June 1999, Volume 5, Number 6

Cover Art By: *Darryl Dennis*

ON THE COVER

6 ActiveX Intranet — Thomas J. Theobald
Because it's platform-specific, ActiveX isn't suitable for most Internet solutions. *Intranet* solutions, however, are a different story. Mr Theobald discusses the issues of creating ActiveX components with Delphi for use on your corporate intranet, and presents three example projects to help get you started.

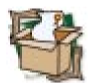
FEATURES

12 DBNavigator
ASCII Made Easy — Cary Jensen, Ph.D.
The ubiquitous ASCII file gets no respect, but is nonetheless extremely useful. Dr Jensen demonstrates — with words and code — how the Table component makes working with these files a snap.

16 On the 'Net
An HTML Generator: Part II — Keith Wood
Last month, Mr Wood introduced the *IHTML* object, which provides an OO approach to generating HTML from a Delphi program. This month, he finishes by demonstrating its use in three applications.

22 Patterns in Practice
The Builder Pattern — Xavier Pacheco
Mr Pacheco discusses the Builder pattern, and uses it to extend the application framework he presented last month — a framework that allows developers to add modules dynamically using Delphi packages.

33 Case Study
HMS Software's TimeControl — Chris Vandersluis
Mr Vandersluis explains why HMS Software chose Delphi Client/Server as the development tool for its TimeControl 3 product. A primary reason was Delphi's strong C/S tools, including SQL Links.



REVIEWS

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

2 Delphi Tools
5 Newsline
36 File | New by Alan C. Moore, Ph.D.

New Products and Solutions

Wise Introduces Wise for Windows Installer

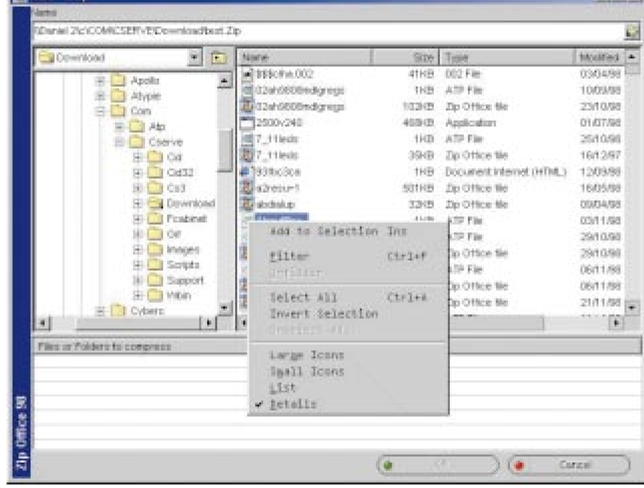
Wise Solutions, Inc., working closely with Microsoft Corp., announced the creation of ***Wise for Windows*** installer, a product that enables application developers to create installation programs compatible with Microsoft's Windows installer technology.

Microsoft's Windows installer is an application installer originally designed for Windows 2000 (formerly Windows NT 5.0). However, both the development environment and the installations created will support Windows NT 4.0 and Windows 95/98. Designed to reduce the administrative requirements of managing Windows workstations, the technology plays a key role in Microsoft's Zero Administration initiative for Windows. The Wise for Windows installer will enable developers to create installations that meet the new Microsoft standard.

Wise Solutions, Inc.
Price: US$679
Phone: (800) 554-8565
Web Site: http://www.wisesolutions.com

Atypie Introduces Zip Office 98

Atypie Software released *Zip Office 98*, a Windows 95/98/NT utility that lets you compress, uncompress, and manage your files.



Zip Office 98 lets you create new compressed archives, as well as add, extract, view, and delete files. It supports other compression formats, including lha, arc, arj, zoo, uue, and others, without using external programs. You can also convert your compressed files from one compression format to another.

The Active Zip feature lets you use your day-to-day programs to work on compressed files, and automatically updates those files inside your compressed archives.

Zip Office 98's Virtual Folders let you organize your downloads from the Internet. While your downloads are stored in the same physical folder, Zip Office 98 lets you define logical folders and sub-folders.

Zip Office 98 provides access to right-click context menus, allowing you to perform Zip compression functions directly from other programs. Its **Send** menu contains all your Microsoft Explorer **Send To** menu selections, allowing you to e-mail files directly from Zip Office 98.

Programmers who would like to include these file compression functions within their software can use Zip Office 98's Open API to extract files directly from any application that was built with Delphi, Access, VB, or other Windows programming languages.

Atypie Software
Price: US$35 for one computer.
Phone: (877) 353-7297 or (32)(2) 346 96 21
Web Site: http://www.zipstore.com

HyperAct Launches eAuthor DB 1.0

HyperAct, Inc. announced ***eAuthor DB 1.0***, a database import plug-in for the company's eAuthor products (eAuthor Help and eAuthor Site), template-based RAD authoring tools for large-scale Web sites and HTML Help projects.

eAuthor DB 1.0 allows the user to create database queries and create import operations that use the data to create and populate pages in the project from eAuthor templates. It enables developers to bind template properties to database fields, or use the eAuthor Authoring Templates Object Model to script import operations.

eAuthor DB 1.0 provides step-by-step wizard design, Visual SQL Builder, a tutorial, and a user guide that includes information to help users import data from databases, such as Borland dBASE, Microsoft Access, Microsoft SQL Server, Corel Paradox, and more.

eAuthor DB 1.0 can be used to create hundreds of pages from database information by binding database fields to template properties; create hierarchies of pages during the import process; define multiple import stages by setting database break fields; create multiple import operations in every stage; and control the import process using JavaScript import operations that provide access to every element in the eAuthor project and the database.

HyperAct, Inc.
Price: US$250
Phone: (402) 891-8827
Web Site: http://www.hyperact.com

New Products and Solutions

InnerMedia Ships DynaZIP-AX 4.0

Inner Media, Inc. announced *DynaZIP-AX 4.0*, a Zip-compatible data-compression toolkit/component for Windows developers. The toolkit provides multi-threaded operations, and lends itself to automated environments, such as Web servers, backup systems, etc.

A 32-bit product, it provides a pair of ActiveX components — one each for Zip and Unzip — and is designed for use with Delphi, Visual Basic, Access, Visual FoxPro, Visual C++, and any other 32-bit programming environment that supports ActiveX components.

The ActiveX components are self-contained, and do not require the installation or use of DLLs beyond those that are already present on a normal Windows machine. Included with DynaZIP-AX 4.0 are sample programs that show, in source code form, how to implement multi-threaded compress/decompress operations.

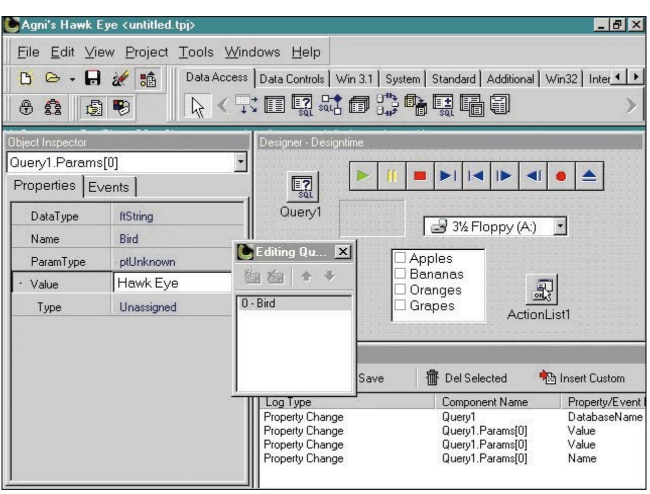
The DynaZIP-AX ActiveX components provide the full range of Zip and Unzip operations, such as read/write/modify, multi-volume (spanning), easy internationalization, multiple callback and progress monitor options, and much more. The parent product, DynaZIP 4.0, contains additional interfaces for database languages, Delphi, C/C++, and others.

Inner Media, Inc.
Price: US$149
Phone: (800) 962-2949 or (603) 465-3216
Web Site: http://www.innermedia.com

Agni Software Releases Hawk Eye 4

Agni Software (P) Ltd. announced *Hawk Eye 4*, a new version of the company's debugger for Delphi components.

Features of the new version include a Delphi Object Inspector look-alike that displays all properties, along with property editors and events; a Component palette like Delphi itself; a Form designer that supports Component Editors; a property and event log; full support for packages; the ability to switch between design-time and run-time environments with a single click; the ability to see your events being fired in the run-time environment; complete support for property and component editors; and more.

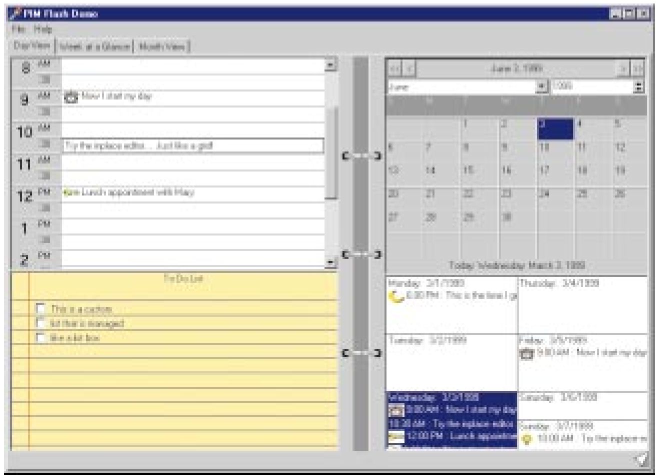


Agni Software (P) Ltd.
Price: Hawk Eye Suite (includes Hawk Eye, Eye Spy, and Eye Trace), US$400; Eye Spy and Eye Trace, US$250.
Phone: +91 80 228 6333, 226 8412, 226 2696
Web Site: http://www.agnisoft.com

Fe Software & Development Announces PIM Flash Components

Fe Software & Development released *PIM Flash*, its suite of over 20 calendaring components for Delphi 3 and 4 developers, including a date calculator that understands repeating holidays, and a set of UI components that make applications look like a real PIM. The standard set comes with a fully customizable calendar and the extensive number of properties you can modify to customize look and feel. Bitmaps and image lists are supported as well.

The professional package includes data-aware versions of the calendar, day view, and week view controls, as well as a set of GUI-oriented controls, such as the Spiral Splitter, Page Corner, List Pad, and others.



Fe Software & Development
Price: Standard Package, US$155.95; Professional Package, US$225.95.
Phone: (818) 838-1932
Web Site: http://www.fesoft.com

New Products and Solutions

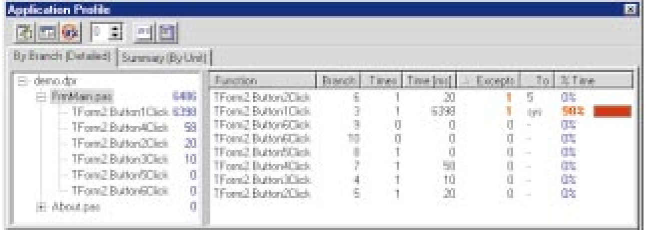
Peter Tiemann and HREF Offer SourceCoder 2.56

HREF Tools Corp. expanded its line of development tools to include reselling ***SourceCoder 2.56***, a utility created by independent developer **Peter Tiemann**.

SourceCoder analyzes and profiles Delphi source code, giving developers quantitative code measurements and enabling them to pinpoint problematic areas. SourceCoder analyzes Object Pascal files and computes metrics for source-code line counts, calculates documentation percentages, and provides complexity measurements. It also checks the scope of variables and points out ambiguities.

SourceCoder offers additional functions, including several code analysis tools; a bug-tracking database; code formatting options; automatic generation of comments per block, function, or unit; a database of projects and functions with links to the code; and many reports. It includes Nassi-Shneiderman charts, with optional profiling results, which are available in a form, as a report, and as HTML.

Peter Tiemann/HREF Tools Corp.
Price: From US$85
Phone: (831) 465-1207
Web Site: http://www.href.com/scoder



SkyLine Tools Announces Barcode Recognition Suite 1.0

SkyLine Tools Imaging announced the ***Barcode Recognition Suite 1.0***, a barcode recognition toolkit for various development environments.

The algorithm developed by SkyLine Tools Imaging is based on the fuzzy logic approach in image recognition. This approach solves the problem of identifying barcode type and can decode a barcode value in a fraction of a second. The user needs only to define the graphical image of the barcode as a graphical file (BMP, TIFF, JPEG, or any other supported format), a DIB handle, HBitmap, or HDC. The suite supports most of the widely used barcode standards, including EAN13, EAN8, UPCA, UPCE, 2OF5, 2OF5I, etc.

The Barcode Recognition Suite comes as a DLL that interfaces with Delphi, Visual Basic, Visual C++, and C++Builder.

SkyLine Tools Imaging
Price: US$1,999 (introductory price).
Phone: (800) 404-3832
Web Site: http://www.imagelib.com

Pegasus Announces Smartscan Xpress BARCODE

Pegasus Software announced the release of ***Smartscan Xpress BARCODE***, a 32-bit ActiveX control that reads over 20 industry formats, such as Code39, CODABAR, Interleaved 2 of 5, UCC128, EAN128, UPC-A, and more. Smartscan Xpress uses ATL technology and does not require MFC. The development kit is less than 630KB and can be used in any environment that hosts ActiveX controls, such as Delphi.

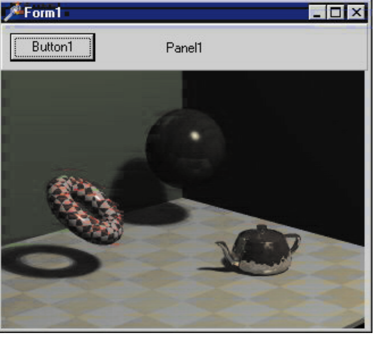
A developer can tell SmartScan to automatically detect all barcodes in an image, or specify an area within the image. SmartScan reports the barcode type, position, and skew angle for all detected barcodes. The detected barcode value is returned as a string, and Smartscan provides comprehensive error reporting.

The barcode detection process can be optimized if you know the barcode types, the orientation of the barcode(s), the maximum number of barcodes, or the color of the barcodes in an image.

Pegasus Software
Price: US$499
Phone: (800) 875-7709
Web Site: http://www.pegasustools.com

Andy Gibson Releases Renderlight

Andy Gibson announced the release of ***Renderlight***, a component for 3D-imaging applications that includes support objects and functions.

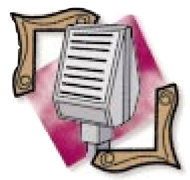


Renderlight works by having code pass geometry into the Renderlight engine, then calling a rendering method. Renderlight then handles the various transformations, clipping, and scan-line aspects. When it needs to color a pixel, it calls an event handler to pass in all the information it has about the polygon — normal information, view coordinates, object coordinates, texture coordinates, materials attached, and the polygon itself. You can then use any of that information to render the pixel.

There are additional support objects for geometry storage (including loading 3DS files with materials), loading JPEG textures, ray tracing, shadows, phong or gouraud shading, reflections, and procedural textures.

Andy Gibson
Price: Freeware (for personal use).
E-Mail: Renderlight@gibsona.demon.co.uk
Web Site: http://www.gibsona.demon.co.uk/index.htm

June 1999



Inprise Announces 10th Annual Inprise and borland.com Conference

Inprise Corp. and borland.com announced the 10th Annual Inprise and borland.com Conference, to be held July 17 through 21 in Philadelphia, PA. Featuring over 200 sessions, the conference will cover Inprise and borland.com tools, including Delphi, JBuilder, C++Builder, VisiBroker, Application Server, AppCenter, and InterBase. Beginning, intermediate, and advanced developers can choose from solution tracks covering such topics as distributed object computing, application deployment and management, application design and methodology, and more. Management track sessions will focus on the needs of IT directors and managers, and development and project managers.

The conference will also feature a hands-on computer lab and exhibit hall. In addition, pre-conference tutorials will be available as four-hour sessions.

Attendees will receive an Inprise/borland.com product worth up to US$350, as well as a conference CD-ROM containing proceedings and code samples from all the tracks.

The registration cost is US$995 before May 28, 1999, and US$1,295 after. To register, visit https://www11.cplan.com/conf99/regform.htm.

borland.com and Marotz Team for Record Delphi Certification

San Diego, CA — Marotz, Inc., a San Diego-based software development company, announced that it has teamed with borland.com, the software development tools division of Inprise Corp., to train, test, and prepare its developers for the Delphi developer certification exam. Marotz programmers had already undergone a six-month-long internal training and certification period when they began the training and testing for Delphi. With a pass rate of over 95 percent, 49 developers became certified, bringing Marotz's total of company-wide certified developers to over 50.

The testing and training was part of a 10-day Marotz company-wide offsite event. Employees from as far as Ukraine took part in the first bi-annual offsite event. Also attending were three certified Delphi trainers from borland.com and a number of Marotz clientele.

For more information on Marotz, visit http://www.marotz.com.

Inprise Licenses Visual dBASE to KSoft

Scotts Valley, CA — Inprise Corp. announced an exclusive international licensing agreement with KSoft, Inc., an Xbase specialist company, to further develop, support, and market Visual dBASE, the PC database development environment for Microsoft Windows.

With the new agreement, KSoft assumes the future development, maintenance, and marketing for Visual dBASE and DOS versions of dBASE internationally. KSoft will also have a joint marketing relationship with borland.com, Inprise's online community for software developers.

For more information on KSoft, visit http://www.dbase2000.com.

borland.com Announces Borland C++Builder 4

Scotts Valley, CA — borland.com announced that Borland C++Builder 4, a new version of its C++ development system, is available through major software distribution channels. Borland C++Builder 4 is a front-to-back C++ compiler and development environment for creating desktop, client/server, multi-tier, and distributed applications that are interoperable with multiple platforms.

borland.com also announced that customers who purchase the Enterprise or Professional versions of C++Builder 4 will receive a free copy of JBuilder 2, borland.com's Java development tool.

