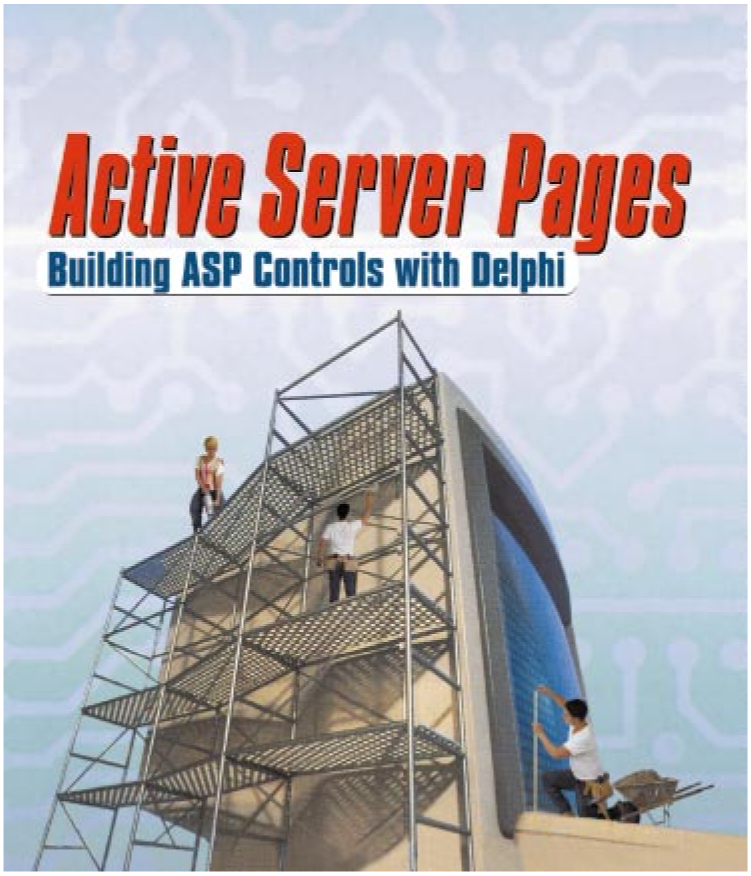
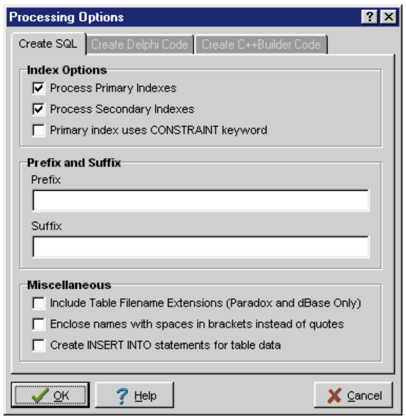
March 1999, Volume 5, Number 3



Cover Art By: *Darryl Dennis*

OCERIS Ships AutoSQL 2.1

OCERIS, Inc. announced *AutoSQL 2.1*, which takes existing Paradox, dBASE, and Access 95/97 tables and creates three types of output. The first is CREATE TABLE and CREATE INDEX statements for creating the tables in a SQL environment. It will optionally create INSERT INTO statements for the data inside the tables. The second and third outputs are Delphi code compatible with versions 2, 3, and 4, and C++Builder code compatible with C++Builder 3.


Processing Options

Create SQL | Create Delphi Code | Create C++Builder Code

Index Options
- [✓] Process Primary Indexes
- [✓] Process Secondary Indexes
- [ ] Primary index uses CONSTRAINT keyword

Prefix and Suffix
Prefix
Suffix

Miscellaneous
- [ ] Include Table Filename Extensions (Paradox and dBase Only)
- [ ] Enclose names with spaces in brackets instead of quotes
- [ ] Create INSERT INTO statements for table data

OK | Help | Cancel

OCERIS, Inc.
Price: US$49.95
E-Mail: oceris@oceris.com
Web Site: http://www.oceris.com

Component Store Announces SQLQuery 2.2

Component Store Ltd. announced the release of *SQLQuery 2.2*, an add-on database component designed for use with Delphi. SQLQuery 2.2 can be used to build thin Win32 or ActiveX clients with Microsoft's SQL Server as a back end. New features in version 2.2 include direct update without *TSQLUpdate* through the *UpdateParams* property, support for optimistic concurrency in *UpdateParams*, asynchronous Open and ExecSQL, InfoPower support, local sort of fields in a result set, and added dataset events for easier SQL error handling.

Component Store Ltd.
Price: US$249 for a single-developer license; multiple developer and site licenses are available.
Phone: (800) 903-4152
Web Site: http://www.component-store.com

toolsfactory Announces ClassExplorer Pro 2.1

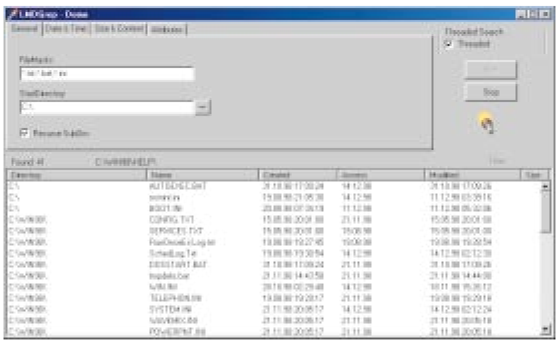
toolsfactory GmbH (LLC) announced *ClassExplorer Pro 2.1*, an integrated software development tool that provides object-oriented code navigation, creation, and documentation for Inprise's Delphi and C++Builder development environments.

ClassExplorer Pro 2.1 supports features such as Class View and Class Hierarchy, which simplify the source view and class navigation of complex projects. Class member creation features simplify the creation of methods, properties, and fields to classes. Code documentation features offer customizable, automatic, online-help generation from source code, including indexes and hierarchical tables.

toolsfactory GmbH (LLC)
Price: US$99
E-Mail: sales@toolsfactory.com
Web Site: http://www.toolsfactory.com

LMD Innovative Releases LMD-Tools 4

LMD Innovative released the *LMD-Tools 4* component package, a set of native Delphi VCL components and routines for various programming tasks.



Version 4 includes customizable, transparent edit and memo controls (without text limitations); an extensive FileGrep component; Calendar controls; an enhanced dynamic-splitter component; dockable toolbars; and improved handling of data containers for bitmap-/wave-files or other data resources (now supporting native data compression).

Both the Standard and Professional editions include over 150 components for Delphi 3 and 4 and C++Builder 3, online help, over 60 demonstration projects, and WPTools Light (LMD-Edition), RTF-Editor, and RTF-Label. Version 3.5 is also included for compatibility with Delphi 1 and 2 and C++Builder 1.

LMD Innovative
Price: Standard Edition, US$149; Professional Edition, US$199 (includes source code of the component library, additional add-ons, and C++Builder support).
Phone: +49 271 355489
Web Site: http://www.lmd.de

Raize Announces Raize Components II

Raize Software Solutions, Inc. announced ***Raize Components II***, the next generation of the company's library of native VCL controls for Delphi and C++Builder.

Raize Components II introduces over 30 new components, including *TRzCheckTree*, *TRzBackground*, *TRzButton*, and *TRzEditListBox*.

TRzCheckTree is a tree view control that associates a checkbox with each node in the tree, and automatically updates the states of parent and child nodes when the state of the current node changes. The *TRzBackground* component enables developers to add gradients and tiled textures to forms, including MDI frames. The *TRzButton* component supports multi-line captions, 3D text styles, and custom button face colors. The *TRzEditListBox* supports automatic run-time editing of items in the list using a popup edit window.

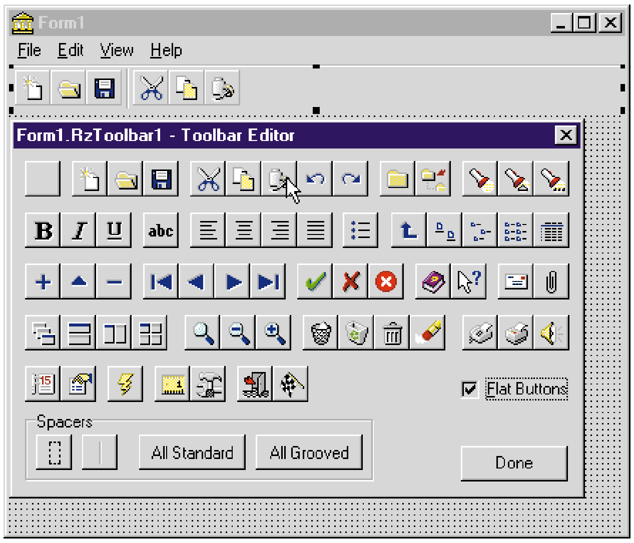
Raize Components II also introduces custom framing properties, which allow developers to select which sides of the frame will appear. As a result, all the Raize Components can appear as line-style controls. In addition, the custom framing properties support showing a second frame style whenever the mouse is positioned over the control, or the control receives the input focus.

The majority of components in Raize Components II have an associated context menu that provides quick access to common settings and properties without requiring the developer to switch to the Object Inspector and search for properties.

Raize Components II introduces specific features for Delphi 4 developers. For example, the *TRzPanel*, *TRzSizePanel*, and *TRzSplitter* components utilize a custom docking manager to manage controls docked in their client areas. The new docking manager displays the captions of docked controls instead of the default "grabber" bars.

Raize Components II provides support for Delphi versions 1, 3, and 4, and C++Builder 3.

Raize Software Solutions, Inc.
Price: US$249
Phone: (630) 717-7217
Web Site: http://www.raize.com



Primoz Gabrijelcic Announces GpProfile 1.1

Primoz Gabrijelcic announced the availability of ***GpProfile 1.1***, a profiler for Delphi 2, 3, and 4.

GpProfile 1.1 is a source-instrumenting profiler that works with Windows 95/98 and NT 4/5. It features multi-threaded program support, the ability to instrument procedures (written in built-in assembler), the ability to show/hide an integrated Source Preview window, a syntax-highlighted source preview, and an API for profiling control. With GpProfile 1.1, profiling results can be exported to standard delimited format.

In addition, GpProfile 1.1 offers conditional API execution with metacomments, a layout manager, the ability to display and browse caller/called hierarchy, context-sensitive help, and free, complete source code (Delphi 4).

Primoz Gabrijelcic
Price: Free
E-Mail: primoz.gabrijelcic@altavista.net
Web Site: http://www.eccentrica.org/gabr

CenturionSoft Announces EuroFonter

CenturionSoft announced ***EuroFonter***, a utility that adds the euro symbol to all TrueType fonts. Any document created and/or received will display the correct symbol, avoiding the risk of dangerous misunderstandings.

EuroFonter offers a wizard-style interface that makes it easy to install and use.

CenturionSoft
Price: US$39.95
Phone: (202) 293-5151
Web Site: http://www.centurionsoft.com

Datasoft Reveals GhostFill SDK

Datasoft (Pty) Ltd announced the release of the *GhostFill Software Developers Kit (SDK)*, a developers kit for the company's productivity add-in for Microsoft Word.

GhostFill is a document assembly tool that simplifies and expedites the production of complex documents. With the SDK, developers can integrate with GhostFill and produce documents from within their applications — even across the Internet.

The GhostFill SDK includes an ActiveX control for integration with Delphi, Microsoft Visual Basic and Visual C++, and others. It also ships with a set of sample applications, comprehensive documentation, and a fully functional copy of GhostFill.

GhostFill's COM-based architecture encourages developers to extend its functionality by adding custom OLE automation servers to its environment.

Datasoft (Pty) Ltd
Price: Free for download.
Phone: +27 21 683 4680
Web Site: http://www.ghostfill.com/sdk

Eagle Software Releases CDK 4 and reAct 4

Eagle Software announced *CDK 4*, a suite of code generation and modification wizards that integrate with Delphi 4. The company also announced ***reAct 4***, a test program generator for Delphi.

CDK 4 includes wizards for building descending, composite, business, linking, embedded, and dialog components, as well as property editors, component editors, and packages.

The CDK Package Wizard allows developers to build package sets consisting of a design-time package and one or more run-time packages. CDK separates and maintains run-time and design-time code, so users are free to focus on the essence of class design.

CDK 4 can also modify any existing Delphi source code. CDK's modification engine parses, then folds, new code directly into the source file.

CDK 4 allows users to edit generated source code manually (in Delphi) and continue to modify the source with CDK wizards interchangeably. Also, CDK generates all code in the user's coding style, so reformatting by hand after generation is unnecessary.

CDK 4 employs a code reuse engine, allowing users to drag and drop "smart code" (CDK Templates) into their class designs. CDK 4 ships with 34 CDK Templates and a wizard for creating custom CDK Templates.

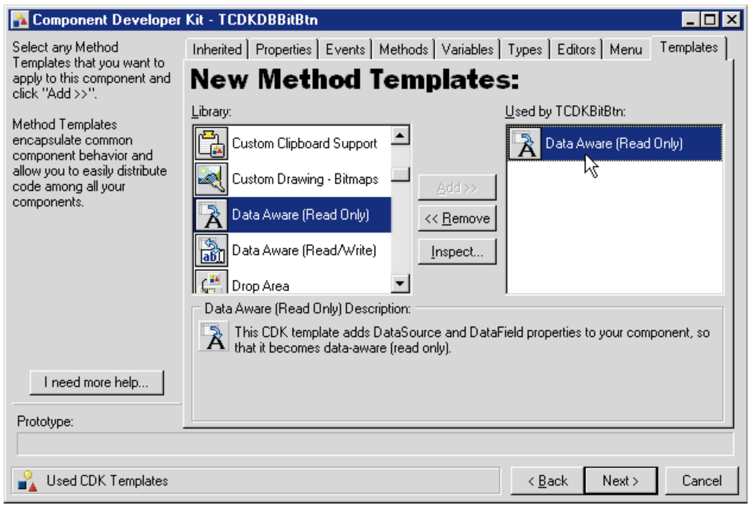
Also announced was reAct 4, which generates the code needed to evaluate any selected component. reAct 4 test programs consist of a run-time component inspector, a test form, and an event log.

With reAct 4, users can dynamically create and destroy instances of the test component, view and change properties, and see the effects of those changes on the test component at run time. Users can see events as they occur; control panel lights flash when corresponding events are triggered.

reAct 4 also includes built-in streaming tests, making it easier to verify the component's ability to save and load its state information to a file.

Because test programs generated by reAct are Delphi programs, users are free to modify and enhance the generated source as needed. reAct 4 is fully compatible with other third-party testing tools that work with Delphi.

Eagle Software
Price: CDK 4, US$289 per copy; reAct 4, US$139 per copy; site licenses are available for both products.
Phone: (310) 441-4096
Web Site: http://www.eagle-software.com



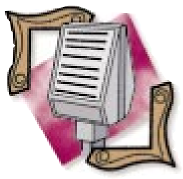
Select any Method Templates that you want to apply to this component and click "Add >>".

Method Templates encapsulate common component behavior and allow you to easily distribute code among all your components.

This CDK template adds DataSource and DataField properties to your component, so that it becomes data-aware (read only).

News LINE

March 1999



Inprise Launches CORBA and Java Tour for Enterprise IT Managers

Scotts Valley, CA — Inprise Corp. announced a North American series of management-level CORBA and Java seminars designed to show IT executives the strategic business benefits of large-scale, platform-independent, and standards-based enterprise application development.

Distributed Enterprise Solutions on Tour consists of two one-day seminars: "CORBA Essentials for Effective Internet Computing" and "Building Distributed Applications with Java." Participants can attend one or both days of the series.

"CORBA Essentials for Effective Internet Computing" will cover the role CORBA plays in enterprise systems and how to use distributed objects to bring high performance to intranet and Internet applications, whether they are written in Java, C++, or other languages.

"Building Distributed Applications with Java" will include discussions on the design and development of distributed systems, the deployment and management of Java applications, the benefits and drawbacks of two- and three-tier computing, and the use of Enterprise JavaBeans as standardized reusable *n*-tier components.

For more information on Distributed Enterprise Solutions on Tour, and to register for a seminar, visit http://www.inprise.com/events/seminars, or call (800) 255-4388. The standard registration fee is US$395 per person for one day, and US$665 for both days. Special corporate/group discounts are available.

Inprise Delivers Enterprise Application Server Solution

New York, NY — Inprise Corp. introduced the Inprise Application Server, a solution that accelerates and simplifies the development, integration, deployment, and management of distributed enterprise applications. The Inprise Application Server enhances an enterprise's competitiveness by streamlining its IT processes and application lifecycle.

Key features of the Inprise Application Server include support for all major programming languages, client interfaces, Web servers, database servers, hardware platforms, and legacy environments, including DCE and COM; an object-communications infrastructure built with Inprise VisiBroker; an open and extensible architecture based on industry standards, such as CORBA, C++, Java, and HTML; support for the Sun Solaris, HP-UX, IBM AIX, and Microsoft Windows NT platforms; JBuilder for Application Server; Web deployment of Internet-based, platform-independent applications; VisiBroker Integrated Transaction Service (ITS); and AppCenter, a distributed applications-level management tool.

For more information and pricing, call Inprise direct corporate sales at (831) 431-1064, or visit http://www.inprise.com/appserver.

