September 1996, Volume 2, Number 9

Cover Art By: *Tom McKeith*

Delphi Patterns

Creating a Generic Notification Mechanism

ON THE COVER

8 Delphi Patterns — James Maycott
Mr Maycott illustrates how to implement a subscribe/notify scheme using a custom component and Delphi events to allow objects to send messages to each other. It's an introduction to message handling and the increasingly popular concept of patterns.

FEATURES

17 Informant Spotlight — Dana Scott Kaufman
We've already learned that Delphi 2 makes creating multithreaded applications a relatively straightforward task. But what about multithreaded queries? There are a few wrinkles, but Mr Kaufman shares the tricks of the trade and provides a working example.

21 DBNavigator — Cary Jensen, Ph.D.
Is there no end to the Delphi 2 bounty? This month, Dr Jensen offers a friendly introduction to the new Object Repository. Say "Good-bye" to the Gallery and "Hello" to Visual Form Inheritance; now it's even easier to share/reuse code.

26 Visual Programming — Tony Yeung
Experts are popular with users and developers alike, so it's good to hear that most experts can be easily implemented with standard Delphi components. But what about "big" experts — the kind that exhaust Windows resources? Mr Yeung shares a custom component and some advice.

32 Delphi Reports — Mark Ostroff
ReportSmith is a robust reporting tool that allows you to build powerful, complex reports featuring graphs, crosstabs, drill-downs, live data pivoting, etc. Mr Ostroff completes his two-part series by discussing a feature new to ReportSmith 32 — the Delphi Connection.

37 Delphi Kiosk — Chip Overstreet
Enterprise applications are moving from two-tier to three-tier architectures for two important reasons: reuse of business rules and easier, cost-effective maintenance of those rules. Mr Overstreet presents some new tools that can help you leverage your current Delphi Client/Server application investment.

41 At Your Fingertips — David Rippy
Our favorite tipster returns in strong fashion. This month he favors us with sage advice on using a single DBNavigator for multiple DBGrids on a form, employing Page Up and PageDown to navigate records in a table, as well as creating marquee-style text on a form.

REVIEWS

43 ReportPrinter Pro 2.0 — Product Review by Bill Todd
46 UDFlib — Product Review by Bill Todd

48 *Delphi 2 Multimedia Adventure Set*
Book Review by Alan C. Moore, Ph.D.
48 *Delphi In Depth* — Book Review by Alan C. Moore, Ph.D.
49 *Building Delphi 2 Database Applications*
Book Review by Larry Clark

DEPARTMENTS

2 Editorial by Zack Urlocker
4 Delphi Tools
6 Newsline
51 File | New by Richard Wagner

Marathon to Delphi

After we had finished Delphi 1, Gary Whizin, the Director of Delphi Development, used to joke that it was like running a 10K race, then discovering you had entered a marathon. It was certainly no sprint to the finish, but rather, a lot of hard slogging. We set our goal and then worked to achieve the milestones that would lead us to the goal. We've found it to be a pretty good way to do things.

Delphi 1 took almost exactly two years of development. In the first year we built on our compiler technology, created the development environment, and built the foundations of the Visual Component Library. In the second year we added more components, more tools, and database support. Although the goal had always been to provide database capabilities in Delphi, the scope and significance of what it means to support client/server development, well, that was the "Heartbreak Hill" of our marathon. We knew our goal. So we paced ourselves, slogged through the hard parts, and yes, we finished on time. I've shipped quite a few products in my career, but Delphi 1 stands alone. It's the product I'm most proud of.

In the case of Delphi 2, we knew what we had signed up for. We set out to create a full 32-bit version with support for Windows 95 and Windows NT. Luckily, we had a head start. We'd begun work on the 32-bit Object Pascal compiler back when Delphi 1 first started. Our goal was to ship Delphi 2 within 1 year.

I try to work out regularly, even during the most hectic periods of the development schedule. I do this to maintain balance in my life and to reduce stress. I was a weekend runner, taking part in the occasional 5K or 10K race — but I always wondered about a marathon. In the fall, after signing off on beta releases of Delphi 2, Molly, a friend of mine at work, asked me if I wanted to run a half marathon. To be honest, she caught me off guard. I was slightly dumbstruck, but also curious about running such a distance. I agreed — after all, the race was still four weeks away. At the time, I was running once or sometimes twice a week, usually somewhere between five and eight miles. What the heck; I figured I'd be able to add 15 minutes to my time for a couple of weeks and increase my distance to 12 miles before the race. How hard could it be? Actually, running a half marathon isn't that hard. At least, not compared to walking down stairs the following day. Or the day after that. It hurt like hell, but I did it. It was the most significant milestone in a very long and personal project.

Who me? Run a m-m-m-arathon? Well, I didn't know about that. But I knew I could add another 15 minutes to my "long run" distance in two or three weeks. I knew my goal; I just didn't think about it too often. Instead, I focused on my short term milestone: my next long run.

Delphi 2 development continued steadily during the fall. I had originally hoped to ship it within 90 days of the Windows 95 release date, but I was a little optimistic. Instead, we added some additional important client/server features, including Data Modules, SQL Explorer, and cached updates. We were into a regular groove with Delphi and things were proceeding on track to meet our ship date.

Meanwhile, my running was getting serious. I made sure I ran my long run every two or three weeks. It was my personal "milestone," much like the regular builds we did of the software. I remember trying to cram a two-hour run in before going from California to Toronto for Christmas. It was the longest lunch break I ever took. I figured it would be a heck of a lot easier to run in the warmth of Santa Cruz than in the misery of a Canadian cold front. In the meantime, I got good at finding pay phones along my route so I could check on how things were going.

Finally, February came around and we were getting ready to sign off the final version of Delphi 2 and send it to manufacturing. It had been hard slogging for a few days to beat the first anniversary of Delphi 1's release. It was a significant emotional deadline for folks on the team, though certainly not a rigid deadline for the company. Wednesday was our "internal" team date by which we wanted to sign off. I didn't really think we'd make that day — and we didn't. Thursday was a flurry of bug fixes, testing, new builds, more minor bugs. By Friday, I was beginning to become concerned. We were still finding a few bugs, and every-

one on the team was getting tired of evening and weekend work. Our sign-off date moved to the weekend. We could still get to manufacturing on Monday morning, well in advance of our anniversary.

Saturday was pretty much a blur. I brought in some food, as did others, and everyone was testing like mad. We found a few last minute glitches; enough to keep fixing the "stop ship" bugs as well as some minor inconveniences. We may not find or fix every bug, but if it's something that can't be easily worked around, we do our best. In a fit of weirdness, someone built a "sign-off shrine" to Delphi in one of the empty offices. We sacrificed old sign-off candidate CDs, leftover Chinese food, Tums, aspirin, cartons, you name it, all of it lighted by the eerie glow of a lava lamp from tech support. Several folks broke into a case of beer that a customer had sent us.

On Sunday, things were looking good. We had a few of the integration folks stay around "on call" so we could build all three of the Delphi products: Delphi Client/Server Suite, Delphi Developer, and Delphi Desktop. Unfortunately, we still found a couple of unlikely bugs in the demonstration programs, which meant rebuilding all three products, an automatic process that takes a couple of hours. We'd stopped finding bugs, so folks had to hang around the office waiting for their official "sign-off." Around noon, things were looking good, so I went for a run. A long run. Just enough time that all three builds would be finished and sign-off would be proceeding with final testing when I got back.

It was a stark, sunny day. The nicest I'd seen in weeks. I chugged past the old Borland buildings on Greenhills road, where my office had been when I started about six years ago. The hills, once intimidating, were just a warm-up to me now. I continued on for about five miles, past De Laveaga Park, where we'd held a few Borland picnics over the years. I continued on for another few miles, spotted a pay phone, and checked my voice mail. No messages. So I kept on running. At just over two hours, I was back into the civilization, if you can call it that, of Scotts Valley. I bought a Gatorade and juice. I called in again; things were proceeding on track.

Now the hard part. I started on Bean Creek. Bean Creek is about an eight-mile run with a very steep hill that extends for nearly a mile. Even though I'd already clocked about 15 miles, I knew this next part would be hard. And it was. But it was also familiar. I'd run Bean Creek dozens of times during training. I knew the curves, the hills. I knew where to look for deer. I knew where to speed up and where to slow down. And if my body forgot for a few miles that I had already run 15 miles, well I wasn't going to let on. Not for a minute. Coming down the final hill I was getting a little buggy. I was talking to myself. Talking myself into it. I had just under two miles to go to get to the next milestone. Just 15 minutes. Step by step. Two miles. Not even.

After four months of training, I'd broken three hours and more than 20 miles.

When I got back to the office, we signed off Delphi 2 in just under 365 days.

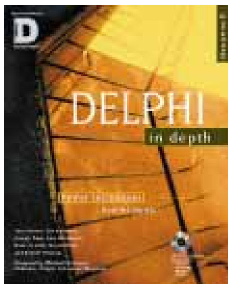
I continued my training for another two months, logging three more "long runs" before tapering down for the race. On April 28th, I ran the Big Sur Marathon in under four hours. It's the race I'm most proud of.

— *Zack Urlocker*

Zack Urlocker is Director of Product Management at Borland International. He's currently working on the next Delphi "marathon." He can be reached on CompuServe at 76217,1053.

New Delphi Book

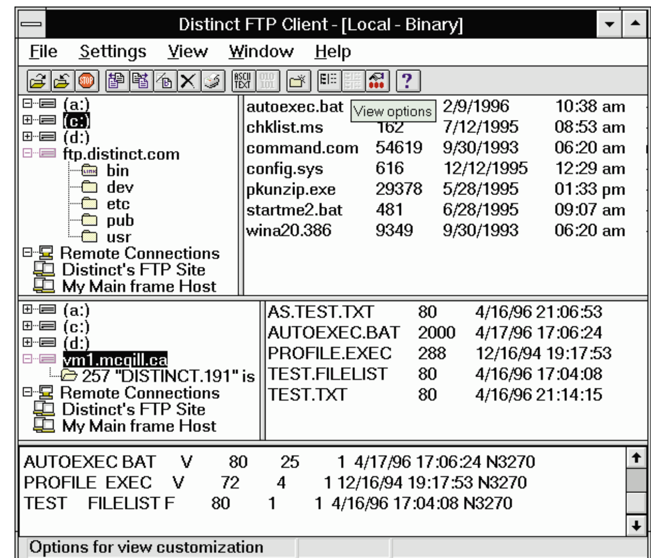
Delphi In Depth
Cary Jensen, Ph.D., et al.
Osborne/McGraw-Hill



ISBN: 0-07-882211-4
Price: US$42.95
(812 pages, CD-ROM)
Phone: (800) 722-4726

Distinct Adds Five New OLE Custom Controls to its Visual Internet Toolkit

Distinct Corp. of Saratoga, CA has shipped the ***Visual Internet Toolkit version 1.2***. Additions to version 1.2 include five OLE custom controls (OCXes): TCP Server, VT220 (terminal), TFTP, Finger, and WhoIs. These new controls enable Delphi 2, Visual C++, FoxPro, PowerBuilder, Access, and Visual Basic programmers to develop 32-bit TCP/IP, Internet, and intranet applications for Windows 95 and Windows NT environments.



The upgraded Visual Internet Toolkit provides developers with OCXes, as well as methods for application development. With the new OCXes, users can integrate a TCP server within their application for multiple client connectivity, while adding VT220 terminal emulation capabilities to their specific environment. The VT220 OCX offers the developer a VT220 emulation with keyboard remapping, as well as color and font support to integrate within their application.

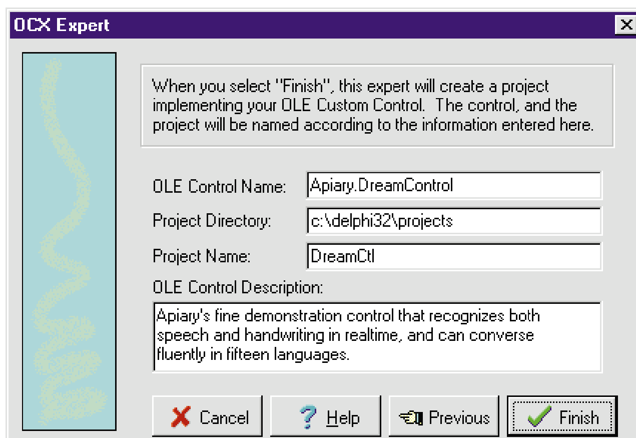
The TFTP OCX is designed for developers needing to integrate TFTP functionality into their application.

The Finger OCX will allow developers to design an application that includes a feature to retrieve detailed information about users on the network. The WhoIs OCX will enable developers to retrieve information about network sites. Visual Internet Toolkit also includes sample programs, and code is provided.

Price: US$295. Distinct also offers a subscription program to developers that includes free upgrades and technical support for one year. The price with subscription is US$545.
Contact: Distinct Corp., 12900 Saratoga Ave., Saratoga, CA 95070
Phone: (408) 366-8933
Fax: (408) 366-1153
E-Mail: Internet: mkgt@distinct.com
Web Site: http://www.distinct.com

Apiary's OCX Expert Converts Delphi 2 VCLs into OLE Controls

Apiary, Inc. of Little Rock, AR announced support for the Delphi 2 product line with its ***OCX Expert***, which provides a way to create 32-bit OLE Controls (OCXes) using Delphi 2.



Prior to the OCX Expert release, generating OLE Controls was accomplished using C++ and Microsoft Foundation Classes (MFCs). Now developers can use the OCX Expert to convert most Delphi 2 VCLs into OCXes with little or no changes. The Expert guides the programmer through a series of question, or steps, to determine the properties and methods that should be exposed in the OCX. After completion, the OCX is portable and can be used in most environments supporting OLE containers such as Paradox 7, Microsoft Word for Windows, Borland C++, Visual Basic 4.0, and others.

A visual representation of the steps required to determine the appropriate properties and methods for an OCX can be found at Apiary's Web site.

Price: US$249
Contact: Apiary, Inc., 10201 W. Markham, Suite 108, Little Rock, AR 72205
Phone: (501) 221-3600
Fax: (501) 221-7412
E-Mail: Internet: info@apiary.com
Web Site: http://www.apiary.com

Delphi TOOLS

New Products and Solutions

TurboPower Releases Memory Sleuth 1.0

TurboPower Software Company of Colorado Springs, CO is shipping *Memory Sleuth 1.0*, an application designed to help programmers find and repair bugs when developing in Delphi 2.

Prior to this release, debugging tools reported errors as hexadecimal addresses that needed to be decoded and matched to a programmer's source code. Memory Sleuth requires no source code changes or complex compile/link switch settings. It tracks memory and resource allocation problems, and reports the line number in the source code where the memory allocation took place.

The program's Snapshot feature captures information about a running program. When a developer's program ends, Memory Sleuth provides statistics on its peak resource usage.

Additionally, Memory Sleuth has the ability to embed debugging messages in an application's source code. These messages appear on the Output Debugging window. Programmers can use these messages to follow their program's execution path, as well as check the value of key variables.

Memory Sleuth 1.0 ships with online documentation, free support, maintenance updates, and a 60-day money-back guarantee.

Price: US$49

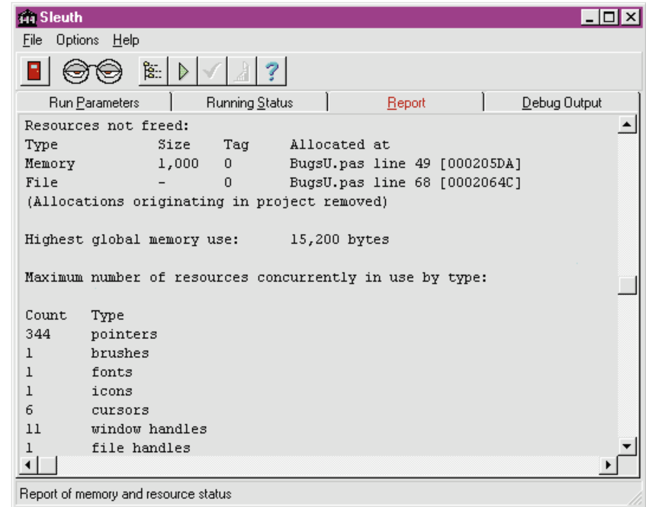
Contact: TurboPower Software Company, P.O. Box 49009, Colorado Springs, CO 80949-9009

Phone: (800) 333-4160 or (719) 260-9136

Fax: (719) 260-7151

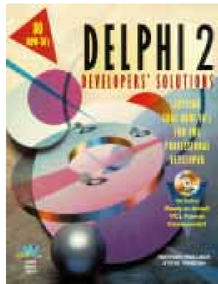
E-Mail: Internet: orders@tpower.com

Web Site: http://www.tpower.com



New Delphi Book

Delphi 2 Developers' Solutions
Nathan Wallace and Steve Tendon
Waite Group Press



ISBN: 1-57169-071-9
Price: US$59.99
(892 pages, CD-ROM)
Phone: (800) 428-5331

New Delphi Component Library for Database Development

In Livermore, CA, **Woll2Woll Software** announced the release of *InfoPower 2.0*, an upgrade to its component library for Delphi 1 and 2.

In version 2, InfoPower features picture-mask validation support. This allows developers to define picture masks that force end-users to enter specific characters, digits, and special characters, and only in specific positions within a field. Picture-masks are used during editing for automatic filling of values and validation when posting records.

InfoPower has also added visual filtering and visual querying components. End-users now have the ability to visually filter a table or query, or modify the WHERE clause of an existing SQL statement. Although the dialog is capable of sophisticated SQL generation, it completely hides the filtering and SQL details from the end-user.

New functionality to the grid includes the ability to embed control types such as bitmaps and spinedits, support for scalable row heights with word-wrapping, and support for selecting multiple records.

Other additions to InfoPower 2.0 include enhanced lookup support, allowing users to fill a drop-down list using a SQL SELECT statement, or records from a dependent table for filtered lookups.

Version 2.0 also has a features-enhanced memo dialog to integrate with third-party spell checkers, allowing the combo box to display alias values, and support for auto-fill of the current date.

Price: US$199; source is available for an additional US$99.

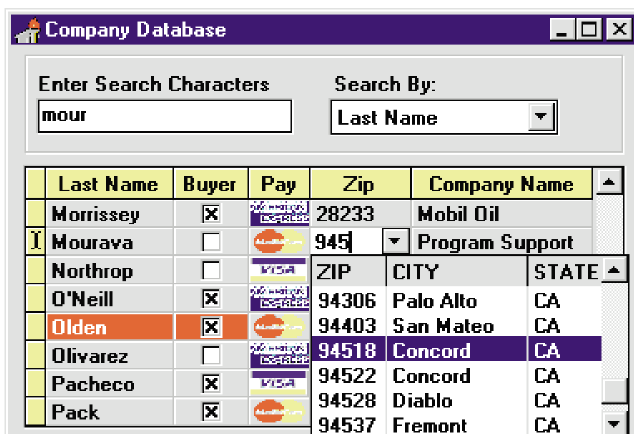
Contact: Woll2Woll Software, 2217 Rhone Dr., Livermore, CA 94550

Phone: (800) 965-2965 or (510) 371-1663

Fax: (510) 371-1664

E-mail: Internet: woll2woll@-woll2woll.com or CIS: 76207,2541

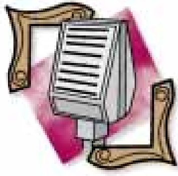
Web Site: http://www.woll2woll2.com



News

L I N E

September 1996



Delphi Client/Server Certification
Borland International has added new programs and resources — including certification for Delphi Client/Server, updated training courses, and an Authorized Education Center Program — to its client/server partner support program. Borland is offering the Delphi Client/Server certification exam, as well as a Train-the-Trainer course. Currently, Borland's Delphi Client/Server Train-the-Trainer course is scheduled for September 16-20 (Atlanta, GA). Borland is also planning special training for Delphi 1 users migrating to Delphi 2. For a certification fact sheet, certification study objectives, course fact sheets, and other information, visit Borland Online at http://www.borland.com.

Borland Announces New International Translation Tools for Delphi

Scotts Valley, CA — Borland International Inc. announced two new tools for translating Delphi applications into international languages: the Delphi Language Pack and the Delphi Translation Suite. These tools allow developers to localize Delphi applications into Danish, Dutch, English, French, German, Italian, Portuguese, Spanish, and Swedish.

The Delphi Language Pack includes system messages for Delphi's Visual Component Libraries (VCLs) and Borland Database Engine (BDE), as well as translated message templates for Delphi forms, dialog boxes, and menus. It also includes the Delphi Language Manager that allows developers to switch between languages.

The Delphi Translation Suite is a localization solution for corporate developers, ISVs, VARs, and system integrators who need to deliver larger-scale applications internationally with a more thorough translation than the Delphi Language Pack can provide. The suite can help translate new or existing applications, and includes all the functionality of the Delphi Language Pack — plus a suite of development tools to automate and test the translation process from one codebase.

Enhancements or bug-fixes to the Delphi Translation Suite apply immediately to all translated versions. The developer's translation staff doesn't need to be technical to do the localization, and cannot access or modify the application's source code. The Suite's GUI conversion features allow the translator to preview how the translation will appear, avoiding message truncation problems and pick letter duplicates. Most importantly, all the work of the Delphi Translation Suite is designed to be re-usable: as new versions of applications are created, only the new or altered items need to be translated.

The Delphi Language Pack and Delphi Translation Suite are scheduled to be available this Fall.

The Delphi Language Pack is priced at US$199.95; the Delphi Translation Suite is US$3,499.95.

Special introductory pricing is available for customers who place orders before September 30, 1996. For more information call Borland at (800) 233-2444.

Delphi Team Beats Usoft and Oracle

Portsmouth, UK — Dustin Thomas, one of Borland's Premier Client/Server Partners, received 1st place in the 1996 RAD Race held during Software Development 96 at the NEC Birmingham using Delphi Client/Server. The RAD Race is a two-day event sponsored by *D3*, APT Data Group's monthly development title, and the consulting group Ovum.

Other teams competing included second place winners Usoft, using Usoft Developer and Oracle; Compsoft using Equinox; ISDE using Prism; and Software Design Associates using Delphi and System Architect. The Software Design Associates team also received a commendation.

The Borland/Dustin Thomas team consisted of developers Jason Voles and Colleen Ridgewell.

The competition benefited the National Association for the Care and Resettlement of Offenders, who will implement the prize winning application.

The application developed was judged by a team of industry professionals, including Henk Bakker of Ovum; *D3*'s Technical Editor Lloyd Blythen; Greg Malewski of Enterprise IT; Mark Whitchorn of the University of Dundee; and NACRO's National Computing Officer Neil Russell.

Delphi 2 Receives Microsoft BackOffice Logo; Borland Products to Support Windows NT 4.0

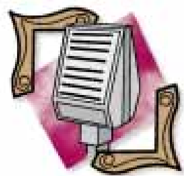
Anaheim, CA — Borland International Inc. announced Delphi Client/Server Suite 2 has received the Designed for Microsoft BackOffice logo.

In addition, Borland announced plans to support Windows NT 4.0 with its application development tools and databases, including Delphi, Borland C++ 5.0, Paradox 7, ReportSmith, InterBase 4.0, and the company's new Intranet development tool, IntraBuilder.

Microsoft's Designed for Microsoft BackOffice logo program allows customers to identify applications that are designed to work with the BackOffice family. The products are tested and approved by Microsoft's own BackOffice Software Testing Labs. Borland's products are also currently applying for Microsoft's new Designed for Windows NT and Windows 95 logo program.

News LINE

September 1996



Borland Posts First Quarter Loss; Wetzel Resigns

Scotts Valley, CA — Borland International Inc. announced it expects revenues for the first fiscal quarter, ending June 30, 1996, in the range of US$34 to US$35 million. As a result, Borland expects to report a substantial operating loss and a loss per share in the range of US$.53 to US$.56. Borland attributed the first quarter revenue decline primarily to lower than expected sales in the US.

Borland also announced the resignation of Gary A. Wetsel as president and CEO. The company has initiated a CEO search while Dr William F. Miller, Chairman of the company's Board of Directors, will serve as acting CEO until a replacement is appointed.

