Borland Delphi 2.0

by Ted Faison

Ted Faison has written several books and articles on object-oriented programming and Windows. He is president of Faison Computing Inc., a firm that develops Delphi, C++ applications, class libraries and software components for Windows. Ted can be reached at tedfaison@msn.com.

Introduction

Everyone is talking about component software and rapid application development. Component software tends to mean different things to different people, and a lot of people associate components with custom controls. Software components are much more than mere custom controls, and no environment that I know of today uses components in a more integrated and sophisticated way than Delphi.

Delphi 1.0 was released by Borland in 1995, and immediately heralded as a landmark product. And justly so. Delphi 1.0 is not simply a "better Visual Basic". It doesn't even make sense to compare Delphi with Visual Basic. We're talking apples and oranges. Just for starters, Delphi produces highly optimized compiled code. No interpreters to mess with or distribute with your applications. With Delphi 1.0 you can just as easily create graphical front ends, database applications, scientific programs or graphics programs. Database applications can be built using a desktop or client-server architecture.

Now there is Delphi 2.0, which adds better support for OLE automation, MAPI applications and Windows 95 controls. Delphi 2.0 generates 32 bit applications, delivering a substantial performance increase over 16 bit Delphi applications. I tested a pre-release copy of the full Client Server version of Delphi 2.0, which runs on both Windows 95 and Windows NT. Because Delphi 2.0 is a 32 bit environment, it doesn't run under Windows 3.1. The Client/Server version comes with integrated PVCS version control system, allowing programmers on large teams to work together on the same files without undoing each other's work. Delphi is available in 3 different configurations, to satisfy the needs of beginners or home users as well as corporate development teams.

Why ObjectPascal?

Probably the most frequently asked question by newcomers to Delphi is, *"Why the heck didn't they use C++ as the native language?*". Just as it seemed like Pascal's days were numbered, Borland comes out with an entirely revamped version of the language and creates Delphi with it! There are two main reasons for ObjectPascal: flexibility and performance. You appreciate the performance both at design time and runtime. Remember that Borland put the *turbo* into Pascal, and the Delphi compiler is nothing less than superb. Compiles execute in seconds. Complete project rebuilds require often less than a minute. One reason is that Pascal in general is a much simpler language than C++. Another is that C++ source files are subject to the header file inclusion nightmare. It

isn't uncommon to compile a simple C++ file and see the compiler reading over 50,000 lines of include files. No such penalties in ObjectPascal.

You'll also appreciate the performance of compiled ObjectPascal code at runtime. The compiler includes a full optimizer, and allows the inline inclusion of assembly language code for critical code.

To create a package like Delphi requires language features that are not available in any off-the-shelf language. Because Borland owns the ObjectPascal language, it doesn't have to ask anybody's permission, or get approval from any ANSI committees to make changes to ObjectPascal. Language features needed to be added to support the visual programming metaphor of Delphi and to deliver intrinsic support for the Windows messaging system.

Here's one example of a language feature Borland added. Delphi components use special *properties* that programmers edit at design time. Properties are special class data members that must be declared in the **published** section for a class. C++ programmers have only 3 choices for access control: **private**, **protected** and **public**. ObjectPascal has those three, plus **published**. Properties are special public members that show up in the Object Inspector, so programmers can edit them at design time. The Object Inspector is a special window that Delphi provides to let you see and edit the properties for a VCL component. VCL, or Visual Component Library, components are Delphi objects you build applications with. Each component type has its own list of properties. Figure 1 shows the Object inspector for a database grid control.

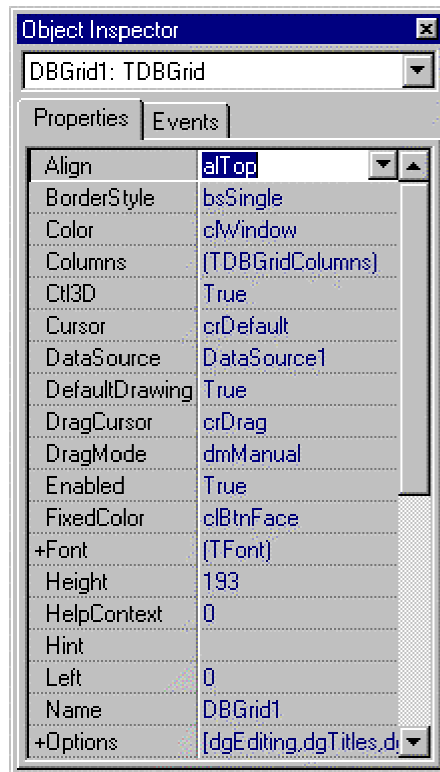

Figure 1 - The Object Inspector, showing the properties for a database grid control

Reading and writing properties is accomplished through special read and write accessors, allowing simple assignment operations to produce desired side-effects. Consider a property declared like this:

```
published
  property Height: Integer read GetHeight write SetHeight;
```

Reading this property will automatically invoke the `GetHeight` method. Writing to it will call the `SetHeight` method. If you have an object called `MyComponent` that uses the `Height` property, then you can write code like:

```
MyComponent.Height := 10;
SomeInteger := MyComponent.Height;
```

For the first statement, the ObjectPascal compiler will substitute the code

```
MyComponent.SetHeight(10);
```

For the second it will produce the code

```
SomeInteger := MyComponent.GetHeight;
```

The neat part is that you can forget the access functions and concentrate on the property, even though reading or writing it may produce side effects, such as refreshing the screen or performing some internal calculations. The value shown for a property in the Object Inspector is obtained by Delphi by calling the property's `Get` accessor. Similarly, when you edit a property value, Delphi changes the internal value by calling the `Set` accessor.

