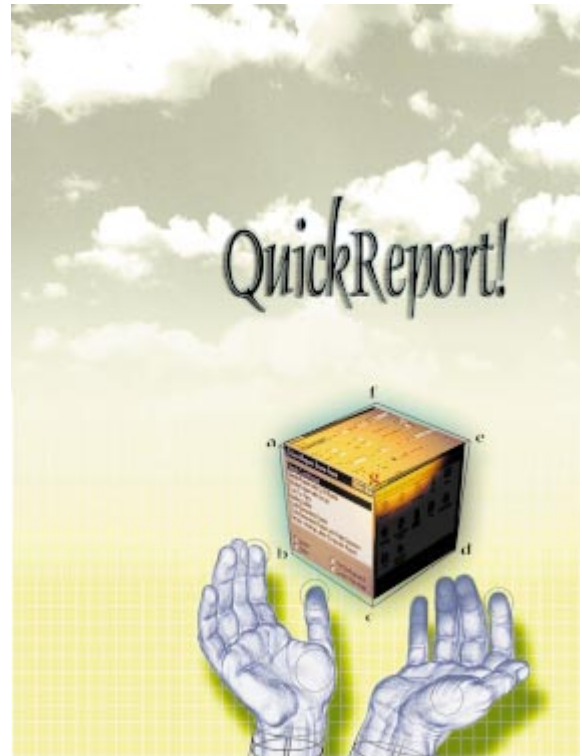


QuickReport!

Delphi's Native Reporting Tool



Cover Art By: Tom McKeith

ON THE COVER



5 QuickReport 2.0: Part I — Cary Jensen, Ph.D.
QuickReport 2.0 is Delphi 3's standard-issue reporting tool; here's how to use it effectively. This first installment demonstrates how to create, print, and preview QuickReports.



10 Extending QuickReport: Part I — Keith Wood
QuickReport is a great add-on for Delphi 2, but these extensions make it even better: a one-piece database grid component, and the ability to print only selected pages.

FEATURES



17 Informant Spotlight
Nice and Nicer — Peter Roth
In other tongues, a "nice" class is one for which the programmer provides needed methods. But what defines "nice" in ObjectPascal, and how can classes be reused — or misused? Here's the story.



23 On the Net
Hop on Pop — Gregory Lee
POP (3) goes Delphi! It may be better to give than receive, but this month we learn how to retrieve e-mail with Delphi, and why the task requires a separate protocol.



27 Columns & Rows
The Paradox Files: Part V — Dan Ehrmann
You know how to manipulate the Paradox file format. But how do you keep it secure? By learning about password protection and Paradox encryption, along with Paradox Table Language options.



31 Greater Delphi
Battening the Hatches — Bill Todd
Another security primer: This third in a series about (encryption-free) InterBase takes on database vulnerability. But not before considering that of the physical surroundings and the OS.



34 Visual Programming
Your First Component Editor — John Ayres
Component editors, as the name implies, are perfect for allowing developers to custom-configure components. Creating this one for *TPanel* can start you down the road to proficiency.



39 OP Tech
Component Creation — Dan Miser
We'd just as soon avoid the possible pitfalls of the intricate process of component creation. To this end, Mr Miser offers these VCL component construction guidelines.



43 Delphi at Work
Serving Many Masters — James Callan
Mr Callan demonstrates various ways DataSource components can function dynamically with DataSets to implement many-to-many relationships.



REVIEWS
55 AppVision's GenerationXpert
Product Review by Alan Moore, Ph.D.



58 Delphi Component Design
Book Review by Alan Moore, Ph.D.



58 Building Internet Applications with Delphi 2
Book Review by Warren Rachele

DEPARTMENTS

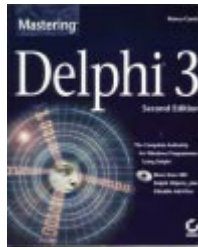
- 2 Delphi Tools**
- 4 Newline**
- 60 File | New** by Richard Wagner



New Products
and Solutions



Mastering Delphi 3
Marco Cantù
SYBEX



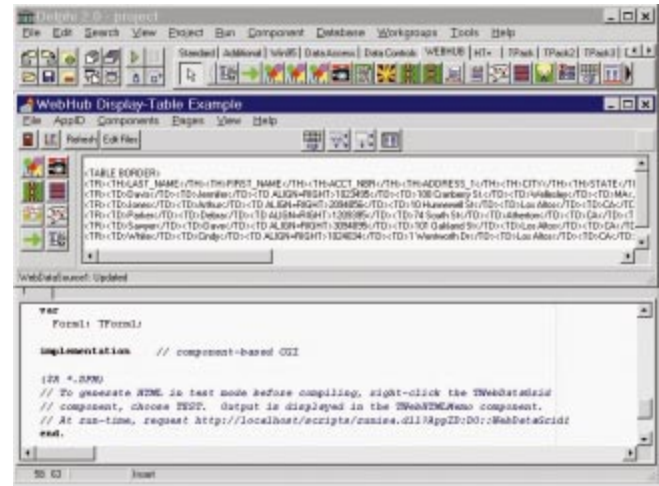
ISBN: 0-7821-2052-0
Price: US\$49.99
(1,476 pages, CD-ROM)
Phone: (510) 523-8233

HREF Tools Launches WebHub 1.0

HREF Tools Corp. announced *WebHub 1.0* is now shipping. Commercially available as an early experience package since October 1995, WebHub uses Delphi 3 Client/Server Suite to provide a framework for building scalable Web applications.

A development environment for Delphi 2 and 3, WebHub offers component-based solutions for multi-user state management, persistent database connectivity, and HTML page generation.

WebHub's components transparently handle the CGI interface for ISAPI, NSAPI, CGI-win, and CGI-bin. It also provides developers with the tools to



write applications for use with Windows NT Web servers. WebHub can be used as middleware for ActiveX, Java, and JavaBean clients, providing functionality not found in JDBC.

Price: Starts at US\$365 for a single developer. WebHub licensing agree-

ments range from US\$95 to US\$1,955 per server, based on capacity.

Contact: HREF Tools Corp., 300 B St., Ste. 215, Santa Rosa, CA 95401

Phone: (707) 542-0844

Fax: (707) 542-0896

E-Mail: WebHubSales@href.com

Web Site: <http://www.href.com>

CT-Connect Now Available from Plains Technology

Plains Technologies, Inc. of Amarillo, TX has released *CT-Connect 1.0*, bringing the FairCom CTree Plus database engine to Delphi.

Written in Delphi for use with Delphi, CT-Connect offers migration of data between CTree Plus data files and Delphi supported databases. It also maintains the methods of Delphi's

TTable, *TDataSource*, and *TBatchMove* components. CT-Connect uses CTree Plus' data types in single- or multi-user mode. Also, CTree Plus programmers will no longer be required to maintain data dictionary files or limited index types.

CT-Connect can be used as an addition to the BDE, or as its replacement. By choosing the CTree and

between table formats; duplicate tables; create bookmarks; display IO performance on reads/writes; and index and buffer use, indexing views on-the-fly, setting filters on-the-fly, and more.

CT-Expert also takes advantage of reading and writing the CTree Plus IFIL and DODA structures in the data files, eliminating the need for external dictionary files.

At press time, a Delphi 32-bit version was planned for release in June from the Plains Technologies Web site.

Price: CT-Connect 1.0 (includes CT-Expert) US\$299; or CT-Expert US\$99.

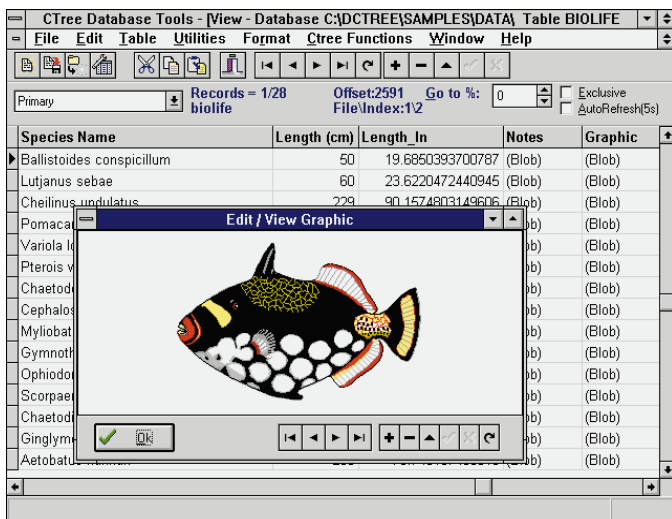
Contact: Plains Technologies, Inc., Wellington Square, 1619 S. Kentucky, Annex D, Ste. 1000, Amarillo, TX 79102

Phone: (806) 359-3650

Fax: (806) 352-6974

E-Mail: frazors@arn.net

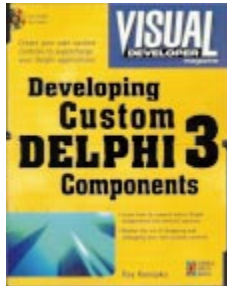
Web Site: <http://www.plainstech.com>



New Products and Solutions



Developing Custom Delphi 3 Components
Ray Konopka
Coriolis Group Books



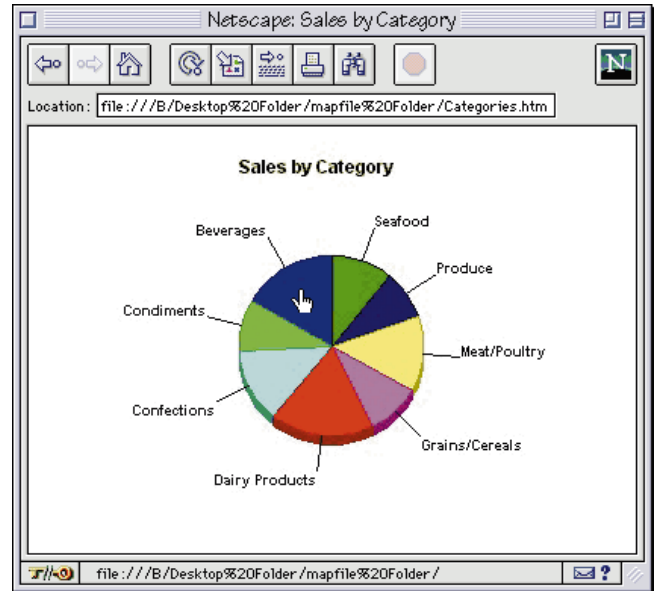
ISBN: 1-57610-112-6
Price: US\$49.99
(725 pages, CD-ROM)
Phone: (800) 410-0192

Pinnacle Publishing Releases Graphics Server 5.0

Pinnacle Publishing's release of *Graphics Server 5.0* adds server-side graphing capabilities to Delphi, Visual Basic, Visual C++, Borland C++, Visual FoxPro, and others. In addition, Graphics Server 5.0 adds support for the Internet.

New features in Graphics Server 5.0 include Web image file output with automatic map file creation for "hot-graphing." Version 5.0 also includes date labels on the X axis, tilt and thickness controls for 3D pie charts, additional box-whisker styles, missing data for overlay graphs, and more.

Graphics Server supports a variety of graph and chart types, including 2D and 3D bar, area, pie, and scatter, as well as log/lin, log/log, high-low-close, open-high-low-close, candlestick, box whisker, gantt, bubble, sur-



face, and times series. Also supported are statistical lines, trend lines, and curve fitting. A built-in end-user interface option also lets users edit and save graphs.

Graphics Server 5.0 will ship with VBX, OCX, and ActiveX support.

Price: US\$349

Contact: Pinnacle Publishing, P.O. Box 4620, Seattle, WA 98104

Phone: (800) 231-1293 or (206) 625-6900

Fax: (206) 625-9102

E-Mail: ppi@pinpub.com

Web Site: <http://www.pinpub.com>

Logic Process Ships DataSentry Data Maintenance Utility and SelfCheck API

Logic Process Corp. of Dallas, TX has released the *DataSentry Data Maintenance Utility*, a Windows 95 and Windows NT 4.0 desktop database maintenance solution for Paradox and dBASE tables.

Extending Borland's Database Explorer, Logic

Process' DataSentry provides database table validation and repair for Delphi developers. Through integration with the BDE and table maintenance libraries, DataSentry supports development of 16- and 32-bit desktop database applications.

DataSentry's data maintenance capabilities include table validation and repair (data and index), as well as table pack, copy, delete, empty, rename, and clone functions. All applicable operations may be performed on any selection of tables in a single session. Tables may be automatically backed up before repair, clone

tables or a Table Structure File may be employed to ensure the success of the rebuild process.

Logic Process has also released the SelfCheck Data Maintenance API, a set of database maintenance routines which can be integrated into your applications. The SelfCheck API enables developers to implement a specific level of validation and repair.

A trial version, site licenses, and electronic distribution with online Help are available.

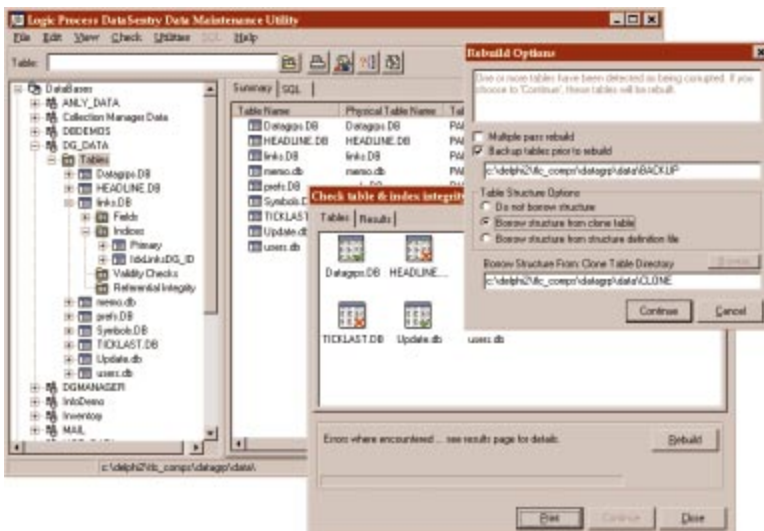
Price: DataSentry, US\$89 (single CPU); current introductory special for DataSentry and SelfCheck bundle, US\$99 (regularly US\$139).

Contact: Logic Process Corp., 10610 Metric Rd., Ste. 110, Dallas, TX 75243

Phone: (214) 340-5172

Fax: (214) 341-9104

E-Mail: sales2@tradelogic.com





Borland Appoints New CFO

Borland has appointed Kathleen M. Fisher, 42, to the position of chief financial officer. Fisher brings more than 15 years of experience with multinational computer and software companies such as AST Research Inc., Western Digital Corp., and Softbank Content Services Inc. She succeeds Paul W. Emery, who recently left Borland. In addition to overseeing Borland's financial management, Fisher will be responsible for Borland's information systems and operations. Fisher's appointment is the latest in a series of new executive appointments by Delbert W. Yocam. Other recent executive appointments at Borland include Rick LeFavre as chief technology officer and John Floisand as vice president of US Sales.

Stops On The World Tour	
Atlanta	Orlando
Boston	Philadelphia
Charlotte	Phoenix
Chicago	Salt Lake City
Columbus	San Diego
Dallas	San Francisco
Denver	Seattle
Houston	Washington, DC
Los Angeles	Calgary
Minneapolis	Ottawa
New Jersey	Toronto
New York	Vancouver

Borland Files Suit Against Microsoft for Unfair Competition

Scotts Valley, CA — Borland has filed a lawsuit in California Superior Court in Santa Clara County against Microsoft. The lawsuit charges Microsoft with recruiting and hiring Borland employees for the specific purpose of damaging Borland's ability to compete with Microsoft in the development tools market, and to slow the company's financial turnaround.

In the past 30 months, Microsoft has hired at least 34 of Borland's top software architects, engineers, and marketing managers, according to a complaint prepared by Wilson, Sonsini, Goodrich & Rosati.

Borland Unveils Multi-Tier Distributed Application Services Suite for Windows NT

Scotts Valley, CA — Borland has announced the Multi-Tier Distributed Application Services (MIDAS) Suite, a set of middleware technologies for Windows NT distributed application development. The result of technologies from Borland and its Open Environment division, MIDAS is compatible with the Entera cross-platform middleware for larger-scale, higher-transaction, and heterogeneous systems.

Application servers built with MIDAS offer 24-hour, seven-day-a-week reliability, opti-

According to the suit, many of these employees now hold strategic positions at Microsoft, mirroring the roles they played at Borland. Microsoft has used large signing bonuses of several millions of dollars and other incentives as a means of wooing Borland employees.

Delphi 3 Includes Microsoft Internet Explorer 3.0

Scotts Valley, CA — Borland announced its Delphi 3 includes Microsoft's Internet Explorer (IE) 3.0 Web browser free of charge. As part of Borland's Golden Gate strategy, Borland's

mized network performance, and a thin-client architecture for application maintenance, distribution, and configuration. Delphi 3 will be the first of Borland's development tools to build MIDAS-enabled applications.

The MIDAS Suite consists of the Business ObjectBroker, the Remote DataBroker, and the ConstraintBroker. The suite is priced at US\$5,000 per server, with volume discounts and special VAR licensing available.

For more details, visit <http://www.borland.com/-midas/> or call (408) 431-1064.

Softbite Announces 1997 Delphi/C++ Builder World Tour with Java

Addison, IL — Borland, The DSW Group, Softbite International, and Informant Communications Group have announced the 1997 Delphi/C++Builder World Tour (with Java).

This World Tour is a five-day event. The first three days focus on Delphi 3 and C++Builder. The last two days cover Java, highlighting JBuilder.

Seminars began in July and

Borland's lawsuit seeks unspecified financial damages and an immediate end to Microsoft's practice of targeting Borland employees in order to hamper the company's ability to compete.

Borland employees and ex-employees are not being sued as part of the legal action.

Windows 95 and Windows NT development tools can create Internet and intranet applications that support IE and other Web browsers, Web servers, and most databases.

Delphi 3 allows developers to build ActiveX components without requiring separate run-time files.

Users can create ActiveX forms and controls from scratch, or turn an existing Delphi or Borland C++Builder VCL component into an ActiveX component. Delphi developers can create ActiveX components for use in many development tools, including IntraBuilder, Visual Basic, Microsoft Office, IE, C++, Java, and PowerBuilder.

For more information on Delphi 3, call Borland at (800) 233-2444, or visit <http://www.borland.com>.

the Informant magazines at the event.

The Delphi World Tour is US\$1,295 for five days; US\$845 for three days; US\$595 for two days; and US\$345 for one day.

Discounts are available for three or more attending from the same company.

For details contact Softbite International at (630) 833-0006, or visit <http://www.softbite.com/tour>.





ON THE COVER

Delphi 3

By Cary Jensen, Ph.D.



QuickReport 2.0: Part I

Delphi 3's Upgraded Reporting Capabilities

Delphi 3 includes QuickReport 2.0 as its reporting tool. Simply put, QuickReport is a VCL-based (Visual Component Library) set of components that permit you to easily include reports in your executables or DLLs. This month's article begins a two part series on QuickReport. This first part covers the basic techniques used to create, print, and preview QuickReports. Next month's article will demonstrate how to create several more-complicated report types, a custom report previewer, as well as generate reports at run time.

The version of QuickReport that ships with Delphi 3 represents a significant upgrade over the 1.0 version included with Delphi 2. As a result, many of the components included in QuickReport 2.0 are different in name and usage from those components previously available. Fortunately, the changes greatly increase QuickReport's utility, and simplify the process of building sophisticated reports.

The new version of QuickReport features:

- a basic report that requires fewer components than the same report in QuickReport 1.0

- a more visual interface
- printing in a background thread (In fact, the *TQuickRep* class implements the *PrintBackground* method, which automates the process of printing a report in a background thread.)
- report generation on-the-fly
- new components (For example, components for printing expressions, displaying the contents of RichText data base fields, as well as a TeeChart component for including business graphs in QuickReports.)
- the ability to combine two or more reports using a composite report
- support for third-party products

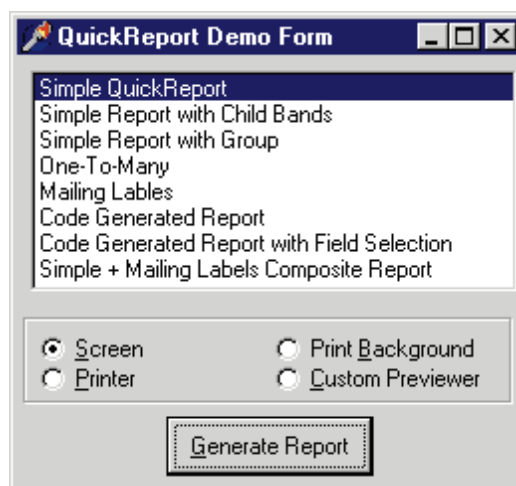


Figure 1: The main form of the QuickReport demonstration project.

This series is intended as an overview, not an in-depth exploration of all aspects of QuickReport. As a result, a selection of reporting techniques are demonstrated, but these represent only a fraction of QuickReport's capabilities. These techniques are demonstrated in the project named quickrep.dpr (see Figure 1).

For additional information on QuickReport, consult the quickrep.hlp file stored in Delphi 3's Help subdirectory. You can also look at qrpt2man.doc, a Word document that serves as the primary source of documentation on QuickReport. (You can

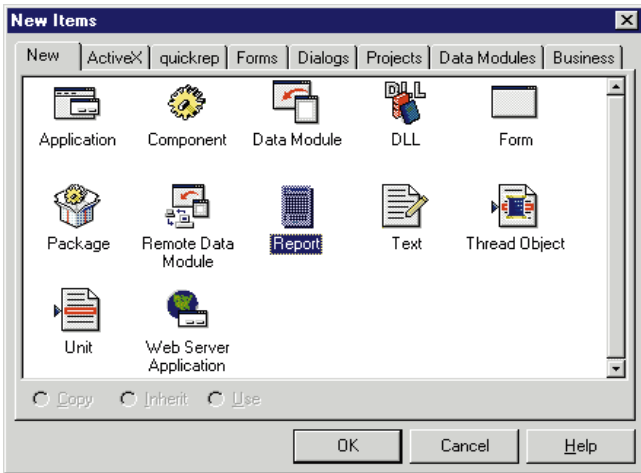


Figure 2: The Report expert icon in the Delphi 3 Object Repository.

use the Windows 95 WordPad if you don't have Word.) This document file can be found in the Delphi 3 \Quickrep subdirectory. In addition, visit the QuSoft Web site at <http://www.qusoft.no>. Consider visiting this site occasionally for updates to the QuickReport units.

QuickReport Basics

You create a QuickReport by adding a QuickRep component to an existing project. To demonstrate a new QuickReport, create a new project and save this project as quickrep.dpr. In all, there are four required steps for creating a QuickReport:

1. add a QuickReport to an existing project,
2. define the bands of the QuickReport,
3. attach the QuickReport to a dataset, and
4. add one or more printable elements to the QuickReport.

In the following sections, we'll review these steps, and investigate QuickReport's additional features.

Adding a QuickReport

There are two ways to add a QuickReport to a project. You can create a new form and add a QuickRep component from the QReport page of the Component palette. Alternatively, you can select **File | New** and click on the Report icon in the Object Repository (see Figure 2). This second technique is easier. I prefer to use a form, because I can more easily add code that is executed when the form and QuickRep component are created and released. Once completed, the QuickReport will resemble Figure 3.

Defining the Report Bands

With QuickReport 2.0, you don't have to explicitly add bands to a QuickRep component. Instead, use the Property Inspector for the QuickRep component to define the bands to appear on the report. You do this by expanding the *Bands* property, then setting its subproperties you want to appear to *True*. There are six *Bands* subproperties (see Figure 4).

In addition to these types of bands, other bands can be added to a report. These include group header and footer bands,

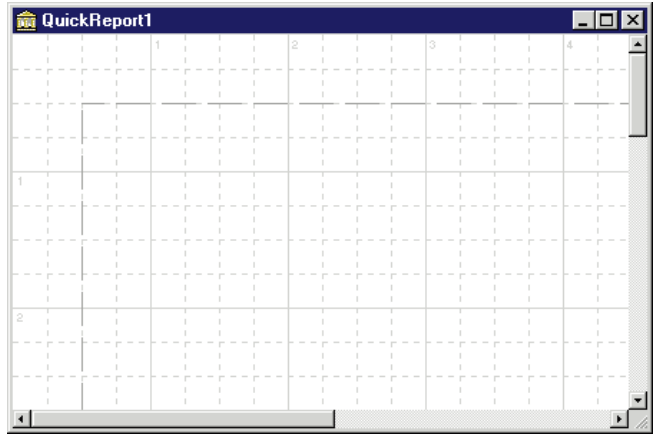


Figure 3: A blank QuickRep component.

Subproperty	Description
<i>HasColumnHeader</i>	Use this band to print column headings for tabular reports.
<i>HasDetail</i>	A detail band is printed once for each record in the master DataSet of the report.
<i>HasPageFooter</i>	This band is printed at the bottom of each page in the report.
<i>HasPageHeader</i>	This band is printed at the top of each page in the report.
<i>HasSummary</i>	This band is printed on the last page of the report. It displays report-wide summary statistics.
<i>HasTitle</i>	A title band is printed once at the beginning of the report and can be used for a cover sheet.

Figure 4: Here are the Bands subproperties.

SubDetail bands, and ChildBands. (These band types are described later in this article, as well as in next month's column.) For our purposes, set the *Bands* subproperties *HasColumnHeader*, *HasDetail*, *HasPageFooter*, and *HasPageHeader* to *True*.

Attaching the QuickReport DataSet

If the report will include data from a database, you must use a DataSet. This DataSet can be placed on the QuickRep

component (or its form), or it can reside on a DataModule. If you place the DataSet in a DataModule, avoid using this same DataSet for user interface elements such as DBGrids. This prevents the QuickReport from being printed in a background thread, or as the user is applying the user interface element. For our report, a single table will be placed on the QuickRep component. Set the table's *DatabaseName* property to *DBDemos*, the *TableName* property to *customer.db*, and the *Active* property to *True*. Next, attach the QuickRep component to the DataSet by setting the QuickRep's *DataSet* property to *Table1*.

If the QuickReport has a Detail band, one copy of the Detail band will be printed for each record in the DataSet. However, you can place filters or use ranges (tables only) on the DataSet to limit the records printed. For example, if you have a table and want to print a particular group of records,

Component	Description
QRLabel	Static, single line of text.
QRDBText	Used to display a field or memo from a DataSet.
QRExpr	Used to display the value of an expression.
QRSysData	Used to display system and report data, such as the date, the time, page number, record number, etc.
QRMemo	Used to define multiple lines of static text.
QRRichText	Similar to QRMemo, except it can include formatted text.
QRDBRichText	Used to display data from a DataSet's formatted memo field.
QRShape	Shape component for a QuickReport.
QRImage	Used to display bitmaps.
QRDBImage	Used to display bitmaps from a DataSet.
QRChart	Used for creating a TeeChart on a QuickReport.

Figure 5: Printable elements on a QuickReport.

use a filter or range to limit the DataSet to that group. Only those records in the filter or range will be printed.

You can also use indexes to control the order that records are printed. Furthermore, you can create calculated or lookup fields for a DataSet, then print them as you would any other field in a table.

Placing Printable Elements

Figure 5 lists the QuickReport components for displaying text or data on a QuickReport. The primary printable components you will use are QRLabel and QRDBText. The QRLabel components are used for simple text, and the QRDBText components are used for fields in the DataSet (including memo, calculated, and lookup fields). The QuickReport shown in **Figure 6** uses QRLabel components for the report title and column headings, while QRDBText components are used for fields. Note also the use of various font properties, including bold face, color, a larger font size for the report title, and the underline font style for the column headings. Creative use of fonts can greatly improve the appearance and readability of your reports.