In addition, as an interim measure, Borland has created an office of the president, reporting to Dr Miller. It includes Paul Gross, senior vice president of Research and Development; Michael Greenbaum, vice president of Marketing and general manager of Client/Server Tools; and David McGlaughlin, vice president, International.

Borland's BAJA Component Event Model Included in Java Beans

Anaheim, CA — Borland International Inc. has announced its BAJA component event model for Java applications development is included in the draft specification for JavaSoft's Java Beans component model specification. Borland and several other industry leaders are participating in a JavaSoft-coordinated effort to develop a portable, platform-neutral Java component API.

Announced during the Borland Developers Conference last month, Borland also demonstrated a working version of its BAJA Property/Method/-Event (PME) programming component model integrated into Latte.

Java Beans will enable developers to write Java applets and applications from reusable components that can interact with other Java applets and applications, as well as with other component models.

Java Beans speeds application development by allowing developers to use existing Java applets and applications.

Java Beans, as exposed by Borland's Latte, provides the primary-level building blocks of functionality for Java developers. Similar to Borland's Delphi, Latte with Java Beans provides an integrated approach to building software with platform independence. For example, Java Beans, written entirely in Java, will allow components to be inserted in any other component architecture, including Microsoft's ActiveX, OpenDoc, and Netscape's LiveConnect.

Borland and Open Environment Amend Merger Agreement

Scotts Valley, CA — Borland International Inc. and Open Environment announced they have modified the terms of the agreement providing for the merger of Open Environment with Borland.

Under the terms of the amendment that was approved by the Board of Directors of both companies, Open Environment shareholders will receive a fixed ratio of 0.66 shares of Borland common stock for each share of Open Environment common stock.

The exchange ratio had been 0.51, but was subject to change based upon the value of Borland common stock over a specified period of time prior to the effective date of the merger.

With this amendment, the exchange ratio will no longer be subject to adjustment based upon the value of Borland common stock.

Open Environment has approximately eight million shares and vested options outstanding, giving an indicated value of approximately US$40 million for the transaction. Borland has approximately 31 million shares outstanding.

Completion of the transaction remains subject to regulatory approvals and approval by the stockholders of Open Environment. The transaction is expected to close later this summer or in early fall.

Delphi World Tour Update

Due to scheduling changes, the Delphi World Tour has announced new dates. The table below outlines the new dates and locations. In addition, the Delphi World Tour added a stop in Phoenix and removed its Frankfurt dates. The Delphi World Tour is sponsored by Borland International, Softbite International, and Informant Communications Group. These four-day seminars cover Delphi 1 and 2 programming techniques.

To receive a complete brochure via fax, call (630) 833-9122 and request document number 5. The brochure can also be requested from Softbite at (630) 833-0006. Softbite's Web site is located at http://www.softbite.com.

City	Dates
Orlando	Sep 10-13
Boston	Sep 16-19
Raleigh	Sep 23-26
Houston	Sep 23-26
Minneapolis	Sep 30-Oct 3
Atlanta	Oct 8-11
Seattle	Oct 8-11
Philadelphia	Oct 14-17
Denver	Oct 21-24
Los Angeles	Oct 22-25
Dallas	Oct 29-Nov 1
Newark	Nov 4-7
Chicago	Nov 4-7
Columbus	Nov 11-14
Phoenix	Nov 12-15
Washington, DC	Nov 19-22
San Francisco	Dec 3-6
Amsterdam	Oct 22-25
London	Oct 28-31

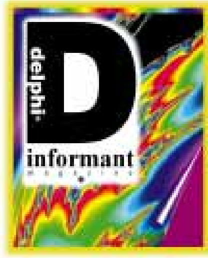
Errors and Omissions

Errors appear in Mark Ostroff's article "Leveraging ReportSmith: Part I" in the August 1996 *Delphi Informant*. On page 28, paragraph nine, third sentence, the correct statement should be:

> In the example shown in Figure 10, entering a variable name of *stateselection* would result in the value being ignored.

Also, to correct his biography, Mark Ostroff has been in the computer field for nearly 18 years and has worked for Borland for over six years.

We apologize for any confusion these errors may have caused.



ON THE COVER

Delphi 1 / Delphi 2 / Object Pascal

By *James Maycott*

Delphi Patterns

Creating a Generic Notification Mechanism in Delphi

In the object-oriented model, objects communicate by sending messages to each other. In Delphi, this translates to the invocation of object methods. Invoking a method requires that a connection exist between the sending and receiving objects. So how do we have an object communicate with one or several other objects to which it does not have apparent connections?

In this article, we'll discuss a solution to this problem that's implemented by a subscribe/notify scheme using a simple custom component and Delphi events. The notification mechanism illustrates the use of four object-oriented design patterns — the *Observer*, *Mediator*, *Proxy*, and *Singleton*.

First, we'll review methods and ways to invoke them in Delphi. Our discussion will then cover events as they relate to the custom component. (Readers already familiar with these topics can skip to the section, "Multi-Object Signaling.")

Invoking Methods

An object's behavior is implemented by the procedures and functions (i.e. its *methods*) that are defined by its class and ancestor classes. In Delphi, methods can be invoked in three ways, through:

- class references,
- object references, and
- method pointers.

A *class reference* can be used to invoke methods that apply to an entire class (*class methods*). *Object references* and *method pointers* can be used to invoke either class methods, or methods pertaining to a specific object instance (*object methods*).

Invoking a method through an object reference or class reference is more common and easier to understand, particularly for the Delphi novice. When a method is invoked through a reference, the method's name — as defined in the object's class — is explicitly stated along with the appropriate parameters (if any). For example, this code invokes a method through an *object reference*:

```
myObject.OneOfItsMethods('A String Parameter');
```

This example invokes a method through a *class reference* (the class is *TEdit*). In this case, the constructor is invoked at the class level to create a new object instance:

```
myEdit := TEdit.Create(nil);
```

You can also invoke a method through an indirect object reference. For example, an object reference can be obtained as the return value from a method, property, or utility routine. In the following code snippet, the *itsAssociatedObject* property returns a reference to another object. *AnotherMethod* is a method of the other object:

```
myObject.itsAssociatedObject.AnotherMethod;
```

A method pointer, as its name indicates, can contain the address of an object or class method. As stated in Delphi's online Help: "Delphi allows you to declare procedural types that are object methods, enabling you to call particular methods of particular object instances at run time."

A variable of procedural method type (e.g. a method pointer) can be assigned any object

or class method provided the method's signature matches that of the type. Once an assignment is made, the method pointer can be used to invoke the method.

Note that the *signature* of a function or procedure is defined by its argument list (the types of the arguments) along with, in the case of a function, the return value's type. Some references, such as Stanley Lippman's *C++ Primer* [Addison-Wesley, 1989], do not include the return value type as part of the signature (others references do). In my example, I include the return value of a function as part of the signature.

Before proceeding, let's cover more terminology:

- *Procedural type*. As used above, this is a type describing a function or procedure with a specific signature.
- *Procedural method type*. Refers to a procedural type that must be an object or class method.
- *Procedure method type*. Used to specifically refer to a procedural method type for a procedure.
- *Function method type*. Used to refer to a procedural method type for a function.

This first code snippet is a function method type declaration:

```
TASampleFunctionMethodType =
  function(x : integer) : integer of object;
```

The following example illustrates a procedure method type declaration:

```
TASampleProcedureMethodType =
  procedure(str : string) of object;
```

Figure 1 shows an example of invoking a function method and procedure method using a method pointer. Here are three items to consider regarding method invocations:

1) When a method is invoked through an object reference, it implies that the sending object has access to the entire **public** interface of the receiving object. That is, the sending object has knowledge of the receiving object's overall purpose and capabilities.
2) When an object method is invoked through a method pointer, the sending object does not necessarily know the class of the receiving object (e.g. its overall purpose). The only thing that the sender knows for sure is the method's signature being invoked (and likely, the message's purpose).
3) With regard to an object method invocation, whether the invocation is through an object reference or method pointer, some type of connection is required between the sending and receiving object. In the former, the connection is a reference to the target object. In the latter, the connection is the address of the method to be invoked.

Delphi Events

All object-oriented programming frameworks provide support for events in some form. These frameworks generate applications that run on top of event-driven operating systems or environments such as Windows, OS/2 PM, or X-Windows. Therefore, there's a clear need to support some type of event abstraction.

```
var
  iVar              : integer;
  DelegateFunction  : TASampleFunctionMethodType;
  DelegateProcedure : TASampleProcedureMethodType;
begin
  { Assume that SomeObject is accessible and that
    FunctionA's signature is the same as that specified
    by TASampleFunctionMethodType. }
  DelegateFunction := SomeObject.FunctionA;
  { Assume ProcedureA's signature is the same as that
    specified by TASampleProcedureMethodType.}
  DelegateProcedure := SomeObject.ProcedureA;
  { Invoking the function method through an
    object reference. }
  iVar := SomeObject.FunctionA(3);
  { Invoking the same method through a method pointer. }
  iVar := DelegateFunction(3);
  { Invoking the procedure method through an
    object reference. }
  SomeObject.ProcedureA('AB');
  { Invoking the same method through a method pointer. }
  DelegateProcedure('AB');
end;
```

Figure 1: Using a method pointer to invoke a function method and procedure method.

Just what is an event? Generally, you can consider an event "something significant" that can happen to an object that may need to be communicated to another object (or objects). These "interested" objects can then take appropriate action depending on the nature of the event.

Different frameworks enforce generating and propagating events in various ways. With Delphi, translating an event's abstraction to its physical implementation occurs as follows:

- A state change occurs to an object — the event generator — that meets the criteria for one of its defined events.
- The generator informs another object (the event receiver) of the event by invoking a method through a method pointer. This method pointer references an event handler in the receiver.
- The receiver reacts to the event by executing code within its event handling method.

Figure 2 is an extract of the event-related code from the source for the signaling component (we'll discuss this in more detail later). For now, here are some important items to note:

- *TSignalChange* is the definition (pre-defined signature) of the event handler method type used to respond to the component's *OnSignalChange* event. Any event handler defined for objects that use this component must be of this type. In general, event handlers must be procedure methods with the first parameter being a *TObject*. This parameter normally contains a reference to the object that invoked the event (the generator or "Sender"). Other parameters can be added as needed depending on the purpose of the event.
- *FOnSignalChange* is a private member of the component class. This attribute will be set to the address of a method (the *OnSignalChange* event handler for the receiver) with a signature that is the same as that defined by *TSignalChange*.
- *OnSignalChange* is a published property of the compo-

```
interface

type

  { 1 }
  TSignalChange = procedure(Sender : TObject;
    strKeyParm : string; strUserInfoParm : string;
    intUserInfoParm : integer; objectParm : TObject)
    of object;

  { The signal client class. }
  TpcsSignalClient = class(TComponent)
  private
    { Event handler method pointer. }
    { 2 }
    FOnSignalChange : TSignalChange;
    { Method that will be invoked by signal server that
      will, in turn, fire the OnSignalChange event. }
    procedure SignalChange(Sender : TObject;
      strKeyParm : string; strUserInfoParm : string;
      intUserInfoParm : integer; objectParm : TObject);

    ...

  published
    { Event handler property that will also show up
      in Object Inspector.}
    { 3 }
    property OnSignalChange : TSignalChange
      read FOnSignalChange write FOnSignalChange;
  end;

  ...

implementation

...

procedure TpcsSignalClient.SignalChange(Sender : TObject;
  strKeyParm : string; strUserInfoParm : string;
  intUserInfoParm : integer; objectParm : TObject);
begin
  { 4 }
  if Assigned(FOnSignalChange) then
    { Only fire event if an event handler has
      been assigned.}
    FOnSignalChange(self, strKeyParm, strUserInfoParm,
                    intUserInfoParm, objectParm);
end;

...
```

Figure 2: Event-related code from our signaling component.

nent that acts as a pseudo-variable for the *FOnSignalChange* **private** attribute. This exposes the event in the Delphi Object Inspector. You can then double-click on the event to create a skeleton of the handler for the owning component. When the handler's skeleton is generated, it's also associated with the event. The address of the handler is assigned to the event property.

Figure 2 illustrates the signaling of the event. For now, it's only important to know that the *SignalChange* method is invoked when the event must be fired. The code simply checks if a method address has been set to handle the event. If so, the method at that address is called.

Multi-Object Signaling

A common scenario in applications, especially complex ones, is the need to signal an event to multiple objects. Consider a

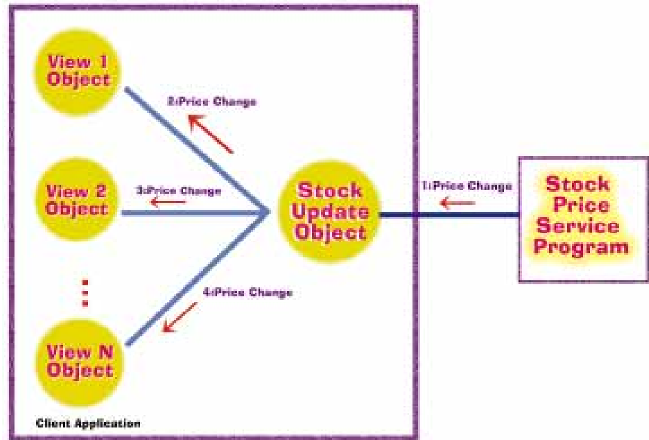


Figure 3: A stock price service program that feeds price data to a client application at regular intervals.

stock price service program that feeds price data to a client application at regular intervals (see Figure 3).

Let's say the client application maintains its connection to the service program through a stock update object. Whenever a change occurs to a stock price, the stock update object receives this information from the service application and passes it to multiple user interface view objects.

Since the number of views can increase or decrease dynamically, what is the best mechanism for signaling these changes to the views? Using standard method invocations would require the stock update object to maintain a reference to each existing view object. The stock update object would then invoke the appropriate method for each view to signal the update. To simplify this process it would make sense to define a specific method for the notification and then implement this method in any class that could connect to the stock update object.

We could try to inherit this method from a base class. However, non-view objects (e.g. a report generator object) may also need to use stock price data. The class hierarchy would, therefore, be difficult to design in Delphi's single inheritance model because it precludes the use of mix-in classes. Clearly, this approach would be awkward, and potentially expensive to implement.

Method pointers, in general, and Delphi events, in particular, are a more natural fit for handling this situation. We could define an event to represent the price change and have each view object implement the corresponding event handler. There are still some complications, however, because we must signal the event to multiple receivers. We must build an event handler method pointer list and maintain it dynamically, as well as provide a way for the views to specify the address of their respective event handlers.

The design decision is whether to build the required mechanism into the stock update object, or into a control object that will act as an intermediary between the stock update and the view objects (see Figure 4). Because this is just one example of a common situation, we'll build a generic, reusable control object.

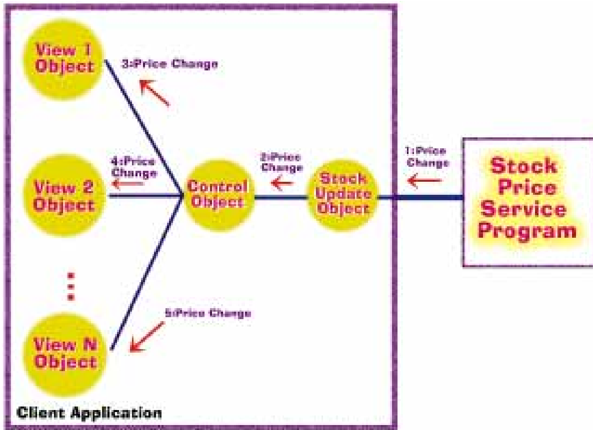


Figure 4: The stock price service program uses an intermediary control object.

The Signaling Component

The PCSSIGNL.PAS unit (see Listing One beginning on page 14) defines two classes, *TpcsSignalClient* and *TpcsSignalServer*. They work together to implement a subscribe/notify mechanism for routing notifications (events) throughout an application.

You can add the *TpcsSignalClient* component to the Component Palette. Any participant object wanting to use the notification mechanism must attach to its own dedicated *TpcsSignalClient* (signal-client) instance. With form objects, you can do this visually by selecting the component from the Component Palette and dropping it onto the form.

To attach to a signal-client instance at run time, an object can invoke the *TpcsSignalClient* class constructor. Once a signal-client is attached to an object, that object can use the signal-client to send notifications to any "interested parties" (i.e. other objects) in the application, and to receive notifications when appropriate.

A notification is always associated with an application entity. The term *entity* as used here, refers to any meaningful piece of an application that can be considered as having a modifiable state. Some examples of application entities are a query that may be refreshed; a file that may be updated; a section of code that may be executed (e.g. a secondary thread under Delphi 2); or any object whose state is meaningful to other objects in the application.

Each application entity is identified by a string key. For example, if we were to use signal-client objects from the stock update example, we may select the following string to represent the stock update object entity:

```
STOCK UPDATES
```

Another good choice for the entity key would be the stock update object's class name. In general, an application's developer must decide which entities are of interest, and assign a unique string key for each entity.

To receive notifications, an object must perform three actions. It must:

1) attach itself to its own dedicated signal-client object as previously described,
2) register for the entities it's interested in, and
3) define a handler for the *OnSignalChange* event generated by its attached signal-client.

If you place a signal-client component on a form, you can establish a skeleton handler at design time by double-clicking on the *OnSignalChange* event property. Otherwise, you must manually enter the entire event handler and associate it with the signal-client's *OnSignalChange* event at run time — most likely, immediately after the signal-client is instantiated. If at some point the object is no longer interested in a notification for a particular entity, it can de-register it.

For an object to send a notification it only needs to be attached to a signal-client instance. The signal-client object provides a method used to generate the notification.

The Signal-Server Object

The signal-client component allows objects to notify others of significant events and receive such notifications. The signaling mechanism also requires a central point of control for handling the notification routing. This is the purpose of the signal-server object, an instance of the *TpcsSignalServer* class.

While any number of signal-client instances can exist, there is only one instance of the signal-server. The creation of this unique instance is encapsulated within the signal-client class. Objects that use the notification mechanism have no direct knowledge of the server object — they only deal with the interface presented by the signal-client.

The signal-server object maintains a master list of all the registration requests made by signal-client objects. Whenever a signal-client requests that a notification be sent on behalf of its attached object, the notification goes through the signal-server. The server looks through its list of registrations for all other client objects that are interested in the entity associated with the notification. The server then invokes a method of each client that in turn fires an event to the client's attached object (delivers the notification).

Notification Content

Each notification contains: a string key identifying the entity associated with the notification; an integer value; a string value; and an object reference. Note that depending on the purpose of the notification, the integer value and object reference can be used as needed.

The implementation of the stock update application in Figure 5 uses signal-clients and a signal-server. Although more objects and messages are involved, a "hard" connection no longer exists between the stock update object and each of the view objects.

Design Patterns in the Notification Mechanism

Any programmer wanting to improve his/her object-oriented skills is advised to pick up a copy of *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [Addison-Wesley,

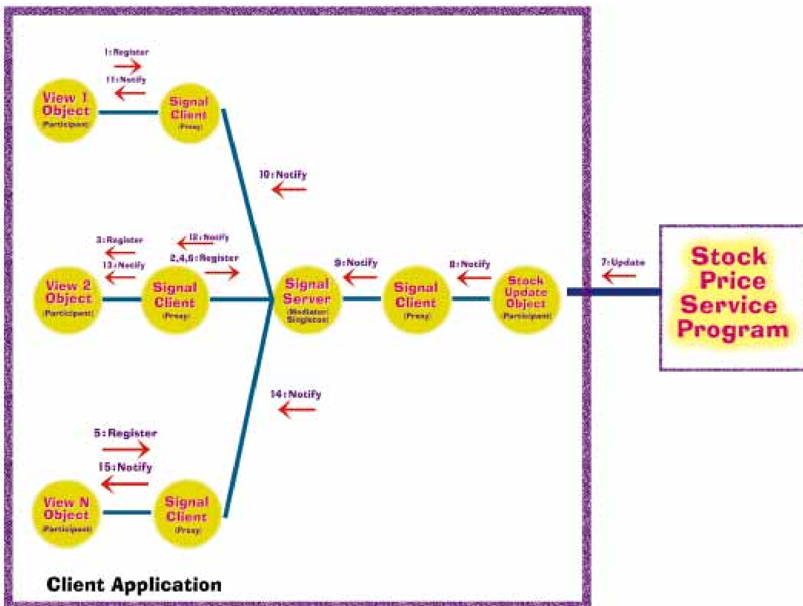


Figure 5: This diagram depicts the implementation of the stock update application using signal-clients and a signal-server.

1995]. This excellent book contains a descriptive list of reusable object-oriented design patterns. The authors define a design pattern as "a recurring pattern of classes and communicating objects that are found in many object-oriented systems." Four of these patterns are used in the implementation of the signal component's notification mechanism. [For a review of *Design Patterns*, see Richard Curzon's "The Gang of Four Speaks of Patterns" in the June 1995 *Delphi Informant*.]

The overall schema is an example of a variation of the Observer pattern. As described in *Design Patterns*, the Observer is used to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." The Observer pattern addresses all the design considerations regarding the signal component mechanism we discussed earlier.

In addition, the pattern also provides more specifics regarding the update of each dependent object's state. One other difference worth noting is that while the signal component mechanism requires an object to attach to a signal-client to participate (a form of composition), the Observer pattern uses inheritance to provide the required interface. Note that the registration, de-registration, and notification methods are derived from abstract base classes.

The Mediator pattern, as described in *Design Patterns*, defines an object that encapsulates how a set of objects interact. The Mediator promotes loose coupling by preventing objects from referring to each other explicitly. In addition, the Mediator allows you to vary the interaction of objects independently. In the notification mechanism, the signal-server instance acts as the mediator between participant objects.

Design Patterns describes the Proxy pattern as a means of "define[ing] a surrogate or placeholder for another object to control access to it." As you view the code for the signal-client component, you'll notice that the methods *RegisterKey*, *UnRegisterKey*, and *ChangeForKey* simply invoke a method in the signal-server with the same name. The signal-server actually handles the notification mediation between participating objects.

You may ask, "Why are we introducing another object, the signal-client, which seems to provide redundant functionality?" The answer is that we want to be able to represent an interface to the notification mechanism as a tangible component that can be used at design time. The signal-client serves this purpose, and acts as a proxy to the signal-server object.

The fourth pattern used is the Singleton. *Design Patterns* defines the Singleton as a way to "ensure a class has only one instance, and provide a global point of access to it." The signal component mechanism uses the Singleton pattern to ensure only one instance of the signal-server object is created, and that the signal-client proxy objects can have access to it.

For a more detailed description of each of these patterns, including the formal class hierarchies and object structures involved, please refer to *Design Patterns*.

Examining the Code

Listing One shows the complete source listing of the notification mechanism. This article is also accompanied by a simple test harness to demonstrate the component's use. In addition to class definitions of the signal-client and signal-server, the PCSSIGNL unit also contains related definitions and variables. The test harness is composed of the TESTSIGM, TESTSIGU, and TESTOBJ units.

TpcsSignalClient

As mentioned, the signal-client acts as a proxy to the signal-server. As such, the *TpcsSignalClient* class consists only of methods and an event property.

TSignalChange is the definition of the procedural method type that represents the *OnSignalChange* event. *OnSignalChange* is fired by a signal-client to inform its attached participant object of a state change to an entity for which it has registered. The parameters for the event are:

- *Sender* — Will always be the participant's signal-client.
- *strKeyParm* — A string key identifying the entity to which the notification applies.
- *strUserInfoParm* — Additional information as a string.
- *intUserInfoParm* — Additional information as an integer.
- *objectParm* — Additional information as an object. If the entity key represents some application object, *objectParm* would typically contain a reference to that object.

The purpose of the *FOnSignalChange* attribute and its corresponding property *OnSignalChange* was addressed in the section "Delphi Events."

RegisterKey is called by a participant object to express interest in an entity identified by *strKeyParm*. When a participant is no longer interested in an entity, it will invoke *UnRegisterKey* — again specifying the string key for the entity as a parameter. If a participant wants to notify other interested parties of a change to an entity's state, it invokes *ChangeForKey*.

The parameters for the *ChangeForKey* method are the same as the last four parameters of the *OnSignalChange* event. A participant does not have to be registered to generate a notification indicating a change to an entity's state. The *RegisterKey*, *UnRegisterKey*, and *ChangeForKey* methods simply forward the request to the signal-server — this is the actual mediator in the mechanism.