The keyword is Reusability

Delphi was designed from the ground up to make components reusable. Practically everything is reusable, and in more ways than one. Assume you created a form to handle TIFF images. Maybe the form has special tools to control image effects. In Delphi you can place the form in a special Component Repository, making the form available for subsequent reuse. To do so, first you create the form, then use the **Project | Add to Repository** command. The repository consists of stored components organized into pages. You select a page for your form, a name, description and author, and the form is stored. The repository is displayed in response to a **File | New…** command, appearing as a tabbed dialog box as shown in Figure 2.

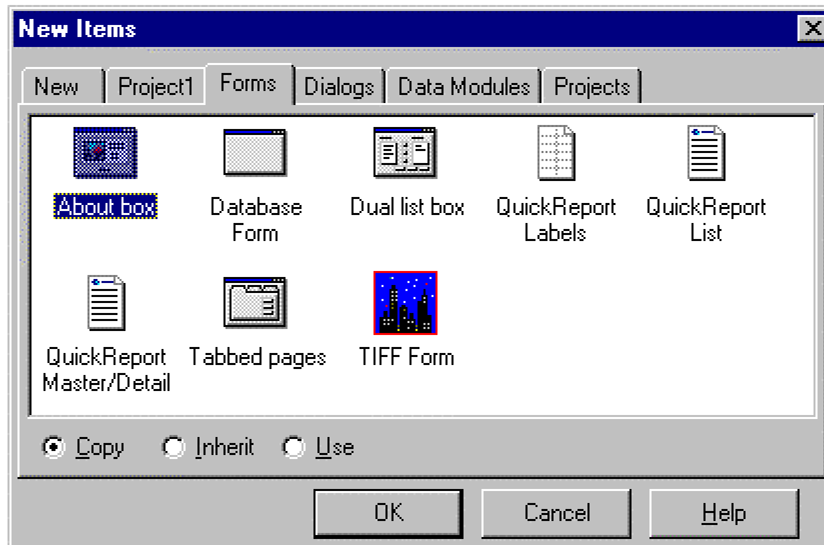

Figure 2 - The Delphi repository, showing the TIFF Form added to the Forms page.

You can add your own pages to the repository, turning it effectively into a Software Component library. Once a component is in the repository, it can be reused in any of three ways: by copying the source code into your project, by using it as a base class for a new component, or by referencing its code – without copying it – in your project.

Almost anything can be reused through the repository. The only essential requirement is that the object be a VCL component, meaning that it must be derived from the Delphi VCL class `TComponent`. You can even reuse an entire application, which is Delphi way of creating new projects, rather than using some kind of Project Expert or Wizard.

C++ programmers use inheritance all the time, but there certain things you just can't inherit in a Windows environment, such as dialog templates. Say you have an About dialog box that looks like the one in Figure 3.

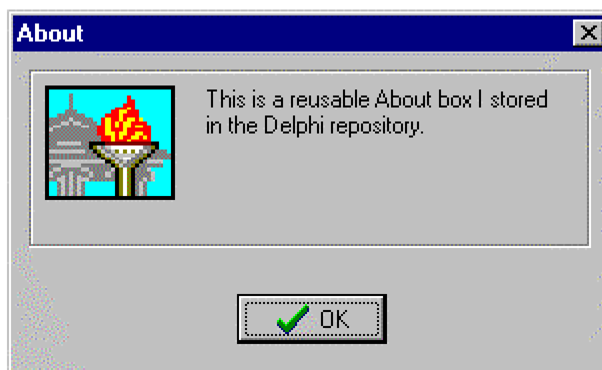

Figure 3 - A simple user form stored in the Delphi Component Repository.

There is no limit to the complexity of forms in the repository. Say you need a form that is almost like one in the repository. For example you need an About box like the one in Figure 3, but it needs a **Help** button, like the form in Figure 4.

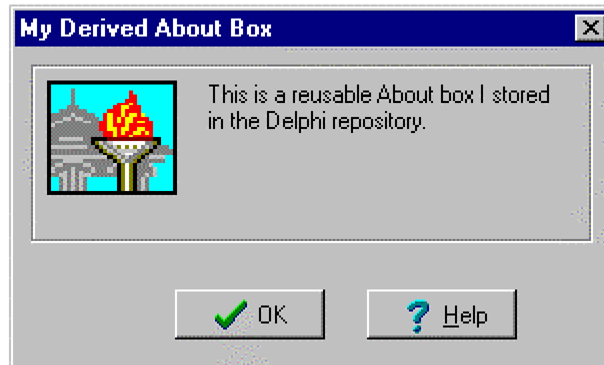

Figure 4 - A modified about box, created through inheritance.