Inprise Announces New Version of MIDAS

Scotts Valley, CA — Inprise Corp. announced version 2 of Inprise's MIDAS (Multi-tier Distributed Application Services), which simplifies and hastens the development, integration, and deployment of thin-client, distributed-database applications.

MIDAS speeds data access across all application tiers, from the client to the database server, through remote-data access and intelligent data synchronization. Enterprises can now develop MIDAS applications with all of Inprise's enterprise tools, including JBuilder, Delphi, and C++Builder. In addition, MIDAS 2 supports Java, CORBA, and COM/MTS. MIDAS 2 now includes support for intelligent master/detail and nested tables, as well as Oracle8i object/relational databases. Finally, through integration with the Inprise Application Server, customers can manage their MIDAS-produced applications.

One of the key offerings of MIDAS 2 is MIDAS Client for Java, which simplifies the development of cross-platform, Pure Java thin clients for distributed-database applications. MIDAS Client for Java includes a set of Java Beans, or components, designed for JBuilder 2. These Pure Java components give developers cross-platform client access to high-performance multi-tier MIDAS applications.

For more information and pricing, call Inprise direct corporate sales at (831) 431-1064, or visit http://www.inprise.com/midas.

LEAD Announces VCL Components for Delphi

LEAD Technologies, Inc. announced support for Inprise Corp.'s VCL (visual component libraries). The new LEADTOOLS VCLs will offer developers the same imaging functionality as the LEADTOOLS ActiveX controls, but will provide this functionality in a native Delphi component format. The VCL will be available throughout the LEADTOOLS product line.

By Ron Loewy

Active Server Pages

Building ASP Controls with Delphi

Active Server Pages (ASP) is at the foundation of Microsoft's Web development architecture. ASP is an extension to Microsoft's IIS (Internet Information Server) Web server, and is available free of charge on any operating system that has IIS or Microsoft's PWS (Personal Web Server) installed. This means most new machines with Windows 98 or Windows NT come with ASP. Adding ASP to older Windows 95/NT installations is easy; the code is available for download from Microsoft's Web site. It's also shipped with several other Microsoft tools, such as Visual InterDev, FrontPage, etc.

ASP is implemented as an ISAPI extension to IIS. Think of it as an optimized CGI program for Microsoft's Web servers that can serve HTML pages created dynamically using some logic. These are not your run-of-the-mill, static HTML pages; code is used to process input from the user, access databases, or use some other mechanism to create what users see in their Web browsers.

Writing ASP applications is relatively easy. ASP code is a combination of HTML and scripts written in JavaScript, VBScript, or any other ActiveScript-capable language engine. Microsoft provides a set of predefined Automation objects that can be used from ASP applications. With some of the common ActiveX and Automation objects available on Windows, writing database-driven applications for the Internet and/or an intranet is easy. (See my article "Much ADO about the Web" in the December, 1998 *Delphi Informant*.) You can write ASP applications using your trusty copy of Notepad (or EDLIN, if you're a real masochist), but most people like to use one of the tools that support ASP development. Microsoft's Visual InterDev comes to mind, and there are many other third-party solutions to choose from.

Extending the functionality of ASP-writing Automation objects is easy. The ADO article I just mentioned shows how to create a simple Automation object that can access a database using ADO, and can be used from an ASP application. The Automation object developed in that article did not know it was used from an ASP application, and could be used from any COM-enabled development tool. In this article, we'll explore the ASP object model, and discuss how to take advantage of it from a Delphi-developed Automation object. We'll also create the Automation object (the project described in this article is available for download; see end of article for details).

The ASP Object Model

Most of Microsoft's new APIs appear as a collection of objects that represent the task at hand. In Microsoft speak, this is commonly referred to as an *object model*. Some of the examples for the use of object models as APIs include the Document Object Model (DOM), which allows object-oriented access to the contents of an HTML document; the ActiveX Data Objects (ADO), which encapsulate the database access primitives as a set of objects; and true to form, the ASP object model, which represents the transactions and entities that create a Web application as a set of objects.

There are five objects you'll need to learn to write efficient Web applications with ASP:

1) The *Request* object is used to hold information about the request the user sent when accessing the application.
2) The *Response* object is used to encapsulate the HTML response the application will send back to the browser.
3) The *Server* object is used to manage the ASP environment. You will usually use it as a launching pad for your own ActiveX and Automation objects.
4) The *Session* object is used to store information specific to the user of the application. This allows the application to maintain state for a specific user (which requires using expanded URLs, or cookies in raw CGI programs).
5) The *Application* object is used to store and manage application information shared among all sessions of the application.

The *Request* Object

The *Request* object encapsulates the information sent from the client's browser to the Web server when the user requests a resource from the ASP application. To understand the *Request* object, let's look at what happens when a user clicks on a link to a page in the ASP application:

- The user clicks on the link's title.
- The browser uses the anchor information defined in the <A...> tag that defines the link to find the server that hosts the ASP application. After a connection is established, an HTTP GET request is sent to the server with the path to the ASP page.
- If the URL had additional information to pass to the application using the HTML `URL?Parameters` syntax, the parameters are sent to the application as part of the request.
- Additional information about the user and the browser is sent in HTTP headers along with the request.
- If the user's browser has had cookies defined in the past for the resource, these cookies are sent in additional HTTP headers.

If the user fills out a form and clicks the **Submit** button that connects to the ASP application via a <FORM ...> tag (whose action points to the ASP application), the request sequence is repeated. Usually, however, an HTTP POST request is sent, and the names of the variables in the forms and the values the user entered are sent after the HTTP headers.

When the request is received by the Web server, it determines this is a request for an ASP script, using the main program's extension (e.g. mypage.asp), and the ASP interpreter is activated with the request information.

ASP parses the request information and allows you to access it as properties and collections of the *Request* object. The *QueryString* property, for example, includes all the parameters passed in the URL after the ? character. If, for example, the URL was http://path-to-server/mypage.asp?Name=Ron, we could find the value of the *Name* parameter using the following syntax:

```
Your Name is <%= Request.QueryString("Name") %>
```

The *ServerVariables* collection provides information about pre-defined environment variables that were passed by the Web server. For example:

```
Request.ServerVariables("PATH_TRANSLATED")
```

will return the path of mypage.asp on the local machine's file system. This can be very useful if you need to access external media, or other files that are installed in the same directory as your ASP pages.

The *Form* collection provides information to values passed in form controls. If, for example, our user entered a credit-card number in a form defined using the following HTML code snippet:

```
<Form Action="http:// path-to-server/mypage.asp">
...
Credit Card Number <input name="CreditCard" Size=16>
...
```

we can access it using the following syntax (in JavaScript):

```
CodeToCheck = Request.Form("CreditCard");
```

Finally, cookies can be accessed via the *Cookies* collection:

```
Last time you visited was <%= Request.Cookies("LastVisit") %>
```

You can do more with the *Request* object and the different properties and collections it provides. Any good ASP reference will include all the information.

The *Response* Object

The *Response* object allows you to create the output the user will see in the browser as a result of his or her request. It provides access to the HTTP headers that are sent with the request, and allows you to build the response from start to end.

Because the result is built sequentially from the headers to the end of the response, you have to set the values of the headers, cookies, etc. before you write the content of the result. The *Response* object has a property called *Buffer* that, when set to True, caches all the output, and doesn't send it to the user until the *Flush* or *End* method has been called. If you don't want to write sequentially to the output, remember to set *Buffer* to True at the start of your ASP page.

The following properties can be used to set the HTTP headers of the page:

- The *Cookies* collection can set cookies that are related to your ASP page. You will get these cookies in the *Request* object the next time the user connects to your page.
- The *ContentType* property sets the type of response you send. By default, `text/html` is assumed, but if your response is an image, you'll need to set it to `image/gif`, `image/jpeg`, etc.
- The *Status* property sets the status that is returned. By default, "200 OK" is returned, and the browser will dis-

play the response. If, however, the user did not provide information, you can set a different response (e.g. "401 Unauthorized" if the user's password wasn't found in your database).

- The *Expires* (or *ExpiresAbsolute*) property can be used to define when the response expires from the browser's cache, and the browser needs to connect to the server again to receive an updated page. Assume, for example, that you write a stock ticker application that displays up-to-date stock prices. Assuming your database is updated every 15 minutes, set *Expires* to `15` to ensure that if the user tries to access the page in 10 minutes, no network bandwidth will be used. But if the user tries after 20 minutes, he or she will get updated information.

You can also write any HTTP header using the *Response* object's *AddHeader* method. Another method related to headers is the *Redirect* method. Use this method to redirect the browser to a new URL. This can be useful if you need to route users to different URLs based on their cookie values, or other information they provide.

After the headers have been set, it's time to create the content. ASP creates the content from the HTML code in the page; your scripts or Automation objects can write to the output stream using the *Response* object's *Write* method. For example:

```
<%
  if (SomeCondition) {
    Response.Write("xxxx")
  }
  else {
    Response.Write("yyyy")
  }
%>
```

Here, the *Response* object is used to change the output based on some condition.

The *Write* method assumes you're writing to an HTML output (ContentType text/html) and will automatically translate your strings to valid HTML representation, e.g. `>` will be translated to `&gt;`. If you want to write values that won't be translated (for example, when creating a GIF image), use the *BinaryWrite* method.

The *Server* Object

The *Server* object has several utility functions useful to an ASP application. *HTMLEncode* and *URLEncode* take plain string data and convert it to string data that can be included in HTML source. For example:

```
<%= Server.HTMLEncode("A < 2") %>
```

will be translated to:

```
A &lt; 2
```

The browser will be able to display this code properly (A < 2), where, if you wrote `A < 2` in the source, the browser will get confused and think that `< 2` ... is a tag it doesn't recognize, and will ignore your code.

The most important function of the *Server* object is the *CreateObject* function. This function is used to start an Automation object. As a Delphi programmer, the *CreateObject* function is the way you can start your Delphi-developed objects to interact with an ASP application. The *CreateObject* function takes a *ProgID* as its parameter. You can determine the *ProgID* of your Delphi object from the Type Library editor. It's the name of the library separated by a dot from the name of the Automation object, e.g. *MyLib.MyObj*.

The *Session* Object

HTTP is a stateless protocol. When a client connects a Web server, the Web server answers and disconnects. The next time you connect to the server, the server has no indication that you called it earlier, and has no way to tell your connection from a connection made by another user hundreds or thousands of miles away from you.

Imagine a simple shopping application. The user wants to browse the available products and collect them in a virtual shopping basket. When finished browsing, the user advances to the checkout line and pays for the selected items. If your application doesn't save the state of the user's shopping basket, how will you know what items the user wants to purchase? How will you be able to differentiate between user A, who wanted to purchase the US$300 Magic food processor, from User C, who wants two laser printers and 15 network cards for a total of US$894?

CGI programs usually solve the state problem using one of two methods: expanded URLs that are generated dynamically by the application, or cookies stored in the user's cookie file. Both approaches are cumbersome. ASP uses the cookies approach behind the scenes and exposes the state information using the *Session* object.

While the *Session* object has some properties that will allow you to recognize the user, or to time out or abandon the session, its most important use is as a storage space for session information. Coming back to the shopping basket problem, you could store the information about user C's chosen products using the following:

```
Session("LaserPrinter") = 2;
Session("NetworkCards") = 15;
Session("FoodProcessor") = 0;
```

The *Session* object's *Session_OnStart* event is called when a new session is created. The code in this event can be used to create session objects, or initialize variables used by the session. The *Session_OnEnd* event is called when the session is terminated.

The *Application* Object

The *Application* object is used to store information that is global in scope to the application, and is shared between all the users of an application. The *Application* object can also be used to start Automation objects used across the application. The *Application_OnStart* event is activated before the first *Session* is created, and can be used to initialize global application variables, or start global Automation objects. The *Application_OnEnd* event is the last event called when the last session used in the application quits.

Delphi and ASP

To write code that takes advantage of the ASP object model, you must start by importing the ASP type library to Delphi. I use tlibimp.exe, which has been available in Delphi's \bin directory since Delphi 3.02. Executing tlibimp.exe on the file ASP.dll, installed with ASP, results in ASPTypeLibrary_TLB.pas and ASPTypeLibrary_TLB.dcr. We'll use the Pascal file in our Automation object. (You'll have to look for asp.dll on your hard disk. I found the version that came with the copy of PWS installation on Windows 95 under C:\Windows\System\INetSrv). Once you import the ASP type library, you're ready to create your Automation object with Delphi.

Like every other Automation object, you'll want to start by creating an ActiveX library; click on **ActiveX Library** on the ActiveX page of the New Items dialog box (**File | New**). I saved my library as DIASP.dpr in my work directory. Now add an Automation object to the project (also from the New Items dialog box). I gave the name ASPObject to the new class, and saved the implementation unit as ASPObj.pas.

Every Automation object that wants direct access to the ASP objects needs to add the ASPTypeLibrary_Tlb unit created when we imported the ASP DLLs to the **uses** statement. We can now add a reference to a scripting context in the object definition. A scripting context is an ASP interface named *IScriptingContext* that provides our object with access to the ASP objects in the context of the session that uses the object.

Our class definition will now look like this:

```
type
  TASPObject = class(TAutoObject, IASPObject)
  private
    FScriptContext: IScriptingContext;
  public
    property ScriptContext: IScriptingContext
      read FScriptContext;
  end;
```

When an Automation object is used by an ASP application, the ASP engine checks if the object implements the *OnStartPage* method before any page processing is performed. If this method is implemented, it is called and a scripting context is passed to the object. From the Type Library editor, I added a new *OnStartPage* method (the completed type library is shown in Figure 1).

I created five *OleVariant* variables to represent the main ASP objects, and I assigned them in this method:

```
procedure TASPObject.OnStartPage(
  AScriptingContext: IUnknown);
begin
  FScriptContext   := AScriptingContext as IScriptingContext;
  FASPRequest      := ScriptContext.Request;
  FASPResponse     := ScriptContext.Response;
  FASPSession      := ScriptContext.Session;
  FASPServer       := ScriptContext.Server;
  FASPApplication := ScriptContext.Application;
end;  // TASPObject.OnStartPage
```

We can now create a simple *HelloWorld* procedure to test our object (again, use the Type Library editor to add the method):

```
procedure TASPObject.HelloWorld;
begin
  ASPResponse.Write('<h2>Hello World</h2>');
end;  // TASPObject.HelloWorld
```

The code for our object is ready. We need to compile the project and register the ActiveX server (**Run | Register ActiveX Server**).

To test the object, I created a simple HelloWorld.asp, and installed it in a virtual directory defined in my Web server as DIASP. This directory must have read and script authorization.

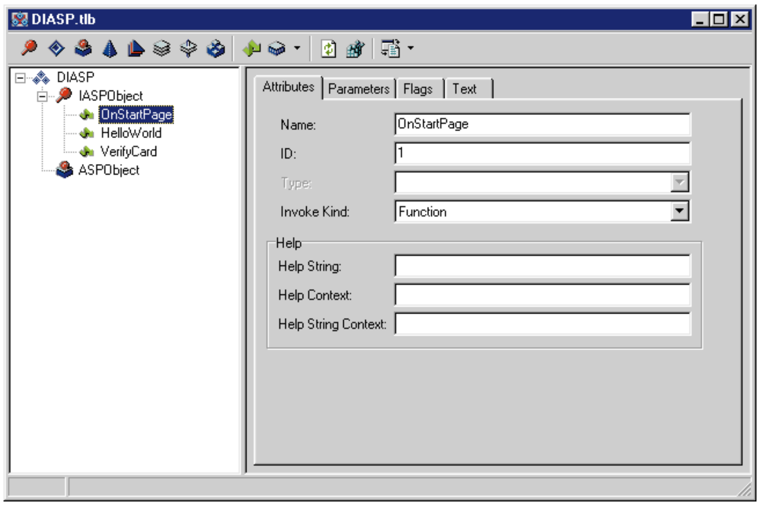
The code that calls our object from the ASP file is simple:

```
<%
Set ASPObj = Server.CreateObject("DIASP.ASPObject")
%>
<H3>This page uses the ASP aware Delphi automation object </H3>
<% ASPObj.HelloWorld %>
```

The result is shown in Figure 2. Notice that our Delphi code accessed the output stream directly via the ASP *Response* object.

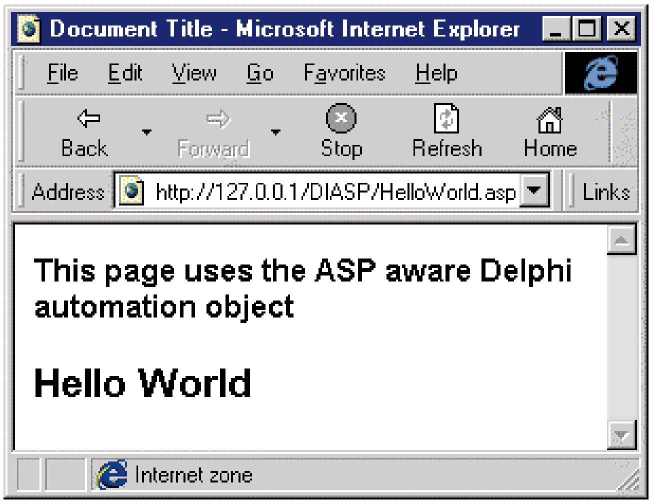
Why Use Delphi?