Some of the new features in Borland C++Builder 4 include rapid distributed development with CORBA and COM; a new, multi-standard flexible C++ compiler; support for the latest ANSI/ISO C++ language specifications; a customizable AppBrowser IDE with integrated two-way visual tool technology; support for Windows 95/98/NT, including multiple monitors, common controls, docking forms and toolbars, and more; Internet tools, including ActiveForms for building Web browser C++ applications and WebBroker for building CGI, WinCGI, ISAPI, and NSAPI C++ applications; a Multi-tier Distributed Application Services (MIDAS) Development Kit, including MIDAS 2; support for industry standards, including Oracle8i database server, MFC, Active Template Library, SQL Server 7, MTS, Object Windows Library, and Visual Component Library; and more.

Borland C++Builder 4 Enterprise has an estimated street price (ESP) of US$2,499 for new users. Borland C++Builder 4 Professional has an ESP of US$799. Borland C++Builder Standard has an ESP of US$99.

For more information or to place orders, call borland.com at (800) 233-2444, or visit http://www.borland.com.

Free Web Site Helps Computer Job Hunters

Brooklyn, NY — Today's data processing, information technology, and computer professionals have a new tool to help them research the current job market in any city in the United States. NSI is a full-service data processing search firm specializing in the placement of computer system professionals on a no-fee basis. NSI's Web site contains hundreds of job openings currently available with major banks, brokerage firms, software and communication companies, entertainment companies, and manufacturing firms throughout the country. Included are details on the various pay and benefits, job descriptions and requirements, step-by-step instructions on how to submit a resume, Web site links, and more. The Web site also provides the latest stock market, business, sports, and weather information.

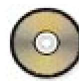
For more information and to view the latest openings, visit http://www.n-s-i.com, or send e-mail to nsi@idt.net. Job seekers may also call (718) 252-2306 or fax (718) 677-6751 to learn more.

Inprise Announces Repurchase of One Million Shares

Scotts Valley, CA — Inprise Corp. announced that since January 1, 1999, it has repurchased approximately 1,000,000 shares of stock at an average price of US$4.86 a share as part of a stock buy-back program authorized by its board of directors.

Since July, the company has repurchased approximately 4,500,000 shares at an average price of US$5.65 out of approximately 5,900,000 shares authorized for repurchase. On July 21, 1998, the Board of Directors approved the repurchase of 1,000,000 shares, which was completed on August 10, 1998. In September, the board authorized the repurchase of up to 10 percent of the company's outstanding shares of common stock on a fully diluted basis, or approximately 5,900,000 shares, inclusive of the 1,000,000 shares repurchased in August.

By Thomas J. Theobald



ActiveX Intranet

Building ActiveX Controls for the Corporate Intranet

Delphi is an excellent construction yard for ActiveX controls, but little attention has been paid to ActiveX for Web applications since their splashy failure in the industry. As a general-use Internet application delivery system, ActiveX is unsatisfactory. The size, platform-specific nature, and potentially hazardous content (from unknown suppliers), all conspire to deny it wide acceptance as an Internet platform. Few users would appreciate the 20 minutes of download time, simply to have the application they wanted crash — and take their machine with it.

These problems, however, are essentially alleviated within the corporate firewall. Size and platform independence, the primary failings of ActiveX, don't matter much within a corporation where 10-megabit network connections and Win32 desktops hold sway. Additionally, with site-specific security measures available, supply of Active content to a user's desktop can be limited to an organization's "trusted" sites. In this context, ActiveX offers several benefits not available to Java, its primary competition:

- **It's cached.** OCX files brought down from the Web are brought once, assuming cache-time and disk allotment aren't exceeded. Java code must be downloaded and run with each hit to a Java-enabled site.
- **It's fast.** ActiveX files are compiled code, and run natively as extensions to the clients' browsers. Java, to date, is interpreted (there are some exceptions for JIT compilers and the JBuilder native compiler), and takes between five and 20 times the amount of time to provide identical functionality if achievable.
- **It can be 100 percent Delphi.** One of the things I dislike about Java is that — it's Java. I took up Delphi specifically because I didn't want to learn C. Perhaps I'm lazy, but I am reticent to give up the skills it's taken me a few years to develop. JBuilder is a great product, and I do see a future in Java, but I don't want to learn the syntax. Installed development shops that have a lot invested in Delphi will find the ActiveX route a potential gold mine in resource usage.

It also shares some benefits with Java:

- **It's instantly updated.** As soon as a new OCX file is delivered to the Web site, new hits draw an updated version of the application.
- **It's thin.** That's right, thin. Although even a small ActiveX application deploys at around a half a megabyte (the sample application in this article deploys at 1078KB), when deployed with run-time packages, that total can shrink to something really puny (the sample mentioned deploys to 337KB under run-time packages). There's an additional benefit in using CAB file compression (more on this later).
- **It can be n-tier.** In fact, I recommend this highly. Because the user has a network connection already (otherwise he or she would not be able to tag your Web site), there isn't anything that should prevent your application from being aware of a middle-tier server. Using *TClientDataSet* in your ActiveX application provides two benefits: It maintains the "Ivory Tower" of data control, and further "thins" the application by not requiring a BDE installation or configuration (all this is maintained at the middle tier). If the only requirements are the OCX, some run-time packages, and a copy of Dbclient.dll, your client should be getting downright skinny.

Design

Although it's outside the scope of this discussion, I'd like to stress that design should be the most important aspect of any application effort. Generally, the better the design, the better the final product; a

poor design will yield a poor application. Unfortunately, too many developers start coding with little or no planning.

ActiveX applications generally require more effort than conventional applications when it comes to analysis and design. Here are some considerations:

- Will the control be solely for Web deployment, or will it be an add-in for other applications? If you're going to pass the control to other developers to use in other tools, you may need to surface some properties and/or methods to make it useful.
- What data access route will you take? The Dbclient DLL to connect to a middle tier? Full BDE installation? Will you need to deploy OLEnterprise or MIDAS? Do you even need a database for this application? This is especially important for ActiveX controls, because CAB deployment of OLEnterprise and the BDE require a little more attention than simply shipping disks and/or checking a box in InstallShield.
- Does the application contain many features that might not be used often? The concern is keeping the client thin. If some of the functionality of the application (e.g. maintenance forms) won't be used often, pack it away in a DLL and include it in a separate CAB.

Development

First things first: If you want a multiple-document interface (MDI) application, ActiveForms aren't going to be much help. If you already have a means by which to emulate MDI applications in an SDI environment, you'll be all right, but ActiveForms don't have a *FormStyle* property. If you're good enough with Delphi to descend your own class that supports MDI, you probably don't need to be reading this, so I'm not going to cover it. Because the likelihood is that the application will get tested a few times before completion, we'll need to set up the **Web Deployment Options** from the **Project** menu. This will allow us to determine where the application will get dropped by Delphi. It also gives the developer an opportunity to isolate the output from the code directory. Once the settings are satisfactory, the developer can start coding.

We'll probably need some kind of Web server to offer our still-in-development project for our own review. Most of us won't have access to a dedicated test-bed Web server. For this reason, I recommend doing ActiveX Web development on an NT machine with Personal Web Server or IIS. Personal Web Server is available from the Microsoft Web site as part of the NT 4 Option Pack at http://www.microsoft.com/windows/downloads/winntw.asp. There is a version for Windows 95 available from the same site, but I don't recommend doing this kind of work in Windows 95 — certainly not with less than 32MB of RAM.

Using Personal Web Server is fairly straightforward, and it comes in handy for many kinds of design work. It also makes publishing documentation on the code you write fairly easy, and makes it available to you for instant changes.

So how do you develop an ActiveX application? The answer is simple: You don't. Just develop your application as a standard stand-alone Delphi application, using the aforementioned considerations. Then, when it's finished, create an ActiveForm shell. Close your finished project, and create a simple ActiveForm, i.e. select **File | New**, then choose **ActiveForm** from the ActiveX page of the Object Repository. Add the files from your normal project to the new ActiveForm project. Delphi creates a new ActiveForm project for you (in many ways, it's similar to a DLL project). Then, simply create the main form of your finished application in the *OnCreate* event handler of the ActiveForm. Setting that form's parent to the ActiveForm and maximizing it will run your application within the OCX.

This is *great* when it comes to producing a quick-and-dirty demonstration of an existing application, and the client wants to see a Web-deployment proof-of-concept. You can have a browser-based version of your application in minutes. This example was created over an omelet at IHOP. It might *not* be so great when you think about possible application architecture issues that haven't considered OCX deployment — particularly if this application wasn't originally intended for Web deployment. Chances are some "weight" will need to be removed (by breaking functionality into DLLs, etc.).

There are some differences between normal development under Delphi and Web deployment. For example, from **Run | Parameters**, the developer will need to locate the browser (this article was written with IE4 in mind; Netscape users can change paths as appropriate) and insert it in the **Host application** drop-down. This feature is available both in OCX and DLL development, and allows debugging of your control/application through the host application. (Usually, IE will be found at C:/Program Files/Plus!/Microsoft Internet):

- **Run | Parameters** should also be set to the HTM file deployed by Delphi when the **Web Deploy** action is chosen. This isn't necessary if you already have a link to it built into your default page, but can save you from following several tags every time you try running something.
- Get accustomed to using **Compile** or **Build All** from the menu, then choosing **Web Deploy** to guarantee you get the latest version of your control. Trusty old F9 might not be what you're looking for here.
- Only the main form needs to be of Active content. All others can be standard *TForm* descendants. If you add more active forms, Delphi will adjust the .HTM file appropriately, and you'll have a bunch of empty sockets in a very long page — not fatal, but messy.
- This method requires slightly more complex project management, and presents difficult unit and system testing of the application.

Build Task One: Create an ActiveForm Shell

To make this easy, we'll copy a demonstration project from the Delphi 3 installation and turn it into a Web-based application. First, create a new directory for your ActiveX project and copy the contents of the demonstration project into it. I've chosen TeeChart (Program files\Borland\Delphi 3\Demos\TEECHART), because it's flashy and somewhat elaborate. (This article discusses Delphi 3, but this technique works for Delphi versions 3 and higher.)

In Delphi, choose **File | New**, then **ActiveForm** from the ActiveX page of the New Items dialog box (see Figure 1). Delphi will gener-

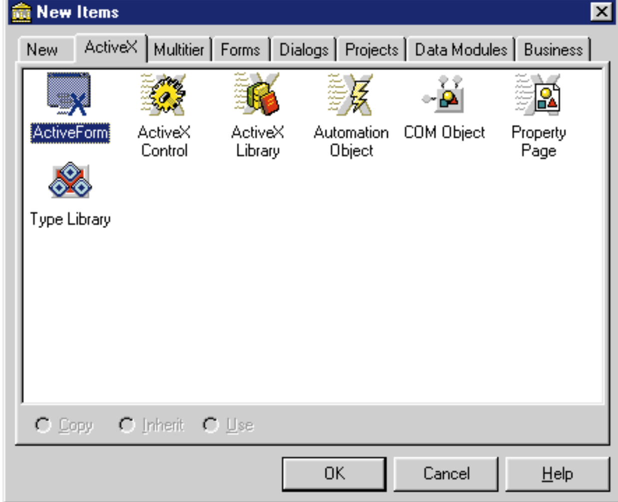


Figure 1: Creating an ActiveForm shell.

ate a new ActiveForm, a type library (which you won't need to worry about for the purposes of this exercise), and a project file (which you also won't need to worry much about). Save all these files into the new directory you just created, and give them some meaningful names. (The projects discussed in this article are available for download; see the end of this article for details.)

View the project manager (or the project source), and add to the ActiveForm project all the files you chose from the demonstration project. In the ActiveForm, add a private variable to the ActiveForm's class definition corresponding to the class of the main form of the demonstration application:

```
TAXTeeChartDemo = class(TActiveForm, IAXTeeChartDemo)
  procedure FormCreate(Sender: TObject);
private
  // Private declarations.
  FEvents: IAXTeeChartDemoEvents;
  // A var must be included in the type declaration
  // for this stunt.
  frmTeeMain: TTeeMainForm;
  procedure ActivateEvent(Sender: TObject);
  ...
```

You'll also need to add the appropriate unit to the ActiveForm's **uses** statement, TeeMain in this case. Now, generate a *FormCreate* event in the ActiveForm, and from there, create an instance of the form into the variable you just declared (e.g. *frmTeeMain*). Finish by parenting it and giving it some formatting to fit the OCX control:

```
procedure TAXTeeChartDemo.FormCreate(Sender: TObject);
begin
  frmTeeMain := TTeeMainForm.Create(Self);
  frmTeeMain.Parent := Self;
  frmTeeMain.Height := Self.Height;
  frmTeeMain.Width := Self.Width;
  frmTeeMain.Show;
end;
```

Now, go to the **Project** menu and choose **Web Deployment Options**. We'll take this a page at a time later in the "Deployment" section. The Project page is the most important right now. Once the top three entries are made (again, see the section titled "Deployment"), accept this dialog box and select **Project | Web Deploy**. There's no speed button for this command, so you'll become familiar with Alt P D. Once Web deployment is complete, you can tag the HTM file directly from Explorer, or follow a link you build into your Web site (see the section titled "Link It All In"). The result is shown in Figure 2.

Generally, following the route of a normal application works just fine — with no concern over browser location, HTM links, etc. Simply build the application as you want it. When finished, re-create the main form as an ActiveForm by taking a blank ActiveForm and pasting the routines and controls from your original main form into it. Adding the remainder of the files from the project finishes the task.

Build Task Two: Transplanting the Code

Again, build your application as a standard Win32 application with no regard to Web planning, and test it until you feel it works properly. Then create a new project directory for constructing your new ActiveX application. Next, select **File | New** and choose **ActiveForm** from the ActiveX page of the New Items dialog box. Save the resulting files, giving them and the components sensible names.

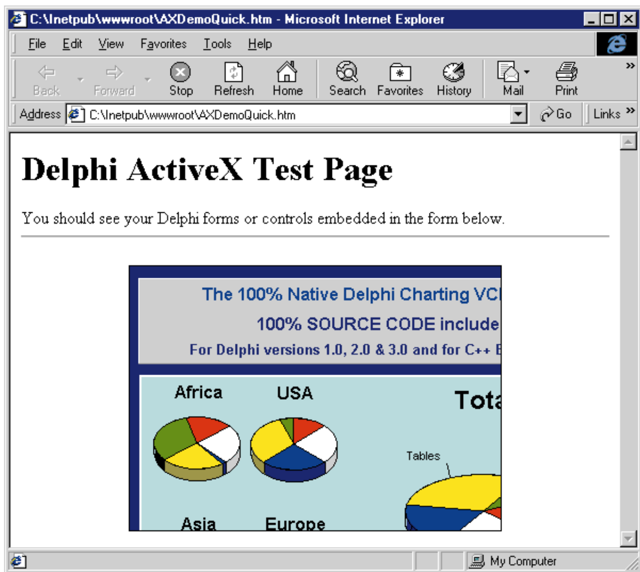


Figure 2: The TeeChart demonstration application as an ActiveForm.

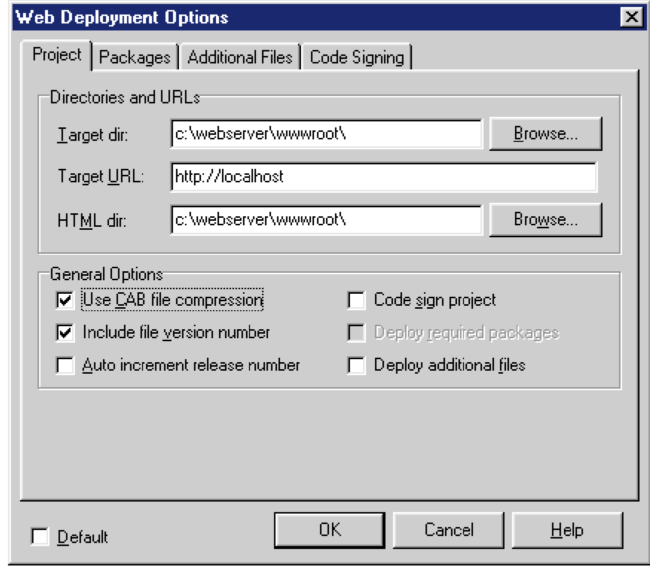


Figure 3: Web deployment options.

Now, copy the contents of the application from step 1 into your new application's directory. Open the main form of the original application while you have the ActiveForm project open. Copy the components, events, and methods of the main form into the ActiveForm. This may require re-creating the events (such as button clicks, etc.), but because you've already written the code, creating the event handlers should be a pretty rudimentary task. Don't modify anything Delphi generated for you, unless you know exactly what you're doing. Continue with deployment as described in "Build Task One."

Deployment

Okay, we've got the application built. Now comes the part where we have some tricks to pull. Deployment of an ActiveX application is fairly simple, but there will be some problems with how Delphi creates some of the output.

The Web Deployment Options dialog box is where you tell Delphi how you want your application deployed on a Web page (see Figure 3). It's a good idea to complete this early — as one of

the first steps in creating your Web application — so you don't have to worry about it once the code starts flying.

The options begin with the Project page, where you determine the physical location of the deployed project. **Target dir** indicates where the CAB or OCX files will be placed. You can supply either a direct drive-mapped path or a UNC path, e.g. C:\Projects\ActiveX Test\Output. **HTML dir** indicates where the .HTM file will go as a drive-mapped path or UNC, e.g. C:\Projects\ActiveX Test\HTML. **Target URL** is the listing that will be placed in the .HTM file as a URL path on the Web server, e.g. http://LocalHost/ActiveXApplication.

There are also **General Options** to consider on this page: **Use CAB file compression** ought to be used, regardless of the size of the OCX file. Simply put, this crunches down the size of the download to a smaller file (generally between 50-75 percent of its original size), and makes it a little faster to download. This is particularly useful when dealing with users who will be accessing your application with a modem.

