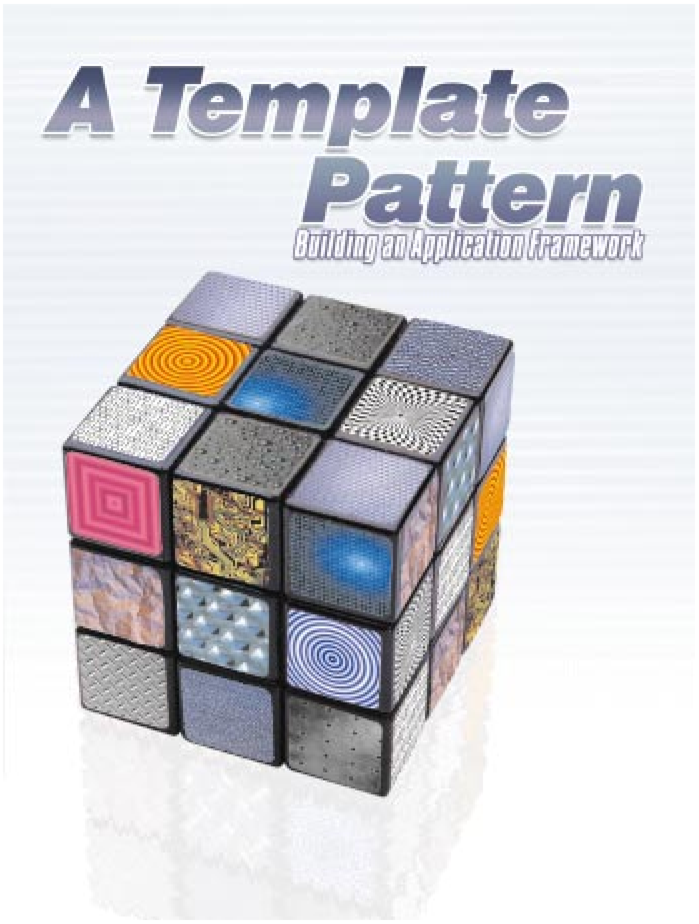
May 1999, Volume 5, Number 5



Cover Art By: *Darryl Dennis*

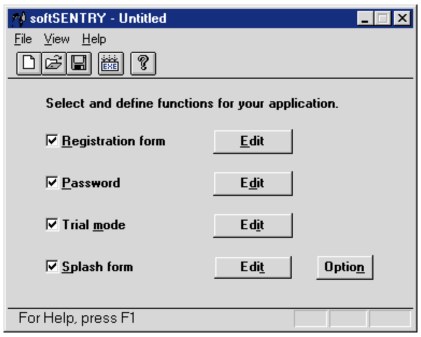
20/20 Software Releases softSENTRY 2.1

20/20 Software, Inc. released *softSENTRY 2.1*, an enhanced version of its trialware and software protection tool that works by injecting directly into executable (.exe) files, or by calling a dynamic-link library (.dll) file.

Protected programs can query internal softSENTRY data, such as System ID, values of user input strings, values of limitation counters, and the active mode in which the program is running. Protected programs can also display a registration form at any time. Enhanced formulas increase the complexity and flexibility of password definitions.

User input strings, such as "name" and "serial number," can be used in password formulas to ensure that the correct registration information must be entered for the password to work. Version 2.1 also adds 99 new operators to further increase the difficulty of breaking a password algorithm.

Additional security measures have been incorporated to protect against attacks on the "footprint" softSENTRY puts on a computer.



Select and define functions for your application.

☑ Registration form
☑ Password
☑ Trial mode
☑ Splash form

20/20 Software, Inc.
Price: US$249 for either 16- or 32-bit Lite versions; US$695 for complete version (includes 16- and 32-bit functionality).
Phone: (800) 735-2020
Web Site: http://www.twenty.com

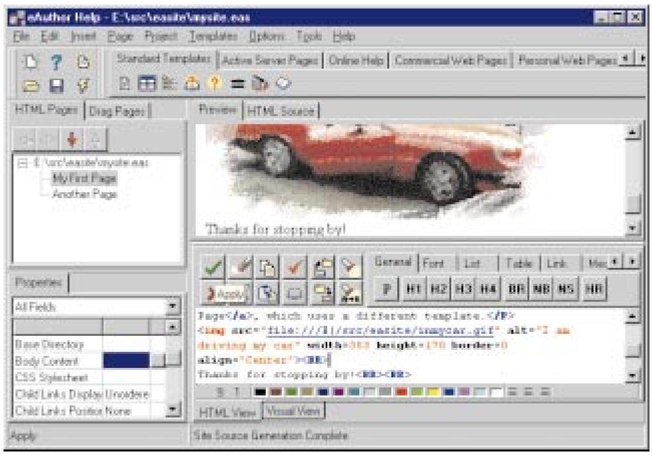
HyperAct Announces eAuthor Help 3.10

HyperAct, Inc. announced version 3.10 of ***eAuthor Help***, its template-based RAD authoring tool. eAuthor was designed to provide a rich authoring environment for large-scale Web sites and HTML Help projects.

This release introduces new features, such as WYSIWYG editing, hard-copy documentation creation, XML document creation, and the eAuthor SDK, which allows eAuthor to be extended with plug-ins, COM-based templates, and more.

With the new ActiveScript support, repetitive documentation creation tasks can be scripted using JavaScript or external application via automation.

In addition, enhancements were added to features available before, such as an improved raw-HTML editor, a faster editing environment, improved spell checker, thesaurus, drag-and-drop capabilities, improved documentation, and more.



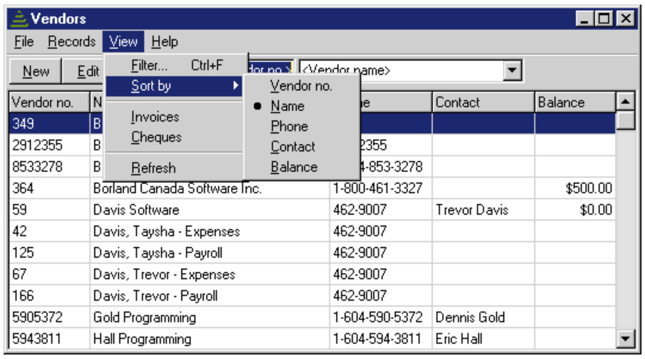
```
Page</a>, which uses a different template.</P>
<img src="file:///E|/src/easite/mmycar.gif" alt="I am
driving my car" width=383 height=170 border=0
align="Center"><BR>
Thanks for stopping by!<BR><BR>
```

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

Davis Business Systems and Psi Computer Consultants Offer BS/1 Small Business

Davis Business Systems Ltd. and **Psi Computer Consultants Pty Ltd.** have developed ***BS/1 Small Business***, a suite of accounting programs designed for use by small- to medium-sized businesses. BS/1 is an ActiveX control that provides software developers a cost-effective way to include a fully functioning accounting system in their own custom application.

BS/1 Small Business ActiveX can be applied in a wider range of languages than a native VCL. A developer can use it for one client's VB application and for another client's Delphi application, without purchasing two versions of the same software.



Vendor no.	N	e	Contact	Balance
349	B			
2912355	B	2355		
8533278	B	4-853-3278		
364	Borland Canada Software Inc.	1-800-461-3327		$500.00
59	Davis Software	462-9007	Trevor Davis	$0.00
42	Davis, Taysha - Expenses	462-9007		
125	Davis, Taysha - Payroll	462-9007		
67	Davis, Trevor - Expenses	462-9007		
166	Davis, Trevor - Payroll	462-9007		
5905372	Gold Programming	1-604-590-5372	Dennis Gold	
5943811	Hall Programming	1-604-594-3811	Eric Hall	

Davis Business Systems Ltd./Psi Computer Consultants Pty Ltd.
Price: From US$79 for a single-user license.
Phone: (604) 462-9007
Web Site: http://www.dbsonline.com

Realsoft Releases SofTrak

Realsoft Development announced the release of ***SofTrak*** (in beta version at press time), an integrated system for customer tracking, support logging, help-desk management, invoicing, licensing, registration coding, version control, and more.

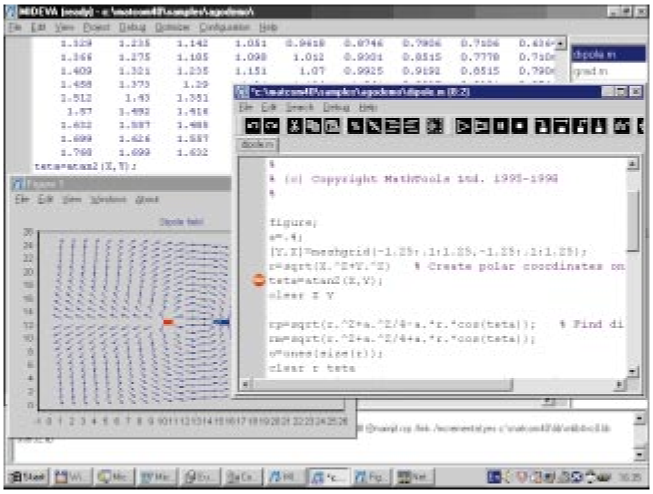
SofTrak was written with Delphi 4 using native components for Win95/98/NT4 and has a Microsoft Outlook-style interface. Databases are xBase-compatible using the Advantage engine. Client/server options are available for larger networks. Registration components are ActiveX and native Delphi, and registration codes use a secure four-stage encryption.

SofTrak's tracking features include dual-pane Customer Database and filtered Tracking Database; auto date/time and username stamping for each entry; unlimited notes section for each entry; tracking of product serial numbers and quantity on hand; custom reports for versions, renewals, and invoices; and more.

Realsoft Development
Price: SofTrak Lite (single user), US$249; SofTrak Professional (up to five users), US$495; additional discounts are available for multiple users.
Phone: (800) 929-3991
Web Site: http://www.realsoftdev.com

MathTools Announces MIDEVA

MathTools Ltd. announced *MIDEVA*, its scientific integrated development environment. MIDEVA presents a complete environment for developing and running scientific applications.



```
1.329   1.235   1.142   1.051  0.9618  0.8746  0.7906  0.7106  0.636
1.366   1.275   1.185   1.098   1.012  0.9201  0.8515  0.7770  0.710
1.409   1.321   1.235   1.151    1.07  0.9925  0.9192  0.8515  0.790
1.458   1.373    1.29
1.512    1.43   1.351
 1.57   1.492   1.418
1.632   1.557   1.488
1.699   1.626   1.557
1.768   1.699   1.632
teta=atan2(X,Y);
```

```
%
% (c) Copyright MathTools Ltd. 1995-1998
%

figure;
a=.4;
[Y,X]=meshgrid(-1.25:.1:1.25,-1.25:.1:1.25);
r=sqrt(X.^2+Y.^2)   % Create polar coordinates on
teta=atan2(X,Y);
clear X Y

rp=sqrt(r.^2+a.^2/4-a.*r.*cos(teta));   % Find di
rm=sqrt(r.^2+a.^2/4+a.*r.*cos(teta));
c=ones(size(r));
clear r teta
```

MIDEVA includes an m-files interpreter, syntax-highlighting editor, source-level debugger, optimizer, and online reference guide. MIDEVA also provides the ability to compile m-files into executables, Delphi/Visual Basic/Excel DLLs, and Debug/Release modes. It is compatible with MATLAB 4.x and 5.

MathTools Ltd.
Price: Single commercial license, US$999; academic license, US$299.
Phone: (212) 208-4476
Web Site: http://www.mathtools.com

Watergate Announces ActiveX and CGI in PC-Doctor

Watergate Software, Inc. announced the integration of ActiveX and CGI technologies into its ***PC-Doctor*** line of diagnostic software products.

The PC-Doctor ActiveX control will allow PC-Doctor to be used in conjunction with Web-based applications by enabling them to display PC-Doctor diagnostic data. Web-based applications can use JavaScript or VBScript to communicate with PC-Doctor's diagnostic and system information capabilities online in real time. PC-Doctor CGI is an HTTP server-callable program that allows for a command line-driven, Web-based interface to PC-Doctor.

PC-Doctor's Modular Core Technology contains over 300 test functions optimized for implementation throughout each stage of the product life cycle. Each primary product — PC-Doctor Factory, PC-Doctor for Windows, PC-Doctor for Windows NT, and PC-Doctor Service Center — provides a specific diagnostic application of the core technology.

Watergate Software, Inc.
Price: Contact Watergate for pricing.
Phone: (510) 596-2080
Web Site: http://www.ws.com

RightDoc Releases RightDoc 1.0

RightDoc Co. announced ***RightDoc 1.0***, an XML-based content management and publishing engine that creates personalized document content from applications to view, print, and generate HTML4, PDF, and PostScript formats.

RightDoc provides for the creation of XML documents in Enterprise or Web server-based Delphi, Visual Basic, Visual C++, Visual J++, Excel, Access, and others. It creates personalized forms, reports, billing statements, legal contracts, letters and more.

With built-in variable text merging, conditional formatting, and conditional processing, RightDoc is application data-driven. Companies can create sets of "Smart" documents, with each document having the ability to alter itself based on application data provided.

Form-based built-in editors create and modify processing tags, styles and style properties, and XML entities.

RightDoc Co.
Price: US$249 per development seat (royalty-free).
Phone: (509) 464-1059
Web Site: http://www.rightdoc.com

Linder Software and Albert's Ambry Announce LSPzip

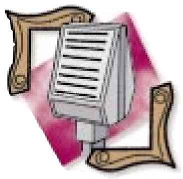
Linder Software, in association with Albert's Ambry, has released the latest version of *LSPzip*, a compression library for developers working in Delphi, C++Builder, Clarion, Visual Basic, or Visual C++ environments.

LSPzip comes with two compression libraries: LSPack and LSZip. The LSZip libraries are compatible with the latest PKZIP archive format.

LSPzip costs US$169 for a single-user license. A trial version may be downloaded from the Albert's Ambry Web site at http://www.alberts.com/authorpages/00013316/prod_698.htm.

News LINE

May 1999



SkyLine Tools Announces the Programmer of The Year Award

SkyLine Tools Imaging presented the Programmer of the Year Award to Shelley Emmerson, Imaging Technology Expert for the Royal Canadian Mounted Police.

Using the SkyLine Tools Imaging Corporate Suite and VideoLab Pro, Emmerson created an application that could input, store, and organize the thousands of images, documents, and video footage resulting from the investigation of the tragic 1998 Swissair Flight 111 crash, which claimed 225 lives when the aircraft went down in more than 200 feet of water near Halifax, Nova Scotia. Emmerson put the application together in 12 hours.

Other tools used were Multi-Edit from American Cybernetics, InfoPower 4 from Woll2Woll, InterBase 5 SQL relational database from InterBase Software, and Crystal Reports from Seagate Software. A Windows NT workstation was utilized as the application server.

InterBase Releases InterBase 5.5 for SCO

Scotts Valley, CA — InterBase Software Corp. announced the availability of InterBase 5.5 for the Santa Cruz Operation (SCO) UNIX OpenServer operating system. InterBase 5.5 for SCO offers performance improvements and enhanced stability, delivering a solution for SCO developers and value added resellers (VARs).

InterBase 5.5 for SCO includes version 1.5 of InterClient, an all-Java JDBC driver. InterClient 1.5 adds direct international support for user-specified character sets.

Stability in InterBase 5.5 has been improved by adding such features as protection for online metadata updates of Triggers and Stored Procedures by the strength of the InterBase 5.5 versioning engine. User Defined Functions (UDFs) have added safety features in Windows, and the UDF library has been expanded.

Performance enhancements include more efficient memory usage and a new multi-threaded ODBC 3.0 driver that adds support for SQL-92 ROLES and international character sets.

For more information on InterBase, visit http://www.interbase.com.

Inprise Strengthens AS/400 Global Partnership with SystemObjects

Amsterdam, Netherlands — Inprise Corp. announced a long-term, world-wide exclusive licensing agreement with SystemObjects Corp. to further develop, support, and market Inprise's Windows-based visual development tools for the IBM AS/400 platform.

SystemObjects is a key Inprise partner, having helped develop Delphi/400 and C++Builder/400. Under the new agreement, SystemObjects assumes the future development, maintenance, and marketing of Delphi/400 and C++Builder/400 internationally. Later this year, SystemObjects plans a Java development solution for the AS/400 based on Inprise's JBuilder.

For more information on SystemObjects, visit http://www.systemobjects.com, or call (800) 586-5516.

HREF Presents Live eSeminars

Santa Rosa, CA — HREF Tools Corp. announced its new live eSeminars using U-VU Network's Internet Conference Service Software. U-VU is a built-with-WebHub system that enables the presenter to broadcast a live, interactive presentation while participants watch visual screens and listen to an audio stream. Online participants log in, see the slide show, listen to the presenter, ask questions by typing them in, and chat (using text) with other users before and after the presentation.

HREF is also partnering with other Delphi developers who want to use the U-VU technology to create their own eSeminars.

The presentations run for about an hour and focus on topics of interest relating to Web development using HREF's line of WebHub products and related utilities.

HREF has completed five presentations with attendees from all over the world, including the US, Brazil, Lithuania, Mexico, Norway, and Australia.

HREF's series "WebHub Tech Talk Radio" brings technical content to developers every two weeks. "WebHub Sizzle" presentations, for developers evaluating the power and usability of WebHub, also air on an ongoing basis.

For more information and a complete schedule, visit http://www.href.com/present.

Oracle Expands Relationship with Inprise for VisiBroker CORBA

Scotts Valley, CA — Inprise Corp. announced it signed a licensing agreement with Oracle Corp. Under the terms of the multi-year agreement, Oracle has selected Inprise's VisiBroker as one of its worldwide standards for CORBA object request broker (ORB) technology. As of press time, VisiBroker has been integrated into Oracle8i, Oracle Application Server, and other Oracle products.

Oracle8i extends Oracle's technology leadership in the areas of transaction processing, data warehousing, mobile computing, and high-availability systems.

Inprise Creates Separate Divisions

Scotts Valley, CA — In conjunction with its fiscal year 1998 and fourth quarter earnings announcement, Inprise Corp. announced a new business structure with the formation of two divisions: Inprise and borland.com.

borland.com plans to become a premier destination Web site that will serve individual developers' needs for a range of advanced Internet products and technologies, including those from third parties. The restructuring will include a streamlining of facilities, headcount, and product lines designed to increase operating efficiency. borland.com offerings will include Borland Delphi, C++Builder, JBuilder, and InterBase.

Solutions offered by the Inprise division will be sold through its direct sales organization and partner channel, and will include Inprise Application Server, JBuilder for AppServer, AppCenter, VisiBroker, ITS and Entera. The division will continue to expand its professional service organization to provide comprehensive integration consulting and training capabilities for the enterprise integration marketplace.

Jim Weil, currently President of Inprise subsidiary InterBase, has been named President of the Inprise division, which will be headquartered in San Mateo, CA. John Floisand, Senior Vice President of worldwide sales, has been appointed President of borland.com, which will remain in Scotts Valley. Both division heads will report to Delbert W. Yocam, Inprise chairman and CEO.

By Xavier Pacheco

The Template Method Pattern

Building a UI Application Framework

In the March, 1999 issue of *Delphi Informant*, we discussed the Singleton pattern. This month, we'll create a simple application framework using a commonly used pattern, the Template Method. This article will present two concepts: designing a framework around which you develop a user interface, and using patterns to accomplish this.

In the following section, I'll explain the concept of frameworks, what they are, and how they help us in designing applications. I'll also explain what frameworks have to do with patterns. In this and future articles, I'll make reference to the book *Design Patterns: Elements of Reusable Object-Oriented Software* [Addison-Wesley, 1994] by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, otherwise known as the Gang of Four (GoF).