The report in **Figure 6** also includes several QRSysData components. These components print information about the report, including the date and time of the report printing, page number, and so on. There are two properties of the QRSysData component that you will want to set. The first is *Data*. This defines the type of information to be

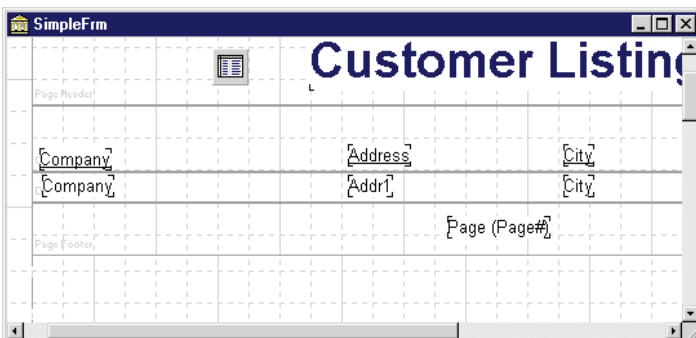


Figure 6: The SimpleFrm form uses QRLabel and QRDBText components, as well as QRSysData components.

included in the field. For example, to display a page number, set *Data* to *qrsPageNumber*. If you want to display the date and time the report was printed, set *Data* to *qrsDateTime*.

The second important property of the QRSysData component is *Text*. This property will prefix the displayed data with a text string. In **Figure 6**, the *Text* property is set to *Page*. Note the use of the space after the 'e' in *Page*. This was necessary to separate the text *Page* from the actual page number. Without this space, the text would run into the page number.

Previewing a QuickReport at Design Time

You can easily preview your QuickReport while you are designing. To do so, right-click on the QuickRep component and select **Preview**. The QuickReport default previewer is displayed, and the report appears. This version of QuickReport is multi-threaded. As a result, the previewer will display the first page of the report as soon as it is formatted. However, QuickReport may continue to format additional pages. Consequently, if the report is long, using the default previewer's VCR-style controls to see the last page of the report will only allow you to view the last formatted page.

QuickReport components, like many other components in Delphi, permit you to add code to event handlers. This code can be used to control various aspects of the printing or previewing of a report. However, because event handlers are not executed when a report is previewed at design time, the image previewed at design time may be quite different from that produced at run time (if you've written event handlers that affect the report output).

Using a QuickReport in an Application

You may never display a QuickRep component to a user at run time. For example, if you place a QuickRep component on a form, you do not display the form to the user. Instead, call the QuickRep's *Preview* or *Print* methods (or alternatively, its *PrintBackground* method).

For example, imagine the simple report is on a form named *SimpleFrm* and it includes a QuickRep component called *QuickRep1*. The following code creates this form, previews the report using the default QuickReport previewer, and releases the form when complete:

```
SimpleFrm := TSimpleFrm.Create(Self);
SimpleFrm.QuickRep1.Preview;
SimpleFrm.Release;
```

If you wanted to print the report instead, you could use:

```
SimpleFrm := TSimpleFrm.Create(Self);
SimpleFrm.QuickRep1.Print;
SimpleFrm.Release;
```

Figure 7 shows a simple report in the default QuickReport previewer. This previewer is automatically created and displayed when you call the QuickRep method *Preview*.

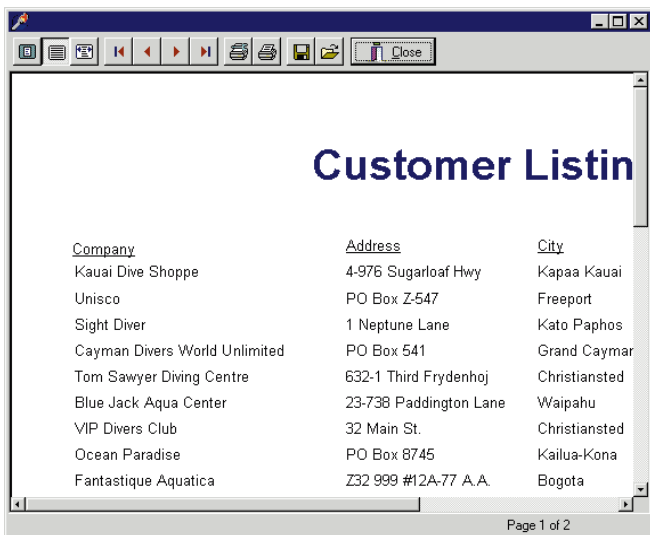


Figure 7: The default QuickReport previewer displaying a QuickReport.

Printing in a Background Thread

With QuickReport 1.0, it's not possible to print a report in a background thread. This is because the QuickReport component in version 1.0 can only be linked to a DataSource, and not directly to a DataSet. You can't have a DataSource point to a DataSet that is active in a background thread.

Now that version 2.0 links directly to a DataSet and doesn't require a DataSource, threaded QuickReports are possible. Normally, this would require you to create a separate Session and Database for the thread. Fortunately, this is not necessary. The *TQuickRep* class includes *PrintBackground*, a method that performs all the necessary steps. Consequently, this single line of code is all that is required to print a QuickReport in a background thread:

```
SimpleFrm.QuickRep1.PrintBackground;
```

Background thread printing requires you to take a number of steps to ensure the report works properly. First of all, as mentioned previously, the report can't use DataSource components. While in a single table QuickReport this is easy enough; however, reports that require multi-table links pose a problem. Typically, to link one table to another you must use a DataSource that points to the master table of the link. (This DataSource is assigned to the detail table's MasterSource property.) Because you can't use a DataSource, you must perform the link through code. (The creation of a master-detail QuickReport that can be used in a background thread will be covered in next month's article.)

When printing in a background thread, you'll need to consider when to release the report. This is especially important because you don't want the report to be created automatically. Instead, you'll want to create the report when it's needed, and release it when it's finished. By only creating the report when needed, you limit the resources required by the application.

The simple solution to this problem is to create the report immediately prior to printing it, and then free the form on which it appears from its *OnClose* event handler. You do this by setting the *Action* parameter of the *OnClose* event handler to *caFree*. This is demonstrated in the code from the *OnClose* event handler for the form SimpleFrm:

```
procedure TSimpleFrm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;
end;
```

Using Child Bands

There is an additional band style that you'll often include in a report — even when it contains only a single DataSet — the ChildBand.

A ChildBand is used to display data from a record that either varies in size, or may be absent altogether. For example, if you have a record that includes a RichText memo field, you might want to place the QRDBRichText component in a ChildBand. This ChildBand will automatically resize at run time, depending on the size of the memo being printed. If the memo field is empty, the ChildBand is simply not printed.

Although the QReport page of the Component palette includes a QRChildBand component, you won't actually place this component in most cases. Instead, you'll set the *HasChild* property of an existing band (a Detail band, a SubDetail, or even another ChildBand) to *True*. For example, if you set the *HasChild* property of a Detail band to *True*, QuickReport automatically places the ChildBand below the Detail band. You can then resize the ChildBand as necessary, and place printable QuickReport components within it (see Figure 8).

A ChildBand itself has a *HasChild* property. This permits a ChildBand to have a ChildBand, which in turn may have another ChildBand. In fact, the QuickReport in Figure 8 has three ChildBands. The first ChildBand displays the Notes field (a memo field) from the Biolife.db table that ships with Delphi. The second ChildBand contains a fixed-

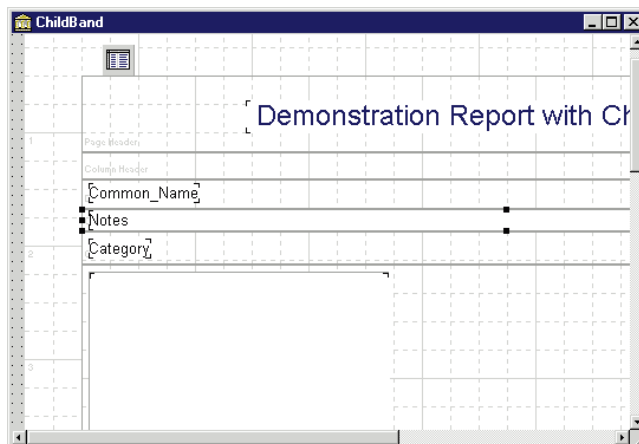


Figure 8: The ChildBand form from the QuickRep project.

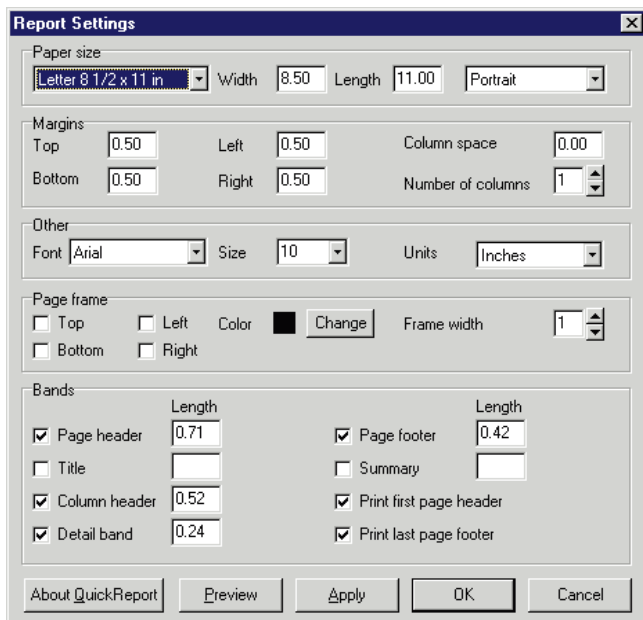


Figure 9: The QuickReport Report Settings dialog box.

sized field (Category). The third ChildBand contains a QRDBImage, which is used to display the bitmap stored in the Biolife.db Graphic field.

While a given ChildBand is not printed if the field or fields it contains are empty, the suppression of a given ChildBand has no effect on other ChildBands associated with the same data record. For example, if the Notes fields for a particular record are empty, the first ChildBand is not printed. However, as long as the Category and Graphic fields of that same record contain data, those ChildBands are printed.

Using the Report Settings Dialog Box

QuickReport comes with a component editor that permits you to easily modify QuickReport's properties. To display this editor, right-click a QuickRep component and select **Report Settings**. QuickReport responds by displaying the dialog box shown in [Figure 9](#).

From this dialog box you can easily adjust paper size and orientation, margins, and column spacing, as well as set the default font. You can control the presence, size, and color of a page frame, select bands for the report and adjust their size, and preview the modified report.

Conclusion

Next month, we'll conclude this series with a look at how to create several different types of reports, including mailing labels, one-to-many reports, and composite reports. You will also learn how to create custom QuickReport pre-viewers, as well as how to create new reports at run time. ▲

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97AUG\DI9708CJ.

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]. Cary is also a Contributing Editor of *Delphi Informant*, as well as 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://gramercy.ios.com/~jdsi>. You can also reach Jensen Data Systems at (281) 359-3311, or via e-mail at cjensen@compuserve.com.





ON THE COVER

Delphi 2 / QuickReport

By *Keith Wood*



Extending QuickReport: Part I

Custom Features Can Save Trees and Sanity

QuickReport is a great add-on that allows reports to be written in native Delphi code and compiled into an executable. It prints information from databases, complete with field-change breaks and grand totals.

QuickReport could be improved, however. For example, the default report previewer doesn't allow the printing of only selected pages. Also, a database grid component would be a great addition. For simple reports, it's sometimes annoying to drop each column separately, then each header, and then try to line everything up. Instead, we could just drop one grid, and have it automatically display the fields from the data source.

Customizing the Previewer

QuickReport provides a default report previewer that allows us to view the report at normal size, at full width, or with the whole page scaled to fit. We can sequentially "step through" the pages of the report, or jump straight to the first or last pages. Reports can be saved, then reloaded at a later time; and of course, the entire report can be printed.

But what if we want to view the report at other resolutions? Or jump to a particular page in the body of the report? Or print only the one page in which we're really interested? Fortunately, QuickReport gives us the tools to write our own report previewer, so we can have it do whatever we want.

The most important piece in writing a new previewer is the QRPreview component. This handles the presentation of the QuickReport; we need only tell it what we want it to do. Open a new form, drop a Panel component on it, and set its *Alignment* property to *alTop*. This holds the controls through which we can interact with the preview component (see [Figure 1](#)). Then drop a QRPreview component onto the body of the form, and set its *Alignment* property to *alClient*, so it fills the remainder of the form.

Zooming In

As mentioned, we want to add the ability to view the report at other resolutions, specifically 50%, 75%, 150%, and 200%, in addition to the standard 100%, full width, and whole page. Rather than designate buttons for these choices, we use a combo box. This provides a text description of the zoom factor, and is more explicit than button images.

The QRPreview component provides a property and two methods to deal with the magnification of the report being shown: *Zoom*, *ZoomToWidth*, and *ZoomToFit*. The first can be assigned the numeric zoom factor as a percentage, while the second fits the page's full width into the viewer, and the third shows the whole page. Mapping from the selection in the combo box to the



Figure 1: The toolbar for the new previewer.

```

{ Change the zoom on the preview }
procedure TfmQuickPreview.ZoomChange(iSize: Integer);
const
  iZoom: array [1..5] of Integer = (50, 75, 100, 150, 200);
var
  i: Integer;
begin
  with grpPreview do
    case iSize of
      1..5: Zoom := iZoom[iSize];
      6:   ZoomToWidth;
      7:   ZoomToFit;
    end;

    { Update menu items and combo box }
    cmbZoom.ItemIndex := iSize - 1;
    for i := 0 to ComponentCount - 1 do
      if Components[i] is TMenuItem then
        with TMenuItem(Components[i]) do
          Checked := (Tag = iSize);
        end;
    end;

    { Change the zoom on the preview via the combo box }
    procedure TfmQuickPreview.cmbZoomChange(Sender: TObject);
    begin
      ZoomChange(cmbZoom.ItemIndex + 1);
    end;

    { Change the zoom on the preview via the pop-up menu }
    procedure TfmQuickPreview.MenuZoomClick(Sender: TObject);
    begin
      ZoomChange(TMenuItem(Sender).Tag);
    end;
end;

```

Figure 2: Zooming a report in the previewer.

appropriate property or method is easily done by using a case statement based on the index of the item selected (see [Figure 2](#)).

To complicate matters, we're adding a pop-up menu that also allows the zoom factor to be changed. The same options appear on the menu, and can be processed like those of the combo box by assigning appropriate values to their *Tag* properties, i.e. their correspondence to the indexes in the combo box list (plus one to facilitate other processing). The menu items have one entry checked to show which resolution is currently in use. To keep this and the combo box synchronized when either changes, we set the combo box item index from the passed *Tag* value (after subtracting one, because the indexes start at zero). We then search through all the menu-item components, and set the *Checked* flag when the item's *Tag* is equal to the current value, clearing all the others.

Page Navigation

Another of our wishes is the ability to immediately jump to any page. The QRPreview component provides a *PageNumber* property we can read to obtain the number of the currently displayed page, or write to to move to a different page. This property can be set through the previewer.

We add the standard page-navigation buttons to the control panel on our form: first, previous, next, and last pages. In the middle of these, we place an edit box for entry of a particular page number, and a label to show us the total number of pages. This latter value is obtained from the *QRPrinter* object.

QuickReport automatically creates a *QRPrinter* object for each application, and uses it to generate the reports for both printing and previewing. One of its properties is *PageCount*, which gives us the total number of pages in the current report.

The current page number is set as the initial value of the edit box. This can be overwritten by the user to move to another page, but must be checked so that it doesn't exceed the bounds of the report. The new page number can't be checked on the *OnChange* event of the edit box, because the user may still be entering the value. This check must wait until the user exits the field, triggering an *OnExit* event, or until is pressed, generating an *OnKeyDown* event.

In each case, the application tries to convert the entered text to a number, and if successful, checks that the number lies within the range of pages for the current report. If either of these fails, the application aborts the procedure and returns to the original page number. If everything is okay, the previewer shows the specified page.

The navigation buttons simply set the page number in the previewer component appropriately. There is no need to check for reaching the start or end of the report when processing the previous- and next-page buttons, because they are enabled only when they are valid.

Menu items appear in the pop-up menu corresponding to the navigation buttons. To make these easier to reach with the mouse, a break is inserted in the menu, causing them to be shown in a new column alongside the zoom options. Set the *Break* property of the first-page menu item to *mbBarBreak*.

A common procedure, *CheckPages*, is called to alter the display after any change to the current page. It updates the edit box with the current page number, and enables or disables the navigation buttons and menu items, according to the position of the displayed page. This is done by comparing the current page number with the start or end of the report. All of this can be seen in [Listing One](#), beginning on page 15.

Printing

Our last big wish is to be able to print only portions of the report. In the standard previewer, the entire report is printed when we press the print button. This calls the *Print* method of the *QRPrinter* object. Further examination reveals *FromPage* and *ToPage* properties of *QRPrinter*; these allow us to specify which pages are to be printed from the report.

We add an edit box to the control panel, allowing entry of the pages to be printed, then add a button to start the printing process. Another button invokes the printer-setup dialog, allowing us to change the settings just before printing.

Rather than allowing for just a single range of pages, we want the ability to list pages by individual value or by range, or any

combination of these at once. Thus, the text entered consists of a list of page ranges separated by commas. Each range can be a single page number, or a range of pages in the format *nn-nn*. Of course, many things could be entered that don't fit this structure, so the previewer must first check that the format is correct before printing anything.

Initially, the entered text is separated at the commas, and placed into a string list. Each entry in that list is then processed to extract the single page number, or range of pages separated by a hyphen. A custom exception, *EPages*, is used to handle any errors encountered during this checking (other than those related to conversion). This keeps the code clean and straightforward. Even when we've extracted valid numbers for the text, there is still the possibility that they're outside the range of values applicable for this report. Again, an exception is raised if they aren't valid.

Once all the specified pages have been checked and found to be correct, the previewer loops through each selection and asks the *QRPrinter* object to print each range in turn. See [Listing Two](#), beginning on page 15.

Finishing Up

We now have our custom report previewer, with all the added functionality that we desired. A few finishing touches and it's ready to go.

The name of the previewed report appears in the window's title bar. The name is obtained from the *Title* property of the *QRPrinter* object. Buttons for saving and reloading reports are added to the control panel, and the standard dialog boxes are used to ask for a filename before invoking the *Save* or *Load* methods of the *QRPrinter* object. Don't forget to add some hints to the buttons and the edit boxes, as a reminder of what they do.

The *QRPrinter* object also has properties that allow more control over the functions available in the standard previewer. These determine whether the print, save, and reload buttons are enabled. It's a simple matter to check these, and add the same abilities to our previewer.

This form is completely generic — a prime candidate for addition to the gallery or repository in Delphi. It could then be easily added to another project that uses a QuickReport.

Only a few steps are required to make use of the new previewer in an application. First, add the preview form to the project, selecting it from the gallery or repository, and add the unit name to the **uses** clause in the **implementation** section of the main form. Then add a new procedure to the **private** section of the main form:

```
private
  procedure PreviewReport;
```

This procedure creates a new previewer and shows it for the requested report:

```
{ Use alternate quick report preview }
procedure TfmQRExtensions.PreviewReport;
begin
  with TfmQuickPreview.Create(Self) do
  try
    ShowModal;
  finally
    Free;
  end;
end;
```

Finally, tell QuickReport to use the new previewer by assigning this procedure to the *OnPreview* event of the *QRPrinter* object:

```
qrPrinter.OnPreview := PreviewReport;
```

Normally this is done in the *OnCreate* event of the main form. That's all there is to it!

A Database Grid

QuickReport provides components to easily report data from a database. Simply drop a *QRDBText* component onto the appropriate band, attach it to a data source, and away we go. Unfortunately, we need to drop one onto the form for each database field we wish to display, making sure they all have the same color and font, and that they all line up. Then we must do the same for the column headings, using *QRLabel* components.

It would be so much easier if we had something like the database grid that Delphi provides for use in normal forms. Then we could just place one of these and attach it to the data source. It would automatically display all the fields, without any alignment problems.

QuickReport does provide a way of implementing this sort of functionality. Just derive a new component from *TQRCustomControl*, and override a couple of methods. The new component is called *TQRDBGrid*, and is described in detail later.

Grid Properties

Obviously, we need a *DataSource* property to identify the data source. We also want a *Color* property and a *Font* property to control the basic appearance of the cells. *BorderStyle* and *Pen* properties allow us to control the frames around the cells, while *AlignToBand* and *Alignment* properties shift the entire grid within its report band. Several of these properties are already present in the *TQRCustomControl* component; all we need to do is expose them.

Each column takes its appearance from the fields of the attached data set. The fields' *Visible* property determines whether they appear in the grid. Then their *Alignment* and *DisplayFormat* properties (if applicable) control how they are presented. The width of each column in the grid comes from the *DisplayWidth* property of the fields.

The *BorderStyle* property allows us to display no frame around the grid cells, to display only the vertical lines at the sides of the cells, or to draw a full box around them. If any

sort of border is requested, the *Pen* property is used to describe the appearance of that border.

The *AlignToBand* property determines whether the entire grid component is shifted when reporting, relative to the band on which it lies. When *False*, the grid is printed as it appears in the design phase. When *True*, the *Alignment* property causes the grid to be positioned appropriately within its band.

One event is defined for the grid. The *OnPrint* event is triggered for each cell of the grid just before it's drawn. It has the following definition, and allows the text, color, and/or alignment of the cell to be altered before it's printed:

```
{ User hook before printing a cell }
TPrintGridEvent = procedure(Sender: TObject;
  FieldName: string; var Value: string; var Color: TColor;
  var Alignment: TAlignment) of object;
```

The name of the current field is passed as well, to assist in identifying when any action should take place. One possible use of this event would be to change the color of certain cells in a particular column to highlight their values, such as cells representing customers who are nearing their credit limits.

Displaying the Grid

Two methods are needed to override the display grid: *Paint* and *Print*. The first is called whenever the grid needs to be drawn on the screen during the design phase. The second is called by QuickReport every time the grid is to be drawn onto the report.

If the grid is not attached to an active data set through its *DataSource* property, then each of these methods calls the default routines inherited from *TQRCustomControl*. This results in the component name being shown during design, and the text "Not connected" being displayed in the report. Without field definitions, it's difficult to draw a meaningful grid!

The width of each column in the grid is derived from the *DisplayWidth* property of the field, which specifies the number of character places to use. This is multiplied by the width of an "n" in the requested font (a standard character width) to obtain the actual column size in pixels. If the total width of all the columns is less than the current width of the entire component, then the component is shrunk to fit just these columns. If the columns extend beyond the width of the grid, then those columns are lost and are not displayed.

Then we step through each field defined, to the data set attached to *DataSource*. If the field is visible, it's drawn onto the screen or report. Any procedure attached to the *OnPrint* event is called to allow final changes to the cell's text, color, or alignment. Then the cell is actually drawn, taking into account the font, color, and alignment attributes set earlier. Finally, any boxing of the cell is performed, using the *Pen* from the grid. This process is fairly involved, but interested readers may follow it in the code accompanying this article (see end of article for download details).

Column Headings

Okay; the grid prints the contents of the visible fields from each database record. Now, how do we know which field is which on the report? Obviously, we need column headings.

Rather than messing with *QLabel* components, and having problems maintaining alignment of headings and columns when we change the report, why not adapt the grid to show the headings as well? All the column sizing and processing remains exactly the same, with the only change being to print the *DisplayName* property of the respective fields, instead of their values.

We also would like the heading grid to be synchronized with the data grid; any changes to the latter should immediately be reflected in the former. Then all we need to do is ensure the horizontal alignment of the two grids, and alter the underlying data set. This is achieved by adding a *HeaderFor* property, which can be set to point to another grid. On doing so, it links to the same data set as the attached grid, and displays the column headings for the defined fields with the same alignment and field widths. The background color for the header can be set separately from that of the data grid. The presence or absence of a value in the *HeaderFor* property determines whether the column headings or data are shown in any particular grid.

The data grid to which we are attaching must be told of the header grid's interest, so that whenever the former changes, it also updates the latter, to keep both synchronized. Attaching to other components raises the problem of what happens when those components are deleted. Delphi comes to the rescue with the *Notification* method, which is called whenever a component is added to or removed from the form. Having identified the component as one attached to this grid, we can easily remove all references to it, and avoid generating exceptions later on. The same applies to the *DataSource* property of the grid (see [Figure 3](#)).

Changes as They Occur

The grid now knows how to draw and print itself when requested, but so far it doesn't know when properties of the

```
{ Check for loss of attached components }
procedure TQRDBGrid.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if Operation = opRemove then
  begin
    if lstHeaders.IndexOf(AComponent) > -1 then
    begin
      lstHeaders.Remove(AComponent);
      lstHeaders.Pack;
    end;
    if (HeaderFor <> nil) and
      (HeaderFor = AComponent) then
      HeaderFor := nil;
    if (DataSource <> nil) and
      (DataSource = AComponent) then
      DataSource := nil;
  end;
end;
```

Figure 3: Acting on notification of deleted components.

attached data set and fields are being altered. If we change the visibility of one of the fields, the appearance of the grid on the screen doesn't reflect this until it's redrawn — perhaps by minimizing, then restoring the window.

Delphi's existing database grid responds immediately to changes in the underlying fields. But how?

If we search the code, we find a link between the database grid and the data source. Because it's aware of both objects, it can be used to pass changes of one through to the other. These links are derived from *TDataLink*, and contain methods called when the data source changes, so that the attached components can be informed, and can update themselves appropriately.

We automatically create and attach a data link when creating the grid. Thus, the grid knows "who" to notify when things change. We then set the *DataSource* property of this link to reflect that of the grid. In fact, we can use this property to hold the value for the grid, and change the reading and writing of the grid *DataSource* property to automatically access that of the link. This avoids doubling up on the property.

Because our grid is reading only from the data source, the interaction is simple: Whenever something in the data source changes, all we have to do is to repaint the grid to pick up the differences. All the notification methods of the data link that we override are similar to the following:

```

procedure TQRGridDataLink.DataSetChanged;
begin
    FGrid.Invalidate;
end;
    
```

Now our grid is always synchronized with the latest alterations to the data set.

What Does It Look Like?

The functionality of the new report previewer and the database grid component are both shown in the demonstration program accompanying this article. The program consists of three forms: the report itself, the report previewer, and the manipulation of the underlying data set.

The report is based on the Customer.db table that comes with Delphi, and consists of two QRDBGrid components: one providing the column headings in the page header band; the other showing the field contents in the detail band. Changes to the fields defined for the data set can be made through the interface form. This shows, on the left, all the fields from the table, with selected properties for each on the right. Select a field, and change its properties as desired. Making a field invisible removes it from the grid when the report is generated.

Some properties of the grid can be set at the bottom of the form: its alignment within the band, whether cell borders are displayed, and the presence and background color of the heading grid. (The *Columns* button invokes the column edi-

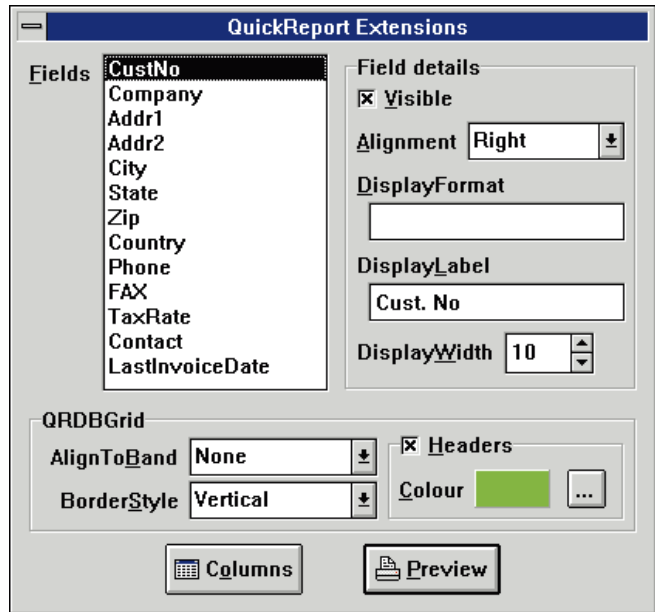


Figure 4: This interface form allows manipulation of the underlying data set fields and TQRDBGrid.

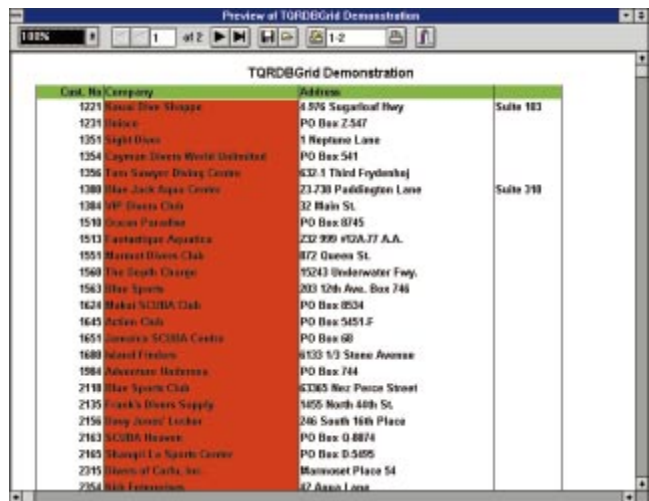


Figure 5: The new report previewer in action.

tor, which is the subject of next month's article.) When all is ready, press the *Preview* button to show the report in the new previewer. The demonstration form appears in Figure 4, while the previewer is shown in Figure 5.

