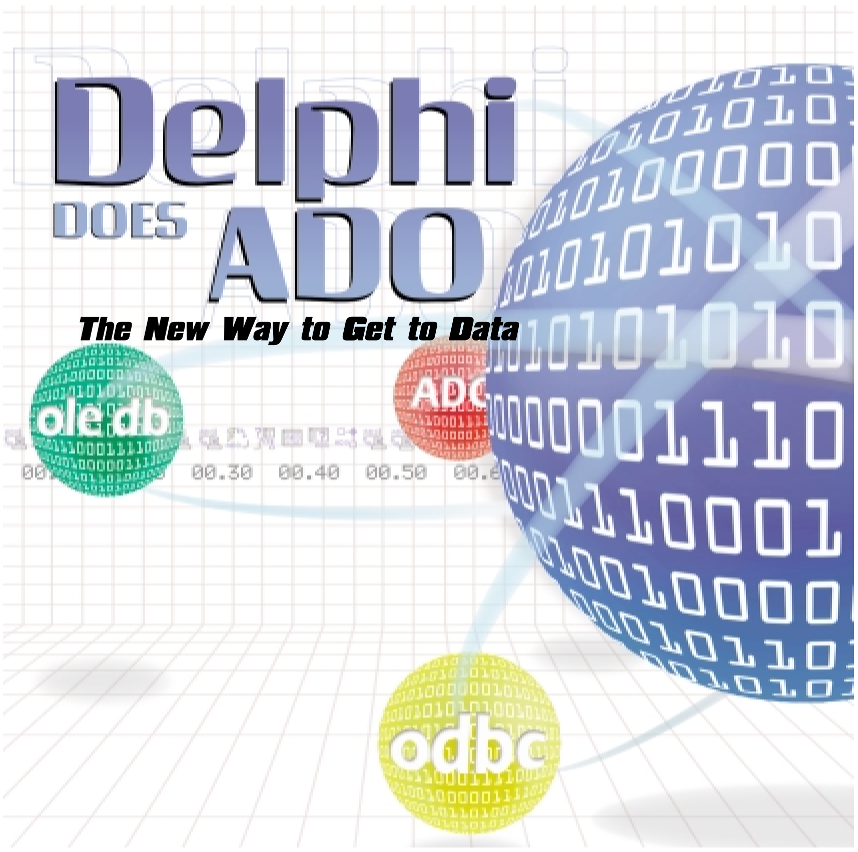
October 1999, Volume 5, Number 10

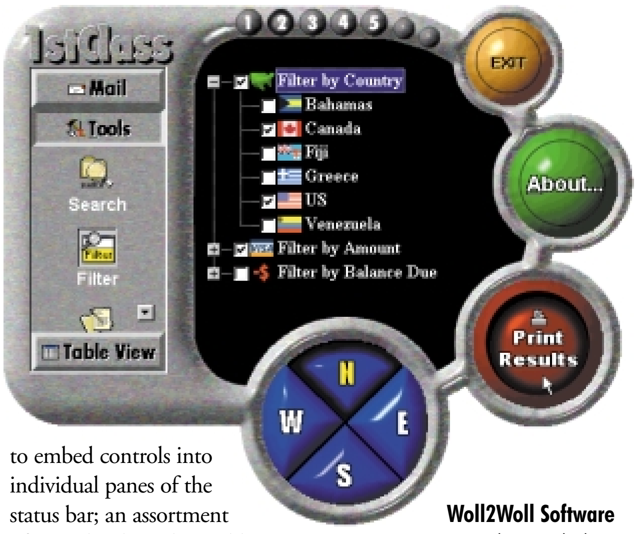


Cover Art By: *Darryl Dennis*

Woll2Woll Releases 1stClass

Woll2Woll Software announced *1stClass*, a new component suite for Delphi and C++Builder.

1stClass' main features include full support for InfoPower; an advanced tree-view control supporting embedded checkboxes, radio-buttons, multi-selection, and more; a sophisticated data-bound tree-view, which can navigate all the tables in your master/detail relationships; the 1stClass TreeCombo for hierarchically organizing and displaying items in a drop-down list; ShapeButton, which has built-in support for displaying buttons as arrows, diamonds, ellipses, rounded rectangles, or triangles; ImageButton, which allows you to load different bitmaps for the up and down states of the button; a StatusBar control allowing you to embed controls into individual panes of the status bar; an assortment of specialized combo and list-box controls; a Label control that supports many special effects, including outlines, shadows, extrusions, and the ability to rotate text; and more.



Woll2Woll Software
Price: 1stClass Standard, US$199; 1stClass Professional, US$299 (includes source code and C++Builder compatibility).
Phone: (800) 965-2965
Web Site: http://www.woll2woll.com

Triple-T Releases Style One 2.0

Triple-T Software announced the release of *Style One 2.0*, a Windows 95/98/NT utility that adds style to Web pages. Creating Cascading Style Sheets (CSS), Style One allows Web developers to enhance Web sites with graphics and a consistent look and feel.

Unlike Java, JavaScript, CGI, and ASP techniques, which are often filtered out by firewalls, style sheets are ignored by older browsers, yet allow newer browsers such as Netscape 4 and Internet Explorer 4 to display sophisticated layouts.

Style One allows Web developers to create styles that will control font type, alignment and colors, background graphics and colors, margins, spacing, and more. Web developers can create and store each style once, and reference it from all Web pages.

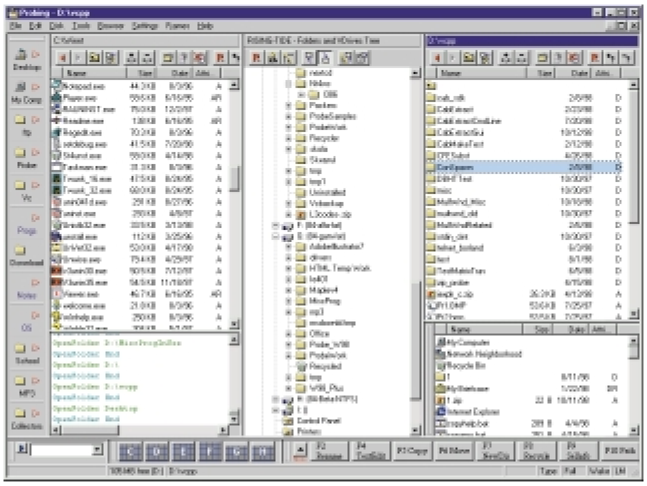
The program supports stand-alone and embedded style sheets. Style One's style sheet parser allows editing of every CSS property. Style One also lets Web developers preview their handiwork using their favorite external browser.

Style One supports level 1 and level 2 CSS standards.

Triple-T Software
Price: US$20
Phone: 10-4702396
Web Site: http://www.3-t.com

Z-Soft Announces FileProbe 1.02

Z-Soft released *FileProbe 1.02* for Windows 95/98/NT, a new version of the company's advanced file management environment for Win32. FileProbe brings much of the power previously only available in command line/scripting environments to the realm of graphical file managers.



Using FileProbe, one can compare folders containing different versions of the same project and highlight the differences. With the FileProbe selection engine, one can quickly select thousands of files across multiple directories meeting a certain filename and/or file date criterion. This enables selective backup operations.

When doing large backup, copy/move, extraction, compression, upload, or download operations, FileProbe runs them in the background, so the program is left active and responsive. Any number of such file operations may be in progress at the same time.

FileProbe can access and modify local drives, networked drives, compressed archives (Zip, Arj, Ace, Cab, Gz, Tar, etc.), MS Windows shell, FTP connections, and more.

Features include regular expression filename matching, multi-folder selections, multiple concurrent file operations, transparent archive contents access, MS Explorer shell compatibility, folder synchronization, bulk file operations, FTP-to-FTP transfers, a generic file search engine, and a highly configurable user interface.

Z-Soft
Price: Between US$25 and US$55, depending on platform and program version; volume discounts are available.
E-Mail: sales@z-soft.com
Web Site: http://z-soft.com

PGC's InstallConstruct 3.1 Is Shipping

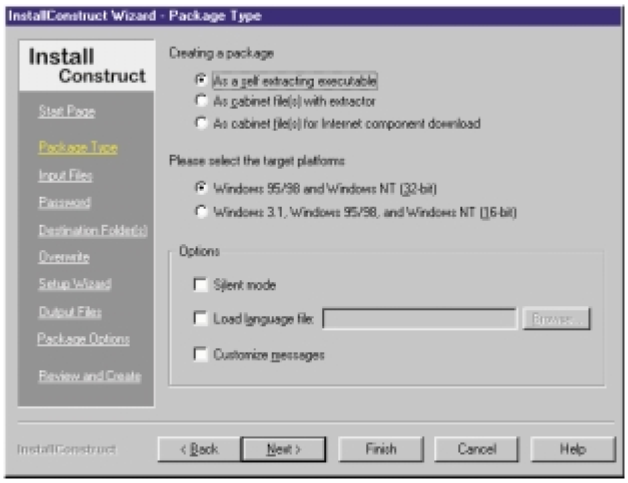
Pacific Gold Coast Corp. announced the release of *InstallConstruct 3.1*, a suite of wizards and tools for creating Installer, Setup Wizard, Uninstaller, and HTML-based Internet Component Download installers for Windows 3.1/95/98/NT applications.

Users of InstallConstruct 3.1 can create and customize Setup Wizard with their own product graphics and logos, and display formatted text for Internet and intranet distribution of program, single, and multi-volume CD-ROM and disk distributions.

InstallConstruct 3.1 also automatically records the selected package options of a project as a package script file (*.adx). These script files not only save you from the repetitive task of creating other similar packages and in updating the existing ones manually, they also support command-line batch processing in unattended operation, so they can be created without user prompts.

This latest release adds user-defined customization capabilities for graphics and 3D logo text, multiple levels of compression, and the creation of stand-alone setup programs. In addition, this new release provides "hidden password" package protection, which protects the package from being extracted externally by other software.

International languages are supported, including English, Danish, Dutch, Finnish, French, German, Japanese, Norwegian, Portuguese, and Swedish.


InstallConstruct Wizard - Package Type

Creating a package

- [✓ As a self extracting executable]
- As cabinet file(s) with extractor
- As cabinet file(s) for Internet component download

Please select the target platforms

- [✓ Windows 95/98 and Windows NT (32-bit)]
- Windows 3.1, Windows 95/98, and Windows NT (16-bit)

Options

- ☐ Silent mode
- ☐ Load language file:
- ☐ Customize messages

Pacific Gold Coast Corp.
Price: US$199
Phone: (800) 732-3002
Web Site: http://www.pgcc.com/ic

Software Productivity Centre Announces ESTIMATE Professional 4.0

Software Productivity Centre Inc. announced version 4.0 of its software project planning and estimation tool, *ESTIMATE Professional*.

ESTIMATE Professional 4.0 allows managers to estimate the effort of developing each component, or to import effort allocations from a scheduler, such as Microsoft Project or a spreadsheet application. Users can then benefit from the tool's ability to account for uncertainty in the resulting forecast of staffing levels, total effort, and risk. The resultant range of possible schedules and their likelihood of being met allows project managers to plan their projects with confidence.

The tool not only takes into account the variability of the development environment, but also the diversity of its users. This latest version allows users to customize the interface according to their personal preferences. This, together with significantly enhanced import and export features, makes the task of project management easier.

ESTIMATE Professional 4.0 injects knowledge of software project behavioral characteristics into the planning process. The tool works hand in hand with Microsoft Project and the like to provide project managers and team leaders with a full solution for maintaining project control.

Software Productivity Centre Inc.
Price: US$695 for single-user license.
Phone: (604) 662-8181
Web Site: http://www.spc.ca

Enterprise One Unveils Jaadu 2.0

Enterprise One announced *Jaadu 2.0*, the company's Web server VCL for Delphi.

Adding the *TJaadu* component to an application allows it to respond directly to requests from Web browser clients. Web-based applications will no longer require the use of Microsoft's IIS server or any other Web server. Because the Web server is compiled directly into the application, Jaadu provides a high-performance solution.

The programmer is free to use all of Delphi's RAD tools without dealing with the DLL issues associated with producing ISAPI-based Web applications. Using the Jaadu components, a few lines of code is all that is needed to fetch a set of data from any database supported by Delphi and use that data to produce formatted HTML output based on a template the programmer supplies. These templates can be created using Microsoft's FrontPage product or any other HTML editor, allowing the developer to take advantage of the latest HTML tools for Jaadu-based projects.

Jaadu 2.0 includes components for the automatic management of user sessions. Also, there is a *TJaaduProcessor* component that greatly simplifies the building of interactive Web sites.

Enterprise One
Price: From US$149 (Jaadu for Delphi Pro).
Phone: (888) 727-5281
Web Site: http://www.jaadu.com

Pegasus Software Releases ImagN' 4 with CadXpress

Pegasus Software released *ImagN' 4*, which offers developers high-speed access to image files from databases, the Internet, intranets, and more. ImagN' 4 adds image file support for WMF and PNG, and AutoCAD 14 support for DXF and DWG image files. The new CadXpress support makes it easy to display AutoCAD files from within a browser, database, or other Windows environment. ImagN' 4 includes image features, such as thumbnails, transparencies, smoothing, improved anti-aliasing, rotate, zoom, crop, copy-to-clipboard, rubberbanding, and more. ImagN' 4 incorporates a new TIF G3/G4 technology to its display engine that should increase the display time by at least two-fold.

ImagN' 4 includes a new annotation/redlining engine that enables developers to add annotations to any image file, including DXF and DWG files. These annotations are displayed in layers, making it easy to password-protect certain layers, or delete unwanted annotations without affecting the image or other annotations. The developer has the choice to add these annotations to the image, or save them in an independent file to be recovered for later use.

ImagN' 4 is currently available as a DLL/VCL. ImagN' is compatible with and has sample code for Delphi, Visual Basic, IE, Access, Visual FoxPro, PowerBuilder, Clarion, C/C++, and more.

Pegasus Software
Price: From US$299 to US$499.
Phone: (800) 875-7009 or (813) 875-7575
Web Site: http://www.pegasustools.com

EvoCorp Releases ActiveIE Component Suite

EvoCorp Corporate Solutions announced the release of the *ActiveIE Component Suite*, a set of native Delphi VCL objects and components that provide a stable environment for Internet Explorer within a Delphi application.

ActiveIE is a safe replacement where current *TWebBrowser* use induces access violations, exhausts system resources, or hangs the system. Because it's built on Internet Explorer's ActiveX controls, ActiveIE remains 100 percent compatible with your current Web browser framework, requiring minimal changes to existing code.

EvoCorp Corporate Solutions
Price: US$30 per single user or developer license.
E-Mail: sales@evocorp.com
Web Site: http://www.evocorp.com

Shaman Delivers Enterprise Shaman 3.2.3

Shaman Corp. announced the implementation of Y2K BIOS testing within *Enterprise Shaman 3.2.3*. Enterprise Shaman is an Internet-based SRM solution that dramatically increases software reliability throughout the corporate enterprise.

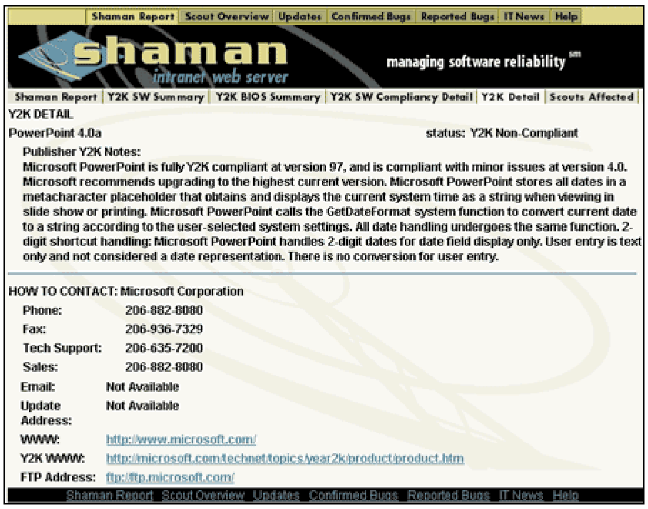
Enterprise Shaman supplies the IT manager with a complete software and hardware inventory of their network, informs them of the Y2K compliance status for their software, reports on compliance levels for each desktop's BIOS, and delivers Y2K software updates as they are released by software vendors.

The new BIOS testing feature uses the industry standard tests licensed from the National Standards Testing Laboratory in addition to a selection of tests and features incorporated by Shaman. The BIOS testing process consists of an automated clock compatibility test, a progression test, a leap year test, a suspect BIOS test, and a reboot test.

The Shaman Y2K BIOS test is an automated process and, once complete, the results are copied to the Shaman Scout, a software agent that resides on the end user's machine that relays software reliability information back to the Shaman Enterprise Server. The Shaman Enterprise Server is a central repository located behind the firewall that collects, organizes, and reports on software reliability information for the network.

The Shaman Enterprise Server automatically generates intranet pages displaying a variety of software reliability information for the network. Through any browser, an IT professional can access the administrator's page that displays Y2K software compliance, BIOS compliance, software and hardware inventory, bug information, available updates, and upgrades for every networked computer.

Shaman Corp.
Price: Contact Shaman for pricing.
Phone: (415) 241-9952
Web Site: http://www.shamancorp.com


Shaman Report | Scout Overview | Updates | Confirmed Bugs | Reported Bugs | IT News | Help
shaman
intranet web server
managing software reliability sm
Shaman Report | Y2K SW Summary | Y2K BIOS Summary | Y2K SW Compliancy Detail | Y2K Detail | Scouts Affected

Y2K DETAIL

PowerPoint 4.0a status: Y2K Non-Compliant

Publisher Y2K Notes:
Microsoft PowerPoint is fully Y2K compliant at version 97, and is compliant with minor issues at version 4.0. Microsoft recommends upgrading to the highest current version. Microsoft PowerPoint stores all dates in a metacharacter placeholder that obtains and displays the current system time as a string when viewing in slide show or printing. Microsoft PowerPoint calls the GetDateFormat system function to convert current date to a string according to the user-selected system settings. All date handling undergoes the same function. 2-digit shortcut handling: Microsoft PowerPoint handles 2-digit dates for date field display only. User entry is text only and not considered a date representation. There is no conversion for user entry.

HOW TO CONTACT: Microsoft Corporation

Phone:	206-882-8080
Fax:	206-936-7329
Tech Support:	206-635-7200
Sales:	206-882-8080
Email:	Not Available
Update Address:	Not Available
WWW:	http://www.microsoft.com/
Y2K WWW:	http://microsoft.com/technet/topics/year2k/product/product.htm
FTP Address:	ftp://ftp.microsoft.com/

Shaman Report Scout Overview Updates Confirmed Bugs Reported Bugs IT News Help

Inprise Announces Borland Delphi 5

Philadelphia, PA — Inprise Corp. announced Borland Delphi 5, a major new version of its rapid application development tool for Windows. Designed to help individual and corporate developers rapidly deliver Windows applications, and extend those applications to the Internet, Delphi 5 simplifies the integration of Windows and browser clients, Web servers, middleware, and back-end database systems. Delphi 5 includes support for HTML 4 and XML.

Delphi 5 includes a number of enhancements that allow organizations to rapidly extend existing systems and build new systems for the Internet, increase the productivity of teams developing applications in larger corporations, and reduce development cycle time. These enhancements include Internet Express, which speeds Internet and XML development by simplifying Data Distribution and optimizing Data Exchange; HTML 4 support to create full-featured dynamic thin-clients for the Web and allow rapid deployment to the Internet with full-featured and responsive client applications; MIDAS 3, which handles the demands of Internet-based applications; ADOExpress for access to all types of information; InterBase Express, a low-maintenance, small-footprint relational database; TeamSource, which lets development teams manage changes to source code; Borland Translation Suite to quickly internationalize or localize applications for new languages and cultures; integrated development environment enhancements; and advanced debugging tools.

