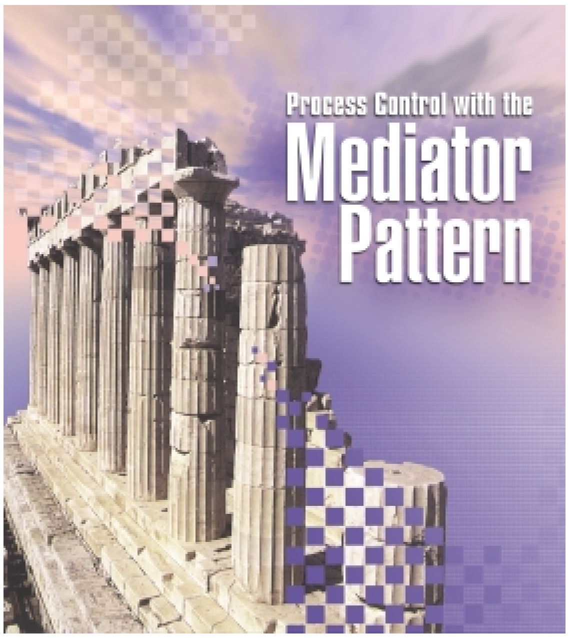
February 2000, Volume 6, Number 2



Cover Art By: *Darryl Dennis*

FileStream.com Ships InstallConstruct 3.2.1

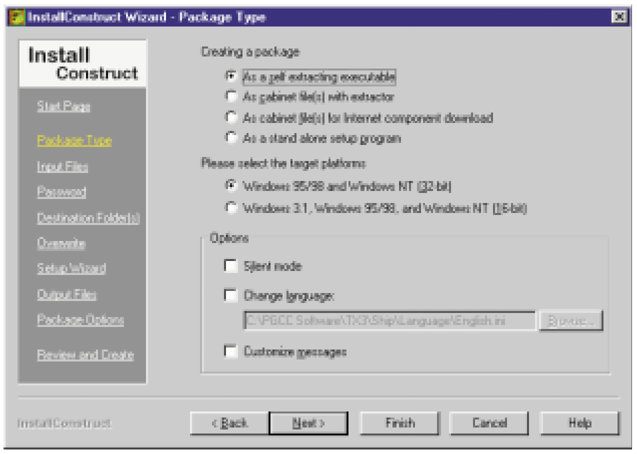
FileStream.com, Inc./Pacific Gold Coast Corp. announced the release of ***InstallConstruct 3.2.1***, designed for creation of Installer, Setup Wizard, Uninstaller, and HTML-based Internet Component Download installers. InstallConstruct is a compact suite of wizards and tools that makes creating Windows 3.1, 95, 98, NT, and Windows 2000 Installers easier, using step-by-step procedures. These installer files are ideal for efficient and professional distribution of groups of program and data files, which are compressed for economy and ready to be installed at the users' convenience.

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

Users of InstallConstruct can create and customize a Setup Wizard with their own product graphics and logo and display formatted text, for Internet and Intranet distribution of program, single and multi-volume CD-ROM, and disk distributions, as well as create and customize Uninstaller. InstallConstruct uses the expanding wizard system, which walks users through the entire creation process to choose from available options, without the need for a programming background or having to write application-specific programming codes.

This latest release adds support of icons to be added to the common group under Windows NT and Windows 2000 so all users of the computer can have access to it; creation of installers for Font Delivery; and the option of keeping backup copies of existing files. In addition, this new release also expands the total path length limitation. International language scripts are supported.

**FileStream.com, Inc./
Pacific Gold Coast Corp.**
Price: US$199
Phone: (800) 732-3002
Web Site: http://www.installconstruct.com


InstallConstruct Wizard - Package Type

Creating a package
- [✓ As a self extracting executable]
- As cabinet file(s) with extractor
- As cabinet file(s) for Internet component download
- As a stand alone setup program

Please select the target platforms
- [✓ Windows 95/98 and Windows NT (32-bit)]
- Windows 3.1, Windows 95/98, and Windows NT (16-bit)

Options
- Silent mode
- Change language: C:\PGCC Software\TX3\Ship\Language\English.ini
- Customize messages

Raize Releases Raize Components 2.5

Raize Software Solutions, Inc. announced ***Raize Components 2.5***, the latest release of the company's library of native VCL controls for Delphi and C++Builder.

Raize Components 2.5 has more than 90 components with the addition of nine new controls. Version 2.5 adds more design-time editors and streamlines some existing editors to make the components easier to use.

Most of the components have been enhanced in the new version. For example, the behavior of the *TRzFrameController* component has been changed. It no longer controls all the components on the form that support Custom Framing. Instead, each component that supports Custom Framing now has a new *FrameController* property. As a result, a *RzFrameController* will only control those components that reference itself through the *FrameController* property. This lets developers use multiple frame controllers to control different sets of components on the same form. Likewise, a single *TRzFrameController* can be used in a DataModule to control components on multiple forms.

Raize Components 2.5 also introduces several features specifically for Delphi 5 developers. For example, several new property categories are registered with Delphi 5, making it easier to locate properties related to specific features in Raize Components. For example, there are categories for Custom Framing, Text Style, and Border Style.

Raize Components 2.5 takes advantage of the owner-draw list support in Delphi 5 property editors by providing new editors that show a preview of the available items.

Raize Components 2.5 supports Delphi 1, 3, 4, and 5, and C++Builder 3 and 4. Raize Components 2.5 comes with complete source code for all components, packages, and design-time editors at no additional charge.

Raize Software Solutions, Inc.
Price: US$249
Phone: (630) 717-7217
Web Site: http://www.raize.com

CoRe Lab Announces ODAC 2.0

CoRe Lab Co. announced ***Oracle Data Access Components*** (ODAC) 2.0, a library of native Delphi components for direct access to Oracle.

ODAC 2.0 is an easier, more flexible, more powerful, and faster way of developing database applications with Oracle. ODAC allows developers to refuse using the BDE for applications working with Oracle only.

Features in ODAC 2.0 include flexible automatic updating; advanced locking and refresh rows; advanced support of Oracle objects, arrays, nested tables, BLOB, and CLOB data types; support of native Oracle8 call interface; embedded SQL Designer to build queries; and more.

CoRe Lab distributes versions of ODAC for Delphi 3, 4, and 5 and C++Builder 3 and 4, Professional and Enterprise editions. ODAC supports Oracle 7.3, 8, and Oracle 8i, including Personal Oracle.

CoRe Lab Co.
Price: US$99 for a single-developer license; US$249 for source code (in addition to developer license).
Phone: (800) 903-4152 (US orders only).
Web Site: http://www.crlab.com

Delphi TOOLS

New Products and Solutions

20/20 Offers PC-Install 7

20/20 Software, Inc. introduced two new versions of its PC-Install program for building software installations in Windows environments: *PC-Install 7* and *PC-Install 7 with Internet Extensions*.

Both versions of PC-Install include unlimited distribution licenses and one year of free technical support. Significant new benefits for software developers in PC-Install 7 include a refined, more flexible, and intuitive interface that makes building installations quicker; several additional editing tools, including global search and replace, template, and direct command editing, multiple undo/redo, and drag-and-drop editing; an expanded list of PC-Install system variables that makes locating and using system resources to control installations easier, more efficient, and transparent to the user; broader support for Visual Basic (VB) projects; automatic file collection for VB 4 through VB 6; added support for automatic file collection in ODBC and Delphi projects; developer-defined local variables that allow collecting nearly unlimited amounts of data from an end user; and a "Smart" uninstall feature that supports incremental removal of installed components.

For the end user, PC-Install 7's added features include a new wizard-style uninstall interface and automatic support for the Add/Remove Programs control panel. Users installing software over the Internet using PC-Install 7 with Internet Extensions no longer need to restart their installation from the beginning when their connection fails. Version 7 restarts automatically at the point in the installation where the connection was lost.

20/20 Software, Inc.
Price: US$249, or US$199 if downloaded from 20/20 Software or its distribution partners; PC-Install 7 with Internet Extensions, US$449, or US$399 if downloaded.
Phone: (800) 735-2020
Web Site: http://www.twenty.com

RightWare, Inc. Announces ARMS 2.0

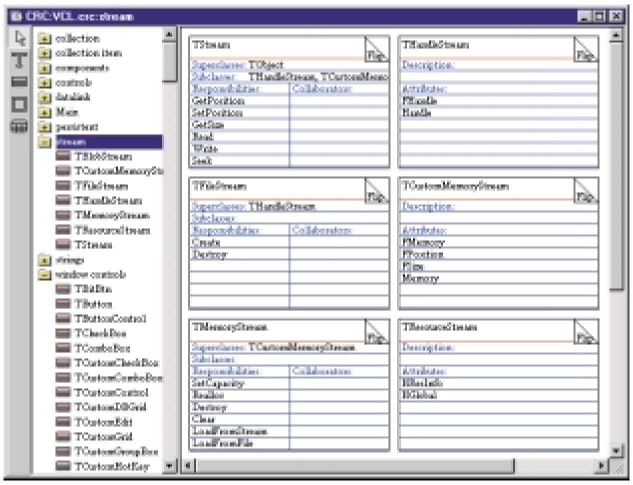
RightWare, Inc. announced the *Active Risk Management System 2.0* (ARMS), a two-component suite of easy-to-use, easy-to-deploy, enterprise-class, team-based, risk-management software.

ARMS allows users to: unify their project team by improving risk communication and have team members stay in sync with changing project risks and get up-to-date information with the ARMS Team Member component; facilitate communication by allowing the team members to engage in online dialog about project risks with the ARMS Discussion Group feature; foster a project environment that does not exclude risk identifiers; create a consistent understanding of the project risks; import resources from Microsoft Project into ARMS Team Manager as users; create and assign action items to team members; get a better picture of how mitigation affects project schedules; print reports, such as the Risk Management Plan, Mitigation Plan, and Top 10 Risk List; export reports to DOC, RTF, PDF, and TXT for further project visibility; optimize the identification of risks by selecting from over 200 risks and over 14 categories; define project risk attributes, such as time frame, project phase found, and status; link risks and track the impact of change on related risks; and more.

RightWare, Inc.
Price: ARMS Team Manager, US$1,599 (Enterprise Edition) and US$999 (Standard Edition); ARMS Team Member, US$1,299 (Enterprise Edition) and US$799 (Standard Edition).
Phone: (877) 717-ARMS
Web Site: http://www.right-ware.com

Excel Software Announces WinTranslator 2.0.2

Excel Software announced *WinTranslator 2.0.2* for software design and code re-engineering. WinTranslator is used in conjunction with Excel's WinA&D product to create class models or structure charts from source code. It can also create CRC cards for the QuickCRC design tool from Delphi, C++, or Java.



The WinTranslator 2.0.2 update makes several improvements to version 2.0, including additional customization options to the re-engineering wizard that steps the user through the process of generating design from code, and additional optimizations for Java re-engineering.

Excel Software
Price: US$495 for a single license; site licenses are available.
Phone: (515) 752-5359
Web Site: http://www.excelsoftware.com

devSoft Releases ICK Version 2.0

devSoft Inc. released version 2.0 of *Internet Commerce Kit (ICK)*, a developer's toolkit for secure access and manipulation of Internet data. The toolkit includes native Internet and intranet development components for development environments such as Delphi, C++Builder, Visual Basic, Visual C++, and others.

The new release introduces vGrid ('virtual' Grid), used to dynamically exchange relational (tabular) data over the Web. The component may be used in both server and client applications to serve and access data.

XML is used as the interchange format. This approach makes it possible to link together a variety of applications from a variety of platforms, built with different development tools. The new release brings a number of significant improvements in the other components of ICK as well, including HTTP, HTTPS, FTP, XMLp, and NetDial. Improvements include better programmatic access to interactive features and security, as well as support for the latest versions of development environments, such as Delphi 5.

ICK 2.0 costs US$245. For more information call (919) 493-5805 or visit http://www.dev-soft.com.

Inprise Announces New Agreement for VisiBroker

Scotts Valley, CA — Inprise Corp. announced that its VisiBroker CORBA Object Request Broker will be utilized in Cisco Systems' CiscoWorks2000 family of enterprise network management products.

As part of the agreement, Inprise's VisiBroker will be embedded in Cisco's Common Management Foundation, a software infrastructure used by CiscoWorks2000 network management applications.

CiscoWorks2000 is a family of network management solutions combining Cisco's switch and router management with Internet standards to transform network management. Inprise's VisiBroker facilitates the development and deployment of distributed enterprise applications that are scalable, flexible, and easily maintained. As systems get more complicated with numerous sub-applications, this combination becomes more important for companies thriving in today's Internet economy.

For more information on Cisco, visit http://www.cisco.com.

Inprise Announces Embedded Database Solution with InterBase Version 5.6

Scotts Valley, CA — Inprise Corp. announced the availability of InterBase version 5.6, the latest version of Inprise's embedded database solution. This new version is now available on the Novell NetWare and Windows platforms and includes updates to SQL functions and roles, as well as performance enhancements.

InterBase 5.6 is certified on NetWare 4.2 and 5.0, Microsoft Windows NT 4.0 SP4, Windows 95, and Windows 98.

InterBase 5.6 Server software for Windows or NetWare is priced at US$200 for a one-user server license. Additional users are US$150 each, US$1,200 for 10 users, or US$2,100 for 20 users. Local InterBase 5.6 software for Windows is available on CD-ROM for US$50. Activation keys for Local InterBase 5.6 for Windows are US$60 each, US$800 for a package of 20, or US$2,000 for a package of 100. There is no Local InterBase for NetWare.

For more information, visit http://www.inprise.com.

Dale Fuller Outlines Strategy in Support of Application Service Providers

San Diego, CA — Inprise Corp. interim President and Chief Executive Officer Dale Fuller unveiled the company's new strategy in support of application service providers (ASPs). In addition, Fuller announced plans to create Inprise AppServices, a new service to integrate software and services from many application service providers into a single suite.

AppServices will allow customers to access business application sources through a Web-based portal that includes a unified suite of communication/collaboration/productivity tools, such as calendaring, messaging, and discussion forums.

Inprise and its partners plan to build and host AppServices. AppServices will enable end users to access their applications and desktop via any networked device, operating system, or protocol, using a standard browser interface.

Inprise's strategy for ASPs consists of three layers.

The first, a "user layer," provides users with a single point of entry and universal registration system from which to access applications from various ASPs being used within a company.

The second, a "transport layer," allows a user to access ASP-hosted applications on different types of devices.

Finally, a "messaging layer" allows different applications from various ASPs to communicate with one another.

According to International Data Corporation, worldwide spending for ASPs will increase from US$150 million in 1999 to over US$2 billion by 2003.

For more information, visit http://www.inprise.com.

ICG Announces the Launch of ComputerBookstore.com

Elk Grove, CA — Informant Communications Group, Inc. announced the launch of ComputerBookstore.com, an online bookstore featuring a wide variety of computing, gaming, certification, training, science, and business books, in addition to training materials, videos, and documentation.

ComputerBookstore.com offers a wide variety of titles from all the major publishers, including Addison-Wesley, IDG, Macmillan, Microsoft Press, O'Reilly & Associates, Osborne/McGraw-Hill, SYBEX, Wiley, and WROX, as well as smaller independent publishers.

In addition, ComputerBookstore.com features professional reviews, e-mail alerts, and other various interactive activities.

ComputerBookstore.com guarantees a savings of up to 41 percent off the suggested retail price for all on-sale items. January 2000 will feature various sales throughout the month, including sales on gaming, Osborne/McGraw-Hill, and Microsoft Press books.

Additional information regarding this new online bookstore is available at http://www.ComputerBookstore.com.

table of CONTENTS

By Xavier Pacheco

Mediator Pattern

Part I: Introduction to Process Control Frameworks

In March of last year (1999), I began a series of articles on design patterns and how to use such patterns within the Delphi VCL framework. In those articles, I discussed the Singleton, Template Method, and Builder patterns.

This month, we'll use the Mediator pattern to illustrate how to solve the problem of process flow control in a batch processing system. I initially planned for this entire example to appear in one article — until I realized that there was too much information. Therefore, the next three articles will focus on this pattern and how to use it in a distributed process control system.

Let's begin by defining the pattern in detail. As stated in the classic, *Design Patterns* [Addison-Wesley, 1995], by Erich Gamma, et al., the Mediator pattern is used to "Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently."

The Mediator pattern is used when objects must communicate with one another in well-defined ways, but the communication is complex. The Mediator pattern can simplify the communication between objects. Additionally, the Mediator pattern can also unbind the dependencies between objects. This facilitates object reuse and customizations. There are four participants to the Mediator pattern (see Figure 1):

- Mediator defines an abstract class or interface for communication between the Colleague classes.
- A ConcreteMediator implements the abstract mediator. Each ConcreteMediator coordinates the Colleague communication.
- The Colleague is an abstract class or interface that defines the class to be managed by the Mediator class.
- ConcreteColleague classes are self-contained classes that communicate with a Mediator class. Knowledge of the Mediator class is typical but not necessary.