OOP Frameworks

The GoF define a *framework* as "... a set of cooperating classes that make up a reusable design for a specific class of software." A framework defines the constraints within which a product can be constructed. In other words, frameworks define the architecture or structure of the final product. Although it might seem a bit restrictive, this is an extremely beneficial practice to use in software development. Developers need not focus on the flow or structure of the user interface (UI), so they can instead focus on specific functionality.

With a tool like Delphi, it's easy to fall into the trap of designing the UI without careful forethought as to how it's supposed to work. A typical UI may start out as a main form with a menu. Later, a page control is added to separate logical functionality. As the UI develops, more functionality is added, removed, and moved into separate forms. Typically, the layout and flow of the UI isn't decided upon until much later in the development process. By first designing a framework, one can reduce the amount of re-coding and redesign that occurs during UI development. A UI framework acts as a plug-in mechanism to facilitate a more structured approach to UI development.

So how do patterns and frameworks relate to one another? A developer uses patterns to build frameworks for software. Patterns and frameworks aren't the same; the GoF have defined three major differences. Design patterns are:

- more abstract than frameworks;
- smaller architectural elements than frameworks; and
- less specialized than frameworks.

The first difference emphasizes that patterns focus more on abstractions or methodologies. Although you can create a concrete framework that structures a specific domain, the same isn't true for a pattern. Frameworks exist in actual code that may be copied and reused in different applications. The same isn't true for patterns, as they must be implemented each time they're used. As for patterns, it's the pattern that's reused, not the implementation of that pattern. Framework implementations, on the other hand, may be reused.

The second difference emphasizes how patterns are smaller in size. A framework may be composed of many patterns, whereas a pattern would never be composed of frameworks. A pattern is more like an algorithm used in a framework, and the framework is what defines the architecture of a system.

The third difference emphasizes how frameworks are written to a specific application domain. For example, you might define a framework for a graphics manipulation application. There isn't a pattern that specifically addresses this domain, although you might use a pattern or combination thereof to construct this specific framework. The key is that the framework would only be useful for this type of application, whereas the patterns used to construct the framework may be used on any type of application.

The Template Method Pattern

As the GoF state in *Design Patterns*, a Template Method pattern is intended to: "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."

The Template Method is probably the most widely used pattern by Delphi developers. It's based on the concept of inheritance; in fact, you'll see examples of the Template Method strewn throughout the VCL code. Figure 1 depicts the structure of the Template Method pattern.

Template Method vs. Abstract Class

The Template Method pattern and abstract classes aren't one in the same. An abstract class provides the declaration of a class interface while deferring its implementation to its subclasses. Therefore, the various implementations of that abstract class provide the ability to create different behaviors for the client.

A Template Method accomplishes the same at the method/algorithmic level. A Template Method is defined *and* implemented in an abstract class to provide structure for an action performed by that class. All descendant classes (concrete classes) override and implement smaller pieces — abstract methods — of the class to help give it identity. Therefore, a Template Method is actually part of an abstract class even though it's usually fully defined and implemented. Look at it as a way to extend the control in which you attempt to define an abstract class. Ideally, the Template Method isn't declared as **virtual** or **dynamic**. The purpose of this method is to establish a consistent process among the descendant classes. For example, a Template Method might look something like this:

```
procedure TSomeObject.TemplateMethod;
var
  i: Integer;
begin
  PrimitiveMethod1;
  for I := 0 to 10 do
    PrimitiveMethod2;
  PrimitiveMethod1;
end;
```

In this example, the *TemplateMethod* method defines a simple template by which two primitive methods are invoked. The implementations of these primitive methods may vary so that they perform entirely different actions. For this to work, we must use object inheritance to allow for the varying behaviors for the primitive methods. Therefore, it can be assumed that the Template Method pattern is dependent on abstract classes, i.e. they go hand in hand.

Method Types

We've nailed down the issue of Template Methods being routines with behaviors that change internally without the client knowing of this change. Because this method is contained in a class, it follows we must provide an inheritance model to allow us to create different subclasses to which we provide the differing method behaviors. I'll focus less on the Template Method itself, and more on the abstract class used to implement a Template Method. In fact, because this is a technique so common in OOP development, I'm surprised we don't see Template Class or Abstract Class presented as discrete patterns.

Another very good book on patterns is *The Design Patterns Smalltalk Companion* by Kyle Brown, Bobby Woolf, and Sherman R. Alpert [Addison-Wesley, 1998]. It presents four types of methods that might appear in an abstract class: template, concrete, abstract, and hook. In Delphi, I'd suggest a fifth type, which I'll refer to as the event (see Figure 2).

Implementing Application Frameworks

In my earlier description of the initial approach to UI design, I described the typical process that a developer might use to put together the UI. Although this might be fine for a small application, or an application being developed by one person, it presents a problem with applications being developed as functional pieces. It's also a problem where different people are working on different parts of the UI.

A better approach would be if you separated the UI/functional pieces and brought them together later — during a build process — after developers have finished developing their parts. This technique is made possible by using a framework. The code that programmers write must adhere to that framework.

The example in Figure 3 depicts the modularity of a framework. The Shell Application owns a View Manager. The View Manager

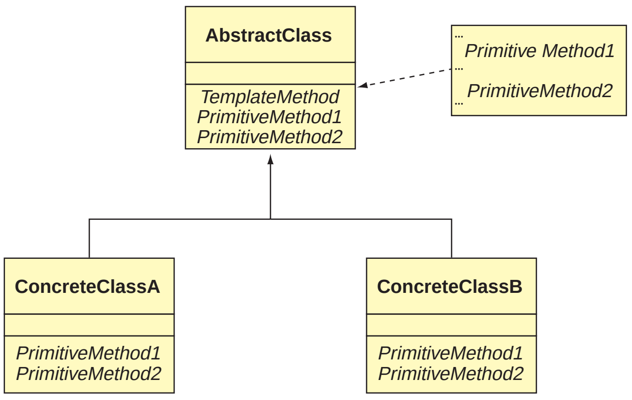


Figure 1: The Template Method pattern structure.

Type	Description
Template	A combination of the remaining four methods forming an algorithm. Subclasses typically don't change this method and simply override the methods they call.
Concrete	A method hard-coded by the abstract class that doesn't get overridden by the subclasses. In Delphi, this is a **static** method.
Abstract	A method declared by, and implemented by, each subclass. In Delphi, the keyword **abstract** is used to define this type of method.
Hook	A method that provides default functionality, and that may be overridden by subclasses. In Delphi, this is the **virtual** or **dynamic** method.
Event	A pointer to a method. This method may be provided by the client of the abstract class. It will be called if it exists by the abstract class. When a method is provided to the method pointer (e.g. *OnClick* event property) it's referred to as an "event handler." Component writers will be familiar with this method type.

Figure 2: Method types in an abstract class.

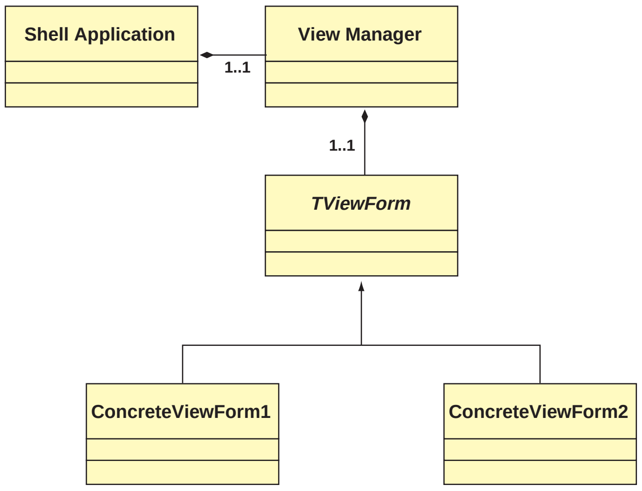


Figure 3: An example UI framework.

```
procedure TWinControl.CreateWnd;
var
  Params: TCreateParams;
  TempClass: TWndClass;
  ClassRegistered: Boolean;
begin
  CreateParams(Params);
  with Params do begin
    // Code removed.
    if not ClassRegistered or
      (TempClass.lpfnWndProc <> @InitWndProc) then begin
        // Code removed.
    end;
    // Code removed.
    CreateWindowHandle(Params);
    // Code removed.
  end;
end;
```

Figure 4: Skeleton of the *TWinControl.CreateWnd* Template Method.

is the intermediary between the Shell Application and each *TViewForm*. *TViewForm* is an abstract class that defines the interface with which the View Manager and Shell Application interact. Developers create implementations of *TViewForm* that adhere to the interface defined by *TViewForm*. Because each *TViewForm* descendant is an independent form, it can be developed apart from the Shell Application. When the functionality for a *TViewForm* is complete, it's incorporated with the main application.

Defining the Abstract *TViewForm*

Listing One (on page 9) shows the source for the abstract *TViewForm* (this and all other source is available for download; see end of article for details). This class declares the methods and properties that enable it to be embedded as a child window. Specifically, it declares two constructors. The constructor that takes a *TWinControl* parameter, *AParent*, assigns the value passed in as *AParent* to a temporary variable. This will be used in the overridden *Loaded* method that sets up the appropriate property values required by this form to be embedded.

TViewForm also declares three hook methods that may be overridden by its descendants. These are "reader" methods for the properties *ViewDescription*, *ViewMenu*, and *ViewToolbar*. Whenever you declare a virtual class containing reader methods for properties, it's a good idea to not keep these methods as abstract, because the descendants of the class may not need to implement them. For example, not all *TViewForm* descendants will return a *TToolBar*, *TPopupMenu*, or string description. Instead of forcing the descendant class to implement these methods, we'll just implement them ourselves to provide default **nil**/empty values, which is what the descendant class is going to have to do anyway.

TViewForm declares one abstract method, *CanChange*. Because *TViewForm* descendants are to be embedded into a *TTabControl*, you may want to prevent the user from switching tabbed pages until a certain operation is complete. The Shell Application doesn't know what functionality the *TViewForm* provides and, therefore, can't prevent the tab switch from occurring. However, the Shell Application can call the *CanSwitch* method to determine if the tab switch is valid.

TViewForm provides only hook and abstract methods, although it carries along concrete, template, and event methods from its own ancestors. For example, one of *TViewForm*'s ancestors is *TWinControl*, which defines two virtual methods that may be overridden by its descendants, *CreateWindowHandle* and *CreateParams*. The declarations of these methods are shown here:

```
TWinControl = class(TControl)
  procedure CreateParams(
    var Params: TCreateParams); virtual;
  procedure CreateWindowHandle(
    const Params: TCreateParams); virtual;
end;
```

Both methods are used in the *TWinControl.CreateWnd* method, which looks similar to Figure 4. I removed much of the code to conserve space. You'll see that both the *CreateParams* and *CreateWindowHandle* methods are called in the context of the *CreateWnd* method. It's possible for descendant classes to override both these methods to provide different behaviors described by the Template Method pattern. In fact, I override the *CreateParams* method to ensure that *TViewForm* can be embedded as a child window. Other than its name, this Template Method pattern is nothing new to most Delphi developers. The same is true for an event method as shown here:

```
procedure TCustomForm.DoClose(var Action: TCloseAction);
begin
  if Assigned(FOnClose) then FOnClose(Self, Action);
end;
```

This procedure shows the *TCustomForm.DoClose* method. *TCustomForm* is another *TViewForm* ancestor. This method checks to see that its field, *FOnClose*, is referring to a method. *FOnClose* is a method pointer that, if valid, gets called. Again, this technique isn't new to component developers.

Creating a *TViewForm* Subclass

Listing Two (on page 10) shows *TViewForm2*, one of three *TViewForm* descendants accompanying this article's example program. I'm illustrating this form because it's more complete. This form is a simple database form containing a database connection, a *TDBGrid* to display the data, a *TToolbar* for navigation, and a popup menu. This form can function independently from the main application. It can also be integrated with the main application, because it adheres to the interface defined by *TViewForm*.

Notice how I've overridden the methods declared by *TViewForm*, and provided the proper results for the Shell Application. For example, *GetViewDescription* returns a valid string, and *GetViewMenu* returns a valid *TPopupMenu* for integration with the Shell Application. Also, examine the *CanChange* method. This method passes False if *Table1* is

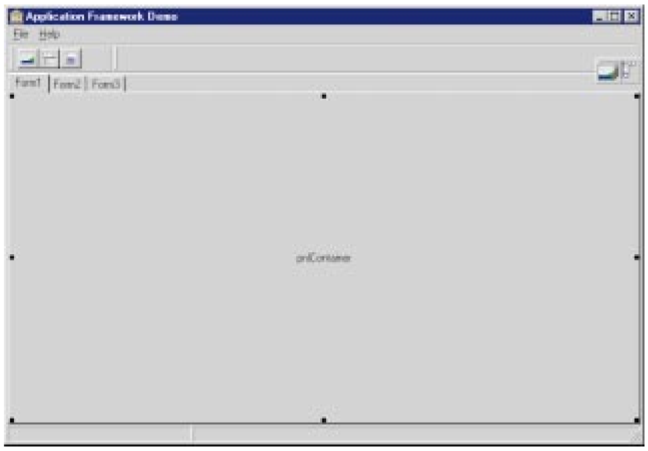
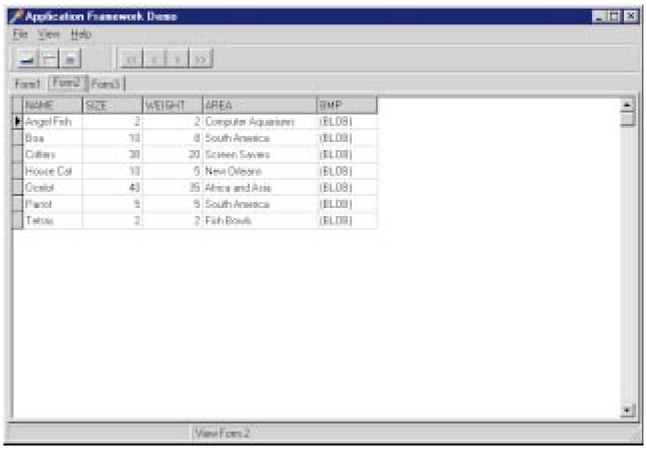


Figure 5: The Shell Application main form.



NAME	SIZE	WEIGHT	AREA	BMP
Angel Fish	2	2	Computer Aquariums	(BLOB)
Boa	10	8	South America	(BLOB)
Critters	30	20	Screen Savers	(BLOB)
House Cat	10	5	New Orleans	(BLOB)
Ocelot	40	35	Africa and Asia	(BLOB)
Parrot	5	5	South America	(BLOB)
Tetras	2	2	Fish Bowls	(BLOB)

Figure 7: *TViewForm2* at run time.

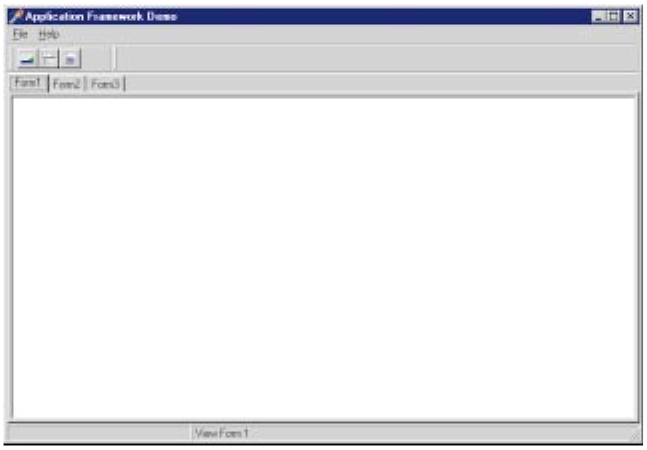


Figure 6: *TViewForm1* at run time.

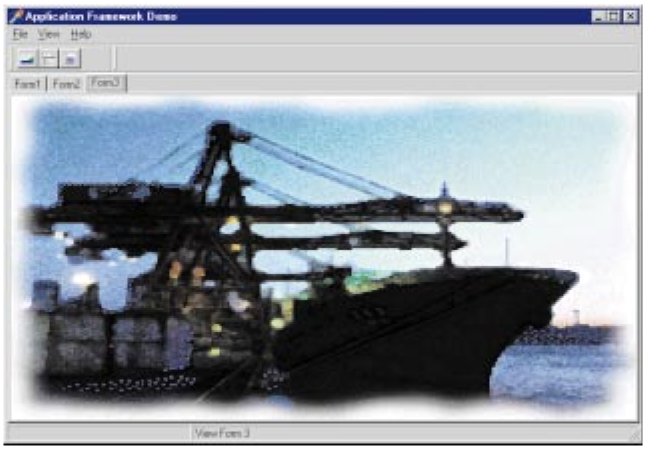


Figure 8: *TViewForm3* at run time.

in a mode other than *dsBrowse*, thus forcing the user to save his or her work. There are similar forms in the example you can study.

Creating the Shell Application

The Shell Application for this framework is shown in Listing Three (beginning on page 10). Figure 5 shows the main form for the Shell Application. Notice that the primary pieces to the main form are the *TTabControl*, *TCoolBar*, and *TMainMenu*. The *TTabControl* contains a *TPanel*, *pnlContainer*, that will serve as the container to the *TViewForm* descendants. When the user selects a new tabbed page, the *TViewForm* corresponding to that tab will be loaded, and the previous form will be unloaded. *TCoolBar* contains two bands. The first band is an application level band. The *TToolButton* objects on this band each correspond to one of the three *TViewForm* descendants and invokes them as normal modal forms. The second band, which appears empty, contains a *TPanel*, *pnlViewTB*, that will contain the *TToolBar* for the *TViewForm* that's returned by the *TViewForm.ViewToolBar* property. Finally, the pop-up menu returned by the *TViewForm.ViewMenu* property is merged into the main form's main menu. I don't use the standard merging capabilities of *TMainMenu*, because I'm going to have multiple levels of merging when I extend this framework in my next article.

As you examine Listing Three, you'll notice another class, *TViewManager*. This class represents another pattern, which I'll discuss in my next article. For now, just know that this is the class with which the main form interacts to reference the *TViewForm* descendants. *TViewManager* is responsible for creating, freeing, and managing the *TViewForm* descendants. I'll discuss *TViewManager*'s functionality in the context of the main form interacting with *TViewForms*.

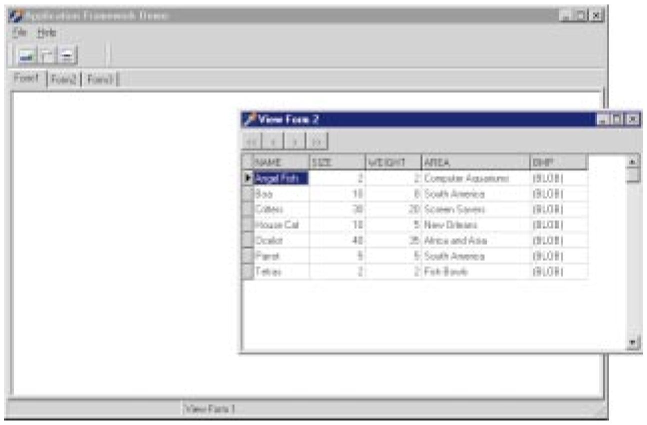
When the user attempts to change a tab on the *TTabControl*, the *OnChanging* event handler for the *TTabControl* is invoked, and the *AllowChange* parameter is set to the value of the currently loaded *TViewForm.CanChange* method. You can see how you might prevent a user from changing tabs and the current *TViewForm*. If the change is valid, *tctrlMainChange*, *TTabControl.OnChange*, is invoked. This method performs several steps. First, it un-merges the current *TViewForm*'s menu. Then, it retrieves the next *TViewForm* and merges its menu and toolbar with the main form. The reason I don't have an *UnMergeToolBar* method is because the *TViewForm*'s toolbar is still owned by *TViewForm*. In the process of freeing the *TViewForm*, its *TToolBar* is also freed. Figures 6, 7, and 8 show the main form with the three *TViewForm* descendants contained in *pnlContainer*.