In C++ you're out of luck: you can't inherit and make changes to dialog templates. You have to create a brand new template by copying the original form, being careful to match all the child Ids of the reused controls, so you can reuse the C++ code attached to the dialog. In Delphi it's much simpler -- you derive a form from the About Box form, and simply add the Help button graphically to it. The resulting derived form inherits the contents of the base form. And don't get the idea that Delphi is really making a copy of the base form and adding the Help button to it. The template for the base form is truly inherited. Changing the base class form affects all forms derived from it, just like with ordinary classes. It gets even better. When I added the Help button to my derived form, I adjusted the position of the inherited OK button, so the buttons would be centered in the form. How did Delphi handle this? It inherited the original OK button, and overrode only its position. Pure elegance… C++ programmers weep!

Multi-threading support is built-in

Many programmers shy away from using threads in their applications, due to the complexities of thread synchronization and maintenance. Delphi makes thread programming simple for many applications, hiding the details of thread start-up, synchronization and termination. Do you need threads in your app? Threads may be just the ticket to solving a user interface problem. Say you need to perform some lengthy operations, such as obtain records from a database or print a file. If you built the code straight into the app, the user has to wait until the operation terminates to continue with the program. If you want to allow the user to abort the operation by clicking a Cancel button somewhere, you need to add a loop that periodically checks the button. With multi-threaded code, things are much simpler: you create a worker thread to handle the lengthy operation, and you monitor the Cancel button in the main thread. If the user hits the Cancel button, the main thread terminates the worker thread and your done.

As an example, assume you have a form called `TMyForm` with a **Start** button to kick off some operation and an **Abort** button to terminate it. The operation itself will be performed by a class derived from the VCL class `TThread`, declared like this:

```
TMyThread = class(TThread)
protected
```

```
    procedure Execute; override;
public
    constructor Create(…your parameters…} );
  end;
```

The `Execute` method is where your lengthy code gets invoked. It overrides the virtual function by the same name in the base class. To use the new thread, you add an instance of it to the form or class that will control the thread, like this:

```
type
  TMyForm = class(TForm)
private
  LengthyOperation: TMyThread;
  end;
```

To start the lengthy operation, you add a handler for the start button like this:

```
procedure TMyForm.StartButtonClick(Sender: TObject);
begin
  LengthyOperation:= TMyThread.Create(… your parameter list…);
end;
```

To abort the operation, you just call the thread's `Terminate` method in the handler for the form's Abort button, like this:

```
procedure TMyForm. AbortButtonClick(Sender: TObject);
begin
  LengthyOperation.Terminate;
end;
```

Pretty simple stuff. The beauty of threads is that they allow you to split a large monolithic program into a collection of smaller tasks that have that ability to run in parallel. Users like programs that let them do more than one thing at a time, and Delphi makes multi-threaded programming much easier than ever before.

No limits in sight

Delphi uses VCL components to completely encapsulate the Windows API, providing a higher-level programming interface to deal with. You might say that Delphi is to C++ what C++ is to C. The result is that you can get the job done much easier and with less code. Consider the process of drawing something on the screen. All Delphi components that are displayable inherit a method called `Paint`. You override this method and use a special Delphi `Canvas` object to draw on. The `Canvas` is a high-level abstraction of a Windows device context. It handles all the drudgery of creating, selecting, deselecting and destroying GDI objects for you. To draw a circle, you add a `Paint` method to your form or component and draw on the `Canvas`, like this:

```
procedure TShape.Paint;
begin
  with Canvas do
        Rectangle(100, 100, 200, 200);
end;
```

It just couldn't get any simpler. Obviously you can change the attributes of the GDI objects before using them, using simple assignment statements on their properties. To create a navy-blue pen, you use the statement

```
Pen.Color := clNavy;
```

and that's it. You don't even have to create the pen, because Delphi does it for you automatically! Because `Color` is a property of `TPen`, changing its value invokes an accessor function that takes care of perfunctory Windows details automatically.

Because Delphi encapsulates the Windows API, it hides many things from programmers. If you need to get under the hood, to handle some low-level or unsupported Windows feature, no problem. If you want direct access to a low-level object, like a Windows device context or a window handle, it's all there for you. If you want to hook the message dispatcher, a simple `WndProc` handler will take care of it. Say you want to do intercept mouse clicks and typed characters for a form. The `WndProc` might look something like this:

```
procedure TMyForm.WndProc(var Message: TMessage);
begin
  case Message.Msg of
   WM_LBUTTONDOWN,  WM_CHAR:
      {do something}
      Exit;
  end;
  inherited;
end;
```

Calling the `inherited` method lets the base class Delphi handler process other events in the default way. Using a `WndProc` is considered a low-level message handling scheme. For all Windows and internal Delphi messages, the proper way to install a message handler is through the `message` keyword. For example, to add a handler for the WM_LBUTTONDOWN message, you add a class method like this:

```
procedure WMLButtonDblClk(var Message: TWMLButtonDown); message WM_LBUTTONDBLCLK;
```

Before adding a message handler like this to a class, you should make sure Delphi hasn't already provided an event handler hook for you. For most commonly used messages and events, Delphi provides easy entry points, using the **Events** page of the Object Inspector. Figure 5 shows the **Events** page for a `TButton` control.

