

Tech Notes

Reasons to Migrate from Delphi 7 to Delphi 2009

By Andreano Lanusse

February 2009

Corporate Headquarters

100 California Street, 12th Floor
San Francisco, California 94111

EMEA Headquarters

York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

Asia-Pacific Headquarters

L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

INTRODUCTION

Many Delphi 7 users wonder whether they'll find compelling reasons to migrate to Delphi 2009. Here they are: a plethora of new features allied to unparalleled developer productivity, all aimed at your ability to create higher-quality applications with improved performance. This article gives a few good reasons to migrate, along with an overview of all the new features added to Delphi since version 7.

WHAT'S NEW IN THE IDE

NEW PROJECT OPTIONS

We've changed the IDE in many different ways in order to make development faster and easier. The project compilation options are now displayed in columns and grouped by categories in a friendly manner. It's also now possible to save your project's configuration options, or build configurations, as you'll see in Figure 1.

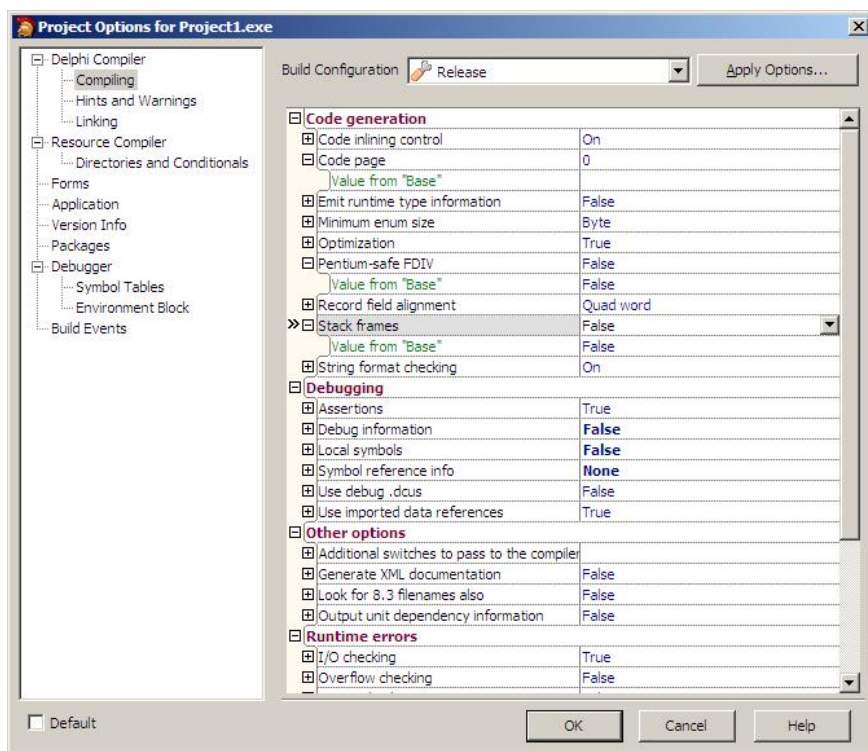


Figure 1 Build Configurations

BUILD CONFIGURATIONS

Compiling and debugging projects are regular tasks for developers. However, the project options that are used to run the final version (release) are not always the same project options you use when debugging. Having to constantly change your project's options is a time-consuming task that you can now avoid, never again being forced to spend lots of time working with the Project Manager. In Delphi 2009 the build configuration options are seamlessly integrated to the Project Manager.

In addition, project configurations can be saved in XML-format OPTSET files. Working with these files you're able to reuse project options from previous projects, no longer having to set them each time a new project is started.

COMPONENT CREATION WIZARD

The Component Creation and Import Wizards have been redesigned to include type libraries, ActiveX controls and assemblies. Both wizards can now install into an existing package or in a new package.

As you see in Figure 2, a new field was added to filter components, making it easier for you to locate the component you want to inherit.

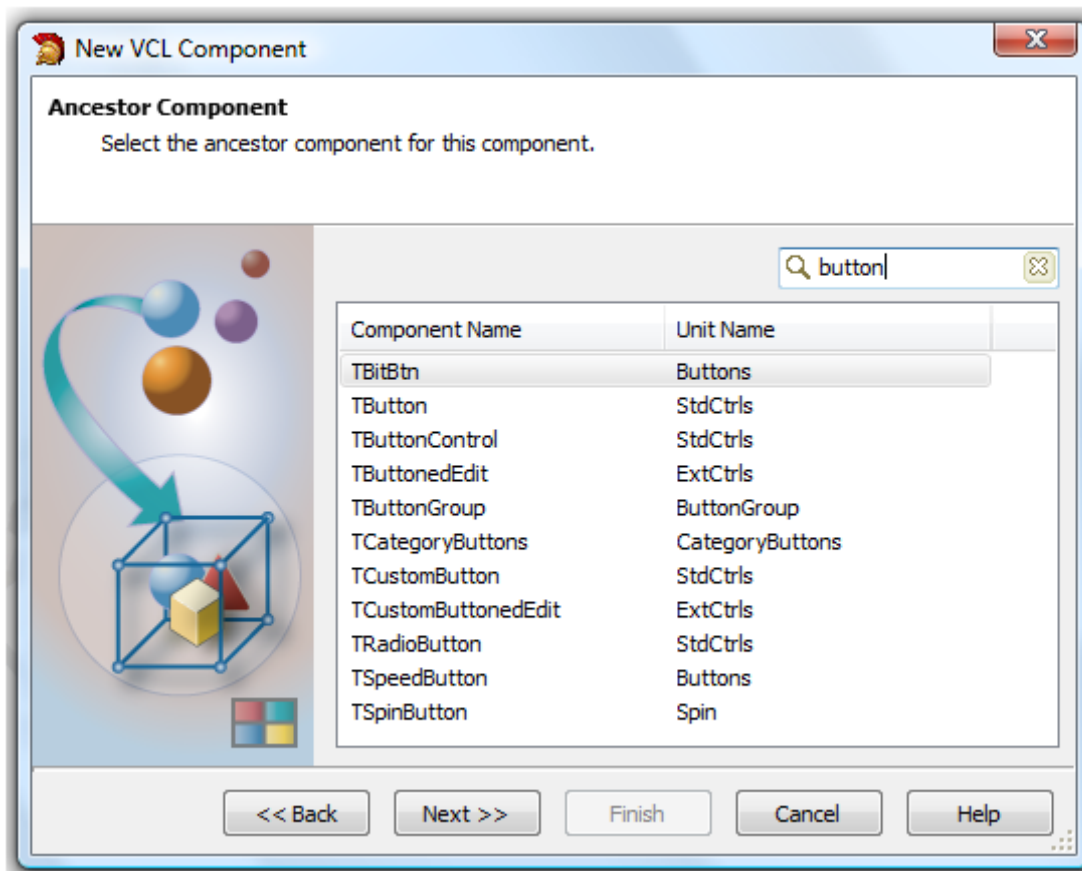


Figure 2 Ancestor Component

COM

COM wizards and the entire type library have been restructured. In fact, the COM Object Creation Wizards are all brand new.

What has changed? A new file type - RIDL (Restricted Interface Definition Language) – was added to the COM architecture. RIDL files work as recording devices projects use to save type libraries. Therefore, the tlb binary file becomes an intermediary file, like dcu, res, obj, and so on. This means developers are now able to recompile tlb files using the command line prompt, and even edit a tlb file using a text editor, while still keeping track of its version.

The type library now uses a text file (the RIDL file), not TLB. This is beneficial because:

- You no longer need to check the tlb file in. It's now automatically generated based on the last ridl
- Different developers can work with the same type library. This is so because the text file can now be merged, something that couldn't be done with the binary file used previously
- The RIDL format provides the Type Library editor with much higher flexibility
- You can easily compare different RIDL files

NEW RESOURCE MANAGER

The Resource Compiler allows you to choose between compiling your resources with BRCC32.exe or RC.exe (Microsoft Platform SDK resource compiler). RC supports the use of Unicode characters in resource files and file names. It also supports the new Windows Vista types (e.g., icons and alpha channel). When you use RC you must define `#include <winresrc.h>` explicitly both for Delphi and C++.

The New Resource Manager allows you to add many resource files (bitmaps, icons, fonts...) to your project.

