February 1998, Volume 4, Number 2

Cover Art By: *Tom McKeith*

Delphi 3 ActiveX

A Handy Introduction

ON THE COVER

FEATURES

REVIEWS

DEPARTMENTS

Delphi TOOLS

New Products and Solutions

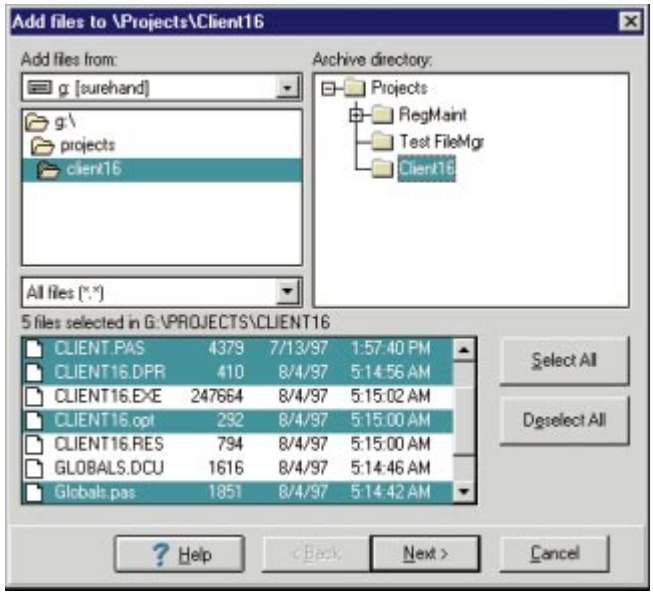
SureHand Software Offers ViCiouS Pro

SureHand Software introduced ***ViCiouS Pro***, a version control system that enables recognition, customization, and creation of logical file families. ViCiouS Pro includes file family definitions for C++Builder, Delphi projects, Delphi forms, Paradox tables, and dBASE tables.

Users have control over definitions for keyword expansion on a per-file extension basis, with the ability to establish standard history blocks for logging comments. ViCiouS Pro also supports keyword expansion within incoming and outgoing files.

ViCiouS Pro supports single- or multiple-user environments. One copy of common units and forms may be shared between multiple projects.

ViCiouS Pro integrates directly with Delphi and C++Builder. Programmers may also access the version control features by using the Archive Explorer. The Admin Tool provides facilities for defining users, file families, and keyword definitions.



ViCiouS Pro is available in 16- and 32-bit versions.

SureHand Software
Price: US$76 per user license, with discounts for multiple users.
Phone: (314) 963-1935
Web Site: http://www.surehand.com

Blinkinc Ships Shrinker 3.2

Blinkinc announced the release of ***Shrinker 3.2***, a utility that compresses and transparently decompresses Windows and real-mode DOS programs. Shrinker 3.2 compresses 16- and 32-bit Windows programs and resources, including EXEs, DLLs, DPLs, OCXs, and ActiveX controls. This version includes support for long filenames and an enhanced Windows user interface. Shrinker 3.2 supports Windows 3.1, Windows 95, Windows 98, and Windows NT 3.51 and 4.0.

Any EXE, DLL, DPL, or OCX compressed with Shrinker will transparently decompress itself into memory at run time. By reducing the amount of data transmitted, and decompressing the program on the local workstation at run time, Shrinker minimizes LAN, WAN, and Internet traffic.

For pricing and information, call (804) 784-2087or visit http://www.blinkinc.com.

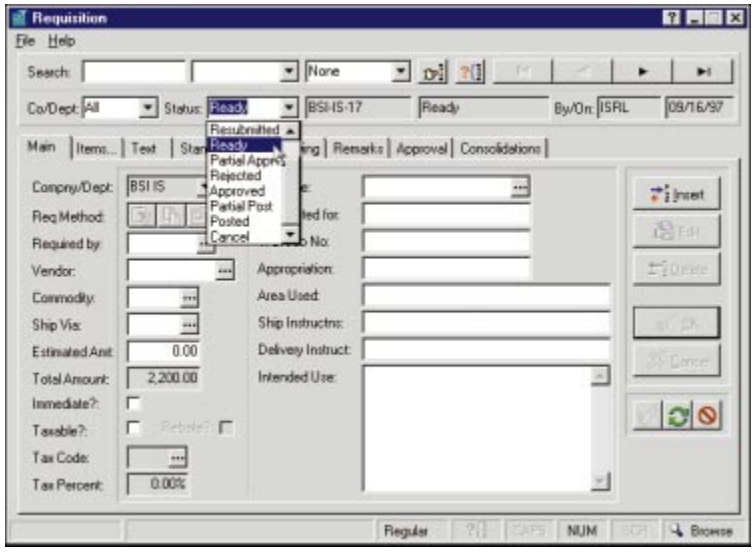
Business Solutions Introduces Client/Server Version of Purchase Manager

Business Solutions, Inc. announced ***Purchase Manager 2.0***, a client/server version of the company's purchasing management system. Written in Delphi, it runs against the Oracle database, and serves medium to large organizations.

Purchase Manager handles regular and quick requisitions, purchase orders, contract orders and releases, RFQ bidding, and shipment orders for raw materials, spare parts, and services. It also includes a comprehensive inventory management system, and easily interfaces with existing accounts payable and financial systems.

Complete Delphi source code, implementation assistance, and consulting are also available. The source code version includes the BSI Guardian Application Security System, also written in Delphi. Although Purchase Manager is written for Oracle, its architecture allows conversion to any BDE-supported database.

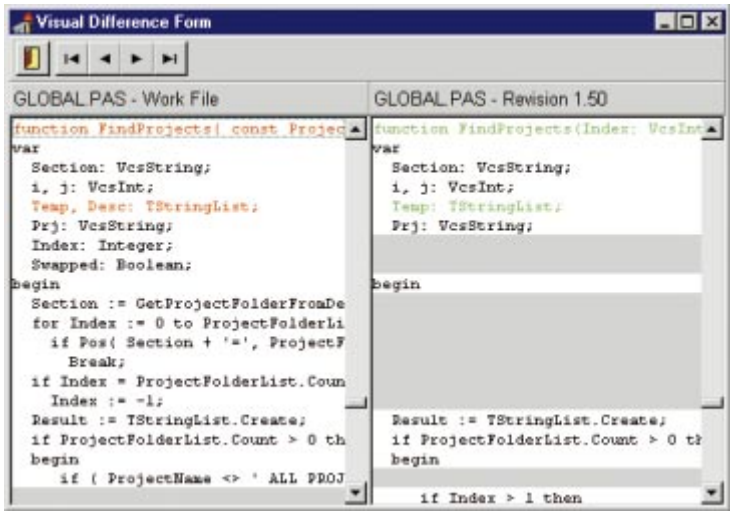
Business Solutions, Inc.
Price: US$10,000, plus US$500 per user; additional US$10,000 for source code.
Phone: (218) 384-4210
Web Site: http://www.bsi-net.com



Delphi TOOLS

New Products and Solutions

Quality Software Components Releases GP-Version 3.5



Quality Software Components, Ltd. released *GP-Version 3.5*, a change control system for Delphi and C++Builder. This version includes user-defined file groupings, which allows users to define files that should be archived as a single entity and specify what file types are dependent on others. With Delphi, GP-Version recognizes which files must remain writeable when checked in.

GP-Version also allows users to define multiple configurations, allowing a different set-up for each client.

The entire functionality of the 32-bit GP-Version is defined in a single .DLL, and doesn't require any third-party tools. An API defined into this .DLL is available upon request. This allows developers to add version control functionality to any file-based application.

Quality Software Components, Ltd.
Price: US$125 per user
E-Mail: info@qsc.u-net.com
Web Site: http://www.qsc.u-net.com

EFD Announces HyperString 2.0 for Delphi

EFD Systems announced *HyperString 2.0*, a comprehensive library of string functions designed for Delphi's 32-bit dynamic string type.

HyperString offers over 200 functions, both high- and low-level, with many coded in hand-optimized assembler. All functions are documented in WinHelp format for integration into the Delphi online Help system, providing access during coding.

Included in the HyperString function set are the usual array of search and edit routines, along with hashing, encryption, compression, fuzzy-matching, numeric expression evaluation, and a unique implementation of dynamic arrays using dynamic strings as containers. Token routines provide parsing of HTML/XML strings, and support the creation of hierarchical data structures inside strings.

For more information visit http://efd.home.mindspring.com.

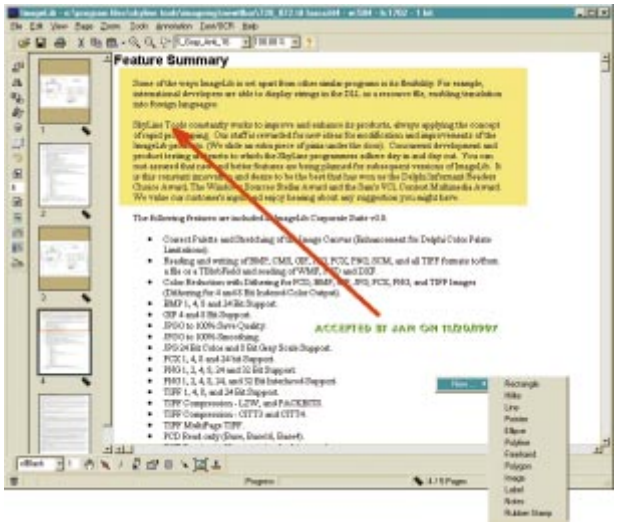
Skyline Tools Releases ImageLib Corporate Suite 3.0

Skyline Tools announced version 3.0 of *ImageLib Corporate Suite*, a document imaging development package that features over 40 image manipulation and correction tools.

The ImageLib Corporate Suite is compatible with Delphi, C++Builder, Borland C++, Microsoft Visual C++, Visual Basic, and others, allowing developers to create desktop, database, Internet, and multimedia applications.

The suite supports JPEG, GIF, PNG, PCX, BMP, Baseline TIFF, TIFF3/CITT, TIFF4/CITT, multi-page TIFF, TIFF Packbits, TIFF LZW, Kodak Photo-CD, TGA, DFX, ICO, EPS, and other formats.

The suite offers OCR features, as well as image-annotation capabilities, such as highlighting, "paste its" (sticky notes), labels, arrows, notes, polygons, circles, and squares. The zoom, scroll, and pan features offer control when moving across enlarged images.



Version 3.0 includes a TWAIN Manager, which scans and stores images in a multi-page TIFF in one step, and supports all TWAIN-compliant digital cameras.

In addition, the suite includes the Thumbnail Image Manager and point-and-click image-correction effects, such as brightening, contrast, gamma correction, color reduction, and rotation, as well as special effects filters (mosiac, page curl, wave, transitions, and more). It also includes a built-in video frame grabber and the ImageLib WebKit.

Skyline Tools
Price: US$599
Phone: (818) 766-3900
Web Site: http://www.imagelib.com

News LINE

February 1998

Borland Announces Delphi Enterprise

Scotts Valley, CA — Borland announced Delphi Enterprise, an integrated development and middleware solution for corporations building secure, fault-tolerant, distributed software applications. Based on Delphi development tools and Entera middleware technology, Delphi Enterprise provides a robust, multi-platform, end-to-end solution that can support large volumes of users and data. Delphi Enterprise allows corporations to deliver enterprise-class applications that leverage their corporate assets, including investments in legacy systems, developer skill sets, infrastructure, and current business practices.

Delphi Enterprise includes technical support, installation, training, and maintenance contracts, as well as optional enterprise consulting services.

Delphi Enterprise prices start at US$45,000, depending on developer seats and server configuration. The product is available directly from Borland and through select Borland partners. For more information, call Borland's direct corporate sales at (408) 431-1064, or visit the Borland Web site at http://www.borland.com.

Borland Reports Second Quarter Fiscal 1998 Results

Scotts Valley, CA — Borland announced results for its fiscal year 1998, second quarter, which ended September 30, 1997. Second quarter revenues for fiscal 1998 were US$42,500,000, compared to US$39,306,000 for the second quarter of the previous fiscal year. (Revenues were up eight percent over the second quarter of fiscal year 1997.)

Year-to-date fiscal year 1998 revenues were US$84,470,000, a nine-percent increase over revenues of US$77,452,000 for the first half of fiscal year 1997.

Net income for the second quarter of fiscal year 1998 was US$1,518,000, or US$.03 earnings per share, as compared to a net loss of US$14,310,000 or US$.40 per share for the second quarter of fiscal year 1997.

Revenues from Borland's client/server, enterprise, and Internet products continued to grow, making up 55 percent of total revenues in the second quarter of fiscal year 1998.

Borland attributes its profitability to continued updates on existing products, as well as new products, such as JBuilder.

Arabic Language Support for Delphi 3

Scotts Valley, CA and *Frankfurt, Germany* — Borland announced the availability of Arabic language support for Delphi 3 Client/Server Suite and the Delphi 2 and Delphi 3 line of tools. With Delphi Arabic Enablement for Windows 95, any existing installation of Delphi 2 or 3 running on Arabic Windows 95 can be upgraded to develop full Arabic applications, including Arabic database, reporting, and display capabilities.

Delphi Arabic Enablement for Delphi 2 and 3 is available from Borland and select distributors for US$99.

SAP Announces Open BAPI Network

Atlanta, GA — Systems, Applications, and Products in Data Processing (SAP) AG launched the Open BAPI Network to encourage the free flow of information among developers using its Business Application Programming Interfaces (BAPI) at customer sites, partner companies, and within SAP itself. The Open BAPI Network ensures that developers building applications complementary to R/3 have access to information for BAPI-based development.

For information or to register for membership visit the SAP Web site (http://www.sap.com).

Borland Signs Letter of Intent, Releases C++Builder Client/Server Suite

Borland signed a letter of intent with IBM Corp. to jointly develop and deliver Java development solutions for IBM AS/400e series business computers.

Borland and IBM will work to provide AS/400 developers with integrated solutions for developing Java applications and applets. The development solutions will be based on Borland's new JBuilder family of pure Java development tools.

Borland also released C++Builder/400 Client/Server Suite, a C++-based RAD tool specialized for the IBM AS/400 series of business computers. Borland's C++Builder/400 combines native connectivity to the AS/400, a reusable object-oriented component library and high productivity visual design tool.

C++Builder/400 Client/Server Suite is available from Borland and authorized Borland/400 business partners. Prices start at US$3,995. For more information call (800) 233-2444 or visit http://www.borland.com/borland400/. International customers can contact their local Borland office or visit http://www.borland.com/borland400/partner.html.

By Dan Miser

Delphi 3 ActiveX

A Handy Introduction

Microsoft has defined a lightweight subset of their OLE technology that is well suited for the Internet. This technology is called ActiveX. Delphi 3 gives developers the ability to easily use existing ActiveX controls in their applications, as well as turn their native VCL components into ActiveX controls. Delphi 3 also provides developers the ability to extend ActiveX controls, and even create custom ActiveX controls from scratch.

This article demonstrates five aspects of ActiveX programming with Delphi 3:

- importing existing ActiveX controls
- converting VCL controls into ActiveX controls
- extending an ActiveX control by adding a property page
- creating custom ActiveX controls
- converting a Delphi form into an ActiveForm

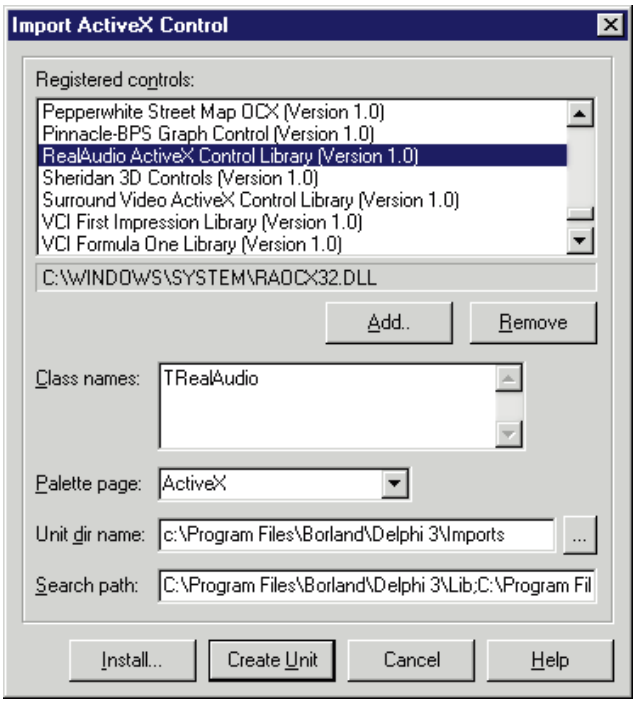


Figure 1: The Import ActiveX Control dialog box.

That's a lot of ground to cover, so let's get to it. (Note: You'll need Microsoft Internet Explorer version 3 or higher to access ActiveX controls on the Web. For Netscape users, a plug-in called ScriptActive will allow ActiveX access. It's available from NCompass Labs at http://www.ncompasslabs.com.)

Importing an Existing ActiveX Control

Because they're components, ActiveX controls fit nicely into Delphi's component-based development strategy. In fact, Delphi 3 gives you the ability to integrate these controls right into the Delphi IDE. Select **Component | Import ActiveX Control** from the menu to display the list of ActiveX controls registered on your system. Any control found in this list can be imported as a Delphi component.

The following example will demonstrate how to install an ActiveX control, and how to use it in the Delphi 3 IDE. To follow this example, you must have Progressive Networks' RealAudio ActiveX control installed on your system; you can find it at http://www.real.com/products/player/-playerdl.html.

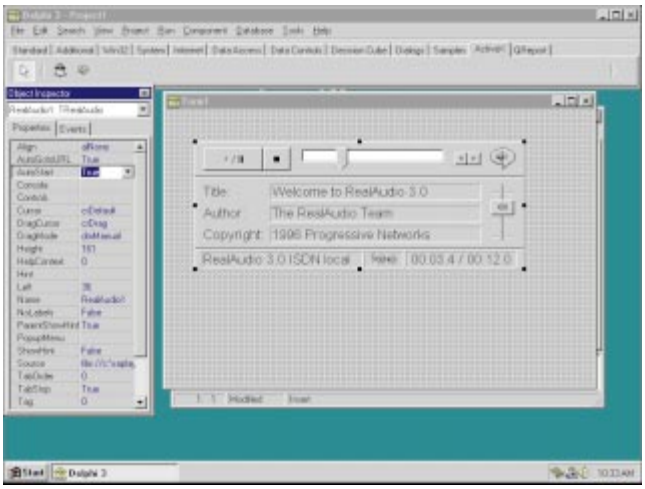


Figure 2: The RealAudio ActiveX control in action.

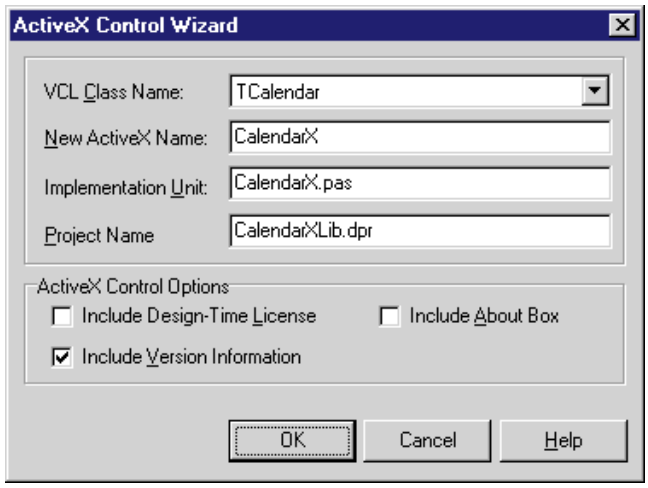


Figure 3: The ActiveX Control Wizard.

Select **Component | Import ActiveX Control**, and modify the dialog box to look like that shown in Figure 1. If you were to click the **Create Unit** button, Delphi would generate a VCL wrapper by reading the type information of the ActiveX control, and translating it into Object Pascal syntax. The source code would then be saved in the directory you specified for **Unit dir name**.

However, because all Delphi 3 components need to be installed to a package before you can use them in the Delphi IDE, you should click the **Install** button. This will create the unit as previously described, then allow you to insert the component into a package by using Delphi's standard component-installation dialog box. Next, you'll be prompted to rebuild the modified package. After recompiling, you can treat the ActiveX control as you would any native VCL component.

