



Cover Art By: Darryl Dennis

ON THE COVER



7 Delphi 4 Multi-tier Techniques — Bill Todd

The Delphi 4 *TProvider* component has a new property that allows you to enforce business rules by blocking inserts, deletes, or posts with an exception, just as you would in a two-tier application. Thus, multi-tier development has never been easier, as Mr Todd demonstrates with six example projects.



14 Multi-tier Database Apps: Part I — Thomas J. Theobald

Mr Theobald explains *n*-tier development from square one, and demonstrates the concepts with an example application portable between two back-end data sources. The demonstration application features four tiers responsible for user interface, business logic, data access, and data storage.



FEATURES

19 Informant Spotlight

MTS Development: Part II — Paul M. Fairhurst

Continuing his three-part series on Microsoft Transaction Server, Mr Fairhurst steps through a demonstration application that implements a three-tier database banking system.



26 DBNavigator

Delphi Database Development: Part V — Cary Jensen, Ph.D.

Backing the discussion with time trials as an example project, Dr Jensen continues his database series by explaining the essential database tasks of navigation and editing.



31 Algorithms

Tree Management — Rod Stephens

From basic terminology to performance issues, Mr Stephens describes tree structures algorithms — and supplies Object Pascal implementations for building them and putting them to use.



37 Delphi Reports Generic Reports — Keith Wood

Taking QuickReport to a new level of reusability, Mr Wood shares techniques for creating generalized, polymorphic reports. Ever used an object procedure?



42 At Your Fingertips

A Quick Spin on NT — Robert Vivrette

Delphi Informant's Technical Editor, Robert Vivrette, shares a number of quick programming tips, from manipulating graphics on Windows NT, to dealing with *GetLastError*, to better string-to-integer conversion.



REVIEWS

46 TSQLBuilder

Product Review by Ron Loewy

48 CodeSite 1.1

Product Review by Alan C. Moore, Ph.D.

DEPARTMENTS

2 Symposium by Jerry Coffey

3 Delphi Tools

5 Newslines

52 File | New by Alan C. Moore, Ph.D.



Simply the Best

As usual, Alan Moore's "File | New" column graces the last page of this magazine. And, as usual, Alan has written a piece of intelligent, engaging commentary. In fact, this one really got me going. I don't disagree with anything Alan says (also as usual), but I would like to expand on some of his points, and offer another perspective. In his column, Alan sings the praises of Delphi 4, but he also chides Inprise — gently, mind you — about some bugs in the initial version of Delphi 4, and the decision to perhaps ship it too early.

Despite the reasonable stance of the article and Alan's well-known affection for Delphi, I fear some of you will zero in on the word "bug" and come away with the impression that we're being unreasonably critical of Inprise and/or Delphi. It also gives me an opportunity to say some things about Delphi, and its place in the software market — and in software history.

Like many of you, I've worked with a lot of programming languages and IDEs in my 18-odd years as a software developer. Delphi is easily the best I've ever worked with. Every other tool — especially when compared to Delphi — suffers from various pitfalls, including, but not limited to: bad or meager documentation, limited feature set, usability lapses, a profusion of bugs large and small, plain flaky behavior, annoying antics to work around (or simply endure), and — last but not least — *the wall*, i.e. things you just can't do.

Alan mentions that some Delphi developers were unhappy that Delphi 4 shipped without most of the hard docs that accompanied previous versions. I'm sorry, but that's the state of the industry; it's just become too expensive to produce the manuals, package and ship them, and remain competitive on price. Another factor is that online help has become so good that the demand for hard docs has

dropped precipitously. They're essentially superfluous; I can see my Delphi 3 docs, still in their shrink-wrap, from where I sit. The context-sensitive help is *that* good. And even if you take the time to look up something in the paper manual, you'll find the same information as online.

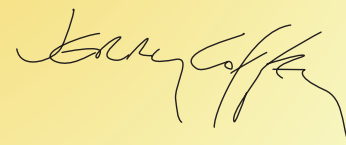
There are also developers out there — I see them in threaded discussions and the like — who deride Delphi's documentation, hard or online. Frankly, I find this irritating; anyone who thinks the Delphi docs are poor has never done serious work with another tool. Inprise's Delphi docs are in a class by themselves; no other software company produces anything close to them in quality or quantity. The one time Inprise (then Borland) clearly stumbled in this respect was with the initial release of Delphi 3. Its online help was essentially broken. Fortunately, an in-line release took care of the problem and brought it back to world-class standards.

Which brings me to maintenance releases. Far more important than the fact that Delphi 4 shipped with some minor bugs, is that Inprise quickly offered maintenance releases to correct the problems. As I write this, the Delphi 4 Update Pack #2 is available at <http://www.inprise.com/devsupport/delphi/downloads>.

(Updated releases of Delphi online help are also there, as are technical papers, FAQs, etc.) Too few developers take advantage of these maintenance releases; I still hear some complaining of Delphi 3's broken help. At this late date, it's the developer's fault if he or she doesn't have the in-lines. Although Inprise has an excellent track record in this regard, all software has bugs, and it's the developer's responsibility to keep up with maintenance releases.

In some respects, Delphi is its own worst enemy; it's so good that the bar has been set very high. Its developer community is holding Delphi to an exalted standard. And that's fine; it's reasonable to expect excellence when it comes to Delphi, and the Inprise R&D team that keeps bringing us spectacular new versions. It's also important to realize you're working with the best software development tool in history.

Thanks for reading.



Jerry Coffey, Editor-in-Chief

Internet: jcoffey@informant.com
Snail: 10519 E. Stockton Blvd.,
Suite 100, Elk Grove, CA 95624

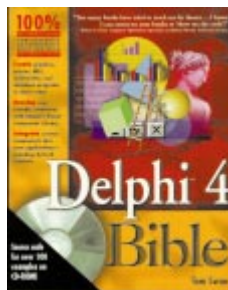


New Products
and Solutions



Delphi 4 Bible

Tom Swan
IDG Books Worldwide

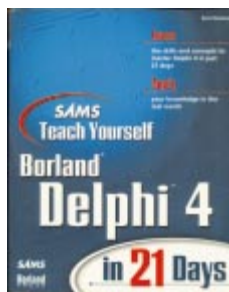


ISBN: 0-7645-3237-5
Price: US\$49.99
(953 pages, CD-ROM)
Phone: (800) 762-4974

SAMS Teach Yourself Borland

Delphi 4 in 21 Days

Kent Reisdorph
SAMS Publishing



ISBN: 0-672-31286-7
Price: US\$39.99 (918 pages)
Phone: (800) 428-5331

Tamarack Announces Rubicon 2.0

Tamarack Associates announced *Rubicon 2.0*, a full-text search engine for all versions of Borland Delphi and C++Builder. Rubicon 2.0 performs full-text searches by indexing all the words in a database or set of documents. The user is then able to perform searches by typing in words or phrases. Rubicon supports And, Or, Near, Not, and Like search logic and wildcards. Searches may be iteratively

narrowed or widened. Search results may be used to filter the search table, navigate to matching records, or create a match or answer table in natural or rank order.

As a database engine, Rubicon can index records stored in table formats supported by the BDE, Advantage Database Server, Apollo, DBISAM, FlashFiler, InterBase Objects, and Titan Access.



Tamarack Associates

Price: US\$299 (includes free 2.xx updates and support)

Phone: (650) 322-2827

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

Inner Media Releases Active Delivery 1.2

Inner Media, Inc. announced *Active Delivery 1.2*, a developer toolkit for creating self-extracting zip files. Active Delivery (AD) enables Web sites to package custom data on demand for delivery over Internet/-intranet connections.

Version 1.2 offers the ability to create and modify Startup menu items; safely replace system and other files that are presently in use; create multi-volume or "spanning" disk sets; make

use of "helper" applications to expand the capability of AD packages; as well as support for Active Server Pages and ISAPI DLLs.

Using any of the provided interfaces, developers can call into AD and tell it whether to create a true 16- or 32-bit executable, what files to add, how the Package is to appear to the end user, what external programs to run, etc. Developers can then ship the AD libraries royalty-free

with any number of products. Code samples are provided for Delphi, C/C++, Visual Basic, Microsoft Visual FoxPro, and Microsoft Access.

Inner Media, Inc.

Price: US\$249; Active Delivery Pro (includes DynaZIP-32), US\$384; current owners of Active Delivery 1.0 may purchase update for US\$69.

Phone: (800) 962-2949

Web Site: <http://www.active-delivery.com>

MathTools Announces MATCOM 4

MathTools Ltd. announced the release of *MATCOM 4*, a compiler for MATLAB capable of compiling MATLAB 4.x

and 5.x code into stand-alone applications, MEX files, or DLLs for Borland Delphi, Microsoft Excel, and Microsoft Visual Basic.

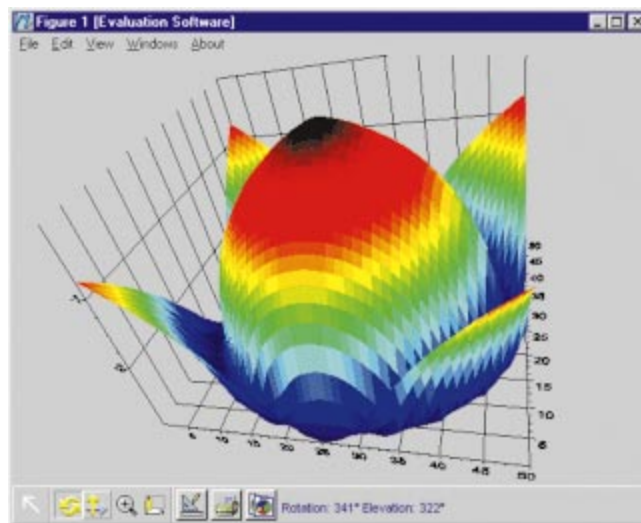
This version of MATCOM features sparse matrices, multi-dimensional matrices, structure type manager, structure arrays, improved compatibility with MATLAB 5, imaging support, exact report of error location, and a built-in accelerator.

MathTools Ltd.

Price: Standard version, US\$699 (commercial license) and US\$199 (academic license); Professional version (includes advanced graphics option and creation of DLLs for Visual Basic/Excel/Delphi), US\$999 (commercial license) and US\$299 (academic license).

Phone: (212) 208-4476

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





SkyLine Ships ImageLib Corporate Suite 4.0

SkyLine Tools announced ImageLib Corporate Suite 4.0, which enables developers using Delphi 3 and 4, Borland C++ Builder 3.0, and Microsoft Visual C++ to incorporate image and multimedia development into their applications. This version offers six additional SnapOn Toolbars, updated Open and Save functions, and improved DLL support. Corporate Suite 4.0 also includes BLOB updates for six file formats, scrollbar width flexibility, thumbnail preview refinements, improved Help files, and internationalization support. Also included are Scale-to-Gray with seven options, a thumbnail manager, Pixel free, and Deskew. The Suite includes ImageLib WebKit, which features support for Progressive Display GIF, PNG, JPEG, Interlaced, Transparent, and Animated GIFs. For more information, call (818) 346-4200, or visit the ImageLib Web site at <http://www.imagelib.com>.

MKO Announces MK QueryBuilder

MK Organisation announced *MK QueryBuilder*, a query tool that extracts and prints data from popular databases. MK QueryBuilder is available for Delphi 3, and comprises three components written in Delphi (source code is provided).

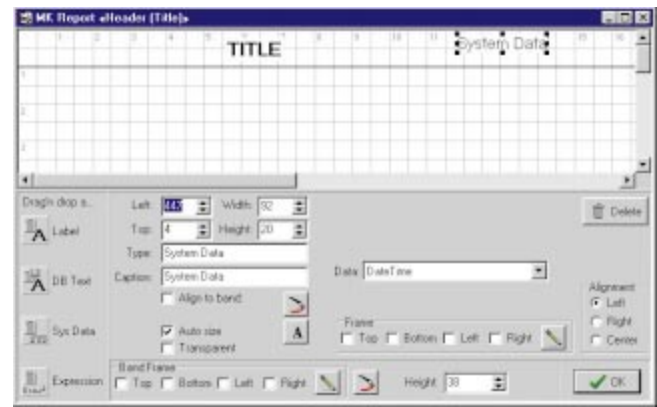
Included are three components: the MKQB component, which gathers Datamodel Designer and the QueryBuilder itself; MKXFER, which allows data exports to various file formats; and MKREPORT, a report designer for the end user (based on QuickReports).

MK QueryBuilder allows developers to graphically design logical data models

Grebar Systems Releases PrintDAT!

Grebar Systems Inc. released *PrintDAT!*, a VCL report component that allows developers to print *TDBGrid*, *TStringGrid*, *TTable*, *TQuery*, and *TDecisionGrid* objects. *PrintDAT!* works like a dialog box component. Reports are compiled into a developer's program or it can be run from within the Delphi IDE.

PrintDAT! can print grids with over 1,000 columns across, using horizontal page breaks, and has run-time options to handle Shrink to Page and snaking newspaper



panels. It auto-sizes report columns according to underlying data and adjusts reports to fit the size and orientation of the paper. Reports can be sent to the printer, a text file, the built-in report viewer, or the Windows Clipboard, or the report can be exported to a spreadsheet using an ASCII-delimited file format.

Grebar Systems Inc.

Price: US\$49; source code is available for an additional US\$99.
Phone: (204) 942-3301
Web Site: <http://www.grebarsys.com>

(links between tables located in different DBMSs, such as ORACLE and SQL SERVER).

MK Organisation

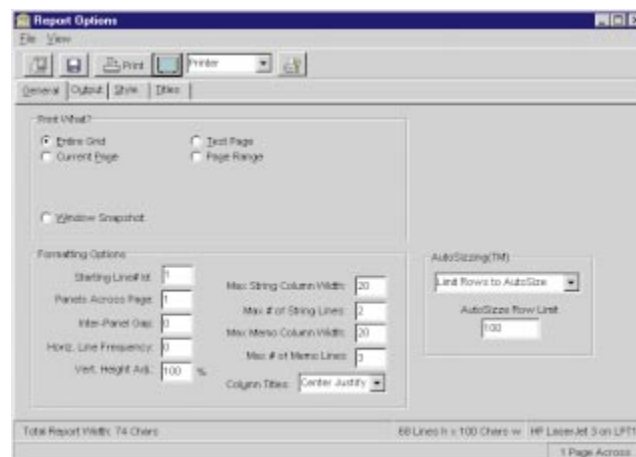
Price: Contact MKO for pricing.
Phone: (33 0) 1 43 58 61 61
Web Site: http://www.mko.fr/html.us/index_us.htm

Adapta Ships AdaptAccounts 6.4

Adapta Software Inc. announced *AdaptAccounts 6.4* for Windows 95/98/NT, a Delphi 4-based version of the company's database accounting applications family. The new line continues the 4-User configuration option, as well as the Developer and Multi-Copy License options. The new version also incorporates InfoPower and ACE Reporter for Delphi 4. Adapta's modular applications include System Manager, General Ledger & Financial Reporter, Accounts Receivable, Accounts Payable, Inventory, Sales, Purchasing, Job Costing, Bill of Materials, and Payroll.

Adapta Software Inc.

Price: General Accounting pack, US\$1,695 for one-user system; Accounting/Distribution pack, from US\$2,995 for one-user system.
Phone: (250) 658-8484
Web Site: <http://www.adapta.com>



January 1999



Inprise Announces New Versions of JBuilder/400 and Delphi/400

Anaheim, CA — Inprise Corp. announced new versions of JBuilder/400 and Delphi/400, the company's Java- and Windows-based development tools for the IBM AS/400 series of business computer.

Inprise offers a family of interoperable AS/400 development tools for enterprises integrating IBM AS/400 systems with the emerging platforms made available by Microsoft

Windows and Sun Microsystems' Java.

Inprise's JBuilder/400 and Delphi/400 allow enterprises to protect and leverage their investments in AS/400 architecture, while they develop and deploy GUI Windows and Web-based applications.

For pricing or other additional information, call (831) 431-1064, or visit <http://www.inprise.com/inprise400>.

Apogee Builds Award-winning Software Application for Beloit

Marlboro, MA — Beloit Corp. received the top ranking for their Bids and Proposals System in *InfoWorld's* annual "InfoWorld 100" list of the most successful client/server application development projects in the world. The system was designed and developed by Apogee Information Systems using Delphi Client/Server.

Beloit is a manufacturer of paper pulp processing machines. The new Bids and Proposals System automates the engineering, ref-

erence selection, cost and price estimation, and text generation process for a new proposal. Proposal teams using the system can span all five of Beloit's divisions, including more than 20 office locations across the United States, South America, Europe, and Asia.

In August, 1998, Apogee won the 1998 Inprise Application of the Year award for its work with ITT Sheraton Corporation. The winning Sheraton application was also developed using Delphi.

Genesis Unlimited Acquires Web Solution Builder

Knoxville, TN — Genesis Unlimited, Inc. announced it has purchased (for an undisclosed amount) all rights to the Web Solution Builder from Shoreline Software. The product allows developers to leverage Delphi to create professional and secure Web-based applications. Genesis Unlimited will enhance and utilize the product for its customer application base.

The agreement is fully endorsed by both companies, and Shoreline

Software will continue to use Web Solution Builder for Web-based application development. Genesis Unlimited will now handle support for Web Solution Builder.

For details, visit Genesis Unlimited's Web site (<http://www.GenesisUnlimited.com>), or Phoenix Business Enterprises (<http://www.pbbe.com>), a wholly-owned subsidiary of Genesis Unlimited that will be handling the Web Solution Builder product.

AverStar Licenses JWatch Technology to Inprise

Burlington, MA — AverStar Inc. announced that Inprise Corp. will use AverStar's JWatch debugging technology for the Java platform in future versions of JBuilder, Inprise's rapid application development tool for Java.

AverStar created JWatch for the enterprise developer who needs to debug multiple concurrent processes distributed across the network. JWatch handles multiple processes, multiple platforms, distributed applications, sourceless applets, Remote Method Invocation, and servlet debugging.

For more information on AverStar or JWatch, visit the AverStar Web site at <http://www.averstar.com>.

Inprise Announces Support for Oracle8i in Enterprise Tools

San Francisco, CA — Inprise Corp. announced support for Oracle8i (Oracle Corp.'s database for Internet computing) in its Delphi and C++Builder tools. With the Oracle8i support, Delphi and C++Builder customers will experience higher productivity in building, deploying, and managing Internet applications.

Oracle8i is designed to be an Internet development and deployment platform. It enhances Oracle8's technology with features that make it easier to create robust and scalable Internet and enterprise intranet applications.

For more information on Oracle8i, visit the Oracle Corp. Web site at <http://www.oracle.com>.



Crystal Reports Update

The cover article of the *October, 1998 Delphi Informant*, “Crystal Reports: Interfacing with the Leading Report Writer,” contained some outdated information. *Delphi Informant* apologizes and offers these updates. We’d like to thank Frank Zimmerman of Seagate technical support for this information.

Contact info. The Seagate Software Web site is at <http://www.seagatesoftware.com>. Tech support is available by phone (604.669.8379), e-mail (support@webacd.seagatesoftware.com), the Web (<http://webacd.seagatesoftware.com>), and fax (604.681.7163). The latest updates to the VCL can be obtained at <http://www.seagatesoftware.com/crystalreports/updates>.

Tables object. The *OnLoadDataFiles* event was redundant with the release of the Crystal 5 VCL, which had a *RetrieveDataFiles* method. With the new 32-bit VCL, this is handled differently again. Crystal no longer uses a *StringList* to represent *DataFiles*. Each major classification in the report’s structure is represented by a subclass in the VCL. Hence, a *Tables* object is a subclass of *TCrpe*. So instead of using the *OnLoadDataFiles* event to fill the *DataFiles StringList*, or instead of using the *RetrieveDataFiles* method, a statement such as this is used:

```
Crpe1.Tables.Retrieve;
```

From there, the individual table names and paths can be accessed either by subscript:

```
Crpe1.Tables[0].Name := 'NewTable.DBF';
```

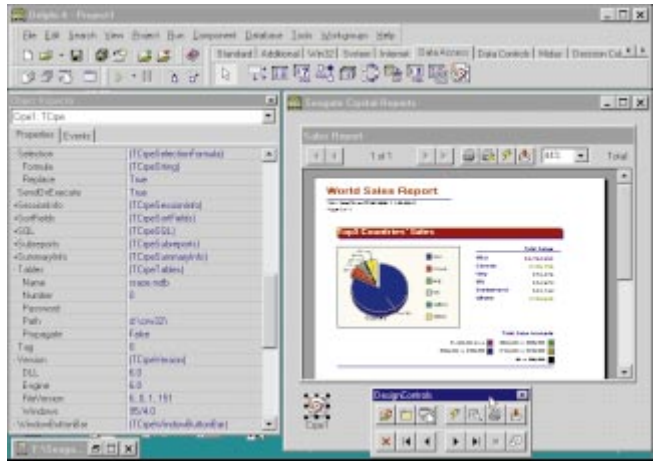
or in a loop using the *Count* method:

```
for cnt := 0 to Crpe1.Tables.Count - 1 do  
    Crpe1.Tables[cnt].Path := 'C:\Newdir\';
```

Export formats. Export formats for WordPerfect, Word for DOS, and Quattro Pro are still available, but only in 16-bit. There are no 32-bit DLLs to allow exporting to these formats from a 32-bit application.

Report format. The report format is not “bit dependant.” A report created in 16-bit Crystal will load without a problem into 32-bit Crystal, and vice versa. U2FCR.DLL is a 32-bit DLL as well. For 16-bit, it is UXFCR.DLL. U2FDOC.DLL and U2FQPDLL do not exist. As previously mentioned, these export formats are only available in 16-bit, the DLL names being UXFDOC.DLL and UXFQPDLL.

New 32-bit VCLs for Delphi 2, 3, and 4. As mentioned, the new VCLs implement extensive use of subclass objects for each report option. Hence, there is a *Tables* object that contains all the properties and methods that relate to handling database tables in a report. There is also a *SQL* object that contains all the properties and methods that relate to SQL query and stored procedure parameters, a *Formulas* object, and so forth. In all, there are 39 new subclass objects. These VCLs are available for free download at <ftp://ftp.img.seagate.com/pub/crystal/delphi>. The latest update to the 16-bit VCL (which does not contain the subclass features of the new 32-bit VCLs) is also available on the same FTP site.



Design time. The new VCL has Active design-time features. Shown here is a report that has been run from the Object Inspector onto a Delphi panel without writing a single line of code. If the Object Inspector is expanded to show all the properties it would take five sheets of 8 1/2 x 11 paper to print!

Feature summary. The new 32-bit VCL features:

- Delphi 2, 3, and 4 versions
- 39 new subclass objects for better organization
- 61 base class properties, 301 subclass properties
- 29 base class methods, 235 subclass methods
- Seven new events, including an *OnError* event for full control over error handling and messages
- 22 new Windows callback events for interacting with Preview Window events
- Full support for all Print Engine features, except the call to send a report an ADO data source
- Full subreport support, including launching of subreports as separate reports
- Full retrieve capability: retrieve all report values directly to the VCL
- Active design-time editing: retrieve report data, edit it, and launch reports without code
- Full support for all the new Crystal 6.0 features: Windows callback events, group options, report summary info, preview window buttons, window cursor shapes
- Visible print job number property
- BDE alias support: aliases can be used directly to change table path, or the path can be extracted from an alias for other purposes (such as setting the report directory)
- Crystal Reports parameter fields and stored procedure parameters can be passed as native Delphi types rather than as strings
- Full support for Area Format and Section Format formulas
- Crystal Reports Print Engine can be loaded and unloaded on-the-fly at run time
- SQL connection can be easily propagated to subreports
- Table location can be easily propagated to subreports
- Section naming convention now matches Crystal Reports short section name format
- Printer can be changed simply by specifying a new printer name
- New, extensive, context-sensitive Help with Delphi code examples for every property and method, plus introductory tutorial sections
- Full-featured sample application for demonstration and testing purposes, with links to the Help file



By Bill Todd



Delphi 4 Multi-tier Techniques

Row-level Business Rules, Real Transaction Processing, and More

One problem many developers encountered while building multi-tier applications with Delphi 3 was implementing business rules in the middle-tier application server. You could create an *OnUpdateData* event handler for the *TProvider* component and validate all the records in the update (delta) packet sent to the application server by a call to *ApplyUpdates* in the client. However, using *OnUpdateData* required you to accept or reject all the records as a set. The only alternative was to use the *TUpdateSQLProvider* component, which Inprise made available with the Delphi 3.02 release. *TUpdateSQLProvider* wasn't an official part of the VCL, but it added an *OnUpdateRecord* event to the events provided by *TProvider*, giving you record-by-record control of the update process.