The *SignalChange* method is invoked by the signal-server when it needs to route a notification to a participant. Its parameters are the same as that of the *OnSignalChange* event.

TpcsSignalServer

The first item to note about the *TpcsSignalServer* class is its placement within the unit. Since the instantiation of the signal-server is encapsulated within the signal-client's class, we must prevent the erroneous creation of other instances. We do this by hiding the signal-server class in the **implementation** section.

The next thing to note is the way the signal-server object is created. If you examine the constructor and destructor for the signal-client, you'll see that the signal-server's *Access* and *Release* methods are called instead of the signal-server's constructor and destructor. *Access* and *Release* ensure that only one instance of the signal-server exists, and control access to that instance.

As you'll recall, this is the intent of the Singleton design pattern. *Access* checks if an instance of the signal-server exists. If not, *Access* creates it through its **private** constructor. If the object already exists, it's returned. In either case, a reference count is incremented to track how many signal-clients are using the signal-server. The *Release* method performs the opposite function. It first decrements the usage count. When the count goes to zero, the signal-server object is destroyed via its **private** destructor.

The signal-server implements its participant registration list as the *TStringList* object *strlstKeys*. The *Strings* value for each entry will contain an entity key. The *Objects* value will reference a signal-client object that has registered for the entity on behalf of a participant. *RegisterKey* (called directly by the signal-client's *RegisterKey* method) adds an entry to *strlstKeys* using *strKeyParm* and *sigclientParm* as the values for *Strings* and *Objects*, respectively.

UnRegisterKey (called directly by the signal-client's *UnRegisterKey* method) iterates through *strlstKeys* and removes the entity specified by *strKeyParm* for the signal-client specified by *sigclientParm*. If an asterisk (*) is specified by *strKeyParm*, all entries corresponding to *sigclientParm* are removed from the list.

ChangeForKey, called directly by the signal-client's *ChangeForKey* method, looks for the first occurrence of the entity specified by *strKeyParm* in *strlstKeys*. For each occurrence of the entity, the associated signal-client's *SignalChange* method is invoked. This, in turn, invokes the signal-client's *OnSignalChange* event, thereby notifying the target participant.

The Test Harness

To demonstrate the mechanism, you can use the simple test harness to create a test program. First, the test program displays a main form with a single **New Form** button (see Figure 6). Each time you press **New Form**, a new instance of a second-level form is displayed. Each second-level form instance has a signal-client component attached, and can act as a participant in the notification mechanism.

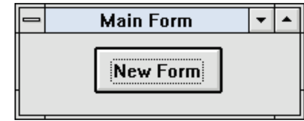


Figure 6: The test project you can build using the downloadable code from this article.

The second-level form features three rows of controls. Each row has a label, three pushbuttons, and a read-only edit field. Each label caption (**Entity1**, **Entity2**, and **Entity3**) represents an application entity with a state associated with it. The **Unregister** and **Register** buttons are used to register and de-register the form for the corresponding entity. The **Signal** button is used to emulate a state change. When you press **Signal**, the value in the corresponding edit field will increase by one. In addition, a notification will be generated that causes every other registered second-level form to increase its corresponding edit field (see Figure 7). Initially, all entities are registered.

The notification mechanism works in either 16-bit or 32-bit mode. To try the signal component and the test program, follow these instructions:

- If you are using the 16- version of Delphi, rename PCSSIGNL.DCR to PCSSIG32.DCR. Copy PCSSIG16.DCR to PCSSIGNL.DCR.
- Copy the files PCSSIGNL.PAS and PCSSIGNL.DCR to a directory where you store custom components.
- In Delphi 2, select **Component | Install** from the menu. In the 16-bit version, select **Options | Install Components** from the menu.

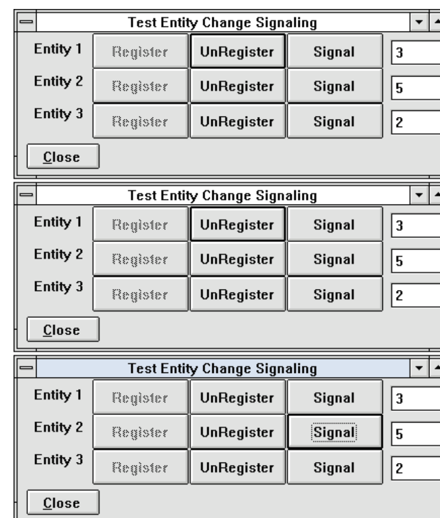


Figure 7: After clicking the **New Form** button three times, three instances of the Test Entity Change Signaling form display. Pressing the **Signal** buttons on the active form causes the numbers for the other entities to increment by one in the other inactive forms.

- Click the **Add** button; in the Add Module dialog box, select the PCSSIGNL.PAS file that you just copied.
- After you have selected the file and returned to the Install Components dialog box, click the **OK** button. This rebuilds your component library. When this is complete, the pcsSignalClient component will be added to your Component Palette under a tab labeled Custom.
- Open a new application project and remove the default form unit that is created for you (do not save it).
- Add the TESTSIGM.PAS unit to your project.
- Save the project and then select **Project | Build All** (in Delphi 2) or **Compile | Build All** (Delphi 1) from the menu.
- Run the project and the main form will be displayed.

Click the **New Form** button three times. This creates three instances of the second-level form. You'll want to reposition the forms to make them clearly visible.

Press the **Signal** button for one of the entities on one of the second-level forms. The corresponding edit field on all the forms will increase by one. The update of the edit field on the same form as the button you pressed occurs as a result of the form's button click handler. The update of the other two forms occurs as a result of notifications.

If you press the **UnRegister** button on a form, as a result of a notification, it prevents the corresponding edit field from being updated. You can re-enable such updates by pressing the **Register** button.

If you press the **Signal** button for **Entity 3**, an Information dialog box is displayed, indicating that non-component objects can also access the notification mechanism (see Figure 8). This is generated by an object that the main form instantiates when it's created. This non-component object is attached to a signal-client object and is registered for **Entity 3**.

Figure 8: The Information dialog box is displayed when you click the bottom **Signal** button on the active form.

The code for the test harness is basic, so I have not included a detailed discussion of it.

Conclusion

A solid understanding of method invocation and event signaling is a critical part of mastering an object-oriented programming environment such as Delphi. Events are well suited for implementing notification schemes in an application. The component presented in this article is a simple example of a small building block upon which more complicated mechanisms can be built. Furthermore, this component illustrates that object-oriented design patterns are not just for the academic elite — they can and should have a place in every programmer's "bag-of-tricks." ∆

The demonstration files referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\-SEP\DI9609JM.

James Maycott is a Senior Software Engineer at PCSI, a leading client/server and Internet/intranet consulting and development firm. He can be reached at (201) 816-8002, or through CompuServe at 75131,1763 (on the Internet use 75131.1763@compuserve.com).

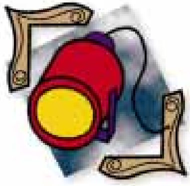
Begin Listing One: The PCSSIGNL Unit

```
unit Pcssignl;

interface

uses
  Classes;

type
  { -Signal Event Type Definition- This event will fire
    in a signal-client component when told to do so by
    the signal-server. The owner/user of this component
    should implement a handler for this event.  The
    strKeyParm will identify what entity has changed, and
    the event handler should act accordingly.
    strUserInfoParm, intUserInfoParm, and objectParm can
    contain additional information about the signal. }
  TSignalChange = procedure(Sender : TObject;
    strKeyParm : string; strUserInfoParm : string;
    intUserInfoParm : integer; objectParm : TObject)
    of object;

  { The signal-client class acts as a proxy to the
    signal-server. }
  TpcsSignalClient = class(TComponent)

  private
    { Event handler method pointer. }
    FOnSignalChange : TSignalChange;
    { Method that will be invoked by signal-server that
      will in turn,fire the OnSignalChange event. }
    procedure SignalChange(Sender : TObject;
      strKeyParm : string; strUserInfoParm : string;
      intUserInfoParm : integer; objectParm : TObject);

  public
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
    procedure RegisterKey(strKeyParm : string);
    procedure UnRegisterKey(strKeyParm : string);
    procedure ChangeForKey(strKeyParm : string;
      strUserInfoParm : string; intUserInfoParm : integer;
      objectParm : TObject);

  published
    { Event handler property that will also appear in
      object inspector. }
    property OnSignalChange : TSignalChange
      read FOnSignalChange write FOnSignalChange;
  end;

procedure Register;

implementation

{ The signal-server class:  Acts as a mediator of
  notifications between participants. Since the instance
  of the server object is only instantiated via a client
  instance, we make the class definition private. }

type
  TpcsSignalServer = class(TObject)
```

```
  private
    { Will hold list of key/client pairs. The string part
      of each entry will be the string key which identifies
      the entity. The object part is the client object that
      subscribed for the entity. }
    strlstKeys : TStringList;
    { Since server object is instantiated and destroyed via
      class methods Access and Release, the constructor and
      destructor are private. }

    constructor Create;
    destructor Destroy; override;

  public
    class procedure Access;
    class procedure Release(
      sigclientParm : TpcsSignalClient);
    procedure RegisterKey(
      sigclientParm : TpcsSignalClient;
      strKeyParm : string);
    procedure UnRegisterKey(
      sigclientParm : TpcsSignalClient;
      strKeyParm : string);
    procedure ChangeForKey(
      sigclientParm : TpcsSignalClient;
      strKeyParm : string; strUserInfoParm : string;
      intUserInfoParm : integer; objectParm : TObject);
  end;

{ Variables private to the TpcsSignalClient and
  TpcsSignalServer classes. Because Delphi's Object
  Pascal does not support class attributes, we use
  implementation variables that can only be accessed
  within this unit. }
var
  { The server instance. }
  theSignalServer : TpcsSignalServer;
  { Number of clients attached to server. }
  intUseCount : integer;

{ --------- TpcsSignalClient Methods follow --------- }

{ The constructor will ensure that the server object is
  created via the singleton design pattern. }
constructor TpcsSignalClient.Create(AOwner : TComponent);
begin
  { Initialize ancestor parts. }
  inherited Create(AOwner);
  { Instantiate / Access the signal server object. }
  TpcsSignalServer.Access;
end;
{ The destructor ensures that the client is detached from
  the server. }
destructor TpcsSignalClient.Destroy;

begin
  TpcsSignalServer.Release(self);
  inherited Destroy;
end;

{ This method will register the client with the server
  for notifications on updates to the entity identified
  by the specified key.

  Parameters:
  strKeyParm : The key for the entity. }

procedure TpcsSignalClient.RegisterKey(
  strKeyParm : string);

begin
  theSignalServer.RegisterKey(self, strKeyParm);
end;

{ This method will tell the server to no longer notify the
  client for the entity represented by the key.

  Parameters:
  strKeyParm : The key for the entity.  If '*', the client
  will no longer be notified for any entity changes that
  it was previously registered for. }
procedure TpcsSignalClient.UnRegisterKey(

  strKeyParm : string);
begin
  theSignalServer.UnRegisterKey(self, strKeyParm);
end;

{ This method will tell the server that the entity
  identified by the key has changed.  The server will then
  notify all registered clients  of the change (except the
  client that signaled the change).

  Parameters:
  strKeyParm : The key for the entity.
  strUserInfoParm : Additional information for change
    (string).
  intUserInfoParm : Additional information for change
    (numeric);
  objectParm : Addtional information for change
    (object) (e.g., the object causing the change, or the
     object that changed). }

procedure TpcsSignalClient.ChangeForKey(
  strKeyParm : string; strUserInfoParm : string;
  intUserInfoParm : integer; objectParm : TObject);
begin
  theSignalServer.ChangeForKey(self, strKeyParm,
    strUserInfoParm, intUserInfoParm, objectParm);
end;

{ This method is invoked by the signal-server. If an event
  handler has been defined for the OnSignalChange event,
  it is fired.

  Parameters:
  Sender : The server object
  strKeyParm : Key that identifies the entity
  strUserInfoParm : Additional information (string)
  intUserInfoParm : Additional information (numeric)
  objectParm : Addtional information (object). }

procedure TpcsSignalClient.SignalChange(Sender : TObject;
  strKeyParm : string; strUserInfoParm : string;
  intUserInfoParm : integer; objectParm : TObject);
begin
  { Only fire event if an event handler has been
    assigned. }
  if Assigned(FOnSignalChange) then
    FOnSignalChange(self, strKeyParm, strUserInfoParm,
                    intUserInfoParm, objectParm);
end;

{ --------- TpcsSignalServer Methods follow --------- }

{ This class procedure checks to see if the signal-server
  object exists, and if not, it instantiates it
  (singleton).  The instance is assigned to theSignalServer
  which can be seen by instances of TpcsSignalClient.
  This method is called in the constructor for
  TpcsSignalClient. }

class procedure TpcsSignalServer.Access;
begin
  if not Assigned(theSignalServer) then
    theSignalServer := TpcsSignalServer.Create;
  intUseCount := intUseCount + 1;
end;

{ This class procedure is called by TpcsSignalClient
  objects (in their destructor) to tell the signal-server
  that it is no longer being used by the specfied client.
```

```
  Parameters:
  sigclientParm : The TpcsSignalClient instance that is
  detaching from the server. }
class procedure TpcsSignalServer.Release(
  sigclientParm : TpcsSignalClient);
begin
  { Remove all items for this client from the server's
    internal list. }
  theSignalServer.UnRegisterKey(sigclientParm, '*');


  { Decrement the client use count. }
  intUseCount := intUseCount - 1;
  { If no clients are attached, destroy the server
    instance. We'll recreate it again when needed. }
  if intUseCount = 0 then
    theSignalServer.Free;
end;

{ Private constructor that is invoked by class Access
  method when the server object needs to be instantiated. }

constructor TpcsSignalServer.Create;
begin
  inherited Create; { Initialize ancestor part of object. }

  { Instantiate string list object for holding key /
    client object pairs. }
  strlstKeys := TStringList.Create;
  { Sort the list by the string key. }
  strlstKeys.Sorted := True;
  { A key may be in the list multiple times. }
  strlstKeys.Duplicates := dupAccept;
end;

{ Private destructor of the signal-server object.
  Invoked by class Release method when use count is 0. }
destructor TpcsSignalServer.Destroy;
begin
  strlstKeys.Free;
  theSignalServer := nil;
  inherited Destroy;
end;

{ This method is invoked by a signal-client to inform the
  server that it wants to be notified when changes occur
  relating to the specified key.

  Parameters:
  sigclientParm : The signal-client that wants to be
    notified when a change occurs relating to the key.
  strKeyParm    : The key. }


procedure TpcsSignalServer.RegisterKey(
  sigclientParm : TpcsSignalClient; strKeyParm : string);
begin
  { Add the key and client object to the string list. }
  strlstKeys.AddObject(strKeyParm, sigclientParm);
end;


{ This method will remove the entries for the specified
  key for the specified signal-client. Once unregistered,
  the client will no longer be notified when a change
  occurs relating to the specified key.

  Parameters:
  sigclientParm : The signal-client that no longer wants
     to be notified on changes relating to the
     specified key.
  strKeyParm : The key.  If '*', all items for the
     specified client will be removed from the server's
     internal list. }
procedure TpcsSignalServer.UnRegisterKey(
  sigclientParm : TpcsSignalClient; strKeyParm : string);
var
  i              : integer;
  intInitialCount : integer;
begin
  intInitialCount := strlstKeys.Count - 1;
  for i := intInitialCount downto 0 do begin
    { Remove specific key only for specified client. }
    if strKeyParm <> '*' then

      begin
        { We must check both the key and the object. }
        if (strlstKeys.Strings[i] = strKeyParm) and
           (strlstKeys.Objects[i] = sigclientParm) then
          strlstKeys.Delete(i);
      end
    else
      begin { Remove all keys for specified client. }
        { We only need to check the object. }
        if strlstKeys.Objects[i] = sigclientParm then
          strlstKeys.Delete(i);
      end;
  end;

end;

{ This method is invoked by a client when it has caused
  the entity associated with the specified key to change.
  The server will go through its internal list of clients,
  and for those that have registered for the key (except
  the client that caused the change), will invoke the
  client's private SignalChange method, which will fire an
  event for the client component (if an event handler has
  been defined).

  Parameters:
  sigclientParm : The signal client that caused the change
    for the entity.
  strKeyParm : The key representing the entity.
  strUserInfoParm : Additional information (string).
  intUserInfoParm : Additional information (numeric).
  objectParm : Additional information (object). }

procedure TpcsSignalServer.ChangeForKey(
  sigclientParm : TpcsSignalClient; strKeyParm : string;
  strUserInfoParm : string; intUserInfoParm : integer;
  objectParm : TObject);
var
  i : integer;
begin
  { First occurrence of key in list. }
  i := strlstKeys.IndexOf(strKeyParm);

  if i <> -1 then { At least one key was found. }
    repeat
      if sigclientParm <> strlstKeys.Objects[i] then
        (strlstKeys.Objects[i] as
         TpcsSignalClient).SignalChange(sigclientParm,
         strKeyParm,strUserInfoParm, intUserInfoParm,
         objectParm);
      i := i + 1;
    until (i = strlstKeys.Count) or
          (strlstKeys.Strings[i] <> strKeyParm);
end;

procedure Register;
begin
  RegisterComponents('Custom', [TpcsSignalClient]);
end;

initialization
  theSignalServer := nil;
  intUseCount := 0;
end.
```

End Listing One



Delphi 2 / Object Pascal

By Dana Scott Kaufman

Threaded Queries

Creating Delphi 2 Multithreaded Applications

One of the more highly-touted features of Microsoft's 32-bit operating systems is the ability to multitask, i.e. perform more than one task at a time. In Windows 95 and Windows NT, multitasking is implemented by the use of processes and threads.

A *process* refers to an instance of a running program. For example, an instance of WordPad and two instances of Notepad represent three processes. A *thread* describes a path of execution within a process. Think of it as the point in your program that is currently being executed. When a process is created by the operating system, the *main thread* is also created. Every process has at least one thread. Once the main thread is running, it can create additional threads within the process. For instance, an address book program might have a thread that searches for a name in a database, and another thread that dials the modem.

When you run a simple Delphi 2 program, you create a program containing a thread. Delphi 2 allows developers to harness this power for use in their programs. The advantage is that by using multithreaded development, programs no longer have to stop functioning while various events occur.

Delphi 2 allows developers to create multithreaded programs quickly and easily. [For a comprehensive introduction to Delphi multithreading, see Joseph Fung's article "Do the Strand" in the June 1996 *Delphi Informant*.]

A common task mentioned in the context of multithreading is the database query. By spawning a thread that executes a lengthy query, the user can still do other things while the thread is running. This article will demonstrate how to implement multithreaded queries with Delphi 2.

Anatomy of a Thread

To successfully create multithreaded applications, you must know the correct place in a thread to execute the queries. In Delphi, threads are created by deriving a new class from the built-in *TThread* class.

Next, the programmer is required to customize several events in the new thread function to make it perform as desired. Commonly, the *Create*, *Destroy,* and *Execute* methods are overridden. Resources required by the thread are allocated in *Create*; those resources are then de-allocated in *Destroy*. The *Execute* method is pivotal to multithreaded applications. After the thread is created, *Execute* is automatically called. All the code used to make the thread perform a specific task should go in the *Execute* method, or in a method called from *Execute*.

Because several threads can access the same resources at the same time, problems can occur. For example, if two threads are writing to a form concurrently, Delphi may be drawing one object and begin drawing

under the influence of another thread, potentially causing errors. Borland's solution to this problem is to call any code of a thread that might have problems with other threads, or the main Delphi thread through the *Synchronize* method.

TThread (and any object descended from it) has a special method named *Synchronize*. The *Synchronize* method suspends all other threads from executing — even the Delphi main thread — while the code contained in the method passed as a parameter to the *Synchronize* method executes. A program can execute only one "synchronized" method at a time. This means developers should only place code that absolutely must be synchronized in the *Synchronize* method. The prototype for a thread's *Synchronize* method is:

```
procedure Synchronize(Method: TThreadMethod);
```

The Golden Thread Rules

Here are some guidelines to follow when you're creating a multithreaded database application.

Each thread that will access a database requires its own session. A *session* is a connection through the Borland Database Engine (BDE) to a database. Each Delphi program that uses databases has one default session automatically assigned. Normally, any data access would go through this session.

With multithreading, more than one database action can occur simultaneously. Thus, a single *TSession* can cause a bottleneck. The solution is to have each thread create its own session. The BDE was designed to be "thread-safe" so that it will allow each database activity with its own session to function independently.

Don't open a *TQuery* or *TTable* component in a synchronized method. As mentioned earlier, when code is executed in a thread's *Synchronize* method, all other threads are suspended. If a query is opened in a synchronized method, all other threads will stop (including the Delphi main thread) while the query is executing. This will effectively make the program single threading because everything will pause while the query is executing. Luckily the BDE is multithread enabled. If we open the *TQuery* or *TTable* without synchronizing it, other threads can still execute.

***TQuery* and *TTable* components should not be attached to a *TDataSource* when they're opened.** This is important. If a *TDataSource* is attached to a *TDataSet* when it's opened, Delphi will try to draw the results on the form if any data-aware components are attached.

To avoid this problem, disconnect the *TDataSource* from the *TDataSet* in the *Synchronize* method before opening the *TDataSet*. After the *TDataSet* is open, reconnect the *TDataSource* in another *Synchronize* method. When the DataSource is reconnected to the *TDataSet*, any components attached to the DataSource (e.g. *TDBGrid*) will display the results. Here's an example:

```
Synchronize(DisconDataSource);   { Disconnect DataSource }
FQuery.Active := True;           { Start Query }
Synchronize(ConnectDataSource);  { Connect DataSource }
```

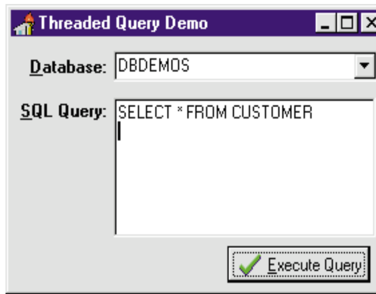


Figure 1: This simple form demonstrates the workings behind multi-threading queries.

```
procedure TMainForm.BitBtn2Click(Sender: TObject);
begin
  { Make sure database was selected. }
  if DatabaseComboBox.Text <> '' then
    { Create new form to hold query result set. Pass query
      information to the form. The TQueryForm contains the
      actual thread. }
    with TQueryForm.Create(Self,Memo1.Lines,
                          DatabaseComboBox.Text) do begin
      { Move form slightly to right. }
      Left := Left + QFormLeft;
      Show;   { Display form. }
      { Check if next form will display off screen. If so,
        start over on left hand side of screen. }
      if ((Left+QUERYFORM_OFFSET+Width)<=Screen.Width) then
        Inc(QFormLeft,QUERYFORM_OFFSET)
      else
        QFormLeft:=0;
    end
  else
    ShowMessage('You must first pick a Database!');
end;
```

Figure 2: The *OnClick* event for the **Execute Query** button.

A Simple Example

To demonstrate how multithreaded queries work, we'll create a program that allows the user to input a query and launch it as a thread in a separate window. The query will execute repeatedly until the user closes the window.

To start, create a main form containing a *TMemo* for the users to enter queries, a *TComboBox* for selecting a Database, and a *TButton* to launch the query. Your form should resemble Figure 1.

The **Execute Query** button creates a new form to run the query and displays the results. The SQL query to be executed and the database to execute the query against are passed as arguments in the *Create* statement (see Figure 2).

The real action occurs in the QueryResults form (see Figure 3). It features a *TDBGrid* to display the query answer set and a button to close the query.