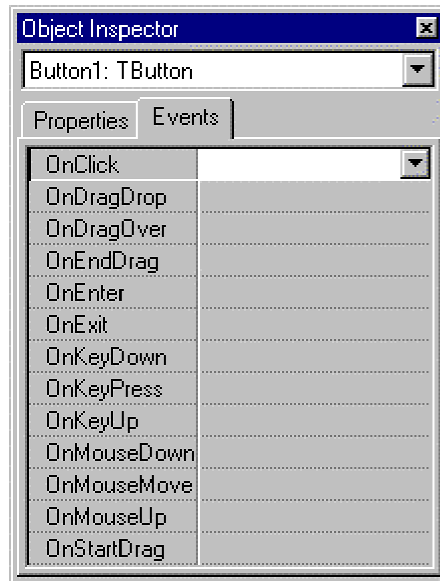

Figure 5 - The Object Inspector's Events page for a `TButton` control.

The **Events** page is certainly the best way to add message handlers to a VCL component. To provide a handler that gets invoked when your button is clicked, you enter a handler name in the `OnClick` field. The handler can have any name you want. By double clicking the empty field, Delphi will create a default name for you, based on the event type and name of the control. Delphi creates an empty handler you for, with correctly formatted arguments in the parameter list. The default handler for the `OnClick` event looks like this:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

All event handlers are passed a pointer to object that sent the message. You don't usually need to know who the sender is, but it's a standard Delphi thing. Anytime you double click a handler field on the **Events** page, Delphi immediately takes you to the unit that contains the handler code.

Database Support

If there is one area in which Delphi really stands out, it is database programming. It's not that Delphi was created specifically for database applications, but rather the fact that programmers need more help with database programming than many other application types. Delphi comes with a complete set of VCL components to handle everything from simple table viewers to multiple-join queries to triggers and stored procedures.

The heart of the architecture is the Borland Database Engine (BDE), which has a role similar to the Microsoft ODBC Driver Manager. When you develop database applications, you typically populate forms with a series of database-aware controls, like edit boxes, listboxes and grids. The `TTable` and `TQuery` components encapsulate the details of connecting with a database. A `TDataSource` component manages the mapping of database fields to data-aware components.

Using a VCL navigator tool, which appears as a set of VCR buttons, you can move the cursor in the result set. Figure 6 shows the relationship between an application and the database components.

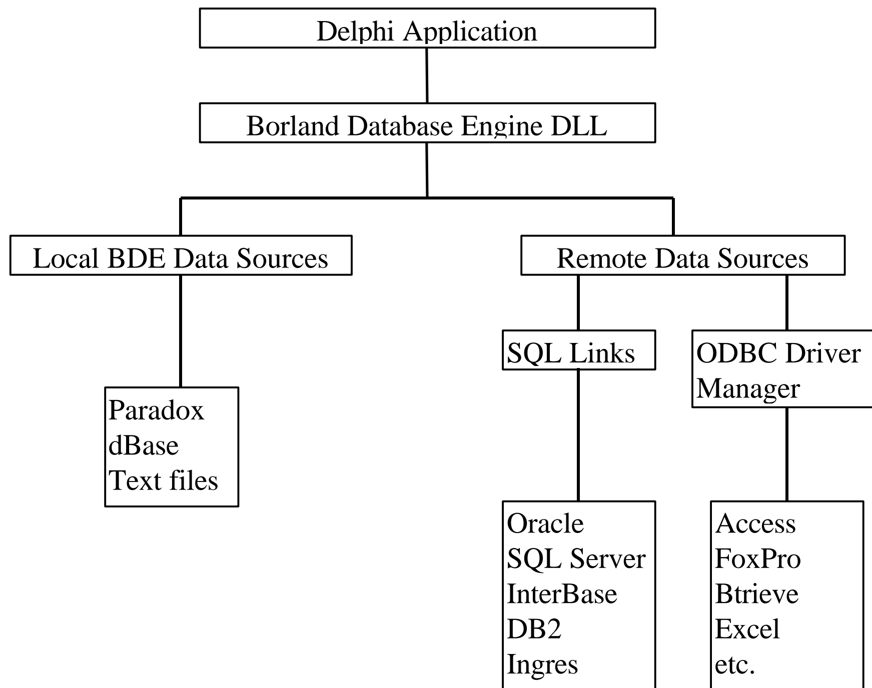

Figure 6 - The architecture of a Delphi database application.

All database access is through the BDE, including access to ODBC data sources. Accessing ODBC data sources incurs a very slight performance hit, compared to direct ODBC access, due to the pass-through of commands from the BDE DLL to the ODBC Driver Manager DLL. For operations that produce multiple records, the overhead is negligible. If your application makes lots of single-record operations, some degradation may be perceptible. I'm talking nuances in timing here. Usually the time required to perform even a single record database operation is eons longer than the transit time of a command through BDE to the ODBC layer.

Access to all client/server SQL databases is through SQL Links, a Borland DLL that uses vendor-specific drivers to talk to the underlying databases. InterBase is Borland's SQL database. It is a full 32 bit high-end client/server database that competes against the likes of Oracle and Sybase SQL Server. InterBase runs on most platforms, including Windows NT, Windows 95, Solaris, HP-UX, IBM AIX and SCO Unix. InterBase supports the creation of a local database architecture, whereby the database is on the same machine as the Delphi application. No networking or client/server stuff involved. This architecture is appropriate for small systems, or for the development and initial test stage for client/server systems. If and when you decide to deploy your application as a client/server system, you can switch over to the full client/server InterBase system, without needing to make any changes to your Delphi application.
I tested Delphi with Oracle 7.2, using Personal Oracle. After getting Oracle running with Delphi, everything proved to work great, but a real application will need to provide custom handlers for the

multitude of database exceptions that may originate, otherwise Delphi will display opaque message boxes that your database application users will be calling you about. Accessing Oracle data was just a matter of adding a `TTable` or `TQuery` component, sprinkling in a `TDataSource`, and putting some database controls on a form. Oracle log-on was supported automatically. Delphi 2.0 is configured to work with Oracle 7.1, and I had to use the BDE Configuration Utility to go in and change a number of low level settings to get the BDE to work with Oracle 7.2, which is the current version.