Delphi 4 has solved this problem in a very different way. In Delphi 4, the *TProvider* component has a new property — *ResolveToDataSet*. *ResolveToDataSet* is *False* by default, which provides the same behavior as Delphi 3. When *ResolveToDataSet* is *True*, updates are applied using the dataset component the provider is connected to. This causes all the dataset's events to fire as though the changes were being made manually or in code. Now you can enforce business rules by blocking inserts, deletes, or posts with an exception, just as you would in a two-tier application.

The sample *EbSrvr* application that accompanies this article demonstrates this using the *BeforePost* event handler from the *OrderTable* object:

```
procedure TEbServer.OrderTableBeforePost
(DataSet: TDataSet);
begin
  with DataSet do
    if FieldByName('ShipDate').AsDateTime <
       FieldByName('SaleDate').AsDateTime
    then
      raise Exception.Create(
        'Ship date must be greater than
        sale date.');
```

This code raises an exception if the *ShipDate* is less than the *SaleDate*. In Delphi 3, raising an exception in the provider's *OnUpdateData* event handler caused an exception in the client application at the call to *ApplyUpdates*. However, raising an exception in one of the *Before* event handlers when *ResolveToDataSet* is *True* doesn't cause an exception in the client. Instead, the client's *ReconcileError* event is fired just as with any error generated by the VCL code in the server, or by the database server itself. This is a vast improvement because all errors, regardless of source, can be handled in the same way in the client.

The sample application uses the *Reconcile Error* dialog box from the *Object Repository* to handle all errors. If you try to post an order record whose *ShipDate* is less than the *SaleDate*, the error will appear in the *Reconcile Error* dialog box with the message text that you passed to the exception's constructor in the application server. The disadvantage of setting *ResolveToDataSet* to *True* is somewhat slower performance. (Note: *EbSrvr* and the

```

procedure TMainForm.JobCodeComboChange(Sender: TObject);
begin
  with MainDm.EmployeeCds do begin
    { If the parameters have not been fetched from the
      server, fetch them. }
    if Params.Count = 0 then FetchParams;
    { Set the parameter values. }
    Params.ParamByName('Job_Code').AsString :=
      JobCodeCombo.Text;
    Params.ParamByName('Job_Grade').AsInteger :=
      StrToInt(JobGradeEdit.Text);
    { If the Employee client dataset isn't active, open it.
      Opening the dataset sends the parameters to the
      server. If it's already open, call SendParams to send
      the new parameter values to the server and refresh
      the dataset. }
    if not Active then
      Open
    else begin
      SendParams;
      Refresh;
    end; // if
  end; // with

  if not MainDm.SalaryCds.Active then
    MainDm.SalaryCds.Open;
end;

```

Figure 1: Setting parameters in the application server's query.

other example projects discussed in this article are available for download; see end of article for details.)

Delphi 4 includes a new component — *TDataSetProvider* — that performs the same function as *TProvider*. However, it always applies updates to a dataset component in the middle-tier application. *TDataSetProvider* doesn't have the ability to apply updates directly to a database server by passing the dataset component. *TDataSetProvider* doesn't use the BDE, so it's the ideal choice if you're building a multi-tier application using a non-BDE database.

Controlling the Application Server

There are two ways for the client to control the application server in a multi-tier application: the first is by passing parameters to a query or stored procedure; the second is by executing custom methods on the server. Passing parameters to a *TQuery*, *TStoredProc*, or *TTable* in the application server was implemented in Delphi 3 using the *Provider* property of *TClientDataSet* to call the *Provider's* *SetParams* method passing as a parameter a variant array containing the parameters.

Delphi 4 provides a new way to do the same thing, as shown by the code in the QCLnt sample application (see [Figure 1](#)). This code is from the *JobCodeCombo* combo box's *OnChange* event handler. *TClientDataSet* now includes a *Params* property. You can fetch the parameters from the *TQuery* or *TStoredProc* component on the server at design time or run time. At design time, right-click the *TClientDataSet* and choose **Fetch Params**. At run time, call the *FetchParams* method.

The preceding code first checks the property *TClientDataSet.Params.Count* to see if any parameters have been fetched. If not, *FetchParams* is called. Once the parameter names and types have been fetched from the server,

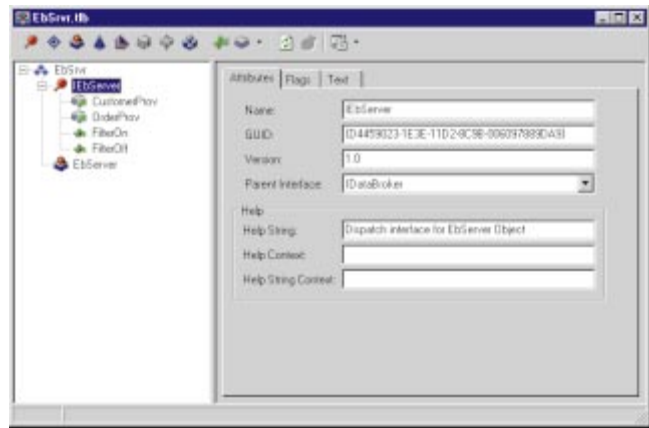


Figure 2: The Type Library Editor with the server's interface selected.

you can assign values to each parameter using the *Params.ParamByName* method.

You can send the parameters to the application server in one of two ways. If the *TClientDataSet* is closed, simply open it. Opening the *ClientDataSet* automatically sends the current parameter values to the application server. If the *ClientDataSet* is already open, call its *SendParams* method to send the new parameters to the application server. The server will automatically close the query or stored procedure, assign the new parameter values, then reopen the query. After calling *SendParams*, be sure to call the *ClientDataSet's* *Refresh* method so it will retrieve the new records from the application server.

Calling custom methods on the application server is no different than calling a custom method in any other automation server. The first step in implementing a custom method in the server that can be called from the client is to add the custom method to the server's interface using the Type Library Editor. Open the Type Library Editor by selecting **Type Library** from the **View** menu, then click on the interface to select it, as shown in [Figure 2](#).

Click the **New Method** button to add as many methods as you need, and to add any parameters and a return value if required. Finally, click the **Refresh Implementation** button to update the type library interface unit, and add the new methods' stubs to the remote data module's unit. In the previous example, two new methods, *FilterOn* and *FilterOff*, were added to the interface. These methods can now be called from the client, using the connection component's *AppServer* property.

Early Binding

By default, the MIDAS connection components use late binding when calling methods of the application server. Although late binding works regardless of the type of connection between the client and the application server, it's slower than early binding. Also, early binding provides compile-time error checking of all your interface method calls. Early binding is only available if you use DCOM for the connection.


```

procedure TEcMainForm.ByName1Click(Sender: TObject);
var
  IServer: IEbServer;
begin
  with MainDm do begin
    IServer := IDispatch(EbConn.AppServer) as IEbServer;
    IServer.FilterOn;
    CustomerCds.Refresh;
  end;
end;

```

Figure 3: Calling a server method with early binding.

To use early binding, you must obtain an interface reference, which is simply a pointer to the interface's vtable (virtual method table), for the application server's interface. This is a two-step process. First, cast the connection component *AppServer* property to *IUnknown* or *IDispatch* to get an interface reference. This converts the *AppServer* property from a variant of variant type *varDispatch* to an interface reference. Next, cast the *IUnknown* or *IDispatch* interface to the interface type of the application server. This causes COM to call *QueryInterface* and return a reference to the application server's vtable. In the code in [Figure 3](#), both casts are performed in a single statement. This code calls the custom *FilterOn* method that was added to the application server's interface using the Type Library Editor.

IEbServer is the application server's interface and *IServer* is an interface reference variable. *EbConn* is the *TDCOMConnection* component. To initialize the interface reference variable *IServer* to point to the interface's virtual method table, the *AppServer* property is first cast to *IDispatch*, then to *IEbServer*. Once the interface variable has been initialized, it can be used to call the methods of the interface using early binding.

Although you cannot use early binding if you aren't using DCOM for the connection, you can improve performance compared to late binding by using the application server's dispatch interface. The code in [Figure 4](#) is nearly identical to the previous example, except that the connection component's *AppServer* property is cast to the dispatch interface type, *IEbServerDisp*.

Controlling the Client

The application server in a multi-tier application can call methods in the client application. This allows events that occur in the server to call event handlers in the client. The first step in allowing the server to call methods in the client is to add another interface to the server's type library, as shown in [Figure 5](#).

This figure shows the type library for the sample *EbSrvr* application after adding a second interface, *IEbClient*. After adding the second interface, select it and add all the client methods that the server will call. In this case, a single method named *ConfirmFilter* was added to *IEbClient*. The client application will include an object that implements this interface.

```

procedure TEcMainForm.ByCustomerNumber1Click(
  Sender: TObject);
var
  IDispServer: IEbServerDisp;
begin
  with MainDm do begin
    IDispServer :=
      IDispatch(EbConn.AppServer) as IEbServerDisp;
    IDispServer.FilterOff;
    CustomerCds.Refresh;
  end;
end;

```

Figure 4: Calling server methods using the dispatch interface.

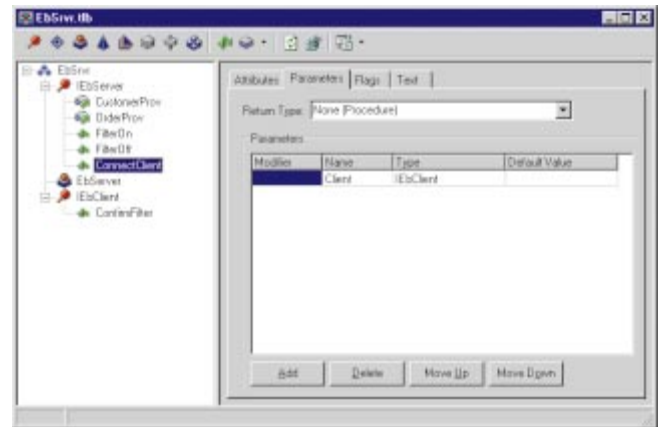


Figure 5: Adding the *IEbClient* interface to the server.

```

type
  TEbServer = class(TRemoteDataModule, IEbServer)
    Database1: TDatabase;
    CustomerTable: TTable;
    CustomerProv: TProvider;
    OrderTable: TTable;
    OrderProv: TProvider;
    procedure EbServerCreate(Sender: TObject);
    procedure EbServerDestroy(Sender: TObject);
    procedure OrderTableBeforePost(DataSet: TDataSet);
    procedure CustomerProvGetDataSetProperties(
      Sender: TObject; DataSet: TDataSet;
      out Properties: OleVariant);
  private
    ClientConnection: IEbClient;
  protected
    function Get_CustomerProv: IProvider; safecall;
    function Get_OrderProv: IProvider; safecall;
    procedure FilterOn; safecall;
    procedure FilterOff; safecall;
    procedure ConnectClient(const Client: IEbClient);
      safecall;
    function IsDatabase: WordBool; safecall;
  end;

```

Figure 6: The server's remote data module with the *ClientConnection* added.

Next, add a method to the server's interface, which takes a reference to the client interface as its only parameter. In [Figure 5](#), this method is called *ConnectClient*. *ConnectClient* assigns its interface reference parameter to a variable that is added to the server's remote data module. The code in [Figure 6](#) is the type

```

procedure TEbServer.FilterOn;
begin
  CustomerTable.Filtered := True;
  if CustomerTable.Filtered then
    ClientConnection.ConfirmFilter('Filter Is On')
  else
    ClientConnection.ConfirmFilter('Filter Is Off');
end;

procedure TEbServer.FilterOff;
begin
  CustomerTable.Filtered := False;
  if CustomerTable.Filtered then
    ClientConnection.ConfirmFilter('Filter Is On')
  else
    ClientConnection.ConfirmFilter('Filter Is Off');
end;

```

Figure 7: Callbacks to the client to confirm the filter state.

declaration for the server's remote data module after the private member variable *ClientConnection* has been added.

ClientConnection is the server's reference to the object that implements the *IEbClient* interface in the client. With this interface reference, the server can call any method of the interface. In this application, the code in [Figure 7](#) is added to the server's *FilterOn* and *FilterOff* methods to confirm to the client that the filter on the Customer table is either on or off.

This code shows the use of the *ClientConnection* interface reference variable to call the *ConfirmFilter* method in the client application. On the client side, an object that implements the *IEbClient* interface must be added to the client application, as shown in the following:

```

TCallback = class(TAutoIntfObject, IEbClient)
  procedure ConfirmFilter(const Msg: WideString); safecall;
end;

```

The *TCallback* object descends from *TAutoIntfObject*, which provides support for the *IDispatch* interface. The implementation code for the *ConfirmFilter* method simply assigns the *Msg* parameter to the *Caption* property of a label on the client's main form so the user can see the message.

The heart of the callback mechanism is in the *OnCreate* event handler for the client application's main form (see [Figure 8](#)). The variable declarations are global to the main form's unit. This code begins by calling the Windows API function *LoadRegTypeLib* to load the server's type library. The parameters are:

- The type library's GUID.
- The type library's major version number.
- The type library's minor version number.
- The national language code of the library.
- An interface reference variable of type *ITypeLib* that is initialized by the call to point to the type library.

LoadRegTypeLib returns a result code indicating whether the type library was successfully loaded or not. If the library is

```

...
var
  ServerTypeLib: ITypeLib;
  TypeLibResult: HRESULT;
  CallBack: IEbClient;
...
  // Set up the callback interface.
  TypeLibResult := LoadRegTypeLib(LIBID_EbSrvr, 1, 0, 0,
    ServerTypeLib);
  if TypeLibResult <> S_OK then begin
    MessageDlg('Error loading type library.',
      mtError, [mbOK], 0);
    Exit;
  end; // if
  // Create an instance of the TCallback object.
  CallBack := TCallback.Create(ServerTypeLib, IEbClient);
  // Get an interface reference to the server.
  Srvr := IDispatch(MainDm.EbConn.AppServer) as IEbServer;
  // Pass the interface handle of the TCallback object to
  // the server.
  Srvr.ConnectClient(CallBack);
...

```

Figure 8: Creating the *Callback* object in the *OnCreate* event handler of the client application's main form.

successfully loaded, an instance of the *TCallback* automation object is created. The type library and interface that the *TCallback* object implements are passed as parameters to its constructor, and the returned value is assigned to the interface reference variable *Callback*. Next, a reference to the server's interface, *IEbServer*, is obtained by casting the connection component's *AppServer* property to the interface type. The final step is the statement:

```
Srvr.ConnectClient(CallBack);
```

which calls the server's *ConnectClient* method passing the interface reference variable for the *TCallback* object as a parameter. As you have seen, the server can now use this interface reference to call any method of the *TCallback* object that is a member of the *IEbClient* interface.

Controlling What the Client Sees

While letting the server call methods in the client is a very powerful way to implement a two-way exchange of information, there are other ways to control what the client sees. There are three ways to limit which fields the client application sees. The first is to use a query as the dataset in the application server, and only select the fields the client application should see.

The second method is to create persistent field objects using the Fields Editor, in the server application, and only create field objects for those fields the client application should see. Note that if you create calculated or lookup fields in the Fields Editor, they will be sent to the client as read-only fields. There is a potential problem with limiting fields using the Fields Editor because you must include the entire primary key if the client will edit, delete, or insert records so that the record will be uniquely identified. If you need to include the primary key so the record can be edited, but do not want the

client application to have access to one or more of the primary key fields, you can select the field object in the Fields Editor and use the Object Inspector to change the field object's *ProviderFlags* property to include the *pfHidden* flag. The effect is similar to rendering a field invisible in a grid by setting its *Visible* property to *False*. The field is there, but it cannot be accessed.

You can also add information to the data packets the server provides to the client. This can be any type of information, and you can specify that it also be included in the delta packets returned to the server when the client applies updates. This means the server can send a round-trip message to itself. The sample *EbClnt* and *EbSrvr* applications use this technique to send the current *Filter* property for the Customer table to the client for display.

To place additional information to the data packets sent by the provider component, begin by creating an event handler for the provider's *OnGetDataSetProperties* event. The event handler gets three parameters: the first is *Sender*, the second is *DataSet*, and the third is *Properties*. *DataSet* is a pointer to the dataset that supplies the provider's data. *Properties* is an *OleVariant* in which you place all the additional information you want included in the data packet. *Properties* must be a variant array, and must include three elements for each attribute you add to the data packet. The first element of the array is a string that contains the attribute's name. The second is a variant that contains its value, and the third is a Boolean value, which is *True* if you want the attribute returned in the delta packets.

To add more than one attribute to the data packet, make *Properties* a variant array of variant arrays, as shown in the code from the *EbSrvr* sample program in [Figure 9](#). This code adds two attributes to the data packet. The first contains the value of the dataset's *Filter* property, and the second the value of the dataset's *Filtered* property. Note that the *Filter* attribute is returned to the server in the delta packets. On the client side, use the *TClientDataSet*'s *GetOptionalParam* method to retrieve the value of any attribute from the data packet. The following code is from the **U.S. Only** menu item's *OnClick* event handler:

```
FilterStringLabel.Caption :=
  CustomerCds.GetOptionalParam('Filter');
```

```
procedure TEbServer.CustomerProvGetDataSetProperties(
  Sender: TObject; DataSet: TDataSet;
  out Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] :=
    VarArrayOf(['Filter', DataSet.Filter, True]);
  Properties[1] :=
    VarArrayOf(['Filtered', DataSet.Filtered, False]);
end;
```

Figure 9: Adding information to the data packet.

This code retrieves the value of the *Filter* attribute as a variant, and assigns it to the *FilterStringLabel*'s *Caption* property.

Real Transaction Control for Local Tables

Although Inprise claims there is transaction support for local Paradox and dBASE tables, it's not true. Transaction support requires the database always be left in a consistent state, i.e. either all or none of the changes that are part of a transaction will occur. For this to happen, the database must roll back all active transactions upon restart after a crash. Transactions for local tables are not rolled back after a crash; instead, any changes that were posted will still exist in the database even though the transaction was never committed.

There is a second major problem with local table transactions. The only transaction isolation level supported is *tiDirtyRead*, which can lead to serious problems. Suppose a physician is entering treatment information for a patient and accidentally enters drug therapy information for the wrong patient. If the record is posted, the change will now be visible to all other users even though the transaction is not committed. What happens if another user prints a list of drugs that must be administered at this point? Even though the physician reviews his/her entries and rolls back the transaction, the patient will still receive the wrong medication.

A much better solution when working with local tables is to use *TClientDataSet* for all data entry. This is easy to do in Delphi 4: Simply drop a *TProvider* and a *TClientDataSet* on a form or data module that already has a *TTable* connected to the table you want to edit. Set the *DataSet* property of the *TProvider* to the table, then set the *Provider* property of the *TClientDataSet* to the *TProvider*. This simple approach assumes that *TClientDataSet* can hold all the table's data in memory. If that's not the case, you will have to use ranges, filters, or queries to restrict the set of records the user works with at one time.

Why is *TClientDataSet* a better solution than local table transactions? First, consider what happens if one user changes a record and posts the change, then another user looks at the same record. The second user will see the unchanged version of the record until the first user calls the *TClientDataSet*'s *ApplyUpdates* method to "commit" his or her transaction. This means that you effectively have read committed transaction isolation instead of dirty read transaction isolation.

Now consider what happens if a user makes several changes and his or her system crashes. Because both the data and changes made using a *TClientDataSet* are held in memory until *ApplyUpdates* is called, they will all be lost. This effectively gives you automatic rollback on restart after a crash. The only time you are vulnerable is during the brief interval between the moment you call *ApplyUpdates* and when the changes have actually been written to disk. If the workstation crashes while the writes are taking place, the database may be either inconsistent or corrupt; however, because the time that the database is in an inconsistent state is very short, the chances are small.

By comparison, when you use local table transactions, the database is in an inconsistent state from the time the user posts the first change of the transaction until the user commits the transaction. This could be several minutes for a transaction that involves manually changing several records. The LclTran demonstration application shows an example of using *TClientDataSet* with local tables.

Using *TClientDataSet* for Flat-file Applications

TClientDataSet is a great tool for building single-user database applications that deal with modest amounts of data. It provides all the features of a relational database — except query support — with nothing to install or configure on the user's machine. All you have to do is distribute the *dbclient.dll* file with your program. The big limitation of *TClientDataSet* is that it holds all the data in memory. However, that is not as bad as it sounds when you consider that 100,000 records, 100 bytes in length, require 10MB of memory.

Using *TClientDataSet* for single-user, flat-file applications is much easier in Delphi 4. One of the most onerous aspects of writing a flat-file application in Delphi 3 is that the only way to create your data tables is in code. In Delphi 4, you can drop a *TClientDataSet* on a form or data module, and define your tables interactively using the Fields Editor. Simply double-click the *TClientDataSet* to open the Fields Editor and add new data fields. When you're done, right-click the *TClientDataSet* and choose **Create DataSet** from the context menu.

Another new feature useful in flat-file applications is the ability to use nested datasets. When you create a master *TClientDataSet*, you can add fields in the Fields Editor whose type is *DataSet*. To use the nested dataset, add another *TClientDataSet* to your project and set its *DataSetField* property to the field of type *DataSet* that you added to the master *TClientDataSet*. Now, open the Fields Editor for the detail *TClientDataSet* and add the fields for the detail dataset. Note that you don't have to add a foreign key field to link the detail to the master, because the detail is actually contained by the master.

One problem you may encounter is trying to add another field to a table after you have created the dataset. You can add the field in the Fields Editor, but you'll get an error when you right-click the *ClientDataSet* and choose **Create DataSet**. To overcome this, select the *TClientDataSet* and open the *FieldDefs* property in the Object Inspector by clicking its ellipsis button. In the Collection Editor, select all the field definitions and delete them. Now, right-click the *TClientDataSet* and choose **Create DataSet** to recreate all the field definitions, including the new field.

The sample Phone application demonstrates this technique. This is a simple, two-table application. The master contains people, and the detail contains phone numbers. One of the advantages of using nested datasets is that both the master and detail are saved in a single file, *phone.ffd*. The

code from the **Save Changes** menu item's *OnClick* event handler first posts any unposted changes in both datasets, then merges the changes and saves the *PersonCds* to the *phone.ffd* file (see **Figure 10**). Note that only the *Person* dataset is saved, because it contains the numbers dataset.

There is, however, a big disadvantage to using nested datasets. You cannot search the entire detail dataset for a record. This would be a serious problem in a Customer/Orders relationship, where searching the entire Orders dataset by order number to find a specific order would be useful. Even in the sample Phone application, it might be nice to search for a person by phone number when you are trying to reconcile your long-distance phone charges.

Maintained Aggregates

Maintained aggregates are another new feature of *TClientDataSet* in Delphi 4. They allow you to maintain the sum, count, min, max, or average of any number of fields in a *TClientDataSet*. What's even more valuable is that they support groups and expressions. You can use any index to group the aggregates. In the case of a composite index, you can also specify the number of fields to group on.

There are two ways to create and use maintained aggregates. The first is through the *Aggregates* property of *TClientDataSet*. Before creating any aggregates, be sure to set the *AggregatesActive* property of the *TClientDataSet* to True. You can set this property at any time — but don't forget it, or your aggregates won't work. Next, click the ellipsis button in the *Aggregates* property to open the Collection Editor. Then, press **Insert** to add an aggregate. Finally, set the aggregate's properties in the Object Inspector. Again, set the aggregate's *Active* property to True so you don't forget. Give the aggregate object a meaningful name, then enter an expression for the *Expression* property. The expression can use the Sum, Min, Max, Avg, or Count operators on a single field, or on an expression involving two or more fields. For example:

```
Sum(TaxRate * SaleAmount)
```

is a valid expression, as is:

```
Sum(Price) - Sum(Cost).
```

```
procedure TMainForm.SaveChanges1Click(Sender: TObject);
begin
  { If there are unposted records post them. }
  if MainDm.PersonCds.State in [dsEdit, dsInsert] then
    MainDm.PersonCds.Post;
  if MainDm.NumberCds.State in [dsEdit, dsInsert] then
    MainDm.NumberCds.Post;
  { Merge the changes in Delta with the data and save it. }
  with MainDm.PersonCds do begin
    MergeChangeLog;
    SaveToFile('phone.ffd');
  end;
end;
```

Figure 10: Saving the master and detail tables in a single file.

```

procedure TMainForm.FormCreate(Sender: TObject);
begin
  TotalLabel.Caption :=
    IntToStr(MainDm.PersonCds.Aggregates.Find(
      'TotalRecords').Value) + ' saved records';
end;

```

Figure 11: Using the *Aggregates.Find* method.