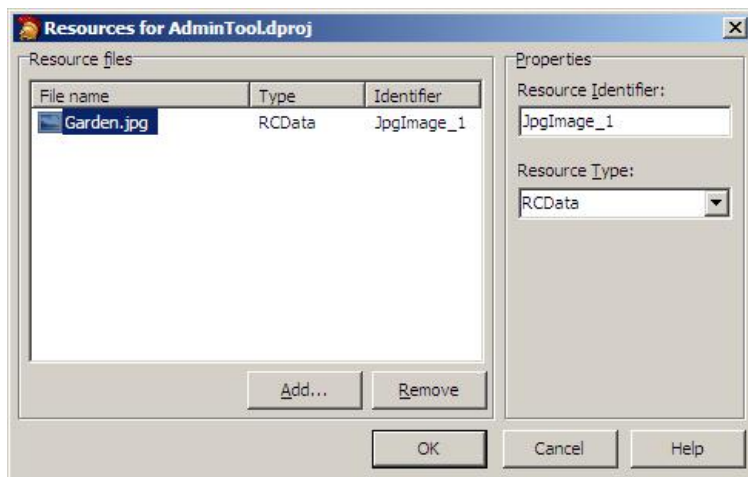


Figure 3 Resource Editor

CLASS EXPLORER

The Class Explorer is a very useful tool that enables you to visualize a project's class hierarchy and its interfaces, as well as add properties, methods and variables to it. These operations can be performed by means of UML, through the use of class models. UML is one of the many resources that were incorporated to Delphi.

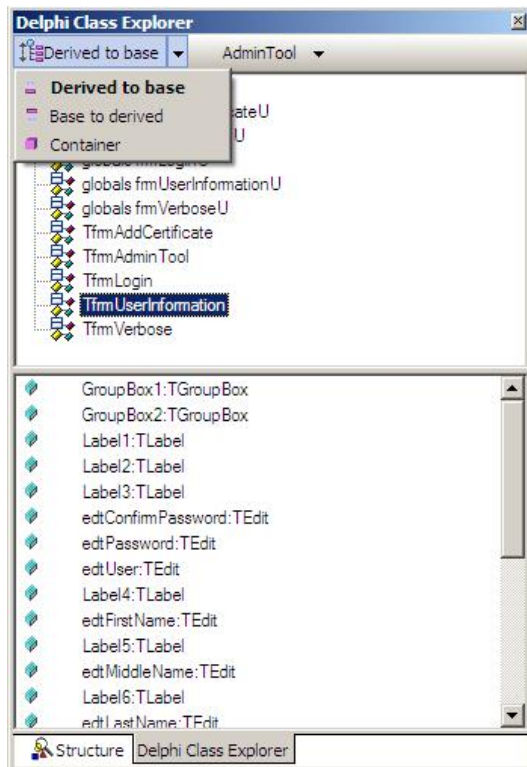


Figure 4 Class Explorer

COMPONENT SEARCH IN THE TOOL PALETTE

In Delphi 2006 you could filter components typing the first couple of letters of their names in the Tool Palette. In Delphi 2007 this feature was enhanced and you were then able to type in any portion of the component name. In Delphi 2009 an Edit field is used to achieve the same result, making this feature clearer and easy to recognize at first glance.

Users who prefer Delphi 7's layout (i.e., components displayed at the upper portion of the IDE) will be glad to know Delphi 2009's IDE can look just like Delphi 7's, except for the component bar.

However, before switching to the old Delphi 7 layout, give the Tool Palette new version a try. Locating components with ease, the orderly arrangement of categories, etc., can provide great productivity gains.

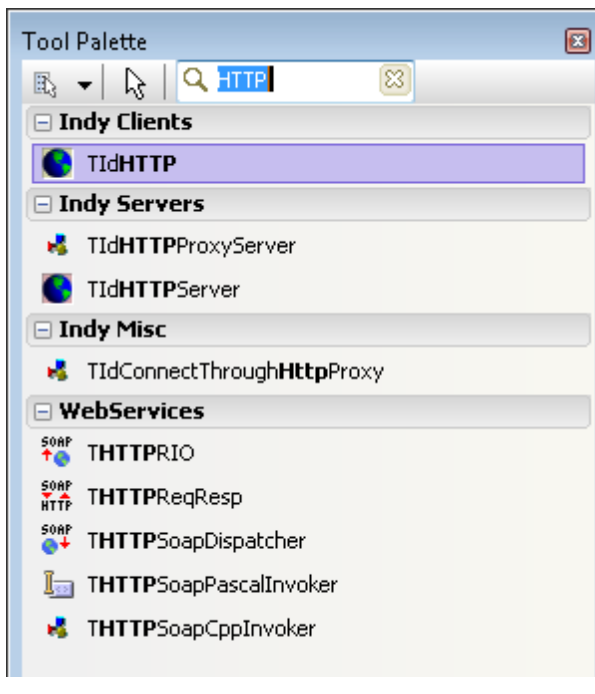


Figure 5 Component Search

CODE EDITOR

The new *Live Templates* feature extends your ability to create code templates in Delphi. These are created as XML files, and help you program with less code.

Block completion is one of the resources involved in enabling automatic begin and end. And who can honestly say he's never had a hard time with begin..end?

Consider this context: you want to change the name of all variables in a selected part of the code. Find..Replace is not a good practice for this situation. It doesn't guarantee only variable names will be changed in the process. Since Delphi 8 you can use Sync Edit to edit different portions of code simultaneously, provided they share the same identifier. Taking the code below as an example, you could select the entire block, make sure sync edit is active and then change the "Comm" variable a single time.

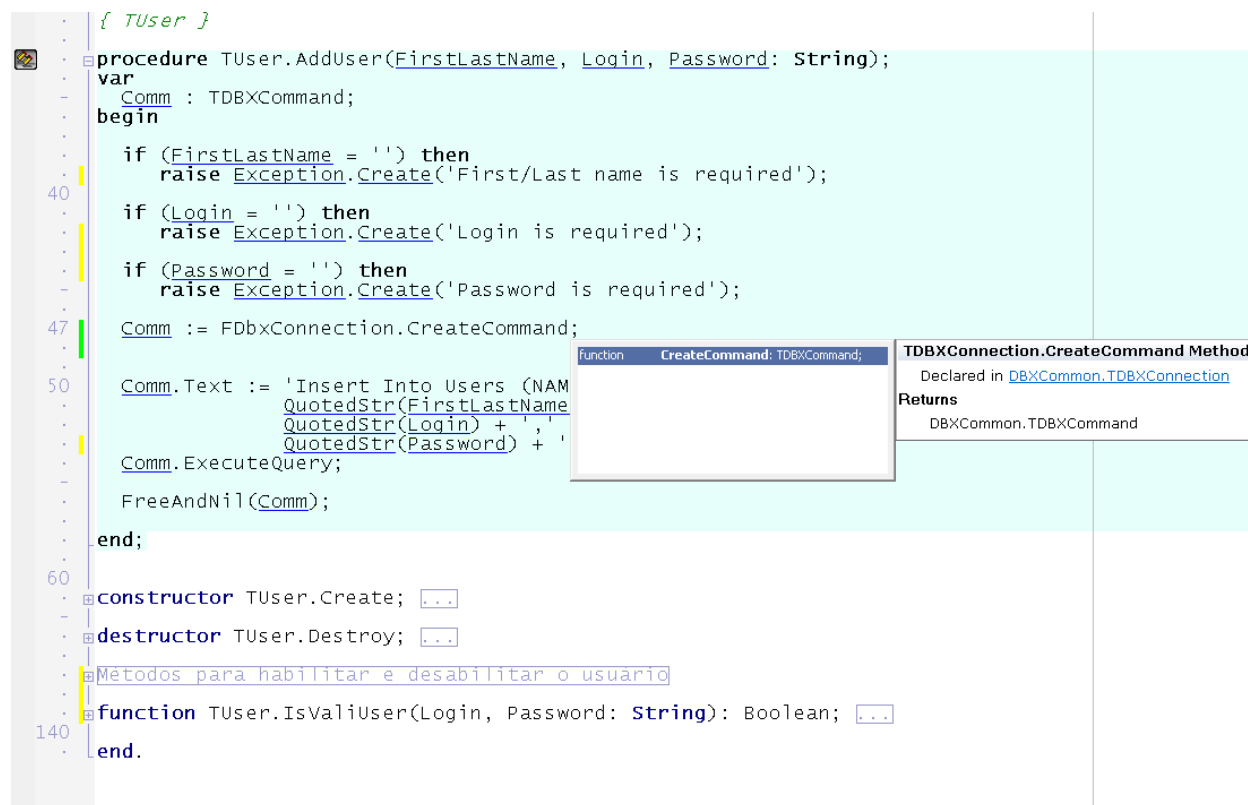


Figure 6 Live Templates

Alongside the code block you see yellow and green marks. The yellow ones are shown for lines that have changed since the last Save. Green marks, in turn, indicate lines that were recently changed and saved.

You also see the smart code line numbering. And you're able to either expand or collapse a method or a class right within the block.

Think about a unit with tens of methods. You hope one day you'll have enough time to stop and set it in order, but never really finds time to do so. The code above holds a region called "Methods for enabling/blocking user access". This region has two methods for either enabling or blocking users. You can't see those methods, unless their region is expanded. A question remains unanswered: wouldn't it be easier to organize and visualize the code with the assistance of such features?

Help from within the code (Help Insight): Press F1 in order to see the documentation of a method, type, class, etc. As you can see, the code above displays the CreateCommand method help info. The same thing happens with any method, type or class, provided it has a description available.