The detail grid has an attached *OnPrint* event that causes the *Company* field to be colored red:

```

{ Colour the Company field red }
procedure TrpQRDBGridDemo.qrdbgDetailPrint(Sender: TObject;
    FieldName: string; var Value: string; var Colour: TColor;
    var Alignment: TAlignment);
begin
    if FieldName = 'Company' then
        Colour := clRed;
end;
    
```

Conclusion

Extending QuickReport in these ways adds useful functionality. The new report previewer allows us to skip to any page at the press of a key, and to print only those pages that we really want. It also provides more zoom options. The database grid

component facilitates the task of generating a report by automatically displaying a collection of fields from the database. Its appearance is controlled by altering its properties and manipulating the field definitions.

Next month we'll extend the QuickReport database grid even further by providing greater control over the displayed columns, allowing each to have its own alignment, color, font, and exact pixel width. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM97AUG\DI9708KW.

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

Begin Listing One — Navigating Pages

```
{ Set label for current page and dis/enable page movement
  buttons/menu items }
procedure TfmQuickPreview.CheckPages;
begin
  edPageNumber.Text := IntToStr(qrpPreview.PageNumber);
  lblPages.Caption := 'of ' +
    IntToStr(qrPrinter.PageCount);
  btnFirstPage.Enabled := (qrPreview.PageNumber > 1);
  btnPrevPage.Enabled := (qrPreview.PageNumber > 1);
  btnNextPage.Enabled := (qrPreview.PageNumber <
    qrPrinter.PageCount);
  btnLastPage.Enabled := (qrPreview.PageNumber <
    qrPrinter.PageCount);
  miFirstPage.Enabled := (qrPreview.PageNumber > 1);
  miPrevPage.Enabled := (qrPreview.PageNumber > 1);
  miNextPage.Enabled := (qrPreview.PageNumber <
    qrPrinter.PageCount);
  miLastPage.Enabled := (qrPreview.PageNumber <
    qrPrinter.PageCount);
end;

{ Change the page number being viewed }
procedure TfmQuickPreview.edPageNumberExit(Sender: TObject);
var
  iPage: Integer;
begin
  try
    iPage := StrToInt(edPageNumber.Text);
    if (iPage < 1) or (iPage > qrPrinter.PageCount) then
      Abort;
    qrpPreview.PageNumber := iPage;
  finally
    CheckPages;
  end;
end;

{ Change pages on ENTER }
procedure TfmQuickPreview.edPageNumberKeyDown(Sender:
  TObject; var Key: Word; Shift: TShiftState);
begin
  if Key = VK_RETURN then
    edPageNumberExit(edPageNumber);
end;

{ Move to first page }
```

```
procedure TfmQuickPreview.btnFirstPageClick(Sender: TObject);
begin
  qrpPreview.PageNumber := 1;
  CheckPages;
end;

{ Move to previous page }
procedure TfmQuickPreview.btnPrevPageClick(Sender: TObject);
begin
  qrpPreview.PageNumber := qrpPreview.PageNumber - 1;
  CheckPages;
end;

{ Move to next page }
procedure TfmQuickPreview.btnNextPageClick(Sender: TObject);
begin
  qrpPreview.PageNumber := qrpPreview.PageNumber + 1;
  CheckPages;
end;

{ Move to last page }
procedure TfmQuickPreview.btnLastPageClick(Sender: TObject);
begin
  qrpPreview.PageNumber := qrPrinter.PageCount;
  CheckPages;
end;
```

End Listing One

Begin Listing Two — Printing Selected Pages

```
{ Print the specified pages, comma-separated list,
  single pages or range nn-nn }
procedure TfmQuickPreview.btnPrintClick(Sender: TObject);
var
  iStartPage, iEndPage, i: Integer;
  sPages: string;
  slPages: TStringList;

  { Extract values in format nn-nn }
  function ExtractRange(sText: string; var iStart,
    iEnd: Integer): Boolean;
  var
    i: Integer;
  begin
    Result := False;
    i := Pos('-', sText);
    try
      if i = 0 then { Single value }
        begin
          iStart := StrToInt(sText);
          iEnd := iStart;
        end
      else { Range of values }
        begin
          iStart := StrToInt(Copy(sText,1,i - 1));
          iEnd := StrToInt(Copy(sText,i + 1,Length(sText)));
        end;
      if (iStart < 1) or (iStart > qrPrinter.PageCount) or
        (iEnd < 1) or (iEnd > qrPrinter.PageCount) then
        raise EPages.Create(
          'Page number(s) outside range of 1 to ' +
          IntToStr(qrPrinter.PageCount));
      if (iEnd < iStart) then
        raise EPages.Create(
          'Ending page is before starting page.');
```

```

        'the form nn-nn.' + #13#10 +
        e.Message, mtWarning, [mbOK], 0);
    Exit;
end;
end;
end;
begin
{ Extract values separated by commas }
slPages := TStringList.Create;
try
  sPages := edPages.Text;
  i := Pos(',', sPages);
  while i > 0 do begin
    slPages.Add(Copy(sPages, 1, i - 1));
    Delete(sPages, 1, i);
    i := Pos(',', sPages);
  end;
  slPages.Add(sPages);

  { Check all values for format }
  for i := 0 to slPages.Count - 1 do
    if not ExtractRange(slPages[i],
      iStartPage, iEndPage) then
      Abort;

  { If all OK, then print each set }
  for i := 0 to slPages.Count - 1 do begin
    ExtractRange(slPages[i], iStartPage, iEndPage);
    with qrPrinter do begin
      FromPage := iStartPage;
      ToPage := iEndPage;
      Print;
    end;
  end;
finally
  slPages.Free;
end;
end;
end;

```

End Listing Two





INFORMANT SPOTLIGHT

Delphi / Object Pascal



By *Peter Roth*

Nice and Nicer

How Nice Are Your Object Pascal Classes?

For many developers new to the Object Pascal language, questions tend to arise after they jump in and get their feet wet. Some of these queries include: What services must classes provide so they are reusable, as in “easily derived from?”; what services must classes provide so they are reusable as fields of another class?; and how can I tell that I have “good” code?

These questions are not new, and experienced programmers have an idea of what a class must have. In C++, for example, certain methods are so important to the proper operation of a program that they are *required*.

If the programmer omits these crucial methods, the C++ compiler generates a default to fulfill the need. Because compilers cannot read the human mind, the generated methods may be less than what is needed.

On the other hand, a class in which the required methods are properly provided is called a “nice” class (coined as such by Martin Carroll and Margaret Ellis in *Designing & Coding Reusable C++* [Addison-Wesley, 1995]).

Object Pascal doesn’t *require* specific methods, but some methods are so important to the proper operation of Object Pascal programs that they *ought* to be part of most classes. Therefore, to be “nice,” an Object Pascal class must contain:

- the *Create* constructor, to make objects (instances of a class);
- the *Destroy* destructor, to destroy objects;
- the *Equals* function, to compare objects;
- the *Copy* procedure, to copy one object into an existing object; and
- the *Clone* constructor, to make duplicates of objects.

The last three of these can be a little tricky to get right, but after some analysis, they are mostly boilerplate.

The nice class is a good start, and is fine for objects that live in memory. For engineering calculations specifically, many objects *persist*; i.e. they are preserved in files. Thus, a class may need to interact with text files and streams to save and restore data from those sources.

A nice class is made “nicer” by providing the following four methods:

- the *Read* constructor, to read text;
- the *Write* procedure, to write text;
- the *Get* constructor, to read from a stream; and
- the *Put* procedure, to write to a stream.

Each of these methods is simple to write. Before getting to the details, however, let’s consider how classes are used and, unfortunately, misused.

How Are Classes Used?

Let’s consider two ways classes are used: as bases from which to derive other classes, and as components of aggregated classes.

When a class is derived from a base class, the derived class is said to be in an *is a* relationship with the base class. That is, a derivative instance may replace a base instance in a calculation because the

derived class is a base class. This relationship is the source of polymorphism.

For example; suppose the job requires class *D* be derived from base class *B*. For *D* to write itself to a file, the inherited part of *B* must also write itself to a file. Therefore, both *D* and *B* will have a *Write* procedure, and *D.Write* will reference *B.Write*. If *B* is a “nicer” class, it can already write itself, so half the job of writing *D.Write* is done and requires no extra effort.

On the other hand, when an object *O* contains another object *D*, then *O* has a *D*. This aggregation of objects is therefore called the *has a* relationship. A Delphi form is an obvious example of an aggregated object. A form has a *TButton*, it has a *TCheckbox*, etc.

If the aggregated object *O* is to write itself to a file, *O*'s fields must also be written to the file. Somewhere in the *O.Write* method will be a call to the contained *D.Write*. If *D* is a “nicer” class, this capability is provided to the *O* developer, and the writing of *O.Write* can be accomplished by simply calling *D*'s method. Again, a small extra effort is required.

How Are Classes Misused?

Consider the statements:

```
var
  a, b: TSomeClass;
begin
  a := TSomeClass.Create;
  b := TSomeClass.Create;
  if a = b then
    writeln('true')
  else
    writeln('False, as expected');
```

The intrepid programmer is trying to compare two objects for equality, but the expression:

```
a = b
```

will always be *False*, regardless of the contents of *a* and *b*. To see why, recall that *a* and *b* are *references*, a kind of pointer. Because a pointer is an address, and because the expression compares different addresses:

```
a = b
```

will always evaluate to *False*. This implies that comparisons must be done with something other than the = operator, and justifies the *Equals* function as a method in the nicer class.

Consider a statement that attempts to copy *b* into *a* using the assignment operator:

```
a := b;
```

This may not be what is intended: *a* is not a copy of *b*, but rather an *alias*. *a* and *b* are still references, so *addresses* are copied by the assignment operator, not *contents*. When references or pointers are copied, the action is known as a *shallow*

copy. When the contents of an object are copied, the action is called a *deep copy*.

To do a deep copy, the programmers *using* a class must access the fields of that class and do the copy in their own code, outside the class. Alternatively, the *designer* of the class can provide a *Copy* procedure and keep this activity where it belongs — in the class.

Finally, although the intent of the statement:

```
a := b
```

is assumed to be “make *a* into a copy of *b*,” it could also be assumed to be a naive attempt to clone *b*; that is, to create *a* as a completely new object. Of course, creating a new object requires a constructor. Therefore, to enable cloning, the class designer must provide a *Clone* constructor as part of the class interface. The programmer can then write:

```
a := TA.Clone(b);
```

Now, let's examine the details of the members of a “nice” class.

The Create Constructor

A constructor performs three tasks when creating an object:

- it allocates storage;
- it initializes the fields of the object; and
- it returns a reference to the created object.

The *Object Pascal Language Guide* [Borland International, 1996] suggests that the constructor may be omitted, and the compiler will nevertheless produce code to instantiate the class, in which case the fields are initialized to 0.

The solution to problems with object-oriented programming languages is one of designing a set of cooperating classes. During the design process, the designer changes the class hierarchy; inheritance patterns are altered; common fields and methods migrate toward the base classes; new classes evolve, older ones are discarded, etc.

In this volatile environment, it's handy to have a constructor to initialize fields. Hence, the “nice” class requires:

- one or more constructors;
- each constructor to invoke its inherited *Create*, or another appropriate inherited constructor, such that all inherited constructors are invoked in turn; and
- the default constructor to be named *Create*, which is declared thus:

```
constructor Create [ ( argument list ) ];
```

and defined thus:

```
constructor TMyClass.Create [ ( argument list ) ] ;
begin
  inherited Create [ ( appropriate arguments ) ] ;
  // User code to instantiate this object's fields.
end;
```

The Destroy Destructor

A destructor should first execute the user code that releases “owned” objects, and finish with a call to the inherited destructor. When this user code completes, the destructor releases storage that was allocated for the object. Note that the word “storage” here refers to *memory*; storage associated with files and other devices must be released with the user code.

The *Object Pascal Language Guide*’s recommendations regarding destructors (albeit enforced by self discipline) are that they produce a nice class destructor:

- There shall be only one destructor.
- It shall be called *Destroy*.
- It shall be declared thus (overriding its parent method, polymorphically):

```
destructor Destroy; override;
```

and defined thus:

```
destructor TMyClass.Destroy;
begin
  // User code to dispose of all of this object's fields,
  // a mirror image of the constructor user code
  inherited Destroy;
end;
```

- When disposing of an object, don’t write:

```
TMyClass.Destroy;
```

instead, write:

```
TMyClass.Free;
```

(In other words, “Don’t call us, we’ll call you.”)

The Equals Function

How will it be used? Assuming class *T1* and the variables:

```
var a, b: T1;
```

then the expression to compare the equality of *a* and *b* will be:

```
a.Equals(b)
```

From this expression, the code follows directly. The declaration is:

```
function Equals(const aT1:T1): Boolean;
```

and the definition is:

```
function T1.Equals(const aT1:T1): Boolean;
begin
  Result := true;
  // for each field in T1
  // if the field is a built-in type
  //   Result := Result and (thefield = aT1.thefield)
  // if the field is another nice class
  //   Result := Result and (theNiceField.Equals(
  //                               aT1.theNiceField))
end;
```

Should *Equals* be virtual? Arguments of all descendants of virtual functions must have the same arguments. That is, the arguments must be the same type as the base class. Given the class *T1*:

```
function Equals(const aT1: T1); Boolean; virtual;
```

then class *T2* derived from *T1* must declare the *Equals* function:

```
function Equals(const aT1: T1); Boolean; override;
```

and define it:

```
function T2.Equals(const aT1: T1); Boolean;
begin
  Result := true;
  // for each field in T2
  // if the field is a built-in type
  //   // This is an unsafe cast.
  //   Result := Result and (thefield =
  //                           T2(aT1).thefield)
  // etc.
end;
```

Not only does this function *look* wrong, the internals require an unsafe type cast. The cast is unsafe because it allows the argument *aT1* to access fields of a descendent which may not exist in the class *T1*, especially if the function is called like this:

```
var
  aT2: T2;
  aT1: T1;
...
if aT2.Equals(aT1) then
  ... // Probably trouble.
```

Therefore, *Equals* shall not be virtual.

What does *Equals* look like in a derived nice class? The definition of *Equals* will have the same form as the *Equals* in *T1*, and will contain an additional statement, as shown in [Figure 1](#). The casts are safe because a derived object may always be cast to its base object. It’s possible to write this statement with a single cast:

```
Result := Result and T1(Self).Equals(aT2);
```

because *T2* is a *T1*, and so is acceptable to the base *Equals* function. However, the form that casts both objects is preferable, because it maintains the symmetry of the statement.

```
function T2.Equals(const aT2:T2): Boolean;
begin
  Result := true;
  // Safe casts
  Result := Result and T1(Self).Equals(T1(aT2));
  // for each field in T2
  // if the field is a built-in type
  //   Result := Result and (thefield = aT2.thefield)
  // if the field is another nice class
  //   Result := Result and (theNiceField.Equals(
  //                               aT2.theNiceField))
end;
```

Figure 1: The definition of *Equals* has the same form as the *Equals* in *T1*, and contains an additional statement.

Should *Equals* have to compare apples and oranges? No. Object Pascal stipulates that different types can never be equal; they are *different*. That's the whole idea behind type safety. The *Equals* function must compare two entities, each of which share the same essence. Therefore, if *a* is a *TApple* (derived from *TFruit*), and *o* is also a derivative of *TFruit*, then the code to compare *a* with *o* must be:

```
if ClassType(o) = ClassType(a) then
  Result := a.Equals(o)
else
  Result := False;
```

which preserves type safety. On the other hand, an expression like:

```
f.Weight = a.Weight
```

compares the size of two pieces of fruit, but in no way implies that the fruits are equal.

Should the argument be passed as a variable, by value, or as a constant? That is, pick one of these statements:

```
function Equals(var aT1: T1): Boolean; // As a variable.
function Equals(aT1: T1): Boolean; // By value.
function Equals(const aT1: T1): Boolean; // As a constant.
```

Again, note that in each case, *aT1* is a reference (a pointer) to an object.

In the first function, the code:

```
var aT1
```

tells the compiler (and the reader) that the reference to *aT1* may be changed by this call. That is, *aT1* will most likely point to a different object after execution of the function. That the simple comparison of two objects could result in getting a different object back is absolutely mind boggling, regardless of what the function is doing. Therefore, reject passing the argument as a variable.

The second function accepts an argument passed by value; it directs the compiler to make a copy of the reference and pass the copy to the function, whose scope becomes that of the function. The function is free to do whatever it wishes with the copy. Again, the human reader wonders how a comparison could require a function to modify even a copy of one of the objects.

In the third function, the phrase:

```
const aT1
```

tells the compiler and the reader that the reference to *aT1* will not and cannot be changed by calling the function. Although this guarantee is similar to passing by value, the notational advantage is significant: The code says what it is supposed to do. Of secondary interest, passing as a **const** may be more efficient than either of the other methods. Therefore, passing as a **const** is appropriate.

```
constructor T1.Clone(const aT1: T1);
begin
  // if this class is derived from T0
  // inherited Clone(aT1)
  // else
  // inherited Create;
  // for each field in T1
  // if the field is a built-in type
  // thefield := aT1.thefield;
  // if the field is another nice class
  // theNiceField := T1.Clone ( aT1.theNiceField))
end;

procedure T1.Copy(const aT1: T1);
begin
  // if this class is derived from T0
  // inherited Copy (aT1)
  // for each field in T1
  // if the field is a built-in type
  // thefield := aT1.thefield;
  // if the field is another nice class
  // theNiceField.Copy ( aT1.theNiceField)
end;
```

Figure 2: The definitions of the *Clone* constructor and *Copy* procedure.

Any of these ways to pass arguments will work, but none prevents the called function from changing the *contents* of the reference. The advantage to choosing **const** is purely notational. On the other hand, the class developer must ensure that the code in the method preserves the “constness” of the argument.

The Clone Constructor and Copy Procedure

Because of the similarity in their intent, these two methods are discussed together. What should they be, functions or constructors?

The idea of cloning implies the creation of a new object, whereas the idea of copying implies placing the values of one object into the values of another, existing object. Hence, *Clone* should be a constructor, and *Copy* should be a function or procedure. Because there is no expected “return signal” from the execution of *Copy*, it should be a procedure.

Using reasoning similar to that of the *Equals* function, the *Clone* constructor and *Copy* procedure declarations for a class *T1* are easily seen to be:

```
constructor Clone(const aT1: T1);
procedure Copy(const aT1: T1);
```

The definitions of the *Clone* constructor and *Copy* procedure are shown in Figure 2. Note that neither method requires type casts.

The Nicer Class

It's time to consider making the nice class even nicer. None of the nice class method names have interfered with the meaning and intent of Object Pascal's reserved words, directives, etc. The nicer class begins to impinge on these areas.

Interfacing with TextFiles. Two methods are proposed: *Read* and *Write*. Although they each interface with a *TextFile*, they

INFORMANT SPOTLIGHT

need not be complementary. That is, *Read* need not be able to interpret a file produced by *Write*.

Read and *Write* have traditionally been associated with file data (for further discussion, see *Pascal User Manual and Report*, 2nd Edition, by Kathleen Jensen and Niklaus Wirth [Springer-Verlag, 1978]). The nicer class maintains this idea, as opposed to the methods used to obtain or set class *properties*, and the myriad other uses to which the words “Read” and “Write” have been applied.

Read. Although *Read* could be a procedure, the interface with a *TextFile* allows the size of an object to be read, as well as the object’s contents. Because these two actions match the actions of a constructor, *Read* is defined as a constructor. The argument may be a *TextFile*, a lexical scanner, etc.

Here’s an example of the simplest *Read* declaration:

```
constructor Read(var f: TextFile);
```

Write is a procedure that writes an object in human-readable form. This versatile method can be extremely useful as an error logger during program development. It can also be used to preserve numerical results, etc.

This is an example of the simplest *Write* declaration:

```
procedure Write(var f: TextFile);
```

The actions these two methods can implement are so diverse that a typical example is not obvious.

Interfacing with *TStreams*. Two methods are proposed: *Get* and *Put*. These methods must be complementary. That is, *Get* must be able to interpret a *TStream* produced by *Put*, and *Put* must produce a *TStream* legible to *Get*. The complementarity is easily maintained because the code for each method is in proximity to the other, and may be easily “eyeballed” for correctness.

Get and Put. *Get* and *Put* were used in the original Pascal language to fetch a single char from input, and write a single char to output, respectively (again, see *Pascal User Manual and Report*, 2nd Edition). They were redefined and used as stream functions in Borland Pascal 7, but have disappeared from Delphi 2. Their use in nicer classes to read and write to streams therefore does not conflict with current usage.

Get. For the same reasons that *Read* is a constructor, so is *Get*. The declaration is:

```
constructor Get(S: TStream);
```

Put. Paralleling *Write*, *Put* is a procedure, and is declared:

```
procedure Put(S: TStream);
```

The state of *S* will change after a call to *Get* or *Put*, so *S* is passed by value (rather than as a **const**) to so indicate. Again,

```
constructor T1.Get(S: TStream);
begin
  // if this class is derived from T0
  // inherited Get(aT1)
  // else
  // inherited Create;
  // for each field in T1
  // if the field is a built-in type
  //   S.Read(theField, sizeof(theField));
  // if the field is another nice class
  //   theNiceField.Get(S);
end;

procedure T1.Put(S: TStream);
begin
  // if this class is derived from T0
  // inherited Put (aT1)
  // for each field in T1
  // if the field is a built-in type
  //   S.Write (theField, sizeof(theField));
  // if the field is another nice class
  //   theNiceField.Put(S);
end;
```

Figure 3: The *Get* and *Put* definitions.

this is a notational decision. The *Get* and *Put* definitions are shown in Figure 3. Note that neither method requires casts.

The Payoff

Nicer classes simplify development because each object is guaranteed to have desirable qualities. As an example, assume the following nicer toy classes:

- *TAnother* contains a string field.
- *TAnotherKid*, a child of *TAnother*, contains a double field.
- *TBase* contains a char field.
- *TDerived* (derived from *TBase*) has a string field and a *TAnotherKid*.

The code in Listing Three shows how easily objects of the classes are created and compared. The classes’ methods make these and other object interactions easy to write. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in `INFORM\97\AUG\DI9708PR`.

Peter N. Roth is president of Engineering Objects International (<http://www.incon-research.com/eoi>) in Fairfax, VA. He teaches and writes software in several languages, but prefers Delphi and C++ . The company’s latest release is ClassBuilder 4 for Delphi 2. Peter can be contacted via e-mail at peteroth@erols.com, or phone at (703) 968-4224.

Begin Listing Three — UTAnoKid.pas

```
{SX+ extended syntax}
unit Utanokid;

interface

uses
  Classes,
  UTAnothe;

type
```

```

TAnotherKid = class( TAnother )
  constructor Create(const aString: string;
    const aDouble: double);
    constructor Clone(const aTAnotherKid: TAnotherKid);
    constructor Get(S: TStream);
    constructor Read(var f: TextFile);

  destructor Destroy; override;

  procedure Copy(const aTAnotherKid: TAnotherKid);
  function Equals(
    const aTAnotherKid: TAnotherKid): Boolean;
  procedure Put(S: TStream);
  procedure Write(var f: TextFile);

  function GetD: double;
  function SetD(const aDouble: double): double;

private
  d: double;
end; {TAnotherKid}

implementation

uses
  SysUtils;

constructor TAnotherKid.Create(const aString: string;
  const aDouble: double);
begin
  inherited Create(aString);
  d := aDouble;
end; {Create}

constructor TAnotherKid.Clone(
  const aTAnotherKid: TAnotherKid);
begin
  inherited Clone(aTAnotherKid);
  d := aTAnotherKid.d;
end; {Clone}

constructor TAnotherKid.Get(S: TStream);
begin
  inherited Get(S);
  S.Read(d, sizeof(d));
end; {Get}

constructor TAnotherKid.Read(var f: TextFile);
begin
  // To be added.
end; {Read}

destructor TAnotherKid.Destroy; {override}
begin
  inherited Destroy;
end; {Destroy}

procedure TAnotherKid.Copy(const aTAnotherKid: TAnotherKid);
begin
  if Self = aTAnotherKid then
    Exit; {Don't clobber Self.}
  inherited Copy(aTAnotherKid);
  d := aTAnotherKid.d;
end; {Copy}

function TAnotherKid.Equals(
  const aTAnotherKid: TAnotherKid): Boolean;
begin
  Result := True;
  if Self = aTAnotherKid then
    Exit; {Saves time?}
  Result := Result and
    TAnotherKid.Equals(TAnotherKid(aTAnotherKid));
  Result := Result and (d = aTAnotherKid.d);

```

```

end; {Equals}

procedure TAnotherKid.Put(S: TStream);
begin
  inherited Put(S);
  S.Write(d, sizeof(d));
end; {Put}

procedure TAnotherKid.Write(var f: TextFile);
begin
  inherited Write;
end; {Write}

function TAnotherKid.GetD: double;
begin
  Result := d;
end; {GetD}

function TAnotherKid.SetD(const aDouble: double): double;
begin
  Result := d;
  d := aDouble;
end; {SetD}

end.

```

End Listing Three





ON THE NET

Delphi 2 / POP3 / SMTP

By *Gregory Lee*



Hop on POP

Internet Delphi: Part II

Last month, we saw how easy it is to add the ability to send e-mail over the Internet with the Simple Mail Transfer Protocol (SMTP). This time, we'll focus on the protocol that lets you retrieve e-mail from an e-mail account: the Post Office Protocol (POP). The current version of this protocol is version 3, so it's generally referred to as POP3.

If you didn't read the last installment on SMTP, you have some catching up to do. One of the topics that we won't review here in detail is a programming method known as a *state machine*. Suffice it to say the technique isn't very complicated, but it's an ideal way to implement support for many Internet protocols. See the first installment for details.

Why?

If there's already a protocol for sending e-mail, why in the world do we need another protocol for retrieving it? Shouldn't we simply assume the mail server's role in an SMTP conversation? That sounds reasonable, but remember that the SMTP protocol essentially requires that the server be available at port 25

at all times; if your personal SMTP server program wasn't available at the exact moment when someone wanted to send you a message, you just wouldn't get it.

Instead, when someone sends you e-mail, it goes from the SMTP server where they dropped it off, into your service provider's post office. In all probability, it was passed around by a handful of systems along the way before it got to your local post office, but that's not really important right now. What's important is the result: When you log into your personal Internet account, you can check your service provider's post office to see if any messages are waiting. That makes a lot more sense than requiring everyone, everywhere, to establish a continuous Internet connection and an SMTP mail server.

A Program Named DelphiEmail

If you want a complete description of the POP3 protocol, you should read RFC 1725, "Post Office Protocol - Version 3" by John G. Meyers and Marshall T. Rose. RFC stands for "Request For Comment" and virtually every Internet standard is documented somewhere in an RFC file (see Figure 1). Here are some Web sites containing indexes you can search for this, and other, RFC documents:

- <http://www.neda.com>
- <http://ds.internic.net/ds/rfc-index.html>
- <http://www.rasip.etf.hr/rfc/>

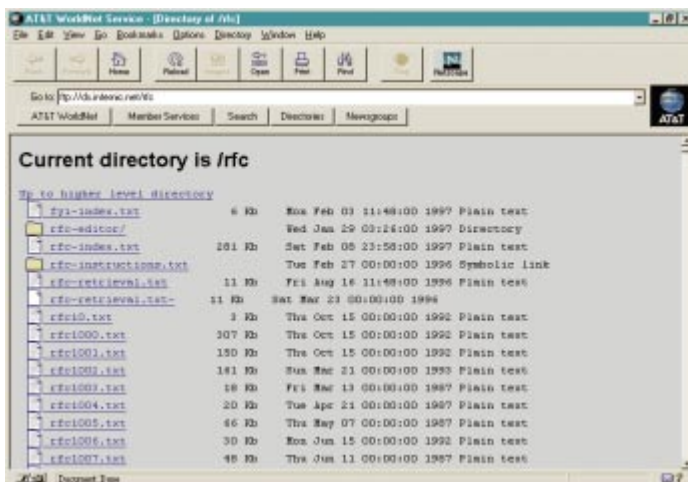


Figure 1: InterNIC's ftp site contains an index to all RFC documents.