To group using an index, set the *IndexName* property to the index to use, and set the *GroupingLevel* property to the number of the last field in the index to group on. For example, if you have an index on the Country, Region, and SalesTerritory fields, set the *GroupingLevel* to 2 to group by Region. Setting the *GroupingLevel* to 0 disables grouping, so the aggregate will use all the records in the dataset. The sample Phone application uses an aggregate named *TotalRecords* to display the total number of records that have been saved.

To use the aggregate value, use the *Aggregates* property's *Find* method to get the value, as shown in [Figure 11](#). The expression:

```
PersonCds.Aggregates.Find('TotalRecords').Value;
```

finds the named aggregate and calls its *Value* method to retrieve the current value of the aggregate.

Another alternative is to create an aggregate field using the Fields Editor. Add a new field to the *TClientDataSet* and choose **Aggregate** for the field type. Also, click the **Aggregate** radio button, then click **OK** to add the field. Select the field in the Fields Editor, then set the *Name*, *Active*, *Expression*, *IndexName*, and *GroupingLevel* properties. You can now access the aggregate like any other field in the dataset. This technique is particularly useful because it allows you to display the value of the aggregate using data-aware controls without writing code.

Conclusion

Delphi 4 brings a host of new features for multi-tier application developers, but even more important are the new features of *TClientDataSet* that are available to all developers. Whether you're developing an enterprise multi-tier application, a traditional two-tier client/server system, or a file-server-based program, *TClientDataSet* offers you maintained aggregates, the briefcase model, better caching than cached updates, better local table transactions than the built-in local table transactions, nested-table, flat-file applications without the BDE, and more.

This is a whole new way to develop any application. ▲

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM99\JAN\DI9901BT.

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





ON THE COVER

Delphi 3 Client/Server / Multi-tier Database Planning and Design

By *Thomas J. Theobald*



Multi-tier Database Applications

Part I: Planning for Database Independence

The use of n -tier technology can reduce the effort of porting applications between databases. Many vertical market firms can expand their customer base by presenting a database-independent product and, with good planning and design, reduce the cost of market penetration dramatically. Traditionally, you were required to recode complete two-tier applications that targeted different database vendors, increasing costs of maintenance, upgrade, and documentation, so a significant potential customer base was required to justify the extra expense incurred. By putting extra effort into the planning stages and performing development of the application in anticipation of multiple back-ends, this extra expense can be reduced significantly.

Models of client/server architecture originally were largely aimed at the two-tier model of client and server, where the two shared the responsibilities of user interface (UI), data management, and business-related logic. When the three-tier model became popular (seemingly by default), it adopted a UI-Business-Data separation of duty. This barely scratches the potential of n -tier technology, however. By adopting a collaborative model, multiple middle-tier objects can

apply their functionality to the data that passes between the client and the server.

There are several methods that provide a UI with database independence by using a tiered application architecture. Although the primary goal is to reduce the task of recoding, multi-tier architecture also provides savings by avoiding the burden of rewrites to documentation and online help, and product support and testing.

General Rules

As a demonstration of that idea, I will design a simple example application that will port between two back-end data sources by using a four-element architecture. While designing this, I will keep in mind these general rules:

- 1) Rewriting code stinks.
- 2) End users should not be required to have an awareness of where or how they are obtaining their information.
- 3) The application should be portable between back ends without having to change the front end or the business logic (BL) partition, i.e. the "business rules."

You'll note that these rules have their roots in object orientation. Rule 1 simply points



out that we should inherit and reuse where possible. Rules 2 and 3 are veiled definitions of encapsulation. Encapsulation will be a key to making this idea work; by providing a standard interface between our application partitions, we will make the job of porting far easier. Treating each partition or partition of the application as a separate and encapsulated object allows division of labor and use of multiple tools at design time, as well as simplifying overall maintenance.

In this article, I'll define four partitions of an application: user interface, business logic, data access, and data storage (not to be billed as an exclusive list; merely the four necessary to achieve the objective of this series). I'll describe the steps you need to take in building an application, as well as provide rough analogies to the simple client/server model in a Delphi application. To start, we'll work from the client to the server, describing the purpose of each partition in turn.

User Interface

General display and user interaction are contained here. This element will be consistent throughout all deployments of the application, regardless of the back end in use. The UI should contain no data-access-specific code, and will exist solely to translate data into information.

Note: If multiple front-end platforms are desired, the use of the strategies presented in this article allow for any number of functionally similar, front-end products to tie into a single middle-tier element. In the most likely circumstance, this would be a single, Java-based front end tying into a CORBA-compliant middle tier. Other options are available, but this would probably receive first consideration in most shops.

Business Logic

Validations, considerations, etc. that are corporation- or application-specific are contained in this partition. This may be malleable, dependent on whether different deployments of the same application (i.e. different clients) have different implementations of the same logic. In fact, this may be separated into corporate and application modules to make two partitions, allowing clients to perhaps include their own "entry point" code in a vertical market application. (Note: This is not as far-fetched as it may seem; a company I used to work for did something along this line with their own application and a run-time Lahey Fortran compiler.)

Data Access

All elements of direct access to a back-end database are stored here. This partition will be built as a single module that governs interaction with a back-end database. Vendor-specific details are provided here (e.g. queries for Oracle vs. InterBase, one of which requires the database name as a prefix). This is where the break from vendor-dependency is made.

The interface to this partition should be standardized to allow multiple versions of this partition to be interchangeable. A version would be created in the previous example for

InterBase, and one for Oracle. When the product is shipped or installed, the appropriate data access (DA) partition would be included with the application. Because the interfaces of the two programs are the same, the front end will not care which one it is accessing.

Data Storage

The actual data on which we operate is stored in the data storage (DS) partition. Databases will contain common entities, each of which may have at least four routines performed upon them: selects, updates, inserts, and deletes. Between platforms, it will probably be obvious that some differences will exist between the database definition language (ddl) scripts of the different vendors. Hopefully, these changes will be minimal, but hopes and reality rarely seem to coincide.

If the developers initially plan the back-end model with multiple vendors in mind, development of this can take place in concert with the design of the DA partitions, respective to the appropriate vendors.

The Process

Step one: The plan. The first step in all applications should be planning. I can't stress that enough. Even if the application in question is tiny, plan it. Always have a goal, and always strive toward that goal. I won't say that work can't be productive without a plan, because some applications do happen to work with seat-of-the-pants coding styles. However, we're talking about some very large and, in most cases, very expensive applications.

When the tools you use cost in the multiple thousands of dollars (and Inprise is extremely inexpensive when compared to much of the competition), the combined costs of developer salaries, operational overhead, and on-the-shelf selling times are dramatic — not to mention that if a bad architecture decision is made early, a good portion of the project will be dedicated to rewriting code. Not only does that mean it's expensive, it's also unpleasant.

We can't afford not to plan. For this step, I highly recommend that you separate your plan into several steps. The first of these is to model the business conceptually. This takes the form of discovering "use cases" and building up your developers to have the general knowledge necessary about the business for which the application is being written. In a vertical market application, this is generally not a big concern (they probably already know the business), but in a consulting environment, this is critical.

The second step is to take the use cases developed and identify the various objects involved — consider everything an object; actions, customers, rules, and inventory items are objects. Arrange them in a hierarchical order and decide how they logically work together. This is where the development team will begin to see patterns and trends surface.

Start with the most general, and work down to the most detailed. The mission statement of the application will be the start: "Track the production and sale of widgets from the

purchase of raw materials to the delivery of finished products to wholesalers and our retail arm.” This will yield the application object and its purpose. The minutiae will be the last: “Invoke the credit-card billing mechanism with the credit-card number given.” This example would define the action of processing a single credit-card number. In theory, the application might then be able to track business flow from the specific lot of raw materials to the mailing address of the customer to whom a specific widget was sold.

If the entire object model was drawn on a whiteboard, it would probably appear as an “org chart,” with the single application entity at the top, extending its “roots” down through more and more divisions to the bottom simple layer of actions and data fields.

The third step is to identify commonalities among the objects identified, and then develop a set of hypothetical classes and ancestor classes. I’ll call these proto-classes, as we have only developed an idea of their role, not their implementation. Note that right now, we haven’t touched a bit of code, and still haven’t made our tool selection. We’re still planning.

The final step of planning is to organize the proto-classes and ancestor proto-classes into manageable, programmable elements. These will become the actual classes and ancestor classes the team will develop to build the application. This step will depend largely on the resources available to the programming team; only they can determine what they can accomplish in a given amount of time. The idea is to be practical about what needs doing, and what might be going overboard on class divisions. I will point out, however, that Rule 1 exists for a reason, and I’ll restate it: Rewriting code stinks. If you find yourself rewriting code more than once, it’s time to further abstract what you’re writing.

Although I’m a freak for planning, I will point out that no plan can foresee reality perfectly, and you shouldn’t be afraid to modify the plan if prevailing conditions demand it. Just make the changes wisely, and you’ll probably avoid most pitfalls.

Step two: Division of function. Okay; we’ve got a plan. Now what? Well, let’s look at what we’ve built so far. We have a business model and a big fat bundle of proto-classes and ancestor proto-classes. We know the operations and objects that will exist in the application. We now need to identify where they belong.

In light of the subject matter, we’ve got to figure out what classes belong in the UI, BL, DA, and DS partitions. Chances are pretty good that we’re going to have analogous classes in several layers. Look at each proto-class you have laid out in your model and ask the following questions about it:

- Does the user need to deal with it? If yes, it belongs at least in the UI.
- Does it exist solely to convey data back and forth? If yes, it belongs at least in the DA.
- Does it exist as a rule the company enforces in general business practice? If yes, it probably belongs to the BL.

- Does the database need to deal with it? If yes, it belongs at least in the DS.

Other questions will come up as you begin dividing these proto-classes among the application partitions. Keep the focus on identifying their roles within the application. Chances are good that a number of additional needs will be identified; create proto-classes for these as you go.

Step three: Develop the partitions. Once the proto-classes have been divided among the logical partitions of the application, it’s time to pick the tools for use and start coding. Tool choice, as always, depends largely on the skill set of the existing development team, and the capacity of the tool to get the job done. My personal favorite is Delphi. Okay, so I’m biased; show me another tool that can program all four layers (C++Builder doesn’t count because it’s simply Delphi in disguise). Considering the audience reading this, I’m guessing you understand.

A well-tuned development team will probably spend a good 40 percent of their budget before they write a line of code. That’s important to note. The more time you spend in good planning, the less time you’ll need to spend correcting. Note that I said *good* planning. Sitting around a table and bandying words like “paradigm shift” and “business rules” are for Dilbert and marketing staff. You and everyone at the table should have a goal and should know most of these words already. Spend your time wisely, figuring out how to overcome problems you can foresee in the upcoming development effort. This is the old pay-me-now-or-pay-me-later problem.

To start with, make sure to target only a single back end. The temptation will exist to try to kill two birds with one stone, but honestly, it will be much easier to build a single DA partition and modify it than to build two DAs with identical interfaces as a parallel effort. If you’ve got the money to support this, fine. If you don’t, however, let the income from the first release fund your development of the second. You build experience by constructing your first practical classes from the proto-classes, and cut time in development of the second set.

Remember that for the purposes of this article, we’re talking about four partitions. I’m going to spell out a couple of ways in which it can be done in Delphi 3 (unfortunately, Delphi 4 wasn’t out when I wrote this).

The actual development will ideally take another 40 percent of the available budget. During the course of the later portions of this time, documentation can be written based on the use cases identified in the planning.

The difference between the two-tier model and the *n*-tier model is in the distribution of processing load; we now have three or more machines available to do our work for us, and we also have means to isolate conceptual functions. In the example I give here, I isolate DA from BL to allow my application to hit two very different brands of data storage: Paradox and InterBase.

First, a short discussion on just how *n*-tier works as it relates to Delphi will be necessary; we'll see the components needed and how they interact. After that, I'll discuss a few models of *n*-tier. Then, we'll see the working example.

N-tier, and What Goes Where

The old methods. On the DS side, we used to have to incorporate most of our business rules at the server if we wanted to avoid re-deploying our software with every change to corporate ideals. Now, we're generally going to be ensuring solely that the data is stored correctly, and we aren't going to give that much of a hoot whether the data is actually correct. This may sound blasé, but really, the data side should only be concerned with data. Dirty or clean, data is simply data. Given those conditions, we'll probably be talking about enforcing referential integrity via either a manually constructed trigger set, or by the database engine's own RI capability. We might also be enforcing normalization (for instance, checking to make sure we're not entering a duplicate lookup record or something). The validity of the data in storage should be checked before it ever gets here.

In the two-tier model, we would have conceivably been entering all sorts of business logic to make a very "fat" server side. Fortunately, we're not doing that here, so no exhaustive SQL will be necessary.

The next option was to place enforcement on the client side. We did something like that in the previous example, in that the client checks its data input prior to allowing it to be updated. The actual action of validation is beginning at the client, but we've used rules supplied from outside our influence. This allows a thin-client approach, as the bulk of the validation code and the processing load sits off-site at one or more middle-tier partitions.

Using Delphi or C++Builder, client-side validation normally takes place at the *TField* level; it shouldn't be a surprise that *TField* has an *OnValidate* event. One could conceivably build a UI that validates the entire record before posting, but I won't get into the gory details of that. Generally, when a user attempts to update or enter a field value, Delphi checks if that field has a validation event, and if it does, it gets fired. The developer's code looks over the new value, and if it passes all the conditions stipulated in the developer's code, it passes. Otherwise, the developer cancels that edit by raising an exception or by some other means. See the UI partition sample code, or the Inprise courseware on field validation (in my example, look in *prUIPartition*, and check its *TEmployeeForm.EmployeesSalaryValidate* method to see an example of this).

Again, that example uses the UI to call the play, but the actual actor that determines a pass/fail result is somewhere outside the UI. I could just as easily have put the validation code in the UI, but that would have defeated the purpose of the example.

New ways of validating. Now for the "new" stuff. Some of the validation may actually take place at a middle tier in a

couple of ways. I'll start by describing how Delphi performs *n*-tier operations.

We know we start at the data server, go to a middle tier through its *TDataSets*, then to a UI with *TClientDatasets* that refer to *TProviders* at the middle tier. How do we make the middle tier actually do a lot of work for us?

Validating after server rejection. The most commonly referred-to method is to use the *TProvider.OnUpdateError* event. If we use explicit *TProvider* controls instead of the auto-instantiated Delphi ones, we can write our own handling of the occurrence of an error at the middle tier when it tries to apply the changes in a delta packet to the server.

There is a good bit of online help about this, but the basic process is that the middle tier receives a delta packet and processes its updates row by row. With each row, if, for some reason, it cannot be applied to the server, the *OnUpdateError* event fires, which allows the developer to insert code designed to handle those conditions and possibly fix or reissue the update rather than send it back to the user for checking (although that is the preferred behavior). Any records left in the update cache after the *TProvider's ApplyUpdates* (records that triggered an *OnUpdateError* and were assigned a response of *rrSkip*) will be reissued to the client dataset for resolution there.

Step-by-step, the process is:

- 1) User applies updates via the *TClientDataSet*.
- 2) The *TProvider* at the middle tier receives the delta packet.
- 3) *TProvider* applies updates in the delta packet row by row.
- 4) Any row with a problem triggers an *OnUpdateError* event.
- 5) Resolution at the middle tier occurs.
- 6) Unresolved errors skipped remain in the delta packet cache.
- 7) The delta packet cache is returned to the *TClientDataSet*.
- 8) If an *OnReconcileError* event is defined at the client dataset, it is fired for each row in the delta packet cache.
- 9) The developer's code may determine how to respond to the error in each row.

One warning: In handling the error and its reconciliation, don't navigate the dataset. This will throw off your record buffer and blow the effort. Delphi is already in a loop to get all the records, and additional navigation could knock it out of whack. Just deal with the data as it is presented to you, and don't mess with where you are looking.

A great (if complicated) example of how an *OnReconcileError* event can be dealt with is in the *RecError* unit, which I've blatantly used from the *EmpEdit* demo. Those of you with Client/Server editions already have this; on my system, it's in Program Files\Borland\Delphi\Objrepos\recerror.pas. One of my biggest rules for development is, if it's been written for you, use it.

Validating before server submission. If you wish to actually have your delta packet validated before submitting it to the server, using a *TProvider.OnUpdateData* event allows you to investigate

what is going to be passed to the server. This view is read-only, so the developer will be unable to apply any changes here. Unlike *OnUpdateError*, however, this event doesn't occur for every row; instead, it occurs once per batch of updates. The developer is responsible for cycling through the dataset. In doing so, he or she can write code to log events, approve data values, etc. Also unlike *OnUpdateError*, the developer cannot do "line-item vetos," but may only pass or fail the entire dataset (the easiest way to cancel the update is to raise an exception in this event). [This process differs in Delphi 4; see Bill Todd's article "[Delphi 4 Multi-tier Techniques](#)" on page 7 of this issue for details.]

Using these two events can allow your middle tier to do more than just pass data back and forth; it really introduces a means by which much of your logic work can get done outside the client system.

Until Next Month

We'll delve into the [code next month](#). However, you can take a look at the example projects now if you'd like a preview (they're all available for download). Each project and the sample data (i.e. the InterBase database) unzip into separate directories for you to examine. [▲](#)

The files referenced in this article are available on the Delphi Informant Works CD located in `INFORM\99\JAN\DI9901.TT`.

Tom Theobald is a senior software developer with Segue Technologies of Alexandria, VA. He began his career with computers as a NetWare engineer, moving later to include NT and Lotus Notes among his acquired skillset. Now a certified Delphi instructor, he makes his trade helping large corporations and government agencies acquire a much more Zen-like attitude toward software development. He can be reached at theobaldt@seguetech.com with any business inquiries, questions, or comments. Death threats and other matters of a personal nature can be forwarded to eviltom@worldnet.att.net.





By Paul M. Fairhurst

MTS Development

Part II: Three-tier Development

Last month, we created a simple MTS (Microsoft Transaction Server) component, installed it into the MTS environment, and called its methods from a standard Delphi client application. If you'll recall, we're developing our components for a fictitious bank named DelphiBank. This forward-looking bank has decided to develop an application to provide its customers with online banking. Each customer will be given a logon account and password, which will allow them to view their account details, including current balance and a list of transactions. They'll also be able to make payments, deposit money, and transfer funds between their accounts.

This month, we'll implement simple logon security. We'll also concentrate on developing server components that access a Paradox database, and use the new BDE's support for pooling database connections from MTS objects. We'll then develop a client application that uses DelphiBank services. (The

complete source for all projects discussed in this article is available for download; see the end of this article for details.)

Before we begin, we'll walk through the structure of a three-tier MTS system, and see how the pieces of the DelphiBank system fit in this structure. We'll then work back from the database through to the MTS server objects, ending with the client application.

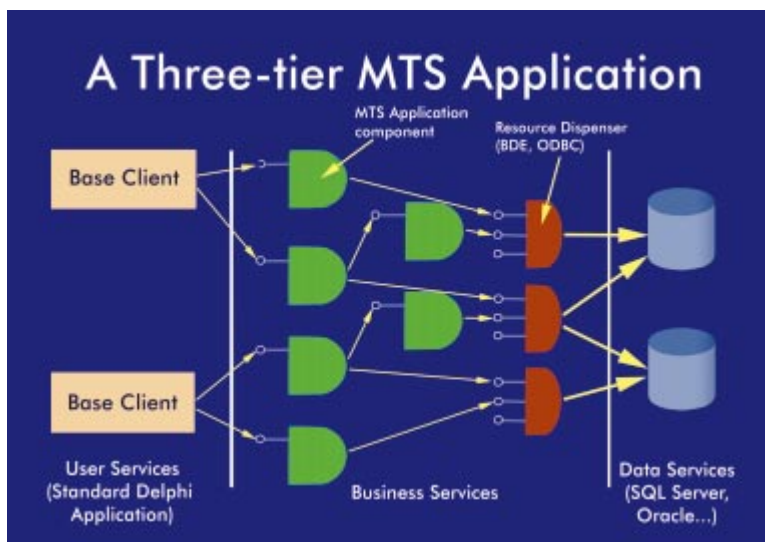


Figure 1: A three-tier application structure.

Structure of an MTS System

In Figure 1, notice the division of the MTS system into three areas: User Services, Business Services, and Data Services. User Services executes on the user's computer and is what the user sees. It's a lightweight, standard Delphi application with no BDE, nor data-aware controls, in sight. In MTS terms, User Services is a Base Client, so named because it directly accesses MTS components from outside of MTS. The Base Client is nothing more than a GUI front-end to Business Services. No enforcement of business rules takes place in this layer, and no database access is allowed.

Indeed, the client is unaware of the presence of a database.

The Business Services layer, also known as the middle tier, contains MTS components. The components provide a gateway between the client and the data. MTS is installed in this layer (which will almost always be on a separate machine from the client, unless you're developing with one PC), which is dedicated to serving client requests. The components expose their functionality through COM interfaces, which can be executed by a Delphi COM client.

Having said this, another MTS object, or any other client that supports the COM model, such as Visual Basic or Visual C++, could call the components. Business rules are implemented in the Business Services layer. An example of a business rule is ensuring a debit to an account doesn't cause that account's balance to drop below the minimum allowed. Another example is allowing only managers to make payments of more than a certain amount. The benefit of putting business rules in this layer is that they can evolve without affecting the client installations. This is important because it lowers support and maintenance budget requirements.

Middle-tier components know how to store data. They use a Resource Dispenser, which provides access to data storage and can dispense connections to resources quickly and efficiently by pooling and re-using them. The new BDE that shipped with Delphi 4, and the latest version of ODBC, are two such resource dispensers. Generally, they use a resource manager, such as Microsoft SQL Server or Oracle, that sits in the Data Services layer. They provide durable, transaction-based storage of data.

The Data Services layer can reside on a separate machine from the Business Services layer, depending on the expected workload. With low loads, running both on the same machine shouldn't present a problem. As a system's complexity and traffic increases, however, they'll each need dedicated servers. Again, such a change will not affect client installations because the BDE is on the Business Services server and only needs reconfiguring in this one place. As a final note, resource managers allow data manipulation through stored procedures. However, if the database is file driven, such as with Paradox or Microsoft Access, the data manipulation will have to be done in the Business Services layer.

The Database

We'll use a Paradox database, so our data manipulation will have to be done with *TQuery* objects in the MTS objects. **Figure 2** shows the database structure; we have four tables to hold information about customers, their accounts, financial transactions on each account, and the type of account, e.g. Current, Savings, etc.

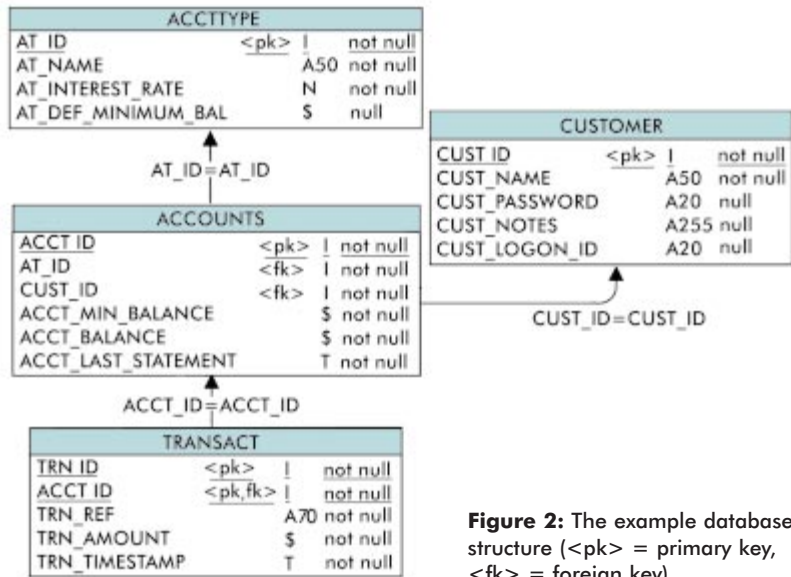


Figure 2: The example database structure (<pk> = primary key, <fk> = foreign key).

We saw Account Types (table ACCTTYPE) in **Part I** of this series. It defines the name of the account type, the default minimum balance, and interest rate applied to accounts of that type, as well as a unique ID. Next, we have Customers (table CUSTOMER). Here we store the customer ID, customer name, logon ID and password, plus any notes about the customer. Each customer can have many accounts (table ACCOUNTS) of different types; hence, in this table, we must store the customer ID and account type ID. We also generate a unique account number for each account and hold the account's current balance, allowed minimum balance, and the date and time of when the last bank statement was sent to the customer.