It's obvious from this sample that it is easy to access the



Name: OnStartPage
ID: 1
Type:
Invoke Kind: Function
Help String:
Help Context:
Help String Context:

Figure 1: Delphi's Type Library editor.


Document Title - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home
Address http://127.0.0.1/DIASP/HelloWorld.asp
Links
Internet zone

This page uses the ASP aware Delphi automation object

Hello World

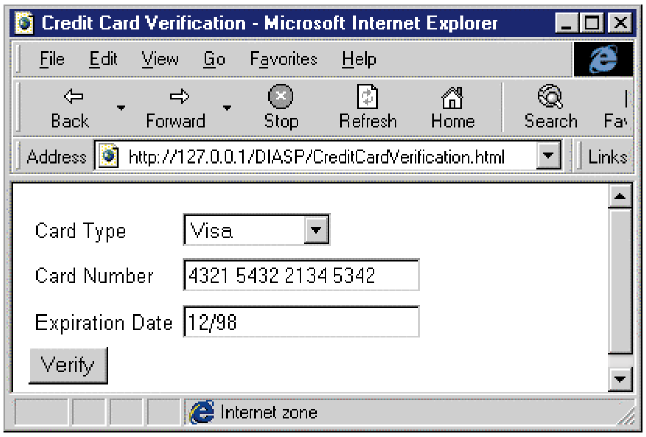
Figure 2: A simple HelloWorld.asp.

```
procedure TASPObject.VerifyCard;
var
  VerifyObject: TCreditCardVerify;
begin
  VerifyObject := TCreditCardVerify.Create(nil);
  try
    VerifyObject.CardNumber :=
      ASPRequest.Form('CardNumber');
    VerifyObject.SetCardTypeByName(
      ASPRequest.Form('CardType'));
    VerifyObject.SetExprDateFromStr(
      ASPRequest.Form('ExprDate'));
    ASPResponse.Write('Checked Validity for Card Number ' +
                      VerifyObject.CardNumber + '<br>');
    case VerifyObject.Valid of
      ccvValid   : ASPResponse.Write('Card is valid!');
      ccvExpired : ASPResponse.Write('Card Expired!');
      ccvInvalid : ASPResponse.Write(
                    'Card did not pass validation!');
    end;
  finally
    VerifyObject.free;
  end;
end;
```

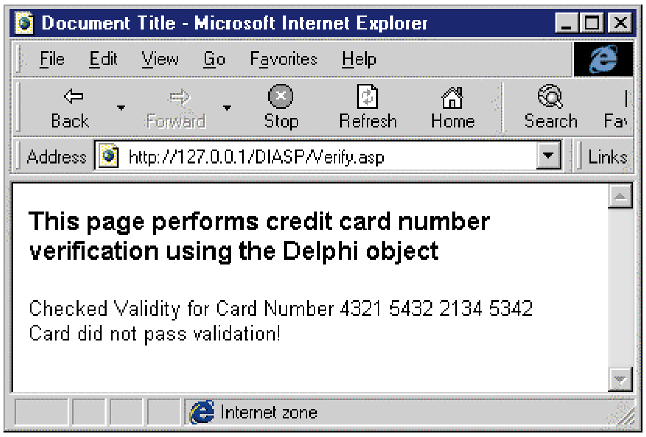
Figure 3: The *VerifyCard* method uses the credit-card verification component.

ASP objects from our Delphi code, but why would we even bother to do that if we can write the same code in VBScript or JScript?

Using Delphi to interact directly with the ASP objects provides several advantages. The first is speed; a compiled Delphi object will be much faster than a script that needs to be interpreted by the Active Script engine used by ASP. If the logic you need to perform is complicated, your users will receive better service if the calculations are performed in a compiled module. The other advantage to using Delphi to write your logic is the power of the language, and the ability to use existing code. I would not bother with Delphi code for simple ASP scripts, but when the need for complicated logic or business rules arises, I would rather use Delphi's strong development tools, debugging


Credit Card Verification - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Fa
Address http://127.0.0.1/DIASP/CreditCardVerification.html
Links
Verify
Internet zone

Card Type: Visa
Card Number: 4321 5432 2134 5342
Expiration Date: 12/98


Document Title - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Fa
Address http://127.0.0.1/DIASP/Verify.asp
Links
Internet zone

This page performs credit card number verification using the Delphi object

Checked Validity for Card Number 4321 5432 2134 5342
Card did not pass validation!

Figure 4: After the verification component does its job, the result is reported back using the ASP *Response* object.

facilities, and existing code modules over the primitive tools available for script development.

Let's create, for example, a credit-card validation routine in Delphi. Assume your Web page receives credit-card orders, and you need to verify that the credit-card number is valid. Usually, you'll need some software that can connect to verify the credit-card information supplied and use your merchant account, but for the purpose of this article, we'll assume that using the credit-card number validation algorithm is enough. While this algorithm won't rival the scheduling problems the NT operating system developers had to face, I wouldn't want to implement it in VBScript.

The CcardVer.pas unit defines a Delphi object *TCreditCardVerify* that, based on the properties *CardType*, *ExprDate*, and *CardNumber*, will return the validity of the card information. We will now add a new method to our object that will access credit-card information sent from an HTML form, verify it, and return a response to the user. (Don't forget to use the Type Library editor to add the new *VerifyCard* method.)

The code for the new method uses the credit-card verification component, as shown in Figure 3. This code uses the ASP *Request* object to access the parameters passed from the CreditCardVerify.html form. After the verification component does its job, the result is reported back using the ASP *Response* object (see Figure 4).

Conclusion

Creating ASP-aware objects with Delphi is almost as easy as creating any other kind of Automation object. If you are writing ASP applications, Delphi code can be used to speed calculations and development time.

Many tasks you've done in Delphi can now be exposed on the Web using the best of both worlds: Delphi's ease of code development and performance with ASP's easy deployment and content authoring. If you have large amounts of Delphi code that you need to Web-enable, and ASP is your Web development technology of choice, the technique described in this article will save you hours of porting.

More information about ASP can be found on Microsoft's Web site (http://www.microsoft.com), or as part of the MSDN that ships with Visual InterDev or Visual Studio. Many authoring tools support the creation of ASP pages. Or, you can always fire up Notepad. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\MAR\DI9903RL.

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.

Patterns / OOP / Persistence / Windows registry

By Xavier Pacheco

The Singleton Pattern

Implementing a Reusable Object for Saving Persistent Data

We're surrounded by patterns, both physical and behavioral. With the exception of occasional acts of spontaneity, most people go about their daily activities according to various behavioral patterns. Likewise, in software development, many problems follow a common theme, and the majority of these problems is solvable by applying object-oriented patterns. In a loose sense, patterns are to object-oriented programming as algorithms are to structured programming.

Unlike our behavioral patterns, we can't depend on programming patterns to pop into our heads automatically. First, we must fully understand the problem at hand. Second, we must be able to determine that there is a pattern that addresses the problem. Finally, we must implement a pattern that we've identified, modified, or created to solve the problem.

This is the first of a series of articles in which I will discuss patterns. Throughout this series, I'll give practical examples to implement the patterns I discuss. The examples I'll illustrate may vary from simple Object Pascal usage, to more complex implementations of COM/DCOM and CORBA. My goals are to:

- introduce you to common patterns or variations thereof,
- present practical solutions to common problems using patterns, and
- promote a frame of thinking when trying to solve common development problems.

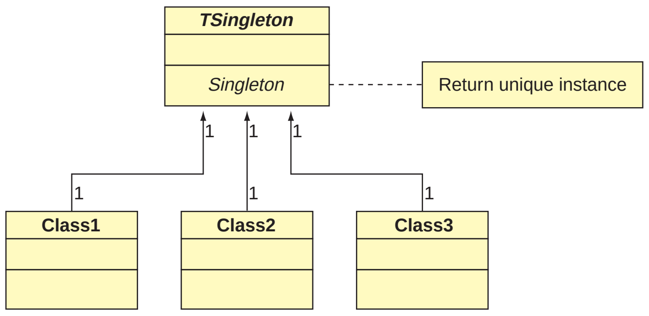


Figure 1: The *Singleton* class structure and its relations.

The Singleton Pattern

The Singleton pattern is a very simple pattern to implement. It's used when you want to ensure that only one instance of a class exists. Additionally, the Singleton pattern is globally available throughout the application. Ideally, it can be overridden to allow programmers to extend the class without necessitating modification to the client code. This last point is not always practical, but ideal if possible. Figure 1 illustrates the structure of the *Singleton* class and its relationship to other classes.

There are several scenarios in which you might need to implement a Singleton pattern. The most common is to encapsulate a particular set of data and/or behaviors so they're available as one instance. This will ensure the data isn't replicated elsewhere in the application, and changed so it becomes invalid. For example, you might want to ensure that an opened file is accessible from only one location within your application.

Some common examples of Singletons used in the VCL are the *TApplication* and *TScreen* classes that exist in all Delphi applications. Others include *TPrinter* and *TClipboard*. I'll illustrate how to implement a Singleton for encapsulating application-wide user configuration data.

Implementing a Singleton Pattern Class

The following list specifies the steps required to implement a *Singleton* class:

1) Define private variables to ensure proper initialization for the *Singleton* class.
2) Define an access function that returns the *Singleton* instance. This function shall create the instance if it's not already created. Otherwise, it will return the previously created instance.
3) Override constructor to test if the object has already been created, or the constructor is called directly. If so, raise an error.
4) Override the *Singleton*'s destructor to destroy a private instance, and set the *Singleton* variable to **nil**.

In the following sections, I'll demonstrate how to create a skeleton *Singleton* class. Later, I'll use this same skeleton to solve a more practical problem.

Defining the internal private variables. Listing One (on page 16) presents a skeleton *Singleton* class, *TSingleton* (this and all accompanying source code is available for download; see end of article for details). Notice I've declared private variables in the **implementation** section of this unit. The purpose of these variables is described in Figure 2.

Defining the access function. The access function for the *TSingleton* class is defined as:

```
function Singleton: TSingleton;
```

This function evaluates `FSingleton <> nil` to determine whether the class has already been instantiated. If the class isn't created, the function creates it; otherwise, it simply returns the reference to *FSingleton*. Notice that before calling the *Create* constructor for *TSingleton*, the function sets the *FExternalCreation* variable to False. This is how I enforce the rule that the client can't directly call the *TSingleton* constructor. Notice that the *Create* constructor raises an exception if *FExternalCreation* is True — the default value for this variable. Therefore, you'll see that the only way to access the *TSingleton* class is to call the *Singleton* function.

Another, and possibly easier, approach would be to make the constructor protected rather than public. I wanted to point out both options, and, because the latter is easier, I won't illustrate it, other than to give it mention.

Notice that I've also provided an alternate access method, a class method defined as:

```
class function Singleton: TSingleton;
```

This method simply calls the access function, *Singleton*. I created this method to illustrate yet another technique for providing an access method.

Define constructor and destructor. As noted earlier, the constructor evaluates the private variable *FExternalCreation*, and raises the appropriate exception. The destructor does the reverse, and also sets the *FSingleton* variable to **nil**.

The **finalization** section of this unit takes care of freeing the *TSingleton* instance, if it hasn't already been freed by checking if *FSingleton* is not **nil**. *FSingleton* is set to **nil** in *TSingleton.Destroy*.

TSingleton Usage

Using *TSingleton* is simple; it's used similarly to the way you use the *TPrinter* and *TClipboard* classes. For example, to invoke the *TSingleton.ShowSingletonName* method, simply refer to the access function as though it were the class, which in effect it is, because it returns a reference to a valid *TSingleton* instance:

```
Singleton.ShowSingletonName;
```

The first time the client makes a reference to the *Singleton* function, the internal *TSingleton* instance is instantiated. From this point, its existence will be present until the client explicitly frees it, or until the application shuts down, at which time the code in the **finalization** block is executed.

The client may also make reference using its own *TSingleton* variable. This isn't a problem because the client's variable and internal variable will be referring to the same instance. Even if the client frees its own variable reference, another call to *Singleton* will simply cause the internal instance to be instantiated again. Therefore, the following code fragments are valid, even though they don't follow good programming practices. I use them here to make a point; I don't really code like this:

```
var
  S: TSingleton;
begin
  S := TSingleton.Singleton;
  S.ShowSingletonName;
end;
```

In the above code, not freeing the *TSingleton* instance is fine because the **finalization** block will free it when the application shuts down. The following code won't fail:

```
var
  S: TSingleton;
begin
  S := Singleton;
  S.Free;
  Singleton.ShowSingletonName;
end;
```

Variable	Purpose
FSingleton	Private variable to refer to the *TSingleton* instance. Client will not be able to directly access this variable.
FExternalCreation	Boolean to indicate if the client directly called the *TSingleton* constructor. A True value indicates an error. Client must use the defined access function.

Figure 2: The variables declared in the implementation of the *TSingleton* class.

Although the call to *S.Free* destroys the internal *FSingleton* instance, the subsequent call to *Singleton.ShowSingletonName* recreates that instance. The same goes for this code:

```
begin
  Singleton.Free;
  Singleton.ShowSingletonName;
end;
```

Now that I've shown you a generic *Singleton* class, I'll show you a more practical use for this class.

Introduction to *TUserConfiguration*

I've worked on quite a number of projects involving user-interface design, and one requirement consistently arises: persistent user options. Although the options that end users want to save may vary, the concept is almost always the same. When users close their applications, some set of data is saved, so the next time the application is launched, those same options are remembered. In many cases, these options have to do with security, e.g. the username, password, and perhaps a list of accessible screens or routines. In other cases, it has to do with the user interface; things such as the main screen location and size might get saved along with other U/I features, e.g. grid-column widths, pane sizes, fonts, etc.

There are many ways to implement this persistent data, and because this is an article on *Singleton* classes, it seems fitting to illustrate how to implement such an object using the Singleton pattern. I prefer the Singleton solution because it allows the client application's modules to access this data with the assurance that when the data is accessed and/or modified from one module, every module realizes the same data.

Listing Two (beginning on page 16) illustrates a simple implementation of the *TUserConfiguration* class. Although it contains quite a bit more code, it still follows the pattern shown from the basic *TSingleton* skeleton, i.e. *TUserConfiguration* is an extension of *TSingleton*. It contains four additional properties: *UserName*, *Password*, *UserID*, and *MainScreenPos*. The first three properties are simple data types, and *MainScreenPos* is of type *TMainScreenPos*, a *TPersistent* descendant. *MainScreenPos* encapsulates four integer values into which I'll store the boundaries for the main screen.

I defined *TMainScreenPos* as a class because I wanted to illustrate how you can store data in a hierarchical manner by taking advantage of Delphi's streaming mechanism and RTTI (run-time type information). Basically, I'm using the same system that stores your *TForm* properties when creating applications in Delphi. All *TPersistent* classes are streamable. The details of RTTI are beyond the scope of this article, but I'll briefly summarize where I use RTTI functionality in the *TUserConfiguration* class. (If you're interested in a more detailed discussion of RTTI, visit my Web site at http://www.xapware.com.)

Making *TUserConfiguration* Persistent

The intent of the *TUserConfiguration* class is to save data to some type of store. As stated earlier, the *TUserConfiguration* class simplifies this by using the functionality provided by Delphi's streaming system and RTTI. Data is made streamable by encapsulating it as published properties of a *TPersistent* class descendant.

The *TStorage* class handles the actual saving of the data. *TStorage* is an abstract class. Two of its descendants, *TStorageCfg* and *TStorageReg*, store the data in a configuration file and system registry, respectively. *TUserConfiguration* uses the *TStorage* class to save/read user-configuration data. *TUserConfiguration* calls *FStorage.SaveUserConf* to save the data, and *FStorage.ReadUserConf* to read it. *TUserConfiguration* doesn't know the type of storage data is saved to, or read from; it only knows that it uses the two abstract methods of the *TStorage* class. It's the responsibility of the *TStorage* descendants to implement the saving and reading of data. (This brings up the topics of generalization and composition; please see my discussion on these issues at the end of this article.)

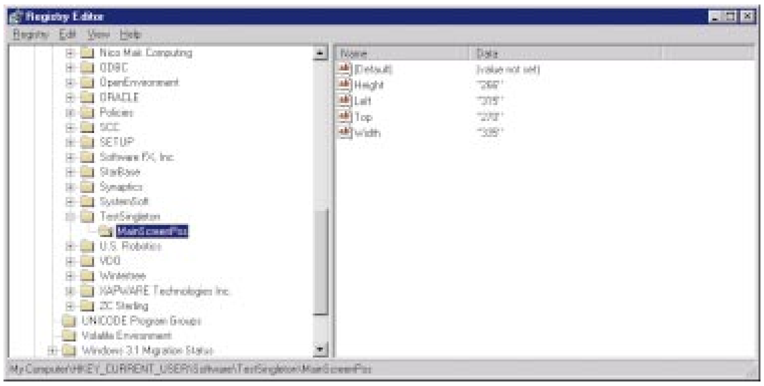
TStorage defines two abstract methods: *SaveUserConf* and *ReadUserConf*. The *SaveUserConf* method is responsible for saving the user configuration information; the *ReadUserConf* method is responsible for retrieving the information back into the *TUserConfiguration* object.