Finding references for a method, class, variable, or any other specific item: Imagine your code holds a class named TCGC, which you want to rename to TCNPJ. Assuming you're a careful developer, tell me where the TCGC class is referenced in the project. Find..Replace won't work this time. Instead, try pressing Shift + CTRL + Enter over class TCGC. The IDE then looks for all references in the project, as seen below.

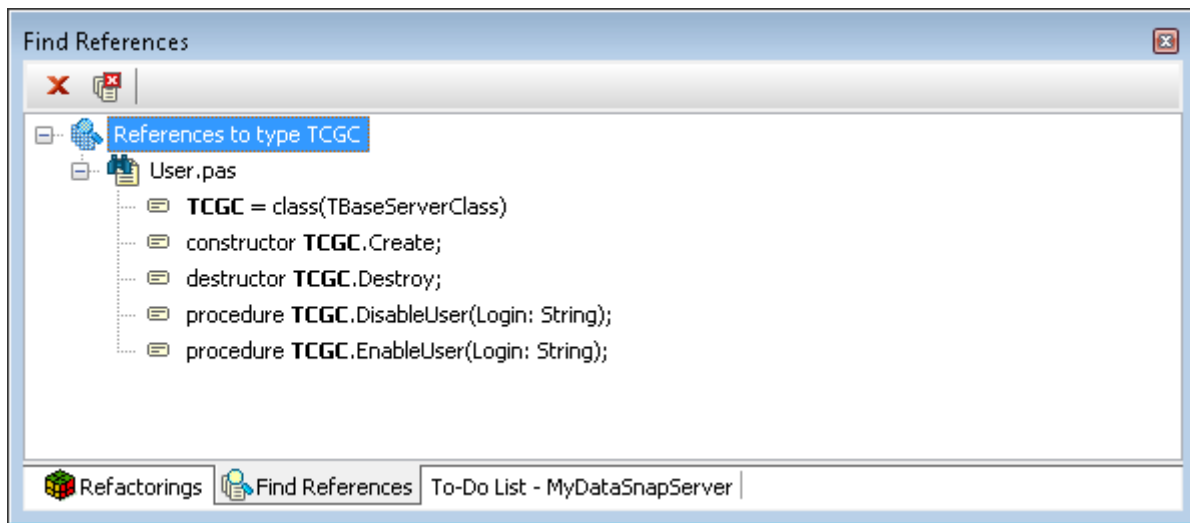


Figure 7 Find References with Live Template

If you're now wondering how to rename all classes to TCNPJ, wait until we discuss Refactorings. Another useful feature is named Surround. It works basically by allowing you to add begin/end, if/begin/end, try/finally, try/except, etc., to a block of code.

CHANGE HISTORY

Files are locally versioned whenever they're changed, even in the absence of version control, thus allowing you to make comparisons between them.

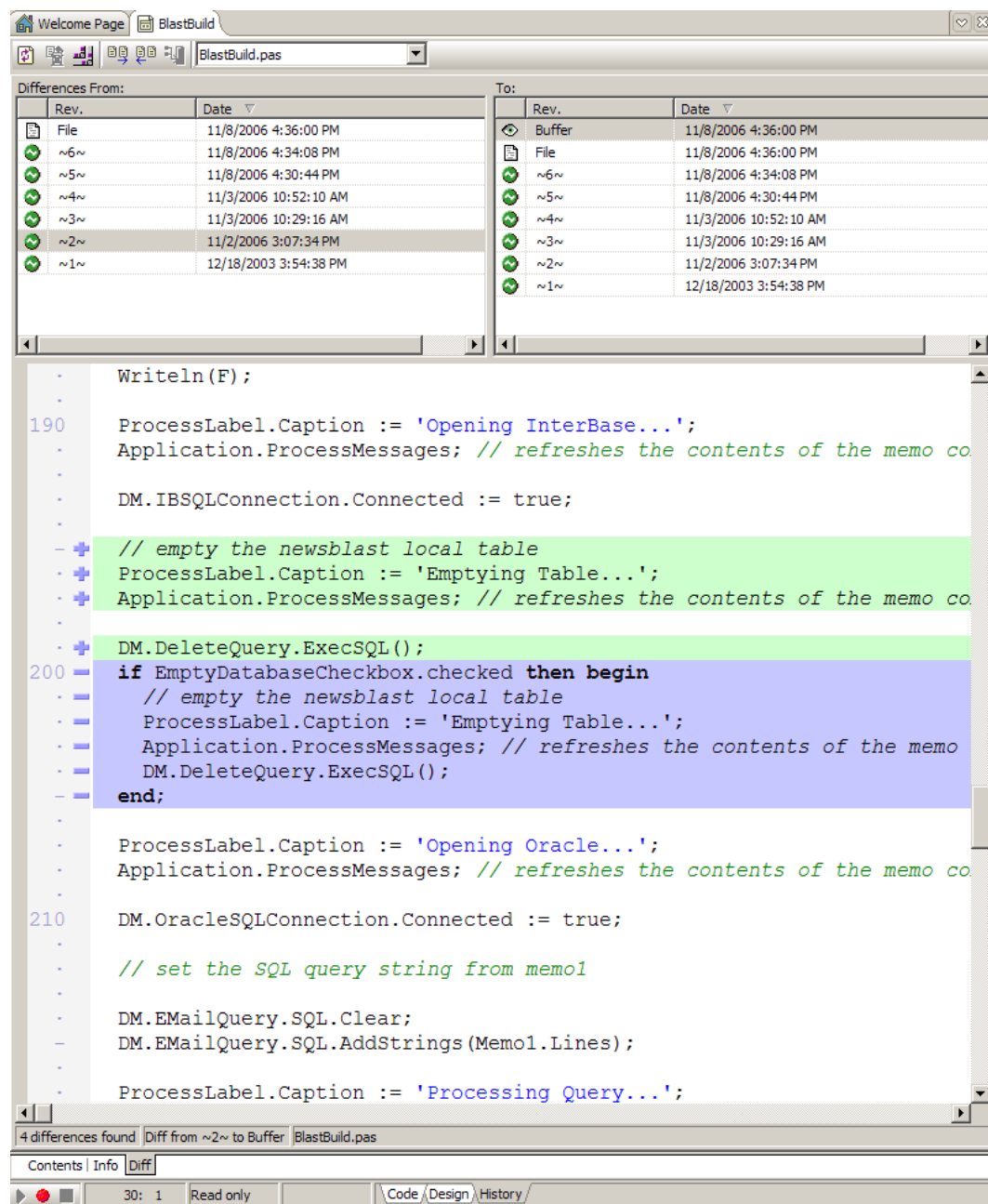


Figure 8 Change History

REFACTORINGS

Delphi 7 users will surely love to try this resource. Refactoring is a technique you can use to restructure and modify your existing code in such a way that the intended behavior of your code stays the same. Refactoring allows you to streamline, simplify, and improve both performance and readability of your application code.

Delphi includes a refactoring engine that evaluates and executes the refactoring operation. The engine also displays a preview of the changes that will occur in a refactoring pane that appears at the bottom of the Code Editor. The potential refactoring operations are displayed as tree nodes, which can be expanded to show additional items that might be affected by the refactoring, if they exist. Warnings and errors also appear in this pane. You can access the refactoring tools from the Main menu and from context-sensitive drop down menus.

UNIT TESTING

Delphi integrates an open-source testing framework, DUnit, for developing and running automated test cases for your applications. This framework simplifies the process of developing tests for classes and methods in your application. Using unit testing in combination with refactoring can improve your application stability. Running a standard set of tests every time a small change is made throughout the code makes it more likely that you will catch any problems early in the development cycle.

DATA EXPLORER

It's now easier to retrieve data from databases. Using the Data Explorer along with drag-and-drop capabilities you can access tables, views, stored procedures and other database items. Besides that, you can also search for data using SQL.

Connections are established through dbExpress. This means Data Explorer supports every database dbExpress supports.

Each connection is assigned an alias which is saved in the dbxconnections.ini file (dbExpress' configuration file). Aliases are treated as shared information, thus facilitating the use of Data Explorer.

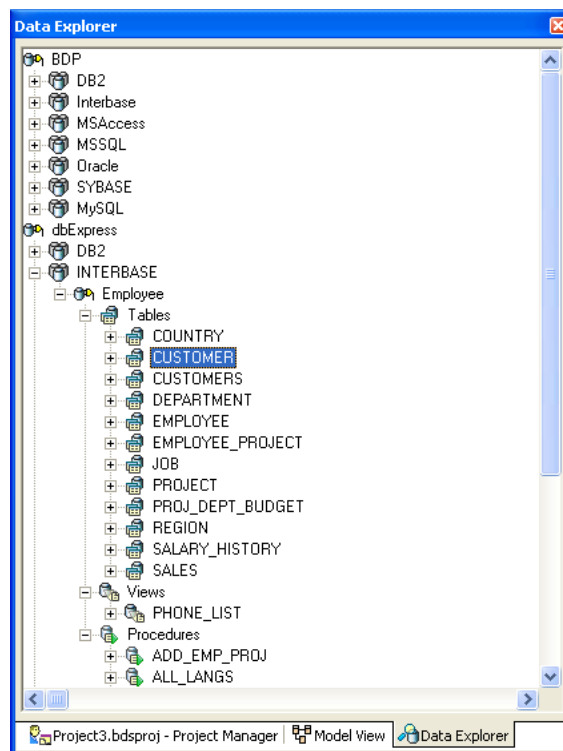


Figure 9 Data Explorer

SQL WINDOW - QUERY BUILDER

The Data Explorer allows developers to easily build complex SQL queries via an intuitive visual query building interface.

