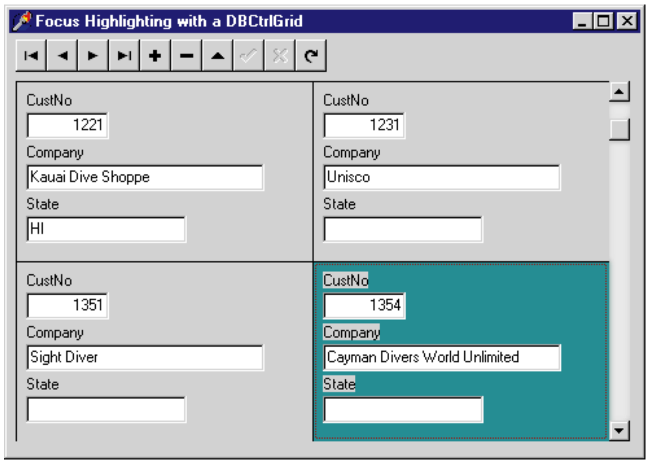
This painting of a DBCtrlGrid is demonstrated in the code that follows. First, the color of DBCtrlGrid's canvas brush is set to *clTeal*. Next, the *Rectangle* method of the canvas is used to paint a rectangle beginning at the upper-left corner of the control, and extending to the lower-right portion. The result is that the canvas background is painted in teal:

```
procedure TForm1.DBCtrlGrid1PaintPanel(DBCtrlGrid:
  TDBCtrlGrid; Index: Integer);
begin
  DBCtrlGrid.Canvas.Brush.Color := clTeal;
  DBCtrlGrid.Canvas.Rectangle(0,0,DBCtrlGrid.Width,
                                  DBCtrlGrid.Height);
end;
```

You might be tempted to conclude that the entire DBCtrlGrid is being painted each time *OnPaintPanel* is called, but this isn't true. In fact, even though this code references the *Width* and *Height* properties of the grid, only the current panel is being painted. In other words, the grid seems to think it's the size of a single panel, and that it's located at the coordinates corresponding to the upper-right corner of that panel. In addition, the *Index* parameter passed to *OnPaintPanel* indicates which panel is being painted. You can compare this to the *PanelIndex* property of the DBCtrlGrid, to highlight the current panel. This is demonstrated in the following code, which is associated with the HILITE4 project:

```
procedure TForm1.DBCtrlGrid1PaintPanel(DBCtrlGrid:
  TDBCtrlGrid; Index: Integer);
begin
  if DBCtrlGrid1.PanelIndex = Index then
    DBCtrlGrid.Canvas.Brush.Color := clTeal
  DBCtrlGrid.Canvas.Rectangle(0,0,DBCtrlGrid.Width,
                                  DBCtrlGrid.Height);
end;
```

Figure 8 shows how this code affects a DBCtrlGrid at run time.


Focus Highlighting with a DBCtrlGrid

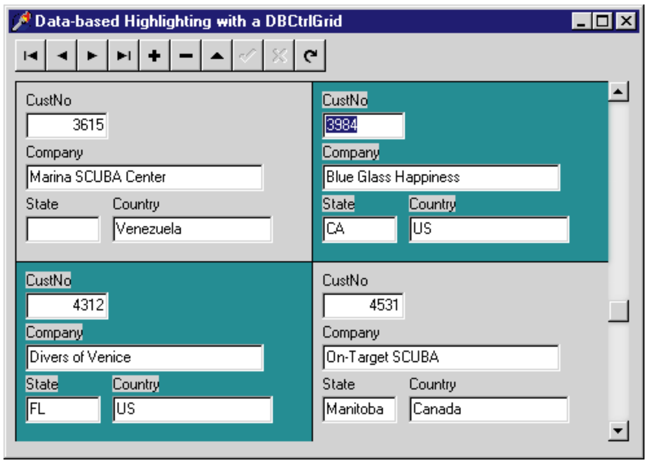
CustNo: 1221
Company: Kauai Dive Shoppe
State: HI

CustNo: 1231
Company: Unisco
State:

CustNo: 1351
Company: Sight Diver
State:

CustNo: 1354
Company: Cayman Divers World Unlimited
State:

Figure 8: Current-record highlighting in a *DBCtrlGrid*.



CustNo 3615
Company Marina SCUBA Center
State Country Venezuela

CustNo 3984
Company Blue Glass Happiness
State CA Country US

CustNo 4312
Company Divers of Venice
State FL Country US

CustNo 4531
Company On-Target SCUBA
State Manitoba Country Canada

Figure 9: The results of panel highlighting.

Data-Based Highlighting in DBCtrlGrids

Highlighting the panels of a DBCtrlGrid is almost as simple as focus-based highlighting. However, instead of using the *Index* parameter and the *TDBCtrlGrid.PanelIndex* property, you use the values of fields in the *DataSet* being displayed in the DBCtrlGrid, as the basis for coloring panels. This technique is demonstrated in the HILITE5 project shown in Figure 9. The following is the *OnPaintPanel* event handler for the DBCtrlGrid in this project:

```
procedure TForm1.DBCtrlGrid1PaintPanel(DBCtrlGrid:
  TDBCtrlGrid; Index: Integer);
begin
  if Table1.FieldByName('Country').Value = 'US' then
    DBCtrlGrid.Canvas.Brush.Color := clTeal;
  DBCtrlGrid1.Canvas.Rectangle(0,0,DBCtrlGrid.Width,
                                   DBCtrlGrid.Height);
end;
```

Conclusion

By highlighting the active field or record, or highlighting selected data, you can improve your user interface, helping your users to more easily locate the information they need. The techniques in this article permit you to easily add these features to your Delphi applications. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\JAN\DI9801CJ.