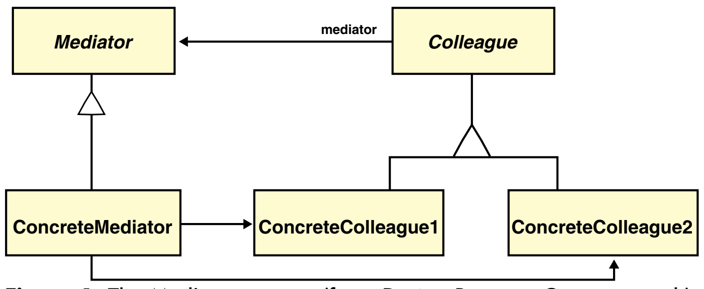


Figure 1: The Mediator pattern (from *Design Patterns*, Gamma, et al.).

Figure 2 shows how the Mediator pattern works when implemented. The ConcreteMediator serves as "traffic cop," controlling the execution of, and communication between, the ConcreteColleague objects. In this scenario, ConcreteColleague objects don't communicate directly with each other; rather, they rely on the Mediator to enforce inter-object communication. This requires a standard form of communication as defined by the abstract class definitions or interfaces.

Uses and Motivation

The Mediator pattern removes the complexities and dependencies of inter-object communication. This objective is especially applicable to process control. Processes by definition are subject to change because of constant improvements in the way we design and handle information. Planning and designing for changes in a process reduce the pain of making modifications.

Any repetitive process we see in business today can benefit from a system design that incorporates the Mediator pattern. Before we delve into the technicalities of an example implementation, however, let's address the motivation of including the Mediator pattern in the design.

Applications focused on managing processes involve the systematic execution of tasks and reporting of task status. If each task were responsible for knowing subsequent tasks and passing behavioral statistics to those tasks, it's easy to see how the resulting application would be constrained

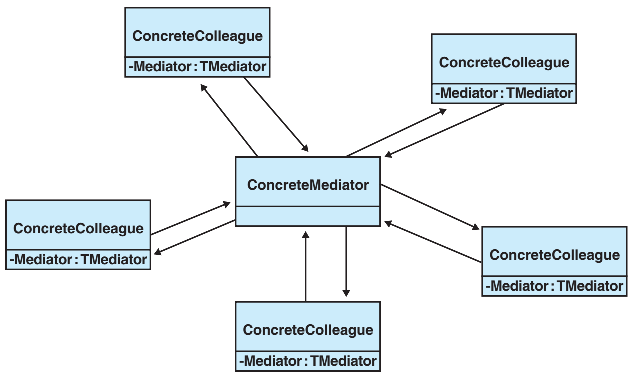


Figure 2: Mediator pattern implementation.

to the initial definition of the process, i.e. the sequence of tasks necessary to complete the process. Changes in the sequence of tasks and/or the passing of behavioral parameters between tasks, in this instance, would require updates to all objects involved — even if the individual task behavior had not changed (see Figure 3).

In Figure 3, you see that if we were to modify the input and/or output parameters of any given task, the changes to the system would impact more than just the changed task. Additionally, this design restricts the execution of tasks to the specified sequence. It becomes difficult for the results of one task to determine the next task to be implemented. A task would have to know about all tasks that it might invoke. This violates the intent to loosely couple each task. The Mediator pattern addresses these issues.

Adding a Mediator between the task objects removes the task-to-task dependencies. The application is now free to alter the sequence of tasks without modifying any of the individual tasks. Additionally, this design can allow generic parameters (process modifiers, performance statistics, error conditions, etc.) of one task to be passed to tasks that aren't necessarily next in sequence. Figure 4 illustrates how this might look; notice the similarities with Figure 2.

Task Control Example

This article's example illustrates a simplified architecture by which you can control a series of tasks/processes. To reduce any possible confusion, "task" refers to an encapsulated unit of work, and "process" refers to a series of tasks coordinated to achieve a defined purpose.

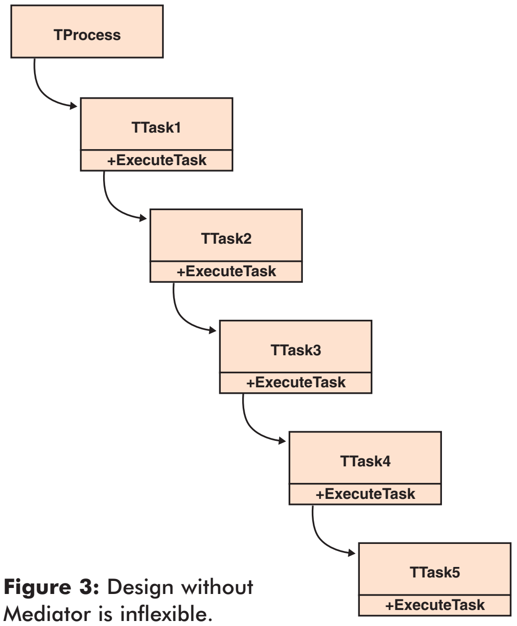


Figure 3: Design without Mediator is inflexible.

The example we'll present is generic for now; we'll expand on its capabilities in later articles. The intent is simply to illustrate how the Mediator controls the invocation of tasks, and how tasks can be invoked non-sequentially.

Defining the *ITask* Interface

ITask defines an interface with a single function, *ExecuteTask* (see Figure 5).

This function takes an OleVariant as a parameter and returns an OleVariant. The reason is due largely to how the interface will be used in a distributed environment. In my initial design of the process control system, I had hard-coded parameters to exactly those needed by the specific tasks. This led to problems when a task's parameters required modification, especially when that task had already been deployed on other machines. I needed a way to re-implement a task and to re-deploy it without having to unregister/register the task on a given machine. Also, I did not want to have to re-compile the calling module that also passed in hard-coded parameters. In a later article, we'll see how to use a *TClientDataset* to implement parameters for each task.

Defining the Mediator Interface

IProcess defines two methods, *ExecuteProcess* and *MessageToProcess* (see Figure 6). Implementations of *IProcess* will serve as the Mediator objects.

MessageToProcess is a method that will be used by each implementation of *ITask* to allow a message to be passed back to the Mediator class of a given *ITask* implementation. *ExecuteProcess* is similar to *ExecuteTask* in that it's invoked by the client of the process.

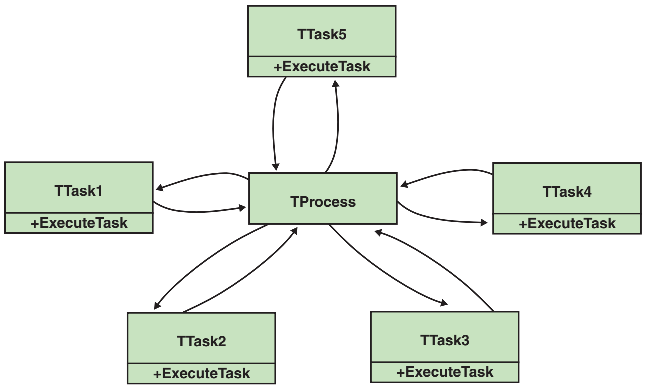


Figure 4: Design with Mediator is very flexible.

```
unit IntfTask;

interface

type
  ITask = interface
    ['{712185C1-810F-11D3-8117-00008638E5EA}']
  function ExecuteTask(AInParams: OleVariant):
    OleVariant;
  end;

implementation

end.
```

Figure 5: The *ITask* interface.

Implementing *ITask*

Figure 7 illustrates the implementation of the *ITask* interface.

As you can see, *TTask* is implemented as an abstract class. This has two primary purposes. First, we want to propagate the requirement for descendant classes to implement the *ExecuteTask* method. Second, we want to provide a mechanism by which the *TTask* descendants would know about, or have a reference to, their Mediator object. This is done through the *TTask*'s constructor.

```
unit IntfProcess;

interface

type
  IProcess = interface
    ['{ 712185C3-810F-11D3-8117-00008638E5EA }']
  procedure MessageToProcess(AMessage: string);
  function ExecuteProcess(AInParams: OleVariant):
    OleVariant;
  end;

implementation

end.
```

Figure 6: The *IProcess* interface.

```
unit TaskClass;

interface

uses
  IntfProcess, IntfTask;

type
  TTask = class(TInterfacedObject, ITask)
  protected
    FMediator: IProcess;
  public
    function ExecuteTask(AInParams: OleVariant):
      OleVariant; virtual; abstract;
    constructor Create(AMediator: IProcess);
  end;

implementation

constructor TTask.Create(AMediator: IProcess);
begin
  inherited Create;
  FMediator := AMediator;
end;

end.
```

Figure 7: *ITask* implementation *TTask*.

```
unit ProcessClass;

interface

uses IntfProcess;

type
  TProcess = class(TInterfacedObject, IProcess)
  public
    function ExecuteProcess(AInParams: OleVariant):
      OleVariant; virtual; abstract;
    procedure MessageToProcess(AMessage: string);
      virtual; abstract;
  end;

implementation

end.
```

Figure 8: Implementing *IProcess* with *TProcess*.

You'll also notice that we've made *TTask* a descendant of *TInterfacedObject* so the *IUnknown* methods are implemented. *IUnknown* is the root definition from which all interfaces descend. *TInterfacedObject* is a class that implements *IUnknown*'s reference counting methods, so your classes don't have to.

The *IProcess* Class

Figure 8 illustrates the implementation of *IProcess* as an abstract class.

TProcess descends from *TInterfacedObject* for the same reasons mentioned for *TTask*. *TProcess* implements the *IProcess* interface, and defines *TProcess* as an abstract class. Again, we want to force implementations of *ExecuteProcess* and *MessageToProcess*.

```
unit DemoProcess;

interface

uses
  Classes, ProcessClass;

type
  TDemoProcess = class(TProcess)
  private
    FMessageStrings: TStrings;
  public
    function ExecuteProcess(AInParams: OleVariant):
      OleVariant; override;
    procedure MessageToProcess(AMessage: string); override;
    constructor Create(AMessageStrings: TStrings);
  end;

implementation

uses
  IntfTask, Task1, Task2, Task3, Task4;

constructor TDemoProcess.Create(AMessageStrings: TStrings);
begin
  FMessageStrings := AMessageStrings;
end;

function TDemoProcess.ExecuteProcess(
  AInParams: OleVariant): OleVariant;
var
  Task: ITask;
  i: Integer;
  InParam: Integer;
begin
  Randomize;
  InParam := AInParams;
  for i := 1 to 10 do begin
    case InParam of
      1: Task := TTask1.Create(Self);
      2: Task := TTask2.Create(Self);
      3: Task := TTask3.Create(Self);
      4: Task := TTask4.Create(Self);
      else
        Task := TTask3.Create(Self);
    end;
    InParam := Task.ExecuteTask(InParam);
  end;
end;

procedure TDemoProcess.MessageToProcess(AMessage: string);
begin
  FMessageStrings.Add(AMessage);
end;

end.
```

Figure 9: *TDemoProcess*, the concrete *TProcess* implementation.

```
unit Task1;

interface

uses TaskClass;

type
  TTask1 = class(TTask)
  function ExecuteTask(AInParams: OleVariant):
    OleVariant; override;
  end;

implementation

{ Process will get executed here. This would consist of
  reading the parameters from AInParams, using them and
  then creating the output params which are passed back
  as Result. }
function TTask1.ExecuteTask(AInParams: OleVariant):
  OleVariant;
begin
  FMediator.MessageToProcess(
    'Executing TTask1.ExecuteTask');
  Result := Random(5);
end;

end.
```

Figure 10: *TTask1*, an implementation of the *TTask* abstract class.

The Concrete *TProcess* Implementation

Figure 9 illustrates a concrete implementation of the *TProcess* abstract class.

As mentioned earlier, we want to illustrate how the Mediator pattern can be used to invoke tasks and how it can do so in a non-sequential fashion. This simulates a scenario where a task can specify the next task to get executed in a sequence. As this series progresses, we'll expand on this design to allow for asynchronous invocation of tasks by the Mediator object.

TDemoProcess is a simple Mediator class that contains a reference to a *TStrings* object and adds strings to those objects in the *MessageToProcess* method. Therefore, *TTask* objects can communicate to the *TDemoProcess* through the *MessageToProcess* method. The reference to *FMessageStrings*, the *TStrings* instance, is set up in the constructor for *TDemoProcess*.

TDemoProcess.ExecuteProcess creates and executes four different implementations of the *TTask* class. We'll use a randomly generated return value from each *TTask* implementation to determine the next *TTask* implementation to invoke in the sequence. We do this 10 times before leaving the procedure. This effectively illustrates non-sequential invocation of *TTask* objects.

The Concrete *TTask* Implementation

Figure 10 illustrates one of five implementations of the *TTask* abstract class.

The implementation of *TTask* is simple; it first passes a message back to its Mediator through the *MessageToProcess* method, then passes back a random number from 0 to 4. As shown in Figure 9, *TDemoProcess* uses this randomly generated result to determine the next *TTask* implementation to invoke.

This implementation of the Mediator pattern is simple, yet illustrative as an expandable model for a Mediator pattern. We've created a simple set-up where a process (*TDemoProcess*) can create and invoke tasks in a non-sequential fashion by using the resulting values from each task to determine the next task to invoke. This is a very simple implementation of a much more complex set-up, where the process determines which task to implement based on status values contained in a database server.

Conclusion

The Mediator pattern is an ideal approach to any system that requires some form of process control, and where extensibility and loosely coupled classes are essential. In the next article in this series, we'll show how to use another pattern to enhance the Mediator capabilities shown here. We'll illustrate how to allow a variable number of parameters to be passed to each task and still maintain loose coupling between each task and between tasks and their Mediator object.

Many thanks to John Wilcher for his feedback and help with this article. A Consulting Manager for Inprise Corp., John provided his experience and some of the initial content for this article. He also provides architectural and design consulting services as a Principal Consultant for Inprise PSO. You can write John at jwilcher@inprise.com, and visit Inprise's PSO Web site at http://www.inprise.com/services. ∆

References

I use these books whenever considering applying a design pattern to a given problem. They are a must for any developer serious about learning and using design patterns:

- *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, et al. [Addison-Wesley, 1995].
- *The Design Patterns Smalltalk Companion* by Sherman R. Alpert, et al. [Addison-Wesley, 1998].
- *Patterns in Java, Volume 1* by Mark Grand [John Wiley & Sons, 1998].

Xavier Pacheco is the president and chief consultant of Xapware Technologies Inc., where he provides consulting services and training. He is also the co-author of *Delphi 5 Developer's Guide* published by SAMS Publishing. You can write Xavier at xavier@xapware.com, or visit http://www.xapware.com.

CORBA / Delphi 4, 5 / VisiBroker / JBuilder

By Dennis P. Butler

CORBA

Part I: Creating a Server

From the stand-alone personal computer, to heterogeneous distributed clients and servers, and everywhere in between, the computing industry has evolved dramatically over the past several decades. Computer professionals are constantly required to use their skills to the utmost in their current environment, and then be able to migrate to the next level when the current framework becomes too complicated or restricting. The evolution of computing continues to point toward a common goal: simplify the work process by better planning, faster development, and sharing of resources.

The CORBA architecture — short for Common Object Request Broker Architecture — was designed to accomplish this goal. The CORBA architecture defines and implements the framework for applications to communicate across such previously unbreakable boundaries as multiple operating systems and programming languages. This capability is achieved through the use of a common interface and information passing mechanism implemented in different programming languages.

There are several factors that set CORBA apart from competitive proprietary information sharing technologies. First of all, CORBA is an open standard; that is, the specification is constantly being reviewed and updated by the OMG, or Object Management Group. This group is made up of hundreds of companies worldwide that decide how to evolve the CORBA specification. This process of evolution has been occurring since 1991, when CORBA 1.0 was released. Another feature that sets CORBA apart from other technologies is that the interface is common among languages, not the implementation of it. Other information-sharing methods rely on operating-system-specific implementations to pass information. While this may be useful in LAN/WAN or intranet environments, where operating systems can be standardized, true distributed applications that need to operate over any OS require a more complete solution, such as CORBA. With CORBA, the client doesn't need to know any details of how the object that it will obtain from a server was implemented.

The starting point for CORBA applications is the interface that applications share when passing information. This common interface that defines what information is going to be passed is called IDL, short for Interface Definition Language. IDL is its own language, although the syntax is similar to that of Java and C++. As its name implies, the only purpose of this language is to define the interface for objects that will be passed between CORBA applications. The implementation and use of these objects is done in the specific target language chosen. The only stipulation here is that the target language has facilities to map to the CORBA architecture.

This is where Delphi comes in. CORBA development is typically associated with C++ or Java development. However, even non-object-oriented languages, such as C and COBOL, have mappings to CORBA, and thus can take advantage of the open architecture. As we'll see later in this article, Delphi employs several methods to use CORBA in an application. CORBA can be implemented through the use of the Type Library editor for easily creating IDL interfaces, through MIDAS to connect to CORBA data, and soon Delphi will gain direct facilities to compile IDL code into Pascal source, which can be used to implement and use the CORBA objects. Much like the IDL2JAVA utility, this IDL2PAS utility will be available soon to Delphi developers to give complete control and flexibility in creating CORBA applications.