While the Delphi part was easy, it took me a long time to get the BDE to connect to an Oracle 7.2 database, mainly for lack of documentation on exactly what changes to make and where. I kept getting a message saying "*Vendor Initialization failed: ORANT71.DLL*". I tried a whole bunch of things without success. Software companies must always remember the cardinal rule that any time an error is reported to a user, it must be accompanied with some kind of information about how to overcome the problem. I resolved the issue simply: I called Borland. Even so, it took a while before I got a hold of someone who knew exactly what the problem was. It turned out I just needed to use the BDE Config tool to blank out the Server Name field, change the Net Protocol to the value '2', and change the vendor init entry to ORA72.DLL! Oh.

It's Alive!

Developing user interfaces for database applications requires laying out controls that are tied to database columns. With grid controls, picture controls, and graphic elements, it's difficult to lay everything out right unless you can see how the actual database data looks on the form. Borland pioneered the concept of *live data* with Delphi, which enables data-aware controls to connect to the underlying database right at design time. You don't have to compile or run any code to see the results in a form. The steps for creating live data forms are essentially trivial: you create a `TTable` or `TQuery` component on the form. You set its `DatabaseName` property to reference your database. You set the `TableName` to indicate which table in the database you want to use, then you set the `Active` property to `True`. That's it. Anything attached to the `TTable` or `TQuery` after that will get data right out of the database table at design time. To access the data you create a `TDataSource` component and set its `DataSet` property to the name of the `TTable` or `TQuery` component to use. Figure 7 shows a simple form at design time, with the grid control displaying live data.

Form1

Order No	Customer No	Sale Date	Ship Via	Ship Cost	Subtotal	Total Invoice	PaymentMethod
307	1480	9/1/92	UPS	$10.00	$295.00	$10,295.00	Visa
310	1481	9/1/92	FedEx	$12.00	$229.50	$6,229.50	Check
313	1909	9/1/92	Walk In	$0.00	$110.00	$8,110.00	Visa
314	1913	9/1/92	FedEx	$12.00	$142.50	$6,142.50	Check
317	1969	9/1/92	FedEx	$12.00	$331.50	$20,331.50	AmEx
320	2001	9/1/92	Walk In	$0.00	$650.00	$3,650.00	Cash
321	2306	9/1/92	Emery	$11.00	$0.00	$8,011.00	Master Card
322	2589	9/1/92	Emery	$11.00	$0.00	$8,011.00	AmEx

Figure 7 - A database form showing live data at design time.

You can scroll through the database data with the grid scrollbar to see all the records in a table or returned by a query. Seeing your data is not only convenient for layout purposes, it lets you verify immediately whether the form is retrieving the data you really want. All the database controls – grids, edit fields, check boxes, radio buttons, labels, listboxes, comboboxes and picture controls – support live data.

Certainly the neatest database control is the grid control. By default, connecting a grid to a data source makes the grid create columns for all the fields returned by the associated `TTable` or `TQuery`. The Object Inspector provides a `Columns` property that gives you access to a full Column editor, allowing you to eliminate columns you don't need. The editor also lets you change the column titles, widths, read-only status, color, and more. You can also make the grid control display check boxes or drop-down combo boxes in a grid field. Using the Object Inspector alone is often all the programming you need to create a fully functional database grid control.

Get me that Report

The purpose of a database application is to extract data from somewhere, and transform it into information for end users. Data is just numbers, information is a display of meaningful numbers, formatted according to user criteria. Many small Delphi applications use simple forms with data-aware controls and grids to show database data, but medium and large applications nearly always need reports to present information to users. Delphi has a built-in report generator, called ReportSmith, which allows you to create formatted reports in a completely graphical environment. ReportSmith lets you create reports with formatted text fields, pictures, and graphical elements like boxes and lines. The program has a Wizard that knows how to create 4 of the most-used report types: columnar, crosstab, form and labels. Starting from a default report created by the Wizard, you can create reports often with no programming at all. For specialized reports, there is an integrated SQL editor that lets you enter you own SQL Select statements.

ReportSmith produces great looking printed reports. You can also export the report data in a number of file formats, including Excel, Lotus 123 and Quattro Pro formats. I was disappointed RTF (Rich Text Format) wasn't a supported export type, because I often include parts of reports into word processing documents, and RTF is a common text exchange format. Microsoft Word knows how to import RTF files, but not the proprietary ReportSmith RPT files. And you can forget using cut and paste to select parts of a report, because ReportSmith doesn't allow you to copy parts of a report to the clipboard.

On the other hand, I was pleased to see that ReportSmith has the capability to produce labels. Lots of high-end report writers lack this feature, requiring you to adopt a complex process of connecting to a word processing application to print mailing labels.