When the form is created, it creates a thread that repeatedly executes the SQL query passed to it (this functionality is hidden from you). This

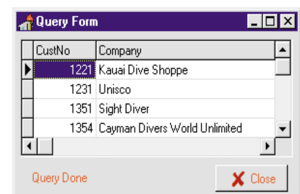


Figure 3: The demonstration QueryResults form creates a thread that repeatedly executes the SQL query passed to it.

makes it easy to launch many different queries. All the main program has to do is create a *TQueryForm* and pass the SQL statements to it. The database and QueryForm will handle the rest. Here is the *TQueryForm*'s *Create* method:

```
constructor TQueryForm.Create(AOwner: TComponent;
  SQLStrings: TStrings; DatabaseName: string);
begin
  inherited Create(AOwner);
  { Create a thread to execute the query }
  FQueryThread := TQueryThread.Create(Self,SQLStrings,
                                      DatabaseName);
end;
```

When the *TQueryForm* is closed, the thread associated with the form must be terminated by adding code to the *FormClose* event. We do this by calling the thread's *Terminate* method:

```
procedure TQueryForm.FormClose(Sender: TObject;
                               var Action: CloseAction);
begin
  FQueryThread.Terminate;
end;
```

A View of the Strand

Now let's look at the thread. Our initial task is to create a new class, *TQueryThread* that is derived from the *TThread* class. Each thread will have several private fields. Each thread needs:

- a *TSession* so it has its own handle to the database,
- a *TQuery* that will do most of the work,
- a *TDataSource* to hook the *TQuery* to a *TDBGrid,* and
- a **private** reference to the form that created the thread, so that it can connect the form's *TDBGrid* to the thread's *TDataSource*.

Also, the *Execute*, *Create,* and *Destroy* methods must be overridden so we can add our own code. Figure 4 is the class declaration for *TQueryThread*.

The *Create* method is passed a reference to the form that created it, as well as a list of SQLStrings containing the query to be run and the database name. *Create* carries out several important tasks. For example, it creates a new *TSession* for itself. Each thread has its own *ThreadId* that is an Integer. We use this to name the session because it's unique in the context of the operating system, and each session must be uniquely named.

```
TQueryThread = class(TThread)
private
  FSession: TSession;
  FQuery: TQuery;
  FDataSource: TDataSource;
  FOwnerForm: TForm;
protected
  procedure Execute; override;
public
  constructor Create( AOwner: TForm;
                      SQLStrings: TStrings;
                      DatabaseName: string);
  destructor Destroy; override;
  procedure DisconDataSource;
  procedure ConnectDataSource;
end;
```

Figure 4: Class declaration for *TQueryThread*.

```
constructor TQueryThread.Create(AOwner: TForm;
                                SQLStrings: TStrings;
                                DatabaseName: string);
begin
  inherited Create(False);
  { Create Session. Use new session for each Thread. }
  FSession := TSession.Create(nil);
  { Name Session with ThreadID, it's unique. }
  FSession.SessionName := IntToStr(ThreadID);
  { Create Query. }
  FQuery := TQuery.Create(nil);
  { Attach session. }
  FQuery.SessionName := FSession.SessionName;
  { Set query. }
  FQuery.SQL.Assign(SQLStrings);
  { Set Database. }
  FQuery.DatabaseName := DatabaseName;
  { Create DataSource. }
  FDataSource := TDataSource.Create(nil);
  { Save reference to the form the thread will
    display data on. }
  FOwnerForm := AOwner;
  { Set Grids DataSOurce. }
  TQueryForm(FOwnerForm).DBGrid1.DataSource := FDataSource;
  { Set thread to Free its resources when Terminated. }
  FreeOnTerminate := True;
end;
```

Figure 5: The demonstration application's *Create* constructor.

We must also create the *TQuery* to execute the SQL query. The *TQuery* is then attached to the newly-created *TSession* and assigned the SQL string and database the user has specified. We must also save a reference to the form that created the thread so we can later access components on the form. The *TDBGrid*, located on the form, is then pointed to the new *TDataSource*.

Lastly, we set the thread's *FreeOnTerminate* property to *True*. This will cause the thread to automatically free its resource when it's terminated. Figure 5 shows the *Create* constructor.

The thread now has all the essential components for running a query.

Executing a Query

Our next task is to create the query-executing routine. To do this we override the *Execute* method (see Figure 6) which will be automatically called after the thread is created. This is where we'll open the query.

The thread will continually re-execute the query until it's terminated. A **while** loop is used to accomplish this. First, we ensure the *TQuery* is connected to a *TDataSource*. We use the *Synchronize* method to guarantee that the code contained in the procedure passed to the method will be running exclusively. This will allow our code to use VCL components safely in the thread.

Next, to assure that the query is closed and disconnected from the DataSource, we change the *TQuery*'s *Active* property to *False* and the *TDatabase DataSource* property to **nil**. This must be done in a synchronized function, because if any visual component is connected to the *TDataSource*, they will be re-drawn to show no data when the DataSet is disconnected from the DataSource. The following method disconnects the query:

```
procedure TQueryThread.Execute;
begin
  { Run until thread is terminated. }
  while not Terminated do begin
    { Disconnect DataSource. }
    Synchronize(DisconDataSource);
    { Set label. }
    TQueryForm(FOwnerForm).Label1.Caption :=
      'Starting Query';
    FQuery.Active := True; { Start Query. }
    { Set label. }
    TQueryForm(FOwnerForm).Label1.Caption := 'Query Done';
    { Connect Datasource,the grid will be filled. }
    Synchronize(ConnectDataSource);
    { Pause so we can see the results of the query. }
    Sleep(1000);
  end;
end;
```

Figure 6: The demonstration application's *Execute* method.

```
procedure TQueryThread.DisconDataSource;
begin
  FQuery.Active := False;  { Close Query }
  { Disconnect DataSource from Query }
  FDataSource.DataSet := nil;
end;
```

Now we can execute the SQL query. We change the label's text so the user knows when the query is executing. Next, we set the *TQuery*'s *Active* property to *True*. Notice we aren't required to open the query in a synchronized method. This allows other threads to continue executing while our query is running.

After the query is open, we change the label to notify the user that the query is done executing. We then re-connect the *TDataSource* to the *TQuery*. Again, we call the procedure that accomplishes this through the *Synchronize* method. When the connection is made, the DataSource will cause the *TDBGrid* on the form to re-draw and display the query results. While doing this we must ensure this thread is the only one associated with the program that is executing. The code to connect the DataSource is:

```
procedure TQueryThread.ConnectDataSource;
begin
  FDataSource.DataSet := FQuery;
end;
```

Finally, we use the *Sleep* function to make the thread pause for a second so the result set is displayed in the *TDBGrid* long enough for the user to see it. After that, the program jumps to the beginning of the loop and restarts the process.

Conclusion

Multithreading programming can be a powerful development technique when used properly. Database programming is one of the best and most widely discussed uses for this type of development.

While not a trivial task, in certain cases, adding multithreaded queries to an application can make for a much more powerful and productive application. Delphi 2 includes all the tools for providing these capabilities to your users. ∆

The demonstration files referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\SEP\DI9609DK.

Dana Scott Kaufman is Chief Technical Officer and a Senior Consultant at Apogee Information Systems, Inc., a consulting and development firm specializing exclusively in Delphi- and Paradox-based applications. He has presented topics on Delphi development at the 1996 Conference to the MAX in the Netherlands, and topics on Java and Paradox at the 1996 Borland Developers Conference. Dana can be reached at (508) 481-1400 or by e-mail at aisys@ix.netcom.com.

By Cary Jensen, Ph.D.

The Object Repository

Visual Form Inheritance and Other Delphi 2 Delights

In Delphi 1 you have two options when you want to reuse forms between applications. One option is to add the form to the Gallery, and then select it any time you need the same form in another application. Alternatively, you can achieve the same effect by selecting **File | Add File** from Delphi's main menu, then selecting the unit associated with that form.

While you can reuse forms these same ways in Delphi 2, the new version of Delphi also lets you inherit from an existing form, rather than simply reusing it. The inherited form will have all the objects and event handlers of the original form. However, unlike form reuse, form inheritance permits you to add additional objects and event handlers — and even override inherited event handlers — without affecting the original form. This feature is referred to as Visual Form Inheritance (VFI).

(It's also possible to reuse a form by adding it to a DLL, and then using the DLL from multiple applications. However, this technique has limitations and is outside a discussion of the Object Repository, the focus of this article. Consequently, DLL-based forms are not included in this discussion.)

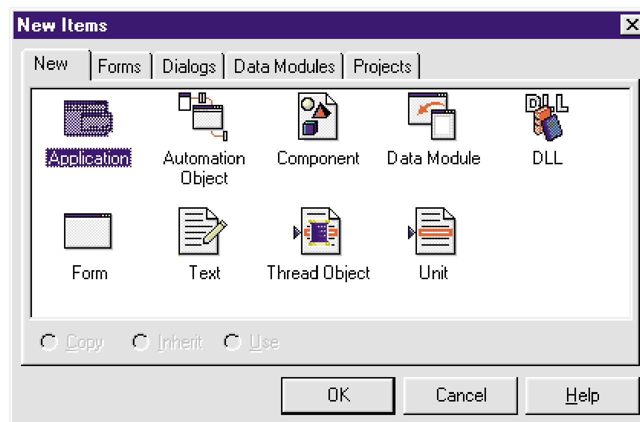


Figure 1: The New page of the Object Repository.

VFI is a feature of the Object Repository, which is a replacement for the Delphi 1 Gallery. There are several types of objects that can appear in the Object Repository, including experts, templates, and container objects (forms and data modules). Experts and templates are features the Object Repository has in common with the Gallery. Container objects, however, exist only in Delphi 2, and are the basis of VFI.

Using Experts

The New page of the Object Repository, shown in Figure 1, contains only experts. Additional experts are located on the Forms page (Database Form), Dialogs page (Dialog Expert), and Projects page (Project Expert). What these experts have in common is that they are code based. When you double-click on an expert, or when you select an expert and click the **OK** button on the Object Repository dialog box, Delphi runs the expert.

The effect of running an expert depends on which one you select. Some experts, such as the DLL expert on the New page of the Object Repository dialog box, simply create a new project containing a single source (.DPR) file that defines the library. You then add to this .DPR file any desired exports clauses, units, functions, procedures, resources, compiler directives, and so on.

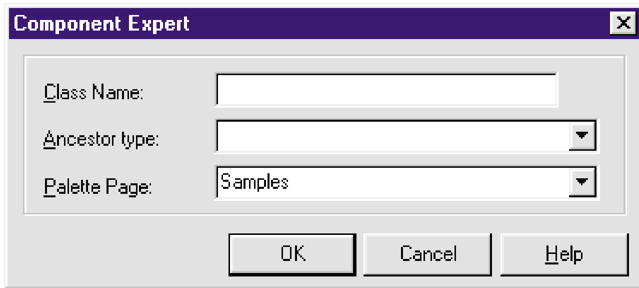


Figure 2: The Component Expert greatly simplifies the process of creating and registering a new component to the Component Palette.

(Note: In Delphi 2, a project file that defines a DLL does not necessarily need a unit. In Delphi 1, every DLL requires at least one unit, even if it is a blank one. Failure to provide at least one unit in a Delphi 1 DLL will result in an error message each time you load the DLL project, but that appears to be all. Unit-less DLLs in Delphi 1 seem to compile correctly.)

Other experts, such as the Component and Thread Object experts on the New page, and the Database Form expert on the Forms page, display a dialog box you use to provide the expert with details about the object you want to create. For example, the Component Expert (see Figure 2) requires you to provide a class name, ancestor type, and Component Palette page on which you want the component to be registered. After you've completed these experts, they generate any required project (.DPR), unit (.PAS), and form (.DFM) files.

If you're ambitious, you *can* create your own experts. However, doing so is one of the more challenging tasks in Delphi (in large part because of the limited documentation). If you are interested in creating your own experts, take a look at the DEMOS\EXPERTS directory, located under the Delphi subdirectory (this is true in both Delphi 1 and Delphi 2). In this directory you'll find one (Delphi 1) or two (Delphi 2) projects that define experts. The critical statement in these projects is a call to the *RegisterProc* function. This function, defined in the EXPTINTF unit, registers a new expert with the IDE (Integrated Development Environment). [Tony Yeung provides one approach to expert creation in his article "Expert Help" beginning on page 26.]

Using Templates

Unlike experts, templates are simply a collection of project files. When you select a template, Delphi:

- prompts you to select a directory in which to copy these files,
- copies the files, and
- opens the copy of the project.

In Delphi 1, the project files are located in subdirectories of the DELPHI\GALLERY directory. There are three templates in Delphi 1: CRTAPP, MDIAPP, and SDIAPP. Delphi 2 template subdirectories are located in the Delphi 2\OBJREPOS directory. Delphi 2 also has three templates: MDIAPP, SDIAPP, and LOGOAPP.

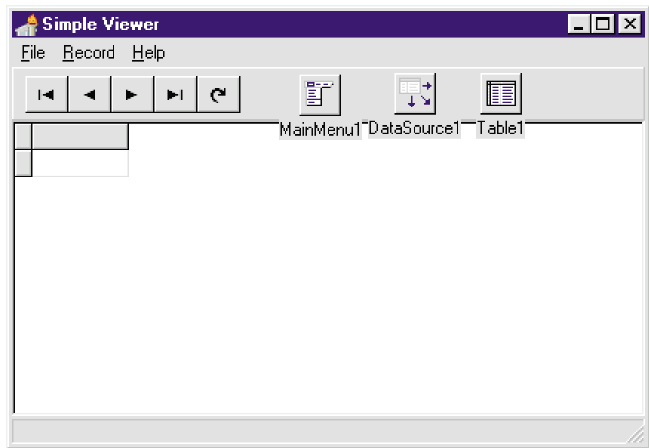


Figure 3: The main form of a project that contains the components and event handlers necessary for a simple viewer.

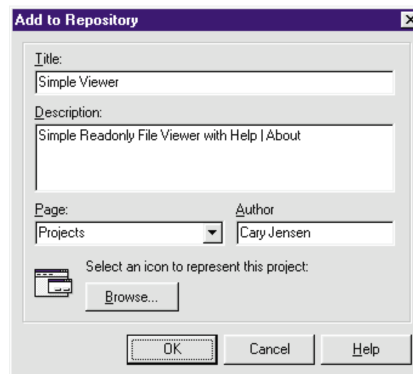


Figure 4: Use the Add to Repository dialog box to create a template from a saved project.

Adding Templates to the Repository

If you are building several applications that have a lot of code, objects, and forms in common, you may want to consider creating a template to hold the objects and code you want to appear in each application. A simple example of such an application, shown in Figure 3, illustrates the main form, which includes a menu. After you have saved this "start from here" application, select **Project | Add to Repository**. Delphi displays the Add to Repository dialog box shown in Figure 4.

Using the Add to Repository dialog box, you specify the title of the template, provide a description, indicate on which page of the repository you want it to appear, and specify the author's name. You can also click on the **Browse** button to select a custom icon for the project. This must be a 32x32 pixel, 16-color icon. After you have completed this form, click **OK** to add the template to the Object Repository. Figure 5 displays this newly added template on the Projects page of the Object Repository dialog box.

Unlike the project templates that ship with Delphi, your project is not automatically copied to the Delphi 2\OBJREPOS directory. Instead, the directory in which you have stored your project template will serve as the home for the template files. To reduce the likelihood you'll delete the template sometime in the future, it's recommended that you always store the projects you want to use as templates in a subdirectory under Delphi 2\OBJREPOS.

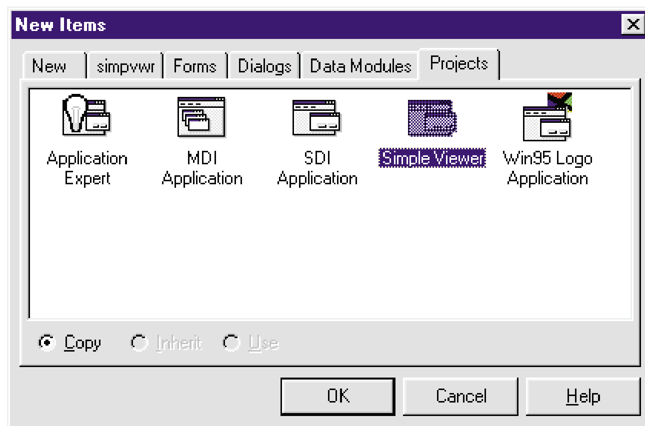


Figure 5: The Simple Viewer project is now a template in the Object Repository.

If you decide to make a change to a template after saving it to the Object Repository, remember two things:

1) These changes will not affect projects previously created from the template.
2) Any subsequent projects created from the template will make use of template changes.

Visual Form Inheritance

VFI is a natural extension of templates — with a twist. As you learned at the beginning of this article, VFI permits form inheritance (unlike templates, which only permit copies). With copies, there is no further relationship between two projects after the template has been selected from the repository. By comparison, when you use the Inherit feature of the Object Repository, the newly created form is a descendant of the original. Therefore, subsequent changes to the original will be inherited by the descendant the next time the project is compiled.

Any form or data module added to the repository can be used in one of three ways: Copy, Inherit, and Use. You define which of these techniques to use by means of the radio buttons that appear at the bottom of the Object Repository. In many cases — such as with all templates and some experts — only the **Copy** radio button is available. Likewise, only the **Inherit** radio button is active when the form or data module listed in the Object Repository is part of the current project. Finally, all three radio buttons, **Copy**, **Inherit**, and **Use**, are available for any form or data module explicitly added to the Object Repository.

When you use **Copy**, the effect is identical to that produced by a template. Specifically, a copy of the form or data module is added to the current project, using the current state of the form or data module. Subsequent changes to the form or data module will not affect the copy, and vice versa. Clearly, using the **Copy** feature is not VFI.

Just as clearly, you *are* using VFI when you select the **Inherit** radio button in the Object Repository. And when you do, the unit that defines the selected object is added to your project. In addition, a second unit and accompanying form file (.DFM) is created, which includes a **type** statement that specifies a new object derived from the selected class.

To see an example, start a new project by following these steps: select **File | New**, select the About box object from the Forms page, click **Inherit**, then click **OK**. Delphi will add the ABOUT.PAS file (the unit that declares the AboutBox form) to your project. In addition, it will generate a new unit that derives a new object from the *TAboutBox* class.

The following is an example of a project source file as it appears immediately after inheriting the About box from the Object Repository:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' { Form1 },
  About in
    '\PROGRAM FILES\BORLAND\DELPHI 2.0\OBJREPOS\ABOUT.pas'
    { AboutBox },
  Unit2 in 'Unit2.pas' { AboutBox2 };

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TAboutBox2, AboutBox2);
  Application.Run;
end.
```

In this example, the derived object is defined in Unit2. The following is the code found within this unit:

```
unit Unit2;

interface

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

type
  TAboutBox2 = class(TAboutBox)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  AboutBox2: TAboutBox2;

implementation

{$R *.DFM}

end.
```

Notice the *TAboutBox2* class is derived from the *TAboutBox* class. Because of this, the *TAboutBox2* class inherits all objects defined for the *TAboutBox* class, as well as any event handlers created for it.

A Warning

Adding the inherited unit to your project makes it rather easy to move to that unit and its associated form, as well as make changes. *It is very important you don't make changes to this original unit or its form.* Any changes to the inherit-

ed unit or its form will be applied to the original, and will affect every object inherited from it the next time the descendant objects are compiled.

Just as you can add member fields or objects to any derived class, you can add objects to a derived form or data module. However, you cannot remove any components from a form that is derived from another form (just as you cannot remove fields or objects from a traditionally derived class). Therefore, choose wisely the form you want to inherit. Don't inherit from a form or data module if it contains any objects you don't want in your derived class.

Selective Inheritance

As mentioned earlier, event handlers are also inherited. In other words, even if you don't add code to an inherited form, all event handlers for the ancestor form are still executed. Fortunately, Delphi makes it easy to skip these inherited event handlers, if necessary.

This can be demonstrated by creating a new application (select **File | Application**). Select **File | New**, then inherit the Dual list box from the Forms page of the Object Repository, as shown in Figure 6. Don't forget to click the **Inherit** radio button.

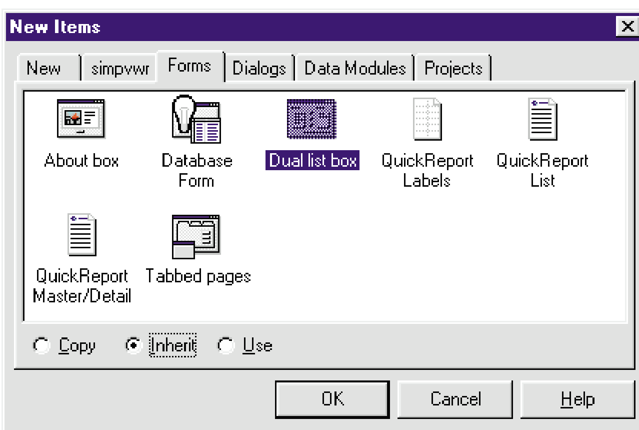


Figure 6: Inheriting from the Dual list box from the Forms page of the Object Repository.

Now imagine you want to change the behavior of one of the buttons on the derived Dual list box, DualListDlg2. Note that the *TDualListDlg* class includes a button named IncAllBtn. When clicked, it moves all objects from the left list box to the right list box. Imagine you want to display a dialog box asking for confirmation before moving the items, even though the event handler declared in the *TDualListDlg* class does not include such a behavior (see Figure 7).

To do this, display the form DualListDlg2. Then double-click the IncAllBtn SpeedButton (the button with two right arrows on it). Delphi will create the following *OnClick* event handler for this button:

```
procedure TDualListDlg2.IncAllBtnClick(Sender: TObject);
begin
  inherited;
end;
```

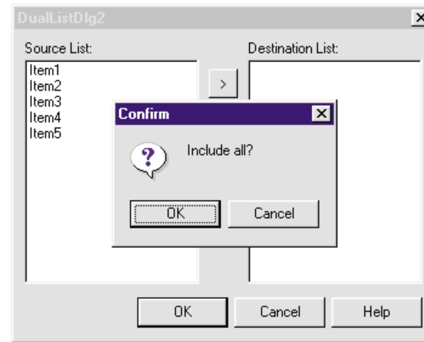


Figure 7: The Confirm dialog box prompts users to "Include all?" when moving objects from the left list box to the right list box.

Notice the reserved word **inherited**. This directive instructs Delphi to execute the inherited event handler. Now consider what happens if you make the following modification to this event handler:

```
procedure TDualListDlg2.IncAllBtnClick(Sender: TObject);
begin
  if MessageDlg('Include all?',mtConfirmation,
                [mbOK,mbCancel],0) = mrOK then
    inherited;
end;
```

With this new event handler, the inherited event handler is only called if the user confirms he or she wants to include all elements from the left list box in the right list box. If the user does not select the **OK** button from the Confirm dialog box, **inherited** is not called, and the inherited event handler is not executed.

After you have made these changes, you need only two more steps before you can execute this demonstration. First, you must go to Unit1 and select **File | Use unit**. From the Use unit dialog box, select Unit2. Finally, you'll want to add a button to Unit1, and display the DualListDlg2 dialog box when that button is clicked. The following is an example of such an event handler:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DualListDlg2.ShowModal;
end;
```

Saving a Form or Data Module to the Object Repository

If you have a form or data module that you want to allow other projects to inherit, simply move to that form, right-click, and then select **Add to Repository**. Delphi displays the Add to Repository dialog box shown in Figure 8. This form is similar to the one you use to add a project template to the repository, except that it only adds a single form or data module. Furthermore, while projects can only be copied, adding forms permits inheritance.

Deriving from the Current Project

If you want to inherit from a form that is already part of the current project, it is not necessary to add that form to the Object Repository. By default, when you have a project open and select **File | New**, the Object Repository will contain a page labeled with the name of your project. On this page you will find the forms and

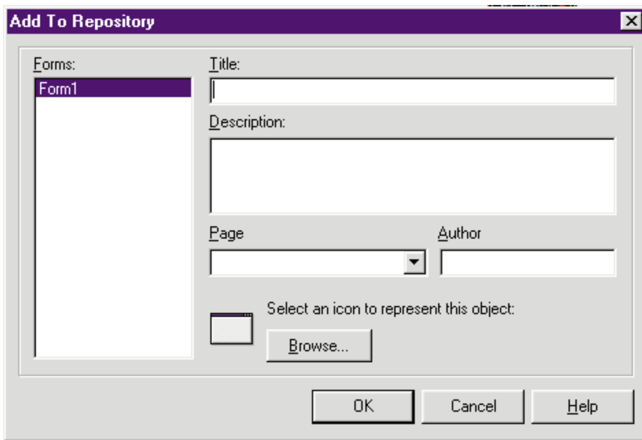


Figure 8: Right-click a form or data module and select **Add To Repository** to display the Add to Repository dialog box.

data modules of your project. These forms and data modules can only be inherited from, they cannot be copied or used.