VisiBroker CORBA

VisiBroker is the ORB (Object Request Broker) used throughout this two-article series and its exam-

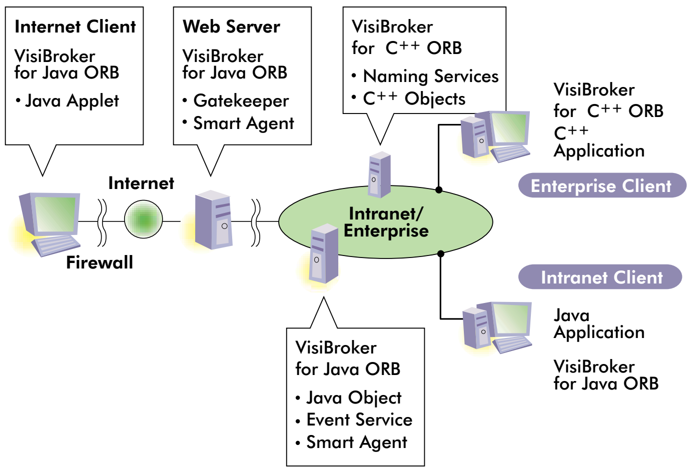


Figure 1: Sample high-level CORBA implementation.

ples. VisiBroker is the Inprise implementation of the CORBA standard that adds many additional features to assist developers when creating applications. Support for thread management and connection management is included, as well as many libraries and other utilities that are created to assist developers when developing CORBA applications. Knowledge of the intricacies of VisiBroker isn't required for this series. For the examples that are used, VisiBroker additions and standard CORBA features are used together, much as they would be for actual production situations.

Let's take a quick look at a high-level diagram of how CORBA fits into distributed applications (see Figure 1).

As you can see in this standard Inprise diagram, there can be many levels of connection across boundaries, such as the Internet, an intranet, or other internal networks. This diagram introduces many VisiBroker-specific items, such as the Gatekeeper, Smart Agent, Naming Services, and Event Service. All we need to grasp from the diagram at this point is that one or more IDL interfaces for CORBA objects has been generated, and instances of those server object implementations are being passed to various clients. As shown in the diagram, clients can include the Internet Client running a Java applet, or the C++ application running on the corporate intranet.

This two-article series will show how Delphi can be used in this network to also take advantage of the CORBA architecture. In this article, we create a CORBA server. Next month, in Part II, we'll address CORBA clients.

Before we get started with Delphi, let's take a quick look at how applications communicate with CORBA. This information will be relevant later in the article when using Delphi to implement this technology. We will start here with a very simple example of how a basic CORBA application can be started between two machines:

- First, the ORB Smart Agent (osagent) must be run on a machine on the network. The osagent will keep track of all server object implementations that have been registered with it, as well as keeping track of other osagents.
- Next, the server application must be run. This will register its object(s) with the osagent to let it know that it has object implementation(s) available for client applications.
- A client application is started. When the client application requires a server object, it will issue a UDP broadcast to find the closest osagent to search for that implementation. The osagent will find the object implementation that the client is looking for, thus allowing a connection to be established between the client and server. The client can now access the server object directly.

Now that we know the steps that take place in a basic CORBA application, we'll apply them to Delphi to see how they're accomplished. For this first example, we'll create an online auction demonstration, where the server will keep track of a particular product, and clients will bid against each other to try to buy the product. For each successful bid, the client application will notify that the bid was successful, and will update the screen to show the high-bid amount. Further bids by other clients will now have to outbid that new highest amount to win the product (which, of course, is a copy of Delphi 5 Enterprise Edition).

Example 1: The Online Auction

Our first step in this example is to create the CORBA object for our server, and create the server that will implement this object. As I mentioned earlier, the CORBA objects are defined by IDL. Delphi developers don't need to know IDL to create their object; instead, this can be done through the use of the Type Library editor. This handy utility allows visual creation of objects and their interfaces. This utility can also be used later to export to IDL for use in other implementations of the object.

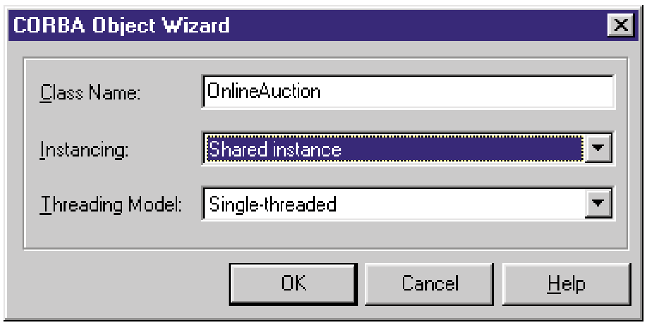
To create the server and its object, start a new Delphi application and save the form and project. You may want to shrink the dimensions of the form, as this will be your server application running on your machine. For this example, I have named the files Cserver.dpr and Cmain.pas. From the main menu of Delphi, select File | New, then select the CORBA Object item on the Multitier page. The CORBA Object Wizard will be displayed (see Figure 2).

As you can see, the object to be defined is named OnlineAuction, will be a shared instance, and will be single-threaded. The information required by this dialog box is described in more detail here.

Class Name. Enter the base name of the object that implements the CORBA interface for your object. Filling in the class name will do two things; it will create a class of this name with a "T" prepended, and create an interface for the class using this name with an "I" prepended.

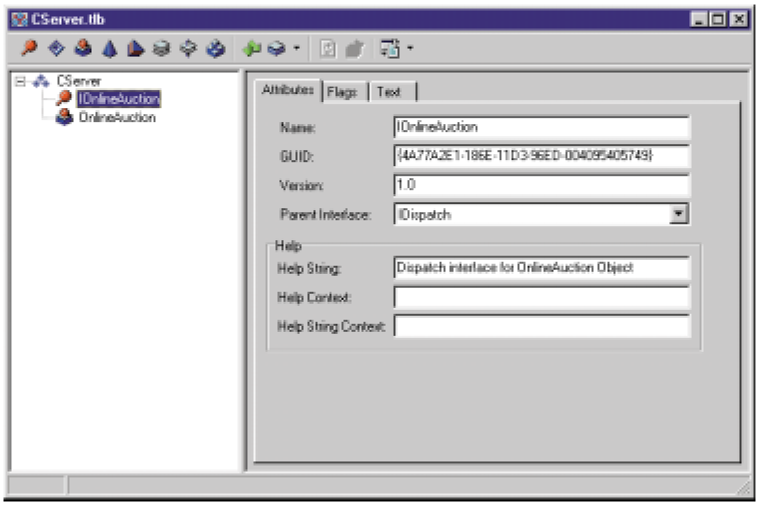
Instancing. Use the Instancing combo box to indicate how your CORBA server application creates instances of the CORBA object. There are two possible values:

- Instance-per-client — A new CORBA object instance is created for each client connection. The instance persists as long as the connection is open. When the client connection closes, the instance is freed.



Class Name: OnlineAuction
Instancing: Shared instance
Threading Model: Single-threaded

Figure 2: Creating the CORBA object interface.


CServer.tlb

CServer
IOnlineAuction
OnlineAuction

Attributes | Flags | Text

Name: IOnlineAuction
GUID: {4A77A2E1-186E-11D3-96ED-004095405749}
Version: 1.0
Parent Interface: IDispatch

Help
Help String: Dispatch interface for OnlineAuction Object
Help Context:
Help String Context:

Figure 3: Delphi's Type Library editor.

- **Shared instance** — A single instance of the CORBA object handles all client requests. Because the single instance is shared by all clients, it must be stateless.

Threading. Use the **Threading Model** combo box to indicate how client calls invoke your remote data module's interface. Again, there are two possible values:

- **Single-threaded** — Each object instance is guaranteed to receive only one client request at a time. Instance data is safe from thread conflicts, but global memory must be explicitly protected.
- **Multithreaded** — Each client connection has its own dedicated thread. However, the object may receive multiple client calls simultaneously, each on a separate thread. Both global memory and instance data must be explicitly protected against thread conflicts.

In this example, the server object will be a shared instance because we want all clients to access the same auction object, so they can bid against each other. If we were writing an object for a banking application, an object could be created that would contain information specific to a banking account of the customer running the client application. In this case, an **Instance-per-client** setting would be more appropriate, since a separate object would be created for each client, making the contents of that object private to the client.

The object is also created for single threading; since only one client request will be processed at a time, multi-threading isn't necessary. Multi-threading is very useful when developing very large applications that require a higher level of flexibility in situations where the server must be able to handle multiple requests that may be occurring simultaneously. For this simple example, we won't take advantage of this feature.

Click **OK** to create the new unit and save the file. This will create the Pascal shell for the CORBA object interface. Instead of having to type in the interface manually, Delphi allows us to visually create the object interface using the Type Library editor (see Figure 3). (The Type Library editor is available from the main menu by selecting **View | Type Library**.)

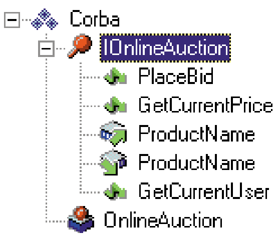


Figure 4: Completed type library tree.

The Type Library editor allows us to specify all the information we need to define the interface of our CORBA object. For this example, we want to create a server object for our online auction that will hold information about the latest high-bid amount and person. We will also add a property for the product that is being bid upon. We also want to add methods to place a new bid, and check information about the current bid. In true object-oriented fashion, properties cannot be modified directly. They must use accessor methods to change their values. As we'll see, the Type Library editor takes care of this as well.

The Type Library editor is also used to define interfaces to COM objects; in fact, this was the original purpose of the Type Library editor. Because of this, it has some features that aren't used for CORBA objects. An example of this is the "Help" information shown in Figure 3. Also, some of the data types that are available in the Type Library editor may not be CORBA-compliant data types. Developers can easily research CORBA data types through the Delphi or VisiBroker Help files.

As we can see in Figure 3, an Interface and CoClass have been created for our CORBA object. All we are concerned with is the Interface. The CoClass that has been created is COM-specific, and can be ignored. Note that the interface has taken our CORBA object name and prepended it with an "I", as mentioned earlier.

Methods and properties are added by right-clicking on the *IOnlineAuction* interface and selecting **Method** or **Property** from the **New** menu. Add these methods to the *IOnlineAuction* interface:

- *PlaceBid* returns an Integer, takes a Double (call it *Amount*) and WideString (call it *CustomerName*) as parameters
- *GetCurrentPrice* returns a Double, no parameters
- *GetCurrentUser* returns a WideString, no parameters

Add the *ProductName* property to the *IOnlineAuction* interface.

When entering the parameters for the methods in the Type Library editor, the following information is required:

- **Modifier** specifies the nature of the parameter, such as whether it should be treated as an **in** parameter, **out** parameter, etc.
- **Name** is the name of the parameter.
- **Type** is the data type of the parameter. The list contains the list of CORBA-compliant data types.
- **Default Value** specifies whether the parameter will have a default value.

For the purposes of this example, the modifier was kept blank for all parameters because no special setting was needed for this example. This defaults the parameter to the **in** setting of IDL; thus modifications made to the parameter variable once the called procedure is completed won't be reflected when control is returned. In fact, the Delphi compiler will show a hint for a parameter within its method if an attempt is made to modify the value of the parameter. There are several types of modifiers that can be used that correspond to IDL parameter types. The most commonly used parameters in IDL are **in**, **out**, and **inout**. This means that parameter information flows into the server, out from the server, or both. This is done with the Type Library editor settings of blank(**in**), out(**out**), and var(**inout**).

When completed, the tree view for the Type Library editor should look like Figure 4. Click on the "Refresh Interface" button (it looks like the two-arrowed recycle symbol) from the Type Library editor to synchronize the source file for the CORBA object. Note: This isn't entirely WYSIWYG, as is the standard Delphi IDE; the "Refresh Interface" button must be clicked.

Once the refresh button has been clicked, the Type Library editor can be closed, and the source file can be saved (csrvobj.pas in this case). We now have our server object interface defined, and the Pascal code shell from which we can add functionality for the object. The Type Library editor has created some files for us, such as the _TLB stub file that's created based on the server application project name. In this example, since the project was named CServer, it's CServer_TLB.pas. Since this file is automatically generated, no additional work is needed on it. This file sets up stub and skeleton classes for the server object, as well as defining several other classes that may be used, such as the CORBA object factory class and the COM CoClass class. The only thing we really need to know at this point is that the CORBA shell has been created for us from how we defined the interface in the Type Library editor, and a TLB file has been created from which we can get our object reference for the server.

Our server source file csrvobj.pas has been filled in from the Type Library editor with the methods and properties that were defined. The empty shell, csrvobj.pas, is shown in Listing One. Now we need to code the implementation of the object. We need to add a few private variables to hold the current high-bid price, the current high-bid customer, and the product being bid on. We also need to initialize these private variables in the constructor for the object. Finally, we need to implement the object with code to provide functionality to the methods that have been created for us. The completed source, with comments, is shown in Listing Two.

All that was provided by Delphi was the code shell; the rest had to be filled in to give the object interface an implementation. We now have the server for our object. To use this server of our CORBA object, all we need to do is add the csrvpas unit to the **uses** clause of any form of a project. When this is done, the initialization code for the object will be fired when that form is used. Thus, the server will be started, and an object will be created that is available for use.

Until Next Month

That's it for this article. Next month, we'll implement CORBA clients, including one written in Java using Borland's JBuilder product. See you then. Δ

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\JAN\DI200002DB.

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 — csrvobj.pas shell

```
unit csrvobj;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  ComObj, StdVcl, CorbaObj, CServer_TLB;

type
  TOnlineAuction = class(TCorbaImplementation,
                         IOnlineAuction)
  protected
    function Get_ProductName: WideString; safecall;
    function GetCurrentPrice: Double; safecall;
    function GetCurrentUser: WideString; safecall;
    function PlaceBid(Amount: Double;
      const CustomerName: WideString): Integer; safecall;
    procedure Set_ProductName(const Value: WideString);
      safecall;
  end;

implementation

uses
  CorbInit;

function TOnlineAuction.Get_ProductName: WideString;
begin

end;

function TOnlineAuction.GetCurrentPrice: Double;
begin

end;

function TOnlineAuction.GetCurrentUser: WideString;
begin

end;

function TOnlineAuction.PlaceBid(Amount: Double;
  const CustomerName: WideString): Integer;
begin

end;

procedure TOnlineAuction.Set_ProductName(
  const Value: WideString);
begin

end;

initialization
  TCorbaObjectFactory.Create('OnlineAuctionFactory',
    'OnlineAuction','IDL:CServer/OnlineAuctionFactory:1.0',
    IOnlineAuction, TOnlineAuction, iSingleInstance,
    tmSingleThread);
end.
```

End Listing One

Begin Listing Two — Implemented csrvobj.pas

```
unit csrvobj;

interface

// Note the included units for ComObj, CorbaObj,
// and Cserver_TLB.
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  ComObj, StdVcl, CorbaObj, CServer_TLB;

// Class is defined as a CORBA class that implements
// the IOnlineAuction interface.
type
  TOnlineAuction = class(TCorbaImplementation,
                         IOnlineAuction)
  private
    // Private variables to hold object information
    // about auction.
    FProductName : WideString;
    FCurrentPrice : Double;
    FCurrentCustomer : WideString;
  public
    // Override create to initialize private variables.
    constructor Create(Controller: IObject;
      AFactory: TCorbaFactory); override;
  protected
    // Accessor methods for FProductName property.
    function Get_ProductName: WideString; safecall;
    procedure Set_ProductName(const Value: WideString);
      safecall;
    // Function to get the current price for the
    // auction product.
    function GetCurrentPrice: Double; safecall;
    // Function to get the current customer for the
    // auction product.
    function GetCurrentUser: WideString; safecall;
    // Function to place a new bid.
    function PlaceBid(Amount: Double;
      const CustomerName: WideString): Integer; safecall;
  end;

implementation

// Included with Delphi to initialize CORBA object.
uses
  CorbInit;

// Overridden create for our object.
constructor TOnlineAuction.Create(Controller: IObject;
  AFactory: TCorbaFactory);
begin
  inherited;
  // Initialize our private variables.
  FProductName := '<NA>';
  FCurrentCustomer := '<NA>';
  FCurrentPrice := 0;
end;

// Method to get the property value for the current
// auction product.
function TOnlineAuction.Get_ProductName: WideString;
begin
  Result := FProductName;
end;

// Method to set the property value for the current
// auction product.
procedure TOnlineAuction.Set_ProductName(
  const Value: WideString);
begin
  FProductName := Value;
end;

// Method to get the current price of the high bid.
function TOnlineAuction.GetCurrentPrice: Double;
begin
  Result := FCurrentPrice;
end;

// Method to get the current customer name of the high bid.
function TOnlineAuction.GetCurrentUser: WideString;
begin
  Result := FCurrentCustomer;
end;

// Method to place a new bid: Take parameters for amount
// of bid and customer who is placing the bid.
function TOnlineAuction.PlaceBid(Amount: Double;
  const CustomerName: WideString): Integer;
begin
  if Amount > FCurrentPrice then
    begin
      FCurrentPrice := Amount;
      FCurrentCustomer := CustomerName;
      Result := 1;
    end
  else
    Result := 0;
end;

// Code provided by Delphi to call the generated CORBA
// object factory to get an object reference for the
// server. Note parameters match what we defined in the
// CORBA object wizard. Since it's in the initialization
// section, the code will run whenever this unit is
// included in the uses section of another unit and the
// server will be started.
initialization
  TCorbaObjectFactory.Create('OnlineAuctionFactory',
    'OnlineAuction', 'IDL:CServer/OnlineAuctionFactory:1.0',
    IOnlineAuction, TOnlineAuction, iSingleInstance,
    tmSingleThread);
end.
```

End Listing Two

By Jeremy Merrill

Modifying VCL Behavior

A Practical Example Using Visual Components

To make a visual component behave differently from its defaults, we generally have to create a new component that descends from the original component's class. This article will show how to dynamically change the behavior of a native Delphi visual component without creating a new class.

```
unit LinkedLabel;

interface

uses
  Messages, Classes, Controls, StdCtrls;

type
  TLinkedLabel = class(TLabel)
  private
    // The associate control.
    FAssociate:  TControl;
    // Puts FAssociate into all caps mode.
    FCapsLock:   Boolean;
    // The distance between the label and the associate.
    FGap:         Integer;
    // True when the label is on top of the associate.
    FOnTop:       Boolean;
    // Saves the original value of FAssociate.WindowProc.
    FOldWinProc: TWndMethod;
    // Used to prevent infinite update loops.
    FUpdating:   Boolean;
  protected
    procedure Adjust(MoveLabel: Boolean);
    procedure SetGap(Value: Integer);
    procedure SetOnTop(Value: Boolean);
    procedure SetAssociate(Value: TControl);
    procedure NewWinProc(var Message: TMessage);
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
    procedure WndProc(var Message: TMessage); override;
  public
    constructor Create(AOwner :TComponent); override;
    destructor Destroy; override;
  published
    property Associate: TControl
      read FAssociate write SetAssociate;
    property CapsLock: Boolean
      read FCapsLock write FCapsLock;
    property Gap: Integer read FGap write SetGap default 8;
    property OnTop: Boolean read FOnTop write SetOnTop;
  end;
```

Figure 1: The *TLinkedLabel* class declaration.