Regarding the **Code sign project** option: Code signing isn't terribly important within an intranet, but a couple of things need to be pointed out about code signing. First, it isn't terribly expensive, so if you intend to serve this application outside the firewall, get a signature. Corporations such as Verisign can supply digital signature registration at reasonable cost to both large corporations and individual developers. Second, once a signature has been obtained, guard it jealously. Only one developer in a shop should have access to the digital signature, and it should be considered volatile property of the company. I'm no specialist on liability law, but I can certainly tell you that a disgruntled employee who Web-deploys "format C:\" with a code signature from your company is going to cause a lot of problems.

The **Deploy additional files** option will probably be necessary, especially if you build your application with run-time packages or DLL files. The Packages page (see Figure 4) will specify those packages used by your ActiveForm project. Each package will be individually configurable to determine whether it's a **Compress in project CAB**, or **Compress in separate CAB** file. It's recommended you use separate CAB files to avoid having to send an overly-large project file, if all that has changed was the project. You can also determine whether to **Use file VersionInfo**, and provide **Target directory** and **Target URL** information. By default, package CAB files will use the same target URL and directory as the project CAB. This doesn't change the fact that they'll probably be loaded into one of the cache directories, unless you send them elsewhere by specifying their destination in the INF file.

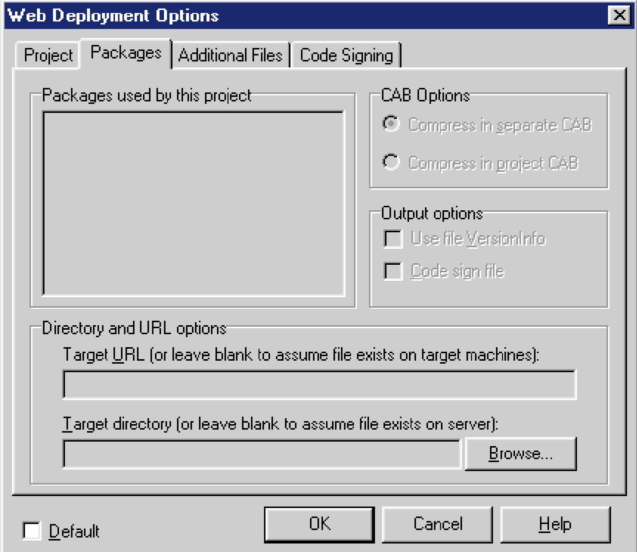


Figure 4: The Packages page specifies packages used by your ActiveForm project.

The Additional Files page is, where necessary, non-package files are included (see Figure 5). These have the same options and notes as package files; basically, this page is included to allow the developer to add run-time files that Delphi isn't normally aware of. This is where the BDE and OLEnterprise installation CAB files should be added to the project, if they're required.

The developer may wish to consider including a configuration file with the OCX application, to allow the user to connect directly to a server via TCP/IP upon running the application. In the ideal circumstance, the application reads the supplied configuration file, sets its *TDatabase*, *TRemoteServer*, or *TMIDASConnection* control appropriately, and saves its settings to the registry. The TCP/IP stack should already be present (or the user wouldn't have been able to

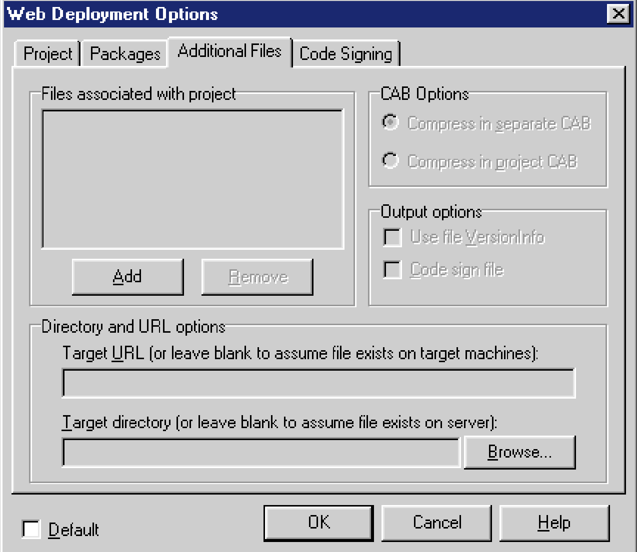


Figure 5: The Additional Files page is where needed non-package files are added.

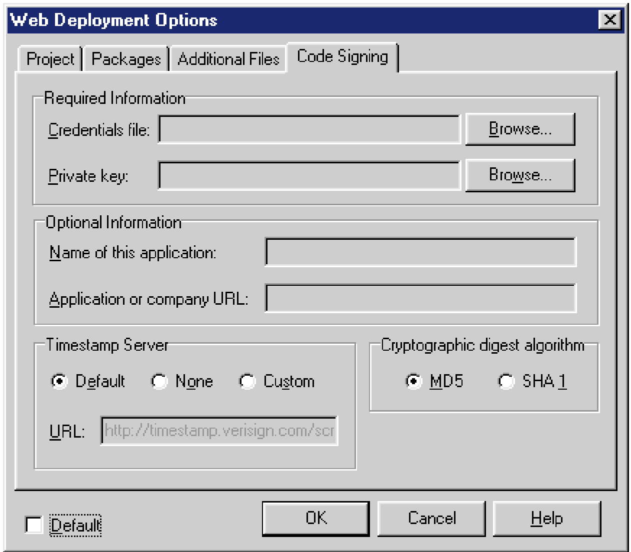


Figure 6: Code signing is provided to allow the developer to specify the details of the signature file.

download the application), and the application can check for its settings in the registry before reading the configuration file.

Code signing is provided to allow the developer to specify the details of the signature file (see Figure 6). Again, any use of a code signature should be carefully governed. Consider it the endorsement of your company (or yourself, if you're a sole developer).

That's Web deployment configuration. Not too bad once you've been through it once or twice. There'll be several additional files generated in the project. MyApplication.INF is an information file containing an entry for every file to be deployed to the Web server for your application to run. It'll look something like this:

```
;Delphi-generated INF file for UnzipProjX.ocx
[Add.Code]
UnzipProjX.ocx=UnzipProjX.ocx
dunzip32.dll=dunzip32.dll

[UnzipProjX.ocx]
file=http://localhost/UnzipProjX.cab
clsid={ 33F47DC3-A735-11CF-A090-00A024B18D7A }
RegisterServer=yes
FileVersion=1,0,0,0

[dunzip32.dll]
file=http:// localhost/dunzip32.cab
FileVersion=1,0,0,0
DestDir=11
```

This file contains information on where to find the individual files to be deployed with your OCX-based application. The `file=` entry of each section denotes the CAB file containing the file for this section. The `clsid` entry refers to the GUID of any ActiveX file (in the previous example, only the application had Active content to require a GUID). `RegisterServer` specifies whether the control should be registered with the operating system. `FileVersion` denotes whether the CAB file will be downloaded to the client. If the CAB file found has a version earlier than the `FileVersion` entry, it won't be sent.

Delphi won't place a `DestDir` entry in your INF file; the developer must add it by hand. `DestDir` indicates the location of the file installed by the browser. By default, the OCX file will be dropped into one of the browser's cache directories; should other files be placed there as well, the application may not know where to find them. `DestDir` can specify different directories, other than the cache directories. Specifically, a value of 10 indicates the Windows OS directory (e.g. C:\Windows on a default Win95 machine, or C:\Winnt on a standard NT box), while a value of 11 directs the file to the system directory (e.g. C:\Windows\System, or C:\Winnt\System32, respectively). This is very important, as package files and DLLs must be found by the application at run time.

The .HTM file (e.g. MyApplication.HTM) is the HTML "wrapper" that will refer to the OCX file generated by Delphi. It will appear like this:

```
<HTML>
<H1> Delphi ActiveX Test Page </H1><p>
You should see your Delphi forms/controls in the form below.
<HR><center><p>
<OBJECT classid="clsid: [your guid here]" codebase=
"c:\MyAppDirectory\ActiveFormName.cab#version=1,0,0,0"
width=[width of your ActiveForm]
height=[height of your ActiveForm]
align=center hspace=0 vspace=0>
</OBJECT>
</HTML>
```

This defines how your ActiveForm application will display itself in the user's browser. Referencing this HTML file with a link from another page is all that is necessary to connect your new application with the outside world.

If you find more than one <OBJECT> reference in your HTM, it probably indicates you've generated more than one ActiveForm in your application. This is a mistake I made early on. Only the main form needs to be of the Active variety. All the other forms in your application can be standard *TForm* descendants with no special attributes.

Other files created as a result of your Web deployment will include the usual DCUs (unless you're a fan of OBJ files), RES, DOF, and backups. Additionally, you'll see an OCX and a CAB file in the target directory. If the output directory is one that doesn't get linked directly by your Web server, the files you'll need to place in reach of the Web server are any CAB files, MyProject.HTM, and MyProject.INF. The only link necessary to your project is to the HTM file. The CAB and INF files will take care of themselves.

Link It All In

Now that we've gone over the bits and pieces, we only need to link to our HTM file, and hit it with a browser. Internet Explorer users won't have a problem with Active content (other than the security settings described later in this article), but Netscape users will need a plug-in to allow them to run an ActiveX. A commonly used plug-in for this purpose is named ScriptActive, and a 30-day trial version is available at http://www.ncompasslabs.com/products.htm.

Security in IE will need to be adjusted to accommodate your testing. I've found the easiest way to accomplish this is to make your host machine a "trusted site," and give trusted sites rights to do whatever they please. The clients downloading your finished product can probably do this as well, because we're talking about intranet deployments, but anyone hitting your control from the Internet might not be so trusting. They will need to adjust specifically to deal with your application.

The security settings in Internet Explorer can be accessed from the **View** menu, under **Internet Options**. The second page of the Internet Options dialog box is devoted to security. In it, any user who needs to access the ActiveX application will need to be able to both download and script the ActiveX control containing the application. Depending on whether the ActiveX application is signed, the security settings for the appropriate status of the control should be set to enable or prompt. Generally, I recommend prompt, but you might get fed up with having to answer "yes" all the time while developing. It's your call how you handle this, but do be careful about accidentally enabling download of content from anywhere on the Internet.

IE4 users have several security zones from which to choose. As this topic is geared toward the corporate intranet, low security settings on the local intranet zone are recommended, and, as I've previously mentioned, it probably would be easiest to add your local host to the trusted sites list. For any Internet zone, however, I certainly recommend that the browser should at least prompt the user for download and scripting of content from any site not in the trusted list.

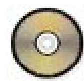
Conclusion

Hopefully, I've made this fairly simple. Creating an ActiveX application is different and sometimes frustrating, but generally

not much different from creating a standard Windows application. I've found configuration of the developer's workstation and deployment issues constitute the main additional effort when dealing with ActiveX applications. With luck, you'll be publishing on the Web in a few hours, and your boss will be more than happy with the splashy content you'll be able to generate. I wish you luck. E-mail me if you have any great success stories with this; I'd like to hear about them. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\JUN\DI9906TT.

Tom Theobald is a senior software developer with Segue Technologies of Alexandria, VA. He began his career with computers as a NetWare engineer, moving later to include NT and Lotus Notes among his acquired skill set. After moving into and cleaning up one too many messes, he decided "to hell with this" and took up software development. Now a certified Delphi instructor, he makes his trade helping large corporations and government agencies acquire a more Zen-like attitude toward software development. His favorite contribution to Inprise was in providing First Union a realistic and achievable image of what the Delphi/Iprise tool set could provide during training of their first cadre. He can be reached at theobaldt@seguetech.com with any business inquiries, questions, or comments. Death threats and other matters of a personal nature can be forwarded to eviltom@worldnet.att.net.



By Cary Jensen, Ph.D.

ASCII Made Easy

Manipulating ASCII Files with the Table Component

ASCII files, the veritable antitheses of database server-based data, are nonetheless important files for receiving data from a wide range of sources, as well as an effective medium for sharing data with others. This month's "DBNavigator" takes a look at several mechanisms that Delphi provides for reading and writing ASCII files using the *TTable* class.

Using tables to read and write ASCII files is not the only solution provided by Delphi. For example, you can use typed or untyped files and perform basic file input/output (I/O) using functions and procedures of the System unit, e.g. *AssignFile*, *ReadLn*, *WriteBuffer*, etc. An advantage to this approach is that applications created using only these techniques do not require the Borland Database Engine (BDE). However, for the database developer whose applications must use the BDE already, the *TTable* class provides a number of handy and relatively easy-to-use methods for using and creating ASCII data.

Overview of ASCII Files

ASCII, which stands for the American Standard Code for Information Interchange, uses 8-bit integers to represent the characters and control codes commonly encountered in data. Each ASCII character or control code has a decimal equivalent. For example, the decimal value 65 represents an uppercase A, 97 represents a lowercase a, and 10 represents a line feed.

An ASCII file is a file that contains only ASCII characters, and is normally considered to hold only text. What makes ASCII files so interesting is that they can be created from almost any source, from "big iron" mainframes to the very earliest personal computers. This makes them a convenient format for importing data from another source, as well as exporting data to be used by some non-database program, such as a spreadsheet or word processor.

In this article, I will limit my discussion to three types of ASCII files: delimited, fixed-length, and simple text. Delimited ASCII files contain two or more data fields. A single character, called a separator, separates these fields. The most common character used for this purpose is the comma. The string data within a delimited file is identified by being preceded and followed by a delimiting character (or delimiter), most often the double quotation mark. The following is an example of what a delimited file may look like:

```
"Plumber","Mark",1000,5.2,3/4/95
"Ramerez","Pablo",1050,16.75,8/15/97
"Johannson","Christina",998,-25.25,9/1/98
```

Fixed-length files don't use separators or delimiters. Instead, the data fields are defined by their position within a record. While the records in a delimited file are of a variable length, in a fixed-length file, each record is the same length. The following is an example of a fixed-length file:

```
Plumber    Mark        1000    5.23/4/95
Ramerez    Pablo       1050    16.758/15/97
Johannson  Christina   998     -25.259/1/98
```

Simple text files do not have individual fields, but are instead composed of a sequence of characters.

An HTML file is a good example. The following is an example of a portion of a simple text file:

```
*************************************************************
                  DELPHI 4 RELEASE NOTES
*************************************************************


This file contains last-minute information about Delphi
4 and additional information that enhances the
usability of Delphi. We recommend you read this entire
file before using Delphi 4.
```

Simple text files are the easiest to work with, and therefore are discussed first.

Reading Simple Text Files

To read a simple text file using a Table component, set the Table's *TableName* property to the name of the text file and open the Table. The *TableName* property can either include the fully qualified path, or you can enter the path in the *DatabaseName* property and only the file name in the *TableName* property. Figure 1 shows the Table and DBGrid page of the example TEXTFILE project with Delphi's README.TXT file loaded into a Table.

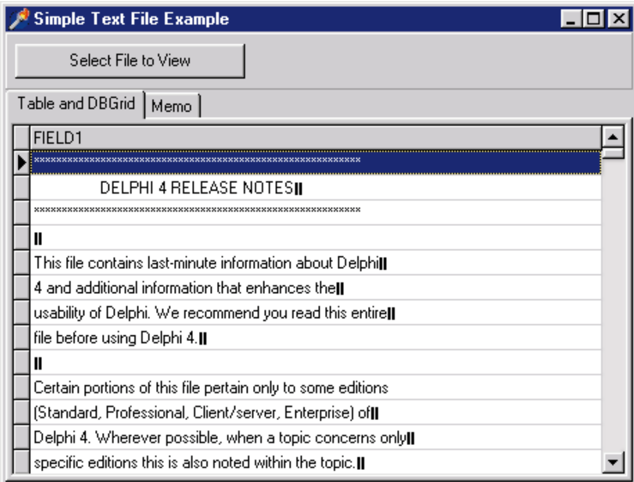


Figure 1: Any text file can be opened using a Table and displayed in a DBGrid.

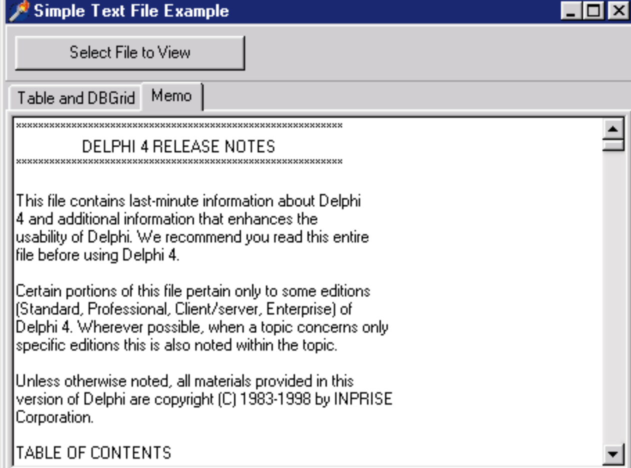


Figure 2: A Memo is often the best control for displaying a simple text file to your users.

(The projects discussed in this article are available for download; see the end of this article for details.)

While using a Table component to access a text file is simple, it's generally used only when your code is going to work with the text in the text file line-by-line. As you can see in Figure 1, the DBGrid, although capable of displaying the text in the Table as a single field, doesn't provide a view of the data suitable for an end user. If you merely want to display text from a simple text file, a Memo component (or other control that encapsulates *TStrings*) is better. This can be seen in Figure 2, which shows a Memo component from the Memo page of the TEXTFILE project.