Saving to a file. The *TStorageCfg* class implements the *TStorage* abstract methods to save and read data to and from a separate configuration file. We'll handle this by using a *TFileStream* object. *TStorageCfg.SaveUserConf* saves the data contained by *TUserConfiguration* to a file. In *TStorageCfg.SaveUserConf*, the *TFileStream.WriteComponent* method writes a component and its streamable properties to the file created by the *TFileStream.Create* constructor. Because *TUserConfiguration* is a *TComponent* descendant, its published data, including other objects and their published data, get stored to the file. *TStorageCfg.ReadComponent* does exactly the opposite by calling *TFileStream.ReadComponent* from the file.

Saving to the registry. The *TStorageReg* class is another implementation of *TStorage* that allows data to be saved to the registry. *TStorageReg* defines the field *FRegKey* that holds the location in the registry.

At first glance, the abstract methods for *TStorageReg* seem quite simple, as they both call a single procedure. The *TStorageReg.ReadUserConf* method calls *RegToComponentProps*, whereas *TStorageReg.SaveUserConf* calls *ComponentPropsToReg*. The two procedures are utilities I've written to save and read a component and its properties (including other objects) to and from the registry. They are defined in the unit *XWRegUtils.pas* shown in Listing Three, beginning on page 18.

Notice that both procedures, *ComponentPropsToReg* and *RegToComponentProps*, make use of an internal procedure, *ProcessProps*. Because the reading and writing of this data is practically identical, I put it into a single procedure, and used a Boolean parameter, *AReadProps*, to distinguish save and read operations. This procedure walks through the published



Name	Data
(Default)	(value not set)
Height	"266"
Left	"375"
Top	"270"
Width	"335"

My Computer\HKEY_CURRENT_USER\Software\TestSingleton\MainScreenPos

Figure 3: Streamed data in the system registry.

(streamable) properties of a component, and writes them to the registry. It also cleverly calls itself recursively to write published objects to the registry in the same hierarchical fashion shown in Figure 3. I won't get into the details of how this procedure works, because it has to do with RTTI and streaming. For now, you may be content in knowing that it's very impressive.

TUserConfiguration Miscellany

The rest of the code contained in the *UserCfg.pas* unit has to do with the housekeeping of the *TUserConfiguration* class. As you might expect, you'll see code that creates and frees any internal classes accordingly, and initializes internal variables.

Take note of the *TUserConfiguration.CreateStorage* method. This method creates the proper *TStorage* descendant, based on the value of *TUserConfiguration.FStorageType*. *CreateStorage* is called from *TUserConfiguration*, the *Create* constructor, and the setter method for the *StorageType* property, *SetStorageType*.

Using the *TUserConfiguration* object is simple. Listing Four (beginning on page 19) illustrates how to retrieve the user data in the *OnCreate* event handler for the main form, and how to save this information in the *OnClose* event handler. The *OnChange* event handlers for two *TEdit* components on the form change the *UserName* and *Password* properties for *TUserConfiguration*.

Generalization or Composition

Earlier, while discussing how the *TUserConfiguration* class worked with the *TStorage* class, I mentioned the issues of generalization and composition, two methods of reusability in object-oriented programming. I'm not going to get into the details of these concepts here. You may visit my Web site for further discussions on basic OOP concepts. My implementation of *TUserConfiguration* is based on composition. I'll explain why I chose composition over generalization.

Generalization is the same as inheritance. Had I implemented *TUserConfiguration* using strictly inheritance, it might have looked like the example shown in Figure 4. In the inheritance model, descendant classes take on the characteristics of their ancestor classes. Consequently, these descendant classes typically have some, if not much, visibility into the ancestor classes' internal methods and elements. The book *Design Patterns: Elements of Reusable Object-Oriented Software* [Addison-Wesley, 1994] by Erich Gamma, et al. describes the inheritance model as "white-box reuse." By the way, if there is any single book that should be read in regards to design patterns, this is the one. Although it's not specific to Delphi, it gives an in-depth rundown of patterns as they might be implemented in any language.

The inheritance model has its advantages. It allows reuse by letting descendant classes take from the functionality of the ancestor class. Descendant classes are used to create "specialized" versions of the ancestor. In *TUserConfiguration*, for example, each descendant would need to override only the methods required for saving and reading data to and from its specific store.

There are two problems with the inheritance model. The first is that the class implementation used by the client is decided upon at compile time. This isn't a major problem because it's possible to have the client refer to the ancestor rather than a specific descendant. For example, *TUserConfiguration* might have two abstract methods for saving and reading data that descendants would need to override. Clients would code to *TUserConfiguration* directly. At some point, the client application would still have to instantiate a specialized version of the ancestor.

The second, and more serious, problem has to do with the exposure of the ancestor class to its descendants. Because methods and internal fields are typically exposed to descendant classes, it's difficult to change the implementation of the parent class without forcing modifications to be made to its descendants. This is typical in cases where the descendant class makes direct references to methods and fields of the parent class' private/protected members.

By using the composition model, the specialized functionality is delegated to an entirely separate class. As with the *TUserConfiguration* class, I've delegated the saving and retrieving of data to the *TStorage* class. *TUserConfiguration* refers to the interface defined for *TStorage*. I use the term "interface" loosely here. This isn't the same as an interface when dealing with COM; however, it's similar. This model

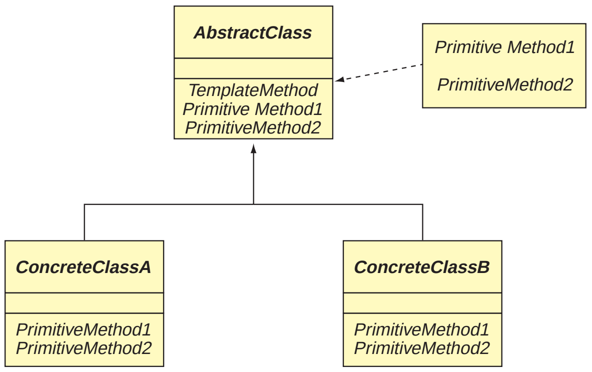


Figure 4: *TUserConfiguration* based on the inheritance model.

more accurately incorporates reusability. Objects deal with each other at the interface level. As long as you never change the interface, objects are self-contained, and the dependencies between classes are minimized. It becomes much easier to change the implementation of classes without necessitating change to other classes used in the system. Additionally, client applications don't need to know anything about the *TStorage* class. It just cares that *TUserConfiguration* can save and read data from a specified store.

When using a composition model, you typically code to interfaces. This requires that you design your classes cautiously, because the interface of a class serves as a "contract" between it and other classes. In the next installment in this series, I'll use this model to design an application framework. You'll be able to add modules to a shell application without having to recompile it.

Conclusion

We've discussed a practical use for the Singleton pattern by illustrating a global user configuration object. We'll be discussing more patterns and presenting real-world uses for them. Programming patterns can consistently save you time; rather than trying to figure out how to solve a particular problem, the solution may already be at hand in a pattern. (I would like to thank Anne Pacheco and Steve Teixeira for proofing this and many of my articles.) ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\MAR\DI9903XP.

Xavier Pacheco is president and chief consultant for Xapware Technologies Inc., where he provides enterprise-level consulting services and training. He is also the co-author of *Delphi 4 Developer's Guide* [SAMS Publishing, 1998]. You can write Xavier at xavier@xapware.com, or visit his Web site at http://www.xapware.com.

Begin Listing One — Sngltn.pas

```
unit Sngltn;

interface

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

type
  // TSingleton class definition.
  TSingleton = class(TComponent)
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    // Class function entry point.
    class function Singleton: TSingleton;
    procedure ShowSingletonName;
  end;

// Function entry point.
function Singleton: TSingleton;

implementation

var
  // Private class variable.
  FSingleton: TSingleton = nil;
  // Boolean valid creation indicator.
  FExternalCreation: Boolean = True;

// TSingleton.
constructor TSingleton.Create(AOwner: TComponent);
begin
  // Test if object has already been created.
  if FSingleton <> nil then
    raise Exception.Create(
      'Singleton class already initialized.');
  // Test if constructor was called external to the
  // Singleton() function.
  if FExternalCreation then
    raise Exception.Create(
      'Call Singleton function to reference this class.');
  inherited Create(AOwner);
end;

destructor TSingleton.Destroy;
begin
  // Set the private variable to nil.
  FSingleton := nil;
  inherited Destroy;
end;

procedure TSingleton.ShowSingletonName;
begin
  ShowMessage(ClassName);
end;

class function TSingleton.Singleton: TSingleton;
begin
  Result := Sngltn.Singleton;
end;

function Singleton: TSingleton;
begin
  if FSingleton = nil then begin
    FExternalCreation := False;
    try
      FSingleton := TSingleton.Create(nil);
    finally
      FExternalCreation := True;
    end;
  end;
  Result := FSingleton;
end;
initialization

finalization
  // Free the global object, only if not freed previously.
  if FSingleton <> nil then
    FSingleton.Free;
end.
```

End Listing One

Begin Listing Two — UserCfg.pas

```
unit UserCfg;

interface

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

```
type
  // Forward declarations.
  TStorage = class;
  TMainScreenPos = class;
  TStorageType = (stConfigFile, stRegistry);

  // TUserConfiguration.
  TUserConfiguration = class(TComponent)
  private
    FStorage: TStorage;
    FUserName: string;
    FPassword: string;
    FUserID: Integer;
    FStorageType: TStorageType;
    FMainScreenPos: TMainScreenPos;
    procedure SetMainScreenPos(
      const Value: TMainScreenPos);
  protected
    procedure SetPassword(const Value: string);
    procedure SetUserName(const Value: string);
    procedure SetUserID(const Value: Integer);
    procedure SetStorageType(const Value: TStorageType);
    procedure CreateStorage; virtual;
  public
    constructor Create(AOwner: TComponent;
      const AStorageType: TStorageType);
    destructor Destroy; override;
    procedure SaveUserConf;
    procedure ReadUserConf;
    property StorageType: TStorageType
      read FStorageType write SetStorageType;
    class function UserConfiguration: TUserConfiguration;
  published
    property UserName: string
      read FUserName write SetUserName;
    property Password: string
      read FPassword write SetPassword;
    property UserID: Integer read FUserID write SetUserID;
    property MainScreenPos: TMainScreenPos
      read FMainScreenPos write SetMainScreenPos;
  end;

  // Base storage class.
  TStorage = class(TObject)
  private
    procedure SaveUserConf(AUserConfiguration:
      TUserConfiguration); virtual; abstract;
    function ReadUserConf(AUserConfiguration:
      TUserConfiguration): Boolean; virtual; abstract;
  end;

  // Configuration file storage.
  TStorageCfg = class(TStorage)
    procedure SaveUserConf(AUserConfiguration:
      TUserConfiguration); override;
    function ReadUserConf(AUserConfiguration:
      TUserConfiguration): Boolean; override;
  end;

  // Configuration file storage.
  TStorageReg = class(TStorage)
    FRegKey: string;
    constructor Create;
    procedure SaveUserConf(AUserConfiguration:
      TUserConfiguration); override;
    function ReadUserConf(AUserConfiguration:
      TUserConfiguration): Boolean; override;
  end;

  // TMainScreenPos.
  TMainScreenPos = class(TPersistent)
  private
    FHeight: Integer;
    FTop: Integer;
    FLeft: Integer;
    FWidth: Integer;
  protected
    procedure SetHeight(const Value: Integer);
    procedure SetLeft(const Value: Integer);
    procedure SetTop(const Value: Integer);
    procedure SetWidth(const Value: Integer);
  published
    property Left: Integer read FLeft write SetLeft;
    property Top: Integer read FTop write SetTop;
    property Width: Integer read FWidth write SetWidth;
    property Height: Integer read FHeight write SetHeight;
  end;

function UserConfiguration: TUserConfiguration;

implementation

uses
  XWRegUtils;

var
  FUserConfiguration: TUserConfiguration = nil;
  FExternalCreation: Boolean = True;

function RemoveExt(const AFileName: string): string;
begin
  Result := Copy(AFileName, 1, Pos('.', AFileName)-1);
end;

function UserConfiguration: TUserConfiguration;
begin
  if FUserConfiguration = nil then begin
    FExternalCreation := False;
    try
      FUserConfiguration :=
        TUserConfiguration.Create(nil, stConfigFile);
    finally
      FExternalCreation := True;
    end;
  end;
  Result := FUserConfiguration;
end;

// TUserConfiguration.
constructor TUserConfiguration.Create(AOwner: TComponent;
  const AStorageType: TStorageType);
begin
  if FUserConfiguration <> nil  then
    raise Exception.Create(
      'UserConfiguration already initialized.');
  if FExternalCreation then
    raise Exception.Create(
      'Call UserConfiguration to reference this class.');
  inherited Create(AOwner);
  FMainScreenPos := TMainScreenPos.Create;
  FStorageType := AStorageType;
  CreateStorage;
  ReadUserConf;
end;

destructor TUserConfiguration.Destroy;
begin
  // Save the setting to storage.
  SaveUserConf;
  FStorage.Free;
  FMainScreenPos.Free;
  FUserConfiguration := nil;
  inherited Destroy;
end;

procedure TUserConfiguration.CreateStorage;
begin
  if FStorage <> nil then
    FStorage.Free;
  case FStorageType of
    stConfigFile: FStorage := TStorageCfg.Create;
    stRegistry:   FStorage := TStorageReg.Create;
  end;
```

```
end;

procedure TUserConfiguration.ReadUserConf;
begin
  if not FStorage.ReadUserConf(self) then begin
    FUserName := EmptyStr;
    FPassword := EmptyStr;
  end;
end;

procedure TUserConfiguration.SaveUserConf;
begin
  FStorage.SaveUserConf(self);
end;

procedure TUserConfiguration.SetMainScreenPos(
  const Value: TMainScreenPos);
begin
  FMainScreenPos := Value;
end;

procedure TUserConfiguration.SetPassword(
  const Value: string);
begin
  FPassword := Value;
end;

procedure TUserConfiguration.SetStorageType(
  const Value: TStorageType);
begin
  FStorageType := Value;
  CreateStorage;
  ReadUserConf;  // If the storage exists, read it.
end;

procedure TUserConfiguration.SetUserID(
  const Value: Integer);
begin
  FUserID := Value;
end;

procedure TUserConfiguration.SetUserName(
  const Value: string);
begin
  FUserName := Value;
end;

class function TUserConfiguration.UserConfiguration:
  TUserConfiguration;
begin
  Result := UserCfg.UserConfiguration;
end;

// TStorageCfg.
function TStorageCfg.ReadUserConf(AUserConfiguration:
  TUserConfiguration): Boolean;
var
  FileStream: TFileStream;
  FName: string;
begin
  Result := False;
  FName := ChangeFileExt(Application.ExeName, '.dat');
  if FileExists(FName) then begin
    FileStream := TFileStream.Create(FName, fmOpenRead);
    try
      FileStream.ReadComponent(AUserConfiguration);
      Result := True;
    finally
      FileStream.Free;
    end;
  end;
end;

procedure TStorageCfg.SaveUserConf(AUserConfiguration:
  TUserConfiguration);
var
  FileStream: TFileStream;
  FName: string;
begin
  FName := ChangeFileExt(Application.ExeName, '.dat');
  FileStream := TFileStream.Create(FName, fmCreate);
  try
    FileStream.WriteComponent(AUserConfiguration);
  finally
    FileStream.Free;
  end;
end;

// TStorageReg.
constructor TStorageReg.Create;
begin
  inherited;
  FRegKey := 'Software\' +
    RemoveExt(ExtractFileName(Application.ExeName));
end;

function TStorageReg.ReadUserConf(
  AUserConfiguration: TUserConfiguration): Boolean;
begin
  RegToComponentProps(FRegKey, AUserConfiguration);
  Result := True;
end;

procedure TStorageReg.SaveUserConf(
  AUserConfiguration: TUserConfiguration);
begin
  ComponentPropsToReg(AUserConfiguration, FRegKey);
end;

// TMainScreenPos.
procedure TMainScreenPos.SetHeight(const Value: Integer);
begin
  FHeight := Value;
end;

procedure TMainScreenPos.SetLeft(const Value: Integer);
begin
  FLeft := Value;
end;

procedure TMainScreenPos.SetTop(const Value: Integer);
begin
  FTop := Value;
end;

procedure TMainScreenPos.SetWidth(const Value: Integer);
begin
  FWidth := Value;
end;

initialization

finalization
  // Free the global object, only if not freed previously.
  if FUserConfiguration <> nil then
    FUserConfiguration.Free;
end.
```

End Listing Two

Begin Listing Three — XWRegUtils.pas

```
unit XWRegUtils;

interface

uses Classes;

// Saves the component specified by AComponent and its
// properties to the system registry in the location
// specified by AregKey.
procedure ComponentPropsToReg(AComponent: TComponent;
  const ARegKey: string);
```