Using Forms from the Object Repository

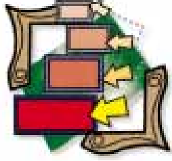
The final topic that relates to the Object Repository is the **Use** radio button. If you decide to use a form or data module from the Object Repository, you will find yourself working with the original file. In other words, **Use** does not add a descendant of the object to your project. Consequently, any changes you make to a form or data module that you select with the **Use** option will affect any classes derived from that form or data module. In general, you should not employ the **Use** option.

Conclusion

While the Object Repository offers all of same features of the Delphi 1 Gallery, it adds Visual Form Inheritance. Using VFI, you can quickly and easily derive new classes based on existing forms and data modules. When used judiciously, VFI can improve the speed with which you develop applications, as well as simplify the process of application revision. ∆

The demonstration forms and report referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\SEP\DI9609CJ.

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



Delphi 1 / Delphi 2 / Object Pascal

By *Tony Yeung*

Expert Help

Using Delphi to Create Complex Experts

Experts — known as "wizards" in Microsoft-ese — have become mainstays of the Windows environment. The good news is that most experts can be easily implemented with standard Delphi components. Use a *TForm* with a *TNotebook*. Add a few pages with some controls. Add **Back**, **Next**, and **Finish** buttons. Throw in some code to tie it all together. Voilà! You have an expert.

Experts are best used in situations where the user must be lead through a complex procedure that requires many steps. However, due to the number of steps, an application's resource consumption, and response times, code complexity and reuse can become difficult to manage.

In this article, we'll discuss some of the problems that can arise with complex experts, and cover an approach to solving them.

Problem: Resource Consumption

If your expert has 10 pages, each with 10 controls, Delphi treats it as a single dialog box with 100 controls (more if you include the notebook, pages, and Back, Next, and Finish buttons).

When your expert is created, all controls on all the pages are also created. Those of us still developing software for Windows 3.x hold our breath as the resource meter drops. Response times can also be affected by a lot of initialization during form creation (e.g. loading list and combo boxes, setting initial states of check boxes and radio buttons, etc.).

You may think you'll never have to write an expert with that many pages — consider yourself fortunate if this is the case. However, experts tend to have many more pages than steps required to complete the task. As the user proceeds through the expert, the subsequent pages that appear depend on the user's response at each step. For example, in an expert that collects information about a person for medical purposes, the first page may prompt the user to enter a person's name, age, sex, and smoking status. There may be subsequent pages that display if the information entered on the previous page indicates that the person is male, female, smokes, or older than 65.

Problem: Code Complexity and Reuse

Because an expert is a single dialog box, one unit exists to house all its code and data. The code managing the various panels and their controls is dispersed throughout the unit.

If you would like to reuse one of the pages from your expert in a second expert, you must extract the code relating to that page from your first expert and add it to the second. A client page may find its way onto most of the experts in an application. (Consider how much work it would be to make a change in the client page in all the experts individually.)

A Solution

This solution considers two main objectives:

1) Alleviate the resource crunch. Some of our more complex experts have nearly 500 controls contained in the 10-15 pages of the notebook. When one of these dialog boxes is open in the development environment, it uses so many resources that it must be closed before opening another dialog box or continuing to run the application.
2) Solve the code sharing problem. Although many of the pages are identical between experts, they are implemented with code copied between each of the dialog boxes. Aside from being difficult to manage, the complex dialog boxes have so much code controlling all of the pages that they may have reached the 64KB limit for code in a single segment.

Our solution involves two classes: *TAutoNotebook* and *TChildForm*.

The *TAutoNotebook* Class

The critical part of the solution is the *TAutoNotebook* component, a subclass of the *TNotebook* component. *TAutoNotebook* adds only a few methods, properties, and a list to accomplish its task. AUTONB.PAS (see Listing Two beginning on page 29) is the source for the *TAutoNotebook* class.

As with *TNotebook*, the *Pages* property is used at design time to identify the individual pages in the notebook. However, the page names do more than just create a blank page for each string in the list. At run time, if the page name corresponds to a form class, an instance of that form will be created and its controls will appear when that page receives focus. Figures 1 and 2 are *TClientDlg* forms appearing independently, and as part of an expert dialog box. Page names that are not form classes will behave as they would with *TNotebook*. Therefore, you can combine form pages with non-form pages.

The component uses a *TList*, *FChildForms*, to keep track of the child forms it creates. This list is initialized in the *Loaded* procedure with a **nil** pointer for each page in the notebook. Any forms created at run time are stored in this list and destroyed automatically in *TAutoNotebook*'s destructor.

The child forms, if needed, can be accessed directly through the *ChildForms* property. You can also override the normal creation order of the child forms by directly calling the public procedure *CreateChild*. Normally, the child forms are created when the page receives focus.

To have the component detect when a page gains focus, one of the two new properties must be used to change pages. *AutoActivePage* and *AutoPageIndex* serve the same purpose as *TNotebook*'s *ActivePage* and *PageIndex*, respectively. If you use *TNotebook*'s properties to change the current page, *TAutoNotebook* won't be called to create the child forms for the pages. This is inconvenient, but necessary, because the method that changes pages in the base class isn't virtual, and therefore can't be overridden.

Figure 1 (Top): An independent *TClientDlg* form.
Figure 2 (Bottom): The *TClientDlg* form as part of the expert.

Unfortunately, the *ActivePage* and *PageIndex* properties can't be hidden by the *TAutoNotebook* class. The *TNotebook* component doesn't follow the pattern of most of the other components because it isn't derived from a *TCustomNotebook* class that implements all the behavior, and a *TNotebook* class that simply exposes the properties.

When the page changes (using *TAutoNotebook's* new properties), the *AutoSetPageIndex* method is called. Before changing to the new page, the *ValidateChild* method is called to perform edits on the current page. If the current page has a child form, *ValidateChild* calls its *CloseQuery* method to allow the child form to perform its own edits. Thus if this child form is used independently or as part of several experts, the same edits will be applied.

Finally, if the edits succeed, *AutoSetPageIndex* calls *CreateChild*. This creates the child form for the new page before using *TNotebook*'s *PageIndex* property to change to the new page.

Creating the Child Form

The *TAutoNotebook* method, *CreateChild*, is where the magic occurs. *GetClass* is used with the page name to determine if the name corresponds to a registered class:

```
frmClass := TFormClass(GetClass(Pages<nPage>));
```

We'll discuss registration in greater detail later.

If the class is found, the return value from *GetClass* can be used as a class reference to create an instance of the form:

```
if frmClass <> nil then
  ChildForms<nPage> := frmClass.Create(Self);
```

To make the form appear as if it were a page in the expert, the current page is assigned as the parent of the child form's controls. To simplify this process, the child forms are designed with a *TPanel* named *AutoNBPanel* that holds all the controls:

```
pnl :=
  ChildForms<nPage>.FindComponent('AutoNBPanel') as TPanel;
if pnl <> nil then
  pnl.Parent := TWinControl(Pages.Objects<nPage>);
```

This allows a search for this panel with *FindComponent* and a reassignment of the parent without processing each control individually. This also solves another problem where the tab order for the controls is lost when they are reassigned individually.

Class Registration

To make *GetClass* aware of the child form classes, they must be registered with a call to *RegisterClass*. According to the *Delphi Component Writer's Guide*, *RegisterClass* is used to register a class with the Delphi streaming system. Classes are not normally required to be registered in this manner. However, to create the child forms without specifying the type at compile time, the call to *RegisterClass* is required.

The **initialization** section of the unit where the form is located is an appropriate place to register the child form classes:

```
initialization
  RegisterClass(TClientDlg);
end.
```

The *TChildForm* Class

Child forms are derived from *TChildForm* to facilitate their management. This class is defined in the *AutoNb* unit. Although Delphi 1 doesn't easily support visual form inheritance, you can derive your forms from a class other than *TForm* by simply changing the class declaration. This works well, provided the new parent class doesn't have any visual components you want to manipulate in design mode.

TChildForm provides some useful procedures that the child forms can override to get control at certain times. The *Initialize*, *FocusPage*, *CancelChanges*, and *CommitChanges* procedures have default empty implementations. Subclasses only have to override these methods if needed.

Initialize is called after the child form is created. A pointer to the expert dialog box is passed to the child form for convenience. Note that the normal creation sequence still applies. You can put class initialization logic in either the *Create* constructor or *OnCreate* event handler. The *Initialize* method, which occurs last in this sequence, is provided if the child form must differentiate its behavior, depending on whether it's running independently or as part of an expert. *Initialize* will only be called in the latter case.

The *FocusPage* procedure is called just after the child form becomes the active page. *FocusPage* corresponds to the *OnPageChanged* event of the *TNotebook* class, and is called after any event handler assigned to this event in the expert dialog box.

CancelChanges and *CommitChanges* are used to cancel or save changes to the child forms. In the sample application that accompanies this article, these methods are called in the event handlers for the **Cancel** and **Finish** buttons in the expert dialog box.

The Sample Application

This article is accompanied by all the necessary files to run the NBDemo sample application. You can build the application by unzipping the files and executing the command line compiler as follows:

```
c:\delphi\bin\dcc nbdemo.dpr
```

To use *TAutoNotebook* in your projects, install the component in the conventional manner.

The NBDemo application shows the *TAutoNotebook* in action (see Figure 3). The application's client and address dialog boxes are used independently and as part of the expert dialog box. Although not implemented in NBDemo, any number of expert dialog boxes can be created, each using the same child forms.

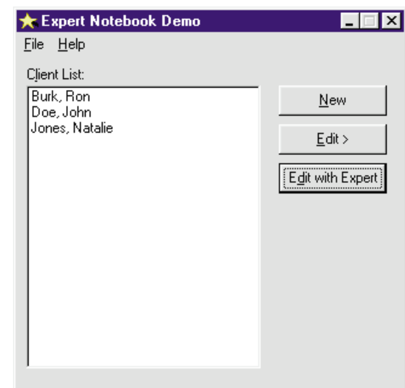


Figure 3: The first page of the Expert Notebook Demo program.

The Clients unit implements a simple client object that uses an .INI file to store data. How this unit functions is unimportant since the specific details of any implementation of *TAutoNotebook* will depend on the objects the application uses and how they access their data.

The Expert

The EXPERT.PAS file is the source for the sample application (see Listing Three beginning on page 31). This dialog box has two normal pages (the first and fourth) and two child form pages. Clearly, most of the code is simply for managing the buttons and changing the pages. As

mentioned earlier, the handlers for the **Cancel** and **Finish** buttons use the *CancelChanges* and *CommitChanges* methods of the *TAutoNotebook* to control the child forms.

The child forms are the client and address dialog boxes. The client dialog box requires the user to enter a first and last name to show that the same edits are performed regardless of whether the dialog box is running independently or as part of the expert (again see Figures 1 and 2). If the user does not enter values into **First Name** and **Last Name**, an error is raised (see Figure 4). The client and address dialog boxes override the *CommitChanges* method to save their changes.

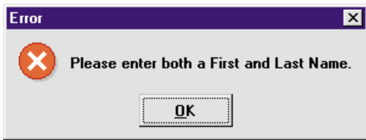


Figure 4: This Error dialog box is displayed if a user fails to enter information into **First Name** and **Last Name** and clicks the **OK** button.

The address page (see Figure 5) illustrates a slightly different approach to performing edits. Instead of waiting until the user clicks on **Next** or **OK** before checking the input, these buttons are not enabled until the input is valid. To accomplish this, the address page uses some of the optional procedures from *TChildForm*, *Initialize*, and *FocusChange*, along with a saved pointer to the expert's **Next** button.

Figure 6 is the last page of the sample application. It's displayed when the user finishes entering the required information.

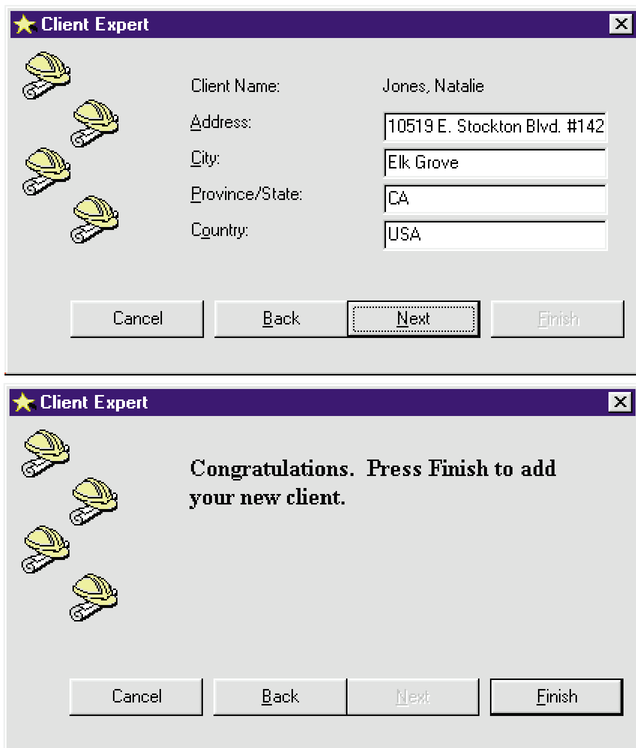


Figure 5 (Top): The address page.
Figure 6 (Bottom): The final page of our sample application.

Conclusion

With some of my current projects, the ability to share a child form among multiple experts is what makes this approach valuable. Before implementing this technique, we simply duplicated the code to manage identical pages in each dialog box — creating a development and maintenance nightmare.

Resource problems are also alleviated because the controls on each of the pages are not created until that page receives focus. As described, only a subset of the pages are shown during the normal use of the expert. Therefore, the total amount of resources required to use any of the experts is significantly reduced. ∆

The demonstration project and other files referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\SEP\DI9609TY.

Tony Yeung is a Systems Consultant with Manulife Financial. He has been developing in the Windows environment with Borland C++ and Delphi for several years. Tony can be reached at 72302.2112@compuserve.com.

Begin Listing Two: The AUTONB Unit

```
unit AutoNb;

interface

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

type
  TChildForm = class(TForm)
  private
  public
    procedure Initialize(Expert : TComponent); virtual;
    procedure FocusPage; virtual;
    procedure CancelChanges; virtual;
    procedure CommitChanges; virtual;
  end;

  TAutoNotebook = class(TNotebook)
  private
    FChildForms : TList;
    { Property Methods }
    procedure AutoSetActivePage(const S: string);
    function AutoGetActivePage : string;
    procedure AutoSetPageIndex(nPage : integer);
    function AutoGetPageIndex : integer;
    function GetForm(nPage : integer) : TForm;
    procedure SetForm(nPage : integer; AForm : TForm);
  protected
    procedure Loaded; override;
    function ValidateChild : Boolean; virtual;
  public
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
    procedure CreateChild(nPage : integer);
    procedure CommitChanges;
    procedure CancelChanges;
    { New properties }
    property AutoActivePage: string
```

```
      read AutoGetActivePage write AutoSetActivePage
      stored False;
    property AutoPageIndex : integer
      read AutoGetPageIndex write AutoSetPageIndex
      stored False;
    property ChildForms[nPage : integer] : TForm
      read GetForm write SetForm;
  end;

procedure Register;

implementation

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

{ Empty default implementations for TChildForm }
procedure TChildForm.Initialize(Expert : TComponent);
  begin end;
procedure TChildForm.FocusPage; begin end;
procedure TChildForm.CommitChanges; begin end;
procedure TChildForm.CancelChanges; begin end;

{ TAutoNotebook Implementation }

constructor TAutoNotebook.Create(AOwner : TComponent);


begin
  inherited Create(AOwner);
  FChildForms := TList.Create;
end;

destructor TAutoNotebook.Destroy;
var
  i : integer;
begin
  for i := 0 to FChildForms.Count - 1 do
    ChildForms[i].Free;
  FChildForms.Free;
  inherited Destroy;
end;

procedure TAutoNotebook.Loaded;
var
  i : integer;
begin
  inherited Loaded;
  { Put a nil pointer in the list for each page }
  for i := 0 to Pages.Count - 1 do
    FChildForms.Add(nil);
  { This will cause the current page to be setup }
  AutoPageIndex := PageIndex;
end;

procedure TAutoNotebook.CommitChanges;
var
  i : integer;
begin
  { Call each TChildForm to commit changes }
  for i := 0 to FChildForms.Count - 1 do
    if ChildForms[i] is TChildForm then

      (ChildForms[i] as TChildForm).CommitChanges;
end;

procedure TAutoNotebook.CancelChanges;
var
  i : integer;
begin
  { Call each TChildForm to cancel changes }
  for i := 0 to FChildForms.Count - 1 do
    if ChildForms[i] is TChildForm then
      (ChildForms[i] as TChildForm).CancelChanges;
end;

function TAutoNotebook.ValidateChild : Boolean;
begin
  { If the current page has a child form,
    call its OnCloseQuery event handler }
  if ChildForms[PageIndex] <> nil then
    Result := ChildForms[PageIndex].CloseQuery
  else
    Result := True;
end;

procedure TAutoNotebook.AutoSetPageIndex(nPage : integer);
begin
  { Only change page if validation of current page is
    okay }
  if ValidateChild then
    begin
      { Create the child if necessary }
      CreateChild(nPage);
      { Let TNotebook change the page }
      PageIndex := nPage;
      if ChildForms[nPage] is TChildForm then
        (ChildForms[nPage] as TChildForm).FocusPage;
    end;
end;

procedure TAutoNotebook.AutoSetActivePage(const S: string);
begin
  AutoPageIndex := Pages.Indexof(S);
end;

function TAutoNotebook.AutoGetActivePage : string;
begin
  Result := ActivePage;
end;

function TAutoNotebook.AutoGetPageIndex : integer;
begin
  Result := PageIndex;
end;

function TAutoNotebook.GetForm(nPage : integer) : TForm;
begin
  Result := FChildForms[nPage];
end;

procedure TAutoNotebook.SetForm(nPage : integer;
                                AForm : TForm);
begin
  FChildForms[nPage] := AForm;
end;

procedure TAutoNotebook.CreateChild(nPage : integer);
var
  frmClass  : TFormClass; { A class reference variable }
  pnl       : TPanel;
begin
  if (FChildForms.Count > nPage) and
     (ChildForms[nPage] = nil) then
    begin

      { Look for a class with the name of the current page }
      frmClass := TFormClass(GetClass(Pages[nPage]));
      if frmClass <> nil then
        begin
          { Class found, create and save in child list }
          ChildForms[nPage] := frmClass.Create(Self);
```

```
          { Look for the panel and reassign its parent }

          pnl :=
            ChildForms[nPage].FindComponent('AutoNBPanel')
            as TPanel;
          if pnl <> nil then
            pnl.Parent:=TWinControl(Pages.Objects[nPage]);

          if ChildForms[nPage] is TChildForm then
            (ChildForms[nPage]
             as TChildForm).Initialize(Owner);
        end;
    end;
end;

end.
```

End Listing Two

Begin Listing Three — EXPERT.PAS

```
unit Expert;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, ExtCtrls, AutoNb,
  StdCtrls;


type
  TExpertDlg = class(TForm)
    nb: TAutoNotebook;
    cmdNext: TButton;
    cmdBack: TButton;
    cmdFinish: TButton;
    cmdCancel: TButton;
    Label1: TLabel;
    Image1: TImage;
    Label2: TLabel;
    procedure cmdNextClick(Sender: TObject);
    procedure cmdBackClick(Sender: TObject);
    procedure cmdCancelClick(Sender: TObject);
    procedure cmdFinishClick(Sender: TObject);
    procedure nbPageChanged(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  ExpertDlg: TExpertDlg;

implementation

{$R *.DFM}

procedure TExpertDlg.cmdNextClick(Sender: TObject);
begin
  nb.AutoPageIndex := nb.AutoPageIndex + 1;
end;

procedure TExpertDlg.cmdBackClick(Sender: TObject);
begin
  nb.AutoPageIndex := nb.AutoPageIndex - 1;
end;

procedure TExpertDlg.cmdCancelClick(Sender: TObject);
begin
  nb.CancelChanges;
end;

procedure TExpertDlg.cmdFinishClick(Sender: TObject);
begin
  nb.CommitChanges;
end;

procedure TExpertDlg.nbPageChanged(Sender: TObject);
var
  IsLastPage : Boolean;
begin
  IsLastPage := nb.AutoPageIndex = nb.Pages.Count - 1;
  cmdFinish.Enabled  := IsLastPage;
  cmdFinish.Default  := IsLastPage;
  cmdNext.Enabled    := not IsLastPage;
  cmdNext.Default    := not IsLastPage;
  cmdBack.Enabled    := nb.AutoPageIndex > 0;
end;

end.
```

End Listing Three

By *Mark Ostroff*

Leveraging ReportSmith: Part II

Making the Delphi Connection

Last month we began this two-part series on Delphi's ReportSmith by explaining when and how to use ReportSmith in your Delphi development, as well as how to call ReportSmith reports using report variables. This month, we conclude this series by addressing the use of the new Delphi Connection.

Getting Connected

The Delphi Connection consists of a new connection type that gets its data from a Delphi 2 DataSet object. To activate the use of the direct Delphi Connection, you must invoke ReportSmith by double-clicking on a *TReport* component within the Delphi 2 IDE. This launches ReportSmith in a Delphi-aware mode. You will then see a new item labeled `Delphi` in the drop-down list of available connection types (see Figure 1). After you select `Delphi` as the connection type, click on the **Server Connect** button below the connection **Type** drop-down list. ReportSmith will then display the names of all the DataSet components (i.e. *TTables* and *TQueries*) in your application.

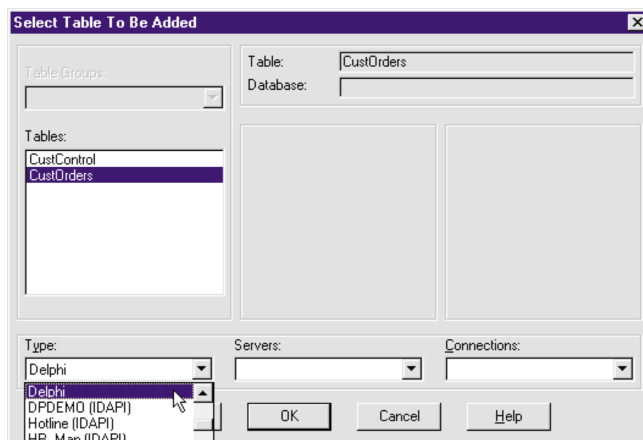


Figure 1: Selecting the Delphi Connection from ReportSmith.

Even if you have more than one DataSet in your Delphi application, you'll only be able to select one. ReportSmith relinquishes all control of the DataSet to Delphi. Thus, all data selection, table joins, data groupings, and sorting are performed by the Delphi DataSet object. As a result, you probably won't use a *TTable* DataSet with the Delphi Connection. The single DataSet restriction usually proves too limiting to make *TTables* a useful DataSet for these kinds of reports.

Where Do You Control Each Option?

Using the Delphi Connection requires a little forethought when setting up the Delphi DataSet. You may have to adjust the way you set up the SQL statement of your *TQuery* to accommodate the results you want in your report. To attain the desired report result, you must also adjust some of the ways you work with ReportSmith. The table in Figure 2 indicates whether an option is controlled by the Delphi DataSet or by ReportSmith.

Using the Delphi Connection

As in 16-bit reporting, report variables can still be used. However, the primary use of report variables in a Delphi/ReportSmith application is to limit the record selection to a user-supplied value. The techniques described here provide an alternative way to limit the reported records. (Note: You may still occasionally need ReportBasic for some reporting tasks.)