All the code associated with the TEXTFILE project can be found in the *OnClick* event handler for the **Select File to View** button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then begin
    Memo1.Lines.Clear;
    Memo1.Lines.LoadFromFile(OpenDialog1.Filename);
    Table1.Close;
    Table1.Tablename := OpenDialog1.Filename;
    Table1.Open;
  end;
end;
```

The Table and Memo components in the TEXTFILE project are configured to be read-only. It's possible, however, to make changes to the data in a simple text file using a Table or a *TStrings* object. Changes made to a Table are posted on a line-by-line basis (unless *CachedUpdates* is used), while changes made to the text in a *TStrings* must be saved by calling the *SaveToFile* method.

Using Fixed-length ASCII Files

The use of a fixed-length ASCII file requires a schema file. A schema file contains a description of your ASCII file's metadata, including field names, types, and sizes. The following is an example:

```
[FIXED]
Filetype=Fixed
CharSet=ASCII
Field1=LastName,Char,20,00,00
Field2=FirstName,Char,20,00,20
Field3=IDNumber,LongInt,11,00,40
Field4=SomeReal,Float,20,02,51
Field5=SomeDate,Date,11,00,71
```

You will immediately recognize this file as being in the same format as an INI file. The file begins with the name of the file that it describes. Delphi assumes the file extension of this file is .TXT. Furthermore, the schema file must use the same file name as your ASCII file, but have the file extension .SCH.

On the line following the ASCII file name is the entry `FileType=`, which is followed by the type of ASCII file. This type can either be the value `Fixed` or `Varying` (the value is not case-sensitive). Next, you must identify the character set. In the US, this value is almost always `ASCII`.

The remainder of the schema file contains descriptions of each of the fields in the ASCII file. Each field is described on a separate line that begins `FieldN=`, where `N` is the ordinal position of the field in the table structure. Consequently, the first field is defined by a line that begins `Field1`, the second field by a line that begins `Field2`, and so forth.

The definition of each field includes five parts: the field name, the field type, the maximum size of the field, the number of decimal places (this applies only to floating point value fields), and the column in which the field begins. The acceptable field type values are shown in Figure 3.

Field Type	Use For
Char	Strings
Float	64-bit floating point numbers
Number	16-bit integer
Bool	Boolean values
LongInt	32-bit long integers
Date	Date fields
Time	Time fields
TimeStamp	Date + Time fields

Figure 3: The values for the field type part of the field definition.

In the preceding schema file example, the first field is declared to have the name LastName, be a string field, contain a maximum of 20 characters, include a meaningless 00 in the decimal part, and finally be found with 0 columns offset from the start of the record. The fourth field, by comparison, is declared to have the name SomeReal, be a floating point number, have a maximum of 20 characters in its value, include two decimal places in its display, and whose value can be found starting at column 52 (offset 51 characters from the beginning of the record).

You can create your fixed-length schema file manually, or you can have Delphi generate it for you. Creating a schema file manually involves entering the properly formatted file definition using any text editor, such as Notepad or WordPad. Just make sure you save the file as a text file using the same name as your data file, but with the extension .SCH.

To have Delphi create your fixed-length schema file, use *BatchMove* (either a BatchMove component, or the *BatchMove* method of the *TTable* class.). This requires that you already have a BDE-supported file type that contains the data from which you want to create a fixed-length ASCII file. If such a file does not already exist, you can create one using the Database Desktop application that ships with Delphi. From the Database Desktop select **File | New | Table**, select Paradox (or any other file type you are familiar with), and then enter the structure of your table in the Create dialog box, as shown in Figure 4.

Once you have a table with the desired structure, use the following steps to create your schema file:

1) In Delphi, create a new application.
2) Place two Table components on your form.
3) Using the *DatabaseName* and *TableName* properties of Table1, select the data file that contains the structure from which you want to create a schema file. For example, set *DatabaseName* to DBDEMOS and *TableName* to CUSTOMER.DB.
4) Select Table2, and set its *TableName* property to the fully qualified file name of the ASCII file for which you want to create a schema file. Include the entire file path, as well as the .TXT file extension. For example, set *TableName* to C:\Program Files\Borland\Delphi4\Projects\CUSTOMER.TXT. Also, set the *TableType* property of Table2 to *ttASCII*.
5) Now place a Button component on your main form, and double-click it to create an *OnClick* event handler. Add one statement to the event handler created by Delphi:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Table2.BatchMove(Table1,batCopy);
end;
```

6) Run your application and click the button.

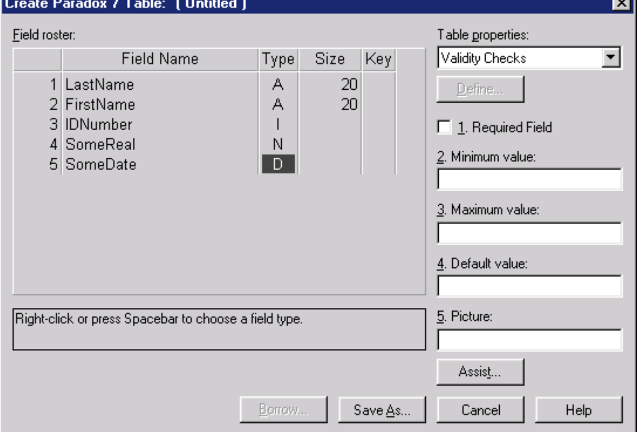


Figure 4: The Create dialog box in the Database Desktop.

That's it; you've created a schema file. If you perform the preceding steps using the CUSTOMER.DB table in the DBDEMOS database, you'll find the following schema file in the directory you entered for your *TableName*:

```
[CUSTOMER]
Filetype=Fixed
CharSet=ascii
Field1=CustNo,Float,20,02,00
Field2=Company,Char,30,00,20
Field3=Addr1,Char,30,00,50
Field4=Addr2,Char,30,00,80
Field5=City,Char,15,00,110
Field6=State,Char,20,00,125
Field7=Zip,Char,10,00,145
Field8=Country,Char,20,00,155
Field9=Phone,Char,15,00,175
Field10=FAX,Char,15,00,190
Field11=TaxRate,Float,20,02,205
Field12=Contact,Char,20,00,225
Field13=LastInvoiceDate,TimeStamp,30,00,245
```

Unlike simple text files, which cannot be edited very easily in data-aware controls such as the DBGrid, fixed-length ASCII files can easily be used in any data-aware control. The only limitation is that the files have no indexes, and therefore cannot be sorted. All records you add are appended to the end of the file. Also, because there are no indexes, you cannot define record uniqueness based on a unique key. Finally, these files cannot be shared because they have no native locking mechanism.

Using Delimited ASCII Files

Delimited ASCII files are only slightly more difficult to use than fixed-length ASCII files, in part because Delphi will not generate a schema file for you. Instead, you must write the schema file for a delimited ASCII file yourself.

There are only three differences between schema files created for delimited ASCII files and those you use for fixed-length files. The first, and most obvious, is the `FileType` entry. While you set this entry to `Fixed` for a fixed-length file, you set it to `Varying` for a delimited file.

The other two differences involve defining the field separator and the string delimiter. The schema file for a delimited ASCII file contains two additional lines immediately following the `FileType=` entry. The first of these is the `Separator=` entry. You use this to define the char-

acter used to separate the fields. As mentioned earlier, this character is often the comma. The second entry is `Delimiter=`, which you use to define the character used to enclose strings. In most cases, this will be the double quote character. The following is an example of a schema file for a delimited ASCII file:

```
[DELIMIT]
FileType=Varying
Separator=,
Delimiter="
CharSet=ascii
Field1=LastName,Char,20,00,00
Field2=FirstName,Char,20,00,20
Field3=IDNumber,LongInt,11,00,40
Field4=SomeReal,Float,11,02,51
Field5=SomeDate,Date,11,00,62
```

While Delphi won't generate a delimited schema file for you, the similarity between the two schema file types provides you with some assistance. Specifically, using the steps given earlier, you can create a fixed-length schema file for your delimited table structure, and then make the three modifications just described to the Delphi-generated schema file.

The use of both a fixed-length ASCII file and a delimited ASCII file is demonstrated in the SCHEMA project, shown in Figure 5. When you view the FIXED.TXT table, the FIXED.SCH schema file permits Delphi to read the following ASCII file:

```
Plumber         Mark        1000       5.203/4/1995
Ramerez         Pablo       1050       16.758/15/1997
Johannson       Christina   998        -25.259/1/1998
```

When you view the DELIMIT.TXT ASCII file, the DELIMIT.SCH schema file permits you to view this file:

```
"Plumber","Mark",1000,5.2,3/4/95
"Ramerez","Pablo",1050,16.75,8/15/97
"Johannson","Christina",998,-25.25,9/1/98
```

Final Notes

By default, the BDE is configured to display two-digit years in dates. So, if you write a date field to an ASCII file, only the last two digits of the year are stored. In most cases, you will want to make sure that Delphi writes all four-year digits of your date data. To do this, you must update the Date format setting using the BDE Administrator located in your system's Control Panel (see Figure 6).

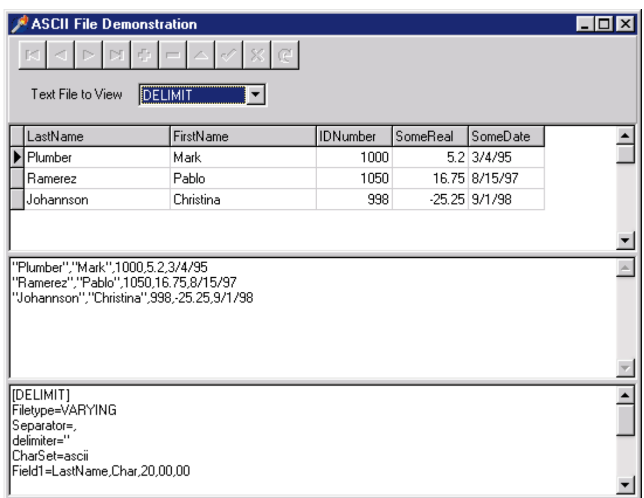


Figure 5: The SCHEMA project main form.

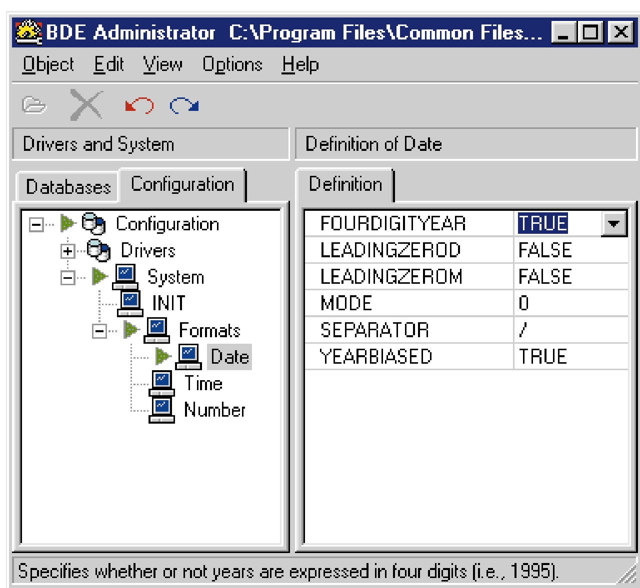


Figure 6: Setting the Date format in the BDE Administrator.

Also, to display the four-digit year in a date field associated with an ASCII file, you must instantiate the *TFields* associated with your Table component, and set the *DisplayFormat* property of your Date fields. For example, setting the *DisplayFormat* of a *TDateField* object to "m/d/yyyy" causes all four digits of the year to be displayed.

Another peculiarity of using ASCII tables is that (at least with my copy of Delphi 4) the Table's *Active* property must be set to False at application startup. While you may want to set the *Active* property to True during design time — so that you can see your data as you work — you should set *Active* to False before running your application. Use the *OnCreate* event handler, or some other similar event handler, to set the *Active* property to True for your tables that use ASCII data. This code may look something like the following:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Table1.Open;
end;
```

Conclusion

ASCII files, while not used in day-to-day database applications, provide an effective medium for passing data to and from Delphi and other applications. Not only can a Table component be used to read simple text files, but with the addition of a schema file, the Table component can read, display, and write delimited or fixed-length ASCII files. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\JUN\DI9906CJ.

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

By Keith Wood

An HTML Generator

Part II: Building with *IHTML* Objects

Last month, we introduced the *IHTML* objects. These provided an object-oriented approach to generating HTML from a Delphi program. They made use of an interface to define the generating ability, allowing any object to produce HTML.

This month, we'll make use of these objects in three applications. The first converts a Pascal source file into an HTML page, just as you would see in the Delphi editor. Next, we'll produce an HTML table that contains a directory structure. Finally, we'll create a program that allows us to generate frameset documents in a more visual way.

Pascal Source

To publish a Pascal source file on the Web, all we need to do is copy the file, add a basic HTML header and body, and surround the actual text with a <PRE> (preformatted) tag. But plain text like this is not as easy to read or skim through as when it's presented in the Delphi editor. In the editor, we can invoke syntax highlighting to allow us to easily distinguish the elements of the Pascal code. Wouldn't it be nice if the Web version looked just the same?

To accomplish this, we need to be able to recognize the various parts of the Pascal code, such as reserved words, strings, numbers, etc. For this, we need a parser. A parser breaks up a stream of characters into individual tokens, or groups of related characters. It then returns each one to the calling program so it can be dealt with.

Delphi provides a basic parser, *TParser*, in the Classes unit. It's designed for use with the reading of components' states from the .DFM file or from the executable. It also functions quite well with Pascal source files, breaking the text into tokens identified as symbols (basically, Pascal reserved words and identifiers), strings (enclosed in quotes), integers and floats, and all other characters (punctuation and operators). White space is ignored. This last point is critical; by ignoring white space, we cannot display the code in its original format. So we'll have to roll our own parser, borrowing heavily from this example.

We declare slightly different tokens for our parser, corresponding to the items we want to highlight: reserved words, identifiers, comments, strings, and numbers. An important new property that we add to the parser allows us to retain all the white space in the file, treating each such character as a token in its own right. Setting *WantAsSource* to True returns all the text from the source file; setting it to False skips over any white space.

The parser works by reading a single character, and from that deciding what sort of token has been encountered. If the token consists of multiple characters, these are collected and returned as the *TokenString* value. An alphabetic character signals the start of an identifier. Once collated, its value is compared to a list of reserved words, and — if found — is flagged as such. This list is set up during the initialization for the parser unit.

Strings are a little more complex because they can include control characters, in the form `#nn`. Furthermore, when returning the tokens as source, we want to include the opening and closing quotes, but when processing as tokens, only these are stripped from the resulting value.

Comments are another problem area. The original *TParser* object doesn't deal with comments, because it was designed to parse the .DFM file contents. But to display the full source code, we need all the comments as well. For each style of comment encountered, we must continue reading characters until we find its end. This is not too hard for comments using braces ({ }), but requires a look-ahead for the other styles, because they each start with a two-character combination.

All the parser functionality is built into the PasParser unit, allowing it to be easily re-used in

other projects. First, you set up the input stream (such as directly from a file, or a memo control via a memory stream), then create the parser and pass it the stream. Finally, you continually call *NextToken*, and process the results until you reach the end of the stream.

Stylish HTML

Once we have our Pascal parser ready, we need to operate on the tokens returned from it. We want to generate an HTML document that preserves the syntax highlighting of Delphi. Fortunately, HTML 4 supports style sheets, allowing us to define each of the styles we require in one place in the document, then apply them throughout. If we ever want to change those styles, we only need to make a small modification. The styles that we implement are for reserved words, strings, comments, numbers, and a default style for the rest of the source code.

The cascading styles sheets used in HTML 4 provide the ability to specify a name for a style, then supply that name when we want the style used. To mark the section of text so displayed, we can surround it with the <SPAN> tag, which allows us to change styles in-line, as shown here:

```
<span class="reserved">if</span> sText =
<span class="string">'Yes'</span>
<span class="reserved">then</span>
...
```

The style sheet that corresponds to this could appear like the following. We create a class of <SPAN> tags with the name of our style, then specify its appearance. The default style applies to everything contained within the <PRE> tag:

```
<style type="text/css">
<!--
span.reserved { color: rgb(0,0,0); font-weight: bold }
span.string { color: rgb(0,128,0) }
span.comment { color: rgb(0,0,128); font-style: italic }
span.number { color: rgb(255,0,0) }
pre { color: rgb(0,0,0) }
-->
</style>
```

To retrieve the actual color schemes to be applied, we need to delve into the registry, down to where Delphi keeps track of such things. Under Delphi 3, the appropriate key is HKEY_CURRENT_USER\Software\Borland\Delphi\3.0\Highlight. Here, we find entries for each of the elements that can be highlighted. Each entry specifies the foreground and background colors (as integers), characters indicating font styles (B, U, and/or I), flags for using the default foreground or background colors, and the indexes within a color grid for the selected colors.

Under Delphi 4, things have changed quite a bit. We now need to look under the HKEY_CURRENT_USER\Software\Borland\Delphi\4.0\Editor\Highlight key, where we find further keys for each of the syntax elements. Within these lower levels, we find entries for the color grid indexes for the foreground and background colors, flags for using the default colors, and separate flags for each font style.

We retrieve these values during the initialization of the containing unit (IHTMLPas) by calling the *LoadFormats* routine. The code automatically knows which registry key to look for through conditional compiles based on the version of Delphi that is running it. The styles are saved in internal variables and are supplied to the calling program on request as a complete style sheet via the *PascalStyleSheet* function.

All that's left now is to step through the Pascal source file with the parser and add each token to a *THTMLText* object that has had a style of *tsPreformat* applied (see Figure 1). For each token, we check its type and apply one of the highlighting styles, if appropriate. This is done with the *THTMLDivision* object, passing False to the constructor to indicate that we want an in-line span rather than a block division. The token text must be "escaped" using the *EscapeText* function, to ensure that HTML control characters in the Pascal code are treated correctly. We then set the tag's class name to the correct value for the token and embed it in the basic text.

All this is encapsulated within the *THTMLPascal* object, which extends the *THTMLBase* object from the *IHTML* collection. It has a new property that allows us to specify the name of the file to be converted, and provides for setting this directly in the constructor.

The actual HTML generation occurs in the *AsHTML* method, as we would expect from an *IHTML* descendant. The output of the *THTMLPascal* object is merely that of the embedded *THTMLText* object that we've built from parsing the code. Once

```
{ Return the Pascal source file formatted as HTML. }
function THTMLPascal.AsHTML: string;
var
  htx: THTMLText;
  hdv: THTMLDivision;
  stm: TFileStream;
  psr: TPascalParser;
  sClass: string;
begin
  try
    { Preformat the rest of the source file. }
    htx := THTMLText.Create(tsPreformat, '');
    try
      { Open specified file. }
      stm := TFileStream.Create(FileName, fmOpenRead);
      { And prepare to parse it. }
      psr := TPascalParser.Create(stm, True);
      while psr.Token <> toEOF do begin
        { Set class of formatting from token type. }
        case psr.Token of
          toReserved: sClass := 'reserved';
          toString:   sClass := 'string';
          toComment:  sClass := 'comment';
          toNumber:   sClass := 'number';
          else        sClass := '';
        end;
        { Add (formatted) text to document. }
        if sClass <> '' then
          begin
            hdv := THTMLDivision.Create(False, '',
                     EscapeText(psr.TokenString));
            hdv.TagClass := sClass;
            htx.Add(hdv);
          end
        else
          htx.Add(THTMLText.Create(tsNormal,
                    EscapeText(psr.TokenString)));
        psr.NextToken;
      end;
    finally
      { Free resources. }
      psr.Free;
      stm.Free;
    end;
    Result := htx.AsHTML;
  finally
    htx.Free;
  end;
end;
```

Figure 1: Formatting the Pascal source as HTML.

added to an enclosing HTML document, the job is done. This is easily achieved, as shown in Figure 2.

The application that accompanies this article (available for download; see end of article for details) allows us to select a Pascal source file from anywhere on the disk, then generate an HTML version of it into the memo displayed on the screen. This can be reviewed before saving the contents to an output file. Finally, you can open your style-sheet-enabled browser and view the beautifully formatted code.