Delphi 5 is scheduled to be available in three editions: Delphi 5 Enterprise, Delphi 5 Professional, and Delphi 5 Standard. Delphi 5 Enterprise has an estimated street price (ESP) of US$2,499 for new users. Delphi 5 Professional has an ESP of US$799. Delphi 5 Standard has an ESP of US$99.95.

For more information, visit http://www.borland.com/delphi.

Inprise Announces the Office of Chief Scientist

Philadelphia, PA — Inprise Corp. announced the appointments of Chuck Jazdzewski, Blake Stone, and Dr Andreas Vogel to the Office of Chief Scientist. Inprise organized the Office of Chief Scientist, which draws from key technical people within the company, to facilitate growth and shape the technical direction for Inprise.

During his tenure at Inprise, Jazdzewski made significant contributions to many products and technologies, including Delphi, JBuilder, OWL, and Turbo Debugger. Prior to his appointment, Jazdzewski held positions as senior staff engineer and C++Builder contributing architect.

Blake Stone, a lead member of Inprise's JBuilder team since he joined Inprise in 1997, was promoted to senior staff engineer and to the Office of Chief Scientist. Stone has been an instrumental architect and visionary for Inprise's JBuilder products, including the upcoming JBuilder for both the Sun Solaris and Linux operating systems.

Dr Andreas Vogel has been a key member of Inprise's engineering team since 1997. Vogel works on the Inprise Application Server and other next-generation, Internet-enabling enterprise products. Before joining Inprise, Vogel was a principal research engineer with Distributed Technology Centre in Brisbane, Australia. He also co-authored *Java Programming with CORBA* [John Wiley & Sons, 1998], *C++ Programming with CORBA* [John Wiley & Sons, 1999], and *Programming with Enterprise JavaBeans, OTS, and JTS* [John Wiley & Sons, 1999].

Inprise Wins Four Awards at JavaOne

Scotts Valley, CA — Inprise Corp. announced that three of its key software products received awards from leading Java publications. *Java Developer's Journal* and *Java Pro* magazine presented Inprise with Reader's Choice Awards for its Inprise VisiBroker, Inprise JBuilder, and Inprise Application Server at the JavaOne Conference and Exposition in San Francisco.

Inprise's VisiBroker won two awards for Best Java Middleware from the readers of *Java Developer's Journal* and *Java Pro* magazine. Inprise JBuilder received the Best Obfuscation Tool Award in the *Java Pro* Reader's Choice Awards, and was also named a finalist in the Best Java Integrated Development Environment category in *Java Developer's Journal* Reader's Choice Awards. In addition, Inprise's Application Server won *Java Pro*'s Reader's Choice Award for Best Middleware EJB support.

Inprise Announces Commitment to Java

San Francisco, CA — Inprise Corp. demonstrated support for Sun Microsystems' Java platform with a series of announcements. Inprise is shipping Borland JBuilder 3, a new version of its of visual development tools for creating platform-independent Java business and database applications, as well as JDataStore, a database written entirely in Java for embedded, Web, and mobile database applications. The company is also previewing its new JBuilder for Solaris product, a professional Java development environment for the Solaris Operating Environment, and introducing its new EJB (Enterprise JavaBeans) Server technology, which will be included in the next version of the Inprise Application Server.

As of press time, JBuilder for Solaris was scheduled to be available by the end of the calendar year (1999). JBuilder for Linux is scheduled to be available early next year. The EJB Server technology will be a primary feature in the next version of the Inprise Application Server, which is scheduled to be available by the end of the year.

To learn more, visit http://www.inprise.com.

table of CONTENTS

By Bill Todd

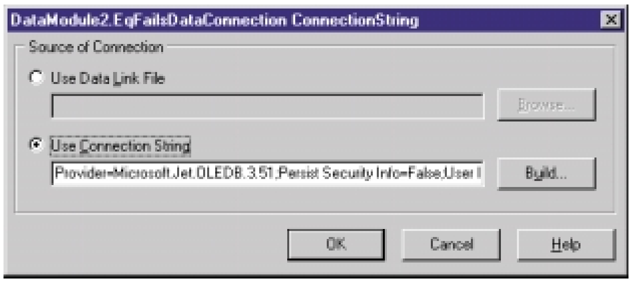
Delphi Does ADO

The New Way to Get to Data

Universal Data Access (UDA) is part of Microsoft's strategy to provide fast access to data in both relational and non-relational data stores. UDA provides a language-independent, easy-to-use API for accessing data in any data source that has a UDA-compatible driver. Like the BDE, this technology makes it easy to access data from multiple data sources in a single program. UDA is implemented using Microsoft Data Access Components (MDAC), which includes Active Data Objects (ADO), Open Database Connectivity (ODBC), and OLE DB.

ADO is the application programming interface of MDAC, and OLE DB is the system-level interface. OLE DB defines a suite of COM interfaces that provide all the data-access capabilities required by any data source, from a relational database to a file system. ODBC is included in MDAC for backward compatibility. While existing ODBC drivers will likely be replaced by OLE DB providers in the future, the Microsoft OLE DB provider for ODBC lets you use any ODBC driver via ADO now. Although ADO is relatively new, OLE DB providers are already available for Microsoft Access, Microsoft SQL Server, and Oracle.

Another major advantage of ADO is that it will be built into all future Microsoft operating systems, including Windows 2000. While this means that today you must install ADO on each PC that will use ADO to access data, that task will vanish in the future. If you want to learn more about UDA and ADO, visit Microsoft's data access Web site at http://www.microsoft.com/data/default.htm. From this page, you can download the ADO redistributable, which allows you to install ADO on Windows 95/98/NT machines, or the MDAC SDK, which contains complete documentation and everything you need to develop your own OLE DB providers. The SDK also includes the ADO redistributable.


DataModule2.EqFailsDataConnection ConnectionString

Source of Connection

Use Data Link File [ ]

[✓ Use Connection String]

Provider=Microsoft.Jet.OLEDB.3.51;Persist Security Info=False;User I

Figure 1: The Connection String Editor.

Everything you need to use ADO with Delphi is on the Delphi 5 CD, including MDAC. Simply go to the MDAC folder on the Delphi 5 CD and run the installation program, MDAC_TYP.EXE. The MDAC installation program is a single .EXE file, so it's easy to install MDAC anywhere you need it. You can also use the MDAC installation program to install MDAC as part of your application's installation if you're using an installation program that supports calling .EXEs (InstallShield Express does not). If you're installing MDAC as part of your application's installation, you'll want to use the "silent" mode to suppress all screen displays. To install in silent mode, use the command:

```
mdac_typ.exe /q:a /c:"setup.exe /qt"
```

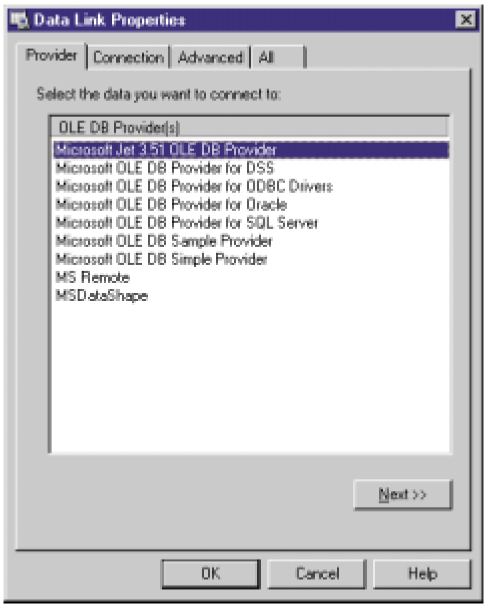
For more information on installing MDAC, including file lists and dependencies, see the MDAC SDK documentation.

Using the ADOConnection and ADODataSet Components

Delphi 5 has a suite of six new components that provide complete ADO support and an easy way to convert existing applications to ADO. To begin building an ADO application, drop an ADOConnection component on a form or data module. The ADOConnection component is the ADO equivalent of the BDE Database component. It allows you to define a connection to a database using its *ConnectionString* property.

Though it's possible to build a connection string manually, it is difficult. The ADO connection string consists of a semicolon-delimited list of many parameters that can easily exceed 150 characters. Fortunately, Microsoft provides a Connection String Editor to make this job easier. To open the Connection String Editor, shown in Figure 1, click the ellipsis button in the *ConnectionString* property's edit box, or double-click on the component.

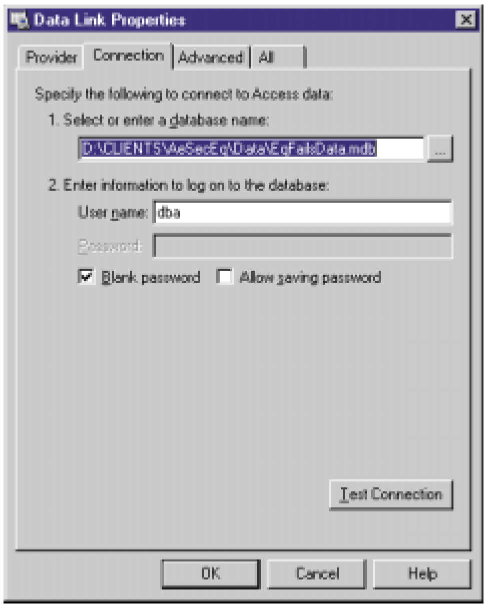
The easiest way to build a connection string is to click the Build button to display the Data Link Properties dialog box, shown in Figure 2. The Provider page lets you choose the driver you want to use.



Select the data you want to connect to:

OLE DB Provider(s)
Microsoft Jet 3.51 OLE DB Provider
Microsoft OLE DB Provider for DSS
Microsoft OLE DB Provider for ODBC Drivers
Microsoft OLE DB Provider for Oracle
Microsoft OLE DB Provider for SQL Server
Microsoft OLE DB Sample Provider
Microsoft OLE DB Simple Provider
MS Remote
MSDataShape

Figure 2: The Data Link Properties dialog box.



Specify the following to connect to Access data:

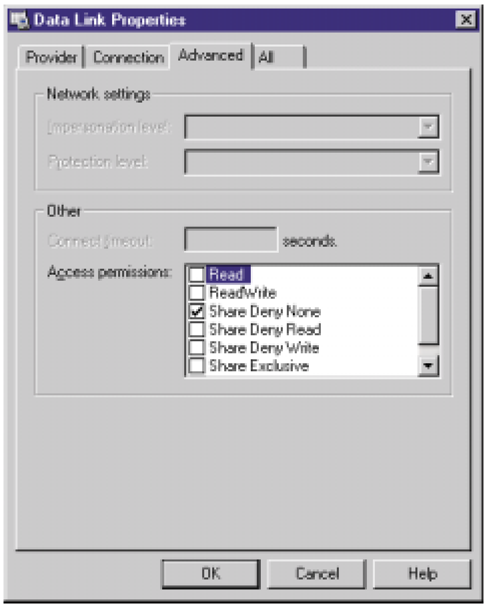
1. Select or enter a database name: D:\CLIENTS\AeSecEq\Data\EqFailsData.mdb
2. Enter information to log on to the database:

User name: dba
Password:
[✓] Blank password [ ] Allow saving password

Figure 3: The Connection page.

What you see on the Connection page depends on the provider you select. Figure 3 shows the Connection page with the Microsoft Jet provider selected, as well as the path to an Access database entered.

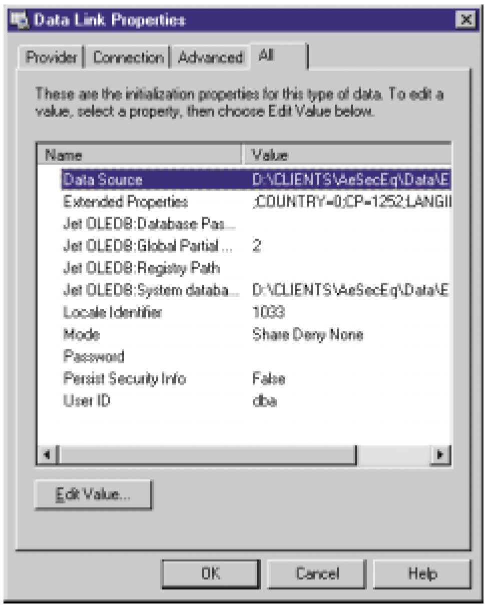
The Advanced page, shown in Figure 4, lets you specify the type of access to the database, and the All page (see Figure 5) lets you edit any value in the connection string. The All page is particularly important if you're connecting to an Access database with user-level security, because it's the only place you can enter the path to the system database.



Network settings
Impersonation level:
Protection level:

Other
Connect timeout: seconds.
Access permissions:
[ ] Read
[ ] ReadWrite
[✓] Share Deny None
[ ] Share Deny Read
[ ] Share Deny Write
[ ] Share Exclusive

Figure 4: The Advanced page.



These are the initialization properties for this type of data. To edit a value, select a property, then choose Edit Value below.

Name	Value
Data Source	D:\CLIENTS\AeSecEq\Data\E
Extended Properties	;COUNTRY=0;CP=1252;LANGII
Jet OLEDB:Database Pas...	
Jet OLEDB:Global Partial ...	2
Jet OLEDB:Registry Path	
Jet OLEDB:System databa...	D:\CLIENTS\AeSecEq\Data\E
Locale Identifier	1033
Mode	Share Deny None
Password	
Persist Security Info	False
User ID	dba

Figure 5: The All page.

Once a value has been assigned to the *ConnectionString* property, you can set the *Connected* property to True, at design or run time, to connect to the database. The ADOConnection component also provides transaction support through its *BeginTrans*, *CommitTrans*, and *RollbackTrans* methods.

The ADODataSet component is really the only one you need to work with data because it allows you to work directly with a table, execute a SQL statement, work with the result set, or call a stored procedure. After dropping an ADODataSet on a form or data module, the first step is to set its *Connection* property. The *Connection* property's drop-down list will display all the available ADOConnection components. Next, you need to set two related properties: *CommandType* and *CommandText*. Set *CommandType* first because it determines how *CommandText* is interpreted. You can set *CommandType* to indicate that you want to connect directly to a table, call a stored procedure, or enter a SQL statement as text. Choosing *cmdTable* as the *CommandType* causes the drop-down list for the *CommandText* property to display the tables in the database.

Once *CommandType* and *CommandText* have been set, using the ADO components is exactly like working with the BDE dataset components. Drop a DataSource component, a DBNavigator, and some data-aware components on your form. Set the *DataSet* property of the DataSource to the ADODataSet component, and set the *DataSource* property of the navigator and data-aware controls.

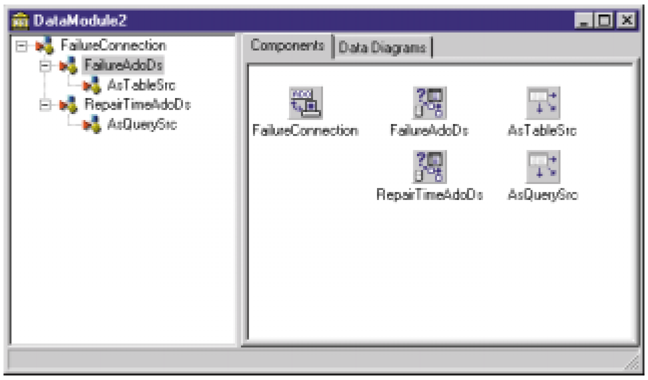
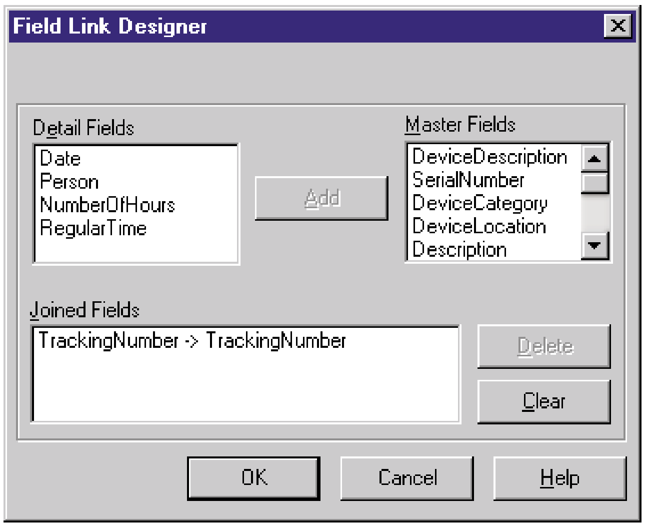


Figure 6: Linked ADODataSet components.

Figure 6 shows a data module containing an ADOConnection, two ADODataSet components, and two DataSource components modeling a one-to-many relationship between two tables in an

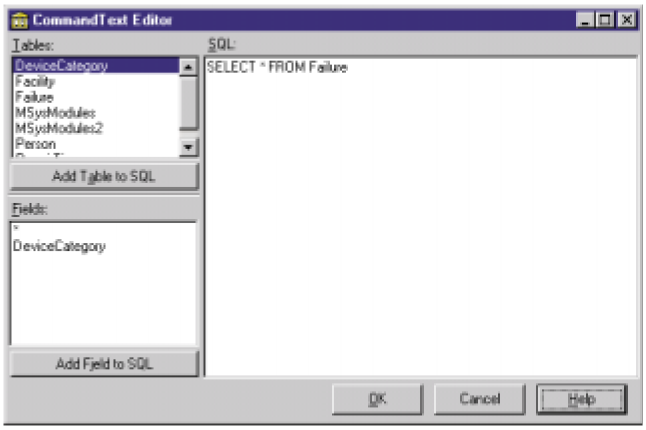


Detail Fields: Date, Person, NumberOfHours, RegularTime

Master Fields: DeviceDescription, SerialNumber, DeviceCategory, DeviceLocation, Description

Joined Fields: TrackingNumber -> TrackingNumber

Figure 7: The Field Link Designer.



Tables: DeviceCategory, Facility, Failure, MSysModules, MSysModules2, Person

Fields: *, DeviceCategory

```
SELECT * FROM Failure
```

Figure 8: The CommandText editor.

Access database. The master table is FailureAdoDs, and the detail table is RepairTimeAdoDs. The datasets were linked by setting the *DataSource* property of the detail dataset to the DataSource component of the master dataset, then setting the *MasterFields* property of the detail dataset.

The property editor for the *MasterFields* property is the Field Link Designer, shown in Figure 7. To link the tables, select the master and detail fields that define the relationship between the tables, and click the **Add** button. In this example, the TrackingNumber field links the tables. If the relation is defined by more than one field, repeat the process of selecting the corresponding master and detail fields and clicking the **Add** button.

To use an ADODataSet with a query result set, change the *CommandType* to *cmdText* and enter the SQL statement in the *CommandText* property. With the *CommandType* set to *cmdText*, the property editor for the *CommandText* property changes to the CommandText editor, shown in Figure 8.

The CommandText editor is a major improvement over the String List Editor, used to edit SQL commands in previous versions of Delphi. It provides a list of tables, and a button to add the table name to the SQL statement, as well as a list of field names for the selected table. Even if you don't use the **Add** buttons, the list of table and field names is very handy. Creating a one-to-many link between the ADODataSet components that execute SQL statements is exactly the same as linking two BDE Query components. The SQL statement for the detail dataset is:

```
SELECT *
  FROM RepairTime
 WHERE TrackingNumber = :TrackingNumber
```

The name of the parameter in the WHERE clause, `:TrackingNumber`, matches the name of the primary key in the master table exactly. The detail dataset's *DataSource* property is set to the master table's DataSource component. Because these two conditions have been met, each time the master dataset is positioned to a new record, the detail dataset is automatically closed, the new value from the master record is assigned to the query parameter, and the detail dataset is opened to retrieve the new set of detail records.