You may have noticed the **Add** and **Remove** buttons on this dialog box. These give you a quick, easy way to install and remove ActiveX controls without resorting to Regsvr32, or some other registration utility. Figure 2 shows a RealAudio component on a form; its *Source* property is set to a RealAudio file, and Play is selected from the context menu. Notice that the file is being played at design time.

Creating ActiveX Controls from VCL Controls

You may be wondering how to create your own ActiveX control that others may use. Delphi comes to the rescue again by providing the ability to convert VCL controls to ActiveX controls. This capability is not restricted to Borland's standard VCL controls; you can even convert your custom VCL controls.

Select **File | New | ActiveX | ActiveX Control** to run the ActiveX Control Wizard. This will generate a fully functional implementation of the ActiveX control, which you can use from within your Delphi applications. For now, let's build a sample ActiveX control based on Delphi's Calendar control (which appears on the Samples page of the Component palette). Modify the ActiveX Control Wizard to look like the dialog box in Figure 3, and click **OK**.

Not all VCL controls can become ActiveX controls; to display a VCL control in the ActiveX Control Wizard, three requirements must be met:

- The VCL control must descend from *TWinControl*. This may force you to alter the parentage of your VCL control, but there is usually a suitable alternative. For example, if you're trying to port a non-visual control to an ActiveX control, you might consider using *TCustomControl* as the ancestor, and provide a simple *Paint* method.
- The VCL control must descend from a control capable of supporting ActiveX conversion. Most Delphi components use the *RegisterComponents* procedure to install themselves to the Component palette. However, if you don't want a control to be converted to ActiveX, you can use the *RegisterNonActiveX* procedure to install the component. A common reason to disallow ActiveX conversion is that the VCL control references other components. Having ActiveX controls talk to each other in this manner would be very difficult, so it's easier to disallow the conversion altogether. Delphi's data-aware controls are a prime example of this. Also, registering a control with *RegisterNonActiveX* will disqualify that control's descendent components.
- The VCL control must be installed in Delphi. If the component isn't installed in the Component palette, it's not registered with Delphi.

Once you click **OK**, Delphi will convert the VCL control to an ActiveX control. Because an ActiveX control is based on OLE, the VCL control can only translate properties that have a corresponding native OLE data type. In addition, you can provide an adapter to convert the nonstandard data type into something that OLE can understand. For example, Delphi comes bundled with adapters, so OLE can use properties of types *TString*, *TPicture*, and *TFont*. If Delphi can't map a data type to a type that OLE can understand, Delphi will omit that property from the generated ActiveX control. You can add properties and methods to the ActiveX control manually by selecting **Edit | Add To Interface**, or editing the type library itself. Or, you can use the Type Library editor (by selecting **View | Type Library**) and have Delphi 3 do most of the work.

```
library CalendarX;

uses
  ComServ,
  CalendarX_TLB in 'CalendarX_TLB.pas',
  CalImpl in 'CalImpl.pas' { CalendarX: CoClass },
  CalPage in 'CalPage.pas' { CalendarPage: TPropertyPage };

exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.TLB}

{$R *.RES}

{$E ocx}

begin
end.
```

Figure 4: The source file for CalendarX.

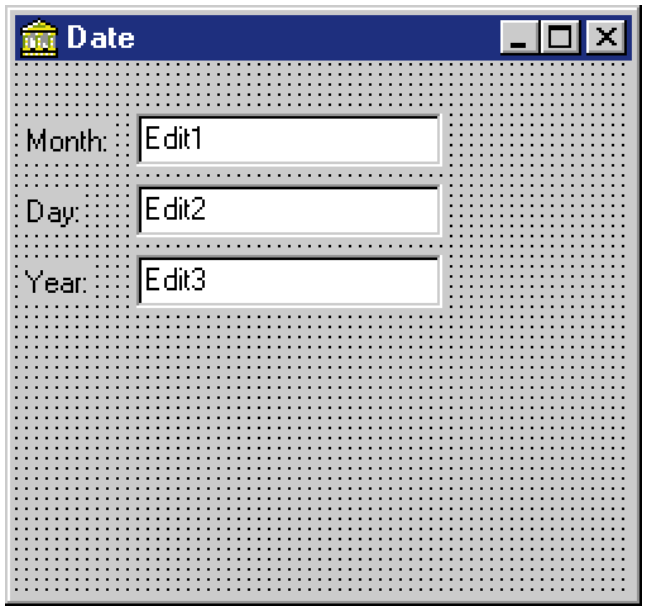


Figure 5: *TCalendarPage* at design time.

Once the code is generated for the ActiveX control, we can manipulate it as we would any other Delphi project. For now, we'll save the control, then compile it by using the **Project | Build All** menu item.

After you've compiled the ActiveX control and registered it with the system, any application that can use ActiveX controls for development can now access this control. Just for fun, you can import your resulting ActiveX control into Delphi using the steps previously covered. This will allow you to view this control as other users might.

Anatomy of an ActiveX Control

Every project created in Delphi needs a project source file (a .DPR file). ActiveX controls are no exception. An ActiveX Library is the project source file for all ActiveX projects. This file exports the four signature methods of an ActiveX control, provides an extension compiler directive, and includes the appropriate type library.

The source file for an example calendar project, CalendarX, is shown in Figure 4. The compiler directive:

```
{$E ocx}
```

tells Delphi what extension to give to the compiled version of the project. To create a new ActiveX Library project from scratch, you would select **File | New | ActiveX | ActiveX Library**.

The CalendarX control is good, but it would be nice to offer an easier way to edit the properties. This is precisely what *property pages* were made for.

Property Pages

Property pages give users a friendly way to alter the properties of an ActiveX control. A property page has a visual control that represents a corresponding ActiveX property. These controls are displayed in a tabbed notebook.

To continue our example project, we'll give CalendarX the ability to edit the date, in a property page. Create a property page by selecting **File | New | ActiveX | Property Page**. Press **OK** to generate skeleton code for a property page. After the form has been created, modify it to look like the one in Figure 5.

The **OK**, **Cancel**, and **Apply** buttons are all provided for you. In addition, the tabbed notebook reflects the number of property pages you've created. The only controls you need to place on a property-page form are those that implement the desired behavior of the property page.

The following steps are necessary to completely implement a property page:

1) Update the visual controls of the property page to match the state of the ActiveX control.
2) Modify the visual controls.
3) If necessary, update the ActiveX control's properties to reflect the changes made in the property page. This is done only if the **OK** or **Apply** button is pressed.

Delphi supplies all the pieces to easily implement this logic. The *UpdatePropertyPage* method will be called whenever the property page is about to be displayed. This is where you set the visual controls to show the ActiveX control's properties. For example, the following method will set the edit controls to the values of the ActiveX control:

```
procedure TCalendarPage.UpdatePropertyPage;
begin
  edMonth.Text := OleObject.Month;
  edDay.Text := OleObject.Day;
  edYear.Text := OleObject.Year;
end;
```

The *TPropertyPage* object contains a reference to the ActiveX control, and places it in a variable called *OleObject*. Access this variable whenever you need to access properties or methods of the ActiveX control.

The ActiveX control must be informed when its properties change. Whenever a user changes a property via a property page, you need to call the *Modified* method of the property page. This tells the ActiveX control that it needs to update itself. For example, the following method is assigned to each Edit component's *OnChange* event:

```
procedure TCalendarPage.EditChange(Sender: TObject);
begin
  Modified;
end;
```

The *UpdateObject* method is called when the ActiveX control is about to set its contents from the property page. Basically, you assign the values you set in *UpdatePropertyPage* back to the *OleObject* of the ActiveX control.

Register the Page

The last step is to bind the property page to the ActiveX control by registering it. This is accomplished by calling the *DefinePropertyPages* procedure inside the ActiveX control's *DefinePropertyPage* method. This method was created in the skeleton code when you created the ActiveX control. Simply pass the CLSID of the page that you want to register. You can find this value in the interface portion of the property-page unit:

```
procedure TCalendarX.DefinePropertyPages(
  DefinePropertyPage: TDefinePropertyPage);
begin
  DefinePropertyPage(Class_CalendarXPage);
  DefinePropertyPage(Class_DFontPropPage);
end;
```

Note also that you'll need to add the name of the property-page unit to the Calendar ActiveX unit (that's where you defined *Class_CalendarXPage*).

Every property page you define creates another tab on the property-page editor. This lets the user concentrate on one logically related group of data at a time. For example, you could create one property page for modifying the fonts used in a control, and another for modifying graphic elements. This would require calling the *DefinePropertyPage* method for each *PropertyPage* you want to register for this ActiveX control.

You can also easily give your control the ability to modify properties of type *TFont*, *TColor*, *TPicture*, and *TStrings*. These property pages will scan through your ActiveX control's properties at run time, and let the user modify individual attributes of these properties. These property pages are implemented in Stdvcl32.dll; therefore, if your control accesses one of the standard property pages (see Figure 6), you'll need to deploy this file as well.

ActiveX from Scratch

We've seen the powerful ability of Delphi to generate ActiveX controls by using the ActiveX Control Wizard, but what do you do if your control isn't listed there? Aside from the simple case where you have full control of the component's parentage, you'll need to create the ActiveX control from scratch. Here's where studying Delphi's generated code really pays off.

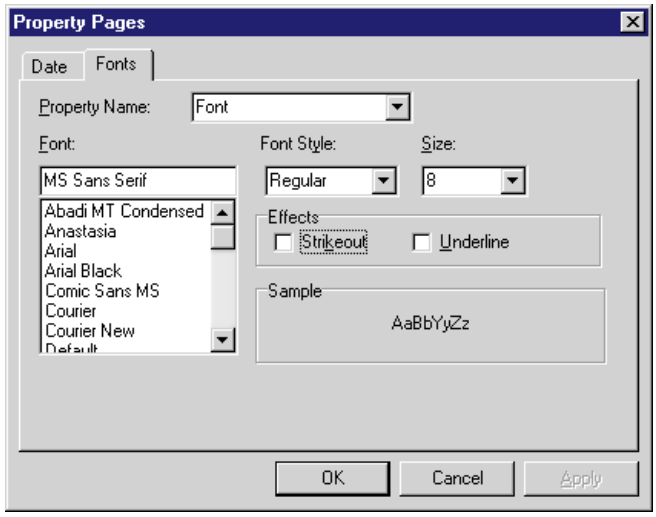


Figure 6: A standard ActiveX property page at run time.

TreeView components are not listed in the ActiveX Control Wizard mainly because the *Items* property is of type *TTreeNodes*. This is not an OLE-compatible type, so there is no automatic way for Delphi to make this structure ActiveX-compliant. However, with a little reengineering, we can expose some of the functionality of the *Items* property, thus allowing the control to be manipulated as an ActiveX control.

To create this ActiveX control, first make sure all the files are closed inside Delphi. Then:

1) Select **File | New | ActiveX Library** to get the project file used by ActiveX controls. This creates the project framework for an ActiveX control.
2) Select **File | New | Automation Object**, and give the object a class name of *TTreeViewX*. This will create a COM-object declaration, as well as a type library. Both these files will be modified throughout the rest of this process.
3) Make the following changes to the source-code statements in the Automation Object unit: The *TTreeViewX* class must descend from *TActiveXControl* instead of *TAutoObj*. The factory-creation class in the initialization section must read:

```
TActiveXControlFactory.Create(ComServer, TTreeViewX,
                              TTreeView,
                              Class_TreeView, 1, '', 0)
```

4) Add the extension directive to the project source file. This step ensures that when you compile the project, the resulting binary file will have the standard ActiveX extension — namely, OCX:

```
{$E ocx}
```

5) Borrowing heavily from a Delphi-generated ActiveX control, make the following modifications to the *TTreeViewX* class:

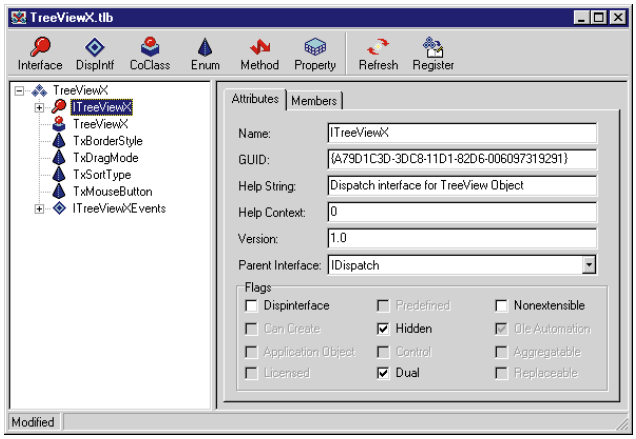


Figure 7: The Type Library editor.

```
TTreeViewX = class(TActiveXControl, ITreeViewX)
  private
    FDelphiControl: TTreeView;
    FEvents: ITreeViewXEvents;
  protected
    procedure InitializeControl; override;
    procedure EventSinkChanged (const EventSink:
                                IUnknown); override;
    procedure DefinePropertyPages (DefinePropertyPage:
                                   TDefinePropertyPage);
                                   override;
```

These methods will provide the link between the VCL control and the ActiveX control. Whenever you make a change to the ActiveX control, you'll actually pass that change on to the VCL control that it represents. See the source code for this month's article for the complete implementation of this control (see end of article for download details).

The skeleton for *TTreeViewX* is now complete. To allow access to the properties, methods, and events of the *TreeView* control, we need to add these elements to the ActiveX wrapper through the type library.

For example, to add the *Indent* property to the control, select **View | Type Library** to invoke Delphi's visual Type Library editor (see Figure 7). Select the *ITreeViewX* interface, and press the **Property** button. Name this property `Indent`. Next, press the **Refresh** button. This will add two methods to the *TTreeViewX* class: *Get_Indent* and *Set_Indent*. This is where the communication between the VCL control and the ActiveX wrapper occurs.

For the *Indent* property, it's a simple matter of reading and writing the value of *FDelphiControl.Indent*. Enumerated types are only slightly more complex to deal with, because you need to typecast the result to the appropriate type. You can learn a lot by looking at one or two Delphi-generated ActiveX control wrappers.

A *TreeView* control without items is like a vintage car without a steering wheel: It might be interesting to look at, but you'll never use it. We need to provide a way to add items to this control without exposing the *TTreeNodes* structure to the outside world. Creating the *AddChild* and *AddRoot*

```
procedure TTreeViewX.AddChild(const RootIndex,
  Index: Integer; const S: WideString);
var
  ARoot, AChild : TTreeNode;
  i : Integer;
begin
  // Find the right root position.
  ARoot := FDelphiControl.Items.GetFirstNode;
  for i := 0 to RootIndex-1 do
    if ARoot <> nil then
      ARoot := ARoot.GetNextSibling;

  // Now find the right child position.
  if (ARoot <> nil) and
     (ARoot.HasChildren) then
    begin
      AChild := ARoot.GetFirstChild;
      for i := 0 to Index-1 do
        if (AChild <> nil) and
           (AChild.GetNextChild(ARoot) <> nil) then
          AChild := AChild.GetNextChild(ARoot);
      FDelphiControl.Items.Add(AChild, S);
    end
  else
    // If sorted, inserted in sort order; else, inserted
    // as last child node.
    FDelphiControl.Items.AddChild(ARoot,S);
end;

procedure TTreeViewX.AddRoot(const Index: Integer;
                             const S: WideString);
begin
  if FDelphiControl.Items.Count = 0 then
    FDelphiControl.Items.Add(nil, S)
  else if (Index>=0) and
          (Index<FDelphiControl.Items.Count) then
    FDelphiControl.Items.Add(FDelphiControl.Items[Index],S);
end;
```

Figure 8: The *AddRoot* and *AddChild* methods.

methods to the type library will give *TTreeViewX* users a way to add items programmatically. The implementation of these methods is shown in Figure 8.

Registering the ActiveX Control

After building an ActiveX control, you can take advantage of Delphi's ability to register the control for you. Select **Run | Register ActiveX Server** to create the required registry settings for the ActiveX control. Once registered with your system, you can use the ActiveX control in any capable environment — including Visual Basic and ActiveX Control Pad.

Of course, selecting **Run | Unregister ActiveX Server** will remove the ActiveX control from the registry. We'll cover this topic in more depth next month, when we explore the issues concerned with deploying ActiveX controls.

ActiveForms

Delphi 3 has supplied you with the ability to easily turn an existing form into an ActiveX control known as an *ActiveForm*. The only real difference between an ActiveForm and an ActiveX control is that the ActiveForm will likely contain several visual controls. You can think of it as a super-component if you want. ActiveForms give you the ability to deploy almost any Delphi form via the Web. However, there are some limitations to what an ActiveForm

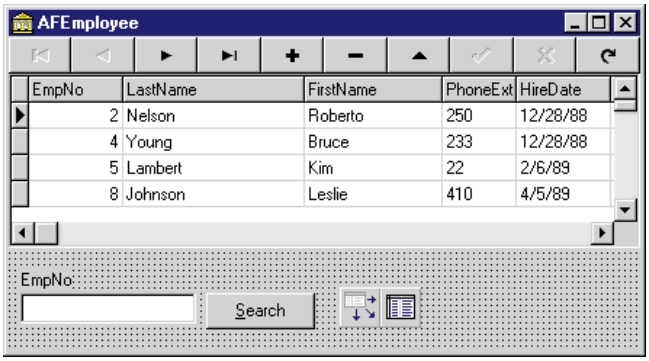


Figure 9: The ActiveForm at design time.

can do. For example, an ActiveForm cannot be a robust, multi-form application. If you need to access multiple screens in your ActiveForm, consider using a Tabbed Notebook interface instead.

Creating an ActiveForm

Let's create a simple ActiveForm to view and edit information from the Employee table found in \Dbdemos. To create an ActiveForm, select **File | New | ActiveX | ActiveForm**. An ActiveX Library will be built for you, if necessary.

(Note: This demonstration requires BDE 4.0 to be installed on every client machine that will access this ActiveForm. You can also deploy a thin-client solution by using MIDAS. For more information on how to use MIDAS in a Delphi application, see http://www.borland.com/midas.)

An ActiveForm is really not much different than a standard Delphi form. For this example, we'll place some standard data-aware controls on the ActiveForm, as shown in Figure 9. Notice that we can still use the standard Table and DataSource components to drive our data-aware controls.

Once all the controls are configured, we can save, compile, and register the ActiveForm as we would any other ActiveX control. At this point, you could use the ActiveX control in Visual Basic, with all its functionality intact. However, it would be nice to give the ActiveX user an interface to control some individual elements of this super-control.

Because an ActiveX control is a DLL, you can't directly access public variables inside the control. Instead, access to the underlying variables of a form must occur through a procedural interface. Use **Edit | Add To Interface** to make a property accessible to the outside world. For even more control over this interface, use the Type Library editor to manipulate all the attributes of the associated type library.

In our example, we would like to give the user a programmatic way to control the contents of the Search edit control's text property. Select **Edit | Add To Interface** to display the dialog box shown in Figure 10. This will create two stub methods to your ActiveForm: *Get_Search* and *Set_Search*. Fill in the two methods to get and set the *edSearch.Text* property of the ActiveForm, respectively. Now you can access *edSearch.Text* from any ActiveX development environment by assigning a value to the *Search* property of the ActiveForm:

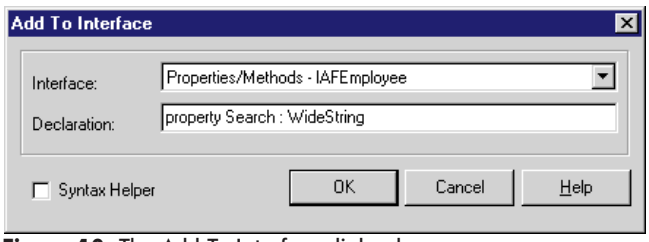


Figure 10: The Add To Interface dialog box.

```
function TAFEmployee.Get_Search: WideString;
begin
  Result := edSearch.Text;
end;

procedure TAFEmployee.Set_Search(const Value: WideString);
begin
  edSearch.Text := Value;
end;
```

We can further extend this control by adding functionality to programmatically search a database, and even control the visibility of all the search-related controls. In addition, because this really is an ActiveX control, we could even add property pages to this ActiveForm, and import the control for use within Delphi.

Conclusion

So there you have it — a whirlwind tour. From simply importing them, to creating your own, in this article we've quickly covered many aspects of Delphi 3 ActiveX programming. In my next article, we'll discuss techniques for deploying ActiveX controls and ActiveForms, including how, when, and why to use run-time packages with your ActiveX controls, and how to deploy them to the Web with INF and CAB files. Δ

The projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\FEB\DI9802DM.