```
// Reads the component properties for the component
// specified by AComponent from the system registry at the
// location specified by AregKey.
procedure RegToComponentProps(const ARegKey: string;
  AComponent: TComponent);

implementation

uses TypInfo, Registry, SysUtils;

procedure ProcessProps(AObject: TObject;
  const ARegSection: string; AReadProps: Boolean;
  ARegIni: TRegIniFile);
var
  PropList: PPropList;
  TypeData: PTypeData;
  i: Integer;
  PropName: string;
  TempObject: TObject;
begin
  TypeData := GetTypeData(AObject.ClassInfo);

  if TypeData.PropCount <> 0 then begin
    GetMem(PropList, SizeOf(PPropInfo)*TypeData.PropCount);
    try
      GetPropInfos(AObject.ClassInfo, PropList);
      for i := 0 to TypeData.PropCount - 1 do begin
       PropName := PropList[i]^.Name;
       // Filter out published properties of TComponent.
       if not ((PropName = 'Name') or
               (PropName = 'Tag')) then
         begin
           // For now, only process integers, strings, and
           // other objects.
           if AReadProps then
             case PropList[i]^.PropType^.Kind of
               tkInteger:
                 SetOrdProp(AObject, PropList[i],
                   ARegIni.ReadInteger(ARegSection,
                                       PropName, 0));
               tkString, tkLString:
                 SetStrProp(AObject, PropList[i],
                   ARegIni.ReadString(ARegSection,
                                      PropName, EmptyStr));
               tkClass: begin
                 TempObject := TObject(GetOrdProp(
                                AObject, PropList[i]));
                 ProcessProps(TempObject, PropName,
                              True, ARegIni);
               end;
             end
           else
             case PropList[i]^.PropType^.Kind of
               tkInteger:
                 ARegIni.WriteInteger(ARegSection,
                   PropName, GetOrdProp(AObject,
                                        PropList[i]));
               tkString, tkLString:
                 ARegIni.WriteString(ARegSection,PropName,
                   GetStrProp(AObject, PropList[i]));
               tkClass: begin
                 TempObject := TObject(GetOrdProp(AObject,
                   PropList[i]));
                 ProcessProps(TempObject, PropName, False,
                              ARegIni);
               end;
             end
         end;
      end;
    finally
      FreeMem(PropList,
              SizeOf(PPropInfo) * TypeData.PropCount);
    end;
  end;
end;

procedure ComponentPropsToReg(AComponent: TComponent;
  const ARegKey: string);
var
  RegIni: TRegIniFile;
begin
  RegIni := TRegIniFile.Create(ARegKey);
  try
    ProcessProps(AComponent, EmptyStr, False, RegIni);
  finally
    RegIni.Free;
  end;
end;

procedure RegToComponentProps(const ARegKey: string;
  AComponent: TComponent);
var
  RegIni: TRegIniFile;
begin
  RegIni := TRegIniFile.Create(ARegKey);
  try
    ProcessProps(AComponent, EmptyStr, True, RegIni);
  finally
    RegIni.Free;
  end;
end;

end.
```

End Listing Three

Begin Listing Four — MainFrm.pas

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    edtUserName: TEdit;
    edtPassword: TEdit;
    lblUserName: TLabel;
    lblPassword: TLabel;
    procedure FormClose(Sender: TObject;
      var Action: TCloseAction);
    procedure FormCreate(Sender: TObject);
    procedure edtUserNameChange(Sender: TObject);
    procedure edtPasswordChange(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

uses UserCfg;

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
begin
  with UserConfiguration do begin
    // Change the storage location to the system registry.
    StorageType := stRegistry;
    if not ((MainScreenPos.Width = 0) or
            (MainScreenPos.Height = 0)) then
      SetBounds(MainScreenPos.Left, MainScreenPos.Top,
                MainScreenPos.Width, MainScreenPos.Height);
    edtUserName.Text := UserName;
    edtPassword.Text := Password;
  end;
end;

procedure TMainForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  with UserConfiguration do begin
    MainScreenPos.Left := Left;
    MainScreenPos.Top := Top;
    MainScreenPos.Width := Width;
    MainScreenPos.Height := Height;
  end;
end;

procedure TMainForm.edtUserNameChange(Sender: TObject);
begin
  UserConfiguration.UserName := edtUserName.Text;
end;

procedure TMainForm.edtPasswordChange(Sender: TObject);
begin
  UserConfiguration.Password := edtPassword.Text;
end;

end.
```

End Listing Four

By Bill Todd

The Briefcase Model

When Your Application Must Travel Well

One of the benefits of using the MIDAS multi-tier application architecture is the ability to build briefcase-model applications. A briefcase-model application is a multi-tier application that allows the user of the client application to save a set of records in local files, disconnect from the network that hosts the application server and database, and edit the data off-line. Later, the user reconnects to the network, and applies any accumulated updates to the database.

There are two architectures that can form the basis for a briefcase-model application. The first is the classic multi-tier application, where the client application runs on one machine. The application server (the middle tier) runs on another machine on the network, and the database server runs on yet another machine. In this case, a MIDAS license is required. In a briefcase application, when the client application starts, it must determine if the application server is available. The easy way to see if the server is available is to set the *Connected* property of the connection component in the client application to True. If the server isn't available, an exception is raised.

One of the nice things about the MIDAS licensing requirements is that you can build briefcase-model applications without having to purchase a MIDAS license. As long as the client application and the application server that provides its data run on the same machine, no license is required.

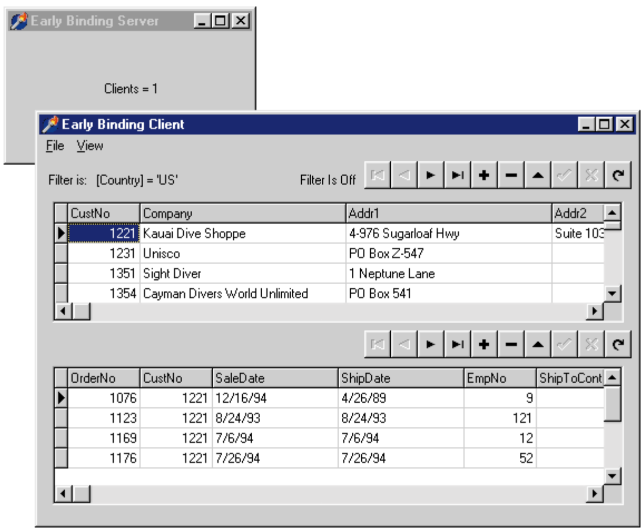
This architecture complicates a briefcase application because the application server is always available. Starting the client when not connected to the network will cause the application server to start, but the server won't be able to connect to the database.

There are two ways to handle this situation. The server can attempt to connect to the database, and, if it fails, can notify the client that the database isn't available. This is by far the more complex solution, because the client will have to call a custom method on the server to determine if the database is available, and, if it isn't, modify its behavior to work with local data, even though the application server is running.

A much simpler approach is to assume the user knows whether she or he is connected to the network, and provide two icons to start the client application. One icon is used when connected to the network, and the other when not connected. The icon that starts the client when the network isn't available can include a command-line parameter that tells the client to run in briefcase mode.

A Sample Application

The sample application accompanying this article (see Figure 1) demonstrates these techniques. (The client and server projects discussed in this article are available for download; see end of article for details.) To run the sample application, compile and run the EbSrvr server first, so it will register itself as an Automation server.



CustNo	Company	Addr1	Addr2
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
1231	Unisco	PO Box Z-547	
1351	Sight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	

OrderNo	CustNo	SaleDate	ShipDate	EmpNo	ShipToCont
1076	1221	12/16/94	4/26/89	9	
1123	1221	8/24/93	8/24/93	121	
1169	1221	7/6/94	7/6/94	12	
1176	1221	7/26/94	7/26/94	52	

Figure 1: The demonstration client and server application at run time.

```
...
{ Try to connect to the server. }
ConnectToServer;
with MainDm do begin
  { If the server is not available, and there is no local
    data to load, notify the user and terminate. }
  if (LoadLocalData = False) and
       (IsConnected = False) then begin
    MessageDlg('There is no data available.',
               mtInformation, [mbOK], 0);
    Application.Terminate;
    Exit;
  end;  // if
  { Open the client datasets. }
  CustomerCds.Open;
  OrderCds.Open;
...
```

Figure 2: Determining if the server is available.

Saving Data

The first requirement of a briefcase client application is the ability to save the data that will be used off-line in local files. The *SaveToFile* method of *TClientDataSet* provides this ability. The following code is from the **Save To Local Drive** menu choice's *OnClick* event handler:

```
procedure TEcMainForm.SaveLocally1Click(Sender: TObject);
begin
  with MainDm do begin
    CustomerCds.SaveToFile(CustomerFileName);
    OrderCds.SaveToFile(OrderFileName);
    AllOrdersCds.SaveToFile(AllOrderFileName);
  end;
end;
```

This code calls the *SaveToFile* method of each of the client dataset components in the program's data module. *SaveToFile*'s parameter is the name of the file to which to save the data. The file can have any name and file extension you choose. In this case, the file names are defined as global constants in the **implementation** section of the main form's unit, so they can be easily changed if necessary.

```
{ If the -L command-line parameter is not present, try to
  connect to the application server. If the connection
  succeeds, set the global variable IsConnected to True. }
procedure TEcMainForm.ConnectToServer;
begin
  { If the local command-line parameter is present,
    don't try to connect to the server. }
  if ParamCount > 0 then
    if UpperCase(ParamStr(1)) = '-L' then begin
      DisableServerFeatures;
      Exit;
    end; // if
  { Try to connect to the server. If the connection
    is established, set IsConnected to True; }
  with MainDm do
    try
      EbConn.Connected := True;
      IsConnected := True;
    except
      DisableServerFeatures;
      Exit;
    end; // try
  { If the server is available, see if the database is.
    If not, shut down the server. }
  with MainDm do
    if not EbConn.AppServer.IsDatabase then begin
      IsConnected := False;
      EbConn.Connected := False;
    end; // if
end;
```

Figure 3: The *ConnectToServer* method.

Calling *SaveToFile* saves both the *Data* and *Delta* properties of the *TClientDataSet*. That is, both the data and the changes that have been made — but not applied to the database — are saved. This allows you to edit off-line and resave the original data and unapplied changes as many times as you wish.

Starting the Client

When the client starts, it must determine if the server is available. This is handled in the main form's *OnCreate* event handler, part of which is shown in Figure 2. This code must deal with three possible conditions upon application startup:

1) Local data is available.
2) Local data is not available, but the application server is available.
3) Neither local nor server data is available.

If local data is available, it must be loaded. If the application server is available, a connection to it must be established. If no data was loaded from local files, data will be fetched from the server automatically when the client datasets are opened. If no data is available from local files or the server, the user must be warned, and the application terminated.

The code in the *OnCreate* event handler begins by calling the *ConnectToServer* method, shown in Figure 3. The main form's unit includes a global variable, *IsConnected*, which is initialized to False. The method then sets that variable to True if a connection can be established to the application server.

The *ConnectToServer* method begins by checking if any command-line parameters are present. If so, it checks if the first parameter is -L. If the -L parameter is present,

```
{ Try to load local data files. If they exist,
  return True; otherwise, return False; }
function TEcmainForm.LoadLocalData: Boolean;
begin
  Result := False;
  with MainDm do begin
    { If local files exist, load them. }
    if FileExists(CustomerFileName) then begin
      CustomerCds.LoadFromFile(CustomerFileName);
      Result := True;
    end; // if
    if FileExists(OrderFileName) then
      OrderCds.LoadFromFile(OrderFileName);
    if FileExists(AllOrderFileName) then
      AllOrdersCds.LoadFromFile(AllOrderFileName);
  end; // with
end;
```

Figure 4: The *LoadLocalData* method.

DisableServerFeatures is called, and this method exits, leaving *IsConnected* set to its default value of False. If the -L command-line parameter isn't present, the *TDCOMConnection* component's *Connected* property is set to True inside a **try..except** block. If the server can't be started, an exception will occur. If the connection is established, the *IsConnected* variable is set to True. If the connection fails, *IsConnected* retains its default value of False, and *DisableServerFeatures* is called.

If a connection to the application server is established, the application server's *IsDatabase* method is called. This method attempts to set the *Connected* property of the server's *TDatabase* component to True. If the connection succeeds, *IsDatabase* returns True; otherwise, it returns False. If the database isn't available, the application server is closed, and *IsConnected* is set back to False.

The *DisableServerFeatures* method:

```
{ Disable features that are only available when
  connected to the application server. }
procedure TEcMainForm.DisableServerFeatures;
begin
  View1.Enabled := False;
  ApplyChanges1.Enabled := False;
  FilterStringLabel.Visible := False;
  FilterLabel.Visible := False;
end;
```

takes care of disabling features of the client application that won't work if the application server isn't running. In this application, there are four features that must be disabled:

- The first is the **View** menu choice, which allows the client to set a filter on the server that displays all customers, or only customers in the US.
- The second is the **Apply Changes** choice on the **File** menu, which calls *ApplyUpdates* for each of the client datasets.
- The third is the *FilterStringLabel* component, which shows the filter expression on the server. While this expression could still be displayed (because it's included in the data packets), there's no way to know if the filter was enabled at the time the data was saved to the local files.

```
procedure TEcMainForm.ApplyChanges1Click(Sender: TObject);
begin
  with MainDm do begin
    with CustomerCds do begin
      { If there is an unposted record, post it. }
      if State in [dsEdit, dsInsert] then
        Post;
      { If there are changes, apply them. If the changes
        are applied successfully, refresh. }
      if ChangeCount > 0 then
        if ApplyUpdates(ChangeCount) = 0 then
          Refresh;
    end; // with CustomerCds
    with OrderCds do begin
      { If there is an unposted record, post it. }
      if State in [dsEdit, dsInsert] then
        Post;
      { If there are changes, apply them. If the changes
        are applied successfully, refresh. }
      if ChangeCount > 0 then
        if ApplyUpdates(ChangeCount) = 0 then
          Refresh;
    end; // with OrderCds
  end; // with MainDm
  { If local files exist, delete them now that the updates
    have been applied. }
  if FileExists(CustomerFileName) then
    DeleteFile(CustomerFileName);
  if FileExists(OrderFileName) then
    DeleteFile(OrderFileName);
  if FileExists(AllOrderFileName) then
    DeleteFile(AllOrderFileName);
end;
```

Figure 5: Applying updates and deleting the local files.

- Fourth is the *FilterLabel* component, which shows whether the filter on the server is currently enabled on the server. This is set by a callback from the server, and isn't available without the server.

The next block of code in the main form's *OnCreate* event handler attempts to load any data in local files by calling the *LoadLocalData* method, shown in Figure 4. This method returns True if local data is found. The method also checks if each file exists, and if so, calls the corresponding client dataset's *LoadFromFile* method to load the data. In this application, there is no way to save order data without saving customer data, so the return value is set to True if the local customer file is found.

Reconnecting

There is one more alteration to the client required in a briefcase-model application. When the user reconnects to the network after making off-line changes, and applies those changes to the database, the local files remain. The next time the client is started, it will reload the same local data and changes. To prevent this, the code for the **Apply Changes** menu choice is changed to delete the local files after updates are applied successfully (see Figure 5).

In Case They Forget ...

There's one more change that's not required (except perhaps to the client). It's possible the user could make changes offline and forget to save them before closing the program. To prevent this, save the files in the main form's *OnClose* event handler, as shown here:

```
procedure TEcMainForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  if not IsConnected then
    SaveLocally1Click(Self);
end;
```

Of course, there's more to any briefcase-model application. Please give the sample application a spin, and study it, before creating your own briefcase solutions. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\MAR\DI9903BT.

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 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.

By Kevin J. Bluck and James Holderness

Shell Notifications

Getting Windows to Share Some of Its Secrets

You may have stumbled across the Windows API function *SHChangeNotify*. According to the Windows docs, it notifies the system of any events that may affect the shell. Well, that's very nice for the shell. There are many interesting events the system monitors: file and directory changes, media insertion and removal, disk free-space updates, etc. For example, you're probably familiar with having Windows Explorer update the CD drive's icon when you insert or remove a CD. That's an instance of shell notifications at work. Very useful, but wouldn't it be nice if you could see those notifications as well?

Until now, tapping into this mechanism required insider knowledge. For some reason, Microsoft decided not to reveal the methods by which Windows Explorer receives these notifications. This article exposes those secrets, and provides a nifty component to give you a head start in using this exciting technique the Delphi way.

A Brief Digression

Before getting wrapped up in the details of signing up for shell notification, you need to know just a little bit about the *PItemIDList* record type. This is a type defined in the standard *ShlObj* unit, and refers to the construct known to shell programmers as a PIDL (pronounced "piddle"). The nature of PIDLs is a broad topic, more than enough to occupy its own article. For the purposes of shell notifications, however, a deep exposure isn't necessary.

Basically, a PIDL is the shell version of the DOS path. If you look at the folder tree in the left pane of Windows Explorer, you can see all the file system folders. You can also see folders that aren't part of the file system, such as Control Panel. Some means was necessary to identify all folders uniquely, whether they were part of the file system or not. Microsoft's solution was the PIDL, a sort of turbo-charged path. It's not a simple string; rather, it's a pointer to a chain of structures that contain identifying information for a given folder. The contents of a PIDL are largely opaque; they weren't intended for direct display or manipulation.

One tricky aspect of PIDLs is that they must often be allocated in one module, and freed in a module written by a different party. This can be problematic, as different development environments often use different memory allocation schemes. For example, using Delphi's *FreeMem* procedure to free memory originally allocated by some C compiler's *malloc* RTL function would most likely end up corrupting the heap. As a result, the memory buffers to contain PIDLs must be allocated and freed by the shell task allocator. This ensures the PIDL's memory will always be allocated and freed using the same scheme, regardless of the development environment used for the module. This functionality is implemented through a COM interface named *IMalloc*. Using anything but this global allocation engine to allocate and free PIDLs is a quick route to an abnormal termination. You can use the *IMalloc* interface directly for this, but we have opted instead to use some "cheater" functions, which you'll find defined in the *kbsnPIDL* unit in the sample files included with this article. (All source in this article is available for download; see end of article for details.)