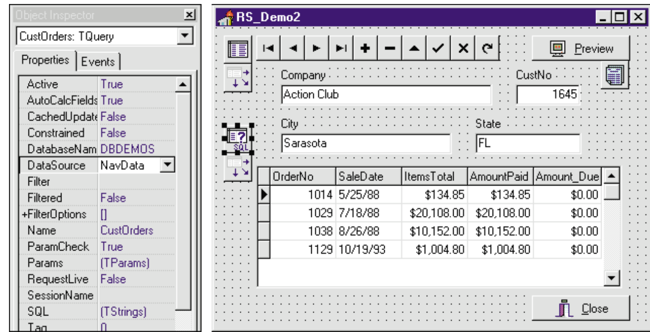
ReportSmith Tool	Option Controlled By
Tables	Controlled mostly by ReportSmith, but limited to selecting a single Delphi DataSet.
Selections	Determined by the Delphi DataSet only.
Sorting	Determined by the Delphi DataSet only.
Derived Fields	Still controlled in part by ReportSmith. However, only derived fields created by ReportBasic macro can be created. Any derived field that needs to be part of the record selection criteria or data grouping should be built as a calculated field in the Delphi DataSet.
Report Variables	Still controlled by ReportSmith.
Database Grouping	Determined by the Delphi DataSet only.
SQL Text	Determined by the Delphi DataSet only.
Report Grouping	Still controlled by ReportSmith. In fact, this is the only way to provide grouping in the report beyond what was created in the Delphi DataSet.
Summary Fields	Still controlled by ReportSmith.
Field Selection Criteria	Determined by the Delphi DataSet only.
Merge Reports	This option is disabled, since the Delphi Connection only allows a single data source.

Figure 2: Control points in Delphi Connection reports.

Limiting the record scope. By default, a Delphi Connection provides all the records in the DataSet to your report. Even if you can't see the records on-screen in your Delphi application, they will be included in your Delphi Connection report. This can present a bit of a challenge if you want to limit which records are included in a report, yet provide access to a broader range of records in your Delphi application.

The answer is to make use of a parameterized query as the DataSet used by the report. This *TQuery* object will be controlled by another DataSet object within your Delphi application. The controlling DataSet object will limit the records displayed at any one time in the report's *TQuery*. By navigating the controlling DataSet, you'll be able to navigate the report's *TQuery* data as well.

The RS_Demo2 project shown in Figure 3 illustrates this technique. It uses a *TDBNavigator* hooked to a *TTable* named CustControl. A *TQuery* named CustOrders is the data source for the DEMO2.RPT report. It is also used in the DBEdit components and the DBGrid that provide the data display. The records selected by the CustOrders *TQuery* are controlled by the CustNo field in the CustControl *TTable*. To provide the coordination between CustOrders and CustControl, the *DataSource* property of CustOrders is set to `NavData`, the *TDataSource* connected to the CustControl DataSet. The SQL for the CustOrders DataSet is shown in Figure 4. (This SQL SELECT statement was built entirely in the Visual Query Builder.)




```
SELECT  CUSTOMER."CustNo",
        CUSTOMER."Company",
        CUSTOMER."City",
        CUSTOMER."State",
        ORDERS."OrderNo",
        ORDERS."SaleDate",
        ORDERS."ItemsTotal",
        ORDERS."AmountPaid" ,
       (ORDERS.ItemsTotal-ORDERS.AmountPaid) AS Amount_Due,
       (CUSTOMER.Company + "  " + CUSTOMER.City + ", " +
        CUSTOMER.State) AS Account
  FROM  "CUSTOMER.DB" CUSTOMER, "ORDERS.DB" ORDERS
 WHERE  CUSTOMER.CustNo = ORDERS.CustNo
   AND  CUSTOMER."CustNo" = :CustNo
ORDER BY CUSTOMER."Company", ORDERS."OrderNo"
```

Figure 3 (Top): The RS_Demo2 project.
Figure 4 (Bottom): The SQL SELECT statement for the CustOrders DataSet.

The WHERE clause that states:

```
CUSTOMER."CustNo" = :CustNo
```

is the key (again, see Figure 4). The colon designates the name of a parameter to use in determining which records are selected and displayed. When the name of a parameter matches the name of a data field in a DataSet pointed to by a query's *DataSource* property, the value of that field in the current record is used in the WHERE clause. Thus, as you navigate the records in CustControl, the parameter used by CustOrders changes, and the query is automatically re-run. The result is that you can scroll through all the customer accounts, yet only display and report on one account at a time. The data display results of this kind of set up are shown in Figure 3.

Creating the Delphi Connection report. To set up a report to use this connection, double-click on the *TReport* component. ReportSmith will be launched with the ability to select `Delphi` from the list of types shown in the lower left of the Select Table To Be Added dialog box. To activate it, click on the **Server Connect** button after you select this connection type. You should now see a list of DataSet names from your currently open Delphi project. Select the `CustOrders` DataSet and click on the **Done** button.

You will also notice at this point that some of the report tool buttons at the top of the dialog box are dis-

abled. If you need to create derived fields or record selections within your SQL, you'll need to do so in the Delphi DataSet. This is why the SQL text of the CustOrders query includes SQL code to create two calculated fields: the Amount_Due field and the Account field (a concatenation of the company's name, city, and state).

You also must remove the DataSet, then add it back into your report if you make any changes to the DataSet once you begin designing your report. Otherwise, any newly created calculated fields will not show up in the list of available fields in ReportSmith.

Group headers and footers in Delphi Connection reports. Your Delphi DataSet is in control of the SQL text being used to build this type of report. Because most of the toolbar buttons affect the SQL code being built, the Delphi Connection will disable many of them. Included in the collection of disabled buttons are the ones that create group headers and footers.

Fortunately, ReportSmith also provides another process for creating these report constructs. Creating group headers and footers thus becomes a two-step process:

1) Create a data field based group. Because even calculated fields are coming from the Delphi DataSet, ReportSmith considers all fields supplied by the Delphi Connection as data fields. Use the Tools | Report Grouping option to call up the Define Groups dialog box. Select the field from the list on the left upon which you want to group the data. Then click the New Group button. You can also click the Group Properties button to further define the group's options.
2) Insert a header or footer (or both) for the new group. After you've created the group, you must create a header or footer for that group. Select Insert Header/Footer from the ReportSmith menu. Then select the appropriate Group Name and activate either the Header or Footer (or both) check box.

You can, of course, repeat this process with multiple report groups. After you have created a header or footer region, the summary buttons in the toolbar are activated. As with any other report, you can now select a column to summarize, and then click on the appropriate summary button.

Design-Time vs. Run-Time Considerations

Because you want to create a new report at design time, you don't want to fill in values for the report name or report directory properties of your *TReport* component. (Otherwise, double-clicking the *TReport* will try to launch ReportSmith with the named report — which doesn't yet exist.) This means you'll have to supply these parameters at run time *before* you try to activate the report. An example of typical code (the code on the Preview button) is shown here:

```
procedure TRS_Demo2.BitBtn1Click(Sender: TObject);
begin
  Report1.ReportName  := 'Demo2.rpt';
  Report1.ReportDir   := ExtractFilePath(ParamStr(0));
  Report1.Run;
  CustOrders.Active   := true;
end;
```

You can also see that the CustOrders DataSet is activated by this code *after* running the report. Why? This brings up another design-time vs. run-time consideration. This issue arises because of the way the Delphi Connection manages the DataSet itself. You'll notice the CustOrders DataSet is active when you first open the RS_Demo2 project. Now double-click on the *TReport* component and open DEMO2.RPT. When you close ReportSmith, you'll see that the CustOrders DataSet is no longer active.

To preserve the safety of data concurrency, the Delphi Connection automatically deactivates the data connection in your Delphi application as it passes control of the DataSet to ReportSmith. This is actually a good idea. Unfortunately, Delphi has no real way to determine when control can be safely returned from ReportSmith, so your code must reactivate the data connection.

Considerations When Your Application Navigates Records

You naturally want the report to reflect the current set of records displayed in your application. This coordination happens automatically when you first activate the report. After that, however, you have two separate applications running in a semi-independent fashion. If you want to maintain the coordination between the two, you must code that for yourself in your Delphi application.

Fortunately, this coordination is rather uncomplicated. Simply recalculate the report whenever you move to a new record in CustControl, the controlling DataSet. Unfortunately, the recalculating of the report will also deactivate the data connection in your Delphi application. So your code needs to handle a couple of side effects: the reactivation of the data connection in your application, and the elimination of the resulting screen flicker. The example shown in Figure 5 is tied to the *OnClick* event of the DBNavigator.

First of all, the reactivation of the CustOrders DataSet must occur after the report is recalculated. The *RecalcReport* is what will cause the deactivation. Second, you want to eliminate any screen updates throughout the process of recalculating the report, the resulting deactivation of the data connection in Delphi, and the coded reactivation of that same DataSet. The code in Figure 5 uses the Windows API function *LockWindowUpdate* to suppress screen redraws on your form. The value this function expects to receive is the window handle of the form to be locked. No window will have a handle of zero, and only one window at a time can have its updates locked. Thus, telling *LockWindowUpdate* to lock window handle zero turns off the screen lock. Unlocking your

```
procedure TRS_Demo2.DBNavigator1Click(Sender: TObject;
  Button: TNavigateBtn);
begin

  if Report1.ReportName <> '' then
    begin
      Screen.cursor := crHourGlass;
      { Eliminate screen flicker }
      LockWindowUpdate(RS_Demo2.handle);
      Report1.RecalcReport;
      CustOrders.Active := true;
      { Restore form updating }
      LockWindowUpdate(0);
      Screen.cursor := crDefault;
    end;

end;
```

Figure 5: If your code must handle side effects, such as the reactivation of the data connection in your application and the elimination of the resulting screen flicker, try this example — in this case tied to the *OnClick* event of the DBNavigator.

form will cause it to automatically perform a refresh, using the currently selected data.

Controlling ReportSmith via DDE

All versions of ReportSmith support DDE as both a client and server. You can take advantage of this ability to control the run-time reporting environment from within your Delphi application.

Setting up the DDE Conversation

Controlling ReportSmith from a Delphi application involves the use of the *TDDEClientConv* component on the System page of the Component Palette. This component allows your application to initiate a DDE conversation with ReportSmith. (ReportSmith acts as the DDE server in this situation.) Figure 6 shows the important *TDDEClientConv* properties. You will also need to focus on several key methods of the *TDDEClientConv* component, as shown in Figure 7.

The RS_PANEL Project

Using DDE to control ReportSmith means you are, in effect, creating an automated end-user. Two main avenues exist for exercising this control: the System and Command topics. The RS_PANEL project illustrates the use of both DDE topics to control ReportSmith. This project builds a Delphi application that uses DDE to control the zoom factor, window state, and toolbar status of the running version of ReportSmith. Figure 8 shows this ReportSmith Control Panel in action.

Using the System *DDETopic* to "Drive by Menu." Your use of the System *DDETopic* becomes a fairly simple matter of driving the ReportSmith menus from your Delphi application. A few issues of note exist in using the *ExecuteMacro* method to drive the System *DDETopic*.

First, the menu command must be placed in a null-terminated string. This means you'll need to use *StrPCopy* to place the command string into a null-terminated

Property	Function
ConnectMode	The *ConnectMode* property determines the type of connection to establish when initiating a link with a DDE server application. A value of *ddeAutomatic* (the default) means the link is automatically established when the form containing the *TDDEClientConv* component is created at run time. For most applications, you want to control when the DDE conversation is established. Set this property to *ddeManual*. This setting will establish the DDE conversation only when the component's *OpenLink* method is called.
DDEService	This property specifies the DDE server application to be linked to a DDE client. Typically, *DDEService* is the file name (and path, if necessary) of the DDE server application's main executable file without the .EXE extension. At design time, you can specify *DDEService* either by typing the DDE server application name in the object inspector, or by choosing **Paste Link** in the DDE Info dialog box.
DDETopic	This property specifies the topic of a DDE conversation. Typically, *DDETopic* is a filename (and path, if necessary) used by the application specified in *DDEService*. See the documentation for ReportSmith for the specific information about specifying a valid *DDETopic*. To control ReportSmith, you will typically use a topic of "System" or "Command". At design time, you can specify *DDETopic* by choosing **Paste Link** in the DDE Info dialog box.
ServiceApplication	The *ServiceApplication* property specifies the main executable file name (and path, if necessary) of a DDE server application, without the .EXE extension. Typically, this is the same value as the *DDEService* property. Sometimes, however, *DDEService* is a value other than the DDE server application's executable file name. In either case, *ServiceApplication* must be specified for Delphi to run an inactive DDE server to establish a DDE conversation. Make sure you supply a value of `ReportSmith` if your application is supposed to start ReportSmith.

Method	Function
CloseLink	Terminates an ongoing DDE conversation. After a link is closed, no DDE communication can take place between the DDE client and server until another link is opened.
ExecuteMacro	Attempts to send a macro command string to a DDE server application. The command string must be a null-terminated string that contains the macro to be executed by the DDE server application. *ExecuteMacro* returns *True* if the macro was successfully passed to the DDE server application. If *ExecuteMacro* was unable to send a command string, *ExecuteMacro* returns *False*. If you need to send a macro command string list rather than a single string, use the *ExecuteMacroLines* method.
OpenLink	Initiates a new DDE conversation. If the conversation was successfully opened, an *OnOpen* event occurs and the *OpenLink* method returns *True*. Otherwise, *OpenLink* returns *False*.

Figure 6 (Top): Key *TDDEClientConv* properties.
Figure 7 (Bottom): Key *TDDEClientConv* methods.

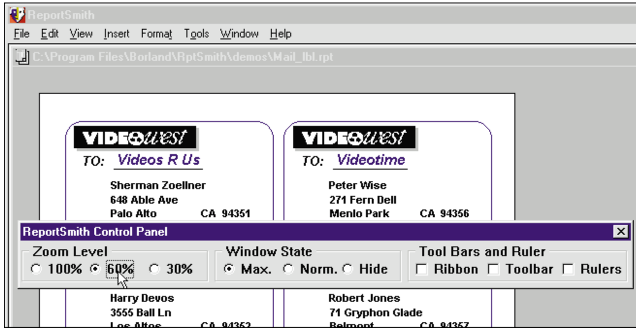


Figure 8 : RS_PANEL controlling ReportSmith's IDE.

```
procedure TRSCtrlPanel.RGWindowStateClick(Sender: TObject);
var
  TheMacro: array[0..79] of Char;
  WinState: string;
begin
  case RGWindowState.ItemIndex of
    0: WinState := '3';    { Maximized }
    1: WinState := '1';    { Normal }
    2: WinState := '0';    { Hidden }
  end;
  StrPCopy(TheMacro,'ShowRS(' + WinState + ')');
  RSWindowCtrl.OpenLink;
  RSWindowCtrl.ExecuteMacro(TheMacro, True);
end;
```

Figure 9: The procedure in RS_PANEL.

array of type *Char*. The command string itself consists of the text of the ReportSmith menu you want to execute. If you need to access a sub-menu, the command string will consist of the main menu's text, followed by the DOS pipe character (i.e. the vertical bar " | "), then followed by the text of the sub-menu. For example, the command to toggle the display status of the ReportSmith Toolbar would be `View|Toolbar`. Notice that no spaces are allowed.

The following code from RS_PANEL illustrates how the text of the currently selected radio button in the RGZoomLevel RadioGroup component is used to drive ReportSmith. *StrPCopy* is used to place the string `View|Zoom(xx)` into an array named TheMacro. The actual value of `xx` is taken directly from the text of the currently selected radio button. Thus, this code is generic, and new values can be added to the *TRadioGroup* without having to change this code:

```
procedure TRSCtrlPanel.RGZoomLevelClick(Sender: TObject);
var
  TheMacro: array[0..79] of Char;
begin
  StrPCopy(TheMacro,'View|Zoom(' +
           RGZoomLevel.Items[RGZoomLevel.ItemIndex]+')');
  RSZoomCtrl.OpenLink;
  RSZoomCtrl.ExecuteMacro(TheMacro, True);
end;
```

Note that this code is using a manual DDE connection. Opening and closing the DDE link will only let you change ReportSmith once per execution. To provide repeatable control of ReportSmith, you need to leave the DDE link open till the Delphi form is closed.

The second issue with using the System topic is that the command string must use the *exact* wording of the target menu item. This becomes an issue in 16- and 32-bit compatibility with some menu items. If the text is different between ReportSmith 2.5 and 3.0, you'll need to add conditional code to handle this.

The *FormActivate* procedure in RS_PANEL contains this kind of conditional code. The menu command to toggle viewing of the ruler is `Rulers` in ReportSmith 3.0, but the 2.5 version uses the singular `Ruler`. *FormActivate* uses a standard Delphi compiler directive to correct the application on start-up of the 16-bit version:

```
procedure TRSCtrlPanel.FormActivate(Sender: TObject);
begin
{$IFNDEF WIN32}
  { 16-bit ReportSmith v2.5's View menu selection uses the
    word "Ruler", not "Rulers" as does the 32-bit version }
  CbxRuler.Caption := 'Ruler';
{$ENDIF}
end;
```

Using the Command *DDETopic* to "Drive by Function." The other *DDETopic* you will likely use is the Command topic. This directly accesses internal ReportSmith external functions. The procedure in RS_PANEL illustrates how the Command topic can be used to execute the *ShowRS* function to control the window state of ReportSmith (see Figure 9).

Notice that all components of a DDE conversation are text-based. Even the Windows API window state-setting index number must be passed to ReportSmith as a string.

Conclusion

ReportSmith is a very powerful reporting tool that can easily be integrated into your Delphi applications. Hopefully this series has shown you a few techniques you can use in your applications. ∆

Some of the material in this article is excerpted by permission from the author's chapter on ReportSmith in the book Delphi In Depth, *copyright 1996 by Osborne/McGraw-Hill. All code examples given here are found on the CD which accompanies the book.*

The demonstration project and report referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\SEP\DI9609MO.

Mark Ostroff has over 18 years experience in the computer industry. He began by programming minicomputer medical research data acquisition systems, device interfaces, and process control database systems in a variety of 3GL computer languages. He then moved to PCs using dBASE and Clipper to create systems for the US Navy, as well as for IBM's COS Division. He also volunteered to help create the original Paradox-based "InTouch System" for the Friends of the Vietnam Veterans' Memorial. Mark has worked for Borland for the past six years as a Systems Engineer, specializing in database applications.

DELPHI KIOSK

By Chip Overstreet

Three-Tier Delphi

Open Horizon's Connection and Application Broker Products

Three-tier architectures are characterized by the separation of the user interface, business logic, and data access logic. Many organizations are implementing three-tier architectures for enterprise applications to realize two key benefits.

First, organizations have spent a great deal of time and effort encoding their business rules. With two-tier architectures, these rules are embedded inside the client application, and are therefore unavailable for use (or reuse) by other applications. With a three-tier architecture, the business rules are located on a shared server, providing the potential for reuse.

Second, centrally locating business rules makes maintenance much easier and more cost effective. No longer does software need to be propagated out to every client workstation each time a business rule is changed.

A set of new products has emerged — known as "second-generation" client/server development tools — that can be used to build three-tier applications (see Figure 1). These include products such as Dynasty, Forte, Magna, and USoft Developer. While these tools offer the ability to build three-tier applications, most have inherent limitations that keep organizations from realizing the full benefits of this new architecture.

Figure 1: Three-tier architectures allow business rules to be partitioned onto multiple platforms.

There are three key limitations to these second-generation client/server tools.

Limited reusability. Most second-generation tools do not support true reusability. That is, you can build shareable business rules, but the only applications that can reuse these rules must also be written with the same tool. To deliver the promise of reuse, these technologies must be able to make their business rules available to other front-end products, such as Delphi Client/Server, Developer/2000, PowerBuilder, Visual Basic, or even Excel.

Maintenance limited to servers. While these tools make it possible to maintain business rules centrally, you must still maintain each client application that uses these rules. That is, whenever a business rule is changed or a new rule is added, each client application must be modified by adding the new calls and the interface information.

As companies rely on rapid change for competitive advantage, their systems must be capable of keeping pace. Therefore when business rules change, the modification should only be required in a single location.

Inability to leverage investments. Vendors of most second-generation tools offer a simple solution: Throw out your "Order Entry

Application" written in Delphi Client/Server and start from scratch using *our* tool.

For most organizations, significant investment has gone into their Delphi Client/Server applications, not only in the design, development, testing, and deployment, but also in training end-users. Most do not want to discard their current systems, and are looking for ways of leveraging their existing investments.

Open Horizon's Connection Application Broker

Connection is a connectivity solution designed to assist organizations in extending departmental client/server implementations to enterprise solutions.

Connection is comprised of client and server components that provide communications between each hardware platform (see Figure 2). In addition to the communications, *Connection* provides a set of *Enterprise Brokers* that enable access to services:

- *Connection Database Brokers* provide integration with databases (relational or non-relational)
- *Security Brokers* provide integration with security services for user authentication, single-sign-on, and data encryption.
- *Directory Brokers* provide integration with directory services for centralized management of resource information.
- *Application Brokers* provide integration with business rules, transactions, legacy applications, etc.

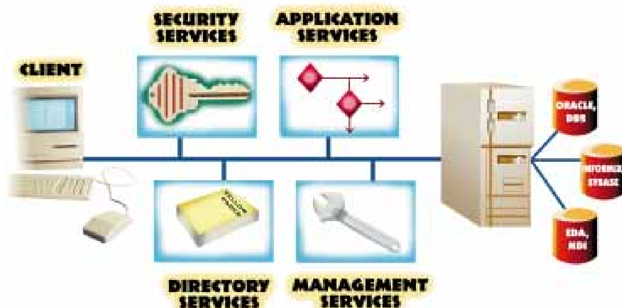


Figure 2: *Connection* provides new or existing client/server tools and applications with transparent integration with enterprise services.

Connection provides new or existing client/server tools and applications with transparent integration with enterprise services. Applications on the client — such as those written in Delphi Client/Server — simply "plug into" *Connection* and then gain immediate and transparent access to the services highlighted above. For example, *Connection* with a *Database Broker* can act as a plug-and-play replacement for products such as Oracle's SQL*Net.

This article discusses how *Connection* can be used to extend traditional "two-tier" tools such as Delphi Client/Server to support three-tier application architectures.

What Is a Broker?

When considering the finance market, a broker is someone who acts as an interface between two or more clients that want to buy and sell financial vehicles, such as stocks, bonds, or mutual funds. For example, to buy 100 shares of stock, you don't have to find a seller on your own. Simply call a broker, and they'll handle the request on your behalf. There is no need for you to know *how* the request is fulfilled. Additionally, a broker can offer advice in real time by informing you of changes in the market, as well as new options that may be available. Finally, your interface to the broker is simple — it's a standard telephone.

What Is the Application Broker?

Open Horizon's *Application Broker* performs a similar task to a financial broker. The *Application Broker* acts as an interface between client applications and server-resident business rules (see Figure 3).

Figure 3: The *Application Broker* acts as an interface between client applications and server-resident business rules, regardless of their implementation.

Client applications such as Delphi Client/Server, Developer/-2000, PowerBuilder, or Visual Basic issue a request to the *Application Broker*, which in turn invokes a business rule on behalf of the client. The business rules can be virtually any callable function — Pascal, C++, or COBOL programs, second-generation "generated" rules, transactions, CORBA-compliant distributed objects, legacy applications, etc.

The *Application Broker* is simply a broker of business rules.

How It Works

Using the *Application Broker* requires four steps: identification, specification, registration, and invocation. Each is described here:

Identification. The *Application Broker* is not a tool for developing business rules. Rather, it's a tool that provides simple access to existing business rules that you have either developed or that are already in use by other applications.

The first step is to identify a business rule that you want to be invoked from your client applications. For example, consider a simple Object Pascal function, called *AddNumbers*, that takes two integers as input and returns the sum of those integers:

```
function AddNumbers(Number1, Number2 : Integer) : Integer;
begin
  AddNumbers := (Number1 + Number2);
end;
```

Specification. The next step is to specify interface information about the rule. This information includes the application name (a mechanism for grouping related business rules), the business rule's name, the input and output parameters, and

optional text descriptions of each. You may also specify such things as error handling information and cursors for handling multiple row returns.

This specification is done using a high-level application interface language (AIL). Using the previous example for *AddNumbers*, the AIL would read:

```
application  mathFunctions  "Math Functions"
  procedure  addNumbers     "Adds Two Numbers"
  takes      int1 integer   "First Number"
  takes      int2 integer   "Second Number"
  returns    int3 integer   "Sum of Two Numbers"
```

This interface file provides the *Application Broker* with the information necessary to invoke the business rule.