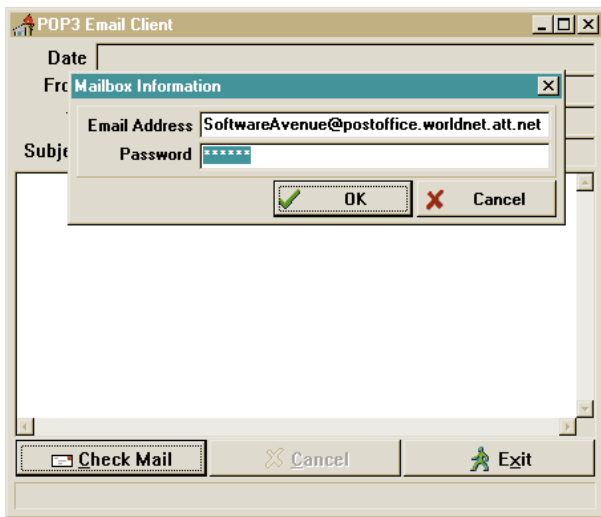


Figure 2: A typical DelphiEmail session.

Getting Started

In a typical DelphiEmail session, the user will start the program and click the **Check Mail** button. This is our cue to contact the mail server and initiate a POP3 session. A typical session is shown in [Figure 2](#).

Before we can connect to the POP3 server, we must know where to find it. Part of the server's location can be taken directly from your e-mail address. In a typical e-mail address, everything on the right side of the @ sign is referred to as the *host name*. For example, in 76455.3236@compuserve.com, the host name is compuserve.com.

The host name gets us most of the way there, but we still need to know where on the host system we can find the POP3 mail server. RFC 1725 tells us that the mail server will be listening for calls at port number 110. You can think of Internet port numbers as the extension numbers in a telephone system. Most established protocols have a “well-known” port number where clients and servers can count on hooking up. The “well-known port” for POP3 just happens to be port 110.

With the host name and port number now in place, we're ready to establish the connection. If you want an explanation of how we're calling routines in WINSOCK.DLL, how the host name gets translated into an IP address, and all that other fun stuff, you can pore through the Windows Socket Specification, or get a copy of the Delphi Finger article that got us started (see the [August 1996](#) issue of *Delphi Informant*). For now, the details of the journey are less important than the destination.

Another State Machine

According to RFC 1725, when we connect to the POP3 server, the server will send us a greeting. If we've requested the data before the greeting arrives (with the `recv` function), the greeting will trigger an `FD_READ` message and a call to our `AsyncEvent` handler.

Like the SMTP greeting, the POP3 greeting message could say almost anything, but according to the POP3

protocol, we can count on one constant element: The first three characters in the greeting message will be +OK. Unlike the SMTP server responses, the POP3 server doesn't use result codes to indicate the success or failure of various operations. In fact, the initial response to every command we can send will be either +OK or -ERR depending on the result — all the more reason to use a state machine to handle the conversation.

A simple state machine can be implemented with one global variable (the current state), and a `case` statement that executes some code or branches to whatever procedure is appropriate given the state. At each stage in the protocol, our program will proceed by sending something new to the server, then waiting for an appropriate response. When the response or acknowledgment message is received, the state machine is bumped to the next level, and processing continues.

In this version of the DelphiEmail program, the state machine is a little more complex than the SMTP version. The reason for this is simple: You can do a lot more with POP3 than you can with SMTP. A POP3 conversation doesn't necessarily proceed along a fixed path the way an SMTP session does. You can use POP3 commands to get statistics about your e-mail, retrieve waiting e-mail, and delete your e-mail. Any and all of these steps are optional.

One result of this added flexibility is an increase in the number of potential values for the global variable `State`. The new definition for the `TState` variable type is:

```
TState = (Inactive, Connected, UsernameSent, PasswordSent,
          WaitingForStats, WaitingToRetrieveMail,
          RetrievingMail, DeletingMessage, QuitSent,
          UnChanged);
```

With the new type established in this way, the layout of the `case` statement is straightforward. Finishing the application is now simply a matter of initiating these states in a particular sequence, and writing a short procedure to handle each state.

Will Our Mystery Guest Please Sign In?

Initially, our `State` is `Inactive`. Once we've connected to the POP3 server, our `State` changes to `Connected`. The `Pop3Engine` procedure handles this state by scanning incoming messages for the +OK we're expecting.

If we get the response we're looking for, we can proceed to the next step, which is to send the server the `USER` command along with the appropriate username. In a typical Internet e-mail address, the username is everything to the left of the @ sign. For example, in 76455.3236@compuserve.com, the username is 76455.3236.

The next time we receive a message from the server, it will filter through the state machine and control will be handed off to the section of code associated with the `UsernameSent` state. This code looks for another +OK. Again, the POP3 protocol doesn't dictate exactly what the

server will send us, but it does require that the first few characters be +OK if we're on track, or -ERR if something's wrong. Once we receive the +OK sequence, we can proceed to the next step.

One of the basic requirements of any e-mail system is that you, and you alone, have access to your e-mail. Obviously, the most straightforward method of protecting an e-mail account is to use a password, and that's what POP3 requires. The format of the line we send to the server is the word PASS, followed by the password that goes with the username we previously supplied with the USER command. After the password line is sent, we set the global *State* variable to *PasswordSent*, so the state machine will know how to proceed when it receives a response.

The code associated with the *PasswordSent* state looks for the appropriate response which, in this case, is another +OK. If the server responds with a -ERR instead, we have something mixed up. The password sent doesn't match the username, and we won't be able to access the account.

Let's assume we have the correct username and password, and that the server has responded with another +OK. At this point, there is no hard-and-fast rule about what we should do. A reasonable course of action, however, would probably be to get the status of any waiting messages. The POP3 command to accomplish this is STAT.

Unlike many of the other commands, the server's response to STAT must follow a consistent format. After the usual +OK, the response will contain two numbers: the number of waiting messages, and their total size in bytes. For example, if five messages are waiting, and their combined size is 496 bytes, the server's response would be:

```
+OK 5 496
```

Ask and You Shall Receive

Now that we're connected to the mail server, we're logged into an e-mail account, and we know how many messages are waiting in the account, we can start retrieving the messages. The POP3 command to get this process started is the word RETR, followed by the number of the message to retrieve. As long as you indicate a valid message number, the server will respond with another +OK, followed by the text of the message requested.

Just like the e-mail we sent with the previous SMTP program, the message will be sent to us in a continuous stream. The only filtering we have to do is to check for a period in the first position of each line. If the first character of a line is a period, we check the second character to determine if this line of the message starts with a period, or if this is a signal that the text of the message is complete.

If the first two characters of a line are periods, this indicates that the message actually contains a line of text that begins with a period. The addition of the leading period is a process called "quoting," and it's up to the receiving program to check for and

Your Mileage May Vary

In last month's article, and again in this installment, we've used the host name parts of e-mail addresses to decide where to contact the mail server. Wouldn't it be nice if everything in life were that simple? Well, everything is not, even e-mail distribution.

Many, if not most, large systems have their SMTP and POP3 servers stationed at a site other than that given in their individual users' e-mail addresses. For example, if you try to contact the SMTP server for worldnet.att.net, you will probably find there isn't even an IP address listed for that host name (Valid name, no record of requested type). In fact, both the SMTP and POP3 servers for users on that system are located at postoffice.worldnet.att.net. So how in the heck does the e-mail you send to a user on that system get where it's going?

Typically, you would drop the e-mail off at the SMTP server on your ISP's system. Again, that server may or may not be located at the host name given in your e-mail address; but even if it's not, at least you only have to keep track of one additional host name instead of every mail-server host name corresponding to every e-mail address you want to send mail to. From there, the mail server on your ISP's system will attempt to find the appropriate mail exchange site for the destination address. It finds this by contacting one or more name servers until it locates the one that knows about worldnet.att.net. Once it's found the appropriate name server, it can inquire about mail exchange site(s) corresponding to users at worldnet.att.net. It will eventually determine that it can hand off the e-mail to an SMTP server at postoffice.worldnet.att.net (which does have a listed IP address). From there, it's up to the mail server on the other end to make sure the message gets to its final destination.

If you find this at all confusing, you're not alone. In fact, entire books have been written on how to set up and administer Internet e-mail systems. We're not talking about a breezy couple of hundred pages either. The book *sendmail* by Bob Costales and Eric Allman [O'Reilly & Associates, Inc., 1996] runs over 1,000 pages. At least one book has been written about the name-server system alone. *DNS and BIND* by Paul Albitz and Cricket Liu [O'Reilly & Associates, Inc., 1992] covers this area in some detail. If you want to know more about how Internet e-mail is distributed and delivered, these are both required reading.

— Gregory Lee

strip out the additional characters. A line containing a single period indicates the end of the message.

Each time WinSock generates an FD_READ message, we'll have another portion of the e-mail waiting in the buffer. Of course, that doesn't mean we'll necessarily get a separate FD_READ read for each line in the e-mail. In fact, it's likely we'll receive several lines at once. The POP3 server and WinSock are going to cooperate to put as much information into the buffer as they can. Because of this, you will often see the leading portion of one line at the end of the buffer, and the remainder at the beginning of the next load. The solution is to set up an intermediary line buffer, so that any partial lines we receive can be deposited there

until the remaining portion of the line comes through. In the *ReceiveMail* function, the global variable *szWork* plays that role.

You've Got It, Now Get Rid of It


After you've successfully retrieved a message from the post office, it's generally a good idea to save the message locally, and delete the message from your e-mail account. That way, you won't waste time retrieving the same messages again the next time you log into the account. Your service provider will probably also appreciate it, because the post office won't have to continue storing your old messages.

The POP3 command to delete an e-mail message is DELE, followed by the message number to delete. Again, it's up to you to make sure you've provided the correct message number. If you wanted, you could delete all the messages in your account without retrieving them. The mail server won't second-guess your commands.

You probably won't be using this sample project to retrieve your e-mail on a daily basis. Even if you did, there's no way (yet) to save your messages to disk, recall them, and so forth. For this reason, I've purposely omitted the DELE command from the sample program. So don't worry, the urgent message your boss just sent you won't get lost in the shuffle (and I won't be flamed by a bunch of angry, recently unemployed Delphi programmers). Besides, it's nice to be able to run the sample program over and over without having to send yourself another e-mail before each session.

Conclusion

Once you've finished counting, retrieving, and deleting messages from your e-mail account, you can say good-bye to the mail server with the QUIT command. The server will respond with +OK, and close its end of the connection.

At this point, turning the DelphiEmail program into a functional e-mail client is simply a matter of combining the SMTP capabilities illustrated in the previous article with the POP3 code just discussed. Of course, a full-featured e-mail program would also allow you to do things like save, recall, reply to, and forward e-mails you've received. It might also allow you to attach binary files to your e-mail, and detach them again upon receipt. Next time, we'll add some of these features, and look at a data-encoding method that makes binary file attachments possible: base64 encoding. 

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM97\AUG\DI9708GL.

Gregory Lee is a programmer with over 15 years of experience writing applications and development tools. He is currently the president of Software Avenue, Inc., which has just released a C++ Builder Edition of their Delphi development tool, Internet Developer's Kit. Greg can be reached by e-mail at 76455.3236@compuserve.com.





By *Dan Ehrmann*

The Paradox Files: Part V

Passwords, Encryption, and Table Language

We have covered a lot of ground in our exploration of the Paradox file format. The first two articles in this series (beginning in the [April 1997](#) issue of *Delphi Informant*) explored the internals of Paradox .DB files, dissecting the structure of a table, how the BDE manages records and blocks, field types, and calculating record size; the third article examined primary and secondary indices; and the fourth article addressed validity checks and referential integrity. In this, the fifth article, we'll discuss password protection and the Paradox encryption mechanisms. We'll also introduce Paradox Table Language options.



Figure 1: Defining a master password for a table.

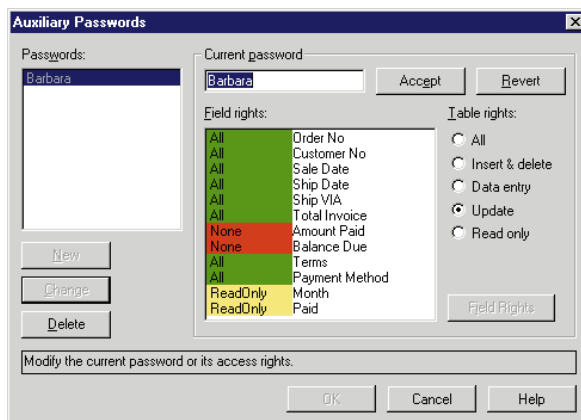


Figure 2: Defining auxiliary passwords for a table.

Encrypted Tables

The Paradox file format supports encrypted tables that can only be accessed by providing a password. Paradox tables support multiple levels of passwords, with different rights to perform operations on a table.

You encrypt a table by defining a *master password* for it. In the Database Desktop, this is accessed by selecting **Table | Restructure Table** (with a table open on the desktop). Under the Table Properties drop-down list select **Password Security**. Click on the **Define** button to access the dialog box shown in [Figure 1](#).

Passwords can be up to 128 characters, but only the first 31 characters are significant. Also, passwords are case-sensitive. The master password defines the *owner* of a table; it has all rights, including the ability to change the master and auxiliary passwords. However, it is also possible to define an auxiliary password that has the right to modify the table's structure.

Auxiliary Passwords

After you enter and verify the master password, the **Auxiliary Passwords** button becomes available. Click this button to see the dialog box shown in [Figure 2](#), which allows you to define auxiliary passwords.

Right	Description
Read only	Users can view the table, but cannot modify its data or structure in any way.
Update	Users can view the table and modify non-key fields only. They cannot insert or delete rows, nor modify key fields.
Data entry	Users can view the table, modify non-key fields and insert new rows. They cannot delete rows, nor modify key fields for existing rows.
Insert & delete	Users can freely insert, delete, or modify records. They cannot change the structure of the table.
All	Users can do almost anything to the table, including changing its structure. They cannot modify the master or auxiliary passwords.

Figure 3: Table access rights for auxiliary passwords.

Right	Description
None	The user cannot view or change data in the field. Its data (but not the existence of the field itself) is completely hidden.
ReadOnly	Users can view the contents of the field, but cannot modify them in any way.
All	Users have all rights to the field, subject to the table rights.

Figure 4: Field access rights for auxiliary passwords.

Click the **New** button to define a new auxiliary password. When you enter the password, the Table rights and Field rights panels become available, allowing you to limit the access rights for each auxiliary password.

The table in [Figure 3](#) describes the effect of limiting table rights for an auxiliary password. Rights are listed in order, from least to most expansive (i.e. each table right includes the capabilities of the previous rights). The table in [Figure 4](#) describes the effect of limited field rights for an auxiliary password. For each field in the table, you can double click on the field or click on the **Field Rights** button to cycle through the three available field rights.

The BDE will not allow you to define rights that are incompatible with each other. For example, if you define Insert & delete table rights, then attempt to limit field rights, the dialog box will change table rights to Update. If you then try to change table rights back to Insert & delete, your modified field rights are changed back to All. This is necessary because you must have all rights to all fields to delete a record. Note that limited field rights are compatible with Data entry table rights, because you don't need to have rights to all fields to insert a record.

In a similar vein, if you add a new field to a table after passwords have been defined, you may need to modify the rights assigned to auxiliary passwords to provide access to this field. This is because the BDE assigns None rights to the field for all passwords that have Read only, Update, or Data entry rights at the table level.

However, you must also be careful about interactions with other table properties. As you saw in last month's article, the Paradox file format allows you to define a field as being Required. You cannot post a record to the table unless a value is provided. However, if users open the table with a password

that grants Data entry table rights, but ReadOnly field rights to the required field, they won't be able to post a new record because they cannot provide a value for this field.

Where Are Passwords Stored?

The master password is stored in the table's header in an encrypted format. Auxiliary passwords are also in the header, encrypted using the master password. This allows the table to be decrypted with no other supporting files or information. If a user or program presents an auxiliary password, the BDE reads the encrypted master password and uses it to decrypt each auxiliary password to see if there is a match.

The BDE Session Password Buffer

When you open a Paradox table using the BDE, a default *session* called (naturally enough) "Session" is created. You can access this session using *TSession* properties, methods, and events (described in detail later in this article). A session object manages a virtual connection to one or more databases; each session is considered to be a different *user* of the database. You can manually create additional *TSession* objects to create additional virtual users.

With respect to passwords, the BDE maintains a password buffer at the session level. This buffer stores all passwords used by Paradox tables within a session, and is big enough to store a maximum of 25 passwords. If the user manually enters a password, it's added to the buffer. Likewise, if your application adds a password in code, it's added to the same buffer. Note that if a password is added to the buffer twice, it's listed twice. If only one instance is removed, the other remains, and access to the encrypted table is still available.

If your application needs to access a Paradox table, and the BDE determines the table is encrypted, it will walk through the list of passwords in the buffer, testing each one against the master and auxiliary passwords stored with the table. For every password in the buffer that matches a valid password for the table, the *union* of rights will be granted.

For example, assume that password "DAN" grants Update table rights and ReadOnly rights to fields 2 and 3 of a table, and password "BARB" grants Data entry table rights and ReadOnly rights to fields 4 and 5 of the same table. If both passwords are in the session buffer and you open this table, users will be granted Data entry table rights overall. They will also be granted full edit rights to all fields, because the ReadOnly rights granted by each password are canceled by the full rights granted by the other password to the fields in question.

Use Caution when Assigning Passwords

This union of rights also creates problems if you use one password to assign different levels of rights to different tables. For example, assume that password "XYZZY" gives full access to one table and password "PLUGH" gives read-only access to the same table. Assume also that password "XYZZY" gives read-only access to a second table.

Along comes user Bob, a low-level clerk who is assigned read-only access to the data in these tables. You have previously told Bob to use “PLUGH” for the first table and “XYZZY” for the second table. However, if Bob accesses both tables, both passwords are in the buffer for his session. If he reopens the form showing the first table, he will have full access, because “XYZZY” provides a higher level of access than “PLUGH.”

There are two lessons to be learned from this scenario:

- 1) When you define passwords for the tables in your application, don't use the same password to provide different levels of access to your tables. Be consistent to ensure that multiple passwords in the session buffer do not grant unintended rights.
- 2) Instead of allowing users to enter table passwords directly, most developers require a user name and password to log in to the whole application. Within a user configuration table, the application reads a level of access for that user, and this level is maintained as a system variable. When access must be provided to an encrypted table, the application then presents a password that is appropriate to the user level. This approach virtualizes Paradox passwords to the application and user level.

Delphi TSession Methods and Events

When your program tries to open an encrypted table, the BDE first checks the session's password buffer for a valid password, as previously described. If a valid password is not found, the BDE reports insufficient password rights, and your program then calls the *OnPassword* event for that session. The default action for this event is to display the Enter password dialog box shown in [Figure 5](#).

You can bypass this dialog box by adding code to this event handler that provides a password to the session. This is done with the *AddPassword* procedure. For example, you might maintain the user's login name as an application variable, and present a different password based on the user (and read from a special configuration table).

You can force the *OnPassword* event to be fired at any time by calling the *GetPassword* function. If the default password dialog box is displayed, this function returns *True* if the user clicked **OK** and *False* if the user clicked **Cancel**. If you add your own code to this event handler, use the *Continue* parameter in the event procedure call to populate the *GetPassword* return value.

To remove one or more passwords for a session, use the *RemovePassword* or *RemoveAllPasswords* procedures, respectively.

Delphi provides no tools to enumerate the passwords for a table, nor the passwords already in the buffer. This functionality is supported by the BDE, but is not displayed in *TSession*. (Note that Paradox itself provides a method to enumerate the passwords for a table, as an aid to documentation.)

Side Effects of Encrypting Tables

BDE operations against encrypted tables run approximately 10 to 15 percent slower than against non-encrypted tables.

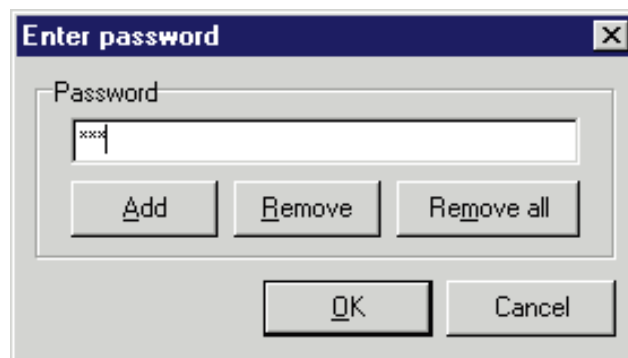


Figure 5: The default password dialog box triggered by the *OnPassword* event.

This is because the BDE must continually pass data read from the table through the decryption algorithm, and insert or updated data through the encryption algorithm.

Another side effect is that encrypted tables do not compress as well when they are zipped. Recall from the first article in this series that Paradox tables use fixed-length block sizes, with fields always occupying a pre-determined amount of space in the table. This invariably results in a large amount of blank space in the table. When this is coupled with the repetitive data often found in normalized tables, it is not unusual to see zip compression ratios of 85 to 90 percent for a typical Paradox table.

When the table is encrypted, however, a mad jumble of ASCII characters, with little or no repetition and no blank space, replaces the data. Compression ratios for encrypted tables usually run no higher than 10 to 15 percent.

How Secure Is the Password?

Paradox's encryption algorithms are certainly not DES — or anything as sophisticated. They use static encryption and decryption rotors that perform multiple (but pre-determined) transformations on each byte of the password. While the algorithm is not documented, it has been reverse engineered by at least two companies: one that markets a replacement Paradox engine, and another that markets a decryption program for people who lose or forget a table password.

The bottom line is that an encrypted table will stop a casual user or experienced PC user who just wants to look at your data. It will not stop a determined hacker who has access to the decryption program, or a lot of time to reverse engineer the decryption algorithm. For most types of corporate data, Paradox encrypted tables provide more than sufficient protection.

Table Language

Paradox tables use 8-bit character sets to store alphanumeric data, giving up to 255 possible characters that can be stored. When you save an alphanumeric character in an Alpha or Memo field, the Paradox file format uses a single byte for that character. Depending on the BDE's *language drivers*, this character can be displayed in different ways.

The BDE supports more than 100 language drivers, many of which are suitable for Paradox tables. These drivers specify a character set that matches a Windows character set (e.g. ANSI 1252) or a DOS code page (e.g. 437 or 850.) To modify the Table Language driver, restructure the table in the Database Desktop. Select **Table | Restructure Table**. Then under the Table Properties drop-down list, select **Table Language** and click on the **Modify** button.

Language drivers control which characters are treated as part of the alphabet, because non-English languages include many accented characters in addition to the standard 26 characters in the English alphabet. A language driver also includes rules for how accented lower-case characters are converted to upper case, and for how an expanded alphabet should be sorted. The driver may also include rules for character substitutions, or the byte value that is used when characters from one language set are added to a table using a different language set.

Note that your Windows regional settings have no effect on the characters that are displayed when a language driver is selected. You do not need to match the language driver for a table with the regional settings defined in Windows. This is because Windows has the ability to use a different Windows character set or DOS code page for each window open on the desktop — even child windows in an MDI application. When you specify a language driver for a table, then display that table within an application, the BDE tells Windows to use the character set or code page that is appropriate to the language driver.

You can easily test this for yourself in the Database Desktop:

- 1) Create a simple table using the default “Paradox ASCII” driver, with an A1 field.
- 2) Populate this table with 10 records representing the ASCII characters from 161 to 170.
- 3) Make a copy of the table. Restructure the copy and change its language driver, for example, to “Pdx ANSI Spanish.”
- 4) View the two tables side by side, as shown in **Figure 6**. Although your regional settings remain the same, you will see different characters in each table for each of the 10 records displayed!

The BDE maintains a current language driver at the session level, using the *TSession.Locale* property. If you open two tables with different language drivers, the BDE will set its current language driver using the last table opened. This can create inconsistent

Pdx ASCII	ID	Text
1	161	í
2	162	ó
3	163	ú
4	164	ñ
5	165	Ñ
6	166	ª
7	167	º
8	168	¿
9	169	□
10	170	¬

Pdx ANSI Spanish	ID	Text
1	161	í
2	162	ø
3	163	£
4	164	æ
5	165	¥
6	166	!
7	167	§
8	168	ˆ
9	169	©
10	170	ª

Figure 6: Viewing two tables with different language drivers.

behavior if you attempt to copy data between these tables; you may not see the extended characters you expect to see.

Note that the BDE does not automatically transliterate from the ANSI character set to the one used by the table. For *TStringField* and *TMemoField* objects, you can set the *Transliterate* property to *True* to force the BDE to perform character translations for you. Setting this property to *True* causes the *AnsiToNative* function to be called whenever a value is written to the field. This function uses *TSession.Locale* to determine the character set or code page to be used for the translation.

You may see a minor performance slowdown if you use different language drivers. If the driver doesn't support binary sorting, where the character sequence is the same as ASCII, an extra level of conversion is needed whenever a sort operation is performed, because the data is not being sorted in a straight line from 0 to 255.

Last Call

Next month, the final article in this series will examine multi-user issues. It will show you how the Paradox file format manages and limits concurrent access to resources, including tables and records, to ensure data integrity. It will also demonstrate the functions of the various network and locking control files. **Δ**

Dan Ehrmann is the founder and President of Kallista, Inc. a database and Internet consulting firm based in Chicago. He is the author of two books on Paradox, and is a member of Team Borland and Corel's CTech. Dan was the Chairman of the Advisory Board for Borland's first Paradox conference, which evolved into the current BDC. He has worked with the Paradox file format for more than 10 years. Dan can be reached via e-mail at dan@kallista.com.





By *Bill Todd*

Battening the Hatches

Inside InterBase: Part III

This third of a five-part series on InterBase examines the three levels of security for InterBase databases: physical security of the database server, operating system security, and that of InterBase itself.

Physical Security

Many organizations overlook physical security. However, if data confidentiality is really important, physical security must be given careful consideration. Remember that anyone who has ever installed a hard disk can remove one in about ten minutes, then access your data at leisure. The only way to prevent this is to make sure that the machine hosting the database is physically secure. The same goes for any copies, including backup tapes.

OS Security

The second level of security concerns the operating system on the database server. InterBase was designed to run on a secure OS; therefore, InterBase database files are not encrypted. Anyone with access to the file can view and copy it with a disk editor. This means you can't have a secure InterBase database on an OS that lacks security, such as Windows 95 or Windows 3.x. Fortunately, securing the database at the OS level is easy, because only the InterBase process directly reads from or writes to the database file, and only InterBase needs access to the .GDB file. Database users need not log on to the machine on which InterBase is running.

Most database authorities recommend that the database run on a dedicated machine, particularly if NetWare or Windows NT is the operating system, to provide the best possible performance and stability. This is also an excellent architecture from a security standpoint, because you can restrict login access to the database administrator only. To access the database, users need only establish communication with InterBase across the network. This arrangement also has economic advantages. For example, 50 users can simultaneously access an InterBase database running on a two-user license of NetWare or Windows NT. Of course you still need a 50-user license for InterBase, but you don't need to license the same users for the OS on the machine that hosts InterBase.

InterBase Security

After you've implemented physical and OS security, it's time to examine the InterBase security features that let you control access to the data and objects in your database. When you install InterBase, a server-security database, ISC4.GDB, is installed in the INTRBASE directory, as is a backup copy, ISC4.GBK. Initially this database contains one user, SYSDBA, whose password is `masterkey`.

Users must be added to the server before they can use Server Manager to connect to the security database. After starting Server Manager, select **File | Server Login** and connect to your server. Next, select **Tasks | User Security** to display the InterBase Security dialog box shown in [Figure 1](#).