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, including *Delphi in Depth* [Osborne McGraw-Hill, 1996]. He is also a Contributing Editor of *Delphi Informant*, and was a member of the Delphi Advisory Board for the 1997 Borland Developers Conference. For information concerning Jensen Data Systems' Delphi consulting and training services, visit the Jensen Data Systems Web site at http://idt.net/~jdsi. You can also reach Jensen Data Systems at (281) 359-3311, or via e-mail at cjensen@compuserve.com.

Begin Listing One — The HILITE2 Project

```
unit hilite2u;

interface

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

type
  THighlight = class(TForm)
    procedure ControlEnter(Sender: TObject);
    procedure ControlExit(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    HighlightColor: TColor;
    UnhighlightColor: TColor;
  end;

var
  Highlight: THighlight;

implementation

{$R *.DFM}

procedure THighlight.ControlEnter(Sender: TObject);
begin
  if Sender is TEdit then
    TEdit(Sender).Color := HighlightColor
  else if Sender is TMemo then
    TMemo(Sender).Color := HighlightColor
  else if Sender is TDBMemo then
    TDBMemo(Sender).Color := HighlightColor
  else if Sender is TMaskEdit then
    TMaskEdit(Sender).Color := HighlightColor
  else if Sender is TDBEdit then
    TDBEdit(Sender).Color := HighlightColor;
end;

procedure THighlight.ControlExit(Sender: TObject);
begin
  if Sender is TEdit then
    TEdit(Sender).Color := UnhighlightColor
  else if Sender is TMemo then
    TMemo(Sender).Color := UnhighlightColor
  else if Sender is TDBMemo then
    TDBMemo(Sender).Color := UnhighlightColor
  else if Sender is TMaskEdit then
    TMaskEdit(Sender).Color := UnhighlightColor
  else if Sender is TDBEdit then
    TDBEdit(Sender).Color := UnhighlightColor;
end;

procedure THighlight.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  HighlightColor := clLime;
  UnhighlightColor := clSilver;
  for i := 0 to Self.ComponentCount - 1 do
    if Self.Components[i] is TEdit then
      begin
        TEdit(Self.Components[i]).OnEnter := ControlEnter;
        TEdit(Self.Components[i]).OnExit := ControlExit;
        TEdit(Self.Components[i]).Color :=
          UnhighlightColor;
      end
    else if Self.Components[i] is TMemo then
      begin
        TMemo(Self.Components[i]).OnEnter := ControlEnter;
        TMemo(Self.Components[i]).OnExit := ControlExit;
        TMemo(Self.Components[i]).Color :=
          UnhighlightColor;
      end
    else if Self.Components[i] is TDBMemo then
      begin
        TDBMemo(Self.Components[i]).OnEnter :=
          ControlEnter;
        TDBMemo(Self.Components[i]).OnExit := ControlExit;
        TDBMemo(Self.Components[i]).Color :=
          UnhighlightColor;
      end
    else if Self.Components[i] is TMaskEdit then
      begin
        TMaskEdit(Self.Components[i]).OnEnter :=
          ControlEnter;
        TMaskEdit(Self.Components[i]).OnExit :=
          ControlExit;
        TMaskEdit(Self.Components[i]).Color :=
          UnhighlightColor;
      end
    else if Self.Components[i] is TDBEdit then
      begin
        TDBEdit(Self.Components[i]).OnEnter :=
          ControlEnter;
        TDBEdit(Self.Components[i]).OnExit := ControlExit;
        TDBEdit(Self.Components[i]).Color :=
          UnhighlightColor;
      end;
end;

end.
```

End Listing One

Alan C. Moore, Ph.D.

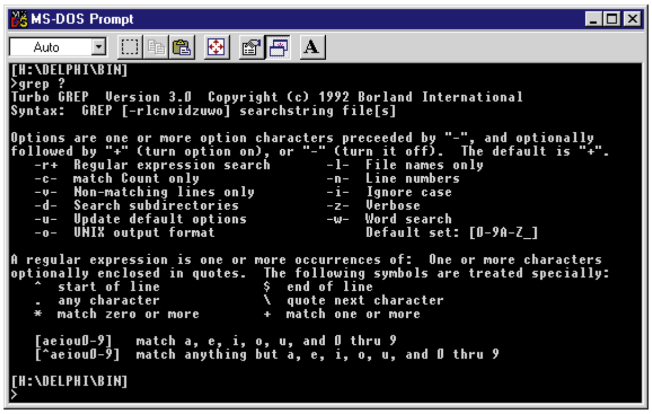
WinGREP

A Powerful Search Tool

Even if you have some UNIX programming in your background, you might not know that GREP is an acronym for Global Regular Expression Print. You certainly *do* know, however, that it's an extremely fast and flexible search utility.

The uses for GREP in Delphi programming are endless. Let's say you plan to use a particular Windows API function, and want to see if and how Borland uses it in the VCL. Or, let's say you have a non-component support class in the current project, but you can't remember the unit in which it's declared. Or you simply want to find the location of every instance of a particular variable or constant in a project. GREP can help with these search tasks and those that are far more elaborate. For example, you can perform searches that filter out specific subsets (e.g. return all words that begin with "work" except those that end with "bak"). A traditional, command-line version of GREP (\Delphi\Bin\Grep.com) is included with Delphi 1 (see Figure 1).