If you will execute a query more than once with different parameters, set the ADODataSet's *Prepared* property to True. This will cause the query plan to be prepared and stored the first time the query is executed. The stored plan will be used for each subsequent execution. This eliminates the time required to parse and optimize the query for all executions except the first.

To work with a stored procedure, set the *CommandType* to *cmdStoredProc*, and choose the stored procedure from the *CommandText* property's drop-down list. Use the ADODataSet's *Parameters* property to assign values to input parameters and retrieve values from output parameters.

Although you can do everything with the ADODataSet component, Delphi 5 also includes the ADOTable, ADOQuery, and ADOStoredProc components. These are designed to resemble the BDE Table, Query, and StoredProc components as closely as possible to make converting an application to ADO easy.

Should You Convert to ADO?

Why convert an existing application from BDE to ADO? Neither the native BDE Access driver nor the Access ODBC driver have been ideal solutions for working with Access databases. Using the ADO Jet driver eliminates these problems. With ADO, your Access applications will correctly detect changes made by other users and warn you when you try to post a record that has been changed by another user since you read it. Also, Autoincrement fields work correctly with default values set for other fields.

The *big* advantage of using ADO with any database, however, is that you are no longer dependent on Borland to update drivers when new releases of the database appear. When a new version of SQL Server or Oracle is released, the new ADO drivers should be available at the same time, and should work because the database vendor writes them.

The ADOCommand Component

In addition to the components for working with datasets, Delphi 5 also provides the ADOCommand component. The ADOCommand component is most useful for executing commands that don't return a result set, such as SQL DDL (Data Definition Language) commands, or a SQL DELETE query.

If you're using one or more ADOConnection components, click the drop-down button in the *Connection* property of the

ADOCommand component and select the connection you want to use. The ADOCommand component, like all the ADO dataset components, has its own *ConnectionString* property so you don't have to use an ADOConnection component. However, in most cases, you'll want to. The connection component provides a single central place to change the *ConnectionString* and any other connection-related properties, as well as providing transaction control methods.

The *CommandText* property of the ADOCommand component contains the command you want to execute, and the *CommandType* property determines whether *CommandText* is interpreted as a text string, table name, or stored procedure name. Set *CommandType* to *ctText* to execute a SQL statement. If the SQL statement includes parameters, you can set their properties using the *Parameters* property editor of the ADOCommand component. Although it makes no sense to use the ADOCommand component to retrieve a dataset from a table, query, or stored procedure, you can do it. The ADOCommand's *Execute* method returns the recordset generated by the command, if any. You can assign the returned recordset to the *RecordSet* property of an ADODataSet to view the records.

Cursor Types

If you're accustomed to working with the BDE dataset components, there are a number of things you'll find different when you use ADO. One of the most striking is the choice of four different cursor types, which you can set using the *CursorType* property of ADODataSet. The first is *ctStatic*, which provides a static dataset that you cannot edit, and that will not show any changes made by other users. A static cursor behaves like the result set from a BDE Query component with its *RequestLive* property set to False.

Choosing *ctOpenForwardOnly* provides a cursor that is identical to a static cursor, except that you can only move forward through the dataset. A forward-only cursor is very efficient and is ideal for generating reports. Setting the *CursorType* to *ctDynamic* provides a cursor that allows you to navigate both forward and backward, as well as see all additions, deletions, and changes made by other users. The *ctKeySet* cursor type is identical to *ctDynamic* except that you can't see records added by other users.

ADO also provides a *CursorLocation* property with two possible values: *clUseClient* and *clUseServer*. Client cursors are somewhat similar to data provided to a MIDAS ClientDataSet, in that all the data is downloaded to the client immediately. For a large dataset, this can impose a significant penalty in time and memory usage. However, client cursors are almost always updateable, support bookmarks, and allow scrolling in both directions. This may not always be true with server cursors. The features available with server cursors will depend on the database and the OLE DB provider you're using.

Transaction Isolation Levels

ADO supports the ANSI SQL-92 standard transaction isolation levels, which are slightly different than those supported by the Delphi Database component's *TransIsolation* property. ADO supports the following four isolation levels:

- **Read Uncommitted.** Read Uncommitted is also called Dirty Read or Browse isolation. At this level of isolation, a transaction can see uncommitted changes made by other transactions.
- **Read Committed.** A transaction at this level cannot see uncommitted changes made by other transactions, but can see committed changes. This means that reading the same record twice may give two different values because the record could have been changed by another transaction that has committed. If a query is re-executed within the transaction, it can also return new records that have been added by another committed transaction that it did not see the first time the query ran.
- **Repeatable Read.** A Repeatable Read transaction will not see any changes made by other transactions to records it has read, even if the other transactions have committed. However, if a query is re-executed within the transaction, it will see new records added by other committed transactions.
- **Serializable.** This isolation level requires that all concurrent transactions interact in ways that produce the same result as though the transactions executed sequentially. A transaction at this level will not see either changed or newly inserted records from other committed transactions.

Of course, the isolation level that you actually get when you choose one of these options depends on the isolation levels that the database you're using supports.

Conclusion

ADO support is the single most important feature for database application developers in Delphi 5. As Microsoft builds ADO into its next generation of operating systems, you'll no longer have to install additional software with your application to access databases. Perhaps more important is the range of data that ADO will provide access to in the future. Looking beyond relational databases, ADO will provide access to e-mail system message stores, the file system on your hard disk, and any other data store in a Microsoft product.

With the full power of Microsoft behind it, ADO will certainly be adopted by other vendors with products that store data. Finally, ADO relieves Borland of the burden of writing drivers. That is a bigger benefit to you than to Borland because it means you'll get better drivers faster, as new versions of data storage products ship. Best of all, because ADO drivers for Access, SQL Server, and Oracle are already available, and because ADO includes an ODBC provider, you can start using it right now. ∆

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

By David Wolfe

Custom Message Handlers

Transforming Windows Messages into VCL Events

Delphi simplifies the process of handling Windows messages through the vehicle of events. For example, getting a button located on an application's main form to respond to a Windows user input message such as WM_LBUTTONDOWN is as simple as selecting the button by clicking on it, double-clicking the *OnClick* event on the Object Inspector's Event page, and entering the appropriate response code:

```
procedure TForm.ButtonClick(Sender: TObject);
begin
  MessageDlg('Delphi is cool!',
             mtInformation, [mbOk], 0);
end;
```

Using events is much easier than writing the gigantic, window procedure **case** statements that traditional Windows programming requires. However, there are times when an application must respond to a user action or system occurrence for which there is no existing event property. This is when understanding the Visual Component Library (VCL) architecture, the Windows messaging system, and their interrelation, becomes important.

```
function WindowProcedure(hWnd: THandle; nMsg: UINT;
  wParam, lParam: Cardinal): Cardinal; export; stdcall;
begin
  Result := 0;
  case nMsg of
    WM_LBUTTONDOWN : // Open a Child Window...
    WM_LBUTTONUP   : // End a Drag Drop Operation...
    WM_MOUSEMOVE   : // Start a Drag Drop Operation...
    WM_SIZE        : // Repaint the Window...
    ...
    WM_QUIT        : MessageDlg('Adios ..', mtInformation,
                                [mbOk], 0);
  else
    DefWindowProc(hWNd, nMsg, wParam, lParam);
  end;
end;
```

Figure 1: A window procedure prototype.

In this article, we'll discus how Windows messages work, and how they're translated into VCL events. Then we'll use what we've learned to develop a custom scroll box component that fires events in response to scrolling occurrences, i.e. WM_VSCROLL and WM_HSCROLL messages.

Windows Application Architecture

Windows is an event-driven, message-based operating system. Messages start as input events (e.g. the user types, moves the mouse, or clicks a button), state changes (e.g. a window resizes or a system font changes), or application broadcasts (e.g. an application sends messages to its own or other application windows). Windows then routes these messages to an application in one of two ways:

- a message can be posted to an application's message queue — a first in/first out structure that temporarily stores messages, or
- a message can be sent directly to an application window for immediate processing.

Messages in the application's message queue are processed by the application's *WinMain* function. The *WinMain* function continually polls the message queue, removing its topmost messages and directing them toward specific windows until a WM_QUIT message is received.

Each window in an application (in Windows, all controls that can receive focus are windows) has an associated window procedure. The window procedure determines how, and to which, messages an application window will respond. Each time a window's class (a structure that describes a window's properties and behavior) is registered through *RegisterClass* or *RegisterClassEx* API calls, the class' window procedure is registered with the Windows kernel. When specific instances of the classes are created through *CreateWindow* or *CreateWindowEx* calls, messages routed to these windows are processed by their associated window procedure.

Messages and the Window Procedure

A window procedure determines a window's behavior. It's traditionally implemented as a large **case** statement with each **case** corresponding to a specific message. The code associated with each **case** is known as a message handler. Message handlers define how a window will respond to different occurrences.

All window procedures must be functions that take four parameters and return a 32-bit signed value; each of the function's four parameters corresponds to a field in a Windows message record. Windows messages are dispatched to a window's window procedure by the *WinMain* procedure's polling of the message queue, or by a direct *SendMessage* call. A typical window procedure might be structured like the function in Figure 1.

```
TMessage = record
  Msg: Cardinal;
  case Integer of
    0: (WParam: Longint;
        LParam: Longint;
        Result: Longint);
    1: (WParamLo: Word;
        WParamHi: Word;
        LParamLo: Word;
        LParamHi: Word;
        ResultLo: Word;
        ResultHi: Word);
  end;

// Easy to understand.
TWMScroll = record
  Msg:        Cardinal;
  ScrollCode: Smallint; { SB_xxxx }
  Pos:        Smallint;
  ScrollBar:  HWND;
  Result:     Longint;
end;
```

Figure 2: A generic message type definition from the Messages unit.

A Windows message is a data record. Significant fields in a message are *nMsg*, which identifies the specific message being sent, and the parameter fields, *wParam* and *lParam*. Some examples of message identifiers are WM_SIZE and WM_LBUTTONUP; they determine which message handler is invoked when a window procedure is called. The "Param" fields contain relevant information specific to a particular message identifier. For example, if the message identifier is WM_LBUTTONDOWN, *wParam* indicates which virtual keys are down, and *lParam* indicates the position of the cursor.

The Messages unit in the \Delphi4\Source\RTL\Win\ directory contains a list of message identifiers and message record types. For each message type, Delphi defines a record type that gives easy-to-understand names to the *wParam* and *lParam* fields. Figure 2 contains a generic, easy-to-understand message type definition from the Messages unit.

We've just covered a lot of ground in a short space. One of Delphi's strengths is that it hides much of the complexity of the Windows messaging system from the application developer using VCL events. Ultimately, however, an understanding of the Windows application architecture is necessary for much serious Windows programming (see Figure 3).

From Messages to Events

The mechanism with which Delphi implements the Windows application architecture is the VCL's message dispatch system.

Delphi classes that correspond to windows, child windows, or common controls descend from the *TWinControl* class. It's important to remember that a Delphi class isn't the same as a window class. Window classes are records that are used as parameters for certain API functions, such as *RegisterClass*. Delphi (or VCL) classes are classes in the object-oriented sense; they are groups of fields, methods, and properties that can be inherited, and they can be polymorphic in their behavior.

TWinControl descendants include *TForm*, *TButton*, *TEdit*, and *TListBox*. In effect, *TWinControl* classes are "wrappers" around Windows common controls. *TWinControl* implements its associated window's window procedure with the *WndProc* method. Any time the window procedure of a *TWinControl*'s associated window is called (through a *SendMessage* call, a *PostMessage* call, or the application picking the topmost message from the message queue), the *WndProc* method is invoked. The mechanism that accomplishes this is complex, but the important thing to understand is that when the window procedure of a window associated

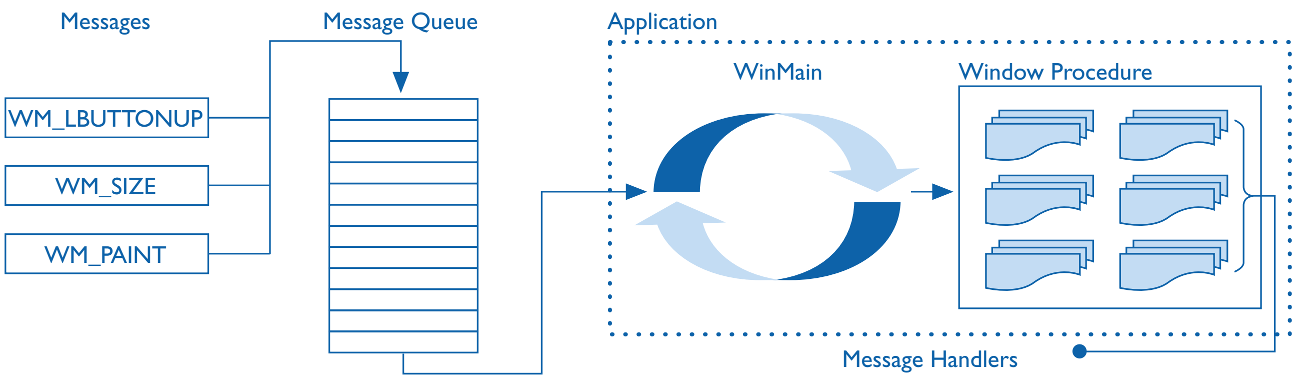


Figure 3: The Windows application architecture.

```
procedure TControl.WndProc(var Message: TMessage);
begin
  if (csDesigning in ComponentState) then
    ...
  if (Message.Msg >= WM_KEYFIRST) and
     (Message.Msg <= WM_KEYLAST) then
    ...
  if (Message.Msg >= WM_MOUSEFIRST) and
     (Message.Msg <= WM_MOUSELAST) then
    ...
  Dispatch(Message);
end;

// TWinControl (override)
procedure TWinControl.WndProc(var Message: TMessage);
var
  Form: TCustomForm;
begin
  case Message.Msg of
    WM_SETFOCUS: ...
    WM_KILLFOCUS: ...
    WM_NCHITTEST: ...
    WM_MOUSEFIRST..WM_MOUSELAST: ...
    WM_KEYFIRST..WM_KEYLAST: ...
    WM_CANCELMODE: ...
  end;
  inherited WndProc(Message);
end;
```

Figure 4: *WndProc* implementations in *TControl*.

with a *TWinControl* is called, the *TWinControl*'s *WndProc* method is fired.

The *WndProc* method is first defined in the *TControl* class, from which *TWinControl* is a descendant. Figure 4 provides a skeleton listing of *WndProc*'s implementation in the *TControl*, and the *TWinControl* class definitions. For more specific information, see the Controls unit in the \Delphi4\Source\VCL directory.

The significance of the *TWinControl WndProc* method skeleton is that it calls its ancestor class' (*TControl*) *WndProc* method, which in turn calls the *TObject.Dispatch* method. *Dispatch* is the key player in the VCL's message dispatch system; it determines whether a message is in the list of message handlers for a given object, or any of its ancestors. If so, *Dispatch* calls the message handler associated with the *Msg* field of its *Message* parameter. If not, it calls the *DefaultHandler* method for the given object.

Message Handlers

A message handler is a method of a *TControl*, or one of its descendants, which is defined by including a message directive in the method's declaration and is fired in response to a specific type of Windows message:

```
procedure WMLButtonDown(var Message: TWMLButtonDown);
  message WM_LBUTTONDOWN;
procedure WMSize(var Message: TWMSize); message WM_SIZE;
procedure WMMove(var Message: TWMMove); message WM_MOVE;
```

For virtually every Windows message, there is a corresponding message-handling method in the VCL. Delphi message handlers constitute the first of three levels of message processing that occur after a Windows message has been picked up by the *WndProc* method. Message handlers are typically named after the message they're designed to "handle," for instance, *WMLButtonDown*.

```
// Level 1
procedure TControl.WMLButtonDown(
  var Message: TWMLButtonDown);
begin
  SendCancelMode(Self);
  inherited;
  if csCaptureMouse in ControlStyle then
    MouseCapture := True;
  if csClickEvents in ControlStyle then
    Include(FControlState, csClicked);
  DoMouseDown(Message, mbLeft, []);
end;

// Level 2
procedure TControl.DoMouseDown(var Message: TWMMouse;
  Button: TMouseButton; Shift: TShiftState);
begin
  if not (csNoStdEvents in ControlStyle) then
    with Message do
      MouseDown(Button, KeysToShiftState(Keys) + Shift,
                XPos, YPos);
end;

// Level 3
procedure TControl.MouseDown(Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Assigned(FOnMouseDown) then
    FOnMouseDown(Self, Button, Shift, X, Y)
end;
```

Figure 5: Procedure calls resulting in the *OnMouseDown* event being fired.

Message handlers ultimately call a second-level, "Do" method, e.g. *DoMouseDown*. "Do" methods handle special case processing and then call a third-level, virtual method, of which *MouseDown* is a good example. Third-level handlers fire events. Figure 5 illustrates the cascade of procedure calls that results in the *OnMouseDown* event being fired.

Events

In Figure 5, we see the following code:

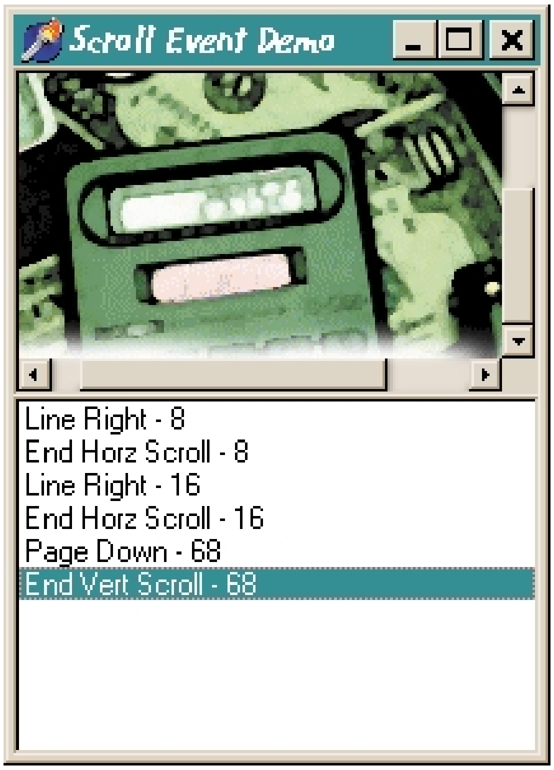
```
if Assigned(FOnMouseDown) then
  FOnMouseDown(Self, Button, Shift, X, Y)
```

In the following example, *FOnMouseDown* points to the code entered in the following block (generated by double-clicking the *OnMouseDown* row on the Events page of the Object Inspector when *Form1* is the "inspected" object):

```
procedure TForm1.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  MessageDlg('Form1 clicked.', mtInformation, [mbOk], 0);
end;
```

The association between the *FOnMouseDown* method variable and the *OnMouseDown* property occurs because events are properties. *FOnMouseDown* is a private field of the *TControl* class, and it stores the *OnMouseDown* method. If code is entered in the *OnMouseDown* event property at design time, then *FOnMouseDown* points to the entry point of that code block.

From an elementary standpoint, events are a way of linking an occurrence to a block of Object Pascal code. From our perspec-



Line Right - 8
End Horz Scroll - 8
Line Right - 16
End Horz Scroll - 16
Page Down - 68
End Vert Scroll - 68

Figure 6: A demonstration of the new scroll box in use.

tive, events are method pointers that are indirectly invoked in the course of handling a Windows message. Whichever perspective one takes, the beauty of Delphi events is that they make responding to Windows messages (e.g. occurrences) an extraordinarily easy and modular task.