Before you do anything else, click the **Modify User** button to display the User Configuration dialog box shown in

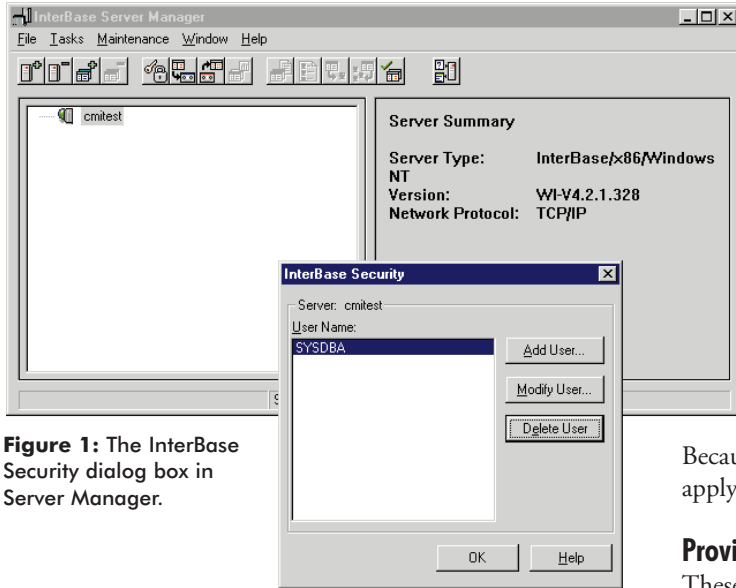


Figure 1: The InterBase Security dialog box in Server Manager.

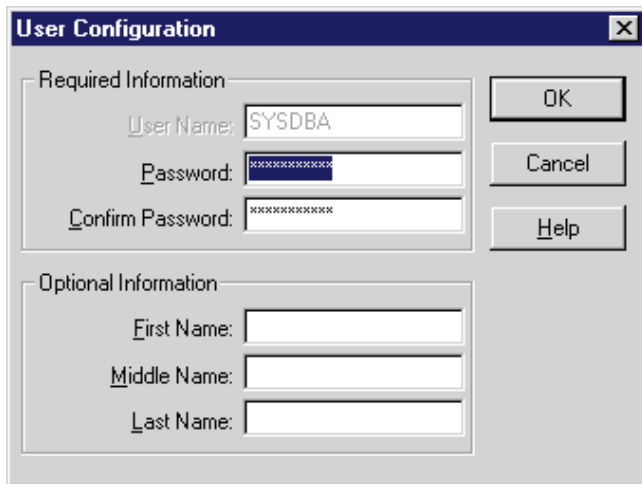


Figure 2: The User Configuration dialog box.

Privilege	Description
SELECT	User can read data from the table using the SQL SELECT statement.
INSERT	User can insert new rows into the table.
UPDATE	User can change data in a row or a table.
DELETE	User can delete rows from a table.
REFERENCES	User can insert record into or modify values in a referential-integrity child table for which this table is the master. Users don't need the REFERENCES privilege to the master table if they have the SELECT privilege.
ALL	All of the privileges listed above.
EXECUTE	This privilege applies only to stored procedures, and grants the user the right to execute them.

Figure 3: Table-level access privileges.

[Figure 2](#), then assign a new password to SYSDBA. Until you do this, everyone who knows the default password has complete access to your data.

Notice that the user name is grayed, and can't be changed. To alter it, delete the name and add a new one. Once you've given SYSDBA a new password, click the **Add User** button and add as many new users as you need. Enter every user that will access any database on your server.

When users have access to a server, they can open any database on that server. However, only SYSDBA and the user that created the database — its owner — can access tables or other database objects. Both the database owner and SYSDBA can drop the database, and both can grant object access to other users, by means of the SQL GRANT command:

```
GRANT <list of privileges>
ON <table or object name>
TO <list of users>
[WITH GRANT OPTION]
```

Because views are virtual tables, all the privileges in [Figure 3](#) apply to updateable views, just as they do to tables.

Providing REFERENCES

These privileges are straightforward, with the possible exception of REFERENCES. To understand REFERENCES, suppose you have an Employee table and a Payroll table. You have declared referential integrity to ensure that it's impossible to add a row to the Payroll table for an employee who does not exist in the Employee table. One of your users is a payroll clerk who needs to maintain the Payroll table, but isn't authorized to access the Employee table. The following GRANT statements provide the required access privileges:

```
GRANT ALL ON Payroll TO Fred
GRANT REFERENCES ON Employee TO Fred
```

The grant of all privileges on the Payroll table lets Fred insert, delete, and update rows. When Fred adds a new row to the Payroll table, the grant of REFERENCES on the Employee table allows InterBase to validate that the employee record exists, but doesn't allow Fred to view the data in the Employee table.

There are three cases in which users, before exercising privileges for a particular table, must have access privileges on other tables:

- 1) To insert, delete, or update in a referential-integrity child, the user must have either REFERENCES or SELECT privileges on the referential-integrity master.
- 2) To perform an update or insert on a table with a check constraint, the user must have SELECT privileges on that table.
- 3) If a table constraint includes a query (SELECT), the user must have SELECT privileges on all tables accessed by the query.

Access privileges granted to a table or view will automatically apply to any new columns added. If you don't want privileges to apply to new columns, create a view and grant users access to it. If you add a column, you can decide at that time whether to add it to the existing view, or create a new view for those who need to work with it.

You can also grant access to tables and views on a column-by-column basis. Only two privileges are available at the column level: INSERT and UPDATE. SELECT and DELETE privileges are available only at the table level. At first it might appear that if you can't grant SELECT privileges at the column level, you can't prevent users from seeing certain columns in a table. However, that's not the case. If you want to allow a user to see a subset of columns, create a view that shows only the subset. Grant the user access to the view, but not the table, and grant the view access to the table.

The ability to grant INSERT and UPDATE access at the column level lets you grant users SELECT access to an entire table or view, allowing them to see all the columns, but change the value of only some. To insert new rows in a table or view, the user must have INSERT access for all columns that have a NOT NULL constraint. Any columns users can't access will be set to their default value, if any, or to null when inserting a new row.

Advanced GRANT

The examples of the GRANT command shown so far in this article have granted privileges to a single user. However, any GRANT statement can include a comma-separated list of users:

```
GRANT ALL ON Customer, Orders, Items  
  TO Mary, Fred, PROCEDURE Update_Status
```

This statement not only grants all privileges to Mary and Fred, but also to the stored procedure Update_Status. This adds another dimension by letting users execute stored procedures to make changes on tables that they couldn't otherwise access. The users need only EXECUTE access to the stored procedure. This lets you provide tightly controlled, limited data access. It's not uncommon for one stored procedure to call another; if procedure A calls B, then A must have EXECUTE access for B.

If you need to grant one or more access privileges to all users, grant them to the special user PUBLIC. Note, however, that any privileges you grant to PUBLIC cannot be revoked from individual users. They can be revoked only from PUBLIC, which will revoke them from all users. Privileges granted to PUBLIC are granted only to users, not to stored procedures or views; you must explicitly grant privileges to each stored procedure and view.

GRANTing to Others

With the SQL GRANT command, not only can you grant access privileges to one or more users, you can also give those users the authority to grant privileges to yet other users, through the optional WITH GRANT OPTION clause. The statement:

```
GRANT ALL ON Payroll TO Fred WITH GRANT OPTION
```

not only grants all access privileges to Fred, but also allows Fred to grant any privilege to other users. Use the WITH GRANT OPTION clause very sparingly. If you grant a privilege to Fred, and Fred grants the privilege to others, revoking Fred's privilege will also revoke the privilege from everyone to whom Fred granted it. It's much better to let only the DBA grant and revoke access privileges, which is done through the SQL REVOKE command.

Revoking Privileges

The syntax of REVOKE is almost identical to the syntax for GRANT. For example, the statement:

```
REVOKE ALL ON Payroll FROM Fred
```

revokes Fred's privileges on the Payroll table. Only SYSDBA, or the user who granted the privilege, can revoke it. If two users both grant a privilege to a third user, both must revoke the privilege before the third user will lose it. This is another example of why it's best to allow only the DBA to grant access privileges. The ALL privilege works the same way in a REVOKE statement that it does in a GRANT. The following statement revokes all privileges except EXECUTE from all users of the Employee table:

```
REVOKE ALL ON Employee FROM PUBLIC
```

Watch What You DROP

You need to be aware of one other InterBase security aspect, over which you have no control. No user, including SYSDBA, can drop an object that's referenced elsewhere in the database. For example, even SYSDBA can't drop a table if that table is referenced in the definition of a view, or in a referential-integrity declaration.

Conclusion

InterBase provides excellent security for your data — if you provide physical security for the server and all database copies, use an OS that lets you prevent unauthorized access to the database files, and carefully grant access to objects within the database. By restricting users' access to objects within the database (by coupling GRANT and REVOKE with views and stored procedures), you can provide any level of control you need. ▲

Bill Todd is President of The Database Group, Inc., a database consulting and development firm based near Phoenix, AZ. He is a Contributing Editor of *Delphi Informant*; co-author of *Delphi 2: A Developer's Guide* [M&T Books, 1996], *Delphi: A Developer's Guide* [M&T Books, 1995], *Creating Paradox for Windows Applications* [New Riders Publishing, 1994], and *Paradox for Windows Power Programming* [QUE, 1995]; and is a member of Team Borland providing technical support on CompuServe. He has also been a speaker at every Borland Developers Conference. He can be reached on CompuServe at 71333,2146, on the Internet at 71333.2146@compuserve.com, or at (602) 802-0178.





Your First Component Editor

Creating a Custom Editor for Panel Components

Delphi stands out among application development systems for several reasons. One is that its development environment can be customized to a remarkable degree. For example, developers can create custom component editors. Component editors add options to the pop-up menu that's displayed when you right-click on a component, and provide a dialog box for interactive editing when the component is double-clicked.

The difference between a property editor and a component editor is that the property editor is used for a specific property type on any component (such as the String List editor for the *Items* property of a ComboBox component, or the *Lines* property of a Memo component), whereas the component editor affects only a single component, such as the Fields editor, when you double-click on a Table component.

This article demonstrates how easy it is to create component editors, which are perfect

for providing a customized, user-friendly method for developers to configure custom components. Component editors can be shipped in the same code as a new component, or they can be created and installed independently. In this article, we'll create a new component editor for Panel components. This component editor will display a dialog box that enables us to choose a style and a color for our panel, and preview it at the same time. It will also add two functions to its pop-up menu, one for selecting the style and one for clearing it.

The Dialog Box

First, create a new project. Place two BitBtn components, a RadioGroup component, a Panel, a ColorGrid (from the Samples page), and a Bevel on the main form, and arrange them to resemble the form shown in [Figure 1](#). Then, set the components' properties as shown in [Figure 2](#).

After the appropriate properties are set, we must add a line of code to make the form functional. In the *OnClick* event for ColorGrid1, add this statement:

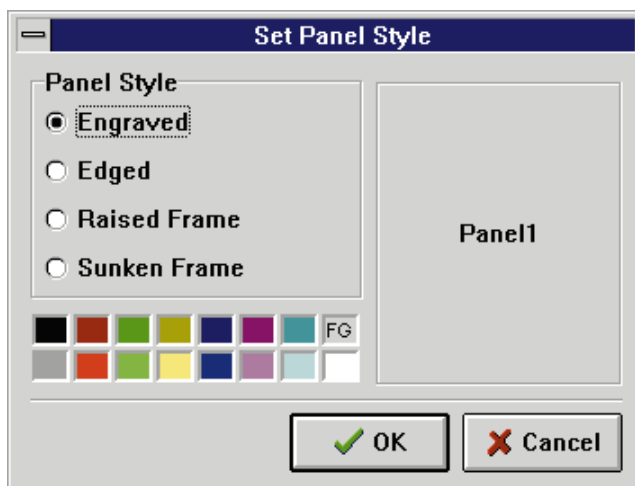


Figure 1: The sample application at design time.

```
Panel1.Color := ColorGrid1.ForegroundColor;
```


Component	Property	Value
BitBtn1	Kind	bkOK
BitBtn2	Kind	bkCancel
RadioGroup1	Caption	Panel Style
	Items	Engraved
		Edged
		Raised Frame
		Sunken Frame
Panel1	ItemIndex	0
	BevelInner	bvRaised
	BevelOuter	bvLowered
ColorGrid1	GridOrdering	go8x2
	BackgroundEnabled	False
	Font.Name	Arial
	Font.Size	8
	ForegroundIndex	7
Bevel1	Shape	bsTopLine
Form1	Caption	Set Panel Style
	Name	PanelStyleSelector
	BorderStyle	bsDialog
	Position	poScreenCenter

Figure 2: Set the values for these component properties.

Then, in the *OnClick* event for *RadioGroup1*, add:

```

case RadioGroup1.ItemIndex of
  0 :
    begin
      Panel1.BevelInner := bvRaised;
      Panel1.BevelOuter := bvLowered;
      Panel1.BorderWidth := 0;
      Panel1.BevelWidth := 1;
    end;
  1 :
    begin
      Panel1.BevelInner := bvLowered;
      Panel1.BevelOuter := bvRaised;
      Panel1.BorderWidth := 0;
      Panel1.BevelWidth := 1;
    end;
  2 :
    begin
      Panel1.BevelInner := bvLowered;
      Panel1.BevelOuter := bvRaised;
      Panel1.BorderWidth := 5;
      Panel1.BevelWidth := 2;
    end;
  3 :
    begin
      Panel1.BevelInner := bvRaised;
      Panel1.BevelOuter := bvLowered;
      Panel1.BorderWidth := 5;
      Panel1.BevelWidth := 2;
    end;
end;

```

We now have a functioning dialog box for our component editor that will enable us to choose a color and four different styles for our panel.

The *TComponentEditor* Object

All new component editors descend from the *TComponentEditor* object. This is an abstract object defined in the *DsgnIntf* unit. When writing a new component editor, we can consider *TComponentEditor* to be declared as:

```

TComponentEditor = class(TObject)
private
  FDesigner: TFormDesigner;
  FComponent: TComponent;
public
  procedure Edit; virtual;
  function GetVerbCount: Integer; virtual;
  function GetVerb(Index: Integer): string; virtual;
  procedure ExecuteVerb(Index: Integer); virtual;
  property Component: TComponent read FComponent;
  property Designer: TFormDesigner read FDesigner;

```

At design time, when you right-click on a component, the component editor associated with that component calls its *GetVerbCount* method. It then calls *GetVerb*, which returns the options to add to the context-sensitive menu. If you then select one of these options, the *ExecuteVerb* method is called, executing the code associated with that menu option.

Double-clicking on a component causes the component editor to execute the *Edit* procedure, which by default calls the *ExecuteVerb* method to execute the code associated with the first option on the pop-up menu. The *FComponent* property gives us access to the component on which the editor is currently working, so we can make appropriate changes to other properties. The *FDesigner* property provides access to the form designer, which is responsible for updating changes made to forms and components at design time.

For this editor, we want to modify the style of the panel, and remove that style. Because the *Edit* procedure, by default, executes the first command in the pop-up menu, the only methods we need to override are *ExecuteVerb*, *GetVerb*, and *GetVerbCount*.

Now that we have designed our form, add *DsgnIntf* to the **uses** clause of our unit. Next, we will derive a new component based on *TComponentEditor*.

Add the following code to the **type** section, after the declaration for the *TPanelStyleSelector* form:

```

TStyleSelector = class(TComponentEditor)
public
  function GetVerbCount: Integer; override;
  function GetVerb(Index: Integer): string; override;
  procedure ExecuteVerb(Index: Integer); override;
end;

```

Our first function, *GetVerbCount*, is called when the right mouse button is clicked, bringing up the pop-up context menu. This function will return the number of options we'll add to the menu. Because we only have two, the code for this function will be:

```

function TGlyphSelector.GetVerbCount: Integer;
begin
  Result := 2;
end;

```

Next, *GetVerb* is called a number of times equal to the result from *GetVerbCount*. This function returns the options to be displayed on the menu, in the order they are to be displayed (see [Figure 3](#)). We need to return the appropriate text, based

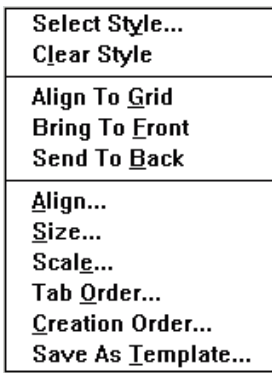


Figure 3: The menu items displayed in order.

on an index passed to the function that represents which position in the menu is needed next. Although we have two options to add to the menu, the index passed to this function is zero-based, so it receives 0 first, then 1. These indexes will be associated with the option that appears in the menu, so the code that performs the action in the next procedure will be based on these indexes. Therefore, our code will resemble:

```
function TStyleSelector.GetVerb(Index: Integer): string;
begin
  case Index of
    0 : Result := 'Select St&yle...';
    1 : Result := 'C&lear Style';
  end;
end;
```

Index 0 is now associated with **Select Style**, and index 1 with **Clear Style**. The only thing left is to write the code that will be performed when the user selects an option. When the user chooses an option from the context-sensitive menu, the *ExecuteVerb* method is called (see Figure 4). It's based on the index associated with that option, so we only need to perform a case statement similar to *GetVerb*, and provide the necessary code for each index. Remember, when a user double-clicks the component, it will automatically perform the code associated with the first option.

For our first option, **Select Style**, we need to show the form we created, and allow the user to select a color and style for the panel. We then assign those properties to the panel, and call *Designer.Modified*. This method informs the form designer that a component has changed visually, and to perform the necessary updates, both on the form and in the Object Inspector. Our second option, **Clear Style**, will simply reset these properties to the defaults for a newly created panel.

In a component editor that is more elaborate than the one illustrated here, we may want to have the component editor reflect the current state of the component when it's activated, rather than just default values. This can be done after the component editor form is created, but before it's presented to the user. For our example, this isn't entirely practical, as a Panel could have properties set to values that wouldn't directly relate to options on our component editor form.

The final procedure that must be added to this unit is *Register*. This procedure will register this component editor with the VCL and Delphi's Open Tools API, and associate it with Panel components. The syntax for this function is:

```
procedure Register;
begin
  RegisterComponentEditor(TPanel, TStyleSelector);
end;
```

The *RegisterComponentEditor* procedure takes two arguments. The first is the component that the editor will be associated with, and the second is the class of the component editor itself. The entire source file is shown in Listing Four beginning on page 37.

Installation and Use

To install our new component editor, save the unit as EDITPANL.PAS. Then perform the following steps:

- For Delphi 1: Click on **Options | Install Components**, then click on the **Add** button.
- For Delphi 2: Click on **Component | Install**, then click on the **Add** button.
- For Delphi 3: Click on **Component | Install Component**.

In each case, add EDITPANL.PAS, as if it were any other component (the EDITPANL.DFM file must be in the same directory as the .PAS file).

After the library is rebuilt, simply drop a Panel component on the form. If you right-click on the component, you'll see the new context menu. Double-clicking the component or choosing the **Select Style** option from the menu will open our form. After you have selected an appropriate style and clicked **OK**, our panel will be modified to reflect the new properties. If we select **Clear Style** from the pop-up menu, it will reset the panel to its default state.

```
procedure TStyleSelector.ExecuteVerb(Index: Integer);
var
  PanelStyleSelector: TPanelStyleSelector;
begin
  case Index of
    0 :
      begin { Select Style was chosen. }
        PanelStyleSelector :=
          TPanelStyleSelector.Create(Application);
        try
          if PanelStyleSelector.Showmodal = mrOK then
            begin
              TPanel(Component).BevelInner :=
                PanelStyleSelector.Panel1.BevelInner;
              TPanel(Component).BevelOuter :=
                PanelStyleSelector.Panel1.BevelOuter;
              TPanel(Component).BorderWidth :=
                PanelStyleSelector.Panel1.BorderWidth;
              TPanel(Component).BevelWidth :=
                PanelStyleSelector.Panel1.BevelWidth;
              TPanel(Component).Color :=
                PanelStyleSelector.Panel1.Color;
            end;
          finally
            PanelStyleSelector.Free;
          end;
        end;
      end;
    1 :
      begin { Clear Style was chosen. }
        TPanel(Component).BevelInner := bvNone;
        TPanel(Component).BevelOuter := bvRaised;
        TPanel(Component).BorderWidth := 0;
        TPanel(Component).BevelWidth := 1;
        TPanel(Component).Color := clBtnFace;
      end;
  end;
  Designer.Modified;
end;
```

Figure 4: The *ExecuteVerb* method.

Conclusion

As we have seen, component editors are easy to create, and can provide an interactive, fun method for developers to use your component. An editor could be complex, with multiple dialog boxes or even full-fledged configuration wizards. Component editors allow you to create highly customized, user-friendly editors for new or existing components, without imposing a system-wide modification that would affect all components, such as a property editor would. Δ

Reference

Developing Custom Delphi Components by Ray Konopka [Coriolis Group Books, 1996].

The file referenced in this article is available on the Delphi Informant Works CD located in INFORM\97\AUG\DI9708JA.

John Ayres is a consultant for Ensemble Systems Consulting in Dallas, using Delphi to produce high-end client/server applications for various Fortune 500 companies. With over 8 years of programming experience, he's worked for a variety of companies, producing a broad range of software, from third-party add-in utilities to games. He keeps himself busy by co-authoring *The Tomes of Delphi: Win32 Core API* (ISBN 1-556225563) and other Windows programming books for Delphi. For more information, visit <http://www.WordWare.com>

Begin Listing Four — EDITPANL.PAS

```
unit Editpanl;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, ColorGrd, StdCtrls, ExtCtrls,
  Buttons, DsgnIntf;

type
  TPanelStyleSelector = class(TForm)
    Panel1: TPanel;
    RadioGroup1: TRadioGroup;
    ColorGrid1: TColorGrid;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    Bevel1: TBevel;
    procedure RadioGroup1Click(Sender: TObject);
    procedure ColorGrid1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

  TStyleSelector = class(TComponentEditor)
  public
    function GetVerbCount: Integer; override;
    function GetVerb(Index: Integer): string; override;
    procedure ExecuteVerb(Index: Integer); override;
  end;

procedure Register;
```

implementation

```
{SR *.DFM}

function TStyleSelector.GetVerbCount: Integer;
begin
  Result := 2;
end;

function TStyleSelector.GetVerb(Index : Integer): string;
begin
  case Index of
    0 : Result := 'Select St&yle...';
    1 : Result := 'C&lear Style';
  end;
end;

procedure TStyleSelector.ExecuteVerb(Index: Integer);
var
  PanelStyleSelector: TPanelStyleSelector;
begin
  case Index of
    0:
      begin { Select Style was chosen. }
        PanelStyleSelector :=
          TPanelStyleSelector.Create(Application);
        try
          if PanelStyleSelector.Showmodal = mrOK then
            begin
              TPanel(Component).BevelInner :=
                PanelStyleSelector.Panel1.BevelInner;
              TPanel(Component).BevelOuter :=
                PanelStyleSelector.Panel1.BevelOuter;
              TPanel(Component).BorderWidth :=
                PanelStyleSelector.Panel1.BorderWidth;
              TPanel(Component).BevelWidth :=
                PanelStyleSelector.Panel1.BevelWidth;
              TPanel(Component).Color :=
                PanelStyleSelector.Panel1.Color;
            end;
          finally
            PanelStyleSelector.Free;
          end;
        end;
      end;
    1:
      begin { Clear Style was chosen. }
        TPanel(Component).BevelInner := bvNone;
        TPanel(Component).BevelOuter := bvRaised;
        TPanel(Component).BorderWidth := 0;
        TPanel(Component).BevelWidth := 1;
        TPanel(Component).Color := clBtnFace;
      end;
  end;

  Designer.Modified;

end;

procedure TPanelStyleSelector.RadioGroup1Click(
  Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
    0 :
      begin
        Panel1.BevelInner := bvRaised;
        Panel1.BevelOuter := bvLowered;
        Panel1.BorderWidth:= 0;
        Panel1.BevelWidth := 1;
      end;
    1 :
      begin
        Panel1.BevelInner := bvLowered;
        Panel1.BevelOuter := bvRaised;
        Panel1.BorderWidth:= 0;
        Panel1.BevelWidth := 1;
      end;
    2 :
```

```
begin
  Panel1.BevelInner := bvLowered;
  Panel1.BevelOuter := bvRaised;
  Panel1.BorderWidth:= 5;
  Panel1.BevelWidth := 2;
end;
3 :
begin
  Panel1.BevelInner := bvRaised;
  Panel1.BevelOuter := bvLowered;
  Panel1.BorderWidth:= 5;
  Panel1.BevelWidth := 2;
end;
end;
end;

procedure TPanelStyleSelector.ColorGrid1Click(
  Sender: TObject);
begin
  Panel1.Color := ColorGrid1.ForegroundColor;
end;

procedure Register;
begin
  RegisterComponentEditor(TPanel, TStyleSelector);
end;

end.
```

End Listing Four





By *Dan Miser*

Component Creation

Construction Guidelines for VCL Components

Components are the essence of Delphi. A well written and well organized component can aid Delphi programmers of all levels. Conversely, a poorly written component can keep any programmer from being productive. This article outlines some simple steps to help ensure your components end up being put to use.

File Organization

The first thing a component user must do when they receive a component is install it to the Component palette. There are three choices for distribution:

- all .PAS files;
- a .PAS registration unit, with supporting .DCU files; or
- all .DCU files.

If there is some private interest that needs to be protected, .DCU files appear to be an attractive option. After all, the component user can't look at the component's source code if it isn't distributed. In addition, if the *Register* procedure of the component is located inside the .DCU, the component user won't have the power to change on which page the component is installed. This defeats the purpose of Delphi's open and configurable environment.

Basic Delphi etiquette requires a component writer to distribute the registration unit as a .PAS file. Supporting .DCU files can also be distributed, if necessary. This allows the component user to install the component on any Component palette page. However, Borland's .DCU format is subject to change (and has changed frequently over the years). A change in the .DCU format from Borland can mean incompatibility for all .DCU files not created with the same version of Delphi (sometimes even minor versions). This means that a component user must have faith that they

can obtain an updated .DCU that can match the .DCU format-du-jour. The moral of this story is that a component user must be offered some way to obtain source code for all of the component's files. For these reasons, many users don't even consider employing a .DCU component solution.

In addition, if the component has any special property editors or component editors, they should be placed in the same unit as the *Register* procedure. This is how Borland distributes their VCL components. For example, looking at `\Delphi\Lib\stdreg.pas` will reveal several property editors and a component editor in the same file that contains registration routines for the standard components.

However, if the property editor contains a form, the property editor declaration should reside in that form's unit. By placing the declaration there, Delphi will only add the overhead of the property editor's form to `COMPLIB.DCL`. Because the property editor's form will never be referenced by an application, either directly or indirectly, the property editor will not be compiled into the final .EXE.

Tying a Component to a DLL

There are two ways to access a DLL function: static import and dynamic import. Using the static import method requires that the DLL filename be known at compile time. In addition, after the function is imported from a DLL via static importing, that func-

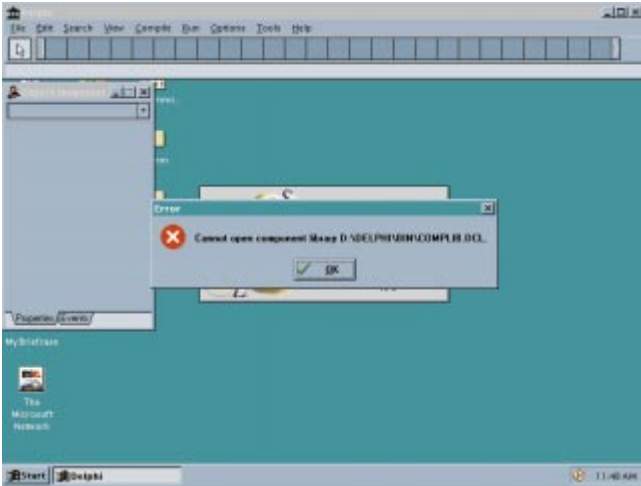


Figure 1: Delphi fails to load COMPLIB.DCL because of bad DLL linking.

tion must exist in that DLL. If the DLL cannot be found, Windows won't let the process run.

Dynamic importing requires more work, but the payoff is tremendous: The programmer can recover gracefully from an error while trying to load a DLL. In addition, the name of the DLL doesn't need to be known at compile time, thus adding flexibility to the design.

When writing a component wrapper around a DLL, the process responsible for loading the DLL is Delphi. This is because when the component was installed in Delphi, it was placed in Delphi's component library, COMPLIB.DCL. If the link to the DLL is static and the DLL doesn't exist, COMPLIB.DCL will render Delphi into a useless state (see [Figure 1](#)). To correct the error, the user must know what caused it. Even then, the user has no way of knowing the name of the offending DLL, or if this error was caused by only one missing DLL. Instead, the usual course of action is to copy a backup version of COMPLIB.DCL to the \Delphi\Bin directory.