The upshot of all this is that every file system object can be represented either as a PIDL or a path. In addition, many non-file system objects also exist that can't be identified by

Constants associated with single events	
SHCNE_ASSOCCHANGED	A file-type association has changed.
SHCNE_ATTRIBUTES	The attributes of an item or folder have changed.
SHCNE_CREATE	A non-folder item has been created.
SHCNE_DELETE	A non-folder item has been deleted.
SHCNE_DRIVEADD	A drive has been added.
SHCNE_DRIVEADDGUI	A drive has been added via the shell.
SHCNE_DRIVEREMOVED	A drive has been removed.
SHCNE_EXTENDED_EVENT	Not currently used.
SHCNE_FREESPACE	The amount of free space on a drive has changed.
SHCNE_MEDIAINSERTED	Storage media has been inserted into a drive.
SHCNE_MEDIAREMOVED	Storage media has been removed from a drive.
SHCNE_MKDIR	A folder has been created.
SHCNE_NETSHARE	A folder on the local computer is being shared via the network.
SHCNE_NETUNSHARE	A folder on the local computer is no longer being shared via the network.
SHCNE_RENAMEFOLDER	The name of a folder has changed.
SHCNE_RENAMEITEM	The name of a non-folder item has changed.
SHCNE_RMDIR	A folder has been removed.
SHCNE_SERVERDISCONNECT	The computer has disconnected from a server.
SHCNE_UPDATEDIR	The contents of an existing folder changed, but the folder wasn't renamed.
SHCNE_UPDATEIMAGE	An image from the system image list has changed.
SHCNE_UPDATEITEM	An existing non-folder item changed, but the item wasn't renamed.
Constants that combine multiple event types	
SHCNE_ALLEVENTS	Specifies a combination of all possible event identifiers.
SHCNE_DISKEVENTS	Specifies a combination of all of the disk event identifiers.
SHCNE_GLOBALEVENT	Specifies a combination of all of the global event identifiers.
Flag used with event constants	
SHCNE_INTERRUPT	The event occurred as a result of a system interrupt.

Figure 1: Shell event constants.

anything but a PIDL. Many shell functions, therefore, require PIDLs as parameters instead of traditional paths, or return PIDLs allocated from within the shell function that you may need to free later. For the purposes of this article, you may consider a PIDL to be a pointer — supplied by the shell — that points to arbitrary data that should not be modified in any way. Functions have been provided in unit *kbsnPIDL* that will convert a file system path to a PIDL, and vice versa. The only unusual aspect is that you should never free a PIDL with the usual VCL functions, such as *FreeMem*; you must use only the *FreePIDL* function provided in unit *kbsnPIDL*.

Getting "In the Loop"

The key to receiving shell change notifications is the *SHChangeNotifyRegister* function. Here's its prototype:

```
function SHChangeNotifyRegister(Window: HWND; Flags: DWORD;
  EventMask: ULONG; MessageID: UINT; ItemCount: DWORD;
  var Items: TNotifyRegister): THandle; stdcall;
```

It's used to register a window with the shell, which will then be notified of all subsequent *SHChangeNotify* events. It's exported from SHELL32.DLL. Like most undocumented functions, it's not exported by name. Therefore, it's necessary to link using the function ordinal. The export ordinal for *SHChangeNotifyRegister* is 2.

The *Window* parameter specifies the handle of the window that should receive the notification messages. This can be any window you desire, but it's usually best to create an invisible window whose only responsibility is handling the notification messages.

The *EventMask* parameter is a bit-mask of all the events you are interested in. You can use any combination of the *SHCNE_xxx* constants, the same ones that are used for the *SHChangeNotify* function, combined with a logical **or** operation. Figure 1 provides a complete listing of these constants.

The *Flags* parameter allows you to specify optional behavior for the notifications. You may filter out interrupt or non-interrupt events, and decide whether to use a proxy window under NT. See Figure 2 for a list of these flags. Typically, both interrupt and non-interrupt flags should be set, as you generally don't care about the ultimate source of the event. At any rate, interrupt events are extremely rare. The SHCNF_NO_PROXY flag allows you to handle the notification more efficiently on Windows NT, but it complicates the message handling procedure, and requires a couple more undocumented functions. We'll explain the whole situation with NT later.

The *MessageID* parameter is the identifier of the message that will be sent to that window. It's recommended you use a value derived from WM_USER for the value of *MessageID* to avoid

Flag	Value
SHCNF_ACCEPT_INTERRUPTS	$0001
SHCNF_ACCEPT_NON_INTERRUPTS	$0002
SHCNF_NO_PROXY	$8000

Figure 2: *SHChangeNotifyRegister* flags.

conflicts with system messages. If you use a single-purpose window as previously described, the value of WM_USER itself is fine. The details of the message handling are explained later.

The *ItemCount* parameter specifies the number of paths you wish to monitor. Usually, this will be 1, but it's possible to monitor many different paths via the *Items* parameter.

The *Items* parameter is a pointer to a record of type *TNotifyRegister*. The following is the definition of this record type:

```
TNotifyRegister = packed record
  pidlPath: PItemIDList;
  bWatchSubtree: BOOL;
end;
```

If the *pidlPath* data member of this record specifies a valid PIDL for a valid folder, you'll receive events that affect the folder itself, as well as any items in the folder. If you set the *bWatchSubtree* data member to True, you'll receive events for the entire sub-tree rooted at the specified folder, e.g. all folders and items below the specified folder in addition to the folder itself. If you set the *pidlPath* data member to **nil**, you'll receive events for every folder and item on the system. If the value of the *ItemCount* parameter is greater than 1, you must supply the same number of *TNotifyRegister* records to the *Items* parameter in the form of a vector, one record packed after another into a buffer large enough to hold all of them.

If the *SHChangeNotifyRegister* function succeeds, the return value is a handle of a shell change notification object. You should save this handle for later use. If the function fails, the return value is 0.

Shut It Off

When you're finished monitoring the notification events, you should pass the notification handle returned by *SHChangeNotifyRegister* to *SHChangeNotifyDeregister*. The export ordinal value of *SHChangeNotifyDeregister* is 4, and the function declaration is as follows:

```
function SHChangeNotifyDeregister(Notification: THandle):
  BOOL; stdcall;
```

The *Notification* parameter takes the handle of a shell change notification object returned by a successful call to *SHChangeNotifyRegister*. As you might guess, if the function succeeds, the return value is True; if it fails, the return value is False.

Getting the Message

After registering to receive shell notification messages, the shell will send its notification messages to the window you specified in the *Window* parameter of the *SHChangeNotifyRegister* function. You must then crack the relevant data out of the message to get useful information. Delphi doesn't define a special message record type for these messages, so simply use the standard *TMessage* type. This is a variant record type, but you should consider it to be defined as shown here:

```
TMessage = record
  Msg:    DWORD;
  WParam: DWORD;
  LParam: DWORD;
  Result: DWORD);
end;
```

The *Msg* data member of the message record will be set to whatever value you specified in the *MessageID* parameter you passed to the *SHChangeNotifyRegister* function. Use this parameter to recognize notification messages, as opposed to the myriad other miscellaneous messages the Windows system will send to your window.

The *LParam* data member will be set to the ID of the event that occurred. This will be one of the *SHCNE_xxx* values shown in Figure 1. It's possible that multiple event IDs could be **or**'ed together, so it would be best to test for the presence of a given ID value, using a logical **and** operation, rather than via an equality test using the = operator.

The *WParam* will be a pointer to a record of type *TTwoPIDLArray*. The name *WParam*, which is C-style shorthand for Word Parameter, is an anachronism. It's now actually a 32-bit long type, just like *LParam*, and so can store a pointer. This record points to an array containing the two PIDLs associated with the event, as shown here:

```
TTwoPIDLArray = packed record
  PIDL1: PItemIDList;
  PIDL2: PItemIDList;
end;
```

Those of you familiar with the *SHChangeNotify* function may be wondering why you only receive PIDLs from shell change notification messages, even though the *SHChangeNotify* function accepts *PChar* and DWORD data types, as well as PIDLs. The reason is that all the data types are automatically converted to PIDLs by the shell before being sent anywhere. For SHCNF_PATH and SHCNF_PRINTER types, this seems obvious enough. For the SHCNF_DWORD type, a 10-byte "fake" PIDL is created, with the two DWORD items immediately following the *cb* data member. Ignore the *cb* data member; it has no use in this situation except as a placeholder. This encoding scheme is encapsulated for your convenience by the *TDWORDItemID* record type, as shown here:

```
TDWORDItemID = packed record
  cb: Word;  { Ignore }
  dwItem1: DWORD;
  dwItem2: DWORD;
end;
```

The SHCNE_FREESPACE event is handled as a special case. When the drive is passed in as a path or a PIDL, it's converted to a DWORD contained by the above encoding scheme, with drives A: to Z: mapping onto bits 0 to 25. For example, drive D:

would map to bit 3, so the value of the *dwItem1* data member would have a value of 8. If there are two drives specified for the event, then two bits will be set in the DWORD. The value of the second DWORD in this case appears to be meaningless.

Windows NT and Memory Maps

This all seems simple enough, at least by the standards of Windows shell programming, but on Windows NT, there is a bit of a problem. NT maintains a careful separation of memory used by different processes. One process attempting to directly access memory owned by another process puts you in the express lane to your favorite exception, the General Protection Fault. You can't simply send a message to a window in a different process, and still expect the structure pointer contained in that message to be accessible.

To get around this, NT actually dumps all the relevant data into a memory-mapped file, which is accessible from any process, then sends the memory map handle and a process ID as the parameters to the message. To remain compatible with Windows 95, somebody obviously has to extract the information from that memory map on the other side. The way this works is that NT automatically creates a hidden "proxy" window whenever you call *SHChangeNotifyRegister*. It is the proxy window that receives the notification message containing the memory map. Its message handler then extracts all the information, and passes on the correct message with the expected data to your window.

Of course, this is not exactly efficient, which is where the SHCNF_NO_PROXY flag comes in. By specifying that flag when calling *SHChangeNotifyRegister*, you're telling NT to not create the proxy window, so the memory map handle gets passed directly to your window in the notification message. It's then up to you to extract the relevant information from the memory map. Fortunately, there are two functions that do all the work for you: *SHChangeNotification_Lock* and *SHChangeNotification_Unlock*.

The export ordinal value of *SHChangeNotification_Lock* is 644, and the function declaration is shown here:

```
function SHChangeNotification_Lock(MemoryMap: THandle;
  ProcessID: DWORD; var PIDLs: PTwoPIDLArray;
  var EventID: ULONG): THandle; stdcall;
```

The *MemoryMap* parameter is a handle to a chunk of memory allocated by the NT system. This handle will be contained in the *WParam* data member of the shell notification message.

The *ProcessID* parameter takes the ID of the process that generated the memory map. This value is contained in the *LParam* data member of the shell notification message.

The *PIDLs* parameter is output-only, and takes a variable of type *pointer* to a record of type *TTwoPIDLArray*. You may initialize the *pointer* variable to **nil** before calling the function. Do not allocate an actual *TTwoPIDLArray* record. When the function returns, this pointer will point to a *TTwoPIDLArray* record that contains the two PIDLs for the notification message, which are normally passed via the *WParam* data member of the message. Don't try to free this pointer directly, either.

The *EventID* parameter is also output-only, and takes a variable of type ULONG, or *Longint*, if you prefer. You may initialize this variable to 0 before calling the function. When the function returns, this variable will contain the *EventID* of the event for this message, which is normally passed by the *LParam* data member of the message.

The return value is a handle to the memory map, which you should save; you'll need it later. If by some strange circumstance the function should fail, it returns a 0.

When you have finished working with the data extracted via *SHChangeNotification_Lock*, you should unlock the memory map so NT can properly dispose of it. This is the purpose of the *SHChangeNotification_Unlock* function. The export ordinal value of *SHChangeNotification_Unlock* is 645, and the function declaration is as follows:

```
function SHChangeNotification_Unlock(Lock: THandle):
  BOOL; stdcall;
```

The *Lock* parameter is the handle you obtained from the call to the *SHChangeNotification_Lock* function. The function returns True if successful, and False on failure.

It's important to note that these functions exist only in Windows NT. If you attempt to link to them while running Windows 95, you'll experience a link failure. Therefore, it's impossible to use the Delphi external method for linking, unless you are completely sure your program will never run on anything but NT. You should use dynamic linking instead by calling the *GetProcAddress* Windows API function after testing which operating system is running. See Figure 3 for an example of using these NT mapping functions.

The Origin of Events

So, now you know how to receive all these shell notifications that are floating around, but who is actually generating them? According to the Windows documentation, "An application should use this function (*SHChangeNotify*) if it performs an action that may affect the shell." That seems to be a bit of wishful thinking. We can't imagine there are many application developers who really give a hoot whether the shell is kept informed of their actions.

Fortunately, the shell seems to generate most of the notifications itself. Sometimes, it may be directly responsible for an event, in which case, it's easy enough for it to make the call to *SHChangeNotify*. However, for things likely to originate in another application, such as file creation, it would presumably have to be monitoring the system somehow to generate the event. The result is that these notifications can be a bit unreliable, and often, there is a noticeable delay between the event and the notification. Also, the shell has only a 10-item event buffer,

```
var
  PIDLs: PTwoPIDLArray;
  EventId: DWORD;
  Lock: THandle;
begin
  // If NT, use the memory map to access the PIDL data.
  if (SysUtils.Win32Platform = VER_PLATFORM_WIN32_NT) then
    begin
      Lock := SHChangeNotification_Lock(THandle(
        TheMessage.wParam), DWORD(TheMessage.lParam),
        PIDLs, EventId);
      if (Lock <> 0) then
        try
          ProcessEvent(EventId, PIDLs);
        finally
          SHChangeNotification_Unlock(Lock);
        end;
    end
  else
    // If this isn't NT, access the PIDL data directly.
    begin
      EventId := DWORD(TheMessage.lParam);
      PIDLs := PTwoPIDLArray(TheMessage.wParam);
      ProcessEvent(EventId , PIDLs);
    end;
end;
```

Figure 3: Example of using *SHChangeNotification_Lock*.

and may decide to consolidate a number of events with a generic SHCNE_UPDATEDIR in case of an overflow.

In short, don't depend on these notifications for mission-critical applications.

Don't Believe Everything You Read

Another problem is that the Windows documentation isn't always completely accurate in its descriptions of the various events. Following are variations from the documentation we've observed after actual implementation.

An SHCNE_ATTRIBUTES is supposed to happen when "the attributes of an item or folder have changed." However, we have only witnessed an SHCNE_ATTRIBUTES event occurring when the printer status changed. Changing file and folder attributes produces an SHCNE_UPDATEITEM event instead.

SHCNE_NETSHARE and SHCNE_NETUNSHARE are supposed to occur when you share or unshare a folder. However, on Windows NT, the SHCNE_NETUNSHARE event never occurs. You get a SHCNE_NETSHARE event on both occasions. On Windows 95, they appear to work as advertised.

An SHCNE_UPDATEIMAGE event is claimed to signify that an image in the system image list has changed. However, images in the system image list should never change. What the event really means is that something that was using that particular icon index in the system image list is now using something else. Typical uses of SHCNE_UPDATEIMAGE include the Recycle Bin changing between empty and full, and the icon associated with a CD drive when a CD is inserted or removed. Document icons, which change as a result of changing a file-type association, do not generate an SHCNE_UPDATEIMAGE. They will produce an SHCNE_ASSOCCHANGED event instead.

SHCNE_MEDIAINSERTED and SHCNE_MEDIAREMOVED aren't generated in response to inserting or removing standard floppy diskettes. The disk drive hardware apparently doesn't support this information.

If you're deleting files into a Recycle Bin, you won't get an SHCNE_DELETE method, as you might expect. You'll actually get a SHCNE_RENAMEITEM. The SHCNE_DELETE comes only after you empty the Recycle Bin. This makes sense if you think about it, because you're actually moving the file from its old path to the Recycle Bin's path, but it might not be completely intuitive at first.

Some events can fire multiple times. This seems to apply to most file events. For example, if you delete a file, you'll probably get notified twice with identical messages. Be prepared for that.

If you want to know more, it's probably best that *you* test the items your particular application will be using, on as many platforms as possible.

The Delphi Way

So much for the messy details demanded by the Windows API. Let's design a component that will hide all this minutiae from those Delphi developers who have better things to worry about.

First, we'll define the component's public interface. There are two main issues with this component: where to watch, and what to watch for. The API provides the capability to filter a few criteria: event type, interrupt or non-interrupt, the folder to watch for events, and whether that folder's sub-folders should also be watched. Naturally, we also need some means of turning this thing on or off.

Identifying the folders to watch is the only unusual aspect of all this. As we mentioned previously, not all folders can be identified by file system paths. Using PIDLs to identify these folders is impractical for the design-time property editor, however, because PIDLs are opaque data types and can't be manually edited by a developer.