In fact, for better or worse, they eliminate the functional need for understanding the messaging basis of Windows application architecture — that is, until it becomes necessary to respond to a message for which there is no associated event. Let's consider one of these situations.

A ScrollBox with *OnScroll* Events

The *TScrollbox* component makes it easy to design applications that require scrolling functionality. Simply place one on a form and insert the necessary controls. There are many times when it's important to know whether the user is scrolling the scrollbox. For example, when vertically scrolling a document in Microsoft Word, a small hint window appears displaying the page of the document corresponding to the current position of the vertical scrollbar. Unfortunately, duplicating this type of functionality using a *TScrollbox* is impossible because it lacks any *OnScroll* events. To rectify this situation, we must create a new component, a *TScrollbox* descendant.

Deriving *TNewScrollBox*

The complete code for this component is available in Listing One (on page 14). (It's also available for download; see end of article for details.) Although going through the process of component creation step by step is beyond the scope of this article, let's consider some of *TNewScrollBox*'s important features:

- We've declared two message handlers: *WMVScroll* and *WMHScroll*. Both have been declared with the **message** directive, and both are called when the *TNewScrollBox* receives a WM_SCROLL message.

```
procedure TForm1.NewScrollBox1HorizontalScroll(
  Sender: TObject; Pos: Smallint;
  EventType: THScrollEventType);
var
  Tmp : string;
begin
  with ListBox1, Items do begin
    case EventType of
      hsLineLeft    : Tmp := 'Line Left';
      hsLineRight   : Tmp := 'Line Right';
      hsPageLeft    : Tmp := 'Page Left';
      hsPageRight   : Tmp := 'Page Right';
      hsThumbPos    : Tmp := 'Horz ThumbPos';
      hsThumbTrack  : Tmp := 'Horz ThumbTrack';
      hsLeft        : Tmp := 'Left';
      hsRight       : Tmp := 'Right';
      hsEndScroll   : Tmp := 'End Horz Scroll';
    end;
    Add(Tmp + ' - ' + IntToStr(Pos));
    ItemIndex := Count-1;
  end;
end;

procedure TForm1.NewScrollBox1VerticalScroll(
  Sender: TObject; Pos: Smallint;
  EventType: TVScrollEventType);
var
  Tmp : string;
begin
  with ListBox1, Items do begin
    case EventType of
      vsLineUp      : Tmp := 'Line Up';
      vsLineDown    : Tmp := 'Line Down';
      vsPageUp      : Tmp := 'Page Up';
      vsPageDown    : Tmp := 'Page Down';
      vsThumbPos    : Tmp := 'Vert ThumbPos';
      vsThumbTrack  : Tmp := 'Vert ThumbTrack';
      vsTop         : Tmp := 'Top';
      vsBottom      : Tmp := 'Bottom';
      vsEndScroll   : Tmp := 'End Vert Scroll';
    end;
    Add(Tmp + ' - ' + IntToStr(Pos));
    ItemIndex := Count-1;
  end;
end;
```

Figure 7: Handling the *OnHorizontalScroll* and *OnVerticalScroll* events.

- We've declared two new event types: *TVScrollEvent* and *THScrollEvent* (one each for Vertical and Horizontal), and two private method variables of that type: *FOnVScroll* and *FOnHScroll*.
- We've declared two method properties (i.e. events), *OnVerticalScroll* and *OnHorizontalScroll*, both of which publish the private method variables *FOnVScroll and FOnHScroll*.
- When *WMVScroll* or *WMHScroll* is called, it calls the *VScroll* or *HScroll* method, which, in turn, fire the *OnVerticalScroll* or *OnHorizontalScroll* event.

The *WMVScroll* and *WMHScroll* message handlers test for a special condition relating to the use of the thumb bar. If the event type is either *ThumbPos* or *ThumbTrack*, the incoming message record holds the current position of the thumb bar while tracking (in the *Pos* field of a message). If the event type is anything else, this *Pos* field isn't used. To allow our event to recognize the position while the thumb bar is being dragged, we test to see if the event type is either of the "Thumb" types. If it is, we send the message's *Pos* value to our event handler. Otherwise, we send the actual current position of the scrollbar.

To demonstrate the use of this new scroll box, I've put together a quick demonstration (see Figure 6). It's a NewScrollBox component and a ListBox component placed on a form.

Inside the NewScrollBox component I put a *TImage* with a graphic larger than the dimensions of the ScrollBox (so it would show the scrollbars). Then I provided code in the *OnVerticalScroll* and *OnHorizontalScroll* handlers, as shown in Figure 7.

Conclusion

Understanding how Windows messages are transformed into VCL events becomes important when it's necessary to extend the VCL. Even if component development isn't the primary order of business, understanding Windows application architecture, and how it relates to the VCL, is a requirement for maturing both as a Delphi developer specifically, and a Windows application developer generally. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\OCT\DI9910DW.

David Wolfe is a programmer/analyst who designs and implements software for the entertainment industry at Media-Services, Inc. When he is not architecting in the UML or programming in Delphi, C++, or Smalltalk, he can be found spending time with his fiancée, Molly, and watching old Kurosawa movies. You can contact David at DavidW@Media-Services.Com.

Begin Listing One — NewScrollBox.pas

```
unit NewScrollBox;

interface

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

type
  TVScrollEventType = (vsLineUp, vsLineDown, vsPageUp,
    vsPageDown, vsThumbPos, vsThumbTrack, vsTop, vsBottom,
    vsEndScroll);
  THScrollEventType = (hsLineLeft, hsLineRight, hsPageLeft,
    hsPageRight, hsThumbPos, hsThumbTrack, hsLeft, hsRight,
    hsEndScroll);
  TVScrollEvent = procedure(Sender: TObject; Pos: SmallInt;
    EventType: TVScrollEventType) of object;
  THScrollEvent = procedure(Sender: TObject; Pos: SmallInt;
    EventType: THScrollEventType) of object;

  TNewScrollBox = class(TScrollBox)
  private
    FOnVScroll: TVScrollEvent;
    FOnHScroll: THScrollEvent;
    procedure WMVScroll(var Message: TWMScroll);
      message WM_VScroll;
    procedure WMHScroll(var Message: TWMScroll);
      message WM_HScroll;
  protected
    procedure VScroll(Pos: Integer;
      EventType: TVScrollEventType); virtual;
    procedure HScroll(Pos: Integer;
      EventType: THScrollEventType); virtual;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property OnVerticalScroll: TVScrollEvent
      read FOnVScroll write FOnVScroll;
    property OnHorizontalScroll: THScrollEvent
      read FOnHScroll write FOnHScroll;
  end;

procedure Register;

implementation

procedure TNewScrollBox.VScroll(Pos: Integer;
  EventType: TVScrollEventType);
begin
  if assigned(FOnVScroll) then
    FOnVScroll(Self, Pos, EventType);
end;

procedure TNewScrollBox.HScroll(Pos: Integer;
  EventType: THScrollEventType);
begin
  if assigned(FOnHScroll) then
    FOnHScroll(Self, Pos, EventType);
end;

procedure TNewScrollBox.WMVScroll(var Message: TWMScroll);
var
  EventType : TVScrollEventType;
begin
  inherited;
  EventType := TVScrollEventType(Message.ScrollCode);
  if EventType in [vsThumbPos, vsThumbTrack] then
    VScroll(Message.Pos, EventType)
  else
    VScroll(VertScrollBar.Position, EventType)
end;

procedure TNewScrollBox.WMHScroll(var Message: TWMScroll);
var
  EventType : THScrollEventType;
begin
  inherited;
  EventType := THScrollEventType(Message.ScrollCode);
  if EventType in [hsThumbPos, hsThumbTrack] then
    HScroll(Message.Pos, EventType)
  else
    HScroll(HorzScrollBar.Position, EventType)
end;

constructor TNewScrollBox.Create(AOwner: TComponent);
begin
  inherited;
  FOnVScroll := nil;
  FOnHScroll := nil;
end;

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

End Listing One

By Eddie Shipman

Agents of Instruction

Using Microsoft Agent with Delphi

If you've seen the television commercial with a Microsoft executive talking with a parrot on his computer, you may have wondered when that technology was going to be available to use in your own programs. With Microsoft's Agent SDK version 2.0, the time is now. Located at http://msdn.microsoft.com/msagent/default.asp, the SDK is a freely downloadable and royalty-free interface that allows you to create interactive agents. In this article, we'll focus on installing the Agent controls and characters, and how to interact with the control using Delphi.

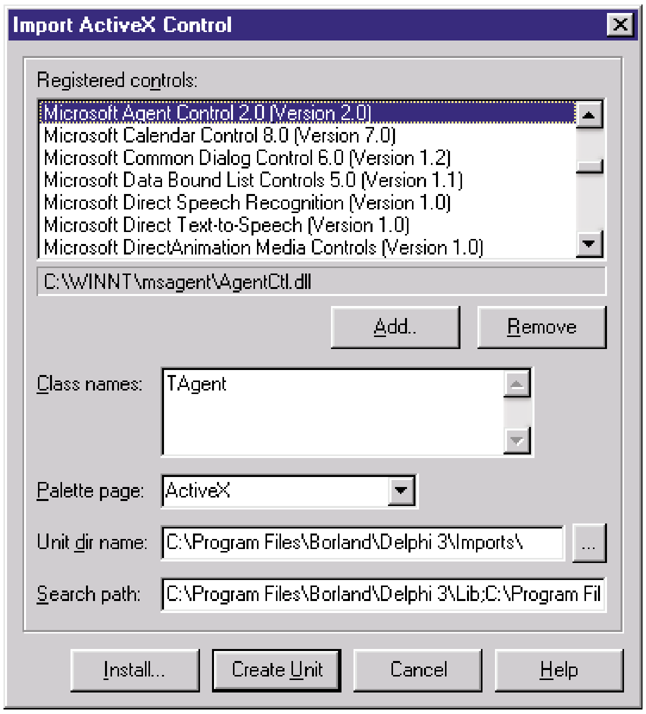
Getting Started

To begin, you must download the Microsoft Agent core components and at least one character file. If you want to have character speech and voice recognition, a text-to-speech engine and a speech recognition engine must be downloaded, as well. You can also download the speech control panel and LISET, Microsoft's Linguistic Information Sound Information Tool, which allows you to modify the settings for the speech engines you install.

Microsoft supplies four characters on their site: Genie, Merlin, Peedy the parrot, and Robby the robot. If you plan on distributing these agents with your application, you must also get a copy of the Microsoft Agent Licensing and Distribution agreement. Before moving on, make a note of these files, which are among those available for download on the Microsoft site: Agent core components; characters; speech recognition engine; L&H TruVoice TTS engine; Visual Basic, Visual C++, and Java code samples; the Microsoft character editor; and LISET.

To successfully install Agent on a target system, you must ensure that the system has a recent version of the

Microsoft Visual C++ run time (Msvcrt.dll), Microsoft registration tool (Regsvr32.dll), and Microsoft COM DLLs. It's best to require that Microsoft Internet Explorer 3.02 (or later) be installed to ensure that the necessary components are on the target system. When Agent is installed, it's considered a system component and can only be uninstalled by re-installing the operating system. It will be included with Windows 2000.

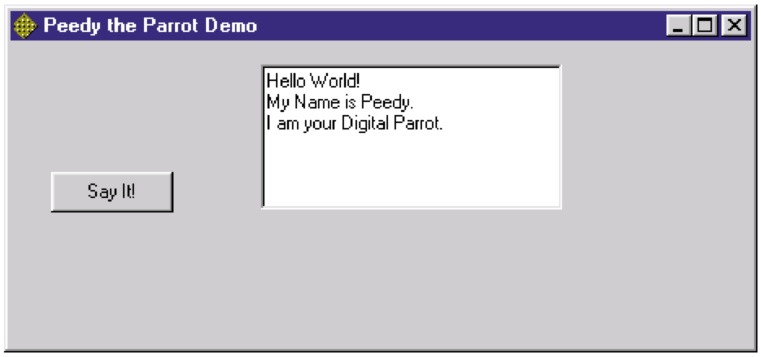


```
Import ActiveX Control

Registered controls:
Microsoft Agent Control 2.0 (Version 2.0)
Microsoft Calendar Control 8.0 (Version 7.0)
Microsoft Common Dialog Control 6.0 (Version 1.2)
Microsoft Data Bound List Controls 5.0 (Version 1.1)
Microsoft Direct Speech Recognition (Version 1.0)
Microsoft Direct Text-to-Speech (Version 1.0)
Microsoft DirectAnimation Media Controls (Version 1.0)
C:\WINNT\msagent\AgentCtl.dll

Class names: TAgent
Palette page: ActiveX
Unit dir name: C:\Program Files\Borland\Delphi 3\Imports\
Search path: C:\Program Files\Borland\Delphi 3\Lib;C:\Program Fil
```

Figure 1: The Import ActiveX Control dialog box.



```
Peedy the Parrot Demo

Hello World!
My Name is Peedy.
I am your Digital Parrot.
```

Figure 2: Creating the new Peedy project.

Programming Agents

The first thing we need to do to use the Microsoft Agent control in Delphi is to install the ActiveX component. Installation of the Agent Core Components will auto-register the Agent ActiveX Control. Then we have to install the character file(s). I used Delphi 3 for the examples in this article. And I decided to use Peedy because of the commercial.

From the **Component** menu, select **Import ActiveX Control**. Scroll down until you see **Microsoft Agent Control 2.0**. Click the **Install** button (see Figure 1). The control will be installed on the ActiveX page of the Component palette.

Now that we have the Agent installed, create a new project and place an Agent component, a Memo component, and a Button component on the form (see Figure 2).

In the *OnFormShow* event handler, write some code to create and activate the Agent control:

```
var
  Form1 : TForm1;
  Peedy : IAgentCtlCharacterEx;

implementation

{$R *.DFM}

procedure TForm1.FormShow(Sender: TObject);
begin
  Agent1.Connected := True;
end;
```

Note that if you don't set the Agent's *Connected* property to True, you'll receive a run-time error indicating that Agent was unable to start.

In the button's *OnClick* event, place the code shown in Figure 3.

Compile and run the project, write some lines into the memo field, click the **Say It!** button, and Peedy will fly onto the screen and speak the words you've typed.

But, where did he go? The character was unloaded before he got a chance to speak, because the speak and play requests are played asynchronously while your code continues to execute. One thing we need to be aware of is the user pressing the **Say It!** button again while the character is loaded and running its

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i : Integer;
begin
  Agent1.Characters.Load('Peedy','Peedy.acs');
  Peedy := Agent1.Characters.Character('Peedy');
  Peedy.LanguageID := $409;
  Peedy.Show(False);
  for i := 0 to Memo1.Lines.Count-1 do
    Peedy.Speak(Memo1.Lines[i],'');
  Peedy.Speak('I''m Done!', '');
  Peedy.Hide(False);
  Agent1.Characters.UnLoad('Peedy');
end;
```

Figure 3: Code for the *OnClick* event.

```
// Turn off automatic error raising;
// be sure to turn it back on.
Agent1.RaiseRequestErrors := False;
// Let the load method raise the error
// through the status property.
LoadRequest := Agent1.Characters.Load('Peedy','peedy.acs');
if LoadRequest.Status <> 0 then begin
  // Unsuccessful Load.
  MessageDlg('The character "Peedy" is already loaded.' +
    Chr(13) + 'Check for previous Load method calls.' +
    Chr(13) + 'The LoadRequest Status was:' +
    IntToStr(LoadRequest.Status), mtError, [mbOK], 0);
  Exit;
end;
```

Figure 4: *LoadRequest* object to catch an error.

animations. Using a **try..except** block, we can catch the error if the user clicks the button again:

```
try
  Agent1.Characters.Load('Peedy','peedy.acs');
except
  on E: Exception do begin
    MessageDlg(E.Message, mtError, [mbOK], 0);
    Exit;
  end;
end;
```

Another way to catch the error is to use a *LoadRequest* object (see Figure 4).

Synchronizing Character Animations

We must track the requests to the control in order to synchronize the control with the code. Now, create an *OnRequestComplete* procedure in the Object Inspector and move the unload code into it:

```
procedure TForm1.Agent1RequestComplete(Sender: TObject;
  Request: IDispatch);
begin
  if Request = MyRequest then
    Agent1.Characters.Unload(AgentName);
end;
```

The *MyRequest* object is defined as:

```
MyRequest: IAgentCtlRequest;
```

Now you'll be able to see Peedy on the screen.

There is another way to control the synchronization of the animations: through the use of bookmarks. If we were to place another call to speak after looping through the memo and format it like this:

```
Peedy.Speak('I'm Done!\mrk=1\', '');
```

we could use the control's *OnBookMark* method to control what was going on:

```
procedure TForm1.Agent1Bookmark(Sender: TObject;
  BookmarkID: Integer);
begin
  if BookMarkID = 1 then
    ShowMessage('I'm Done');
end;
```

This will show a message box after he is finished speaking.

```
Request1 := Peedy.Speak(Knock knock','');
Genie.Wait(Request1);
Request2 := Genie.Speak('Who's there?','');
Peedy.Wait(Request2);
Request1 := Peedy.Speak('Orange.','');
Genie.Wait(Request1);
Request2 := Genie.Speak('Orange who?','');
Peedy.Wait(Request2);
Request1 := Peedy.Speak(
  'Orange you glad you didn't tell this awful joke?','');
Genie.Wait(Request1);
```

Figure 5: Genie and Peedy interacting.

There are other tags that can be used to modify speech output in the *Speak* method. You can use these tags to change the characteristics of the output expression of the character. From the Microsoft documentation, Speech tags follow this format:

- All tags begin and end with a backslash character (\).
- The single backslash character is not enabled within a tag. To include a backslash character in a text parameter of a tag, use a double backslash (\\).
- Tags are case-insensitive, e.g. \pit\ is the same as \PIT\.
- Tags are whitespace-dependent, e.g. \Rst\ is not the same as \ Rst \.

```
procedure TForm1.Agent1RequestComplete(Sender: TObject;
  Request: IDispatch);
var
  Response : string;
  i        : Integer;
  p        : TPoint;
begin
  if Request = HideRequest1 then begin
    Agent1.Characters.UnLoad('Genie');
    WaitRequest1 := Peedy.Speak(
      'Please enter your name in the box below.', '');
  end;
  if Request = HideRequest2 then begin
    Agent1.Characters.UnLoad('Peedy');
    Request := nil;
    Close;
  end;
  if Request = WaitRequest1 then begin
    Peedy.Play('GestureDown');
    Response := InputBox('Input Name',
      'Please enter your name.', 'You don't have a name?');
    Peedy.Play('RestPose');
    if Pos('You don't have a name?', Response) = 0 then
      WaitRequest2 := Peedy.Speak(
        'Thank you, your name is ' + Response + '.','')
    else
      begin
        Peedy.Play('Uncertain');
        WaitRequest2 :=
          Peedy.Speak('Hey, don't you have a name?','')
      end;
  end;
  if Request = WaitRequest2 then begin
    Peedy.Speak(
      'Select Yes or No to make me click the other button',
      '');
      WaitRequest3 := Peedy.Play('GestureDown');
  end;
  if Request = WaitRequest3 then begin
    P := ScreenToClient(Point(Peedy.Top+Peedy.Height+50,
                        Peedy.Left+Peedy.Width+40));
    i := MessageDlgPos(
      'Do you want me to click the button?',
      mtConfirmation, [mbYes, mbNo], 0, P.X, P.Y);
    if i = mrYes then
      Button2.Click;
    else
      WaitRequest4 := Peedy.Speak(
        'You selected "No" so I did not click ' +
        'the button.','');
  end;
  if Request = WaitRequest4 then begin
    Peedy.Play('RestPose');
    Peedy.Play('Wave');
    Peedy.Speak('Goodbye, thanks for trying Microsoft ' +
           'Agent Technologies','');
    HideRequest2 := Peedy.Hide(False);
  end;
end;
```

Figure 6: The *OnRequestComplete* event.