The main form retrieves the *TViewForm* descendants using one of two methods. One is illustrated in the *tctrlMainChange* method in the call to the *TViewManager.GetViewForm* methods shown below:

```
FViewManager.GetViewForm(tctrlMain.TabIndex, pnlContainer);
```

TViewManager.GetViewForm is an overloaded method that creates a *TViewForm* instance and assigns the second parameter, in this case *pnlContainer*, as the parent. *TViewManager* does this by calling the appropriate constructor of *TViewForm*. The *TViewManager* method is called in *TMainForm.ShowViewFormModal*:

```
ViewForm := FViewManager.GetViewForm(ViewIndex);
```



	NAME	SIZE	WEIGHT	AREA	BMP
▶	Angel Fish	2	2	Computer Aquariums	(BLOB)
	Boa	10	8	South America	(BLOB)
	Critters	30	20	Screen Savers	(BLOB)
	House Cat	10	5	New Orleans	(BLOB)
	Ocelot	40	35	Africa and Asia	(BLOB)
	Parrot	5	5	South America	(BLOB)
	Tetras	2	2	Fish Bowls	(BLOB)

Figure 9: Modal *TViewForm2*.

This method returns a reference to a *TViewForm*. *ShowViewFormModal* demonstrates how you can show the same form previously embedded in a *TPanel* as a separate modal form (see Figure 9). Finally, notice that the main form's *OnCreate* event handler invokes the *tctrlMainChange* method to load the first *TViewForm*.

Conclusion

The Template Method pattern is really an "OOP Patterns" term for something most Delphi programmers have been doing since Delphi 1. In this article, I discussed the Template Method pattern and started on the initial design of an application framework. I'll use this framework in my next article to discuss another useful and commonly used pattern, the Builder pattern. I'll also illustrate how to extend this framework to embed fully functional, run-time modules by using add-in packages. In closing, I'd like to thank David Streever and Anne Pacheco for their technical and grammatical review of this article. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\MAY\DI9905XP.

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

Begin Listing One — *TViewForm*

```
unit xwViewFrm;

interface

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

type
  { TViewForm serves as the abstract class for Views within
    the ShellApp framework. }
  TViewForm = class(TForm)
  private
    FAsChild: Boolean;   // Form created as child indicator.
    FTempParent: TWinControl;   // Temporary parent window.
  protected
    procedure CreateParams(var Params: TCreateParams);
      override;
    procedure Loaded; override;
    function GetViewDescription: string; virtual;
    function GetViewMenu: TPopupMenu; virtual;
    function GetViewToolBar: TToolBar; virtual;
  public
    // Constructor to create as a normal form.
    constructor Create(AOwner: TComponent); overload;
      override;
    // Constructor to create as a child form.
    constructor Create(AOwner: TComponent;
      AParent: TWinControl); reintroduce; overload;
    function CanChange: Boolean; virtual; abstract;
    property ViewDescription: string
      read GetViewDescription;
    property ViewMenu: TPopupMenu read GetViewMenu;
    property ViewToolbar: TToolBar read GetViewToolBar;
  end;

implementation

{$R *.DFM}

{ TviewForm. }
constructor TViewForm.Create(AOwner: TComponent);
begin
  FAsChild := False;
  inherited Create(AOwner);
end;

constructor TViewForm.Create(AOwner: TComponent;
  AParent: TWinControl);
begin
  FAsChild := True;
  FTempParent := aParent;
  inherited Create(AOwner);
end;

procedure TViewForm.Loaded;
begin
  inherited;
  if FAsChild then begin
    align := alClient;
    BorderStyle := bsNone;
    BorderIcons := [];
    Parent := FTempParent;
    Position := poDefault;
  end;
end;

procedure TViewForm.CreateParams(
  var Params: TCreateParams);
begin
  Inherited CreateParams(Params);
  if FAsChild then
    Params.Style := Params.Style or WS_CHILD;
end;

function TViewForm.GetViewMenu: TPopupMenu;
begin
  Result := nil;
end;

function TViewForm.GetViewToolBar: TToolBar;
begin
  Result := nil;
end;

function TViewForm.GetViewDescription: string;
begin
  Result := EmptyStr;
end;

end.
```

End Listing One

Begin Listing Two — *TViewForm2*

```
unit ViewFrm2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, xwViewFrm, ComCtrls, ToolWin, Grids,
  DBGrids, Db, DBTables, Menus;

type
  TViewForm2 = class(TViewForm)
    Table1: TTable;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    ToolBar1: TToolBar;
    ToolButton1: TToolButton;
    ToolButton2: TToolButton;
    ToolButton3: TToolButton;
    ToolButton4: TToolButton;
    PopupMenu1: TPopupMenu;
    InsertRecord1: TMenuItem;
    EditRecord1: TMenuItem;
    DeleteRecord1: TMenuItem;
    procedure ToolButton1Click(Sender: TObject);
    procedure ToolButton2Click(Sender: TObject);
    procedure ToolButton3Click(Sender: TObject);
    procedure ToolButton4Click(Sender: TObject);
    procedure PopupMenuClick(Sender: TObject);
  protected
    function GetViewToolBar: TToolBar; override;
    function GetViewMenu: TPopupMenu; override;
    function GetViewDescription: string; override;
  public
    function CanChange: Boolean; override;
  end;

var
  ViewForm2: TViewForm2;

implementation

{$R *.DFM}

{ TViewForm2. }
function TViewForm2.CanChange: Boolean;
begin
  if Table1.State <> dsBrowse then
    ShowMessage(
      'Cannot change pages until you save the record');
  Result := Table1.State = dsBrowse;
end;

function TViewForm2.GetViewDescription: string;
begin
  Result := 'View Form 2';
end;

function TViewForm2.GetViewMenu: TPopupMenu;
begin
  Result := PopupMenu1;
end;

function TViewForm2.GetViewToolBar: TToolBar;
begin
  Result := ToolBar1;
end;

procedure TViewForm2.ToolButton1Click(Sender: TObject);
begin
  inherited;
  Table1.First;
end;

procedure TViewForm2.ToolButton2Click(Sender: TObject);
begin
  inherited;
  Table1.Prior;
end;

procedure TViewForm2.ToolButton3Click(Sender: TObject);
begin
  inherited;
  Table1.Next;
end;

procedure TViewForm2.ToolButton4Click(Sender: TObject);
begin
  inherited;
  Table1.Last;
end;

procedure TViewForm2.PopupMenuClick(Sender: TObject);
begin
  inherited;
  ShowMessage((Sender as TMenuItem).Caption);
end;

end.
```

End Listing Two

Begin Listing Three — The Shell Application

```
unit MainFrm;

interface

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

type
  TViewManager = class(TObject)
  private
    FCurrentView: TViewForm;
  public
    constructor Create;
    destructor Destroy; override;
    { GetViewForm retrieves the View, giving it a parent
      window into which it embeds itself. }
    function GetViewForm(const ViewIndex: Integer;
      AParent: TWinControl): Boolean; overload;
    { GetViewForm retrieves a specified view by index. This
      method retrieves a reference to a view instance which
      does not embed itself. }
    function GetViewForm(const ViewIndex: Integer):
      TViewForm; overload;
    { CloseCurrentViewForm closes and destroys the current
      embedded view form. }
    procedure CloseCurrentViewForm;
    { MergeViewToolBar merges the current views toolbar
      with the AViewToolBarParent parameter. }
    procedure MergeViewToolBar(
      AViewToolbarParent: TWinControl);
    { MergeViewMenu merges the current views menu with the
      main menu specified by AAddMenu. AMainMenu is used as
      the owner of the newly created menu. }
    procedure MergeViewMenu(AMainMenu: TMainMenu;
      AAddMenu: TMenuItem);
    { UnmergeViewMenu unmerges and destroys the menu added
      by the ViewMenu. }
    procedure UnmergeViewMenu(AMenuItem: TMenuItem);
  end;

  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    stbrMain: TStatusBar;
    tctrlMain: TTabControl;
    pnlContainer: TPanel;
    CoolBar1: TCoolBar;
    ToolBar1: TToolBar;
    ToolButton1: TToolButton;
    ToolButton2: TToolButton;
    ToolButton3: TToolButton;
    pnlViewTB: TPanel;
    mmiView: TMenuItem;
    mmiHelp: TMenuItem;
    mmiAbout: TMenuItem;
    ilMain: TImageList;
    procedure mmiExitClick(Sender: TObject);
    procedure tctrlMainChanging(Sender: TObject;
      var AllowChange: Boolean);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure tctrlMainChange(Sender: TObject);
    procedure ToolButton1Click(Sender: TObject);
    procedure ToolButton2Click(Sender: TObject);
    procedure ToolButton3Click(Sender: TObject);
  private
    FViewManager: TViewManager;
    procedure ShowViewFormModal(const ViewIndex: Integer);
  end;

var
  MainForm: TMainForm;

implementation

uses
  ViewFrm1, ViewFrm2, ViewFrm3;

{$R *.DFM}

{ TViewManager. }
procedure TViewManager.CloseCurrentViewForm;
begin
  if FCurrentView <> nil then begin
    FCurrentView.Free;
    FCurrentView := nil;
  end
end;

constructor TViewManager.Create;
begin
  inherited Create;
  FCurrentView := nil;
end;

destructor TViewManager.Destroy;
begin
  if Assigned(FCurrentView) then
    FCurrentView.Free;
  inherited Destroy;
end;

function TViewManager.GetViewForm(const ViewIndex: Integer;
  AParent: TWinControl): Boolean;
begin
  CloseCurrentViewForm;
  case ViewIndex of
    0: FCurrentView := TViewForm1.Create(nil, AParent);
    1: FCurrentView := TViewForm2.Create(nil, AParent);
    2: FCurrentView := TViewForm3.Create(nil, AParent);
  end;
  FCurrentView.Show;
  Result := True;
end;

function TViewManager.GetViewForm(
  const ViewIndex: Integer): TViewForm;
begin
  Result := nil;  // Default.
  case ViewIndex of
    0: Result := TViewForm1.Create(Application);
    1: Result := TViewForm2.Create(Application);
    2: Result := TViewForm3.Create(Application);
  end;
end;

procedure TViewManager.MergeViewMenu(AMainMenu: TMainMenu;
  AAddMenu: TMenuItem);
var
  MenuItem: TMenuItem;
  i: Integer;
begin
  if FCurrentView <> nil then
    if FCurrentView.ViewMenu <> nil then begin
      for i := 0 to FCurrentView.ViewMenu.Items.Count-1 do
        begin
          MenuItem := TMenuItem.Create(AMainMenu);
          MenuItem.Caption :=
            FCurrentView.ViewMenu.Items[i].Caption;
          MenuItem.OnClick :=
            FCurrentView.ViewMenu.Items[i].OnClick;
          AAddMenu.Add(MenuItem);
        end;
      AAddMenu.Visible := True;
    end;
end;

procedure TViewManager.UnmergeViewMenu(
  AMenuItem: TMenuItem);
var
  i: Integer;
begin
  for i := AMenuItem.Count - 1 downto 0 do
    AMenuItem.Delete(i);
  AMenuItem.Visible := False;
end;

procedure TViewManager.MergeViewToolBar(
  AViewToolbarParent: TWinControl);
begin
  if FCurrentView.ViewToolbar <> nil then begin
    FCurrentView.ViewToolBar.EdgeBorders := [];
    FCurrentView.ViewToolbar.Parent := AViewToolbarParent;
  end;
end;

{ TMainForm. }
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FViewManager := TViewManager.Create;
  tctrlMainChange(nil);
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  FViewManager.Free;
  FViewManager := nil;
end;

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

procedure TMainForm.tctrlMainChanging(Sender: TObject;
  var AllowChange: Boolean);
begin
  AllowChange := FViewManager.FCurrentView.CanChange;
end;

procedure TMainForm.tctrlMainChange(Sender: TObject);
begin
  FViewManager.UnmergeViewMenu(mmiView);
  FViewManager.GetViewForm(tctrlMain.TabIndex,
                           pnlContainer);
  FViewManager.MergeViewToolBar(pnlViewTB);
  FViewManager.MergeViewMenu(mmMain, mmiView);
  stbrMain.Panels[1].Text :=
    FViewManager.FCurrentView.ViewDescription;
end;

procedure TMainForm.ShowViewFormModal(
  const ViewIndex: Integer);
```

```
var
  ViewForm: TViewForm;
begin
  ViewForm := FViewManager.GetViewForm(ViewIndex);
  try
    ViewForm.ShowModal;
  finally
    ViewForm.Free;
  end;
end;

procedure TMainForm.ToolButton1Click(Sender: TObject);
begin
  ShowViewFormModal(0);
end;

procedure TMainForm.ToolButton2Click(Sender: TObject);
begin
  ShowViewFormModal(1);
end;

procedure TMainForm.ToolButton3Click(Sender: TObject);
begin
  ShowViewFormModal(2);
end;

end.
```

End Listing Three

By Kevin J. Bluck and James Holderness

From the Shell

Part II: More Undocumented Shell Dialog Boxes

Last month, we looked at several useful dialog boxes whose functions are not provided in Comdlg.dll, nor are they clearly documented. Instead of trying to duplicate interfaces by building a dialog box manually, we showed how to access several commonplace system dialog boxes you may need, but are difficult to find. To close out this two-part series, we'll show you several more functions and how to use them.

A sample program accompanies this series and demonstrates each dialog box function (see Figure 1). The first article covered five dialog boxes: Browse for Folder, About Windows, Format, Change Icon, and Run. This month we tackle the rest.

Finding Files

Many people want the Find dialog box. It's the handy utility provided by the shell to find files based on a variety of criteria (see Figure 2). The ability to spawn this dialog box from your own application is provided by the *SHFindFiles* function. It's exported from Shell32.dll, and the ordinal value is 90:

```
function SHFindFiles(SearchRoot: PItemIDList;
  SavedSearchFile: PItemIDList): LongBool;
  stdcall;
```

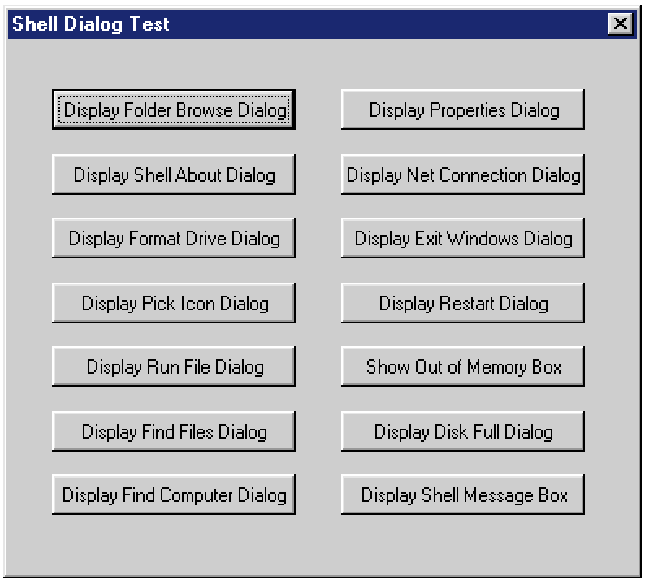
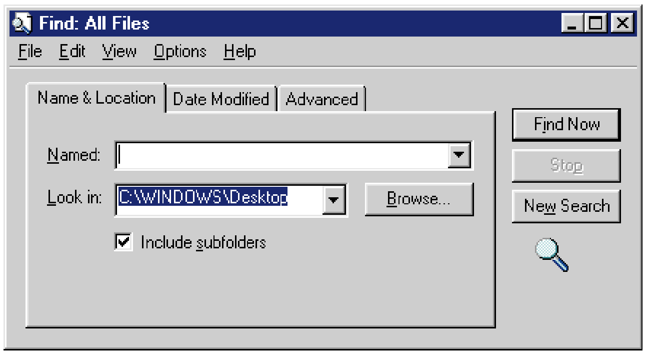

Shell Dialog Test
Display Folder Browse Dialog
Display Properties Dialog
Display Shell About Dialog
Display Net Connection Dialog
Display Format Drive Dialog
Display Exit Windows Dialog
Display Pick Icon Dialog
Display Restart Dialog
Display Run File Dialog
Show Out of Memory Box
Display Find Files Dialog
Display Disk Full Dialog
Display Find Computer Dialog
Display Shell Message Box

Figure 1: The demonstration program.

The *SearchRoot* parameter allows you to begin a search in a particular folder. This is the same effect you get if you select **Find** on the context menu of a folder you've right-clicked. You may pass this parameter as **nil** to allow the user to begin searching at the Desktop. The *SavedSearchFile* parameter allows you to specify a file saved from a previous search (a .FND file) that will initialize the dialog box to match the saved state. This is the effect you would get from opening a .FND file in the Explorer. You may also pass this parameter as **nil**. Passing both parameters as **nil** produces the dialog box you would get from selecting **Find | Files or Folders** from the **Start** menu.

If you specify a non-**nil** *SearchRoot* PIDL, it's your responsibility to free that PIDL after calling *SHFindFiles*. However, if you pass a non-**nil** *SavedSearchFile* PIDL, you mustn't try to free that PIDL if the function succeeds, as an error will occur if you do. We can only hope the shell will free it when it's done doing whatever it does with it. If the function fails, however, you must free the PIDL yourself. [For a description of PIDLs and their use, see "Shell Notifications" by Kevin J. Bluck and James Holderness in the March, 1999 *Delphi Informant*.]

Unlike most dialog box functions, this function is non-modal. Instead, it starts the dialog box in a separate thread, then returns immediately. The return value will be True if the dialog box was successfully spawned, False if there was an error. If the user doesn't explicitly close the dialog box, it will automatically close when your process terminates. Keep in mind that you have no direct way of telling what the user does with the dialog box. The best way for your application to become aware of files the user eventually finds is to sup-



Named:

Look in: C:\WINDOWS\Desktop

Include subfolders [✓]

Figure 2: The Find dialog box.

port OLE drag-and-drop, so the user can drag found files from the dialog box into your application.