How is this possible? The secret is to intercept the Windows messages being sent to the control. This can be accomplished by using a *TControl* property named *WindowProc*, which essentially points to a component's Windows message event handler.

To demonstrate this technique, we'll create a LinkedLabel component, which will link itself to any *TControl* and dynamically modify its behavior. *TLinkedLabel* will descend from *TLabel*, and will feature four additional published properties:

- *Associate* — the companion control whose behavior we'll be modifying.
- *CapsLock* — when this Boolean property is True, certain types of associate controls will process lower-case keystrokes as upper case. This doesn't work with all controls, because not all controls respond to the WM_CHAR message in the same way. Testing reveals that Edit, MaskEdit, Memo, and RichEdit controls all respond to the *CapsLock* property, while ComboBox does not. Obviously, *CapsLock* will have little or no effect on many other components, such as a Button or CheckBox control.
- *Gap* — the distance between the LinkedLabel and its associate control.
- *OnTop* — this Boolean property determines whether the LinkedLabel will appear to the left of, or on top of, the associate control.

In addition, *TLinkedLabel* will keep the *Enabled* and *Visible* properties of the LinkedLabel and its associate synchronized. It will also maintain a set distance and orientation from the associate control. This means that when you move the LinkedLabel, the associate moves with it, and vice versa.

Let's take a look at the *TLinkedLabel* class declaration, shown in Figure 1.

Now let's look at the different methods of this component in detail, starting with the constructor. Note that when creating a new object, all of its associated memory is cleared. This will automatically set *FAssociate* and *FOldWinProc* to **nil**, and *FCapsLock*, *FOnTop*, and *FUpdating* to False, all without having to explicitly initialize them in the constructor. Therefore, the only thing we need to set in the constructor is the default *Gap* value:

```
implementation

constructor TLinkedLabel.Create(AOwner: TComponent);
begin
  inherited;
  FGap := 8;
end;
```

Now we come to the *Adjust* method, which is responsible for positioning the LinkedLabel component or the associate control, depending on the value of the *MoveLabel* parameter. As you'll see in the code, the actual position of the LinkedLabel in relationship to the associate is based on the *Gap* and *OnTop* properties (see Figure 2). Although *OnTop* only provides us with two possible orientations, there are many other possibilities that could easily be programmed into this component. However, adding a lot of "bells and whistles" to *TLinkedLabel* is not the focus of this article, and has, therefore, been entrusted to the reader.

```
procedure TLinkedLabel.Adjust(MoveLabel: Boolean);
var
  dx, dy: Integer;
begin
  if (Assigned(FAssociate)) then begin
    if (FOnTop) then
      begin
        dx := 0;
        dy := Height + FGap;
      end
    else
      begin
        dx := Width + FGap;
        dy := (Height - FAssociate.Height) div 2;
      end;
    if (MoveLabel) then
      begin
        Left := FAssociate.Left - dx;
        Top  := FAssociate.Top - dy;
      end
    else
      begin
        FAssociate.Left := Left + dx;
        FAssociate.Top  := Top + dy;
      end;
  end;
end;
```

Figure 2: The *Adjust* method.

At this point, we come to the set methods of the *Gap* and *OnTop* properties (see Figure 3). These are needed so we can reposition the LinkedLabel when the *Gap* or *OnTop* values are modified.

```
procedure TLinkedLabel.SetGap(Value: Integer);
begin
  if (FGap <> Value) then
    begin
      FGap := Value;
      Adjust(True);
    end;
end;

procedure TLinkedLabel.SetOnTop(Value: Boolean);
begin
  if (FOnTop <> Value) then
    begin
      FOnTop := Value;
      Adjust(True);
    end;
end;
```

Figure 3: The set methods of the *Gap* and *OnTop* properties.

Now we come to the *SetAssociate* method (see Figure 4).

```
procedure TLinkedLabel.SetAssociate(Value: TControl);
begin
  if (Value <> FAssociate) then begin
    if (Assigned(FAssociate)) then
      FAssociate.WindowProc := FOldWinProc;
    FAssociate := Value;
    if (Assigned(Value)) then
      begin
        Adjust(True);
        Enabled := FAssociate.Enabled;
        Visible := FAssociate.Visible;
        FOldWinProc := FAssociate.WindowProc;
        FAssociate.WindowProc := NewWinProc;
      end;
  end;
end;
```

Figure 4: The *SetAssociate* method.

To understand it, we need to discuss the *WindowProc* property in more detail. *WindowProc* is defined as of type *TWndMethod*. *TWndMethod* can be found in the Controls unit with the following definition:

```
TWndMethod = procedure(var Message: TMessage) of object;
```

Notice that *FOldWinProc* is also defined as a *TWndMethod*, and that the *NewWinProc* method has the same parameter structure as *TWndMethod*. This allows us to point *FOldWinProc* to the current value of *WindowProc* and assign *WindowProc* to the *NewWinProc* method.

Why do we need to use *FOldWinProc* if *WindowProc* is just another event property? Because the difference between *WindowProc* and any other event property is that *WindowProc* is already pointing to an existing event handler. If we simply point *WindowProc* to our own method, the control will no longer be able to respond to any Windows messages. To solve this problem, we set *FOldWinProc* to the current value of *WindowProc* before pointing *WindowProc* to the *NewWinProc* method.

In *NewWinProc*, we call the old message handler, via *FOldWinProc*, after and acting upon specific Windows messages. Because we modify the *WindowProc* property on the associate control, it's important that we restore its former value before changing to a new associate component.

It's also important that we don't leave the associate's *WindowProc* property pointing to a routine that no longer exists. We therefore call *SetAssociate(**nil**)* in the destructor, which, as we've seen, will restore *WindowProc* to its original value:

```
destructor TLinkedLabel.Destroy;
begin
  SetAssociate(nil);
  inherited;
end;
```

```
procedure TLinkedLabel.NewWinProc(var Message: TMessage);
var
  Ch: Char;
begin
  if (Assigned(FAssociate) and (not FUpdating)) then begin
    FUpdating := True;
    try
      case(Message.Msg) of
        WM_CHAR:
          if (FCapsLock) then begin
            Ch := Char(TWMKey(Message).CharCode);
            if (Ch >= 'a') and (Ch <= 'z') then
              TWMKey(Message).CharCode := ord(UpCase(Ch));
          end;
        CM_ENABLEDCHANGED:
          Enabled := FAssociate.Enabled;
        CM_VISIBLECHANGED:
          Visible := FAssociate.Visible;
        WM_SIZE, WM_MOVE, WM_WINDOWPOSCHANGED:
          Adjust(True);
      end;
    finally
      FUpdating := False;
    end;
  end;
  FOldWinProc(Message);
end;
```

Figure 5: The *NewWinProc* method.

```
procedure TLinkedLabel.WndProc(var Message: TMessage);
begin
  if (Assigned(FAssociate) and (not FUpdating)) then begin
    FUpdating := True;
    try
      case(Message.Msg) of
        CM_ENABLEDCHANGED:   FAzssociate.Enabled :=Enabled;
        CM_VISIBLECHANGED:   FAssociate.Visible := Visible;
        WM_WINDOWPOSCHANGED: Adjust(False);
      end;
   finally
      FUpdating := False;
    end;
  end;
  inherited;
end;
```

Figure 6: Instead of tapping into the *WindowProc* property, we override the *WndProc* method.

In addition, we don't want to be pointing to an associate that no longer exists. By overriding the *Notification* method, we can know when the associate control is destroyed, and reset our pointer to the associate accordingly:

```
procedure TLinkedLabel.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  if ((Operation = opRemove) and
      (AComponent = FAssociate)) then
    SetAssociate(nil);
end;
```

Now we come to the *NewWinProc* method (see Figure 5). Here, we simply look for specific Windows messages being sent to the associate component. It's important to realize that although this method is only called by the associate control, it's actually part of the LinkedLabel, i.e. Self = LinkedLabel, not the associate control. This is identical to creating an *OnClick* event handler for a button. The *OnClick* event handler is created as part of the button's parent form and is not a new method extending the *TButton* class.

If you examine this routine, you'll see we make no attempt to process Windows messages. We react to specific messages, then let the associate process them normally by calling *FOldWinProc*. In the case of the WM_CHAR message, we change part of the message, causing the component to think an upper-case character was pressed.

Finally, we look at two different messages to see if the associate has been moved. This is because components that descend from *TWinControl* will get a WM_MOVE message when they're moved, and other visual components (such as a Label) will get the WM_WINDOWPOSCHANGED message. The WM_SIZE message is examined, because if the *OnTop* property is False, the position of the LinkedLabel will change based on the height of the component.

The last method of our component is where we make adjustments to the associate when the LinkedLabel is changed (see Figure 6). Rather than override existing methods of *TLabel* to do this, we employ the same technique we used to modify the associate's behavior. Notice that instead of tapping into the *WindowProc* property, we override the *WndProc* method. How is this the same technique? If you look at *TControl*'s constructor, you'll see that *WindowProc* is initialized to point at the *WndProc* method. So in essence, we are overriding the same method, but in a cleaner way, and without having to store the original value of *WindowProc*.

One final point should be made about the previous component. You'll notice the use of *FUpdating* in both *NewWinProc* and *WndProc*. This variable is used to alert the LinkedLabel and the associate that the other component is making a change. If you don't do this, it's easy to create an infinite updating loop, or get unexpected results. Here's one flow of events that demonstrates the need for the *FUpdating* variable:
- The user drags the LinkedLabel to a new position.
- *WndProc* receives a WM_WINDOWPOSCHANGED message, and fires *Adjust(False)* to move the associate.
- *Adjust* sets *FAssociate.Left* to the new value as part of repositioning the associate.
- *FAssociate* fires off a WM_MOVE message, indicating it has changed position.
- *NewWinProc* detects the WM_MOVE message and calls *Adjust(True)* in an attempt to move the LinkedLabel to match the associate's new position.

As you can see, we haven't even gotten a chance to change the associate's *Top* property to match the LinkedLabel's new position before the associate tries to move the LinkedLabel. By using the *FUpdating* variable, the associate will not notice the WM_MOVE message and won't try to call *Adjust* to reposition the LinkedLabel.

A Couple of Issues

There are a couple of problems with the *TLinkedLabel* component that I did not address in this article. The following are brief descriptions:
- You can cause all kinds of problems if you link two or more LinkedLabels to the same component, and then destroy one or more of them. You can end up breaking the link to other LinkedLabels, and even cause the linked component's *WindowProc* to point to a non-existent routine.
- If you link a LinkedLabel to a component on a different form, the *Notification* method won't be called when that component is destroyed. Calling *FreeNotification* when the component is linked will fix this, but that doesn't really address the problem. The real problem is that we allowed it

to be linked to the component on the other form in the first place. What we really want to do is restrict associates to only those components with the same parent as the LinkedLabel. Although it's not difficult to do this, it's a little tricky to only show eligible components in the Associate properties drop-down list in the Object Inspector.

Conclusion

That's about it. Replacing the *WindowProc* of an existing component does have its limitations, but can be a very useful technique. I can't think of any other reasonable way to design a component like *TLinkedLabel* and have the associate control move the LinkedLabel when the associate is moved. I'm not going to try and list other possible uses for this technique, because they are countless and limited only by a programmer's ingenuity. ∆

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\JAN\DI200002JM.

Jeremy Merrill is an EDS contractor in a partnership contract with the Veteran's Health Administration. He is a member of the VA's Computerized Patient Record System development team, located in the Salt Lake City Chief Information Officer's Field Office.

By Rick Spence

Visual Form Inheritance

Part I: An Introduction and Primer

If you use Visual Form Inheritance extensively, your development time will decrease, your applications will have fewer errors, and your UI will have a more consistent look and feel. Are these bold claims justifiable? This article shows how to get the most from a Delphi feature known as VFI for short. Read on and see for yourself.

What Is Visual Form Inheritance?

VFI refers to Delphi's ability to create new forms that either inherit, or are derived from, existing forms. VFI shares the advantages of inheritance in general, the primary one being the ability to capture the similarities of different classes in a hierarchy. In this case, the classes are forms generated by Delphi's Form designer. Put another way, VFI allows you to write code in one place that's common to multiple forms. Writing code just once obviously increases productivity and decreases errors. This applies to forms in one application and between applications.

There's more to VFI than code reuse, however. This is where the "Visual" part of VFI comes in. You can use the Delphi Form designer to lay out forms, and have that layout shared among forms. Having forms share a layout leads to a more consistent look and feel, and improves productivity.

Using VFI also allows you to design your forms in a hierarchy. All changes to forms higher in the hierarchy affect all inheriting forms. Likewise, you can add components and/or code to forms, thus changing all inheriting forms.

Assume you're creating an application to manage customers and their invoices. You determine you need three general maintenance forms: one for the customers, a second for their invoices, and a third for the parts you're selling. Although each form is displaying different information, certain elements of the forms are the same. Say, for example, that each form needs a page control with two tab sheets: one with a database grid, the other with table-specific edit controls. Each form also needs buttons to save and cancel changes, add and delete records, etc.

You could design each form separately, writing code for each. A better approach would be to start with a generic form that contains controls and code common to all the forms, then use VFI to create the forms based on the generic form, customizing each new form as required.

Once you've created these forms, you can still add functionality to the generic form. For example, you may want to add a background logo to each form or implement a generic search facility. All you need to do is add this in the generic form, and the specific forms automatically inherit this look and behavior.

Understanding how VFI works requires a working knowledge of a language's implementation of inheritance in general. A quick review is in order.

Inheritance in Brief

Inheritance is an object-oriented programming feature that allows you to create new classes based on existing classes. The existing class is called a superclass; the new class is called a subclass. Classes, of course, contain data and code. Subclasses can see the public code and data of their superclasses.

Delphi supports single inheritance, meaning classes directly inherit from only one class. (Some other languages support multiple inheritance, which allows one class to inherit from more than one superclass.) Delphi also supports repeated inheritance, which allows Class A to inherit from Class B, and Class B to inherit from Class C. In this case, Class A can see the public data and code of Class B and Class C (see Figure 1).

