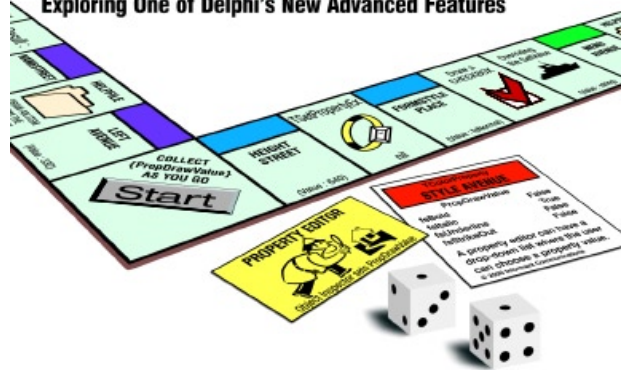


Owner-drawn Property Editors

Exploring One of Delphi's New Advanced Features



Cover Art By: Arthur Dugoni

ON THE COVER



5 OP Tech

Owner-drawn Property Editors — Ray Lischner

New in Delphi 5 is the ability for a property editor to draw anything to display a property's name and value, and if the property has a drop-down list, you can draw each list item, as Mr Lischner explains.

FEATURES



9 Greater Delphi

CORBA: Part II — Dennis P. Butler

Mr Butler ends his two-part exploration of CORBA development by describing how to build CORBA clients with Delphi (both early and late binding) and with JBuilder.



16 Visual Programming

Visual Form Inheritance: Part II — Rick Spence

Mr Spence winds up his two-part introduction to Delphi's woefully underdocumented VFI capabilities by creating a generic table maintenance application.



21 DBNavigator

Interfaces Revisited: Part I — Cary Jensen, Ph.D.

More than a method, less than a class, but not an object, the Object Pascal interface defies brief description. Dr Jensen is up to the task, however, and provides an updated introduction.



25 On the 'Net

SAX for Delphi — Keith Wood

Mr Wood explains the importance of SAX (Simple API for XML) and then demonstrates its power by building an impressive example application for Delphi versions 3, 4, and 5.



30 Sound + Vision

FormShaper — Peter Morris

Tired of the standard rectangular windows? Have an exotic UI request from a client? Mr Morris shares his FormShaper component, and shows us how to think outside the box.



REVIEWS

33 ASTA 2.1

Product Review by John Rendell

DEPARTMENTS

2 Delphi Tools

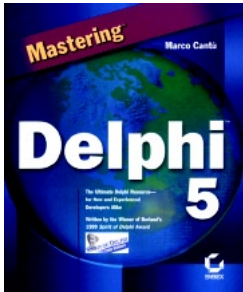
3 Newsline

36 File | New by Alan C. Moore, Ph.D.





Mastering Delphi 5
Marco Cantù
SYBEX



ISBN: 0-7821-2565-4
Price: US\$49.99 (1,085 pages)
Web Site: <http://www.sybex.com>

Woll2Woll Announces InfoPower 2000

Woll2Woll Software announced the availability of *InfoPower 2000*, a new version of its visual component suite for Delphi and C++Builder.

A major architectural change has been made to InfoPower's database design by providing true virtual dataset support. This allows developers to use any *TDataSet* natively (without requiring a *TtwDataSet* descendant). As a result, developers can directly use Delphi 5's new ADO and InterBase data objects, or any third-party engine.

InfoPower 2000 features a new hierarchical data inspector, a multi-purpose component for editing or viewing data. It can be used to group related fields (even from different datasets), provide a hierarchical view of the data, embed InfoPower or *1stClass* controls, or without a dataset just like the Delphi Object Inspector.

InfoPower 2000 provides the ability to create true paperless forms just like the real hard-copy form they are based on, using the new custom framing and glyph effects in the edit controls.

Developers can make the *TtwDBNavigator* control appear transparent with flat, transparent navigator buttons, each of which now contains

Invoice No	Buyer Name	Open	Payment Method	Purchase Date	Total Invoice	Balance Due
1	Cathy W...	<input checked="" type="checkbox"/>	Visa	11/16/1998	\$248.59	\$33.00
2	John Henry	<input type="checkbox"/>	Discover	05/09/1999	\$120.82	\$0.00
3	Steve Forrest	<input type="checkbox"/>	Credit Card		\$637.40	\$0.00
					\$1,006.81	\$33.00

TActionList support.

The new version also provides enhanced RichEdit word processor control by integrating Microsoft Word's Spell and Grammar Checker (Delphi 5 only).

Also, InfoPower 2000 includes advanced filtering capabilities, significant performance enhancements, and improved support for SQL parsing. When using ADO or Delphi 5's InterBase objects, it additionally supports filters on calculated fields in a

developer's dataset, wildcard searches, and more. Automatic Aliases for user-entered text allow filtering on mapped values that are stored in the database.

In addition, InfoPower 2000 also enhances its

grid with registry and .INI support for saving and loading user column positions and sizes. The grid now displays multiple-line cell-level tool tips when the cell text doesn't fit.

Woll2Woll Software

Price: InfoPower 2000 Standard, US\$199; InfoPower 2000 Professional, US\$299 (includes source code, C++Builder compatibility, and support for older versions of Delphi).

Phone: (800) 965-2965

Web Site: <http://www.woll2woll.com>

Name	John R Freeman
Address	2216 Peterson Drive Livermore, CA, 41557 USA
Street	2216 Peterson Drive
City	Livermore
State	CA
Zip	41557
Country	USA
RichEdit	This is the direct contact for the head of the manufacturing department. Contact him first.
Sabbatical	<input checked="" type="checkbox"/>
Employment Data	
Salary	\$22.5 Per Hour
PayType	Per Hour
PayRate	22.5
Schedule	10:02:52 AM->11:02:55 AM

ZieglerSoft Releases ZieglerCollection one v. 1.60 and Crt32 v. 2.05

ZieglerSoft released *ZieglerCollection one v. 1.60*, a new ver-

sion of the development tool for Delphi and C++Builder.

In addition to Delphi 5 support, new features have been added, such as forms, buttons, check boxes, etc. that can take any shape. All registered customers get free upgrades to the new version.

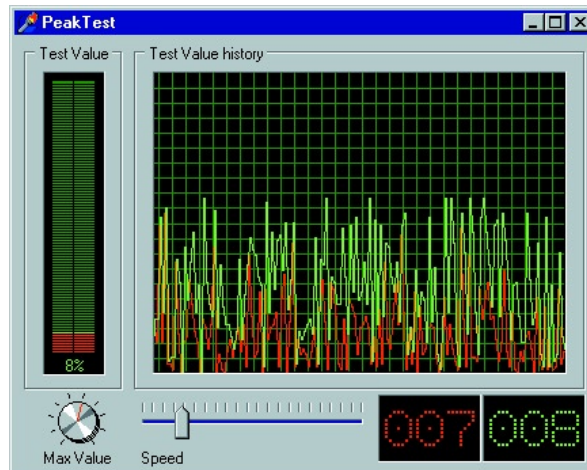
The company also released *Crt32 v. 2.05*, a new version of its tool for moving Turbo/Borland Pascal programs, using the old Crt unit, to Delphi. The product now supports Delphi 5.

ZieglerSoft

Price: US\$52 each.

Phone: (+45) 9811 3772

Web Site: <http://www.zieglersoft.com>





LWE Releases MATHEMATICS DLL TOOLKIT! Version 5.0

LWE Research, Inc. announced *MATHEMATICS DLL TOOLKIT! Version 5.0*, its unit conversion and financial math package for Delphi 5. *MATHEMATICS DLL TOOLKIT!* is a DLL-based toolkit of mathematical functions for the experienced programmer and the novice programmer alike.

MATHEMATICS DLL TOOLKIT! 5.0 provides SI and British numerical conversion functions for area, force, heat capacity, enthalpy, heat, energy, work, heat flux, heat flow, heat-transfer coefficients, length, mass, mass flux, molar flux, mass-transfer coefficients, power, pressure, temperature, thermal conductivity, viscosity, volume, and weight (in troy measurements) properties.

The toolkit also provides DLL functions for financial math cal-

culations, including compound interest, annuity future, and present value calculations (front- and back-end loading), continuous compounding, yield to maturity, multiple compounding, multiple compounding annuity future, and present value calculations (front- and back-end loading). It provides class wrappers to compute future value, present value, interest, number of years, and payment amount. Set three values and the class will compute the fourth. The package also provides component wrappers for docking with the Delphi 5 palette.

All functions in LWE Research's *MATHEMATICS DLL TOOLKIT!* are designed to provide 19 significant figures of accuracy. Working in Windows 95/98/NT, programmers can write the code, display any

output the way they want it, and let *MATHEMATICS DLL TOOLKIT!* do the calculations.

MATHEMATICS DLL TOOLKIT! comes with *MATHEMATICS DLL TOOLKIT!* DLL file, class DCU, and component DCU; an HTML manual explaining all functions; information on floating point limits for Borland and Microsoft compilers; a sample EXE describing how to attach to the DLL; sample EXE Delphi code describing how to use the DLL calls or classes; and notices of updates.

LWE Research, Inc.

Price: Single-user license, US\$100; professional license US\$300; site licenses and custom DLLs are available (call for more details).

Phone: (800) 201-4559

Web Site: <http://www.lweresearch.com>

ESB Announces ESB Professional Computation Suite 1.1.2

ESB Consultancy announced *ESB Professional Computation Suite 1.1.2* (ESBPCS), a collection of over 1,600 routines and over 50 components for Borland Delphi 4 and 5 that are aimed at making computations easier for the developer.

Areas covered include extended currency types, date and time manipulation, trigonometry, statistics, linear regression, probability distributions, special functions, optimized math, vectors, matrices, equation solving, fractions, business math, accounting, complex numbers, geometry, unit conversions, and more.

Components include a variety of custom-designed edit controls for the various data types, including Vector Editing and

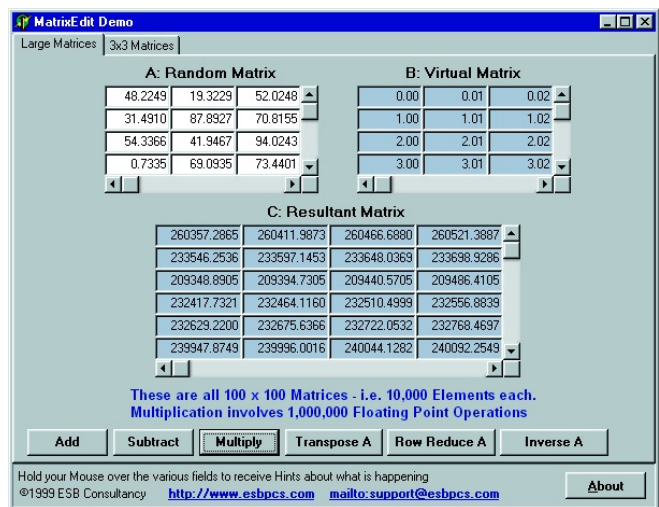
Matrix Editing. There are components to aid in statistics and unit conversion, and many of the components have data-aware versions.

ESB Consultancy

Price: US\$99 for single-developer license.

Phone: +61-8-9093-2133

Web Site: <http://www.esbpcs.com>



SkyLine Tools Announces ImageLib Corporate Suite 5.0

SkyLine Tools Imaging announced the release of *ImageLib Corporate Suite 5.0* for Delphi 5, the company's imaging toolkit for Delphi developers.

Features of the new version of ImageLib Corporate Suite include upgraded memory for the larger images produced by newer digital cameras; upgraded

TWAIN scanning that meets the specifications for newer scanners; a features package that allows annotations to be customized by the developer; a magnifying glass feature, which allows the user to zoom into a specific area of the image rather than zooming into the entire page (it also lets the user move the magnifying glass

around on the screen, and continues to magnify only the area under the "glass"); and an improved IIDocumentImage, with flicker-free annotations.

SkyLine Tools Imaging

Price: US\$599

Phone: (818) 346-4200

Web Site: <http://www.skylinetools.com>



March 2000



Inprise Announces VisiBroker 3.3 for Delphi

Scotts Valley, CA — Inprise Corp. announced the availability of VisiBroker 3.3 for Delphi. With VisiBroker 3.3 for Delphi, developers can create and deploy distributed applications in computing environments, including Windows and Linux. VisiBroker for Delphi provides a complete CORBA environment for Delphi. The product is available as a free download, for developers who have purchased Delphi 5 Enterprise Edition, at Inprise's Web site (<http://www.borland.com/visibroker/delphi>).

Inprise's VisiBroker for Delphi is designed to facilitate the development of CORBA applications that are scalable, flex-

ible, easily maintained, and based on industry standards. CORBA simplifies the building of applications that are distributed and interoperable across the Internet and multiple platforms, including Windows,

Tamarack Hosts Newsgroup Search Engine for Delphi and C++ Builder

Palo Alto, CA — Tamarack Associates is hosting a newsgroup search engine at <http://www.tamaracka.com> that indexes Delphi and C++ Builder newsgroups. The archive of messages goes back to May, 1997, and includes newsgroups from Borland, Advantage, TurboPower, Vista Software, Woll2Woll, and

Linux, and Java. CORBA is particularly important to the Linux community, where it is also used for inter-application communication in Linux desktop environments, including both GNOME and KDE.

others. Searches against the 1.5 million-record database are performed using Tamarack Associates' Rubicon 2 full-text search engine. Search results are returned in either date (most recent to least recent) or rank order and may be viewed by article or thread. The site focuses on Borland development tools.

Inprise Collaborates with Sun on Delivery of Java 2 Platform for Linux

Scotts Valley, CA — Inprise Corp. announced it has jointly produced a Linux version of the Java 2 Platform, Standard Edition (J2SE) with Sun Micro-

systems. The agreement enables programmers to develop and run applications based on the Java 2 platform and deploy them on Linux workstations and serv-

ers, as well as simplify moving existing Java applications to the Linux operating system.

Earlier this year, Inprise announced a free download of the beta JBuilder Just-In-Time compiler (JIT) for Linux. Now Sun is licensing the JIT from Inprise to complete the Java 2 Software Development Kit, necessary for running high-performance Java applications on the Linux OS.

The first release of the Linux port of the Java 2 platform developed by Sun and Inprise is available immediately at <http://developer.java.sun.com/developer/earlyAccess/j2sdk122/> and will be included in Inprise's JBuilder. Sun and Inprise plan to ship the final version of the Linux port in early 2000.

Inprise Announces Inprise Application Server 4.0 and VisiBroker for Java 4.0

New York, NY — Inprise Corp. announced the shipment and availability of Inprise Application Server 4.0 and VisiBroker for Java 4.0.

Based on open industry standards, the products deliver a foundation for customers to expand their presence on the Web and provide them with the technology infrastructure needed to support enterprise-strength Internet business applications.

Inprise Application Server combines the benefits of Enterprise JavaBeans (EJB) and CORBA, enabling customers to integrate existing IT resources with new, powerful Web applications. By leveraging the proven scalability and reliability of VisiBroker, customers can create e-business applications that can handle the high volume of transactions required for doing business on the Web.

The Inprise Application Server also provides comprehensive support for the Java 2 Platform, Enterprise Edition. Inprise is redefining the value that developers can expect from an appli-

cation server by combining the open, cross-platform, and cross-language framework of CORBA with the transactional business logic of EJB technology.

VisiBroker is the object request broker designed to facilitate the development and deployment of distributed enterprise applications that are scalable, flexible, easily maintained, and based on industry standards.

For more information, visit <http://www.borland.com>, or call (800) 632-2864.

Inprise/Borland Open-sources InterBase 6

Scotts Valley, CA — Inprise Corp. announced plans to jump to the forefront of the Linux database market by open-sourcing InterBase 6, the new version of its cross-platform SQL database. Inprise plans to release InterBase in open-source form for multiple platforms, including Linux, Windows NT, and Solaris.

The source code for InterBase 6 is scheduled to be published during the first part of this year.

The company also announced it plans to continue to sell and support InterBase 5.6 through normal distribution channels. Inprise plans to announce further details of its roll-out plans for the InterBase open-source project on its Web site (<http://www.inprise.com>).

InterBase 6 is a cross-platform SQL database designed for business-critical, mobile computing, and Internet-based applications on Linux, Windows NT, Solaris, and UNIX.





By Ray Lischner



Owner-drawn Property Editors

Exploring One of Delphi's New Advanced Features

The Object Inspector is, of course, familiar to all Delphi users. At the heart of the Object Inspector lie property editors; each property you see in the Object Inspector has a corresponding instance of a property editor class.

Among Delphi's advanced features is the ability to define new property editors, provide new functionality for existing properties, or define the method for setting and displaying new properties on new components. Before Delphi 5, the Object Inspector could display only text for a property value. New in Delphi 5 is the ability for a property editor to draw anything it wants to display a property's name and value. If the property has a drop-down list, you can draw each list item, too. This article tells you how to use the new owner-drawn feature of property editors.

Property Editor Refresher

A property editor is a class that inherits from *TPropertyEditor*. You register a property editor class for certain property types, property names, and components. The Object Inspector checks the type and name of each property it must display, and chooses an appropriate property editor class. Then it creates an instance of the class — one instance per property. When you select a different component, the Object Inspector frees all the property editor objects and creates new ones for the new component.

A property editor determines how to display a property's value and how the user can set a new property value in the Object Inspector. For example, *TIntegerProperty* calls *IntToStr* to display the integer property value as a string, and *StrToInt* when the user types a new property value. *TColorProperty*, on the other hand, also uses an integer-type property, but interprets the value of the integer as a color, mapping the color value to a name (such as *clBlack* or *clBtnFace*) if possible.

```
type
  TVisualFontProperty = class(TFontProperty)
  public
    procedure PropDrawValue(Canvas: TCanvas;
      const Rect: TRect; Selected: Boolean); override;
  end;
```

Figure 1: Class declaration for *TVisualFontProperty*.