For example:

```
// Peedy Whispers.
Peedy.Speak('\Chr="Whisper"\I'm Whispering', '');
```

There are many other tags that can be used to modify the characteristics of the speech output. Please reference the document, SpeechOutputTags.doc, for more information. It's included in the Agent documentation, available at http://msdn.microsoft.com/workshop/imedia/agent/agentdevdl.asp#allzip.

Moving the Character

So you want to move the character around — maybe place him on top of your form? This is done with the *MoveTo* method. Place the following code before *Peedy.Show*:

```
Peedy.MoveTo(Form1.Left, Form1.Top-Peedy.Height, 0);
```

It appears that placement isn't as straightforward as it may seem. You can get the dimensions of the character with the *Height* and *Width* properties, but it doesn't seem to place the character directly on the top-left edge of the form. There seems to be a transparent border around some of the characters. You can also have the character gesture at any portion of the screen. This code makes him gesture at the *Button1* component. *GestureAt* takes screen coordinates, so be sure to convert them with *ClientToScreen*:

```
p := ClientToScreen(Point(Memo1.Left, Memo1.Top));
Peedy.GestureAt(p.x, p.Y);
```

There are many animations included with these characters. All of the animations for the Microsoft-supplied characters are documented in the Agent documentation (file name, alldocs.zip) on Microsoft's Agent site. To get the animations to play, use the *Play* method:

```
Peedy.Play('Greet');  // "Peedy" Bows.
```

Interacting with Other Agents

Agent allows you to load more than one character. These loaded characters can interact with each other, although you're the one controlling the interaction. For instance, you can have more than one character loaded with each character doing animations and responding to the other, as well as to the user. You use *Request* objects along with the *Wait* method to accomplish this interaction.

For now, we'll load Peedy and Genie and have Genie wait and respond to something Peedy says (see Figure 5). Any animations pending in Peedy's queue before Genie's *Wait* method will play without interruption.

Interacting with Users

There are several ways for the character to interact with the user: input balloons, input boxes, message dialog boxes, and speech recognition. The speech recognition uses a *Command* object for the program, or Agent to react to the user interaction. There is also a third-party extension to Agent called HTMLBalloons written by Costas Andriotis. It allows the use of input balloons similar to the ones used with Office Assistants, and can be obtained at http://www.agentstation.com.

Let's demonstrate how to use the input boxes and message dialog boxes to interact with the user. The *OnRequestComplete* event is used in this example because of the asynchronous mode of the actions of the agent characters (see Figure 6).

The most common way to interact with a user is through speech recognition. If you haven't seen the commercial, Peedy asks the executive if he wants to schedule a meeting in his Outlook calendar. The executive gives the command for Peedy: "Sure." Peedy replies, "How about Saturday, 2:00 PM?" The executive says, "OK." Peedy then flies away into the background. This is done through the use of the *Commands* object of the character.

First, let's define a couple of simple commands. One will tell Peedy to calculate; when the Agent control processes the command, he'll play the Process animation in which he pulls out a calculator and starts calculating. The commands will also appear in the popup menu that appears when you right-click on the character or his tray icon. Be aware that you cannot use V or H as accelerators for your command captions, because these are accelerators for the default menu items when you have a command defined.

The format to add a *Command* object to the character's commands collection is:

```
Commands.Add(const Name: WideString;
  Caption, Voice, Enabled, Visible: OleVariant);
```

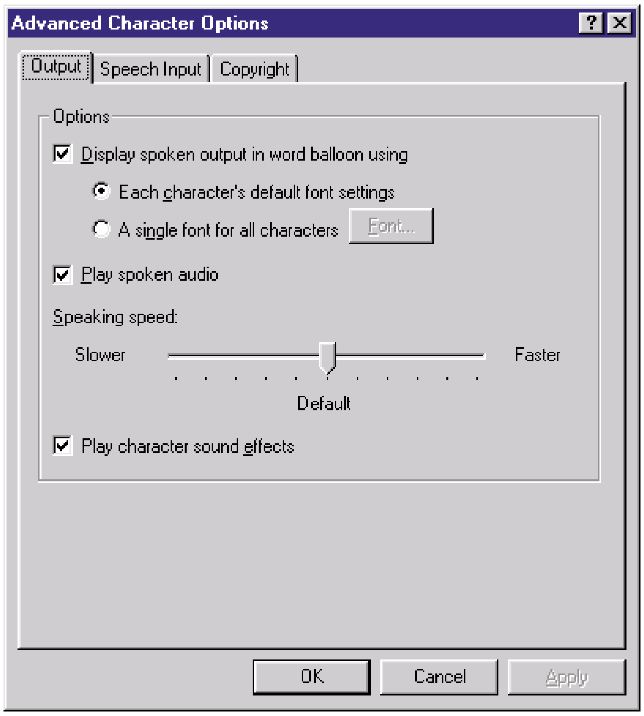
In our case, to define a command to calculate, we use this code:

```
Peedy.Commands.Add('Calculate', '&Calculate', 'calculate',
                   True, True);
```

We also added a command to bring up the character's property sheet:

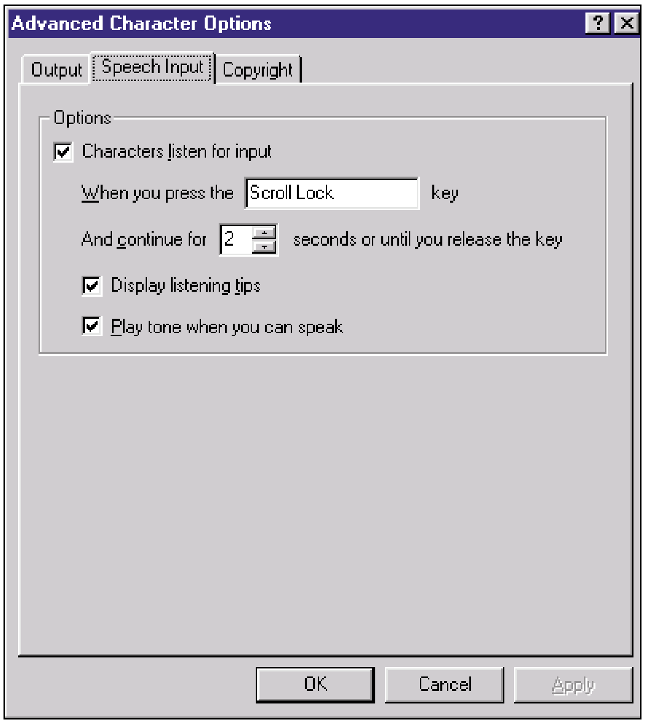
```
Peedy.Commands.Add('Property Sheet',
  '&Show Property Sheet', 'property', True, True);
```

This property sheet allows us to change the key to press when we want the character to begin listening to commands, and other things pertaining to our character (see Figures 7 and 8).



[✓] Display spoken output in word balloon using
(●) Each character's default font settings
( ) A single font for all characters
[✓] Play spoken audio
Speaking speed:
[✓] Play character sound effects

Figure 7: Output options in the Advanced Character Options dialog box.


Advanced Character Options

Output | Speech Input | Copyright

Options

☑ Characters listen for input

When you press the Scroll Lock key

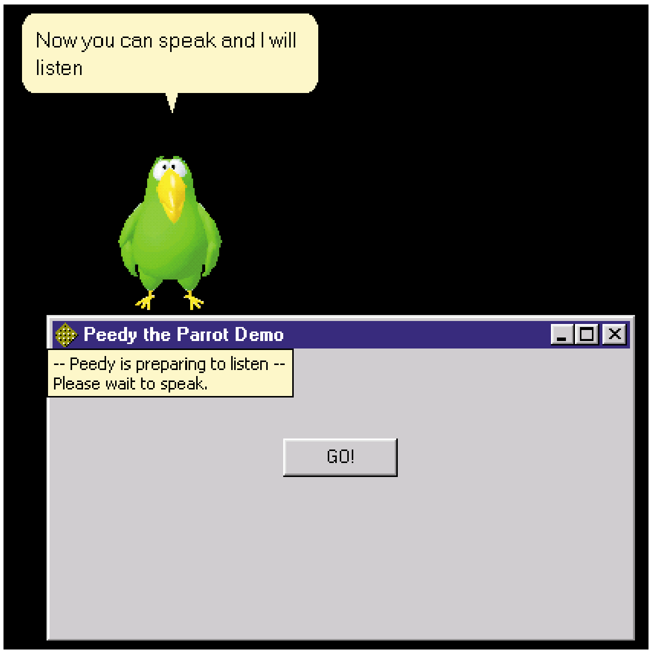
And continue for 2 seconds or until you release the key

☑ Display listening tips

☑ Play tone when you can speak

OK | Cancel | Apply

Figure 8: Determining speech characteristics in the Advanced Character Options dialog box.


Now you can speak and I will listen

Peedy the Parrot Demo

-- Peedy is preparing to listen --
Please wait to speak.

Figure 9: A tip box signifies Peedy is in listening mode.

When selecting these commands from the pop-up menu, it's processed exactly as it would be if the character were in listening mode. For the character to begin listening to your commands, we must place him in listening mode. This is done with the *Listen* method:

```
Peedy.Listen(True);
```

Of course, you'll need a microphone attached to your sound card for the speech recognition engine to listen to your commands. In the following code, we'll add the commands, and then play an animation that will trigger the *Listen1* request when it's done:

```
{ Both commands have alternate voice commands. The
  character will perform the same operation when hearing
  either of the commands; e.g. (calculate|figure) if you
  say "calculate" or "figure" it means the same thing to
  the character. }
Peedy.Commands.Add('Calculate', 'C&alculate',
                   '(calculate|figure)',True,True);
Peedy.Commands.Add('Goodbye', '&Goodbye',
                   '(goodbye | bye)',True, True);
Peedy.Speak('Now you can speak and I will listen','');
Listen1 := Peedy.Play('Alert');
```

In the *Listen1* request, we will set Peedy to listening mode:

```
if Request = Listen1 then
  Peedy.Listen(True);
```

You'll see a listening tip box come up when the character is in listening mode (see Figure 9).

In the *Agent1Command* procedure, we'll listen for the specific voice commands (see Figure 10).

```
procedure TForm1.Agent1Command(Sender: TObject;
  UserInput: IDispatch);
var
  VoiceStr : string;
begin
  // Get the actual word spoken in the string, VoiceStr.
  VoiceStr := IAgentCtlUserInput(UserInput).Voice;
  if IAgentCtlUserInput(UserInput).Name = 'Calculate' then
    begin
      Listen2 := Peedy.Speak('I heard you say ' +
                             VoiceStr,'');
      Peedy.Listen(True);
    end;
  if IAgentCtlUserInput(UserInput).Name = 'Goodbye' then
    begin
      Peedy.Speak ('I heard you say '+ VoiceStr,'');
      Peedy.Play ('Wave');
      Peedy.Speak('Goodbye','');
      HideRequest2 := Peedy.Hide(False);
    end;
end;
```

Figure 10: Code for listening for specific voice commands.

Conclusion

Microsoft Agent technology is an exciting new option in user interface design. With these new techniques, you can build totally interactive CBTs, Web sites, demonstrations, and other programs. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\OCT\DI9910ES.

Eddie is a Senior Delphi Developer at Academic Software, Inc. in Austin, TX. He lives in Round Rock with his wife Vickie and pet Pomeranian, Tasha. He can be reached via e-mail at shipman@inetport.com.

By Keith Wood

Open Tools at Work

A Delphi Add-in to Automate Module Versioning

When working on a moderate to large project, it quickly becomes apparent that we need to keep track of the different versions of the modules being developed. For example, we must ensure the latest version is the one released to the testers, and ultimately the user. And during testing, we need to know which version of each module is being used so we can trace any errors.

Delphi provides built-in hooks for third-party version-control systems bundled with its Client/Server editions. But, what if we want something simpler, like a label on each form that we update each time we save the unit? We can create a Delphi add-in that automates this task for us, relieving us of the burden of remembering to perform this change each time.

Note: The code demonstrated in this article was written using Delphi 2 and 3. However, it also compiles correctly in Delphi 4. Its Delphi 5 compatibility was unknown at press time.

Versioning Add-in

The add-in we create integrates with the Delphi IDE through interactions with Delphi's *ToolServices* object, which is available to experts and add-ins. From this object, we gain access to Delphi's menu structure, we are told about modules that are opened and closed, and we can alter properties of components on forms.

At the simplest level, we want the add-in to update the text of a label each time the module is saved. To do this, we need to keep track of which modules are open and when they are being saved. We can then access the label component, determine its current value, and increment it before writing it back.

To make the add-in more useful, let's expand the requirements to update a specified property of a named component, rather than just hard-coding *lblVersion.Caption*. Let's also allow the version label to be either an incrementing

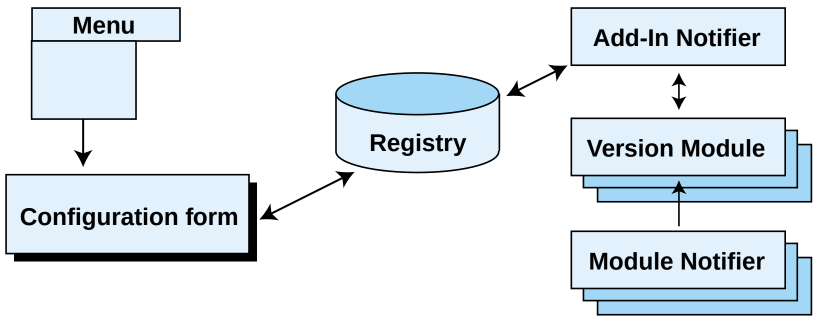


Figure 1: Structure of the versioning add-in.

```
{ Create a project notifier and install it.
  Also add menu item. }
constructor TVersionAddIn.Create;
var
  imnMainMenu: TIMainMenuIntf;
  imiMenuItem, imiParentMenu: TIMenuItemIntf;
begin
  inherited Create;
  { Add menu item for versioning options. }
  imnMainMenu := ToolServices.GetMainMenu;
  if imnMainMenu <> nil then
    try
      imiMenuItem :=
        imnMainMenu.FindMenuItem('ProjectOptionsItem');
      if imiMenuItem <> nil then
        try
          imiParentMenu := imiMenuItem.GetParent;
          if imiParentMenu <> nil then
            try
              imiOptions := imiParentMenu.InsertItem(
                imiMenuItem.GetIndex + 1,
                '&Versioning Options...',
                'KWood_VersionAddIn', '', 0, 0, 0,
                [mfVisible], imiOptionsClick);
            finally
              imiParentMenu.Free;
            end;
        finally
          imiMenuItem.Free;
        end;
    finally
      imnMainMenu.Free;
    end;
  { Install notifier. }
  ainVersion := TVersionAddInNotifier.Create(imiOptions);
  if not ToolServices.AddNotifier(ainVersion) then
    MessageDlg('Couldn''t load'#13#10 + GetName,
               mtError, [mbOK], 0);
end;
```

Figure 2: Constructing the add-in interfaces.

number, or the date and time the module was saved. Finally, let's have the user confirm each change to the version, or have the add-in automatically update it whenever the module is saved. Finally, it would be nice if the whole system could be customized at the project level; we may want to have different schemes in different projects, with the ability to turn it all off for certain applications.

Creating the Add-in

The add-in is created by subclassing the *TIExpert* class. We override the basic information providing methods of this class to supply the name of the new add-in, its unique identifying string, and its style. The latter is set to *esAddIn*, indicating that we will control all interaction with the add-in. Normally, Delphi would set up an interface to the expert for us, based on its style (through the Object Repository for form and project experts, and through the **Help** menu for standard experts).

Our add-in has a series of interacting objects behind the scenes to implement all the functionality required. These are shown in Figure 1. The first entry point is via the menu to the configuration form. This allows the user to customize the add-in for each project, with the options being saved in the Windows registry.

The second entry point is through the add-in notifier that tracks changes to the current project within Delphi. It reads the settings for the project from the registry and manages a series of version module and module notifier objects that monitor the activities of individual modules within the project. We set up both of these entry points in the **constructor** of the add-in, as demonstrated in Figure 2.

For the menu item, we use the *GetMainMenu* function of the *ToolServices* object to provide access to Delphi's main menu. We then locate the position in the menu where we want our item to appear. In our case, this is after the **Project | Options** item.

The *FindMenuItem* function gets us to the required menu item, but only if we know the name of the menu item. Some of the standard menu item names are shown in Figure 3. From here, we retrieve a reference to the **Project** menu itself with the *GetParent* function. Next, we insert our new menu item immediately after the "Versioning Options" item with the *InsertItem* function. Finally, don't forget to free all the interfaces that we've acquired.

We need to keep a reference to this new menu item, so it can be freed when the expert is destroyed. The result of the user selecting this new item is a call to the *imiOptionsClick* method, which we passed as one of the parameters to its construction. This procedure simply creates the configuration form, then displays it and processes any changes made through it.

The add-in notifier, which we create and load, handles the other side of our add-in's interactions. This notifier keeps track of the options set by the user for the current project and is informed by Delphi when projects and/or modules are opened or closed.

Configuration

Selecting the **Versioning Options** menu item invokes the configuration form (see Figure 4). This allows the user to customize how the add-in functions with their project.

The options exist at two levels: a global default and project-specific values. When a project is opened, the add-in tries to find its particular settings. If these are not available, the defaults are used instead. The options are saved in the registry under the key \Software\KWood\Version Add-In. Beneath this are keys for the global values (Defaults) and each project by name. To toggle between the two levels, we use a checkbox at the top of the form.

At each level, we can specify the name of the component and its property to be updated. This property must be able to accept the version label to be maintained. Two types of labels can be supplied: an integer or a date and time. Further checkboxes allow the user to enable or disable the actions of the add-in for this project as a whole and to automatically update the version, or to ask the user to confirm the change for each module.

When the OK button is pressed, any changes are written back to the registry, and are updated in the add-in notifier that is tracking the current project.

Project Tracking

The add-in notifier is informed when a project is opened or closed. Upon opening a new project, the notifier retrieves its name and uses this to load the versioning options applicable to it from the registry. It also enables the Options menu item with the following statement:

```
imiOptions.SetFlags([mfEnabled],
                    [mfEnabled]);
```

This method allows us to alter the menu flags specified in the first parameter and to set them to the values from the second parameter. All other flags retain their original values.

Upon closing a project, the notifier frees all the notifiers it has set up to monitor the individual modules, clears the project name, and disables the menu item.

When a file is opened or added to the project, the add-in notifier creates an object and associated module notifier to track the new module's activities. These are added to a list of open modules. Of course, as the file is closed or removed from the project, these items are destroyed.

Module Tracking

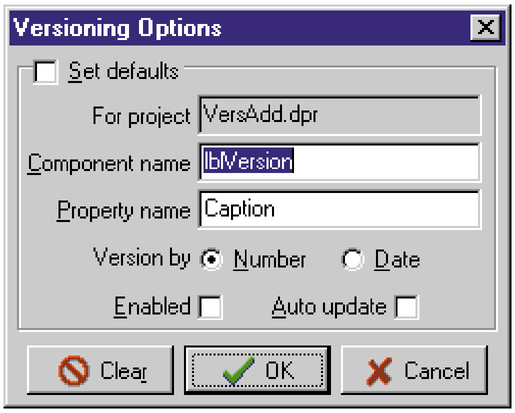
For each module opened, we want to be informed when the user saves it. At this point, we can update the version label as requested. To do this, we use the *TIModuleNotifier* class and derive a new subclass from it to implement our actions. Once created, the notifier must be associated with the particular module to monitor. To achieve this, we call the module interface's *AddNotifier* method. Before the notifier is destroyed, we must remove it from that interface with the *RemoveNotifier* method.