Finally, each account can have many transactions (table TRANSACT). Here we hold a unique transaction ID, the account ID, a reference string (such as "DEP" for deposit), the amount of the transaction, and a timestamp of when the transaction took place. In the case of transferring money from one account to another, two transactions would appear: one deducting the money from account 1, and the other depositing the money into account 2.

Server Component Structure

The server component contains four MTS objects: *AccountTypes2*, *Accounts*, *Customers*, and *Transactions*. Each handles operations on a table in the database, and implements the *IAccountTypes2*, *IAccounts*, *ICustomers*, and *ITransactions* interfaces, respectively. We implemented an *IAccountTypes* interface in **Part I**. Because the interface methods have changed, COM standards dictate that we define an entirely new interface so we don't break existing clients of *IAccountTypes*. The accepted way of doing this is to name the new interface with a succeeding number.

You can see the objects and interfaces in **Figure 3**, which shows the type library for our MTS component, *DelphiBankServer2*, in the Delphi Type Library Editor. The *IAccounts* interface is open, revealing its methods. The *Accounts* object implements *IAccounts*, and thus supports all

the functions in this interface. When a client creates an *Accounts* object, they know all methods of *IAccounts* will be implemented.

The method selected in Figure 3 is *ListByCustID*, which returns a list of accounts for the given customer ID. You can see the function parameters on the Parameters page on the right-hand side of the Type Library Editor. Every function in every interface starts with the parameter *ClientKey*, and ends with *strDebug*. The client key is obtained when a customer is logged on, and it's used to time out client connections. (We'll implement client key functionality in Part III of this series.) The debug string is assigned whenever an error or exception occurs inside an object method. Debugging MTS server components can be tricky, and simple feedback like this proves extremely useful when developing. The debug messages can also be optionally passed back to the client, usually when a business rule is blocking the action. Notice the third parameter, *varResultSet* of type *OleVariant*. This is how we pass back a query result set from a database query to the client. We'll see how to do this shortly.

As mentioned before, each object handles a particular table in the database. Also, each object has its own data module that contains all the *TQueries* it needs to perform its tasks, as well as a *TDatabase* component that provides a database connection for them. All four data modules are descended from a common data module, *DelphiBankCommonDM*, that gives the data modules access to field-name constants and some common code. Each data module is created and managed by its related object. When it's created, a data module automatically opens a connection to the database, and closes it when the data module is destroyed. This way, the queries contained in the data module are always available.

We will now focus on the *Accounts* object, in particular the *ListByCustID* and *Transfer* methods. The functionality of the remaining objects is coded in a similar way. Once you understand how the *Accounts* object works, you should have no problem understanding how the others work.

The Accounts Object

Every MTS object descends from *TMtsAutoObject*, which provides the back-drop functionality for your object to interact with MTS. There are two methods here that can be overridden: *OnActivate* and *OnDeactivate*. You know from last month's article that MTS activates and deactivates objects to conserve system resources. When this happens to your object, these methods get fired. Therefore, this is the place to allocate and free resources (e.g. a database connection) held by your object.

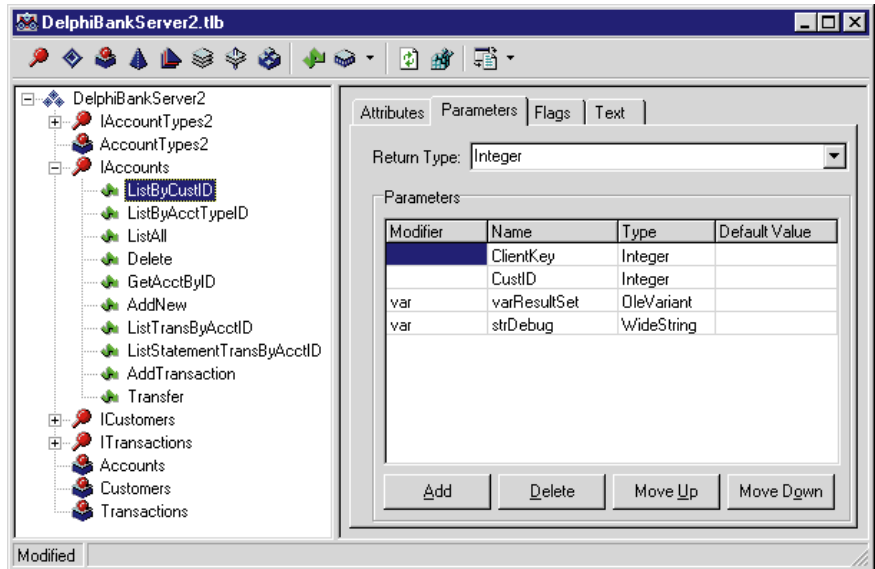


Figure 3: The MTS component's type library.

```

procedure TAccounts.OnActivate;
begin
  try
    { Create our own datamodule. }
    dmAccounts := TdmAccounts.Create(nil);
    if Assigned(dmAccounts) then
      dmAccounts.DbDelphiBank.Open;
  except
  end;
end;

procedure TAccounts.OnDeactivate;
begin
  try
    if Assigned(dmAccounts) then begin
      dmAccounts.DbDelphiBank.Close;
      dmAccounts.Free;
    end;
    dmAccounts := nil;
  except
  end;
end;

```

Figure 4: Example *OnActivate* and *OnDeactivate* methods.

If you look at Figure 4, you'll see that is exactly what we do. In *OnActivate*, we create the object's data module and open the database connection. In *OnDeactivate*, we do exactly the opposite. You may think that opening and closing connections like this would be time consuming. However, if you open the BDE Administrator, go to the Configuration page, then open **System**, select **Init**, and change the option **MTS Pooling** to **True**, a little magic happens. This option does two things. First, the BDE will pool connections to databases from MTS objects, which means that new connections are quickly allocated from old connections. Secondly, MTS objects that use transactions and connect to databases through the BDE will automatically have their work enlisted in a database transaction. Therefore, it's very important that you turn this option on if you want full transactional support.

At the time of writing, there is much activity in the news-groups about MTS pooling in the BDE. Quite a few people

```

...
try
  CheckDbOpen;
  { Set up the query. }
  with dmAccounts do begin
    { Run the query. }
    QryAccountReadByCustID.ParamByName(
      cFieldCustID).Value := CustID;
    QryAccountReadByCustID.Open;
    { Package the result. }
    CreateVarArrayFromDataset (varResultSet,
      QryAccountReadByCustID);
    QryAccountReadByCustID.Close;
  end;
  SetComplete; { OK. }
  Result := cOKResult;
except on E: Exception do begin
  SetAbort; { Problem encountered. }
  dmAccounts.QryAccountReadByCustID.Close;
  strDebug := 'TAccounts.ListByCustID() - ' + E.Message;
  Result := cErrResult;
end;
end;

```

```

var
  TransactionsIntf : ITransactions;
begin
  ...
  try
    OleCheck(ObjectContext.CreateInstance(
      CLASS_Transactions,ITransactions,TransactionsIntf));
  except
    raise Exception.Create(
      'Could not create Transactions object');
  end;
  ...
end

```

Figure 5 (Top): A portion of the *TAccounts.ListByCustID* method.

Figure 6 (Bottom): Creating a *Transactions* object from inside an MTS object.

seem to be having problems with it — myself included. I'm not entirely convinced that this new feature is 100-percent solid. If you experience problems, try turning it off. Be aware, though, that if a transaction fails, it won't be rolled back. Finally, another important thing you should know is that you must put BDEMTS in the *uses* clause of your MTS objects. Without it, they won't take part in MTS transactions. The MTS object wizard should have done this automatically, but unfortunately, does not.

The *ListByCustID* method in *Accounts* takes a customer ID and returns a list of accounts for this customer (see [Figure 5](#)). Here, the *CheckDbOpen* method simply checks that the data module has been successfully created, and the database connection has been opened (by *OnActivate*). If this isn't the case, an exception is immediately raised and the method aborts. You can see from the code that if an exception occurs, a call to *SetAbort* is made, which stops the current transaction from committing. Any open queries are closed, and the *strDebug* variable is filled with the exception message. A non-zero return code is returned by the function to indicate failure. If the database has been opened successfully, a query is filled with the customer ID

and executed against the database. The result set is then packed into a variant array (more on this shortly), which is returned to the client.

Acquiring the Data

Querying the database is straightforward. Things get a little more tricky when we want to insert or update information. For instance, the *Transfer* method has to debit money from the source account, deposit it into the destination account, and fill out two bank transactions in the TRANSACT table. Moreover, it must enforce business rules and ensure it all happens within one transaction so money is never lost.

Luckily, MTS objects can enlist the help of other MTS objects, so if another object already implements the functionality you need, you can capitalize on it. Because the *Transactions* object implements functionality to add a new bank transaction with the *AddNew* method, that's exactly what we're going to do.

Here's the declaration of the *Transfer* function:

```

function TAccounts.Transfer(ClientKey, SrcAcctID,
  DestAcctID: Integer; const TrnRef: WideString;
  TrnAmount: Currency; ForceFailure: WordBool;
  var strDebug: WideString): Integer;

```

You can see it takes the usual client key, debug string, source and destination account, transaction reference, and the amount to transfer. It also takes a parameter named *ForceFailure*, which allows the caller to force a failure in the middle of the transfer to demonstrate and test transaction rollback. If a failure occurs, the balance of the source and destination accounts shouldn't change, and there should be no record of a bank transaction.

The first thing the *Transfer* function does, after checking that the database is open, is to create a *Transactions* object to help it complete its task (see [Figure 6](#)). We use the object context's *CreateInstance* function. We pass in the Class ID (GUID) of the *Transactions* COM object (created for us in the type library), the interface we want on the object, and the variable that will hold the interface pointer if the call succeeds. The COM utility function, *OleCheck*, checks the return status of the call, and raises an exception if the call fails. If the call succeeds, the created object is enlisted inside the *Accounts* current transaction.

[Figure 7](#) shows the code that makes the transfer. The *AddToBalance* function first checks that adding the transaction amount won't cause the balance of the account to drop below the minimum allowed (remember the amount can be negative). This is a business rule in action. If the rule is broken, an exception is raised and the transaction will be rolled back. Otherwise, a query is set up and executed, which adds the transaction amount to the destination account's balance. The *Transactions* object we created is then used to add a new bank transaction to the destination account. Finally, after checking for a forced failure, we proceed to deduct the transaction amount from the source account in a similar way, and add a bank transaction. If

```

{ Deposit the amount to the destination balance. }
if not AddToBalance(ClientKey, DestAcctID, TrnAmount,
  dmAccounts, strDebug) then
  raise Exception.Create(strDebug);

{ Deposit the money from the destination. }
if TransactionsIntf.AddNew(ClientKey, DestAcctID, TrnRef,
  TrnAmount, strDebug) <> cOKResult then
  raise Exception.Create(strDebug);

{ If we are testing a transaction failure, do it now. }
if ForceFailure then
  raise Exception.Create(
    'Forced failure - Amount deposited but not deducted.');
```

```

{ Deduct the amount from the source balance. }
if not AddToBalance(ClientKey, SrcAcctID, -TrnAmount,
  dmAccounts, strDebug) then
  raise Exception.Create(strDebug);

{ Deduct the money from the source. }
if TransactionsIntf.AddNew(ClientKey, SrcAcctID, TrnRef,
  -TrnAmount, strDebug) <> cOKResult then
  raise Exception.Create(strDebug);

SetComplete; { OK. }
Result := cOKResult;
TransactionsIntf := nil;
```

Figure 7: The core of TAccounts.Transfer.

all goes well, we make our call to *SetComplete*, and set our *Transactions* object interface pointer to *nil*, which releases it.

Getting Data Back to the Client

I promised earlier that I'd show you how we get a result set of data from a database query back to the client from inside an MTS object. Getting singular information back, such as the customer name or account balance, is easy because we simply use *var* parameters. A result set is different. It contains a variable number of columns of information of differing types, and a variable number of rows of actual information. A result set such as this can be represented with a two-dimensional array. However, we won't know anything about the type of data we're passing back until run time. Furthermore, the only data that can be passed back has to be of an Automation-compatible type so that COM can marshal it. Variants are Automation-compatible and can hold data of varying types. The solution then is a two-dimensional variant array.

From a given result set with *m* columns and *n* rows, we'll create and populate a variant array with its structure, as shown in Figure 8. The column display labels are always placed in row 0.

[0,0] Col Label	[1,0] Col Label	[2,0] Col Label	[m-1,0] Col Label
[0,1] Row 0 Data	[1,1] Row 0 Data	[2,1] Row 0 Data	[m-1,1] Row 0 Data
[0,2] Row 1 Data	[1,2] Row 1 Data	[2,2] Row 1 Data	[m-1,2] Row 1 Data
[...]	[...]	[...]	[...]
[0,n-1] Row n-2 Data	[1,n-1] Row n-2 Data	[2,n-1] Row n-2 Data	[m-1,n-1] Row n-2 Data
[0,n] Row n-1 Data	[1,n] Row n-1 Data	[2,n] Row n-1 Data	[m-1,n] Row n-1 Data

Figure 8: Structure of the variant array result set. The array is [0..m-1, 0..n] for *m* columns and *n* rows.

The rest of the rows, if any, are populated with the values from the result set. Advanced information, such as display width and custom constraints — the kind of thing you get in a *TField* — isn't passed back using this method. That doesn't mean you couldn't write some code to do it. Admittedly, it may not be as pretty, or as easy, as using *TDatasets* and data-aware controls, but this is the price you pay for maximizing server resources.

Incidentally, the MIDAS services that come with Delphi 4 are aimed at this area. They take the hassle out of getting information back from the Business Services layer, but at the expense of a server license when you deploy. Our way of doing things doesn't use MIDAS. We have to move data back ourselves, but it's a whole lot cheaper. The method that does the packaging of a dataset into a variant array is named *CreateVarArrayFromDataset*. It's a member of the *TDmDelphiBankCommon* data module. You pass in an open dataset from a query or stored procedure, and it returns a created and fully populated two-dimensional variant array that can be sent back to the client (again, see Figure 8).

We've looked at the server in some depth. Before we move on and look at the client, however, let's examine the object context methods in detail so you're familiar with all of them.

The Object Context in Detail

The object context can be directly accessed with the *GetObjectContext* function. This returns an *IObjectContext* interface (its methods are shown in Figure 9). We've already seen *CreateInstance*, *SetComplete*, and *SetAbort* in action. The *IsSecurityEnabled* and *IsCallerInRole* methods are used when employing MTS security features, which we'll see next month in Part III. *IsInTransaction* returns True or False to indicate if the current object is executing inside a transaction.

When you create an MTS object, you must specify its transactional requirements:

- Requires — MTS will make sure the object runs in a transaction, creating a new one if necessary.
- Requires New — MTS will create a new transaction for the object even if the client already has one running.
- Supported — Indicates the object doesn't necessarily need a transaction, but will work quite happily inside one if the client has one running.
- Does Not Support — The object is probably old legacy code and doesn't understand MTS transactions, i.e. it doesn't call *SetComplete* or *SetAbort*.

Function	Description
<i>CreateInstance</i>	Instantiates another MTS object. All context information is transferred to the created object, so the new object will be enlisted in any current transaction.
<i>EnableCommit</i>	Tells MTS the object hasn't necessarily finished its work, but the transaction can be committed in its current form. The object is not deactivated and retains its state between method calls.
<i>DisableCommit</i>	Tells MTS the object hasn't necessarily finished its work, and that the transaction cannot be committed in its current form. The object isn't deactivated, and retains its state between method calls.
<i>SetComplete</i>	Tells MTS the object has finished its work, and the transaction can be committed. The object is deactivated on exit from the method.
<i>SetAbort</i>	Tells MTS the object has finished its work, but the transaction can never be committed. The object is deactivated on exit from the method.
<i>IsCallerInRole</i>	Indicates if any object's direct caller is in a particular role (used in security).
<i>IsInTransaction</i>	Indicates if the object is currently taking part in a transaction.
<i>IsSecurityEnabled</i>	Indicates whether MTS security is enabled.

Figure 9: The *IObjectContext* interface methods.

The remaining two methods in the *IObjectContext* interface are *EnableCommit* and *DisableCommit*. Remember stateful and stateless components? Well, if you're writing a stateful component, you'll want your object to retain its state between method calls. So, instead of calling *SetComplete* or *SetAbort* (which deactivates the object), call these. The object won't be deactivated when the method finishes, and properties in your object will keep their values. The difference between the methods is transactional. With *EnableCommit*, you're signifying that any currently running transaction could be committed by MTS in its current form. With *DisableCommit*, you're forbidding the current transaction from being committed until you say so.

The Client

This completes our discussion of the server. All we need now is a client. The client itself was written by an excellent Delphi programmer and good friend of mine, Mark Smith. The interesting thing about this is that I created the type library, stubbed all the functions with a "Not Implemented" message, and gave it to him. I then proceeded to implement the server functionality while he simultaneously developed the

client. It really does work well this way, and it forces you to think very hard about your problem domain before you can give someone a type library full of methods!

I'm not going to say too much about the client code, as I prefer to concentrate on MTS server functionality. Suffice it to say, it imports the type library from the server, and has some classes that support creating and freeing of instances of the server objects. The actual implementation of the client is an MDI-style application, as shown in **Figure 10**. You can see I have logged in as myself (user ID "paul"; password "paul") and as William Gates (user ID "william"; password "william"). With each login ID is presented a list of accounts and their details, such as current balance, last statement date, etc. If you double-click on an account, you're presented with all the transactions on the account since the last statement. The application allows you to pay a bill, deposit money, and transfer money from one account to another. When you perform such an operation, a message is sent to all windows in the application that are interested in any account you just modified to notify them that a change has occurred. So if you open the bank transactions window for account A, and transfer money from account A to account B, the bank transactions window would refresh and immediately display the new transfer. The whole application is strictly a front-end for the server objects, and performs no verification of what you enter — that's the Business Services layer's task.

The amazing thing is, if this was a delivered system, and the customer wasn't happy with the current client (say they wanted a Web version), you could completely scrap it and not lose any Business Services or Data Services functionality. The division between the client and server is so clean that the MTS components would remain intact, and work perfectly for the new client. In fact, the client itself needn't be a Delphi appli-

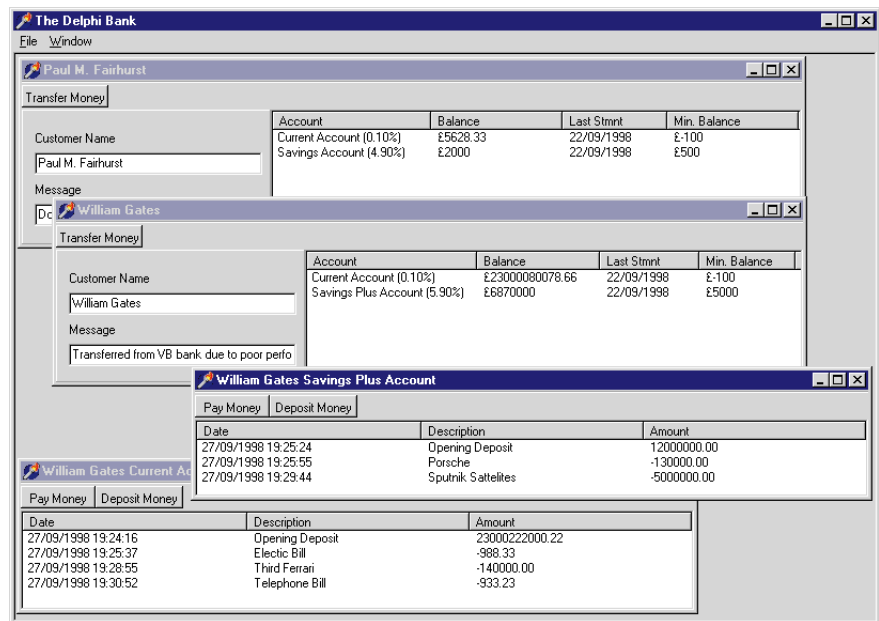


Figure 10: The DelphiBank client in action.

cation. Because the objects are COM objects, you could drive them with, say, Microsoft Active Server Pages, or a Cold Fusion Web server. That's the sort of code reuse the industry desperately needs.

Conclusion

We've covered a lot of ground in Part II. We've seen how to structure a three-tier MTS application. We've talked about the *IObjectContext* interface and its methods. We've seen how one MTS object can use another while keeping the work inside one transaction. We also learned how to get result data from a dataset back to the client, and we've looked at a real client using our component's services in a real-world scenario.

In the **third** and final part of this series, we'll take a look at how MTS security can block unauthorized calls to your components and how Roles can help you implement business logic by grouping together related users. We'll look at the MTS explorer in more detail, and see how it monitors and provides statistics for transactions. We'll also see how easy it is to move our client to another machine by using the MTS client installation export utility. Finally, we'll look at the Shared Property manager, which allows objects to quickly share information with each other. **△**

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\JAN\DI9901PF.

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





DBNAVIGATOR

TDataSet / TField

By Cary Jensen, Ph.D.



Delphi Database Development

Part V: Navigation and Editing

Over the past few months, this column has undertaken a systematic re-examination of Delphi database development. This month's installment of "DBNavigator" continues this series with a look at basic dataset navigation and editing. You might recall that Table components, as well as many types of Queries and StoredProc components, return a set of one or more records. This is also true of ClientDataSet components. These components permit you to move a cursor between the various records in the set, and to make changes to these records (if they aren't set to read-only).

(It's worth noting that some Query and StoredProc components don't reference a set of records. For example, a Query that holds a SQL CREATE TABLE statement doesn't have a cursor for a set of records. These datasets are special, and the descriptions found in this month's "DBNavigator" don't apply.)

Basic Navigation

When a DataSet is first opened, either by setting its *Active* property to True or by calling

its *Open* method, the internal cursor for the DataSet is set to point to the first record in the set. It's then possible to navigate the DataSet using the methods *First*, *Last*, *Next*, and *Prior*. These same methods are called by components, such as *TDBNavigator*, when the user clicks on the corresponding buttons.

There are other methods that can be used to navigate records. These include *MoveBy*, which permits you to move forward and backward by some number of records that you specify. Other methods move the cursor by searching records based on their contents. These methods include *FindKey*, *FindNearest*, and *Locate*. These searching methods will be discussed in detail in next month's "DBNavigator."

There are two Boolean properties that are also useful when your code needs to navigate records. These are *BOF* (beginning of file) and *EOF* (end of file). *BOF* returns True when your code has attempted to move beyond the beginning of a table. This happens when you're already on the first record and you issue a call to the *Prior* method, or you call *MoveBy* with a negative parameter

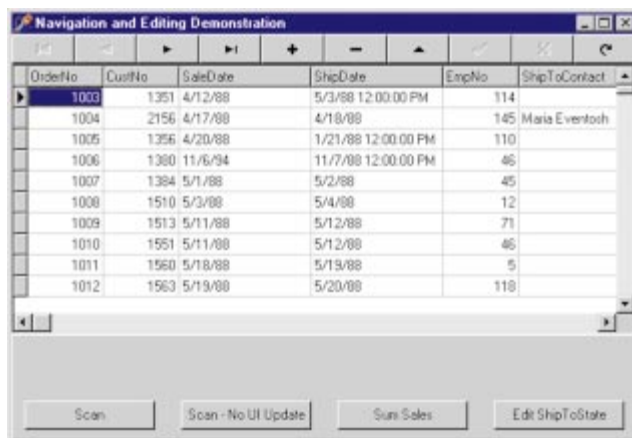


Figure 1: The EDITDEMO project demonstrates basic navigation and record editing.

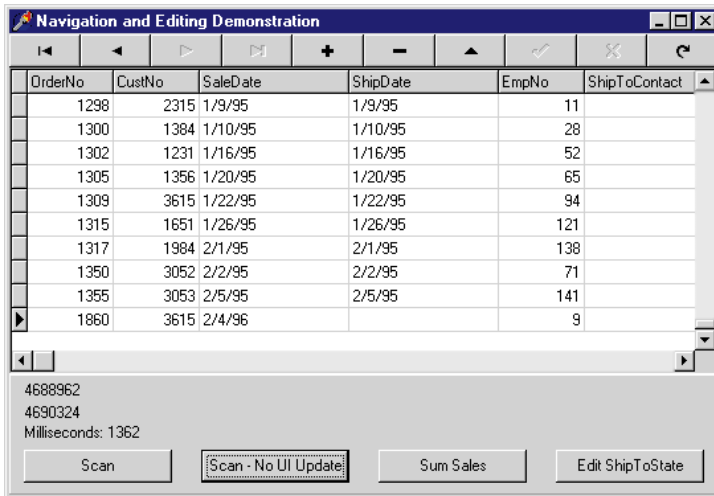


Figure 2: Code associated with the **Scan** button displays the scanning speed.