```
{ Create the HTML document. }
htm := THTMLDocument.Create('Pascal source for ' +
                            filFiles.FileName);
{ Add formatting for Pascal elements. }
htm.StyleSheet := PascalStyleSheet;
{ Add a heading. }
htm.Add(THTMLHeading.Create(1, 'Pascal source for ' +
                            filFiles.FileName));
{ Format the Pascal source file. }
htm.Add(THTMLPascal.Create(filFiles.FileName));
{ Footer. }
htm.Add(THTMLHorizRule.Create(AsPercentage(100),
                              ahDefault));
htm.Add(THTMLText.Create(tsNormal,
 'Generated by IHTML(c) objects - Written by Keith Wood'));
{ Copy results to the memo field. }
merHTML.Text := htm.AsHTML;
```

Figure 2: Converting a Pascal source file into an HTML document.

```
{ Create the HTML document. }
htm := THTMLDocument.Create('Directory listing for ' +
  dirDirectories.Directory + '\' + fltFilters.Mask);
{ Add a heading. }
htm.Add(THTMLHeading.Create(1,'Directory listing for ' +
  dirDirectories.Directory + '\' + fltFilters.Mask));
{ Then create the directories/files listing. }
htm.Add(THTMLDirectory.Create(dirDirectories.Directory,
  fltFilters.Mask, cbxSubdir.Checked));
{ Footer. }
htm.Add(THTMLHorizRule.Create(AsPercentage(100),
                              ahDefault));
htm.Add(THTMLText.Create(tsNormal,
 'Generated by IHTML(c) objects - Written by Keith Wood'));
{ Copy results to the memo field. }
merHTML.Text := htm.AsHTML;
```

Figure 3: Converting a directory structure into an HTML document.

```
<html>
<head>
<title>A simple frameset document</title>
</head>
<frameset cols="20%, 80%">
<frameset rows="1*, 2*">
<frame src="./images/athena.jpg">
<frame src="./images/athena.jpg">
</frameset>
<frame src="DemoFramesSource.htm">
<noframes>
<p>This frameset document contains:</p>
<ol>
<li><a href="./images/athena.jpg">Some neat contents
</a></li>
<li><img src="./images/athena.jpg" border=0
  alt="A neat image"></li>
<li><a href="DemoFramesSource.htm">
The code that produced this</a></li>
</ol>
</noframes>
</frameset>
</html>
```

Figure 4: An HTML frameset document.

Directory Structure

Displaying a directory structure on the Web can be done through FTP. But by doing it programmatically, we have much more control over what is shown and how it appears. We can achieve this by encapsulating the generation of the directory structure in a new *IHTML* object, *THTMLDirectory*. It's derived from *THTMLBase* because it doesn't contain any other tags.

We want to be able to specify the starting directory for our generation, a mask to be used in selecting the files displayed, and a flag indicating whether we should check into the subdirectories under this root. These are set up as properties of the object, along with a constructor to facilitate their initialization.

When the HTML is requested, we use an internal *THTMLTable* object to construct the directory representation. Once completed, we simply return the HTML from the table as the result of the directory object, as shown in Listing One (beginning on page 20).

The columns in the table show the directory or file name, its size, and the date and time it was last modified. A header row is built to display appropriate titles. To improve the presentation of the final display, we make use of the <COLGROUP> and <COL> tags available for tables in HTML 4 to specify that the "size" column should be aligned to the right.

We then start the generation process for the root directory. First, we retrieve all the subdirectories in the current directory and save them to a string list. Next, we add all the files that match the specified mask to that list. Finally, we step through each list item (conveniently sorted for us) and create a table row filled with its details. If the entry is a subdirectory and we are scanning all subdirectories, we call this process recursively to complete the structure. We keep track of the level of the subdirectories so we can indent the items found therein, and thus display the inherent hierarchy within the HTML page.

Now that we've encapsulated the directory structure in an HTML-generating object, producing the actual document is very simple, as shown in Figure 3.

HTML Frameset

A frameset document describes to a browser how to break up the screen to show multiple documents. The <FRAMESET> tag replaces the normal <BODY> tag for the document, and contains a number of individual frames or embedded framesets. Attributes of the <FRAMESET> tag indicate how much space is to be devoted to each row and column.

This space is measured in exact pixels (an integer value), as a percentage of the available screen space within the browser (number and percent sign (%)) or as an amount relative to the other rows or columns (number and asterisk (*)). For example, see the HTML page in Figure 4. This document displays two columns, the first being 20 percent of the total width available, and the second column occupying the other 80 percent. The first column is then subdivided into two rows by an embedded frameset. The first row takes up one third of the available space, and the other takes up two thirds. Frame tags then specify the actual contents of those regions.

To cater to browsers that cannot handle frames, the content within the <NOFRAMES> tag is shown instead. This can include any of the normal HTML tags. In this case, we provide a reference to one

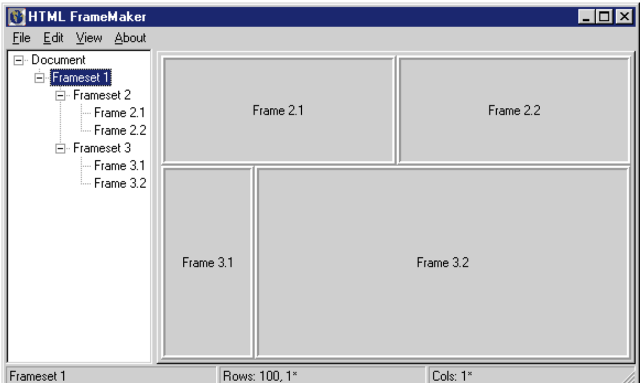


Figure 5: Displaying a frameset document during design.

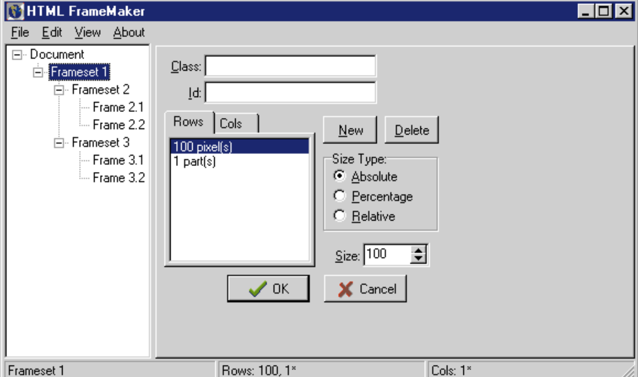


Figure 6: Setting a frameset's properties.

image, display the second one directly, and have a link to the document that filled the third frame.

Often, the coding of a page like this is done by hand, with no immediate feedback about the appearance of the page. The effects of resizing the browser must wait until the document is loaded for testing. Coming from the Delphi world, we expect a graphical way of doing things, and, using the *IHTML* objects, we can achieve just that.

Frameset Designer

Because we're ultimately generating framesets and frames, and the *IHTML* objects already provide this functionality, we can build on that foundation and merely extend it. Our two new classes, *TMakerFrameSet* and *TMakerFrame*, are subclasses of *THTMLFrameSet* and *THTMLFrame*, respectively. They inherit the HTML-generating abilities of their ancestors, and we add additional properties and methods to interact with the visual display.

Within a frame, we simply track its height and width for display on the screen, and its frameset parent. For a frameset, we need it to know its identifying number, the framesets or frames that it contains, and its immediate parent frameset (if one exists). Additional methods allow us to change a child frame into an embedded frameset, and vice versa. We also need the frameset to generate child frame objects once we supply the dimensions for the rows and columns. And finally, we need the frameset to interact with our user interface to display its contents.

The user interface is divided into two parts, with the left side containing a tree view that shows the frameset hierarchy within the document (see Figure 5). On the right, we display either the properties for the current node in the tree, a graphical display of the entire frameset, or the HTML that is generated.

Upon opening, we're presented with a default frameset document that contains a single frameset with a single frame. This frameset is referenced within the program as the base frameset, and is the one ultimately added to the HTML document. The user then modifies the properties of the objects within the hierarchy, views the results on the screen, and generates the final HTML document.

Changing Properties

As properties of the document as a whole, we have the page's title and the text to be displayed in browsers that cannot handle frames. Although the latter is entered as plain text, there's no reason HTML tags cannot be placed in the content to format the result.

The properties of a frameset (see Figure 6) allow us to specify any class or id values, and the number and dimensions of the rows and/or columns for this object. Buttons enable us to add or remove rows or columns. For each one, we can select the type of the size measurement, then enter its actual value. A text description of the assigned size appears in the row or column listbox to show the results. Pressing the OK button saves any changes and generates appropriate child frames based on the rows and columns specified.

Changes to the class or id values update the frameset's properties directly, while alterations to rows or columns necessitates a recalculation of the number of rows and columns belonging to the frameset and their sizes. Existing frames within the bounds of the new values are retained, while extraneous ones are destroyed. Any new frames required are generated with default values.

For a frame, the properties are numerous. First, we have the name of the frame (to be used as the target of a link on a page) and the URL from which to obtain the frame's initial content. Next, we have the class and id values (as for the frameset), flags to indicate whether the frame can be resized or has a border, the sizes of margins around the frame, and how the frame is to deal with scrolling should its contents be too much to display all at once. Each of these controls updates the corresponding property within the frame object itself, and will be generated automatically into the resulting document when requested.

Display and Generation

The frameset object itself manages the display of the frames defined within this program. One method requests it to fill a tree view with nodes representing itself and the frames it contains. Another method causes it to generate panels of the appropriate size for each of its child frames. These panels are contained within the base panel that sits on one of the page control tabsheets on the main form.

The calculations involved in determining the correct size for each frame are fairly complex. We must add up the absolute sizes (actual pixels) and total the relative sizes for those frames that have them. Then we can assign the panel dimensions for those panels with absolute or percentage sizes (taking into account scaling if the absolute sizes don't add up to the total available), while keeping track of the space remaining. This unallocated space is then split up between those panels with relative sizes. All this is done within the confines of the dimensions of the panel representing the parent frameset, allowing it to be applied to embedded framesets, as well.

A method is assigned to each panel that's so generated, that causes a click on the panel to select the corresponding item in the tree view, based on its caption. Clicking on an object in the tree view or in the graphical display selects it, causing its major

details to be shown in the status bar, and its properties page to be loaded (if properties are being viewed).

Generating the final document is performed in the usual way. After adding the base frameset object to the document (assigned to its *FrameSet* property), we call the document's *AsHTML* method, which invokes all the others' in turn.

Because the frame maker objects we use are derived from the *IHTML* objects, these can be added directly within the frameset and called upon to do their bit. We add them in the *AsHTML* method, rather than when they are created, because they may change somewhat before being finally generated. Because we want to retain these objects after generating the document, we must ensure that the *OwnContents* property of the frameset, and the document itself, is set to False.

The resulting document is saved to the specified file, and can then be viewed as HTML source by selecting the **View | View HTML** option on the menu. Or, power up your browser to see what it looks like when rendered.

Conclusion

The applications presented in this article make use of the *IHTML* objects introduced last month to produce customized HTML documents. By encapsulating the desired behaviors in new *IHTML* objects, we greatly simplify the generation of the final documents. The applications presented here produce documents displaying Pascal source code, generate pages showing a directory structure, or create frameset documents visually.

The extensibility of the *IHTML* hierarchy and the use of an interface for the main generation ability means we can use them in any number of situations. Only our imaginations limit the applications. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\JUN\DI9906KW.

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.

Begin Listing One — Directory structure to HTML

```
{ Generate an HTML document showing the selected
  directory structure. }
function THTMLDirectory.AsHTML: string;
var
  htb: THTMLTable;
  hcg: THTMLTableColumnGroup;
  htr: THTMLTableRow;
begin
  try
    { Create the table to hold the directories/files. }
    htb := THTMLTable.Create(AsPercentage(100), 0);
    { Align the file size column to the right. }
    hcg := THTMLTableColumnGroup.Create('', 0, 0);
    hcg.Add(THTMLTableColumn.Create('', 0, 0));
    hcg.Add(THTMLTableColumn.Create(
      'text-align: right', 0, 0));
    hcg.Add(THTMLTableColumn.Create('', 0, 0));
    htb.Add(hcg);
    { Create a new HTML table row. }
    htr := THTMLTableRow.Create;
    htr.AlignHoriz := thLeft;
    { And add headings to it. }
    htr.Add(THTMLTableHeading.Create('Name'));
    htr.Add(THTMLTableHeading.Create('Size'));
    htr.Add(THTMLTableHeading.Create(
      Indent(1) + 'Modified'));
    { Add the row to the table. }
    htb.Add(htr);
    { Generate the directory tree into the table. }
    GenerateHTMLForDir(htb, Directory, 0);
    Result := htb.AsHTML;
  finally
    { Free all the HTML objects. }
    htb.Free;
  end;
end;

{ Generate directory structure for a specified directory. }
procedure THTMLDirectory.GenerateHTMLForDir(
  htb: THTMLTable; sDirectory: string; iLevel: Word);
const
  { To sort directories (d) before files (f). }
  sSortType: array [Boolean] of string = ('f', 'd');
var
  slsFiles: TStringList;  { To sort the files. }
  src: TSearchRec;        { File details found in search. }
  i, iFound: Integer;     { Working variables. }
  htr: THTMLTableRow;     { The current HTML table row. }

  { Is the current file entry a directory? }
  function IsDirectory(iAttr: Integer): Boolean;
  begin
    Result := ((faDirectory and iAttr) <> 0);
  end;
begin
  try
    slsFiles := TStringList.Create;
    slsFiles.Sorted := True;
    slsFiles.Duplicates := dupIgnore;
    if IncludeSubDirectories then begin
      { Find subdirectories in specified directory. }
      iFound := FindFirst(sDirectory + '\*.*',
                         faDirectory, src);
      while iFound = 0 do
        with src do begin
          { If not self or parent directory reference, then
            add to list. }
          if IsDirectory(Attr) and
             not ((Name = '.') or (Name = '..')) then
            slsFiles.AddObject(sSortType[True] + Name,
              TFileInfo.Create(Name, Size, Time, Attr));
          { Any more? }
          iFound := FindNext(src);
        end;
      FindClose(src);
    end;
    { Find matching files in specified directory. }
    iFound := FindFirst(sDirectory + '\' + Mask,
                       faAnyFile, src);
    while iFound = 0 do
      with src do begin
        { If not self or parent directory reference,
          then add to list. }
        if not ((Name = '.') or (Name = '..')) then
          slsFiles.AddObject(sSortType[IsDirectory(Attr)] +
            Name, TFileInfo.Create(Name,Size,Time,Attr));
        { Any more? }
        iFound := FindNext(src);
      end;
    FindClose(src);
    { Now process in sorted order. }
    for i := 0 to slsFiles.Count - 1 do
      with TFileInfo(slsFiles.Objects[i]) do begin
        { Create a new HTML table row. }
```