Finding Computers

Another function closely related to *SHFindFiles* is *SHFindComputer*. This function shows the same dialog box you would get if you clicked the **Start** menu and selected **Find | Compute**. Its interface is identical to *SHFindFiles*, except it completely ignores the parameters you send it. Apparently, they have been reserved for future expansion. Just pass **nil** to both parameters. The return values are identical to *SHFindFiles*, and like that function, the dialog box is non-modal. So the function returns immediately while the dialog box remains open, and there is no direct means of telling what the user did with the dialog box. *SHFindComputer* is exported from Shell32.dll by the ordinal value of 91:

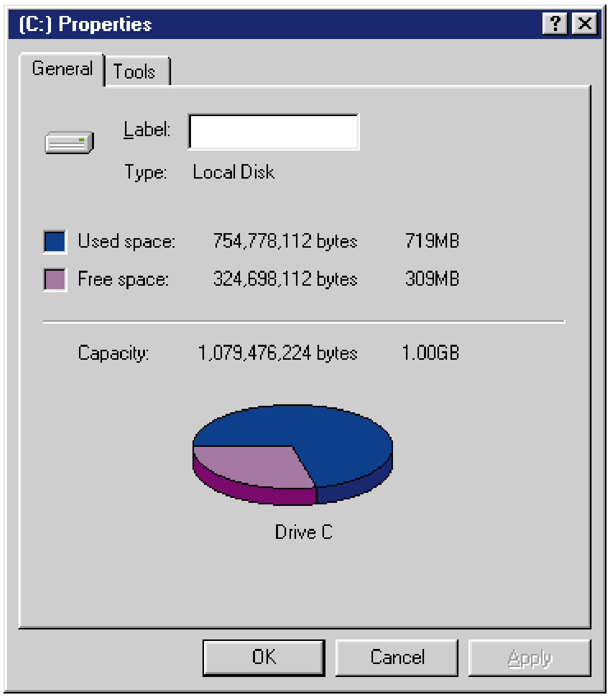
```
function SHFindComputer(Reserved1: PItemIDList;
  Reserved2: PItemIDList): LongBool; stdcall;
```

Browsing for Files

There really isn't any compelling reason why you need to use this next function. *GetFileNameFromBrowse* is nothing more than a simplified wrapper around the *GetOpenFileName* function, which is the function you call when you want to display the standard Open dialog box. Obviously, you already have access to everything this function can do by using the standard VCL *TOpenDialog* component or the *GetOpenFileName* API function. However, for some applications, it might be nice to be able to browse for a file with a single function call without the tedious process of filling in all the members of the OPENFILENAME structure, or instantiating an instance of *OpenDialog*. The function is exported from Shell32.dll by ordinal value 63:

```
function GetFileNameFromBrowse(Owner: HWND;
  FileName: Pointer; MaxFileNameChars: DWORD;
  InitialDirectory: Pointer; DefaultExtension: Pointer;
  Filter: Pointer; Caption: Pointer): LongBool; stdcall;
```

Most of the parameters to this function correspond directly with members of the OPENFILENAME structure. The *Owner* parameter identifies the window that owns the dialog box. The *FileName* parameter points to a buffer that contains a file name used to initialize the dialog box's Edit control. When the function returns, this buffer contains the full path of the selected file. It's advisable to provide a buffer capable of storing MAX_PATH characters plus a null terminator. The *MaxFileNameChars* parameter specifies the size, in characters, of the buffer pointed to by the *FileName* parameter. The *InitialDirectory* parameter points to a string that specifies the initial file directory when the dialog box appears. If the *FileName* parameter contains a fully qualified file name with path, the *InitialDirectory* parameter is



Label:

Type: Local Disk

🟦	Used space:	754,778,112 bytes	719MB
🟪	Free space:	324,698,112 bytes	309MB
	Capacity:	1,079,476,224 bytes	1.00GB

Figure 3: The Properties dialog box — in this case for a drive.

ignored, and the path from the *FileName* parameter is used instead. The *DefaultExtension* parameter points to a buffer containing the default extension the dialog box will search for. The *Filter* parameter points to a buffer containing pairs of null-terminated filter strings that will be shown in the **Files of Type** drop-down list. The *Caption* parameter points to a string to be shown in the title bar of the dialog box. For further details on all these parameters, see the documentation on the Windows OPENFILENAME data structure.

If the user selects a file to open, the return value is True. It's False if an error occurs, the user chooses the **Cancel** button, or the user chooses the **Close** command from the System menu.

Displaying Object Properties

Another handy undocumented dialog box function is *SHObjectProperties*. This function can be used to display the Properties dialog box for a drive, folder, or file (see Figure 3). It can also be used to display the properties for a printer object. The function is exported from Shell32.dll by ordinal value 178:

```
function SHObjectProperties(Owner: HWND; Flags: UINT;
  ObjectName: Pointer; InitialTabName: Pointer):
  LongBool; stdcall;
```

The *Owner* parameter identifies the window that owns the dialog box. The *Flags* parameter specifies the type of object whose name is passed in the *ObjectName* parameter. These are the possible flag values:

```
OPF_PRINTERNAME = $01;
OPF_PATHNAME    = $02;
```

The *ObjectName* parameter points to a string containing the path name, or the printer name whose properties will be displayed. If a printer is local, you may use only the actual printer name. If a printer is from the network, you need to use the entire UNC-style name, in the form \\COMPUTERNAME\PRINTERNAME. The *InitialTabName* parameter points to a string containing the name of

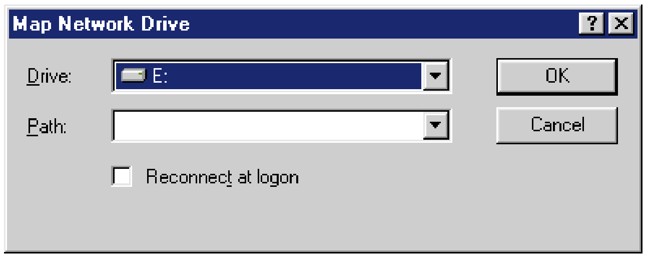


Figure 4: The Map Network Drive dialog box.

the tabbed page that will initially be shown in the dialog box. If the *InitialTabName* parameter is **nil**, or the string doesn't match the name of any tab, the first tab on the property sheet will be selected.

If the function succeeds, the return value is True. If the function fails, the return value is False. To get extended error information, call the API function *GetLastError*. Note that this dialog box is non-modal, similar to the *SHFindFiles* dialog box, so when the function returns, the dialog box will almost certainly still be open. There is no way of knowing when the user has closed the dialog box.

Networking

The next two functions allow your user to connect to network resources. *SHNetConnectionDialog* (see Figure 4) is available on Windows 95 and NT, and is exported from Shell32.dll by ordinal 160:

```
function SHNetConnectionDialog(Owner: HWND;
  ResourceName: Pointer; ResourceType: DWORD):
  DWORD; stdcall;
```

SHStartNetConnectionDialog is available only on NT. It shows the same dialog box as *SHNetConnectionDialog*, but starts it non-modally in another thread and returns immediately. This function is exported from Shell32.dll only on NT by ordinal value 215:

```
function SHStartNetConnectionDialog(Owner: HWND;
  ResourceName: PWideChar; ResourceType: DWORD):
  DWORD; stdcall;
```

The parameter lists are basically identical. The *Owner* parameter takes the handle of the window that will own the dialog box. The *ResourceName* parameter points to a null-terminated string specifying the fully qualified UNC path of the network resource to connect to. Specifying this parameter results in a dialog box that is "pre-set" to the named resource and doesn't allow the user to change the resource. If you pass **nil** to this parameter, the dialog box allows the user to specify the resource.

The *ResourceType* parameter can be set to one of two values: RESOURCETYPE_DISK or RESOURCETYPE_PRINT. These values will produce different dialog boxes. The first allows you to assign a drive letter to a network disk resource, while the second allows you to map a parallel port name, such as LPT2, to a network printer. However, for some reason, RESOURCETYPE_PRINT doesn't work on NT. If you pass this value on NT, the function fails. There are also some other constants in the RESOURCETYPE_XXX family, but none of the others work for this function on any platform.

If the function succeeds, the return value is NO_ERROR. If the user cancels the dialog box, it returns –1 ($FFFFFFFF). If the function fails, the return value is some other error code. To get more detailed error information, call the *GetLastError* API function.

Shutting Down the System

The next two functions, *ExitWindowsDialog* and *RestartDialog*, deal with the problem of shutting down and restarting the operating system. They may seem out of place in this article because they're not really much more than extensions of the *ExitWindowsEx* API function, but they both produce dialog boxes as part of the process. Both functions are exported from Shell32.dll. The ordinal export value for *ExitWindowsDialog* is 60, and the ordinal for *RestartDialog* is 59. These function declarations are shown in:

```
procedure ExitWindowsDialog(Owner: HWND); stdcall;
```

and:

```
function RestartDialog(Owner: HWND; Reason: Pointer;
  ExitType: UINT): DWORD; stdcall;
```

ExitWindowsDialog is probably the less useful of the two. It's the dialog box displayed when you select Shut Down from the Start menu. The dialog box doesn't always seem to actually use *Owner* as a parent. On Windows 95, the owner window will receive a WM_QUIT message if the operation is successful. On Windows NT, the owner window doesn't appear to be used at all. There is no return value for the function, so you have no way of knowing what the user selected, or whether the operation was canceled. Presumably, your application will be receiving shutdown messages from Windows fairly soon if the user decided to quit, but there is no way to know for sure at the time of the function call.

RestartDialog is used when changes are made to the system that require a shutdown or restart before they can take effect. The *Owner* parameter identifies the window that will own the dialog box. The *Reason* parameter points to a string that is displayed in the dialog box, explaining the reason for the shutdown. The *ExitType* parameter specifies the type of shutdown that will be performed if the user selects the Yes button. You can use a subset of the values used by *ExitWindowsEx* in addition to a few new values. The following is the complete list:

```
EWX_LOGOFF          = $00;
EWX_SHUTDOWN        = $01;
EWX_REBOOT          = $02;
EW_RESTARTWINDOWS   = $42;
EW_REBOOTSYSTEM     = $43;
EW_EXITANDEXECAPP   = $44;
```

The return value is IDYES if the user chose to perform the shutdown. It is IDNO if the operation was canceled.

There are a couple of other points about *RestartDialog* you should note. The reason displayed in the dialog box always has some default text appended to it asking the user to confirm the operation. It's therefore advisable you always end your reason text string with a space, or a CR/LF. The title of the dialog box is always set to "System Settings Change." Finally, the return value cannot be used to determine the success of the operation. It only signifies the choice made by the user. If the restart operation failed for some reason, the return value will still be IDYES.

Note that for either of these functions to work correctly, users must have the SE_SHUTDOWN_NAME privilege enabled in their profile. This is usually not an issue in Windows 95, but some installations of NT are set up to prevent certain users from shutting down the system.

This is particularly common on server machines, which would deny other users needed services if shut down at the wrong time. This can be an insidious bug, because developer accounts typically have "Local Administrator" privileges and can shut down the system, but when a user tries the same thing, the privilege is unavailable. Be sure to test your application using typical user accounts before release.

Out of Memory!

Here's an undocumented dialog box function of dubious value, but we mention it anyway for the sake of completeness. *SHOutOfMemoryMessageBox* is the standard shell dialog box used when the system is low on memory. It's exported from Shell32.dll by the ordinal value of 126:

```
function SHOutOfMemoryMessageBox(Owner: HWND;
  Caption: Pointer; Style: UINT): Integer; stdcall;
```

It makes a call to the standard Windows API function *MessageBox*, passing its three parameters along with the standard system error message ERROR_OUTOFMEMORY, i.e. "Not enough storage is available to complete this operation," or the local language equivalent.

The *Owner* parameter specifies the parent window for the dialog box. The *Caption* parameter points to a null-terminated string used for the dialog box title. If *Caption* is **nil**, the title of the parent window is used instead. The *Style* parameter can be set to any combination of the MB_XXX constants used by the *MessageBox* function, but it's typically set to `(MB_OK or MB_ICONHAND)`. The return values are identical to those for *MessageBox*. Check the documentation for that function if you would like full details on the MB_XXX constants and the return value.

When the actual *MessageBox* call is made, MB_SETFOREGROUND is added to the *Style* flags, but if that first call fails, a second *MessageBox* call is made, this time with MB_SYSTEMMODAL added to the *Style* flags. MB_SYSTEMMODAL, in combination with MB_ICONHAND, should cause the message box to display regardless of available memory. Theoretically, anyway; in practice, we've observed a bug in the function that prevents the second call from ever being made. In the event the system really is out of memory, this function will likely be incapable of displaying anything. However, it still returns the result of the *MessageBox* call, so you should be able to tell when the function has failed by checking for a return value of zero.

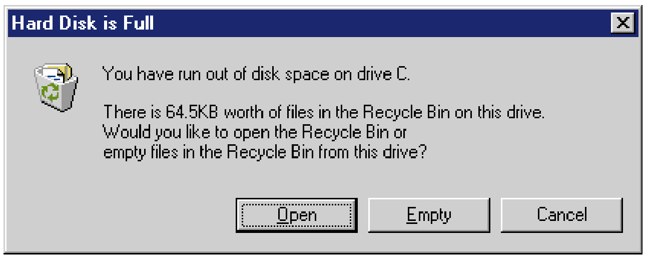
Out of Space!

Another resource-oriented function is *SHHandleDiskFull*. Its name is a bit of hyperbole in our opinion. It certainly can't handle a full disk all by itself. It can provide a useful tool, however, to deal with users who never empty their Recycle Bins (see Figure 5). This function is generally called by an application in response to a disk operation that is failing because of insufficient free disk space. When called, if the user has anything in that disk's Recycle Bin, this function gives the user the opportunity to empty their Recycle Bin, hopefully freeing up a significant amount of disk space. If there is no Recycle Bin on the specified drive, or the Recycle Bin is already empty, this function does absolutely nothing. This function is exported from Shell32.dll by ordinal value 185:

```
procedure SHHandleDiskFull(Owner: HWND;
  Drive: UINT); stdcall;
```

The *Owner* parameter identifies the dialog box's owner window. It seems to make no difference if you pass 0 to this parameter. The *Drive* parameter specifies the zero-based number of the drive that is running out of space. This is the same scheme used in *SHFormatDrive*, where 0 = A:, 1 = B:, and so on.

The thing that concerns us about this function is that it's not clear how one should use it. It really can't be used as a standard error dialog box whenever a disk runs out of space, because the dialog box won't show at all when there is nothing in the Recycle Bin, and there is no return value to provide information about whether the dialog box was ever shown. It's also difficult to know whether the application can immediately retry the operation after this procedure returns, because there's no direct way to know if the user has freed any space. The user may have chosen to merely open the Recycle Bin, or may have done nothing. It seems the application will have to monitor disk free space on its own, and merely use this dialog box as an attempt at a "quick fix" second chance before resorting to failing the operation.


Hard Disk is Full

You have run out of disk space on drive C.

There is 64.5KB worth of files in the Recycle Bin on this drive.
Would you like to open the Recycle Bin or
empty files in the Recycle Bin from this drive?

Figure 5: The Hard Disk is Full dialog box.

Generic Shell Message Boxes

The last set of functions we'll cover is the family of generic message dialog box functions provided by the shell. *ShellMessageBox* is just a wrapper around the Windows API function *MessageBox* that allows you to use either string resource identifiers, or standard null-terminated strings, as well as allowing additional inclusion strings in the message string in a manner similar to the Windows API *FormatMessage* function. *ShellMessageBox* is exported from Shell32.dll by ordinal 183:

```
function ShellMessageBoxA(Module: THandle; Owner: HWND;
  Text: PChar; Caption: PChar; Style: UINT;
  Parameters: array of Pointer): Integer; cdecl;
```

Technically, this function is called *ShellMessageBoxA*, as it is an ANSI-only variant, even on NT. There is also a UNICODE variant called *ShellMessageBoxW*, which is exported by ordinal 182, but this variant is available only on NT:

```
function ShellMessageBoxW(Module: THandle; Owner: HWND;
  Text: PWideChar; Caption: PWideChar; Style: UINT;
  Parameters: array of Pointer): Integer; cdecl;
```

The *Module* parameter takes the handle of the module that provides the string resources for the dialog box. You should use the *GetModuleHandle* Windows API function to retrieve that handle. The *Owner* parameter is the usual handle to the owner window. The *Text* parameter points to a null-terminated string containing the text you would like displayed in the dialog box. It may alternatively be the resource ID of a string resource contained in the module identified by the *Module* parameter. This text may include "escape sequences," which the function will replace with the additional text parameters passed in the *Parameters* parameter in the same manner provided by the API function *FormatMessage*. These escape sequences take the form "%#", where "#" is the ordinal position of the extra string parameter in question. For example, "%1" will be replaced by the first string in the

Parameters open array, "%3" will be replaced by the third, and so on. The *Caption* parameter points to a null-terminated string that specifies the text shown in the dialog box title bar. Again, a resource ID may be used instead of a string pointer. If this parameter is left **nil**, the caption of the window specified in the *Owner* parameter is used instead. The *Style* parameter is a bit-mask of flags, the same ones used in the Windows API *MessageBox* function. This parameter can be set to any combination of the MB_XXX constants used by the *MessageBox* function. The return value is also identical to that of the *MessageBox* API function. Check the documentation for that function if you would like full details on the MB_XXX constants and the return value.

As for the *Parameters* parameter, alert readers have probably already noticed that the Microsofties have done something very naughty. They've exported a function using the **cdecl** calling convention instead of the standard **stdcall**. Plus, to add insult to injury, they've made use of C language-specific variable parameter lists! This was quite lazy on their part. *FormatMessage* shows they know how to do the same thing in a more language-independent fashion, by passing an array of 32-bit values that reference values to insert in the formatted message. This **cdecl** situation, of course, makes it rather difficult to translate these functions to Delphi, because Delphi doesn't directly support variable parameter lists. Well, that's what we get for messing with undocumented functions. To deal with this problem, the *Parameters* parameter is typed as an open array of *Pointer*. An open array parameter is the closest simulation available in Delphi to the concept of variable parameter lists in C. Interested readers are welcome to examine the inline assembly code required to set up the **cdecl** call stack correctly in the source code included with this article (see end of article for download details). At any rate, this is where pointers to the "extra" strings required to replace any escape codes are provided. Note that because of the mechanics of open arrays, you must specify at least one pointer value here. If you have no escape sequences to replace, and therefore no additional strings to pass, simply stick one **nil** value into the brackets.

Componentization

In general, wrapping a component around a system dialog box is one of the simplest tasks in component development. The only dialog box component with any real complexity to it in this project is the one encapsulating *SHBrowseForFolder*. *RunFileDlg* is slightly complex, as it incorporates a notification message scheme. All the rest are quite straightforward. In general, they consist of little more than a constructor to initialize data, some properties that correspond directly to the parameters expected by the underlying API function, and an *Execute* method to invoke the dialog box at run time. Some of them, such as *ExitWindowsDialog*, are so simple that they aren't even worth a component. We'll just mention a few general design principles common to all these dialog box components, rather than bore you with the excruciating details of their rather simple and repetitive implementations.

The efficiency-minded reader may question why we bothered overriding the constructors for many of these components. The compiler automatically zero-initializes all *TObject* descendants' data storage upon creation, so there is no real reason to do so in a constructor; that we have done so in many cases is unnecessary. We believe, however, that the gain in code readability and ability to see the exact expected default values are well worth the trivial amount of extra code.

We have converted all flag types to enumerated and set types. This serves two purposes: It enables rigorous type-checking by the compiler, and it prevents the user from having to remember "magic numbers" for use in the design-time editor. To avoid translating these enumerated types back and forth to the corresponding Windows constants in the middle of other code, we've implemented "translation" functions that handle this task. This ensures that translation is done only in one easily identifiable place in the unit.

The common denominator for all dialog-wrapping components is the *Execute* method. In every case, the meat of the API functionality is in that method. All the other code associated with the component is strictly in a supporting role, implementing the mechanics of the Delphi component paradigm. If you want to see a practical example of how a particular API call is used, check the corresponding component's *Execute* method in the code accompanying this article (again, see end of article for download details).

We've added a few stand-alone functions to invoke the simpler dialog boxes, or to invoke default instances of the more complicated ones. Sometimes, a developer just wants to call a single function for a general dialog box rather than monkey with a component. If you want to do any significant customization, however, you should use the component. Of course, you're also welcome to call the API functions directly, if you like.

Browsing Magic

By far, the most complex of these dialog box components is *TkbBrowseForFolderDialog* (discussed last month). The *SHBrowseForFolder* API function has a complicated initialization process and a callback mechanism to be implemented. Here, we present the highlights of encapsulating this function.

The most difficult aspect of this function for the uninitiated is that it deals with the dreaded PIDL, formally known in Delphi as the *PItemIDList* record type, defined in the VCL unit *ShlObj*. Basically, a PIDL is the shell version of the DOS path. Every file system object can be represented either as a PIDL or a path. In addition, many non-file system objects also exist, such as Control Panel, which can't be identified by anything but a PIDL. Because the *SHBrowseForFolder* dialog box allows you to select folders that are not part of the file system, it uses PIDLs for input and output. For the purposes of this article, you may consider a PIDL to be a pointer supplied by the shell that points to arbitrary data that should not be modified in any way. Functions have been provided in the unit *kbsdPIDL* that will convert a file system path to a PIDL and vice versa, assuming the PIDL in question points to a file system object. The most unusual aspect about PIDLs is that you should never free a PIDL using the usual RTL functions like *FreeMem*; you must use only the *FreePIDL* function provided in *kbsnPIDL*.