A property editor implements its type-specific behavior by overriding one or more methods of *TPropertyEditor*. Most property editors override *GetValue*, which gets the property value as a string, and *SetValue*, which converts a string to a property value and sets the property's value in the selected component. For more information about writing property editors, see the *DsgnIntf.pas* file that ships with Delphi 5 (in the *Delphi5\Source\Toolsapi* directory by default), and Delphi 5 online Help (under "property editors, creating").

The Basics

At its most basic level, implementing an owner-drawn property editor is simply a matter of overriding the *PropDrawValue* method of *TPropertyEditor*. For example, *TColorProperty* overrides *PropDrawValue* to show a small color swatch in the Object Inspector. To understand how to use *PropDrawValue*, consider writing a new property editor for *TFont*-type properties. The new property editor will display the font in use by using the font to write the font's name.

Delphi already has a property editor, *TFontProperty*, which adds the ellipsis button in the Object Inspector so the user can select a font from the Windows font chooser dialog box. Derive your new property editor from *TFontProperty* as shown in Figure 1.

Delphi calls *PropDrawValue* when it needs to display the property value in the Object Inspector. It passes a canvas to draw on and the bounds of the drawing area. The *Selected* parameter isn't used, so you can ignore it.

Note that Delphi doesn't set the clipping region to the given rectangle. Be sure you confine your drawing to that rectangle, or else you'll be obscuring the values of other properties.

The only task that *TVisualFontProperty* does is to select the font to use for drawing the property value. It sets the font name, style, and color, but only if

the color is different from the background color. The font size is left as the default, so you don't run into problems when the font property has a very large or very small size. Figure 2 shows the implementation of *PropDrawValue*.

The property value is ordinarily (TFont), which is not very informative. Override the *GetValue* method to display something more helpful, say the font name and size, as shown in Figure 3.

You can draw anything you want on the canvas. For example, the property editor for icons and bitmaps is *TGraphicProperty*. It displays the property value as a boring string, e.g. (TIcon). The property editor would be more useful if it displayed a miniature icon. *TVisualGraphicProperty* overrides *PropDrawValue* to do just that.

The property editor for *TPicture* is similar, so the common work is done by a subroutine, *DrawGraphic*. This subroutine stretches the graphical object to fit in the space available in the Object Inspector. It maintains the original aspect ratio, and scales the image to fit in the smaller of the available height or width. Windows doesn't stretch icons, so *DrawGraphic* calls *StretchIcon* to draw the icon on a bitmap and stretch the bitmap. Figure 4 lists these subroutines.

DrawGraphic does all the hard work, so *PropDrawValue* is easy to write. It makes sure the property has an actual graphical object. If not, it lets the inherited method do its thing (namely, display (None) as the property value). Figure 5 shows the *PropDrawValue* method.

Owner-drawn Name

You can also override the *PropDrawName* method, which works the same way as *PropDrawValue*, but draws the property's name. Most properties don't need any special treatment for the property name, but one example I like is to show the *Name* property in boldface. It's an important property, and this makes it easy to find.

```
// Instead of a boring (TFont) font value, show the font
// name and size using the chosen font style. The user
// might have chosen a large size, so select only the font
// name, leaving the size as the default size. Set the font
// color only if it is different from the background color,
// or else the user could not see the font name.
procedure TVisualFontProperty.PropDrawValue(
  Canvas: TCanvas; const Rect: TRect; Selected: Boolean);
var
  Font: TFont;
begin
  Font := TFont(GetOrdValue);
  if Font <> nil then begin
    if ColorToRGB(Font.Color) <> ColorToRGB(c1BtnFace) then
      Canvas.Font.Color := Font.Color;
    Canvas.Font.Name := Font.Name;
    Canvas.Font.Style := Font.Style;
  end;
  inherited;
end;
```

Figure 2: Definition of *TVisualFontProperty.PropDrawValue*.

```
function TVisualFontProperty.GetValue: string;
var
  Font: TFont;
begin
  Font := TFont(GetOrdValue);
  if Font = nil then
    Result := inherited GetValue
  else
    Result := Format('%s, %d', [Font.Name, Font.Size]);
end;
```

Figure 3: Overriding the *GetValue* method.

Figure 6 shows the *TBoldComponentNameProperty* class and the *PropDrawName* method.

Drop-down Lists

A property editor can have a drop-down list where the user can choose a property value. Delphi uses the owner-drawn feature to improve the

```
// Windows doesn't stretch icons, so if the icon
// doesn't fit, draw it on a temporary bitmap
// and stretch the bitmap.
procedure StretchIcon(Canvas: TCanvas;
  const Rect: TRect; Icon: TIcon);
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
  try
    Bitmap.Height := Icon.Height;
    Bitmap.Width := Icon.Width;
    Bitmap.Canvas.Brush.Color := c1BtnFace;
    Bitmap.Canvas.FillRect(Rect);
    Bitmap.Canvas.Draw(0, 0, Icon);
    Canvas.StretchDraw(Rect, Bitmap);
  finally
    Bitmap.Free;
  end;
end;

procedure DrawGraphic(Canvas: TCanvas; const Rect: TRect;
  Graphic: TGraphic; const Value: string);
var
  R: TRect;
  HeightRatio, WidthRatio: Single;
begin
  Canvas.FillRect(Rect);
  // Fit the graphic into the given space. Maintain the
  // aspect ratio and adjust the height or width to fill
  // the given space.
  HeightRatio := (Rect.Bottom - Rect.Top) / Graphic.Height;
  WidthRatio := (Rect.Right - Rect.Left) / Graphic.Width;
  R := Rect;
  if HeightRatio < WidthRatio then
    R.Right := R.Left + Trunc(Graphic.Width * HeightRatio)
  else
    R.Bottom := R.Top + Trunc(Graphic.Height * WidthRatio);
  if (Graphic is TIcon) and
    ((HeightRatio > 1) or (WidthRatio > 1)) then
    StretchIcon(Canvas, R, TIcon(Graphic))
  else
    Canvas.StretchDraw(R, Graphic);
  // To the right of the graphic, let the inherited editor
  // draw the usual text, e.g. TIcon.
  R.Left := R.Right;
  R.Right := Rect.Right;
  R.Top := Rect.Top;
  R.Bottom := Rect.Bottom;
  Canvas.TextRect(R, R.Left+1, R.Top+1, Value);
end;
```

Figure 4: Drawing a graphical object on the Object Inspector's canvas.

```
procedure TVisualGraphicProperty.PropDrawValue(
  Canvas: TCanvas; const Rect: TRect; Selected: Boolean);
var
  Graphic: TGraphic;
begin
  Graphic := TGraphic(GetOrdValue);
  if (Graphic = nil) or Graphic.Empty or
    (Graphic.Height = 0) or (Graphic.Width = 0) then
    inherited
  else
    DrawGraphic(Canvas, Rect, Graphic, GetVisualValue);
end;
```

Figure 5: *PropDrawValue* for a graphical object.

drop-down lists for the *TColor* and *TCursor* property editors. You can do the same by overriding the *ListDrawValue*, *ListMeasureHeight*, and *ListMeasureWidth* methods, which are shown in **Figure 7**.

```

type
  TBoldComponentNameProperty =
    class(TComponentNameProperty)
    public
      procedure PropDrawName(Canvas: TCanvas;
        const Rect: TRect; Selected: Boolean); override;
    end;

procedure TBoldComponentNameProperty.PropDrawName(
  Canvas: TCanvas; const Rect: TRect; Selected: Boolean);
var
  Style: TFontStyles;
begin
  Style := Canvas.Font.Style;
  Canvas.Font.Style := Canvas.Font.Style + [fsBold];
  try
    inherited;
  finally
    // Restore the style so Delphi can draw
    // the property value.
    Canvas.Font.Style := Style;
  end;
end;

```

Figure 6: Drawing the Name property's name in boldface.

```

procedure ListDrawValue(const Value: string;
  Canvas: TCanvas; const Rect: TRect; Selected: Boolean);
procedure ListMeasureHeight(const Value: string;
  Canvas: TCanvas; var Height: Integer);
procedure ListMeasureWidth(const Value: string;
  Canvas: TCanvas; var Width: Integer);

```

Figure 7: Methods for owner-drawn drop-down lists.

```

// Draw an item in the drop-down list. Display the checked
// or unchecked box for each item.
procedure TSetPropertyEx.ListDrawValue(const Value: string;
  Canvas: TCanvas; const Rect: TRect; Selected: Boolean);
var
  IsChecked: Boolean;
  OrdValue: Integer;
begin
  OrdValue := GetOrdValue;
  IsChecked := GetEnumValue(EnumInfo, Value) in
    TIntegerSet(OrdValue);
  Canvas.FillRect(Rect);
  Canvas.TextRect(Rect, Rect.Left + Checked.Width + 2,
    Rect.Top + 1, Value);
  if IsChecked then
    Canvas.Draw(Rect.Left + 1, Rect.Top + 1, Checked)
  else
    Canvas.Draw(Rect.Left + 1, Rect.Top + 1, Unchecked);
end;

procedure TSetPropertyEx.ListMeasureHeight(
  const Value: string; Canvas: TCanvas;
  var Height: Integer);
begin
  if Height < Checked.Height then
    Height := Checked.Height;
end;

procedure TSetPropertyEx.ListMeasureWidth(
  const Value: string; Canvas: TCanvas;
  var Width: Integer);
begin
  Width := Width + Checked.Width + 2;
end;

```

Figure 8: Owner-drawn list for set elements.

ListDrawValue is similar to *PropDrawValue*, but now the *Selected* parameter means that the user has selected the list item. Delphi automatically sets the canvas colors for unselected and selected list items, so you can usually ignore the *Selected* parameter.

The *Value* parameter is the string to display. Delphi gets these strings by calling *GetValues*, one of the standard methods of *TPropertyEditor*. (You can learn more about *GetValues* in Delphi's online Help.)

Before the Object Inspector displays the list, though, it calls *ListMeasureHeight* and *ListMeasureWidth* to learn the size of each list item. Set the *Height* or *Width* parameter to the desired height or width. The drop-down list uses the maximum size of all list items and displays every item in the same-sized rectangle, so don't assume an item's height must match the height of the rectangle passed to *ListDrawValue*.

As the user scrolls through the list, Delphi calls *ListDrawValue* appropriately to draw the newly visible list items. The user might scroll forward and backward many times. If the list item takes a lot of time to draw, you should draw it in a separate bitmap and let *ListDrawValue* quickly display the bitmap.

The next example is a property editor for set-type properties. The drop-down list shows the set elements, and a check box next to each element tells you whether that element is a member of the set. The check box is one of the standard Windows check box bitmaps. The property editor retrieves the bitmap once and displays the checked or unchecked bitmap as needed. The global variables *Checked* and *Unchecked* store these bitmaps as *TBitmap* objects. The listing in **Figure 8** shows the owner-drawn list methods for *TSetPropertyEx*.

The same logic for drawing check boxes can be used to display each set element for displaying the values of Boolean-type properties.

```

// Draw a check box and the Boolean text label,
// i.e. True or False.
procedure DrawBoolCheckBox(Canvas: TCanvas;
  const Rect: TRect; const Value: string);
begin
  Canvas.FillRect(Rect);
  Canvas.TextRect(Rect, Rect.Left + Checked.Width + 2,
    Rect.Top + 1, Value);
  if Value = BooleanIdents[False] then
    Canvas.Draw(Rect.Left + 1, Rect.Top + 1, Unchecked)
  else
    Canvas.Draw(Rect.Left + 1, Rect.Top + 1, Checked);
end;

{ TSetElementPropertyEx }
// Display a check box for each item, showing whether it's a
// member of the set. The user cannot click to check or
// uncheck, but double-click works.
procedure TSetElementPropertyEx.PropDrawValue(
  Canvas: TCanvas; const Rect: TRect; Selected: Boolean);
begin
  DrawBoolCheckBox(Canvas, Rect, Value);
end;

{ TBoolPropertyEx }
// Display a check box for ByteBool, WordBool, and LongBool
// items. The user cannot click to check or uncheck,
// but double-click works.
procedure TBoolPropertyEx.PropDrawValue(Canvas: TCanvas;
  const Rect: TRect; Selected: Boolean);
begin
  DrawBoolCheckBox(Canvas, Rect, Value);
end;

```

Figure 9: Owner-drawn Boolean property editors.

TBooleanPropertyEx applies to properties of type Boolean. The *ByteBool*, *WordBool*, and *LongBool* properties are similar, but require a different property editor. Figure 9 shows the simple code needed for these property editors. The check box is slightly confusing because you expect to single-click the check box to change its value. Delphi doesn't support single-click, but a double-click invokes the property editor's *Edit* method. For a set element or Boolean property, a double-click toggles the property value. Perhaps Delphi 6 will support single-click interaction in property editors.

Using the Property Editors

The final task is to register these new property editors. Most of the editors are easy to register, but the new set-type property editor poses a problem. Each set is a distinct type, and you must register the property editor separately for each. Fortunately, Delphi has a little-known feature that lets you register a property editor for all set-type properties. Instead of registering a property editor for a single property type or name, you supply a property map function. The function takes an object and property information as arguments and returns the property editor class or *nil*. In this case, the map function checks the property type and returns the new set property editor for all properties whose type is *tkSet*. Figure 10 shows the *Register* procedure and the map function.

After you write the *Register* procedure, all that's left to do is to bundle the new property editors in a package and install the package in Delphi. Close existing forms to make sure you dispose of the old property editors. Open a new form and you can see the new property editors. Figure 11 shows an example of the visual font, icon, set, and Boolean property editors.

Other New Property Editor Features

This article covers owner-drawn property editors, but Delphi 5 has other new features, which you can explore on your own.

```
// Register the set property editor for all
// set-type properties.
function SetMapper(Obj: TPersistent; PropInfo; PPropInfo):
  TPropertyEditorClass;
begin
  if PropInfo.PropType.Kind = tkSet then
    Result := TSetPropertyEx
  else
    Result := nil;
end;

procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TFont), nil, '',
    TVisualFontProperty);
  RegisterPropertyEditor(TypeInfo(TGraphic), nil, '',
    TVisualGraphicProperty);
  RegisterPropertyEditor(TypeInfo(TComponentName),
    TComponent, 'Name', TBoldComponentNameProperty);
  RegisterPropertyEditor(TypeInfo(Boolean), nil, '',
    TBooleanPropertyEx);
  RegisterPropertyEditor(TypeInfo(ByteBool), nil, '',
    TBoolPropertyEx);
  RegisterPropertyEditor(TypeInfo(WordBool), nil, '',
    TBoolPropertyEx);
  RegisterPropertyEditor(TypeInfo(LongBool), nil, '',
    TBoolPropertyEx);

  RegisterPropertyMapper(SetMapper)
end;
```

Figure 10: Registering the new property editors.

GetVisualValue is similar to *GetValue*, but it can return a different string that is meant solely for displaying a value — not for editing. *GetValue* and *SetValue* work as a team, converting a property value to a string and back again. Sometimes, you want to display a property value with a different string than what you use to edit the property value, in which case, you override *GetVisualValue* to return the display string and *GetValue* to return the string for editing.

A new attribute, *paFullWidthName*, tells the Object Inspector to display the property name across the full width of the Object Inspector, leaving no room for the property value. At first glance, this seems like a strange attribute for a property editor, but many components have a property named *About*, whose sole purpose is to have an ellipsis button (*paDialog*), which brings up an About dialog box. This property has no meaningful value, so *paFullWidthName* might be helpful. Override *PropDrawName* and you can include your company logo (if it's small enough) next to the property name.

These new features aren't documented in the Help files, but if you have the Professional or Enterprise Edition, you can read the source code in `\Toolsapi\DsgnIntf.pas`. Using the property editors that come with Delphi as starting points, you can create interesting and effective property editors. ▲

The files referenced in this article are available on the Delphi Informant Magazine Complete Works Companion Disk in INFORM\00\MAR\DI200003RL.

Ray Lischner is the author of *Delphi in a Nutshell* [O'Reilly & Associates, 2000] and other books and articles about Delphi. He talks about Delphi and programming at conferences and user group meetings across the country. Ray also teaches Computer Science at Oregon State University.

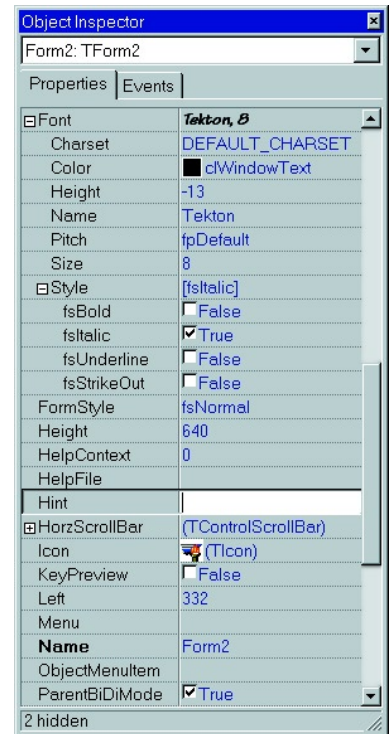


Figure 11: The owner-drawn property editors in action.





By Dennis P. Butler

CORBA

Part II: Creating Clients

Our server has been set up (last month in [Part I](#) of this two-article series), and should provide objects as necessary for any clients looking for an instance of *TOnlineAuction*. Now it's time to create a client to access and use this object. In CORBA, there are two ways a client can get an instance of a server object. The first is known as *early binding*, or *static binding*. This means the client has knowledge of what type of CORBA object it's going to interface with. This means that another file, known as the stub, will be used to handle the passing of data between the client and server — processes known as *marshaling* and *unmarshaling*, respectively. The complexity of the marshaling process is taken care of for us by the stub, which makes it much easier to implement.

The other way to access the server object from the client is known as late binding, or dynamic binding. Dynamic binding is also commonly referred to as DII, or Dynamic Invocation Interface. This means the client has no prior knowledge of the server object, and thus knows nothing about the structure of objects that it can access. An observation at this point is that the client stub doesn't get used, since the client doesn't have the facility to know the structure of server objects at design time. This knowledge is what is provided by the stub to the client.