The *TIModuleNotifier* class declares two abstract methods. We're interested in the *Notify* method, but we still need to supply code for the *ComponentRenamed* method, even though it does nothing:

Caption	VCL Menu Object	Version
&File	*FileMenu*	
&New...	*FileNewItem*	
New Applica&tion	*FileNewApplicationItem*	
New &Form	*FileNewFormItem*	
New Data &Module	*FileNewDataModuleItem*	
&Print...	*FilePrintItem*	
E&xit	*FileExitItem*	
&Edit	*EditMenu*	
&Undelete	*EditUndoItem*	
&Redo	*EditRedoItem*	
Cu&t	*EditCutItem*	
&Copy	*EditCopyItem*	
&Paste	*EditPasteItem*	
&Delete	*EditDeleteItem*	
Se&lect All	*EditSelectAll*	
&Search	*SearchMenu*	
&Find...	*SearchFindItem*	
&Replace...	*SearchReplaceItem*	
&View	*ViewsMenu*	
&Project Manager	*ViewPrjMgrItem*	
Pro&ject Source	*ViewPrjSourceItem*	
&Project	*ProjectMenu*	
&Options...	*ProjectOptionsItem*	
&Run	*RunMenu*	
&Run	*RunRunItem*	
&Component	*ComponentMenu*	
&New...	*ComponentNewItem*	2
&New Component...	*ComponentNewItem*	3
&Install...	*ComponentInstallItem*	2
&Install Component...	*AddtoPackage1*	3
Create Component &Template...	*ComponentInstallCompositeItem*	3
Open &Library...	*ComponentOpenLibraryItem*	2
&Rebuild Library	*ComponentRebuildItem*	2
Install &Packages...	*InstallPackagesItem*	3
Configure &Palette...	*ComponentPaletteItem*	
&Database	*DatabaseMenu*	
&Explore	*Borland_DbExplorerMenu*	
&SQL Monitor	*Borland_SQLMonitorMenu*	
&Form Expert...	*Borland_FormExpertMenu*	2
&Form Wizard...	*Borland_FormExpertMenu*	3
&Tools	*ToolsMenu*	
&Options...	*ToolsOptionsItem*	2
Environment &Options...	*ToolsOptionsItem*	3
&Repository...	*ToolsGalleryItem*	
&Tools...	*ToolsToolsItem*	2
Configure &Tools...	*ToolsToolsItem*	3
&Help	*HelpMenu*	
&Help Topics	*HelpContentsItem*	2
&Contents	*HelpContentsItem*	3
&Topic Search	*HelpTopicSearchItem*	2
&Keyword Search	*HelpTopicSearchItem*	3
&How to Use Help	*HelpUsingHelpItem*	
&Windows API	*HelpAPIItem*	
&About...	*HelpAboutItem*	

Figure 3: Selected Delphi menu items.

```
{ It does nothing, but we must override this abstract method. }
procedure TVersionModuleNotifier.ComponentRenamed(
  ComponentHandle: Pointer; const OldName, NewName: string);
begin
  { Do nothing. }
end;
```



Set defaults [ ]
For project: VersAdd.dpr
Component name: lblVersion
Property name: Caption
Version by: [✓ Number] Date
Enabled [ ] Auto update [ ]

Figure 4: Configuring the versioning expert.

The *Notify* method can tell us many things about its attached module. These include signals when the module is renamed or deleted, when its code editor or form designer are selected, when the code or form are modified, and when the module is being saved. See the *EditIntf* unit in the Source\ToolsAPI directory for the appropriate values to check.

Of course, the only notification we want to know about is when the module is about to be saved, i.e. *ncBeforeSave*. We must act on the "before save" so that any changes we make are written to the disk. First, we check whether versioning has been enabled for this project. If it is hasn't, we do nothing.

Otherwise, we retrieve an interface to the form itself using the *GetFormInterface* method of the module interface object. Then, we search for the component specified by the user for this project with the *FindComponent* function. If we find such a component, we then attempt to update the named property, as shown in Listing One (on page 24).

When versioning by a number, we must first read the current value from the component before incrementing it and writing it out again. We can check the type of the property with the *GetPropTypeByName* method to ensure that we can set the version label. Then we can obtain the current value through the *GetPropValueByName* method. If it's a string or variant property, we convert the value to a number, defaulting to zero if unsuccessful. Confirmation of the change is requested if the user hasn't selected automatic updates during configuration. Finally, we increment the value and set it with the *SetPropByName* method, converting back to a string if necessary.

Versioning by dates is a little simpler. We still retrieve the current date and time label for use in the confirmation dialog box, but we don't have to worry about different formats for the property.

After updating the new version label, Delphi continues and saves the module and new label to disk. Our versioning task has been completed with minimal interaction from the user. Appropriate error handling and checking in the code means that the lack of a component with the specified name doesn't disrupt the program flow. Thus, not all modules within a project need to be versioned.

The resulting version label could be invisible by default and would only be shown when a certain action occurred, such as pressing a particular key combination or a mouse click in a particular spot. Alternately, a conditional compile section, such as the following:

```
{ $IFDEF VERSIONING }
lblVersion.Visible := True;
{ $ENDIF }
```

can be set up, based on a flag defined at compilation time. Then, for a testing version of the application, under the **Project | Options** menu item, on the Directories/Conditionals tab, you would add VERSIONING to the list of Conditional defines before doing a complete recompile with the **Project | Build All** menu item. For a user release, you would remove this value from the list before compiling.

Installation

Installing the versioning add-in into Delphi 3 involves the following steps:
1) Close Delphi.
2) Copy the versioning add-in DLL to an appropriate directory, e.g. \Delphi\Bin.
3) Start the Registry Editor by selecting **Run** from the Windows 95 **Start** menu, then entering regedit.
4) Under the key HKEY_CURRENT_USER\Software\Borland\Delphi\3.0\Experts, add a new string value by right-clicking and selecting **New | String Value**. Enter its name as `VersAdd`.
5) Modify the value for this entry by right-clicking it and selecting **Modify**. Enter the full path and file name for the DLL from step 2.
6) Close the Registry Editor and restart Delphi. The **Versioning Options** menu item should appear on the **Project** menu to show the add-in was installed correctly.

The same steps apply for Delphi 2, except that the registry key is 2.0 instead of 3.0. Thereafter, you can configure the add-in to work with the component and property you prefer. Add one of these components to your form, rename it appropriately, and save the unit. The property should be updated with the new version for this unit.

Conclusion

Although it's nowhere near a full version control system, the add-in described here shows how we can automate tasks within the Delphi IDE through the use of its Open Tools API. For this utility, we dealt with menu interfaces, with project and module notifiers, with module and form interfaces, and finally with component interfaces. All this allows us to track what is happening inside Delphi and to intervene at the appropriate points for our purposes. Our add-in maintains a designated property of a specified component with minimal action from ourselves. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\OCT\DI9910KW.

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. He has enjoyed exploring Delphi since it first appeared, and works with it occasionally. You can reach him via e-mail at kwood@ccsc.com.

Begin Listing One — Updating the version label on a module save

```
{ When notified of file being saved, look for specified
  component. If found ask whether to increment and save. }
procedure TVersionModuleNotifier.Notify(
  NotifyCode: TNotifyCode);
var
  cpiComponent: TIComponentInterface;

  { Update component's property as a number. }
  procedure UpdateAsNumber;
  var
    sVersion: string;
    iVersion, iVersInc: Integer;
    iPropType: TPropertyType;
  begin
    { Check property type. }
    iPropType := cpiComponent.GetPropTypeByName(
                   vmdModule.Notifier.PropertyName);
    if not (iPropType in [ptInteger, ptFloat, ptString,
             ptLString, ptLWString, ptVariant]) then
      begin
        MessageDlg('Invalid property type for ' +
          vmdModule.Notifier.ComponentName + '.' +
          vmdModule.Notifier.PropertyName + ' -'#13#10 +
          'must be string, numeric or variant for ' +
          'versioning', mtError, [mbOK], 0);
        Exit;
      end;
    { Extract current value. }
    try
      if iPropType in [ptInteger, ptFloat] then
        cpiComponent.GetPropValueByName(
          vmdModule.Notifier.PropertyName, iVersion)
      else
        begin
          cpiComponent.GetPropValueByName(
            vmdModule.Notifier.PropertyName, sVersion);
          iVersion := StrToInt(sVersion);
        end;
    except
      iVersion := 0;
    end;
    iVersInc := iVersion + 1;
    { Update to new value? }
    if vmdModule.Notifier.AutoUpdate or
       (MessageDlg('Update version of file ' +
       vmdModule.FileName + #13#10 + 'from ' +
       IntToStr(iVersion) + ' to ' + IntToStr(iVersInc) +
       '?', mtConfirmation, [mbYes, mbNo], 0) = mrYes) then
      begin
        if iPropType in [ptInteger, ptFloat] then
          cpiComponent.SetPropByName(
            vmdModule.Notifier.PropertyName, iVersInc)
        else
          begin
            sVersion := IntToStr(iVersInc);
            cpiComponent.SetPropByName(
              vmdModule.Notifier.PropertyName, sVersion);
          end;
      end;
  end;

  { Update component's property as date and time. }
  procedure UpdateAsTime;
  var
    sVersion, sVersInc: string;
    dVersion: TDateTime;
    iPropType: TPropertyType;
  begin
    { Check property type. }
    iPropType := cpiComponent.GetPropTypeByName(
                   vmdModule.Notifier.PropertyName);
    if not (iPropType in [ptString, ptLString,
                          ptLWString, ptVariant]) then
      begin
        MessageDlg('Invalid property type for ' +
          vmdModule.Notifier.ComponentName + '.' +
          vmdModule.Notifier.PropertyName + ' -'#13#10 +
          'must be string or variant for versioning',
          mtError, [mbOK], 0);
        Exit;
      end;
    { Extract current value. }
    try
      cpiComponent.GetPropValueByName(
        vmdModule.Notifier.PropertyName, sVersion);
      dVersion := StrToDateTime(sVersion);
    except
      sVersion := '<none>';
    end;

    sVersInc := DateTimeToStr(Now);
    { Update to new value? }
    if vmdModule.Notifier.AutoUpdate or
       (MessageDlg('Update version of file ' +
         vmdModule.FileName + #13#10 + 'from ' + sVersion +
         ' to ' + sVersInc + '?', mtConfirmation,
         [mbYes, mbNo], 0) = mrYes) then
      cpiComponent.SetPropByName(
        vmdModule.Notifier.PropertyName, sVersInc);
  end;

begin
  { Check for file being saved. }
  if (NotifyCode = ncBeforeSave) and
     vmdModule.Notifier.Enabled then
    { Get interface to this module. }
    with vmdModule.ModuleInterface do
      { Get interface to the form. }
      with GetFormInterface do
        try
          { Look for component. }
          cpiComponent := FindComponent(
            vmdModule.Notifier.ComponentName);
          try
            { And ask to update if found. }
            if Assigned(cpiComponent) then
              case vmdModule.Notifier.VersionBy of
                iVersionByNumber: UpdateAsNumber;
                iVersionByDate:   UpdateAsTime;
              end;
          finally
            cpiComponent.Free;
          end;
        finally
          Free;  { Form interface. }
        end;
end;
```

End Listing One

By Robert Vivrette

CodeRush 4

Everything You Ever Wanted to Add to Delphi

One of the sayings that third-party developers live by is: "There is always room for improvement." This saying is particularly applicable to those third-party vendors who make add-ons for the Delphi development environment. We've seen a lot of products along these lines: add-on component libraries, integrated wizards for creating custom projects, and — in the case of this review — products that extend the capabilities of the Delphi Code editor.

Although Delphi's Code editor has some wonderful capabilities, developers often formulate their own list of features they wish it had. I've used a number of products that try to address this issue. Most of them however, take the approach of creating a really fancy, full-function editing system, having it run alongside Delphi, and trying (with various degrees of success) to keep your Delphi code and forms synchronized with what you're doing in this separate editor.

CodeRush takes a different approach. Because Borland published the interface to their editor in the Open Tools API, CodeRush is able to enhance the Delphi Code editor itself, rather than running as a separate editor. The advantages of this are obvious: There's no need to concern yourself about switching between Delphi and an external editor, and no need to worry about whether code or design-time elements are synchronizing properly.

I had an opportunity to see some of the new capabilities of this latest version of CodeRush at the Borland developers conference in Philadelphia this last July. Mark Miller (the architect of CodeRush and owner of Eagle Software) is passionate about his work, and it shows in the products he creates.

So let's take a look at some of the features of CodeRush.

Smart Keyboard Templates

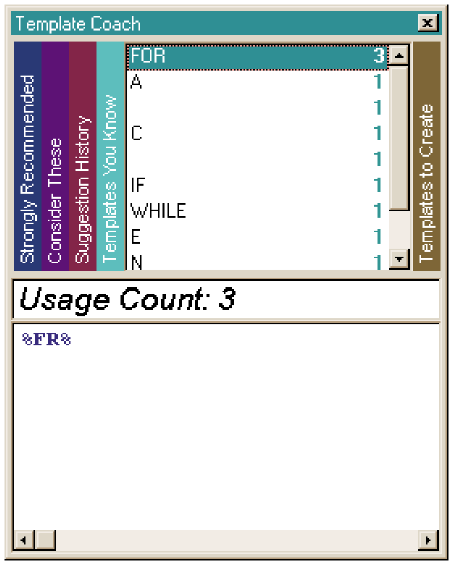
The Delphi Code editor already has a rudimentary template facility. It has the ability to take short (usually two- or three-character) key combinations and expand them into larger sections of code. For example, if you were to type `ifeb` and then hit Delphi's template hotkey (Ctrl J), it will replace these four characters with a full **if..then..else** block with **begin..end** blocks.

CodeRush takes this idea to the limit. Whereas Delphi templates are fairly static, those used in CodeRush have all sorts of embedded smarts. For example, one of the options in CodeRush is to define your coding style, i.e. how you indent your **begin..end** blocks under an **if** statement, etc. The templates are then aware of your preferences and adapt accordingly. Another example is that templates know where they are; for example, expanding the **procedure** block template generates different behavior in the **interface** section, where it might appear as `procedure GetALine;`, as opposed to the **implementation** section, where it might appear as `procedure TForm1.GetALine;`. You can define your own templates or use any of the more than 1,000 included.

One thing that has always disturbed me with code templates, however, is that it's virtually impossible to remember them all. To be honest, in previous versions of CodeRush, I turned them off because they would expand character combinations that I really didn't know were defined as templates. It just became too much of an irritant as I was constantly undoing unwanted template expansions. I almost felt like deleting every template and defining them one at a time so my brain could absorb the acronyms for them.

This latest version of CodeRush addresses this issue quite nicely with the Template Coach (see Figure 1). This window is dockable, like all the other windows in Delphi (and CodeRush for that matter). Its purpose is to give you tips concerning your use of the CodeRush templates. For example, it highlights the templates you already know (and are making good use of), as well as spotting opportunities for you to create your own templates. For example, if you're typing the same piece of code repeatedly, it adds a recommendation to make a template from that piece of code. Also, if you type blindly along, ignoring many of the templates available, the Template Coach will give you tips on templates it recommends you use.

The Template Coach is a tremendous idea, and it opens up the power of code templates to those — like me — who have been intimidated or annoyed by them in the past.

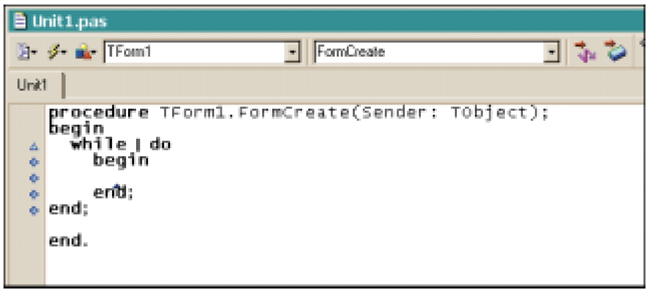


```
FOR    3
A      1
       1
C      1
       1
IF     1
WHILE  1
E      1
N      1
```

Usage Count: 3

```
%FR%
```

Figure 1: The Template Coach.

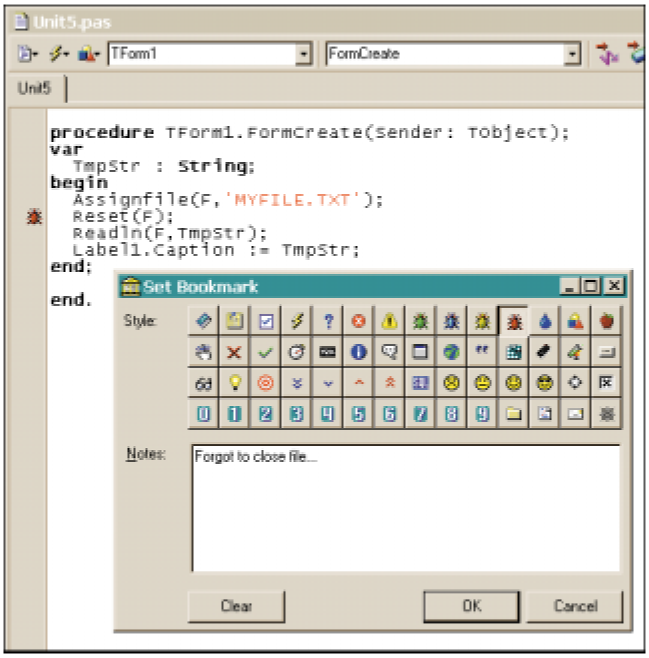


```
procedure TForm1.FormCreate(Sender: TObject);
begin
  while | do
    begin

    end;
end;

end.
```

Figure 2: Using the visual stack-based markers in the code templates.



```
procedure TForm1.FormCreate(Sender: TObject);
var
  TmpStr : String;
begin
  AssignFile(F,'MYFILE.TXT');
  Reset(F);
  ReadLn(F,TmpStr);
  Label1.Caption := TmpStr;
end;

end.
```

Notes: Forgot to close file...

Figure 3: There are many glyphs to use as bookmarks.

Visual Stack-based Markers

Programmers rarely sit in one piece of code for long; they hop over to this function or that one, open up a different source code file, browse through for a routine they want to use, etc. Stack-based markers are used for exactly this purpose. Although Delphi has a rudimentary form of these (the little back and forward buttons on the top-right of the editor window), CodeRush's visual markers are a real gem.

These markers can be used in two ways. First, they can be used as a kind of breadcrumb-dropping mechanism. If ever you're going to move away from the spot where you're working, you can drop one of these markers and go wherever you like. When you're done and want to return to the marker, simply hit Esc, and — *voilà!* — back you go. As indicated by their name, these markers are stack-based, meaning you can place a bunch of these markers in sequence, then jump back in sequence by repeatedly hitting Esc.

A second way these markers are used is within the code templates. When a template is defined, you can indicate locations of where markers should be placed. For example, in Figure 2, I typed `wh`, followed by the space bar. This triggered one of the **while** code templates, which inserted a **while** statement with a **begin..end** block. In addition to placing the cursor in the next logical spot (the condition portion of the **while** statement), it has also placed a marker within the **begin..end** block. Now, as soon as you're done defining the condition, you simply hit Esc and you're transported down to within the **begin..end** block. Ready to type!