Dan Miser is a software developer residing in Southern California with his wife and daughter. He has been a Borland Certified Client/Server Developer since 1996, and is a frequent contributor to *Delphi Informant*. You can contact him at http://www.iinet.com/users/dmiser.

By Tom Stickle

ActiveX Scripting

Adding Scripting to Your Delphi 3 Applications

For advanced applications that require customization, a scripting language can be a useful feature. Unfortunately, numerous factors have deterred developers from providing scripting support in their applications: Creating a new scripting language from scratch absorbs a great deal of development time while increasing long-term training and support costs, and licensing a scripting engine from a third-party company is a potentially expensive undertaking that creates a dependency on the vendor.

With the arrival of ActiveX scripting, developers finally have a vendor-independent platform that enables them to add standard scripting support to their applications. By using the Component Object Model (COM), ActiveX scripting provides a common set of interfaces that allow developers to access various scripting languages in the same manner. Your application need not care what language the script is written in, because the script executes in a separate object known as a *scripting engine*. For example, if your application supports ActiveX scripting, you can use Microsoft's Visual Basic Scripting Edition (currently available free of royalties), VBA, JavaScript, or Perl — without changing a single line of code.

The ActiveX scripting implementation consists of two inter-related COM objects: the scripting engine and the scripting host. The scripting engine is the COM object that processes scripts, i.e. VBScript or JScript. The scripting host is a COM object that is tied to our application. Figure 1 shows the basic architecture of ActiveX scripting. The idea is that the scripting host object is responsible for creating our application's custom COM objects, as well as handling communication from the scripting engine to the application. This is accomplished by "fastening" the scripting host to the scripting engine at the time the engine is initialized.

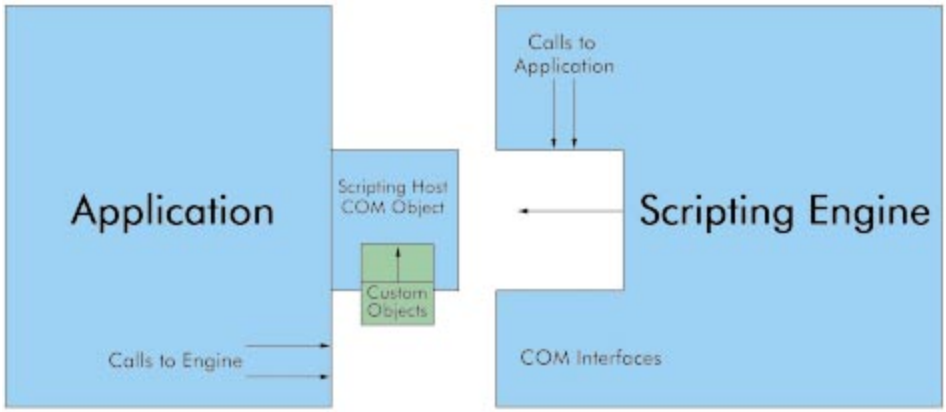


Figure 1: The basic architecture of ActiveX scripting.

The Scripting Engine

To successfully implement ActiveX scripting in an application, you first need a basic knowledge of the interfaces and methods available. In this article, we'll tackle the commonly used interfaces and methods

```
function TScriptingDemo.InitEngine : Boolean;
var
  CatID: TGuid;
begin
  Result := False;

  { Choose CatID based on the menu selection. }
  if tmVBScript.Checked then
    CatID := CatID_VBScript
  else
    CatID := CatID_JScript;

  ActiveScript := IActiveScript(CreateComObject(CatID));

  { Get Interface pointer for
    IActiveScriptParse Interface. }
  if (ActiveScript.QueryInterface(IID_IActiveScriptParse,
                          ActiveScriptParse) <> S_OK) then
    Exit;

  { Instantiate Site Object. }
  ActiveScriptSite :=
    IActiveScriptSite(TActiveScriptSite.Create);

  { Register Site Object with the ActiveScript Engine. }
  if ActiveScript.SetScriptSite(
       ActiveScriptSite) <> S_OK then
    Exit;

  { Initialize the engine. }
  if (ActiveScriptParse.InitNew <> S_OK) then
    Exit;

  { Now fire up the scripting engine. }
  if ActiveScript.SetScriptState(
       SCRIPTSTATE_CONNECTED) <> S_OK then
    Exit;

  { If we made it this far then we have succesfully
    initialized the engine! }
  Result := True;
end;
```

Figure 2: Starting a scripting session.

with enough detail to get you started. For more information, consult Microsoft's Web site at http://www.microsoft.com.

The scripting engine is made available through a series of COM interfaces. Although there are several optional interfaces that can be used in different scenarios, our application will interact with the following three:

- *IActiveScript*. This interface is the primary means of communicating from our application to the scripting engine. It provides a mechanism for starting and stopping the scripting engine.
- *IActiveScriptParse*. This is an interface to the parser that allows us to submit a script at run time.
- *IActiveScriptError*. An instance of this interface is returned to us by the scripting engine when an error occurs. We can use it to request more detailed error information.

For our demonstration application, the Activescp.pas file (see Listing One beginning on page 15) contains the Delphi wrapper for accessing the scripting engine's automation interfaces (see end of article for download details). With the drudgery of creating the wrapper out of the way, we can now focus on the details of those interfaces and methods.

Getting Started

To link our application to the scripting engine, we create a COM object of our own, known as the Script Site Object. You'll notice the interface declarations for this object in the Activescp.pas file as well, but unlike the scripting engine, we are responsible for implementing this object. The Script Site Object is responsible for providing the scripting engine with any available application objects and an interface for retrieving error information such as syntax and run-time script errors. For our demonstration, we'll be implementing two interfaces:

- *IActiveScriptSite*. This interface provides the primary method of communicating from the scripting engine to our application.
- *IActiveScriptSiteWindow*. This is a simple interface that passes the scripting engine a window handle from our application. By implementing this interface, any windows that are created by our script will appear to be a seamless part of our application.

As seen in Listing Two (beginning on page 16), the Scriptsite.pas file contains the implementation for this object. Although we are implementing all methods, we are really only adding processing for two methods:

- *OnScriptError*. This is triggered whenever a design- or run-time error occurs when processing a script. By implementing this method, we're able to provide users with meaningful information about an error, such as the line and column of a syntax error.
- *GetWindow*. This is triggered whenever the scripting engine needs to obtain a window handle.

Starting the Engine

As mentioned earlier, one of the great benefits of the ActiveX Scripting architecture is the ability to switch scripting engines without modifying your application code. You may be wondering how our application can tell the difference between the VBScript Engine interfaces and the JScript interfaces. After all, each interface has the same name and is defined by the same *globally unique identifier* (GUID). We are able to differentiate the various engines because the engine provider is required to group the interfaces together with a special type of GUID known as a *category identifier* (CatID). The scripting engine we integrate with is determined by which CatID we submit when instantiating the COM interfaces. This means we can easily swap scripting engines without changing a single line of code, even at run time.

You can now create a Delphi method to handle the steps required to start our scripting session (see Figure 2). We first create an instance of the scripting engine by calling *CreateComObject* and pass in the category identifier of the scripting engine we want to instantiate. When doing this, be sure the variables storing those interface pointers are global, since Delphi 3 will release them as they go out of scope. Once we have a handle to *IActiveScript*, we can easily query out the *IActiveScriptParse* interface. At this point, we instantiate our *IActiveScriptSite* object and pass it into the engine. This step bridges the two objects together for the

```
function TScriptingDemo.ParseScript(
  const ScriptText: WideString): Boolean;
var
  Ei: TExcepInfo;
  Flags: DWord;
  VarOut: OleVariant;
begin
  Flags := SCRIPTTEXT_NULL;
  Result := (activeScriptParse.ParseScriptText(
              ScriptText, nil, nil, nil,
              0, 0, Flags, VarOut, Ei) = S_OK);
end;
```

Figure 3: Submitting a script for parsing and execution.

```
procedure TScriptingDemo.StopEngine;
begin
  ActiveScript.SetScriptState(SCRIPTSTATE_UNINITIALIZED);
end
```

Figure 4: Reinitializing the scripting engine.

session. The last step is to call *InitNew*, then change the engine state to connected. We have now successfully fired up the scripting engine.

Parsing the Script

Now that we have a mechanism for starting the engine, we must have a way to submit our script for parsing and execution. This is accomplished through the use of the *IActiveScriptParse* interface and its *ParseScriptText* method. We can conveniently bundle this call into a Delphi method (see Figure 3).

Resetting the Engine

In many cases, you'll probably want the ability to submit a script, execute it, but then reset the engine so you can repeat the process with another script. This can be done by changing the script state from connected to uninitialized. The Delphi method shown in Figure 4 accomplishes this.

Firing a Script Subroutine

Fortunately, the scripting engine gives us the ability to selectively fire individual subroutines or functions in the script from our Delphi application. This is possible because the scripting engine exposes all subroutines through an automation interface at the time the script is parsed. We can now treat the methods in the script as if they were methods in any standard dual automation object.

A *dual automation object* is essentially an ordinary COM object. As with any COM object, it has a vtable that contains pointers to its methods and properties. The big difference is that it also contains a *dispatch interface* — or *dispinterface* — that assigns a unique integer identifier, referred to as a *dispatch identifier* (DispID), to each method and property. The second major difference is that a dual automation object inherits from an interface, called *IDispatch*, which provides a method for calling any other methods in the vtable based on its DispID.

The clear advantage of this mechanism is that we now have the ability to execute any routine in the script at run time.

```
procedure TScriptingDemo.FireMethod(
  const EventName: WideString);
var
  Disp: Integer;
  DispParams: TDispParams;
  Ei: TExcepInfo;
  InvKind: Integer;
  ReturnVal: POleVariant;
  ScriptDispatch: IDispatch;
begin

  if ActiveScript.GetScriptDispatch(
       nil, ScriptDispatch) <> S_OK then
    Exit;

  { Initialize dispatch id. }
  Disp := -1;

  { Get a dispatch id number that corresponds to
    the EventName name. }
  ScriptDispatch.getIDsOfNames(GUID_NULL, @eventName, 1,
                               LOCALE_USER_DEFAULT, @disp);

  { See if anything was found for the EventName --
    Spelling Error will fail! }
  if disp = -1 then
    begin
      ShowMessage(Format(
        'The method %s was not found in the script...',
        [EventName]));
      Exit;
    end;

  { Set the type of invocation to method. }
  InvKind := DISPATCH_METHOD;

  { This structure can contain up to 32 arguments to pass
    in, but we will not pass any in the demo. }
  ReturnVal := nil;
  DispParams.rgvarg := nil;
  DispParams.rgdispidNamedArgs := nil;
  DispParams.cArgs := 0;
  DispParams.cNamedArgs := 0;

  { Fire the Event via ID Binding. }
  ScriptDispatch.Invoke(Disp, GUID_NULL, 0, InvKind,
    DispParams, ReturnVal, @ei, nil);
end;
```

Figure 5: ID binding allows for generic code to call a routine in script.

We simply query the *IDispatch* interface for the DispID that corresponds to the name of a script routine, and then invoke that method based on the DispID. This technique is known as *ID binding*, and enables us to create generic code for calling a routine in the script (see Figure 5).

Exposing Application Objects to the Script

For the scripting engine to have true significance to an application, we must be able to expose an application's Automation objects to the script, so that users can easily access them without knowledge of COM. After all, one of the major advantages a scripting language provides is that users can control the logic of complex tasks without having to know all the details. For example, a script writer may add the following code to a script:

```
if CountDown = 0 then
  Rocket.Launch
end if
```

```
procedure TMyObj.ShowCelcius(DegreesF: Integer);
var
  DegreesC : Integer;
begin
  DegreesC := (DegreesF - 32) * 5 div 9;
  MessageDLG(Format(
    '%d degrees Fahrenheit is equal to %d degrees Celcius.',
    [DegreesF, DegreesC]) , mtInformation, [mbOK], 0 );
end;
```

Figure 6: The *ShowCelcius* method.

```
function TActiveScriptSite.GetItemInfo(
  ItemName: WideString; dwReturnMask: DWord;
  out UnkItem: IUnknown; out TypeInfo: ITypeInfo): HResult;
var
  ObjDispatch: IDispatch;
begin
  { This method is called when the engine wants information
    about an object that we submitted by calling
    AddNamedItem. Using Delphi 3, the instances created
    here will be freed when they go out of scope. }

  { Does the engine want the Automation object's IUnknown
    pointer? }
  if (dwReturnMask = SCRIPTINFO_IUNKNOWN) and
     (ItemName = 'MyObj') then
    UnkItem := CoMyObj.Create;

  { Does the engine want the Automation object's
    type information? }
  if (dwReturnMask = SCRIPTINFO_ITYPEINFO) and
     (ItemName = 'MyObj') then
    begin
      ObjDispatch := CoMyObj.Create;
      { Get a handle to our Automation object's
        type library. }
      ObjDispatch.GetTypeInfo(0,0,TypeInfo);
    end;

  Result := S_OK;
end;
```

Figure 7: The *GetItemInfo* method.

In this scenario, the three lines of VBScript handle the flow of control, but the Delphi *Rocket* object provides the intelligence and power to handle the details.

To demonstrate how you can expose the complex logic of an application to the scripting language, you can create a simple COM Automation object using Delphi 3. Although you might normally store your application objects in an ActiveX Library (.DLL), for simplicity, you can add it to the current project. The first step is to request a new ActiveX object by selecting File | New | ActiveX | Automation Object to invoke the Type Library editor. Add a new interface named *IMyObj*, with a method called *ShowCelcius*.

You can implement *ShowCelcius* as shown in Figure 6. This is a simple routine that converts Fahrenheit temperatures to Celcius. Be sure to add Dialogs and Demo_TLB to the **uses** statement, register the type library using the Type Library editor, and then register the new object with the OS.

We need to make two modifications to make our new object available to scripting applications. When you initialize the scripting engine, you can add the name of your object to the engine's namespace by adding the following to your *InitEngine* method before the connection is made:

```
Flags := SCRIPTITEM_ISVISIBLE;
ActiveScript.AddNamedItem('MyObj', Flags);
```

When a script is parsed, the engine will now be aware of an object called *MyObj*. If the script contains a reference to *MyObj*, the scripting engine will call the *GetItemInfo* method of our Script Site Object to obtain an instance of the actual object; add the code shown in Figure 7 to accomplish this.

Testing

Now lets look at ActiveX scripting in action. In our sample code editor, create a simple routine that references our COM object and calls a method in it:

```
Sub Main
  Dim ANumber
  ANumber = InputBox( _
              "Please enter a temperature (F) to convert")
  MyObj.ShowCelcius ANumber
End Sub
```

When you fire off this code in the editor, you'll be prompted for a Fahrenheit temperature; a message box will then appear with the appropriate conversion. Of course, you should try running the code with a syntax error to see how the engine reports it through the Script Site Object.

The last thing to try is to switch the sample application's scripting engine from VBScript to JScript. Because JScript has no built-in support for simple input and output statements such as the Visual Basic MsgBox and InputBox, we would have to use our custom Automation objects to collect input and display output to the user. For simplicity, we can type in the following code:

```
function Main() {
  MyObj.ShowCelcius(98);
}
```

Installing the sample. For the sample application to work on your machine, you must install the Visual Basic Scripting edition. The latest version of VB Scripting edition is available at http://www.microsoft.com/vbscript.

Conclusion

The addition of ActiveX scripting to your Delphi applications provides the ability to include powerful application objects in a familiar, easy-to-use, scripting language. Your users can then manipulate the flow of an application while leaving the complex details to you. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\FEB\DI9802TS.

Tom Stickle lives in Phoenix, AZ. He can be reached at (602) 598-1890, or via e-mail at tstix@dancris.com.

Begin Listing One — ACTIVESCP.PAS

```
unit Activscp;

interface

{ Interface specification for ActiveX Scripting. }
uses
  Windows, comobj, activeX;

{ IActiveScript.AddNamedItem input flags. }
const
  SCRIPTITEM_ISVISIBLE      = $00000002;
  SCRIPTITEM_ISSOURCE       = $00000004;
  SCRIPTITEM_GLOBALMEMBERS  = $00000008;
  SCRIPTITEM_ISPERSISTENT   = $00000040;
  SCRIPTITEM_CODEONLY       = $00000200;
  SCRIPTITEM_NOCODE         = $00000400;

  SCRIPTITEM_ALL_FLAGS = (SCRIPTITEM_ISSOURCE +
                          SCRIPTITEM_ISVISIBLE +
                          SCRIPTITEM_ISPERSISTENT +
                          SCRIPTITEM_GLOBALMEMBERS +
                          SCRIPTITEM_NOCODE +
                          SCRIPTITEM_CODEONLY );

{ IActiveScript.AddTypeLib() input flags. }
const
  SCRIPTTYPELIB_ISCONTROL     = $00000010;
  SCRIPTTYPELIB_ISPERSISTENT  = $00000040;
  SCRIPTTYPELIB_ALL_FLAGS     =
    (SCRIPTTYPELIB_ISCONTROL  + SCRIPTTYPELIB_ISPERSISTENT);

{ IActiveScriptParse.AddScriptlet() and
  IActiveScriptParse.ParseScriptText() input flags. }
const
  SCRIPTTEXT_NULL          = $00000000;  { Added for demo. }
  SCRIPTTEXT_ISVISIBLE     = $00000002;
  SCRIPTTEXT_ISEXPRESSION  = $00000020;
  SCRIPTTEXT_ISPERSISTENT  = $00000040;
  SCRIPTTEXT_ALL_FLAGS   = (SCRIPTTEXT_ISVISIBLE
                          + SCRIPTTEXT_ISEXPRESSION
                          + SCRIPTTEXT_ISPERSISTENT);

{ IActiveScriptSite.GetItemInfo() input flags. }
const
  SCRIPTINFO_IUNKNOWN   = $00000001;
  SCRIPTINFO_ITYPEINFO  = $00000002;
  SCRIPTINFO_ALL_FLAGS =
    (SCRIPTINFO_IUNKNOWN + SCRIPTINFO_ITYPEINFO);

{ IActiveScript.Interrupt() Flags. }
const
  SCRIPTINTERRUPT_DEBUG          = $00000001;
  SCRIPTINTERRUPT_RAISEEXCEPTION = $00000002;
  SCRIPTINTERRUPT_ALL_FLAGS =
    (SCRIPTINTERRUPT_DEBUG +
     SCRIPTINTERRUPT_RAISEEXCEPTION);

{ Script state enumerations. }
type
  SCRIPTSTATE = LongInt; { (0..5); }

const
  SCRIPTSTATE_UNINITIALIZED = SCRIPTSTATE(0);
  SCRIPTSTATE_INITIALIZED    = SCRIPTSTATE(5);
  SCRIPTSTATE_STARTED        = SCRIPTSTATE(1);
  SCRIPTSTATE_CONNECTED      = SCRIPTSTATE(2);
  SCRIPTSTATE_DISCONNECTED   = SCRIPTSTATE(3);
  SCRIPTSTATE_CLOSED         = SCRIPTSTATE(4);

{ Script thread state values. }
type
  SCRIPTTHREADSTATE = LongInt; { 0..1 }
const
  SCRIPTTHREADSTATE_NOTINSCRIPT = 0;
  SCRIPTTHREADSTATE_RUNNING = 1;

{ Thread IDs }
type
  SCRIPTTHREADID = DWORD;

const
  SCRIPTTHREADID_CURRENT  = SCRIPTTHREADID(-1);
  SCRIPTTHREADID_BASE     = SCRIPTTHREADID(-2);
  SCRIPTTHREADID_ALL      = SCRIPTTHREADID(-3);

{ GUIDs }
const
 { Category IDs }
 CATID_VBScript: TGUID =
   '{ B54F3741-5B07-11CF-A4B0-00AA004A55E8 }';
 CATID_JScript: TGUID =
   '{ F414C260-6AC0-11CF-B6D1-00AA00BBBB58 }';

 { Class IDs }
 IID_IActiveScriptParse: TGUID =
   '{ BB1A2AE2-A4F9-11CF-8F20-00805F2CD064 }';
 IID_IActiveScriptSite: TGUID =
   '{ DB01A1E3-A42B-11cf-8F20-00805F2CD064 }';
 IID_IActiveScriptSiteWindow: TGUID =
   '{ D10F6761-83E9-11cF-8F20-00805F2CD064 }';

 { String version of GUIDs }
 Class_IActiveScriptSite =
   '{ DB01A1E3-A42B-11cf-8F20-00805F2CD064 }';
 Class_IActiveScriptSiteWindow =
   '{ D10F6761-83E9-11cF-8F20-00805F2CD064 }';

type
  POleVariant = ^OleVariant;
  IActiveScript = interface;  { Forward declarations. }
  IActiveScriptParse = interface;
  IActiveScriptSite = interface;
  IActiveScriptSiteWindow = interface;

  { IActiveScript Interface - Methods in VTable order. }
  IActiveScript  = Interface(IUnknown)
    function SetScriptSite(ScriptSite: IActiveScriptSite):
      HResult; stdcall;
    function GetScriptSite(const iid: TIID; out vObj):
      HResult; stdcall;
    function SetScriptState(ScriptState: LongInt):
      HResult; stdcall;
    function GetScriptState(out ScriptState: LongInt):
      HResult; stdcall;
    function Close:HResult; stdcall;
    function AddNamedItem(ItemName: WideString;
      dwFlags: DWord): HResult; stdcall;
    function AddTypeLib(const GuidTypeLib: TGUID;
      wVerMajor, wVerMinor, wFlags: Word):
      HResult; stdcall;
    function GetScriptDispatch(StrItemName: Pointer;
      out ScriptDispatch: IDispatch): HResult; stdcall;
    function GetCurrentScriptThreadID(
      wScriptThreadID: Word): HResult; stdcall;
    function GetScriptThreadState(wScriptThreadID,
      wScriptState: Word): HResult; stdcall;
    function InterruptScriptThread(wScriptThreadID: Word;
      ExcepInfo: PExcepInfo; wFlags: Word):
      HResult; stdcall;
    function Clone(Script: iActiveScript): HResult;
stdcall;
    function GetScriptThreadID(wWin32Thread,
      wScriptThreadID: Word): HResult; stdcall;
  end;

  { IActiveScriptParse Interface - Methods in VTable order. }
  IActiveScriptParse  = interface(IUnknown)
    function InitNew: HResult; stdcall;
    function AddScriptlet(DefaultName, ScriptCode,
      ItemName, SubItemName, EventName, Delimiter:
      WideString; wSrcContextCookie: Word; StartLine:
      Integer; wFlags: Word; StrName: WideString;
```

```
      var ExcepInfo: TExcepInfo): Integer; stdcall;
    function ParseScriptText(MainScript: WideString;
      ItemName: Pointer; UnkContext: IUnknown;
      EndDelimiter: Pointer; dwSourceCookie: DWORD;
      StartLineNo: Integer; dwFlags: DWord;
      var VarOut: OleVariant; var ExcepInfo: TExcepInfo):
      HResult; stdcall;
  end;

  { IActiveScriptError Interface --
    Methods in VTable order. }
  IActiveScriptError = interface(IUnknown)
    function GetExceptionInfo(var excepInfo: TExcepInfo):
      HResult; stdcall;
    function GetSourcePosition(out wContextCookie: Word;
      out lineNo: UINT; out charPos: Integer):
      HResult; stdcall;
    function GetSourceLineText(wsSourceLine: WideString):
      HResult; stdcall;
  end;

  { IActiveScriptSite Interface --
    We are responsible for implementing this. }
  IActiveScriptSite = interface(IUnknown)
    [Class_IActiveScriptSite]
    function GetLCID(var wLCID: TLCID): HResult; stdcall;
    function GetItemInfo(StrName: WideString;
      dwReturnMask: DWord; out UnkItem: IUnknown;
      out TypeInfo: ITypeInfo): HResult; stdcall;
    function GetDocVersionString(var VersionString: TBSTR):
      HResult; stdcall;
    function OnScriptTerminate(var VarResult: OleVariant;
      var ExcepInfo: TExcepInfo): HResult; stdcall;
    function OnStateChange(ScriptState: LongInt):
      HResult; stdcall;
    function OnScriptError(pAse: IActiveScriptError):
      HResult; stdcall;
    function OnEnterScript: HResult; stdcall;
    function OnLeaveScript: HResult; stdcall;
  end;

  { IActiveScriptSiteWindow is aggregated
    into IActiveScriptSite. }
  IActiveScriptSiteWindow = interface(IUnknown)
    [Class_IActiveScriptSiteWindow]
    function GetWindow(var Hwnd: THandle):
      HResult; stdcall;
    function EnableModeless(FEnable: WordBool):
      HResult; stdcall;
  end;

implementation

end.
```