The advantage of DII is that clients can be created that may never need to be rebuilt when a server object is changed; the client code can remain constant through many changes to the server object that it uses. This is done through the use of another CORBA construct known as the Interface Repository. This holds run-time type information about what is available to the client, and allows the client to use the available services. A drawback to this approach, as compared to early binding, is that it is more complex, slower, and requires more work for the developer.

In our Delphi example, it's still relatively simple, through the use of the type library files, and the *TAny* class. Since we know what server implementation we're looking for, we can access the methods directly using *TAny*. In actual production situations, the client may not know what server objects and methods are available, and may need to have more complicated code to accommodate this. For this article, both early and late binding clients will be created for use with the CORBA server we've created.

(There is also the capability in CORBA to provide a DSI, or Dynamic Skeleton Interface. Like DII for the server, this allows the CORBA servers to have no knowledge at compile time of what objects will be available to them. This is also done using an Interface Repository to store information. This technique won't be covered in this series, but it's important to know that it's available.)

Before we go any further, we should provide more explanation for these new CORBA terms that we have introduced:

- **Client Stub and Server Skeleton.** These two CORBA features are used to convert information to be passed between the client or server into the CORBA packet format to be sent over the network. The stub is a layer between the client and the ORB layer and provides a means for the client application to send information to the server. The server skeleton is a layer between the server and the ORB layer that converts the parameters and other information sent by the client, so it can be used by the server in performing the action the client requires.
- **Marshaling and Unmarshaling.** Marshaling is the process of converting parameter values and other information so they can be sent over the network. Marshaling is accomplished by the stub to send information to a server. Unmarshaling is the opposite process, where the server skeleton converts information that has been sent over the network into the parameter values and calls the appropriate function for the client.
- **Run-time Type Information.** RTTI is information available at run time about objects in a system. Delphi and CORBA both incorporate



Figure 1: The Online Auction client.

this feature. It's especially important when creating functionality that will wait until run time to see what types of objects are available for use, or for providing different types of functionality based on what objects are being used.

Since it's easier, the early binding client will be created first. By using the Type Library editor, we have all the files we need for our early binding client. The Type Library editor creates a stub file for us in the form of a TLB file. When we create the client, we'll need to add this file to the **uses** section of the form, so we have a reference to the structure of the server object. We also need to add `CorbaObj` to the **uses** section to perform the binding necessary to communicate across the ORB.

Figure 1 contains an image of our client. It has functionality to refresh the bid information, enter information for a new bid, and place the bid.

Early Binding Client

As mentioned earlier, the early binding client uses the type library file generated by the Type Library editor to get a reference to the CORBA object that our server will create. The client code we have to access and use this object is shown in Listing One (beginning on page 12). In this client, we implement all the methods from our server object. We're able to do this because we know the structure of the server object through the `IOnlineAuction` interface. We get the reference to the interface to the server object when the client starts by getting an instance from the CORBA factory. We can then perform any operation on the object.

We must take several steps before running this client. The ORB Smart Agent must be running on the network somewhere — on the server machine, or on some other machine. To do this, run:

```
osagent -C
```

from the command line, or run `VisiBroker Smart Agent` from the `VisiBroker` folder installed in the Delphi folder. The `-C` at the command line designates that the `osagent` will run in the taskbar; otherwise it will not appear there, so it may not be apparent whether it's running while you're testing. Once the ORB Smart Agent is running, start the server application.

Once the server has been started, it's a good idea to ensure that the server objects are available for any clients. The `VisiBroker` utility,



Figure 2: Selecting **Export to CORBA IDL**.

`osfind`, can be used to do this. Run `osfind` from the command line on the client machine to display a list of available objects within the subnet of the machine. This will verify that the client has access to the necessary server objects. For complicated implementations of CORBA, the `osagents` can be configured to look for server objects, or other `osagents` outside the current subnet. Although this isn't covered in this article, suffice it to say there are facilities to allow the client to look virtually anywhere for a server object, as long as the `osagents` have been set up correctly.

The final step is to run several clients. These should automatically get a reference to the server by including the type library file that was generated, and the client should have access to all server functions. In our example, we can launch many clients from different machines (within the same network subnet), and make successive bids against the server.

As you can see, not much is required to create an early-binding client to our server object in this simple example. DII is a little more complicated, as we'll see next.

Late Binding Client

As described earlier, the late binding client has no knowledge of the structure of available server objects at design time, and must use a facility called the Interface Repository to find what is available. In this example, we'll implement this client and describe the requirements, benefits, and drawbacks in using this approach.

Before we get to writing the client, there are a few requirements that must be met. First, the interface for the object must be registered with an interface repository. To do this, we must first have an IDL file. This can be created easily by returning to the Type Library editor and selecting **Export to CORBA IDL**. This is done by dropping down the last button on the right of the toolbar (see Figure 2). In this example, the CORBA IDL setting must be selected. The MIDL export setting will not work with the interface repository functionality we are going to use.

This will create the IDL file that corresponds to the server object defined previously. The filename will be `<ProjectName>.IDL` wherever the server application source has been saved. This IDL must then be registered with an interface repository. The `osagent` and the server should be started before starting the interface repository. Then, start the interface repository. At this point, we can load the IDL for our CORBA object into the repository, so clients can see it's available on one of the available servers. The interface repository can be started by running the following statement from the command line:

```
irep <Repository Name>
```

The repository name can be anything you want and will launch the Interface Repository application. Once it's open, you can either select `File | Load` from the main menu, and select the IDL file that was exported above, or run the following statement from the command line:

```
id12ir <IDL File Name>
```

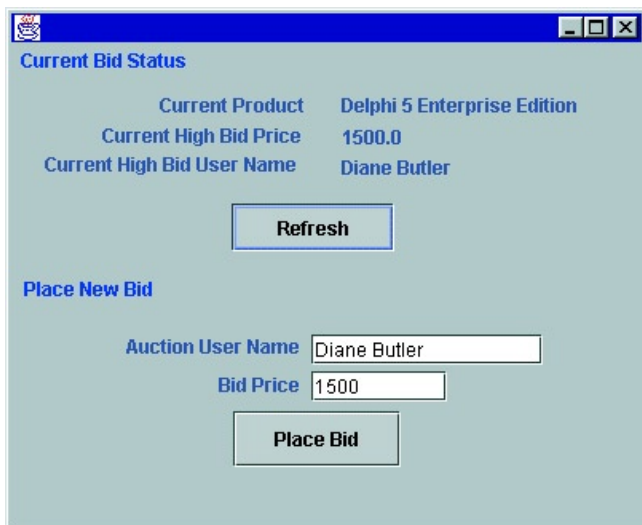



Figure 5: Java CORBA client.

a reference to the server object through the use of automatically generated Helper files. By doing this, an object reference is obtained, and is used in the same manner as the Delphi client. Helper files and other CORBA files are generated from the IDL2JAVA utility, which was run when the CServer.idl file was compiled. JBuilder uses this method to create the stub and skeleton files, as compared to using the Type Library editor in Delphi.

Once an object reference has been obtained, the code for the frame itself is similar to the Delphi application. The Delphi CORBA server has no knowledge of what language is being used for requests; Delphi and Java clients make virtually identical calls to the server object through their stub files. Our Java client could have been running on a UNIX machine located on a different continent from our Delphi server. As long as the CORBA subnet or osagents were configured correctly, these separate processes could talk to each other just as easily as if they were on the same machine. This simple Java/Delphi example provides a mere glimpse of the full potential of CORBA.

Conclusion

There's no doubt that CORBA will continue to gain momentum in enterprise computing due to its tremendous assets: flexibility, language independence, and a wide range of capabilities for virtually every distributed need. Delphi combines these assets with RAD development to make CORBA programming easier and faster for the developer, without sacrificing CORBA's capabilities. As we saw in these simple examples, Delphi is an ideal platform for setting up CORBA clients and servers for many types of applications.

Inprise developers also have advanced CORBA capabilities available through the use of the MIDAS technology. MIDAS allows users to create complicated queries easily through Delphi, and pass query results back from remote datasets using CORBA as the transportation format. This technology is especially powerful, because developers don't need to create complicated objects to hold query output; MIDAS automates this task, creating stub and skeleton classes automatically. The MIDAS technology is available in several Inprise development tools and will continue to play a key part in RAD CORBA development.

Going forward, Delphi developers can expect to see more CORBA support in new releases of Delphi. The IDL2PAS utility, when released, will give Delphi developers access to all CORBA features

and will not limit implementations to the framework that Delphi has provided. This will provide the best of both worlds: RAD development for standard CORBA tasks as covered in this series, and granular CORBA development for more specific and complicated implementations through IDL2PAS.

Delphi has long been regarded as the best Windows development tool. With the merging of CORBA technology to Delphi, this reputation will only grow as Delphi's capabilities now reach across previously unbreakable boundaries, such as multiple operating systems and languages. Δ

The files referenced in this article are available on the Delphi Informant Magazine Complete Works Companion Disk in INFORM\00\MAR\DI200003DB.

Dennis P. Butler is a Senior Consultant for Inprise Corp., based out of the Professional Services Organization office in Marlboro, MA. He has presented numerous talks at Inprise Developer Conferences in both the US and Canada, and has written a variety of articles for various technical magazines, including *CBuilderMag.com*. He can be reached at dbutler@inprise.com, or (508) 481-1400.

Begin Listing One — Implemented cclient.pas

```

unit cclient;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, CorbaObj, CServer_TLB, StdCtrls, Mask,
  Buttons;

type
  TfrmStaticCorbaClient = class(TForm)
    lblCurrentProduct: TLabel;
    lblBidPrice: TLabel;
    lblProduct: TLabel;
    lblCurrentHighBidPrice: TLabel;
    lblPrice: TLabel;
    btnRefresh: TBitBtn;
    edtBidPrice: TEdit;
    lblCustomerName: TLabel;
    edtUserName: TEdit;
    btnMakeBid: TBitBtn;
    lblCurrentHighBidUser: TLabel;
    lblUser: TLabel;
    lblBidStatus: TLabel;
    lblPlaceNewBid: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure btnRefreshClick(Sender: TObject);
    procedure btnMakeBidClick(Sender: TObject);
  public
    // Our interface to the server object.
    AuctionInterface : IOnlineAuction;
  end;

var
  frmStaticCorbaClient: TfrmStaticCorbaClient;

implementation

{$R *.DFM}

```



```

// On create, we immediately establish connection to server
// using interface defined in type library stub and do
// client initializations.
procedure TfrmStaticCorbaClient.FormCreate(Sender: TObject);
begin
  // Call factory to get reference to the server object.
  AuctionInterface :=
    TOnlineAuctionCorbaFactory.CreateInstance('');
  // Set the product name; resets it for each client. This
  // wouldn't be done in production, but it's done here to
  // demonstrate use of accessor methods created by Type
  // Library editor for object properties.
  AuctionInterface.Set_ProductName(
    'Delphi 5 Enterprise Edition');
  // Refresh with server to get latest information.
  btnRefresh.Click;
end;

// Refreshes information from server. This example doesn't
// implement server callbacks, so refreshes must be
// done manually.
procedure TfrmStaticCorbaClient.btnRefreshClick(
  Sender: TObject);
begin
  // Update price and customer name information for
  // current product.
  lblPrice.Caption := FloatToStrF(
    AuctionInterface.GetCurrentPrice, ffCurrency, 18, 2);
  lblUser.Caption := AuctionInterface.GetCurrentUser;
  lblProduct.Caption := AuctionInterface.Get_ProductName;
end;

// Call object to place a new bid against the server.
procedure TfrmStaticCorbaClient.btnMakeBidClick(
  Sender: TObject);
begin
  // Do some client-side data checking to save speed.
  if edtUserName.Text = '' then
    begin
      ShowMessage('You must enter a user name first. ');
      Exit;
    end;
  // Validate floating point value.
  try
    StrToFloat(edtBidPrice.Text);
  except
    ShowMessage('Invalid amount entered. ');
  end;
  // Place Bid.
  case AuctionInterface.PlaceBid(
    StrToFloat(edtBidPrice.Text), edtUserName.Text) of
    0 : ShowMessage('Bid amount insufficient. ');
    1 : ShowMessage('Bid successful! ');
  end;
  // Refresh information.
  btnRefresh.Click;
end;
end.

```

End Listing One

Begin Listing Two — Implemented cclient_dii.pas

```

unit cclient_dii;

interface

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

type
  TfrmDynamicCorbaClient = class(TForm)
    lblCurrentProduct: TLabel;

```

```

    lblBidPrice: TLabel;
    lblProduct: TLabel;
    lblCurrentHighBidPrice: TLabel;
    lblPrice: TLabel;
    btnRefresh: TBitBtn;
    edtBidPrice: TEdit;
    lblCustomerName: TLabel;
    edtUserName: TEdit;
    btnMakeBid: TBitBtn;
    lblCurrentHighBidUser: TLabel;
    lblUser: TLabel;
    lblBidStatus: TLabel;
    lblPlaceNewBid: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure btnRefreshClick(Sender: TObject);
    procedure btnMakeBidClick(Sender: TObject);
  public
    // Servers declared as type TAny; special type for
    // CORBA interfaces using DII.
    AuctionFactory,
    AuctionServer : TAny;
  end;

var
  frmDynamicCorbaClient: TfrmDynamicCorbaClient;

implementation

{$R *.DFM}

procedure TfrmDynamicCorbaClient.FormCreate(
  Sender: TObject);
begin
  try
    // Bind to ORB instance for object factory.
    AuctionFactory :=
      ORB.Bind('IDL:CServer/OnlineAuctionFactory:1.0 ');
    // Create reference to server object from factory.
    AuctionServer := AuctionFactory.CreateInstance('');
  except
    ShowMessage('Failed to connect to server. ');
    raise;
  end;
  // Refresh information on screen.
  btnRefresh.Click;
end;

procedure TfrmDynamicCorbaClient.btnRefreshClick(
  Sender: TObject);
begin
  // Update price and customer name information for
  // current product.
  lblPrice.Caption := FloatToStrF(
    AuctionServer.GetCurrentPrice, ffCurrency, 18, 2);
  lblUser.Caption := AuctionServer.GetCurrentUser;
  lblProduct.Caption := AuctionServer.Get_ProductName;
end;

procedure TfrmDynamicCorbaClient.btnMakeBidClick(
  Sender: TObject);
var
  r1BidPrice : Double;
  sBidUser : WideString;
begin
  // Do some client-side data checking to save speed.
  if edtUserName.Text = '' then
    begin
      ShowMessage('You must enter a user name first. ');
      Exit;
    end;
  // Validate floating point value.
  try
    StrToFloat(edtBidPrice.Text);
  except
    ShowMessage('Invalid amount entered. ');
  end;
  // Place Bid; use local variables as intermediaries
  // to calls.

```

```

sBidUser := edtUserName.Text;
r1BidPrice := StrToFloat(edtBidPrice.Text);
case AuctionServer.PlaceBid(r1BidPrice, sBidUser) of
  0 : ShowMessage('Bid amount insufficient. ');
  1 : ShowMessage('Bid successful!');
end;
// Refresh information.
btnRefresh.Click;
end;

end.

```

End Listing Two

Begin Listing Three — CorbaClient package

```

// Title:      Corba Java Client
// Version:    1.0
// Copyright:  Copyright (c) 1999
// Author:     Dennis Butler
// Company:    Inprise Corporation
// Description: CORBA Client for Delphi Server
package CorbaClient;

import java.util.*;
import java.awt.*;
import com.sun.java.swing.*;
import borland.jbcl.layout.*;
import java.awt.event.*;
import borland.jbcl.control.*;

public class Frame1 extends DecoratedFrame {
    public static void main(String[] args) {
        Frame1 frame1 = new Frame1();
        frame1.show();
    }

    // CORBA Object Factory and Object Interface.
    CServer.OnlineAuctionFactory pOnlineAuctionFactory;
    CServer.IOnlineAuction pOnlineAuction;

    Double r1Total = new Double(0.0);
    XYLayout xYLayout1 = new XYLayout();
    JLabel jLabel1 = new JLabel();
    JLabel jLabel2 = new JLabel();
    JLabel jLabel3 = new JLabel();
    JLabel jLabel4 = new JLabel();
    JLabel jLabelCurrentBid = new JLabel();
    JLabel jLabelCurrentProduct = new JLabel();
    JLabel jLabelCurrentUser = new JLabel();
    JButton jButton1 = new JButton();
    JLabel jLabel5 = new JLabel();
    JLabel jLabel6 = new JLabel();
    JLabel jLabel7 = new JLabel();
    JTextField jtfUserName = new JTextField();
    JTextField jtfBidPrice = new JTextField();
    JButton jButton2 = new JButton();

    public Frame1() {
        try {
            // Initialize the ORB.
            System.out.println("Initializing the ORB");
            org.omg.CORBA.ORB orb =
                org.omg.CORBA.ORB.init((String[]) null, null);

            // Bind to OnlineAuctionFactory object.
            System.out.println(
                "Binding to OnlineAuctionFactory object");
            pOnlineAuctionFactory =
                CServer.OnlineAuctionFactoryHelper.bind(
                    orb, "OnlineAuction");

            // Get an instance of OnlineAuction.
            System.out.println(
                "Getting an instance of OnlineAuction");
            pOnlineAuction =
                pOnlineAuctionFactory.CreateInstance(
                    "NewOnlineAuction");