The solution is to use two properties. One property is a new enumerated type, *TkbSpecialLocation*, which encapsulates the list of Windows API constants that correspond to various "special" folders, e.g. Control Panel. These constants can be used with the *SHGetSpecialFolderLocation* API function to obtain a PIDL to that folder. By setting a property to one of these enumerated values, special locations can be selected without requiring the developer to type in the data for the PIDL. One of the values of *TkbSpecialLocation* is *kbslPath*. Setting this value will enable a second property to allow the developer to enter a specific file system path to monitor. Here's the final list of published properties:

```
property Active:           Boolean
property HandledEvents:    TkbShellNotifyEventTypes
property InterruptOptions: TkbInterruptOptions
property RootFolder:       TkbSpecialLocation
property RootPath:         TFileName
property WatchChildren:    Boolean;
```

```
TkbShellNotifySimpleEvent = procedure(Sender: TObject;
  IsInterrupt: Boolean) of object;

TkbShellNotifyIndexEvent = procedure(Sender: TObject;
  Index: LongInt; IsInterrupt: Boolean) of object;

TkbShellNotifyGeneralEvent = procedure(Sender: TObject;
  PIDL: Pointer; Path: TFileName;
  IsInterrupt: Boolean) of object;

TkbShellNotifyRenameEvent = procedure(Sender: TObject;
  OldPIDL: Pointer; OldPath: TFileName; NewPIDL: Pointer;
  NewPath: TFileName; IsInterrupt: Boolean) of object;

TkbShellNotifyGenericEvent = procedure(Sender: TObject;
  EventType: TkbShellNotifyEventType; PIDL1: Pointer;
  PIDL2: Pointer; IsInterrupt: Boolean) of object;
```

Figure 4: Event procedure type definitions.

```
property OnAnyEvent:           TkbShellNotifyGenericEvent
property OnDiskEvent:          TkbShellNotifyGenericEvent
property OnGlobalEvent:        TkbShellNotifyGenericEvent
property OnAssociationChanged: TkbShellNotifySimpleEvent
property OnAttributesChanged:  TkbShellNotifyGeneralEvent
property OnDriveAdded:         TkbShellNotifyGeneralEvent
property OnDriveRemoved:       TkbShellNotifyGeneralEvent
property OnExtendedEvent:      TkbShellNotifyGenericEvent
property OnFolderCreated:      TkbShellNotifyGeneralEvent
property OnFolderDeleted:      TkbShellNotifyGeneralEvent
property OnFolderRenamed:      TkbShellNotifyRenameEvent
property OnFolderUpdated:      TkbShellNotifyGeneralEvent
property OnFreespaceChanged:   TkbShellNotifyGeneralEvent
property OnImageUpdated:       TkbShellNotifyIndexEvent
property OnItemCreated:        TkbShellNotifyGeneralEvent
property OnItemDeleted:        TkbShellNotifyGeneralEvent
property OnItemRenamed:        TkbShellNotifyRenameEvent
property OnItemUpdated:        TkbShellNotifyGeneralEvent
property OnMediaInserted:      TkbShellNotifyGeneralEvent
property OnMediaRemoved:       TkbShellNotifyGeneralEvent
property OnNetworkDriveAdded:  TkbShellNotifyGeneralEvent
property OnResourceShared:     TkbShellNotifyGeneralEvent
property OnResourceUnshared:   TkbShellNotifyGeneralEvent
property OnServerDisconnected: TkbShellNotifyGeneralEvent
```

Figure 5: Design-time published events.

For those developers who like to get down and dirty, we'll surface a couple of run-time properties that allow them to meddle at the API level. There are really only two bits of information we can provide: the notification handle, and the actual root PIDL. These properties are:

```
property Handle: THandle
property RootPIDL: PItemIDList
```

Because this component encapsulates a notification mechanism, it should be clear that events are at its heart. We'll want to pre-crack the event data for the developer's convenience, of course. Every event will include the *Sender* parameter, as usual, and a *Boolean* value identifying whether the event was generated by an event. Some review of the API documentation reveals that we have five basic patterns of "data" parameters:

1) No extra parameters.
2) One DWORD.
3) One non-**nil** PIDL, which might represent a path.
4) Two non-**nil** PIDLs, which might represent paths.
5) "Generic" events that have two raw PIDLs, either of which might be **nil**.

The definitions of the procedural types representing these five event categories are shown in Figure 4. You may notice that PIDL parameters are represented as *Pointer* types, rather than *PItemIDList* types. This is because *PItemIDList* is defined in the *ShlObj* unit, which is not added automatically to form units when the component is dropped. This has the annoying quality of causing compiler errors when an event is assigned, unless unit *ShlObj* is manually added to the form's **interface** part **uses** clause.

Deciding what the events will be was fairly simple. There should be a component event for each possible shell event. In addition, we'll define events to encapsulate the three "collective" events defined by the Windows API (for those hardy souls who like working with raw Windows data). The list of events and their definitions are found in Figure 5.

The last area of the public interface to consider, the run-time methods, lends a few candidates for consideration. It's handy to have auxiliary methods for setting the *Active* property on and off. Also, virtually any component that handles system data needs some sort of reset capability. These methods are shown here:

```
procedure Activate;
procedure Deactivate;
procedure Reset;
```

There are, of course, many details to implementing a component that go beyond the core functions that the component encapsulates. As there are many other excellent references that cover the details of implementing custom components in Delphi, this article will not cover them. Instead, it will concentrate only on those pieces of the component's implementation that directly relate to the specific problem of shell notifications.

The implementation of shell notifications revolves around the calls to *SHChangeNotifyRegister* and *SHChangeNotifyDeregister*. Everything we do in this component will be in support of those function calls. Let's outline how the public properties and methods relate to those functions.

The property most directly linked to the calls is *Active*. Setting this property to True will cause *SHChangeNotifyRegister* to be invoked with parameters governed by the other four published properties. As you might guess, setting it to False will cause *SHChangeNotifyDeregister* to terminate the notifications.

The mechanics of calling the API functions are delegated to two private methods, *StartWatching* and *StopWatching*, which will be discussed later. Meanwhile, here's a code snippet from the private property writer method, *SetActive*, which illustrates the logic at work:

```
// Do nothing if the new value is the same as the old.
if (NewValue <> Self.FActive) then
  { If we're activating, start watching. }
  if (NewValue) then
    Self.StartWatching;
  else  { If we're deactivating, stop watching. }
    Self.StopWatching;
```

The other four properties (besides *Active*) can substantially change the notification model, if they are modified. There is no way to update these "on the fly" when *Active* is True, so it's necessary to "reset" the shell notification if these properties are changed while *Active* is True. This is the purpose of the *Reset* method. It simply calls the private methods *StopWatching* and *StartWatching*, if the component is *Active*. This has the effect of stopping the notifications with *SHChangeNotifyDeregister*, and calling *SHChangeNotifyRegister* with the current values of the component's properties. The property writer methods for these four properties call the *Reset* method after updating the component's internal data member corresponding to that property.

A Window All Our Own

Next, we consider how to manage the shell notification messages, which will result from the call to *SHChangeNotifyRegister*. A window must be available to process all the notification messages generated by the shell. How best to provide this window? We could use the application's main form, but that would require hooking that form's window procedure, a messy undertaking at best. It seems simplest to generate our own invisible window, whose sole purpose is to handle those notification messages, and over whose destiny we have absolute control. This step eliminates problems with conflicting messages and clashing hooks.

The Delphi VCL thoughtfully provides a couple of functions to facilitate this scheme. They are *AllocateHWnd* and *DeallocateHWnd*, found in the Forms unit. *AllocateHWnd*'s entire purpose in life is to generate a handle to an invisible window, using a window message-handling procedure you provide. Exactly what we need! Now, in the component's constructor, we can get and store a handle to a window that has nothing better to do than manage our icon's messages. Here's a call to this handy method:

```
{ Allocate a message-handling window. }
Self.FMessageWindow := AllocateHWnd(Self.HandleMessage);
```

As you can see, the call is trivial. What's important is the message-handling procedure we passed. This is where the notification messages from the shell are received, and where we have the opportunity to dispatch them. *AllocateHWnd* takes a single argument of *TWndMethod*, a class-member procedure that takes a single argument of type *TMessage*. It's up to us to provide that procedure. Figure 6 shows our component's message-handling procedure, to give you the idea.

```
procedure TkbShellNotify.HandleMessage(
  var TheMessage: TMessage);
var
  PIDLs:   PTwoPIDLArray;
  EventId: DWORD;
  Lock:    THandle;
begin
  { Handle only the WM_SHELLNOTIFY message. }
  if (TheMessage.Msg = WM_SHELLNOTIFY) then
    begin
      { If this is NT, use the memory map to access the
        PIDL data. }
      if SysUtils.Win32Platform=VER_PLATFORM_WIN32_NT then
        begin
          Lock := SHChangeNotification_Lock(THandle(
            TheMessage.wParam), DWORD(TheMessage.lParam),
            PIDLs, EventId);
          if (Lock <> 0) then
            try
              Self.ProcessEvent(EventId, PIDLs);
            finally
              SHChangeNotification_Unlock(Lock);
            end;
        end
      { If this is not NT, access the PIDL data directly. }
      else
        begin
          EventId := DWORD(TheMessage.lParam);
          PIDLs   := PTwoPIDLArray(TheMessage.wParam);
          Self.ProcessEvent(EventID, PIDLs);
        end;
    end  { if }
  { Call the default Windows procedure for any other message. }
  else
    TheMessage.Result := DefWindowProc(Self.FMessageWindow,
      TheMessage.Msg,TheMessage.wParam,TheMessage.lParam);
end;
```

Figure 6: An example message-handling method.

As we discussed earlier, we first check the message identifier to verify that this incoming message is a WM_SHELLNOTIFY message. We ignore all others, and send them to default handling, because none of the miscellaneous messages typically broadcast to every window in the system interest us. WM_SHELLNOTIFY, if you were wondering, is a constant we define ourselves, not one provided by Windows. Setting it equal to WM_USER is the easiest thing to do, and perfectly safe because we use this window only for handling shell notification messages.

Once we're satisfied this is indeed a notification message, we decode the *LParam* and *WParam* values to determine which event the message is relating to us, and the associated PIDL data. Notice how we extract the data using the *SHChangeNotification_Lock* API call if the component is running on NT. You'll also see that the actual detailed handling of the message is delegated to another private method, *ProcessEvent*, to avoid repeating that rather substantial bit of code.

The private method *ProcessEvent* (see Figure 7) is where we crack the PIDLs out of the *TTwoPIDLArray* record, convert them to file-system paths if possible, and dispatch them to the appropriate event handler. The centerpiece of this method is a **case** statement that calls the event-handling method corresponding to the event type. These event-

```
procedure TkbShellNotify.ProcessEvent(EventID: DWORD;
  PIDLs: PTwoPIDLArray);
var
  EventType:   TkbShellNotifyEventType;
  PIDL1:       PItemIDList;
  PIDL2:       PItemIDList;
  Path1:       TFileName;
  Path2:       TFileName;
  IsInterrupt: Boolean;
begin
  { Crack open the Two-PIDL array. }
  PIDL1 := PIDLs.PIDL1;
  PIDL2 := PIDLs.PIDL2;
  { Try to convert PIDLs to Paths. }
  Path1 := GetPathFromPIDL(PIDL1);
  Path2 := GetPathFromPIDL(PIDL2);
  { Determine if event is interrupt-caused. }
  IsInterrupt := Boolean(EventID and SHCNE_INTERRUPT);
  { Iterate through possible events and fire as appropriate.
    This is necessary because event IDs are flags, and
    there may be more than one in a particular message. }
  for EventType := Low(TkbShellNotifyEventType) to
                   High(TkbShellNotifyEventType) do begin
    { Skip the "multi" event types.
      They will be fired as needed below. }
    if (EventType in [kbsnAnyEvent, kbsnDiskEvent,
                      kbsnGlobalEvent]) then
      Continue;
    { If the current event type is flagged... }
    if ((ShellNotifyEnumToConst(EventType) and
         EventID) <> 0) then begin
      { Fire appropriate "multi" events for this event. }
      Self.AnyEvent(EventType, PIDL1, PIDL2, IsInterrupt);
      if ((ShellNotifyEnumToConst(kbsnGlobalEvent) and
           ShellNotifyEnumToConst(EventType)) <> 0) then
        Self.GlobalEvent(EventType, PIDL1,
                         PIDL2, IsInterrupt);
      if ((ShellNotifyEnumToConst(kbsnDiskEvent) and
           ShellNotifyEnumToConst(EventType)) <> 0) then
        Self.DiskEvent(EventType,PIDL1,PIDL2,IsInterrupt);
      { Fire specific event. }
      case (EventType) of
        kbsnAssociationChanged:
          Self.AssociationChanged(IsInterrupt);
        { Other event-handling methods with
          appropriate parameters... }
        kbsnServerDisconnected:
          Self.ServerDisconnected(PIDL1,Path1,IsInterrupt);
      end; { case }
    end; { if }
  end; { for }
end;
```

Figure 7: An abbreviated event-processing method.

handling methods are necessary because events are often left unassigned by the developer, and calling these **nil** handlers directly would cause exception faults. There is one such method for each published event that accepts the raw PIDLs and paths, performs any additional processing that may be required to extract useful information, and calls the event handler if it has been assigned.

The other wrinkle is the **for** loop, which iterates through the possible event types, comparing each to the *EventID* parameter using a logical **and** comparison to see if the matching event handler should be fired. This is necessary because the *EventID* parameter is actually a bitmap of flags, and, as such, might include more than one event flag. This prohibits a simple equality comparison using the = operator.

```
procedure TkbShellNotify.StartWatching;
var
  NotifyPathData: TNotifyRegister;
  Flags:          DWORD;
  EventType:      TkbShellNotifyEventType;
  EventMask:      DWORD;
begin
  { Initialize Flags. }
  Flags := SHCNF_NO_PROXY;
  if kbioAcceptInterrupts in Self.InterruptOptions then
    Flags := Flags or SHCNF_ACCEPT_INTERRUPTS;
  if kbioAcceptNonInterrupts in Self.InterruptOptions then
    Flags := Flags or SHCNF_ACCEPT_NON_INTERRUPTS;
  { Initialize EventMask. }
  EventMask := 0;
  for EventType := Low(TkbShellNotifyEventType) to
                   High(TkbShellNotifyEventType) do
    if (EventType in Self.HandledEvents) then
      EventMask :=
        EventMask or ShellNotifyEnumToConst(EventType);
  { Initialize Notification Path data. }
  NotifyPathData.pidlPath      := Self.RootPIDL;
  NotifyPathData.bWatchSubtree := Self.WatchChildren;
  { Register for notification and store the handle. }
  Self.FHandle := SHChangeNotifyRegister(
    Self.FMessageWindow, Flags, EventMask, WM_SHELLNOTIFY,
    1, NotifyPathData);
  { If registration failed, set Active to False. }
  if (Self.Handle = 0) then
    Self.Deactivate;
end;
```

Figure 8: The *StartWatching* private method.

The Heart of the Matter

Now, with all these supporting tasks worked out, we can finally get to the heart of the matter — the long-awaited call to *SHChangeNotifyRegister*. It might seem a bit anticlimactic, but this function is found in only one place throughout the entire component. This place, of course, is the *StartWatching* private method. Let's work our way through it, as shown in Figure 8.

First, we initialize the flags. We'll always specify SHCNF_NO_PROXY, because we're prepared to handle the message correctly on NT. We'll also set the SHCNF_ACCEPT_INTERRUPTS and SHCNF_ACCEPT_NON_INTERRUPTS flags, as determined by the value of the *InterruptOptions* property.

Next, we initialize the *EventMask* parameter to specify the shell events we're interested in monitoring. This is accomplished by iterating through all possible values of the *TkbShellNotifyEventType* enumerated type, and setting the corresponding flag bits using a logical **or** operation each time we find a value, which is set in the *HandledEvents* property.

The last parameter to set up is the folder path information. We simply fill the necessary *TNotifyRegister* record with the values found in the *RootPIDL* and *WatchChildren* properties.

Finally, we can make the call to *SHChangeNotifyRegister* and save the handle returned by that function.

The *StopWatching* method is quite simple by comparison. It merely calls the *SHChangeNotifyDeregister* function, giving it the handle saved from the call to *SHChangeNotifyRegister*, and resets the internal data member containing the handle value to zero. Here's the entire method:

```
procedure TkbShellNotify.StopWatching;
begin
  { Deregister the notification handle
    and set the handle to nil. }
  SHChangeNotifyDeregister(Self.FHandle);
  Self.FHandle := 0;
end;
```

The result is a component that makes shell notifications almost trivial to access. Go ahead. Try it out with the sample event monitor provided with this article's companion source code. Feel free to forget all that Windows API complexity.

The Bottom Line

The Shell Notifications API gives you powerful insight into the internal workings of the Win32 shell. Your applications can use it to react to all sorts of important system events, giving you that extra edge over the competition. The *TkbShellNotify* component gives you this API in a convenient and simple package, making these services almost trivial to exploit. Now, get out there and write something amazing! Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\MAR\DI9903KB.