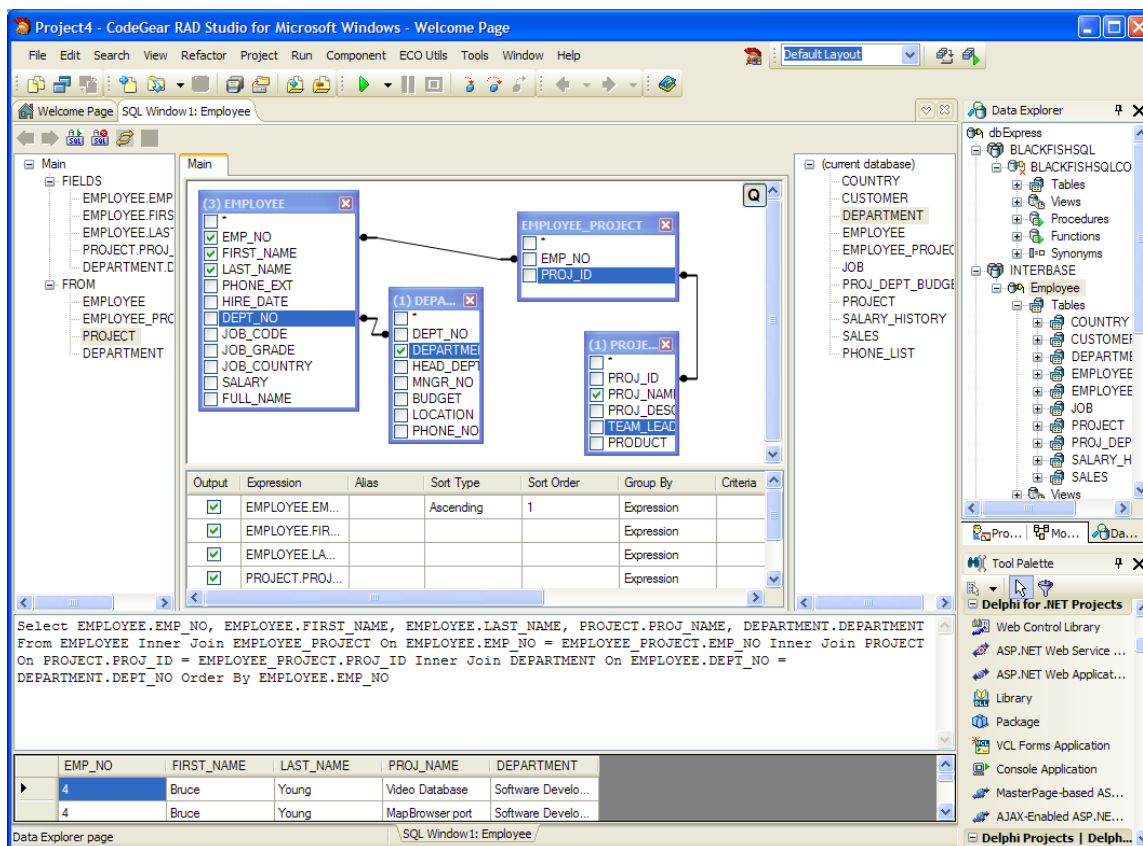


Figure 10 SQL Window

DEBUGGER

The debugger now comes with the “Thread View and Wait Chain Traversal” feature, available only for Windows Vista. This resource helps you locate deadlocks and thread contentions.

During the debug process you need to visualize the content of variables. We most often use the Watch List for that, but the higher the number of items added the more confusing the visualization gets. Variables added to Watch List can now be grouped based on a name provided by the developer. The groups are then represented as tabs in the Watch List.

After the debug is finished, all units that are opened during the process are automatically closed; only units that were open before the debug process was started are kept.

There are lots of new things regarding the visualization and usability of local variable window, call stack and others. There is also a new tree view for the content of objects that are being debugged, shown in Figure 11:

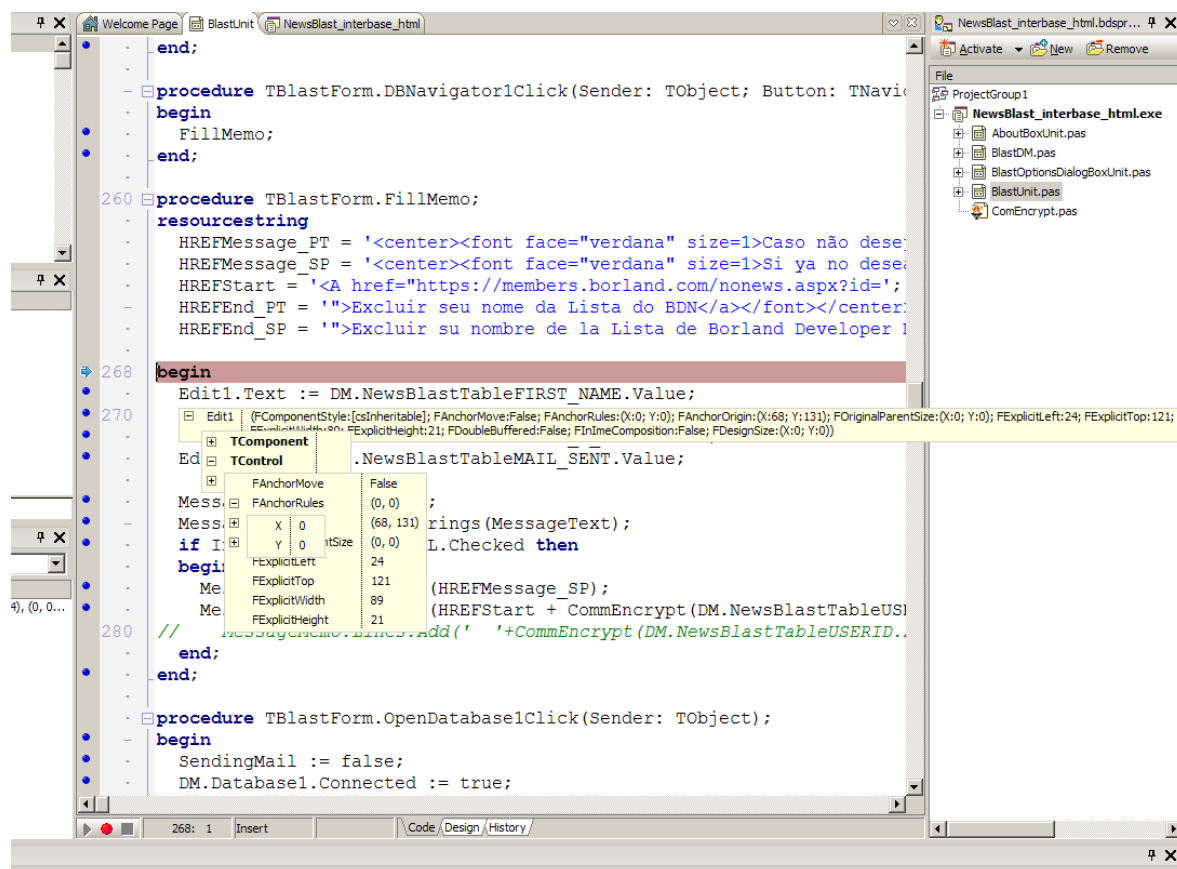


Figure 11 Debugger

WHAT'S NEW IN THE VCL AND RTL

Since Delphi 7, the VCL and RTL have been continuously enhanced. Besides complete support to Windows Vista, Unicode..., new components have been added, while existing components have been improved by the addition of new functionalities.

All these add up to better component usability, the creation of rich interfaces and the ability to use Windows Vista's new functionalities. In this chapter we focus on what's new related to the existing components and the changes in RTL classes.

WINDOWS VISTA

Applications compiled in Delphi 2007 (or later) are 100% compatible with Windows Vista. The VCL has been updated to support the new characteristics of this OS, while new components were also added: TFileOpenDialog, TFileSaveDialog and TTaskDialog.

New classes have also been created; as, for instance:

- TCustomFileDialog
- TCustomFileOpenDialog
- TCustomFileSaveDialog
- TCustomTaskDialog
- TFavoriteLinkItem
- TFavoriteLinkItems
- TFavoriteLinkItemsEnumerator
- TFileTypeltem
- TFileTypeltems
- TTaskDialogBaseButtonItem
- TTaskDialogButtonItem
- TTaskDialogButtons
- TTaskDialogButtonsEnumerator
- TTaskDialogProgressBar
- TTaskDialogRadioButtonItem

Dialog box components are now displayed in a Vista-like fashion. You are now probably wondering what happens to applications when you run them on Windows XP. There's no reason to be concerned about it. VCL recognizes the OS, using its specific resources and the appropriate interface.