```

```

}
catch(org.omg.CORBA.SystemException e) {
    System.err.println("System Exception");
    System.err.println(e);
}

try {
    jbInit();
}
catch (Exception e) {
    e.printStackTrace();
}
}

private void jbInit() throws Exception {
    xYLayout1.setHeight(311);
    xYLayout1.setWidth(400);
    jLabel1.setText("Current High Bid Price");
    jLabel4.setForeground(Color.blue);
    jLabelCurrentProduct.setText("< NA >");
    jLabelCurrentUser.setText("< NA >");
    jButton1.setText("Refresh");
    jButton1.addMouseListener(
        new java.awt.event.MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                jButton1_mouseClicked(e);
            }
        });
    jLabel5.setForeground(Color.blue);
    jLabel6.setText("Auction User Name");
    jLabel7.setText("Bid Price");
    jButton2.setText("Place Bid");
    jButton2.setText("Place Bid");
    jButton2.addMouseListener(
        new java.awt.event.MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                jButton2_mouseClicked(e);
            }
        });
    jLabel5.setText("Place New Bid");
    jLabelCurrentBid.setText("< NA >");
    jLabel4.setText("Current Bid Status");
    jLabel3.setText("Current High Bid User Name");
    jLabel2.setText("Current Product");
    this.setLayout(xYLayout1);
    this.add(jLabel1, new XYConstraints(57, 50, -1, -1));
    this.add(jLabel2, new XYConstraints(93, 31, -1, -1));
    this.add(jLabel3, new XYConstraints(21, 68, -1, -1));
    this.add(jLabel4, new XYConstraints(6, 3, -1, -1));
    this.add(jLabelCurrentBid,
        new XYConstraints(207, 51, 183, -1));
    this.add(jLabelCurrentProduct,
        new XYConstraints(207, 31, 182, -1));
    this.add(jLabelCurrentUser,
        new XYConstraints(207, 70, 176, -1));
    this.add(jButton1,
        new XYConstraints(138, 100, 102, 30));
    this.add(jLabel5, new XYConstraints(8, 146, -1, -1));
    this.add(jLabel6, new XYConstraints(72, 182, -1, -1));
    this.add(jLabel7, new XYConstraints(130, 206, -1, -1));
    this.add(jtfUserName,
        new XYConstraints(188, 182, 145, -1));
    this.add(jtfBidPrice,
        new XYConstraints(188, 205, 85, -1));
    this.add(jButton2,
        new XYConstraints(139, 230, 105, 35));
}

// Place new bid button.
void jButton2_mouseClicked(MouseEvent e) {
    Double r1Total = new Double(jtfBidPrice.getText());

    if (pOnlineAuction.PlaceBid(
        r1Total.doubleValue(),
        jtfUserName.getText())==0) {
        Message m = new Message(this, "Sorry",
            "Bid Amount
Insufficient");
        m.show();

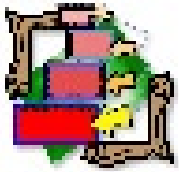
```

```
    }
    else {
        Message m2 = new Message(this, "Success",
                                "Bid Successful");
        m2.show();
    }
}

// Refresh button.
void jButton1_mouseClicked(MouseEvent e) {
    Double r1Total =
        new Double(pOnlineAuction.GetCurrentPrice());
    // Update price and customer name information
    // for current product.
    jLabelCurrentBid.setText(r1Total.toString());
    jLabelCurrentUser.setText(
        pOnlineAuction.GetCurrentUser());
    jLabelCurrentProduct.setText(
        pOnlineAuction.Get_ProductName());
}
}
```

End Listing Three





By Rick Spence



Visual Form Inheritance

Part II: Generic Table Maintenance

Last month we discussed the merits and mechanics of Visual Form Inheritance (VFI). I mentioned that one of the best ways to use VFI was to produce a generic table maintenance form. Regardless of the table you're maintaining, your users will need to perform the standard add, edit, and delete operations. VFI allows you to write most of this code and perform most of this form layout once, in a generic superclass. You can then create forms that inherit from this form, designing these new forms with data and components specific to the actual table you're editing.

This article describes a basic framework you can use as the basis for your data-editing forms. We'll discuss two layouts for the main form. If you choose to use this code, you can either start with one of these or design your own. Either way, you should be able to use most of the code and ideas in this article. And if you subsequently change your mind about the layout, you only have to make the change in one place because you're using VFI.

Generic Table Maintenance Features and Layout

The generic table maintenance form allows users to add new records, edit existing records, and delete and browse records. There are other functions you could implement here, but in the interest of restricting the length of this article, we'll leave those as an exercise for you.

Because we don't know which table the form will be working with, we can't lay out editing controls specific to a table. This will be done in the table-specific subclasses that inherit from this generic form. It's our goal, though, to place as much code and perform as much of the layout as possible in this generic form. Let's start with a basic form layout that should work with most tables.

The *PageControl* of our generic maintenance form contains two pages, labeled Form View and Browse View (see Figure 1). The Browse View page contains a database grid, which will list records from the table being edited. The Form View page is empty; subclasses will lay out this page with data-aware controls specific to the actual table being edited.

Note that the generic form includes a *TDataSource* component, but doesn't include any dataset (*TQuery*, *TTable*, or *TStoredProc*) components. There are two problems with placing a dataset component directly on the generic form:

- 1) You're tied to using that style of access, e.g. *TTable*. The way we have the form, it will work with any of these three *TDataSet* components.
- 2) You can't use existing datasets stored in a data module. Our approach allows dataset components to be located anywhere. They could be in a data module, or in a subclass of this generic form.

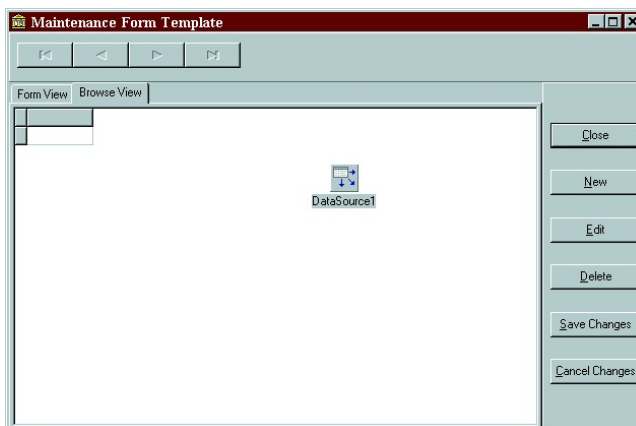


Figure 1: First layout of the generic database maintenance form.

The grid and the navigator are linked to the data source. As you'll see, it's the subform's responsi-

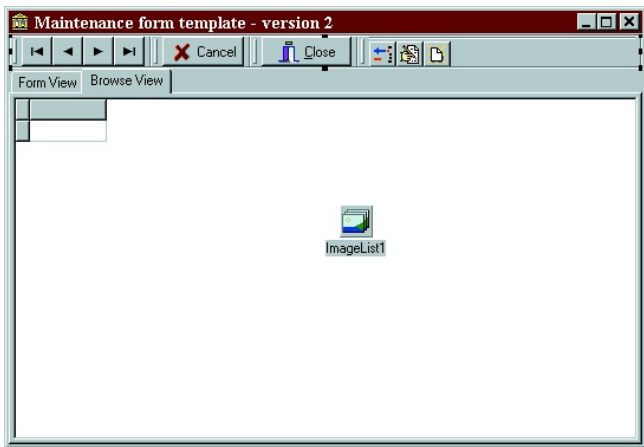


Figure 2: Second layout of the generic database maintenance form.

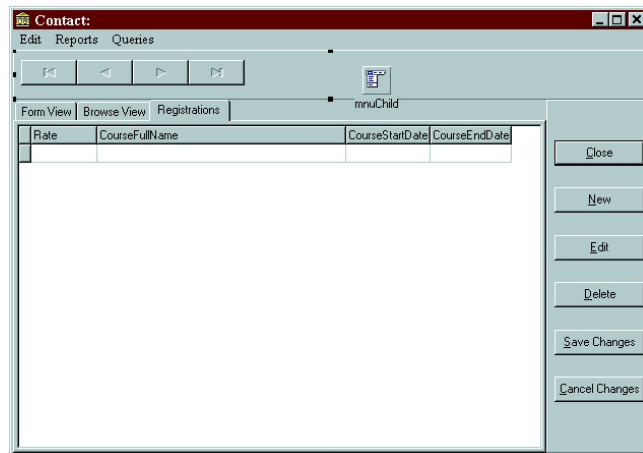


Figure 4: An additional page added to the child form.

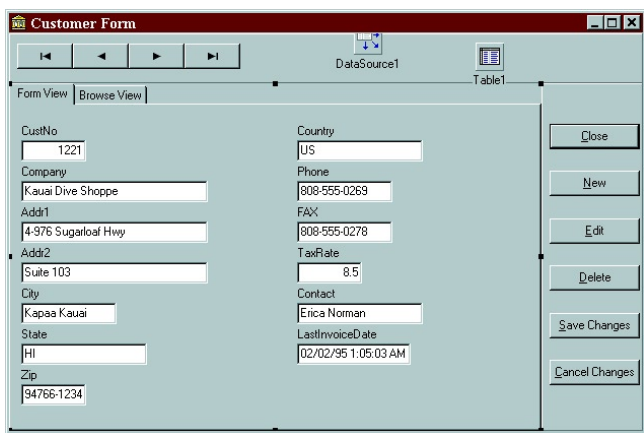


Figure 3: The Customer Form, inherited from the generic form, used to edit the Customer.db table from the DBDemos database.

bility to attach a dataset to the data source. This is the only link between the form and the actual table being maintained.

Figure 2 shows an alternative layout, which still uses the *PageControl* and two tab sheets, as well as a more modern *Coolbar* as the container for the action buttons. You probably have your own ideas for layout; whatever style you use, you should still be able to use the code in this article. Again, the only link between this form and the table being edited is through the data-source component.

The generic form is responsible for enabling/disabling the buttons and other components when appropriate. The design doesn't allow editing, adding, or deleting records when the *Browse View* tab is selected. All these actions must be performed with the *Form View* tab selected. Furthermore, it won't support auto editing. Auto editing, which is the default behavior of the data source, allows users to initiate editing by simply clicking in a data-aware control. I prefer to require users to explicitly request editing by clicking the *Edit* button. The *Edit* button can ensure that the *Form View* tab is selected, enable the data-aware controls (they should be disabled until the dataset is in edit mode), and move the dataset into edit mode.

Subclass Responsibilities

Obviously, the generic form requires the subform to perform certain functions. Most importantly, the subclass must associate the superclass' data-source component with an actual dataset. It must also lay out the subclass' *Form View* page with controls linked

to fields in the actual table. Figure 3 shows a table-specific form, inherited from the generic form, used to edit the Customer.db table from Delphi's DBDemos database.

The subform includes a *TTable* component configured to access the Customer table in the DBDemos database. The subform also sets the *DataSet* property of the *DataSource* component to reference this *DataSet*. Note that the *DataSource* component was introduced in the superclass, but the subclass is setting one of its properties. If you looked at the .DFM file for the subform, you wouldn't see the declaration of the *DataSource* component, but you would see its *DataSet* property being set. The subform, of course, is responsible for laying out the *Form View* page with data-aware controls specific to the table being edited (again, see Figure 3).

To summarize, the subclass' responsibilities are:

- 1) attaching the *DataSource* component to a dataset; and
- 2) laying out the *Form View* page with controls specific to the table being edited.

You could eliminate the second responsibility by performing a default layout if the user doesn't supply one. The superclass could implement a method to dynamically create data-aware controls from the fields in the table. The actual coding of this isn't too difficult; however, formatting issues arise when there are more fields as a result of more room on the tab sheet.

The most common mistakes programmers make when working with VFI is to forget to implement the subclass responsibilities. In this case, it's not too difficult to implement the subclasses. There are only two things they must do, but, in the real world, the interface between the superclass and the subclass can be more involved. In my designs, I always try to implement some default behavior in a superclass, allowing the subclasses to override this behavior if they need to. When the subclass absolutely must implement something — as in this case, where it must associate the *datasource* with a *dataset* — I write code in the superclass to verify that the subclass actually implemented it. We'll cover this in the next section.

Note that the layout inherited from the superclass is often only the starting point for subforms. In addition to adding controls to the *Form View* page, subforms can also add additional action buttons and pages (see Figure 4). The child form has added a *Registrations* page to show the registrations for each student, as well as a menu. Because this is an MDI child form, its menu will merge with its parent.

Generic Table Maintenance Implementation

Now we'll take a look at the generic superclass. Note that we'll present the class as a finished design, but that's rarely the way it works in practice. In the real world, it's doubtful you'll get the design perfect the first time; rather, it will evolve as you start working with subclasses. Class design is an iterative process. You're continually finding redundancies in subclasses and "removing" them by moving them up the class hierarchy. Also, note that the superclass is relatively simple. You'll probably want to extend it to allow ordering, filtering, and searching of records.

One of the requirements of the generic superclass is to disable the data-aware controls when the user isn't editing or adding

```
// Generic routine to enable/disable children of WinParent.
// Pass the WinControl whose children you want to set,
// True to enable the controls, and False to disable.
procedure TMaintainTemplateFrm.SetChildControls(
    WinParent: TWinControl; State: Boolean);
var
    i : Integer;
begin
    with WinParent do
        for i := 0 to WinParent.ControlCount - 1 do begin
            Controls[i].Enabled := State;
            // If this control can have children (i.e. it's a
            // TWinControl which has a Controls property),
            // enable/disable them.
            if Controls[i] is TWinControl then
                SetChildControls(TWinControl(Controls[i]), State)
        end;
    end;
end;
```

Figure 5: Generic recursive code to enable/disable a windowed control's children.

```
procedure TMaintainTemplateFrm.DeleteBtnClick(
    Sender: TObject);
begin
    if MessageDlg('Delete this record', mtConfirmation,
        [MBYes, MBNo], 0) = mrYes then
        DataSource1.DataSet.Delete;
end;

procedure TMaintainTemplateFrm.NewBtnClick(
    Sender: TObject);
begin
    DataSource1.DataSet.Append;
    EditMode;
end;

procedure TMaintainTemplateFrm.SaveBtnClick(
    Sender: TObject);
begin
    DataSource1.DataSet.Post;
    BrowseMode;
end;

procedure TMaintainTemplateFrm.CancelBtnClick(
    Sender: TObject);
begin
    DataSource1.DataSet.Cancel;
    BrowseMode;
end;

procedure TMaintainTemplateFrm.EditBtnClick(
    Sender: TObject);
begin
    EditMode;
    DataSource1.DataSet.Edit;
end;
```

Figure 6: Generic code implementing the Edit, Delete, New, Save, and Cancel buttons.

records. The easiest way to implement this is to disable the page, but that doesn't "gray out" the edit controls. The only way to have the edit controls grayed out is to explicitly disable each one. The superclass, of course, doesn't know the names of the controls on the Form View page. They're introduced in the table-specific subclasses and are different in each subform. There are two ways to implement this disabling:

- 1) Have each subform implement the disabling/enabling of its controls.
- 2) Write the code once in a generic manner in the superclass.

Naturally, we'll opt for the second approach. We need to implement a generic routine that will enable/disable all the child controls of the Form View page. Controls that can have other controls as children are based on the *TWinControl* class. *TWinControl* defines two properties you can use to access its children: *ControlCount*, which denotes the number of children, and *Controls*, which is the array of references to those controls.

To disable all the children of a *TWinControl* called *WinControl*, it's tempting to write:

```
for i := 1 to WinControl.ControlCount - 1
    WinControl.Controls[i].Enabled := False;
```

Although this works, it won't gray out the children of any other windowed control. For example, imagine you used this code to disable the controls on the Form View page, and one of those controls was a *GroupBox* with its own children. This code would not gray out the controls inside the group box. To implement this nested disabling, you need a recursive routine (see [Figure 5](#)).

Here's how you would call *SetChildControls* to disable the Form View page's children:

```
setChildControls(FormTab, False);
```

The generic form operates in one of two modes. In browse mode, the user is browsing records. In edit mode, they're either editing an existing record or adding a new record. The superclass defines two routines that take care of enabling/disabling controls when moving between the modes. When switching to edit mode, the superclass needs to set focus to a control on the Form View page. Because these controls are defined in the table-specific subclasses, the superclass doesn't know which controls exist. Therefore, it locates the first *TWinControl* on the page. The *GetFirstEditControl* method, shown in [Listing One](#) (beginning on page 19), shows this. The superclass declares *GetFirstEditControl* as **virtual** and **protected** so the subclasses can override it if necessary. This is a good example of the superclass providing default behavior and allowing subclasses to override it.

The last thing to implement is the code for the **Edit**, **Delete**, **New**, **Save**, and **Cancel** buttons. They follow a similar form: They move the form into the appropriate mode; access the dataset using *DataSource.DataSet*; and call the appropriate *DataSet* methods (see [Figure 6](#)). It couldn't be simpler.

There is additional code in the template we won't look at here. The *onCloseQuery* event, for example, asks the user whether to save or cancel changes before closing the form. There's also code that prevents the user from moving between pages when edits are pending (the Pascal file is available for download; see end of article for details).

```

procedure TMaintainTemplateFrm.FormCreate(Sender: TObject);
begin
    // Check to ensure child form is a good boy...
    // 1. DataSource.DataSet must be set.
    // 2. Form View tab sheet must be layed out.
    if DataSource1.DataSet = nil then begin
        ShowMessage(
            'Template: Forgot to set DataSource.DataSet');
        PostMessage(Self.Handle, WM_Close, 0, 0);
        Exit;
    end;

    if FormTab.ControlCount = 0 then begin
        ShowMessage('Template: Forgot to layout form tab');
        PostMessage(Self.Handle, WM_Close, 0, 0);
        Exit;
    end;
    // Always start on Browse View tab sheet.
    DataPage.ActivePage := BrowseTab;
    // And start in Browse mode.
    BrowseMode;
end;

```

Figure 7: Superclass code ensuring that the subclass adheres to its responsibilities.

Note the implementation of the **Delete** button. It simply asks the user to confirm deletion of the record. It's likely that subclasses will override this method and issue a message pertinent to the record being deleted. However, when you write code for this **Delete** button in a subclass, Delphi generates a call to the superclass method:

```

procedure TfrmCust.DeleteBtnClick(Sender: TObject);
begin
    inherited;
end;

```

The **inherited** keyword refers to calling a method with the same name in the superclass. In this case, you don't want this, so remove the call.