End Listing One

Begin Listing Two — SCRIPTSITE.PAS

```
{ This implements the required
  IActiveScriptSite Interface. }
unit ScriptSite;

interface
uses
  Windows, SysUtils, comobj, activeX, Activscp, Dialogs,
  Forms, ComServ, Demo_TLB;

type
  { TActiveScriptSite Declaration. }
  TActiveScriptSite = class(TComObject, IActiveScriptSite,
                            IActiveScriptSiteWindow)
  protected
    { IActiveScriptSite }
    function GetLCID(var wLCID: TLCID): HResult;
      virtual; stdcall;
    function GetDocVersionString(var VersionString: TBSTR):
      HResult;  virtual; stdcall;
    function GetItemInfo(ItemName: WideString;
      dwReturnMask: DWord; out UnkItem: IUnknown;
      out TypeInfo: ITypeInfo): HResult;  virtual; stdcall;
    function OnScriptTerminate(var VarResult: OleVariant;
      var ExcepInfo: TExcepInfo): HResult; virtual; stdcall;
    function OnStateChange(ScriptState: LongInt): HResult;
      virtual; stdcall;
    function OnScriptError(pAse: IActiveScriptError):
      HResult;  virtual; stdcall;
    function OnEnterScript: HResult; virtual; stdcall;
    function OnLeaveScript: HResult; virtual; stdcall;
    { IActiveScriptSiteWindow }
    function GetWindow(var Hwnd: THandle): HResult; stdcall;
    function EnableModeless(FEnable: WordBool):
      HResult; stdcall;
  end;

implementation

{ TActiveScriptSite - Protected Implementation. }
function TActiveScriptSite.GetLCID(var wLCID: TLCID):
HResult;
begin
  { No need for us to do anything here. }
  Result := S_OK;
end;

function TActiveScriptSite.GetItemInfo(
   ItemName: WideString; dwReturnMask: DWord;
   out UnkItem: IUnknown; out TypeInfo: ITypeInfo): HResult;
var
  ObjDispatch : IDispatch;
begin
  { This method is called when the engine wants information
    about an object that we submitted by calling
    AddNamedItem.  Using Delphi 3, the instances created
    here will free when they go out of context. }

  { Does the engine want the Automation object's
    IUnknown Pointer? }
  if (dwReturnMask = SCRIPTINFO_IUNKNOWN) and
     (ItemName = 'MyObj') then
    UnkItem := CoMyObj.Create;

  { Does the engine want the Automation object's type
    information? }
  if (dwReturnMask = SCRIPTINFO_ITYPEINFO) and
     (ItemName = 'MyObj') then
    begin
      ObjDispatch := CoMyObj.Create;
      { Get a handle to our Automation object's
        Type Library. }
      ObjDispatch.GetTypeInfo(0,0,TypeInfo);
    end;

  Result := S_OK;
end;

function TActiveScriptSite.GetDocVersionString(
  var VersionString: TBSTR): HResult;
begin
  { Tell engine that we will accept its default,
    i.e. not implemented. }
  Result := E_NOTIMPL;
end;

function TActiveScriptSite.OnScriptTerminate(
  var VarResult: OleVariant;
  var ExcepInfo: TExcepInfo): HResult;
begin
  { This tells us that the script is completed. }
  Result := S_OK;
end;

function TActiveScriptSite.OnStateChange(
  ScriptState: LongInt): HResult;
begin
  { Alerts us when engine states are changing. }
  Result := S_OK;
```

```
end;

function TActiveScriptSite.OnScriptError(
  pAse: IActiveScriptError): HResult;
var
  wCookie: Word;
  ErrString: string;
  ExcepInfo: TExcepInfo;
  CharNo: LongInt;
  LineNo: LongInt;
begin

  wCookie := 0;
  LineNo  := 0;
  CharNo  := 0;

  if Assigned(pAse) then
    begin
      pAse.GetExceptionInfo(ExcepInfo);
      pAse.GetSourcePosition(wCookie, LineNo, CharNo);

      ErrString := concat(ExcepInfo.bstrSource, ' ',
                          ExcepInfo.bstrDescription, ' ',
                          'Line ', intToStr(LineNo),
                          '. Column ', intToStr(CharNo));
      ShowMessage(ErrString);
    end;

  pAse := nil;
  result := E_FAIL;  { Halt script execution! }
end;

function TActiveScriptSite.OnEnterScript(): HResult;
begin
  Result := S_OK;
end;

function TActiveScriptSite.OnLeaveScript(): HResult;
begin
  Result := S_OK;
end;

{ IActiveScriptSite Window Implementation. }
function TActiveScriptSite.GetWindow(
  var Hwnd: THandle): HResult;
begin
  { ActiveX Scripting uses this to get a window handle from
    our application. This allows the script engine to
    display information on the interface, such as a
    dialog box. }
  Hwnd := Application.Handle;
  Result := S_OK;
end;

function TActiveScriptSite.EnableModeless(
  FEnable: WordBool): HResult;
begin
  { Causes the host to enable or disable its main window
    as well as any modeless dialog boxes. We won't need
    this for our demo. }
  Result := S_OK;
end;

initialization
  TComObjectFactory.Create(ComServer, TActiveScriptSite,
    IID_IActiveScriptSite, 'ActiveScript Host', '',
    ciMultiInstance);
end.
```

End Listing Two

By Cary Jensen, Ph.D.

Insightful Delphi

Delphi 3's Code Insight Feature

One of the most powerful features of Delphi 3's editor is virtually unknown. This feature, called Argument Value Lists, is part of Code Insight. This month's "DBNavigator" provides an introduction to Code Insight, including the largely undocumented Argument Value Lists.

Code Insight is a powerful new feature of the Delphi editor that provides additional online help for the various declarations visible from the unit you are editing. Code Insight is possible because Delphi is continuously parsing your code in the background while you work. So long as the statements you are entering can be parsed (and are more or less syntactically correct), Code Insight works to analyze your code, as well as that in any unit you are using. In fact, Code Insight doesn't even require the source code (the .PAS file) — the .DCU file alone provides enough information. This information is used to generate and display popup lists and hints, either when you pause for a moment (as with Code Completion), or when you specifically request it using shortcut keystrokes.

There are five basic features of Code Insight:

1) Code Completion
2) Code Parameters
3) Argument Value Lists
4) Code Templates
5) Tooltip Expression Evaluation

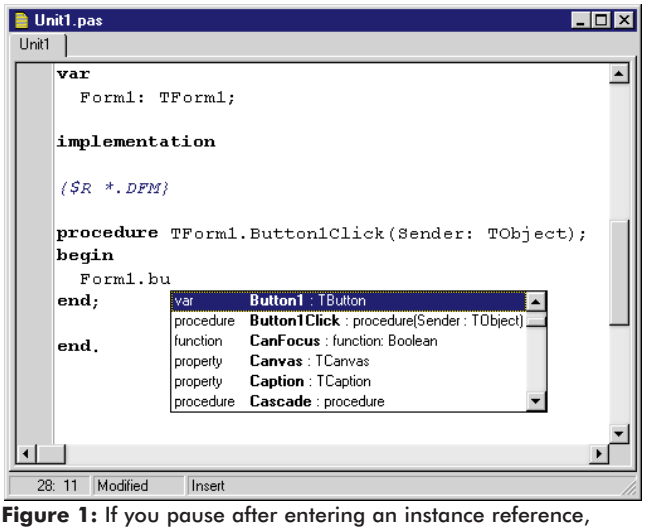


Figure 1: If you pause after entering an instance reference, Code Completion displays the members that are visible from the reference.

Code Completion

Code Completion is a feature that generates and displays a list of the visible members of an instance or class reference. There are two advantages of Code Completion. First, it reminds you which members (that is, fields, properties, and methods) are accessible from your reference. Second, it permits you to select a member from this list to have the member inserted automatically into the editor, rather than requiring you to type the entire reference.

If you've used Delphi 3, you have no doubt encountered Code Completion. For example, if you select File | New Application, add a button to the default form, add an event handler to this button, then type a reference to an instance of the form followed by a dot (the member reference operator), Code Completion displays the list shown in Figure 1.

By comparison, if you enter a reference to the *TForm1* class, rather than an instance of the class, a slightly different list is displayed. This list, shown in Figure 2, displays those members that are visible from a reference to the class itself.

As mentioned earlier, the list displayed by Code Completion respects the visibility of the declared members of the class. For example, within a unit where the class *TForm1* is declared, the list generated for an instance of

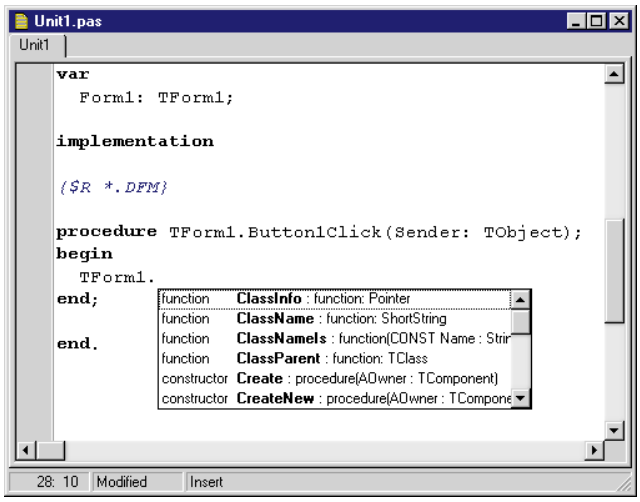


Figure 2: Code Completion can also display the members visible from a class reference.

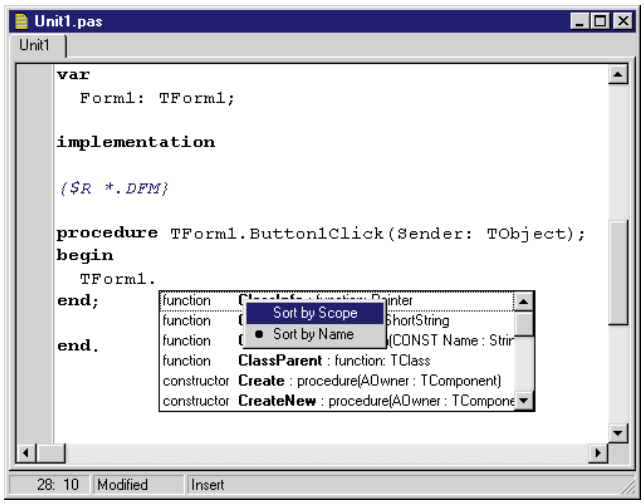


Figure 3: Right-click the list displayed by Code Completion to change its sort order.

the *TForm1* class includes private declarations, because private members are accessible within that unit. However, if the *TForm1* class is declared in another unit (and that unit appears in an appropriate **uses** clause for the unit you are editing), Code Completion will display only the public and published members of a *TForm1* instance. Protected members are only displayed when you are editing a method associated with the class within which they are declared, or a method in a class that descends from the one in which they are declared. Again, this corresponds to member visibility.

When you install Delphi 3, this list is sorted alphabetically by default. You can, however, change the sort order by right-clicking the list and selecting **Sort by Scope**, as shown in Figure 3. You can return to an alphabetically-sorted list by right-clicking and selecting **Sort by Name**.

Once the Code Completion list is displayed, you can have Code Insight enter one of the listed members at your cursor by first highlighting the desired member and then pressing Enter. There are two ways to highlight a member. One way is to search incrementally. Specifically, begin typing the name of the member you want. As you type, Code Completion will move through the list, highlighting the member whose name most closely matches what you typed. The second technique is to use your cursor keys, ↑, ↓, Home, End, etc. You can use these techniques concurrently. For example, you can begin by using an incremental search to move to the general vicinity of a member in the list, then use the cursor keys to select it. For obvious reasons, incremental searching is usually only effective when the list is sorted alphabetically.

You can cancel the list displayed by Code Completion by pressing Esc. Furthermore, if you have an instance or object reference and the Code Completion list is not currently displayed, you can force its display by pressing Ctrl Space.

Code Parameters

The Code Parameters feature of Code Insight provides on-the-fly syntax display for the arguments of methods, procedures, and functions that are visible from your unit. For example, imagine you are typing the following into a unit that is using the Dialogs unit:

```
if MessageDlg(
```

Shortly after typing the open parenthesis, Code Parameters will display the syntax of the parameters of this function, as shown in Figure 4. The parameter you are currently entering is displayed in bold. If there is more than one parameter, the bold type face in the Code Parameters window advances to the next parameter as you complete each one.

While Code Parameters, like Code Completion, is displayed automatically, the Help window will be removed if you move your cursor to another line of code. If this happens, you can re-display the Help window by moving your cursor back to the argument list and pressing Ctrl ⇧ Shift Space.

Argument Value Lists

Argument value lists can be generated when you are entering an expression, such as the actual parameter in an argument list or the expression on the right side of an assignment statement. Unlike Code Completion and Code Parameters, the

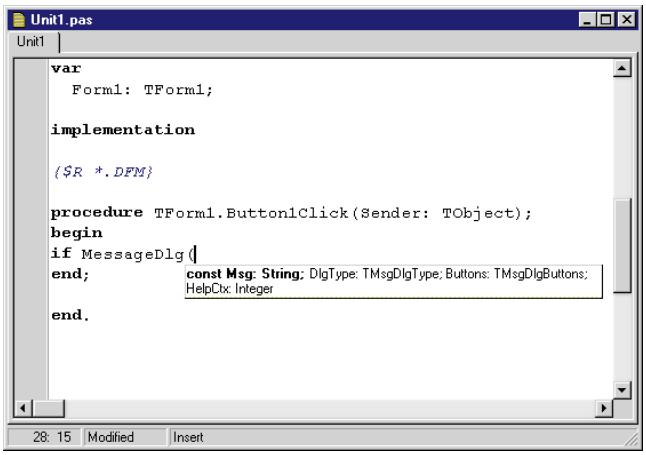


Figure 4: Code Parameters displays the syntax of the argument list for a function, procedure, or method.

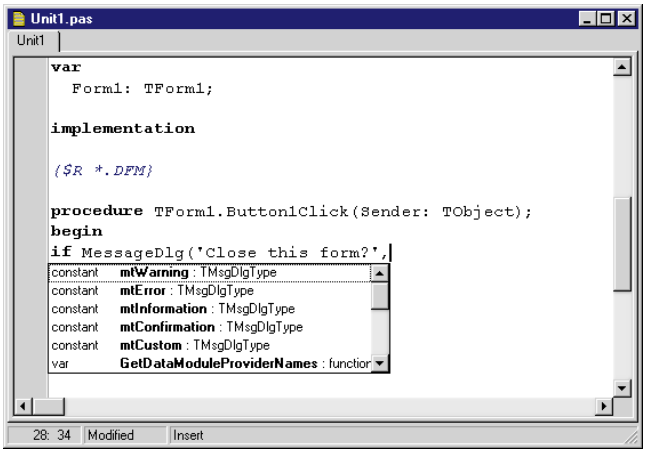


Figure 5: Press Ctrl Space to see the list of expressions valid in the current context.

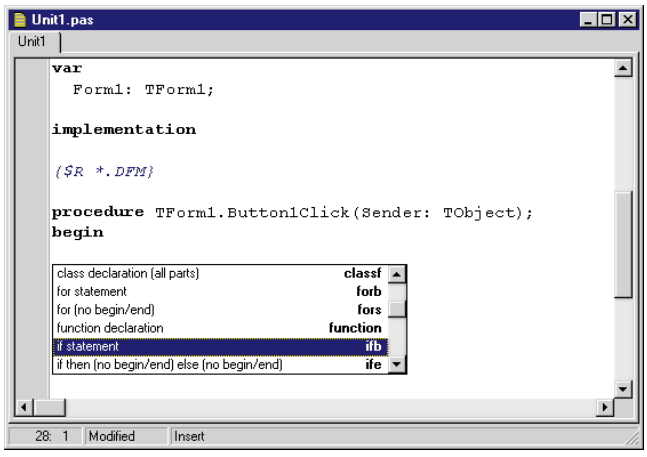


Figure 6: Press Ctrl J in the editor to display defined code templates.