What about WinGREP? How is it different from the Grep.com that ships with Delphi 1, or from other multi-file searching tools? As you know, there's a world of difference between DOS and Windows applications, particularly if we're talking about DOS command-line applications. For the latter, you must remember — and accurately enter — the various option letters. On the other hand, WinGREP provides all the common Windows conveniences we've become accustomed to. In fact, it integrates into many popular IDEs (see Figure 2).



```
[H:\DELPHI\BIN]
>grep ?
Turbo GREP  Version 3.0  Copyright (c) 1992 Borland International
Syntax:  GREP [-rlcnvidzuwo] searchstring file[s]

Options are one or more option characters preceeded by "-", and optionally
followed by "+" (turn option on), or "-" (turn it off).  The default is "+".
   -r+  Regular expression search       -l-  File names only
   -c-  match Count only                -n-  Line numbers
   -v-  Non-matching lines only         -i-  Ignore case
   -d-  Search subdirectories           -z-  Verbose
   -u-  Update default options          -w-  Word search
   -o-  UNIX output format                   Default set: [0-9A-Z_]

A regular expression is one or more occurrences of:  One or more characters
optionally enclosed in quotes.  The following symbols are treated specially:
   ^  start of line             $  end of line
   .  any character             \  quote next character
   *  match zero or more        +  match one or more

   [aeiou0-9]   match a, e, i, o, u, and 0 thru 9
   [^aeiou0-9]  match anything but a, e, i, o, u, and 0 thru 9

[H:\DELPHI\BIN]
>
```

Figure 1: Originally from Turbo Pascal, Grep.com is still included with Delphi 1.

As you can see from the main WinGREP window — Figure 3 shows the results of a search — most of the search options are only a click away. In addition, all commands include fly-over hints. Once you've completed your search, you can jump right to the line of code that contains the text for which you're searching in the IDE. WinGREP supports many popular IDEs, including all three versions of Delphi, C++Builder, and many more (again, see Figure 2 for a partial list.) Let's examine some features in detail.

General Options

WinGREP contains all the general options you would expect. It comes in 16- and 32-bit versions with identical functionality, including support for long file names. With the 32-bit version, you can add WinGREP to the Windows Explorer menu and simply right-click on any file to initiate a search. Of course, you can either perform a search or a search-and-replace operation. And it's easy to navigate files. It also includes command-line parameters — to set WinGREP's startup directory and alternate configuration filenames — and you can customize colors and fonts.

WinGREP's strength lies in its search capabilities, providing all the expected search options and letting you know what's taking place through its status indicator. Best of all, these

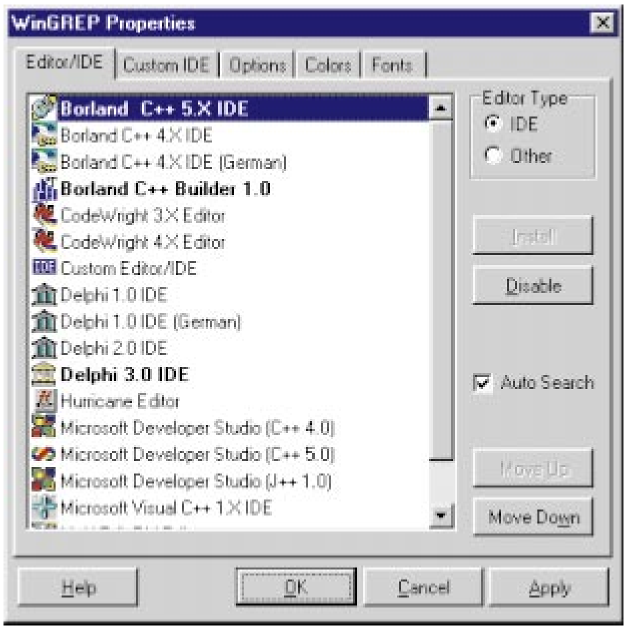
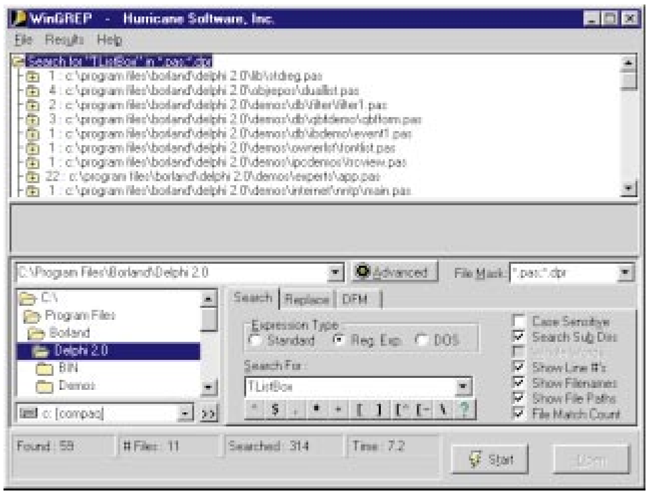


Figure 2: Some of the IDEs that WinGREP supports.



```
Search for "TListBox" in *.pas;*.dpr
 1 : c:\program files\borland\delphi 2.0\lib\stdreg.pas
 4 : c:\program files\borland\delphi 2.0\objrepos\duallist.pas
 2 : c:\program files\borland\delphi 2.0\demos\db\filter\filter1.pas
 3 : c:\program files\borland\delphi 2.0\demos\db\gbldemo\gblform.pas
 1 : c:\program files\borland\delphi 2.0\demos\db\ibdemo\event1.pas
 1 : c:\program files\borland\delphi 2.0\demos\ownerlst\fontlist.pas
 1 : c:\program files\borland\delphi 2.0\demos\ipcdemos\monview.pas
22 : c:\program files\borland\delphi 2.0\demos\experts\app.pas
 1 : c:\program files\borland\delphi 2.0\demos\internet\nntp\main.pas
```

C:\Program Files\Borland\Delphi 2.0 — File Mask: *.pas;*.dpr — Search For: TListBox

Found: 59 | # Files: 11 | Searched: 314 | Time: 7.2

Figure 3: The main WinGREP window.

searches are fast. I compared the same search with WinGREP to one conducted with Norton Navigator. On a Pentium, WinGREP finished in seven seconds, while Norton Navigator took about 10. Delphi 3's search tool took even longer.