Defensive Coding

Whenever you write generic code — or code that will be used by another programmer — it's best to write that code as defensively as possible. Your code defines an interface for the user of the code, which may require the user to use your interface in a certain way or, as in this case, for a subclass to implement something. Your code can't assume the users are using it correctly; in fact, it should assume the worst.

Take the example of this generic maintenance form. Imagine another programmer attempting to use this form, but neglecting to associate the datasource with a dataset. When the user clicks any of the buttons that operate on the table, Delphi raises an exception, because your code is executing something like this:

```
DataSource.DataSet.SomeMethod
```

The other programmer will be able to find his or her error eventually, but it would be easier if the superclass code could detect that the subclass had not implemented what it was supposed to implement and report the error in a friendlier manner.

In this example, the superclass' *FormCreate* event can detect the omission and display a message stating just that. The awkward part is deciding what to do when you detect the error. Here, there's no point in having the form display, because the user can't do anything.

This isn't as easy as it sounds. The form's *FormCreate* event can't execute the *Close* method, and raising an exception doesn't help either. The exception is intercepted by the VCL, and the form proceeds to display and execute.

The best solution I've been able to come up with is to use the Windows API to post a **WM_CLOSE** message to the window. By posting the message, it's put into the queue for this window. When the window is finished with its creation process, it continues to process messages and then receives the close message. **Figure 7** shows the *FormCreate* event for the superclass, which checks to ensure the subclass has fulfilled its responsibilities. If not, it closes the form.

Conclusion

This article shows how to use Visual Form Inheritance to implement a generic table maintenance form. The superclass contains all the code and layout common to all forms. Subclasses can implement table-specific layout and code. The goal is to place as much code, and perform as much of the layout as possible in the superclass, which leads to more consistent user interfaces, faster development time, and fewer errors.

In the interest of length, we only implemented a small number of features in the superclass. You may want to extend it to allow users to select in which order they want to browse (e.g. populate a combo box with index names), search for, and filter records. Have fun. **▲**

The files referenced in this article are available on the Delphi Informant Magazine Complete Works Companion Disk in INFORM\00\MAR\DI200003RS.

Rick Spence is technical director of Database Programmers Retreat (<http://www.dp-retreat.com>), a training and development company with offices in Florida and the UK. You can reach Rick directly at 71760.632@compuserve.com. General inquiries should be directed to Dpr@Aug.com.

Begin Listing One — The GetFirstEditControl method

```

// Utility routines to enable/disable buttons and to set
// focus to first edit control.
procedure TMaintainTemplateFrm.EditMode;
var
    FirstEditControl : TWinControl;
begin
    // Enable all the controls on the Form View tab sheet.
    // Simply enabling/disabling the tab does not gray
    // out the controls.
    setChildControls(FormTab, True);
    DataPage.ActivePage := FormTab;
    NewBtn.Enabled := False;
    EditBtn.Enabled := False;
    DeleteBtn.Enabled := False;
    SaveBtn.Enabled := True;
    CancelBtn.Enabled := True;
    DbNav.Enabled := False;
    // Set focus to first control, on the Edit tab, which can
    // receive focus.
    FirstEditControl := GetFirstEditControl;
    FirstEditControl.SetFocus;
end;

```

```
procedure TMaintainTemplateFrm.BrowseMode;
begin
  // Disable all the controls on the Form View tab sheet.
  // Simply enabling/disabling the tab does not gray
  // out the controls.
  setChildControls(FormTab, False);
  NewBtn.Enabled := True;

  EditBtn.Enabled := not DataSource1.DataSet.Eof;
  DeleteBtn.Enabled := not DataSource1.DataSet.Eof;

  SaveBtn.Enabled := False;
  CancelBtn.Enabled := False;
  dbNav.Enabled := True;
end;

// Return the first TWinControl on the Form View tab.
function TMaintainTemplateFrm.GetFirstEditControl:
  TWinControl;
var
  lFound : Boolean;
  i : Integer;
begin
  lFound := False;
  i := 0;
  while (not lFound) and (i <= FormTab.ControlCount - 1) do
    begin
      lFound := (FormTab.Controls[i] IS TWinControl);
      if not lFound then
        i := i + 1;
    end;
  Assert(lFound, 'Template: No windowed controls found');
  Result := TWinControl(FormTab.Controls[i]);
end;
```

End Listing One





By Cary Jensen, Ph.D.

Interfaces Revisited

Part I: Declarations, Implementation, and Method Name Resolution

I first wrote about interfaces in the [April, 1998 issue](#) of *Delphi Informant Magazine*. There have been several important developments in the almost two years since. The first is that Delphi 4 introduced a new and important way to implement an interface in a class. The second — and arguably more important — is that the use of interfaces is quickly extending well beyond its original purpose: the native support for COM (Microsoft's Component Object Model). As a result, it seems like the right time to revisit this important topic.

But just what is an interface and why is it important? An interface is a definition of methods and properties that can be implemented by a class. Interfaces provide for assignment compatibility (often called polymorphism) between different objects that implement a common interface. More importantly, interfaces provide for polymorphism without the constraints normally associated with polymorphism through inheritance. Specifically, although two classes do not inherit a particular method from a common

ancestor, they can be treated polymorphically with respect to that method if that method is defined for an interface that both classes implement. Another way to look at it is that interfaces permit you to define an interface (methods and properties) completely independent of an object's implementation.

This article begins by discussing why interfaces are important in Object Pascal, and continues with a detailed look at the characteristics of interfaces. This discussion includes interface declarations, interface implementation requirements, and method name resolution. In Part II, this series will continue with an explicit example of interface usage, including how to use the interface implementation by delegation introduced in Delphi 4.

Why Add Interfaces to Delphi?

The easy answer is to provide native support for COM. However, the support for COM provided for by interfaces is a by-product of their function. Specifically, interfaces provide an elegant mechanism for treating objects polymorphically, even when they have no common ancestors.

This concept is difficult to describe, but an example can help illustrate it. Consider a situation where you want to create a group of user-interface (UI) components that share a common capability, such as being able to load their string data from a resource file. Furthermore, you might want to introduce a new method, named *LoadStrings*, in each of your new UI components, and from the implementation of this method, load that component's strings from a resource file, possibly based on the component's *Tag* property.

```
// Pseudocode.
interface
uses controls, stdctrls;

type
  TNewButton = class(TButton)
  public
    procedure LoadStrings; virtual;
  end;
  TNewLabel = class(TLabel)
  public
    procedure LoadStrings; virtual;
  end;

implementation

procedure TNewLabel.LoadStrings;
begin
  // Code to load the NewLabel's strings
  // from a resource file.
end;

procedure TNewButton.LoadStrings;
begin
  // Code to load the NewButton's strings
  // from a resource file.
end;
```

Figure 1: While this code compiles, it's not possible to call *LoadStrings* in a polymorphic fashion.

So far, so good. However, another aspect of this framework is being able to generically instruct each of your components to initiate their loading processes. In Delphi 1 and 2, where interfaces had not yet been introduced, this would have been all but

```
// Pseudocode.
interface

uses controls, stdCtrls;

type
  TLoadable = class(TObject)
  public
    procedure LoadStrings; virtual; abstract;
  end;
  TNewButton = class(TButton, TLoadable)
  public
    procedure LoadStrings; override;
  end;
  TNewLabel = class(TLabel, TLoadable)
  public
    procedure LoadStrings; override;
  end;

implementation

procedure TNewLabel.LoadStrings;
begin
  // Code to load the NewLabel's strings
  // from a resource file.
end;

procedure TNewButton.LoadStrings;
begin
  // Code to load the NewButton's strings
  // from a resource file.
end;
```

Figure 2: If Delphi supported multiple inheritance, you could create another class (*TLoadable*) that includes an abstract, virtual *LoadStrings* method.

```
// Code sample.
interface

uses stdCtrls;

type
  ILoadable = interface
    procedure LoadStrings;
  end;
  TNewButton = class(TButton, ILoadable)
  public
    procedure LoadStrings;
  end;
  TNewLabel = class(TLabel, ILoadable)
  public
    procedure LoadStrings;
  end;

implementation

procedure TNewLabel.LoadStrings;
begin
  // Code to load the NewLabel's strings
  // from a resource file.
end;

procedure TNewButton.LoadStrings;
begin
  // Code to load the NewButton's strings
  // from a resource file.
end;
```

Figure 3: This code includes one interface declaration and two class declarations.

impossible. Consider the pseudocode shown in [Figure 1](#) (which will compile).

While the code does compile, it's not possible to call the *LoadStrings* method of these objects in a polymorphic fashion. For example, the following method won't compile:

```
procedure DoLoadStrings(LoadableObject: TControl);
begin
  LoadableObject.LoadStrings;
end;
```

The problem is that, although *LoadStrings* may be a legitimate method of a particular *TControl* instance you pass to *DoLoadStrings*, it's not a method of the *TControl* class. Only if *LoadStrings* were a public or published method of *TControl* would this method compile correctly.

The Multiple Inheritance Approach

If Delphi supported multiple inheritance, this problem could be solved fairly easily. Specifically, you could create another class (let's call it *TLoadable*) that includes an abstract, virtual *LoadStrings* method. Then, when you declare each of your custom UI components, you could specifically declare them to descend from both their natural ancestor (*TNewButton* from *TButton* and *TNewLabel* from *TLabel*) and *TLoadable*. The pseudocode shown in [Figure 2](#) demonstrates how this might look.

Now all you would have to do is change the signature of *DoLoadStrings* to look like the following:

```
procedure DoLoadStrings(LoadableObject: TLoadable);
begin
  LoadableObject.LoadStrings;
end;
```

Because this re-written method takes a *TLoadable* object as its parameter, and because *TLoadable* objects have a visible *LoadStrings* method, everything should work fine. The only problem is that Delphi doesn't support multiple inheritance, and, consequently, this code won't compile.

The Solution: Interfaces

This is where interfaces come in. An interface is a declaration not unlike the abstract virtual *TLoadable* class shown in [Figure 2](#). Furthermore, an interface can be used by two or more classes to make those classes assignment compatible, in the same way that inheritance provides polymorphism.

Consider the code sample shown in [Figure 3](#). This code includes one interface declaration, named *ILoadable*, as well as two class declarations. Each of these classes implement the *ILoadable* interface.

In the language of interfaces, we say that both *TNewButton* and *TNewLabel* implement the *ILoadable* interface. Furthermore, because both of these classes implement a common interface, they are assignment compatible with an interface reference. Therefore, we can make this framework work generically by changing the single parameter of our *DoLoadStrings* method to be of an *ILoadable* type. The following code segment demonstrates how this completed method looks:

```
procedure DoLoadStrings(LoadableObject: ILoadable);
begin
  LoadableObject.LoadStrings;
end;
```

Because *DoLoadStrings* can take any *ILoadable* object, the following code fragment is completely legal:

```
var
  NewButton1: TNewButton;
  NewLabel1: TNewLabel;
begin
  NewButton1 := TNewButton.Create(Application);
  DoLoadStrings(NewButton1);
  NewLabel1 := TNewLabel.Create(Application);
  DoLoadStrings(NewLabel1);
```

Interface Declarations in Object Pascal

As you can see in the preceding sample code, an interface is declared in a type declaration, much like a class. However, interface declarations are different from class declarations in a number of important ways. For example, an interface consists only of method and property declarations. There are no member fields in an interface.

The member fields of a class are used for holding data in an instance of a class. Interfaces, unlike classes, can never be instantiated. Therefore, an interface can never hold data, which is why an interface cannot contain member field declarations.

That interfaces have no member fields restricts how interface properties are declared. Specifically, while a class can implement the read and write parts of a property by using direct access, interfaces can only use accessor methods. Specifically, when a property is declared in an interface, the read part of the property is specified using a function method, and the write part is defined using a procedure method.

The following is an example of a simple interface declaration that contains a *ShowMessage* method and a *MessageText* property. The remaining two methods, *GetMessageText* and *SetMessageText*, are also considered methods of the interface:

```
type
  IShowMessage = interface
    function ShowMessage: Boolean;
    function GetMessageText: string;
    procedure SetMessageText(Value: string);
    property MessageText: string
      read GetMessageText write SetMessageText;
  end;
```

By comparison, a class declaration can specify a member field in both the read and write parts. This is called *direct access*, because an object reading the property will be reading directly from the member field, and any object writing to the property will be writing directly to the member field.

All methods in an interface declaration are considered virtual and abstract by definition. Consequently, you never use the virtual or abstract directives in an interface declaration. In addition, there are no visibility identifiers (public, published, and so forth) in an interface declaration. All methods and properties declared in an interface are treated as though they were public, although a class implementing the methods can use visibility identifiers to control their visibility in the class.

A class implements an interface by including the name of the interface in the parentheses that follow the class keyword in its type declaration. This interface name is separated from the ancestor class name by a comma. Furthermore, a single class can implement two or more interfaces. When more than one interface is being implemented,

you include a comma-separated list of those interfaces following the ancestor class name.

Implementing the Methods of an Interface

A class that implements an interface is required to provide for the implementation of every method declared in that interface. This implementation may be provided either by explicit declaration or by inheritance from an ancestor class. For example, if the *TNewLabel* class is declared to implement the *ILoadable* interface, and this interface declares a single method named *LoadStrings*, then the *TNewLabel* class must either inherit a method with an appropriate signature named *LoadStrings*, or it must explicitly declare and implement it. Furthermore, if the inherited *LoadStrings* method is an abstract one, a concrete implementation of this method must be provided for in the *TNewLabel* class implementation.

Although a class that implements an interface must implement all methods declared in that interface, it is not required to implement all, or even any, of the interface properties. For example, in the *IShowMessage* interface described previously, a class implementing *IShowMessage* doesn't need to have a *MessageText* property.

This point can be confusing, but makes a lot of sense. The interface property belongs to the interface. Whether the implementing object contains the property is irrelevant. In fact, to make matters worse, an object that implements an interface can have a property that has the same name as a property declared in the interface, and yet the interface property and the object property can be completely unrelated. The compiler can tell which property is being accessed based on whether the qualifier of the property is an interface reference (in which case the interface's accessor methods are invoked), or an object reference (in which case whatever mechanism is used by the object to implement the property is used).

The Interface Hierarchy

Interfaces are organized in a hierarchy, much like Delphi classes. In Delphi, all interfaces, with the exception of *IUnknown*, descend from an existing interface. *IUnknown* is the highest-level interface, meaning that all interfaces necessarily descend from it. When you see an interface declaration that doesn't specify an ancestor declaration (like the *IShowMessage* interface declared in the previous code sample), the interface descends from *IUnknown*. Interfaces that descend from an interface other than *IUnknown* include the ancestor interface in the interface type declaration. This is demonstrated in the following interface declaration:

```
type
  INewInterface = interface(IDispatch)
  ...
```

When a class implements an interface, it is responsible for implementing not only all the methods of that interface, but any of that interface's ancestor interface methods. Consider the following declaration, which contains the Object Pascal declaration of the *IUnknown* interface:

```
IUnknown = interface
  ['{ 00000000-0000-0000-C000-000000000046 }']
  function QueryInterface(const IID: TGUID; out Obj):
    HRESULT; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
end;
```

Knowing that *IShowMessage* descends from *IUnknown*, any class that implements *IShowMessage* must not only implement the three

ShowMessage methods, but also the three *IUnknown* methods. Fortunately, as described earlier in this article, that implementation can be inherited. For example, the *TComponent* class implements the *QueryInterface*, *_AddRef*, and *_Release* methods. As a result, any object that descends from *TComponent* can implement the *ShowMessage* method by declaring and implementing only the three *ShowMessage* methods. The *IUnknown* method implementations are satisfied by inheritance.

Interface References

An interface reference can point to an instance of an object that implements the interface. Interface references can be variables, formal parameters, object properties, and even return values from functions. In the *DoLoadStrings* procedure described earlier, the interface reference was a formal parameter (*LoadableObject*).

You use an interface reference to invoke the methods of the interface, as well as read and/or write the properties of the interface. Doing so invokes the particular implementation of the method or property accessor methods of the object being referred to. For example, passing a *TNewButton* object to the *DoLoadStrings* causes the particular implementation of *LoadStrings* defined by the *TNewButton* class to be invoked. By comparison, passing a *TNewLabel* to *DoLoadStrings* results in the *LoadStrings* behavior defined within *TNewLabel*. However, an interface reference is limited to working with only those properties and methods defined by the interface; any methods, properties, or fields of the object not part of the interface definition cannot be accessed from an interface reference.

Consider again the *DoLoadStrings* method. Although you can pass any *ILoadable*-implementing object to this method, you can only use the formal parameter *LoadableObject* to invoke the *ILoadable* methods. Specifically, even though you may pass a *TNewButton* instance to *DoLoadStrings*, the compiler won't permit you to access the *Enabled* property of the *TNewButton* from *LoadableObject* — or any other *TNewButton* property for that matter. Only *LoadStrings*, *QueryInterface*, *_AddRef*, and *_Release* can be invoked, because they are the only methods of the *ILoadable* interface. (Remember that *ILoadable* descends from *IUnknown*, by definition.)

One more point concerning interface references is in order: You can never cast an interface reference as an object. For example, you cannot cast *LoadableObject* to *TNewButton*.

Interface Method Name Resolution

As you've already learned, any object that implements an interface is required to implement the methods defined within that interface. This can pose a problem if the interface declares a method whose name is already in use by an object that needs to implement the interface.

For example, imagine there is a class named *TMessageObject* that must implement the *ShowMessage* interface, but has inherited the method *ShowMessage* from its ancestor. This is a problem if the inherited method is conceptually different from the interface method of the same name. In this case, it's necessary to implement a method corresponding to the interface method, and distinguish it from the inherited method.