An Example

To give you a taste of Delphi programming, I'll show an example of a small multi-threaded program. Using C or C++ you would expect to see lots of low level Windows API calls, like `CreateThread`, and deal with details like thread local storage and whatnot. Of course you can get to these details in Delphi if you want, but I'm guessing you don't until there is just no other alternative, and Delphi offers lots of alternatives. My example is called BOUNCE, a rather trivial program which shows three objects bouncing around inside separate regions of a window. Each object is a black square with a colored body. As it moves, it leaves a black wake behind, eventually painting over the entire bounce region. Figure 8 shows how BOUNCE looks after running shortly.

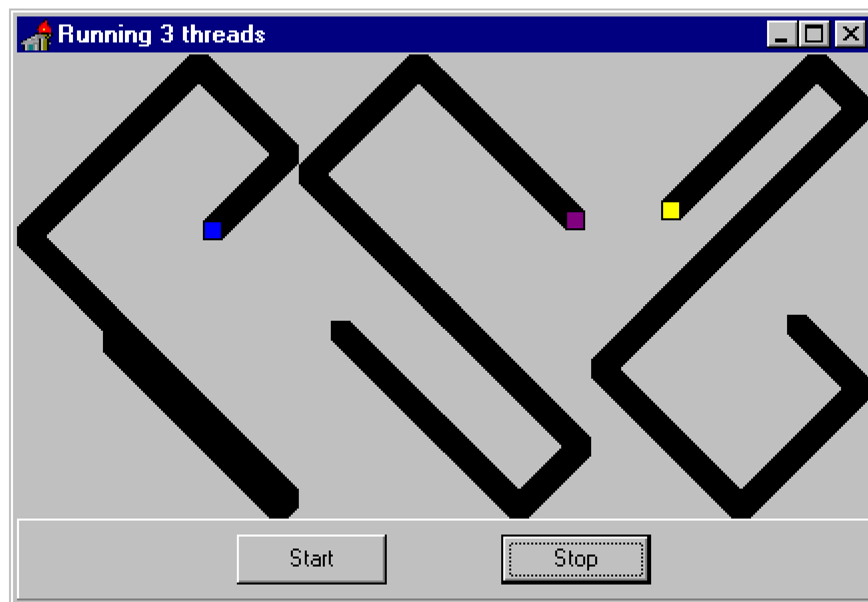

Figure 8 - The paths drawn by running BOUNCE briefly.

Each object starts off in a random position, moving downwards to the right. When it hits the boundary of the enclosing region, it bounces off and continues. The **Start** and **Stop** buttons control the 3 threads. The **Stop** button suspends them, the **Start** button makes them resume. Each object is animated by a separate thread, implemented as a class derived from the VCL class `TThread`, as shown in Listing 1.

```
unit ThreadTest;

interface

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

type
  TThreadBouncingObject = class(TThread)
  private
    ContainmentBox: TPaintBox;
    BoundsCheckBox: TRect;
    P: TPoint;
    XIncrement, YIncrement: Integer;
    procedure UpdateScreen;
  protected
    procedure Execute; override;
  public
    constructor Create(Box: TPaintBox);
  end;

const
  WIDTH = 10;
  HEIGHT = 10;

implementation

{$R *.DFM}

constructor TThreadBouncingObject.Create(Box: TPaintBox);
begin
  inherited Create(True);
  ContainmentBox := Box;
  BoundsCheckBox := Rect(0, 0, Box.Width - WIDTH, Box.Height - HEIGHT);
  P.X := Random(ContainmentBox.Width - WIDTH);
  P.Y := Random(ContainmentBox.Height - HEIGHT);
  XIncrement := 1;
  YIncrement := 1;
end;

procedure TThreadBouncingObject.Execute;
begin
  repeat
    begin
      {update the X position}
      P.X := P.X + XIncrement;
      if (XIncrement > 0) and (P.X >= BoundsCheckBox.Right) or
         (XIncrement < 0) and (P.X <= BoundsCheckBox.Left) then
        XIncrement := -XIncrement;

      {update the Y position}
      P.Y := P.Y + YIncrement;
      if (YIncrement > 0) and (P.Y >= BoundsCheckBox.Bottom) or
         (YIncrement < 0) and (P.Y <= BoundsCheckBox.Top) then
        YIncrement := -YIncrement;
      UpdateScreen;
      Sleep(10); {pause briefly}
    end
  until Terminated;
end;

procedure TThreadBouncingObject.UpdateScreen;
begin
    ContainmentBox.Canvas.Rectangle(P.X, P.Y,
                                    P.X + WIDTH,
```

```
                                    P.Y + HEIGHT);
end;

end.
```

Listing 1 - The code in unit ThreadTest, implementing the thread used in BOUNCE.

The constructor for class `TThreadBouncingObject` randomizes the initial position for each bouncing object, and saves a pointer to the containment bounce region. The thread execution is carried out by the `Execute` method, which runs an infinite loop moving the object. At the end is a `Sleep` call, which suspends the thread briefly, to slow down the bouncing objects. On my machine (a 100 Pentium), the objects were bouncing around so fast I couldn't tell what they were doing.