During a search, you can set the file size and date filters, and you can select which file attributes to include in the search (see Figure 4). As Figure 3 shows, you can also set options such as case sensitivity, subdirectory search, and expression type. The latter can be a standard expression, a regular expression, or a DOS expression. You can also specify what information to show in the results, including line numbers, file names, and paths. The results of the search are shown as it progresses. Therefore, you can stop the search at any time if you see a particular match you're looking for. You can load and/or save search result sets to or from a file. You can even export search results to text.

Special Delphi Features

WinGREP has a number of special features of particular interest to Delphi programmers, chief among which is its

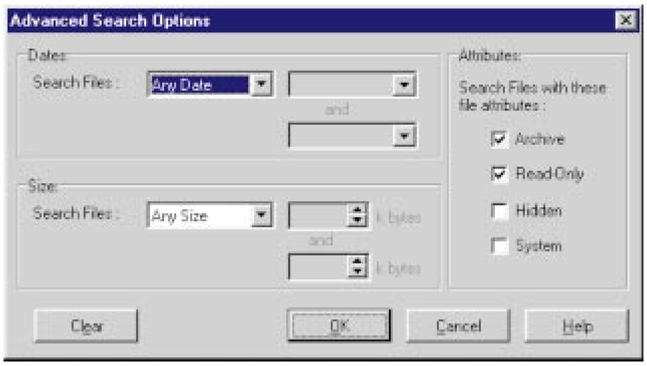


Figure 4: You can set file size, date filters, and which file attributes to include in a search.

ability to search binary .DFM files for text. You can search all the form definition files in your project for specific information, such as color constants, specific components, and so on. As previously mentioned, WinGREP is fully integrated into the Delphi IDE. Therefore, you can perform searches without leaving Delphi, and can jump to the line of source code where the match is found, right in the Delphi IDE. If you purchased WinGREP 3.0 a while back and can't get it to work with Delphi 3, don't worry — you can download a free update patch from Hurricane Software's Web site that will solve that problem.

INFORMANT FACT FILE

WinGREP 3.0 is a powerful, multi-file searching tool for Delphi and other programming languages. It integrates smoothly with many IDEs and comes with its own programmer's editor. It's very fast and flexible, and provides special Delphi support for searching .DFM files.

Hurricane Software, Inc.
2401 SE 7th
Blue Springs, MO 64014
Phone: (888) 946-4737 or (816) 373-9252
Fax: (816) 228-8372
Web Site: http://www.hurricane-soft.com
Price: Single user, US$59. Contact Hurricane Software for multi-user and upgrade information.

Conclusion

I was a bit hesitant to accept this particular assignment, as I felt I already had sufficient searching tools and didn't really need to look at another one. I'm glad I didn't give in to my hesitation; from this point on, WinGREP will be my searching tool. Give it a try. I think you'll find it easy to use, fast, and powerful. Most importantly, it provides excellent support for Delphi. I recommend this product to any Delphi programmer who needs a professional-strength searching tool. Δ

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 at acmdoc@aol.com.

By Alan C. Moore, Ph.D.

Raize Components 1.6

A Significant Upgrade to an Outstanding Component Library

No doubt many readers have heard of Ray Konopka. Among other distinctions, he's the Delphi columnist for *Visual Developer Magazine*, a popular presenter at Borland conferences, and the author of the classic *Developing Custom Delphi 3 Components* [Coriolis Group Books, 1997]. What you may not know is that he is also the chief architect of Raize Components, an impressive series of Delphi components from Raize Software Solutions, Inc.

Raize Components 1.6 includes 50 new components (not including the custom ancestors), that can be divided into two large groups: standard and data-aware. Some, such as *TRzLabel* and *TRzDBLabel*, are direct descendants of components in the Visual Component Library (VCL), while others go deeper to the more basic components, such as *TWinControl* and *TGraphicControl*. I'll discuss the components in three groups: those inherited from *TGraphicControl*, those inherited from *TWinControl*, and those inherited directly from *TComponent*.

TGraphicControl-Related Components

Of the components in this group (see Figure 1), only two are descendants of higher-level VCL components: *TRzLabel* and *TRzDBLabel*; both descend from *TCustomLabel*. The remaining 17 descend directly, or indirectly (through Raize base and custom classes), from *TGraphicControl*. These include the familiar status pane, progress bar, speed button controls, various status controls, and more.

```
TGraphicControl
 ||||||└─TCustomLabel
 ||||||         |└─TRzLabel
 ||||||         └──TRzDBLabel
 |||||└──TRzBorder
 ||||└───TRzCustomStatusPane
 ||||           |||└TRzStatusPane
 ||||           ||└─TRzDBStatusPane
 ||||           |└──TRzCustomGlyphStatus
 ||||           |            |└─TRzGlyphStatus
 ||||           |            └──TRzDBStateStatus
 ||||           └───TRzPollingStatus
 ||||                        ||└─TRzClockStatus
 ||||                        |└──TRzKeyStatus
 ||||                        └───TRzResourceStatus
 |||└────TRzCustomProgressBar
 |||            |└─TRzProgressBar
 |||            └──TRzDBProgressBar
 ||└─────TRzSpacer
 |└──────TSpeedButton
                 └─TRzRapidFireButton
```

Figure 1: Raize components descending from *TGraphicControl*.