```
        htr := THTMLTableRow.Create;
        { And add cells to it. }
        htr.Add(THTMLTableDetail.Create(
          Indent(iLevel) + Name));
        if IsDirectory(Attr) then
          htr.Add(THTMLTableDetail.Create('Dir'))
        else
          htr.Add(THTMLTableDetail.Create(IntToStr(Size)));
        htr.Add(THTMLTableDetail.Create(Indent(1) +
          DateTimeToStr(FileDateToDateTime(Time))));
        { Add the row to the table. }
        htb.Add(htr);
        { Recurse into subdirectories if appropriate. }
        if IsDirectory(Attr) and IncludeSubDirectories then
          GenerateHTMLForDir(htb, sDirectory + '\' + Name,
                             iLevel + 1);
      end;
  finally
    { Tidy up. }
    for i := 0 to slsFiles.Count - 1 do
      slsFiles.Objects[i].Free;
    slsFiles.Free;
  end;
end;
```

End Listing One

By Xavier Pacheco

The Builder Pattern

Extending Frameworks — Building Add-in Packages

In the March, 1999 and May, 1999 issues of *Delphi Informant*, I presented the Singleton and Template Method patterns. They are likely to be widely used because they are easy to implement, and can be applied to a wide range of problems. This month, we'll discuss an equally useful pattern, although one that is slightly more complex: the Builder pattern.

We'll discuss the Builder pattern, and use it to extend the application framework presented last month. This framework will allow developers to work on separate modules that may be added dynamically to compiled applications by using Delphi's package technology.

As I study patterns, I find it essential to think about them in the context of the language and object framework within which they'll be implemented. It seems that much of what is available in the form of patterns literature presents patterns from a purely object-oriented context, not taking into account a particular framework, e.g. VCL, MFC, OWL, etc. This isn't a criticism, and I recommend that you make the effort to understand these patterns from this perspective. This column, however, intends to present patterns from a different perspective; the goal is to illustrate how and when to use and implement patterns within the context of Delphi and the VCL framework.

Before we get to the Builder pattern, however, we need to discuss two aspects of patterns: pattern classifications and how patterns are described.

Pattern Classifications

Patterns can be classified as one of three types: creational, structural, or behavioral. Creational patterns have to do with how the objects making up the pattern are created; structural patterns have to do with how the objects within the pattern are composed; and behavioral patterns have to do with how the objects interact with each other, and the clients of those patterns.

The Singleton and Builder patterns are creational patterns, and the Template Method pattern is a behavioral pattern. Keep in mind that a pattern is often a part of another pattern or creates another pattern. For example, the Builder pattern is often used to create a Composite pattern, which is a structural pattern.

Pattern Descriptions

I've examined several books on patterns and have determined there is no single rule by which patterns are described. However, each of the sources is consistent in how it describes each pattern by using a uniform set of elements. These elements consist of descriptions, definitions, pictures, and examples (see Figure 1).

Element	Meaning
Name	Name by which a pattern will be referred to by developers. It should match the pattern's purpose.
Classification	Classification to which the pattern belongs: creational, structural, or behavioral.
Synopsis	Concise sentence describing the pattern.
Intent, Problem, or Context	Statement describing what the pattern does, or the problem the pattern addresses.
Motivation	Description of how the pattern applies itself to a given problem.
Structure	Typically, graphical representations of the pattern using some form of object-modeling notation, such as UML. This may also present a definition of the participants.
Forces	Considerations of the problem that lead to the solution.
Solution	Describes how the pattern applies itself to the problem and addresses the various forces.
Applicability	Areas where the pattern may be used.
Known Uses	Areas where the pattern is known to be used in the context of the language being discussed.
Consequences	Pros and cons of using the pattern.
Collaborations	How objects (participants) of the pattern interact to accomplish a solution.
Examples	An example of the pattern's implementation in a given language.
Related Patterns	Other patterns that resemble, or might be used instead of, the pattern.
Participants	The elements (objects, interfaces) that make up a pattern's structure.

Figure 1: Pattern description elements.

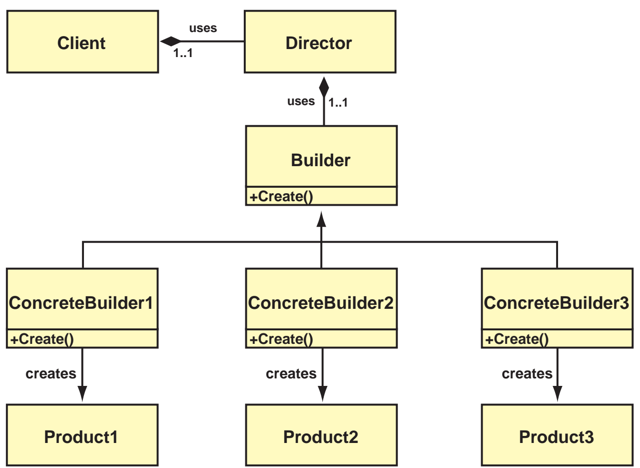


Figure 2: The Builder pattern structure.

For this and future articles, I'll use a variation of this more formal method of describing the patterns I present, including only the Name, Classification, Synopsis, Applicability, Structure, Known Uses, and Examples (see the sidebar "Inside the Builder Pattern" on page 26). However, I do encourage you to refer to those sources cited at the end of this article for a more complete description of the patterns I discuss.

Participants

There are five participants to the Builder pattern, including the Client (i.e. the user of the pattern), as shown in Figure 2. The functions of these participants follow:

- The Director uses a Builder interface to construct Concrete Builders. The Director may also act as an intermediary between the Client and the Builder.
- The Builder is an abstract interface that defines the methods required to tell it how to construct itself.
- The Concrete Builder is a specific implementation of the Builder interface that implements its abstract construction methods.
- The Product is the item being built.
- The Client is the user of the item being built — the complex object.

Structure Description

Notice that the Client has a one-to-one association with the Director. The Client makes a request to the Director for a specific Product. The Director, in response to this request, uses the abstract Builder interface to construct the Concrete Builder based on the criterion given by the Client. The Concrete Builder then returns an instance of the Product. This result may go directly back to the Client, or it may go through the Director for the Client to use.

If you're familiar with patterns, you may recognize that this structure is similar to the Abstract Factory and the Composite patterns. The Abstract Factory is another creational pattern, very much like the Builder pattern, but it differs in the way its resulting product is presented to the Client. Instead of providing a single, self-contained product, the Abstract Factory returns pieces that make up a product. It's up to the Client or the Director to construct these pieces. (I'll discuss the Abstract Factory in a later article.) The Composite is a structural pattern used to compose objects into hierarchical structures. In fact, the Builder and Abstract Factory are often used to create Composites.

Known Uses

The classical example of the Builder pattern is that of a graphics image viewer, where the viewer may render images in multiple formats. The viewer doesn't concern itself with the type of image to render, but rather the process by which it obtains an image. The viewer uses a Director that manages the creation process of each image format. A builder object, one for each format of image, handles an image's specific creation logic.

Another use — to which Delphi developers may more closely relate — is that of Experts. It's possible to add behavior to the Delphi IDE by providing modules in the form of packages. These modules contain the objects that implement pre-defined interfaces that are part of the Delphi Open Tools API. The structure of Open Tools is similar to that of the Builder pattern. This add-in capability is a powerful feature that allows anyone to enhance the IDE with additional functionality. This capability can be incorporated into applications you build using a technique similar to that of the Open Tools API. The rest of this article presents an example of how to implement such a framework using a Builder pattern.

Add-in Packages Example

The add-in packages demonstration I've written illustrates a simplified method by which you can add functionality to an application dynamically — without having to modify and re-compile the application. I do this using Delphi packages.

Before packages came about, we would have accomplished this by using a similar method through dynamic-link libraries (DLLs). One of the problems with DLLs is that they're not VCL aware. It's almost impossible to effectively decompose functionality from the application. You would have to deal with the mess of exporting VCL objects from the DLL.

This isn't a problem with packages. Although packages are DLLs, they're special-purpose DLLs that know about the VCL. Delphi and its applications know how to obtain the run-time type information about objects contained within packages. Therefore, forms and other VCL objects in your packages can easily be integrated into your Delphi applications.

Package Layout for the Shell Application

Take a look at Figure 3. It shows how the Shell Application uses packages to obtain add-in functionality. We'll discuss these packages in-depth in this section.

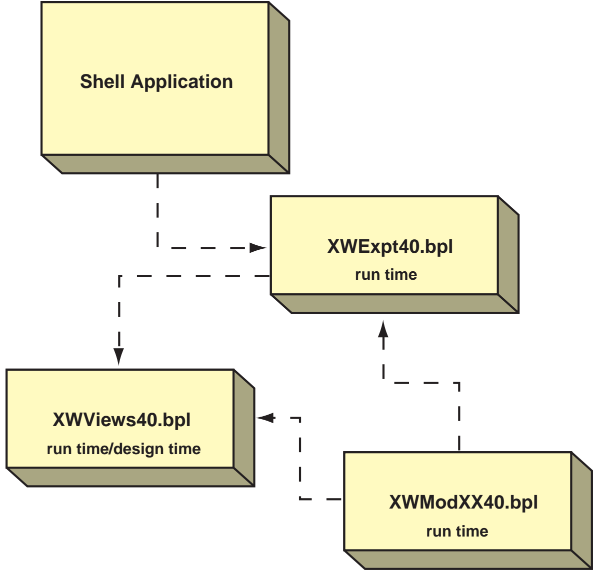


Figure 3: How the Shell Application uses packages to obtain add-in functionality.

XWViews40.bpl. This lowest-level package is a run-time and design-time package that contains the *TViewForm* from last month's article. Recall that this form overrides the methods to allow it to be instantiated as both an independent form, and as a contained form. This package will be used by both the XWExpt40.bpl and XWModXX40.bpl packages. (Note: The only reason XWViews40.bpl is also a design-time package is because the *TViewForm* is going to serve as the ancestor to all forms used by modules. You may want to write a Form Expert or install the form into the repository, so you don't have to open the *TViewForm* unit every time you create a new project.)

XWExpt40.bpl. The Module Interface is defined as a semi-abstract class in this package. This package also contains the definition of the Module Manager. It's used by the Shell Application and by each module instance (add-in package). The Module Manager serves the purpose of the Director in the Builder pattern structure, whereas the Module Interface serves the purpose of the abstract *Builder* class.

XWModXX40.bpl. Module instances that implement the Module Interface are contained in the XWModXX40.bpl package, where the "XX" in the package name would be replaced by a two-digit identifier. For example, package 1 would be XWMod0140.bpl, and package 2 would be XWMod0240.bpl. You may add any number of these packages to the Shell Application. These modules must contain the implemented Module Interface and any additional forms that adhere to the *TViewForm* definition. Also, this module can contain any additional data modules or other forms necessary to provide the module-specific code. The XWModXX40.bpl packages are run-time packages that are dynamically loaded by the Module Manager object. They're registered in the Windows registry so the Shell Application will know of their existence.

Add-in Builder Structure Description

Figure 4 shows how the Builder pattern is structured for add-in packages. The Shell Application (the Client) has a one-to-one association with the XWModuleManager (the Director). The Shell Application makes requests to the Module Manager to load a particular package. The menus that allow the user to select the package to load are built dynamically and are added to the Shell Application's main menu at run time, based on information contained in the Windows registry. It's the Module Manager that searches through the Windows registry, and builds a menu that is merged with the Shell Application's menu. The Module Manager also provides the *OnClick* event handlers for these menu items that load the corresponding package.

The Module Manager has a one-to-one association with the *TXWModuleInterface* class used to refer to the currently loaded module. Each *TXWModuleInterface* instance implements the methods required to build itself. This includes the construction of any sub-forms, data modules, menus, and toolbars that get integrated with the Shell Application. My code shows how the Shell Application provides a parent *TMenuItem* for module-level (package), and view-level (form) menus. The Shell Application also provides a parent container for a module-level and view-level toolbar. Toolbars provided by these objects are integrated into the Shell Application's main Coolbar component. Figure 5 shows what the Shell Application looks like when one of the modules is loaded.

The *TXWModuleInterface* contains the *TViewForms* that get integrated into the Shell Application by keeping track of a current view form. When the user changes pages on the Shell Application, the View Form for that page is created and displayed within a Panel component. This is similar to how we achieved form containment in last month's article.

The code used to make all of this work is somewhat extensive to explain in text. Therefore, I've included ample remarks in the source, where I can more clearly explain what I'm doing. (The complete source for the demonstration application discussed in this article is available for download; see the end of this article for details.) In the following sections, I'll highlight some key points.

Loading and Unloading Packages

When the user selects one of the modules from the Shell Application's main menu, the Module Manager loads the package specified after unloading the currently loaded package. The Delphi routines to load and unload a package are *LoadPackage* and *UnloadPackage*. You'll see this code in the *TXWModuleManager.LoadModule* and *TXWModuleManager.UnloadModule* methods in Figure 6. These two routines function exactly like the *LoadLibrary* and *UnLoadLibrary* API functions in that they load/unload the specified module. *LoadModule* returns a valid library handle if it succeeds.

Class Name Retrieval of the Module Interface Class

Neither the Shell Application nor the Module Manager know of the class names of the implemented modules contained in the module implementation packages. Therefore, I store these class names in the Windows registry (see Figure 7).

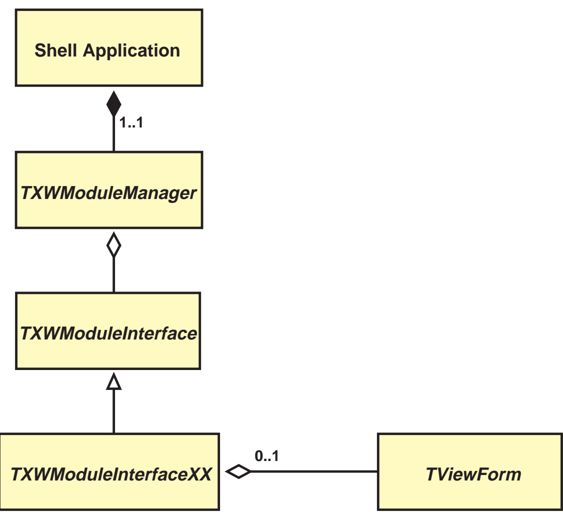


Figure 4: The Builder pattern structure for add-in packages.

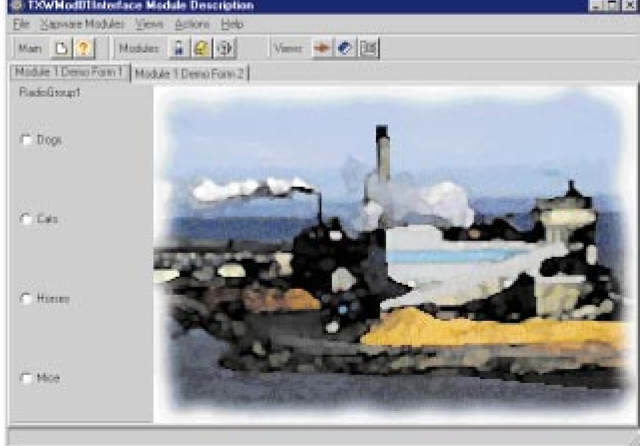


Figure 5: Shell Application with one of the modules loaded.

Ideally, the package would support a function such as *GetPackageProcAddress* to retrieve an address to an exported routine — just as with DLLs. This isn't the case, although it's possible to pull routine names out of a package by performing some hairy, compiler-specific voodoo. Perhaps this is a topic for another article, but for now, I'll opt for the much simpler technique of putting each class name in the registry. (A .reg file is included in the download file to facilitate this process.) To retrieve this class name, I wrote the helper function *GetModuleClassName*:

```
function GetModuleClassName(
  const AModuleName: string): string;
{ The class name of the TXWModuleInterface is provided in
  the registry. This method retrieves that class name. }
var
  IniFile: TRegIniFile;
begin
  IniFile := TRegIniFile.Create(cModRegLocation);
  try
    Result := IniFile.ReadString(RemoveExt(AModuleName),
                                 'ClassName', EmptyStr);
  finally
    IniFile.Free;
  end;
end;
```

```
procedure TXWModuleManager.LoadModule(
  const AModuleName: string);
var
  ModClassName: string;
  XWMIC: TXWModuleInterfaceClass;
begin
  // Call any event handlers provided by the client.
  if Assigned(FBeforeLoadModule) then
    FBeforeLoadModule(Self);
  // Unload any previously loaded modules.
  if FCurrentModuleHandle <> 0 then
    UnloadModule;
  try
    // Load the specified module.
    FCurrentModuleHandle := LoadPackage(AModuleName);
    // Return the class name that needs to be instantiated
    // into ModClassName.
    ModClassName := GetModuleClassName(AModuleName);

    { Create an instance of the class using the FindClass
      procedure. Note, this requires that the class already
      be registered with the streaming system using
      RegisterClass. This is done in a unit initialization
      section in each module package. }
    XWMIC :=
      TXWModuleInterfaceClass(FindClass(ModClassName));
    FCurrentModule := XWMIC.Create(FViewParentWindow,
      FModuleLevelMenuParent, FModuleLevelTBParent);

    if Assigned(FAfterLoadModule) then
      FAfterLoadModule(Self);

    { Load the module's menus and toolbars. }
    FCurrentModule.LoadModuleLevelMenu;
    FCurrentModule.LoadModuleLevelToolbar;
  except
    on E: Exception do begin
      UnloadModule;
      raise;
    end;
  end;
end;

procedure TXWModuleManager.UnloadModule;
begin
  if FCurrentModuleHandle <> 0 then begin
    if Assigned(FBeforeUnloadModule) then
      FBeforeUnloadModule(Self);
    { Unloaded any module level and view level menus before
      unloading and freeing this module. }
    if FCurrentModule <> nil then begin
      FCurrentModule.UnloadModuleLevelMenu;
      CloseView;
      FCurrentModule.Free;
      FCurrentModule := nil;
    end;
    // Unregister any classes provided by the module.
    UnRegisterModuleClasses(FCurrentModuleHandle);
    // Unload the module package.
    UnloadPackage(FCurrentModuleHandle);

    FCurrentModuleHandle := 0;
    // Call any event handlers provided by the client.
    if Assigned(FAfterUnloadModule) then
      FAfterUnloadModule(Self);
  end;
end;
```

Figure 6: The *LoadModule* and *UnloadModule* procedures.

Figure 8 highlights the module class name entry in the Registry Editor.

Instantiating Classes Contained in Packages

Now that the Module Manager knows of the class name, how can it actually instantiate the class? To create a class, you need a class reference. The *FindClass* and *GetClass* functions return class references that are registered with the streaming system. The following code snippet illustrates how the class name is obtained, and how the class instance is created:

```
// Return the class name that needs to be instantiated
// into ModClassName.
ModClassName := GetModuleClassName(AModuleName);
XWMIC := TXWModuleInterfaceClass(FindClass(ModClassName));
FCurrentModule := XWMIC.Create(FViewParentWindow,
  FModuleLevelMenuParent, FModuleLevelTBParent);
```

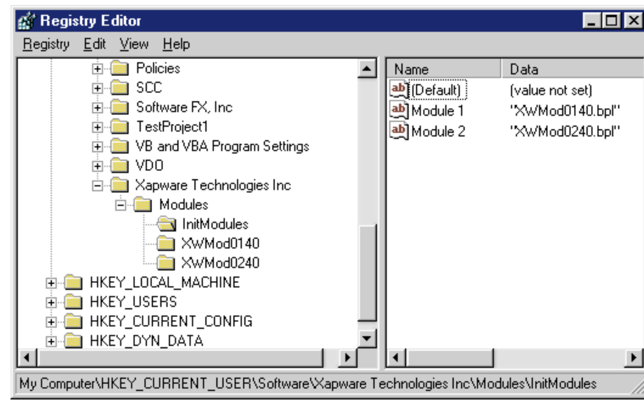


Figure 7: The module registry entries.

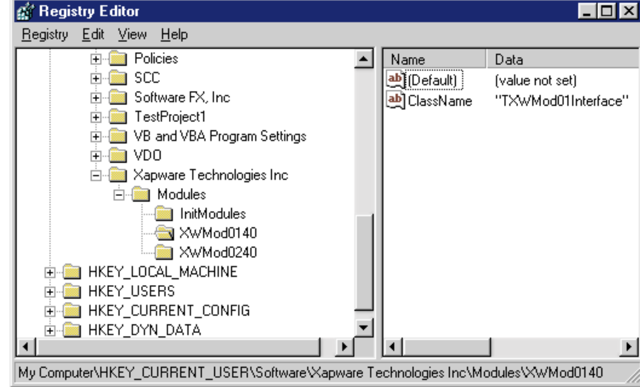


Figure 8: The module class name entry.

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  { Create the FModuleManager and assign to its properties
    the menus and panels that will house the module and
    view level menus and toolbars. }
  FModuleManager := TXWModuleManager.Create;
  FModuleManager.ModuleLevelMenuParent := mmiViewSelect;
  FModuleManager.ModuleLevelTBParent   := pnlModule;

  FModuleManager.ViewParentWindow := pnlMain;
  FModuleManager.ViewMenu := mmiActions;
  FModuleManager.ViewTBParent := pnlViews;

  // Merge the Module Manager's menu.
  mmMain.Merge(FModuleManager.ModuleManagerMenu);
  // Hook up the event handlers for After and Before
  // loading modules.
  FModuleManager.AfterLoadModule := AfterLoadModule;
  FModuleManager.BeforeUnloadModule := BeforeUnloadModule;
end;
```

Figure 9: The main form's *OnCreate* event handler is responsible for merging the menus.