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Enabled := False;
  LabelStart1.Caption := IntToStr(GetTickCount);
  Table1.First;
  while not Table1.EOF do begin
    Table1.Next;
  end;
  LabelEnd1.Caption := IntToStr(GetTickCount);
  LabelTotal1.Caption := 'Milliseconds: ' +
    IntToStr(StrToInt(LabelEnd1.Caption) -
      StrToInt(LabelStart1.Caption));
  Button1.Enabled := True;
end;

```

Figure 3: The **Scan** button's *OnClick* event handler.

whose value is less than the current absolute record number. Likewise, *EOF* returns True when you attempt to move beyond the end of a table. Similar to what happens with *BOF*, this occurs if you call *Next* while on the last record of a table, or call *MoveBy* with a positive parameter whose value is greater than the number of records remaining in the table. Merely moving your cursor to the first or last record isn't sufficient to cause a *BOF* or *EOF* to return True, respectively. You must actually attempt to move beyond the first or last record for this to happen.

The following code example demonstrates the use of *First*, *Next*, and *EOF*. This code navigates to every record in a table pointed to by the Table component named *Table1*, beginning with the first record in the index order:

```

Table1.First;
while not Table1.EOF do begin
  // Do something with the record.
  Table1.Next;
end;

```

This type of operation is often referred to as a "scan." In a real scan, your code would do something as it visits each record, such as accumulate a summary calculation, or make a change

to some or all records visited. Once the basic data read and write operations are described, more meaningful versions of this scan will be shown.

The use of this simple scan is demonstrated in the EDITDEMO project, shown in [Figure 1](#) (available for download; see end of article for details). This project contains a Table component that points to the ORDERS table from the sample files installed with Delphi. The contents of the Table component are displayed in a DBGrid. If you click on the **Scan** button, you'll see visible results of the scan operation, as the records of the DBGrid scroll rapidly from the first to the last records in the ORDERS table.

The code associated with the **Scan** button also includes instructions that display an indication of the scan's performance. This display involves three Label components. One Label displays the operating system tick count immediately before the scan operation, the second displays the tick count following the completion of the scan, and the third displays the number of ticks that passed during the scan. Each tick represents a millisecond. [Figure 2](#) shows the results of a typical scan of the ORDERS table on a 150 MHz machine with 64MB of RAM. In this case, the entire scan requires slightly longer than one second (1,362 milliseconds, to be precise). The code shown in [Figure 3](#) is associated with this button's *OnClick* event handler.

While being able to watch the records scroll may be entertaining, it slows down the scanning process tremendously. This is because the DBGrid must be repainted each time the cursor moves to another record. Painting operations are among the most resource intensive, so they should be avoided if possible. Fortunately, the *DataSet* classes include the *DisableControls* method, which you can use to suppress the notification that the dataset otherwise sends to any associated DataSources. Because it's the DataSource that instructs the DBGrid to repaint itself after a DataSet record navigation, calling *DisableControls* permits a DataSet to navigate without repainting data-aware controls.

There is a danger associated with using *DisableControls*, however. Specifically, you must be sure to call *EnableControls* following a call to *DisableControls*, or your data-aware controls will be inoperative. Unfortunately, simply including a call to *EnableControls* following a scan isn't sufficient. You must include the call to *EnableControls* within the **finally** portion of the **try..finally** block, which you enter immediately following the call to *DisableControls*. This approach is necessary to guarantee the execution of *EnableControls* if an exception occurs following the call to *DisableControls*, but before *EnableControls* is executed.

The use of *DisableControls* and *EnableControls* with a scan is demonstrated in the **Scan - No UI Update** button in the example project. The code shown in [Figure 4](#) is from this button's

```

procedure TForm1.Button2Click(Sender: TObject);
begin
  Button2.Enabled := False;
  Table1.DisableControls;
  LabelStart2.Caption := IntToStr(GetTickCount);
  try
    Table1.First;
    while not Table1.EOF do begin
      Table1.Next;
    end;
    LabelEnd2.Caption := IntToStr(GetTickCount);
    LabelTotal2.Caption := 'Milliseconds: ' +
      IntToStr(StrToInt(LabelEnd2.Caption) -
        StrToInt(LabelStart2.Caption));
  finally
    Table1.EnableControls;
    Button2.Enabled := True;
  end;
end;

```

Figure 4: Code for the **Scan - No UI Update** button's *OnClick* event handler.

OnClick event handler. Figure 5 shows how the example project might look after clicking this button. Note the enormous performance improvement over the scan without *DisableControls*. When *DisableControls* was used, the scan took 10 milliseconds, or 1/100th of a second — a fraction of the time required when *DisableControls* wasn't used.

Reading Data

While the *TDataSet* classes permit you to navigate data, they don't provide the ability to read the data. That capability is provided through a collection of classes referred to as *TFields*. While there is a class named *TField*, you don't work directly with it. Instead, you work with one of its descendant classes, such as *TStringField*, *TBlobField*, or *TIntegerField*.

When a dataset is first opened, one *TField* descendant is created for each of the columns in the underlying data file. Which *TField* descendant is created depends on the data type of the corresponding column, which is dynamically determined by the BDE. Each *TField* can be used to read the data in the associated column of the dataset. In addition, if a particular *TField* is not read-only, and the dataset can be edited, you can also write to the field using *TField*.

There are two general techniques you can use to access the *TFields* of a dataset. The first is to use the properties and methods of the dataset component itself. The *Fields* property of a dataset is an array of pointers to *TFields* that are automatically created when the dataset is opened. The *TFields* in *Fields* are indexed based on the structure of the underlying table. For example, *Table1.Fields[0]* references the *TField* associated with the first field in the table, while *Table1.Fields[2]* references the third.

Using the *Fields* property gives you the most efficient access to the individual fields of a dataset, but these references aren't very informative. If you'd rather refer to a *TField* based on the field's name, use the *FieldByName* method, which takes a single string parameter. When calling *FieldByName*, you pass the name of

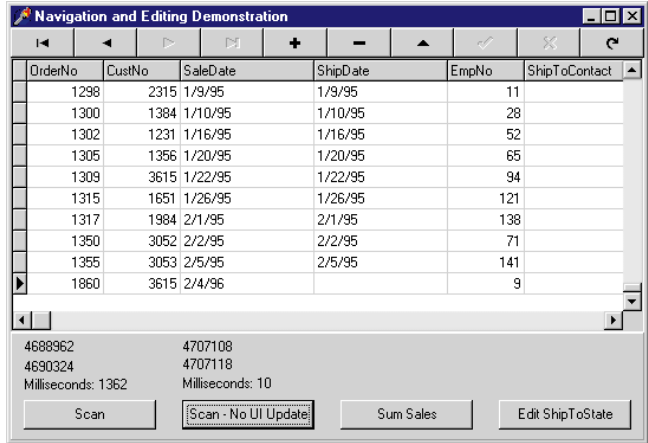


Figure 5: Using *DisableControls* greatly improves scan performance.

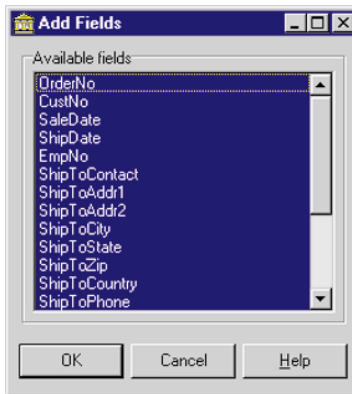


Figure 6: Use the Add Fields dialog box to add individual fields to the Fields Editor.

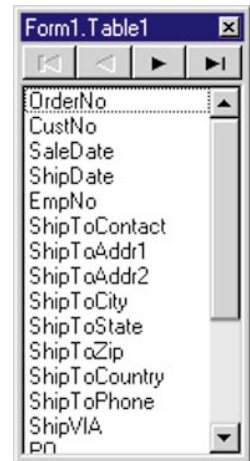


Figure 7: The Fields Editor lists the currently design-time-instantiated *TFields* for a dataset.

the underlying field to which you want access, and it returns a *TField* reference to that field. For example, to reference the *OrderNo* field of a table named *Table1*, you can use *Table1.FieldByName('OrderNo')*. Accessing *TField* descendants using *FieldByName* is slower than using the *Fields* property, because, internally, *FieldByName* must first look up the ordinal position of the field in the table's structure. However, *FieldByName* is easier to read, thereby reducing the number of errors due to incorrect column references.

The second means of accessing the *TFields* of a dataset is to instantiate the individual *TField* components at design time. You do this using the Fields Editor, which is displayed when you right-click a dataset and select **Fields Editor** (or simply double-click on it). From the Fields Editor, right-click and select **Add Fields**. This displays the Add Fields dialog box, shown in Figure 6. Select the fields you want to add, then click **OK**. The instantiated fields then appear in the Fields Editor, shown in Figure 7. (In Delphi 4, you can instantiate all fields simply by displaying the Fields Editor and pressing **Ctrl(F)**.)

The *TFields* instantiated using the Fields Editor are given default names produced by adding the name of the dataset to the name of the field. For example, in the EDITDEMO project, the instantiated *OrderNo* field is named

```

procedure TForm1.Button3Click(Sender: TObject);
var
    SumofSales: Currency;
begin
    SumOfSales := 0;
    Button3.Enabled := False;
    LabelStart3.Caption := IntToStr(GetTickCount);
    Table1.DisableControls;
    try
        Table1.First;
        while not Table1.EOF do begin
            SumOfSales := SumOfSales +
                Table1.FieldName('ItemsTotal').AsCurrency;
            Table1.Next;
        end;
        LabelEnd3.Caption := IntToStr(GetTickCount);
        LabelTotal3.Caption := 'Milliseconds: ' +
            IntToStr(StrToInt(LabelEnd3.Caption) -
                StrToInt(LabelStart3.Caption));
    finally
        Table1.EnableControls;
        Button3.Enabled := True;
        ShowMessage(FormatCurr('$ #,###.##', SumOfSales));
    end;
end;
    
```

Figure 8: A scan demonstration.

Table1OrderNo. When you use this reference in your code, you access the OrderNo column for whichever record the corresponding dataset is currently pointing to. This is always the case. A *TField* represents a column, but the dataset references the record.

To use a *TField* descendant, reference its properties. For example, the *AsString* property of a *TIntegerField* reads the contents of the corresponding column as a string, while the *Value* property returns the contents of the underlying column as a variant. While *TField* descendants have methods, most of these are intended for the internal use of the component.

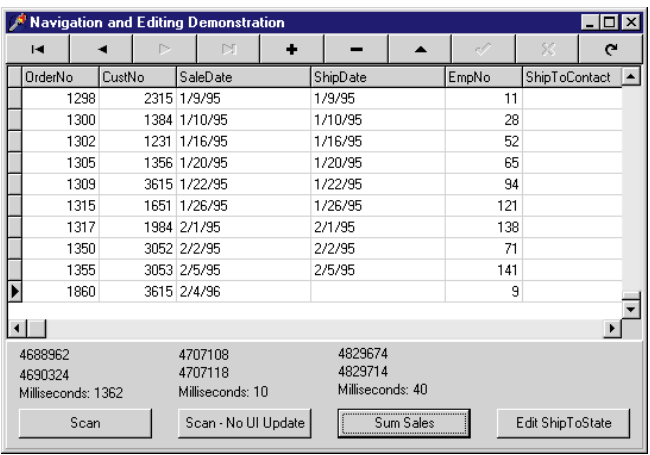
The code shown in **Figure 8** demonstrates a scan. During the scanning operation, the value of the ItemsTotal field is accumulated in a *Currency* variable. The total value is displayed following the end of the scan. The performance of the scan when data is being read is shown in **Figure 9**.

Note: This type of operation — calculating the sum of a field — can be performed using a SQL query. In fact, many of the features you produce using a scan can also be produced using SQL. Indeed, in many instances, a SQL query is much faster than a scanning operation. However, there are situations where SQL queries can't be used, such as when you want to perform a scan on the result set returned by a query.

Editing Data

Editing data during a scan is more involved than simply reading data. In short, you must place a dataset in the *dsEdit* state before changing a record, and you must explicitly post the change before attempting to move off the record. Posting returns a dataset to the *dsBrowse* state.

You place a dataset in the *dsEdit* state by calling its *Edit* method. For some databases, such as Paradox, placing a



```

procedure TForm1.Button4Click(Sender: TObject);
var
    Undo: Boolean;
begin
    Table1.First;
    if Table1.FieldName('ShipToState').AsString = '' then
        Undo := False
    else
        Undo := True;
        Button4.Enabled := False;
        LabelStart4.Caption := IntToStr(GetTickCount);
        Table1.DisableControls;
        try
            Table1.First;
            while not Table1.EOF do begin
                try
                    Table1.Edit;
                    if Undo then
                        Table1.FieldName('ShipToState').Value := ''
                    else
                        Table1.FieldName('ShipToState').Value :=
                            Table1.FieldName('PaymentMethod').Value +
                            ' ' + Table1.FieldName('OrderNo').AsString;
                    Table1.Post;
                    Table1.Next;
                except
                    // Handle a failure to edit or post here. In this
                    // example, ignore records that cannot be edited or
                    // posted.
                end; // try-except
            end; // begin

            LabelEnd4.Caption := IntToStr(GetTickCount);
            LabelTotal4.Caption := 'Milliseconds: ' +
                IntToStr(StrToInt(LabelEnd4.Caption) -
                    StrToInt(LabelStart4.Caption));
        finally
            Table1.EnableControls;
            Button4.Enabled := True;
        end; // try-finally
    end;
    
```

Figure 9 (Top): Scan performance while data is being read.
Figure 10 (Bottom): Code associated with the **Edit ShipToState** button in the EDITDEMO project to enclose the editing of each record within its own **try..except** block.

dataset in the *dsEdit* state creates a record lock. For tables such as these, if a record that you want to edit is already locked by another user, the call to *Edit* fails, raising an exception.

Even for datasets that don't place pre-emptive locks when you enter the *dsEdit* state, the post operation can still fail, also by raising an exception. This will occur if the record being posted is rejected by the underlying dataset due to key violations or invalid data. Likewise, if the record being posted was

OrderNo	CustNo	SaleDate	ShipDate	EmpNo	ShipToContact
1298	2315	1/9/95	1/9/95	11	
1300	1384	1/10/95	1/10/95	28	
1302	1231	1/16/95	1/16/95	52	
1305	1356	1/20/95	1/20/95	65	
1309	3615	1/22/95	1/22/95	94	
1315	1651	1/26/95	1/26/95	121	
1317	1984	2/1/95	2/1/95	138	
1350	3052	2/2/95	2/2/95	71	
1355	3053	2/5/95	2/5/95	141	
1860	3615	2/4/96		9	

4688962	4707108	4829674	4864044
4690324	4707118	4829714	4864364
Milliseconds: 1362	Milliseconds: 10	Milliseconds: 40	Milliseconds: 320

Figure 11: Writing to a table during a scan increases scan duration over simply reading.

updated by another user (after the record was read, but before it was posted), the posting will fail.

If you write scanning code that writes to a dataset, you must ensure that exceptions raised by the *Edit* and *Post* methods are handled. This requirement is in addition to the previously discussed **try..finally** block that ensures the eventual call to *EnableControls*. A typical scenario is to enclose the editing of each record within its own **try..except** block, handling any failures within the **except** clause. If you want to continue to scan additional records — even if one or more records can't be updated due to exceptions — this **try..except** structure must appear inside the scan's **while** loop. This technique is demonstrated by the code associated with the **Edit ShipToState** button in the EDITDEMO project (see [Figure 10](#)). [Figure 11](#) depicts how the project appears after clicking this button.

Admittedly, this example is contrived; it alternates between assigning a value to the ShipToState field, and then clearing the field. This meaningless edit was performed so that if you download this example project and run it, you can always return the ORDERS table to its original state (where the ShipToState field is blank for every record). However, the example does serve its intended purpose by demonstrating the basics of record editing during a scan operation.

Conclusion

Using Delphi's various DataSet components, you can easily navigate between the multiple records of a table. The process of reading and writing the values of columns, however, makes use of *TField* descendants. *TFields* can be accessed by the *DataSet.Fields* property, the *DataSet.FieldByName* method, or by working directly with design-time instantiated *TField* components.

In next month's "DBNavigator," this series continues with a look at record searching methods. [▲](#)

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\JAN\DI9901CJ.

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





By Rod Stephens



Tree Management

The Care, Feeding, and Implementation of Delphi Trees

Last month's algorithmic excursion involved networks. Using node and link classes, it showed how to implement a dynamic network data structure, and use that structure to find the shortest path through a network. This article discusses another kind of dynamic data structure: the tree. Trees are well suited for managing hierarchical relationships among data items. They can also be used to maintain sorted lists while allowing fast searching and retrieval. This article explains trees, and shows how you can use different tree structures in your Delphi programs.

Tree Terms

Before you study trees, you should know some basic terminology. You can define a *tree* recursively as a data structure with a *root node* connected to zero or more subtrees. Trees are customarily drawn with the root node at the top, as shown in Figure 1. The tree in Figure 1 consists of a root node labeled A connected to two subtrees with roots B and C. The subtree with root B has a root node connected to two subtrees with roots D and E. The *trivial* trees with roots C, D, and E are connected to no other nodes.

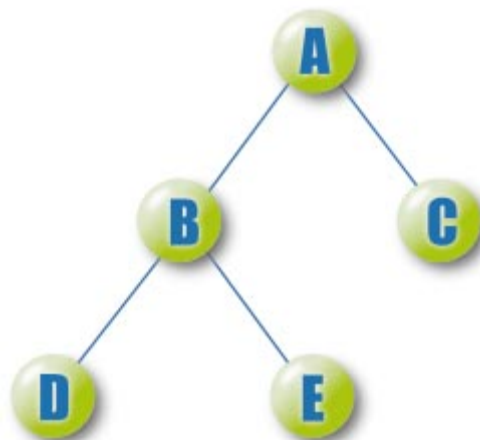


Figure 1: A tree.

The nodes in a tree are connected by links or *branches*. Each branch defines a parent-child relationship between the two *nodes* it connects. The upper node is the *parent* and the lower node is the *child*. The nodes along the path from a node to the root are the node's *ancestors*. Conversely, if a node is another's ancestor, then that node is the ancestor's *descendant*. For example, in Figure 1 nodes A and B are ancestors of node E. Nodes D and E are the descendants of node B.

The *degree* of a node is the number of children it has. The degree of the tree is the largest degree of any node. Trees of degree two are called *binary trees* and trees of degree three are sometimes called *ternary trees*. A leaf is a node with degree 0 (like nodes C, D, and E). All other nodes are called *internal nodes*.

A node's *height* or *depth* in the tree is one plus the number of its ancestors. The height of the tree is the largest of the nodes' heights. The tree in Figure 1 has degree 2 (nodes A and B have degree 2) and height 3 (nodes D and E have height 3).

As you can see, tree terminology is a mishmash of words pirated from genealogy (parent, child, ancestor) and botany (branch, node, leaf).

```

var
  root : TBinaryNode;

// Build a tree like the one in Figure 1.
procedure BuildFig1Tree;
var
  child : TBinaryNode;
begin
  // Create the root node A.
  root := TBinaryNode.Create;
  root.node_value := 'A';
  // Create node B.
  child := TBinaryNode.Create;
  child.node_value := 'B';
  root.left_child := child;
  // Create node C.
  child := TBinaryNode.Create;
  child.node_value := 'C';
  child.left_child := nil;
  child.right_child := nil;
  root.right_child := child;
  // Create node D.
  child := TBinaryNode.Create;
  child.node_value := 'D';
  child.left_child := nil;
  child.right_child := nil;
  root.left_child.left_child := child;
  // Create node E.
  child := TBinaryNode.Create;
  child.node_value := 'E';
  child.left_child := nil;
  child.right_child := nil;
  root.left_child.right_child := child;
end;

```

Figure 2: Code that builds the binary tree shown in Figure 1.

Planting Trees

You can represent a tree in Delphi using a class to represent the tree nodes. The class defines the node's value and includes references to the node's children. For example, you could use the following code to define a binary node class:

```

type
  String10 = string[10];
  TBinaryNode = class(TObject)
  public
    node_value : String10;
    left_child, right_child : TBinaryNode;
  end;

```

Using this class, a Delphi program can build a binary tree. For example, the code shown in Figure 2 builds a tree similar to the one shown in Figure 1.

To build a tree of larger degree, you can add more child references to the node class. For example, you might define a node for a ternary tree like this:

```

type
  String10 = string[10];
  TBinaryNode = class(TObject)
  public
    node_value : String10;
    left_child, middle_child, right_child : TBinaryNode;
  end;

```

For trees of higher degree, using separate child variables is cumbersome. You can build a more flexible node class by

```

// Find the node with a particular value.
function TBigNode.FindNodeValue(target_value: String10):
  TBigNode;
var
  i      : Integer;
  child  : TBigNode;
begin
  // Assume we will not find the target.
  Result := nil;
  // See if this is the node we want.
  if (node_value = target_value) then
    // It is. Return this node.
    Result := Self;
  else
    // It is not. Make the children search for it.
    for i := 0 to children.Count - 1 do begin
      child := children.Items[i];
      // See if this child can find the target.
      Result := child.FindNodeValue(target_value);
      // If the child found it, we are done.
      if (Result <> nil) then break;
    end;
  end;
end;

```

Figure 3: This code searches a subtree for a target value.

using a linked list, *TList*, or some other expandable object to store each node's children:

```

type
  String10 = string[10];
  TBigNode = class(TObject)
  public
    node_value : String10;
    children   : TList;
  end;

```

Tree Recursion

The beginning of this article defined a tree recursively as a root node connected to zero or more subtrees. That recursive nature translates into many useful tree operations. A program can implement many tree operations by invoking a simple method provided by the root node. That node can perform the operation, or pass the request down to its children. They in turn can handle the request, or pass it down to their children. The request continues moving down through the tree until a node handles it or every node has passed up the chance and the task remains unhandled.

For example, suppose you have a large tree that uses the *TBigNode* class described earlier. This class stores child nodes using a *TList* object. The function *FindNodeValue*, shown in Figure 3, searches the subtree rooted at a specific node looking for a target value. It returns the node with the indicated value.

FindNodeValue first determines whether the current node has the target value. If so, it sets its *Result* value to this node. If this node does not have the target value, the function examines each of the node's children. For each child, *FindNodeValue* recursively invokes the child's *FindNodeValue* function. That makes the child search its subtree for the target value. If any child successfully finds the target, the function stops looking.

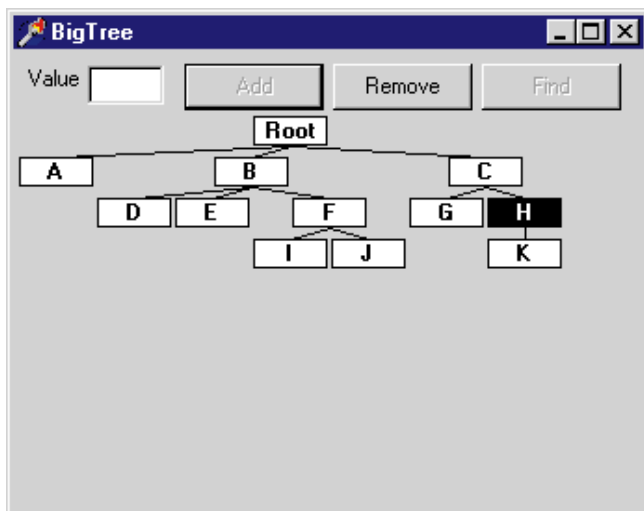


Figure 4: The example program BigTree lets you add, remove, and search for nodes.

A Delphi program can search an entire tree for a target value simply by invoking the root node's *FindNodeValue* function:

```
target_node := root.FindNodeValue(txtValue.Text);
```

This is typical of many tree operations. A program can perform the operation for an entire tree by invoking a recursive procedure defined by the tree's root node.

Testing Trees

The example program BigTree, shown in [Figure 4](#), lets you manage a tree. Enter a value in the text box, select a node, and click the **Add** button to add a new child beneath the node you have selected. Select a node other than the root and click the **Remove** button to remove the node and all of its descendants from the tree. Enter a value and click the **Find** button to make the program search for the value.

The BigTree program uses several recursive routines that work much the same as the *FindNodeValue* function described earlier. One of the more important of these is the class destructor shown in [Figure 5](#). When it's invoked, the destructor frees each of the node's children. The children free their children, which free their children, and so forth, until every node in the subtree is freed. This destructor makes it easy for a program to destroy an entire tree. All it needs to do is free the root node, and the rest of the tree is destroyed automatically.

The *TBigNode* class used by the program BigTree uses a *TRect* variable named *position* to define the node's placement on the form. One of the more interesting parts of the class initializes the positions for the nodes in the tree. It can then use the positions to draw the tree.

The *SetPosition* procedure sets the position for a node. This procedure, shown in [Figure 6](#), recursively sets the positions of the node's children. It then centers the node over its children's subtrees.