Kevin J. Bluck is an independent contractor specializing in Delphi development. He lives in Sacramento, CA with his lovely wife Natasha. He spends his spare time chasing weather balloons and rockets as a member of JP Aerospace (http://www.jpaerospace.com), a group striving to be the first amateur organization to send a rocket into space. Kevin can be reached via e-mail at kbluck@ix.netcom.com.

James Holderness is a software developer specializing in C/C++ Windows applications. He also runs a Web site on undocumented functions in Windows 95 (http://www.geocities.com/SiliconValley/4942). He is currently working for FerretSoft LLC (http://www.ferretsoft.com), where he helps create the Ferret line of Internet search tools. James can be reached via e-mail at james@ferretsoft.com or jholderness@geocities.com.

By Bill Todd

ReportBuilder Pro 4.0

Newcomer Is Worth Switching To

Although ReportBuilder is a relative newcomer to the family of Delphi reporting tools, it's been worth the wait. If you're looking for a single reporting tool with different layouts for different pages, side-by-side bands, newspaper-style columns, end-user reporting, and fast drag-and-drop report layout, this is it. (This review is based on a pre-release version of ReportBuilder 4.0, so there may be some differences between the features described here and the shipping product.)

To build a report, begin by dropping a *TTable* and *TDataSource* on a Delphi form. Next, move to the ReportBuilder page of the Component palette, and drop a *TppBDEPipeline* and *TppReport* on your form. The BDEPipeline component is one of a family of pipelines that supply data to your report. Use BDEPipeline to get data from a BDE (Borland Database Engine) database, or the DBPipeline if you're working with a BDE replacement. The TextPipeline component lets you use data in a text file for your report, and the JITPipeline supplies data from a non-database source. You can also create your own pipeline components to supply data from proprietary sources. Set the pipeline component's *DataSource* property to connect it to your DataSource component, and you're ready to double-click the Report component to open the designer, shown in Figure 1.

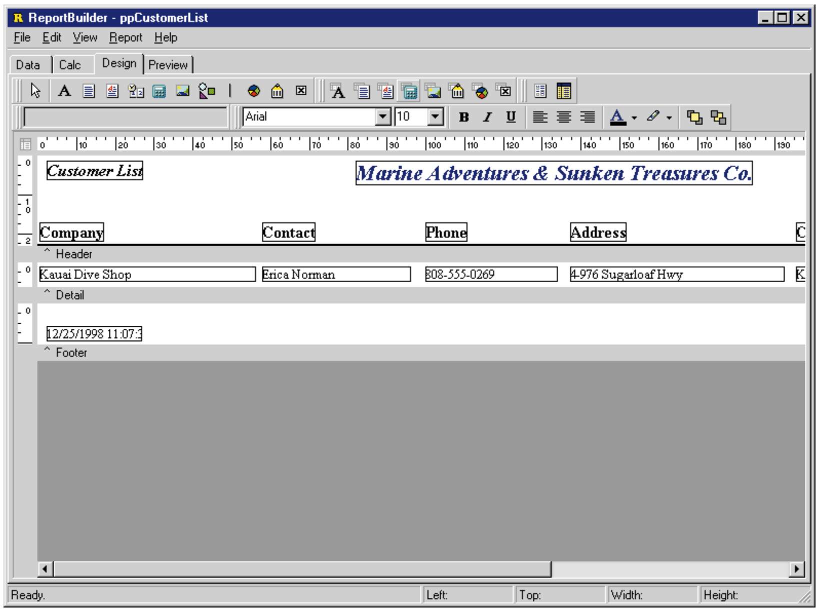
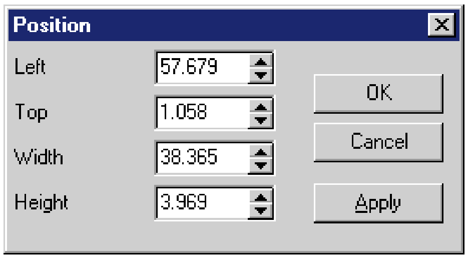


Figure 1: The ReportBuilder report designer.

Ergonomically Speaking

The designer is well laid out and easy to use. The first row of toolbars includes the non-data-aware, data-aware, and advanced components. The next row begins with a toolbar (which is empty in Figure 1). The contents of this toolbar vary depending on the component you select in the designer. For example, if you select a data-aware component, the toolbar contains a drop-down list of data pipeline components, and a drop-down list of fields from the currently selected pipeline. To the right is a text-control toolbar that will be familiar to anyone who has used a Windows word processor. At the end of the formatting toolbar are Bring To Front and Send To Back buttons for working with layered components.

The third row of toolbars begins with one that is unique: the nudge bar. The four buttons on this toolbar let you nudge the selected components one pixel up, down, left, or right. You can use a keyboard combination of Ctrl plus the appropriate arrow key to achieve the same effect. Either method makes precise


Position

Left 57.679

Top 1.058

Width 38.365

Height 3.969

Figure 2: The Position dialog box.

alignment easy. The next set of toolbars provides a full set of sizing and alignment options, including grow-to-largest and shrink-to-smallest, in both the horizontal and vertical directions; align left, right, or center; space the selected components evenly either horizontally or vertically; and center the selected components horizontally or vertically within their band.

You can also position and size components with great precision by right-clicking and displaying the Position dialog box (see Figure 2). The Position dialog box lets you specify size and position to three decimal places. The designer makes extensive use of context menus, so if you want to do something to a component, simply right-click on it; the pop-up menu will likely offer the choice you need.

From the **View | Toolbars** menu, you can open the Report Tree and Data Tree windows, shown in Figure 3. The Report Tree allows you to see all the components in each band of your report, and to select one or more components. This is handy for working with layered components. The Data Tree shows all the fields for each pipeline component on the report. You can drag fields from the Data Tree and drop them in any band of your report. When you drop a field, the appropriate data-aware control is created automatically. You can also choose whether dropping a field creates a data-aware control, a label containing the field name, or both.

Suite Components

ReportBuilder has a full suite of data-aware and non-data-aware components to print any type of data on your report. However, the two components on the Advanced toolbar — Region and SubReport — really set ReportBuilder apart. The Region component allows you to group components, including subreports, so they will be positioned as a group, both at design time and run time.

The SubReport component is a complete report, including bands, that you can position within another report. The power of subreports is limited only by your imagination. For example, you can use subreports to create a report that consists of several sections, each of which has its own layout and data source. When you add subreports to a report, a tab is added to the bottom of the report designer for each subreport. Simply click on a tab to go to the design view for that subreport. You can also place subreports in regions, and place them next to each other within bands of the main or other subreports. This means you could print a report that shows information about a customer, displays a list of that customer's locations, and displays a list of the products that customer has ordered. Subreports also give you the ultimate in power for creating multi-level master-detail reports.

ReportBuilder gives you total control of relative positioning when you place multiple components that can stretch, such as subreports or memos, in the same band. Suppose you want to print a memo, and below the memo, you need to print some other fields from a database. Simply set to True the *ShiftWithParent* property of the components below the memo, and they will move down as the memo component expands. Suppose you also want to put a frame around the memo component in your report. Simply drop a Shape component on top of the memo, set its *StretchWithParent* property to True, and send it to the back. Now the Shape component will automatically expand and contract to fit the size of the memo in each record as it's printed. If you have multiple stretching components positioned vertically in the same band, you can set each component's *ShiftRelativeTo* property to ensure they print in the correct order relative to each other.

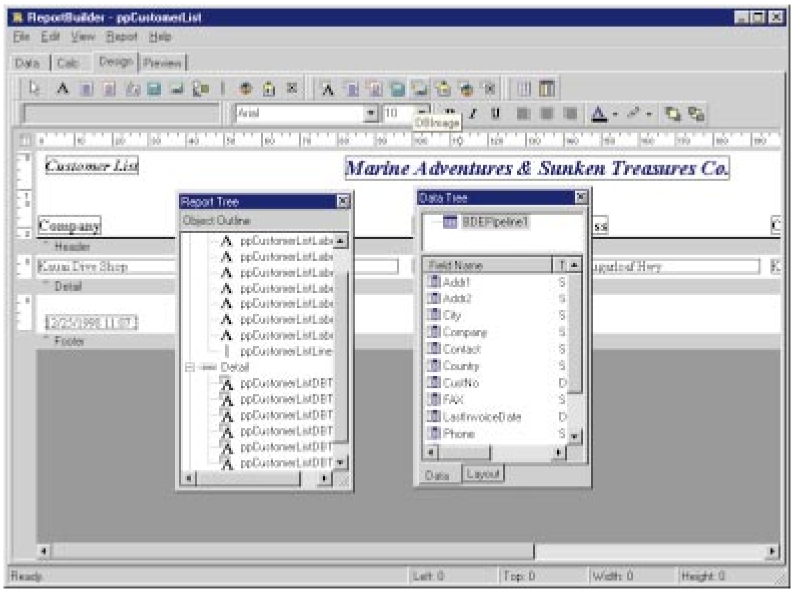


Figure 3: The Report Tree and Data Tree windows.

Number Crunching

ReportBuilder allows you to create calculated values in several ways. First, of course, you can add Delphi calculated fields to your datasets. Second, the *TppDBCalc* component lets you calculate the sum, count, min, max, or average for a group of records. Finally, the *TppVariable* component lets you add code to its *OnCalc* event handler to generate values any way you wish. If the value of one *TppVariable* component uses the value of another, you can set its *CalcOrder* to ensure the variables calculate in the correct order to produce a correct result. ReportBuilder also lets you choose to generate reports in one pass or two. One pass is faster, but using a two-pass report lets you display grand totals at the beginning of the report, or use "page *n* of *m*" page numbers.

Ease of Use

ReportBuilder lets you give your end users the ability to design and save reports with unparalleled ease and control. If you can safely turn your users loose in your database, a query wizard helps them create SQL statements to fetch the data on which to report. If the structure of the database is too complex (or the tables too large) to let an untrained user create queries and reports, you can create data views for the user to work with. A data view is a class that provides a level of abstraction between the database and the user. You create data views that contain query components that return a subset of data from one or more tables. End users employ the data views to create reports without direct access to the database.

The report designer for end users is identical to the designer used by developers. One feature of the designer particularly useful for end-user reporting is the ability to design a report and save it as a template. Templates can be saved to individual disk files, or to a BLOb field in a database. By creating and saving a template, you can give your end users a starting point from which they can more easily create the reports they need. ReportBuilder also includes a data dictionary component that allows you to create more readable aliases for table and field names. If you build a data dictionary, end users will only see the more meaningful alias names.

ReportBuilder 4.0 has its own programming language, RAP (Report Application Pascal). You and your end users can use RAP to write event handlers for any object in a report. This lets you easily create the code for calculated values in the report, and change the properties of any object as the report runs. For example, you could change the color of a field depending on whether the value is positive or negative. RAP is great for developers because it means you create and deploy complete reports, including event handlers, without having to change your executable. For end users, RAP provides the means to build very sophisticated reports. When you implement end-user reporting, ReportBuilder lets you set the level of RAP access available to your end users so you can match the complexity to their needs.

The Bottom Line

I hate creating reports. One of the first things I look for in a reporting tool is how fast it lets me create the dozens of simple reports that most applications require. The icing on the cake is the Report Wizard. While it's a great tool for end users, it saves time for developers as well. After setting up your data sources and pipeline components, fire up the Report Wizard. It will let you choose a report style, select the fields to be displayed, and define the groups you require; click the Finish button to create your report layout. Even if the report requires additional manual customization, getting the basic layout created without having to manually place and align each component is a real time saver.

ReportBuilder ships with an excellent user manual in the form of a Word document. The foundation of the manual is a series of tutorials of gradually increasing complexity that teach you how to use all the power of the product with a minimum investment of time. The tutorials cover everything from creating master-detail and master-detail-detail reports, to putting reports in DLLs and using the JITPipeline component to print a report from a string grid. The manual also contains an excellent introduction to the report designer, as well as an explanation of all its tools and time-saving features, where to find them, and how to use them. If you invest a few hours in working through the manual and tutorials, you'll find that you are instantly productive when you start building reports in your applications.

INFORMANT FACT FILE

ReportBuilder 4.0 is a relative newcomer to the family of Delphi reporting tools, but if you're looking for a single reporting tool with different layouts for different pages, side-by-side bands, newspaper-style columns, end-user reporting, and fast drag-and-drop report layout, this is it. I've looked at many reporting tools, and none of them compare with ReportBuilder 4.0.

Digital Metaphors Corp.
16775 Addison Road, Suite 613
Dallas, TX 75248

Phone: (972) 931-1941
E-Mail: info@digital-metaphors.com
Web Site: http://www.digital-metaphors.com
Price: ReportBuilder, US$249; ReportBuilder Pro, US$495; upgrade from ReportBuilder to ReportBuilder Pro, US$246.

ReportBuilder 4.0 is available in two versions. The Pro version includes the end-user reporting capability. Both versions support Delphi 1 through 4, and include full source and a 30-day, money-back guarantee.

Conclusion

I shudder at the thought of changing reporting tools. I will still have to support all my old applications using the old tool for a long time, and I will have to endure the pain of becoming proficient with a new tool. But ReportBuilder is worth the trouble. The ability to produce any kind of report my clients need, as well as provide end-user reporting using a single tool, will make life so much easier in the future that the change is worth the effort. I've looked at a lot of reporting tools, and none compare with ReportBuilder 4.0. Δ

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 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.

The Multimedia APIs

Unlike TAPI and some of the newer APIs, the multimedia APIs have been a part of Delphi since its inception. Unfortunately, the documentation and examples (in Delphi and elsewhere) are minimal. Increasingly, I get messages from developers struggling to work with these APIs and frustrated by the lack of information, so I'm writing a book on the topic (*The Tomes of Delphi: 32-bit Multimedia Programming* will be published by Wordware this summer). I will also be exploring selected multimedia topics in these pages. This column, which presents an overview of the multimedia APIs, is the first such contribution.

The 32-bit multimedia APIs fall into three general groups — low-level, mid-level, and high-level — with low-level APIs more device-specific, high-level APIs more generic, and mid-level APIs somewhere in between. In Windows 3.x, the MCI was the highest-level API available. With Windows 95, a higher-level interface was introduced: the *MCIWnd* class.

Low-level APIs. The low-level functions require more work, but provide a high degree of control over various media devices. These APIs include functions to work with various multimedia file types, audio, audio mixers, and joysticks. They also include the most precise timing functions in the Windows API. Considering multimedia's time-critical requirements, this should come as no surprise. As Bob Swart demonstrated at last summer's Inprise Conference, these timing functions can even be used to profile applications.

The Waveform API provides a low-level interface to audio devices. Most of these functions begin with "Wave," or more specifically, "WaveIn" or "WaveOut." The former group provides functions for recording .WAV files, the latter for playback. Because this is a low-level API, you generally need to call several of these functions. Often grouped with these files, but at a higher level, the *PlaySound* function provides an easier means of playing .WAV files. Closely related to these functions are the auxiliary audio functions that control auxiliary audio devices. All of these are prefixed with "Aux," e.g. *AuxSetVolumn*.

Another important set of low-level functions are those that control the MIDI (Musical Instrument Device Interface), enabling communication with a sound card's built-in synthesizer. With these MIDI functions, you can work with sounds directly, select various patches (instruments), and perform many sound-playing operations on the fly.

The low-level multimedia input/output functions provide a variety of file operations with buffered and unbuffered files, files in standard Resource Interchange File Format (RIFF), memory files, or files using custom formats. Similarly, there are low-level functions for working with .AVI files. These include a large number of routines for file I/O, streaming AVI data, and editing AVI streams. Finally, there are several functions for working with joysticks, and a handful of timer functions.

The MCI. The Media Control Interface is a mid-level API that provides a fairly easy means of working with a variety of multimedia files and devices, so you can create a sophisticated multimedia application with minimum coding. Best of all, the MCI provides two approaches (two sets of commands) for performing these tasks. The first, *mciSendString*, is ideal for prototyping applications; the second, *mciSendCommand*, is faster and better suited for the finished product. The command strings consist of words arranged in English-like sentences. For example, the words associated with the playback of a sound include "load," "play," and "stop." For example:

```
mciSendString("pause movie", nil, 0, nil);
```

Conveniently, these command strings have corresponding command messages, but messages are more complicated. The command message corresponding to the above command string looks like this:

```
mciSendCommand(MovieDevice, MCI_PAUSE, 0, DWORD(nil));
```

where *MovieDevice* is the handle of the movie-playing device returned by calling another command message using the MCI_OPEN command. As you can see, there's quite a bit more involved with command messages than with command strings. Sometimes, you're required to supply the third and fourth parameters (flags and parameters, respectively). In a running application, command messages are much faster than command strings, because the latter must be parsed.

A high-level class. The *MCIWnd* class is even easier to use, providing a quick way to create a multimedia control and associate it with some type of multimedia event. For example, the following statement (based on code from Delphi's online help) creates a button on a form that, when pressed, plays a video clip:

```
MCIWndCreate(hwndParent, g_hinst, WS_VISIBLE or WS_CHILD or
  MCIWNDF_SHOWALL, 'sample.avi');
```

Because this column is introductory in nature, it probably raises more questions than answers. However, I plan to explore these topics in more detail in future articles. I also plan to devote another column to sources of information on multimedia programming, both books and Internet sites. In the meantime, I'd like to hear from you concerning your experiences, questions, and discoveries in working with multimedia in Delphi. ∆

— 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.