While *TRzStatusBar* is a descendant of *TWinControl*, the various panes that can be dropped on it are descendants of *TGraphicControl*. In Figure 2, taken from an educational application I'm currently updating, I use four of the panes (from left to right): *TRzStatusPane*, *TRzResourceStatus*, *TRzClockStatus*, and *TRzGlyphStatus*. You can also drop other controls on the *TRzStatusBar*. (In this figure, you can also see the powerful toolbars you can build with other components we'll be discussing later.)

With these tools, you can create attractive and informative status panels. Best of all, you don't have to bother with the details of getting the system time, resource information, and so forth. Finally, because *TRzStatusBar* is a direct descendant of *TCustomControl* (not Delphi 2/3's *TStatusBar*), it works well in all versions of Delphi, and adjusts automatically to the operating environment.

Of the various pane components, I especially like the *TRzGlyphStatus* component, which I use in my application to give a visual indication of whether a course file is loaded.

Another *TGraphicControl* descendant, *TRzBorder*, provides a simple way to add a custom border to a control that doesn't have one, or add custom dividers within a control.

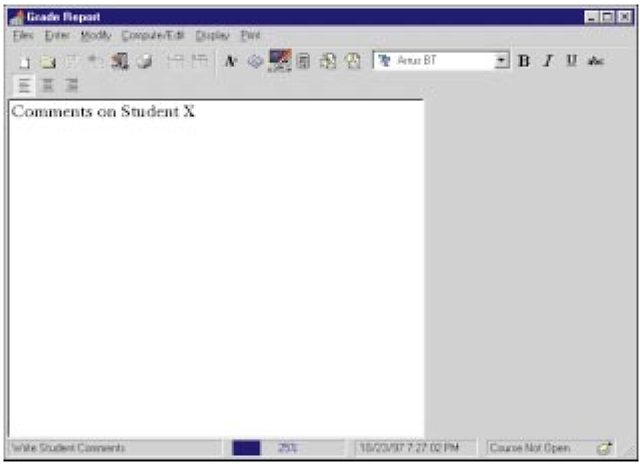


Figure 2: Sample application with *TRzToolbar* at top and *TRzStatusBar* at bottom with four custom panes.

```
TWinControl
 ├─TCustomComboBox
 │  ├─TRzCustomComboBox
 │  │  ├─TRzComboBox
 │  │  ├─TRzColorComboBox
 │  │  ├─TRzFontComboBox
 │  │  └─TRzLineComboBox
 │  ├─TDBComboBox
 │  │  └─TRzDBComboBox
 │  │     └─TRzDBLineComboBox
 │  └─TDriveComboBox
 │     └─TRzDriveComboBox
 ├─TCustomControl
 │  ├─TCustomPanel
 │  │  └─TRzCustomPanel
 │  │     ├─TRzCustomRadioGroup
 │  │     │  ├─TRzRadioGroup
 │  │     │  └─TRzDBRadioGroup
 │  │     ├─TRzPanel
 │  │     ├─TRzSplitter
 │  │     ├─TRzStatusBar
 │  │     └─TRzToolbar
 │  └─TRzTrackBar
 │     └─TRzDBTrackBar
 ├─TCustomEdit
 │  ├─TRzLineEdit
 │  └─TCustomMaskEdit
 │     ├─TDBEdit
 │     └─TRzDBLineEdit
 ├─TCustomListBox
 │  ├─TRzCustomListBox
 │  │  └─TRzListBox
 │  │     └─TRzTabbedListBox
 │  ├─TDBListBox
 │  │  └─TRzDBListBox
 │  ├─TRzCheckList
 │  ├─TDirectoryListBox
 │  │  └─TRzDirectoryListBox
 │  └─TFileListBox
 │     └─TRzFileListBox
 ├─TRzButtonEdit
 │  └─TRzDBButtonEdit
 ├─TRzCustomButton
 │  ├─TRzCustomCheckBox
 │  │  ├─TRzCheckBox
 │  │  └─TRzDBCheckBox
 │  └─TRzRadioButton
 └─TRzSpinEdit
    └─TRzDBSpinEdit
```

Figure 3: Raize components descending from *TWinControl*.

The final *TGraphicControl* component, and descendant of *TSpeedButton*, is *TRzRapidFireButton*, which continuously broadcasts click messages as long as the user holds down the left mouse button over it.

TWinControl-Related Components

Some of the common controls in the *TWinControl* group descend from Borland custom classes. As Figure 3 shows, three custom classes and seven descendant classes are based on *TComboBox*, and two custom classes and six descendant classes are based on *TCustomPanel*.

You'll notice that quite a few components descend directly from *TWinControl* or *TCustomControl*. As Konopka pointed out at the 1997 Borland Developers Conference, not everyone is completely satisfied with some of the components in the VCL that ships with Delphi. The controls in this library offer considerably more functionality and flexibility.

While some Raize components enhance the functionality of standard Windows controls, others allow behavior we expect in the Windows environment, such as being able to resize adjacent controls. Delphi now includes a splitter component to enable this. However, I consider the Raize *TRzSplitter* component superior to Delphi's and to a similar splitter in another Delphi library. With *TRzSplitter*, you simply drop any control onto each of the two sectors at design time, and you're ready to operate.