The TaskMessageDlg function was designed to support Windows Vista. It has the same functionality seen in MessageDlg, except for a few additional parameters that support Windows Vista's characteristics. When you run your application on Windows XP, MessageDlg is automatically executed. VCL is there to ensure it.

The UseLatestCommonDialogs global variable defines that all dialog components (TOpenDialog, TSaveDialog, TOpenPictureDialog and TSavePictureDialog) should follow Windows Vista's design whenever it receives a TRUE statement.

As an example, this is how Open and Save Dialogs would look like in Windows Vista:

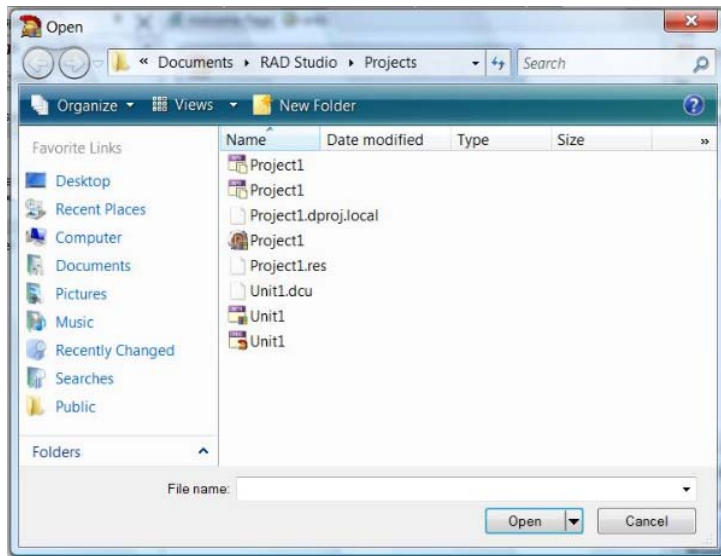


Figure 12 Open Dialog in Windows Vista

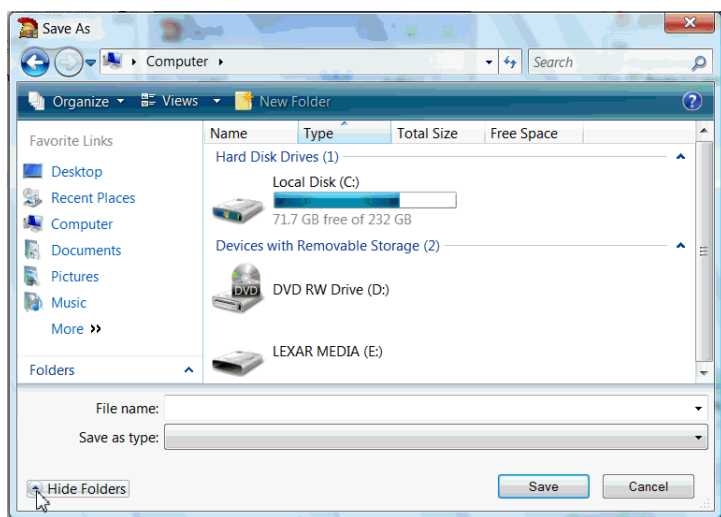


Figure 13 Save Dialog in Windows Vista

The units below were enhanced to support Window's new APIs.

- UxThemes – new
- DwnApi – new
- ActiveX – updated
- Windows – updated
- Messages – updated
- CommCtrl – updated
- ShlObj - updated

TACTIONMANAGER

New properties - DisabledImages, LargeImages e LargeDisabledImages - that allow you to define large and disabled images, based on the TImageList component.

PNG SUPPORT

The Image component supports the PNG format natively.

TBITMAP

Support to 32-bit alpha bitmaps, along with the addition of the AlphaFormat property.

TBUTTONGROUP

This component allows you to group many different buttons in a panel.

TBUTTON

Windows Vista now has two new button styles, both supported by Delphi by means of its TButton class.

CommandLink is a button with a different, friendlier design. Using it you can add a more detailed description of the button functionality.

SplitButton opens a list of options when clicked. This list is presented as a PopMenu. You can also assign images to the items.

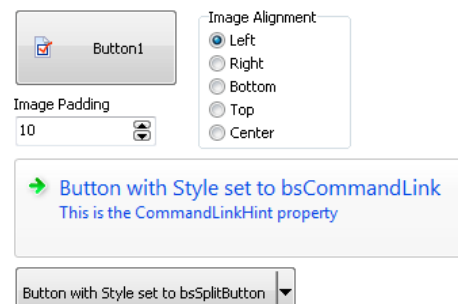


Figure 14 New Button Styles

TListView AND TTreeView

TListView has been improved and now allows you to define basic and advanced groups. The advanced group support enables deeper customization of groups (requires Vista), allowing you to define images for each of them.

TTreeView allows enabling/disabling nodes and images for expanded items.

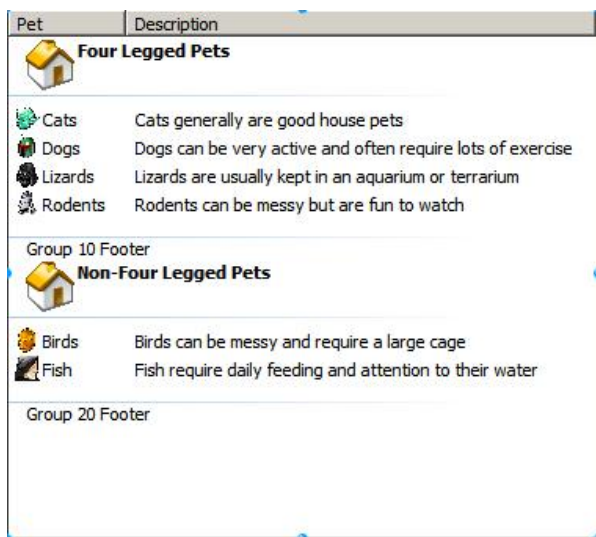


Figure 15 TListView

TButtonEdit

The new TButtonEdit component allows you to add images within the Edit field. Images can be placed either at the right or left side. You can also use events to control image clicking – by means of the onLeftClick and onRightClick events.

TBALLOONHINTS

The new hints now come in Windows Vista style and allow the addition of titles, descriptions and images, which will make your user notifications much friendlier.

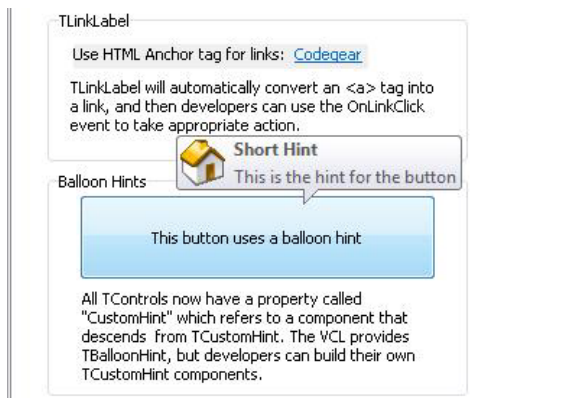


Figure 16 Balloon Hints

TCATEGORYPANELGROUP

This new component is very useful. It works like an Outlook bar, onto which you can add many different panels. Each of these panels can contain any VCL component. You can define a title, an image, the alignment and an icon for each of the panels, being able to expand and collapse them.

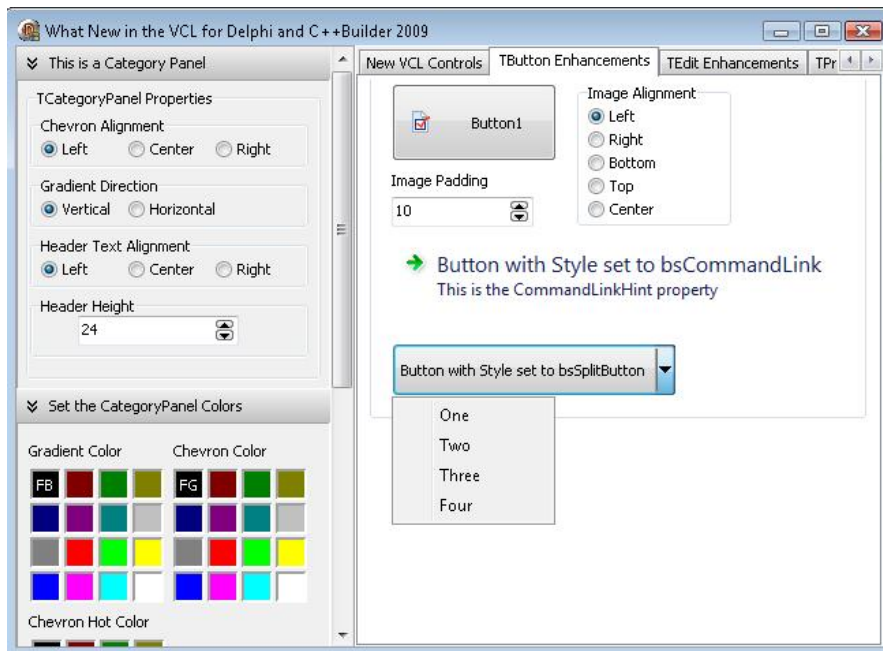


Figure 17 Panel Group

TLINKLABEL

LinkLabel is a kind of label that allows you to add HTML tags that affect the appearance of the component.