Argument Value Lists feature must be specifically requested by pressing Ctrl Space. The list generated displays the constants, functions, and variables that are consistent with the argument required by the expression.

I think this is the most powerful feature available in Code Insight. Unfortunately — and ironically — it is unknown to most Delphi developers. In fact, the section of the *User's Guide* that ships with Delphi 3 fails to even mention Argument Value Lists. I even attempted to locate some mention of it in the online Help while writing this article, but was unsuccessful. I know it must be in the online Help somewhere, because that's how I learned of this feature (shortly before Delphi 3 shipped). However, I am unable to locate a reference now.

In any case, it *is* a Delphi 3 feature, and a powerful one at that. For example, imagine that while you are entering the function *MessageDlg* you cannot recall which values are acceptable for the second argument, *DlgType*, which is of the type *TMsgDlgType*. Such a situation is perfect for Argument Value Lists. With your cursor poised to enter the second argument, press Ctrl Space. When you do, Code Insight generates and displays a list of the symbols that are visible to your unit that match the data type of the required expression. The list generated for the second argument of the *MessageDlg* function is shown in Figure 5.

As with Code Completion, while the argument value list is displayed you can begin typing the name of the symbol you want, use your cursor keys to navigate the list, or both. Once the symbol you want entered at your cursor is highlighted, press Enter to have Code Insight enter the value.

Also similar to Code Completion, you can sort argument value lists alphabetically or by scope. In most cases it's best to have this list sorted by scope, because that will place all expressions of the matching type as required by the expression at the top of the list. This is useful because the list not only includes symbols that match your expression exactly, but also includes any variants that are visible from your unit (because variants are assignment-compatible with a wide range of expression types). To change the sort order of the argument value list, right-click the list and select the sort order you want.

Code Templates

A code template is a pre-defined snippet of code that can be inserted into the editor at your request. For example, if you want to enter an **if** statement, position your cursor at the location where you want the code template to be entered, then press Ctrl J. Code Insight displays a list of the existing code templates, as shown in Figure 6.

As with the argument value list, you can navigate this list of code templates by using ↑, ↓, Home, End, etc., or by conducting an incremental, alphabetical search of the entries. Once the template you want is highlighted, press Enter to select it. Code Insight responds by entering the code, and placing your cursor within it.

The code template list that appears when you press Ctrl J has two columns. The first column contains the template description; the second column contains a shortcut, or mnemonic, for the template. The shortcut contains no spaces, and must be unique for each template. For example, the shortcut for one of the **if** templates is **ifeb**. Once this list of templates is displayed, you can perform an incremental search on the shortcut, use your cursor keys, or both. After you've highlighted the entry of the template you want, press Enter to have Code Insight enter the template at the position of your cursor.

If you know the shortcut associated with a particular code template, you can access the template directly by typing the shortcut, then pressing Ctrl J. If the characters of the shortcut you have typed are unique to that shortcut, the code template is entered without the template list being displayed.

If two or more shortcuts share the same characters as the one you entered, a short list of only those templates whose shortcut names match what you have entered is displayed. For example, if you enter `if`, then press Ctrl J, the code template list contains all templates whose shortcuts begin with "if", as shown in Figure 7.

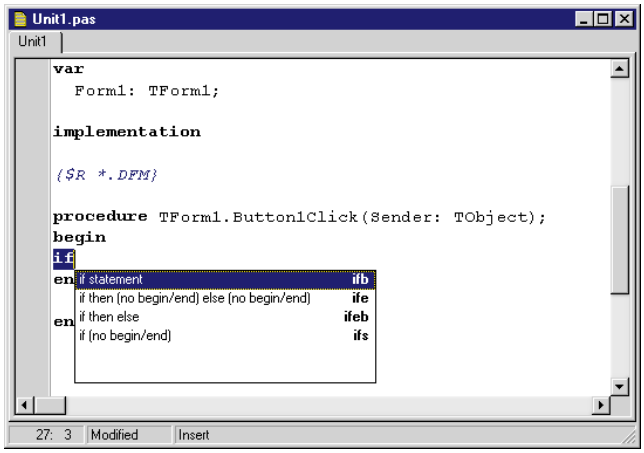


Figure 7: If you enter only part of a shortcut name, and that part matches two or more templates, Code Insight displays a short list of the matching template shortcut names from which you can choose.

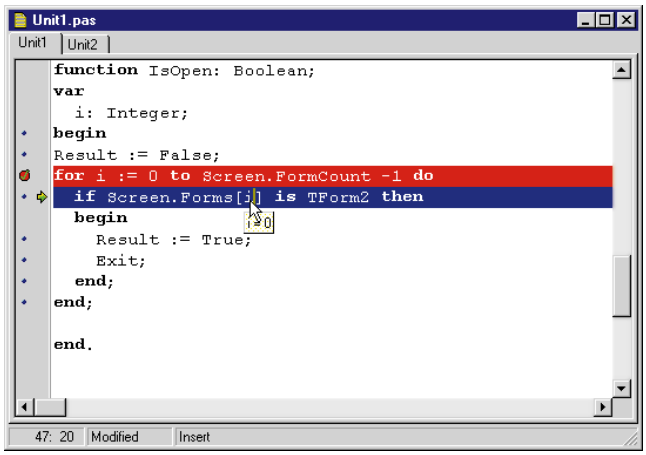


Figure 9: Tooltip Expression Evaluation permits you to inspect the value of variables and properties at run time, without using **Run | Evaluate/Modify** or setting watches.

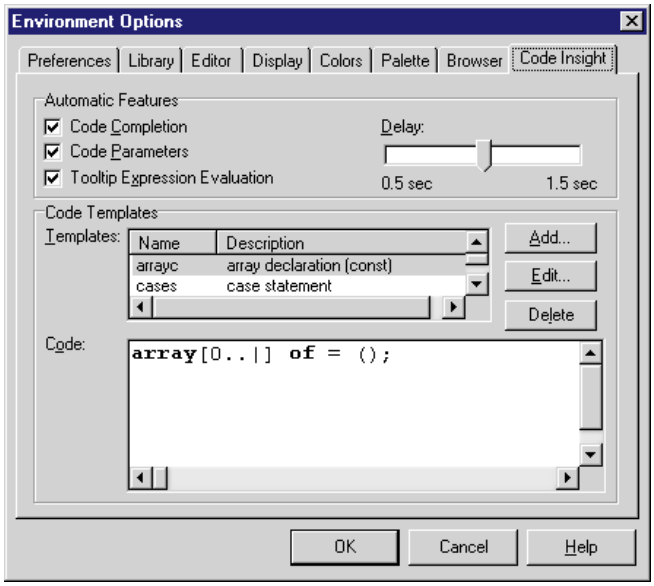


Figure 8: The Code Insight page of the Environment Options dialog box.

Creating Your Own Templates

While Delphi ships with a number of useful code templates, you can easily add custom code templates. To do so, select **Tools | Environment Options** from Delphi's main menu. Next, select the Code Insight page of the Environment Options dialog box, as shown in Figure 8.

To add a new template, select **Add**. In the Add Code Template dialog box enter a shortcut name and a description. Remember that the shortcut name must be unique. Also keep in mind that the incremental search of the template list is based on the shortcut name. For example, click **Add** and enter the shortcut name `repeat`, followed by the description `repeat statement`. Press Enter to continue. Now, within the Code field of the dialog box, enter the following:

```
repeat
  |
until expression;
```

Notice that the vertical bar (|) was placed where one or more statements must be entered. The vertical bar character defines where you want the cursor placed after the template has been entered. If you now enter `repeat` in the code editor and press Ctrl J, the entire template will be entered, and your cursor will appear on the line following the keyword **repeat**.

You can also easily modify existing code templates. To change the name or description of a template, select the template in the template list and select **Edit**. To change the text of the template itself, select the template in the Templates list, then simply modify the displayed code in the Code field.

The template text is written to an ASCII file named delphi32.dci, which is stored in Delphi's \Bin folder. If you find that editing, adding, or deleting a template using the Code Insight page of the Environment Options dialog box is awkward, you can edit this ASCII file directly.

Tooltip Expression Evaluation

Tooltip Expression Evaluation is a feature of Delphi's integrated debugger. It permits you to easily inspect the value of expressions at run time without having to resort to using **Run | Evaluate/Modify**, or setting watches. Instead, all you need to do is position your mouse over an expression, and the value of that expression will be displayed in a fly-by window, as shown in Figure 9.

Sometimes the value of an expression is unavailable because of compiler optimizations. If this is the case, the fly-by window will indicate the value is unavailable. You can reduce the likelihood of this happening with Tooltip Expression Evaluation by un-checking the Optimization checkbox on the Compiler page of the Project Options dialog box. However, remember to turn compiler optimizations back on before testing and delivering your project.

Controlling Code Insight

The features of Code Insight are controlled from the Code Insight page of the Environment Options dialog box. Code Completion, Code Parameters, and Tooltip Expression

Evaluation are enabled and disabled using the checkboxes at the top of this dialog box. If you disable either Code Completion or Code Parameters, you can still access these Code Insight features by pressing Ctrl Space and Ctrl Shift Space, respectively. However, I know of no way to access Tooltip Expression Evaluation if you disable this automatic feature. The **Delay** trackbar permits you to control how long Code Insight waits before automatically displaying the enabled Code Insight features.

Conclusion

Code Insight is a valuable new productivity tool for Delphi 3 developers. It can significantly reduce keystrokes, and the time you spend in Delphi's online Help. Δ

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

By Adam Chace

What's in the Package?

Design-time and Run-time Packages in Delphi 3

Packages offer Delphi developers a new way to deploy applications with run-time libraries. Forget the hassles of trying to incorporate VCL objects into standard DLLs; with Delphi 3's new packages feature, deploying shared libraries of Delphi units and components is a cinch, and you don't have to change a single line of code. This article will provide an overview of what packages are, and how to use them.

Like standard DLLs, packages are essentially just bundles of code that are referenced by a given application, be it an .EXE, an ActiveX control, a .DLL, or even another package. Borland has given packages a .DPL extension to differentiate them from common DLLs, but their architecture is almost identical to any implicitly linked DLL. These libraries are used at design time by the IDE, and can then be incorporated at run time by your application.

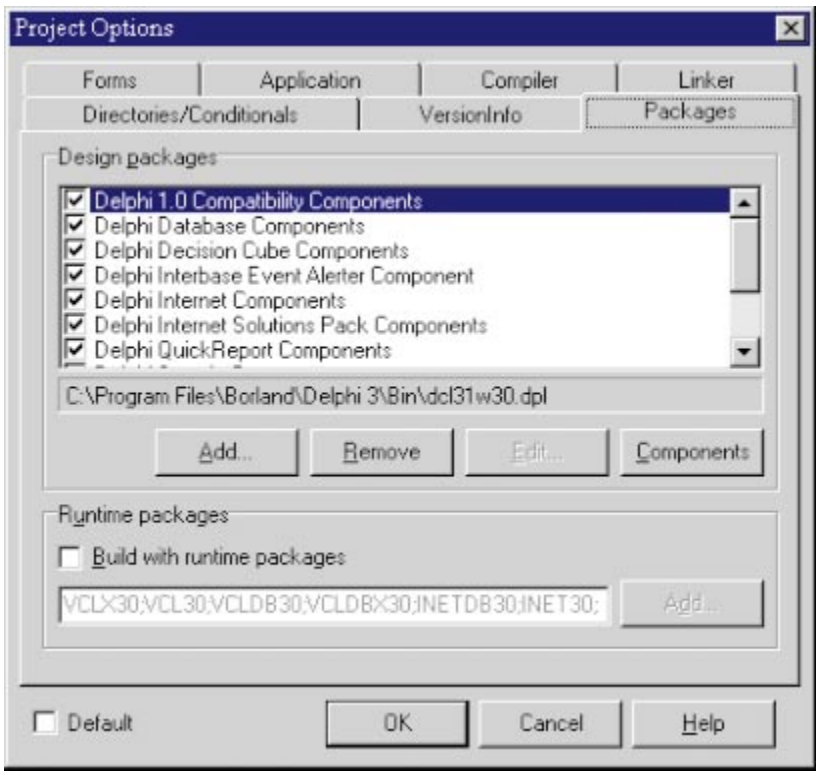


Figure 1: Adding and removing design-time packages.

The .DPL file isn't the only file associated with a package, however. There's also the .DCP file, a single file collection of all the DCUs your package contains; and the .DPK file, the editor file that specifies which .PAS files a particular package contains, and which .DPL files a package requires. The .DCP file is only of interest if a package is distributed without source code or the associated DCUs. In these cases, the .DCP file is used when you compile your package. The .DPK file is used any time you edit a package, as we'll soon see, and is modified every time you add or remove items to or from a package.

The New Component Palette Model

Although you may not know it, if you've developed in Delphi 3, you're already using this new feature. That's because the VCL model has been replaced with a new one, based entirely on design-time packages. Design-time packages are the new way of installing and removing components in Delphi. The Delphi 3 Component palette is made up entirely of design-time packages, which in turn, are collections of Delphi components and units. To configure the palette, select **Project | Options** and click on the Packages tab to display the dialog box shown in Figure 1.

From this dialog box, you can load and unload packages of components without having to recompile your entire Component palette (unlike previous Delphi versions). This is a major time saver, especially if you're doing a lot of switching between projects. You can also get a quick snapshot of what components a particular package contains by clicking on the **Components** button. Adding or removing packages is simply a matter of selecting the .DPL file you want by checking or unchecking the box, and clicking **Add** or **Remove**. Once you add a new package, Delphi displays the message box shown in Figure 2, telling you whether the package was successfully installed, and if so, what components were added as a result.

From this dialog box, we can also edit any packages for which we have the .DPK file. If we highlight a package and click **Edit**, we'll see a dialog box that allows us to specify which components and units this package contains, as well as any other packages it requires (see Figure 3). This form also provides access to the package options (which allow us to specify whether this package is a run-time or a design-time package, or both), the package description, compiler information, directory information, and other pertinent details about this particular package.

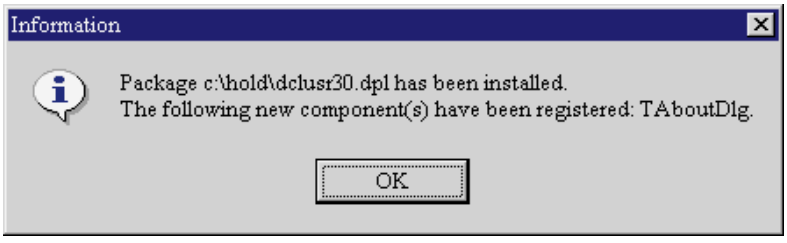


Figure 2: Delphi tells what components the package has installed.

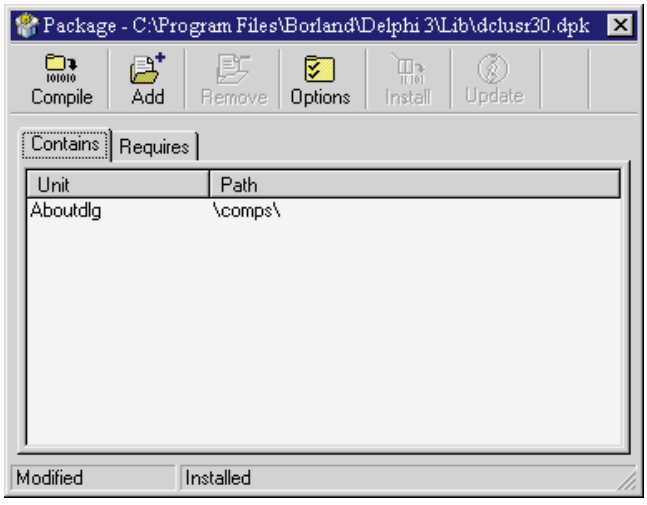


Figure 3: Editing package contents.

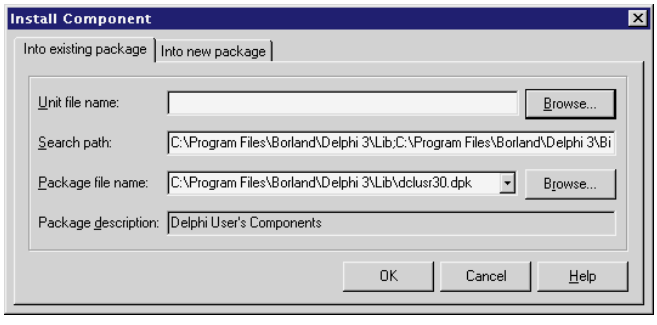


Figure 4: Installing "loose" components.

Using Non-package-based Components in Delphi 3

Soon, most of your third-party components will be distributed in the form of design-time packages. But what about your old components, or newer components that aren't in packages? Can you still use these in Delphi 3?

The answer, of course, is yes. Borland has provided an empty design-time package named DCLUSR30.DPL. Any non-package-based component can be placed in it. To place a component in this package, select **Components | Install** to display the dialog box shown in Figure 4. All you need to do is select your .PAS or .DCU file, and you're done. As long as there are no errors in the code, the component will install automatically as the package is loaded onto the palette.

If you are adding DCUs this way, make sure they were compiled in Delphi 3, because the DCU architecture has changed. If you don't have the source code for a component that wasn't compiled in Delphi 3, then you'll be unable to install the component into a package.

Running with Packages

That about covers design-time packages, which, for the most part, were created to make configuring the palette quicker and easier. Now, we get into the more interesting topic of run-time packages. Run-time packages are typically the same .DPL file as their design-time counterpart, but are used in deployment rather than development. With a typical Delphi application, all the necessary code for components and units used in the application is compiled directly into the executable, but this can produce a large executable that can be tedious to update, especially if it is being done via modem or the Internet. Building your project with run-time packages allows you to reduce your update size at the cost of a larger initial deployment. The initial delivery must include your using application (.EXE, .DLL, etc.) and all the required run-time packages. Once these packages are delivered, only the using application needs to be updated, as long as no code in the library files is changed.

Let's see an example of the impact of using run-time packages. The application, PACK.EXE, was built as a single form with a Label, Edit, DataSource, Table, ClientSocket, and DBListBox component on it. Each of these components is contained in a design-time package, and has a corresponding run-time package (which again, is usually just the same file). If we were to compile this application conventionally, we would see it has a file size of 392,192 bytes (see Figure 5).

Now, we want to indicate that we will be distributing this application not as a single executable, but rather as an executable with run-time packages. To do so, we select **Project | Options**

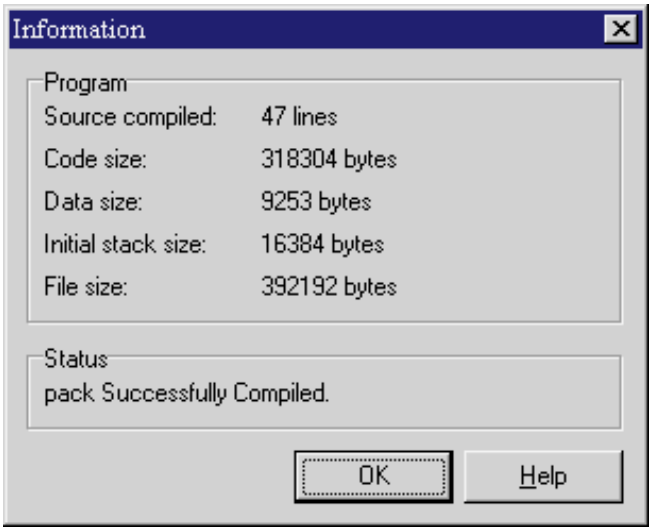


Figure 5: Our application compiled without using packages.

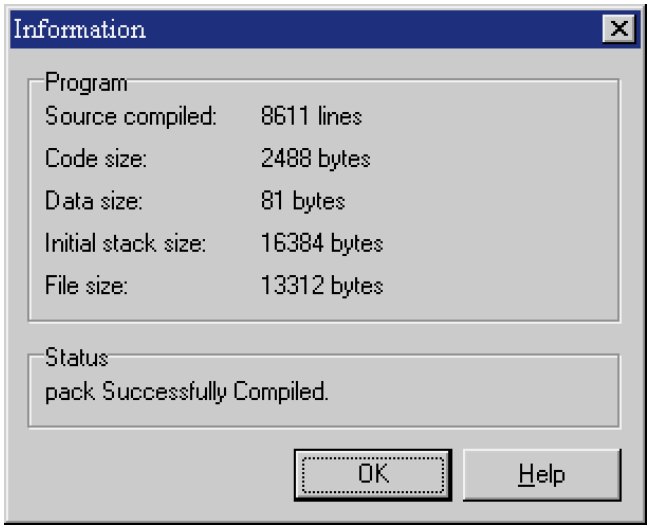


Figure 7: The same application compiled with run-time packages.