```
// Free any children and the children list.
destructor TBigNode.Destroy;
var
  i      : Integer;
  child  : TBigNode;
begin
  // Free the children.
  for i := 0 to children.Count - 1 do begin
    child := children.Items[i];
    child.Free;
  end;
  // Free the children list.
  children.Free;
  inherited Destroy;
end;
```

```
const
  BOX_WID = 40;
  BOX_HGT = 16;
  BOX_HGAP = 2;
  BOX_VGAP = 6;

// Position the node and its descendants. Update start_x so
// it indicates the rightmost position used by the node and
// its descendants.
procedure TBigNode.SetPosition(var start_x : Integer;
  start_y : Integer);
var
  i, xmin : Integer;
  child   : TBigNode;
begin
  // Set the node's top and bottom.
  position.Top := start_y;
  position.Bottom := start_y + BOX_HGT;
  // Record the leftmost position used.
  xmin := start_x;
  // If there are no children, put the node here.
  if (children.Count = 0) then
    start_x := xmin + BOX_WID;
  else
    begin
      // This is where the children will start.
      start_y := start_y + BOX_HGT + BOX_VGAP;
      // Position the children.
      for i := 0 to children.Count - 1 do begin
        // Position this child.
        child := children.Items[i];
        child.SetPosition(start_x, start_y);
        // Add a little room before the next child.
        start_x := start_x + BOX_HGAP;
      end;
      // Subtract the gap after the last child.
      start_x := start_x - BOX_HGAP;
    end;
  // Center this node over its children.
  position.Left := (xmin + start_x - BOX_WID) div 2;
  position.Right := position.Left + BOX_WID;
end;
```

Figure 5 (Top): By freeing the node's children, the *TBigNode*'s destructor recursively destroys an entire subtree.

Figure 6 (Bottom): The subroutine *SetPosition* recursively positions the node's children and then centers the node over them.

SetPosition takes as parameters the minimum X and Y coordinates it's allowed to use to position the node. As the procedure moves through the tree, these values are updated so a node always lies below its parent, and to the right of any previously positioned subtrees at or below its depth in the tree. When it

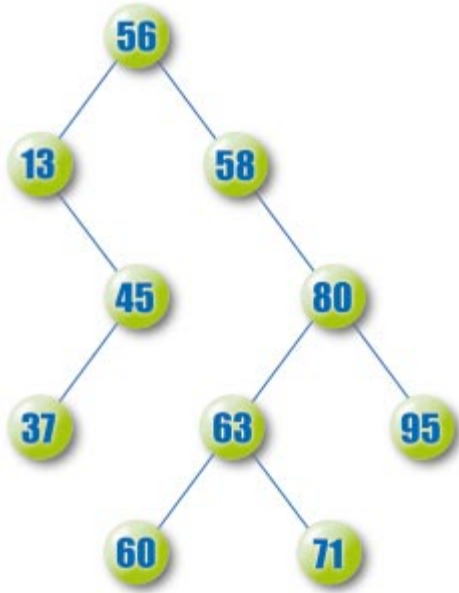


Figure 7: A sorted binary tree.

finishes, *SetPosition* leaves the *start_x* parameter holding the value of the maximum X coordinate it used to position its subtree. This becomes an input for future calls to the procedure.

SetPosition procedure begins by setting the node's top and bottom position values. These are determined solely by the position of the node's parent. *SetPosition* then saves the minimum X coordinate value it is allowed to use. If the node has no children, the procedure adds the width of a node, *BOX_WID*, to the minimum X coordinate to allow room to draw its node.

If the node has children, the procedure recursively invokes each child's *SetPosition* procedure to position the subtree rooted at the child. It increases the minimum Y coordinate the children can use by *BOX_VGAP*, so there will be some vertical space between the node and its children.

Each call to a child's *SetPosition* procedure updates the variable *start_x* to indicate the maximum X coordinate used to position that child's subtree. Between each child's call, *SetPosition* adds the amount *BOX_HGAP* to *start_x* to set the minimum X value the next child can use to position itself. This puts some space between the child subtrees.

When it finishes positioning the node's children, *SetPosition* subtracts the amount *BOX_HGAP* that it added to *start_x* after it positioned the last child's subtree. At this point, the variable *start_x* holds the largest X coordinate used by the last child's subtree.

SetPosition finishes by centering its node over the minimum and maximum X coordinate values used by all of the children. You can see the result in [Figure 4](#). For example, node F is centered over the subtrees rooted at its children, I and J. These subtrees both include a single node, so it's easy to see that node F is properly centered.

Note that node B is not centered over its children, nodes D, E, and F. Instead it's centered over the subtrees rooted at those children. These subtrees include X coordinates ranging from the left edge of node D to the right edge of node J, at the bottom of the F subtree.

The program *BigTree* uses similar recursive techniques to draw the tree (each node recursively draws its subtree), determine which node was clicked by the user (each node recursively decides whether the clicked point lies within its subtree), and delete a node (each node recursively searches its subtree for the target node).

Another Sort of Tree

There are many different kinds of trees with different degrees and different management policies. The B+tree, for example, is a high degree tree that is commonly used by databases to store table indexes. A simpler example is the sorted binary tree. In this kind of tree, items are arranged so each node's value is greater than its left child's and smaller than its right child's. [Figure 7](#) shows a small sorted tree.

Adding a node to a sorted binary tree is not quite as easy as adding one to an unsorted tree. To add a new value, the program begins at the tree's root. It compares the new node's value to the root's value. If the new value is smaller, the program continues to search for the node's location by examining the root's left child. If the new value is larger than the root's value, the program searches for the node's location by examining the root's right child. The program continues searching down the tree until the node it wants to examine is not present. It then inserts the new node at that point.

For example, suppose you want to insert the value 48 in the tree shown in [Figure 7](#). The program first compares 48 to the root node's value 56. Because $48 < 56$, the program searches the root's left child. It then compares 48 to 13.

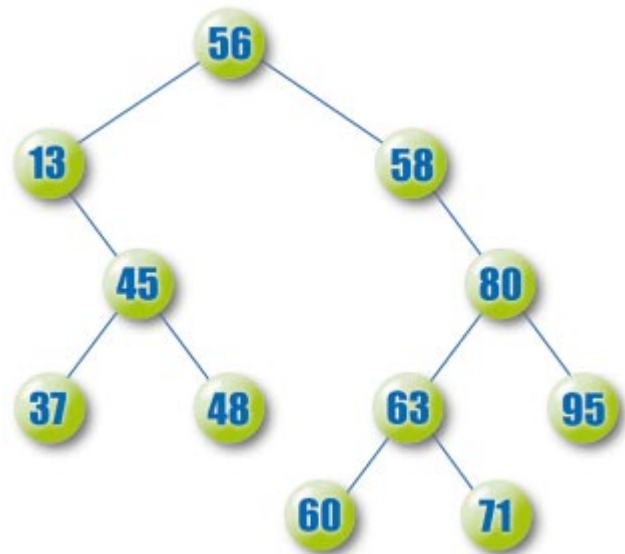


Figure 8: The sorted binary tree from [Figure 7](#) after the value 48 has been added.

```

// Add a new node to this subtree.
procedure TSortNode.AddNode(new_value : Integer);
begin
  // See which child to examine.
  if (new_value < node_value) then
    begin
      // Add it to the left subtree.
      if (left_child = nil) then
        begin
          // Create the new child here.
          left_child := TSortNode.Create;
          left_child.node_value := new_value;
        end
      else
        begin
          // Add it to the subtree.
          left_child.AddNode(new_value);
        end;
      end
    end
  else
    // Add it to the right subtree.
    if (right_child = nil) then
      begin
        // Create the new child here.
        right_child := TSortNode.Create;
        right_child.node_value := new_value;
      end
    else
      // Add it to the subtree.
      right_child.AddNode(new_value);
    end;
  end;
end;

```

Figure 9: Code that inserts a value in a sorted binary tree.

Because $48 > 13$, the program examines that node's right child and compares 48 to 45. Because $48 > 45$, the program should next examine the node's right child. In this case there is no right child, so the program inserts the new node as the right child of the node containing 45. The result is the tree shown in Figure 8.

Like so many tree operations, you can write a node insertion routine recursively. Each node's *AddNode* procedure compares its node value to the new value. It then invokes the appropriate child node's *AddNode* procedure. To add a node to a tree, a program invokes *AddNode* for the tree's root. Figure 9 shows the *AddNode* procedure for the *TSortNode* node class.

Climbing Trees

No discussion of trees would be complete without some mention of tree traversal. Traversal is the process of visiting all the nodes in the tree in a specific order. Four main traversals are defined for trees: preorder, inorder, postorder, and breadth-first.

In a preorder traversal, a node lists itself before listing its children. In an inorder traversal, a node lists its left child, then itself, then its right child. A postorder traversal lists a node's children before listing the node itself. Finally, a breadth-first traversal lists all nodes at a given depth in the tree before it lists those at the next level. For example, the preorder, inorder, postorder, and breadth-first traversals of the tree shown in Figure 8 are:

- preorder: 56, 13, 45, 37, 48, 58, 80, 63, 60, 71, 95

```

// Return the subtree's preorder traversal.
function TSortNode.Preorder : string;
begin
  Result := IntToStr(node_value);
  if (left_child <> nil) then
    Result := Result + ' ' + left_child.Preorder;
  if (right_child <> nil) then
    Result := Result + ' ' + right_child.Preorder;
end;

// Return the subtree's inorder traversal.
function TSortNode.Inorder : string;
begin
  Result := '';
  if (left_child <> nil) then
    Result := Result + left_child.Inorder + ' ';
  Result := Result + IntToStr(node_value);
  if (right_child <> nil) then
    Result := Result + ' ' + right_child.Inorder;
end;

// Return the subtree's postorder traversal.
function TSortNode.Postorder : string;
begin
  Result := '';
  if (left_child <> nil) then
    Result := Result + left_child.Postorder + ' ';
  if (right_child <> nil) then
    Result := Result + right_child.Postorder + ' ';
  Result := Result + IntToStr(node_value);
end;

```

Figure 10: Sorted node traversal code.

- inorder: 13, 37, 45, 48, 56, 58, 60, 63, 71, 80, 95
- postorder: 37, 48, 45, 13, 60, 71, 63, 95, 80, 58, 56
- breadth-first: 56, 13, 58, 45, 80, 37, 48, 63, 95, 60, 71

Note that the inorder traversal of a sorted binary tree lists the items in sorted order. This gives a simple method for sorting a list of items: Add them one at a time to a sorted binary tree, then produce the tree's inorder traversal.

Figure 10 shows code used by the *TSortNode* class to produce traversals for a node's subtree. They are generally similar in structure. Each routine adds the node's value to a string, together with the traversals for the node's children. The main difference between these routines is the order in which they combine the node's value with the child traversals.

Breadth-first traversals are a little different from the others. When it visits a node, the traversal routine cannot immediately follow the child nodes down into the tree like the other functions do. It must visit other nodes at the current depth in the tree first.

Figure 11 shows code that performs a breadth-first traversal. This code uses a *TList* object to build a list of nodes it must output. While this list isn't empty, the code removes the first node from the list and adds its value to the traversal. It then adds the node's children to the list for later output.

All the nodes at the same level in the tree are added one after each other in the list, so they are later removed together before any of their children are removed. That produces the

```

// Return the subtree's breadth-first traversal.
function TSortNode.BreadthFirst : string;
var
  nodes : TList;
  node : TSortNode;
begin
  Result := '';
  // Create the node list.
  nodes := TList.Create;
  // Put the root node on the list of nodes.
  nodes.Add(root);
  // While the list of nodes is not empty...
  while (nodes.Count > 0) do begin
    // Output the first item and remove it from the list.
    node := nodes.Items[0];
    nodes.Delete(0);
    Result := Result + IntToStr(node.node_value) + ' ';
    // Add the node's children to the stack.
    if (node.left_child <> nil) then
      nodes.Add(node.left_child);
    if (node.right_child <> nil) then
      nodes.Add(node.right_child);
  end;
  // Destroy the list of nodes.
  nodes.Free;
end;

```

Figure 11: Code to perform a breadth-first traversal.

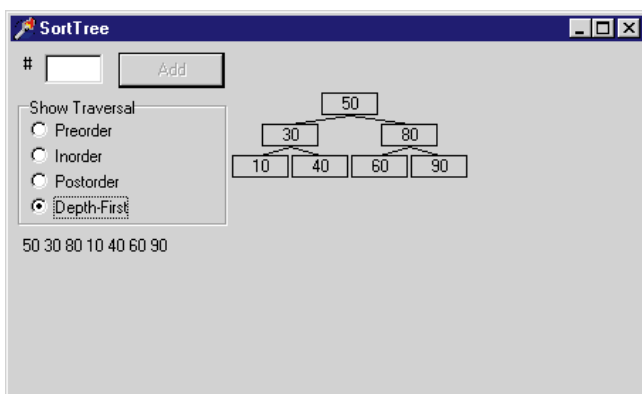


Figure 12: The example program SortTree displaying a depth-first traversal.

breadth-first traversal. Note that unlike the other traversal routines, this one is not recursive.

The SortTree example program shown in Figure 12 lets you build a sorted binary tree. Enter a number in the text box and click the Add button to add a new node to the tree. If you click on a radio button, the program uses the code shown in Figures 10 and 11 to display the appropriate traversal for the tree.

Conclusion

This article hardly scratches the surface of the things you can do with trees and other dynamic data structures. You can quickly add, remove, and rearrange the items in a tree to represent new relationships. With a little extra work, you can create more exotic kinds of trees, such as B-trees, B+ trees, and AVL trees. Using data structures like these, you can represent and manipulate data with complex hierarchical relationships efficiently. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\JAN\DI9901RS.

Rod's book *Ready-to-Run Delphi 3.0 Algorithms* [John Wiley & Sons, 1998] has lots more to say about trees and other dynamic data structures. For more information, visit <http://www.delphi-helper.com/da.htm>. You can contact Rod via e-mail at RodStephens@delphi-helper.com.





DELPHI REPORTS

Delphi 2, 3, 4 / QuickReport

By Keith Wood



Generic Reports

Using Polymorphism to Create Reusable Reports

Many applications require reporting capabilities to describe and distill the data they manipulate. To make these reports more useful, we often need to supply parameters to control various aspects of the reports, e.g. limit the range of records selected, control breaks and subtotaling, set the sort order, etc. However, we don't want to create a separate form for each report to retrieve these values from the user — especially when the same sort of values are typically required for several reports. It would be better if we could have a common way of eliciting these details, then handling them in a generic way.

This article describes an approach to gathering report parameters and manipulating them through the magic of polymorphism, allowing each report to do its own thing with a standard call and set of parameters. The reporting tool used here is QuickReport, but the technique could be applied to any other tool as well.

Report Parameters

We want a single form to ask for all possible parameters from the user for all our reports. Having all the processing in one form allows us to reuse the code effectively because many reports share parameters. It also reduces

maintenance costs, since we only have to make changes in one place.

For our example, we have several reports running against the DBDemos database that comes with Delphi. These are divided into two main groups: those for customers and those for orders. The parameters we might want to ask for are a customer, state, an order, and a date or date range.

We must then specify which of the possible parameters are available for use within each report. Because there can be a combination of any or all of these, we use an enumerated type, and associate a set of these values — *Params* — with the report:

```
type
  // Possible parameters to be requested
  // from user.
  TReportParameter = (rpCustomer, rpState,
                      rpOrder, rpDate,
                      rpDateRange);
  TReportParameters = set of TReportParameter;
```

Figure 1: The parameters form at design time, showing all possible panels. Alternating panels are colored here to highlight their presence.

Each set of parameters resides on its own panel on the form and corresponds to one of the enumerated values in the set (see [Figure 1](#)).

These panels provide us with an easy way to hide items that aren't required for a particular report. We just set the *Visible* property of the panel based on the presence of the appropriate enumerated value:

```
pn1DateRange.Visible := (rpDateRange in Params);
```

Setting the *Align* property to *allTop* for these panels ensures they reposition themselves automatically as they appear or disappear. Setting the correct *TabOrder* enables a consistent flow through the fields regardless of which are present. We can then set the form to the right size by using the position and dimensions of the final panel. This last panel is always displayed because it contains the **OK** and **Cancel** buttons used to initiate the report or close the form, as well as an option to preview the report or send it directly to the printer.

Even though we display the parameter panels to the user, some reports may require a value to be entered, while others can accept a blank value. To cater to this, we associate *Required*, another set of the aforementioned enumerated values, with each report. The elements in this set specify which parameters must be non-blank, i.e. which are required. This set should be a subset of those available to the report.

Sorting and Filtering

Another common feature among reports is the ability to specify an ordering for the printed records. This is achieved by letting the user pick from a list of possible sequences. Each report may have its own set of orderings, so it's specified with that report definition.

To allow a descriptive field to be displayed to the user while having the actual field names behind the scenes, we make use of a string list's ability to associate an object with each item. Unfortunately, a string is not an object and cannot be stored directly. To easily overcome this limitation, we can wrap it in an object, *TString*, before loading the list:

```
type
  // Wrapper around a string.
  TString = class(TObject)
  public
    Value: string;
    constructor Create(sValue: string);
  end;
```

Preparing the sorting list as if it were an .INI file facilitates loading the list of sort options. Each element consists of the descriptive text, followed by an equals sign (=), then the fields to be used. Elements are separated by a carriage return and line feed. Upon loading this into a string list via its *Text* property, we can access the two parts of the elements with the *Names* and *Values* properties. These are then transferred into the combo box that appears on the screen as the *Items* and *Objects* (see [Figure 2](#)).

Applying a filter to a report allows us to restrict the records displayed to those that are appropriate to the report selected. We could have exactly the same report behind two menu

items with the only difference being in the filter. This value appears in each report's definition as a string that's added to the WHERE clause of the report's query.

```
procedure LoadSortBy(sSortBy: string);
var
  sIsSort: TStringList;
  i: Integer;
begin
  sIsSort := TStringList.Create;
  try
    // Load sort options from array; already delimited by
    // line feeds so Text property loads into separate
    // lines in string list.
    sIsSort.Text := sSortBy;
    // Then separate display and field values.
    for i := 0 to sIsSort.Count - 1 do
      cmbSortBy.Items.AddObject(sIsSort.Names[i],
        TString.Create(sIsSort.Values[sIsSort.Names[i]]));
    pn1SortBy.Visible := (cmbSortBy.Items.Count > 1);
    cmbSortBy.ItemIndex := 0;
  finally
    sIsSort.Free;
  end;
end;
```

Figure 2: Loading the sorting options.

```
type
  TReportRec = record
    // Menu entry under which to appear, format:
    // 'upper level|lower level.'
    MenuEntry: string;
    // Name of QuickReport class.
    Report: string;
    // Title of the report.
    Title: string;
    // Set of user parameters available for entry.
    Params: TReportParameters;
    // Set of parameters that are required;
    // should be subset of above.
    Required: TReportParameters;
    // Possible orderings, format:
    // '<display>=<field(s)>' + sSep + ...
    SortBy: string;
    // Filter to be applied to WHERE clause of query.
    Filter: string;
  end;

const
  sSep = #13#10; // Separator for sort by options.
  iNumReports = 7; // Number of entries in array below.
  // Records containing details of each report. Place in
  // order of appearance on menu.
  recReports: array [1..iNumReports] of TReportRec =
    ((MenuEntry: '&Customers|&List'; Report: 'TqrfReport1';
    Title: 'Customer List'; Params: [rpCustomer, rpState];
    Required: []; SortBy: 'Customer No=CustNo' + sSep +
    'Company=Company' + sSep + 'State=State, Company' +
    sSep + 'Zip Code=Zip, Company'), (MenuEntry:
    '&Customers|Last &Invoiced'; Report: 'TqrfReport1';
    Title: 'Customer Last Invoiced'; Params: [rpState,
    rpDate]; Required: [rpDate]; SortBy:
    'Customer No=CustNo' + sSep + 'Company=Company' +
    sSep + 'Last Invoiced Date=LastInvoiceDate desc' +
    sSep + 'State=State, Company' + sSep +
    'Zip Code=Zip, Company'), (MenuEntry:
    '&Orders|&List by Customer'; Report: 'TqrfReport2';
    Title: 'Orders by Customer'; Params: [rpCustomer];
    Required: [rpCustomer]; SortBy: 'Order No=OrderNo' +
    sSep + 'Sale Date=SaleDate, OrderNo' + sSep +
    'Total Amount=ItemsTotal, OrderNo'),
    ...
```

Figure 3: Defining the reports to the parameter module.

Report Definitions

To manage all the details pertaining to a particular report, we define a record structure and initialize an array of these records with the appropriate values (see [Figure 3](#)). Note that the syntax for initializing a record is a field name, followed by a colon (:), then the field's value. Fields are separated by semi-colons (;) and must be specified in the same order as in the declaration. Those fields at the end of the record that don't have values dif-

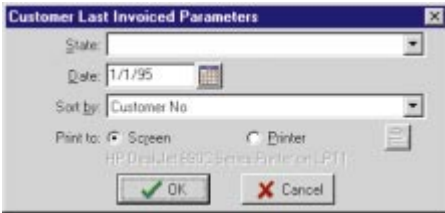


Figure 4: The report parameters form for the Last Invoiced report.

ferent from the default can be omitted from the list. [Figure 4](#) shows the report parameters screen in action for the second report defined in [Figure 3](#).

Adding a new report is then a simple matter of incrementing the report counter, *iNumReports*, and inserting the report's definition into the array declaration. The additional fields in the report definition allow us to specify the menu entry that invokes the report, its title (for display on the parameters form and on the report itself), and the name of the class that implements the report.

To pass the parameter values into the reports themselves, we need a transfer mechanism that works for a variety of data types in a generic way. Typically, the parameters are of three basic types: numbers, strings, and dates. Each of these can easily be converted to and from strings, suggesting that a string list is the ideal way to pass them around. A string list has the advantage of being able to handle a variable number of parameters without trouble. Furthermore, by using its *Values* property, a string list can be accessed for specific parameter values by name.

Polymorphic Reports

To enable all the reports to be handled in a common way, we make use of the polymorphic abilities of Delphi objects. This is the process of invoking a single method, but having each report deal with the call in its unique way.

All our reports have several features in common: a report title, a date-time stamp, and page numbers. We can place these into a base report, along with some default behavior, then inherit them into all our application reports. This helps us maintain a standard look and feel for the reports, and eases the maintenance of this appearance by localizing the changes to the base form.

Each report needs to handle the parameters passed to it in its special way. To allow this to happen, we define a method, *SetParams*, in the base report, and make it **virtual**. Then, in any report derived from the base one, we can override the behavior of this method to perform something specific to that report. The method is declared in the **protected**

section of the class to make it available to those subclasses, but not to the world at large:

```
type
  TqrfBase = class(TForm)
  protected
    procedure SetParams(slsParams: TStrings); virtual;
```

Because there is one parameter — the title — that is consistent across all reports, we set it as the only action in the *SetParams* method in the base report. Having some code in the method also means we don't need to label the method as abstract. Thus, any subclass of the base report can use this implementation if it so desires, without any further work.

All our other reports must derive from this base report, enabling them to inherit the definitions and default behavior of the latter. In these subclassed reports, we can change the behavior of the *SetParams* method by overriding the definition in the base report (see [Figure 5](#)). We then code the new functionality required. The code from the ancestor class can still be accessed through the use of the **inherited** keyword.

The *SetSelection* method in the base report, which loads a description of the selected parameters and sort option, isn't declared as **virtual** because its implementation doesn't change for each report. By declaring it in the base report, it's available to any derived report through normal inheritance.

Invoking the Report

Clicking the **OK** button on the report parameters screen validates the parameters selected based on the *Required* field for that report. If all is well, it then builds up the string list that contains the parameters. Any non-blank selections are added to the list, as is the title of the report, and any filter and sort order applicable to that report. We then create an instance of the nominated report and ask it to process these parameters. Finally, the report is previewed, or printed as requested.