Persistent Bookmarks

This is the feature of CodeRush that I personally find the handiest. We're all familiar with the way Delphi shows little icons in the gutter on the left side of the editor window. Delphi typically only uses this for showing breakpoints and the current execution point of the application being debugged.

CodeRush's persistent bookmarks significantly extend this capability by allowing you to place various bookmarks in the gutter. You can select from a wide variety of glyphs to use as bookmarks (see Figure 3). Interestingly, most of the glyphs have unique properties. For example, there are button glyphs that can be assigned an action when they're clicked and e-mail glyphs that will launch your default e-mail client and address a message to a specific person. There is even a checkbox glyph you can set to, obviously, check or uncheck with an action associated with each. As an example, you could put checkbox bookmarks next to conditional defines and have the uncheck option clear the conditional define and the checked option enable it. These are just a few of the hundreds of behaviors that bookmarks can perform.

To make managing your bookmarks easier, there is also a dockable window that shows all bookmarks that have been placed in the currently open project (see Figure 4). By clicking on any of the bookmarks in this window, you are immediately taken to the location of that bookmark. And bookmarks are persistent, so even after you close your files or project, the bookmarks will be there when you come back.

Instant Declaration and Initialization

I don't know about you, but when I write a procedure or function, I generally write all the code first and declare the local variables last. This is where CodeRush's Instant Declaration and Initialization feature really shines. All you need to do is visit the variables one at a time and hit Ctrl Alt V on each. When

CodeRush sees this key combination, it looks at the surrounding context of the use of a variable and proposes a variable declaration for it. For example, when I positioned the cursor on the a in the **for** loop in Figure 5 and hit Ctrl Alt V, it added the local declaration specifying *a* as an Integer. When I did the same for *TmpStr*, it correctly detected that it should be a string and added its declaration appropriately.

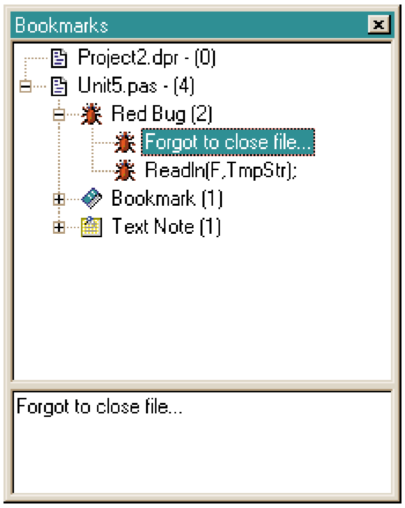


Figure 4: Clicking on a bookmark immediately takes you to the location of that bookmark.

Going one step further, you can also trigger this kind of declaration on string literals. In this case, it created a local string constant named *sReadingLineNumber*. The result is shown in Figure 6.

Embedded Diagramming

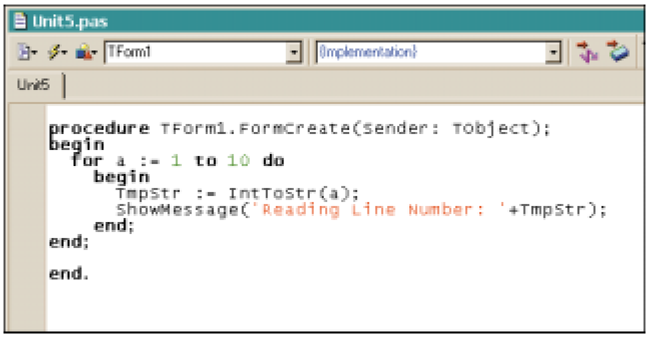
How many times have you wished you could insert a diagram in your code explaining the relationship of some object or class in relation to another? Look no further; the Professional version of CodeRush allows you to create and edit diagrams inside your source code (see Figure 7). Although the Standard edition of CodeRush only allows you to view and print these diagrams, the Professional version gives you a full suite of tools for making any kind of diagram you can imagine. Also included is the ability to link diagram portions together and provide hotlinks between elements of the diagram and associated pieces of source code.

INFORMANT FACT FILE

From the documentation, and online dynamic tutorial, to the many details, CodeRush is solid, well thought out, and well designed. If you feel the Delphi IDE is perfect and needs no improvement, you won't see any need for CodeRush's capabilities. If you're like other Delphi developers who would love it if the Delphi Code editor could just do "this one thing," then each of CodeRush's many enhancements will be welcome.

Eagle Software
12021 Wilshire Blvd., Suite 655
Los Angeles, CA 90025

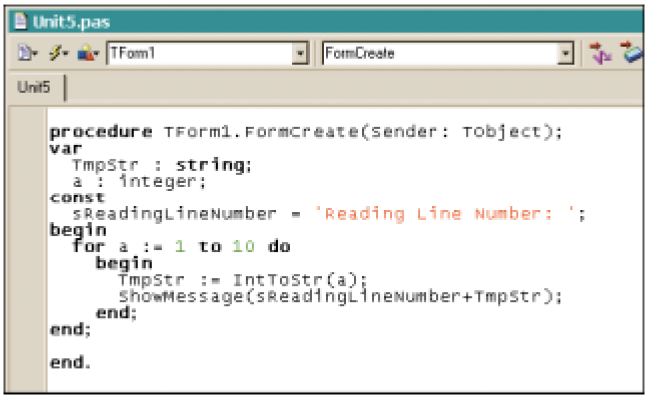
Phone: (310) 441-4096
Web Site: http://www.eagle-software.com
Price: US$199 (Standard version); US$369 (Professional version); site licenses are available.



```
procedure TForm1.FormCreate(Sender: TObject);
begin
  for a := 1 to 10 do
    begin
      TmpStr := IntToStr(a);
      ShowMessage('Reading Line Number: '+TmpStr);
    end;
end;

end.
```

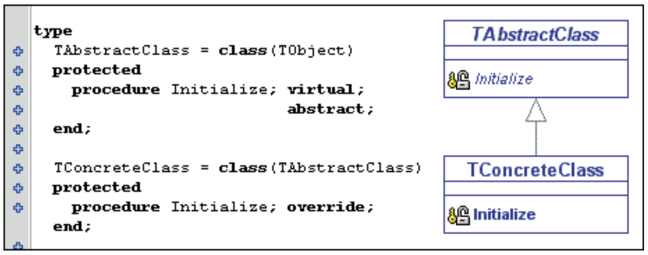
Figure 5: Before invoking CodeRush's instant variable declaration feature.



```
procedure TForm1.FormCreate(Sender: TObject);
var
  TmpStr : string;
  a : integer;
const
  sReadingLineNumber = 'Reading Line Number: ';
begin
  for a := 1 to 10 do
    begin
      TmpStr := IntToStr(a);
      ShowMessage(sReadingLineNumber+TmpStr);
    end;
end;

end.
```

Figure 6: Variables automatically defined.



```
type
  TAbstractClass = class(TObject)
  protected
    procedure Initialize; virtual;
                          abstract;
  end;

  TConcreteClass = class(TAbstractClass)
  protected
    procedure Initialize; override;
  end;
```

Figure 7: You can create and edit diagrams inside your source code.

The Tip of the Iceberg

I've really only touched on a few of the capabilities that CodeRush adds to your Delphi development environment. It would take about three or four times the space I had available in this review to really do this product justice. One huge untouched area is the fact that CodeRush is highly extensible. There is a wide range of (mostly free) plug-ins for CodeRush that add to its features, and the ability to make your own plug-ins is also included. If you want to learn more about the additional capabilities CodeRush provides, I encourage you to check out their excellent Web site at http://www.eagle-software.com.

Conclusion

As you might have guessed, I like CodeRush quite a bit. Interestingly, I was asked to review CodeRush over another individual, because I would be more likely to bring out its failures. I failed at that task; I couldn't find any. The entire product, from the documentation and online dynamic tutorial to the many details, is really wonderfully solid, well thought out, and well designed.

If you feel the Delphi IDE is perfect and needs no improvement, then I guess you won't see any need for CodeRush's capabilities. However, if you're like the 99.9 percent of Delphi developers out there who would love it if the Delphi Code editor could just do "this one thing," then each of CodeRush's many enhancements will be welcome. At US$199 for the Standard version and US$369 for the Professional version (which adds diagram creation and editing), CodeRush is an easy buy, considering the many amazing capabilities it adds. Go get it! Δ

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

By Robert K. Leahey

ModelMaker 5

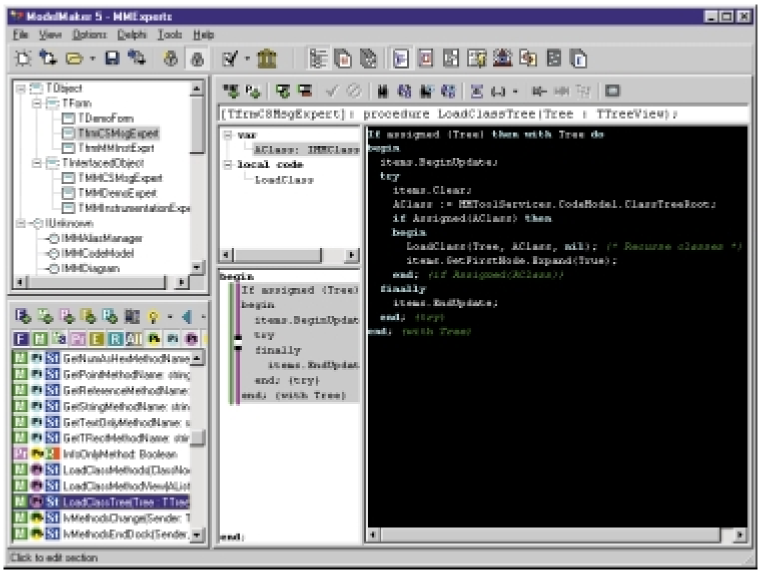
The Ultimate Delphi CASE Tool

I recently had a job interview that, naturally, included a technical evaluation. I had to develop a test project on site, so the manager led me to a PC and turned me loose in Delphi to create my masterpiece. I was only 10 minutes into development when I was shaken with a realization: This machine did not have ModelMaker. I was doomed!

Okay, not really. In fact, I got the job, but it did serve to show me how dependent I had become upon a tool. ModelMaker 5 from ModelMaker is a Delphi CASE tool for analysis and design. However, that's like saying Delphi is a programming environment; it's accurate, but somehow fails to convey the scale of the application.

I happened upon ModelMaker a year ago when I was searching for a UML tool that supported Object Pascal. I found several products, most of which seemed to support Delphi as an afterthought, and were in the US$800+ price range. ModelMaker (then in version 4) distinguished itself by being affordable (US$199) and by being specifically for Delphi. I bought it and began happily using it for class diagrams. Figure 1 shows a screenshot of ModelMaker's main interface.

As time went by, however, ModelMaker began supplanting the other tools I was using for various facets of development:



```
[TfrmCSMsgExpert]: procedure LoadClassTree(Tree : TTreeView);
```

```
var
  AClass: IMMClass
local code
  LoadClass
```

```
begin
  If assigned (Tree)
  begin
    items.BeginUpdat
    try
    finally
      items.EndUpdat
    end; {try}
  end; {with Tree}
end;
```

```
If assigned (Tree) then with Tree do
begin
  items.BeginUpdate;
  try
    items.Clear;
    AClass := MMToolServices.CodeModel.ClassTreeRoot;
    if Assigned(AClass) then
    begin
      LoadClass(Tree, AClass, nil); (* Recurse classes *)
      items.GetFirstNode.Expand(True);
    end; {if Assigned(AClass)}
  finally
    items.EndUpdate;
  end; {try}
end; {with Tree}
```

Figure 1: ModelMaker with Class Tree and Method Implementation view active.

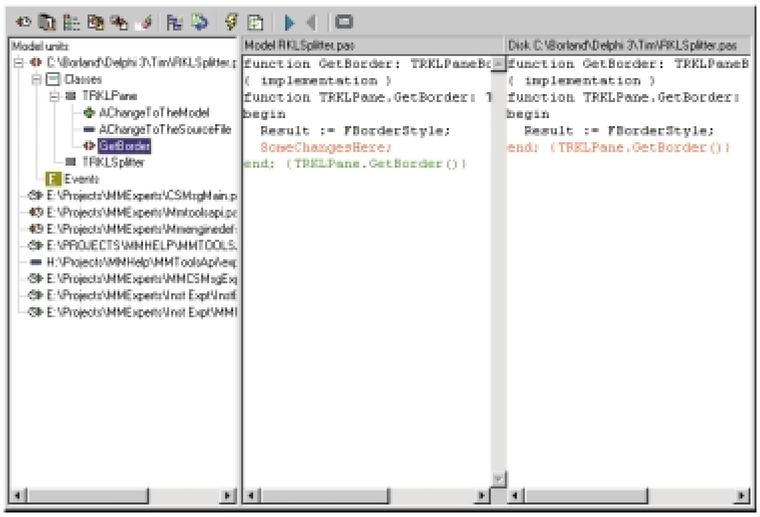
- Like many developers, my code-documentation requirements are high; ModelMaker streamlines the sometimes redundant task of code documentation.
- I was using a help-generation tool to document my custom components. It would parse my code and look for specially formatted comments to include in my Help file. ModelMaker automated entering those formatted comments, then completely replaced my help-generation tool.
- For code differencing, I was using external products, such as PVCS or Visual Source Safe, that would compare the source code as text and point out the differences, which I would then attempt to synchronize in Delphi. ModelMaker offers powerful code comparison options (compare as text, as classes, by time stamp, etc.) and lets me correct differences in the editor.

In addition, ModelMaker reverse-engineers code, handles unit management, integrates class diagrams with code generation, supports design patterns, greatly simplifies class/component creation, enables COM and object-interface support, and, with version 5, offers an Open Tools API similar to Delphi's that allows for development of third-party experts to extend ModelMaker's functionality.

Reverse Engineering

ModelMaker offers excellent reverse-engineering capabilities, quickly breaking down a unit into its classes and their members and converting that information into ModelMaker's Active Model format. Immediately after importing a source file, you'll be able to begin editing and regenerating it with ModelMaker's various options having their effects. For instance, at my previous job I would often import a unit of code and regenerate it with my source commenting options set simply to clean it up and get a little documentation into an otherwise sparsely documented legacy unit. In addition to adding the comments, doing this would allow me to set the order in which the method implementations would appear (alphabetically, access methods last, etc.), which is nice for trying to negotiate someone else's code. It's even possible to reverse-engineer your code documentation!

Imagine this scenario: You're brought on to maintain a large project with many units and classes and far too little project documentation to guide you. All too common, eh? Fear not! You start up your trusty ModelMaker, import all the units, and begin building your own project documentation right from the code. You enter ModelMaker's Diagram Editor and, using the Visualization Wizard, effortlessly create several UML-style class diagrams displaying relations. Because ModelMaker also imported what code documentation there was, you use the Documentation Wizard to comment what's left, then hit the Help File Generation Wizard to create a quick Help file for the project's classes. In a short time, you've created an impressive amount of documentation (without actually knowing anything about the project), all from the initial code import.



```
Model units:
C:\Borland\Delphi 3\Tim\RKLSplitter.p
  Classes
    TRKLPane
      AChangeToTheModel
      AChangeToTheSourceFile
      GetBorder
    TRKLSplitter
  Events
E:\Projects\MMExperts\CSMsgMain.p
E:\Projects\MMExperts\Mmtoolsapi.pa
E:\Projects\MMExperts\Mmenginedef
E:\PROJECTS\MMHELP\MMTOOLS.
H:\Projects\MMHelp\MMToolsApi\exp
E:\Projects\MMExperts\MMCSMsgExp
E:\Projects\MMExperts\Inst Expt\Inst
E:\Projects\MMExperts\Inst Expt\MMI
```

```
Model RKLSplitter.pas
function GetBorder: TRKLPaneBo
{ implementation }
function TRKLPane.GetBorder: T
begin
  Result := FBorderStyle;
  SomeChangesHere;
end; {TRKLPane.GetBorder()}
```

```
Disk C:\Borland\Delphi 3\Tim\RKLSplitter.pas
function GetBorder: TRKLPaneB
{ implementation }
function TRKLPane.GetBorder:
begin
  Result := FBorderStyle;
end; {TRKLPane.GetBorder()}
```

Figure 2: ModelMaker's Difference view in Structured Difference mode.

Code Documentation

Code documentation was one of the things that impressed me the most when I first started using ModelMaker. The possibilities here are nearly endless. Because of ModelMaker's macro implementation, it's possible to automate large portions of the task of documentation. User-defined and pre-defined macros can be used with a page full of documentation options to create some heavily customized, yet automated, comments.

Many packages that allow reverse-engineering will ignore your code comments when parsing your code, thus forcing you to somehow re-integrate your documentation. ModelMaker, on the other hand, makes it possible for you to import your code comments with your code. By formatting your comments correctly, ModelMaker will import your documentation and apply it to the appropriate method or object within the active modeling engine. As a package, this is one impressive, well-designed, time-saving set of features.

Help Generation

It's not a WinHelp editor; you're on your own for that. However, ModelMaker will quickly and unobtrusively create a Help project file (.HPJ) and a topic file (.RTF) from your model and your model's documentation that you can then compile. The topic file is created using RTF templates that you can edit to produce the look you want.

For those who are not used to directly editing RTF files to create Help files, this may not be much help (unless you accept the default results, which aren't bad). For those who can wrestle with RTF, this is an excellent feature that can produce good, detailed component (or project) documentation with minimum fuss.

Differencing

So you've been doing all your development in ModelMaker, making changes, generating the code, then compiling in Delphi. Then one morning, you come in and some non-ModelMaker-using Philistine has edited your source by hand. In Delphi, of all things! The source code is now out of synch with the model. Ack! Well, fear not, grasshopper; ModelMaker understands that these things happen. It will help you find the path.

The Differences view, shown in Figure 2, offers a smorgasbord of methods for determining who's up-to-date, including a time-stamp comparison (for one or all units); a structured comparison, which shows differences by class and member; a file comparison, which is a straight text compare; and a class comparison, which will compare two selected classes.

Once the differences have been determined, you have a variety of options for synchronizing the model. You can re-import entire units or individual methods, or you can overwrite the source file from the model.

Class Diagrams

It's ironic. This feature is why I originally purchased ModelMaker, but now that I know more about UML, I can see that ModelMaker is a bit lacking here. Basically, the Diagram Editor in ModelMaker is intended to give a view (or views) into the model. ModelMaker supports classes, interfaces, relationships, and annotation symbols, but if you're looking for a full UML editor, ModelMaker isn't it — yet.

Having spoken to Gerrit Beuze (one of the authors of ModelMaker) about this issue, he agrees and states that over the coming months, improving UML support will be one of their priorities. That's no small thing. The rate at which these guys come out with increased functionality is astounding. Every few weeks, they come out with a new build, and those builds often have more improvements than many products' major releases.