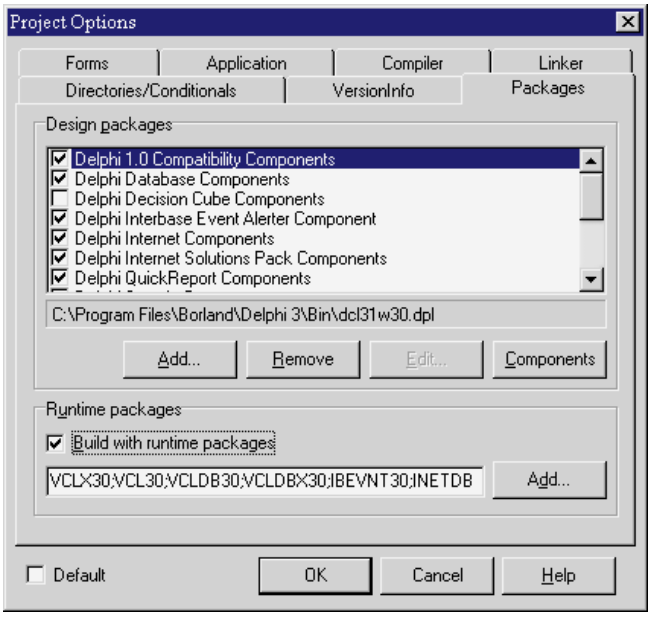


Figure 6: Choosing to build with packages.

and click the **Build with runtime packages** checkbox (see Figure 6). We can now, if desired, add or remove any run-time packages from this list.

Why would we want to remove a package name from this list at this point? Say, for instance, we're using a single function in a single unit included in a large package. We may want to simply compile the code into the executable, rather than deploy a sizable file for only one function. We may also know that we will be updating a particular component a great deal for this application. Because we ideally want components and code in packages to be static, we would avoid having to constantly update this .DPL file and the executable by removing the run-time package that it's contained in from this list.

Once you've decided whether you want to remove any files from this list, click **OK**, and build the application. It's important to note that although a run-time package appears in the list, it will only be required by the executable if it uses code contained in that package. It isn't necessary to go through this list and remove those that you don't think you need. The compiler takes care of referencing the packages used for you. So, once we click **Build**, we see the application is now only 13,312 bytes (see Figure 7)!

The next step is to determine which run-time packages need to be distributed with this application. We can do this in several ways:

- Determine which design-time packages are used by your application, and create a list of their corresponding run-time packages.
- Examine the application with an editor like Windows Quick View, and note the .DPL files that are referenced in the file.
- If it's an .EXE or a .DLL, use a freeware tool like PFinder to obtain the list.
- Because we have built a simple Windows executable, we can use PFinder to create the list, as shown in Figure 8. (This free utility is available from Apogee Information Systems, Inc., and can be downloaded from http://www.apogeeis.com/delphi.)

Now, we simply need to deploy VCL30.DPL, VCLDB30.DPL, and INET30.DPL once, with our executable. After that, as the project changes, we can update our 13KB executable. This is especially helpful during the testing phase of a project, where you may be distributing new builds of an application daily.

Another benefit of using run-time packages is that a user-machine, like standard DLLs, only needs one copy of any one package. So, if a project will be distributed as a suite of using applications, you can save a lot of space by only deploying the components they share once in a run-time package, rather than redundantly compiling the component code into each using program.

That's all there is to it. Run-time packages are an easy way to reduce your deployment costs, and you don't need to make

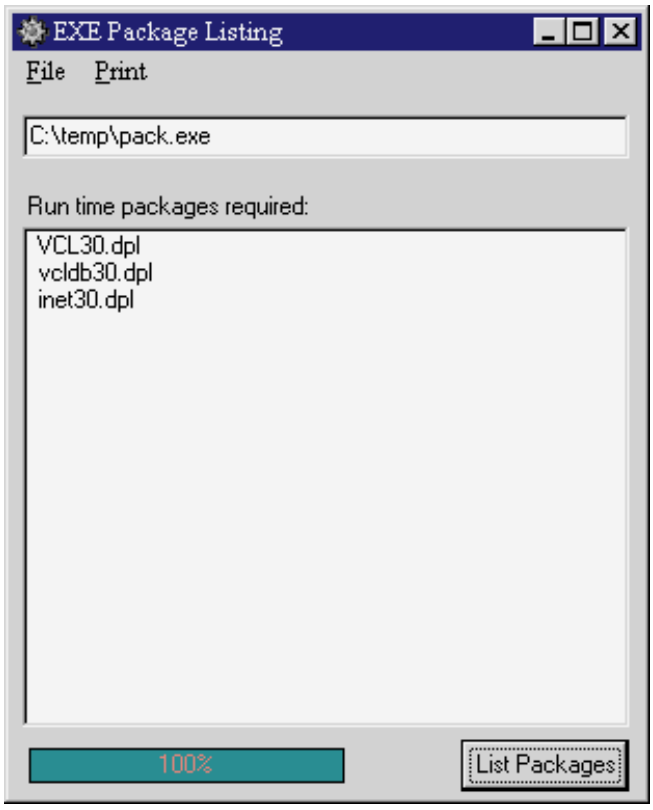


Figure 8: PFinder package-listing utility.

any coding changes. In fact, you can decide to deploy with run-time packages as late as your very last build, without having to be concerned with the issues of conventional DLLs.

Conclusion

As we have seen, packages are an exciting new feature for Delphi programmers to exploit. While design-time packages streamline the use of the component library, run-time packages provide the real power. Delphi programmers can easily segment and distribute portions of functionality, independent of the .EXE file. This significantly reduces the amount of effort involved with distributing application updates. Δ

Adam Chace is an Application Developer with Apogee Information Systems, Inc., a Boston-based consulting firm specializing in Delphi Client/Server systems. He presented "Packages in Delphi 3" at the 8th annual Borland Developers Conference in Nashville, TN and is a Delphi 3 Client/Server certified developer. You can reach him at achace@apogeeis.com.

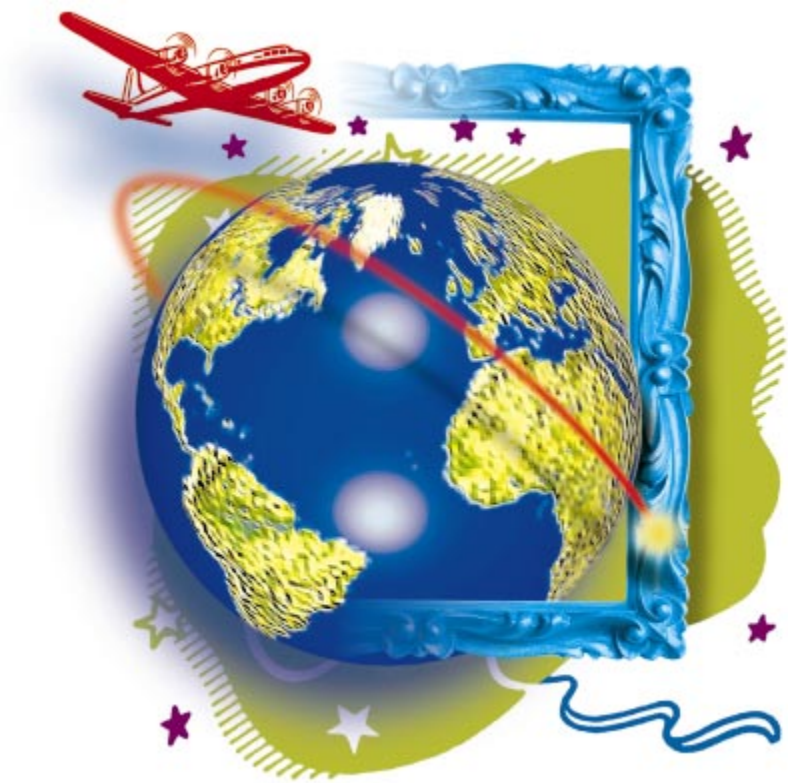
By Keith Wood

Picture This on the Web

A CGI Program to Deliver Database Pictures

Delphi 3 makes the creation of Internet-capable programs even easier than previous versions of Delphi. It contains basic classes that provide the necessary functionality to produce applications that run as ISAPI or NSAPI extensions, or as Common Gateway Interface (CGI) or Win-CGI executables.

This article explores the requirements for generating a CGI program with Delphi 3. It looks at the new Web module components, and how they interact with the HyperText Transfer Protocol (HTTP) to create a document in response to a request. To show how this works, we'll build a CGI program that delivers pictures from a database.



Delphi's Web Components

Delphi 3 Client/Server Suite provides several new components designed to ease the process of creating Web-aware programs. These components, defined in the HTTPApp, ISAPIApp, CGIApp, and DbWeb units, are described here. Their relationship to each other is shown in Figure 1.

In a Web program, *TWebApplication* replaces the normal application object. It must be subclassed to handle the different versions of Web extensions available. When its *Run* method is called, it creates wrappers for the incoming request and outgoing response, before searching for a Web dispatcher to translate between the two.

TWebModule is derived from *TDataModule* via *TCustomWebDispatcher,* and so provides a visual work area on which to develop our application. On this form, we can add other Web components and/or database-access controls. Built into this component is the ability to dispatch Web requests to different actions, based on the additional path information that accompanies the request. These are dealt with by the Web action items.

A *TWebActionItem* is invoked by a *TWebModule* when its particular path information is received with a request. Having multiple action items in our program allows us to respond differently to requests that have some common basis. For each action, we can specify the type of HTTP

request to respond to, the extra path to look for, and whether it's the default for the module. Finally, we supply a handler for the *OnAction* event to actually construct the response.

TWebRequest encapsulates an HTTP request that is received and processed by our program. It provides access to the details about that request, including any parameters that are sent by the client. This component is subclassed to allow the different types of Web extensions to retrieve the data in the appropriate way.

TWebResponse allows us to easily send a reply. It has methods that accept the document we generate, and deliver it back through the Web server to the client. Input for the document can be supplied as text, as a redirection, or from a stream. Again, this component is subclassed to deal with the differing ways of handling Web requests.

TPageProducer handles the generation of an HTML document in an easy-to-use manner. It contains the text of the document internally, or the name of a file that provides this text. References to substitution tags, delimited by the usual angle brackets and starting with a pound sign (#), can be embedded in the document. Each tag has an identifier associated with it, and these are used in the *OnHTMLTag* event to provide the appropriate replacement text.

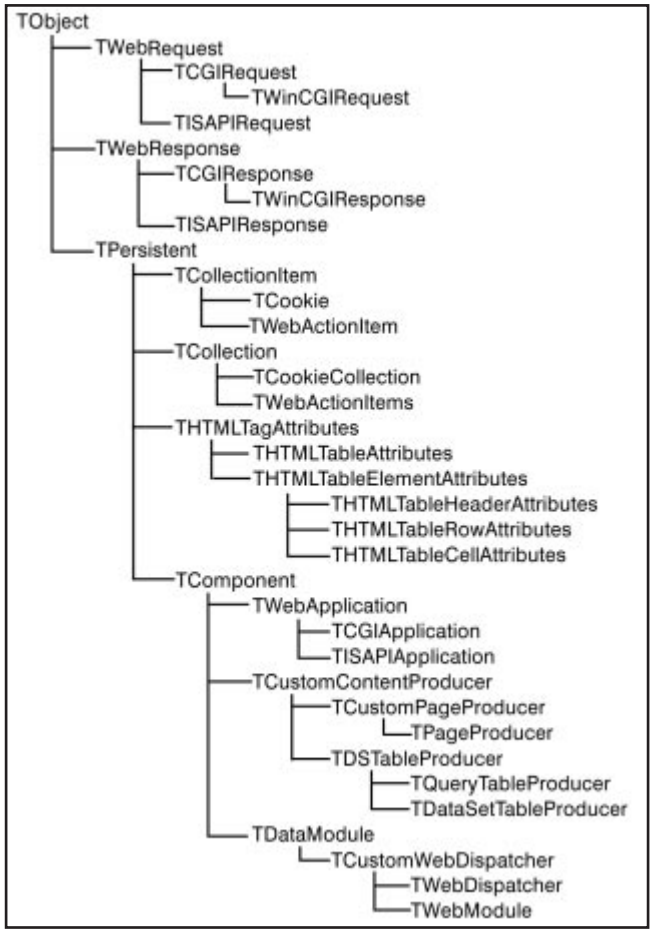


Figure 1: The object model of new Web components.

```
<HTML>
<HEAD>
<TITLE>Sample Page</TITLE>
</HEAD>
<BODY>
<H2>Sample Page</H2>
<P>This page displays a single image below:</P>
<IMG SRC="/images/Athena.jpg">
</BODY>
</HTML>
```

Figure 2: Sample HTML referencing an image.

TQueryTableProducer and *TDataSetTableProducer* provide an easy way to generate an HTML table from the contents of a query or table attached to a database. Its properties allow header and footer text to be specified, as well as control the appearance of the HTML table and its columns and rows. For specialized formatting, the *OnFormatCell* event is used to manipulate the cell contents even further, such as including a hypertext link.

Using object-oriented techniques, Delphi 3 allows us to build a single Web-response program, and have it work with all the standard extension formats: ISAPI/NSAPI, CGI, and WinCGI. All we need to do is change the Web module wrapper and recompile.

Let's put all of this together to produce a CGI application that extracts a picture from a database record, and returns it to the Web browser client.

Web Graphics

HTML documents define what appears on Web pages. It consists of straight text, interspersed with formatting and metadata commands (tags), which are delimited by angle brackets (< >). These tags allow the specification of headings and character and paragraph formatting, as well as define the links that make the Web what it is.

Another tag allows images to be included on the page. The graphics that are displayed this way are usually in one of two formats (.GIF or .JPEG), but the images themselves are not contained in the HTML document. The tag simply has a reference to another Web document that provides the content. See Figure 2 for an example of an HTML page that displays an image.

When the browser receives a document containing one of these tags, it must send another request to retrieve the picture. To distinguish between the different document types available on the Web, MIME (Multipurpose Internet Mail Extensions) encoding is used. This includes a two-part type to describe the contents of a document. Normal Web pages have the type text/html, while the two standard image formats are image/gif and image/jpeg, respectively. Web servers can be set up to automatically supply this content encoding as part of the response to a client, based on the extension of the file requested.

Usually, these images are stored in separate files on the Web server. With a large site, managing these files can be difficult. If, instead, we use a database to manage these pictures, then we need a way of extracting them again for delivery over the Web.

Field Name	Format	Comment
PICTURE_NO	AutoIncrement	Unique id for each image
PICTURE_TEXT	Alpha 30	Description of the image
PICTURE_TYPE	Alpha 30	The MIME type of the image
PICTURE_BLOB	BLOB	The actual image

Figure 3: Fields in the demonstration table.

Delivering Graphics

To deliver a picture in response to a client request, we need to duplicate what the server would do with a file. This means we must supply the document type, along with other metadata, which describe the length of the response, and indicate its beginning. After these headers, we send the actual content of the image in its original binary format. Obviously, this comes from a BLOB field in the database table.

For the purposes of this article, we'll create a very simple database table that has the minimum fields necessary to effect delivery of the images. We'll then generate a Paradox table specifying the fields (see Figure 3), using the Database Desktop tool. Save the table as WebPics.db.

To load appropriate pictures into the database, use the WPLoad program that accompanies this article (see end of article for download details). It allows us to browse for files that contain .GIF or .JPEG images, and to place these in the database with a description. The image type is automatically extracted from the file, and is placed in the type field.

We also need to add an alias in the BDE to reference this new table. The alias is WebPics, and its path is set to point to the directory that contains the database table we just created.

Building the Web Module

The first step in creating our new Web module is to call on the Web Server Application expert. To do this, select **File | New**, and then the Web Server Application icon. Here, we are asked what type of Web module we require. In this case, select **CGI**, and press **OK**.

```
{ Load details about a scheme from the registry. }
function TwmdWebPics.LoadScheme(sId: string): Boolean;
begin

  Result := True;

  with regSchemes do
    try
      if not OpenKey(sRegKey + '\' + sId, False) then
        Abort;
      sSchemeId := sId;
      sSchemeName := ReadString(sNameKey);
      sAliasName := ReadString(sAliasKey);
      sUserId := ReadString(sUserKey);
      sPassword := Coded(ReadString(sPasswordKey));
      slsOtherParams.Text := ReadString(sOtherKey);
      sTableName := ReadString(sTableKey);
      sKeyField := ReadString(sKeyKey);
      sBlobField := ReadString(sBlobKey);
      sTypeField := ReadString(sTypeKey);
    except
      Result := False;
    end;
end;

{ Extract a picture from the database and return it. }
procedure TwmdWebPics.wmdWebPicswacGetPicAction(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  sSelect: string;
  stmHeader: TStringStream;
begin
  SetFields;
  { Check for valid scheme. }
  if not LoadScheme(slsHTTPFields.Values['SCHEME']) then
    Response.StatusCode := 400
  else
    begin
      { Initialize database with scheme details. }
      with dbsWebPics do begin
        AliasName := sAliasName;
        Params.Clear;
        if sUserId <> '' then
          Params.Add('username=' + sUserId);
        if sPassword <> '' then
          Params.Add('password=' + sPassword);
        if slsOtherParams.Count > 0 then
          Params.AddStrings(slsOtherParams);
        Open;
      end;

      { Find the required record and extract the image. }
      with Response, qryWebPics do
        try
          sSelect := sBlobField;
          if sTypeField <> '' then
            sSelect := sSelect + ', ' + sTypeField;
          SQL.Clear;
          SQL.Add('SELECT ' + sSelect);
          SQL.Add('FROM ' + sTableName);
          SQL.Add('WHERE ' + sKeyField + ' = ' +
                  slsHTTPFields.Values['ID']);
          Open;
          try
            ContentStream := TBlobStream.Create(
              TBlobField(FieldByName(sBlobField)), bmRead);
            { Set image type. }
            if sTypeField <> '' then
               ContentType :=
                 FieldByName(sTypeField).AsString
            else
              try
                stmHeader := TStringStream.Create('');
                stmHeader.CopyFrom(ContentStream, 0);
                ContentStream.Position := 0;
                if Pos('JFIF',Copy(stmHeader.DataString,
                                   1,10)) = 7 then
                  ContentType := 'image/jpeg'
                else if Pos('GIF', Copy(
                       stmHeader.DataString,1,10)) = 1 then
                  ContentType := 'image/gif';
              finally
                stmHeader.Free;
              end;
          except
            StatusCode := 500;
          end;
        except
          StatusCode := 404;
        end;
    end;
end;
```

Figure 4: Deliver a picture across the Web.

We are presented with a blank Web module, wrapped in the code appropriate for a CGI application. Place a Query component on the Web module, and enter the SQL code to extract the image type and the BLOB field, selecting a parameter for the record id.

Next, double-click on the Web module itself, or on its *Actions* property in the Property Inspector, to display the Web actions editor. Add a new action, give it a name, and mark it as the default. On the Events tab in the Object Inspector, create a handler for its *OnAction* event.

In this event, we execute the query with the id value passed in as a parameter (see the code in Figure 4). We then attach the response to a stream created from the BLOB field, and set the content type from the database field. Some error handling caters for not being able to open the database table. If no record is found for that id, then no image is returned.

HTTP requests can be submitted to this program in two ways: GET or POST. In a GET request, the user's parameters are appended to the URL itself after a question mark (?). With a POST, the parameters arrive separately from the URL.

Data passed in these ways end up in different properties of the *Request* object, both with the same structure. To ease processing in the remainder of the program, we determine which property is active, depending on the *MethodType*, and save a reference to it. All subsequent references to the parameters then use this pointer:

```
{ Set pointer to request fields depending on request method. }
procedure TwmdWebPics.SetFields;
begin
  if Request.MethodType = mtPost then
    slsHTTPFields := Request.ContentFields
  else
    slsHTTPFields := Request.QueryFields;
end;
```

Generic Picture Delivery

This works fine for our single database and its fields, but what if we want this capability on any image in any database table? We need to make the application more generic, and enable it to be configured for each specific case, without having to recompile it.