```
type
  TqrfReport3 = class(TqrfBase)
  protected
    procedure SetParams(slsParams: TStrings); override;
  end;

  // Set up query from parameters.
  procedure TqrfReport3.SetParams(slsParams: TStrings);
  begin
    inherited SetParams(slsParams);
    with qryReport do begin
      // Set order number.
      ParamByName('OrderNo').AsInteger :=
        StrToInt(slsParams.Values[sOrder]);
      Open;
    end;
    qryItems.Open;
  end;
```

Figure 5: Overriding the *SetParams* method to process the parameters specific to this report.

```

// Create the report, pass parameters to it, and show it.
class procedure TqrfBase.ShowReport(sReport,sTitle:
string;
  sIsParams: TStrings; bPreview: Boolean);
var
  clsReport: TqrfBaseClass;
begin
  // Find the class with this name. Each such class must be
  // registered with Delphi, e.g.
  // initialization
  // RegisterClass(TqrfReport1);
  clsReport := TqrfBaseClass(FindClass(sReport));
  // Create a new report from this class.
  with clsReport.Create(Application) do
    try
      // Pass the selected parameters.
      SetParams(sIsParams);
      // And preview or print the report.
      if bPreview then
        qrpQuickReport.Preview
      else
        qrpQuickReport.Print;
    finally
      Free;
    end;
  end;
end;

```

Figure 6: Creating the report form and viewing or printing it.

To create the correct report (recalling that the parameters module doesn't want to know about all the possible descendants of the base report), we invoke a **class** method on that base report that's always available (see Figure 6). To this, we pass the name of the required report form as a string — the *Report* field in each definition. The application can locate the appropriate class from this name, and build us an instance of it to work with.

For this *FindClass* function to work, however, it's necessary to register the classes with the application beforehand. This is achieved through the *RegisterClass* procedure that's called from each child report unit in its **initialization** section:

```

initialization
  // Tell application about this form.
  RegisterClass(TqrfReport3);

```

This ensures the class is registered before it will be used.

De-coupling the Reports

De-coupling is the process of removing links between classes and/or units. It attempts to restrict the knowledge necessary for one object to interact with another. By doing this, we limit the communication between the two, thereby lessening the possibility of breaking those links by changing one of the objects.

In our case, we move all the report knowledge into the report parameters module, but we still want to access the reports from the menu. For this, we provide a single method in the reports module that can be called from the main form to construct the menu. All the main form needs to do is call this single method and pass across a reference to the menu item under which all the reports are to

```

// Construct a menu for the reports known to this unit and
// attach underneath the specified menu item.
class procedure TfrmRepParams.BuildMenu(
  mniParent: TMenuItem);
var
  i, iPos: Integer;
  mniAddTo, mniItem: TMenuItem;
  sCaption: string;

  // Find upper-level menu, or create it if necessary.
function FindMenuItem(mniCurrent: TMenuItem;
  sCaption: string): TMenuItem;
var
  i: Integer;
begin
  // Check current menu items.
  for i := 0 to mniCurrent.Count - 1 do
    if mniCurrent.Items[i].Caption = sCaption then
      begin
        Result := mniCurrent.Items[i];
        Exit;
      end;
  // Not there; create a new item.
  Result := TMenuItem.Create(mniCurrent);
  try
    Result.Caption := sCaption;
    mniCurrent.Add(Result);
  except
    Result.Free;
    raise;
  end;
end;

begin
  // Process full array of reports.
  for i := 1 to iNumReports do
    with recReports[i] do begin
      // Locate parent menu (if applicable).
      mniAddTo := mniParent;
      sCaption := MenuEntry;
      iPos := Pos('|', sCaption);
      // There may be several levels to menu.
      while iPos > 0 do begin
        mniAddTo := FindMenuItem(
          mniAddTo, Copy(sCaption, 1, iPos-1));
        sCaption := Copy(sCaption, iPos + 1,
          Length(sCaption) - iPos);
        iPos := Pos('|', sCaption);
      end;
      // Add new menu item.
      mniItem := TMenuItem.Create(mniAddTo);
      try
        with mniItem do begin
          Caption := sCaption;
          Tag := i; // Index into table here.
          // Return to this unit to process.
          OnClick := ReportMenuClick;
        end;
        mniAddTo.Add(mniItem);
      except
        mniItem.Free;
        raise;
      end;
    end;
  end;
end;
end;

```

Figure 7: Building the reports menu.

appear. This would be invoked in the *OnCreate* event of the main form, as shown here:

```

// Ask the report unit to build the reports menu.
procedure TfrmGenReports.FormCreate(Sender: TObject);
begin
  TfrmRepParams.BuildMenu(mniReports);
end;

```


That is the entire extent of the interaction with the reports module from the rest of the system. There's not much that can change here! The *BuildMenu* method is declared as a **class** method of the parameters form. This means it belongs to the class as a whole, and not to any particular instance of the class. It also means we can call the method without having any instances of the class to work with. This is useful, given that we create an instance of the form in response to a selection from the menu. As you can see, we refer to the class itself when invoking the method.

Within the menu-building method, we loop through the array of reports and build a menu item for each one (see [Figure 7](#)). These are attached to the menu item passed in by the main form, and down through other intermediate levels, as specified in the *MenuEntry* value for each report. Different levels are separated by a pipe (|) and should include ampersands (&) in the appropriate spots if menu accelerators are required. The result of clicking on one of these menu items is to invoke another method that also resides in the reports form as a **class** method.

Because all the report menu items come back to this one method, we need to be able to identify which one was requested. The easiest way to do this is to use the *Tag* property of the menu items, and set this equal to the index into the array of report details. This way, there is no hard-coding of *Tag* values, and we can safely add new reports by simply inserting them into the array.

Within the response method, we check that the *Tag* value is a valid index into the array. If it is, we create an instance of the report parameters form tailored for the nominated report. The rest of the system doesn't know — and has no need to know — what is happening within the reports menu.

Demonstration

The demonstration project that accompanies this article (available for download; see end of article for details) includes

a main form with a menu, the report parameters form, the base report, and three subclassed reports for actual use. It is built around the DBDemos database that comes with Delphi, and illustrates the techniques described in this article.

Pick a report from the menu and fill in the required parameters. Choose whether to preview it, or send it straight to the printer and watch the report appear.

As mentioned earlier, QuickReport is used as the reporting tool in this demonstration, but with minor modifications the techniques could be applied to other tools.

Conclusion

Through the use of object-oriented techniques and code reuse, we have produced a generalized form for requesting report parameters from the user, and applying them to each of the reports in the system. This should reduce maintenance costs since we only have to make the changes in one place.

Furthermore, we have de-coupled the reporting subsystem from the rest of the application — allowing it to be treated as a black box — with only a single call required to initialize it. This isolates the reports from the rest of the system, and severely limits the effects of any changes in either area. **Δ**

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\JAN\DI9901KW.

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





AT YOUR FINGERTIPS

Delphi / Tips

By Robert Vivrette



A Quick Spin on NT

And Other Useful Tips

No doubt many of you are familiar with the *BitBlt* Windows API function. For those of you who aren't, it stands for "Bit Block Transfer" and is simply one of the functions Windows uses to paint images on the screen. Its close relative is the *StretchBlt* function, which performs a similar task, but allows you to stretch or shrink the image you are drawing. Delphi wraps each of these commands in far simpler-to-use mechanisms: the *Draw* and *StretchDraw* methods.

In addition to a few other relatives (including *MaskBlt* for masking operations, and *PatBlt* for painting an area with a pattern), Windows also includes a little-known function called *PlgBlt*. Part of the reason why it isn't in many peoples' graphics vocabulary is that it's currently an NT-only function (as is *MaskBlt*). This means that users of Windows 3.1/95/98 are out of luck; it simply won't work.

However, if you use Windows NT and need to do some fancy footwork with graphics, you'll find *PlgBlt* to be an incredibly powerful tool. In a nutshell, *PlgBlt* performs similarly to *StretchBlt*, except that the destination of the painting doesn't need to be rectilinear (hence the "Plg" in its name, which stands for parallelogram). All *PlgBlt* needs to perform this magic is three points on a canvas.

Now wait a minute, you say. Why only three points? Well, as it turns out, it really does use four points to make this parallelogram. However, it doesn't trust you to generate the fourth point. If you were to place three dots on a piece of paper in a specific order, there would be only one spot where you could place the fourth and still have a parallelogram. Note here that the sequence of those first three points is important in making this determination.

As a demonstration on the use of *PlgBlt*, I wrote a quick little application that allows the user to spin a bitmap in real-time. As you can see from the code in [Listing One \(on page 45\)](#),

there is no rocket science here, but the capability *PlgBlt* adds is undeniably cool. (This code is available for download; see end of article for details.) Even if you are using a non-NT development environment, there are a couple of interesting techniques you may find useful.

As you can see, there are only four methods in use: *FormCreate*, *FormDestroy*, *FormPaint*, and *FormMouseMove*. Our objective for this demonstration is to place a *TImage* on a form, add a graphic to it, and when the application is run, allow the user to grab the corners and rotate the image. We'll start by creating a new form, placing a *TImage* in the center, and loading the *TImage* with a bitmap.

The first thing that happens is the *FormCreate*. Here we do a few setup-type things. There's a *TBitmap* we declared earlier that we're going to use to prevent flicker when we rotate the image (more on that a little later). In *FormCreate*, however, we must set up that bitmap to be the width and height of the form.

Next, we set up an array of points. There are four points in this array (numbered 0 to 3); we'll be using them for two purposes: The first is to pass the array into the *PlgBlt* command (remember, it will ignore the last point); the second purpose is to provide locations for drawing handles on the image (so we can see where to grab when rotating it). All I do in *FormCreate* is place the points in their proper locations — one at each corner of the image.

Next, we calculate a *MidPt*. This *TPoint* variable will hold the rotational center of the image, and is an average of the four corners of the image. The next variable defined is *R*, which will hold the distance (radius) from one corner of the image to the *MidPt*. This is used in the rotation calculations later on. Next, we fill an array of four real numbers (doubles actually) with the initial angle that each corner of the image has in relation to the midpoint. This is also used in our rotation calculations. Lastly, we set the *OverHandle* variable to a known value. This variable is used to track the handle number that the mouse is over. When it is -1, it's not over a handle.

The *FormDestroy* method simply cleans up a little by deleting the background bitmap we created in the *FormCreate*.

FormPaint is where *PlgBlt* comes into play. Earlier, I noted we were going to use the background bitmap to avoid flicker. The way this is done is that all drawing is done on this bitmap first. When it's all done, the entire contents are plastered on the form's canvas with a single call to its *Draw* method. Flicker occurs when you have multiple (and conflicting) draw actions occurring in the same space. Because the background bitmap is an in-memory *TBitmap*, we can draw on it all we want and the user won't see it until it is transferred to the form. Because there's a single drawing event, there is no flicker.

Back to *FormPaint*. First, we clear the background bitmap using *FillRect*. Then we do our call to *PlgBlt*. The parameters it expects are as follows (partially excerpted from the Win32 API Help):

- *hdcDest*. This identifies the destination device context. This is the handle to the destination canvas (*BkBmp.Canvas.Handle*).
- *lpPoint*. This is our array of points used to identify the first three corners of the destination parallelogram. The upper-left corner of the source rectangle is mapped to the first point in this array, the upper-right corner to the second point in this array, and the lower-left corner to the third point. As I mentioned earlier, the lower-right corner is calculated for you. It doesn't hurt that we pass in all four points, the *PlgBlt* function simply ignores the last one.
- *hdcSrc*. This identifies the source device context. This is the handle to the Images canvas (*Img.Canvas.Handle*).
- *nXSrc*. This specifies the x-coordinate of the upper-left corner of the source rectangle.
- *nYSrc*. This specifies the y-coordinate of the upper-left corner of the source rectangle.
- *nWidth*. This specifies the width of the source rectangle.
- *nHeight*. This specifies the height of the source rectangle.

The last three parameters (*hbmMask*, *xMask*, and *yMask*) have to do with an optional bitmap that can be used to mask colors. We aren't using these in this example, so we set them all to zero. If the call to *PlgBlt* is successful, we draw the four handles on the image, using the same array of points we passed into *PlgBlt*. If the call is not successful, it generally is a sign that we are not running on an NT platform, so we make an appropriate mention using *TextOut*. After this is done, the background bitmap is transferred all at once to the form, using its *Draw* method.

The last method in this demonstration, *FormMouseMove*, is the mouse management for working with the handles on the corners of the image. When the mouse is moved, we first look to see if the left mouse button is down. If so, we also check to make sure the *OverHandle* variable contains the number of a handle that has been grabbed. Assuming this is also true, we determine the angle from where the mouse is to the mid-point of the image (calculated earlier). We use this angle to re-calculate the locations of each of the four handles on the image.

If the left mouse button isn't down, we make sure the *OverHandle* variable is reset, then use *PtInRect* to determine if the mouse is over any of the four handles. If so, we store that handle's number in *OverHandle*, and switch the cursor to a hand pointer.

That's pretty much it for this demonstration. Notice that there is no code to rotate the image as such (the *PlgBlt*

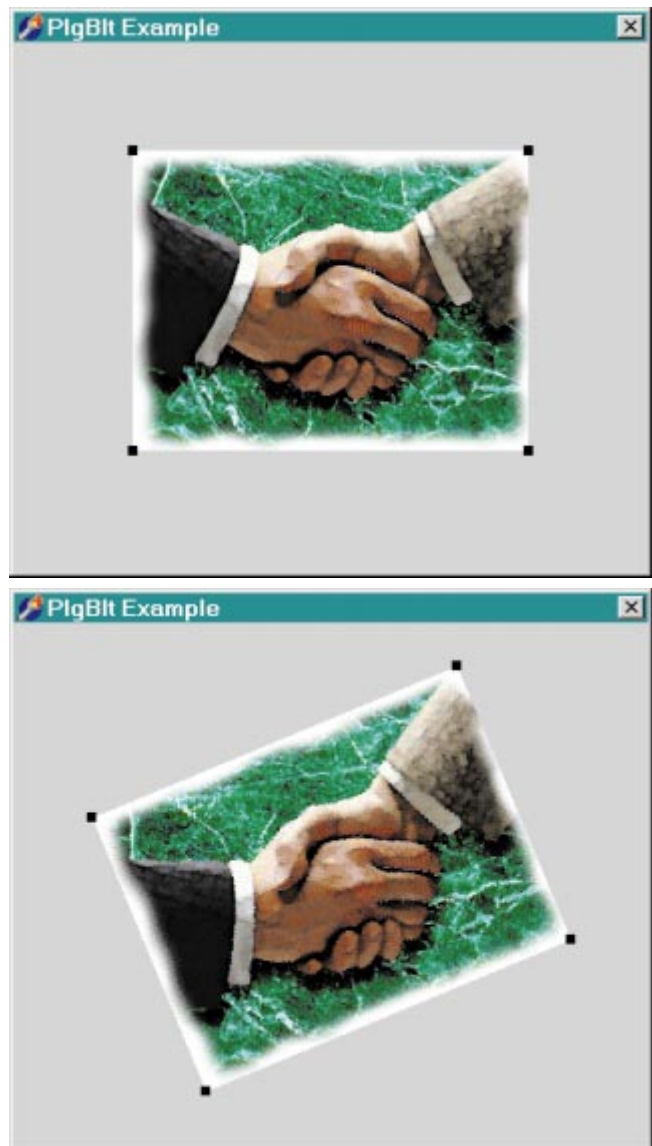


Figure 1 (Top): The example program in its original state.
Figure 2 (Bottom): The example program when rotated.

routine handles all that for us). When you run the application, you'll see something like what is shown in [Figure 1](#).

Grabbing one of the handles and spinning it a little counter-clockwise would result in what is shown in [Figure 2](#).

These pictures really don't do the demonstration much justice. Suffice it to say that the rotation is perfectly smooth and very fast, thanks in part to the amazing power of *PlgBlt*.

System Error Text Messages

How many times have you seen a reference similar to this in the Windows API Help? "If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information, call *GetLastError*."

Apparently, while programmers around the world were sleeping, Microsoft decided it was getting too difficult (and inefficient) to return all possible error values as the result of most function calls. Therefore, they reworked the idea of obtaining error values through a separate API function, namely *GetLastError*. Many API functions, however, retain some backward compatibility, and still return the error results themselves (in addition to setting the value *GetLastError* uses).

However, we need to keep up with the times; the function return values for errors we've come to rely upon will likely disappear. A good example of this is the *LoadLibrary* command. When successful, it returns the handle of the specified library (DLL or EXE). When it fails, it *used* to return a small integer value (between 0 and 32). Now, however, if it fails, it simply returns a zero (NULL), and requires that you go to *GetLastError* to get the real error result.

Obviously, this is a much more efficient solution to obtaining error values. After all, mixing error values with success values as the return result of a function requires you know which ones are errors and which ones are "good."

So everything's wonderful now, right? Well — no! If you look at *GetLastError*, it's still simply returning a numeric result that must be interpreted by your program. What if you want to get a text message of what the error is? Microsoft does include another function, named *FormatMessage*, that can take the result of *GetLastError* and make an intelligible error message out of it. The problem is that *FormatMessage* isn't the easiest function to use.

Fortunately, Delphi comes to the rescue again. In Delphi's SysUtils unit, there's a very useful wrapper function for *FormatMessage* named *SysErrorMessage*. Whenever you get an error result from an API function, you simply pass *GetLastError* into a call to *SysErrorMessage*. The result is a string representation of the error. For example, this statement:

```
ShowMessage(SysErrorMessage(GetLastError));
```

will display the result of *SysErrorMessage* in a *ShowMessage* dialog box. Pretty handy! Do keep in mind, however, that the value stored in *GetLastError* is temporary. If you call a function and get an error result, make sure you call *GetLastError* immediately. Its value may change if you call other API functions before you retrieve its value.

Note also that you can use *SetLastError* to set your own error values. This is normally done for functions stored in DLLs. You can learn more about this in Win32 API Help under the section on *SetLastError*.

String-to-Integer Conversion

One of the more frequently used functions in a programmer's repertoire is *StrToInt*. It simply converts a string value into an integer value. Programmers often use this to extract a numeric value (from an edit box, for example). A program may ask for the number of airplanes sold during a particular month; the user enters the value in the edit box, and the programmer takes that value and converts it to an integer for internal processing. However, *StrToInt* has an annoying quality. If the function cannot convert the value to a valid integer (perhaps there are non-numeric characters in the string), it generates an exception, such as "1234b is not a valid integer value." In some cases, this may be a desired effect, but you often don't want to generate an exception. You could, of course, turn off run-time exceptions, but that can hide exceptions generated in other areas.

Do you wish there was a function that would do this conversion and not whine about invalid data? Wish no longer ...

The *StrToIntDef* function is an extension of *StrToInt* that allows you to specify a default value in the event the string cannot be converted to a valid integer value. All you do is specify the string to convert, and as a second parameter, provide the default value you want it to send back in the case of a conversion error. The main advantage is that this is entirely silent, so you can tell you had a conversion error (the default value was returned) without disrupting the flow of the application.

Remember, both *StrToInt* and *StrToIntDef* are in the SysUtils unit, so you will need to include it in your `uses` clause. ▲

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\JAN\DI9901RV.

Robert Vivrette is a contract programmer for Pacific Gas & Electric, and Technical Editor for *Delphi Informant*. He has worked as a game designer and computer consultant, and has experience in a number of programming languages. He can be reached via e-mail at RobertV@mail.com.

Begin Listing One — PlgBltU.pas

```

unit PlgBltU;

interface

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

type
  TForm1 = class(TForm)
    Img: TImage;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure FormMouseMove(Sender: TObject;
      Shift: TShiftState; X,Y: Integer);
  private
    P          : array[0..3] of TPoint;
    OAng       : array[0..3] of Double;
    OverHandle : Integer;
    BkBmp      : TBitmap;
    MidPt      : TPoint;
    Ang,R      : Double;
  end;
var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
var
  Pt : Integer;
begin
  BkBmp := TBitmap.Create;
  BkBmp.Width := Width;
  BkBmp.Height := Height;
  P[0] := Img.BoundsRect.TopLeft;
  P[3] := Img.BoundsRect.BottomRight;
  P[1] := P[0]; Inc(P[1].X,Img.Width);
  P[2] := P[3]; Dec(P[2].X,Img.Width);
  with Img do
    MidPt := Point(Left+Width div 2,Top + Height div 2);
  with Img do
    R := Sqrt(Sqr(Width div 2) + Sqr(Height div 2));
  for Pt := 0 to 3 do
    with P[Pt] do
      OAng[Pt]:= ArcTan2(Y-MidPt.Y,X-MidPt.X)+Pi;
      OverHandle := -1;
  end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  BkBmp.Free;
end;

procedure TForm1.FormPaint(Sender: TObject);
var
  Pt : Integer;
begin
  with BkBmp.Canvas do begin
    Brush.Color := clBtnFace;
    FillRect(ClipRect);
    if PlgBlt(Handle,P,Img.Canvas.Handle,0,0,
      Img.Width,Img.Height,0,0,0) then
      begin
        Brush.Color := clBlack;
        for Pt := 0 to 3 do
          with P[Pt] do
            FillRect(Rect(X-3,Y-3,X+3,Y+3));
          end
        else
          TextOut(0,0,'PlgBlt supported only on WinNT');
        end;
    Canvas.Draw(0,0,BkBmp);
  end;
end;

```