Delphi provides for the mapping of interface methods onto implemented methods of a different name. For example, consider the following declaration of *TMessageObject*:

```
type
  TMessageObject = class(TParentMessageObject, IShowMessage)
    FMessText: string;
    function IShowMessage.ShowMessage := DisplayMessage;
    function GetMessageText: string;
    procedure SetMessageText(Value: string);
    function DisplayMessage: Boolean;
  end;
```

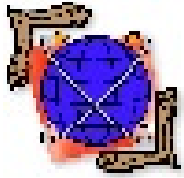
This declaration specifies that *TMessageObject* descends from *TParentMessageObject*, and implements *ShowMessage*. Because *TMessageObject* already has a *ShowMessage* method (by inheritance), it's necessary to map the *ShowMessage.ShowMessage* method to a different method name, which in this case is *DisplayMessage*. When using method resolution to map an interface method onto a new method name, the interface method — and the one to which it is being mapped — must have the same argument list and return value.

Given the preceding type declaration, if the *ShowMessage* method for an instance of *TMessageObject* is called using an object reference (a reference of type *TMessageObject*), the inherited *ShowMessage* method is executed. However, if an instance of *TMessageObject* is assigned to an *ShowMessage* variable and the *ShowMessage* method is called, the *DisplayMessage* method will execute.

Conclusion

This discussion will continue in **Part II** of this series. Part II will discuss how using objects with interface references differs significantly from standard object usage. You will also find a detailed discussion of interface implementation by delegation introduced in Delphi 4. 

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 Magazine*, and an internationally respected trainer of Delphi and Java. For more information, visit <http://www.jensendatasystems.com>, or e-mail Cary at cjensen@compuserve.com.



By Keith Wood



SAX for Delphi

Demonstrating XML's Simple API Standard

In last month's issue of *Delphi Informant Magazine*, I introduced XML and explained how to parse it using the Microsoft XML parser ("Generating XML"). This involved using a customized application specific to this parser. In the XML world, there's a standard called SAX (Simple API for XML), which defines a set of interfaces that allow XML parsers and document handlers to interoperate easily. This article sets out the Delphi version of SAX, and provides a native XML parser and a wrapper for the Microsoft version. A simple XML viewer is used to demonstrate the functionality.

XML in Brief

XML is a meta-language, i.e. a language for specifying other languages. It's a subset of SGML, designed to be powerful, yet easy to use. XML documents consist of tags, enclosed in angle brackets (<, >), that contain text and/or other tags. They have a strong hierarchical structure, facilitating the automatic processing of their contents. Using XML, we can declare exactly what that structure is in domain-specific ways.

The definition of that structure is to be found in the document type definition (DTD), which is usually kept in a separate document. Each XML document that claims to conform to a particular DTD must have a reference to it at the start of the file. For smaller documents, the DTD can be specified within the XML file, or it can be absent

altogether. XML documents without DTDs can still be usefully processed, they just don't claim to obey any structure.

XML has advantages over HTML in that it defines the structure and meaning of the embedded data, whereas HTML simply describes how to present that data without knowing what it means. Thus, XML documents allow for the easy interchange of data and can be processed in meaningful ways automatically — searching for particular elements, formatting for any number of different output formats, or exported from one program for import into a second application and further processing there.

SAX

SAX was developed by the XML-DEV mailing list under the leadership of David Megginson. The aim was to produce an event-driven interface to allow plug-and-play between the actual parser and the application that used the contained data. Details of the standard and a Java implementation are available at <http://www.megginson.com/SAX>.

SAX defines seven interfaces, two classes, and two exceptions (see the table in Figure 1). There are four handler interfaces, one each for resolving entities (*EntityResolver*), handling entities and notations (*DTDHandler*), dealing with the document structure and content (*DocumentHandler*),

SAX name	Type	Delphi name
<i>AttributeList</i>	interface	<i>ISAXAttributeList</i>
<i>DTDHandler</i>	interface	<i>ISAXDTDHandler</i>
<i>DocumentHandler</i>	interface	<i>ISAXDocumentHandler</i>
<i>EntityResolver</i>	interface	<i>ISAXEntityResolver</i>
<i>ErrorHandler</i>	interface	<i>ISAXErrorHandler</i>
<i>Locator</i>	interface	<i>ISAXLocator</i>
<i>Parser</i>	interface	<i>ISAXParser</i>
<i>HandlerBase</i>	class	<i>TSAXHandlerBase</i>
<i>InputSource</i>	class	<i>TSAXInputSource</i>
<i>SAXException</i>	exception	<i>ESAXException</i>
<i>SAXParseException</i>	exception	<i>ESAXParseException</i>

Figure 1: SAX definitions and Delphi equivalents.

and lastly, fielding any errors (*ErrorHandler*). These handlers are brought together in the parser, described by its own interface (*Parser*). The parser steps through the actual XML document and calls the appropriate methods in the handlers as necessary.

The other two interfaces define how to specify the attributes for a tag (*AttributeList*) and the whereabouts of the parser in the source document (*Locator*). The latter allows for better error reporting if something goes wrong.

Two helper classes are included: *HandlerBase* and *InputSource*. The former is a default implementation of all the handler interfaces. The parser uses an object of this type if no other handler is specified. Typically, these default methods do nothing. By deriving our handler from this base, we need only override those methods that we want to have do something useful. The *InputSource* class defines an input stream containing the XML document, and can load that stream from local storage or across the Internet. It also contains details about where the document came from and how it was encoded.

For error reporting, two exceptions are defined: *SAXException* and *SAXParseException*. The first is the parent for all SAX exceptions and includes a property allowing us to embed any other exception within it. Thus, we can wrap a normal exception to make it appear as a SAX exception. *SAXParseException* is derived from *SAXException* and adds properties to identify the originating document and the position within it that caused the error.

Handling Documents

The *DocumentHandler* interface is the one most users of XML documents are interested in. It provides a series of callback routines that are invoked by the SAX parser at the appropriate times.

The *StartDocument* method signifies the beginning of a new XML document, allowing the handler to perform any necessary initializations. Similarly, *EndDocument* is called to terminate the document and release any held resources. As elements are read from the XML document, the *StartElement* method notifies us of their presence. The content of each tag is then processed through further calls before the *EndElement* method is invoked. Even for an empty tag, both of these routines are called.

Other content is denoted by calls to the appropriate routine: *Characters* for text data, *IgnoreableWhiteSpace* for all whitespace between tags, and *ProcessingInstruction* for embedded instructions.

The *SetDocumentLocator* method allows the document handler to tie into the parse process, and to be able to find out where we are in the source document. This means that the document handler can perform further validations on the data, such as verifying date or numeric formats, etc., and report any violations while indicating the characters in error.

SAX Parser

The *Parser* interface enables us to register each of the four types of handlers before invoking a parse on a particular XML document. When starting a parse, the *EntityResolver* is queried to determine the proper name of the document before loading it. This allows us to supply a source document, given its public or system ID. In this way, we can redirect document references or look them up in an external table. While parsing, the *DocumentHandler* and *DTDHandler* routines invoked as elements are encountered. Hopefully, the *ErrorHandler* is not called upon to deal with errors that may arise.

```
{ Interface to handle XML documents. }
ISAXDocumentHandler = interface(IUnknown)
[ '{ 17FA1882-38A2-11D3-9ABD-8A0 70 457C716 }' ]
  procedure SetDocumentLocator(Locator: ISAXLocator);
  procedure StartDocument;
  procedure EndDocument;
  procedure StartElement(Name: TSAXString;
    Attributes: ISAXAttributeList);
  procedure EndElement(Name: TSAXString);
  procedure Characters(Text: TSAXString);
  procedure IgnoreableWhiteSpace(Text: TSAXString);
  procedure ProcessingInstruction(
    Target, Data: TSAXString);
  procedure Comment(Text: TSAXString);
end;

{ Interface to XML parser. }
ISAXParser = interface(IUnknown)
[ '{ 17FA1884-38A2-11D3-9ABD-8A0 70 457C716 }' ]
  function GetEntityResolver: ISAXEntityResolver;
  procedure SetEntityResolver(Handler: ISAXEntityResolver);
  function GetDTDHandler: ISAXDTDHandler;
  procedure SetDTDHandler(Handler: ISAXDTDHandler);
  function GetDocumentHandler: ISAXDocumentHandler;
  procedure SetDocumentHandler(
    Handler: ISAXDocumentHandler);
  function GetErrorHandler: ISAXErrorHandler;
  procedure SetErrorHandler(Handler: ISAXErrorHandler);
  property EntityResolver: ISAXEntityResolver
    read GetEntityResolver write SetEntityResolver;
  property DTDHandler: ISAXDTDHandler
    read GetDTDHandler write SetDTDHandler;
  property DocumentHandler: ISAXDocumentHandler
    read GetDocumentHandler write SetDocumentHandler;
  property ErrorHandler: ISAXErrorHandler
    read GetErrorHandler write SetErrorHandler;
  procedure Parse(URI: TSAXString);
  procedure ParseSource(Source: TSAXInputSource);
end;
```

Figure 2: SAX *DocumentHandler* and *Parser* interfaces in Delphi.

As well as parsing the document internally, an implementation of the interface could be a wrapper around an external parser, i.e. a driver for that parser. This is what we're doing with the Microsoft XML parser.

Exceptions

SAX defines two exception types: *SAXException* and its descendant *SAXParseException*. As well as enabling us to easily identify all SAX exceptions, *SAXException* provides for wrapping any other exception, such as an I/O error, that might occur during processing. This allows these miscellaneous exceptions to be treated in a common way and still have access to the originals as necessary.

SAXParseException retains this ability and adds further details about an error during parsing. Specifically, it notes the public and system IDs of the document (its name/location) and the line and column position where the error was detected.

The *ErrorHandler* interface defines three levels of errors: *Warning*, *Error*, and *FatalError*. A warning can be safely ignored and the parsed document will be useable. An error can also be ignored, although the resulting document cannot be processed. The parser may continue working only to reveal any further errors. And finally, a fatal error results in the parser stopping completely. The default actions for these routines are to ignore warnings and errors, and to raise an exception for a fatal error.

Into Delphi

Translating the SAX specification into Delphi isn't difficult.

```

{ Abstract base class for SAX parsers. }
TSAXCustomParser =
  class(TInterfacedObject, ISAXParser, ISAXLocator)
  private
  ...
  protected
  procedure ParseInput(stmXML: TSAXInputSource);
    virtual; abstract;
  public
  constructor Create;
  destructor Destroy; override;
  { ISAXParser }
  ...
  procedure Parse(URI: TSAXString);
  { $IFDEF VER10 0 } { Not Delphi 3. }
    overload; virtual;
  procedure Parse(Source: TSAXInputSource); overload;
  { $ENDIF }
    virtual;
  procedure ParseSource(Source: TSAXInputSource); virtual;
  ...
  end;

```

Figure 3: Using conditional compilation to handle Delphi versions 3 through 5 in the one source file.

Delphi 3, 4, and 5 provide the interface construct, allowing you to directly copy the API's declarations. Of course, there are some minor changes in keeping with Delphi's naming standards: Interfaces start with "I," classes with "T," and exceptions with "E," as well as prefixing each name with "SAX" as a way of producing unique names within the wider Delphi world. Also, most methods starting with "get" were defined without that prefix, again in keeping with the feel of Delphi.

Each interface is assigned a GUID to allow for its presence to be determined at some later stage. The methods are basically copied from the specification and some types are altered to reflect Delphi's capabilities. The results of translating the *DocumentHandler* and *Parser* interfaces can be seen in [Figure 2](#). All the interfaces and supporting classes can be found in the *XMLSAXI* unit. (This unit, and all other source discussed in this article, is available for download; see the end of this article for details.)

Some larger changes were also made. A *Comment* method was added to the *DocumentHandler* interface to deal with these parts of the document. For normal processing, these can be safely ignored, but applications such as viewers and editors are interested in such things. Also, the handlers were implemented as read/write properties even though the specification has them as (read-only) function calls. Again, this felt more like Delphi and allows for the testing of a handler's presence within the parser.

In the *DTDHandler* interface, a *DocumentType* method was added to receive notifications of the DTD declaration within an XML document. This provides the handler with enough information for it to retrieve the DTD if it desires.

Classes were defined for the exception types, along with the input source, a default handler and parser. *TSAXInputSource* encapsulates a memory stream and can load this from either a local file or across the Internet. For the latter, it makes use of the *THTTP* class in Delphi 3 or the *TNMTTP* class in Delphi 4 and 5. All Internet access can be disabled by defining *NOHTTP* as a conditional define.

TSAXHandlerBase implements the four handler interfaces, although each routine generally does nothing. This provides a convenient base upon which to build more useful handlers. All we need to do is derive our handler from this base and override only the methods we're interested in, safe in the knowledge that the remainder will function in a predictable way. An instance of this base class is created by the *XMLSAXI* unit as *HandlerBase*, making it always available for use in parsers. Note that we need to increment the reference count for the handler so it doesn't get released as we make use of it within the parsers:

```

initialization
  { Create a default handler object. }
  HandlerBase := TSAXHandlerBase.Create;
  HandlerBase._AddRef; { Need to keep it around. }
finalization
  HandlerBase._Release;

```

The default parser, *TSAXParser*, is an abstract class that handles all the nitty-gritty of registering the handlers and performing the preparations for a parse. Practical parsers can inherit from this class and implement the protected *ParseInput* method to perform the actual processing of the XML document in the supplied stream.

Implementation

Of course, an interface declaration is just that — a blueprint for how things should work. To be useful, we need to implement that specification in actual classes. Because many parsers could use the attribute list, its implementation was placed into its own unit, *XMLSAXAtt*. Here we find a *TSAXAttribute* class to hold details about an individual attribute, and the *TSAXAttributeList* class, which manages a *TStringList* to provide the necessary interface methods.

A document handler can be found in the *XMLDocModel* unit. It implements the document handler interface and generates a simplified document model as it goes. This model is then available to the calling document for further processing through the *XMLDocument* property. Recall that XML tags are structured like a tree. This is reflected in the class that makes up the document model. *TXMLElement* defines a single node with a type, name, and value. It may also have attributes and a list of sub-

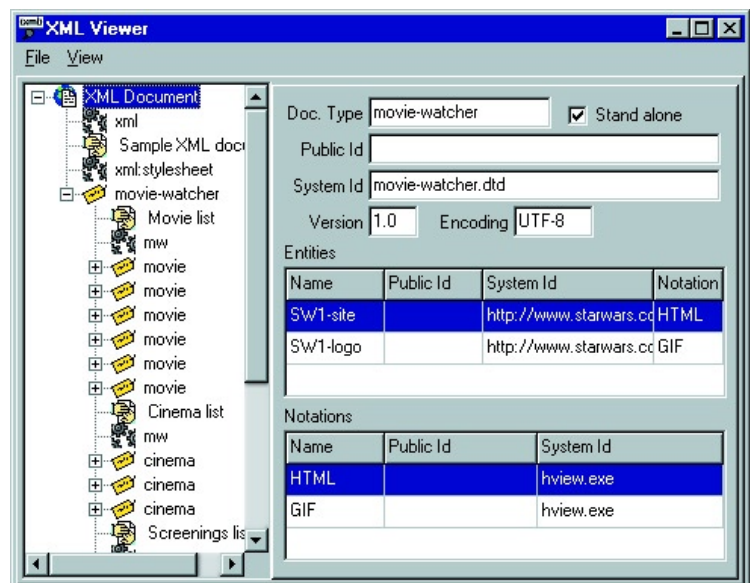


Figure 4: A simple XML viewer using SAX.

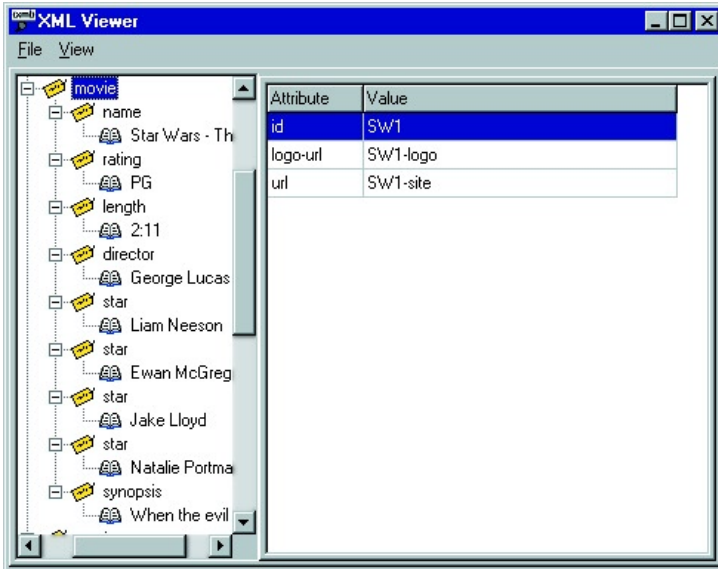


Figure 5: Element attributes in the XML viewer.

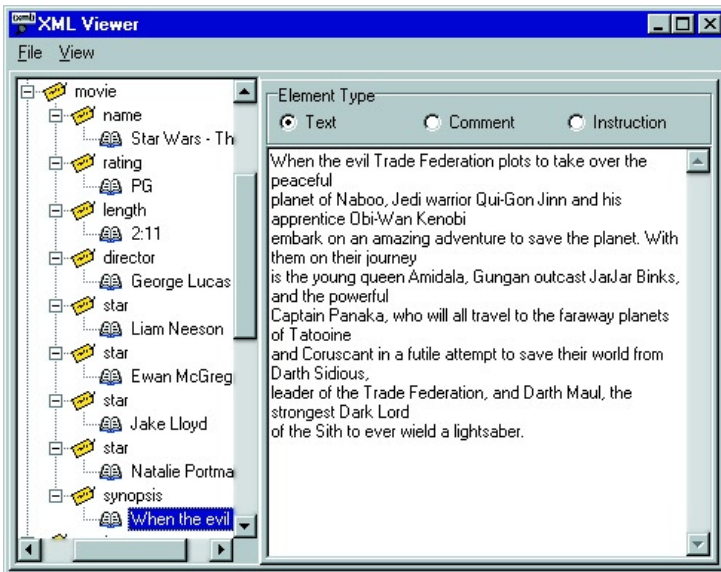


Figure 6: Text entries in the XML viewer.

elements, each of which is also a *TXMLElement*. The document tree can be easily processed through recursive routines to extract the necessary information.