To access a database, we need several pieces of information: the BDE alias, the user id, and a password. Furthermore, to access a table and extract an image, we require the names of the table, its key field, and the BLOB and image type fields.

For this exercise, we store all this data in the registry. Under a common key, \Software\Kwood\WebPics, we maintain one key for each database scheme that we wish to access. Within this key are the individual values for the data identified above. To hide the password from prying eyes, a coding scheme is used. In this case, it's a simple substitution cipher, but a safer one could be used if deemed necessary.

To retrieve an image, we need two parameters with the request: the id of the scheme to use, and the id of the record that holds the picture. On processing such a request, we must first read the registry, using the scheme id to retrieve the other values. The connection ones are then placed into a *TDatabase* object, and the database is opened.

Next, a query is constructed from the remaining values and the record id. Once this is opened, we extract the image and its type, and return these to the client (see Figure 4).

As an added feature, if the image type field is not specified, we determine this from the picture itself. Each file format has an identifying mark in its header. In the case of .JPEG files this consists of the characters `JFIF` at positions 7 through 10 (starting from 1), while .GIF files start with the string `GIF`.

Maintaining Schemes

Having the scheme details in the Windows registry makes it more difficult to maintain them (as opposed to an .INI file). So, we enhance the WebPics application to perform this maintenance task as well. This requires an extra Web action, added after double-clicking on the Web module form, which we set to respond to the /config additional path information.

If no additional parameters accompany the request, we display a list of all the schemes the program currently knows about. This is achieved by using a *TPageProducer* component, and responding to its *OnHTMLTag* event, to substitute for the list of schemes. An HTML table is constructed in code to replace the tag. If we were storing data in a database, we could use a *TQueryTableProducer* or *TDataSetTableProducer* component, instead. To protect against later name changes, we use the *Request.ScriptName* property to fill in the HTML when referring to the program itself.

From this list of schemes, the user can select links that allow them to add a new scheme, update an existing one, or to delete one. The add and update options pass back a hidden parameter-called action, which is set to Get.

Along with this, the id of the scheme to retrieve (or zero when adding) is specified. This action parameter is looked for in the program, which causes it to generate a page containing all the details for a single scheme, and allows these to be altered. The pages come from two *TPageProducer* components, where each page is set up as an HTML form. The forms' actions call the program again, passing back the fields for a scheme.

After entering the requested data, the user sends the new details, which now arrive with an action of Add or Update,

```
{ Accept request and perform configuration actions. }
procedure TwmdWebPics.wmdWebPicswacConfigureAction(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  sAction: string;
begin
  SetFields;
  sAction := slsHTTPFields.Values['ACTION'];

  { Display a single scheme's details. }
  if sAction = 'Get' then
    begin
      if slsHTTPFields.Values['ID'] = '0' then
        Response.Content := wppAddScheme.Content
      else if LoadScheme(slsHTTPFields.Values['ID']) then
        Response.Content := wppUpdateScheme.Content
      else
        Response.StatusCode := 400;
    end
  else
    { Apply changes to the registry (if applicable) and
      redisplay complete list. }
    begin
      if sAction = 'Delete' then
        DeleteScheme
      else if sAction = 'Add' then
        AddScheme
      else if sAction = 'Update' then
        UpdateScheme;
      Response.Content := wppListSchemes.Content;
    end;
end;
```

Figure 5: Configuring the database schemes.

these being the labels on the respective submit buttons. The information is processed and stored in the registry.

If a delete link is selected (one for each scheme), the action parameter reads Delete, and the id of the scheme is passed along. The corresponding key in the registry is removed as this request is processed.

Following an Add, Update, or Delete, the default action of listing the schemes is performed again, ready for the next activity. All of this processing can be seen in the code in Figure 5.

So, to initiate a maintenance session with the WebPics program, simply call it with the configuration directive attached: http://localhost/cgi-bin/webpics.exe/config.

Demonstration

The WebPics CGI program is demonstrated through another CGI application: ListPics. This was constructed in the same manner as described above, and performs two main actions.

The default action (see Figure 6), with no parameters, produces a Web page that lists the description and image type for all the pictures in our database (up to a maximum of 20). Each has a hypertext link attached to it that displays that image on its own page. The bulk of this processing is done by a *TDataSetTableProducer* component and its *OnFormatCell* event.

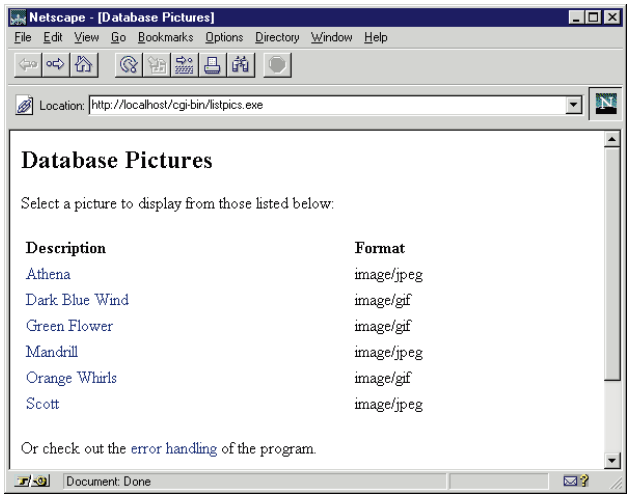


Figure 6: The list of images from the database.

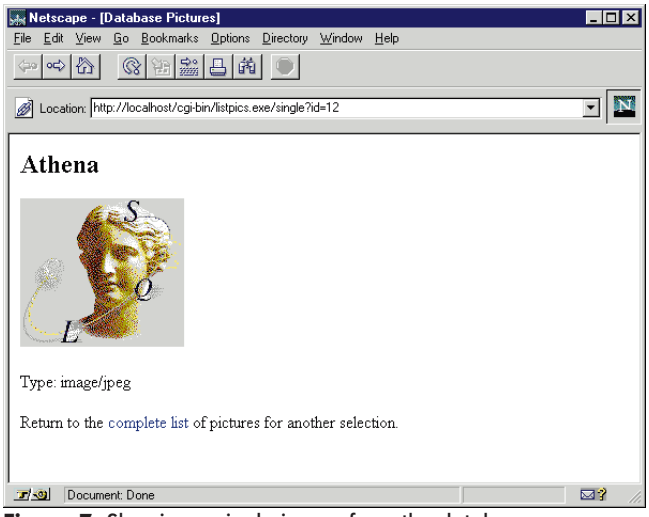


Figure 7: Showing a single image from the database.

The second action (see Figure 7), responding to the /single additional path information, generates a page showing the description of one image, its type, and the actual image itself. It requires a parameter of the id for the record to be displayed. In turn, it passes this on to the WebPics program to extract the image from the database. Two *TPageProducer* components are used to present the standard page or an error page, as appropriate.

To view the HTML for each page, simply use the **View | Source** option of your browser. The workings of this application can be seen by looking at the code that accompanies this article.

Conclusion

We've seen how we can create CGI Web applications using Delphi 3's new components. We've written a utility that can deliver images out of a database and across the Web, as well as an application to access those pictures easily.

Of course, just about anything can be stored in a BLOB field, so this technique can be used to send sounds or even applets and programs.

A large part of the future seems to be involved in providing content through the World Wide Web. Of even more use is the ability to interact with the client in deciding what data they want to see. Interpreting their requests, and constructing an appropriate response, requires programming of some sort. Using a fully fledged programming environment such as Delphi allows us to perform just about any action required to achieve this end. The new Web components of Delphi 3 make this even easier. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\FEB\DI9802KW.

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

Status Problems

While running this program through the WebSite Web server, and viewing it with Netscape, I kept getting an error message: "Unknown status reply from server: !0".

Tracing the content sent from the Web server, it appeared that the HTTP header was being incorrectly sent. It was going as: "HTTP/1.0 OK", rather than the required: "HTTP/1.0 200 OK".

To overcome this, I needed to modify the source code for the CGIApp unit. When sending the response in the *SendResponse* method of the *TCGIResponse* component, I altered the first header line to provide the format above:

```
{ The following line did not appear to work with WebSite
  server, so I replaced it with the one immediately below.
AddHeaderItem(StatusString, 'Status: %s'#13#10); }
AddHeaderItem(HTTPRequest.ProtocolVersion + ' ' +
              StatusString, '%s'#13#10);
```

This version is available in the code accompanying this article, as unit CGIApp2.

— *Keith Wood*

By Alan C. Moore, Ph.D.

Abbrevia and LockBox

TurboPower's New File/Data Manipulation Libraries

When we think of file/data manipulation, we generally think of the common operations of opening, saving, copying, deleting, and so forth. In the changing world of information technology, two new operations must be added to the list: encryption and compression. Of course, Delphi provides excellent support for all the common operations, but practically none for the last two.

Now, TurboPower has helped fill that void with two of its newest component libraries: Abbrevia, which provides support for working with PKZip-compatible files; and LockBox, which supports the encrypting and decrypting of files. While the two libraries are quite different, there's at least one situation in which I can imagine using the two of them together: transmitting sensitive data over the Internet.

I'll begin by discussing each of these libraries separately, concentrating on their uses and their special features. Then I'll examine some of the common features in both libraries. We'll begin with Abbrevia, then turn our attention to LockBox.

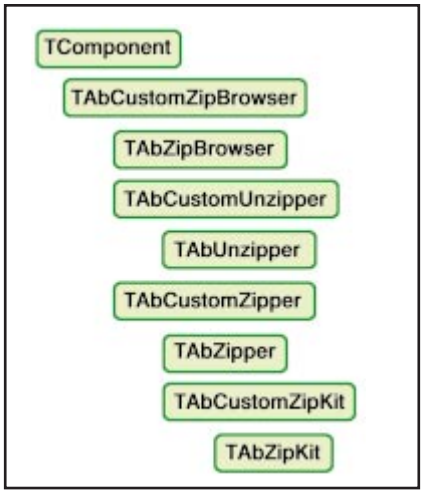


Figure 1: Abbrevia's hierarchy of non-visual components.

The Many Ways to Zip

It's difficult to imagine any programmer reading this who's not familiar with compressed files, particularly PKZip files. That particular application, or more precisely collection of applications, is one of the great shareware success stories. So much so that the PKZip-compression method has become the *de facto* standard in the computer industry. Many applications, including Norton Navigator for Windows 95, now include support for PKZip-compatible files. With Abbrevia, any Delphi (or C++Builder) programmer can now add such support. Exactly what kind of support are we talking about? Let's see.

Abbrevia supports the compressing and decompressing of files, adding them to, or deleting them from, an archive, and viewing the contents of an archive — all common operations. It also provides support for some of the less common operations, including creating a self-extracting archive (an .EXE file that, when executed, extracts all the archived files stored within it), and even compressing or decompressing on-the-fly without saving to a file. Let's take a look at the components and classes that make this possible.

Abbrevia consists of one visual component, several non-visual components, and various low-level support classes. For most applications you can select one or more components to provide the functionality you need. When you need greater control, you can use the same support classes the components use. The hierarchy of non-visual components is shown in Figure 1; the hierarchy of support classes is shown in Figure 2. Note that all the support classes are derived from *TObject*. *TAbArchiveItem* describes a single file in an archive while *TAbArchive* describes an entire archive. Both classes are abstract. Two useful classes are descended from these: *TAbZipItem* and *TAbZipArchive*, respectively.

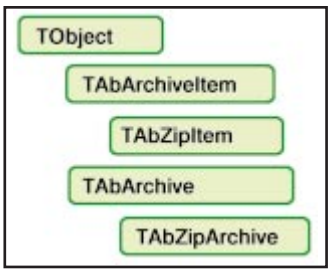


Figure 2: Abbrevia's hierarchy of support classes.

Powerful and Flexible Non-visual Components

As you can see from Figure 1, just about all the non-visual components are based on custom components whose properties are defined but not published. This makes it easy to derive your own specialized components, but I doubt it would ever be necessary. As the name implies, *TAbZipBrowser* allows you to inspect the contents of an archive, but not perform any other operations, such as extracting or adding files.

Two derivatives of this component, *TAbUnZipper* and *TAbZipper*, provide the additional functionality. Both inherit useful properties from *TAbZipBrowser*, which allow you to set the base directory for adding and extracting files, the file name of the archive itself, and the Items array that contains the names of the files in the archive. There are also several useful events. Particularly important is *OnProcessItemFailure*, which allows you to handle exceptions that arise in working with files in an archive.

The extracting component, *TAbUnZipper*, adds a few new properties related to extracting files and using password protection. Its method, *ExtractFiles*, takes care of the drudgery of extracting the files from the archive. Events such as *OnConfirmOverwrite* and *OnNeedPassword* give you and your users a good deal of flexibility in unzipping files.

The archive-building component, *TAbZipper*, adds new properties and events. The *CompressionMethodToUse* property determines how files will be stored in an archive. The *AutoSave* property controls when changes are made. The *DosMode* property can be used to make an archive DOS compatible (restricted to DOS-compatible filenames). The *AddFiles* method allows you to easily add files to an archive. Events such as *OnConfirmSave* and *OnRequestBlankDisk* allow you to anticipate and handle a variety of file compression scenarios.

The last of the non-visual components, *TAbZipKit*, combines the functionality of the previous two components and adds a powerful new feature: the ability to work with hidden compression. What is hidden compression? It's transparently compressing an application's data files upon exiting, and decompressing them when the application is loaded. This technique would be especially useful with large data files or with data files to which the user needs to add password protection for security reasons.

We'll be discussing data security in some detail shortly when we examine the LockBox library, but first let's take a look at Abbrevia's crowning glory: *TAbZipOutline* (see Figures 3 and 4).

Power, Flexibility, and Ease of Use

If, for some reason, you need or want to control the visual interface, the non-visual components we've been discussing so far will meet your needs. However, Abbrevia's *TAbZipOutline* component makes it even easier to provide compression capabilities for your applications. It's descended from *TwinComponent*, not *TOutline*, to prevent access to some of the latter's methods and properties that would be inappropriate here. It provides all the functionality of the *TAbZipKit* component along with an outline display appropriate for viewing and working with the files in a PKZip-compatible archive.

Figure 3, taken from one of the example programs, shows this component in use. There are a few more features of this library I want to describe, particularly the low-level compression classes. I will postpone that discussion, however, until we look at the common features in Abbrevia and

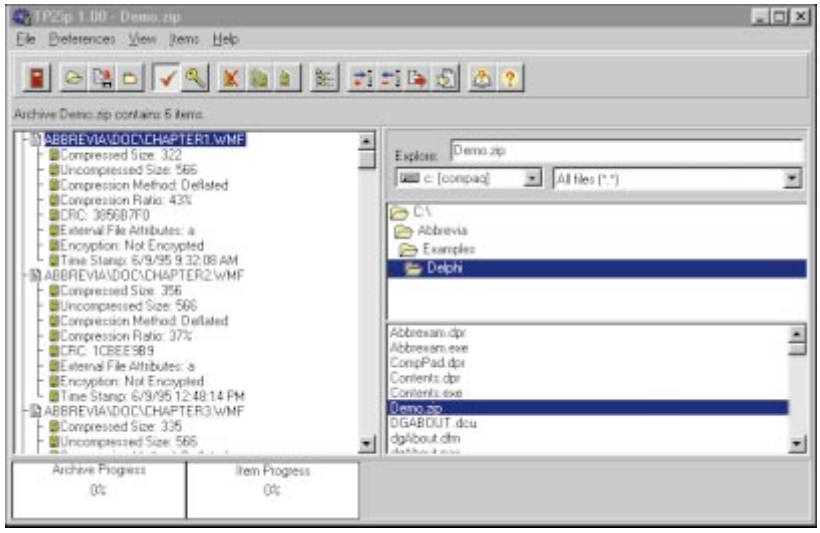


Figure 3: Abbrevia's main example program showing all of the .ZIP file functionality.

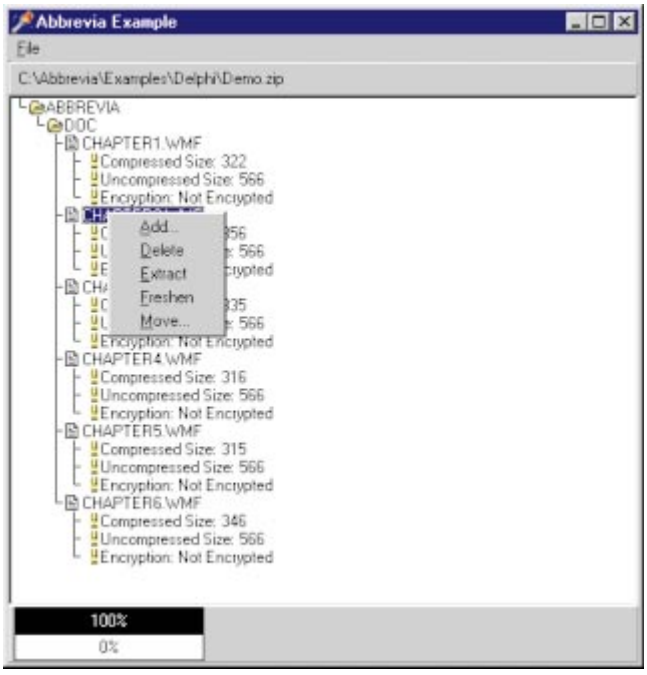


Figure 4: Abbrevia's crowning glory, its visual component, *TAbZipOutline*.

Name	Type	Key Size	Speed (in milliseconds)
Triple Data Encryption Standard	Block	128	360
Data Encryption Standard (DES)	Block	56	270
Blowfish Cipher	Block	128	250
LockBox Quick Cipher	Block	128	210
LockBox Cipher	Block	128	160
Random Number Generator 64-bit	Stream	64	120
Random Number Generator 32-bit	Stream	32	70
LockBox Stream Cipher	Stream	128	60

Figure 5: The ciphers available in LockBox.

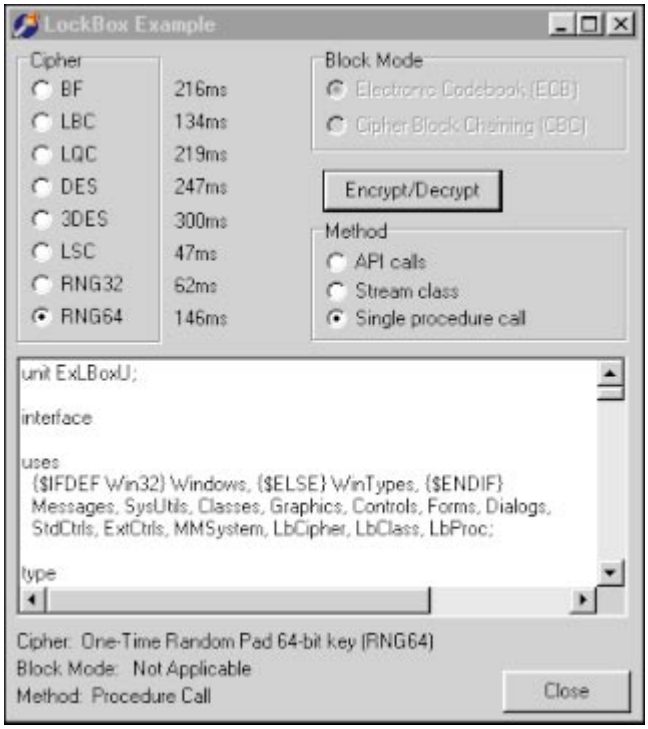


Figure 6: LockBox's example program, ExLBox, demonstrating the use of block ciphers.

LockBox. Now let's put on our spy outfits and enter the world of cryptography.

Data Security with LockBox

Like Abbrevia, LockBox provides many options and several levels at which you can work. Before starting this review, I wasn't aware of the variety of data-encryption methods available. The methods in LockBox fall into two general groups: those using block processing and those related to streams. Within these you can choose from many encryption algorithms and key sizes, depending on the requirements of your application. Generally, there is a tradeoff between the degree of security and speed of execution. The tougher you make your data-encryption system to crack, the longer the data processing will take.

Figure 5 summarizes the different ciphers available in LockBox. You'll notice a number of obvious patterns. While key size is one of the factors related to speed (the larger the key, the slower the processing), it's by no means the only one. Key size is also one of the main factors in ensuring security. You can't decipher data without the key used to encrypt it. So the question becomes: What's the maximum amount of time it takes to find the key by trying all the possible combinations? The manual discusses this issue in some detail. *Byte* magazine points out that, while a 40-bit DES key could be cracked in about 0.4 seconds, a 128-bit DES key could take up to 157,129,203,952,300,000 years!