```

procedure TForm1.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X,Y: Integer);
var
  Pt          : Integer;
  TmpRect     : TRect;
begin
  if ssLeft in Shift then
    begin
      if OverHandle = -1 then
        Exit;
      Ang := ArcTan2(Y-MidPt.Y,X-MidPt.X) -
        OAng[OverHandle]+Pi;
      for Pt := 0 to 3 do
        P[Pt] := Point(MidPt.X-Round(R*COS(Ang+OAng[Pt])),
          MidPt.Y-Round(R*Sin(Ang+OAng[Pt])));
      Paint;
    end
  else
    begin
      OverHandle := -1;
      for Pt := 0 to 3 do begin
        with P[Pt] do
          TmpRect := Rect(X-3,Y-3,X+3,Y+3);
          if PtInRect(TmpRect,Point(X,Y)) then
            begin
              Cursor := crHandPoint;
              OverHandle := Pt;
            end;
          end;
        if OverHandle = -1 then
          Cursor := crDefault;
        end;
      end;
    end.
  end.
End Listing One

```





NEW & USED

By Ron Loewy

TSQLBuilder

An Affordable Visual Query Tool for Delphi Applications

Unlike many Delphi developers that spend time writing database applications using the Delphi data access and data control components, my brush with the BDE and Delphi database programming was limited to a small project I developed three years ago with the help of Woll2Woll's InfoPower component set. My new project required the ability to import data from different databases into a help-authoring tool. The design of the import application made it clear that we'd need to provide end users with the ability to define database queries and specify how they intend to bind the information into the generated help topics. Because we don't know the source of the data, the users must create their own SQL queries to select the data.

My experience is that some programmers prefer to use visual design tools for their SQL queries, unless the query builder they use is incapable of a SQL feature, or they are the authors of a SQL book available in the library of the computer science department in a university near you. It was clear we needed to look for a visual query builder to include with our data import application. The first tool I checked, *TSQLBuilder* from Conclusion Software, proved to be a capable contender. When it took less than 20 minutes to roughly integrate it with my application, I abandoned the search for other options.

TSQLBuilder arrives as a collection of Delphi units and DFM files that you install into Delphi's component library by compiling the *TSQLBuilder.DPK* file provided with the product. Once you install the package, two components are added to the Samples page of the Component palette. *TBtmEdit* is used internally by the SQL Builder forms and *TSQLBuilder*, the visual query builder.

TSQLBuilder is designed to work like a file open or save dialog box; all you need to do to activate it from your application is call its *Execute* method. The resulting SQL query is available in the component's *SQL* property. From there, it's simple to obtain the SQL source, feed it to your *TQuery* component, and continue with your application's logic.

While my needs were limited to the ability to define queries visually, you can do more with *TSQLBuilder*. The ability to print the query results is built into the component for simple ad hoc reports. You can set the *DefaultPrinting* property to *False* and handle all the printing chores by writing code for the *OnPrintHeader*, *OnPrintRow*, and *OnPrintFooter* events.

If your users need to reload the query they were defining from persistent storage, the component provides *LoadFromStream* and

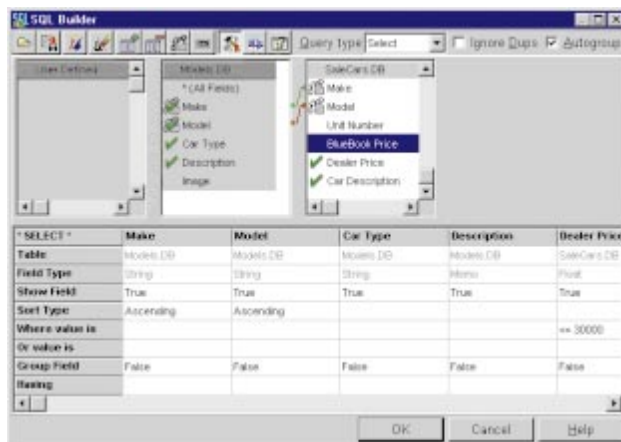


Figure 1: The SQL Builder dialog box in action.

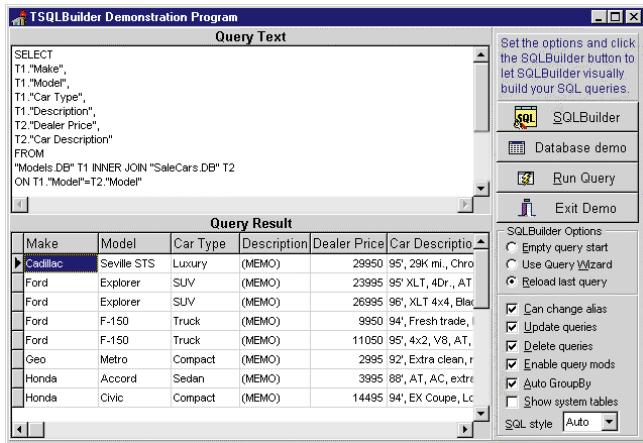


Figure 2: The SELECT statement created from the query shown in Figure 1, and the resulting answer set.

SaveToStream functions. I combined the SQL Builder's stream in the applications database definition file, but you can save queries to BLOB fields, file streams, memory streams, or any other stream your application creates.

When the user activates the SQL Builder, he or she is presented with a form for defining the tables and fields of the query (see Figure 1). A toolbar provides access to functions such as query load and save, a table selection wizard, adding and removing a table, clearing all tables, a table join editor, adding a SQL-defined field, selection of the query type (select, update, delete), and the ability to switch between the visual query builder, a SQL editor, and an instant results table.

Once a table is added, it appears in the top part of the query builder form. The user can double-click on any field to add it to the query. Every added field appears in the field grid at the bottom of the form. The user can define parameters such as sort type, selection, grouping, and visibility for each field. If the user adds another table to the query, the query builder tries to create a join based on field names. The user can fine-tune the join operation with the **Edit Join** button, or create new links by dragging and dropping fields from one table to another. The results of a query are shown in Figure 2.

Using *TSQLBuilder* is so simple that most developers will be able to take advantage of the tool by playing with the sample project provided. The sample project shows how to use the *TSQLBuilder* component, and combine the results with a *TQuery*. It also provides a sample of using a database to store different queries and load them using the *LoadFromStream* method.

The product comes with Microsoft Write files containing end-user information, a programmer's guide, and other useful documentation. Every property, event, and method of the component is described in the programmer's guide. The package also comes with a compiled .HLP file you can give to your users that documents the query building process. Documentation for our product is HTML-based; Conclusion was kind enough to provide us with the end-user documentation in that format upon request.

The documentation provided is enough to get you started in a hurry. Any questions you might have are answered by inspecting the sample project. The product also comes with complete source code.

I wish the query builder could be embedded in the main application and not appear as a dialog box, and a closer replication of the Microsoft Access query builder tool would also be nice, because many users are familiar with its interface.

Nevertheless, *TSQLBuilder* is a great tool for end users that need to create ad hoc SQL queries. It integrates simply with Delphi applications and provides nice touches, such as online documentation for end users. Last, but not least, the price of US\$65 (or US\$59 via e-mail) is a bargain. Δ

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

INFORMANT

FACT FILE

TSQLBuilder 1.62 offers an easy way to provide ad hoc query capability to the end users of your Delphi application. It integrates easily with Delphi applications, and provides nice touches such as end user documentation. It's available at a great price for all versions of Delphi.

Conclusion Software, Pty. Ltd.
P.O. Box 106
Kemps Creek NSW 2171
Sydney, Australia

Voice: +61 2 4774 8703
Fax: +61 2 4774 9086
E-Mail: info@conclusion.com
Web Site:
<http://www.conclusion.com>
Price: on disk US\$65; by e-mail US\$59





NEW & USED

By Alan C. Moore, Ph.D.

CodeSite 1.1

A Revolutionary New Debugging Tool

Debugging is one of the least glamorous aspects of programming. It's also one of the most important. Careful as we are, we can never eliminate the need for debugging, nor can we change its mundane character. However, we can find ways to make the process more efficient and less time consuming. CodeSite, a revolutionary new debugging tool from Raize Software Solutions, Inc. takes a giant step in that direction. First we'll review some of Delphi's built-in debugging tools. Then we'll examine CodeSite's capabilities.

As developers, we appreciate the many powerful debugging tools built into Delphi, from the several ways of tracing or stepping through code, to the new "fly-over" information on current variable values introduced in Delphi 3. These tools and techniques can be helpful in many situations, but not all. For example, the integrated debugger can give you an excellent snapshot of the current state of your application, but can't provide any history of how you got there. And while Delphi's debugger is useful for following program flow in a normal application, it doesn't work for IDE extenders — such as experts — or for components at design time.

Using a traditional approach, CodeSite takes care of the dirty work, and allows you to easily log program information. (The first time I used CodeSite was when it was in beta and I was writing the Project Identifier Plug-in for Eagle Software's CodeRush. Even after I crashed Delphi with my plug-in, I was able to find in what method the access violation occurred, and why.) Let's examine that approach and CodeSite's interface to it.

A Traditional Technique; a New Tool

Several years ago, when I was working with Turbo Pascal, I would add *Write* and *Writeln* statements to track interrelated variables as

they changed to see how they affected other variables and program flow. In a complex routine or unit, you often have no other choice. Perhaps you've used a similar approach, such as keeping track of one or more variables by writing their values to a label or panel. Or you may have used the common approach of using *ShowMessage* to display a message box containing program information. This approach has problems: First, you must convert all your data to a string representation; second, the dialog box causes the application's focus to change. Also, when the *ShowMessage* dialog box is closed, your application will need to repaint itself.

While this kind of technique may be necessary and appropriate, it's fraught with dangers. As the previous example implies, keeping track of debugging data *within the application you're debugging* can affect that application. It could even introduce problems. If the program crashes, you could lose valuable debugging information. The effort to create your own debugging display is also time consuming, especially if you consider the additional time wasted when you must remove the debugging statements before shipping the final version. CodeSite supports this kind of approach, while at the same time removing most, or all, of these obstacles.

CodeSite consists of two separate, but interrelated, elements: a *CodeSite* object (defined in the *RzCSIntf* unit) that you add to the *uses* clause of any unit you wish to debug; and the CodeSite viewer, which displays the results of the debugging statements. The *CodeSite* object allows you to send a wide variety of data (e.g. strings, integers, Boolean variables, date/time values, etc.) to the viewer. The real power of CodeSite becomes obvious in the viewer, which we'll examine in detail after we discuss the *CodeSite* object.

CodeSite features many methods for sending messages to the CodeSite viewer (see [Figure 1](#)). They provide a way for sending a wide variety of programming-related data, much more than simply string data. You can also control whether CodeSite is enabled, by setting the *Enabled* property. This is particularly useful when you want to debug at a certain location after a certain number of iterations. It's analogous to setting conditional breakpoints. The *Enabled* property also allows you to keep CodeSite messages in your code without a negative impact on performance. When a CodeSite method is encountered in code, it first checks the state of the *Enabled* property before performing its function. If it's *False*, the method exits.

Many of the methods shown in [Figure 1](#) have variations for formatted strings, e.g. *SendFmtNote*. Others allow switching between different message types (*csmInfo*, *csmWarning*, *csmError*, etc.), e.g. *SendIntegerEx*. CodeSite also includes functions to *Clear* the contents of the viewer, send a message to the *ScratchPad* pane, *AddCheckPoint* in the code, or *AddSeparator* in the display. You can easily access these messages using the CodeSite Message Expert dialog box if you're using it with CodeRush (see [Figure 2](#)). Even easier are the keyboard short cuts. These work with the stand-alone and CodeRush implementations. With the CodeRush implementation, you save the particular variable to the Clipboard (a Boolean, such as *IsFileSelected*) and type the short cut, e.g. "csb" for CodeSite Boolean. The result is:

```
CodeSite.Boolean('IsFileSelected', IsFileSelected);
```

(Note that Delphi's short cuts work differently from those in CodeRush and don't support the kind of Clipboard technique outlined here.) This, in turn, produces output such as this in the CodeSite viewer:

```
IsFileSelected = False
```

The particular icons used to identify the different types of messages are useful in finding particular kinds of information. There are icons for Information messages (*csmInfo*), Warning messages (*csmWarning*), Error messages (*csmError*), etc. [Figure 3](#) shows many of these icons in the CodeSite viewer.

While CodeSite supports most of the commonly used data types, it can't automatically display the symbolic names of custom enumerated types (although it can show those values

Method	Description
<i>SendMsg</i>	Sends a message to the CodeSite viewer. Each message has an associated type.
<i>SendFmtMsg</i>	Like <i>SendMsg</i> , but has the ability to format the message string.
<i>SendNote</i>	Sends a message string (with an icon showing a yellow note icon) to the CodeSite viewer.
<i>SendError</i>	Sends an <i>Error</i> message string (with an icon showing a red circle with a white "x") to the CodeSite viewer.
<i>SendWarning</i>	Sends a <i>Warning</i> message string (with an icon showing a yellow triangle with a black exclamation point) to the CodeSite viewer.
<i>SendAssigned</i>	Sends a message to the CodeSite viewer indicating whether the <i>Value</i> parameter is assigned.
<i>SendBoolean</i>	Sends the results of an evaluation of the <i>Expression</i> parameter as a message to the CodeSite viewer.
<i>SendColor</i>	Sends the color symbol name (if available) of the <i>Value</i> parameter and the RGB values for the corresponding color, and sends the formatted result as a message to the CodeSite viewer.
<i>SendInteger</i>	Sends the formatted <i>Value</i> parameter (an <i>Integer</i>) as a string message to the CodeSite viewer.
<i>SendPoint</i>	Sends the formatted <i>Value</i> parameter (a <i>TPoint</i>) as a string as a message to the CodeSite viewer.
<i>SendRect</i>	Sends the formatted <i>Value</i> parameter (a <i>TRect</i>) as a string message to the CodeSite viewer.
<i>SendString</i>	Sends the formatted <i>Value</i> parameter (a string) as a string message to the CodeSite viewer.
<i>SendProperty</i>	Sends the current value of a property (simple or class type) to the CodeSite viewer as a <i>Property</i> message.
<i>SendObject</i>	Sends the current value of an object (<i>Obj</i> parameter) to the CodeSite viewer as an <i>Object</i> message with a description of the published properties and their current values to the inspector pane.
<i>SendStream</i>	Sends a <i>Stream</i> message to the CodeSite viewer. The first parameter is a message string used to identify the object; the second parameter is the actual object, the details of which are shown in the inspector pane.
<i>SendStringList</i>	Sends a <i>StringList</i> message to the CodeSite viewer. The first parameter is the message string; the second is the string list itself, which is shown in the inspector pane.
<i>EnterMethod</i>	Sends a message to the CodeSite viewer, which increments the indent level and adds a message to the message view.
<i>ExitMethod</i>	Sends a message to the CodeSite viewer, which decrements the indent level and adds a message to the message view.

Figure 1: Selected CodeSite methods for sending messages to the CodeSite viewer.

NEW & USED

as integers). With just a little bit of programming, however, you can use CodeSite's methods to display these values in this way. The enumerated type is defined as:

```
TCommentStringType = (cstBracket, cstParenStar,
    cstDoubleSlash, cstString);
```

To transform the enumerated type into its string representation, use the following procedure (courtesy of Ray Konopka):

```
procedure CSSendCommentString(const Msg: string;
    CommentStrType: TCommentStringType);
var
    S: string;
begin
    S := GetEnumName(TypeInfo(TCommentStringType),
        Ord(CommentStrType));
    CodeSite.SendFmtMsg('%s = %s', [Msg, S]);
end;
```

Having defined that procedure, you can now apply the following statement to the current status of any variable of this type:

```
CSSendCommentString('CommentStringType', ACommentStrType);
```

Because I track this data in a procedure that calls a recursive function, the debugging output demonstrates both the usefulness of the EnterMethod/ExitMethod pair and this custom statement. Following is a segment in which the recursive function is called twice. Note the indentation that CodeSite produces:

```
Method ——> : CheckedOut
Info : BPos = 22
Info : NewBPos = 23
Info : BPos = 42
Method ——> : CheckedOut
Info : BPos = 42
Info : NewBPos = 43
Info : CommentStringType = cstDoubleSlash
Method <<— : CheckedOut
Info : result = True
Info : CommentStringType = cstDoubleSlash
Method <<— : CheckedOut
```

The above comes from a log file that I saved to a .TXT file. In the viewer (which we'll discuss soon), these lines would have icons instead of the Method and Info keywords. You can also save log files in CodeSite's native format (.CSL for CodeSite log). This native CodeSite log file is powerful, allowing you to save the information recorded in the viewer (a .TXT file contains only the message strings). In addition, with the CodeSite log file, you can pass CodeSite logs across platforms and among team members. In all, the CodeSite object contains 25 methods that provide considerable power and control in debugging your applications.

The CodeSite Viewer

The CodeSite viewer has three panes: a message list, an inspector pane, and a scratch pad (again, see Figure 3). You'll

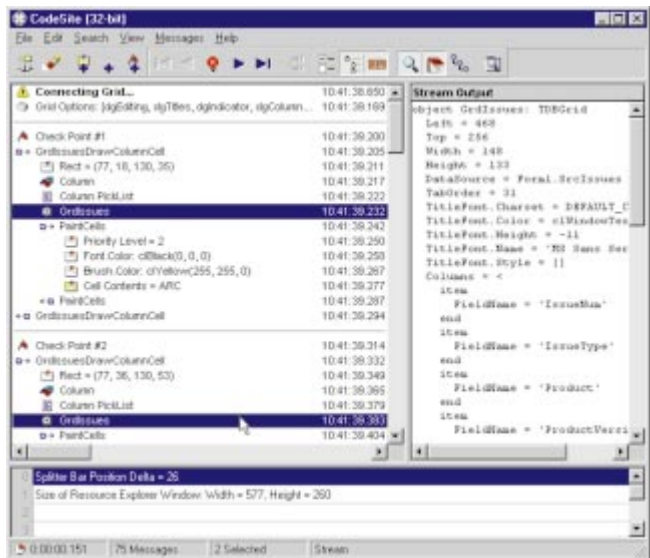


Figure 2 (Top): One way to add debugging messages to source code is by using the CodeSite Message Expert.

Figure 3 (Bottom): The three panes of the CodeSite viewer from the CodeSite Help file: message list, inspector pane, and scratch pad.

recall that there are commands associated with each of these panes, most of which belong with the message list. Commands can be executed from the main menu, or from the context menus shown by right-clicking the mouse on any of the panes. Most of the commands are also available on the toolbar. Status information is shown on the status bar.

As you would expect, the message list is the most important. Any message you send to the CodeSite viewer, except for scratch pad messages, appears in this pane. You can annotate items, as well as insert note messages anywhere in the log. Using the CodeSite Preferences dialog box, you can change the appearance of the message list and the font used to display different types of messages, and enable or disable the message icons that identify the different kinds of messages (see Figure 4).

CodeSite gives you the option of setting a time stamp for each message, allowing you to include time-stamp informa-

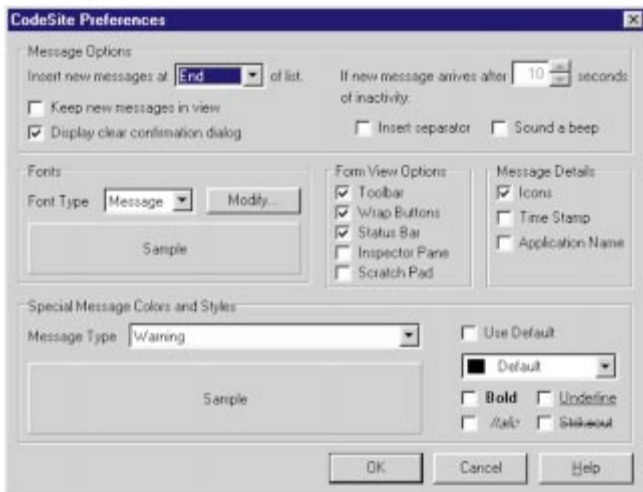


Figure 4: The CodeSite Preferences dialog box allows you to modify the appearance of the message list.

tion with each line of output. This time-stamp feature raises CodeSite to the level of an incipient profiler. Whether this potential will be fully realized in future versions is uncertain, but it's certainly an exciting prospect. Even in its current form, it's very powerful. For example, if you select two or more items in the message list, the lower-left status pane shows the time difference between the first and last items. Also, if you have an enter or exit method message selected and press the Find Matching Method toolbar button while holding down **⇧** (Shift), both the enter and exit messages are selected, and the status pane shows how long that method took to execute.

The inspector pane displays detailed information, depending on what type of message is selected in the message list. If a *csmObject* message is selected, for example, the inspector pane will show an Object Inspector-like view of all published properties and their object values. If a *csmStream* message is selected, the object's stream representation is displayed in a memo field. You can also display the call stack for the currently selected method in the inspector pane if that option is enabled.

Finally, the scratch pad is a pane to which you can send non-persistent messages. You can also write up to 100 lines. Each new message overwrites the current contents of the line. There are various situations where you might want to use the scratch pad, such as when tracking changes in the mouse position.

In addition to the three panes, there are several other very useful features in the CodeSite viewer: the message navigator buttons, which allow you to easily move around in a large log file; text searching in the message list; and the ignore messages option.

Special Utilities

As if all of this weren't enough, there are now several CodeSite utilities that increase this wonderful product's

power. If you use CodeSite with CodeRush, the CodeSite viewer appears in its own panel. However, there may be times when you need or want to use the separate CodeSite Message Router, a tray icon program that displays a dialog box allowing you to redirect CodeSite messages. If you're using CodeSite with CodeRush, the keyboard templates are automatically installed and available. If not, you'll want to download another utility, the CodeSite.dci file. Because Delphi 3 supports keyboard templates — which you can add through the Code Insight page of the Environment Options dialog box — you can easily add CodeSite templates to Delphi, with or without CodeRush.

There are two more utilities that will be of interest to some users. The CodeSite Compatibility Unit consists of a set of procedures that are identical to the interface procedures used by GExperts' debugging window and DebugLog window. For developers working with database programming, the CodeSite SQL Monitor Unit turns the CodeSite viewer into a SQL Monitor; you can view SQL messages as if you were using the SQL Monitor utility.

Conclusion

CodeSite is a powerful and exciting new Delphi debugging tool. With it, you can gather all kinds of data that would otherwise be very hard to obtain. Its keyboard templates make it very easy to add debugging statements to your code. And because each debugging statement begins with "CodeSite," it's also easy to remove them. With all of its features, it'll certainly become a vital part of many Delphi developers' toolkits. I recommend it highly. ▲

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

INFORMANT
FACT FILE

CodeSite 1.1 from Raize Software Solutions, Inc. is a revolutionary debugging tool based on the traditional technique of sending debugging information to a window or log file. It consists of a CodeSite object and a CodeSite viewer. The former allows the developer to track a wide variety of program data and to control the viewer. The viewer displays debugging information in useful ways, and is highly customizable.

Raize Software Solutions, Inc.
2111 Templar Drive
Naperville, IL 60565

Phone: (630) 717-7217
Fax: (630) 717-7329
E-Mail: sales@raize.com
Web Site: <http://www.raize.com>
Price: US\$79.95

Delphi 4: The Best Release Ever?

Each new version of Delphi has included enhancements to the Visual Component Library (VCL), and Delphi 4 is no different (for an in-depth summary, see Robert Vivrette's article "The Best Just Got Better" in the September, 1998 *Delphi Informant*). By now, many of you have upgraded to Delphi 4, and you've probably reached an initial assessment.

How do the changes in Delphi 4 compare with those in Delphi 3? The most striking change in Delphi 3 was its introduction of packages; Delphi 4's most significant changes are to the Integrated Development Environment (IDE) and the Object Pascal language, many of which Dr Cary Jensen previewed in his article "Delphi 4" in the July, 1998 *Delphi Informant*. In these (and other) accounts based on pre-release versions, the emphasis was on the positive and significant changes to this leading Windows development tool.

There's another side to the story, however: a story of bugs and corporate decisions. First, let's review some of the major changes and welcome enhancements.

Major Changes and Enhancements. One addition that affects both the IDE and the VCL is the new docking capability. In the IDE, the ability to dock, or attach, one window to another is very helpful in customizing your working environment. All the windows and toolbars can be dragged to various locations on the desktop or attached to other windows.

Now you can dock the Watch window to the lower panel along with the Message window. I've been waiting for this feature for a long time, but you need to be careful: If you drag a package editor window to improve its visibility (with docking turned on), it may seem to disappear if it becomes docked to a window in the background! Additionally, with the new *DragKind* and *DockSite* properties (and the existing *DragMode* property), you can now add docking behavior to components in your programs.

Docking support is just one of several enhancements to the VCL. At the lowest level, *TObject*'s new procedures, *BeforeConstruction* and *AfterConstruction*, give you added control over the behavior of classes and components. *TControl*'s new property, *Constraints*, allows you to specify the minimum and/or maximum width

and height of a control without resorting to API calls.

Even more significant are the changes to the Object Pascal language, which are more substantial than any previous version since Delphi 1. These include the new *Int64* data type, *Longword* (32-bit unsigned integer), dynamic arrays, and changes to some existing types. For C++ veterans, there are welcome additions, including routine or method overloading (multiple declarations of functions and procedures), and default parameters.

When you consider the Code Explorer, the enhancements to the Code Editor (such as Class Completion), the Code Browser, and the major debugger enhancements, you'll likely conclude that Delphi 4 is indeed a major step forward. But what about the criticisms?

Criticisms. One of the first criticisms after version 4's release concerned Inprise's decision to not supply some of the printed manuals that had accompanied earlier versions. I disagree with this criticism, because the Help files accompanying the initial Delphi 4 release were much better than those that accompanied the first release of Delphi 3, and the printed manuals are available at a reasonable price.

Soon after the release, vigilant users began to discover bugs, the most infamous of which was the *ItemIndex*-bug in *TCustomListBox*. For a detailed account of the bug and a fix (if you haven't downloaded and installed the maintenance release), see Mark Miller's excellent Internet article at <http://www.eagle-software.com/FixingTheItemIndexBug.htm>. This bug was particularly problematic because it could potentially affect *TCustomListBox*'s direct decedents and decedents of *TCustomDBGrid*. In all, seventeen components were affected (again, see Miller's article for details).

Here's a bug you can investigate: Drop two *TPanel* components onto a form, setting the *Align* property of the first to *alRight*, and the *Align* property of the second to *alLeft*. Change the *Align* property of the second

panel first to *alRight*, then back to *alLeft*, and finally back to *alRight*. Whoops! Where did that panel go? Do you notice anything strange about the left border of the form? That's right. The right border of the panel coincides with the left border of the form — and you can't drag the panel back into view unless you change its *Align* property again.

And there are others still. How did this situation come about? The consensus among many I spoke with at the Inprise Conference was simply that Delphi 4 was released too soon. There needed to be more rigorous testing to ensure that this essential tool — mission critical for so many developers — was as bug-free as possible. Even at the cost of delaying the release for a month while beta testing continued, ensuring the usability of the VCL components would have been worth it in the long run.

With its enhanced IDE, its new native Internet components, its productivity tools, and its expanded Object Pascal language, Delphi 4 represents a giant step forward; in my opinion, it's the best release yet. But Inprise must consider the results of the apparently premature release of Delphi 4 (with its VCL bugs) and take steps to avoid this in the future. Then all of us — Inprise and the host of Delphi users — can look to the future with confidence, anticipating solid, reliable, and impressive Delphi upgrades each year. ▲

— Alan C. Moore, Ph.D.

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