TPOPUPACTIONBAR

Now supports ActionBar styles.



THEADERCONTROL AND THEADERSECTION

New checkbox support.

NEW PANELS

The traditional TPanel is a visual container for other components. Within TPanel you are able to accommodate visual controls wherever you want. In other words, it works with absolute positioning (the Top and Left coordinates of the control refer to the panel's upper left corner).

Inspired by similar Java concepts (namely, the Layout Manager, which defines how controls are distributed within the container) we can say we now have three kinds of layout managers:

- **TPanel** : absolute type, or XY. Components are placed in fixed, precise positions.
- **TFlowPanel** : components are placed in a sequence, according to a given order (similar to an HTML page, neither using tables nor CSS stylesheets).
 - The flow is determined by the FlowStyle property, which accepts the options you see below. In order to understand the naming convention, keep in mind that components are accommodated according to the direction defined by the first pair (e.g., LeftRight). When there's no space left in the panel, the direction is redefined by the second pair (e.g., TopBottom):
 - fsLeftRightTopBottom: left to right, top down (default)
 - fsRightLeftTopBottom: right to left, top down
 - fsLeftRightBottomTop: left to right, bottom up
 - fsRightLeftBottomTop: right to left, bottom up
 - fsTopBottomLeftRight: top down, left to right
 - fsBottomTopLeftRight: bottom up, left to right
 - fsTopBottomRightLeft: top down, right to left
 - fsBottomTopRightLeft: bottom up, right to left
 - Another relevant property is AutoWrap. When True, indicates the flow will be “broken” towards the other direction in case the panel runs out of space. When False, components outside the boundaries of the panel will not be visible.
 - An interesting use for this kind of panel is in the automatic generation of forms, with fields being dynamically defined either in a database or a file. This way we do not need to be concerned about positioning each of the fields.

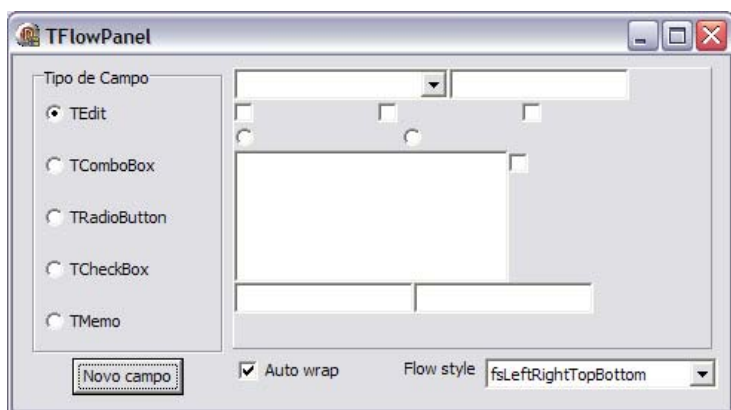



Figure 18 Form Auto Generation

- **TGridPanel** : the panel is partitioned by lines and columns, with each of the resulting cells holding a component (similar to the use of HTML tables).
 - Components are arranged according to the order of the lines (top down) and, within each line, column-wise (left to right).
 - The number of lines is determined by the RowCollection property, which can contain various objects of the TRowItem class. Each item has two properties:
 - SizeStyle: determines the standard by which line height is specified in the Value property:
 - ssAbsolute: number of pixels
 - ssAuto: the number is disregarded, with line height being automatically calculated
 - ssPercent: percentage in comparison to the panel height
 - Value: a number expressing the height, according to the SizeStyle property.
 - Similarly, the number of column is determined by the ColumnCollection property, which contains objects of the TColumnItem class. TColumnItem has the same properties TRowItem does, differing only in that they relate to the column width, not the row's.
 - The ExpandStyle property determines an action for whenever someone tries to add a component into a panel that is out of free cells. Its possible values are:
 - emAddRows: a new line is added to the panel to accommodate the components
 - emAddColumns: new columns are added to accommodate the components
 - emFixedSize: an exception is raised when there's no more free space to accommodate new components.

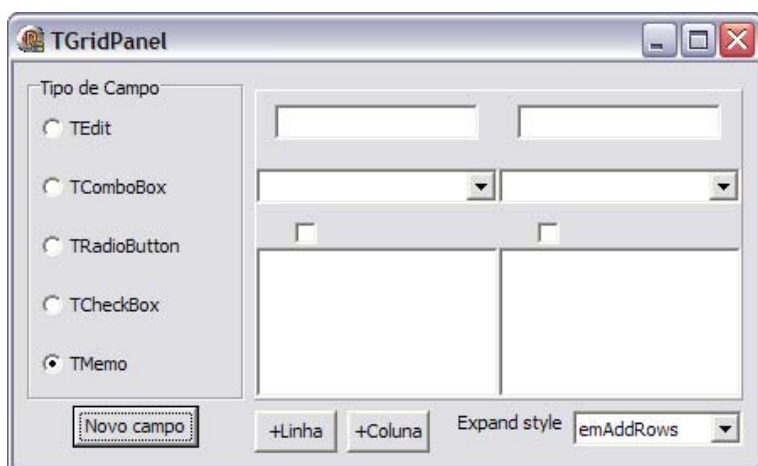


Figure 19 Expand Property Style

TCategoryButtons

This component allows you to create buttons and group them into categories, similar to what's seen in an Outlook bar. It helps you refine the design of your applications.

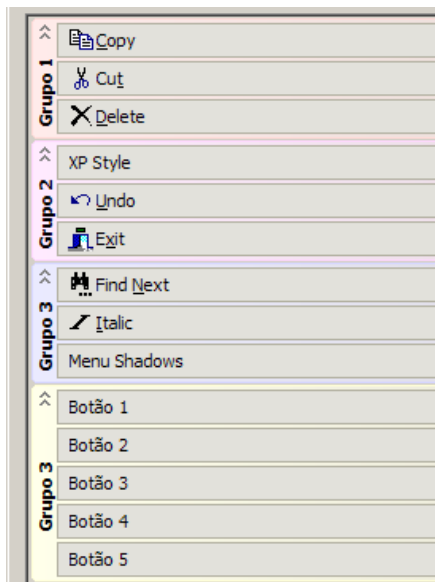


Figure 20 TCategoryButtons

TTrayIcon

Think of those icons you see beside the taskbar clock, in the Windows tray. What if you could place your application right there too? It's now more than simple to do so. All you have to do is place a TTrayIcon component (Win32 tab) in the main form. Set just a few properties and you're done:

- **Icon:** Stores the icon that is displayed in the tray. You can use your application's icon or an icon that describes a status or situation. This icon can be changed at anytime.
- **Icons:** references a TImageList containing a bunch of bitmaps or icons which will be used in the animation.
- **Animate:** When True, keeps swapping the icons in the Icons list. The index of the icon that is being currently displayed can be either retrieved or changed using the IconIndex property.
- **AnimateInterval:** millisecond interval of the icon swapping process. The OnAnimate event is generated after each iteration, allowing you to define an action to be taken.
- **BalloonTitle** and **BalloonHint:** Title and text for the balloon, displayed by the ShowBalloonHint method. The balloon can be closed by a simple click (either on it or over the dialog's X). However, it goes away automatically after the interval predefined in the BalloonTimeout property (milliseconds).
- **PopupMenu:** it's common to associate a popup menu to the application's icon, enabling users to quickly access the most commonly used commands. All it takes for you to do so is reference the menu in this property. To access it, left-click the icon.

You can hide the main form (using methods `Hide` or `Visible := False`) without halting the application. In this case it's essential to provide the tray icon with a menu or event (`OnClick`, for instance) to ensure the control is passed back to you right after.