The key word here is "registered." When a package is loaded, none of the classes it defines are registered with the loading applications. This is equally the case for component writers who should be well familiar with the *RegisterComponent* method. It's the registration methods that expose the classes to the calling application. Each package must properly register the module interface class when the package is loaded. This can be done in the **initialization** section of the unit that defines the module interface class.

Likewise, the class must also be unregistered when the package is unloaded. This can be done in the unit's **finalization** section. The calls made are *RegisterClass* and *UnRegisterClass,* respectively:

```
initialization
  RegisterClass(TXWMod01Interface);

finalization
  UnRegisterClass(TXWMod01Interface);
```

One of the things I do when I unload the package in the *TXWModuleManager.UnloadModule* method is to call *UnRegisterModuleClasses*. This ensures that all classes registered by a module get unregistered.

Menu Merging

In the add-in package demonstrations, there are several layers of menus that get merged with the Shell Application's main menu. To be technically accurate, only the Module Manager's menu gets merged with the Shell Application's main menu. I use the normal merging technique through the *TMainMenu.Merge* method (see Figure 9). To merge the other layers (Module, View), I must copy these menus and programmatically add them to the main menu. The reason for this is because only one menu can be merged using the *Merge* method.

In addition to the Shell Application's main form, the Module Manager, and the Abstract Module Interface (see Listing One, beginning on this page), the code that accompanies this article contains an example implementation of the Module Interface (see Listing Two, beginning on page 27). The code provided includes two implementations of the module interface, including one that contains database access.

Conclusion

The Builder pattern is a useful creational technique for structuring complex objects resulting in a self-contained product. In my next article, I'll examine the Abstract Factory, a pattern similar to the Builder, but with a different outcome. Finally, many thanks to Anne Pacheco for her grammatical review of this, and many other of my articles. ∆

Sources

Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, et al. [Addison-Wesley, 1994]; *The Design Patterns Smalltalk Companion* by Kyle Brown, Bobby Woolf, and Sherman R. Alpert [Addison-Wesley, 1998]; *Patterns in Java, Volume 1* by Mark Grand [Wiley Computer Publishing, 1998].

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\JUN\DI9906XP.

Inside the Builder Pattern

Name: Builder Pattern

Classification: Creational

Synopsis: "Separate the construction of a complex object from its representation so that the same construction process can create different representations" (from *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, et al.[Addison-Wesley, 1994]).

Applicability: Used where the logic required for building complex objects is separated from the client of that object. Also, this construction logic must facilitate the building of multiple representations, or, rather, variations of complex objects that may work with that same client.

Structure: Shown in Figure 2.

— Xavier Pacheco

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

Begin Listing One — MainFrm.pas

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Menus, ToolWin,
  ComCtrls, xwModExpt, ImgList;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    mmiHelp: TMenuItem;
    mmiAbout: TMenuItem;
    mmiHowHelp: TMenuItem;
    mmiContents: TMenuItem;
    cbMain: TCoolBar;
    tbShell: TToolBar;
    tbcMain: TTabControl;
    stsbrMain: TStatusBar;
    pnlMain: TPanel;
    mmiViewSelect: TMenuItem;
    mmiActions: TMenuItem;
    tbtn1: TToolButton;
    tbtn2: TToolButton;
    imMain: TImageList;
    pnlModule: TPanel;
```

```
    pnlViews: TPanel;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure tbcMainChange(Sender: TObject);
    procedure FormClose(Sender: TObject;
      var Action: TCloseAction);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiAboutClick(Sender: TObject);
  private
    FModuleManager: TXWModuleManager;
    procedure OpenView(const AViewNum: Integer);
    { : After load module event handler. }
    procedure AfterLoadModule(Sender: TObject);
    { : Before Unload module event handler. }
    procedure BeforeUnloadModule(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
begin
  { Create the FModuleManager and assign to its properties
    the menus and panels that will house the module and
    view level menus and toolbars. }
  FModuleManager := TXWModuleManager.Create;
  FModuleManager.ModuleLevelMenuParent := mmiViewSelect;
  FModuleManager.ModuleLevelTBParent    := pnlModule;

  FModuleManager.ViewParentWindow := pnlMain;
  FModuleManager.ViewMenu := mmiActions;
  FModuleManager.ViewTBParent := pnlViews;

  // Merge the Module Manager's menu.
  mmMain.Merge(FModuleManager.ModuleManagerMenu);
  // Hook up the event handlers for After and Before
  // loading modules.
  FModuleManager.AfterLoadModule := AfterLoadModule;
  FModuleManager.BeforeUnloadModule := BeforeUnloadModule;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  // Unmerge the Module Manager's menu and free it.
  mmMain.UnMerge(FModuleManager.ModuleMainMenu);
  FModuleManager.Free;
end;

procedure TMainForm.OpenView(const AViewNum: Integer);
begin
  { When a new view (Form) is loaded, set its parent window
    to that of the Module Manager's parent window. }
  if FModuleManager.CurrentModule <> nil then
    FModuleManager.OpenView(AViewNum, True);
end;

procedure TMainForm.tbcMainChange(Sender: TObject);
begin
  OpenView(tbcMain.TabIndex)
end;

procedure TMainForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  FModuleManager.UnloadModule;
end;

procedure TMainForm.AfterLoadModule(Sender: TObject);
begin
  { When a module is loaded, retrieve its caption and
    create its tabs on the TTabControl component based on
    its captions. Also, get its icon. }
  Caption := FModuleManager.CurrentModule.ModuleDescripton;
  tbcMain.Tabs.Assign(
    FModuleManager.CurrentModule.ViewCaptions);
  // Invoke the call to open the first view.
  tbcMainChange(nil);
  Icon := FModuleManager.CurrentModule.ModuleIcon;
end;

procedure TMainForm.BeforeUnloadModule(Sender: TObject);
var
  M: TMemoryBasicInformation;
  i: Integer;
begin
  { Before unloading a module, we must free any hanging
    component provided by the module. }
  if FModuleManager.CurrentModule <> nil then begin
    for i := Application.ComponentCount - 1 downto 0 do
      begin
        VirtualQuery(GetClass(
          Application.Components[i].ClassName),
          M, SizeOf(M));
        if (FModuleManager.CurrentModuleHandle = 0) or
           (HMODULE(M.AllocationBase) =
             FModuleManager.CurrentModuleHandle) then
          Application.Components[i].Free;
      end;
    Caption := 'Xapware Shell Application';
    tbcMain.Tabs.Clear;
    Icon := Application.Icon;
  end;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.mmiAboutClick(Sender: TObject);
begin
  cbMain.Bands[2].Control := nil;
end;

end.
```

End Listing One

Begin Listing Two — XWMod01Intf.pas

```
unit XWMod01Intf;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, Menus, XWModExpt, xwViewFrm,
  ComCtrls, Mod01MgrFrm;

type
  TXWMod01Interface = class(TXWModuleInterface)
  private
    FViewCaptions: TStringList;
    FModuleIcon: TIcon;
    FModuleToolBar: TToolBar;
    FModuleMgrForm: TMod01MgrForm;
  private
    procedure ModuleViewsMenuOnClick(Sender: TObject);
  protected
    function GetModuleCaption: string; override;
    function GetModuleDescription: string; override;
    function GetModuleFileName: string; override;
    function GetModuleIcon: TIcon; override;
    function GetModuleToolBar: TToolBar; override;
    function GetModuleLevelMenu: TPopupMenu; override;
    // View Methods.
    function GetViewCount: Integer; override;
    function GetViewDescriptions: TStringList; override;
    function GetViewCaptions: TStringList; override;
```

```
    function GetCurrentView: TViewForm; override;
  public
    constructor Create(AParentWindow: TWinControl;
      AModuleLevelMenuParent: TMenuItem;
      AModuleLevelTBParent: TPanel); override;
    destructor Destroy; override;
    function CanSwitchModules: Boolean; override;
    function ModuleLogOn(const AUserName,
      APassword: string; const LenderID: Integer):
      Boolean; override;
    function GetView(const AIndex: Integer;
      AWithParent: Boolean): TViewForm; override;
    procedure CloseView; override;
  end;

implementation

uses
  DemoFrm1, DemoFrm2;

const
  { : Specify a valid form count. }
  FViewCount = 2;

{ TXWMod01Interface. }

function TXWMod01Interface.CanSwitchModules: Boolean;
begin
  // Pass true as a default for this example.
  Result := True;
end;

procedure TXWMod01Interface.CloseView;
begin
  if FCurrentView <> nil then begin
    UnloadViewLevelMenu;
    FCurrentView.Free;
    FCurrentView := nil;
  end;
end;

constructor TXWMod01Interface.Create(
  AParentWindow: TWinControl;
  AModuleLevelMenuParent: TMenuItem;
  AModuleLevelTBParent: TPanel);
var
  MenuItem: TMenuItem;
  i: Integer;
begin
  inherited Create(AParentWindow, AModuleLevelMenuParent,
                   AModuleLevelTBParent);
  FCurrentView := nil;
  FModuleIcon := TIcon.Create;
  // The icon is stored in the resource file for the
  // module. Retrieve it.
  FModuleIcon.Handle := LoadIcon(hInstance, 'MODULE1');
  // Create a TStringlist of Captions for each view
  // in the system.
  FViewCaptions := TStringList.Create;
  FViewCaptions.Add('Module 1 Demo Form 1');
  FViewCaptions.Add('Module 1 Demo Form 2');
  FModuleLevelMenu := TPopupMenu.Create(nil);
  // Add Menu items for each module.
  for i := 0 to FViewCaptions.Count - 1 do begin
    MenuItem := TMenuItem.Create(FModuleLevelMenu);
    MenuItem.Caption := FViewCaptions[i];
    MenuItem.OnClick := ModuleViewsMenuOnClick;
    FModuleLevelMenu.Items.Add(MenuItem);
  end;
  { Create the management form used for centralizing some
    logic and for design-time manipulation of visual
    components used with this module. Otherwise, I'd have
    to programmatically create these components and set
    their properties. It's easier to just use them off this
    hidden form. }
  FModuleMgrForm := TMod01MgrForm.Create(Application);
end;

destructor TXWMod01Interface.Destroy;
begin
  FModuleLevelMenu.Free;
  FModuleIcon.Free;
  FViewCaptions.Free;
  if FCurrentView <> nil then
    FCurrentView.Free;
  FModuleMgrForm.Free;
  inherited;
end;

function TXWMod01Interface.GetModuleCaption: string;
begin
  Result := Format('Module %s', [ClassName]);
end;

function TXWMod01Interface.GetModuleDescription: string;
begin
  Result := 'TXWMod01Interface Module Description';
end;

function TXWMod01Interface.GetModuleFileName: string;
begin
  Result := 'XWMod01Intf.pas';
end;

function TXWMod01Interface.GetModuleIcon: TIcon;
begin
  Result := FModuleIcon;
end;

function TXWMod01Interface.GetModuleToolBar: TToolBar;
begin
  Result := FModuleMgrForm.ToolBar1;
end;

function TXWMod01Interface.GetModuleLevelMenu: TPopupMenu;
begin
  Result := FModuleLevelMenu;
end;

function TXWMod01Interface.GetViewCaptions: TStringList;
begin
  Result := FViewCaptions;
end;

function TXWMod01Interface.GetViewCount: Integer;
begin
  Result := FViewCount;
end;

function TXWMod01Interface.GetViewDescriptions:
  TStringList;
begin
  Result := nil;
end;

function TXWMod01Interface.ModuleLogOn(
  const AUserName, APassword: string;
  const LenderID: Integer): Boolean;
begin
  // This could provide logic to perform a database logon.
  Result := True;
end;

procedure TXWMod01Interface.ModuleViewsMenuOnClick(
  Sender: TObject);
begin
  ShowMessage(Sender.ClassName);
end;

function TXWMod01Interface.GetView(const AIndex: Integer;
  AWithParent: Boolean): TViewForm;
begin
  if ((AIndex >= FViewCount) or (AIndex < 0)) then
    raise Exception.Create('Invalid view index');
```

```
  if FCurrentView <> nil then
    FCurrentView.Free;

  { This method will create the view as both a child window
    and as a regular form depending on the value
    of AWithParent. }
  if (AWithParent = True) and (FViewParentWindow<>nil) then
    case AIndex of
      0: FCurrentView := TDemoForm1.Create(
                           Application, FViewParentWindow);
      1: FCurrentView := TDemoForm2.Create(
                           Application, FViewParentWindow);
    end
  else
    case AIndex of
      0: FCurrentView := TDemoForm1.Create(Application);
      1: FCurrentView := TDemoForm2.Create(Application);
    end;
  Result := FCurrentView;
end;

function TXWMod01Interface.GetCurrentView: TViewForm;
begin
  Result := FCurrentView;
end;

initialization
  RegisterClass(TXWMod01Interface);

finalization
  UnRegisterClass(TXWMod01Interface);