This leads directly into the problem of specifying the Browse dialog box's root folder. It's possible to limit the scope of the user's browse by specifying a root folder lower in the hierarchy than Desktop. For example, if you specify the C:\ folder, the user will be unable to browse anything that isn't part of the C: drive. The tricky part is that this root folder must be specified as a PIDL. Furthermore, the root folder isn't necessarily a file system path. How do you allow the user to specify both file and non-file root folders in the design-time editor?

The solution is to use two properties. One property is a new enumerated type, *TkbsdSpecialLocation*, which encapsulates the list of Windows API constants that correspond to various "special" folders, such as Control Panel. These constants can be used with the *SHGetSpecialFolderLocation* API function to obtain a PIDL to that folder. By setting a property to one of these enumerated values, special non-file folders can be specified without requiring the developer to type in the data for that folder's PIDL. One of the values of *TkbsdSpecialLocation* is *kbsdPath*. Setting this value will enable a second property to allow the developer to enter a specific file system path to use as the root folder. You can see the implementation of this scheme in the *Execute* method fragment shown in Figure 6.

Once this root folder problem is solved, the remaining initialization is straightforward. We simply go through the required list of parameters in *TBrowseInfo*, and translate the corresponding component property values into each data member. Once *TBrowseInfo* is ready, we call the function.

While the dialog box is displayed, it's issuing calls to a callback function that we must implement. Our implementation is shown in Figure 7. The sole purpose of this callback is to invoke the component's event dispatch methods, passing the necessary data to each. A couple of items are worthy of mention. We've used the "user-defined" parameter of the callback to pass a pointer to our own component. This is important, because the callback is not an object member function, and therefore has no *Self* variable to refer to. A second issue was making this "alien" function capable of calling the component's event dispatch methods. These are protected access, and technically not available from outside the component. To solve this problem, we exploited a little-known feature of Delphi. Regardless of the specified access level, all of an object's data members and methods are available to any code contained in the same unit. This allows the callback to directly call the *TkbBrowseForFolderDialog* class' protected event dispatch method. This would not have been possible if the callback was implemented in a different unit.

```
{ If the RootFolder property specifies to use the Path
  property as the root, fetch the PIDL for that path and
  load it into the pidlRoot member of BrowseInfo. }
if (Self.RootFolder = kbsdPath) then
  begin
    BrowseInfo.pidlRoot := GetPIDLFromPath(Self.RootPath);
  end  { if }
{ If the specified root is Desktop, just set a nil PIDL. }
else if (Self.RootFolder = kbsdDesktop) then
  begin
    BrowseInfo.pidlRoot := nil;
  end  { else if }
{ If RootFolder isn't specifying a path for the root, try
  to fetch the PIDL for some special folder. If the folder
  is not recognized, just leave the root PIDL nil to get a
  default tree. }
else
  begin
    BrowseInfo.pidlRoot :=
      GetSpecialLocationPIDL(Self.RootFolder);
  end; { else }
```

Figure 6: Setting the root folder for *SHBrowseForFolder*.

```
function BrowseForFolderCallback(DialogHandle: HWND;
  MessageID: UINT; PIDL: LPARAM; ComponentPointer: LPARAM):
  Integer; stdcall;
var
  DialogComponent: TkbBrowseForFolderDialog;
begin
  { If the value we expect to point to the dialog component
    is not nil... }
  if (ComponentPointer <> 0) then
    begin
      DialogComponent :=
        TkbBrowseForFolderDialog(ComponentPointer);
      { Based on which message is invoking the callback,
        invoke the appropriate event dispatch method for
        the referenced component. We are cheating a bit;
        these are actually protected methods, but we can
        access them from outside the class because this
        code is in the same unit. }
      case (MessageID) of
        BFFM_INITIALIZED:
          DialogComponent.Initialize(DialogHandle);
        BFFM_SELCHANGED:
          TkbBrowseForFolderDialog(DialogComponent).Change(
            DialogHandle, PItemIDList(PIDL));
      end;
    end;
  { Always return 0. }
  Result := 0;
end;
```

Figure 7: Setting the root folder for *SHBrowseForFolder*.

The fact that the callback function receives the dialog box's handle allows us to cheat the system a little more. The *SHBrowseForFolder* function has no means of directly specifying the dialog box's title bar caption. However, because we can store the handle after the BFFM_INITIALIZED callback invocation, we can use the *SetWindowText* API function to set the dialog box's caption to our desired value. This is exactly what we do in the component's *Initialize* method, before calling the event handler — a little sly, and it works very well. Having the window handle offers a great deal of leverage to a knowledgeable Windows programmer. In fact, you could exploit this to further modify the dialog box and its various child windows if you like resizing, changing text, sending messages, receiving notifications, and so forth. The details of these enhancements will be left as an exercise.

Conclusion

As we stated last month, these standard shell dialog boxes provide the ability to integrate your application with the Windows shell. You can use them as drop-in solutions to common problems, saving time and improving the polish of your applications. The component and function wrappers give you these almost unknown function interfaces in convenient Delphi style, making these useful dialog boxes almost trivial to incorporate. Have fun, and write something amazing! ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\MAY\DI9905KB.

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

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

By Rod Stephens

Map Coloring

A Look at Four- and Five-color Algorithms

A map can be an incredibly useful tool, particularly if you don't like to stop and ask for directions. A complicated map can be quite confusing, however. For hundreds of years, cartographers have been making maps easier to read by displaying different countries or regions in different colors. If no two adjacent regions have the same color, it's much easier to see where one ends and the next begins.

In 1853, F. Guthrie speculated that it was possible to color any map this way using, at most, four colors. It wasn't until 1976, 123 years later, that this fact was proven by Appel and Haken. Unfortunately, the proof is computer assisted. It uses a program to exhaustively examine a huge number of graphs, and doesn't provide an efficient algorithm for four-coloring a map.

This article describes two algorithms for coloring maps (both are available for download; see end of article for details). The first is a somewhat inefficient algorithm that four-colors a map. While the algorithm is theoretically inefficient, in practice, it is quite fast for maps with a reasonably large number of regions.

The second alternative is an efficient algorithm that colors a map using five colors. Whether you color a map with four or five colors usually doesn't matter. The only reason you might need to use exactly four colors is if you have a somewhat unusual computer that can display only four colors, or if you are trying to impress your friends with your algorithmic prowess. In these cases, use the inefficient four-coloring algorithm, and be patient.

Exhausting Work

One way to four-color a map is to simply examine all the possible combinations of colors for the regions until you find a combination that works. If there are R regions on the map, there are R^4 possible combinations of colors. If R is large, this can be a huge number of combinations. For example, if R is only 10, R^4 is 20,000. Because the program searches all these combinations in a rather simple-minded manner, this is called an *exhaustive search*.

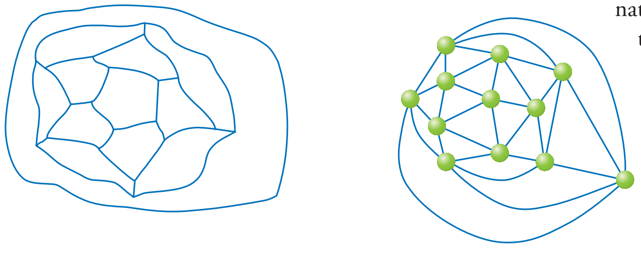


Figure 1: A map and its adjacency graph.

The first step in any map-coloring algorithm is to convert the map into an *adjacency graph*. Each *node* in the graph corresponds to a *region* on the map (the two terms are used interchangeably in the rest of this article). Two nodes in the graph are connected with a link if their regions share a border in the map. Figure 1 shows a small map and its corresponding adjacency graph.

The example program Color4 uses an exhaustive search to four-color maps (see Figure 2). Much of the program's code is dedicated to loading, editing, drawing, and otherwise manipulating maps. This code is long and irrelevant to the map-coloring algorithm, so it's not described here.

Color4 uses the *TRegion* class to store information about regions. The code for this class is shown in Figure 3, with region loading, editing, and other irrelevant routines omitted. *TRegion*'s *points* array stores the points that define the region's borders. The program uses this array to draw the region. It also compares the points in two regions' borders to see if the regions are adjacent. The *neighbors TList* contains a list of the adjacent regions. This list gives the links between the nodes in the adjacency graph. The *color_number* variable holds the index of the color for the region. When the algorithm is finished, this will be a number between 1 and 4.

As far as the map-coloring algorithm is concerned, *TRegion* provides only two interesting methods: *HasLine* and *CheckNeighbor*. *HasLine* returns True if the region's border contains a specific line segment. *CheckNeighbor* uses *HasLine* to see if a specified region is adjacent to this region. For each border segment in this region, *CheckNeighbor* calls the other region's *HasLine* function to see if the regions share the segment. If *HasLine* ever returns True, the regions are neighbors, so the procedure adds the two regions to each other's neighbor list.

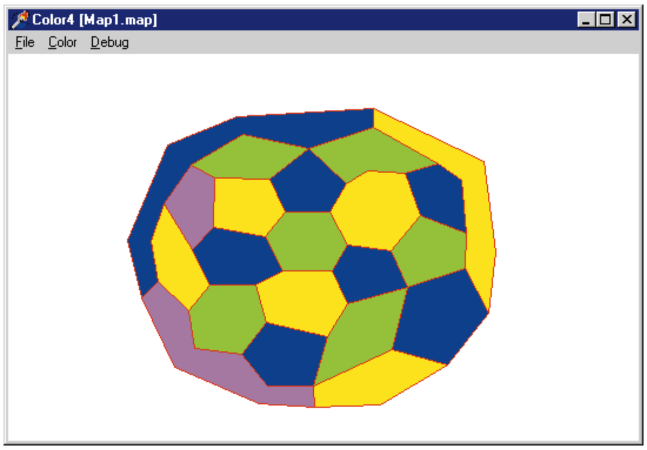


Figure 2: The example program Color4 uses an exhaustive search to four-color maps.

The program's *TMapForm* class includes only two interesting class variables: *regions* and *colors*. The variable *regions* is a *TList* object that holds a list of all the regions on the map. The *colors* array is an array of color values that the program uses to color the regions. The value *colors[0]* is a background color used for regions that haven't yet been assigned colors. For example, when you first load a map, all the regions have this color.

TMapForm has three routines that deal with map coloring: *AssignColors*, *FindNeighbors*, and *AssignOneColor*.

When the user invokes the **Color Nodes** command from the **Color** menu, the program invokes the *AssignColors* procedure, shown in Figure 4. *AssignColors* resets the regions' colors and clears their neighbor lists. It then calls *FindNeighbors* to find the regions' current neighbors. *AssignColors* then assigns an arbitrary color to node 0 in the graph. It doesn't matter which color the program uses for this first node. The color helps determine the colors of the other nodes, but a solution is possible no matter what color is used first.

The procedure then calls function *AssignOneColor*, passing it the parameter 1. That tells *AssignOneColor* to begin assigning colors

```
type
  // Array of points.
  TPointArray = array [1..1000000] of TPoint;
  // Pointer to array of points.
  PPointArray = ^TPointArray;

  // Map region class.
  TRegion = class
    public
      // # points on the region's border.
      num_points    : Integer;
      // The points on the region's border.
      points        : PPointArray;
      // List of neighboring TRegions.
      neighbors     : TList;
      // Region number starting with 0.
      number        : Integer;
      // Color number for the region.
      color_number  : Integer;

      constructor Create; virtual;
      ...
      // Region loading, editing, etc. declarations omitted.

      function HasLine(a1, b1, a2, b2: Integer): Boolean;
      procedure CheckNeighbor(rgn: TRegion);
      procedure AddNeighbor(nbr: TRegion);
  end;

// Set some defaults.
constructor TRegion.Create;
begin  inherited Create;
  num_points    := 0;
  neighbors     := TList.Create;
  number        := -1;
  color_number  := 0;
end;

// Return True if the region contains this line segment.
function TRegion.HasLine(a1, b1, a2, b2: Integer): Boolean;
var
  i, x1, y1, x2, y2 : Integer;
begin
  // Assume we will find it.
  Result := True;
  x1  := points^[1].X;
  y1  := points^[1].Y;
  for i := 2 to num_points do begin
    x2  := points^[i].X;
    y2  := points^[i].Y;
    if ((x1=a1) and (y1=b1) and (x2=a2) and (y2=b2)) then
      Exit;
    if ((x2=a1) and (y2=b1) and (x1=a2) and (y1=b2)) then
      Exit;

    x1  := x2;
    y1  := y2;
  end;

  // We didn't find the line.
  Result := False;
end;

// See if this region is adjacent to the indicated one.
// If so, add the regions to each other's adjacency lists.
procedure TRegion.CheckNeighbor(rgn : TRegion);
var
  i, x1, y1, x2, y2 : Integer;
begin
  x1  := points^[1].X;
  y1  := points^[1].Y;
  for i := 2 to num_points do begin
    x2  := points^[i].X;
    y2  := points^[i].Y;
    if (rgn.HasLine(x1, y1, x2, y2)) then
      begin
        // They are neighbors. Add them to each other's
        // adjacency lists.
        neighbors.Add(rgn);
        rgn.neighbors.Add(Self);
        Exit;
      end;
    x1  := x2;
    y1  := y2;
  end;
end;

// Region loading, editing, etc. routines omitted.
...
```

Figure 3: The *TRegion* class.

to the nodes, starting with node 1. How that assigns colors to all the nodes is described a little later.

The *FindNeighbors* procedure, shown in Figure 5, examines every pair of regions. It invokes one of the pair's *CheckNeighbor* procedures to see if the two regions are neighbors. If they are, *CheckNeighbor* adds them to each other's neighbor lists.

The heart of the exhaustive search is the *AssignOneColor* function, shown in Figure 6. This function takes as a parameter the index of the next region it should consider. If that index is greater than the largest index of any region, all the nodes have been assigned colors successfully. That means the current assignment of colors is a valid four-coloring. Function *AssignOneColor* sets its return value to True to indicate that it found a valid coloring, then exits.

If it has not yet found a valid coloring, the function tries to give the region it is considering each of the four colors in turn. For each color, the function determines whether one of the region's neighbors has already used that color. In that case, the program cannot also assign the color to this region.

If the color isn't already used by any of the region's neighbors, *AssignOneColor* assigns that color to this region. It then recursively calls itself to assign a color to the region with the next index. If that call to the function returns True, the program has found a valid four-coloring, so this call to *AssignOneColor* sets its return value to True and exits.

If the recursive call returns False, the program cannot find a valid coloring with this region having the assigned color. The function continues to try the other colors. If the function cannot find a valid coloring using any of the four colors for this region, it resets the region's color, sets its return value to False, and exits. As the calls to *AssignOneColor* return back up the call stack, they will make new color assignments for the previously colored regions, and try again to color this region. The recursion will continue to try color combinations until it finds one that works.

The Color4 program uses this code to four-color maps. Use the **File** menu to open a map file, or draw your own map. Click the left mouse button to start a new region or add to the current region. Click the right button to finish the region. To make two regions neighbors, use the same end points for at least one of their edges.

```
// Four-color the map.
procedure TMapForm.AssignColors;
var
  rgn_i : Integer;
  rgn   : TRegion;
begin
  // Reset all the colors and clear the neighbor lists.
  for rgn_i := 0 to regions.Count - 1 do begin
    rgn := regions.Items[rgn_i];
    rgn.neighbors.Clear;
    rgn.color_number := 0;
  end;
  // Make the adjacency lists.
  FindNeighbors;
  // Only continue if there are regions.
  if (regions.Count > 0) then
    begin
      // Assign the first region a color.
      rgn := regions.Items[0];
      rgn.color_number := 1;
      // Assign the other colors using an
      // exhaustive search.
      if (not AssignOneColor(1)) then
        ShowMessage(
          'Error: Could not find a valid coloring.');
    end;
end;
```

Figure 4: The *AssignColors* procedure four-colors the map.

```
// Create the adjacency lists for the regions.
procedure TMapForm.FindNeighbors;
var
  i1, i2     : Integer;
  rgn1, rgn2 : TRegion;
begin
  // Compare each region to every other and see if they
  // are adjacent.
  for i1 := 0 to regions.Count - 2 do begin
    rgn1 := regions.Items[i1];
    for i2 := i1 + 1 to regions.Count - 1 do begin
      rgn2 := regions.Items[i2];
      rgn1.CheckNeighbor(rgn2);
    end;
  end;

  // Display the neighbor lists if desired.
  if (mnuShowNeighborLists.Checked) then
    ShowNeighborLists;
end;
```

Figure 5: The *FindNeighbors* procedure finds the neighbors for each region.

```
// Assign a color for the node with the indicated index.
// Recursively assign colors for the other nodes. Return
// True if we find a valid assignment.
function TMapForm.AssignOneColor(rgn_i: Integer): Boolean;
var
  color_num, nbr_i : Integer;
  rgn, nbr         : TRegion;
begin
  // If rgn_i >= regions.Count, then all regions have been
  // assigned and we have a valid solution.
  if (rgn_i >= regions.Count) then
    begin
      Result := True;
      Exit;
    end;
  // Try each possible color for this region.
  rgn := regions.Items[rgn_i];
  for color_num := 1 to 4 do begin
    // See if this color is available for this region.
    for nbr_i := 0 to rgn.neighbors.Count - 1 do begin
      // See if this neighbor has used the color already.
      nbr := rgn.neighbors.Items[nbr_i];
      if (nbr.color_number = color_num) then Break;
    end;
    // See if the color is usable.
    if (nbr_i >= rgn.neighbors.Count) then
      begin
        // Assign this color to the region.
        rgn.color_number := color_num;
        // Recursively assign colors to the other regions.
        if (AssignOneColor(rgn_i + 1)) then
          begin
            // We found a valid assignment.
            Result := True;
            Exit;
          end;
      end;
    // If everything worked and we found a complete
    // assignment, we have already exited. Otherwise
    // we continue trying other colors.
  end;
  // Blank our color so we can try again later.
  rgn.color_number := 0;
  // We found no valid assignments.
  Result := False;
end;
```

Figure 6: The *AssignOneColor* procedure recursively assigns colors to regions until it finds a valid four-coloring.

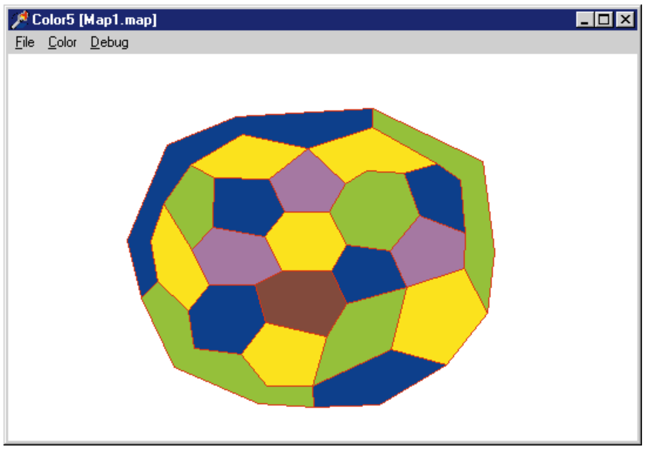


Figure 7: The example program Color5 five-colors maps.

When you have loaded or created a map, select the **Color Nodes** command from the **Color** menu, or press F9.

Planar Postulates

The four-coloring algorithm used by Color4 exhaustively examines color combinations until it finds one that works. For reasonably small problems, this algorithm is fast, and the program has no trouble. If a map contains many regions, however, exhaustive search can be impractical.

If the map contains R regions, there are R^4 possible color combinations. If R is 1000, R^4 is one trillion. If your computer can examine one million combinations per second, it will take more than 11 days to search them all. The program will probably find a valid coloring before it searches all the combinations, but there is no guarantee of quick success. In cases like this, you can use the five-coloring algorithm demonstrated by the Color5 program, shown in Figure 7. This program doesn't always find a four-coloring, but it's faster for very large maps.