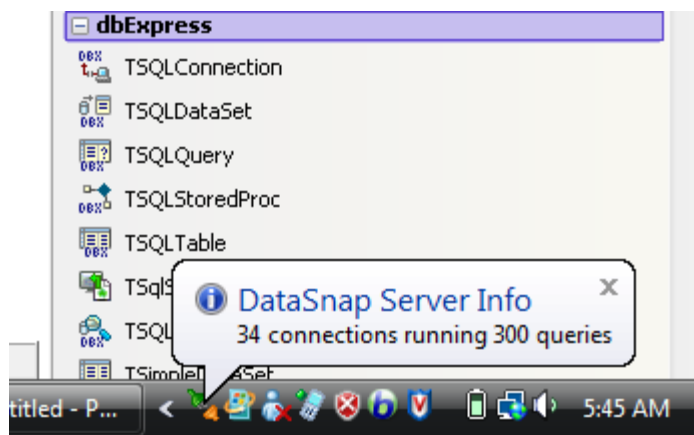


Figure 21 Tray Icon

INTELLIMOUSE SUPPORT

IntelliMouse is how we call the support for mouse wheel scrolling in your application. Support for this technology was first introduced to VCL in Delphi 2006. In order to use it, simply declare the `IMOUSE` unit in your application.

TOBJECT

The father of all components in Delphi has also been enhanced:

- New methods
 - class function `UnitName` : string
 - function `Equals(Obj : TObject) : Boolean`; virtual
 - function `GetHashCode` : Integer; virtual
 - function `ToString` ; String; virtual
- A few additional Overloads for the following methods:
 - `MethodAddress`
 - `FieldAddress`
- The type of return of the functions below has changed from `ShortString` to `String` in order to support Unicode
 - `ClassName`
 - `MethodName`

Other components - `TPanel`, `TProgressBar`, `TTrayIcon`, `TScreen`, `TRadioGroup` – also provide a lot of improvements.

NEW MEMORY MANAGER AND NEW RTL FUNCTIONS

Many RTL functions have been updated to improve the performance of your applications. FASTMM is the most relevant of such improvements. A new memory manager aimed at Win32 applications, FASTMM enables applications to have better performance by performing the compilation in Delphi 2006, and identifies memory leaks just declaring

```
ReportMemoryLikeonShutdown := True
```

It can't be overemphasized that simply by compiling your applications in Delphi 2006 or later, you experience performance gains, while also being able to detect memory leakages.

VCL – MARGINS AND PADDING

Three seems to be a magic number these days... Besides three new visual components, the VCL has also been improved with three additional classes:

- TMargins
- TPadding
- TCustomTransparentControl

The TControl class now has an additional property (Margins, of TMargins class). TMargins is used in the Margins property of TControl and its descendants. TMargins helps define the relative position between components on a form, and between the edges of the form and the component. For example, when you set the left margin for a component to 10 pixels, the component will not come closer than 10 pixels to the edge of the container, or to another component on the left edge. The number of pixels by which two components are separated is the sum of the pixels of both components.

The TWinControl class adds an extra property – Padding, of TPadding class: a TMargins descendent.

Padding adds space along the edge the control. Child controls that are aligned to the parent are positioned inside the control according to this spacing. Padding does not affect child controls which are not aligned to the parent control, nor does it affect the size of the ClientArea.

Padding is the opposite of Margins. Margins affects the positioning of the control itself inside the parent control, but Padding affects how all aligned child controls are positioned with respect to the parent control.

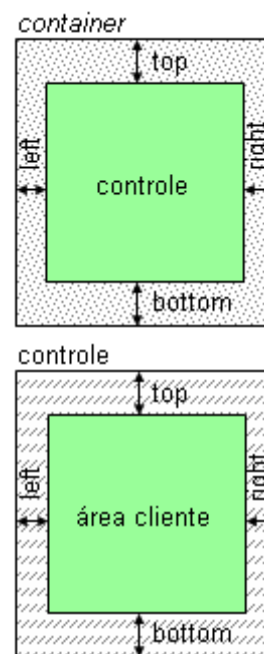


Figure 22 Margins and Padding

TRANSPARENT CONTROLS

The new `TCustomTransparentControl` class can be used as a basis for components that need to be there while pretending they're not. Oh, oh. Well... think of it as the glass of a window or door. You know it's there, although you're unable to visually perceive it. To see the difference, do this test: create a new VCL application (File | New | VCL Forms Application – Delphi for Win32). Place two `TButtons` and two `TLabels` in it, see Figure 22.

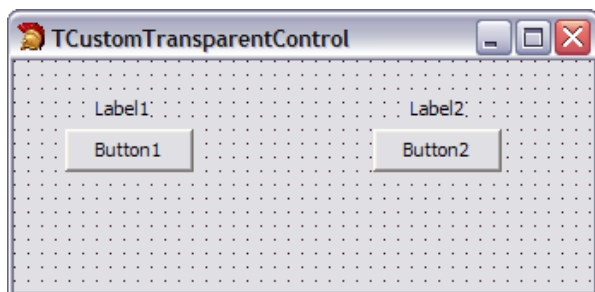


Figure 23 Transparent Controls

Set both buttons' `Top` property to 40; Button 1 and 2's `Left` properties to 30 and 210, respectively. When you're done, type the code as below. Note that the form implements the events `OnCreate`, `OnDestroy` and `OnClick`, with both buttons sharing the same `OnClick` event.

I've created a `TCustomTransparentControl` descendent – `TTransparentControl` – and a `TCustomControl` descendent - `TOpaqueControl`. They are dynamically created by the form's `OnCreate` event, being positioned right above the other existing controls. I've also added an `OnClick` event to our customized controls for you to observe their behavior. The result can be seen in Figure 24.

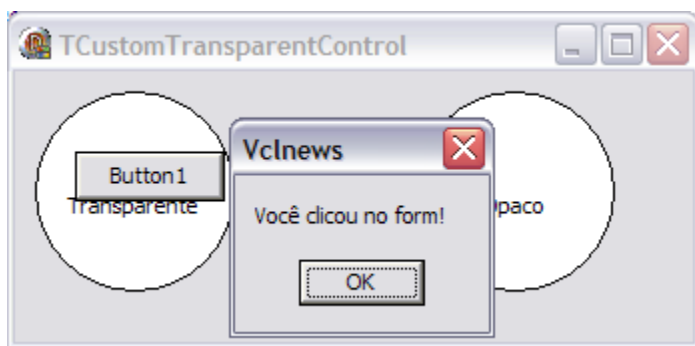


Figure 24 Transparent Control Example

```
type
  TTransparentControl = class(TCustomTransparentControl)
  protected
```

```
    procedure Paint; override;
end;

TOpaqueControl = class(TCustomControl)
protected
    procedure Paint; override;
end;

TfCustomTransparentControl = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure FormClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
private
    InvCon : TTransparentControl;
    VisCon : TOpaqueControl;
    procedure ControlClick(Sender: TObject);
end;

var
    fCustomTransparentControl: TfCustomTransparentControl;

implementation

{$R *.dfm}

{ TTransparentControl }
procedure TTransparentControl.Paint;
const txt = 'Transparent';
begin
    Canvas.Ellipse(0,0,Width,Height);
    Canvas.TextOut((Width - Canvas.TextWidth(txt)) div 2, Height div 2, txt);
end;

{ TOpaqueControl }
procedure TOpaqueControl.Paint;
const txt = 'Opaque';
begin
    Canvas.Ellipse(0,0,Width,Height);
    Canvas.TextOut((Width - Canvas.TextWidth(txt)) div 2, Height div 2, txt);
end;

{ Form }
procedure TfCustomTransparentControl.FormCreate(Sender: TObject);
begin
    InvCon := TTransparentControl.Create(Self);
    InvCon.Parent := Self;
    InvCon.SetBounds(10,10,100,100);
    InvCon.OnClick := ControlClick;

    VisCon := TOpaqueControl.Create(Self);
    VisCon.Parent := Self;
    VisCon.SetBounds(200,10,100,100);
    VisCon.OnClick := ControlClick;
```



```
end;

procedure TfCustomTransparentControl.FormDestroy(Sender: TObject);
begin
    InvCon.Free;
    VisCon.Free;
end;
procedure TfCustomTransparentControl.Button1Click(Sender: TObject);
begin
    ShowMessage('You have clicked on ' + (Sender as TButton).Caption);
end;

procedure TfCustomTransparentControl.ControlClick(Sender: TObject);
begin
    ShowMessage('You have clicked on control ' + (Sender as
TControl).ClassName);
end;

procedure TfCustomTransparentControl.FormClick(Sender: TObject);
begin
    ShowMessage('Form clicked!');
end;
```

RIBBON CONTROLS

Most of you are already familiarized with the new Ribbon interface (used in Office 2007). Such interfaces are designed to facilitate user access to your application's menu options.

The VCL comes with Ribbon Controls, a group of components that allows you to create Ribbon-style Delphi interfaces.

Ribbon Controls integrate with Action Manager, meaning applications with Actions and the traditional menus can be easily migrated to Ribbon.

The architecture behind Ribbon Controls is very simple. From a Ribbon control you are able to add a Tab (which contains groups). Each of these groups contains buttons with customized appearance. Additionally, the Ribbon control includes the *Quick Access Toolbar* and the *Application Menu*.

See an example application with Ribbon Controls in Figure 24.