Regarding the available options, Figure 5 doesn't tell the complete story. Each of the five block ciphers comes in two forms: one using Electronic Codebook (ECB) mode and one using Cipher Block Chaining (CBC) mode. While ECB is slightly faster, CBC offers greater security because it uses the encryption results from processing the previous block to process the current block. Then we have the various stream modes.

All the stream methods are faster than the block methods. The Triple Data Encryption Standard is by far the slowest because it uses a 128-bit key and applies the DES encryption algorithm to the data three times. Generally, block and stream ciphers are appropriate for different programming situations: Block ciphers work well with data in memory while stream ciphers are ideal for operations involving files. I'll have more to say about the latter when I discuss the stream support in LockBox and Abbrevia. The block ciphers are demonstrated fully in the example program, ExLBox (see Figure 6). You can use this program to test the relative speed of the various methods. In addition to its many encryption methods, LockBox also provides helpful programming choices.

Low or High: Choose Your Level

As with many of its other component libraries, TurboPower's LockBox provides you with the option of working with low-level API routines or high-level classes. The low-level routines give you more control while the high-level classes make your work easier. The low-level routines fall into several groups:

- GenerateXKey (where X is one of several key types) allows you to generate the random key to use in encrypting/decrypting data.
- InitEncryptX (where X can be any of the first seven basic encryption methods) allows you to prepare the encryption system.
- EncrytX (where X can be any of 13 encryption methods) allows you to encrypt or decrypt data.

LockBox provides high-level classes for working with memory or file steams (which we'll discuss soon) and high-level procedures and functions for working with any of its encryption methods (stream based or block based). Some of the later routines encapsulate the functionality of the low-level routines we looked at earlier, and add file-writing capabilities. Others allow you to create your own key-generation procedure(s) using one of several hashing algorithms. By now you should have a good idea of the main features of both of these libraries. Now let's discuss some of the common features of the two libraries.

Flowing with the Stream

As Ray Lischner points out in *Secrets of Delphi 2* [Waite Group Press, 1996], streams are the preferred way of working

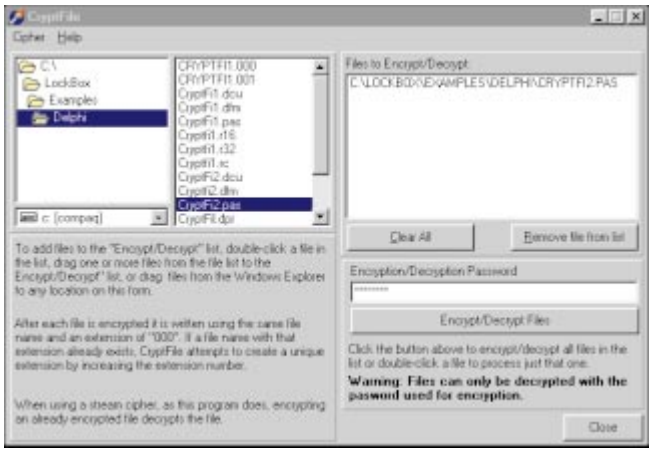


Figure 7: The example program, CryptFile, demonstrating the use of stream ciphers.

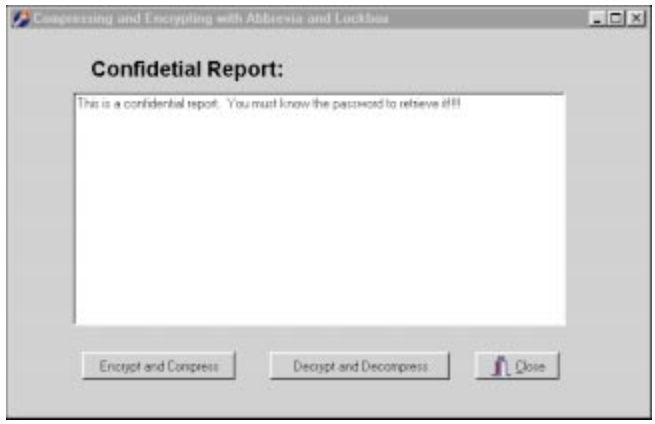


Figure 8: A new sample program, ConfiZip, using components from Abbrevia and routines from LockBox to encrypt and compress confidential messages.

with files in Delphi. They give you the ability to work with data in memory and copy data structures from one medium to another (e.g. memory to file). Both these libraries take advantage of this, and in doing so increase their value to us.

Abbrevia includes two routines (always used together) that allow you to compress and decompress data without having to hassle with intermediary files. Using *DeflateStream*, you can save an application's data files in compressed form when you, or your users, are finished with them, then load them again when needed. All this is handled transparently.

Likewise, LockBox provides stream classes, which allow you to save encrypted data to a file and later retrieve it using one of six encryption methods. Figure 7 shows an example program, CryptFile, that demonstrates the use of stream ciphers. Here again, the encryption is handled automatically and is completely transparent to the user. From a security point of view, it's nearly impossible for anyone to make sense of the data stored in the encrypted file without first knowing the method used and the encryption key.

As a bonus, I've included a sample program with this article, ConfiZip, which uses components from Abbrevia and routines from LockBox. It first encrypts a text file (using Abbrevia's built-in password dialog box) then saves the file in compressed format. Naturally, the program also allows you to reverse this process. Figure 8 shows the layout of the main form; the code for the main form is given in Listing Three (on page 37) and indicates how little programming is required to take advantage of these component/class libraries. (You can download all the source code and the 32-bit executable file: see end of article for details).

Other Common Features: Quality We've Come to Expect

TurboPower's excellent tradition of documentation continues with these libraries. And, of course, the manuals include a full description of all the components (Abbrevia), classes, and low-level routines. In addition, each manual provides an excellent introduction to the area of programming it addresses, data compression and data encryption, respectively. In Abbrevia, you learn a little of the history of data compression, its various methods, and sources of additional information. In LockBox, you learn detailed information about the various encryption methods, the importance of keys, and the uses for stream and block methods. Again, there's a list of references if you wish to learn more.

As with TurboPower's other libraries, Abbrevia and LockBox include comprehensive online Help (mirroring the material in the manuals), full source code, and excellent example programs. This company continues to provide free technical assistance and a 60-day, money-back guarantee for its products. You can download a fully functional demonstration version of Abbrevia from TurboPower's Web site, and find out first hand if it meets your needs. There's no demonstration version of LockBox because some of its encryption methods are too powerful to be exported from the United States.

Conclusion

Using native Delphi classes and components, Abbrevia and LockBox provide excellent compression and encryption functionality for Delphi programmers. Once again, TurboPower demonstrates its leadership in the third-party Delphi marketplace. That leadership is based on meticulous attention to detail, excellent documentation, and responsiveness to the needs and wishes of customers. If you need to add PKZip-compatible compression or high-level encryption to your applications, I think you will be more than satisfied with these excellent products. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\FEB\DI9802AM.

INFORMANT FACT FILE

Abbrevia is an excellent collection of components and classes that provides a complete solution to working with PKZip-compatible files. Its crowning glory, *TAbZipOutline*, allows you to easily work with any of the files in an archive (e.g. browsing, adding, or retrieving). LockBox is a comprehensive collection of encryption/decryption classes and low-level routines for adding data security to an application. It includes a large number of options to cover various programming situations and needs. Both libraries provide support for Delphi streams, and come with excellent documentation and example programs. Abbrevia is available in a demonstration version. Both come with a 60-day, money-back guarantee.

TurboPower Software Company
P.O. Box 49009
Colorado Springs, CO 80949-9009
Phone: (800) 333-4160 or (719) 260-9136
Fax: (719) 260-7151
Web Site: http://www.turbopower.com
Price: Abbrevia, US$199; LockBox, US$249 (available only in the US and Canada).

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

Begin Listing Three — Unit ConfZipU;

```
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons, AbZipper, AbArcTyp,
  AbZBrows, AbUnZper;

type
  TForm1 = class(TForm)
    Memo1: TMemo;
    Label1: TLabel;
    EncryptAndCompressBtn1: TBitBtn;
    DecryptAndDecompressBtn1: TBitBtn;
    BitBtn1: TBitBtn;
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    AbUnZipper1: TAbUnZipper;
    AbZipper1: TAbZipper;
    procedure EncryptAndCompressBtn1Click(Sender: TObject);
    procedure DecryptAndDecompressBtn1Click(
      Sender: TObject);
  private
    { Private declarations }
    EncryptPassword : string;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

uses
  Abdlgpwd, LbProc, LbCipher;

{$R *.DFM}

procedure TForm1.EncryptAndCompressBtn1Click(Sender: TObject);
var
  Dlg : TPassWordDlg;
  Key : TKey128;
  EncryptedFile : string;
begin
  if (Memo1.text='') then
    begin
      MessageDlg('You haven't entered any text to save',
                 mtError, [mbOK], 0);
      Exit;
    end;

  Dlg := TPassWordDlg.Create(Application);
  EncryptPassword := '';

  try
    Dlg.ShowModal;
    if Dlg.ModalResult = mrOK then
      EncryptPassword := Dlg.Edit1.Text;
  finally
    Dlg.Free;
  end;

  if EncryptPassword = '' then
    Exit;

  ChDir(ExtractFilePath(Application.ExeName));
  SaveDialog1.Title :=
    'Enter Name of Text File to Archive';
  SaveDialog1.Filter := 'Text files (*.txt)|*.TXT';

  if SaveDialog1.Execute then
    begin
      if Pos('.', SaveDialog1.FileName) = 0 then
        SaveDialog1.FileName :=
          Concat(SaveDialog1.FileName, '.txt');

      try
        Memo1.Lines.SaveToFile(ChangeFileExt(
          SaveDialog1.FileName, '.txt'));
      finally
      end;

      with AbZipper1 do begin
        BaseDirectory :=
          ExtractFilePath(SaveDialog1.FileName);
        AbZipper1.Filename :=
          ChangeFileExt(SaveDialog1.FileName, '.zip');
        EncryptedFile :=
          ChangeFileExt(SaveDialog1.FileName, '.xxx');
        GenerateLMDKey(Key, SizeOf(Key), EncryptPassword);
        LBCEncryptFile(SaveDialog1.FileName, EncryptedFile,
                       Key, 16, True);
        AddFiles(SaveDialog1.FileName, 0);
        Save;
      end;

      if MessageDlg('Delete Text File?', mtConfirmation,
                    [mbOK, mbCancel], 0) = idOK then
        DeleteFile(ChangeFileExt(SaveDialog1.FileName,
                                 '.txt'));
    end;
end;

procedure TForm1.DecryptAndDecompressBtn1Click(
  Sender: TObject);
var
  TextFile : string;
  Dlg : TPassWordDlg;
  Key : TKey128;
  EncryptedFile : string;
begin

  Dlg := TPassWordDlg.Create(Application);
  EncryptPassword := '';

  try
    Dlg.ShowModal;
    if Dlg.ModalResult = mrOK then
      EncryptPassword := Dlg.Edit1.Text;
  finally
    Dlg.Free;
  end;

  if EncryptPassword = '' then
    Exit;

  OpenDialog1.Title := 'Open Zip File';
  OpenDialog1.Filter := 'Zip files (*.zip)|*.ZIP';

  if OpenDialog1.Execute then
    begin
      TextFile := ExtractFileName(ChangeFileExt(
                    OpenDialog1.FileName,'.txt'));
      EncryptedFile :=
        ChangeFileExt(OpenDialog1.FileName,'.xxx');
      with AbUnZipper1 do begin
        BaseDirectory :=
          ExtractFilePath(OpenDialog1.FileName);
        ChDir(BaseDirectory);
```

```
        Filename := OpenDialog1.FileName;
        ExtractFiles(EncryptedFile);
        GenerateLMDKey(Key, SizeOf(Key), EncryptPassword);
        LBCEncryptFile(EncryptedFile, TextFile,
                       Key, 16, False);
      end;

      try
        Memo1.Lines.LoadFromFile(TextFile);
      finally
      end;

    end;
end;

end.
```

End Listing Three

High Performance Delphi 3 Programming

High Performance Delphi 3 Programming is a repackaging of last year's unfortunately titled *Kick Ass Delphi Programming*. Publisher Keith Weiskamp explained that the original titles in the *Kick Ass* series met some resistance with stores' buying agents, making it difficult to get these valuable volumes into the hands of programmers. In this retitled edition, a handful of new chapters have been added, and one has been removed; however, the rest of the text remains fundamentally the same.

The reader will find no introductory Delphi or Object Pascal material here; the book's intended user level is accurately labeled as Intermediate to Advanced. The material and topics are spread evenly across this spectrum. Seven authors contribute to the work, with each exhibiting a vastly different style.

Jim Mischel authored four useful chapters about advanced utility topics; the first three of these appeared in the book's previous edition. Chapters covering console applications and the development and use of DLLs are must-reads for consulting and commercial developers. The explanations and examples are simple, and provide a solid basis for further development.

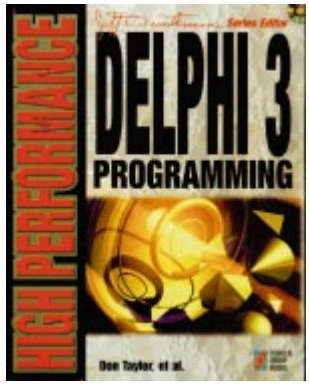
The chapter detailing drag-and-drop through the Windows Shell interface, one of the holdovers from the first edition, now acts as a "point" to the following chapter's counterpoint: implementation of a drag-and-drop interface with OLE/ActiveX (whose code represents a much more elegant solution than that of its predecessor). The chapter also offers an excellent introduction to OLE technology. The author suggests additional readings to help develop a deeper understanding of the topic.

John Penman contributed one new chapter in addition to the two chapters brought from the older book. His area of interest is in developing Internet applications using the Winsock API. The lead chapter develops *CsSocket*, a wrapper component for Windows Sockets. This offers a good introduction to Winsock. The *CsSocket* component is used as the basis for the two following chapters, which describe a pair of FTP components providing both client and server functionality. The FTP client was initially presented in the first edition; utilizing the *CsSocket* component, it can stand alone or be added to other projects. Penman adds the counterpart to the client component in a new chapter that details the development of an FTP server component, encapsulating the server side of the FTP protocol.

Jon Shemitz offers three chapters — one with co-author Ed Jordan — which are unrelated in content, but go a long way toward demonstrating the depths to which Delphi can be plumbed in application development.

The first chapter in the set details the building of a Fractal generator. The mathematics and use of assembler functions make this a good Sunday afternoon project that will result in a broader skill set. Chapter 9, the collaborative effort of the two authors, is a collection of ideas and problem solvers on various topics ranging from the inner workings of *TPersistent* descendants, to how to build a Delphi application that acts as its own setup program, to streaming data to the Clipboard. The chapter is a good read filled with possibilities that the programmer can file away until needed.

A new chapter entitled "Models, Views, and Frames" is one argument for adding this volume to your library even if you possess the earlier edition. Mr Shemitz's clear explanation of utilizing frames and views to embed one form inside another transmits the insight needed



to create a truly modern user interface. The implementation of setup wizards and property sheets, both affecting the same data objects, is a complex endeavor. By leading the reader through several implementation choices (and the traps associated with each) the author indicates which methods provide the most elegant solution. This chapter takes considerable study time, but the results are well worth the work.

The remaining chapters are fundamentally unchanged from the previous publication. Terence Goggin opens up the Math Unit, and shows some useful functionality that lacks documentation and popular usage. Bugs are identified, and a clever way of handling dynamic data and static arrays is explained in the course of developing a component that adds statistical functions to a project. Goggin also introduces building dynamic user interfaces. A single chapter by Richard Haven explores Hierarchical

Data and its representation. Working this concept from the example of data contained in a relational database, the author offers numerous techniques for viewing and navigation. Like a few other chapters in the book, this topic requires dedicated study, and has a limited and specialized audience.

Don Taylor writes the final four chapters in a style he pioneered with *Delphi Programming Explorer*. Readers are reintroduced to Ace Breakpoint, a private investigator who combines adventure *noir* with development exploration. This unique text intersperses Mr Breakpoint's adventures with pages from his "casebook" of programming topics, which can be integrated into a programmer's repertoire: packing dBASE and Paradox tables, floating toolbars, DLLs, and the Windows 95 core, among others. While the narrative may be an acquired taste, readers will find much useful information.

The CD-ROM that accompanies the book contains all the examples listed in each chapter. The code compiled without a problem in Delphi 3 — an improvement from the earlier edition, whose code contained many difficulties and errors associated with the transition from Delphi 1 16-bit to Delphi 2 32-bit. The CD's installation program, which moves projects to the hard disk for compilation purposes, gives the reader a choice of which demo programs to include.

A programmer's library typically consists of two shelves: the "tutorial and methods" collection, and the *How to*, *Secrets of*, and *High Performance* collection. Books on this second shelf help programmers solve problems and add depth to their programming skills. If *Kick Ass Delphi Programming* is already on that shelf, the reader should review the new material to determine if it warrants the purchase price. Otherwise, this well written and well organized collection of advanced development information deserves a place in your programmer's library.

— *Warren Rachele*

High Performance Delphi 3 Programming, by Don Taylor, et al., Coriolis Group Books, 14455 N. Hayden Road, Suite 220, Scottsdale, AZ 85260,
(602) 483-0192, http://www.coriolis.com.

ISBN: 1-57610-179-7
Price: US$49.99
(635 pages, CD-ROM)

Packages

The Potentials and the Pitfalls

The introduction of packages was one of the major changes in Delphi 3. Before packages, if you wanted to work with a particular subset of the VCL, your options were somewhat limited. Of course, you could create and load different versions of complib.dcl. However, this was a hassle, and rather wasteful of disk space. With packages, those hassles are gone. Unfortunately, they've introduced a new set of problems. But let's start with the good news.

You Gotta Take the Good ...

In Delphi 3, if you select **Component | Install Packages**, you see all the packages currently installed and the various options you have: You can add a single package or a collection of packages, remove packages, edit packages, and examine the components in a package. If you encounter a problem or conflict with a particular package, you can simply de-select it by clicking the check box. With the Package Editor you can create your own new packages, and with the Package Collection Editor you can create collections of packages to distribute to other developers.

Delphi 3 Help points out that "design-time packages ... simplify the tasks of distributing and installing custom components." That's true. When I first installed third-party component libraries in Delphi 3, I was amazed at how smoothly the process went. Once the setup program was finished and I fired up Delphi 3, the new components appeared, already installed on the palette. What a difference from Delphi 1 and 2!

... with the Bad

Unfortunately, I soon encountered problems: incompatibilities between packages from some of the third-party tool and component producers. These conflicts occurred because one product depended upon components in the other, but was using a different version of those components than the one installed on my system. I received the following not-so-informative message: A device attached to the system is not working properly and the package cannot be installed (or something to that effect). Because I knew what I had installed recently, I was able to un-check one or more packages, getting one to work, but not both.

Gradually, some sanity has emerged in how to work with packages. But this has more to do with the third-party developers' commitment to correcting the problem than with Borland's leadership. Thanks to developers such as Ray Konopka, component writers now have some guidelines to help them avoid these conflicts. In *Developing Custom Delphi 3 Components* [Coriolis Group Books, 1997], Mr Konopka imparts some excellent advice to the developer who will be distributing components in packages:

- Always use a unique prefix in naming your components to avoid conflicts with components created by other developers.
- Make sure your package names are unique.
- Never put property or component editors in run-time packages; these belong only in design-time packages.
- Make certain that the registration procedures are not in the component unit (as we've become accustomed to doing), but in a separate registration unit. Otherwise, your component could also be inadvertently installed by someone using a property editor that is dependent upon it, but written and distributed by someone else.

Leading vendors such as TurboPower are now aware of the problems that can arise with packages, and are taking aggressive steps to shield us from them. That particular company, for example, is giving each new product release a slightly different package name from its predecessor to avoid version conflicts. I greatly appreciate such thoughtfulness. However, I can't help but wonder if all this could have been avoided in the beginning if Borland had put as much energy into warning us about the pitfalls of packages as they did touting their advantages. What do you think? Let me know your views, experiences, problems, and solutions in working with packages. Δ

— 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 via e-mail at acmdoc@aol.com.