The main form creates 3 side-by-side regions of type `TPanel`, which are used as the containment boxes for the bouncing objects. The thread objects are created inside the `FormCreate` member, and destroyed in the `FormDestroy` member. These two functions are similar to regular constructors and destructors, except they are called as handlers by Delphi. The code for the main form is shown in Listing 2.

```
unit TestForm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, ThreadTest, StdCtrls;

type
  TForm3Threads = class(TForm)
    Panel1: TPanel;
    PaintBox1: TPaintBox;
    PaintBox2: TPaintBox;
    PaintBox3: TPaintBox;
    ButtonStart: TButton;
    ButtonStop: TButton;
    procedure FormDestroy(Sender: TObject);
    procedure ButtonStartClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure ButtonStopClick(Sender: TObject);
  private
    Object1: TThreadBouncingObject;
    Object2: TThreadBouncingObject;
    Object3: TThreadBouncingObject;
  public
    { Public declarations }
  end;

var
  Form3Threads: TForm3Threads;

implementation

{$R *.DFM}

procedure TForm3Threads.FormDestroy(Sender: TObject);
begin
  Object1.Terminate;
  Object1.Free;
  Object2.Terminate;
  Object2.Free;
  Object3.Terminate;
  Object3.Free;
```

```
end;

procedure TForm3Threads.ButtonStartClick(Sender: TObject);
begin
  Object1.Resume;
  Object2.Resume;
  Object3.Resume;
end;

procedure TForm3Threads.FormCreate(Sender: TObject);
begin
  Randomize;   {for the object initial positions}
  Object1 := TThreadBouncingObject.Create(PaintBox1);
  Object2 := TThreadBouncingObject.Create(PaintBox2);
  Object3 := TThreadBouncingObject.Create(PaintBox3);
end;

procedure TForm3Threads.ButtonStopClick(Sender: TObject);
begin
  Object1.Suspend;
  Object2.Suspend;
  Object3.Suspend;
end;

end.
```

Listing 2 - The code in unit `TestForm`, the main form used in BOUNCE.

Note that objects are created in 2 phases in Delphi. The declaration itself isn't enough, as in C++. You actually have to call an object's constructor to initialize it. The arrangement is similar to the way GDI objects are handled by Microsoft Visual C++: you declare things like `CPens` and `CBrushes`, and subsequently must invoke their `Create` members.

The Development Environment

In this day and age, application development programs are expected to integrate the various phases of code editing, compiling and linking. The Delphi integrated development environment does this, going one step further. Because Delphi supports user interface design with parallel code generation, it also has a Forms Designer that works in tandem with the Code Editor window. There is a button on the toolbar that lets you flip back and forth between a form's graphical view and its source code. If you add controls to a form, the code window is updated immediately and automatically.

Delphi projects usually entail multiple forms and code units, and Delphi makes it easy to manage the code units by keeping them all on a single tabbed window. Changing unit is a one-click operation, and all the names of all open units are always visible on the tab bar. Most other programs employ the older MDI approach, where you switch between files using the Window menu.

The Delphi Integrated Development Environment (IDE) is good, but not perfect. Its use of multiple overlapping windows sharing the screen tends to become a nuisance, with so many independent windows to manage. There is an Object Inspector, the Code Editor, the Forms Designer, the main toolbar, etc. Figure 9 shows a typical view of the Delphi IDE.