end.
```

End Listing Two

By Alan C. Moore, Ph.D.

Youseful

Bill White's Useful Installation Creation System

Many of you have used one of the two leading installation applications: InstallShield and Wise Installation System. I've used the latter, and found it convenient for many situations. Something has troubled me for the longest time, though: Why had no one developed a native-VCL, component-based, installation system? Well, someone has. Bill White Software Production's powerful collection of installation components gives you all the power and flexibility of the two major products, but allows you to work exclusively within the Delphi environment.

This tool, Youseful, offers three ways to distribute your application: traditional (disk-based), Internet-based, and hybrid. With the traditional approach, you create a self-extracting .EXE file containing the installation routine and all the files that make up your application. If you plan to distribute your application on floppies, and the self-extracting executable won't fit on one floppy, you can use a series of floppies. With Internet distribution, users can use their browsers to install the software directly from an Internet site by clicking on its link. The user doesn't have to first download a self-extracting .EXE and then run it, so the process is straightforward.

Figure 1: Youseful's powerful eight-page Wizard allows you to make all of the basic decisions in building your installation. For example, the Groups page provides the means to create file groups on the user's machine.

Hybrid distribution has elements of both. First, Youseful creates a self-extracting .EXE containing the installation routine, but not all the files your application will need. Users will need to download the additional files from one or more remote servers. One of the great advantages of this approach is that the size of the self-extracting .EXE can be kept very small, i.e. 200-300KB.

A Super Component with a Powerful Component Editor

The core of the Youseful library is the *TInstall* component. By simply dropping a *TInstall* component on a form, double-clicking the component to bring up the component editor, and setting the preferences, you can produce your installation routine quickly and easily. This component editor is actually a complex Wizard with up to 16 pages, allowing you to make all necessary decisions (see Figure 1). The Wizard automatically adds the components you need to execute the particular kind of installation system you need. This library also includes two Help systems: a standard Windows Help system that provides information on all of the components, and an HTML Help file that includes several helpful tutorials. I completed the main tutorial quite quickly and found it provides an excellent introduction to the library's commonly-used features.

The first page in the Install component dialog box, Welcome, gives you the opportunity to enter general data used throughout the Wizard: your company's name, the product's name, and the

product's version. It also allows you to add password protection to your installation routine, and select one of over 14 languages to use.

The Appearance page allows you to control the appearance of your installation program by selecting various components and setting their properties. Most of this can be done from within this Wizard. On this page, you must select either the Maximize Window or Hide Window options. In the former case, your installation application will occupy the whole screen while the various dialog boxes appear. In the latter case, only those dialogs boxes appear. The background window can contain a backdrop (which could be dithered, as has become customary), a product label, and/or a billboard.

The Groups page allows you to establish various file groups (folders/directories) that will be created on the user's machine, as well as set properties for them (again, see Figure 1). Those properties include Install path, Dialog options, Description, and Install types. The Install path is the directory in which files will be installed; Description is used in the file group selection dialog box to help users determine if they wish to install the particular file group; Install types determines in which types of installations the file group will be included. Install types include all the common ones: Full, Compact, Typical, and Always.

The Registry page gives you the opportunity to automatically add appropriate keys to the Windows registry. In the top window, you enter a root key (e.g. hkCurrentUser, hkLocalMachine, etc.) and the key path for each entry. In the lower window, you enter the value of the key. By right-clicking on these panes, you're generally prompted to make the next required decision. The whole process is very smooth.

The Shortcuts page allows you to establish shortcuts during the installation of your application. You can easily create desktop shortcuts to files (including URLs), but you can't create such shortcuts to folders without some coding. This is one of the few limitations in this library. You can add a new group to the Start menu, which will appear at the top of the menu, or you can add an application to the Startup folder.

The Dialogs page allows you to add a number of built-in and custom dialog boxes to your installation. Youseful supports four dialog sequences and two dialog events. Dialog sequences involve one or more dialog boxes connected in such a way that the user can move forward or backward in the sequence. Dialog events display special dialog boxes that would not generally be part of the browse sequence but that need to be presented to the user in certain circumstances. These circumstances include indicating that files are being installed and prompting the user to insert a disk.

The four types of sequences are:
1) before any files are installed,
2) following a successful installation,
3) following an unsuccessful installation, and
4) prompting the user with the option to reboot the computer.

With Delphi, you can add dialog boxes to your installation as project forms, allowing you to customize them. If you make changes to Youseful's dialog boxes, those changes will be local and affect only the project you're creating.

The last two pages — Loading and Builder — complete the process. The Loading page allows you to specify whether *TInstall* will automatically install when the user runs the program (by setting the *RunAutomatic* property), and whether a Loader program will be used. If the latter option is chosen, a Setup.exe file (self-extracting) will be created that contains the application files, the installation project, and pre-setup files that you can specify on this Wizard page. The Builder page automatically saves all your work and compiles/builds the setup project. This convenient automation feature is something that could be emulated in other single-purpose component libraries.

There are several other pages available in Youseful's Wizard that support less-used features. As with the pages we've already discussed, the functionality is usually implemented by adding properties to your main form.

The Internet page is only needed if you plan to distribute your application via the Internet. On this page, you define the path to the server from which the files in your installation will be installed. The INI page allows you to make entries to an .INI file. Two more pages (BDE and Aliases) allow you to install all or part of the BDE, and work with aliases on the user's computer. A new page has been added with version 4, enabling you to install ODBC as part of your installation. Some of the options on these pages affect properties of the main *TInstall* component rather than add additional components to the main form.

The *TInstall* component also includes events and methods for customizing the behavior of the installation. In addition to the main *TInstall* component, Youseful includes other components that work in concert with, and are generally installed automatically by, the main component's wizard. These include components to help you use the BDE, and compress or decompress files. There are other file-related components, including one to manage .INI files, a registry-manipulating component, link-managing components, etc.

INFORMANT FACT FILE

Youseful is a VCL-based installation tool that supports three ways to distribute an application, including over the Internet. Its powerful Wizard leads you through the process of creating an installation project from beginning to end, including saving and compiling. You can fully customize an installation's appearance, use standard or custom installation dialog boxes, install the BDE, work with .INI files or the registry, establish file groups, and permit users to uninstall your application.

Bill White Software Productions
3117 Raymond Drive
Atlanta, GA 30340

Phone: (770) 457-1225
E-Mail: support@youseful.com
Web Site: http://www.youseful.com
Price: US$99 (includes 16- and 32-bit versions). Major upgrades are US$25; minor upgrades are free. Available in a shareware version.

Feature	Youseful	Wise
Type of tool	VCL-based	Script-based
Uses comprehensive wizard	Yes	Yes
Allows customization of installation's appearance	Yes	Yes
Includes standard dialog boxes	Yes	Yes
Supports adding custom dialog boxes	Yes	Yes
Supports installation of BDE	Yes	Yes; provides more network options
Supports file groups, specifying target folders	Yes	Yes
Supports "intelligent" upgrades with patch utility	No	Yes, in Enterprise edition
Supports uninstall	Yes	Yes
Supports Web-based and traditional installations	Yes	Yes, in Enterprise edition

Figure 2: A comparison of Youseful to the Wise Installation System.

Comparing Youseful

Some readers might be interested in a comparison between Youseful and one or both of the popular tools I mentioned at the beginning of this review. As mentioned, I have a very high opinion of the Wise Installation System (see Bill Todd's review in the October, 1998 issue of *Delphi Informant* for an excellent assessment). Figure 2 shows a comparison of some of the major features of the two products.

You'll notice that Youseful compares favorably with Wise. Still, in some circumstances, you may need something like the Wise Enterprise edition to solve problems. For example, I know of several developers who use Wise's patch facility to simplify the process of upgrading an application or tool.

Conclusion

The overriding appeal of Youseful is its component-based approach. If you prefer to do most of your development in Delphi, you should take a look at this product (available in a shareware version). Its powerful and extremely Youseful Wizard places it in a select group of RAD tools, making it an excellent model for future component libraries. I strongly recommend it. ∆

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

Case Study

By Chris Vandersluis

Keeping Time with Delphi

HMS Software Uses Delphi to Rewrite TimeControl

In 1995, HMS Software undertook a complete rewrite of its commercial timesheet system, TimeControl — a highly flexible and deployable commercial timekeeping solution. Having evaluated several development platforms, including Microsoft Visual C++, HMS chose Delphi Client/Server as the tool for its TimeControl 3. HMS based this decision on Delphi's strength in client/server tool sets, and its ability to directly support several target databases through SQL Links.

Founded in 1984, HMS Software specializes in implementing project control systems, and has frequently been asked to create department-wide timesheets for integration into the project-management process. The numerous timekeeping systems HMS created in its first decade were written in xBASE, starting with dBASE and ending with Microsoft Visual FoxPro in later years.

TimeControl was in its second incarnation when the TimeControl 3 project began. The first Windows version, TimeControl 2, was released in 1995 as a port from FoxPro for DOS to FoxPro for Windows. Even before the release of TimeControl 2, however, it was apparent the product would require an architectural change if it were to continue to grow. "The easiest thing for us would have been a port to FoxPro," said Stephen Eyton-Jones, Director of Technical Services for HMS. "We even did some prototyping in VFP. The problem was, it just didn't provide the tools we needed, or the performance required for this kind of application."

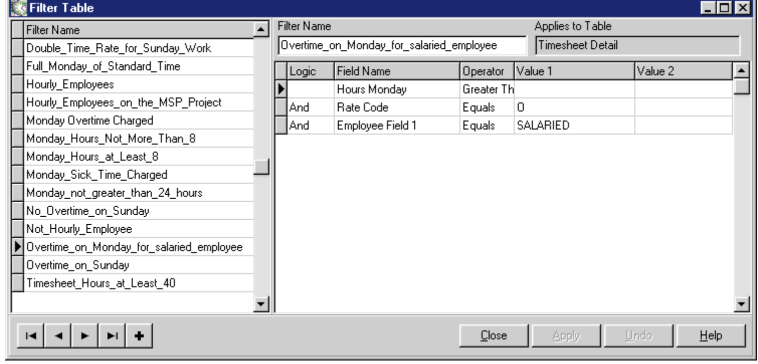


Figure 1: The Filter Table dialog box in TimeControl.

The Issues

The design constraints on TimeControl 3 were extensive. First, because a corporate timesheet system is perhaps the only database application in the enterprise likely to be distributed to every employee, it must be highly deployable. Second, as an enterprise-wide product, the database architecture would have to support the target architecture of the market. In the case of TimeControl, the requirement was support for multiple databases, including FoxPro, Oracle, Microsoft SQL Server, InterBase, Sybase, and Access. FoxPro would also provide support for legacy systems. Because these systems would be installed "sight unseen," provision of an installation script was necessary to allow for effortless installation by the client, regardless of the database.

Data security represented a third issue. As a financially-oriented system, the TimeControl database structure must protect the data itself. Also, access to information within the application must be on an as-needed basis. Some data would be unavailable to some users, and some would be made available in a read-only format.

Finally, reporting would have to be in a format familiar to financial personnel, but offer enough flexibility to meet both finance and project management requirements. For example, TimeControl 3 would have to be able to dynamically link to several commercial project-management scheduling systems, including Microsoft's Project, Primavera's P3, and Welcom's Open Plan.

Installation

Because installation was a major issue, HMS selected Wise Installation System. Custom database script

HMS is a provider of high-quality, leading-edge project management products and services, enabling organizations to be more effective. HMS released its first iteration of TimeControl in 1994. Now a full client/server system, TimeControl works for firms of all sizes from several employees, to several hundred, to several thousand. The new design allows for many new functions to be added to the system, including links to additional project management systems, and the ability to interface or integrate with major financial systems.

Third-party tools: Delphi, Wise Installation System, Formula One.

HMS Software
1000 St-Jean, Suite 711
Pointe Claire, QC H9R 5P1
Canada
Phone: (514) 695-8122
Fax: (514) 695-8121
Web Site: http://www.hmssoftware.ca

dialog boxes were required to allow users to select a database, then prompt for the necessary BDE settings. At that time, neither InstallShield nor Wise Installation System included all the functionality required to support multiple BDE entries. Therefore, HMS had to do significant modifications involving over 20 registry settings. Also, documentation for such an installation was non-existent. (Advances in later versions of both products have since simplified this process.)

Architecture

The early stages of development presented some immediate challenges. With all the databases that were supported, Query components were the best available method of interacting with the tables. However, each database carried its own quirks on how the ANSI92 standard for SQL queries was interpreted. In particular, reporting and timesheet-validation routines required an easy, menu-driven filtering capability where filters might even be defined within other filters.

For example, a filter name might be, "Show all timesheets where overtime was charged." A more complex filter might build upon this, e.g. "Of the timesheets where overtime was charged, show only those charged by salaried employees." The Filter Table dialog box would show fields, appropriate operators — such as "equals," "contains," "not empty," and "begins with" — and the corresponding values (see Figure 1). More complex statements were permitted by allowing new lines preceded by "and," "or," and "or not" operators.

The Filter Table dialog box would have to resolve to a valid SQL statement. However, because each database might (and often does) require a unique syntax of that statement, HMS developed a SQL Generator that first checks BDE settings to determine what database is in use, then generates the proper SQL statement for that database.

Security

TimeControl includes salary information for all staff. Therefore, HMS had to ensure that data was secure. Access to raw data is controlled by a "gateway" database, with only one table of two columns. An encrypted username and password are stored in the table's single record. This username/password combination is used to access the main database, allowing DBAs to provide more specific access to different data elements of the main database as required.

From an application standpoint, TimeControl includes functionality grouped into user profiles (see Figure 2). These profiles include lists of exceptions to controls viewable by the user. User profiles are linked to the user table and accessed during the log-on process. A user profile may contain menu items, report items, and even field items that are to be restricted.

Any field in TimeControl can be declared "read-only," "value-hidden," or "invisible" within the profile. Read-only is exactly as it

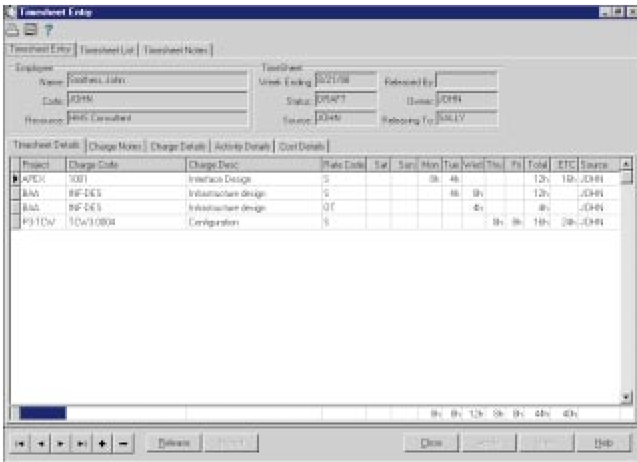


Figure 2: TimeControl's Timesheet Entry dialog box.

sounds: The field label and value are visible, but the value can't be changed by users with this profile. Value-hidden means that the field label is visible, as is the control for the value, but the value itself isn't displayed. An invisible property would make both the field label and the field value invisible. These properties are read as dialog boxes are created, making each dialog box, grid, or report in the system a customizable item. Administrators might use this functionality to make a "project rate" visible, while concealing an "actual rate" or "payroll rate."

Reporting

For reporting, HMS used Visual Components' Formula One for final presentation, and a report wizard to pre-condition the data. Although Formula One offers a choice of either sending the data via ODBC or populating the cells directly as objects, HMS was committed to avoiding ODBC as much as possible. Therefore, after the data was conditioned using a report wizard, the information was inserted into each cell in turn until the report was ready to be presented.

Happy Ending

Since its release in January, 1998, market acceptance of TimeControl 3 has been spectacular. Major organizations, such as Anderson Consulting, Bombardier/Canadair, Credit Suisse/First Boston Bank, EDS, and the US Navy, have adopted the system. The attraction lies primarily in its architecture and ability to adapt to the client's database standard.

Development for TimeControl 3 has now shifted toward an even greater degree of deployability. Scheduled for release in early 1999, TimeControl 3.1 includes a Java interface written using Inprise's JBuilder Client/Server edition. Designed in a true, three-tiered architecture, TimeControl on Java includes a pure Java middle tier connecting to the database using JDBC (or, in the case of SQL Server, the JDBC/ODBC Bridge). RMI is used to connect to a thin applet. The applet, designed to run in almost any browser, is paper-thin, running about 50KB in size. Data controls have all been placed in the middle-tier application, ensuring that there is no need to compromise data security by leaving an open port through the firewall.

So far, only the TimeControl timesheet itself has been created in this environment. Initial response to the new architecture is so positive, however, that HMS has already identified it as key to future development. Applications like TimeControl are most

attractive to the marketplace when they include both a client/server application and a browser interface. Delphi and JBuilder have enabled HMS to deliver both. ∆

Chris Vandersluis is the president of HMS Software, based in Montreal, Canada. He has been involved in the automation of project control environments since 1983 and has been published in a variety of publications, including *Fortune Magazine* and *Computing Canada*. He can be reached at chrisv@hmssoftware.ca.

Together at Last: Borland and Project JEDI

Is there any long-time Borland developer who did not feel at least a little anxiety when the company decided to change its name to Inprise? Is there anyone familiar with Project JEDI — who has witnessed its ups and downs and its near disappearance — who is not pleasantly surprised that we are still talking about this important voluntary initiative? Even more, is there anyone who could have predicted that after a year and a half of its rather tenuous existence that it is now on the brink of its most important initiative? Recent developments at Inprise and Project JEDI are intertwined in interesting ways.

When Inprise announced it was splitting into two divisions, including the Web-based "Borland Division," it received mixed reviews in some of the Internet discussion groups. Because the reorganization entailed laying off a number of employees, one could view the glass as "half full" or "half empty." For some, it portended an intermediate step toward Inprise divesting itself of its development tools — Delphi, C++Builder, and JBuilder — so that it could concentrate fully on its other objectives. For others, however, it indicated at least the possibility that Inprise was ready to demonstrate in a meaningful way that it recognized the importance of its developer base.

It does feel good to be able to say "Borland" again. But the new initiative at Inprise seems more than cosmetic. The new developments in Project JEDI and its partnership with Borland are even more encouraging. It seems Borland folks have been lurking on the JEDI discussion thread for a long time, considering when the best time might be to get involved. As I have stated on several occasions in the past, this project has had its ups and downs. That is understandable, given its voluntary nature and the busy lives we developers tend to live. So, one of the considerations that must have affected Borland's decision was making certain that there was an administrative structure in place that would make any joint endeavors workable.

Project JEDI has been largely dormant for several months while several of us committed to the project have continued to discuss ways to get it "moving again." In the past six months, a new leader has emerged: Thomas Guarino. A developer who lives in Indiana, Tom has been working hard to keep the Project moving forward. He recently invited several of us to be a part of a new administrative group, including Michael Beck of Ohio and another founding JEDI leader, Helen Borrie of Australia.

Among others involved in the new initiative are two very well-known Delphi figures, both of whom are making important contributions: Charlie Calvert and Bob Swart. Bob has been a strong and important supporter of the Project from the beginning, making his Header Conversion tool freely available to JEDI conversion teams. He has promised to revise that tool to handle the new specification required for those conversions, which Borland will distribute.

Charlie has become, in effect, the liaison between Project JEDI and the Delphi R&D Team. He is also very visible at the new Borland site with his Tech Voyage page, http://www.borland.com/techvoyage, which includes a number of his articles on a variety of Delphi and C++Builder topics. There's a link on that page to a description of the Borland JEDI Initiative at http://www.borland.com/techvoyage/jediinitiative.html. For the latest on Project JEDI, be sure to visit the Project JEDI Web site at http://www.delphi-jedi.org.

One of the first challenges for the project concerned the stringent requirements that a conversion must meet to receive Borland's endorsement. While there was a lively discussion about this among the administration team, and later on the JEDI discussion thread, most seemed to understand the need to accept the specification. At the same time, Project JEDI will maintain its autonomy and identity. As I write this, many things are still in a state of flux. However, regardless of the level of success of this joint venture between Borland and Project JEDI, I believe we can draw one conclusion from it: The Delphi community — both within Borland and among those who depend upon it for their outstanding tools — continues to show signs of vitality and innovation. 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 on the Internet at acmdoc@aol.com.