The best way to ensure this problem does not occur is to use dynamic importing of the DLL functions. This requires a call to *LoadLibrary*, followed by a call to *GetProcAddress*, for each and every function that must be called in the component (see [Figure 2](#)). The call to *AddExitProc* in the unit's **initialization** section is included for compatibility with Delphi 1 and 2. In Delphi 3, Borland recommends the **finalization** section be used instead of *AddExitProc* for exit behaviors such as this.

By calling *LoadLibrary*, the error checking can now occur at the component level. This provides more control than the error checking and reporting done by Windows. In [Figure 2](#), the component checks to make sure the DLL was loaded successfully. If it wasn't, a message is shown explaining the problem. (Note that if an exception is raised here, Delphi will still be unable to load the component library.) Next, the functions from the DLL are imported using *GetProcAddress*. If there is an error importing these functions, the result of the *GetProcAddress* call will be `nil`. Otherwise, the result will be a

```
implementation

const
  DLLName = 'GOOD.DLL';
  DLLProcedure1 : procedure = nil;
var
  DLLHandle : THandle;

function LoadDLL : Boolean;
var
  OldFlag: Word;
begin
  Result := False;

  { Turn off system error message. }
  OldFlag := SetErrorMode(SEM_NOOPENFILEERRORBOX);
  DLLHandle := LoadLibrary(DLLName);

  { Restore system error messages. }
  SetErrorMode(OldFlag);

  {$IFDEF WIN32}
  if DLLHandle = 0 then
  {$ELSE}
  if DLLHandle <= HINSTANCE_ERROR then
  {$ENDIF}
  begin
    ShowMessage('Cannot find file: '+DLLName);
    Exit;
  end;

  @DLLProcedure1 := GetProcAddress(DLLHandle, 'DLLProcedure1');
  Result := True;

end;

procedure UnloadDLL; far;
begin
  FreeLibrary(DLLHandle);
end;

initialization

  if LoadDLL then
    AddExitProc(UnloadDLL);

end.
```

Figure 2: A dynamic import.

valid address which your component will use to call the DLL function. This is more sophisticated than letting Delphi display a cryptic error message.

Note also that the return result of *LoadDLL* is an indication of whether the DLL was loaded successfully. If it was, *ExitProc* is set up to free the DLL when the application ends (or the component is destroyed).

16- and 32-bit Resource Files

The only difference between 16- and 32-bit resource files is the signature block. The format of the individual standard resources has not changed from 16- to 32-bit environments. This means that when all the appropriate resources are bound together into one resource file, the difference is slight. However, the difference is enough to require every resource file built in 16-bit Windows to be rebuilt for access in 32-bit Windows. The easiest way to keep these files portable is to save the 16-bit .RES file as

an RC script, while saving all the standard resources in their native file format.

An RC file is nothing more than an ASCII file with a specific format. RC files are used by resource compilers to identify which resources to include, and where to find them. In its simplest form, the format of an RC file is as follows:

```
nameID ResourceType filename
```

Some of the basic *ResourceTypes* available for this format are: BITMAP, CURSOR, and ICON. A sample RC script would look like this:

```
TCLICKLABEL BITMAP "CLKLABEL.BMP"
```

This would assign the bitmap located in the file CLKLABEL.BMP to a bitmap resource type, and would be identified with the resource name TCLICKLABEL. Because the RC script and bitmap file format are both accessible in 16- and 32-bit Windows, the only thing that must be done is compile the resource file to a .RES format for the appropriate environment.

Both Delphi 1 and 2 contain a resource compiler (written by Borland) that will create resource files from RC script files for both environments. The original Microsoft Windows Resource Compiler for 16- and 32-bit environments can be obtained from any number of Microsoft Developer resources.

To compile an RC script to a 16-bit resource file, make sure \Delphi\Bin is in the path, and type:

```
BRC -R -FOFILENAME.R16 FILENAME.RC
```

32-bit resource files are created in a similar fashion. Make sure the Delphi 2 binary directory is in the path, and type:

```
BRC32 -R -FOFILENAME.R32 FILENAME.RC
```

The 32-bit version also has the capability to create 16-bit resource files. This can be accomplished by adding a -31 flag (for Windows 3.1) to the previous command:

```
BRC32 -R -31 -FOFILENAME.R16 FILENAME.RC
```

For more information on RC files, including a complete description of resource types available, see "RC (Resource Statements)" in the online Help in WINAPI.HLP.

Now that the resource files are created, it's time to put them to use.

16- and 32-bit Component Portability

All well written components should have their own icon to display on the Component palette. This helps to readily identify a component. If this file cannot be found, default icons will be used to identify the component. While this may be fine for extremely limited production, it is of critical importance to distinguish components — both in name and icon

— from all other components. Imagine how difficult it would be to use the Component palette if everyone decided to not create custom icons.

Delphi has a clearly documented system for registering components. For components that need to be installed in the component library, Delphi finds the *Register* procedure in that unit to determine which component should be installed, and to which Component palette page it should be added. Next, Delphi looks for a .DCR file with the same file name as the registration unit. This .DCR file is simply a resource file that contains a bitmap. This bitmap will be placed on the Component palette to represent the newly registered component. If Delphi finds the .DCR file, it binds that resource file implicitly into COMPLIB.DCL.

After Delphi has linked the resource file into the component library, it looks for a bitmap resource with the same name as the installed component's class name. For example, if a component named *TClickLabel* is being installed, the bitmap resource must be named TCLICKLABEL for Delphi to attach the bitmap to the component. In Delphi 1, it's necessary to name this bitmap resource using all capital letters. This is in direct contradiction to the documentation, but it is necessary. If Delphi doesn't find this resource, it assigns the component's parent's bitmap to represent the component on the Component palette.

This information has led many component authors (who wish to provide a 16- and 32-bit solution) to adopt the file organization represented in Figure 3. However, this scheme severely limits unit filenames. As you can see, filenames can be a maximum of 6 characters plus the 16/32 designation to keep the files easily documented. In the finite world of 8.3 filenames, it is hard enough to come up with a meaningful and relevant unit filename. Add to this the population of existing and future Delphi components, and the chances of a unit name collision increase dramatically.

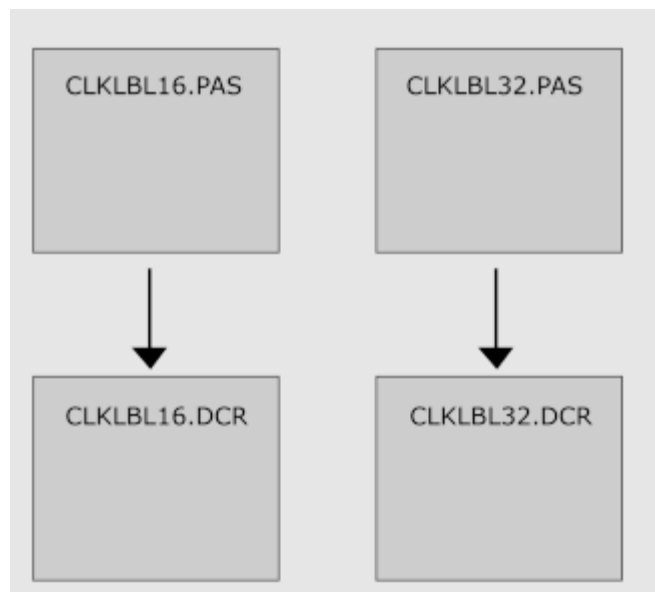


Figure 3: File organization for automatic component installation.

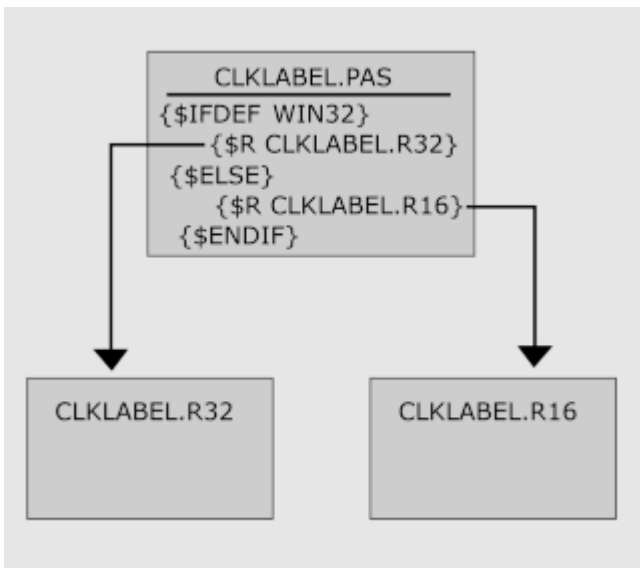


Figure 4: File organization for optimized component installation.

```

unit CompReg;

interface

procedure Register;

implementation

uses
  Classes, ClkLabel;

{$IFDEF WIN32}
{$R *.R32}
{$ELSE}
{$R *.R16}
{$ENDIF}

procedure Register;
begin
  RegisterComponents('DMC', [TClickLabel]);
end;

end.

```

Figure 5: A typical registration unit for a component.

Borland provides a mechanism to deal with the minutiae of component installation. It works well for most cases; however, in true Borland tradition, if their way doesn't work for all cases, their architecture is open enough to allow anyone to change it. By implementing the file organization shown in [Figure 4](#), Delphi can share a component between the two environments with minimal work. [Figure 5](#) is a typical component registration unit.

Notice in [Figure 4](#) that we did not name our resource files with the .DCR extension. Therefore, Delphi cannot implicitly link the resource file, and should assign a default icon. However, the conditional directive `{$IFDEF WIN32}`, combined with the include resource file directive `{$R}`, will bind the appropriate resource file to `COMPLIB.DCL` explicitly. Delphi doesn't care how the component's bitmap was placed in `COMPLIB.DCL`; it only cares about finding a bitmap resource with the same name as the component.

There's an additional benefit to placing the component's palette bitmap in a non-DCR resource file. All other resources for this component can now be placed in that resource file. This will centralize all resources, making it easier to manage. For example, if a custom cursor was also required for a component, it could be placed in the same file as the component bitmap by simply adding the appropriate line to the RC file.

Conclusion

This article has shown several pitfalls to avoid during the intricate process of component creation. In addition, tips on how to automatically share a component between 16- and 32-bit Windows were given. By following these simple steps, Delphi programmers everywhere can start to realize the potential of Delphi's most powerful force — well written components. ▲

Dan Miser is a Senior Software Engineer at COMPS InfoSystems in San Diego. He recently spoke about the new features of Delphi 3 at the Borland Developers Conference in Nashville. He is also a Borland Certified Delphi Client/Server Developer. You can contact him at <http://www.iinet.com/users/dmiser>, or dmiser@compu-serve.com.





By *James Callan*

Serving Many Masters

Implementing Many-to-Many Database Relationships

It's been said that "No man can serve two masters." And while this may hold true for people, it need not for data sets. Most practical client/server database applications mandate supporting many-to-many relationships between tables. Unfortunately, book and manual examples focus on single-table and master-detail forms, while detail-detail forms are given scarce treatment. This article aims to correct the oversight by giving many-to-many relationships their due; it shows you how to recognize the need for many-to-many relationships, and how to implement them in Delphi.

Real-World Databases

A database is a model of the real world, adequate for representing interesting facts about real-world entities. Typically, characteristics of each entity (person, place, thing, or concept) and the relationships it has with other entities are of primary importance. For example, a company's payroll system will use a table containing data about employees. How else would it know whom to pay?

Similarly, an order-entry system would maintain data about orders. This order data likely would be stored in a master-detail, mated-table pair. One table contains data about the order, while the mated table keeps data about each line of the order. Each order is uniquely identified by a primary key, e.g. an order number. Order lines are correspondingly matched with their related order by having their primary keys contain the order numbers to which they relate. This dyad is termed the *standard master-detail relationship*, and is extremely common in real-world systems.

Master-detail tables represent many realistic situations, but can be inadequate for some real-world scenarios. Consider a purchasing system, for instance. Parts are purchased from suppliers. Each part can be supplied by multiple suppliers. Equally important is

knowing which parts are supplied by a specific supplier. Representing this situation in a master-detail arrangement would fail to produce the desired results. The classic master-detail relationship breaks down between the Parts and Suppliers tables.

This article focuses on recognizing when to abandon master-detail relationships and use alternate techniques. We will first introduce a moderate example, and demonstrate how the real-world entities are modeled. We'll then enhance the model to produce a workable table design, demonstrate how to build many-to-many relationship support into your forms, and conclude with some design tips.

Matchmaker, Matchmaker

Consider the problem of matching people with tasks for which they are skilled: We have people; we have skills; we also have tasks. For our example, we'll consider people and skills.

Each person can acquire multiple, useful skills. Likewise, multiple people may possess the same desirable skill. As in the more complex procurement problem we considered earlier, the matchmaker problem has a many-to-many relationship at its core. Two useful questions this application could answer are: "What skills has a given person acquired?"



Figure 1: The ERD showing the two essential skills.

and, “Which people have acquired a specific skill?” Ultimately, we’ll build a form that answers both these questions. First, however, we must design a database.

Many-to-Many Modeling

Planning is the first step in building anything worthwhile. In the database world, a data model is the equivalent of planning. Data models are comprised of, among other things, entities and relationships. Since Codd and Date laid the foundations of relational databases nearly thirty years ago, many modeling techniques have been introduced. Entity-relationship modeling has stood the test of time.

In an Entity-Relationship Diagram (ERD), entities are represented by boxes, while relationships are represented by lines drawn between the boxes. A relationship between two entities can be one-to-one, many-to-one, one-to-many, or many-to-many. With a many-to-one relationship between entity A and entity B, we know that one or more instances of A are related to a single instance of B. Similarly, with a many-to-many relationship between entity A and entity B, we know that one or more As are related to one or more Bs.

Let’s get specific and model our match-making problem. At first glance, we need two entities. One entity — we’ll call it PERSON — will represent a person. (By convention, entities are given singular names.) The other entity — we’ll call it SKILL — will represent a skill that could be acquired by a person. In determining how the entities PERSON and SKILL relate to one another, we must consider both perspectives of their relationship. We’ll consider these perspectives in the context of a company that needs to match the right person to the right task.

Each person in the company can contribute one or more skills. Similarly, each skill can be one of many contributions that a person makes to the company. These two basic requirements result in the Entity-Relationship Diagram illustrated in Figure 1. The figure uses an extended form of standard Information Engineering notation used for diagramming ERDs.

The end-points of the figure’s relationship line hold special significance. Each has three prongs and an oval. The three-pronged symbol, termed a *crow’s foot* in database circles, indicates one or more instances. The oval indicates whether a relationship is optional. Formally, the symbols mean that each person *may* (due to the “optional” indicator) be related to one or more skills. Americanized interpretation of the same symbols would conclude that each person is related to zero or more skills. Both interpretations are equivalent.

Taking the other side, we can conclude that each skill may be related to one or more persons. Astute readers will notice that

we’ve lost a bit of the richness associated with our original requirements. Don’t be alarmed. The words above and below the relationship line are used to give meaning to the relationship. Because there are two perspectives to every relationship, we need two phrases. By convention, the relationships are read from top to bottom, and from left to right.

Looking again at **Figure 1**, we read the ERD as follows: “Each PERSON may be a value-added contributor via one or more SKILLS, and each SKILL may be a value contribution for one or more PEOPLE.” Now this sounds a lot like the original requirements. We’ve constructed a model of the matching problem. Unfortunately, our model has a problem.

Resolving Many-to-Manys

The problem with many-to-many relationships is that, despite being depicted that way in diagrams, they’re seldom implemented as two tables. Decades of analysis have taught us that beneath every many-to-many relationship lies a richer semantic relationship begging further study. This additional analysis often leads to improved models and better applications.

Let’s reconsider the procurement problem. In addition to knowing merely that a supplier supplies a part, you also want to know what the supplier charges for the part. You may also want to keep a history of the supplier’s price changes. You may even need to deal with two suppliers calling the same part different names. In the match-making case, we must deal with the issue of *recency*, i.e. how current is a person’s skill? And to what degree has the skill been certified? Likewise, we may have to contend with the technology-obsolescence juggernaut: Training and skills must be updated constantly.

To address these issues, we’ll need to keep track of facts such as when a particular skill was acquired, whether the acquisition involved certification, and who made the certification. How long a given skill can be deemed “useful” could also be helpful. Given these additional requirements, we can conclude that the act of acquiring a skill is itself interesting. For want of a better term, we’ll call this “act” a PERSONAL SKILL ACQUISITION. We can then resolve the many-to-many relationship with two many-to-one relationships, as shown in **Figure 2**.

Our expanded model introduces a new type of end-point for relationship lines. The short bar is also an optionality



Figure 2: Resolving the many-to-many relationship with two many-to-one relationships.

indicator, but it means “mandatory” rather than “optional.” The new model in Figure 2 can thus be read: “Each PERSON may be a contributor via one or more PERSONAL SKILL ACQUISITIONS, and each PERSONAL SKILL ACQUISITION must be acquired by one and only one PERSON.”

Continuing, it reads: “Each PERSONAL SKILL ACQUISITION must be for one and only one SKILL, and each SKILL may be acquired through one or more PERSONAL SKILL ACQUISITIONS.”

This new, resolved ERD contains richer meaning, and truly captures the essence of our problem. The ERD also permits us to answer the additional questions posed earlier. A more compelling reason to resolve the many-to-many relationship is to give some semantic significance to what will ultimately become a “link” or “join” table in our application. We can now proceed with our table and application design.

Although this example application was written using Delphi 2 and makes use of the new DataModule component, it can be constructed in Delphi 1 without the data module. (It will work in Delphi 3 without changes.)

Application Overview

The goals of our skill-sets application are to be able to answer which people have acquired a specific skill, to know the particulars of their skill acquisition, and to be able to ascertain all the skills a particular person has acquired. Another useful feature would be the ability to add, edit, and delete entries. These basic requirements are sufficient to demonstrate how to implement many-to-many relationships.

We’ll keep it simple, yet interesting. The People table will maintain an employee ID, first name, and last name. (By convention, tables are named in the plural.) The Skills table will maintain a skill name (SkillID), a description, the type of skill (Regulated, Unregulated, Normal, etc.), whether the skill is renewable, and the number of months that the skill is expected to be considered current (Duration). These extra details will serve to make the example more realistic. We’ll keep the intersection-entity information from the PERSONAL SKILL ACQUISITION in a join table called PerSkill (also a table-naming convention). The PerSkill table will contain the employee ID, skill ID, the date the person acquired the skill (Acquired), whether the skill was certified (Certified), who did the certification (Certification), and any notes or details (Details) about this person’s skill.

To answer the question of what skills Tom, Dick, and Harry have, we’ll need a form with people in one region, and skill information in another. To answer the secondary question of who has a particular skill, we’ll also need a form with skills in one region, and information about people in another. Both these questions require information from all three tables. We’ll structure the tables in a master-detail-detail arrangement using dynamic SQL queries.

People.db					
Field Name	Type	Size	Key	Required	Default
EmpID	A	10	*	Y	
FirstName	A	20		Y	
LastName	A	25		Y	

Skills.db					
Field Name	Type	Size	Key	Required	Default
SkillID	A	10	*	Y	
Descrip	A	35		Y	
SkillType	A	1		Y	N
Renewable	A	1		Y	N
Duration	S				9999

PerSkill.db					
Field Name	Type	Size	Key	Required	Default
EmpID	A	10	*	Y	
SkillID	A	10	*	Y	
Acquired	D		*	Y	
Certified	A	1		Y	N
Certification	A	10			
Details	M	128			

Figure 3: Using the Database Desktop, create the People, Skills, and PerSkill tables.

To best illustrate some of the techniques for representing many-to-many relationships in applications, we’ll purposely limit ourselves to two forms. Therefore, in two forms we must be able to add, edit, and delete people, skills, and personal skills. We also need to answer the two burning questions posed earlier. Although a variety of two-form methods could accomplish this task, we’ll use a Browser form that permits editing on both the People and Skills tables, and answers the two questions. The second form will be used to capture the acquisition of new skills. This arrangement, as noted in the “Afterthoughts” section, permits generalization to more complex queries later.

Creating Tables

Using the Database Desktop, create the People, Skills, and PerSkill tables as depicted in Figure 3. Set the default values for the SkillType, Renewable, Duration, and Certified fields as indicated in the figure. To set the default values for a field, select the field, choose **Validity Checks** under **Table Properties**, and type the default value in **Default Value**.

Under the **Tools** menu of the Database Desktop, use the **Alias Manager** to create a public alias, named **Many2Many**, that points to the directory where you created the tables. We’ll use this alias later to de-reference the tables we just created.

Rousing the Browser

Figure 4 illustrates the final layout of our completed Browser form. Using Delphi, begin a new application and drop a Panel and two GroupBox components, relatively equal in size, on the default form. Align the Panel to *alBottom* and align each of the GroupBoxes to *alTop*, as shown in Figure 4. Change the form’s *Caption* property to **SkillSets Browser**. Set its *ClientHeight* to 400, and its *ClientWidth* to 541. Next, set its *BorderStyle* to *bsDialog*. Change the top group box’s *Name* property to **MasterGB**, and change its *Caption* to **&People**. Name the bottom

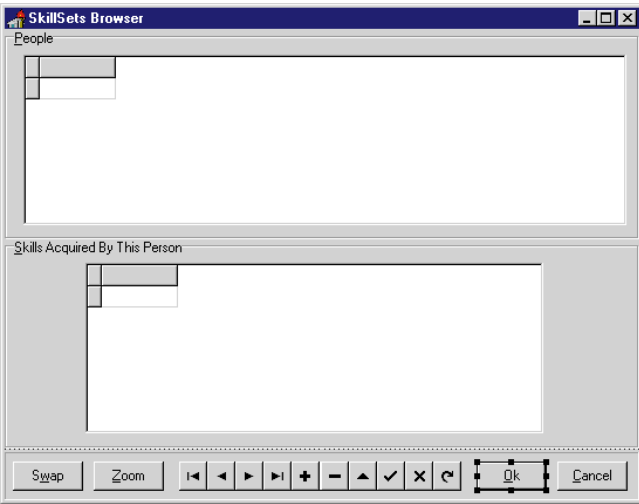


Figure 4: The final layout of the Browser form.

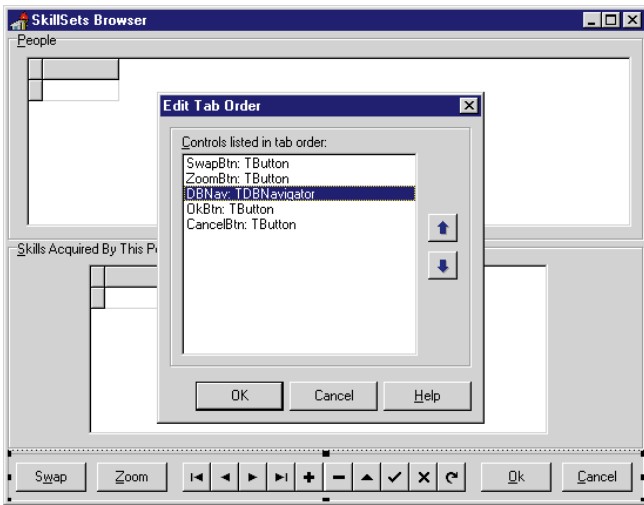


Figure 5: Setting the tab order of the objects.

group box DetailGB, and make its *Caption* read &Skills Acquired By This Person. Set the *Height* property for both group boxes to 178.

Remove *Panel1's Caption*, and set its *BevelOuter* property to *buLowered*. Next, position four Button components and a DBNavigator on the Panel, as illustrated in Figure 4. A Button width of 60 makes this possible. Name the DBNavigator DBNav, and set its *TabStop* property to *True*. Set the Buttons' *Name* and *Caption* properties from left to right, as follows: *SwapBtn*, *S&wap*; *ZoomBtn*, *&Zoom*; *OkBtn*, *&OK*; and *CancelBtn*, *&Cancel*.

Next, drop a DBGrid component in each of the group boxes. Name the top grid *MasterGrid*, and position it by setting its *Left* property to 16, its *Top* to 21, its *Width* to 514, and its *Height* to 145. Similarly, name the lower grid *DetailGrid* and set its *Left* to 69, *Top* to 20, *Width* to 390, and *Height* to 145. Under the *DetailGrid's Options* property, set *RowSelect* to *True* (more about this later). Right-click on the form surface (try between the lower group box and *Panel1*), and set the tab order to *MasterGB*, *DetailGB*, then *Panel1*. Right-click on *Panel1* and set the tab order as shown in Figure 5. Finally, set *OkBtn's Default* property to

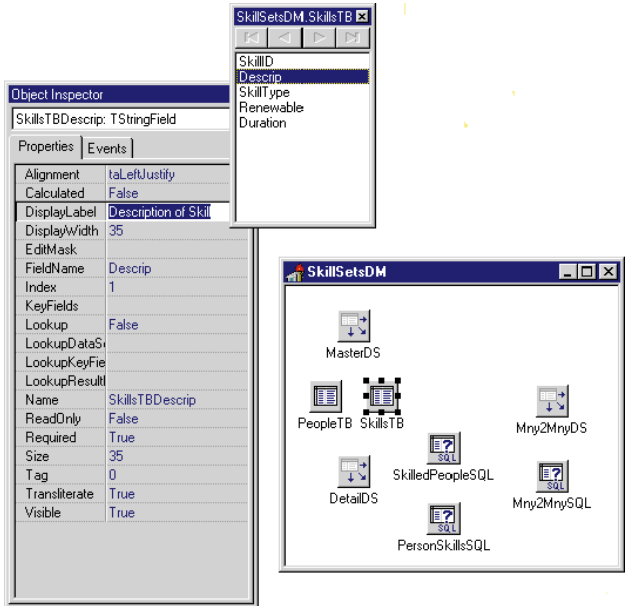


Figure 6: The arrangements of the components for our data module.

True. Your form should look exactly like Figure 4. Save the form as *Browser*, and the application as *Mny2Mny*.

Data, Data Everywhere

Following an object-oriented approach, we'll separate our form from our data with a DataModule component. If you ever need a many-to-many form similar to *Browser* for a production application, you'll be glad we did this. How does it go? "Theft is the highest form of reuse." If you're following along in Delphi 1, simply lay out the components on one of the grids, exactly as you always have before.

Within Delphi's *File* menu, select *New Data Module* to create a new data module for the project. Name the data module *SkillSetsDM*. On the data module, drop three *DataSource* components, naming them *MasterDS*, *DetailDS*, and *Mny2MnyDS*. Next, add two *Table* components, naming them *PeopleTB* and *SkillsTB*. Next, drag over three *Query* components, naming them *SkilledPeopleSQL*, *PersonSkillsSQL*, and *Mny2MnySQL*. Finally, arrange the components as shown in Figure 6.

This last step isn't necessary, but you might find it useful. Human-factor experiments at Oracle Corp. in the United Kingdom found that spatial arrangements assist people in remembering complex ERDs. I suspect the same would hold true for data modules, if similar experiments were performed today. Set the *MasterDS* object's *DataSet* property to *PeopleTB*. Set the *DetailDS* object's *DataSet* property to *PersonSkillsSQL*, and its *AutoEdit* property to *False*. Set the *Mny2MnyDS* object's *DataSet* property to *Mny2MnySQL*.

Set *PeopleTB's DatabaseName* to *Many2Many*. (Remember the alias we created when we created the tables?) Set its *TableName* to *People*. Note: If you delete the *.DB* extension that appears by default in local table names, you'll find it easier to upsize your application later. Now, double-click the

PeopleTB component, and right-click the Field Inspector to **Add New Fields** for the *People* table. Add all three fields.

Similarly, set *SkillsTB*'s *DatabaseName* to *Many2Many*, and its *TableName* to *Skills*. Use the Field Inspector to add all the skill fields to the data module as well. Using the Field Inspector with the Object Inspector (as shown in Listing Five beginning on page 51) set the *SkillsTBDescription* field's *DisplayLabel* to *Description of Skill*, and its *DisplayWidth* to *38*. Change *SkillsTBSkillType*'s *DisplayLabel* to *Type*. Change *SkillsTBRenewable* and *SkillsTBSkillType*'s *Alignment* property to *taCenter*. Next, change *SkillsTBSkillID*'s *DisplayLabel* to *Skill ID*. Finally, change *SkillsTBDuration*'s *DisplayLabel* to *Duration (Months)*. Save the data module as *SkillSet*.