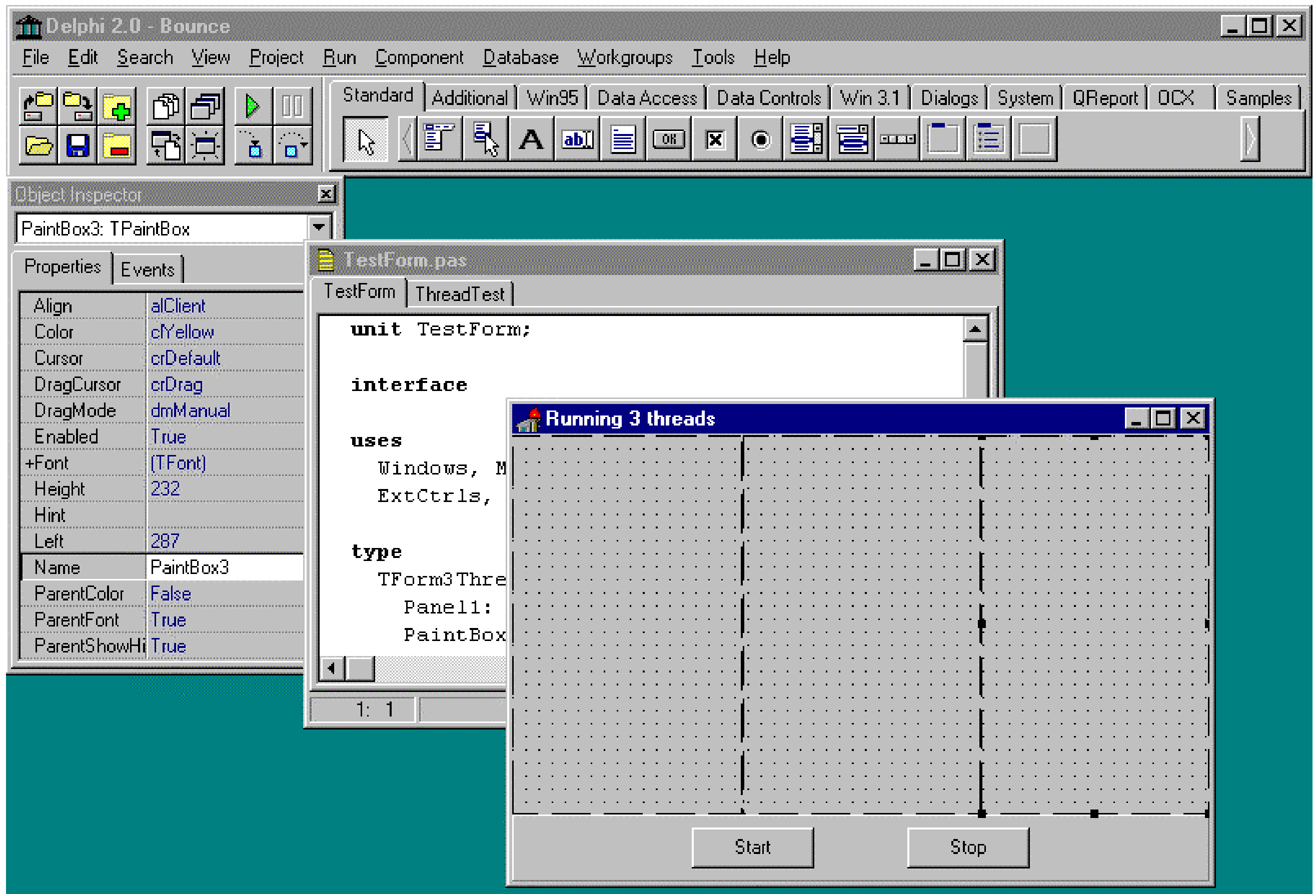

Figure 9 - A look at the Delphi IDE with its overlapping windows.

I would find it much easier to manage a single main window subdivided into moveable sub-panes, as in Microsoft Visual C++ or even Borland C++ ClassExpert. Borland added the 3-pane ClassExpert environment to C++ two years ago, reminiscent of the one found in SmallTalk. The top left pane lists all the classes in a project. The top right pane lists all the member functions for the class selected in the left pane. The bottom pane is an editor that displays the code for the member function clicked in the top right pane. Together, the 3-pane window gives you instant navigation capabilities for all functions in an entire project. Somehow the Delphi group went for the overlapped window look. The Object Inspector is nice for small projects, but inconvenient to use to get around in a project. What might be nice is to have some kind of object navigation outline pane, where you can see a complete hierarchy of all the objects in a project.

The tabbed toolbar is a nice touch, given the number of tools provided by Delphi. Because you can install your own tools, it is possible to run out of space on the toolbar, so Borland provided a set of scroll bars. You can also create new tabs on tool bar, so you can organize your tools any way you want.

I don't know about you, but when I work, I open lots of files. I always look for code to cut and paste, so I don't have to debug any more code than I have to. The easiest way to find code is to use an integrated GREP-like search tool, like the one found in Borland C++ or Visual C++. With an integrated tool, you can launch regular expression searches across entire directory structures. Files containing the target text are listed in a second window. Clicking on an item opens the corresponding file on the line containing the text. Alas, Delphi doesn't have a GREP tool, so while in Delphi I usually keep a session of Borland C++ or Visual C++ open to run searches in.

The Integrated debugger is very easy to use. The tool bar has buttons to start, pause, step into and step over code. A step out button is not available. You would use that button when you're single stepping inside a function, and want to return to the calling function. Most programming language IDEs have a step out command, including Borland C++. Why they left it out in Delphi is a mystery to me. May be they didn't want to crowd the screen too much. We programmers are a hard bunch to please… No matter what you give us, we'll always need more.

Another minor item I missed was an interactive tooltip for object inspecting at debug time. When you put a breakpoint in your code, you often want to inspect the value of a variable. Visual C++ pioneered the concept of tooltip inspection: You just move the mouse over a variable. If the variable is scalar and in scope, a tooltip window automatically appears, showing the variable's value. Cool! In Delphi, you have to right-click the variable, then chose the **Evaluate/Modify** command from the popup menu. If the object is complex, meaning it has data members or fields, Delphi just gives you a comma separated list of their values. I would have preferred a 2-column formatted table, with the first column giving the field name, and the second the value.

Conclusion

Delphi's database support in absolutely excellent. I consider it much better than that available in either Borland C++ or Visual C++, although the latest release of Borland C++ contains many of the database controls available in Delphi. The support for database exceptions, live data, grids, and BLOBs make Delphi a nearly ideal database development environment. Delphi makes it possible to develop local database applications that can easily be converted into full client/server ones without affecting your ObjectPascal code.

Overall I give Delphi a solid A+, and I think it deserves to be nominated Product of the Year again. Whether you program in C++, SmallTalk, Visual Basic or other, if you need a tool to create Windows applications fast, you should definitely take a look at Delphi. Delphi is one of the hottest products around. Its ObjectPascal language is perfect for writing Windows code, even though some programmers out there may be reluctant to switch to a new language. Of course Pascal is not new, having been around nearly as long as C. Don't let ObjectPascal scare you away. Anybody who has written a line of code in any object-oriented language will easily pick up ObjectPascal programming. Classes, data members, member functions….they're all there. The ObjectPascal syntax is easy to learn, and Delphi can easily reduce your development time from months to weeks – even days. Try it!