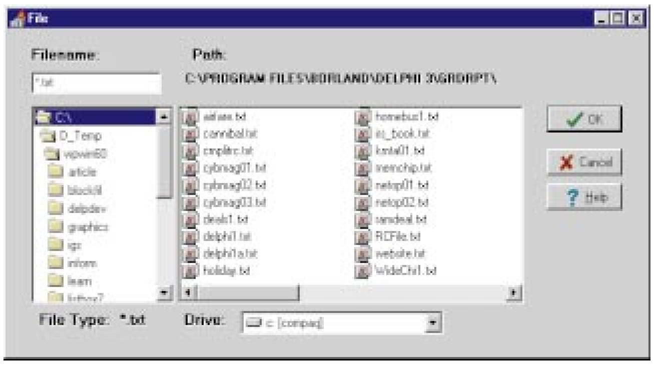


Figure 4: A directory list box and a file list box using *TRzSplitter*.

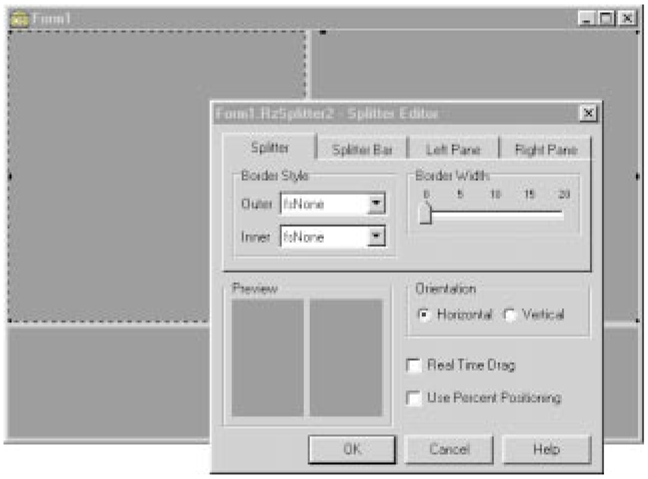


Figure 5: *TRzSplitter* at design time with three individual panels.

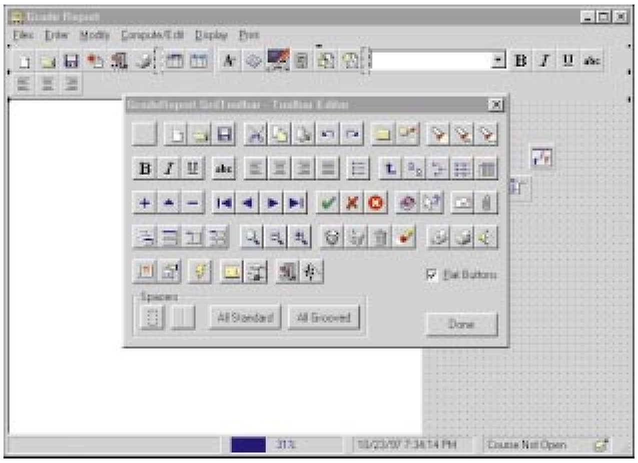


Figure 6: The Toolbar Editor is just one of many component and property editors included in this library.

You don't have to write any code to affect sizing operations. Figure 4 shows a directory list box and a file list box using a *TRzSplitter*. What if you need or want three or more adjustable panels? No problem — simply place splitters within splitters, and you're ready to go.

Figure 5 shows the design-time look of a form that uses two *TRzSplitter*s to create three panes. Both of these splitters have their align properties set to *alClient*. It also shows the powerful component editor you can use to visually manipulate many of the properties.

A few other *TWinControl* descendants deserve special mention. Most of the file-managing components are in this group, including *TRzFileListBox*, *TRzDirectoryListBox*, and *TRzDriveComboBox*. With these components, you can easily set up an attractive and powerful file-managing dialog box. They include support for long filenames, multiple columns, shell icons, and updated three-dimensional glyphs in the style of Windows 95. Figure 4 uses these components and some of the Raize enhanced labels.

One of my favorite components is *TRzToolbar*. As you probably know, toolbars began to appear in applications even before Windows 95 was released. They have become so popular that they're now a standard interface object. However, working with Delphi's built-in toolbars (i.e. with panels and speed buttons) is hardly trivial; you need to make a lot of manual adjustments to get the proper appearance.

The Raize solution is much better. With the Toolbar Editor, you can easily construct a toolbar in minutes. The *TRzToolbar* will even wrap to multiple lines as the width changes. This wonderful component editor (see Figure 6) includes more than 60 commonly used tool bits that you can easily add to your toolbar. Of course, you can use your own custom tool bits, as I've done in this application. You have complete control over how they're arranged, whether the appearance is flat, and whether the control is enabled. You can even add separators and change their order.

TComponent Descendants

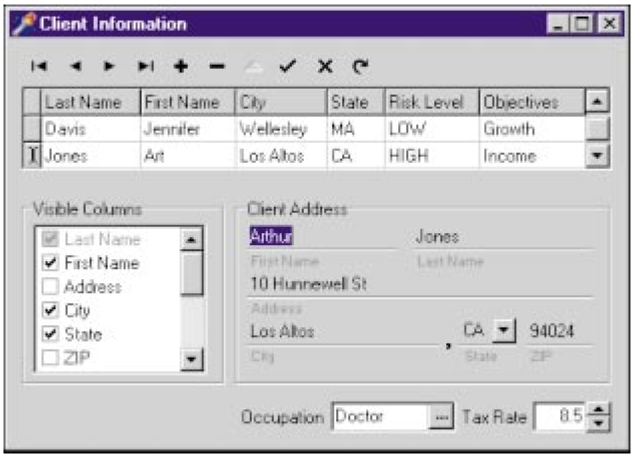
With the exception of *TRzLauncher* (which allows you to launch an application from within your main application, and notifies you when the launched application terminates), the components in this group are specialized dialog boxes. *TRzLookupDialog* provides a method, other than using a combo box, for looking up strings in a list. *TRzDBLookupDialog* is its data-aware cousin; and *TRzSelDirDialog* provides a convenient way to prompt the user for a directory name and the ability to create a new directory.