In Delphi 4 and 5, we have the option of overloading methods

Icon	Node type
	XML document
	Element tag
	Comment
	Processing instruction
	Text

Figure 7: Icons for node types.

and supplying default values and can employ this in several places within the implementations. For example, in the *TSAXCustomParser* definition, there are two versions of the *Parse* method: one that takes a URI identifying the document source, and one that takes an input source. In Delphi 3, we need two differently named methods to achieve this, but in Delphi 4 and 5 we overload the one method name to handle both situations. This makes it easier for the user because they don't have to

remember the name of the other version. To handle Delphi versions 3 through 5 in the one source file, we can use conditional compiles, as shown in Figure 3.

Finally, there are two implementations of a SAX parser: a native Delphi version, *TSAXDelphiParser*, in the *XMLSAX* unit, and a wrapper for the Microsoft XML parser, *TSAXMSParser*, in the *XMLSAXMS* unit. The Delphi version neither validates the document with its DTD, nor does it supply default and fixed values for attributes that don't appear in the document. Also, it doesn't parse external entities. Even so, it works well with stand-alone XML documents, even when they include internal entities.

XML Viewer

Putting all of this into practice, we can now build a simple XML viewer. Provided with a file name (as a command-line parameter, or via the menu), it opens the file, parses the XML, and displays it to the screen in a tree view (see Figures 4 through 6).

For each node within the tree, we identify its type with an icon (see Figure 7) and display its details when it's selected. The document displays the DTD name, XML version, etc., along with any unparsed external entities and notation declarations. Element nodes may have a series of attributes attached that are shown in a string grid. Other nodes (text, comments, and processing instructions) have their type displayed as a radio button selection, with their actual contents appearing in a memo.

Menu items allow us to swap parsers, load a different document, or exit the program. Others provide for expanding or collapsing the entire tree structure beneath the selected node, and for viewing the source document behind the structure.

To load this simple view, we create a document model handler as previously described and connect it with our favorite XML parser (as a SAX parser).

Again, we must increment the reference count for the document handler, because we are going to be using it over and over for the different parsers. Note that the document handler also implements the *DTDHandler* interface, allowing it to process DTD declarations:

```

xdhDocument := TXMSSimpleDocHandler.Create;
xdhDocument._AddRef; { Keep it around. }
...
xprSAXParser := TSAXDelphiParser.Create(xdhDocument,
xdhDocument, nil, nil);
    
```


The resulting document model is processed recursively, as shown in Listing One (on page 29) and creates corresponding nodes in the tree view as we go. A pointer to the element in the model is saved as the Data for the node, with the node's caption coming from the element's name or value. Appropriate icons are specified through the *ImageIndex* property based on the element's type. As the user steps from node to node, the attached element object is retrieved and additional details are displayed on the right half of the form.

The application uses the Delphi XML parser by default. To use the Microsoft XML parser instead, simply select it from the

File | Parser | Microsoft menu option. Remember that the same document handler is used in both cases. This is the whole point of SAX!

Conclusion

SAX is a standard that allows XML parsers and document consumers to interoperate in a well-defined and interchangeable manner. Any SAX-compliant document handler is able to make use of any SAX-compliant parser, and vice versa. This allows us to separate these two functions and to reduce the maintenance burden. Furthermore, as new parsers are made available, we can move to them with minimal changes to our applications.

The SAX implementations described here provide a native Delphi XML parser, as well as a wrapper around the Microsoft XML parser. A document handler that generates a simple document model is also provided. Together, these allow us to write a basic XML viewer quite easily. 

References

XML specification: <http://www.w3.org/XML>

SAX specification: <http://www.megginson.com/SAX>

Microsoft XML parser: <http://msdn.microsoft.com/xml/default.asp>

XML information: <http://www.xml.com>,

<http://www.xmlsoftware.com>

The files referenced in this article are available on the Delphi Informant Magazine Complete Works Companion Disk in INFORM\00\MAR\DI200003KW.

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.

Begin Listing One — Loading an XML document

```
{ Load an XML document. }
procedure TfrmXMLViewer.LoadDoc(sFilename: string);
var
  i: Integer;

  { Add current element to the tree view, then recurse
  through children. }
  procedure AddElementToTree(xe1Element: TXMLElement;
    trnNode: TTreeNode);
  var
    i: Integer;
    sName: string;
    trnNew: TTreeNode;
  begin
    { Display meaningful text. }
    if xe1Element.ElementType in [xtComment, xtText] then
      begin
        sName := Copy(xe1Element.Value, 1, 20);
        if Length(xe1Element.Value) > 20 then
          sName := Copy(sName, 1, 17) + '...';
        end
      end
    else
```

```
      sName := xe1Element.Name;
      trnNew := trvXML.Items.AddChildObject(trnNode, sName,
        xe1Element);
      trnNew.ImageIndex := Ord(xe1Element.ElementType);
      trnNew.SelectedIndex := trnNew.ImageIndex;
      for i := 0 to xe1Element.Contents.Count - 1 do
        AddElementToTree(xe1Element.Contents[i], trnNew);
      end;
begin
  pgcDetails.ActivePage := tshDocument;
  trvXML.Items.Clear;
  { Load the source document. }
  memSource.Lines.LoadFromFile(sFilename);
  dlgOpen.FileName := sFilename;
  { Parse the document. }
  xprSAXParser.Parse(sFilename);
  { Extract document level information. }
  edtDocType.Text := xdhDocument.Name;
  edtPublicID.Text := xdhDocument.PublicID;
  edtSystemID.Text := xdhDocument.SystemID;
  edtVersion.Text := Format('%3.1f', [xdhDocument.Version]);
  edtEncoding.Text := xdhDocument.Encoding;
  cbxStandAlone.Checked := xdhDocument.StandAlone;
  with xdhDocument.Entities, stgEntities do begin
    RowCount := 2;
    if Count > 0 then
      RowCount := Count + 1
    else
      Rows[1].Clear;
    for i := 0 to Count - 1 do begin
      Cells[0, i + 1] := Items[i].Name;
      Cells[1, i + 1] := Items[i].PublicID;
      Cells[2, i + 1] := Items[i].SystemID;
      Cells[3, i + 1] := Items[i].Notation;
    end;
  end;
  with xdhDocument.Notations, stgNotations do begin
    RowCount := 2;
    if Count > 0 then
      RowCount := Count + 1
    else
      Rows[1].Clear;
    for i := 0 to Count - 1 do begin
      Cells[0, i + 1] := Items[i].Name;
      Cells[1, i + 1] := Items[i].PublicID;
      Cells[2, i + 1] := Items[i].SystemID;
    end;
  end;
  { Add the structure to the tree view. }
  AddElementToTree(xdhDocument.XMLDocument, nil);
  trvXML.Items[0].Expand(False);
end;
```

End Listing One





By Peter Morris



FormShaper

Say Good-bye to Rectangular Forms

Developers employ many tricks to make their applications stand out from the crowd. One of the most popular methods is using non-rectangular forms. Another way is altering the shapes of buttons, panels, etc. However, these tricks are very primitive. Consider the following code:

```
procedure TForm1.FormResize;
var
  Region : HRGN;
begin
  Region :=
  CreateEllipticRgn(0,0,width,height);
  SetWindowRgn(Handle, Region, True);
end;
```

First, a variable of type HRGN is defined, which is a handle to a region (and could simply be a *THandle*). Then, whenever the form is resized, an elliptical region matching the size of our form is created using *CreateEllipticRgn*. The region is applied to the form using:

```
SetWindowRgn(HandleOfObjectToShape,
  HandleOfNewRegion, RedrawImmediately);
```

(Note: Windows owns regions associated with controls, so there is no need to destroy the current region before applying a new one.)

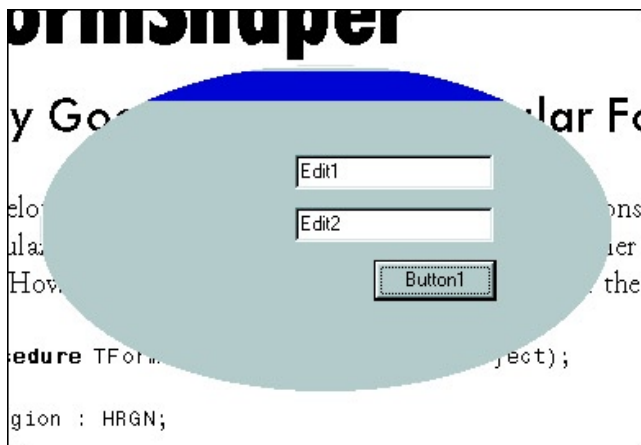


Figure 1: An elliptical form created with *CreateEllipticRgn*.

If you've never seen this before, you may be excited (see Figure 1). Don't get too excited, though. After a short time, you realize that an elliptical form is almost as boring as a rectangular one. This article reviews some new methods that are available to developers for spicing up an otherwise mundane application.

Playing with Shapes

Windows provides a suite of region-related commands to make life more interesting:

- *CreateEllipticRgn*
- *CreateEllipticRgnIndirect*
- *CreatePolygonRgn*
- *CreatePolyPolygonRgn*
- *CreateRectRgn*
- *CreateRectRgnIndirect*
- *CreateRoundRectRgn*
- *GetPolyFillMode*
- *GetRegionData*
- *GetRgnBox*
- *InvertRgn*
- *OffsetRgn*

Although these commands offer more flexibility, they still only give us the ability to create rather straightforward form shapes. This is where combining regions comes into play. This list is by no means complete, so we'll discuss additional commands throughout this article.

Combining Regions

Now we're going to start getting more complicated. With Windows, it's possible to create a number of regions and combine them using the *CombineRgn* command. Regions may be added together, subtracted from each other, or XORed (areas that do not overlap are combined; areas that do overlap are excluded).

Value	Description
RGN_AND	Combines the regions only where they overlap.
RGN_COPY	Creates a copy of the region (SourceRegion1).
RGN_DIFF	Makes a differential region (difference between Source1 and Source2).
RGN_OR	Adds the two source regions together.
RGN_XOR	Areas that don't overlap are combined; overlapping areas are excluded.

Figure 2: Ways of combining methods.

```

procedure TForm1.Resize;
var
  Region,
  RectRegion : HRgn;
begin
  Region := CreateEllipticRgn(0,0,Width,Height);
  RectRegion := CreateRectRgn(32,32, Width-32, Height-32);

  // Combine Region with RectRegion;
  // store the result in Region.
  CombineRgn(Region, Region, RectRegion, RGN_XOR);

  // Once applied to our form, Region will be owned by
  // Windows, while RectRegion will not. In this case we
  // need to destroy RectRegion.
  DeleteObject(RectRegion);
  SetWindowRgn(Handle, Region, True);
end;
    
```

Figure 3: Combining a rectangle to achieve an interesting effect.

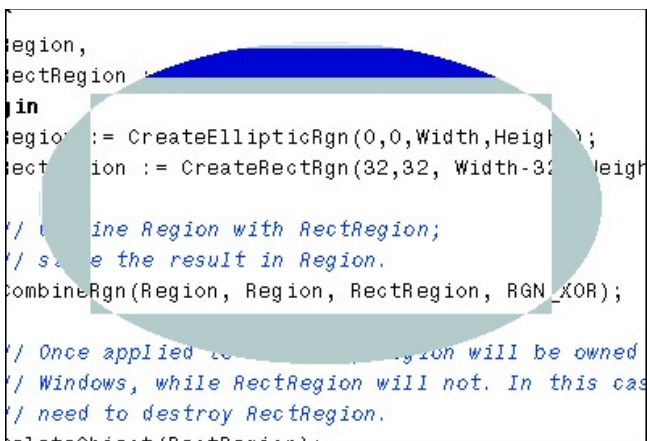


Figure 4: The result of the code in Figure 3.

```

// Code above here will load a black and white bitmap
// into a TBitmap variable called BMP.
with BMP do
  for Y:=0 to Height-1 do
    for x:=0 to Width-1 do begin
      // Is this pixel black?
      if Canvas.Pixels[X,Y] = clBlack then begin
        // If this is the first pixel, we create a new
        // region. Otherwise we add it to the region we
        // have created so far.
        if FRgn = 0 then
          FRgn := CreateRectRgn(X,Y, X+1, Y+1)
        else
          begin
            EXCL := CreateRectRgn(X,Y, X+1, Y+1);
            CombineRgn(FRgn, FRgn, EXCL, Rgn_XOR);
            DeleteObject(EXCL);
          end;
      end;
    end;
end;
    
```

Figure 5: Creating a region from a bitmap.

The syntax of *CombineRgn* is:

```

CombineRgn(NewRegion, SourceRegion1, SourceRegion2,
  CombineMode);
    
```

Available modes of combining regions are shown in Figure 2.

You could combine a rectangle to our previous elliptical form using RGN_XOR (see Figure 3) for a strange effect (as shown in Figure 4), but this process is time consuming.

Even though we now have an extensive set of tools for altering a form's shape, we can't easily make regions suitable for applications. I could introduce you to the *CreatePolygonRgn* command, which creates your region from an array of points. But this method is time consuming, as well, and people tend to sacrifice quality to save time. A region with poorly shaped edges is an eyesore and much worse than a standard Windows rectangle.

The question is: Can we easily make complex regions quickly and maintain high quality? Luckily, the answer is "Yes."

Creating a Region from a Bitmap

At this point, it would be nice to be able to introduce the *CreateBitmapRgn* command. It would allow us to draw a simple black-and-white mask for the shape of the form using our favorite graphics package and import it directly into our application. Unfortunately, there is no such command.

Therefore, we must write this routine ourselves. This process involves reading each pixel from a black-and-white bitmap mask: If the pixel is white, we ignore it; if it's black, we create a region the size of one pixel and add it to a larger region, which will eventually be applied to our form.

The problem with this method is that, although it offers the ability to manipulate our form's shape with pixel precision, it's too slow. Anyone who has ever tried the *Canvas.Pixels* property will know that this method could take as long as one minute for an average size bitmap. Luckily, there are a few ways we can reduce or eliminate this problem. But first, Figure 5 shows one way to create our region from a bitmap.

The code in Figure 5 is quite simple, but if we tried to use it in an application at run time, our application would take far too long to start. The solution to our problem lies with two more Windows commands: *ExtCreateRegion* and *GetRegionData*.

Loading and Saving a Region

After our region is created, it's easy to acquire its data to write out to disk, and then read it back as a Windows region. Luckily, this is instantaneous, and — even better — it's easy:

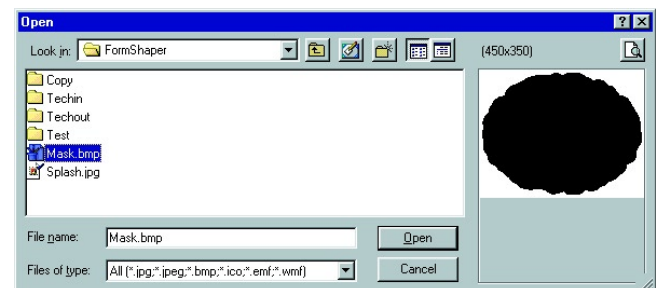


Figure 6: Selecting a 2-bit, black-and-white bitmap as the mask.

```
// Passing nil tells GetRegionData that we just
// want the size in bytes.
RegionSize := GetRegionData(FRgn, 0, nil);
// Allocate the correct amount of memory.
RegionData := AllocMem(RegionSize);
// Get the data for the region.
GetRegionData(FRgn, 0, RegionData);
```

The data now held in *RegionData* can be quite easily written to a stream:

```
// So that we know how many bytes to read back later.
TheStream.Write(RegionSize, SizeOf(Integer));
// Write the stream data.
TheStream.Write(RegionData^, RegionSize);
```

Reading the stream back into a region is equally simple:

```
// Read back the size.
TheStream.Read(RegionSize, SizeOf(Integer));
// Allocate the memory to hold our region.
RegionData := AllocMem(RegionSize);
// Read the region data back in to memory.
TheStream.Read(RegionData^, RegionSize);
```

Finally, making the *RegionData* back into a Windows region is as simple as this:

```
Region :=
  ExtCreateRegion(nil, RegionSize, TRgnData(RegionData^));
SetWindowRgn(Handle, Region, True);
```

Choosing the Correct Implementation

The final step is choosing the correct way to implement this technology. An easy way would be to write a “region-making” tool to create region files from bitmaps. Your application could then load these regions at run time. The problem is that you must ship many small files along with your application.

I’ve written a *TFormShaper* component to implement this method. This allows me to apply regions at design time and save the *RegionData* in the same way a *TImage* would: embedded within the application. (*TFormShaper*’s source code, and a demonstration application, is available for download; see the end of this article for details.)

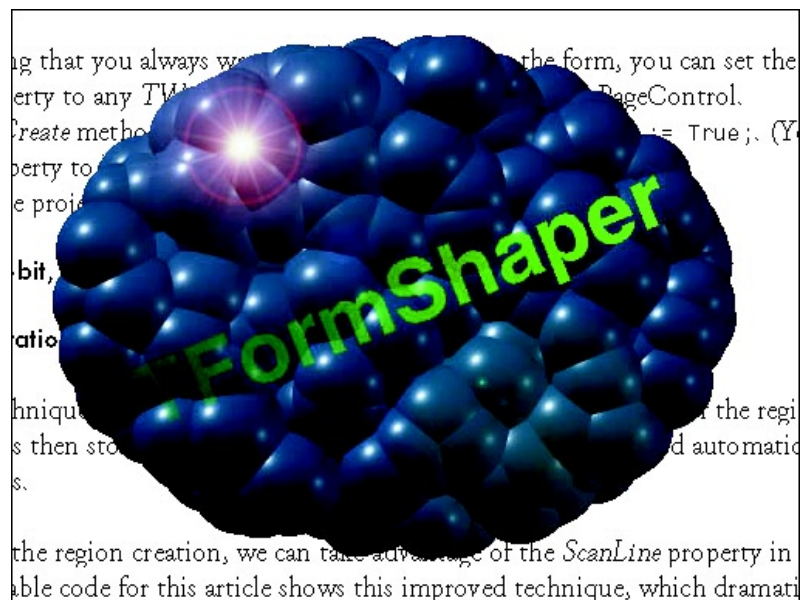


Figure 7: The demonstration application at run time.