Checking Our Perspective

Before we get too far, we probably should make sure we're on the right track. Because one of the purposes of the Browser form is to answer the two questions about people and skills, we'll need the capability to change perspectives. What do I mean by perspective?

Referring back to **Figure 2**, we find that to answer the question "What skills do Tom, Dick, and Harry have?" we need only consider the data model from the PERSON(s) perspective. We begin with a PERSON, and follow the relationship to the PERSONAL SKILL ACQUISITIONS for the PERSON. Any additional information we need can be obtained by following the relationship from the PERSONAL SKILL ACQUISITION down to the related SKILL.

To answer "Who has this skill?" we switch perspectives, beginning instead with SKILL. We follow the relationship from SKILL to the PERSONAL SKILL ACQUISITION for a particular skill. Again, any additional information can be obtained by following the relationship from the PERSONAL SKILL ACQUISITION down to the related PERSON. In the first case, the PERSON is the driving force; in the second, SKILL drives.

Let's do a partial integration effort, and check to make sure we can properly swap perspectives. We'll need a bit of code, however. To swap between perspectives, we'll encapsulate the code required to establish the field characteristics for the fields that make up the data sets in each view. I also anticipate that we'll need a variable to keep track of which perspective we're currently viewing. A Boolean flag will do nicely.

Make the *SkillSetsDM* the active window, and press **F12** to view the code. Add the following declarations:

```
private
  { Maintains many-to-many viewing perspective. }
  ViewingByPeople: Boolean;
public
  procedure ByPeople;   { To view by People. }
  procedure BySkills;  { To view by Skills. }
end;
```

Dynamically Changing Field Properties

When we defined the *SkillsTB* component, we set up static field components and provided display properties to control their captions and width. This can also be done at run time (again, see Listing Five).

The *PeopleTB* fields are dynamically altered at run time in the *ByPeople* method. This need not be repeated as we do here, but it is a useful technique for certain types of data-driven forms. Depending on your application, you could retrieve your display characteristics from an .INI file.

Grid Resizing

When I first set up the Browser, I realized the size of the *People* grid was dramatically smaller than the grid required for *Skills*. After swapping, the grid looked off-center and rather untidy. To correct this problem, I determined some re-sizing to improve the appearance. The complete source for the Browser is in Listing Six on page 52. You may need to play around with the numbers to make it perfect on your system. (Resolution-independent, dynamic resizing is a topic for another article.)

After adding *SkillSet* to your *uses* clause, use the Object Inspector to change *MasterGrid*'s *DataSource* property to *SkillSetsDM.MasterDS*. Database components within used data modules become visible to drop-down lists in the Object Inspector after the modules are added to your form's *uses* clause. Now, go ahead and set the *DataSource* for *DetailGrid* to *SkillSetsDM.DetailDS* as well.

To the form's *OnActivate* event, add a call to the *ViewByPeople* method. Double-click on *SwapBtn* to add to its *OnClick* event handler the following code:

```
procedure TForm1.SwapBtnClick(Sender: TObject);
begin
  if ViewingByPeople then
    ViewBySkills
  else
    ViewByPeople;
end;
```

The final step for this first integration is to add a *Close* statement to the *OnClick* events for *OkBtn* and *CancelBtn*. Compile and run the application.

Testing the Swap

You should see the Employee ID, First Name, and Last Name, as shown in **Figure 7**. For now, don't be concerned about the bottom grid. Now click the **Swap** button, or simply press **Alt W** to swap the perspective. You should now be looking at a view of Skills, as illustrated in **Figure 8**. Again, other than the group box's caption, don't worry about the lower grid for now.

Referring back to our data model in **Figure 2**, you'll find that we're toggling the top grid between the two bottom entities (PERSON and SKILL) in the ERD. If you haven't played around with dynamic forms and control re-sizing, this will be a pretty neat trick.

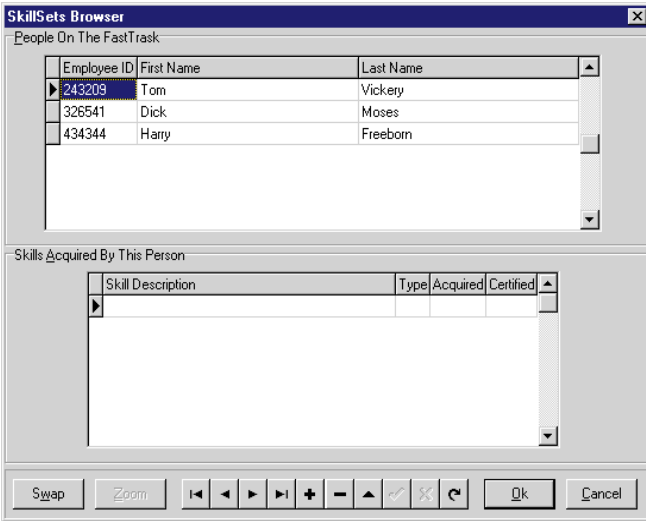


Figure 7: Viewing information by people.

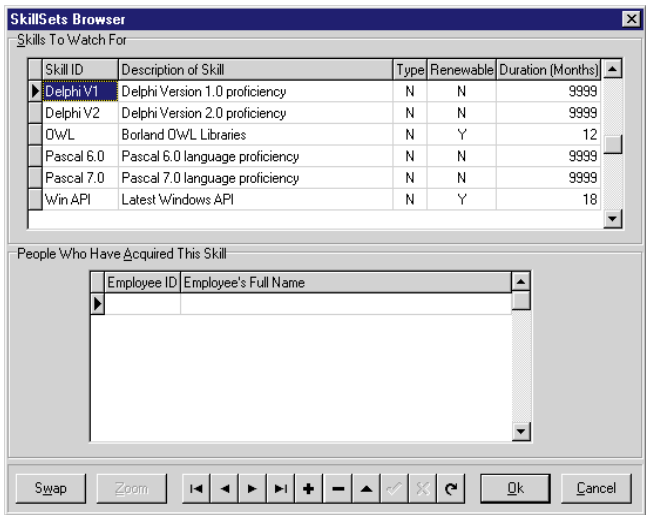


Figure 8: Viewing the data by the skills of each person.

Taking Inventory

The first question we'll attack is "What skills does a particular person have?" Let's go back to the SkillSets data module, and work on our queries. Change the *PersonSkillsSQL*'s *DatabaseName* to *Many2Many*, and its *DataSource* to *MasterDS*. The *DataSource* property for a Query component is used to automatically instantiate any parameters in your query during the *DataSource*'s *OnChange* event. It's primarily used to coordinate master-detail DataSet relationships.

To answer our question, we need to follow the PERSON entity to the PERSONAL SKILL ACQUISITION, and from there to the SKILL. Each relationship on the ERD represents a join clause in our SQL statement. Thus, our SQL statement will be a three-table, dual-join SELECT. Next, click on the SQL glyph to raise the String list editor for the query's SQL statement, and enter the SQL statement shown in Figure 9.

This SELECT sets up table aliases A, B, and C to assist in resolving prospective ambiguities between column names. Using the :EMPID parameter that it gets from the *MasterDS*

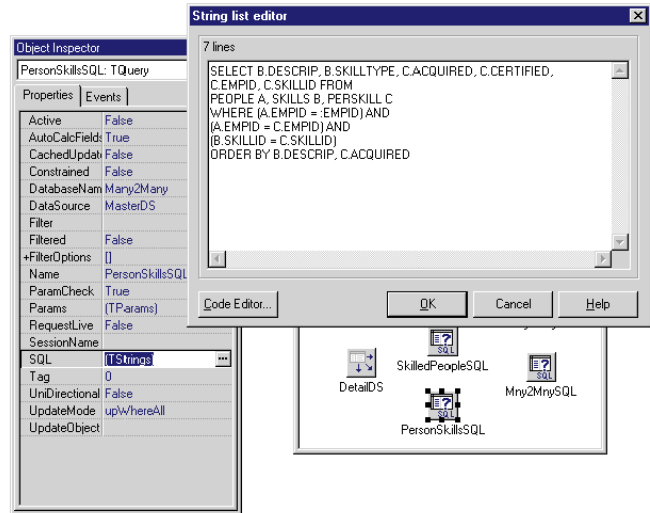


Figure 9: Using the String list editor to enter the SQL SELECT statement.

DataSource, the query first selects the appropriate person from the People table, using *EmpID* (the primary key for the table). It then uses the *EmpID* column to perform a join on the *EmpID* column of the *PerSkill* table (partial primary key) to locate any records for that person. The last step is to use each record of the resulting intersection set to join the *SkillID* column of the *PerSkill* table with the *Skills* table (primary key for the table). The resulting columns come from both the *PerSkill* and *Skills* tables. The final *ORDER BY* clause sorts the resulting data appropriately.

To verify the query, complete the following steps. Inspect the Parameters dialog box, and set the *EMPID* data type to *String*. Next, open the People table by setting the *PeopleTB*'s *Active* property to *True*. Finally, open the query by setting the *PersonSkillsSQL*'s *Active* property to *True*. If all is well, you should see both grids activated on the Browser. To know for sure, of course, we need data.

Details, Details

To tidy up the lower grid, we need to perform some cosmetic surgery on the field objects. Double-click the *PersonSkillsSQL* component, and add the field objects just as for the tables earlier. Change *PersonSkillsSQLDESCRIP*'s *DisplayLabel* to *Skill Description*, and its *DisplayWidth* to 40. Set *PersonSkillsSQLSKILLTYPE*'s *DisplayLabel* to *Type* and its *Alignment* to *taCenter*. Set *PersonSkillsSQLACQUIRED*'s *DisplayLabel* to *Acquired*, *Alignment* to *taCenter*, and *DisplayWidth* to 7. Change *PersonSkillsSQLCERTIFIED*'s *DisplayLabel* to *Certified*, and its *Alignment* to *taCenter*. Set the *Visible* property for *PersonSkillsSQLEMPID* and *PersonSkillsSQLSKILLID* to *False*. The grid should look slightly small, but much better.

Finding Skilled People

To answer the question "Which people have acquired a particular skill?" we follow a similar approach as before, but in reverse. Change the *SkilledPeopleSQL*'s *DatabaseName* to *Many2Many*, and its *DataSource* to *MasterDS*. Enter the SQL statement for the *SkilledPeopleSQL* query as shown in Figure 10. You'll notice

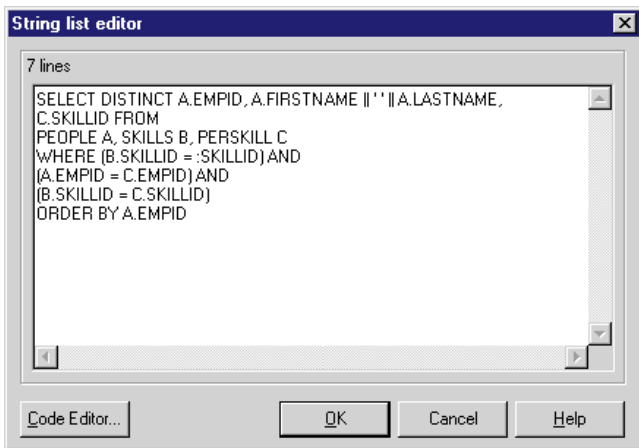


Figure 10: The *SkilledPeopleSQL* statement.

special characters in this query. Rather than select First Name and Last Name as separate columns, I've concatenated them. This is a useful feature when dealing with names, addresses, and similar data. Examine the online Help for Local SQL for other similar features.

SQL junkies will notice the *DISTINCT* keyword. Because a person can acquire more than one skill, nothing prevents a person from acquiring the same skill more than once. (This is the recency issue mentioned earlier.) We have, in fact, planned for this. The primary keys for the *PerSkill* table are *EmpID*, *SkillID*, and the date *Acquired*. Because more than one occurrence of the same skill is permitted, the same person can be selected in the SQL statement. The *DISTINCT* keyword prevents this.

Set the data type for the *SKILLID* parameter to *String*, and perform the following steps to test the query. Change the *DataSet* for *MasterDS* to *SkillsTB*, and change the *DataSet* property for *DetailDS* to *SkilledPeopleSQL*. Next, close the *PersonSkillsSQL* query.

Finally, open the *SkillsTB* table and the *SkilledPeopleSQL* query. It looks great except for those funky column labels. Double-click the *SkilledPeopleSQL* component and add all three columns. Set *SkilledPeopleSQLEMPID*'s *DisplayLabel* to *Employee ID*, *SkilledPeopleSQLFirstNameLastName*'s to *Employee's Full Name*, and *SkilledPeopleSQLSKILLID*'s *Visible* property to *False*. It should look much better now.

Adding People and Skills

We now need to enable editing for people and skills. This will require some minor changes to the *SkillSetsDM* methods that you previously added, and the addition of two event handlers for the grids in our Browser.

Before we embark on our code changes, go through the *SkillSetsDM* and close *SkilledPeopleSQL*, *SkillsTB* and *PeopleTB*. Set the *MasterDS*'s *DataSource* back to *PeopleTB*, and the *DetailDS*'s *DataSource* back to *PersonSkillsSQL*. We will be opening the tables and queries on an "as needed" basis.

In the *ByPeople* and *BySkills* methods, remove the comments surrounding the last two lines in each procedure. Next, in the

Browser form, add an *OnEnter* event handler for *MasterGrid*, as follows:

```
procedure TForm1.MasterGridEnter(Sender: TObject);
begin
  DBNav.DataSource := SkillSetsDM.MasterDS;
  ZoomBtn.Enabled := False;
end;
```

Add a similar *OnEnter* event handler for *DetailGrid*, as follows:

```
procedure TForm1.DetailGridEnter(Sender: TObject);
begin
  DBNav.DataSource := SkillSetsDM.DetailDS;
  ZoomBtn.Enabled := True;
end;
```

The *Zoom* button will be used to invoke our second form, to permit editing of new skill acquisitions. Because it's good to have a context for zooming, we'll permit it only in the lower grid. Now compile and run the form. Swap the view a couple of times. While viewing by people, the Browser will appear as in Figure 7. When viewing by skills, it appears as in Figure 8. Using the navigator, you can add, edit, and delete records in the top grid. Use the Browser to add the data depicted in Figures 7 and 8.

Editing Across the Join

The final step in our example is to permit editing in the joined table. You'll recall that *DetailGrid* has its *RowSelect* property set to *True*. This property doesn't permit editing. Editing three tables linked by a dual join can be a bit tricky. It can be done, but it requires the new *UpdateSQL* component. What makes it less than trivial in our example is that we must work around a bug in the BDE interface that doesn't like Null fields in link tables. You'll recognize the bug as spurious statements, triggered by the BDE interface during edits or deletions, that read, "Beginning of Table" or "End of Table." Fear not; there is a workaround.

Add a second form to the project, and remove the form from the project's auto-create list using the Project Viewer's Project Options dialog box. Lay out the form as shown in Figure 11. Set the properties for the form's controls, as shown in the .DFM file (see Listing Seven beginning on page 52). Save the form as *M2MEdit*.

Returning to the *SkillSetsDM*, set *Mny2MnySQL*'s *DatabaseName* to *Many2Many*, *DataSource* to *DetailDS*, and *RequestLive* property to *True*. Use the String list editor to enter the following SQL statement:

```
SELECT EMPID, SKILLID, ACQUIRED, CERTIFIED,
       CERTIFICATION, DETAILS
FROM PERSKILL
WHERE EMPID = :EMPID
AND SKILLID = :SKILLID
```

Set the data types for the *EMPID* and *SKILLID* parameters to *String*.

Mny2MnySQL's parameters are filled in automatically from *DetailDS*. Since both detail queries that *DetailDS* toggles

Figure 11: The design of the People Skills Definition form.

between contain both EMPID and SKILLID in their result set, either detail query can be active. Now, switch to code view to add some data-module code.

Fooling the BDE Interface

Earlier I mentioned a bug in the BDE interface regarding link-table editing. The bug surfaces only when the link table is empty or has double Null keys. We'll add two methods designed to circumvent the problem. The first, *CheckAutoAdd*, determines whether the table is empty, and if so, automatically places the table into insert mode. The only reason to zoom on an empty intersection is to add the first entry, so we save the user a step. We also prevent the error, and that's the clever bit.

The second procedure supports adding new records by supplying the side of the key that doesn't change during additions. When viewing by people, this will be the employee ID for the current person. When viewing by skills, it will be the skill ID for the current skill. These records remain fixed until the modal dialog box is closed.

After adding a public definition for *CheckAutoAdd*, add the *OnNewRecord* event handler for the *Mny2MnySQL* DataSet and the *CheckAutoAdd* method (again, see Listing Five).

Skillful Zooming

When adding entries to the PerSkill table, half the key will always be known and static. We need a visual cue to let users know which half of the key must be selected from a drop-down list. To this end, we'll add the *FixThePerson* and *FixTheSkill* methods to the Many2ManyDlg form. Each method is designed to dynamically alter the form, so as to properly inform the user.

In addition to adding standard modal dialog-box close functionality to the OK and Cancel buttons, we'll also need an activator for *CheckAutoAdd*, and a clean-up routine to close the query when we close the form. The form's *OnActivate* and *OnClose* events work well. We must also remember to add the SkillSet file to M2MEdit's uses clause. The source code for this is shown in Listing Eight on page 54.

The last step is to create the Many2Many dialog box, activate the query, fix up the dialog box for editing, make the edits, and refresh the lower grid after returning. Listing Eight contains the *OnClick* event handler that must be added to the Browser's *ZoomBtn* to accomplish these items. Add the M2MEdit file to the Browser's uses clause. (Purists may want to move the query activation inside the Many2Many dialog box.)

After making the code additions, run the form. Both the keyboard and mouse can be used to run the application. The **Alt P** key combination navigates to the upper grid, **Alt S** moves to the lower, **Alt W** swaps the View, and the **Alt Z** zooms. Using **Tab** and **Enter**, users can add and edit data or leave the form.

Dynamic Duo

This article has demonstrated various ways in which DataSources can dynamically operate with DataSets to achieve a practical way of implementing many-to-many relationships within Delphi client/server applications. Multiple masters can be served. Hopefully, you've seen an idea or two that can help you when you next encounter this pattern.

Helpful Resources

The need to properly model complex applications cannot be over-stressed. Although we've touched briefly on Entity-Relationship Modeling, full coverage of this subject is beyond the scope of this article. For a complete coverage of Entity-Relationship Modeling, see the book *CASE*Method Entity-Relationship Modeling* by Richard Barker [Addison-Wesley, 1990], or *The Data Modeling Handbook* by Michael C. Reingruber and William W. Gregory [John Wiley & Sons, Inc., 1994]. For an excellent treatment of design patterns, consult *Design Patterns for Object-Oriented Software Development* by Wolfgang Pree [Addison-Wesley, 1995].

The start-up delay in opening the initial tables for this program is an example of processing delays that often pervade database applications. You may want to consider changing the mouse cursor to keep your users patient.

Afterthoughts

We developers are seldom lucky enough to find an article that addresses precisely what we need when we need it. If you're working with an application that requires dynamic SQL or joins between tables, and you're using auto-increment primary keys, you're possibly experiencing some problems.

Delphi's database-field objects use OLE variants to store their data, and the auto-increment data type is not directly compatible with an integer data type. In such situations, simply

setting the *MasterSource* property to a *DataSource* will cause a variant-conversion exception. The workaround to this is to add code to instantiate your SQL query's parameters from the master *DataSet's* fields directly within the master *DataSet's OnDataChange* event handler.

Relegating the join table edits to a separate dialog box permits you great freedom in how you organize your application. Making the dialog box separate supports placing it in a DLL, and calling it from multiple locations within your application. It also provides a user-initiated action by which what I term "second-stage" queries can be activated. There is no point is tying up resources needlessly.

Although the example in this article mixed Table objects and Query objects, the entire project can be built exclusively using Query objects. This is important when considering upsizing your application. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\AUG\DI9708JC.

James Callan, formerly a consulting director with Oracle Corp., is currently president of Gordian Solutions, Inc., an information-technology consulting provider in Cary, NC. Jim's latest book on client/server excellence, *Collaborative Computing with Delphi 3* (Wordware, 1997), is currently available in bookstores. He can be reached at (919) 460-0555, or by e-mail at 102533.2247@compuserve.com.

Begin Listing Five — SkillSet.pas

```
unit SkillSet;

interface

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

type
  TSkillSetsDM = class(TDataModule)
  MasterDS: TDataSource;
  DetailDS: TDataSource;
  Mny2MnyDS: TDataSource;
  PeopleTB: TTable;
  SkillsTB: TTable;
  SkilledPeopleSQL: TQuery;
  PersonSkillsSQL: TQuery;
  Mny2MnySQL: TQuery;
  PeopleTBEmpID: TStringField;
  PeopleTBFirstName: TStringField;
  PeopleTBLastName: TStringField;
  SkillsTBSkillID: TStringField;
  SkillsTBDDescrip: TStringField;
  SkillsTBSkillType: TStringField;
  SkillsTBRenewable: TStringField;
  SkillsTBDuration: TSmallintField;
  PersonSkillsSQLDESCRIP: TStringField;
  PersonSkillsSQLSKILLTYPE: TStringField;
  PersonSkillsSQLACQUIRED: TDateField;
  PersonSkillsSQLCERTIFIED: TStringField;
  PersonSkillsSQLEMPID: TStringField;
  PersonSkillsSQLSKILLID: TStringField;
```

```
SkilledPeopleSQLEMPID: TStringField;
SkilledPeopleSQLFIRSTNAMELASTNAME: TStringField;
SkilledPeopleSQLSKILLID: TStringField;
procedure Mny2MnySQLNewRecord(DataSet: TDataSet);
private
  ViewingByPeople: Boolean;
  { Maintains Many-to-many viewing perspective. }
public
  procedure ByPeople; { To view by People. }
  procedure BySkills; { To view by Skills. }
  procedure CheckAutoAdd; { Join table BDE workaround. }
end;

var
  SkillSetsDM: TSkillSetsDM;

implementation

{$R *.DFM}

procedure TSkillSetsDM.ByPeople;
begin
  SkilledPeopleSQL.Close;
  ViewingByPeople := True; { Set view indicator. }
  MasterDS.DataSet := PeopleTB;
  with PeopleTB do begin
    if not Active then
      Open;
    FieldByName('EmpID').DisplayLabel := 'Employee ID';
    FieldByName('FirstName').DisplayLabel := 'First Name';
    FieldByName('FirstName').DisplayWidth := 30;
    FieldByName('LastName').DisplayLabel := 'Last Name';
    FieldByName('LastName').DisplayWidth := 30;
  end;
  DetailDS.DataSet := PersonSkillsSQL;
  PersonSkillsSQL.Open;
end;

procedure TSkillSetsDM.BySkills;
begin
  PersonSkillsSQL.Close;
  ViewingByPeople := False; { Reset view indicator. }
  MasterDS.DataSet := SkillsTB;
  if not SkillsTB.Active then
    SkillsTB.Open;
  DetailDS.DataSet := SkilledPeopleSQL;
  SkilledPeopleSQL.Open;
end;

procedure TSkillSetsDM.CheckAutoAdd;
begin
  with Mny2MnySQL do begin
    if not Active then
      Open;
    if BOF and EOF then
      begin
        Edit;
        Insert;
      end;
  end;
end;

procedure TSkillSetsDM.Mny2MnySQLNewRecord(
  DataSet: TDataSet);
begin
  with Mny2MnySQL do
    if ViewingByPeople then
      FieldByName('EMPID').AsString :=
        PeopleTB.FieldByName('EMPID').AsString
    else
      FieldByName('SKILLID').AsString :=
        SkillsTB.FieldByName('SKILLID').AsString;
end;

end.
```

End Listing Five

Begin Listing Six — Browser.pas

```

unit Browser;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, Grids, DBGrids, DBCtrls,
  StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    MasterGB: TGroupBox;
    DetailGB: TGroupBox;
    SwapBtn: TButton;
    ZoomBtn: TButton;
    OkBtn: TButton;
    CancelBtn: TButton;
    DBNav: TDBNavigator;
    MasterGrid: TDBGrid;
    DetailGrid: TDBGrid;
    procedure FormActivate(Sender: TObject);
    procedure SwapBtnClick(Sender: TObject);
    procedure OkBtnClick(Sender: TObject);
    procedure CancelBtnClick(Sender: TObject);
    procedure MasterGridEnter(Sender: TObject);
    procedure DetailGridEnter(Sender: TObject);
    procedure ZoomBtnClick(Sender: TObject);
  private
    ViewingByPeople: Boolean;
    procedure ViewByPeople;
    procedure ViewBySkills;
  public
    end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

uses SkillSet, M2MEdit;

procedure TForm1.ViewByPeople;
begin
  SkillSetsDM.ByPeople; { Set up data sources. }
  ViewingByPeople := True;
  MasterGrid.Width := 468;
  MasterGrid.Left := 33;
  MasterGB.Caption := '&People On The Fast Track';
  DetailGB.Caption := 'Skills &Acquired By This Person';
  DetailGrid.Width := 397;
end;

procedure TForm1.ViewBySkills;
begin
  SkillSetsDM.BySkills; { Set up data sources. }
  ViewingByPeople := False;
  MasterGrid.Width := 509;
  MasterGrid.Left := 16;
  MasterGB.Caption := '&Skills To Watch For';
  DetailGB.Caption :=
    'People Who Have &Acquired This Skill';
  DetailGrid.Width := 378;
end;

procedure TForm1.FormActivate(Sender: TObject);
begin
  ViewByPeople;
end;

procedure TForm1.SwapBtnClick(Sender: TObject);
begin
  if ViewingByPeople then
    ViewBySkills
  else

```

```

  ViewByPeople;
end;

procedure TForm1.OkBtnClick(Sender: TObject);
begin
  Close;
end;

procedure TForm1.CancelBtnClick(Sender: TObject);
begin
  Close;
end;

procedure TForm1.MasterGridEnter(Sender: TObject);
begin
  DBNav.DataSource := SkillSetsDM.MasterDS;
  ZoomBtn.Enabled := False;
end;

procedure TForm1.DetailGridEnter(Sender: TObject);
begin
  DBNav.DataSource := SkillSetsDM.DetailDS;
  ZoomBtn.Enabled := True;
end;

procedure TForm1.ZoomBtnClick(Sender: TObject);
begin
  Many2ManyDlg := TMany2ManyDlg.Create(Application);
  SkillSetsDM.Mny2MnySQL.Open;

  with Many2ManyDlg do begin
    if ViewingByPeople then { Show the dialog box. }
      FixThePerson
    else
      FixTheSkill;
    ShowModal;
    Free;
  end;

  if ViewingByPeople then { Refresh the view. }
    ViewByPeople
  else
    ViewBySkills;
end;

end.