Data-Aware Components

Of the 50 components in the library, 15 are data-aware (see Figure 7). These include a progress bar; spin edit; list, check, and combo boxes; status panes; a track bar; and more. Most of these components are in the demonstration program included with the product, and are available at the Raize Software Solutions Web site (http://www.raize.com). While some components add data-aware capabilities not previously

Data-Aware Component	Special Feature(s)
TRzDBButtonEdit	Uses embedded buttons to display custom dialog boxes and lookup dialog boxes.
TRzDBCheckBox	Supports multiline captions, 3D text styles, and custom glyphs.
TRzDBComboBox	Supports intuitive keyboard searching and automatic completion.
TRzDBLabel	Includes rotation of text.
TRzDBLineComboBox	Like *TComboBox*, supports intuitive keyboard searching and automatic completion, but appears as a line.
TRzDBLineEdit	Like *TRzLineEdit*, shown as a single line; appearance of the line is programmable.
TRzDBListBox	Features fast keyboard searching.
TRzDBLookupLineComboBox	Enhanced *TDBLookupComboBox* that appears as a single line.
TRzDBLookupDialog	Displays contents of a dataset for user to select; more flexible than a lookup combo.
TRzDBProgressBar	Percentage value displayed can be automatically calculated from the data in a table.
TRzDBRadioGroup	Includes many new ways to customize its appearance including border properties.
TRzDBSpinEdit	Can accept integer and floating point values.
TRzDBStateStatus	Shows current state of dataset.
TRzDBStatusPane	A *FieldLabel* property can specify a static text string to prefix the data stored in a dataset.
TRzDBTrackBar	A *Values* string list property is populated with the values to be written to a database table.

Figure 7: The data-aware Raize components.



Last Name	First Name	City	State	Risk Level	Objectives
Davis	Jennifer	Wellesley	MA	LOW	Growth
Jones	Art	Los Altos	CA	HIGH	Income

- ✓ Last Name
- ✓ First Name
- ☐ Address
- ✓ City
- ✓ State
- ☐ ZIP

First Name: Arthur
Last Name: Jones
Address: 10 Hunnewell St
City: Los Altos
State: CA
ZIP: 94024
Occupation: Doctor
Tax Rate: 8.5

Figure 8: Some of the Raize data-aware components, including line-oriented ones.

available, most add useful functionality and visual versatility. For example, the line-oriented components in Figure 8 enable you to mimic the look of paper forms.

Flexibility through Properties

Not only does this library include a large number of components, it also has a large number of properties to fine-tune those components. This provides a high degree of flexibility. One good example is the *CustomThumb* property used with the track bar components. If you don't like any of the three built-in thumb-style properties, you can create and use your own.

Similarly, you have many options in changing the appearance of most of the controls in the library. Earlier, I mentioned the splitter component. If you want to change its appearance, you can alter the border styles of each pane individually, change the splitter style and width, and so on, using the Object Inspector. You can also use the custom component editor, which provides a preview area that shows the effect of the changes before you accept them.

INFORMANT FACT FILE

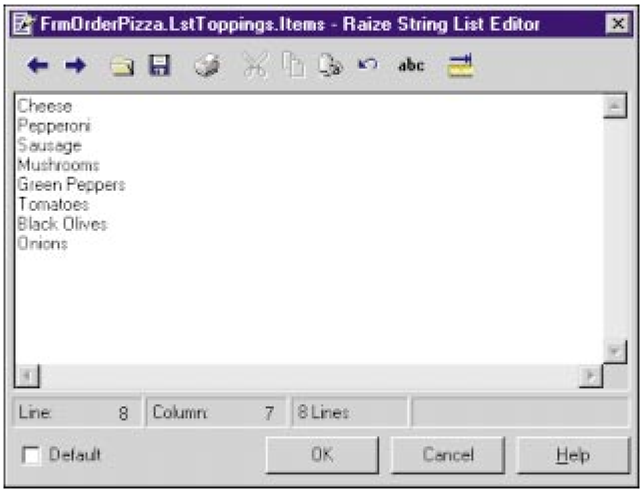
Raize Components 1.6 is an outstanding library of standard and data-aware controls that will add flare and functionality to your applications. It includes full source code, an extensive demonstration program, full support for all three versions of Delphi (and C++Builder), and online support.

Raize Software Solutions, Inc.
2111 Templar Dr.
Naperville, IL 60565
Phone: (630) 717-7217
Web Site: http://www.raize.com
Price: US$199.95 for standard (discounts for five or more copies); US$30.00 for upgrade; US$209.95 for bundle (Raize Components 1.6 and *Developing Custom Delphi 3 Components*).

Furthermore, Raize Components' custom classes make it easy to create your own descendants. You can choose from the fully implemented classes or the custom classes. Additionally, because the full source code for all the components is provided, you can find out how everything works, and get ideas on extending their capabilities.

Documentation and Other Features

One of the exciting aspects of this library is the addition of a few property editors. We discussed the Toolbar Editor earlier. Another one of my favorites is the Raize String List Editor (see Figure 9). It includes many welcome features such as indenting/unindenting, opening/saving to file, cutting/pasting operations, printing, undoing, and so forth.



Cheese
Pepperoni
Sausage
Mushrooms
Green Peppers
Tomatoes
Black Olives
Onions

Line: 8 Column: 7 8 Lines

Figure 9: The Raize String List Editor is a welcome substitute for the one that comes with Delphi.