In many ways, the example program Color5 is similar to the example program Color4; both use the same *TRegion* class and the same code to load and manipulate maps. Their differences are in how they find map colorings. Color5 uses two key facts about planar graphs to produce the five-coloring. A graph is planar if it can be drawn in a flat plane without any links crossing each other. Adjacency graphs, such as the one shown in Figure 1, are always planar.

Fact 1

The first useful fact about planar graphs is that nodes with fewer than five neighbors are easy to color. First, remove the node from the graph and color the remaining nodes. When they are colored, restore the removed node and look at its neighbors. Because the node has fewer than five neighbors, the neighbors cannot have used up all five colors. Pick an unused color and assign it to the node.

This fact alone would be enough to color the entire graph if every node had fewer than five neighbors. In fact, removing a node reduces the number of neighbors of each of its neighbors, so some of them may now have fewer than five neighbors. The program can continue like this indefinitely unless it eventually reaches a state where every node has at least five neighbors. For example, every node in the graph shown in Figure 1 has exactly five neighbors.

Fact 2

The second fact about planar graphs is useful when every node in the graph has at least five neighbors. When that is the case, it can be

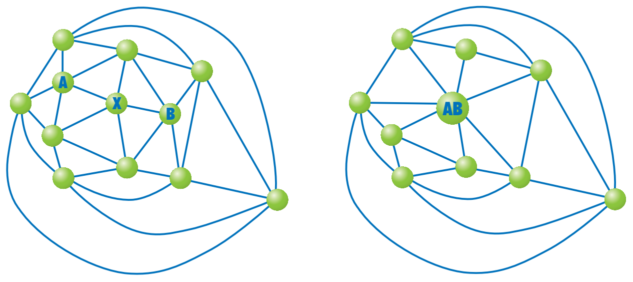


Figure 8: Combining nodes in an adjacency graph.

shown that there is a node X with exactly five neighbors, where two neighbors, A and B, are not neighbors of each other, and each has at most seven neighbors of its own. For example, the graph on the left in Figure 8 shows such nodes A, X, and B. Node X has five neighbors. Its neighbors A and B are not neighbors of each other, and they each have no more than seven neighbors.

Because nodes A and B are not neighbors, the program can assign them the same color. To continue processing the graph, the program removes node X and combines nodes A and B into a single node. It then continues coloring the remaining nodes. Figure 8 shows this combination process. Notice that nodes that were neighbors of both nodes X and node A or B have fewer neighbors than they did before. Now there are some nodes with fewer than five neighbors, so the program can use the first observation to continue processing the graph.

When it has finished processing the smaller graph, the program restores nodes X, A, and B. It gives nodes A and B the color it gave the combined node in the smaller graph. Because node X has exactly five neighbors, and nodes A and B have the same color, X's neighbors can have used at most four of the five colors available. Pick one of the remaining colors and assign it to node X.

Five-coloring

The example program Color5 uses these two facts to five-color maps. As it removes nodes from the graph, it places information about the nodes on a stack. When the graph is empty, the program removes the items from the stack in last-in-first-out (LIFO) order, using the information to assign colors to the nodes. In English, the algorithm is:

```
1) While the graph is not empty, do:
   a. If there is a node N with fewer than 5 neighbors, then:
      i)   Add node N and its current neighbor list to the stack.
      ii)  Remove node N from the graph.
   b. Else:
      i)   Find a node X with exactly five neighbors, two of which (A
           and B) are not adjacent and have at most seven neighbors.
      ii)  Add node N and its current neighbor list to the stack.
      iii) Remove node N from the graph.
      iv)  Add nodes A and B to the stack.
      v)   Combine nodes A and B by adding node B's neighbors to
           node A's neighbor list.
      vi)  Remove node B from the graph.
2) While the stack is not empty, do:
   a. If the next pair of objects on the stack is a node and its neigh-
      bor list saved in step 1.a.i or step 1.b.ii, then:
      i)   Examine the node's neighbors and assign the node a color
           that is not used by its neighbors.
   b. Else (the next pair contains two nodes A and B saved in step
      1.b.iv):
      i)   Assign node B the same color already assigned to node A.
```

The main difference between Color4 and Color5 is the *AssignColors* procedure, shown in Listing One (beginning on this page). This version of the procedure starts by creating two *TList* objects. The *orig_regions* list will contain a copy of the original region list. The procedure uses this list to restore the graph after it has torn it apart during the color calculations. The second *TList* object is the stack object. The program copies the region list into *orig_regions*, resets each region's color, and empties each region's neighbor list. It then calls procedure *FindNeighbors* to create the new neighbor lists.

While the regions list holds at least one region, the program removes a node from the graph. It first searches for a node with fewer than five neighbors. If it finds one, it adds the node and its neighbor list to the stack. It then calls the region's *RemoveFromGraph* procedure to remove the node from the neighbor lists of its neighbors. The program leaves the node's neighbor list alone so it will later be able to find the node's neighbors.

If the program cannot find a node with fewer than five neighbors, it calls the *FindNonAdjacent* function for regions until it finds a node with five neighbors, two of which are non-adjacent with at most seven neighbors of their own. It saves that region and its neighbor list on the stack, and removes the region from the graph. It then calls the *AssociateWith* procedure to associate one of the neighbor nodes with the other. It adds both neighbors to the stack, and removes the first from the graph.

Finally, when the region list is empty, the program empties the stack. When it finds a node and its neighbor list on the stack, the program assigns the node a color not already taken by its neighbors. When it finds two nodes, it assigns the second node the same color as the first node. *AssignColors* finishes by restoring the original region list.

The last new pieces of Color5 are support routines provided by the *TRegion* class, shown in Listing Two (beginning on page 24). The *RemoveFromGraph* procedure searches a node's neighbors and removes the node from the neighbors' neighbor lists. The *FindNonAdjacents* function searches a node's neighbors for two non-adjacent neighbors that have at most seven neighbors. The function returns True if it finds two such neighbors.

The *AddNeighbor* procedure checks whether a node is already the neighbor of another node. If it isn't, the routine adds each node to the other's neighbor list. Finally, the *AssociateWith* procedure associates one node with another. For each neighbor in the first node's neighbor list, the routine uses *AddNeighbor* to add the second node to the neighbor's list. This merges the two nodes in the graph, as shown in Figure 8. The example program Color5 uses this code to five-color maps. The interface is very similar to that of the example program Color4. Load or create a map, and press F9 to five-color it.

Conclusion

These two algorithms are useful tools for any amateur cartographer. The five-coloring algorithm used by Color5 is quite fast. At each step, it removes one node from the graph, so if the graph contains *N* nodes, the program can only run for *N* steps. The steps are somewhat complicated, but they are far shorter than the steps that may be needed by an exhaustive search.

Eventually, the five-color algorithm empties the graph, and can use the information in its stack to color the nodes one at a time. Even for enormous maps, this is fast, and the algorithm can finish in a reasonable amount of time. The exhaustive search used by Color4 can be slow for large maps. In practice, however, it is quite fast for maps of reasonable size. Because it produces a slightly better result and is much simpler than the other algorithm, exhaustive search is usually a better choice. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\MAY\DI9905RS.

Rod Stephens is the author of several books, including *Ready-to-Run Delphi 3.0 Algorithms* [John Wiley & Sons, 1998]. You can reach him at RodStephens@delphi-helper.com, or see what else he's up to at http://www.delphi-helper.com.

Begin Listing One — *AssignColors*

```
// Five-color the map.
procedure TMapForm.AssignColors;
var
  orig_regions, stack, nbrs : TList;
  i, rgn_num, n1_num        : Integer;
  rgn, n1, n2               : TRegion;
  color_used                : array [1..5] of Boolean;
  obj                       : TObject;
begin
  // Prepare the lists we need.
  orig_regions := TList.Create;
  stack := TList.Create;
  // Save a copy of the region list because we will
  // mess it up. Also reset the region's colors and
  // clear their adjacency lists.
  for i := 0 to regions.Count - 1 do begin
    rgn := regions.Items[i];
    orig_regions.Add(rgn);
    rgn.color_number := 0;
    rgn.neighbors.Clear;
  end;
  // Make the adjacency lists.
  FindNeighbors;
  // Push regions onto the stack.
  while (regions.Count > 0) do begin
    // Look for a region with degree < 5.
    rgn := nil;
    for rgn_num := 0 to regions.Count - 1 do begin
      rgn := regions.Items[rgn_num];
      if (rgn.neighbors.Count < 5) then Break;
    end;
    // If we found node with fewer than 5 neighbors,
    // add it and its neighbor list to the stack.
    if (rgn_num < regions.Count) then
      begin
        // Push rgn and its neighbor list onto the stack.
        stack.Add(rgn);
        stack.Add(rgn.Neighbors);
        // Remove rgn from the graph.
        rgn.RemoveFromGraph;
        regions.Delete(rgn_num);
      end
    else
      begin
        // There is no node with degree < 5. Search
        // for one with degree = 5 and 2 non-adjacent
        // nodes n1 and n2 with degree <= 7.
        for rgn_num := 0 to regions.Count - 1 do begin
          rgn := regions.Items[rgn_num];
          if (rgn.FindNonAdjacents(n1, n2)) then Break;
        end;
        // If we still did not find one, something
        // is wrong.
        if (rgn_num >= regions.Count) then
          begin
          ShowMessage('Error finding node to remove.');
          Break;
        end;
        // Save rgn and its adjacency list.
        stack.Add(rgn);
        stack.Add(rgn.neighbors);
        // Remove rgn from the graph.
        rgn.RemoveFromGraph;
        regions.Delete(rgn_num);
        // Associate n1 and n2.
        n1.AssociateWith(n2);
        // Push n1 and n2 onto the stack.
        stack.Add(n1);
        stack.Add(n2);
        // Remove n1 from the graph.
        n1.RemoveFromGraph;
        // Find n1 in the region list.
        for n1_num := 0 to regions.Count - 1 do begin
          n2 := regions.Items[n1_num];
          if (n1 = n2) then
            begin
              regions.Delete(n1_num);
              Break;
            end;
        end;
      end; // End if we did not find a node with degree < 5.
  end; // End while (regions.Count > 0) do.

  // The graph is empty. Produce the coloring.

  // Pop regions off the stack and color them.
  while (stack.Count > 0) do begin
    // See if the next item in the stack is a
    // neighbor list or a region.
    obj := stack.Items[stack.Count - 1];
    if (obj.ClassNameIs('TList')) then
      begin
        // Get the neighbor list.
        nbrs := stack.Items[stack.Count - 1];
        stack.Delete(stack.Count - 1);
        // Get the corresponding region.
        rgn := stack.Items[stack.Count - 1];
        stack.Delete(stack.Count - 1);
        // Assign rgn a color different from those
        // used by its neighbors.
        for i := 1 To 5 do
          color_used[i] := False;
        for n1_num := 0 to nbrs.Count - 1 do begin
          n1 := nbrs.Items[n1_num];
          color_used[n1.color_number] := True;
        end;
        // See which color is left.
        for i := 1 to 5 do
          if (not color_used[i]) then Break;
        // If we did not find an unused color,
        // something is wrong.
        if (i > 5) then
          ShowMessage('Error finding color for node.')
        else
          rgn.color_number := i;
        // End if the next item in the stack
        // is a neighbor list.
      end
    else
      begin
        // The next item in the stack is a region.
        // Get the region.
        n1 := stack.Items[stack.Count - 1];
        stack.Delete(stack.Count - 1);
        // Get the associated region.
        n2 := stack.Items[stack.Count - 1];
        stack.Delete(stack.Count - 1);
        // Assign n2 the same color as n1.
        n2.color_number := n1.color_number;
      end; // End if neighbor list/node ...
  end; // End while (stack.Count > 0) do
```

```
  // Restore the original region list.
  regions.Destroy;
  regions := orig_regions;
end;
```

End Listing One

Begin Listing Two — *TRegion*

```
// Remove the node from the graph.
procedure TRegion.RemoveFromGraph;
var
  nbr_num : Integer;
  nbr     : TRegion;
begin
  // Remove links from neighbors to here.
  for nbr_num := 0 to neighbors.Count - 1 do begin
    nbr := neighbors.Items[nbr_num];
    nbr.RemoveNeighbor(Self);
  end;
end;

// Find two mutually non-adjacent neighbors with
// degree <= 7, if they exist.
function TRegion.FindNonAdjacents(
  var n1, n2: TRegion): Boolean;
var
  n1_num, n2_num, n3_num : Integer;
  n3                     : TRegion;
  non_adjacent           : Boolean;
begin
  // If this node has more than 5 neighbors, it won't do.
  if (neighbors.Count > 5) then
    begin
      Result := False;
      Exit;
    end;
  for n1_num := 0 to neighbors.Count - 2 do begin
    // See if n1 has degree <= 7.
    n1 := neighbors.Items[n1_num];
    if (n1.neighbors.Count <= 7) then
      begin
        // n1 has degree <= 7. Find another.
        for n2_num:=n1_num+1 to neighbors.Count-1 do begin
          // See if n2 has degree <= 7.
          n2 := neighbors.Items[n2_num];
          if (n2.neighbors.Count <= 7) then
            begin
              // See if n1 and n2 are non-adjacent.
              non_adjacent := True;
              for n3_num := 0 to
                  n1.neighbors.Count-1 do begin
                n3 := n1.neighbors.Items[n3_num];
                if (n3 = n2) then
                  begin
                    // They are adjacent.
                    non_adjacent := False;
                    Break;
                  end;
              end;
              // If the nodes are non-adjacent, we're done.
              if (non_adjacent) then
                begin
                  Result := True;
                  Exit;
                end;
            end; // End if (n2.neighbors.Count <= 7) ...
        end; // End for n2_num = n1_num + 1 to ...
      end; // End if (n1.neighbors.Count <= 7) then
  end; // End for n1_num = 0 to neighbors.Count - 2 do
  // We did not find a usable pair of nodes.
  Result := False;
end;

// If this region is not yet in our neighbor list, add it
// and add us to its list.
procedure TRegion.AddNeighbor(nbr : TRegion);
```

```
var
  n1_num : Integer;
  n1     : TRegion;
begin
  // Examine our neighbors.
  for n1_num := 0 to neighbors.Count - 1 do begin
    n1 := neighbors.Items[n1_num];
    // If the node is in the neighbors list, do nothing.
    if (n1 = nbr) then Exit;
  end;
  // Update the neighbor lists.
  neighbors.Add(nbr);
  nbr.neighbors.Add(Self);
end;

// Associate this node with the target. Copy this node's
// neighbors into target's neighbor list.
procedure TRegion.AssociateWith(target : TRegion);
var
  n1_num : Integer;
  n1     : TRegion;
begin
  // Examine all neighbors.
  for n1_num := 0 to neighbors.Count - 1 do begin
    n1 := neighbors.Items[n1_num];
    if (n1 <> target) then
      begin
        // Add n1 to target's neighbor list and
        // vice versa.
        n1.AddNeighbor(target);
      end;
  end;
end;
```

End Listing Two

Delphi 3, 4 / Interfaces / OOP / HTML

By Keith Wood

An HTML Generator

Part I: Putting the Delphi Interface Construct to Work

In the May, 1996 issue of *Delphi Informant*, I introduced the *THTMLWriter* component in the article "An HTML Generator." This component allowed us to generate HTML from a Delphi program, with the full power and flexibility that a Delphi program provides. The approach used a single component to encapsulate the required behavior, and defined methods to generate the HTML.

Since the release of Delphi 3, we've had an alternate way to implement the generation of HTML — a more object-oriented approach. Delphi 3 introduced us to the interface, an abstract definition of a set of methods that an object can express. In this article, we'll redesign the HTML generator as a set of objects, the *IHTML* collection, that encapsulates one or more HTML tags (this article assumes a basic knowledge of HTML). We'll also update these objects to take into account changes for HTML 4.

```
{ Base class that implements the IHTMLProducer interface. }
THTMLBase = class(TObject, IHTMLProducer)
private
  FStyle: string;
  FId: string;
  FTagClass: string;
  FLanguage: string;
  FDirection: THTMLDirection;
  FTitle: string;
  FAccessKey: Char;
  FTabIndex: THTMLNumber;
  FOtherAttributes: string;
protected
  function QueryInterface(const IID: TGUID; out Obj):
    HResult; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
  property Style: string read FStyle write FStyle;
  property Id: string read FId write FId;
  property TagClass: string read FTagClass write FTagClass;
  property Language: string read FLanguage write FLanguage;
  property Direction: THTMLDirection
    read FDirection write FDirection;
  property Title: string read FTitle write FTitle;
  property AccessKey: Char
    read FAccessKey write FAccessKey;
  property TabIndex: THTMLNumber
    read FTabIndex write FTabIndex;
  property OtherAttributes: string
    read FOtherAttributes write FOtherAttributes;
  function BaseAttributesAsHTML: string;
public
  function AsHTML: string; virtual; stdcall;
end;
```

Figure 1: The *THTMLBase* class declaration, which is the basis of the HTML-generating hierarchy.

Interfaces

An interface defines a set of methods that determines the interactions expected of an object. The methods are not implemented in the interface (in this way, it's similar to an abstract class), but must be coded for each object that expresses that interface. Other differences from abstract classes are that interfaces can only have method and property declarations, and all properties must be accessed through functions or procedures. Also, all attributes must be **public**, and interfaces can have no constructor or destructor.

Delphi has a single inheritance model, which means that each class can be derived from only one other class, inheriting the latter's properties and methods. Interfaces allow us to simulate multiple inheritance through an object, expressing one or more of these sets of definitions. Like the normal class hierarchy, interfaces form their own hierarchies and are all ultimately derived from *IUnknown*. This interface defines the basic functionality required to discover what interfaces are available in an object, and to reference count accesses to them.

Our interface, *IHTMLProducer*, consists of a single method, *AsHTML*, that returns the contents of the implementing object formatted for inclusion in an HTML document. Its definition is as follows:

```
// The HTML producing interface; a single function that
// returns HTML formatted text.
IHTMLProducer = interface(IUnknown)
  ['{ 1265C6A2-5791-11D2-A65A-0000C08699E7 }']
  function AsHTML: string; stdcall;
end;
```

Positioning the cursor at the correct spot and pressing Ctrl Shift G enters the GUID (globally unique identifier) that identifies this interface. This value is required because later, we'll be testing for the existence of this interface within an object. The *AsHTML* function should be declared with the **stdcall** directive, because it may be called across process boundaries.

Any class that expresses this interface must define a function that implements this one routine. We don't require anything else from that class, and don't care what else it can or can't do.

THTMLBase

The base of our HTML-generating hierarchy is *THTMLBase*. This class implements the *IHTMLProducer* interface and declares the basic attributes that exist in most of the HTML tags (see Figure 1). It's derived directly from *TObject*, and we make use of the inherited *GetInterface* method to implement the *QueryInterface* function required by the *IUnknown* interface.

Normally, interfaces are reference counted, i.e. the number of references to each is tracked, and the object that implements the interface is destroyed when no one can access it any longer. This works well when you're dealing with the objects only through their interfaces. For our purposes, however, we're creating objects, changing properties specific to them, then generating the HTML through the interface. We control the creation and destruction of the objects and don't want the interface's scheme to interfere with that. To this end, we implement the *_AddRef* and *_Release* functions defined in the *IUnknown* interface to return a value of -1, disabling any processing dependent on them.

The basic HTML attributes declared in the *THTMLBase* class are set up as simple properties, directly referencing internal variables. We aren't concerned with changes to these values until we actually generate the HTML. They're defined as **protected** so they're not externally visible, but they can be exposed by subclasses. The *BaseAttributesAsHTML* method formats these values as HTML tag attributes, and returns that string. Again, it's **protected** so that it can be used only by subclasses.

At this stage, the *AsHTML* method required by the interface does nothing, returning a blank string. This method is overridden by each subclass to generate the HTML tag that it encapsulates.