To use the *TFormShaper* component, select **Component | Install** from the Delphi IDE and add the *cFormShaperReg.pas* unit to a design-time package.

After the *cFormShaperReg.pas* unit is installed on your Component palette, follow these steps:

- 1) Drop a *TFormShaper* component on the form, right-click on the component, and select **Import Mask** (I have provided one with my downloadable code, as shown in Figure 6). An Open dialog box will appear. Select a 2-bit (black and white) bitmap, which will be used as the form shape (black = solid, white = transparent). When you specify the file and click OK, the Mask image will be processed and a region of the shape of the Mask will be created. It might take a couple seconds, so be patient.
- 2) Set the form’s *BorderStyle* property to *bsNone*; otherwise the size of the banner will shift the Mask vertically.
- 3) Rather than assuming that you always want to apply the shape to the form, you can set the *ControlToShape* property to any *TWinControl* descendant, e.g. a *Panel* or *PageControl*.
- 4) In the form’s *FormCreate* method, add the statement: `FormShaper1.Active := True;` (You can also set its active property to True at design time.)
- 5) Compile and run the project (see Figure 7).

The advantage of this technique is that we have done all the grunt work (the creation of the region) at design time. The region is then stored internally in the component and will be loaded automatically when the application runs.

To improve the speed of the region creation, we can take advantage of the *ScanLine* property in *TBitmap*. The downloadable code for this article shows this improved technique, which dramatically reduces the time necessary to build the region from the Mask bitmap.

Conclusion

In this article, we looked at ways to improve the look of your forms. If you’re looking for a way to spruce up your development environment, sample some of the methods we discussed.

Who knows? You may set a trend. ▲

The files referenced in this article are available on the *Delphi Informant Magazine Complete Works Companion Disk* in `INFORM\00\MAR\DI200003PM`.

Peter Morris is 26 years old and married with two children. He works from home as a computer programmer for Insite Technologies Ltd., writing a wide variety of non-standard-looking Windows applications. He loved breaking away from the Windows GUI standards, as he now writes special graphical effects and animated components — the sort of things that make people say, “Wow, how did he do that?” You can reach Peter at MrPMorris@hotmail.com.





NEW & USED

By John Rendell

ASTA 2.1

The Fast Way to Internet Applications

ASTA Technology Group claims ASTA 2.1 is "... the simplest way to create robust, full-featured, Internet-enabled applications" — a pretty bold claim for a product that competes with several other *n*-tier development tools. Because I'm working on a local-table-to-InterBase conversion of a large staffing system, I couldn't pass up the chance to test ASTA to see if it lives up to its claim.

ASTA is a tool that allows development of three-tier Internet-enabled database applications. This means that the database runs on one tier, an application server on a second tier, and clients on a third.

In traditional two-tier development, the client issues SQL statements, and the server responds with corresponding data. In a three-tier environment, the SQL is usually coded into the application server. The client triggers the application server to issue the SQL statement, and the application server returns the resulting rows from the server. Needless to say, three-tier applications require a lot of planning in creating the server (the implications of which are too lengthy to go into in this review).

ASTA differentiates itself by allowing traditional two-tier practices to be used in a true three-tier environment. (Figure 1 shows an abstract implementation diagram.) SQL can be generated from the application server or the client, allowing for fast application conversions.

After working with other three-tier products, I was a little concerned about the learning curve. Working with ASTA was a nice surprise. After a few tutorials, I felt comfortable enough to start converting a large test application. ASTA provides an application server and client datasets for accessing data from the server.

Servers

There are 10 optimized ASTA servers, depending on your database needs. Because I use IBOjects for accessing InterBase tables, I was happy to see an ASTA IBOjects server. Other servers include Direct Oracle Access, ADO, ODBC, ODBC Express, IBExpress, and Advantage. Documentation is included to allow the coding of other custom servers.

All ASTA servers can run in one of three modes:

- **Single** (default): Client requests are queued and processed in a FIFO (first-in-first-out) order. Only a single connection to the database server is used.
- **Pooled**: A predefined number of connections to the server are pooled, allowing for simultaneous threaded access to the server. For example, if created for 10 sessions, 10 connections to the server are opened at run time (even if no clients are connected). Clients use whatever connection is available. Additional threaded sessions are dynamically created on an as-needed basis.
- **Persistent**: Each client connects to a session that is persistent for that client. A database connection is used for each client. Because a unique session is opened for each client, it allows the developer more flexibility in controlling how data streams back to the client.

After selecting the threading model, all I had to do was compile and run the server; no additional



Figure 1: ASTA allows traditional two-tier practices to be used in a true three-tier environment.

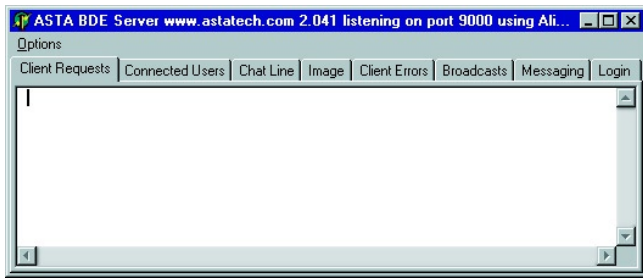


Figure 2: The ASTA server.

work was necessary. Unlike MIDAS, the ASTA server has some nice client management tools (see Figure 2):

- Connected clients
- Messaging
- Chat lines
- Client errors
- Broadcast messages
- Login stamping

With these tools, client requests can be logged (great for debugging) in the Client Request window. Clients can also be disconnected from the server. The chat tools are nice, but I decided not to use them.

The application server is run as an executable. There is also an application available that reloads ASTA servers, as well as an NT service that manages ASTA servers.

Clients

Once the server was running, I had to create a client to actually use the ASTA server. ASTA uses IP sockets to connect to the ASTA server (unlike MIDAS, you cannot use DCOM). Each client application must have an ASTA Client Socket. Because the Client Socket determines where the ASTA server resides, I had to provide an IP address or machine name so I could use some of the *ASTAClientDataSet* properties at run time.

ASTA uses a custom *ASTAClientDataSet* to converse with the ASTA server. Again, ASTA's ease of use shines with its *SQLWorkbench* property. Selecting this property opens a dialog box with just about everything you need to build your SQL statements, as shown in Figure 3.

By default, all *ASTAClientDataSets* are read-only. By changing the *EditMode* property, you can have ASTA apply updates on each record post, or keep them cached until you decide to apply the changes. Again, ASTA has made this easy to do with a dialog box that guides you through selecting the update mode, key field, and the table to update.

Because the *ASTAClientDataSet* is *TDataSet* compatible, you can use any data-aware component you'd like to edit and display data.

ASTAClientDataSets are also memory-based, which allows fast population of controls. *ASTAClientDataSets* can also be used in briefcase mode, allowing the data to be stored to a local drive and retrieved for later editing/updating.

Advanced Options

With version 2, ASTA introduced some advanced tools for doing more work on the server. There are two components: *ASTAProvider* and *ASTABusinessObjectManager*. The *ASTABusinessObjectManager* allows for custom SQL code to be issued from the server. There are several examples included with ASTA that show how to use this component.

My favorite, however, is *ASTAProvider*. This is a new component that allows SQL to be generated on the server side and have some server-side control on posting of records via *Before/After* events. *ASTAProviders* also have one of the most revolutionary ways of dealing with multiple clients and updates: broadcasts of single record changes.

To give you an idea of how broadcasts work (and why I think they are so great), the following is a requirement in the test application. In the test application, I have a ToDo list. Each user has a list of items that need to be completed. It's possible (and probable) that other users will send ToDo lists to other users. Each user's list must show these items as soon as possible. Some users may have 10 items, some hundreds.

Traditional methods of client/server and *n*-tier development have no direct way of notifying other clients of changes that have been made. A few ways around this is to use *Database* events (still requiring the client to issue another query to find what changed) or to refresh the query (in most cases requiring the query to be closed and reopened). *ASTAProviders* have made this process extremely easy:

- Add an *ASTAProvider* to the application server and point it to a *TDataSet*.
- Add an *ASTAClientDataSet* to the client application. Choose the Server Provider from the *ProviderName* property.
- Change *SQLGenerateLocation* to Server (default is Client).
- Change *RegisterForBroadcasts* to True.

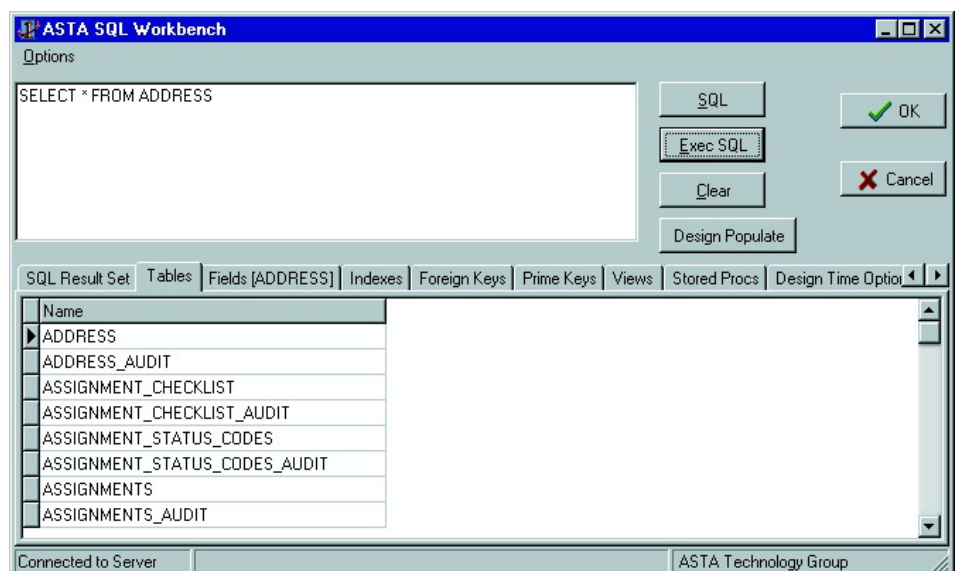


Figure 3: Selecting the *SQLWorkbench* property brings up this dialog box.

INFORMANT

FACT FILE

ASTA is an Internet development tool that delivers on its promise as being "... the simplest way to create robust, full-featured Internet-enabled applications." If you're looking for a fast way to move your existing applications to the Internet (or even on a local LAN), look no further — ASTA has you covered.

ASTA Technology Group
24 Ragged Ridge Rd.
Camden, ME 04843

Phone: (800) 699-6395
E-Mail: info@astatech.com
Web Site: <http://www.astatech.com>

Price: ASTA Entry Suite (includes components and license for one server), US\$399; additional server licenses, US\$249 (a license is required for each server); source code, US\$249; site/enterprise licenses are available.

Once these steps are complete, any change to the provider's dataset is broadcasted back to all listening clients. (However, the *OnProviderBroadcast* method must be coded to do something with it — ASTA's easy, but not that easy!)

Why is this revolutionary? Assume there are 100 client connections. Each client may have 100 items on a ToDo list. If the only option to update these clients is to close and reopen the query, then every few minutes, the server would have to fetch and return 10,000 records — regardless of whether there were any changes. By using broadcasts, you've reduced this to only having the server broadcast a changed record to everyone. ASTA automatically suppresses the broadcast back to the issuing client — so the client that issued the update won't receive its own changes.

- Better online documentation (as noted previously).
- Server as an NT Service (ASTA Technology Group is working on this now).

Conclusion

I liked ASTA so much I decided to use it on my current project. The ability to broadcast changes made the decision a no-brainer. ASTA makes moving applications from any format to client/server or the Internet very easy. From simple SQL at the client level, to full-blown server processing, ASTA has it covered. ▲

John Rendell is President of Little Wiggler Software, Inc., specializing in upsizing database applications. When not playing in Deja 80s — a retro band — he can be reached at john@little-wiggler.com.

Deployment

Once the actual application server is configured to access the database, the ASTA server only needs to be copied to the server and then run. On the client side, no configuration is needed. Because all the database connections happen at the application server, there really is nothing to configure (except the IP address of the server, which can be read in from a central .ini file or stored in the registry).

One of the overlooked benefits of using a three-tier environment is security. By having the server on its own machine, it's possible to make the actual database server invisible; all database access is controlled by the application server. ASTA only requires a user-defined port to be opened on a firewall for Internet access.

Tech Support

Tech support is first rate. Questions on the list server were answered within 24 hours.

Suggested Improvements

There are many example projects and some online Help, but I really wish the documentation was better; the online documentation is not complete (many of the advanced functions and procedures are not listed). Having to hop from an example to the online Help to figure out how something works is not very productive.

Another minor nit pick: When compiling the source code, there are a lot of warnings and hints, most of it due to coding style. Not a big deal, but something I'd like to see cleaned up.

Wish List

As no product is perfect, I would like to see the following things in future versions:

- Load balancing/Fail Over.
- Automatic reconnecting — it's currently possible with code, but I'd like to see it as an automatic feature.



Multimedia Resources on the Internet

In the **August, 1999 issue** of *Delphi Informant Magazine*, I wrote about multimedia resources in book form. I also promised to devote another column to exploring some of the resources available on the Internet. I'll begin by discussing links that have a direct connection to Delphi and then mention more general links that provide valuable supplementary information. Delphi multimedia sites can be divided into two groups: Those dealing with general audio and those dealing with MIDI. We'll begin with the more general sites.

General Delphi multimedia sites. Swift Software (<http://www.swiftsoft.de>) provides a full line of professional multimedia software, including Delphi components for manipulating wave data, working with mixers, and handling .avi files. There are also components that go beyond the ordinary multimedia operations to filter sounds, create sound effects, and work with newer technologies — including DirectSound (3D) and MPEG. (I haven't tested these, so I can't comment on their quality.) There are also a number of freeware Delphi tools and components on this site. These include a Debug Monitor with a Delphi unit that sends commands and messages to the debug window; a TimeStamp Expert, an add-on that automatically creates headers and time stamps in your code; a freeware MP3 BladeEncoder Component; and Video for Windows, a Delphi import unit for the VFW SDK. These tools and components include source code.

If you're just getting started in multimedia and would rather "do it yourself" than use preexisting components, be sure to visit Alex Simonetti Abreu's cool site, Athena's Place (<http://www.bhnet.com.br/~simonet/howtoprojs.htm>). It consists of eight how-to projects with full source code — most of which is related to multimedia. These projects include examples for creating and saving a wave file to your own custom format, saving a wave file to a BLOB field, creating high-resolution timers using the multimedia services, and detecting multimedia devices and setting their volumes. There are also examples of creating and using resource-only DLLs (which can be used with multimedia data), extracting version information from DLLs, EXEs, VXDs, and more.

Dave's MIDI Software (<http://www.netcomuk.co.uk/~dave.ch/midisoft.html>) is a major Delphi MIDI site, which contains links to many other sites. Here you will find his valuable freeware collection, MidiComp. It includes MIDI input and output components for Delphi, as well as demonstration applications that use these components. I've tried some of these components and recommend them for your consideration.

Delphi Multimedia (http://www.kobira.co.jp/sakura/d_multi.htm) has a number of nice downloads. While you're there, be sure to check out David J. Taylor's SweepGen application.

My favorite Delphi multimedia site is probably Colin Wilson's (<http://www.wilsonc.demon.co.uk/delphi.htm>). The site features a large collection of freeware multimedia components and demonstration applications. These include Mixer Demo, which encapsulates the master volume control along with balance, mute, and bass and treble, as well as MIDI volume, balance, and mute controls. Additional examples include Mixer Explorer, which displays the mixer components supported on a particular system in a tree view; the feature-rich MIDI Controls & Sequencer application; MIDI Controls Demo, which plays MIDI files and displays MIDI events in a "piano-roll style"; a MIDI Jukebox, which uses the MIDI Controls and the mixer components; and the Multimedia

Level Data and Meter application, which includes an LED Ladder control and a Multimedia Meter Data control. These applications include full source code. There are also a number of NT-specific routines, among many others. Indeed, this is a very rich site well worth visiting.

Other multimedia sites. Though not a Delphi site, EDN Access (<http://www.ednmag.com/reg/1995/110995/>) contains articles on various technologies, including one from 1995 entitled "Multimedia Codecs Move beyond Basic Conversion." Although it's a bit dated, this article is very detailed. The Synth Zone (<http://www.synthzone.com/sampling.htm>) is devoted to audio samples and sampling resources. It includes a very large list of links to sites where you can find tools and sounds. Another site with many links, Audio Lab's Audio & Acoustic Links (http://audiolab.uwaterloo.ca/aa_links.html), provides access to additional sites containing useful technical information.

The Audio section of the Online Communicator (<http://www.communicator.com/audio1.html>) has a number of useful links to audio technology sites, including some related to MPEG audio. The "Channel 1" file library is a good source for shareware and freeware programs and utilities, including several hundred multimedia items. Unfortunately, you won't find much source code here.

A similar site, 32bit.com (<http://www.32bit.com/software/listings/multimedia>), contains a large multimedia section. This site is well organized and easy to navigate. Again, there are a large number of utilities but without source code.

The Audio File Format FAQ (<http://home.sprynet.com/~cbagwell/audio.html>) has up-to-date information on many audio file formats. It also has links to some interesting articles, including one on audio effects algorithms.

The MidiWeb (<http://www.midiweb.com>) includes a section on programming (C/C++ and Visual Basic). Also, in other pages on the site, you can download MIDI files.

Finally, The MIDI Farm (<http://www.midifarm.com>) is a vastly comprehensive MIDI site that contains information on commercial products and a large collection of MIDI files and other resources.

— 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.