On the whole, the documentation is pretty good. The manual includes a brief description of each component and each new property. You can learn quite a bit by studying the component source code and example program source code, although the latter doesn't include all the components. As you would expect from the author of *Developing Custom Delphi 3 Components*, the code is very well written.

I wish there had been a bit more information in the manual and in the Help files. Short examples showing each component and property being used within a programming situation would have been a great addition; I guess I've been spoiled by TurboPower's incredibly detailed manuals. However, I really did not find this a major impediment to using these great components.

A Superb Library

This is an outstanding library of Delphi components — solid, reliable, fast, and versatile. You should take a serious look at Raize Components 1.6, particularly if you're developing in 16 and 32 bit, and want your Delphi 1 versions to be as attractive and modern as possible. I find myself relying more on them as time goes on, and I couldn't be happier. ∆

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

A Few Awards ...

... and a Good-Bye

The developer tools market is constantly changing, which makes accurately determining future winners and losers in our industry a difficult, if not impossible, task. Nonetheless, as we begin this new year, I'll attempt to do just that. The envelope please ...

100% Pure Hype Award: Push Technology. What a difference six months makes! Back in early 1997, many industry pundits proclaimed that push technology would bring an end to the Web as we knew it. That prediction seems laughable today; sure, push has its place, but it will ultimately serve a minor role in the Web framework.

50% Pure Hype Award: Java. Java certainly has earned the hype award of the 1990s. Even people who don't know how to use a mouse know about Java; in fact, I can hardly go to my dentist or grocery store without someone asking me about it these days. But when you strip away all the hype, how important is Java as an application development language and platform? Is it "our future" as Sun claims, or is it "just a fad" as some in the Wintel community say? My belief is that the eventual answer will land somewhere between these extremes. Java is an important technology, but for the foreseeable future, it won't dominate the industry. Specifically, look for Java to gain importance on the server side, but fall short on the client side.

Your Father's Oldsmobile Award: Client/Server RAD Tools. While developers are perhaps more productive than ever with client/server RAD tools such as Delphi and Visual Basic, you have to admit they're not the sexy tools they once were. Today the "cool tool" label is given only to Web application development tools such as NetDynamics or Visual InterDev. Nonetheless, losing their sex appeal isn't such a bad thing. After all, the feature set of client/server tools is as mature as it will likely get for some time, enabling developers to focus on stability and reliability rather than the next version of their tool.

It's About Time Award: Distributed Objects. If you've followed object-oriented technology this decade, you've heard for years about distributed object architectures, but it wasn't until 1997 that this technology moved from high-cost, "bleeding edge" solutions to affordable, mainstream technology, as evidenced in Borland's MIDAS. And while the industry will continue to battle over the standard of the future — DCOM or CORBA — it's clear that distributed objects are becoming a legitimate framework on which to build even "budget" applications.

Cool Technology That's Destined to Fail Award: RandomNoise Coda. RandomNoise Coda is a tool that enables you to create "Pure Java" Web sites; that's right — Web sites without using HTML. While a Java-only solution may offer some capabilities that can't be fulfilled by HTML, it's never going to wildly succeed in the long run. No matter how popular Java gets, it will never surpass the document-publishing aspect of the Web.

Destined for Success Award: Extensible Markup Language (XML). While Java isn't about to replace the ubiquity of HTML, this isn't to say that HTML will always be the best platform on which Web applications will be built. On the contrary, HTML has some inherent limitations that make creating and maintaining Web applications kludgy at best. XML, a simplified version of Standard Generalized Markup Language (SGML), offers a solution by providing a bridge between the simplicity of HTML and the flexibility and power provided by SGML or Java. XML offers developers the extensibility needed to *manage* data, not simply *present* it. Additionally, XML extends rather than replaces HTML, which — unlike products such as RandomNoise Coda — doesn't force people to abandon an entrenched standard.

Mother Hen Award: Microsoft. One reason Microsoft is so successful in propagating the use of its technology is the way it provides for developers using their products and technologies. Microsoft developer relations may not be perfect, but they sure beat other companies in this area. The Microsoft Developer's Network (MSDN) program and Microsoft's Web site are prime examples of how a technology company can adequately arm developers with the information they need. Microsoft knows it's not enough to produce great technology; developers need to be equipped with solutions.

Righting the Ship Award: Microsoft. Microsoft was late to focus on the Web, but has since worked hard to play catch-up with Netscape and others in the arena. At least in the area of client-side Web technology, they're clearly emerging as a technology leader. With such innovations as the Document Object Model and Scriptlets, their Dynamic HTML technology solution is extremely well thought out, and is actually a breeze for developers to use. If Redmond extends this innovation into other areas of the Web, watch out!

Young Turks Award: NetObjects and Allaire. With the flood of Web tools on the market, it's inevitable that many of the names you hear today will be gone before the next millenium. As the Web market matures, you'll notice the inevitable "shake out," but when the dust settles, NetObjects, in the HTML-authoring and design market, and Allaire, in the server-side application market, will be among the winners.

That's All He Wrote

I want to end on a personal note: This is my final "File | New" column. At the time of this writing, I am finalizing plans to focus my energies on the Web tools market. Knowing that this professional transition will take me away from the pressing issues facing Delphi developers, I decided to "pass the pen" to another Delphi developer. Alan Moore, a regular *Delphi Informant* contributor, will continue "File | New;" I look forward to reading his insights in future issues. I have enjoyed the opportunity to write this column over the past two years, and I want to thank those of you who have corresponded with me via e-mail during this time. ∆

— Richard Wagner

Richard Wagner is Chief Technology Officer of Acadia Software in the Boston, MA area, and Contributing Editor to Delphi Informant. *He welcomes your comments at rwagner@acadians.com.*