That being said, the Diagram Editor is an excellent view into the model. Some cool features of the Diagram Editor include:

- Visualization Wizard — a tool for adding multiple classes and relationships to a diagram.
- Drag/Drop — is there a class in the model that you want to add? Drag it into the diagram. Associations will be automatically visualized.
- Class editing — selecting a class in the diagram is the same as selecting it in the class tree. Its members are displayed in the Member List, allowing editing of the class from within the diagram. Likewise, if *TMyClass* and *TMyOtherClass* are both in the diagram, and you want *TMyClass* to have a property of type *TMyOtherClass*, use the Add Property tool to drag an association between them. Not only will the relationship be displayed in the diagram, the property will also be automatically added.
- Interface Implementation — to have a class in the diagram implement an interface, use the Add Interface Support tool to

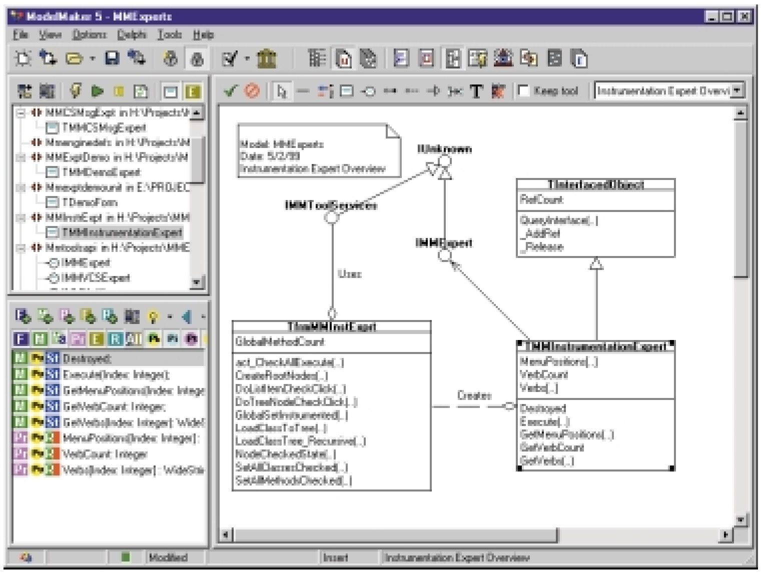
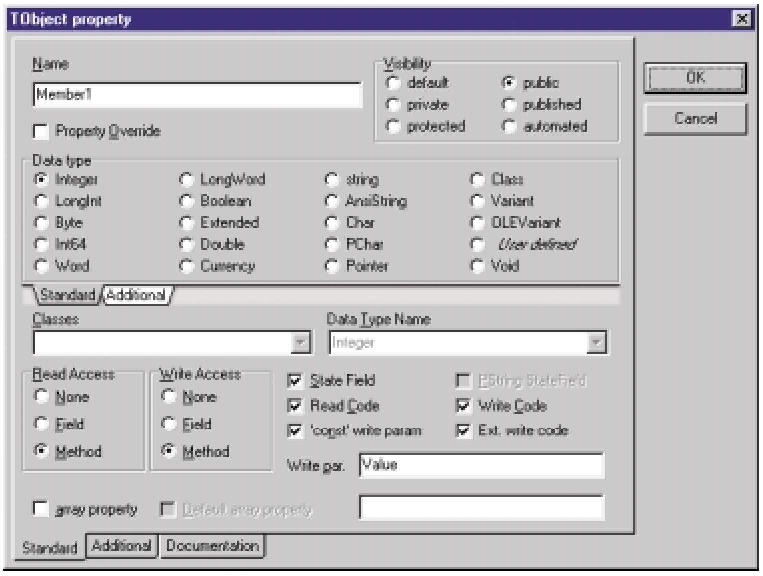


Figure 3: ModelMaker with Unit view and Diagram Editor active.



TObject property

Name: Member1

☐ Property Override

Visibility: ○ default ○ private ○ protected ● public ○ published ○ automated

Data type: ● Integer ○ LongInt ○ Byte ○ Int64 ○ Word ○ LongWord ○ Boolean ○ Extended ○ Double ○ Currency ○ string ○ AnsiString ○ Char ○ PChar ○ Pointer ○ Class ○ Variant ○ OLEVariant ○ User defined ○ Void

Standard | Additional

Classes:

Data Type Name: Integer

Read Access: ○ None ○ Field ● Method

Write Access: ○ None ○ Field ● Method

☑ State Field ☑ Read Code ☑ 'const' write param ☐ PString StateField ☑ Write Code ☑ Ext. write code

Write par. Value

☐ array property ☐ Default array property

Standard | Additional | Documentation

OK | Cancel

Figure 4: ModelMaker's TObject property dialog box.

drag an association between them. The interface will be added to the class' declaration.

In spite of ModelMaker's limited UML support, I'm still able to use the Diagram Editor for productive brainstorming and project documentation.

Class Creation

Part of why I became lost in that job interview was that I had become heavily dependent on ModelMaker's automation. Adding a new class or interface is easy, but fleshing out that new class or interface is ridiculously easy (from now on, I'll simply say "class" when referring to a class or an interface). Any time a class is selected, all of its members are displayed in the member list (see the lower-left corner of Figure 3), which is filterable to show members by kind or scope.

Clicking any of the **Add** buttons (property, field, method, event, or method resolution clause) displays the appropriate editor dialog box, and allows you to completely define the new member. Each of these dialog boxes allows total control over the members, but the Property Editor deserves particular attention (see Figure 4).

Note that we're only looking at the first page of options. The second page allows you to control such things as storage and default specifiers, implementation mapping, indexed properties, etc. As you can see on this page, you can specify the property's settings, such as visibility and type; but most notably, to the right of the **Read Access** and **Write Access** radio button groups, you see a set of options that include **State Field**, **Read Code**, **'const' write param**, **Write Code**, and **Ext. write code.** I point out these options as an example of how much ModelMaker automates.

If you add a property, the read and write access defaults to field; if you add *MyProperty*, you will also get the private field *FMyProperty*, added by ModelMaker. If you then change the read or write access to Method, *GetMyProperty* or *SetMyProperty* is automatically added. All of this is maintained by ModelMaker. Now, assuming you're using access methods, the aforementioned options allow you to do things such as automatically add standard "read code" to your *GetMyProperty* method or extended "write code" to your *SetMyProperty* method.

Suffice it to say that, with the right options set, you can click **Add Property** and have the property, the property's state field, and its read and write access methods (complete with read and write code) automatically added. That's a whole lot of code for just a few clicks.

Speaking of maximum code for minimum clicks, I have to mention a little thing called the Creational Wizard. It's an odd name, but it's a powerful tool. As you create fields (and state fields for properties), you can specify that a field is to be initialized. If you then run the Creational Wizard, it will check your object for initialized fields and suggest code to place in the object's constructor and destructor. As you update your class, you can run the wizard to update that automated code (just something else you don't have to remember to do).

Interface Support

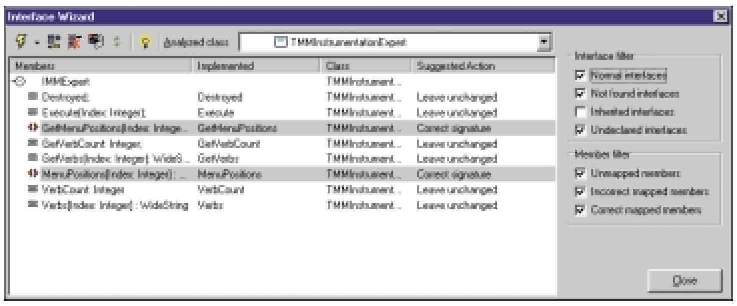
ModelMaker offers thorough COM and Object Interface support, adding interfaces to the Class Tree and simplifying creation and implementation of interfaces through things such as the Interface Wizard (see Figure 5). The Interface Wizard compares an object and the interface it implements, displaying disparities and suggestions for correcting the implementation. You can accept some or all of the wizard's suggestions, and ModelMaker will implement the suggested changes for you.

Design Pattern Support

ModelMaker actively supports the following patterns: Visitor, Observer, Wrapper, Singleton, Mediator, Decorator, ReferenceCount, and Lock. By "actively," I mean that code to support the selected pattern is added and maintained by ModelMaker, and if you choose to remove the pattern, all the code is removed for you. Most of the patterns have wizards to simplify integration of the pattern into existing classes.

Open Tools API

With version 5, ModelMaker has introduced an Open Tools API similar to Delphi's. Though still in its infancy (version 4 as of this writing), the API shows a great deal of promise. Currently, it doesn't give much access to the UI, but it does provide near-total access to the model. Take this as an example of its power: The ModelMaker developers are creating


Interface Wizard

Analyzed class: TMMInstrumentationExpert

Members	Implemented	Class	Suggested Action
IMMExpert		TMMInstrument...	
Destroyed;	Destroyed	TMMInstrument...	Leave unchanged
Execute(Index: Integer);	Execute	TMMInstrument...	Leave unchanged
GetMenuPositions(Index: Intege...	GetMenuPositions	TMMInstrument...	Correct signature
GetVerbCount: Integer;	GetVerbCount	TMMInstrument...	Leave unchanged
GetVerbs(Index: Integer): WideS...	GetVerbs	TMMInstrument...	Leave unchanged
MenuPositions[Index: Integer]: ...	MenuPositions	TMMInstrument...	Correct signature
VerbCount: Integer	VerbCount	TMMInstrument...	Leave unchanged
Verbs[Index: Integer] : WideString	Verbs	TMMInstrument...	Leave unchanged

Interface filter
- [✓] Normal interfaces
- [✓] Not found interfaces
- [ ] Inherited interfaces
- [✓] Undeclared interfaces

Member filter
- [✓] Unmapped members
- [✓] Incorrect mapped members
- [✓] Correct mapped members

Figure 5: ModelMaker's Interface Wizard.

INFORMANT FACT FILE

ModelMaker is a Delphi CASE tool that provides integrated support for reverse engineering, code maintenance, class diagrams, documentation, design patterns, code differencing, COM/object interfaces, and class hierarchy maintenance, all from a single active model.

ModelMaker
Stenenkruis 27 B
6862 XG Oosterbeek
Netherlands

E-Mail: info@modelmaker.demon.nl
Web Site: http://www.modelmaker.demon.nl
Price: US$199 for a single-user license.

the new ModelMaker UI using it. They are able to test their new UI designs in the existing ModelMaker builds. (It also allows me to send them some UI ideas without having access to the ModelMaker source.)

I've already written two experts that I use constantly. One is a CodeSite (by Raize Software Solutions) messaging expert that I can use to insert any of the CodeSite object methods into my model. The other is designed to take advantage of a new Instrumentation feature in ModelMaker. This feature was added to support products, such as CodeSite and GpProfile. Every method in ModelMaker has an Instrumentation option that, if selected, will add the contents of certain macros at the beginning and ending of the method. My expert is designed to simplify the toggling of the option for many methods at once.

Just as the third-party experts have extended Delphi immensely, expect some heavy-duty experts to extend ModelMaker as well.

Cons

No application is perfect, and as highly as I recommend ModelMaker, it does have some problems.

The printed and online documentation for ModelMaker is improving, but still slightly lacking. What's ironic about this is that I've been volunteering to create online Help files for ModelMaker. I have some files in place, and I update them when I can, but they are far from comprehensive. The current printed documentation for ModelMaker is thorough, but it's from version 3.3. There's still a great deal that is applicable in the document, and Gerrit has said that he is updating the file (removing the out-of-date, version-specific material) and creating one that will be more of a "What-to-do," and the Help files will be "How-to-do."

One common Object Pascal capability that is conspicuously unsupported is conditional defines. Although you can enter conditional code in the implementation editors, you can't, for instance, enclose a property within conditional statements. Gerrit acknowledges this limitation and says that, though he would love to offer that functionality, his team decided (rightly, I think) that the considerable amount of effort required to support this feature was better spent on other capabilities, e.g. COM support.

Another complaint I have — and this one stems from my philosophy of UI design — is that not all commands are available from all UI metaphors. There are menu people, toolbar people, and keyboard people. I tend to be a combo keyboard-and-toolbar person, but for some functionality, I am forced to go to the ModelMaker main menu or a context menu, rather than have a toolbar button to click. The good news is that the ModelMaker team is currently developing the new UI and welcomed my feedback, so this point may be moot by the time this is in print.

Conclusion

You may have noticed that I mention speaking to Gerrit Beuze a lot. This is not a result of some special relationship I have with him; rather, it's a result of the amazing support that ModelMaker offers. ModelMaker's support is one of the things that won me over early. Their response time is excellent, and they are always ready to hear your suggestions for their product. They have repeatedly shocked me by releasing a new build that includes a feature from a suggestion that I may have made only days before.

I actually use this tool more throughout my day than I do Delphi. Obviously, I use Delphi to create forms and compile code and such, but for virtually everything else, I use ModelMaker. By the way, even compiling can be triggered from ModelMaker (choose from **Syntax Check**, **Compile**, or **Build All**).

ModelMaker is worth well more than the asking price. My suggestion is that you download the evaluation demonstration immediately. ∆

Robert has spent the last five years in the Dallas/Ft. Worth area developing Delphi applications. Prior to that, he specialized in electronic publishing and graphic design (skills now put to use as a Web developer). Robert is also a musician and has played professionally for 15 years.

An Interview with Ray Konopka

Ray Konopka is president of Raize Software Solutions, Inc., an independent consulting firm and producer of Delphi tools. He has been the chief architect of its products, including Raize Components and the new debugging tool, CodeSite. A contributing editor of *Visual Developer Magazine*, writing the "Delphi by Design" column, and author of the best-selling *Developing Custom Delphi Components* and *Developing Custom Delphi 3 Components* from Coriolis Group Books, Konopka is a frequent and popular speaker at international developer conferences and user groups. Information about his books and products can be found at http://www.raize.com.

DI: You're involved in a variety of activities: You produce software, write books and articles, do some consulting, and give talks and presentations. What are some of the challenges of such a varied career, and what strategies have you developed to "keep it all together?"

Konopka: It sure does sound like a lot of different tasks, doesn't it? Well, it certainly helps that I'm a highly organized person, but the main reason I can juggle so many tasks is that I try to reuse the results of my work as much as possible.

For example, I might be working on one of our products and come across a technique that is not widely known or documented — or is just plain cool. I then use the knowledge I gained from the experience and write an article about the topic for my "Delphi by Design" column. This in turn could easily become a new presentation that I give at developer conferences. Of course, each task requires expressing the information in slightly different ways, but the technical details are reused over and over.

DI: I understand that Raize Components grew out of your work as a consultant. Could you talk a bit about how that happened?

Konopka: Shortly after I started using Delphi (way back in early 1994 — it wasn't even called Delphi back then), I had envisioned creating a set of commercial components. As a result, I had already created several custom components by the time I started Raize Software Solutions in February of 1995. Unfortunately, I didn't have the funding to create a commercial product, so I began consulting.

Back then, I was often asked to create components that provided features that were not available in Delphi 1. I arranged it with all of our clients that my company would own the rights to all general-purpose components that I created. Of course, the client retained all proprietary components.

Those general-purpose components provided much of the material that became part of my first book. In fact, while writing the book, one of my goals was to present a set of components that developers would consider on par with a commercial product. It was clear I reached this goal when I started to receive messages from developers asking if it was okay for them to use the components in their applications.

It was at this time that we decided to produce the commercial version of Raize Components. Of course, we knew we needed to create something way beyond what was presented in the book, but we started with 16 of the book's components as a foundation. After enhancing these components, we created 27 new components for the first version of Raize Components.

DI: I'd like to explore the world of component writing with you a bit. At one time many developers viewed writing components as an "advanced topic;" is this still the case?

Konopka: One of my goals in writing *Developing Custom Delphi Components* was to make component writing more approachable. While I believe this is now true, it really comes down to what you want to create. For example, do you think many developers would consider object-oriented programming an advanced topic? At one time they did, but I would hope that this is no longer the case. Of course, there are still areas within OOP which are considered advanced. For example, design patterns.

My point is that the same can be said with respect to component writing. For example, the process of creating custom components is well documented, and there are even tools available to help you. However, I believe everyone would consider creating a new custom grid component to be an advanced task.

DI: I'd like to discuss come general Delphi issues. While you support C++Builder in your various products, Delphi seems to be central to your programming activities. What do you consider its most important strength? What is the one area of Delphi you feel needs to be improved the most?

Konopka: There are two aspects of Delphi that I consider extremely important. The first is Object Pascal. It's an extremely powerful and elegant language, and more importantly, it's easy to read and maintain. The second is the fact that components are objects. The VCL is truly the power that drives Delphi.

As for improvements, I don't believe I could single out one particular area. There are areas throughout the product that could be improved. For example, I would very much like to be able to register multiple component editors for a given component. It is possible to manage this now by chaining editors together, but there are several problems with the approach and it really should be built into Delphi.

In addition, I'd like to see component templates replaced with a way to build true compound components visually. I'd also like to see the Delphi IDE move toward an interface similar to Visual Studio. That is, an MDI-style approach where I can create a work area that is used for all my projects. The docking support in Delphi 4 provides some of this, but it's managed on a per-project basis and is focused around the Code Editor rather than a main frame window.

DI: One of the more pleasant surprises that accompanied the release of Delphi 4 was the almost immediate release of a Delphi 4 version of the Raize Components. While some companies were able to provide timely updates, others were not. What steps did you have to take to be ready with the update? Were there any problems?

Konopka: As a tools and component provider for Delphi, we're eligible to participate in Delphi field tests. As each new field test is released, we check to make sure our products work with the new build. As for problems, there were some minor changes required, but the VCL has essentially been unchanged since version 1.

DI: There have been many changes at Inprise and the company has certainly had its ups and downs. Often I see discussions on the Internet about Inprise in general and Delphi in particular, focusing on the direction in which this company is moving and the implications for its flagship product. Could you share some of your observations and views? What will the future hold for Delphi?

Konopka: Back in 1998, I would have answered this question by commenting on how at the Inprise Conference in Denver, the major push by Inprise was the Enterprise and Distributed Computing. Certainly valid topics, but not everyone is working in these areas. It appeared that Inprise was sacrificing its current customer base in its attempts to attract the enterprise. Furthermore, it was unclear what role Delphi was going to play in all of this. For example, it did not go unnoticed that Delphi was not even mentioned during the conference's opening keynote.

Fortunately, I now have a much more positive outlook on all this. I believe the Inprise/Borland.com split is a great move. As two companies, Inprise can continue to focus on the enterprise, while Borland can once again focus on their development tools, and more importantly on individual developers. It's important to note that the two customer bases are not mutually exclusive — every corporate enterprise development team is made up of individual developers.

As for the future of Delphi, there are several factors that will determine the future of Delphi. First and foremost, Borland needs to continue to make Delphi the best development tool for Windows. However, in the long term, I think it would make sense for Borland to consider creating Delphi for other platforms, such as Windows CE and Linux.

DI: I think that would please a lot of developers. Like many others, I've been interested in the health of the Delphi community, which I think is very good. I see the proliferation of discussion groups, user groups, Web sites, and organizations like Project Jedi as an indication that developers are willing to reach out and help each other. Could you comment on this?

Konopka: The Delphi developer community is like no other that I have been involved with. Delphi developers are a very loyal group of people and have demonstrated again and again their willingness to help other developers. Just consider the number of free Delphi components that are available on the Web. To other developers, this is a great resource, but as a commercial provider of Delphi components and tools, we are forced to view all this free software as competition. No easy feat, but we have managed.

Unfortunately, not all the free software that is available is legitimate. There have been several instances where technology that we developed has appeared in a free product. Some developers believe that copyrights of other products do not apply if their product is given away for free with source code. If this type of behavior continues, it will deteriorate the Delphi third-party commercial market.

DI: Are you planning to write any new books in the near future?

Konopka: At the time of this interview, there are no formal plans for a new book. However, I am talking with the publisher about a third edition of *Developing Custom Delphi Components*.

DI: I'd like to end with a very general question. Windows is very popular right now as a computing environment, particularly in the home computing market. Do you see any indications that this will change in the future?

Konopka: I believe Windows will remain popular for quite some time. There is simply too much money invested in the technology (machines and software) for companies and consumers to switch to something different. This is not to say that something else won't come along. Linux is getting a lot of press these days, just like Java did a year or two ago. But like Java, Linux is being targeted more at the server market. I believe client machines and consumer PCs will continue to be Windows-based for quite some time. ∆

— *Alan C. Moore, Ph.D.*

Note: This is an abridged version of the interview. The complete interview is available on the Delphi Informant Web site at http://www.DelphiMag.com.

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.