Registration. After the AIL has been constructed (or generated using a GUI front-end tool), it's then passed through a compiler. The compiler accomplishes two tasks. First, it generates a C++ wrapper that is automatically linked to the business rule. Second, it registers the interface information inside the *Application Broker*.

The *Application Broker* can be considered as a meta-dictionary that contains the information necessary to locate and invoke one or more business rules.

Invocation. Once the business rule has been registered, it's then available to be invoked from any client application. To invoke the *AddNumbers* business rule from within Delphi Client/Server, the developer simply binds two integers and formats a string containing the call information. *Connection* uses stored-procedure-call semantics for specifying this information. With Delphi Client/Server, simply use the built-in functionality for generating a stored procedure call — there is no need to write C code, remote procedure calls, or even be aware of the network.

Communication from first-tier to second-tier. Under the covers, *Connection Client* uses RPCs to issue the request. Stored-procedure-call semantics are used because virtually every application and tool on the market has built-in support. Even off-the-shelf applications such as Microsoft Excel or WordPerfect can be used to invoke a business rule because they have the ability to issue a stored procedure call.

As a Delphi Client/Server developer, you do not even require knowledge of whether the business rule is implemented as a C function, a distributed transaction, a legacy COBOL application, or even a CORBA-compliant distributed object.

Dynamic Binding. Ongoing maintenance is one of the most time-consuming and costly tasks in client/server computing. Three-tier client/server architectures help to reduce maintenance costs because business rules are maintained in a single location on the server.

Nevertheless, three-tier architectures do not inherently address situations where new rules are introduced, or where interface information changes. When these changes occur, each client application using the rule must be identified and updated. One of the main reasons why CORBA and OLE are gaining popularity is that the discovery and binding of rules (distributed objects) is performed at run time. Applications require no prior knowledge of the rules or their interface information.

The *Application Broker* provides support for either a static or dynamic binding model. With the static model, you hard-code the name, and input and output parameters of the business rule into each client application.

With the dynamic binding model, all this information can be obtained at run time, thereby eliminating the need to modify every client application each time a change is made. The dynamic model also assists greatly in the development life cycle. The business rules developers can work independently building C functions, transactions, etc., and registering them within the *Application Broker*. The Delphi Client/Server developer can simply query the *Application Broker* at any time to get the latest view of available business rules.

Migrating Delphi Client/Server Applications to Support Three-Tier

Connection provides you with a simple mechanism for supporting three-tier architectures without the need to discard your Delphi Client/Server applications and starting from scratch.

As described earlier, *Connection* with a *Database Broker* can be used as a simple replacement for products such as SQL*Net from Oracle (although *Connection* has been architected to provide higher performance). With this solution, you can continue to run your Delphi Client/Server application as it currently works in a two-tier environment.

With the addition of the *Application Broker* onto the server, the infrastructure is in place to invoke business rules that reside on the server. The migration can be incremental. Start by identifying the business logic inside your Delphi Client/Server applications that you want to redeploy as a centralized, shareable business rule. Once the rule has been specified and registered within the *Application Broker*, simply replace the client-resident Delphi business logic with a stored procedure call.

Additional Services

This article has focused mainly on extending Delphi Client/Server applications to support three-tier architectures. However, there are additional requirements that typically surface when organizations begin to deploy client/server applications across the enterprise. One of these is support for enhanced security, with requirements for user authentication, single-sign-on, resource authorization, and data protection. Another is support for directory, or naming services, which enables location information about resources, including databases, to be stored centrally rather than within each application or on each client workstation (such as in an .INI file).

Connection provides customers with the ability to add *Security Brokers* and *Directory Brokers* to the base *Connection* product, thereby providing this support to Delphi Client/Server applications transparently.

Conclusion

Organizations are moving rapidly to embrace three-tier application architectures to obtain the benefits of reusability and simplified maintenance. However, many of these benefits are only partially realized with the tools on the market today.

As a Delphi Client/Server application developer, you now have the ability to leverage your favorite tool to support three-tier architectures without throwing out your existing investment. At the same time, you can obtain true reusability of business rules, as well as simplified maintenance on both the client and server. ∆

For more information on *Connection*, please contact Open Horizon by phone at (415) 598-1200, e-mail at info@openhorizon.com, or visit their Web site at http://www.openhorizon.com.

Chip Overstreet is the Vice President of Marketing and Business Development for Open Horizon. Prior to Open Horizon, he worked for Ochre Development, a second-generation client/server tools company, and Oracle Corporation in both the US and Australia.

AT YOUR FINGERTIPS

Delphi / Object Pascal

By *David Rippy*

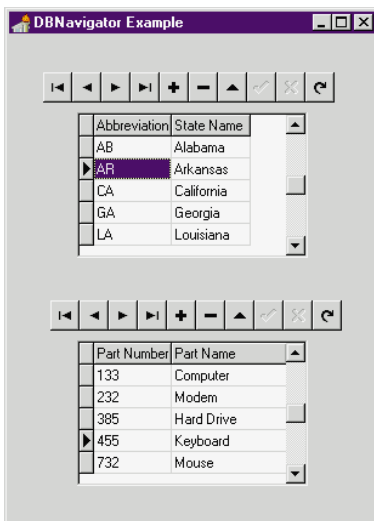
You are the product of your own brainstorm.

— Rosemary Konner Steinbaum

How can I use a single DBNavigator for multiple DBGrids on a form?

On a form containing two or more DBGrids, it's common for developers to use a separate DBNavigator component for navigating each DBGrid (see Figure 1). This works fine in most cases, but can become a problem if screen real estate is minimal. To solve this problem, use a single DBNavigator component that is smart enough to control each of the DBGrids on a form (see Figure 2).

Surprisingly, all that is needed is one line of code added to each of the DBGrids. With this line of code, the DBNavigator will always "know" which DBGrid should receive instructions (i.e. First Record, Next Record, Insert, Delete, etc.).



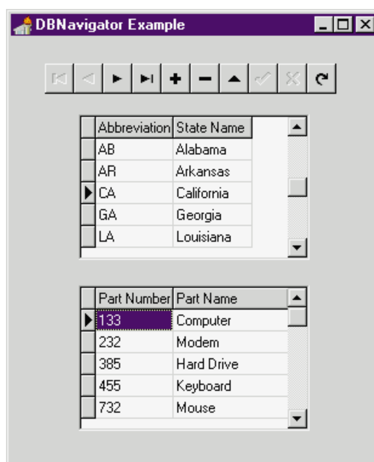
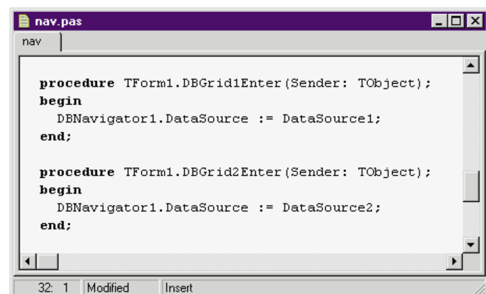


Figure 1 (Top): This form contains two DBNavigator components.
Figure 2 (Bottom): The DBNavigator in this form controls both DBGrids.

Examine the *OnEnter* event handlers for DBGrid1 and DBGrid2 in Figure 3. As you can see, whenever the user positions the cursor in DBGrid1 or DBGrid2, the *DataSource* property of DBNavigator1 is set to the same DataSource as the DBGrid with focus. This means you can freely switch between the two grids, and use DBNavigator1 for traversing both. — *D.R.*



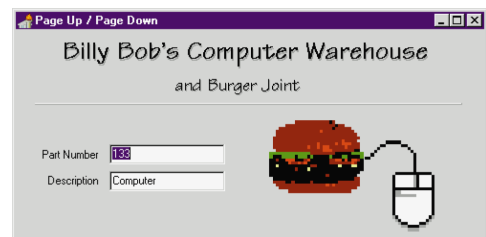
```
procedure TForm1.DBGrid1Enter(Sender: TObject);
begin
  DBNavigator1.DataSource := DataSource1;
end;

procedure TForm1.DBGrid2Enter(Sender: TObject);
begin
  DBNavigator1.DataSource := DataSource2;
end;
```

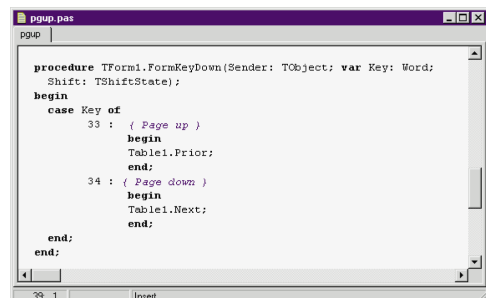
Figure 3: *OnEnter* event handlers for DBGrid1 and DBGrid2.

How can I use Page Up and PageDown to navigate records in a table?

For forms such as the one in Figure 4, it's nice to give users the ability to navigate records using key strokes in addition to the mouse. The default behavior of a form does not provide this, but it's a simple feature to implement.

In this example, the user can advance to the prior or next record in the table using Page Up and PageDown, respectively. All the code necessary for this function is located in the *KeyDown* event handler of *Form1*





```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  case Key of
        33 :  { Page up }
            begin
            Table1.Prior;
            end;
        34 : { Page down }
            begin
            Table1.Next;
            end;
  end;
end;
```

Figure 4 (Top): Page Up and PageDown will advance records forward and backward.
Figure 5 (Bottom): The *KeyDown* event handler for *Form1*.

(see Figure 5). In the event handler, a **case** statement checks the value of *Key* (a parameter passed in as part of the event handler). If the value of *Key* is *33* (VK_PRIOR), the user pressed Page Up, and the *Prior* method is called to advance to the prior record in the table. If the value of *Key* is *34* (VK_NEXT), the user pressed PageDown, and the table's *Next* method is invoked.

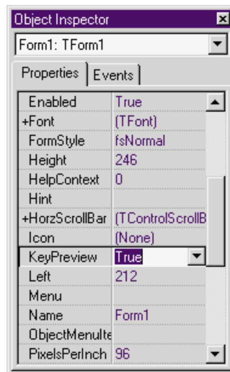


Figure 6: You must set the *KeyPreview* property to *True*.

The final — and most important — step in this tip is to set the *KeyPreview* property of the form to *True* (see Figure 6). *KeyPreview* passes most key events to the Form for processing, allowing the developer to insert key-handling code, such as we have done in this example. If *KeyPreview* is set to *False* (the default), the code in the form's *KeyDown* event handler will not execute.

TIP: You may be wondering how to determine the values for specific keys such as Page Up and PageDown. Add the following code to your form's *KeyDown* method (remembering to set the *KeyPreview* property to *True*):

```
ShowMessage(IntToStr(Key));
```

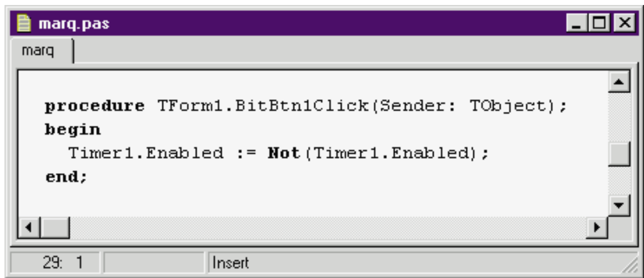
Now, when you press a key, its corresponding *Key* value will be displayed in a dialog box.

Note that I am using the actual numbers here, but you may want to use the virtual key constants (i.e. VK_PRIOR and VK_NEXT) which can be found in the WinTypes.PAS (Delphi 1) or Windows.PAS (Delphi 2) files located in the \SOURCE\RTL\WIN directories. Using these constants will help keep your code much more readable. — *D.R.*

How can I create marquee-style text on my form?

Creating marquee-style text is a unique way to jazz up your forms, and requires minimal effort to implement. First, drop a *TLabel* component onto your form, such as the one shown in Figure 7. You can apply any color, size, and font combination. Next, set the *Caption* property of *Label1* to the message you want to scroll. For best results, add a few extra spaces to the end of the message.

Figure 7: The *TLabel* component for the marquee-style text example.



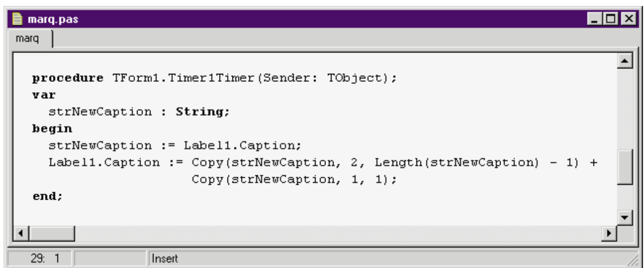


Figure 8 (Top): This code will enable and disable the *TTimer*.
Figure 9 (Bottom): *TTimer1*'s *OnTimer* event handler.

You also need to place a *TTimer* component on the form. Set the *Interval* property of the *TTimer* to *500* (milliseconds). The *Interval* property determines the rate that your message scrolls — the smaller the value, the faster the scrolling. A value of 500 will update the message twice a second. The button on the sample form toggles the *Enabled* property of the *TTimer* to start and stop the scrolling (see Figure 8).

The *OnTimer* event handler for the *TTimer* contains all of the string manipulation necessary to give the illusion of marquee-style text (see Figure 9). In a nutshell, we are simply taking the first character of *Label1*'s *Caption* and moving it to the end of the line. That's it! Be sure to experiment with different colors and typefaces. — *D.R.*

Quick Tip: Using Application.ShowHint

Most applications these days use Hints to indicate the function or purpose of a given control. Did you know you can enable or disable all the Hints in your application with only one line of code?

```
{ Set to False to disable all hints }
Application.ShowHint := True;
```

— *D.R.* Δ

The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\SEP\DI9609DR.

David Rippy is a Senior Consultant with Ensemble Corporation, specializing in the design and deployment of client/server database applications. He has contributed to several books published by QUE. David can be reached on CompuServe at 74444,415.

NEW & USED

By Bill Todd

ReportPrinter Pro 2.0

Nevrona Designs' Upgraded Report Writer

If you read the review of ReportPrinter 1.1 in the December 1995 issue of *Delphi Informant* and decided you didn't want to use a code-based reporting tool, it's time to look again. When it first appeared on the Delphi scene, ReportPrinter 1.1 from Nevrona Designs broke a lot of new ground. With version 2.0, Nevrona has taken the product a step further.

ReportPrinter 1.1 was the first report writer, that I am aware of, built entirely from Delphi components, and capable of producing the entire reporting system in the executable file. Because it is not a visual band-oriented tool, ReportPrinter also broke tradition in the world of reporting tools by allowing you to create reports with code. Although creating reports with code is more time consuming in many cases, this functionality gives you the power and flexibility to create unique reports.

Using ReportPrinter Pro 2.0, you can now build on this powerful foundation to bring together the best of both worlds. If you need to create reports entirely in code, you still can. For example, to create a report where the layout of each page is different, using code lets you do it. You can still print anything, including graphics and memos, in any order, anywhere on a page, just as you could in ReportPrinter 1.1.

Figure 1: This multilevel report was produced with just two lines of code.

Improvements

ReportPrinter Pro 2.0 brings forth an entire suite of report shell components to make creating most report formats easy with just a few lines of code. For example, Figure 1 shows a one-to-many-to-many report using the sample Customer, Orders, and Items tables that ship with Delphi. This report includes headings at each level, and a total quantity in the footing for the Items table. To create this report, you need to write only two lines of code.

If you need to make labels, take a look at the *TLabelShell* component in Figure 2. *TLabelShell* makes labels a snap because it includes over 150 pre-defined Avery label types. If you are using a custom label, simply enter the dimensions, number down and across, and you're ready to go. This report required 33 lines of code, most of it suppressing blank lines.

The *TDBTablePrinter* component's Table Editor is another feature that makes creating columnar reports much easier in ver-

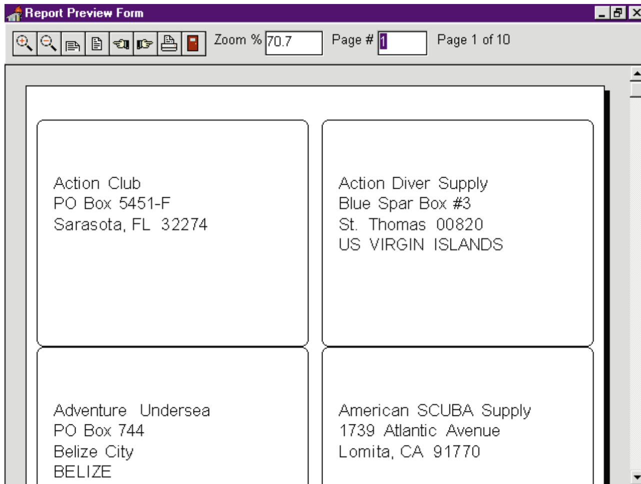


Figure 2: Labels printed with the new *TLabelShell* component.

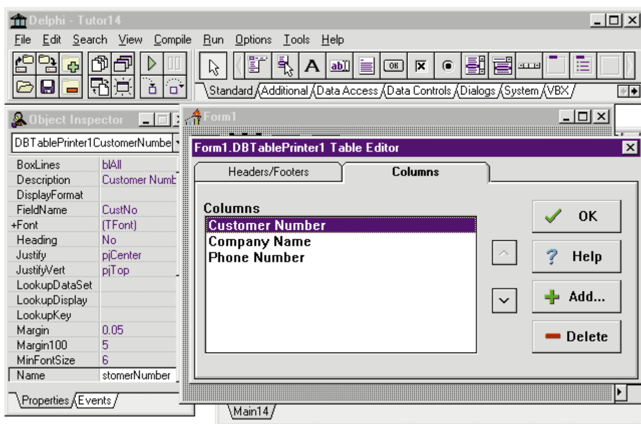


Figure 3: The *TDBTablePrinter* component's Table Editor.

sion 2.0 (see Figure 3). The Columns tab of the Table Editor lets you quickly add columns to a report, rearrange the columns, and set their properties. The Table Editor also lets you set the properties of the various headers and footers in the report, and determine which ones will be active in each report.

When creating a report with code, the new *TReportSystem* component is a true time saver. It combines all the functionality of the four components from version 1.1, enabling you to view the report as it is created in the screen previewer.

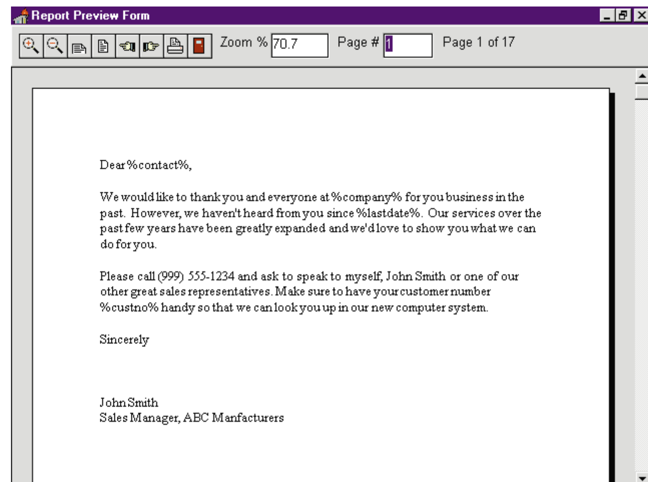
To see descriptions of the new components in ReportPrinter 2.0, take a look at the table in Figure 4. In addition to a rich set of properties that allow you to customize reports at design time, each new component has a complete set of events to customize each component's behavior.

Manipulating Text

In version 2.0, ReportPrinter has added some major memo printing capabilities. It still allows you to print any memo field anywhere on a report page, but now there is even more power. You can easily manipulate the text of memos before printing. Figure 5 shows a form letter stored in a text file

Component	Description
TReportSystem	Integrates the functionality of the *TReportPrinter*, *TReportFiler*, *TFilePrinter* and *TFilePreview* components from version 1.1.
TDetailShell	Prints detail sections in multi-level reports.
TMasterShell	Prints master or detail levels in a multilevel report. *TMasterShell* can be linked to other *TMasterShell*, or to *TDetailShell* components to create as many levels of detail as needed.
TReportShell	This is the top of the shell component hierarchy. It adds report header and footer, as well as page header and footer capabilities to the features provided by *TMasterShell*. A typical multilevel database report is built from one *TReportShell* component, and as many *TMasterShell* and *TDetailShell* components as needed.
TLabelShell	A specialized component for printing labels.
TDBTablePrinter	This component is designed to print tabular reports from database tables. You can link multiple *TDBTablePrinter* components together to print master detail reports to any level.
TTablePrinter	Identical to *TDBTablePrinter* except that it doesn't get its data from a database table. Instead, you supply the data in code.

Figure 4: A description of ReportPrinter 2.0's new components.



Dear %contact%,

We would like to thank you and everyone at %company% for you business in the past. However, we haven't heard from you since %lastdate%. Our services over the past few years have been greatly expanded and we'd love to show you what we can do for you.

Please call (999) 555-1234 and ask to speak to myself, John Smith or one of our other great sales representatives. Make sure to have your customer number %custno% handy so that we can look you up in our new computer system.

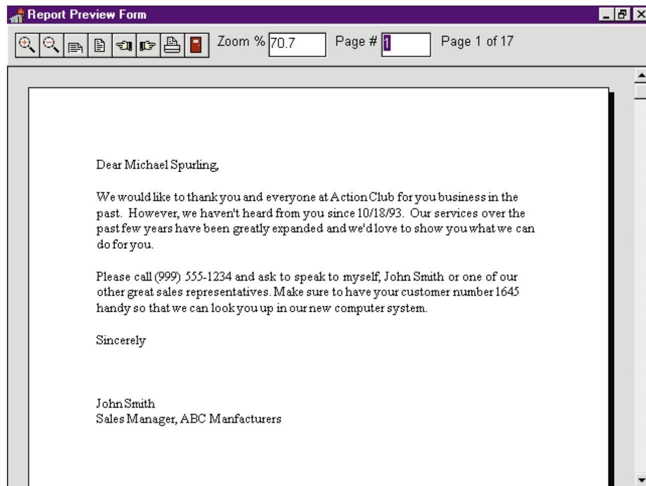
Sincerely

John Smith
Sales Manager, ABC Manfacturers

Figure 5: A form letter stored in a text file.

that contains embedded tokens (i.e. `%company%`) in the text. You can replace the tokens with other values — as the report is being printed — to produce the letter shown in Figure 6. Storing the text in an external file makes it simple for users to modify the body of the letter without changing the program. You can also store text containing tokens in memo fields in a table, and replace the tokens with data from another table, or with values entered by the user.

A tribute to its excellent design, the new ReportPrinter components were built from the original ReportPrinter 1.1 class library. This means all the programs using ReportPrinter 1.1 are completely compatible with the new version.


Report Preview Form

Zoom % 70.7 Page # 1 Page 1 of 17

Dear Michael Spurling,

We would like to thank you and everyone at ActionClub for you business in the past. However, we haven't heard from you since 10/18/93. Our services over the past few years have been greatly expanded and we'd love to show you what we can do for you.

Please call (999) 555-1234 and ask to speak to myself, John Smith or one of our other great sales representatives. Make sure to have your customer number 1645 handy so that we can look you up in our new computer system.

Sincerely

John Smith
Sales Manager, ABC Manfacturers

Figure 6: Here the tokens in Figure 5 have been replaced with their respective values.

Documentation Upgrade

On a final note, ReportPrinter 1.1's documentation was sorely limited, but that is no longer the case. ReportPrinter Pro 2.0 includes a professional caliber 196 page manual. It features 22 demonstration programs documented in 56 pages of tutorial text. This makes learning the product fun and easy, instead of slow and frustrating. ReportPrinter Pro 2.0 ships in both 16-and 32-bit versions, with full source code.

Conclusion

If you want to use only one report writer for all your Delphi programs, this is it. ReportPrinter 2.0 allows you to compile a program for easy distribution; create standard reports, graphics, and memos and form letters with embedded data quickly; modify page layout from page to page; and add precise positioning for pre-printed forms.

Whether you need all these features or just a few, ReportPrinter is a tool that can handle any reporting task. ∆

Bill Todd is President of The Database Group, Inc., a Phoenix area consulting and development company. He is co-author of *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, 1993]; a member of Team Borland; and a speaker at every Borland database conference. He can be reached at (602) 802-0178, or on CompuServe at 71333,2146.