Consider the following sample class produced by Delphi's Form designer:

```
type
  TFrmRichEdit = class(TForm)
    btnCancel: TButton;
    btnOK: TButton;
    rtfNotes: TRichEdit;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

The class is named *TFrmRichEdit* and inherits from the *TForm* class. *TForm* is one of many classes declared in the VCL. *TForm* and its superclasses define data and code common to all types of forms. This is where you'll find the form's font, caption, and color. It's also where you'll find the *Close*, *Show*, and *ShowModal* methods. Thus, *TForm* and its ancestors capture the similarities of all forms. If Delphi didn't support inheritance, the Form designer would have to generate all these common methods and data in every form you create.

To instantiate the *TFrmRichEdit* class, you first need to declare an object of that class, which Delphi's Form designer does for you:

```
var
  frmRichEdit: TfrmRichEdit;
```

Then you can instantiate the class by calling its constructor, *Create*, passing the form's owner, *Application*:

```
frmRichEdit := TFrmRichEdit.Create(Application);
```

There's inheritance in action again. Who declares the *Create* method? One of *TFrmRichEdit*'s superclasses, i.e. *TCustomForm*.

Once you've created an object based on a subclass, you can directly access that object's data and methods:

```
frmRichEdit.btnOk.Enabled := False;
```

and the data and methods of its superclasses:

```
frmRichEdit.Caption := 'RTF Editor';
frmRichEdit.Show;
```

Caption is declared in *TControl* (several levels up in the hierarchy), and the *Show* method is declared in *TCustomForm*.

So far we've discussed the public data and code of a class. What do we mean by "public?" And are there other ways to declare data and code? Public refers to the visibility, or scope, of the declarations. A class' public data and code are visible to the user of the class. In the previous sample class, *btnCancel*, *btnOk*, and *rtfNotes* are all public instance variables.

There are two other ways to declare instance variables and code in a class: You can make them private or protected. Again, if you review the class declaration just shown, you will see two empty sections labeled **private** and **public**. These are sections where you can add your own code and data. Any additions you make to the **private** section of the class are only visible to methods of the class. Users of the class can't see the private data or methods.

Delphi's Form designer doesn't generate a **protected** section for you, but you can add one. Any declarations you make in the **protected** section are visible to methods of the class, just as the declarations in the **private** section. The difference between **private** and **protected** has to do with subclasses. Subclasses can see the **protected** declarations, but not the **private** ones.

With that quick review behind us, let's put VFI to use.

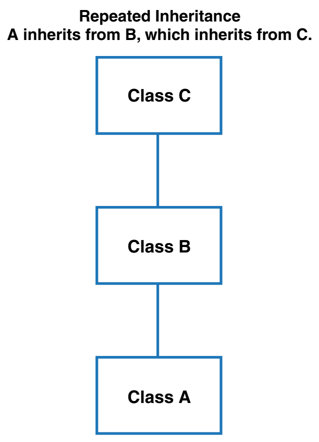


Figure 1: Repeated inheritance.

Mechanics of VFI

To create forms based on existing forms, you need to use Delphi's Object Repository. Using the Object Repository, you can create a form based on a form already in the Object Repository, or in your current project. As you'll see, you use the same IDE menu items.

To create a new form in your application, select **File | New Form**. To create a form based on an existing form, select **File | New**. This opens the Object Repository. Now you can choose what you want to create. You can add your own forms to the Object Repository, but there are some forms supplied with Delphi you can use as starting points. Select the Forms page, and you'll see forms labeled **Dual list box** and **About box** (see Figure 2). These forms are available for any application to use.

Notice the **Copy**, **Inherit**, and **Use** radio buttons at the bottom of the form. These tell Delphi how you want your new form to relate to the form in the Object Repository:

- **Copy** means you want to duplicate the form in the Object Repository in your own application. Your new form will be completely independent from the form in the Object Repository; if you subsequently change the form in the Object Repository, this won't change the new form.
- **Inherit** means the new form is based on the form in the Object Repository and retains its link to this form; if you change the form in the Object Repository, this will also change all forms inherited from this form. This is the most common VFI option.

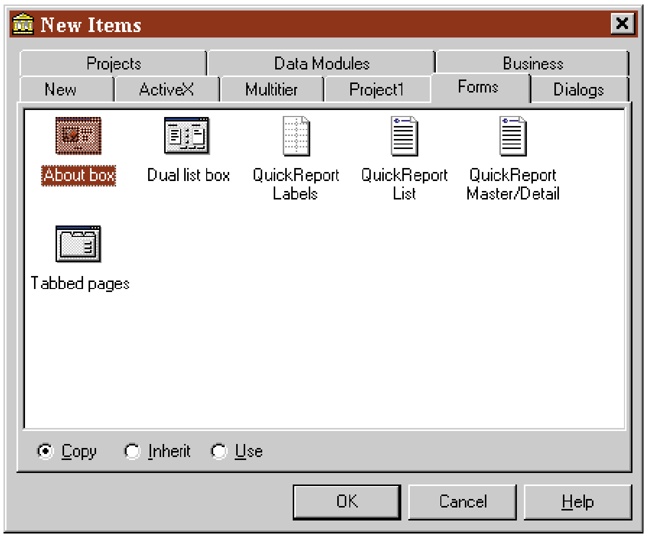


Figure 2: The Forms tab of the Delphi 4 Object Repository.

- **Use** means you want to directly use the form in the Object Repository. Any changes you make directly change the form in the Object Repository, thus changing forms inherited from this form.

Forms in your current application are also available for reuse. When you open the Object Repository, there should be a tab sheet whose caption is the name of your project. Select this and you'll see the forms in your current project. The only difference between working with forms in the Object Repository and working with forms in your application is that you can only inherit from forms in your existing application, i.e. only the **Inherit** radio button is available. It's the only option that makes sense.

Once you've created a new form that inherits from a form in the Object Repository, you can still change the new form. You can add new components and write new methods. You can also change things you inherit from the form in the Object Repository. You can move and resize inherited components, but you can't delete a component you inherited. Attempting to do so will result in a design-time error message. Although you can't delete an inherited component, you can achieve the same effect by making it invisible by setting its visible property to False.

Once you've changed a property of an inherited component, you've broken the link for that property between the subform and the superform. For example, assume you move a button to the left in a subform, then move the same button in the form in the Object Repository. Because the subform has broken the link to the *Left* property from the superform, the component in the subform doesn't move (it would if you hadn't broken the link). You can reestablish the link by right-clicking on the component, and selecting **Revert to inherited**. This undoes all changes to the subclass component.

When working with methods you inherit from a superclass, you have several options. You can either replace the superclass method altogether, or execute your own code in addition to the superclass method. Let's look at an example. On the Forms page of the New Items dialog box, double-click the **Dual list box** form with the **Inherited** radio button selected. Double-click on the > button; this button enables you to move a selected item from the left list box to the right list box. The IDE will generate the following method template for you:

```
procedure TDualListDlg2.IncludeBtnClick(Sender: TObject);
begin
  inherited;

end;
```

The keyword **inherited** means "Call a method with the same name in the superclass." The code in the superclass is what actually moves the selected items from the left list box to the right list box:

```
procedure TDualListDlg.IncludeBtnClick(Sender: TObject);
var
  Index: Integer;
begin
  Index := GetFirstSelection(SrcList);
  MoveSelected(SrcList, DstList.Items);
  SetItem(SrcList, Index);
end;
```

This code is only called because the subclass explicitly makes a call to it using the **inherited** keyword. You're free to place your additional code before or after the inherited call. You're also free to remove the **inherited** keyword, thus removing the call.

A Quick Example

Imagine you've created several forms inheriting from the **Dual list box** form in the Object Repository. You've customized the subforms with information specific to your application, and now you want to add drag-and-drop capabilities to each form; you want to allow your users to drag items from the left list box into the right list box. If you hadn't used VFI, you would have to implement the drag-and-drop capability in each subform. However, because you used VFI, you can add the code in one place: the form in the Object Repository. Here's a step-by-step approach:

1) Select the **Dual list box** form in the Object Repository, with the **Use** radio button selected.
2) Set the *DragMode* property of the left list box, *SrcList*, to *dmAutomatic*. This automatically enables dragging when the user holds the left mouse button down with an item selected. The alternative mode is *dmManual*, in which case your program must explicitly enable dragging in code.
3) Write the *onDragOver* event for the right list box, *DstList*, as:

   ```
   Accept := (source = srcList);
   ```

   The *onDragOver* event is fired when the user drags an item over the control. The event is used to determine whether the control is a valid destination for the dragged item. The code we wrote tells Delphi to accept the drop if the source of the drag is the *SrcList* list box.
4) Write the *onDragDrop* event of the *DstList* list box as:

   ```
   IncludeBtnClick(Sender);
   ```

   The *onDragDrop* event is fired when the user drops the item. This code simply fires the same event as if the user had clicked the > button, so there's no point in duplicating that code.

VFI under the Hood

One of my favorite things about Delphi is that nothing's hidden. Everything we've just done is implemented in source code. To see how VFI actually works, all you need to do is examine the code pro-

```
type
  TDualListDlg = class(TForm)
    OKBtn: TButton;
    CancelBtn: TButton;
    HelpBtn: TButton;
    SrcList: TListBox;
    DstList: TListBox;
    SrcLabel: TLabel;
    DstLabel: TLabel;
    IncludeBtn: TSpeedButton;
    IncAllBtn: TSpeedButton;
    ExcludeBtn: TSpeedButton;
    ExAllBtn: TSpeedButton;
    procedure IncludeBtnClick(Sender: TObject);
    procedure ExcludeBtnClick(Sender: TObject);
    procedure IncAllBtnClick(Sender: TObject);
    procedure ExcAllBtnClick(Sender: TObject);
    procedure MoveSelected(List: TCustomListBox;
      Items: TStrings);
    procedure SetItem(List: TListBox; Index: Integer);
    function GetFirstSelection(List: TCustomListBox):
      Integer;
    procedure SetButtons;
  private
    { Private declarations }
  public
    { Public declarations }
end;
```

Figure 3: Class declaration when you use the Dual list box from the Object Repository.

duced by the Form designer. Let's start by looking at the class declaration generated when you simply copy a form from the Object Repository into your application. We'll use the Dual list box as our example again. Figure 3 shows the class declaration.

And all the code is right there in the **implementation** section (Figure 3 doesn't show that). The important line is the one that declares the class:

```
TDualListDlg = class(TForm)
```

This class inherits directly from *TForm*. Everything about Dual list box is in this .PAS file, and its associated .DFM file.

Now consider Figure 4, which shows the entire Pascal file when you inherit from the form in the Object Repository.

Again, the line declaring the class is important:

```
TDualListDlg3 = class(TDualListDlg)
```

This shows that the new form simply inherits from another class, *TDualListDlg*. *TDualListDlg*, of course, is the form in the Object Repository. Note how the new form's **uses** clause includes that unit (DUALLIST). Also note that the project source code, the .DPR file, includes that unit, as well:

```
uses
  Forms,
  Unit1 in 'Unit1.pas' { Form1 },
  Unit2 in 'Unit2.pas' { Form2 },
  DualList in '..\OBJREPOS\DUALLIST.pas' { DualListDlg },
  Unit3 in 'Unit3.pas' { DualListDlg3 };
```

Because your new form inherits from another form, your project must include both forms.

Changes you make to your subform are simply made to the subform's .PAS and .DFM files. For example, if your subclass changes a component's position, that change is recorded in the subclass' .DFM file. In fact, all you will see in the subform's .DFM file are the subform properties that are different from its superclass. That's the best way to see exactly which properties were changed.

```
unit Unit3;

interface

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

type
  TDualListDlg3 = class(TDualListDlg)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  DualListDlg3: TDualListDlg3;

implementation

{$R *.DFM}

end.
```

Figure 4: Pascal file for the Dual list box inherited from the Object Repository.

The final thing we need to look at in this section is the structure of the Object Repository itself. You may be surprised to learn that the Delphi Object Repository is nothing more than an interface to an .INI file. The Delphi 4/5 Object Repository is stored in the file Delphi32.DRO in the \bin directory of your Delphi installation. Open it with Notepad or some other text editor to see for yourself. Figure 5 shows the portion concerning the Dual list box.

The important line is the section heading. As you can see, it tells Delphi where the code is located on the disk. There are similar entries for each form in the Object Repository.

VFI in the Real World

Technical articles often paint a rosy picture of development. Just follow these steps and you're in programming utopia. In the real world, VFI can save you time and make your programs more consistent. But it does take effort. The first time you use VFI, you'll spend a lot of time designing your form hierarchies. Here's how it usually goes the first time you use it.

You start developing the way you always did: You create forms and write code. Then you realize that several forms look pretty much the same, and you're writing similar code in each form. At this stage, you decide VFI will help, so you find the common layouts and code and move those into a generic form. You place that form in the Object Repository, then rework the child forms to inherit from the main form. But it doesn't end there. It's unrealistic to think you'll come up with a perfect design the first time. What usually happens is you continue developing the subforms, continually looking for things you can move up the hierarchy to avoid duplication. I consider this class design a very similar process to database normalization: You're trying to avoid redundancy and duplication, and it takes time.

This type of design work is an iterative process. You might not see productivity benefits the first time you try it. Then again, you might. An example springs to mind from one of my own development projects. We designed a standard interface for editing database tables. We provided all the usual features — add, edit, delete, search for records, etc. — and allowed the user to access these activities from the menu, as well as from the buttons located in a Coolbar. All the common code and layout was stored in a common superclass (which we use in all our applications), and the actual editing forms were derived from this common form. The user didn't care for the Coolbar, and wanted it changed to a simpler interface. We had almost 30 forms developed this way, but all we had to do was replace the Coolbar with a Toolbar in one form and recompile.

```
[C:\PROGRAM FILES\BORLAND\DELPHI4\OBJREPOS\DUALLIST]
Type=FormTemplate
Name=Dual list box
Page=Forms
Icon=C:\PROGRAM FILES\BORLAND\DELPHI4\OBJREPOS\DUALLIST.ICO
Description=Dialog box with two list boxes. Supports moving
items from one list to the other.
Author=Borland
DefaultMainForm=0
DefaultNewForm=0
Ancestor=
```

Figure 5: Portion of Delphi Object Repository concerned with the Dual list box.

The real benefits of VFI, though, materialize the second or third time you use a design. Remember that you can use the Object Repository to share forms between applications. In our shop, the first stage of our design is to list forms required for an application, and note their similarities. We then decide whether to use existing templates we've already developed, modify those slightly, or develop new ones. Whichever way we go, we rarely start from scratch.

If you want to get started using VFI, I'd suggest you start with a database maintenance form. Decide how you want the form to look, design it once, and place it in the Object Repository. Place as much common code in this generic form as possible, then create actual forms containing data specific to the actual tables, inheriting from this generic form. As you write code in the actual forms, consider whether that code could be moved up the hierarchy. For each piece of code you write, ask yourself whether that code is specific to the data in this form, or whether it's generic and other forms could use it. If it's generic, move it up the hierarchy.

Conclusion

VFI allows you to visually create new forms based on existing forms, and provides the same benefits as inheritance in general. This, in turn, leads to faster development time and more consistent and reliable applications.

Understanding how VFI actually works requires a good understanding of Delphi's object model and inheritance in particular. As you saw, VFI is nothing more than a language mechanism (inheritance), some IDE features, and an .INI file! I consider it one of the most powerful features of Delphi, and use it extensively. Next month, I'll present a generic database maintenance form I use in development, and show how this substantially reduces my own development time. ∆

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.

By T. Wesley Erickson

Time Travels

Of Time Zones, Daylight Savings, and other Delights

If your application uses dates and times, what will happen when it's deployed to users in different time zones? Have you taken the effects of Daylight Savings Time into consideration?

We know that many locations do not recognize Daylight Savings Time; but what about locations in the Southern Hemisphere, such as Brazil and portions of Australia, where its implementation is the opposite of that in the Northern Hemisphere? Your particular application may not be affected, but if you need to convert between local time and Universal Coordinated Time (UTC) for any reason, you should consider the effects that time-zone changes will have on your program.

Who cares about changes in time zones? Your users might. Consider the relatively trivial example of a telephone dialer: Wouldn't it be nice if users were notified of the local time when calling a phone number outside of the local calling area? This would require a lookup table of area codes to time zones, but it would certainly be a reasonable enhancement to a dialing program. Some types of programs are vitally concerned with time-zone changes, particularly technical programs, or those relating to navigation and astronomy, to name a few.

You could ask your users to specify time-zone settings when installing your product, but they already specified their date, time, and time zone when they set up their computers. Besides, mobile computing is so pervasive, a user might easily work in multiple time zones in a single day. It seems intrusive to ask the user for information already in the registry. At the same time, it's your responsibility to confirm that their settings make sense and to offer alternatives if necessary.

A computer running Windows 95/98/NT maintains its internal time as Universal Coordinated Time, and displays the local time based on the user's time-zone setting, and the current state of Daylight Savings Time.

If all your application needs is the current UTC or local time, we could simply call the Win32 API procedures *GetSystemTime* or *GetLocalTime*. These functions return a data structure of type *SystemTime*:

```
type
  SystemTime = record
    wYear: Word;
    wMonth: Word;
    wDayOfWeek: Word;
    wDay: Word;
    wHour: Word;
    wMinute: Word;
    wSecond: Word;
    wMilliseconds: Word;
  end;
```

Most of the elements of the *SystemTime* record are self-explanatory; *wDayOfWeek* is an unsigned 16-bit integer (i.e. Word) value in the range 0-6, which identifies the day of the week, corresponding to Sunday through Saturday.

The elements of *SystemTime* can be used to assign a Delphi *DateTime* variable:

```
var
  ST : SystemTime;
  DT : TDateTime;
begin
  // Get Universal Coordinated Time (UTC).
  ST := GetSystemTime(ST);
  with ST do
    DT := EncodeDate(wYear, wMonth, wDay) +
      EncodeTime(wHour, wMinute, wSecond,
                 wMilliseconds);
end;
```

Note that *SysUtils.Now* is implemented in exactly this fashion, using *GetLocalTime* (*SysUtils.Now* replaces *GetCurrentTime*, which is obsolete).

Similarly, to determine if Standard or Daylight Time is in effect, we can call the Win32 API function *GetTimeZoneInformation*:

```
var
  Error  : Double;
  TZInfo : TTimeZoneInformation;
begin
  Error := GetTimeZoneInformation(TZInfo);
  case Error of
    0 : { Unknown } ;
    1 : { Standard Time } ;
    2 : { Daylight Time } ;
  end;
end;
```

Time Zone Information

GetTimeZoneInformation also returns a data structure (a record) that contains the current time-zone settings, and the information needed to convert between local and UTC times:

```
type
  TTIMEZONEINFORMATION = record
    Bias: Longint;
    StandardName: array[0..31] of WCHAR;
    StandardDate: TSystemTime;
    StandardBias: Longint;
    DaylightName: array[0..31] of WCHAR;
    DaylightDate: TSystemTime;
    DaylightBias: Longint;
  end;
```

Elements of *TTimeZoneInformation* are shown in Figure 1.

The dates and times represented by *StandardDate* and *DaylightDate* are implemented as a set. It's not permissible to have only one or the other specified; either both dates are specified (meaning Daylight Savings Time is implemented), or both are unspecified, in which case *wMonth* must be set to zero as a flag. Dates may be stored in either of two formats:

- **Absolute.** *wYear*, *wMonth*, *wDay*, *wHour*, *wMinute*, *wSecond*, and *wMilliseconds* are combined to refer to a specific date and time.
- **Relative.** Also called "day-in-month" format, refers to a particular occurrence of a day of the week, e.g. the last Sunday of the month.

Bias	Difference in minutes between local time and UTC, based on the formula UTC = local time + *Bias*.
StandardName	Null-terminated string identifying Standard Time, e.g. Pacific Standard Time. May be empty, or set to user's preference with *SetTimeZoneInformation*.
StandardDate	Record of type *SystemTime* that specifies the date and time when Standard Time begins.
StandardBias	Minutes added to *Bias* during Standard Time (normally zero).
DaylightName	Null-terminated string identifying Daylight Time, e.g. Pacific Daylight Time. May be empty, or set to user's preference with *SetTimeZoneInformation*.
DaylightDate	Record of type *SystemTime* that specifies the date and time when Daylight Time begins.
DaylightBias	Minutes added to *Bias* during Daylight Time (normally 60).

Figure 1: Elements of *TTimeZoneInformation*.

Relative dates are, by far, the more common, and are implemented as shown here:

- *wYear* must be set to zero as a flag.
- *wMonth* identifies the month in which the change occurs.
- *wDayOfWeek* identifies the day of the week (0-6 corresponding to Sunday-Saturday).
- *wDay* identifies which occurrence of *wDayOfWeek* (1-5 where 1 is the first occurrence, and 5 means "last *wDayOfWeek* in the month").

The resulting Relative date is combined with the encoded time from the *SystemTime* record to specify the exact date and time when the change occurs.

Using this information, we can create a function that will tell us whether Daylight time is in effect for a given date and time. We'll assume for this discussion that *TZInfo* is a global variable of type *TTimeZoneInformation* that was returned by an earlier call to *GetTimeZoneInformation*, perhaps during *FormCreate*, as shown in Listing One (beginning on page 25).

Converting between Local Time and Universal Time

We now have enough information to convert local times to UTC, and vice versa. On Windows NT, we might use *SystemTimeToTzSpecificLocalTime*, which converts UTC to local time in a specific time zone, but this function isn't available to users of Windows 95. We'll continue to assume that *TZInfo* is a global variable of type *TTimeZoneInformation* returned by an earlier call to *GetTimeZoneInformation*, as shown in Listing Two (on page 26).

We have one more step to make the program complete. Our program can decipher time-zone information and convert local times to UTC and back. How do we handle a situation where the time zone changes while our program is running? The answer is found in the Borland FAQ database (FAQ2020D). First, add the following code to the **private** declaration section of your application:

TIMEZONE.EXE

TIMEZONE.EXE is an example application designed to demonstrate using Delphi to access time-zone information, to convert between local time and UTC, and to respond to system-wide changes in the time-zone setting (see Figure A).

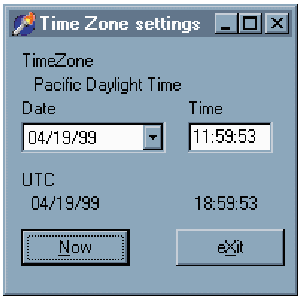


Figure A: The Time Zone application.

The program starts up initialized to the current local date and time, and lists the current time-zone setting (top), as well as the UTC and date that correspond to the local time and date in the edit controls.

To keep the program simple, user interaction is limited to two edit controls and two buttons. The user may select a date using the DateTimePicker. The user may enter a time in the **Time** edit control. If the string entered by the user fails to convert to a valid time, the exception handler clears the control and sets the time to midnight.

The display is updated when any of the following actions occur:

- The user clicks one of the controls.
- The user selects a date, and the pop-up calendar closes.
- The user tabs from one field to another.
- The user presses Enter↵ after making an entry in the **Time** edit control.

Two buttons are provided: **Now** sets the date and time to the computer's current date and time; **Exit** terminates the program.

— T. Wesley Erickson

```
private
  procedure WMTIMECHANGE(var Message: TWMTIMECHANGE);
    message WM_TIMECHANGE;
```

Then, add this *wmTimeChange* procedure to the **implementation** section:

```
procedure TFormName.wmTimeChange(
  var Message: TWMTIMECHANGE);
begin
  // Get new Time Zone information.
  Error := GetTimeZoneInformation(TZInfo);
  // Use value returned by GetTimeZoneInformation
  // for error trapping.
  case Error of
    0 : { Unknown } ;
    1 : { Standard Time } ;
    2 : { Daylight Time } ;
  end;
  // Update the form using new date/time settings...
end;
```

The *wmTimeChange* procedure can perform whatever action is necessary to update your application with the newly changed time zone.

Conclusion

Hopefully, you will find this little foray into the Win32 API to be time well spent. Your application is now ready for prime time. Feel free to check out my Time Zone application discussed in the sidebar TIMEZONE.EXE. ∆

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\JAN\DI200002TE.

T. Wesley Erickson is a Fire Suppression Captain with the Oceanside Fire Department in Southern California. A firefighter for over 23 years, Wes holds a BS in Computer Science and a teaching credential. His first involvement with computers was in 1966, participating in a program at the Data Processing Installation at MCB Camp Pendleton while he was in high school. Wes learned more than he ever wanted to know about time zones while writing a lunar co-longitude program, named the winner of the 1998 Computing Challenge by the Computing Section of the Association of Lunar and Planetary Observers, a world-wide organization of amateur astronomers. When not computing or star-gazing, he can be found sailing, cycling, enjoying the company of his wife Mary, or playing with grandson Jordan. You may contact Wes at twesley@compuserve.com, or visit his Web site at http://home.pacbell.net/twerick.

Begin Listing One

```
function DaylightSavings(DT: TDateTime): Boolean;
var
  D, M, Y, WeekNo : Word;
  DTBegins, STBegins : TDateTime;
begin
  // Get Year/Month/Day of DateTime passed as parameter.
  DecodeDate(DT,Y,M,D);
  // If TZInfo.DaylightDate.wMonth is zero,
  // Daylight Time not implemented.
  if (TZInfo.DaylightDate.wMonth = 0) then
    Result := False
  else  // Daylight Time is implemented.
    begin
      // If wYear is zero, use relative SystemTime format.
      if (TZInfo.StandardDate.wYear = 0) then
        // Relative SystemTime format.
        // Calculate DateTime Daylight Time begins using
        // relative format. wDay defines which wDayOfWeek
        // is used for time change: wDay of 1 identifies
        // the first occurrence of wDayOfWeek in the month;
        // 2..4 identify the second through fourth
        // occurrence. A value of 5 identifies the last
        // occurrence in the month.
        begin
          // Start at beginning of Daylight month.
          DTBegins :=
            EncodeDate(Y, TZInfo.DaylightDate.wMonth, 1);
          case TZInfo.DaylightDate.wDay of
            1, 2, 3, 4 :
              begin
                // Get to first occurrence of wDayOfWeek.
                // Key point: SysUtils.DayOfWeek is
                // unary-based; TZInfo.Daylight.wDay is
                // zero-based.
                while (SysUtils.DayOfWeek(DTBegins) - 1) <>
                      TZInfo.DaylightDate.wDayOfWeek do
                  DTBegins := DTBegins + 1;
                WeekNo := 1;
                if TZInfo.DaylightDate.wDay <> 1 then
                  repeat
                    DTBegins := DTBegins + 7;
                    Inc(WeekNo);
                  until WeekNo = TZInfo.DaylightDate.wDay;
                // Encode time Daylight Time begins.
                with TZInfo.DaylightDate do
                  DTBegins := DTBegins + EncodeTime(
                                wHour, wMinute, 0, 0);
              end;
            5 :
              begin
                // Count down from end of month to day of
                // week. Recall that we set DTBegins to the
                // first day of the month; go to the first
                // day of the next month and decrement.
                DTBegins := IncMonth(DTBegins, 1);
                DTBegins := DTBegins - 1;
                // Find the last occurrence of
                // the day of the week.
                while SysUtils.DayOfWeek(DTBegins) - 1 <>
                       TZInfo.DaylightDate.wDayOfWeek do
                  DTBegins := DTBegins -1;
                // Encode time Daylight Time begins.
                with TZInfo.DaylightDate do
                  DTBegins := DTBegins + EncodeTime(
                                wHour, wMinute, 0, 0);
              end;
          end;  // case.
          // Calculate DateTime Standard Time begins using
          // relative format. Start at beginning of
          // Standard month.
          STBegins :=
            EncodeDate(Y, TZInfo.StandardDate.wMonth, 1);
          case TZInfo.StandardDate.wDay of
            1, 2, 3, 4 :
              begin
                while (SysUtils.DayOfWeek(STBegins) - 1) <>
                      TZInfo.StandardDate.wDayOfWeek do
                  STBegins := STBegins + 1;
                WeekNo := 1;
                if TZInfo.StandardDate.wDay <> 1 then
                  repeat
                    STBegins := STBegins + 7;
                    Inc(WeekNo);
                  until (WeekNo = TZInfo.StandardDate.wDay);
                // Encode time Standard Time begins.
                with TZInfo.StandardDate do
                  STBegins := STBegins + EncodeTime(
                               wHour, wMinute, 0, 0);
                end;
            5 :
              begin
                // Count down from end of month to day of
                // week. Recall we set DTBegins to first
                // day of the month; go to the first day of
                // the next month and decrement.
                STBegins := IncMonth(STBegins, 1);
                STBegins := STBegins - 1;
                // Find last occurrence of day of the week.
                while SysUtils.DayOfWeek(STBegins) - 1 <>
                       TZInfo.StandardDate.wDayOfWeek do
                  STBegins := STBegins -1;
                // Encode time at which Standard Time begins.
                with TZInfo.StandardDate do
                  STBegins := STBegins + EncodeTime(
                               wHour, wMinute, 0, 0);
              end;
          end;  // case.
        end
      else
        begin  // Absolute SystemTime format.
          with TZInfo.DaylightDate do begin
            DTBegins := EncodeDate(wYear, wMonth, wDay) +
                        EncodeTime(wHour, wMinute, 0, 0);
          end;
          with TZInfo.StandardDate do begin
            STBegins := EncodeDate(wYear, wMonth, wDay) +
                        EncodeTime(wHour, wMinute, 0, 0);
          end;
        end;
      // Finally! How does DT compare to DTBegins and
      // STBegins?
      if (TZInfo.DaylightDate.wMonth <
         TZInfo.StandardDate.wMonth) then
        // For Northern Hemisphere...
        Result := (DT >= DTBegins) and (DT < STBegins)
      else
        // For Southern Hemisphere...
        Result := (DT < STBegins) or (DT >= DTBegins);
    end;
end;
```

End Listing One

Begin Listing Two

```
function LocalTimeToUniversal(LT: TDateTime): TDateTime;
  var UT: TDateTime; TZOffset: Integer;
  // Offset in minutes.
begin
  // Initialize UT to something,
  // so compiler doesn't complain.
  UT := LT;
  // Determine offset in effect for DateTime LT.
  if DaylightSavings(LT) then
    TZOffset := TZInfo.Bias + TZInfo.DaylightBias
  else
    TZOffset := TZInfo.Bias + TZInfo.StandardBias;
  // Apply offset.
  if (TZOffset > 0) then
    // Time zones west of Greenwich.
    UT := LT + EncodeTime(TZOffset div 60,
                          TZOffset mod 60, 0, 0)
  else
    if (TZOffset = 0) then
      // Time Zone = Greenwich.
      UT := LT
    else
      if (TZOffset < 0) then
        // Time zones east of Greenwich.
        UT := LT - EncodeTime(Abs(TZOffset) div 60,
                Abs(TZOffset) mod 60, 0, 0);
  // Return Universal Time.
  Result := UT;
end;

function UniversalTimeToLocal(UT: TDateTime): TDateTime;
  var LT: TDateTime; TZOffset: Integer;
begin
  LT := UT;
  // Determine offset in effect for DateTime UT.
  if DaylightSavings(UT) then
    TZOffset := TZInfo.Bias + TZInfo.DaylightBias
  else
    TZOffset := TZInfo.Bias + TZInfo.StandardBias;
  // Apply offset.
  if (TZOffset > 0) then
    // Time zones west of Greenwich.
    LT := UT - EncodeTime(TZOffset div 60,
                          TZOffset mod 60, 0, 0)
  else
    if (TZOffset = 0) then
      // Time Zone = Greenwich.
      LT := UT
    else
      if (TZOffset < 0) then
        // Time zones east of Greenwich.
        LT := UT + EncodeTime(Abs(TZOffset) div 60,
                              Abs(TZOffset) mod 60, 0, 0);
  // Return Local Time.
  Result := LT;
end;
```

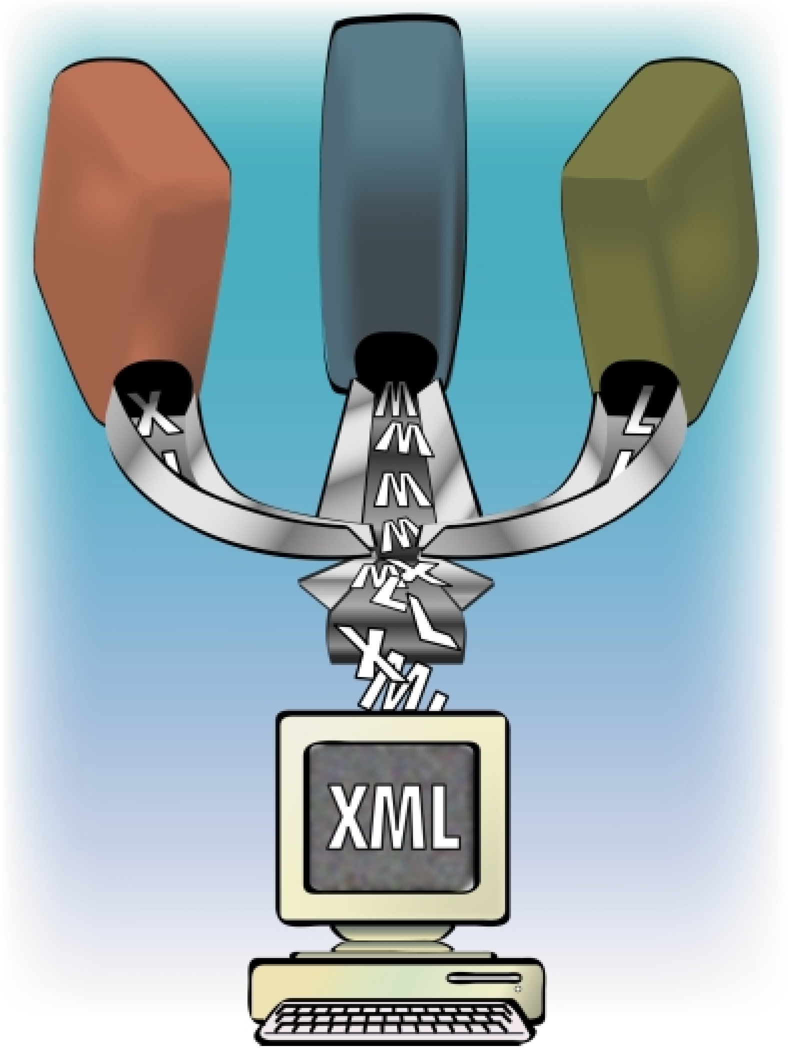
End Listing Two

By Keith Wood

Generating XML

And Delivering It Across the Web

In previous articles, we were introduced to XML (Extensible Markup Language), and we've seen how to process XML documents in a generic and reusable manner. This article looks at generating the XML itself, and delivering XML documents across the Internet.



XML as Data

XML documents encode the structure and content for domain-specific data. Their hierarchy of tags provides a format that's easy to manipulate programmatically, while remaining legible to humans. XML provides a mechanism to transfer data between applications that is independent of their underlying programming language and operating system.

Because XML documents contain data, it seems natural to want to generate them from existing data sources, i.e. databases. In general, each record from a table becomes an element in the XML document. Within this element appear sub-elements for each of the record's fields. Fields that form the primary key of the record should be specified as the ID attribute of the record element. We place an element identifying each table around these record elements. Around these, in turn, we have the document element that corresponds to the database.

Records from dependent tables appear as elements within their parent record. The linking fields need not appear as field elements nor as IDREF attributes, because the position of the record element within the parent adequately describes the relationship.

Across the Internet

When we want to generate HTML dynamically from a Delphi application, we can use the *TWebModule* object (created using the Web Application Wizard), and the *TPageProducer* object (from the Internet tab on the Component palette). *TWebModule* objects handle the intricacies of dealing with the various server protocols (ISAPI, NSAPI, CGI, Win-CGI). Indeed, they’re set up so we can write a program for one protocol, then easily convert it for use with another.

A *TPageProducer* object allows us to provide an HTML template — embedded as text or from an external file — that is output on request. Within that template, special tags are intercepted by the component and presented to us for replacement. These tags are embedded in angled brackets as usual, and start with a pound sign followed by the name of the tag. For example:

```
<#movies>
```

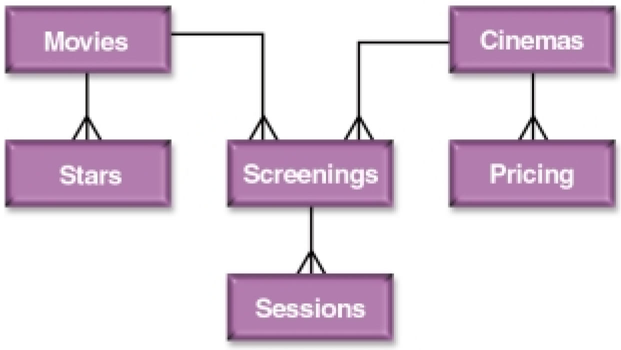
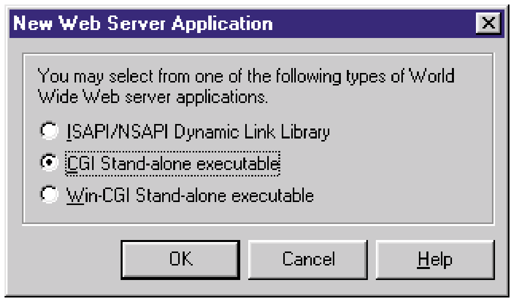


Figure 1: The example database.

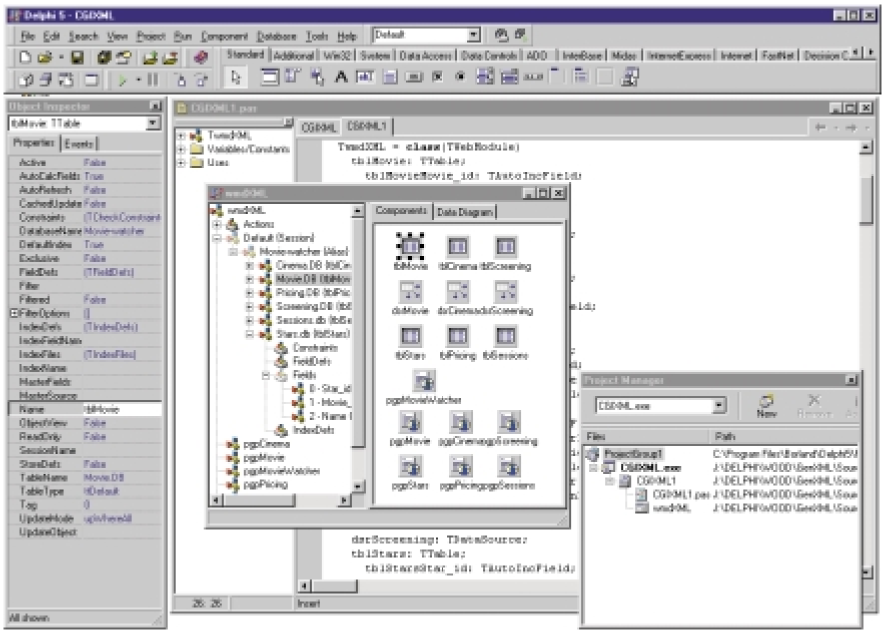


New Web Server Application

You may select from one of the following types of World Wide Web server applications.

( ) ISAPI/NSAPI Dynamic Link Library
[✓ CGI Stand-alone executable]
( ) Win-CGI Stand-alone executable

Figure 2: The Web Server Application wizard.



```
TwmdXML = class(TWebModule)
  tblMovie: TTable;
    tblMovieMovie_id: TAutoIncField;
```

```
dsrScreening: TDataSource;
tblStars: TTable;
  tblStarsStar_id: TAutoIncField;
```

Figure 3: Putting the example application together.

They may optionally have attributes specified in the usual HTML style. The *OnHTMLTag* event is triggered for each of these tags found, providing us with the type of the tag, its name, and any attributes. We respond by using this information to supply the text that replaces the tag in the output document. Enhancements to the basic PageProducer provide for the generation of HTML tables directly from a query or other data set.

This process can just as easily be used to generate XML instead of HTML. We simply replace the HTML snippets in the PageProducer objects with XML snippets, using the replacement tag mechanism to substitute the values from the database.

Watching Movies

To help demonstrate the techniques described in this article, we’ll use a database that contains information about local movies, and where and when they’re showing. (The example projects for this article are available for download; see end of article for details.) The overall structure of the database is shown in Figure 1. For this demonstration, the tables have been set up as Paradox files.

The Movies table holds the name, rating, length, director, and synopsis of each film, with the stars appearing in the related Stars table. Similarly, the Cinemas table holds the name, phone number, address, directions, and facilities for each theater. Prices for the different session times are described in the Pricing table attached to the Cinemas table. In the Screenings table, a movie and a cinema are brought together, detailing show dates, and any restrictions or theater features. The Sessions table holds the actual times for each showing, along with a reference to the pricing scheme that applies.

Generation

To create the XML dynamically, we start with a new application using the Web Server Application wizard (see Figure 2). The wizard is accessed by selecting Web Server Application from the New page of the New Items dialog box (select **File | New** from Delphi’s main menu).

We’ll select the CGI option, although there is no reason not to use one of the other types. Into this Web module we place tables and data sources corresponding to the database described previously, link them appropriately, and activate them (see Figure 3).

Next, we add several PageProducer components: one for the overall document, and one for each of the tables. Our main PageProducer contains the XML prolog and highest-level tags — those corresponding to the database and main tables (see Figure 4).

Within the PageProducer objects for each table is an XML document fragment describing the record structure. The entire snippet is enclosed in a tag indicating the type of record. Following this are the individual fields and an enclosing tag for any dependent tables. Using the field names as the substitution tag names makes the processing simpler during replacement. See the fragment for the movie element in Figure 5.

The event handler for the main PageProducer's *OnHTMLTag* event replaces the contents of the table tags with XML representing the records. For each one, it must step through all the records in that table and apply the appropriate template from another PageProducer, over and over. The *GetRecords* method performs this task within the application, taking the table and PageProducer as parameters. Event handlers for the PageProducer objects associated with each table simply use the tag name to find a field value from the table, or invoke the *GetRecords* method for dependent tables (see Figure 6).

```
<?xml version="1.0" standalone="yes"?>
<!--DOCTYPE movie-watcher SYSTEM "/movie-watcher.dtd"-->
<?xml:stylesheet type="text/xsl" href="/mw3.xsl"?>
<movie-watcher>
<movies>
<#movies></movies>
<cinemas>
<#cinemas></cinemas>
<screenings>
<#screenings></screenings>
</movie-watcher>
```

Figure 4: Main document outline for XML.

```
<movie id="<#movie_id>"<#logo_url><#url>>
  <name><#name></name>
  <rating><#rating></rating>
  <length><#length></length>
  <director><#director></director>
  <starring>
<#stars>  </starring>
  <synopsis><#synopsis></synopsis>
</movie>
```

Figure 5: Document fragment for the movie element.

```
// Cycle through all the records in the table
// and generate the XML snippet.
function TwmdXML.GetRecords(tbl: TTable; pgp:
  TPageProducer): string;
begin
  Result := '';
  with tbl do begin
    First;
    while not EOF do begin
      Result := Result + pgp.Content;
      Next;
    end;
  end;
end;

// Generate movie-watcher XML document.
procedure TwmdXML.pgpMovieWatcherHTMLTag(Sender: TObject;
  Tag: TTag; const TagString: string; TagParams: TStrings;
  var ReplaceText: string);
begin
  if TagString = 'movies' then
    ReplaceText := GetRecords(tblMovie, pgpMovie)
  else if TagString = 'cinemas' then
    ReplaceText := GetRecords(tblCinema, pgpCinema)
  else if TagString = 'screenings' then
    ReplaceText := GetRecords(tblScreening, pgpScreening);
end;

// Add details for a movie.
procedure TwmdXML.pgpMovieHTMLTag(Sender: TObject;
  Tag: TTag; const TagString: string; TagParams: TStrings;
  var ReplaceText: string);
begin
  if TagString = 'stars' then
    ReplaceText := GetRecords(tblStars, pgpStars)
  else
    ReplaceText :=
      tblMovie.FieldByName(TagString).DisplayText;
end;
```

Figure 6: Handling tags for the main document and movie element.

Formatting for individual fields is handled through the normal Delphi mechanisms: the *DisplayFormat* property, or the *OnGetText* event on the field itself. Some special formatting is performed using the latter (see Figure 7).

The autoincrementing identifiers for each table are made unique by prefixing them with a character corresponding to the table name (document-wide uniqueness is required of XML IDs). Some fields are presented as attributes and aren't included in the document if their values are blank. Boolean fields are indicated by the presence or absence of an empty tag. Finally, the memo fields must provide their actual value rather than their type.

We generate the entire XML document by creating a default Web action for the module. Within this we set the content type to text/xml and invoke the main PageProducer to create the actual content. Finally, we indicate that we've supplied the Web response by setting the *Handled* parameter to True. All this appears in the demonstration project CGIXML.dpr.

Once the application is compiled and placed in our Web server's CGI directory, we can call it up and view the results (see Figure 8). To do this, we'll need to have Internet Explorer version 5 installed, as it's currently the only browser to support XML. Also, we need to place the HTML style sheet for this document, mw3.xsl, into the server's normal document area so it can be retrieved.

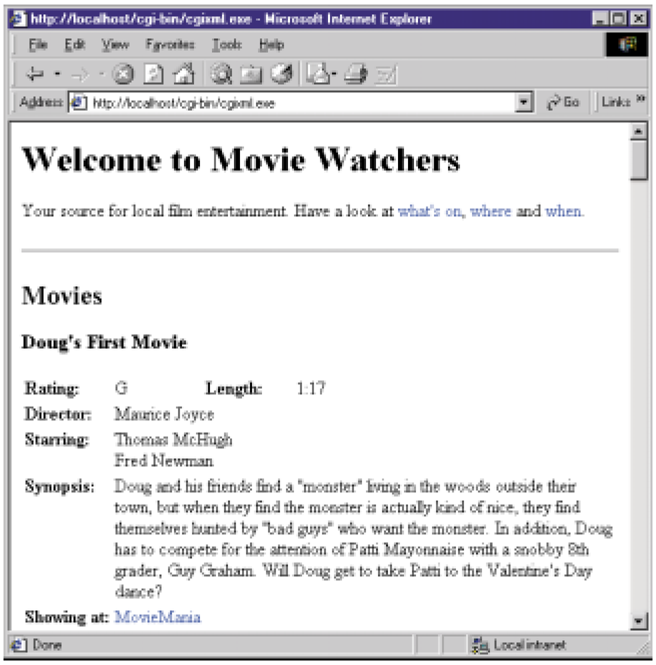
TRecordPageProducer

In the first example, we manually cycle through all the records in

```
// Make id unique.
procedure TwmdXML.IDGetText(Sender: TField;
  var Text: string; DisplayText: Boolean);
begin
  Text := Copy(Sender.FieldName, 1, 1) + Sender.AsString;
end;

// Include attributes only if present.
procedure TwmdXML.AttributeGetText(Sender: TField;
  var Text: string; DisplayText: Boolean);
begin
  if Sender.AsString <> '' then
    Text := ' ' + ModifyName(Sender.FieldName) +
            '="' + Sender.AsString + '"';
end;

// Include empty field tag only if flag in DB set.
procedure TwmdXML.EmptyFieldGetText(Sender: TField;
  var Text: string; DisplayText: Boolean);
begin
  if Sender.AsBoolean then
    Text := '<' + ModifyName(Sender.FieldName) + '/>';
end;

// Display longer text.
procedure TwmdXML.MemoGetText(Sender: TField;
  var Text: string; DisplayText: Boolean);
begin
  Text := Sender.AsString;
end;
```

Figure 7: Specialized formatting for various fields.

a table to generate the section of the XML document that's derived from it. As we've seen, this process was repeated several times on the different tables, all of which are doing basically the same thing. In true Delphi tradition, we now capture that process within a component, making it available for future use with minimum effort.


http://localhost/cgi-bin/cgixml.exe - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/cgi-bin/cgixml.exe
Go
Links
Done
Local intranet

Welcome to Movie Watchers

Your source for local film entertainment. Have a look at what's on, where and when.

Movies

Doug's First Movie

Rating: G Length: 1:17
Director: Maurice Joyce
Starring: Thomas McHugh
Fred Newman
Synopsis: Doug and his friends find a "monster" living in the woods outside their town, but when they find the monster is actually kind of nice, they find themselves hunted by "bad guys" who want the monster. In addition, Doug has to compete for the attention of Patti Mayonnaise with a snobby 8th grader, Guy Graham. Will Doug get to take Patti to the Valentine's Day dance?
Showing at: MovieMania

Figure 8: The formatted document in the browser.

```
// Iterate through the records in the dataset.
function TRecordPageProducer.ContentFromStream(
  Stream: TStream): string;
var
  stmNoRecs: TStream;
begin
  Result := '';
  if Assigned(FDataSet) then
    if FDataSet.Active then
      if FDataSet.RecordCount > 0 then
        // Cycle through all the records.
        with FDataSet do begin
          First;
          while not EOF do begin
            Stream.Position := 0;
            Result :=
              Result + inherited ContentFromStream(Stream);
            Next;
          end;
          Exit;
        end;
  // No data found.
  if FNoRecsFile <> '' then
    stmNoRecs := TFileStream.Create(FNoRecsFile,
                   fmOpenRead + fmShareDenyWrite)
  else
    stmNoRecs := TStringStream.Create(FNoRecsDoc.Text);
  if Assigned(stmNoRecs) then
    try
      Result := inherited ContentFromStream(stmNoRecs);
    finally
      stmNoRecs.Free;
    end;
end;
```

Figure 9: Generating a document snippet for each record.

The PageProducer component (from the Component palette's Internet tab) allows us to generate a section of a document from a template. The QueryTableProducer and DataSetTableProducer components transform the contents of a query, or any data set respectively, into an HTML table for inclusion in a document. What we want is somewhere in between: to be able to process each record in a data set, but without the hard-coded HTML output.

To achieve this, we create our own class, *TRecordPageProducer*, which generates its section of the document for each record in the attached data set. It builds on the abilities of *TPageProducer* in that the document fragment can be specified as either embedded text or a file reference, and inherit the substitution operations on fields within the snippet.

We create the new component and derive it from *TPageProducer*. Next, we add the new properties. Obviously, we add one to refer to the attached data set, *DataSet*, as well as *NoRecsFile* and *NoRecsDoc*, to allow for the reporting of a lack of data. The constructor and destructor are overridden to allocate and release the string list used by the *NoRecsDoc* property, and the *Notification* method is overridden to clear our reference to the data set if it's deleted.

So much for housekeeping; now we can add the new functionality. Browsing through the code for *TPageProducer* we find that all content requests end up going through the *ContentFromStream* method. This means that if we override this one method to cycle through each record, it will work no matter how the content is requested.

In our version of the method, we first check if *DataSet* exists, is open, and contains data. If so, we reposition the data set to the beginning before stepping through each record and applying our template to it (see Figure 9). Here we make use of the functionality of the ancestor to perform the processing of the template through the call to the inherited *ContentFromStream*. Note that we must reset the template stream to the beginning each time around the loop as it is processed within the inherited method.

To automatically substitute field values for tags with their names, we must override another inherited method. Some examination reveals the *DoTagEvent* routine as the one we want. In *TPageProducer*, it simply calls the *OnHTMLTag* event handler if it exists. Instead, we want it to try to match the tag name with a field name, and only call the event handler if that fails (see Figure 10). An exception is raised if the data set isn't active, or if the field doesn't exist. We trap this and redirect processing to the user event instead.