THTMLContainer

Many HTML tags, such as the paragraph and table tags, contain other HTML tags. Because this is common, we create a new class that provides this functionality. Therefore, any class derived from this one automatically inherits the containership abilities.

THTMLContainer extends *THTMLBase* and maintains an internal list of objects that it contains (see the class declaration in Figure 2). It contains methods to add objects to the list and to clear it out, as well as properties to determine the number of items currently held and to access each in turn. As an object is added to the internal list, it's checked to ensure that it expresses the *IHTMLProducer* interface. An exception is raised if this isn't the case.

```
{ Base container class for HTML. Any number of HTML
  producers can be added to this container and each will
  generate its HTML in turn. }
THTMLContainer = class(THTMLBase)
private
  lstHTML: TList;
  FOwnContents: Boolean;
  function GetCount: Word;
  function GetItems(Index: Integer): TObject;
public
  constructor Create;
  destructor Destroy; override;
  function AsHTML: string; override; stdcall;
  function ContentsAsHTML: string;
  procedure Add(objHTML: TObject);
  procedure Clear;
  property Count: Word read GetCount;
  property Items[Index: Integer]: TObject
    read GetItems; default;
  property OwnContents: Boolean
    read FOwnContents write FOwnContents;
  end;
```

Figure 2: The *THTMLContainer* class declaration, which is the basis for HTML tags that contain other tags.

The *ContentsAsHTML* method calls the *AsHTML* method of each of the objects in the list in turn (hence the need to check their type on adding), and combines the results. It's declared as **protected** so it's available to subclasses without being generally visible. The *AsHTML* method for this object simply calls the *ContentsAsHTML* method.

To facilitate memory management, we add another property to control what happens to the contained objects when the container is cleared or destroyed. When *OwnContents* is True, all the objects in the list are freed when the list is cleared. When it's False, the contained objects are left alone. By default, the *OwnContents* property is set to True. Because the HTML document itself is a container, we only need to keep a reference to it, and free it when we're finished. The document object, in turn, frees all the objects it contains, removing the need to track each individually.

Generating HTML

The actual HTML generation is done through the *AsHTML* function, which is declared in the interface. Each subclass overrides this method to produce HTML appropriate to the tag it encapsulates. Because the object generates all the HTML it requires, there's no need to keep track of opening and closing tags (with the possibility that these will become unsynchronized).

As an example, let's look at the paragraph tag. This is wrapped in the *THTMLParagraph* class (see Figure 3). It declares two properties of its own, *Alignment* and *Text*, and exposes several of the basic properties from *THTMLBase*. Its constructor allows us to create a basic paragraph that contains some text, while initializing the *Alignment* property to the default value.

THTMLParagraph subclasses *THTMLContainer*, which means we can add other tags to the paragraph. All these different elements are combined within the *AsHTML* function. It first specifies the tag header for an HTML paragraph (`<p`), and follows this with the value of the *Alignment* property (if not the default), and those of the basic attributes that have been set (through the *BaseAttributesAsHTML* method). This completes the opening paragraph tag, denoted by `>`.

The contents of the paragraph are added next. This comes in two parts: First, the *Text* property, which provides a quick way of adding

```
{ An HTML paragraph. }
THTMLParagraph = class(THTMLContainer)
private
  FText: string;
  FAlignment: THTMLAlignmentHoriz;
public
  constructor Create(sText: string); virtual;
  function AsHTML: string; override; stdcall;
  property Text: string read FText write FText;
  property Alignment: THTMLAlignmentHoriz
    read FAlignment write FAlignment;
  property Style;
  property Id;
  property TagClass;
  property Language;
  property Direction;
  property Title;
  property OtherAttributes;
end;

{ Initialise. }
constructor THTMLParagraph.Create(sText: string);
begin
  inherited Create;
  FText := sText;
  FAlignment := ahDefault;
end;

{ Return formatted HTML. }
function THTMLParagraph.AsHTML: string;
begin
  { Check for errors.}
  if (heStrictHTML4 in HTMLErrorLevel) and
     (Alignment <> ahDefault) then
    raise EHTMLError.Create(
      Format(sDeprecatedAttribute, ['p']));
  { Generate HTM. }
  Result := '<p' + AlignHorizAttribute('align',Alignment) +
    BaseAttributesAsHTML + '>' + EncodeSpecialChars(Text) +
    ContentsAsHTML + '</p>'#13#10;
end;
```

Figure 3: The *THTMLParagraph* class encapsulates an HTML paragraph.

```
var
  htm: THTMLDocument;
begin
  try
    htm := THTMLDocument.Create(
             'IHTML Objects Demonstration');
    htm.AddHeaderTag(THTMLMetadata.Create(
      'Keywords', 'IHTML;Demonstration'));
    htm.AddHeaderTag(THTMLComment.Create(
      'This page demonstrates the IHTML objects'));
    htm.Add(THTMLHeading.Create(
      1, 'IHTML Objects Demonstration'));
    htm.Add(THTMLParagraph.Create(
    'Welcome to the demonstration of the IHTML objects.'));
    memHTML.Text := htm.AsHTML;
  finally
    htm.Free;
  end;
end;
```

Figure 4: Generating a simple HTML document.

straight text to the paragraph, and then the HTML from the tags contained within this one. We use the inherited *ContentsAsHTML* method to generate the latter. Note that the text is passed through a routine that converts special characters to their HTML equivalents. Finally, we add the closing paragraph tag, `</p>`.

THTMLDocument

Producing an HTML document then becomes the task of the *THTMLDocument* object. It has properties that allow us to specify the header details for the document, including the title (mandatory), a base document, and a style sheet. Methods allow us to add additional tags to the header block; these can be link specifications, metadata, scripts, or objects.

Additional properties provide for the setting of the document's color scheme (although this approach is now deprecated) and/or a background image. The *THTMLDocument* object is derived from *THTMLContainer*, and so the body text and other content are placed with the *Add* method.

Because of the default behavior that containers free the objects they contain on their own destruction, we only need to keep track of the document itself. Everything we add to it will be released once we free the main document. Thus, we could generate a simple document with the code in Figure 4. The resulting page is being copied into a memo field. The full hierarchy of objects provided by the *IHTML* collection is shown in Figure 5.

Merging Documents

Rather than having to generate the entire document within our Delphi program, we can have the common HTML text in an external file, and simply substitute for the parts that change. This is achieved through the *THTMLStream* object.

It takes a stream as input, and parses it for special tags to be replaced. These tags are identified by starting with a pound sign (#), the same convention that Delphi uses for its page producer components. For each such tag found, an event is generated that allows us to specify the new value. One such tag value is built into the object itself: the `<time>` tag that inserts the current date and time. It takes an optional *format* attribute that can specify the Delphi formatting string to be used. So, if we code the following in the HTML document:

```
<#time format="d mmmm, yyyy hh:nn ampm">
```

it might appear as:

```
6 November, 1998 08:23 PM
```

As an extension to this scheme, we can include another file into the first one by following the pound sign with ^ (caret) and the name of that file. The new file is also parsed in the same manner, allowing for further substitutions.

Data Tables

To facilitate the generation of HTML tables from information in a database table, we have the *THTMLDataTable* object. This is declared in the *IHTMLDB* unit, because it doesn't encapsulate one of the basic HTML tags.

It derives from the *THTMLTable* object, and uses its inherited abilities to generate the actual table HTML. The main new prop-

- *THTMLBase*
  - *THTMLComment*
  - *THTMLContainer*
    - *THTMLAnchor*
    - *THTMLButton*
    - *THTMLDivision*
    - *THTMLDocument*
    - *THTMLFieldSet*
    - *THTMLForm*
    - *THTMLFrameSet*
    - *THTMLHeading*
    - *THTMLImageMap*
    - *THTMLLabel*
    - *THTMLList*
    - *THTMLListItem*
    - *THTMLNoScript*
    - *THTMLObject*
    - *THTMLParagraph*
    - *THTMLSelectField*
    - *THTMLSelectGroup*
    - *THTMLTableBase*
      - *THTMLTable*
        - *THTMLDataTable*
      - *THTMLTableCellBase*
        - *THTMLTableDetail*
        - *THTMLTableHeading*
      - *THTMLTableColumnGroup*
      - *THTMLTableRow*
      - *THTMLTableRowGroup*
    - *THTMLText*
  - *THTMLFrame*
  - *THTMLHorizRule*
  - *THTMLImage*
  - *THTMLImageMapArea*
  - *THTMLInlineFrame*
  - *THTMLInputField*
    - *THTMLButtonField*
    - *THTMLCheckboxField*
    - *THTMLFileField*
    - *THTMLHiddenField*
    - *THTMLImageField*
    - *THTMLPasswordField*
    - *THTMLRadioField*
    - *THTMLResetField*
    - *THTMLSubmitField*
    - *THTMLTextField*
  - *THTMLLineBreak*
  - *THTMLLink*
  - *THTMLMetadata*
  - *THTMLObjectParam*
  - *THTMLScript*
  - *THTMLSelectOption*
  - *THTMLStream*
  - *THTMLStyleSheet*
  - *THTMLTableColumn*
  - *THTMLTextareaField*

Figure 5: The hierarchy of *IHTML* objects.

erty is *DataSet*, which allows us to indicate where to obtain the data. Additional properties allow for the inclusion or exclusion of a header row, as well as the specification of its color scheme. The inherited abilities provide for the usual customization of the table.

The dataset attached to this object determines what data is displayed. All the visible fields from that source are displayed in turn. Their formatting can be controlled through the usual mechanisms for manipulating dataset fields. Memos have their entire contents added to the table, while all other fields show the *DisplayText* value. All fields can have a default alignment, or use the alignment from the field itself by setting the *UseFieldAlign* property to True.

Events allow for the properties of an entire row, or for each cell to be overridden. For cells, this includes any alteration of the displayed text itself. To ease the process of creating links within this table, we can specify fields to be used as the hot-spot text, *LinkField*, and for the destination, *LinkTarget*. These are then automatically formatted as the table is generated.

Error Reporting

Each *IHTML* object encapsulates an HTML tag, so it can perform its own error checking. Before generation of the HTML, the objects check for problems, such as missing mandatory fields and deprecated tags and/or attributes.

Setting the global variable *HTMLErrorLevel* determines the level of error checking. This value is a set of the levels of error reporting required. The levels are *heStrictHTML4*, which checks for deprecated items in HTML 4, and *heErrors*, which reports missing mandatory attributes. The default value is *heErrors*. Any errors found are notified by raising an *EHTMLError* exception. This derives directly from *Exception* without adding anything new.

Delphi 4 Bonuses

Delphi 4 also introduced changes to the Object Pascal language. These include the overloading of method declarations, and the provision of default values for parameters in method calls.

To enhance the abilities of the *IHTML* objects we just described, we can make use of these new capabilities. Using conditional compiles, we can set up extensions to be used when compiled under Delphi 4. But first we need to be able to identify when this occurs.

All versions of Delphi define a standard symbol to identify that version: In Delphi 2 it's VER90, in Delphi 3 it's VER100, and in Delphi 4 it's VER120. (Whatever happened to VER110?) Using this symbol in a conditional compiler directive allows us to target the enclosed code for that version. We extend the parameter lists for the constructors of some of the *IHTML* objects under Delphi 4, allowing the more common additional attributes to be set. By supplying default values for these, we don't require the user to enter them all, if they're not necessary. For example, in the *THTMLParagraph* constructor, we add parameters for the class, ID, and inline style of the paragraph:

```
{ $IFDEF VER120 }  { Delphi 4 }
constructor Create(sText: string; sTagClass: string = ' ';
  sId: string = ''; sStyle: string = ''); virtual;
{ $ELSE }
constructor Create(sText: string); virtual;
{ $ENDIF }
```

Then, under Delphi 4, we could code any of the following:

```
htm.Add(THTMLParagraph.Create('A basic paragraph.'));
htm.Add(THTMLParagraph.Create(
  'A formatted paragraph.', 'format1'));
htm.Add(THTMLParagraph.Create(
  'An individually formatted paragraph.', 'format1',
  'para1'));
htm.Add(THTMLParagraph.Create(
  'A specially formatted paragraph.', 'format1','',
  'background-color: red'));
```

Lazy Wrappers

The objects described in this article provide the functionality necessary to generate HTML from a Delphi program, but the class names can be lengthy, which requires more typing. A solution to this is to produce a wrapper unit that reduces these names and makes them easier to enter. Each class to be abbreviated is simply assigned to the new shortened class name. Here's an example:

```
type
  TP = THTMLParagraph;
  TH = THTMLHeading;
  TA = THTMLAnchor;
```

Then we would be able to add a new paragraph to an HTML document with the following statement:

```
htm.Add(TP.Create('A shorter paragraph call'));
```

To make the code even shorter, we could subclass the original class and replace its methods with abbreviated versions as well. For example, we could subclass *THTMLParagraph*, as shown in Figure 6. The implementation of these constructors is just a call to the longer original versions. Now we can have the following:

```
htm.Add(TP.New('An even shorter paragraph call'));
```

Feel free to apply this technique to whichever of the *IHTML* objects you desire.

Demonstration

The demonstration program that accompanies this article allows you to generate five documents. (The demonstration program is available for download; see end of article for details.) In each case, the HTML page is displayed as source on the screen, and is saved to a file that can be opened in your browser. The name of the file is given each time.

The first example illustrates many of the common tags used in HTML documents. These include headings, formatted text, lists, images, and forms. The second demonstrates the use of tables and draws a chessboard complete with pieces (assuming that the supplied graphics are located in a subdirectory called images). Next, there is an example of a frameset document. It is based on the example in the HTML specification and displays three frames, two with images and the last with the source code.

HTML tables can also be generated from database tables, as the fourth example demonstrates. The contents of the Biolife table (minus the graphic) are displayed within the browser. Note that the generating object automatically handles memo fields. Two events are attached to the table creation to color every second row light blue and the length columns red.

```
TP = class(THTMLParagraph)
public
{ $IFDEF VER120 }  { Delphi 4 }
  constructor New(sText: string; sTagClass: string = '';
    sId: string = ''; sStyle: string = ''); virtual;
{ $ELSE }
  constructor New(sText: string); virtual;
{ $ENDIF }
end;

{ $IFDEF VER120 }  { Delphi 4 }
constructor TP.New(sText: string; sTagClass: string = '';
    sId: string = ''; sStyle: string = ''); virtual;
begin
  Create(sText, sTagClass, sId, sStyle);
end;
{ $ELSE }
constructor TP.New(sText: string); virtual;
begin
  Create(sText);
end;
{ $ENDIF }
```

Figure 6: Subclassing *THTMLParagraph*.

Lastly, we show how HTML templates can be combined with substituted text, or other documents, to generate new pages. An event is attached to handle the replacement of the marked tags. In each case, except for the frames example, the Delphi source code that generated the document is available through a link at the bottom of the created page. In the frames example, the source code appears directly in one of the frames.

Conclusion

The interfaces introduced in Delphi 3 allow us to employ a more object-oriented approach to the generation of HTML from a Delphi program. Instead of the monolithic component created in the previous effort, we now have a set of smaller, integrated objects that encapsulate specific tags within an HTML document. Properties allow us to customize the tags without forcing us to fill in lots of unnecessary parameters in a method call. Furthermore, if the objects presented here don't work the way you would prefer, the object-oriented approach allows you to extend or replace them without affecting the rest of the hierarchy.

Using interfaces means we can add the HTML generating abilities to any object we desire, and have it interact seamlessly with the objects previously described. These new objects can appear anywhere within the class hierarchy and have any sort of inherited abilities. Next month, we'll look at more applications for this approach, including programs for generating Pascal source code to HTML, converting a directory structure to HTML, and producing a frameset definition document in a more visual way. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\MAY \DI9905KW.

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. Often working with Delphi, he has enjoyed exploring it since it first appeared. You can reach him via e-mail at kwood@ccsc.com.

By Cary Jensen, Ph.D.

Delphi Database Development

Part VIII: Validating Data

For most of the past year, this column has taken a systematic look at Delphi database development. This month's installment continues this series with a look at client-side data validation.

Data validation, as defined here, involves confirming that data entered by the user is acceptable before permitting it to be posted to a database, e.g. the data entered into a field (column) is within an acceptable range of values, and that data has been supplied for all required fields. It also involves verifying that data within a given record is consistent, e.g. when a payment type field indicates that a credit card was used, the corresponding credit card number field has been entered.

Overview of Data Validation

There are a number of ways that data validation can be implemented. If a database server is being used (e.g. InterBase, Oracle, or Microsoft SQL Server), you can define constraints and triggers on the server. Server-based validation ensures that all applications storing information on the server must abide by the rules defined there. If the data is being accessed in an *n*-tier environment, such as that supported by Inprise's MIDAS (Multi-tier Distributed Application Services) technology, these business rules can be defined on the application server. Finally, it's possible to define the data validation rules on the individual client applications.

Each of these approaches has its advantages and disadvantages. For example, placing business rules on the database (whether it's a remote database or a local one) has the advantage that the rules are applied regardless of how the data is being added to the database. Whether the data is coming from client applications in a client/server environment, being inserted from an application server, or being imported directly to the database, these rules are respected. The drawback to this approach is that any change to the back end requires all data-based triggers and constraints to be redefined for the new server, e.g. when your company replaces Microsoft SQL Server with Oracle.

The advantage of storing the rules on the application server is that all thin-client applications using the application server will use the rules. The drawback is that the rules are not enforced until the client application sends the user's data updates. This may occur after the user has made numerous inserts, updates, or deletions. If the user has repeatedly violated the same business rule, this may not be discovered until after the user has made extensive invalid edits.

Client-side validation can be used in any situation, including those stand-alone applications that don't involve a database server or an application server. Furthermore, client-side validation can be applied without involving a network round trip. The drawback to client-side validation is that it must be repeated for each client application. This introduces the potential for inconsistent application of business rules. Likewise, client-side validation involves binding the user interface to business logic, an association many developers want to avoid.

From a maintenance standpoint, it's ideal when all validation can be placed in only one of these three layers. In reality, though, effective validation often involves placing it in several layers. For example, although a majority of business rules may be applied at an application server layer, it might be desirable to create some validation on the client to decrease network traffic. Likewise, a primary key on a remote database table has the effect of reinforcing record uniqueness — regardless of rules defined in other layers.

Writing triggers and constraints on the database server is a server-specific topic. For information on applying business rules at this layer, refer to your database documentation. Writing business rules for the application server layer is a MIDAS topic. You can find some discussion of these issues in Bill Todd's article "Delphi 4 Multi-tier Techniques" in the January, 1999 issue of *Delphi Informant*, as well as the online Help. The remainder of this article focuses on the application of client-side validation.

Validating Data on the Client

Client-side validation is that which is defined in your Delphi application with which the user interacts. In general, there are four types of client-side validation: keystroke-level, field-level, record-level, and database-level. Keystroke-level validation involves accepting or rejecting individual keystrokes. Field-level validation is applied as each field is entered into a record. Record-level validation occurs when an individual record is being posted. Database-level validation is applied to sets of records as they are being applied to the underlying database. Each of these types of validation is introduced separately in the following sections.

Keystroke-level Validation

Keystroke-level validation involves evaluating each character as it is entered into a field, and the rejection of those characters that are invalid. There are two ways to apply keystroke-level validation:

- Write an event handler that executes after each character is entered, and raise an exception (silent or otherwise) when an invalid character is encountered.
- Use the *EditMask* property of a *TField*.

Writing an event handler to evaluate every keystroke entered by the user is the most intrusive of all data validation, and is therefore the least often used. This type of validation is usually achieved by adding an *OnKeyPress* event handler to an individual DBEdit control (or similar single-field data control), and evaluating each character as it is entered. This typically means that even if the field is displayed in a DBGrid (or some other multi-field control), the user is not permitted to edit it there. Instead, the user is required to select from a menu or click a button to display a modal dialog box in which the single field control appears. (A modal dialog box is one that must be accepted before the user can return to the form from which the dialog box was invoked.) Only after an acceptable value has been entered is the underlying field updated by your code.