INFORMANT FACT FILE

Nevrona Designs' ReportPrinter Pro 2.0 report writer allows you to produce reports compiled into a program for easy distribution, create standard reports, graphics, and memos and form letters with embedded data, and modify page layout from page to page. ReportPrinter Pro 2.0 also handles the task of precise positioning for pre-printed forms.

Nevrona Designs
1930 S. Alma School Rd., C-204
Mesa, AZ 85210-3043
Voice: (602) 491-5492
Fax: (602) 530-4823
E-Mail: Internet: tech@-nevrona.com or CIS: 70711,2020
Web Site: http://www.nevrona.-com/designs
Price: ReportPrinter Pro 2.0, US$149.00; upgrade from version 1.1, US$49.

New & Used

By Bill Todd

UDFlib

Adding Functionality to Your InterBase Applications

One of the shortcomings of InterBase is that it only includes eight functions you can use in your SQL queries. Fortunately, InterBase also supports user-defined functions. This allows you to write your own functions in the form of a C/C++ library or DLL, and attach the function library to any InterBase database.

Using the user-defined function feature of InterBase, MER Systems, Inc. has created UDFlib. This library consists of over 100 functions you can add to any InterBase application.

The UDFlib functions are grouped into eight categories:
1) C string functions
2) conversion functions
3) date/time functions
4) double precision functions
5) float functions
6) integer functions
7) SmallInt functions
8) VarChar functions

Figure 1 shows the functions available for both character and VarChar strings. When working with VarChars just add the letter "v" to the beginning of the function name. For example, to take a substring of a character string use the substring function. To extract a substring from a VarChar use vsubstring.

The list of functions for other categories is just as complete. The date/time category includes functions to subtract two dates, add a time to or subtract a time from a date, compute the Julian date, return the number of the week the date falls into, and many others.

Function	Description
ltrim	Removes leading blanks from a string
ltrimc	Removes leading characters from a string
rtrim	Removes trailing blanks from a string
rtrimc	Removes trailing characters from a string
alltrim	Removes blanks from both ends of a string
alltrimc	Removes a user-specified character from both ends of a string
center	Pads a string to the specified length and centers it by adding the specified character(s) to the beginning and end of the string
cstradd	Concatenates two strings
cstrdelete	Deletes a specified number of characters from any location in a string
cstr_plus_int	Converts an integer to a string and concatenates it to a string
int_plus_cstr	Converts an integer to a string and concatenates a string to it
lefts	Copies the left most N characters of a string
rights	Copies the right most N characters of a string
len	Returns the length of a string
lower	Converts a string to lower case
lpad	Pads a string to any length by adding the specified character to the beginning of the string
pad	Pads a string to any length by adding the specified character to the end of the string
parse	Returns the characters between any occurrence of the specified character
pos	Returns the position of one string within another
proper	Capitalizes the first letter of each word in the string
quarter	Given a date, returns a string that identifies the year and quarter ("96Q1")
replicate	Fills a string with N occurrences of the specified character
reverse	Reverses the characters in a string
substring	Returns any substring of a string

Figure 1: UDFlib string functions.

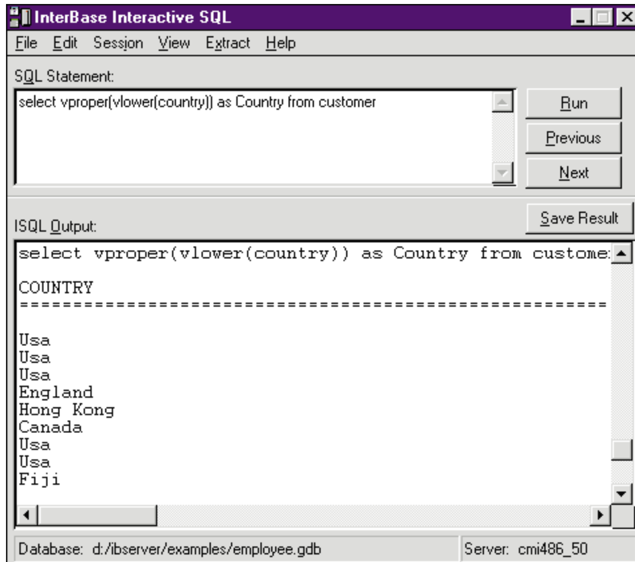


Figure 2: The SQL query.

Installing UDFlib

UDFlib, like all user-defined functions, must be installed in each database it's used. Using the script file, UDFLIB.SQL, that ships with the product simplifies the process. First, run the install program that places the four UDFlib files in the IBSERVER\BIN directory. Next, start the InterBase ISQL utility, connect to your database, and enter the command:

```
IN UDFLIB.SQL;
```

Exit from ISQL and you are done with the installation.

Using UDFlib

Using the functions in UDFlib is as simple as using the functions built into InterBase.

Simply include the functions in any SQL statement. For example, Figure 2 shows the SQL query:

```
select vproper(vlower(country)) as Country from customer
```

This uses the vlower function to force the value in the Country column of the Customer table to lower case, and the vproper function to change the first letter of each word to upper case.

About the only thing you need to be careful of is using the appropriate function for the type of column.

UDFlib includes a complete online Help file with a topic for every function. The Help topic includes not only the syntax for the function, but a stored procedure and a WISQL query example as well. Although InterBase is available on many platforms, the current version of UDFlib is available only for InterBase for Windows 95 and Windows NT.

Conclusion

If you develop client/server applications with InterBase for Windows 95 or Windows NT, UDFlib is a must-have product. It adds the functions you need for every task you may encounter in virtually any application.

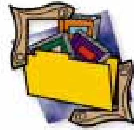
If you would like to try before you buy, download a demonstration version from either the BDEVTOOLS or DELPHI forums on CompuServe, or from MER Systems' Web site at http://www.mers.com. ∆

INFORMANT FACT FILE

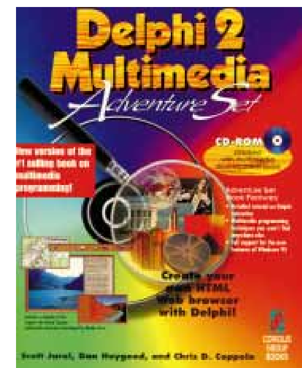
UDFlib is a library of over 100 functions that can be added to an InterBase application. The UDFlib functions are grouped into eight categories: C string functions, conversion functions, date/time functions, double precision functions, float functions, integer functions, SmallInt functions, and VarChar functions. UDFlib is ideal for the client/server developer working with InterBase for Windows 95 and Windows NT.

MER Systems, Inc.
58 Worthington Ave.
Richmond Hill, Ontario
Canada L4E 2S5
Phone: (416) 410-5166
Fax: (416) 410-5167
E-Mail: Internet: rschieck@mers.com
Web Site: http://www.mers.com
Price: US$250 per server

Bill Todd is President of The Database Group, Inc., a Phoenix area consulting and development company. He is co-author of *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]; a member of Team Borland; and a speaker at every Borland database conference. He can be reached at (602) 802-0178, or on CompuServe at 71333,2146.



Delphi 2 Multimedia Adventure Set



Delphi and Multimedia. I've waited a long time for a book that deals adequately with this topic — and I suspect I'm not alone. As multimedia PCs become the norm, we will be expected to incorporate more media in our applications. Let's face it, text and simple graphics don't cut it any more. As users of our applications see new Windows programs with sound, animation, and video clips, they will come to expect *all* applications to include at least some of these features. While Delphi and the Windows multimedia API provide most of the basic tools we need, until now there has been a paucity of detailed information on how to use these tools, particularly the more advanced ones. *Delphi 2 Multimedia Adventure Set* by Scott Jarol, Dan Haygood, and Chris D. Coppola finally fills the vacuum.

Many Delphi books explicitly indicate the level of experience the reader should have. While the introduction leaves you with the impression that no particular background is required, I feel some experience in Delphi and Object Pascal is necessary to benefit from this book. However, if you have worked your way through any of the introductory Delphi books, you shouldn't have any great difficulties. And you'll certainly learn a lot about writing Object Pascal code.

As you'll soon discover, the book's organization is both topical and progressive. Large sections deal with hypertext, graphic manipulation and animation, and sound. Some sections begin with introductory chapters explaining basic principles; most end with the creation of an impressive application.

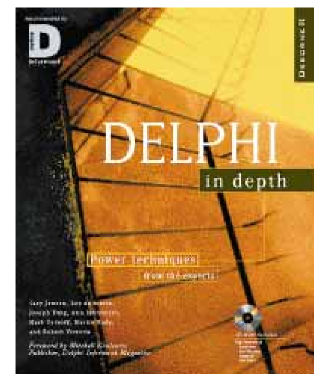
The book begins — where most general-purpose Delphi books conclude their discussion of multimedia — with Delphi's MediaPlayer control. The first topic is hypermedia. Dan Haygood introduces hypermedia and the Hypertext Markup Language (HTML). He then provides an example of a hypermedia engine to create an interface between Delphi and HTML. His code provides excellent examples of techniques for parsing, formatting, and displaying text. He also provides an excellent introduction to the Windows 95 Multimedia System. He concentrates on a discussion of sound and the use of .WAV files, and adds multimedia capabilities to the hypermedia engine.

The second section, written by Chris D. Coppola, explains graphical imaging. Again, the coverage is excellent. The text and code examples are easy to follow, and the topics introduced — animation techniques, dissolving images, and other special visual effects — should be of interest to many programmers.

In the next section, Haygood explains the art of hyperimaging, or creating hotspots on the screen. Haygood expands on earlier projects and concepts, and introduces more advanced topics. First, Haygood expands the hypermedia engine to become a functional Web browser. Then Coppola introduces the art of animation delineating, including topics such as

"Delphi 2 Multimedia Adventure Set"
continued on page 49

Delphi In Depth: A Heavyweight Winner



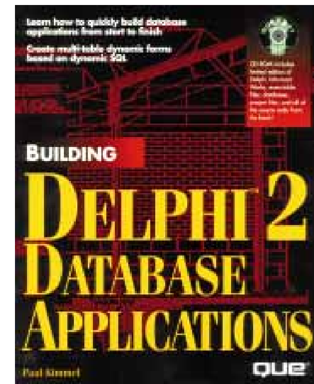
Delphi In Depth touts itself as "the Heavyweight of All Delphi Guides." In this age of exaggerated claims, one might initially dismiss such a statement. Nevertheless, after spending many hours with this book and its accompanying CD-ROM, my opinion is that this self-assessment seems justified.

Unlike many early Delphi books, *Delphi In Depth* doesn't simply paraphrase and expand Borland's documentation. Rather, this book follows the new trend, setting the standard for multi-topic or general Delphi resources. Written by an impressive collection of authors headed by Cary Jensen and Loy Anderson, it addresses many cutting-edge topics which should interest most Delphi programmers. While it's not the longest book available, it is filled with useful tips, insightful advice, and solid programming examples. So let's take an in-depth look.

The publisher claims this book is targeted at every programming level. However, a slight caveat is in order. If you're new to programming, and to Delphi, you may have difficulty using this book as your introduction. However, the opening chapters provide an excellent introduction to Delphi for programmers who have worked in other languages. The focus — as it should be — is on Delphi's unique qualities, including its seamless incorporation of database capabilities. Someone moving to Delphi from C++ or Visual Basic — let alone traditional Pascal — hardly needs a hundred or so pages devoted to Object Pascal basics. But such a developer must understand Delphi's use of properties and events, its exception-handling capabilities, as well as its approach to Object Oriented Programming (OOP). These topics, and the essential features of the Visual Controls Library (VCL) and Integrated Development Environment (IDE), are covered thoroughly, albeit succinctly. One useful discussion is of creating and using new Delphi classes, and

"Delphi In Depth: A Heavyweight Winner"
continued on page 50

Building Delphi 2 Database Applications Fails to Deliver



Paul Kimmel's *Building Delphi 2 Database Applications* approaches a subject users have been anticipating. Prior to getting this title, I had received requests to recommend a book specifically addressing Delphi's database aspects. At the time, I couldn't. Regrettably, this is still true.

I expected a comprehensive, focused presentation of database topics from this text. What I found instead was a routine introduction to Delphi, coupled with a disappointing treatment of its most elementary database operations. Even the length is disappointing — this one barely breaks 400 pages (thanks to its index). Brevity may be a virtue, but omission of essential topics is not — especially with a US$49.99 price tag.

The book is divided into four sections: Introduction; Database Management; Creating an Interface; and Designing Reports. The first and third sections cover the same subjects you'd find in any general Delphi text. The second section focuses on database issues, and the fourth covers ReportSmith.

The real problem lies in what is *not* covered.

The subject of client/server databases is dispensed within a couple of paragraphs, which simply state you need a more expensive version of Delphi, and "you *might* need a client/server platform to develop Oracle or Sybase database applications" (emphasis added). One sidebar begins, "While this book does not go into detail about specifics pertaining to client/server development, ..." *Detail? Specifics?* It gets nowhere near the subject.

One other disclaimer occurs while dismissing *TDatabase*, *TStoredProcedure*, and *TBatchMove* as irrelevant, except in client/server environments: "The puzzle of building client/server database applications has lots of pieces, many of which aren't covered in this book; some aren't covered in any book."

SQL is treated almost as poorly — only 13 incredibly shallow pages are devoted to the topic. After stating "The intent of this chapter is not to endorse the use of SQL specifically," Kimmel continues with, "This book would be incomplete, however, if SQL were left out completely." Considering the subject is databases, I should think so.

ODBC is covered in 15 pages, but only addresses how to perform one-time conversions from foreign table formats into dBASE or Paradox. The bulk of this chapter discusses converting an ASCII text file to Paradox, which hardly elevates ODBC to its highest purpose.

It's hard to imagine a database book that doesn't discuss searching, but *TTable*'s *GotoKey* and *GotoNearest* methods aren't even mentioned. In fact, the index contains no reference to any of the words "search," "locate," or "find," except in unrelated contexts. On the other hand, the section on Database Management begins by describing Database Desktop's Sort Table dialog box, one of the least useful options for a relational database.

Although the title references Delphi 2, the text doesn't distinguish between version 1 and 2 features, nor does it even mention there *are* two versions (or

"Building Delphi 2 Database Applications Fails to Deliver" *continued on page 50*

Delphi 2 Multimedia Adventure Set (cont.)

sprite and cell animation. Again, the treatment is thorough, and includes excellent code examples.

The final two chapters address audio topics. Coppola explains .WAV files, and introduces more sophisticated techniques (beyond high-level functions such as *sndPlaySound*). In the final chapter, Scott Jarol introduces the Musical Instrument Digital Interface (MIDI), which opens up a whole new world of audio possibilities. For me, this final chapter was the most interesting and helpful (as I am a programmer/musician), but also the most disappointing. I loved what Jarol presented — I just wish he had taken it further. After all, MIDI is probably the least explored of all the Delphi multimedia topics. However, Jarol's explanation of some of the low-level MIDI functions, and particularly the use of Windows callback functions and message callbacks, should prove invaluable to many aspiring multimedia developers.

Delphi 2 Multimedia comes with a well-designed CD-ROM, which includes all the book's code in two versions: in non-compiled form in the programs directory; and in compiled form (with .EXE files) in the projects directory. I hope this approach will become standard for programming books in the future. I found it particularly helpful to begin my exploration by simply running all the programs, without having to compile them in Delphi. The CD-ROM also includes useful shareware, utilities (including the Games SDK), and demonstrations. And there's no superfluous junk!

The only downfall of this book is the scattered references to Visual Basic, apparently carried over from a previous series of multimedia books published by the Coriolis Group. As you read through the text, you'll be amused (or aggravated) that references to Visual Basic remain.

I highly recommend *Delphi 2 Multimedia Adventure Set* for any programmer intent on incorporating multimedia into their applications. This book is clear, comprehensive, and interesting.

— *Alan C. Moore, Ph.D.*

Delphi 2 Multimedia Adventure Set by Scott Jarol, Dan Haygood, and Chris D. Coppola, Coriolis Group Books, 7339 E. Acoma Drive, Suite 7, Scottsdale, AZ 85260, (602) 483-0192, http://www.coriolis.com.

ISBN: 1-883577-64-0
Price: US$44.99
844 pages, CD-ROM

Building Delphi 2 Database Applications Fails to Deliver (cont.)

that Delphi 2 requires a 32-bit environment). A sentence tracing Delphi's history simply indicates Delphi 2 "was released within months after the first version rolled off the shelf and shortly after Windows 95 became available."

The contents reinforce the impression that the book was written for Delphi 1 and simply had a "2" grafted onto the title. The section on Designing Reports covers ReportSmith, but makes no mention of QuickReport.

In short, this book misses its target. The book is rated as suitable for accomplished or expert users; it decidedly is not. At best, it might serve as a low-level introduction to database development. However, many of the general books on Delphi offer as much (or more) information on database topics. Since those books also address the other aspects of Delphi, and cost no more than this work, there's simply no reason to settle for such a poor effort.

I am eager to see a good book on developing database applications in Delphi, but this is not the one. *Building Delphi 2 Database Applications* fails to deliver what its title promises.

— Larry Clark

Building Delphi 2 Database Applications by Paul Kimmel, QUE, 201 West 103rd Street, Indianapolis, IN 46290, (800) 428-5331.

ISBN: 0-7897-0492-7
Price: US$49.99
404 pages, CD-ROM

Delphi In Depth: A Heavyweight Winner (cont.)

transforming a class into a component. The authors then demonstrate how to make the new component installable in either Delphi version through the use of conditional compiler directives.

The introductory chapters offer much to the relatively new Delphi programmer. But what about experienced Delphi programmers? If you've been programming in Delphi for a while, you'll probably find much of the material in the first two-thirds of the book familiar. However, most chapters contain hidden gems which will be of interest to even the most experienced Delphi programmer. For example, following the introductory chapters is one dealing with writing property editors. I found this an absolute delight. While there is only basic information on writing custom components, this chapter rivals Ray Konopka's discussion of writing property editors in *Developing Custom Delphi Components* [Coriolis Group Books, 1996]. This alone should attract interest from experienced Delphi developers, as Borland's documentation on this is minimal.

The chapter on resource files was also a pleasant surprise. Although the discussion of working with .WAV files didn't initially impress me, I soon found something new — .WAV files can be included in resource files just like bitmaps, cursors, string tables, and so on, and can thus be embedded in the final executable (.EXE) file.

Midway through, *Delphi In Depth* provides an excellent introduction to database issues. All essential aspects for using database components, aliases, the Fields editor, and client/server techniques are included. There is also a tutorial on the available report-generating tools. This section explores more advanced topics, such as using Dynamic Data Exchange (DDE) in ReportSmith, as well as the latter's macro language, ReportBasic. Also included are two chapters on graphics — introductory and advanced — covering topics such as animation and preventing screen flicker.

For experienced developers, the final third of the book will be most useful. If you are moving to Delphi 2, the thoughtful discussion of multithreading will be invaluable. The authors present both the uses and potential pitfalls of using threads. The final section of the book provides a real-world programming example (a case study of the WebHub components) and explores new territory for many Delphi programmers.

For developers interested in the Internet and the World Wide Web, these last few chapters should provide an excellent starting place. The authors discuss both practical issues (such as accommodating Web surfers visiting your page) and programming issues (including integrating the new Web-specific languages, such as Java, with Delphi.) Needless to say, if you're going to program in Java, you'll need more than what's provided here. These chapters will, however, help you deal with some of the cross-language integration issues that arise when using Delphi to add power to Web applets.

But how does *Delphi In Depth* deal with the capabilities of, and differences between, the two versions of Delphi currently available? Very well! The authors are careful to clarify which approaches are available in each version. Where necessary, they provide techniques to accomplish the same task in both versions. The code on the CD-ROM is similarly organized. As a programmer who is still skipping between the two versions of Delphi, I appreciate the care the authors have taken in presenting things this way.

When I encounter a book with such a wealth of valuable information, my biggest complaint is: I want more! For example, I would have loved to see the authors devote a chapter or two to advanced component writing. However, I think the aim here is to serve the needs of the largest cross-section of programmers in the Delphi community.

I highly recommend *Delphi In Depth* for anyone who has experience in Delphi and wants to increase their expertise, as well as for any programmer migrating from Delphi 1 to Delphi 2.

— Alan C. Moore, Ph.D.

Delphi In Depth by Cary Jensen, Loy Anderson, Joseph Fung, Ann Lynnworth, Mark Ostroff, Martin Rudy, and Robert Vivrette. Osborne/McGraw-Hill, 2600 Tenth Street, Berkeley, CA 94710, (800) 722-4726.

ISBN: 0-07-882211-4
Price: US$42.95
812 pages, CD-ROM

Web-Enabling Delphi

If you or your company are not already starting to look at the Web as a development platform, I am betting you will soon. However, one of the hindrances to developing Web applications is you not only need to learn a new environment, but new tools as well. Fortunately for Delphi developers, we can apply our existing knowledge of Object Pascal and VCLs to build Web applications. This month, I'll address the "Web application framework," and how Delphi can be used within it.

Web Application Framework

If you are new to the Web, perhaps the notion of using it as a development environment seems perplexing. We are all probably familiar with the architecture of a typical Delphi client/server application: a Delphi .EXE running on a client PC connected to a database server on a LAN or WAN. However, because of the Web's distributed nature, a Web application is often composed of many different pieces, using a variety of technologies dispersed on the client and server.

Client Side

On the client side, the main component required is a Web browser. For the user, the browser serves as the "window" to the Web. For most Web applications, you'll want to use Netscape Navigator, Microsoft Internet Explorer, or another popular browser, but you may have occasion to embed a Web browser directly into your Delphi application. Delphi's new version 2.0 Internet Update contains an ActiveX control you can use for this purpose. Alternatively, you can use OLE Automation to take control of a browser to automate a task or perform a custom process (see Joseph Fung's article "Using OLE Automation to Access the Internet" in the May 1996 *Delphi Informant*).

However, as the Web has matured, increased demands have been placed on the client side to display multimedia and other MIME types, as well as to manage "executable content" within the browser. Rather than beefing up Web browsers into "mega-apps," the industry has chosen to handle these tasks using client extensions. The three most important client extensions include Java applets, ActiveX controls, and Netscape Plug-Ins. While Delphi cannot create Java applets, you can use it to create ActiveX controls and Netscape Plug-Ins.

Server Side

On the back end, the Web server handles requests for static HTML pages, but you'll need to extend the server to provide capabilities the server does not support by default. The two common methods of doing this are to use a CGI (Common Gateway Interface) or a native server API.

Since the Web began, CGI has been the *de facto* standard means of interfacing external programs with a Web server. Using CGI, you can execute a CGI script or program on the server to generate dynamic HTML for the client (see Keith Wood's article "An HTML Generator" in the May 1996 *Delphi Informant*). For example, when an HTML form is submitted, it can run a CGI program to process the form and return an HTML document to the client showing a result. You can use Delphi to create a CGI program that can be executed on any Web server running on a Windows platform. There are many CGI components available on the Web, such as one by HREF located at http://www.href.com. Also look for freeware and shareware components at http://www.delphi32.com.

A second approach to integrating with the Web server is through its native API. Two of the most popular are the Netscape Server API (NSAPI) and the Microsoft Internet Server API (ISAPI). The major advantage to using a server API is that they are much more efficient than CGI programs. CGI requires a separate instance of the program be executed for each client request, but a server API extension is loaded as a DLL. Consequently, CGI programs can eat more memory, and also limit the amount of data sharing that can be performed. On the other hand, the primary disadvantage to their use is that your solution is proprietary (specific to that server). A CGI program is much more flexible, and can be used on multiple Web servers. Delphi can be used to create NSAPI and ISAPI server DLLs. I have not yet seen a NSAPI component class available, but an ISAPI class we developed is available at http://www.acadians.com/delpisap.htm.

Hodge Podge of Tools

A Web application is often an assembly of pieces on the client and server sides that, when combined, form a complete application. Within such an environment, it is obvious that a hodge podge of tools are often required to pull off a Web-based application. If you can live with its major limitation — being tied to the Windows platform — Delphi serves as a viable solution for many of these Web development tasks.

— *Richard Wagner*

Are you using Delphi to build Web applications? If so, drop me a note at rwagner@acadians.com and let me know how you are using it.

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

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