```
// Replace field references automatically.
procedure TRecordPageProducer.DoTagEvent(Tag: TTag;
  const TagString: string; TagParams: TStrings;
  var ReplaceText: string);
begin
  try
    ReplaceText :=
      FDataSet.FieldByName(TagString).DisplayText;
  except
    inherited DoTagEvent(Tag, TagString,
                         TagParams, ReplaceText);
  end;
end;
```

Figure 10: Automatically replacing field references.

Using this new component, instead of the basic PageProducer, makes our code that much simpler. The original application has been updated in CGIXML2.dpr to demonstrate the new abilities.

Enhancements

The applications presented here demonstrate how to produce an XML document from an existing database on demand. Recall, however, that Web modules are able to accept additional parameters from the user. These can be used to further customize the output, either by providing a subset of the data in the first place, or by referring to a different style sheet from within the document. The latter allows for presenting the same XML document in different ways, and can include its own selection criteria. In fact, we could generate a customized style sheet, as well as the original XML.

Another useful enhancement would be some sort of automatic generation of the XML document directly from the database. This could take the form of an expert that allows us to select the database, tables, and fields required, specify how each field is to be presented (as an element or an attribute), then create the Web module for us. Alternately, it could work from the DTD and match this with the fields in a selected database.

Conclusion

The Internet technologies built into Delphi enable us to quickly generate server-side applications for processing and delivering data. We can use these abilities to produce XML documents, as well as conventional HTML documents. The full functionality of Delphi can be brought to bear on the problem, allowing us to access databases and to customize the produced documents.

To make the processing of information from data sources easier, we created the *TRecordPageProducer* component that cycles through each record in its attached data set and applies its HTML/XML template to each one. ∆

References

- XML specification: http://www.w3.org/XML
- 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 CD located in INFORM\00\JAN\DI200002KW.

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

By Alan C. Moore, Ph.D.

TSyntaxMemo

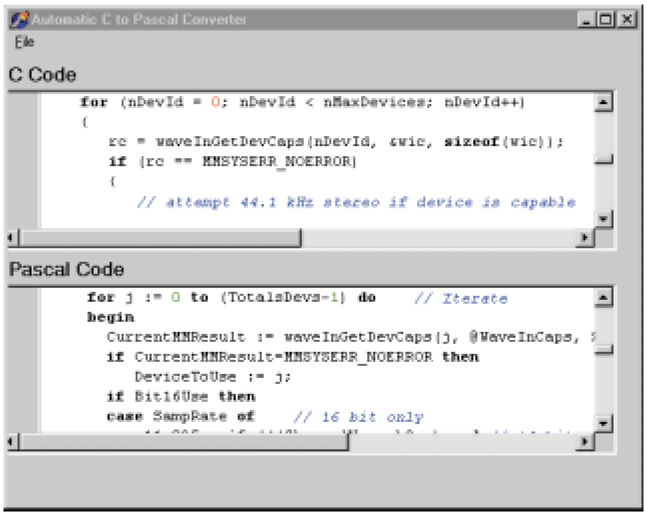
The Easy Road to Syntax Highlighting

If you display source code in your applications (e.g. code editors, experts, or similar tools), you want that code to look professional. In most cases, you want to duplicate the syntax highlighting of the application that produced or compiled the code. If you're working with Delphi code, you would like your editor/viewer to have the same look as the editor in Delphi's IDE. You could go to a lot of trouble parsing the code and using a RichEdit control to display it. I've tried it; it's a lot of work.

On the other hand, you could try the excellent set of components we'll be examining in this review: dbRock Software's TSyntaxMemo. Although this library consists of only three components, there's a great deal more power here than you might expect. As we'll see, you can take advantage of the main features of this library by simply dropping a couple of components on a form and setting a few properties. Or you can get under the hood and do some amazing things. We'll begin with an overview of the library, then take a closer look at its components and their properties.

An Overview

The library consists of three components: *TSyntaxMemo*, the main component; *TDBSyntaxMemo*, a data-aware version of the main component; and *TSyntaxMemoParser*, a non-visual component that provides the parsing functionality for the other two. *TSyntaxMemoParser* is essential, so it's a property of the two memo components; it's required to enable syntax-highlighting functionality. In fact, with either *TSyntaxMemo* or *TDBSyntaxMemo*, you can have multiple parsers. Supporting *TSyntaxMemoParser* is a powerful scripting language, and an arsenal of built-in parsers for most of the common programming and scripting languages: Pascal (Delphi), C, Java, HTML, etc. Let's start by taking a detailed look at the main component.



```
for (nDevId = 0; nDevId < nMaxDevices; nDevId++)
{
   rc = waveInGetDevCaps(nDevId, &wic, sizeof(wic));
   if (rc == MMSYSERR_NOERROR)
   {
      // attempt 44.1 kHz stereo if device is capable
```

```
for j := 0 to (TotalsDevs-1) do    // Iterate
begin
  CurrentMMResult := waveInGetDevCaps(j, @WaveInCaps, 
  if CurrentMMResult=MMSYSERR_NOERROR then
     DeviceToUse := j;
  if Bit16Use then
  case SampRate of    // 16 bit only
```

Figure 1: C and Pascal code displayed in the Automatic C to Pascal Converter.

Though *TSyntaxMemo* isn't a descendant of *TMemo*, you can use it exactly as you would Delphi's *TMemo* component. A syntax highlighting editor component, *TSyntaxMemo* implements all properties, methods, events, and messages of *TMemo*, as well as many new ones. In one version of my Sound Component Expert, I concluded the component-producing expert by displaying an edit window whose main component was a *TMemo*. When I learned of this library and had a chance to try it out, I was delighted with the difference that substituting a *TSyntaxMemo* component for a *TMemo* component made. Figure 1 shows an even more impressive use of these components. In this deceptively named Automatic C to Pascal Converter, there is C code in the top pane and Pascal code in the lower pane.

So, how does *TSyntaxMemo* perform its magic? *TSyntaxMemo* depends on one or more *TSyntaxMemoParser* components to analyze code in preparation for syntax highlighting. You can use up to six different *TSyntaxMemoParser* components with each *TSyntaxMemo* component. Of course, each *TSyntaxMemoParser* component would define a different source code format to be highlighted. The selected *TSyntaxMemoParser* component is controlled through the *ActiveParser* property of *TSyntaxMemo*.

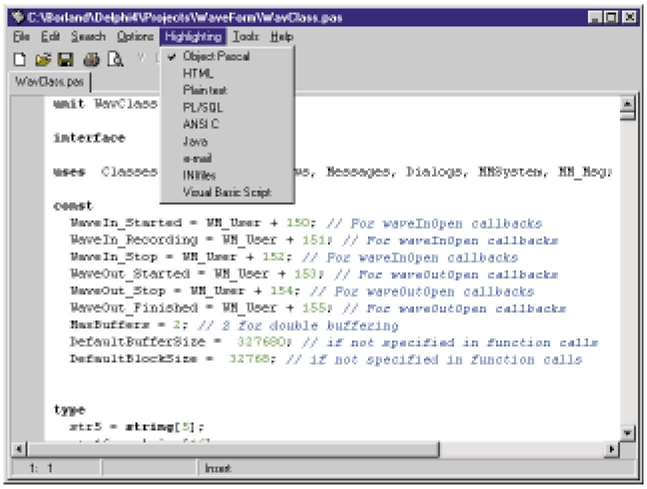
TSyntaxMemo supports OLE-based drag and drop, allowing text from other applications to be easily moved to and from editors created using *TSyntaxMemo*. Unlimited undo and redo is built in. The library also includes support for bookmarks, regular expressions in search queries, display of glyphs in a customizable gutter, and file sizes limited only by available memory.

TSyntaxMemo sports a powerful arsenal of nearly 70 new properties. I'll discuss some of them in a general way and a few of the more important ones in more detail. The *ActiveParser* and various *Parser?* properties give you complete control over which language (and what type of syntax highlighting) you're currently supporting. *LanguageNames* provides the names of those languages you're supporting, so you can display them in a menu. This is demonstrated in one of the example programs we'll examine shortly. *ClipCopyFormats* controls which formats can be copied to the Clipboard, and *CursorTokenText* provides the name of the type of format at the cursor position.

Users have come to expect certain options in a modern editor. With properties to support word wrap, undo/redo, setting tabs, and bookmarks, TSyntaxMemo provides just about all the editing capabilities you might want. The *Options* property we'll be examining provides even more choices in this and other areas. A powerful example program, TSMEd, demonstrates most of the library's features (see Figure 2). The menu item selected in this example shows clearly the languages supported.

Support for text color, font styles, and background color is essential (it's included in the three *def_?* properties), but the control over the appearance of this control goes much further. With the *Gutter*, *GutterColor*, *GutterFont*, and *GutterGlyph* properties, you can control the width, color, font, and glyphs used in the gutter — if you choose to use the gutter, that is. Likewise, the *LineGlyphs*, *LineColor*, and *LineTextColor* properties control the glyphs, the background color, and the text color for a single line. *SelColor* and *SelTextColor* control the background color and the text color for selected text. Finally, margin color allows you to get or set the background color for the margin area between the left-edge gutter and the text area.

In sum, the *TSyntaxMemo* properties give you a great deal of control over the elements of a code editor — from navigation, streaming data, and searching, to special highlighting. One of these properties is particularly rich. The *Options* property is a set of some 20 choices that control editing and other behaviors. The options give you and



```
unit WavClass

interface

uses  Classes                us, Messages, Dialogs, MMSystem, MM_Msg;

const
  WaveIn_Started = WM_User + 150; // For waveInOpen callbacks
  WaveIn_Recording = WM_User + 151; // For waveInOpen callbacks
  WaveIn_Stop = WM_User + 152; // For waveInOpen callbacks
  WaveOut_Started = WM_User + 153; // For waveOutOpen callbacks
  WaveOut_Stop = WM_User + 154; // For waveOutOpen callbacks
  WaveOut_Finished = WM_User + 155; // For waveOutOpen callbacks
  MaxBuffers = 2; // 2 for double buffering
  DefaultBufferSize =  327680; // if not specified in function calls
  DefaultBlockSize =  32768; // if not specified in function calls


type
  str5 = string[5];
```

Figure 2: The languages supported in the example program TSMEd.

your users a great deal of control over printing, word wrapping, word selection, and more.

There are also over 60 methods. Many you would expect, such as *LoadFromFile*, *LoadFromStream*, *SaveToFile*, and *SaveToStream*. You can also save and retrieve bookmarks, and manipulate them in other ways. The various categories into which the properties fall — navigation, printing, word wrapping, and word selection — are also supported by these methods. There are also methods for working with special glyphs and marks. Finally, some 25 events allow you to monitor and respond to these and other kinds of activity during editing.

With the data-aware version of this component, *TDBSyntaxMemo*, you can edit or display a database's BLOB (binary large object) test field with all the syntax highlighting you need. This component uses the *Text* property to represent the contents of the BLOB field. As in any memo component, it allows multiple lines of text. A descendant of *TSyntaxMemo*, *TDBSyntaxMemo* contains the additional properties — *DataField* and *DataSource* — necessary for it to work with a BLOB field in a table.

The final component, *TSyntaxMemoParser*, performs the actual work of analyzing text in a particular language, and determines how that text should be highlighted. It has considerably fewer properties and methods, and is based on a powerful scripting language that controls the parsing.

Although you can use the *Compile* property only at design time, you can use the *CompileScript* method at run time to compile the parser-defining text in the file specified by the *Script* property. Similarly, by assigning to the *Script* property at design time, the specified script will be compiled, and the compiled version of the script will be saved with the application. However, at run time you need to call the *CompileScript* method to explicitly compile a script. *UseRegistry* must be True or *RegisterKey* will have no effect. Parsers rely on scripts, because it's these text files that define the key elements of a language that may require special formatting for displaying or printing.

Parser Scripts

TSyntaxMemoParser uses scripts to define the analysis and appearance of text in a *TSyntaxMemo* control. This library includes a number of built-in scripts for popular languages and a good deal of information to help you build your own. With these scripts, you can define most aspects of the editing environment, such as *TabColumns*, *Gutter* settings, the property editor display, and auto-replace entries. These scripts are written in a standard way, with specific sections that carry out certain functions. The dbRock Software Web site includes an excellent example of techniques for defining special syntax circumstances. Learning this scripting language isn't trivial, particularly if you want to accomplish complex tasks. Once you've mastered it, however, you can extend the capabilities of these components considerably.

INFORMANT FACT FILE

TSyntaxMemo is a powerful set of syntax-highlighting components for Delphi and C++Builder. TSyntaxMemo includes full integration with AddictSpell and supports on-the-fly spell-checking as in recent versions of popular word processors. This feature-rich component set supports unlimited undo/redo, use of Clipboard, and printing of highlighted text. It includes a good Help file and several excellent demonstration programs. Customers who register for the current version will be entitled to free upgrades to future versions of TSyntaxMemo.

dbRock Software
26 Kemp Avenue
Paisley PA3 4JS
Scotland
United Kingdom

Phone: 07712 899461
E-Mail: dbrock@dbrocksoftware.com
Web Site: http://www.dbrocksoftware.com
Price: US$169

Conclusion

I'm most impressed with this library. It's clearly the best solution for any developer who needs to add syntax highlighting functionality to an application — either in a browser or a code editor. TSyntaxMemo comes with full source code, a very good Help system, and excellent support. One of the features I almost forgot to mention is the excellent built-in property editor. Though the price is a bit high, it's worth remembering that you will never need to pay for an upgrade — they're free. Visit the dbRock Software Web site. If nothing else, the downloadable demonstrations highlight the awesome power of Delphi, and the talent of the excellent developers we have in our midst. I recommend these components highly and without reservation. ∆

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\JAN\DI200002AM.

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.

Delphi 5: A Portent?

By now most of you have read numerous reviews and articles about Delphi 5. Many of you have already upgraded. Rather than simply go over that territory again, I would like to take a different approach and try to answer these questions: How is the release of Delphi 5 different from that of Delphi 4? And what does this indicate about changes in Inprise's approach?

Delphi is Inprise/Borland's flagship product. One indication of this is that the Delphi sessions have been the best attended at recent Inprise conferences. In the first Delphi session, the presenter asked, "When should we release Delphi 5?" As you can probably imagine, many in the audience shouted back, "Now!" But the presenter was ready for that and came back with "Wrong answer! We'll release it when it's ready." Although it may seem the obvious answer, it does represent a shift in philosophy, a return to the emphasis on quality that made this company great in the first place. It represents a shift of strategy for the company compared to the rushed release of Delphi 4. The strategy embodies a rediscovery of one of the cardinal rules of software production: Better late than lousy.

In reviewing the new features, I'll begin with one that is seldom mentioned: the Help files. Because I have been working on a book on Delphi and multimedia, the expanded Help in this and other Microsoft API areas caught my attention early on. The new Help is more extensive and better organized. If you want to get a good idea of the other improvements, check out the help topic, "What's new in Delphi." I'll outline some of the highlights.

An IDE overhaul. The enhancements to Delphi 5 fall into various categories, including the IDE, debugger, database functionality, and VCL. The most impressive changes are to the IDE. In fact, I can't recall any version of Delphi that matches this explosion of new capabilities. Based on the reaction I saw to these features at the conference, and since, it certainly seems that Inprise is working to keep Delphi the best development environment on the market. Let's examine some of the details.

All versions of Delphi 5 allow you to customize and load different desktop layouts (including debugging desktops), and they feature an Object Inspector with a new look that includes images in drop-down lists and categories of properties. Now you can even modify the editor's properties by customizing its key mappings. There was a session on this latter capability at the Inprise conference, the first time I remember a Borland engineer delving this deeply into Delphi's inner workings. The Project Manager includes a number of new features that will remind CodeRush users of certain plug-ins under version 4. Among many others, the file-management capabilities now include drag-and-drop copying of files from a Windows folder into a project. To-Do Lists, which allow you to keep track of project-related tasks, are available in the Professional and Enterprise editions.

Better support for debugging. Another area that includes major improvements is the debugger (see Robert Vivrette's article, "Delphi 5 Drill-Down," in the August, 1999 *Delphi Informant Magazine* for further details). The debugging windows now include drag-and-drop capabilities. Breakpoints, always important in debugging, are more powerful than ever: You can now organize them into groups, enabling or disabling them at will. You can even associate various actions with your breakpoints. There is a new floating-point debugging window, as well as several new debugging commands and options.

Of course there are many new database features, new and improved components, and even some new tools in the Enterprise edition. The database enhancements have been covered extensively in this magazine (in particular, see Dr Cary Jensen's "Delphi 5," also in the August, 1999 *Delphi Informant Magazine*). One of the frequently mentioned new features is frames — similar to compound components in many ways, but more flexible and dynamic. I was immediately intrigued with how these frames might compare with the so-called "features" I wrote about in my review of Nevrona Designs' RAD tool, Propel (see the July, 1998 of *Delphi Informant Magazine*). I learned that although frames are certainly more powerful than compound components, they are not nearly as powerful as Propel's features, which are more flexible. With features you can manipulate (delete, move, add to) the contained components without changing the main feature, and you can encapsulate the behavior of the original components in the composite feature. The president of Nevrona Designs has informed me that he intends to release the latest incarnation of Propel, ND-Patterns, as a free tool, charging only for the source code and technical support. Visit their Web site at http://www.nevrona.com for more details.

Besides this wonderful assortment of new features and improvements, I am equally impressed with what isn't in Delphi 5: The bugs that plagued the release of Delphi 4. Of course, there are bugs — there are always bugs. But the number and severity seems insignificant, especially when compared to the last release. This is a direct result of Inprise obeying the better-late-than-lousy rule.

I agree with my colleagues who've praised this release of Delphi. It's an outstanding development tool! Further, I predict that if Inprise continues to exercise such attention to quality and detail, the future will be bright for Delphi and Inprise.

— 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.