```

End Listing Six**Begin Listing Seven — M2MEdit.dfm**

```

object Many2ManyDlg: TMany2ManyDlg
  Left = 290
  Top = 235
  BorderStyle = bsDialog
  Caption = 'People Skills Definition'
  ClientHeight = 330
  ClientWidth = 324
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OnActivate = FormActivate
  OnClose = FormClose
  PixelsPerInch = 96
  TextHeight = 13
  object Label1: TLabel
    Left = 8
    Top = 3
    Width = 63
    Height = 13
    Caption = '&Employee ID:'
    FocusControl = EmpIDLKUF1d
  end
  object Label2: TLabel
    Left = 128
    Top = 3
    Width = 53
    Height = 13
    Caption = 'First Name:'

```

```

end
object Label3: TLabel
  Left = 220
  Top = 3
  Width = 54
  Height = 13
  Caption = 'Last Name:'
end
object Label4: TLabel
  Left = 8
  Top = 58
  Width = 78
  Height = 13
  Caption = '&Skill Description:'
  FocusControl = SkillLLKUFld
end
object Label5: TLabel
  Left = 8
  Top = 113
  Width = 45
  Height = 13
  Caption = '&Acquired:'
  FocusControl = AcquiredFld
end
object Label6: TLabel
  Left = 191
  Top = 113
  Width = 73
  Height = 13
  Caption = 'Certification &By:'
  FocusControl = CertificationFld
end
object Label7: TLabel
  Left = 8
  Top = 165
  Width = 35
  Height = 13
  Caption = '&Details:'
  FocusControl = DetailsMEMO
end
object DBEdit1: TDBEdit
  Left = 128
  Top = 19
  Width = 89
  Height = 21
  TabStop = False
  Color = clBtnFace
  DataField = 'FirstName'
  DataSource = DataSource1
  MaxLength = 20
  TabOrder = 1
end
object EmpIDLKUFld: TDBLookupComboBox
  Left = 8
  Top = 19
  Width = 93
  Height = 21
  DataField = 'EMPID'
  DataSource = SkillSetsDM.Mny2MnyDS
  DropDownRows = 9
  DropDownWidth = 300
  KeyField = 'EmpID'
  ListField = 'EmpID;FirstName;LastName'
  ListSource = DataSource1
  TabOrder = 0
end
object DBEdit2: TDBEdit
  Left = 216
  Top = 19
  Width = 97
  Height = 21
  TabStop = False
  Color = clBtnFace
  DataField = 'LastName'
  DataSource = DataSource1
  MaxLength = 25
  TabOrder = 2
end
object SkillLLKUFld: TDBLookupComboBox
  Left = 8
  Top = 74

```

```

  Width = 305
  Height = 21
  DataField = 'SKILLID'
  DataSource = SkillSetsDM.Mny2MnyDS
  KeyField = 'SkillID'
  ListField = 'Descrip'
  ListSource = DataSource2
  TabOrder = 3
end
object AcquiredFld: TDBEdit
  Left = 8
  Top = 128
  Width = 65
  Height = 21
  DataField = 'ACQUIRED'
  DataSource = SkillSetsDM.Mny2MnyDS
  MaxLength = 0
  TabOrder = 4
end
object CertifiedCB: TDBCheckBox
  Left = 107
  Top = 132
  Width = 73
  Height = 17
  Caption = 'Certified'
  DataField = 'CERTIFIED'
  DataSource = SkillSetsDM.Mny2MnyDS
  TabOrder = 5
  ValueChecked = 'Y'
  ValueUnchecked = 'N'
end
object CertificationFld: TDBEdit
  Left = 190
  Top = 130
  Width = 121
  Height = 21
  DataField = 'CERTIFICATION'
  DataSource = SkillSetsDM.Mny2MnyDS
  MaxLength = 10
  TabOrder = 6
end
object DetailsMEMO: TDBMemo
  Left = 8
  Top = 180
  Width = 305
  Height = 65
  DataField = 'DETAILS'
  DataSource = SkillSetsDM.Mny2MnyDS
  TabOrder = 7
end
object OkBtn: TButton
  Left = 156
  Top = 301
  Width = 75
  Height = 25
  Caption = '&Ok'
  Default = True
  TabOrder = 9
  OnClick = OkBtnClick
end
object CancelBtn: TButton
  Left = 240
  Top = 301
  Width = 75
  Height = 25
  Caption = 'Cancel'
  TabOrder = 10
  OnClick = CancelBtnClick
end
object DBNavigator1: TDBNavigator
  Left = 64
  Top = 252
  Width = 200
  Height = 25
  DataSource = SkillSetsDM.Mny2MnyDS
  TabOrder = 8
  TabStop = True
end
object DataSource1: TDataSource
  AutoEdit = False

```

```

    DataSet = Table1
    Left = 96
    Top = 42
end
object Table1: TTable
    Active = True
    DatabaseName = 'Many2Many'
    ReadOnly = True
    TableName = 'PEOPLE'
    Left = 124
    Top = 42
end
object DataSource2: TDataSource
    AutoEdit = False
    DataSet = Table2
    Left = 256
    Top = 152
end
object Table2: TTable
    Active = True
    DatabaseName = 'Many2Many'
    ReadOnly = True
    TableName = 'SKILLS'
    Left = 284
    Top = 152
end
end
end

```

End Listing Seven

Begin Listing Eight — M2MEdit.pas

```

unit M2MEdit;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics,
    Controls, Forms, Dialogs, Grids, DBGrids, DB, DBTables,
    StdCtrls, DBCtrls, ExtCtrls, Mask;
type
    TMany2ManyDlg = class(TForm)
        DBEdit1: TDBEdit;
        EmpIDLKUFld: TDBLookupComboBox;
        DataSource1: TDataSource;
        Table1: TTable;
        DBEdit2: TDBEdit;
        SkillLLKUFld: TDBLookupComboBox;
        DataSource2: TDataSource;
        Table2: TTable;
        AcquiredFld: TDBEdit;
        CertifiedCB: TDBCheckBox;
        CertificationFld: TDBEdit;
        DetailsMEMO: TDBMemo;
        Label1: TLabel;
        Label2: TLabel;
        Label3: TLabel;
        Label4: TLabel;
        Label5: TLabel;
        Label6: TLabel;
        Label7: TLabel;
        OkBtn: TButton;
        CancelBtn: TButton;
        DBNavigator1: TDBNavigator;
        procedure OkBtnClick(Sender: TObject);
        procedure CancelBtnClick(Sender: TObject);
        procedure FormActivate(Sender: TObject);
        procedure FormClose(Sender: TObject);
        var Action: TCloseAction;
    private
    public
        procedure FixThePerson;
        procedure FixTheSkill;
    end;

var
    Many2ManyDlg: TMany2ManyDlg;

```

implementation

```

{$R *.DFM}

uses SkillSet;

procedure TMany2ManyDlg.FixThePerson;
begin
    EmpIDLKUFld.Enabled := False;
    SkillLLKUFld.Enabled := True;
    EmpIDLKUFld.Color := clBtnFace;
    SkillLLKUFld.Color := clWindow;
end;

procedure TMany2ManyDlg.FixTheSkill;
begin
    SkillLLKUFld.Enabled := False;
    EmpIDLKUFld.Enabled := True;
    EmpIDLKUFld.Color := clWindow;
    SkillLLKUFld.Color := clBtnFace;
end;

procedure TMany2ManyDlg.OkBtnClick(Sender: TObject);
begin
    ModalResult := mrOK;
end;

procedure TMany2ManyDlg.CancelBtnClick(Sender: TObject);
begin
    ModalResult := mrCancel;
end;

procedure TMany2ManyDlg.FormActivate(Sender: TObject);
begin
    SkillSetsDM.CheckAutoAdd;
end;

procedure TMany2ManyDlg.FormClose(Sender: TObject;
    var Action: TCloseAction);
begin
    SkillSetsDM.Mny2MnySQL.Close;
end;

end.

End Listing Eight

```





NEW & USED

By *Alan C. Moore, Ph.D.*

AppVision's GenerationXpert

Getting in Touch with Expert Automation

Delphi's environment is customizable and extendible; we can change its look and behavior in many ways. From the Code Editor to the tool bar, we can install individual new components or entire libraries. Among the more important tools in Delphi's arsenal are its experts (or Wizards as they're called in Delphi 3), which free us from hours of code-writing drudgery.

While we're all familiar with their capabilities, how many of us have considered writing experts? While Borland provides a *Component Writer's Guide* and online Help, no such detailed information exists to help us write experts. Information *is* available in a few sample programs (in Delphi's \demos\experts directory), and in the expert interface itself, DsgnIntf.pas (in the doc director.). Also, books such as Ray Lischner's *Secrets of Delphi 2* [Waite Group Press, 1996] and *Hidden Paths of Delphi 3* [Informant Press, 1997] have begun to fill the gap. But for the programmer in need of a quicker solution, this still may not suffice.

Enter GenerationXpert. With the introduction of AppVision's GenerationXpert, all of this is about to change. This very special Delphi expert enables us to create experts of our own. It's available in 16- and 32-bit versions, and in two distinct flavors: a DLL that runs in the Delphi environment (like most experts), and a stand-alone .EXE. Their interfaces are identical: Each

is a multi-page notebook that allows you to define all the basic elements of your expert.

A Guided Tour

As experts go, GenerationXpert is fairly involved, consisting of nine or ten dialog pages, depending on the internal choices you make. On the first page, you choose whether to create a new expert, or modify one you created earlier. On the second page, you choose the expert's type and style. On the third page, you provide a filename, class name, expert name, and description for your expert.

The appearance of the next page depends on the type of expert you choose. For example, if you choose a standard expert, you're asked to provide the text that Delphi will include under the **Help** menu. On the other hand, if you choose a form or a project expert, then you're asked for a bitmap to represent your new expert in the Object Repository.

Then, regardless of the type, you can choose a modal or a modeless dialog box. And in the

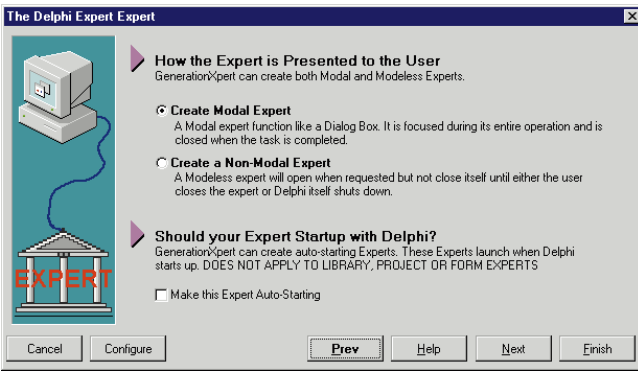


Figure 1: GenerationXpert lets you choose to have a standard expert load along with Delphi.

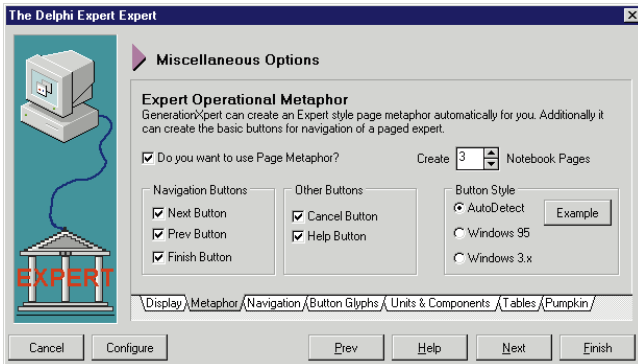


Figure 2: The Metaphor page lets you choose navigation buttons, and specify the number of pages.

case of a standard expert, you can choose to have it loaded along with Delphi (see [Figure 1](#)). Then you must provide names for the expert's entry function (and exit function, if modeless), form, and unit.

On the next page, as mentioned, you can choose to have GenerationXpert create a .DLL file (the usual type of expert), and optionally, a stand-alone .EXE file. On the same page, you can choose to install the expert in Delphi by having the program make an appropriate entry in Delphi.ini.

Next is Miscellaneous Options, which contains seven tabbed pages:

- The Display page allows you to set your expert's border style, border icons, form caption, and alignment.
- The Metaphor page (see [Figure 2](#)) lets you set the number of pages your expert will have, and how they'll be navigated.
- The Navigation page provides additional navigation options.
- The Button Glyphs page (see [Figure 3](#)) allows you to select the bitmaps for the three navigational buttons (**Next**, **Prev**, and **Finish**).
- The Units & Components page (see [Figure 4](#)) provides some delightful and unexpected options. You can add one or more commonly needed units (PRINTERS, MMSYSTEM, INIFILES, and FILECTRL), any of four common dialog boxes (OpenFile, SaveFile, Print, and Printer Setup), and the *TBatchMove* and/or *TReport* components.

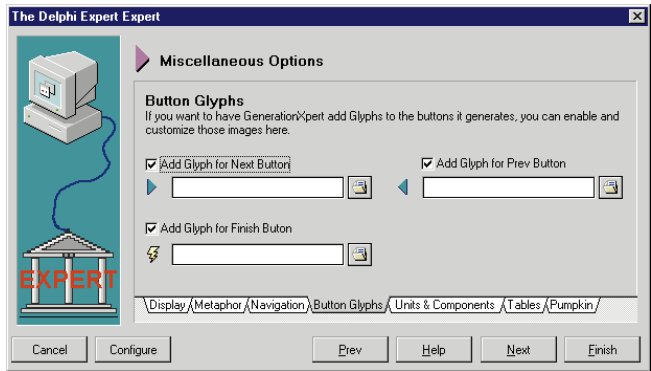


Figure 3: The Button Glyphs page allows you to assign bitmaps to your expert's buttons.

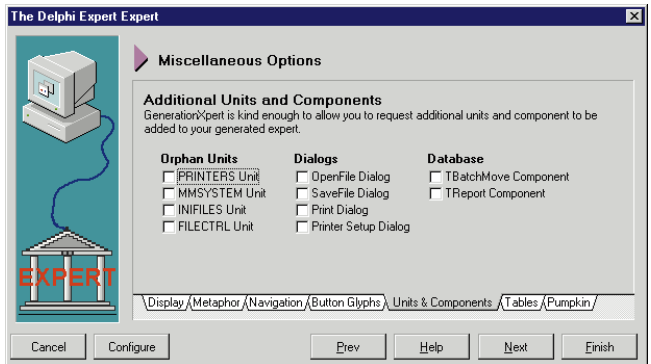


Figure 4: The Units & Components page reveals some unexpected surprises.

- The Tables page allows you to choose among the database engines on your system, create a list of tables, then edit them.
- The final subpage allows you to create connections to one of AppVision's other products, a Delphi project manager called Pumpkin. If you click **Finish** on any other page, you'll end up here. Just two things are left to do: Save the expert definition (in a .GXD file), and generate the code for the expert. Both these tasks are accomplished on this page.

Strengths and Weaknesses

A good deal of thought went into GenerationXpert's structure and internal logic. You can't move on to the next page until you've entered data in the crucial fields. The program keeps track of your choices, and presents you with appropriate follow-up pages; for example, only if you choose a project or a form expert will you be asked to create/choose a bitmap representation. Why? Because these, unlike a standard expert, are available within the Project and Form galleries, respectively.

The product does what it promises, and this is certainly a major strength. Another obvious strength is that GenerationXpert is the first comprehensive expert-generating tool for Delphi. All the weaknesses I'll mention are ones you would expect in a first-generation tool that has yet to be significantly modified.

The most serious shortcoming is the lack of any printed manual or tutorial. Because expert-writing is new for most developers, this is a real problem. Another problem is that you

can't import previously-designed form units into your expert. In other words, you'll have to do all the form-design work after you've generated the expert skeleton. (According to the application's author, form addition is among the top priorities for future versions.)

Sometimes a particular feature suggests both strengths and weaknesses. Unfortunately, it offers no way to add additional components or units beyond the pre-defined choices. Nevertheless, GenerationXpert does what any good expert should: It lets you make the decisions, then does the grunt work. Imagine coding all the decisions I described previously!

Who Needs It?

One likely market for GenerationXpert would be programmers who do a great deal of custom work, and find themselves repeatedly starting with the same form template and applying similar, but slightly different, design steps with each new application. The process could be automated by writing a form expert that requested the variable information, then generated a custom form.

Another possibility is for those who need a small, specific tool (an expert) to generate a series of related components. Of course, something like this could easily be done with one of the component-creation tools now available. However, if we know specifically what changes to make to a series of base components, and if those changes remain consistent from one component to the next, we may actually do better to create a simple expert, to really automate the process.

I'm sure you can think of many more uses. If you want to learn more about Delphi experts or write some of your own, you should definitely consider this product. **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 on the Internet at acmdoc@aol.com.

INFORMANT
FACT FILE

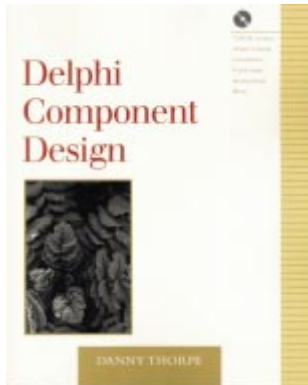
GenerationXpert is a Delphi expert that generates the skeletal code for standard, form, and project experts. It can generate modal or modeless experts that integrate into Delphi as DLLs, or as stand-alone .EXE files. While it provides the means to create a multi-page dialog form, it can't import existing forms. It produces a definition file (.GXD) that can be modified later, and a unit file (.PAS) that can be compiled. GenerationXpert is definitely worth the attention of developers who need to create experts (Wizards).

AppVision Software
171 Belvedere Ave.
San Carlos, CA 94070
Phone: (415) 631-8913
Fax: (415) 631-8914
E-Mail: sales@appvision.com
Web Site: <http://www-appvision.com>
Price: US\$89 (32-bit upgrade is free.)





Delphi Component Design



Even before I saw the first mention of *Delphi Component Design* by Danny Thorpe — a glowing endorsement by Jerry Coffey, Editor-in-Chief, in the February 1997 *DI* — there was a buzz of excitement about it on the COBB Delphi discussion group which I help moderate. As a long-time member of Borland's Research and Development team (Pascal and now Delphi) who has contributed his technical expertise to a number of Delphi publications, Danny Thorpe is widely known among Delphi developers. So there was much excitement and anticipation about his new book. Such excitement among advanced programmers seemed remarkable; developers in this category are highly selective, and tend not to buy many Delphi books.

Any book about writing Delphi components is going to include a significant number of new custom components demonstrating various techniques of component design, right? Wrong! While *Component Design* contains excellent code examples and an interesting utility or two, the goal is not to fill your

Component palette with a plethora of new custom components. Rather, it is to help you acquire the knowledge needed to create complex components and to work subtle magic in the Delphi environment. So don't expect to find a flashy new progress bar, a floating tool box, or an enhanced panel to house compound components. What you *will* find includes a detailed discussion of Delphi's component streaming process, its message system, the proper (and improper) use of excep-

Building Internet Applications with Delphi 2

Building Internet Applications with Delphi 2 by Davis Chapman, et al. is a text that should find its way onto the shelves of any programmer interested in creating software with Internet connectivity, regardless of language. From the fundamentals of Internet data transport to the intricacies of the Netscape API, this book packs in nearly every detail required by a programmer delving into development for the world-wide network.

Using Delphi as the demonstration tool, *Building* emphasizes the detailed presentation of the structures and protocols of the Internet. To lay a sound foundation for the reader, Chapman provides a good summary of the items comprising a complete Internet tool suite. The opening chapters introduce FTP, e-mail, Usenet News, and other World Wide Web tools; each is given a more thorough treatment in later chapters. A well written chapter on

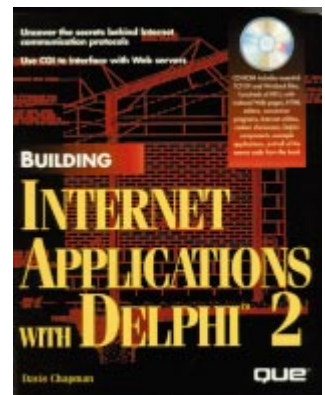
tions, and how to take advantage of Run-Time Type Information. And that's just skimming the surface.

Component Design begins with a concise but insightful history of the development of Delphi, tracing its roots to Borland's Turbo Pascal. After describing the two types of Delphi programmers — the application developer and the component creator — Thorpe provides an excellent overview of the component design process, including the various privacy levels, the basic classes

on which components and their support classes are built, and the extensibility/functionality dichotomy.

Many questions face the component architect: When is it better to declare a method dynamic instead of virtual? What is metadata and how can it be used to control the behavior of components? When and how should we use the *TReader* and/or *TWriter* methods? What are the best ways to optimize a compo-

"Delphi Component Design"
continued on page 59



Internet connectivity educates the reader on the protocols and addressing standards that are the basis for the existence of the Internet. The TCP/IP, IP addressing, and Domain Name Service segments are useful to those who need a broad perspective on the topics. The presentation is well balanced; each segment contains a brief introductory paragraph, followed by a more technical discussion.

Critical to the success of any Windows Internet development endeavor is a thorough understanding of the Windows Sockets (WinSock) API. The presentation in this book is not exhaustive, but it provides the foundation needed to understand the remainder of the projects in the book. Each of the steps required for an Internet connection through a Windows socket is explained using example calls written in Object Pascal. Readers may find it helpful to supplement

the book by viewing the WinSock.PAS header file (to see where the code fits into the overall picture). The WinSock chapter concludes by providing the snippets to encapsulate the WinSock object with a wrapper.

The project in the WinSock chapter provides a good example of what to expect from the rest of the book's projects. Readers will quickly learn they'll need to load the code examples from the accompanying CD to build the projects. (Heed the instructions

"Building Internet Applications with Delphi 2"
continued on page 59

Delphi Component Design (cont.)

nent? (And when is it a waste of time to code in assembler?) These and many other important questions are answered in the middle section of *Component Design*, which is devoted to the implementation details of writing a component. Toward the beginning of this section, Thorpe discusses the various helper classes — *TList*, *TCollection*, *TStrings*, etc. — that perform many of the behind-the-scenes tasks of components. What seems to be missing is a complete example of using such classes in building a new component. While he provides some useful tips for building new components with *TCollection* and its supporting class *TCollectionItem*, Thorpe

missed a great opportunity by not providing a specific example of using this helper class in building a new component.

Thorpe provides an interesting discussion of advanced graphics techniques in a chapter on Visual Component Library subsystems. He makes a strong case for using the Fractal Image File format instead of the better-known JPEG, GIF, or TIFF formats. He introduces a *TFIFImage* graphics class to provide support for this format, and includes a sample application and a related utility on the accompanying CD-ROM.

Delphi provides many tools to help us design and implement custom components,

including the new Open Tools Application Programming Interface in Delphi 2, Property Editors, and Component Editors. These tools are discussed in-depth in the concluding chapters, along with a brief look at Delphi Experts. There is also a chapter on optimization techniques that includes an excellent discussion of memory management. Here Thorpe's masterful understanding of Delphi's compiler really shines. This chapter, as well as others, will be of interest to any serious Delphi developer.

Delphi Component Design is definitely not an appropriate book for new Delphi developers. However, if you have experience writing compo-

nents and want to explore more advanced aspects, this book will interest you. And despite what its title might infer, *Delphi Component Design* has much to offer any advanced Delphi programmer, whether component creator or application developer.

— Alan C. Moore, Ph.D.

Delphi Component Design by Danny Thorpe, Addison-Wesley Developers Press, One Jacob Way, Reading, MA 01867-3999, (800) 447-2226, <http://www.aw.com/devpress>.

ISBN: 0-201-46136-6
Price: US\$36.95
(348 pages, CD-ROM)

Building Internet Applications with Delphi 2 (cont.)

regarding the read-only attributes while installing the code, or you'll waste a lot of time.) All code is provided in 16- and 32-bit versions, for either Delphi 1 or 2, and compiles easily. Although the example in Chapter 6 calls a DLL, the remaining examples in the book use Delphi exclusively. Preferring to work from complete code listings, I found the Pascal snippets difficult to place in context; it was helpful to load the project into Delphi.

The core of *Building* examines the development of individual Internet tools. Beginning with the development of an FTP Client and Server, the detailed entries also include SMTP and POP Mail Clients, an Internet News Client/Reader, and an associated UUEncoder/Decoder. The presentation is well balanced. The communications transaction being discussed is explained point by point. The Delphi procedure required to implement this data exchange

or process accompanies the text.

The World Wide Web and its protocols and tools are examined in the third section of *Building*. The chapters that examine CGI and the Netscape API present exciting possibilities for Delphi programmers who may not have considered the tool's applicability for developing these applications.

The final section of *Building* is devoted to useful appendices. Commands and response codes — the currency of the Internet — are detailed for each tool in the Internet suite. Message formats are also provided for the e-mail and HTTP datagrams. A chapter is set aside to discuss RFC (Request for Comment) Standards Documents. The CD contains the full text of many of these documents, and also includes FYI and FAQ documents.

Filling out the appendix section is a chapter on C/C++

conversion to Object Pascal that provides a chart-based reference to the process. This subject could fill more than the 10 pages allotted, and should be considered only an indicator of the possibilities.

As mentioned, the accompanying CD is useful beyond the project code it provides. Shareware and freeware tools for accessing the Internet and World Wide Web are also included. A nice addition is the Web pages supporting the shareware (listed in a separate directory). This saves time by bypassing the network in favor of the stored pages.

Building is valuable on two levels: It's a solid reference and teaching tool for programmers in any language who find themselves in need of technical information about the Internet, its protocols, and standards; and for Delphi programmers, it provides an excellent selection of learning opportunities for their selected tool in devel-

oping Internet applications. The material is exclusively aimed at advanced programmers; nothing is provided about the basics of Object Pascal and Delphi. *Building Internet Applications with Delphi 2* is for a specialized group of programmers, but, with the increasing importance of Internet connectivity in the commercial software market, this audience should see a continued expansion. Having this information in your toolbox may make the difference when bidding on your next project.

— Warren Rachele

Building Internet Applications with Delphi 2 by Davis Chapman, et al. QUE Corp., 201 W. 103 Street, Indianapolis, IN 46290, (800) 858-7674.

ISBN: 0-7897-0732-2
Price: US\$49.99
(624 pages, CD-ROM)



The All-Wagner Team

The component-based architecture of Delphi and other development tools has radically altered the way developers build applications. While the business logic specific to an application must still be created internally, savvy developers are leveraging the vast array of commercial, shareware, and freeware third-party VCL components.

If you've looked on the Web, you've undoubtedly noticed that locating shareware and freeware components isn't difficult. Finding *quality* components within this pool, however, may not always be such a simple task. While you can feel assured using shrink-wrapped packages like InfoPower or Raize Components, you must sort through a slew of shareware and freeware components to find useful and reliable ones. Nonetheless, quality components are out there, and I'm highlighting them this month.

In trying to select specific components to cover, I limited my attention to one category that deserved special attention: freeware components that include full source code. Not only is their price right for developers, but their source code can be a great learning tool. Besides, the authors deserve recognition for spending time and energy to provide free, fully functional components and source code for the developer community.

Just as John Madden picks only football players he's seen play for his All-Madden Team, I'm selecting only components I've worked with, i.e. it's likely there are many worthy of being recommended that I haven't used. Disclaimers aside, here are the winners:

Best Look Award: *TExplorerButton*. The user-interface enhancements by Microsoft first seen in Internet Explorer (IE) 3.0 produced a flurry of activity by Delphi developers to emulate this look and feel. Fabrice Deville's *TExplorerButton* stands out from the others I tried in this category. While Delphi 3 allows you to

work with Explorer-style buttons, *TExplorerButton*'s automatic handling of color shading makes it superior to what is provided by Borland within the package. <http://www.tornado.be/~fdev/components.html>

Instant Productivity Award: *TFileDrag*. Adding file drag-and-drop capabilities from Windows Explorer into your application is a breeze with Erik C. Nielsen's *TFileDrag* component. You can insert this VCL component into your form and have it supply a list of files being dropped into your application. Then you can do what you like with them. <http://SunSITE.icm.edu.pl/delphi/ftp/d20free/fdrag10.zip>

Workman Award: *xProcs*. While not as glamorous as some of the VCL components we're talking about, Fabula Software's *xProcs* is a freeware collection of 150 general-purpose functions that are useful in almost any application. <http://ourworld.compuserve.com/homepages/stefc/xprocs.zip>

Plug-and-Play Award: *TJustOne*. Preventing multiple instances of an application involves some knowledge of the Windows API. Steve Keyser's *TJustOne* component allows you to add this capability to your application without writing a single line of code. <http://carbohyd.siobc.ras.ru/torry/vcl/system/justone.zip>

Speed 'Em Up Award: *THETreeView*. The TreeView is one of the most powerful organizing controls for the Windows user interfaces. However, Delphi's *TTreeView* is slow in performing certain

routines. Haakon Eines' *THETreeView* is an optimized version of the standard Delphi component, and can tremendously expedite loading and saving. <http://SunSITE.icm.edu.pl/delphi/ftp/d20free/hetrvview.zip>

Office 97 Award: *TOfficeLabel*. Alexander Meeder's *TOfficeLabel* (and *TOfficeButton*) emulates the look of the labels and buttons of the Personal Assistant in Microsoft Office 97. These controls look particularly attractive on wallpapered forms. <http://carbohyd.siobc.ras.ru/torry/vcl/packs/ocontrlr.zip>

Colorizer Award: *TColorButton*. Emulating the capabilities of IE, Steven Costa Martins' *TColorButton* makes picking a color easy and intuitive. This button component allows a user to push a button to display the common control's Color dialog box. After a color is selected, the button face reflects the chosen color. <http://www.intermid.com/delphi/download/colorbtn.zip>

Now that you've seen my favorites, I'd like to hear from you. What other quality freeware or shareware components have been useful to you? ▲

— Richard Wagner

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