A more common, and more easily applied form of keystroke-level validation involves the use of an edit mask. An edit mask is a pattern defined for an individual *TField*. At run time, Delphi ensures that only characters the mask permits are accepted. Any character not conforming to the mask is automatically rejected, without raising an exception.

In addition to simply rejecting or accepting characters, edit masks can also be used to perform run-time case conversions of character data. For example, using an edit mask, you can ensure that a particular sequence of characters is accepted as upper-case characters, even if the user entered them in lower case.

To create an edit mask, you must either instantiate *TField* descendants for your fields at design time (by right-clicking your DataSet component and using the Fields Editor to instantiate the fields), or you must assign the *EditMask* property of a *TField* at run time (using either a DataSet's *Fields* property or *FieldByName* method). Using design-time instantiated fields is the easiest, because it permits the property to be configured at design time.

The following is a simple edit mask:

```
>LL<
```

The `>` and `<` parts of this mask convert the entered characters into upper case, while the two `LL` characters require that exactly two letters be entered. This mask is useful for applications requiring the entry of a US state name abbreviation. While this technique doesn't ensure a valid state abbreviation is entered, it does ensure it's in the correct form.

Field-level Validation

Field-level validation involves evaluating the contents of an individual field as its value is being updated in the underlying record buffer. The record buffer is a holding area that stores an image of a record while it's being edited. It allows the user to modify a record in memory without affecting the underlying database as each field (column) is updated.

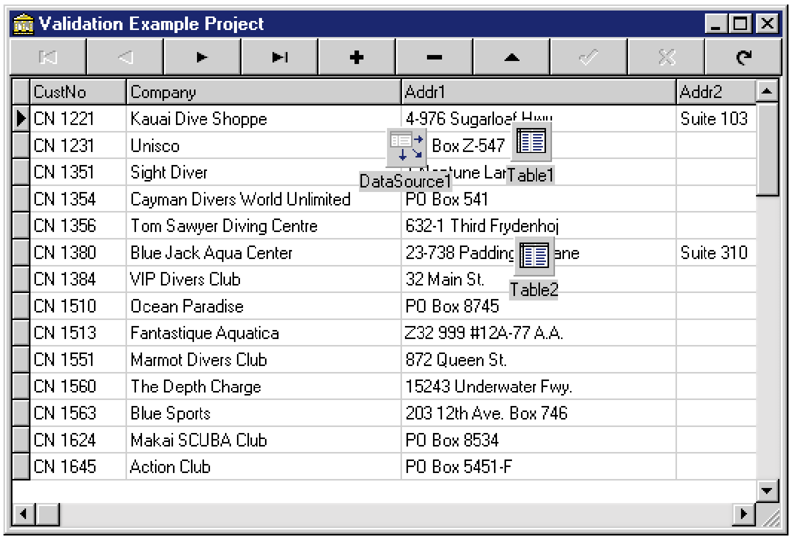
If a user enters invalid data into a field for which field-level validation is defined, they are prohibited from leaving the field until the data is corrected. While not as intrusive as keystroke-level validation, field-level validation tends to interrupt the flow of the user's work.

There is only one way to provide field-level validation with Delphi. This technique involves adding an *OnValidate* event handler to the *TField* associated with the field for which validation is needed. From within this event handler, you evaluate the contents of the field. If your code determines the value is invalid, you raise an exception. Raising an exception within an *OnValidate* event handler has the effect of preventing the value entered into the field from being updated to the underlying buffer. Furthermore, whatever action the user was attempting to perform is prevented, whether it's navigation to a different field in the same record, or some action that would have otherwise resulted in the current record being posted. Only after the user corrects the invalid data are they permitted to proceed.

Field-level validation is demonstrated in the VALID project (available for download; see end of article for details.) The project's main form, shown in Figure 1, contains a table, named *Table1*, that demonstrates a variety of validation techniques. All fields for this table were instantiated at design time using the Fields Editor. To display the Fields Editor, right-click the Table component and select **Fields Editor**. To instantiate *TField* components for each field associated with the table, press C A. (In versions of Delphi before Delphi 4, you must right-click the Fields Editor and select **Add Fields**. This results in the display of the Add Fields dialog box, wherein you select all field names and press Enter).

When the fields have been instantiated, select the field to which you want to add field-level validation, and add an *OnValidate* event handler. The following is the *OnValidate* event handler added to the Company field from the VALID project:

```
procedure TForm1.Table1CustNoValidate(Sender:
TField);
begin
  if CustNo.AsInteger < 1000 then
    raise EInvalidCustNoException.Create(
      'Customer numbers must be greater than
      1000');
end;
```



Validation Example Project

	CustNo	Company	Addr1	Addr2
▶	CN 1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
	CN 1231	Unisco	Box Z-547	
	CN 1351	Sight Diver	une La	
	CN 1354	Cayman Divers World Unlimited	PO Box 541	
	CN 1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
	CN 1380	Blue Jack Aqua Center	23-738 Padding ane	Suite 310
	CN 1384	VIP Divers Club	32 Main St.	
	CN 1510	Ocean Paradise	PO Box 8745	
	CN 1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	
	CN 1551	Marmot Divers Club	872 Queen St.	
	CN 1560	The Depth Charge	15243 Underwater Fwy.	
	CN 1563	Blue Sports	203 12th Ave. Box 746	
	CN 1624	Makai SCUBA Club	PO Box 8534	
	CN 1645	Action Club	PO Box 5451-F	

Figure 1: The main form of the VALID project.

From within this event handler, the value of the Customer number field, CustNo, is evaluated. If the value of this field is less than 1000, an exception is raised. As a result, the user is prevented from entering a value that is not one of the acceptable values. Furthermore, because this validation is applied at the field level, the user cannot continue to work with the record until this invalid value is correct.

There is another feature of this code worth noting. Specifically, instead of raising a generic exception, a custom exception was raised. This exception was defined by the following statements, which appear in a **type** clause in the project's main form unit:

```
ECustomException = class(Exception);
EInvalidCustNoException = class(ECustomException);
```

In general, whenever you raise an exception in code, it's considered good form to raise a custom exception, i.e. one defined by you. Doing so permits any future exception handling code you add to distinguish between exceptions raised by you and those generated by Delphi's components. In this case, a class named *ECustomException* is defined, and all explicitly raised exceptions are declared to descend from it.

Record-level Validation

Record-level validation is used to prevent a record from being posted when it contains invalid data. Unlike keystroke-level and field-level validation, the user is not prevented from entering invalid data. Indeed, a user may move freely throughout a record, entering invalid data all over the place. Before the user can post the record, however, this invalid data must be corrected. While some may argue this is not efficient, imagine how difficult it would be for you to complete a written form, such as your income taxes, one field at a time, with each field needing to be correct before you could continue to the next.

Delphi provides three ways to create record-level validation. Two involve properties, and one makes use of an event handler. Each of these is discussed in the following sections.

Before doing so, however, a comment is in order. The first two techniques, constraints and the *Required* property, can be applied at the field level. This might lead you to treat them as field-level validation, but doing so is incorrect. Field-level validation is applied on a field-by-field basis, which occurs during navigation, but does not necessarily involve the current record being posted. These techniques are clearly examples of record-level validation, because the rules you define are only applied if the corresponding record is being posted.

Constraints

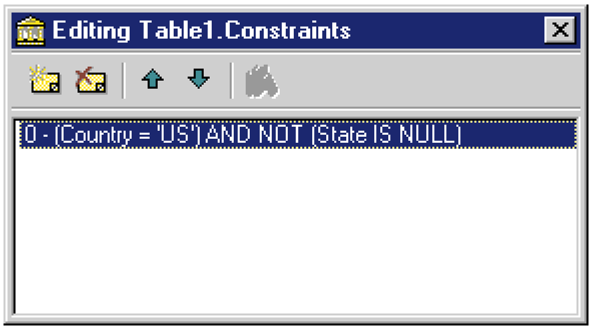
Constraints permit you to define Boolean SQL statements that are executed before the record is posted. If the expression evaluates to False, an internal exception is raised, preventing the record from being posted.

There are two ways to place constraints. You can use the *CustomConstraint* string property of individual *TField* components, or the *Constraints TCheckConstraints* property of your *TTable* or *TQuery* components. (*TStoredProc* components don't have a *Constraints* property. Furthermore, do not confuse this *Constraints* property with the *TControl.Constraints TSizeConstraints* property, which controls the size of visible components.)

The example project VALID employs the *CustomConstraint* property to ensure the Contact field isn't left blank. The value of the *CustomConstraint* property for the *Table1Contact TField* is "Contact IS NOT NULL." When a record is being posted, and one of the *TField CustomConstraint* SQL expressions evaluates to False, an exception is raised, and the string associated with the *TField*'s *ConstraintErrorMessage* is displayed to the user. In this example, the *ConstraintErrorMessage* property contains the string "You must enter a contact name."

Using the *CustomConstraint* property requires you either instantiate *TFields* at design time, or assign this property to *TFields* at run time. By comparison, the *Constraints* property for *TTable* and *TQuery* components permits you to define one or more constraints without working with individual *TFields*. The *Constraints* property contains one or more *Constraint* objects, each of which defines a *CustomConstraint* and an *ErrorMessage*. (The *Constraints* property also permits you to import constraints defined by a remote server. This is a useful feature for propagating server-side constraints to the client side, which can reduce network traffic. Using imported constraints is outside the scope of this article.)

You add a constraint to the *Constraints* property by displaying the *Constraints* property editor, shown in Figure 2. After the *Constraints* property editor is displayed, click the Add New



0 - (Country = 'US') AND NOT (State IS NULL)

Figure 2: The *Constraints* property editor.

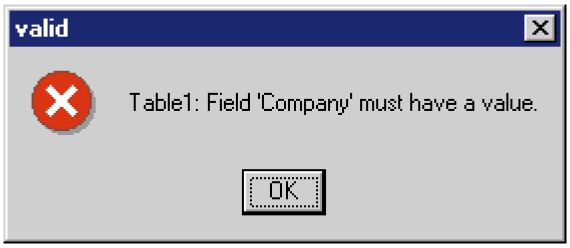


Table1: Field 'Company' must have a value.

Figure 3: The error message displayed when a required field is left blank.

button and define a *CustomConstraint* and an *ErrorMessage* string for the new *Constraint*. You can add as many constraints as you like, and each constraint can reference one or more fields in the table or query.

In the VALID project, a single constraint is added to *Table1*'s *Constraints* property. The value of the *CustomConstraint* property of this constraint is "(Country = 'US') AND NOT (State IS NULL)." When this expression returns False, the following error message is displayed to the user: "When Country is US you must supply a value for the State field."

The *Required* Property

The example of a *TField.CustomConstraint* property in the preceding section demonstrates how to require a user to supply a value for a field. However, there is an easier way of doing this. Instead, simply set the *Required* property of a *TField* to True. This will cause Delphi to verify that the associated field has been assigned a value before permitting the record to be posted. If the field is null, Delphi raises an exception and displays an error message.

The *Table1* Company field in the VALID project has the *Required* property set to True. If you attempt to enter a new Customer record and fail to enter a value in the Company field, the error message shown in Figure 3 is displayed.

Because setting the *Required* property is easier than defining a *CustomConstraint* for a field, you might wonder why I introduced *Constraints* first. The answer is that the error message displayed by Delphi when a Required field is left blank is automatically generated. By comparison, when you define a *CustomConstraint*, you also get to define the *ConstraintErrorMessage* property, permitting you to control the text of the message displayed to the user.

BeforePost

While constraints and the *Required* property are useful, the most flexible technique is to write a *BeforePost* event handler for your datasets. From a *BeforePost* event handler, you can evaluate any value in the current record, as well as examine data in other datasets, before deciding if the record is valid. If you determine the record is valid, you do nothing, and permit the default posting behavior to execute. If your code finds the record is invalid, you raise an exception, which has the effect of preventing the record from being posted.

Following is an example of code that appears in the VALID project. This code uses a second Table component to point to the Customer table; however, this one is sorted by Company name. If the record is being posted in an inserted record, this code verifies that the Company name doesn't already appear in the database. If it does, a custom exception is raised, and an error message describing the problem is displayed:

```
procedure TForm1.Table1BeforePost(DataSet: TDataSet);
begin
  if Table1.State = dsInsert then
    if Table2.FindKey([Table1Company.AsString]) then
      raise EDuplicateCustomer.Create(
        Table1Company.AsString +
        ' is already in the Customer table');
end;
```

This code demonstrates that you can reference multiple tables from a *BeforePost* event handler. However, it's unlikely you would write this specific test. Company name uniqueness can be assured using a unique index. Also, in some databases it might be perfectly valid to have two companies with the same name. This database would otherwise permit such a duplication because record uniqueness is ensured through the use of a unique Customer number.

Database-level Validation

Client-side database-level validation involves caching the user's edits to two or more records, then evaluating this work before permitting it to become a permanent part of the database. Unlike the other types of validation described here, database-level evaluation can involve data in multiple tables. A complete discussion of database-level evaluation is beyond the scope of this article. Therefore, this section will serve to describe the basic approach.

There are two parts to database-level validation. The first is that all changes, including inserts, deletions, and modifications, must be cached on the client side until validation is ready to take place. This can be accomplished in one of two ways. You can use cached updates or the *TClientDataSet* component. Cached updates can be used by any Delphi developer using Delphi 2 or later. The *TClientDataSet* component is only available in the client/server versions of Delphi 3 and 4. (I hope borland.com will make this tremendously useful component available in all versions of Delphi with Delphi 5.)

The second part of database-level validation is that when changes are applied, either from cache or from a client dataset, they are done without the context of a transaction. The transaction, which is applied using a *TDatabase*, permits all the user's edits to be cancelled, if necessary.

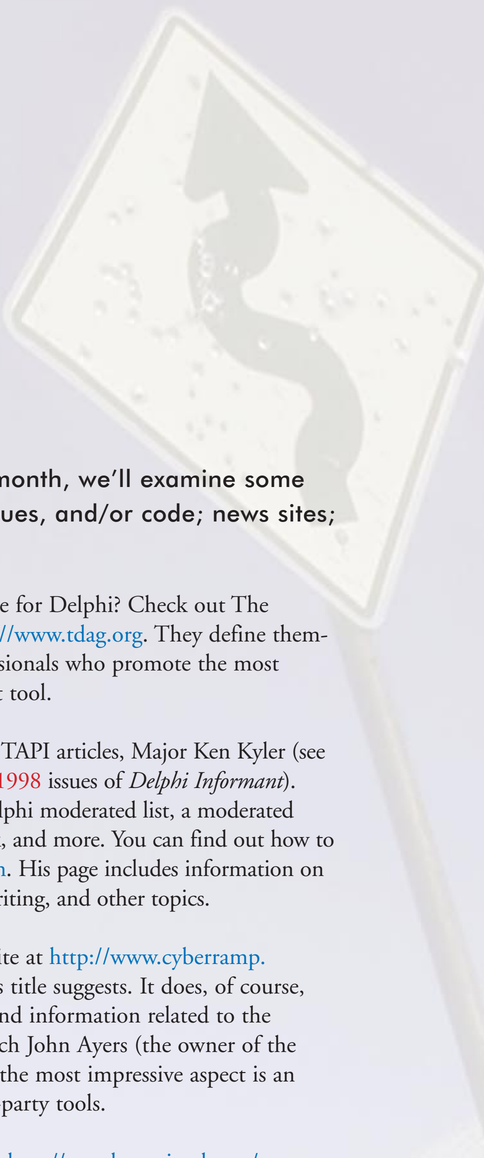
Conclusion

Client-side validation permits your code to evaluate data before it's applied to the underlying database, and to reject values that are not acceptable. While not appropriate for all applications, client-side validation nonetheless deserves a place in the repertoire of all Delphi database developers. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\MAY\DI9905CJ.

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

table of CONTENTS



Delphi on the Web: Off the Beaten Path

Last month we examined some general sites of interest to Delphi developers. This month, we'll examine some Delphi sites you may not know about: sites devoted to components, tools, techniques, and/or code; news sites; and special interest sites.

Developer's Corner Journal (http://www.dcjournal.com) is similar to some of the Pascal sites we examined last month. However, this site focuses on two Inprise tools: Delphi and C++Builder. Particularly rich in content, DCJ includes sections on beginners' issues, the Windows GUI, Internet programming, experts, database development, and much more.

Brad Stowers' Delphi Free Stuff (http://www.delphifreestuff.com) is just that. It includes over a dozen of Brad's components, as well as those of other developers. It also includes links, experts, tips, and examples demonstrating advanced techniques, including working with the Windows API. There's an open invitation and willingness to provide a home for other "freeware components that need a distribution point."

Conrad Herrmann's DAX FAQs at http://pweb.netcom.com/~cherrman/daxfaqs.htm is devoted to Delphi's ActiveX capabilities. It includes information and code examples related to Delphi's ActiveX Class Framework, covering Delphi 3 and 4. It covers bugs in IE4.01 and Delphi 3.02, as well as work-arounds.

The Delphi Pages at http://www.delphipages.com is an attractive site with a wealth of tools. The Delphi News portion, central to the site, is excellent. It includes pages devoted to applications, tips, components, chat, a Delphi forum, links, and more.

Delphi user groups can be an effective means of sharing information and programming techniques. But what about those who don't live close enough to such a group to participate? Enter the Virtual Delphi User Group, located at http://balticsolutions.com/vdug. VDUG provides a number of useful services. In addition to its monthly newsletters, it provides information on Delphi components and Internet sites.

The Search Site for Software Developers at http://developers.href.com is indispensable for Delphi developers. It provides a powerful means to search many Usenet and vendor newsgroups. You can find information on obscure topics, assessments of programming tools, and answers to tough questions. It also provides access to files on the Delphi Super Page and links to other top sites.

Richey's Delphi-Box at http://inner-smile.com/delphi4.htm contains a wealth of information and links. In addition to the expected links to Web sites, FTP sites, and Inprise sites, there are sections devoted to less-usual topics: Delphi user groups and Delphi job offers.

Would you like to be an advocate for Delphi? Check out The Delphi Advocacy Group at http://www.tdag.org. They define themselves as Delphi users and professionals who promote the most advanced Windows development tool.

You may recall my partner in the TAPI articles, Major Ken Kyler (see the July, August, and September 1998 issues of *Delphi Informant*). Ken is sponsoring new lists: a Delphi moderated list, a moderated Delphi Database list, Delphi Talk, and more. You can find out how to subscribe at http://www.kyler.com. His page includes information on Web page authoring, technical writing, and other topics.

The Tomes of Delphi Support Site at http://www.cyberramp.net/~jayres/ is a lot more than its title suggests. It does, of course, provide a good deal of updates and information related to the important series of books of which John Ayers (the owner of the site) is the principle author. But the most impressive aspect is an exhaustive page of links to third-party tools.

Advanced Delphi Programming at http://members.tripod.com/~delphipower/index.htm is also worth visiting. It has information on several 32-bit Delphi programming topics, such as working with shell extensions in Delphi; displaying the Properties page for a file, folder, or drive; and using *SHFileOperation* to copy files in Delphi.

Last year, I discussed Project Jedi, a project to develop and make available conversions of Windows APIs otherwise unavailable to Delphi developers. Of course, the Project Jedi site at http://www.delphi-jedi.org has information on this important endeavor. But the site also includes a good deal of additional information and links. There is an excellent tutorial written by Andreas Prucha on converting C headers; it provides a wealth of information on this difficult topic. This site also includes links to user groups and an interview with John Ayers.

I hope to revisit this important topic. In the meantime, I continue to invite your helpful input. ∆

— Alan C. Moore, Ph.D.

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

table of CONTENTS