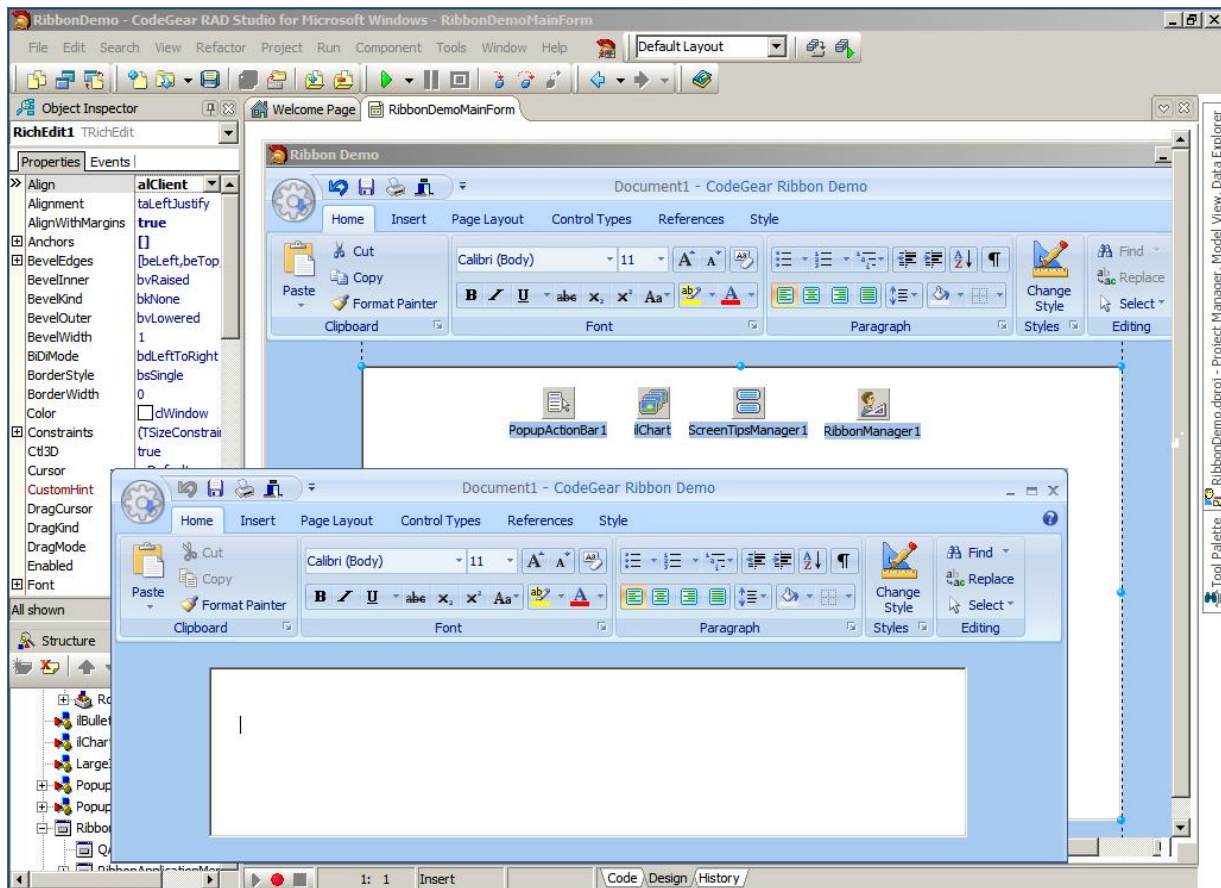


Figure 25 Ribbon Controls

100% UNICODE

One of the greatest challenges faced by our R&D team was that of incorporating Unicode support throughout the entire VCL – and then the IDE – because Delphi is developed in Delphi.

Throughout this version's development process we've had meetings with a number of component development companies. Allen Bauer, our Chief Scientist for Delphi, posted comments on the Unicode support in his blog. Initiatives like those proved essential in allowing third-party components to be quickly available to Delphi 2009, as well as in keeping developers updated about how to work with Unicode.

Unicode is a standard that allows computers to consistently represent and handle text from any existing system of writing.

- *The Unicode Standard: Version 5.0. 5. ed. Addison-Wesley Professional, 2006. 1472 p*

Many character sets – like Chinese, Japanese, and Russian, along with others of Asian background – are represented by means of Unicode. The most commonly used encodings are UTF (Unicode Transform Format) and UCS (Universal Character Set). To learn more about Unicode, visit: <http://en.wikipedia.org/wiki/Unicode>.

The result of all this is a Delphi that is 100% Unicode, by all means. You're now probably wondering if the migration is that easy. Definitely yes, once many things are handled by the VCL and the compiler.

One of the most relevant changes in this version, String types were previously based on the ANSI standard. Now they're based on UNICODE. The AnsiString and WideString types still work the same way, except regarding their data size in bytes.

Unicode changes, in short:

- String maps UnicodeString, no longer AnsiString
- Char now maps WideChar (2 bytes, not 1 byte) being a UTF-16 character
- PChar maps PWideChar
- AnsiString maps the old String type

No changes were applied to:

- AnsiString
- WideString
- AnsiChar, PAnsiChar
- Short String contains AnsiChar elements
- Implicit conversions still work.
- The user's active code page controls the mode (ANSI vs. Unicode), so that ANSI strings are still supported.

Operations that do not depend on character size:

- String concatenation
- Standard String functions. E.g., Length, Copy, Pos, and so on.
- Operators. E.g., <string> < comparison> <string>, CompareStr(), CompareText(), etc.
- FillChar (<struct or memory>)
- Windows API

Operations involving the character size (measured in bytes) may require a few changes. Nothing too complicated, but here's a tip: pay special attention to code in which you:

1. Assume Sizeof (Char) is 1.
2. Assume the size of a string equals the number of bytes in the string.
3. Handle String or PChars directly.
4. Saves or reads a string from/to a file.

Items 1 and 2 do not apply to Unicode, because in Unicode the Sizeof (Char) is 2 bytes and the size of a string is twice as big as the number of bytes. Besides, the code that reads and saves files must understand the right number of bytes to perform those operations, for a character is no longer represented as 1 byte.

As you can see, migrating is very easy. The benefit of having Unicode support is that of allowing Delphi developers to distribute their applications worldwide. Brazil is currently one of the most relevant software developers in the global market. Many Brazilian companies distribute their applications to China, Japan, Russia and other countries where Unicode is crucial.

In 2007 the Russian government acquired 1 million Delphi licenses to be used in teaching primary and high school students to develop software with Delphi. Therefore, Unicode support is vital for that country.

NEW LANGUAGE AND COMPILER RESOURCES

INLINE DIRECTIVE

The Delphi compiler allows functions and procedures to be tagged with the inline directive to improve performance. If the function or procedure meets certain criteria, the compiler will insert code directly, rather than generating a call. Inlining is a performance optimization that can result in faster code, but at the expense of space. Inlining always causes the compiler to produce a larger binary file. The inline directive is used in function and procedure declarations and definitions, like other directives.

```
program InlineDemo;
{$APPTYPE CONSOLE}

uses
  MMSystem;

function Max(const X,Y,Z: Integer): Integer;inline
begin
  if X > Y then
```

```
        if X > Z then Result := X
            else Result := Z
    else
        if Y > Z then Result := Y
            else Result := Z
    end;

const
    Times = 10000000; // 10 million
var
    A,B,C,D: Integer;
    Start: LongInt;
    i: Integer;

begin
    Random; // 0
    A := Random(4242);
    B := Random(4242);
    C := Random(4242);
    Start := TimeGetTime;
    for i:=1 to Times do
        D := Max(A,B,C);
    Start := TimeGetTime-Start;
    writeln(A, ', ', B, ', ', C, ': ', D);
    writeln('Calling Max ', Times, ' times took ', Start, ' milliseconds. ');
    readln

end.
```

When the above code is executed, the Max method is called 10 million times. The numbers below vary depending on your machine. Using a Pentium M 1.8GHz with 2GB RAM we've obtained the following results:

With inline	Without inline
25 milliseconds	68 milliseconds

The inline directive is a suggestion to the compiler. There is no guarantee the compiler will inline a particular routine, as there are a number of circumstances where inlining cannot be done. The following list shows the conditions under which inlining does or does not occur:

- Inlining will not occur on any form of late-bound method. This includes virtual, dynamic, and message methods.
- Routines containing assembly code will not be inlined.
- Constructors and destructors will not be inlined.
- The main program block, unit initialization, and unit finalization blocks cannot be inlined.
- Routines that are not defined before use cannot be inlined.
- Routines that take open array parameters cannot be inlined.
- Code can be inlined within packages, however, inlining never occurs across package boundaries.
- No inlining will be done between units that are circularly dependent. This included indirect circular dependencies, for example, unit A uses unit B, and unit B uses unit C which in turn

uses unit A. In this example, when compiling unit A, no code from unit B or unit C will be inlined in unit A.

- The compiler can inline code when a unit is in a circular dependency, as long as the code to be inlined comes from a unit outside the circular relationship. In the above example, if unit A also used unit D, code from unit D could be inlined in A, since it is not involved in the circular dependency.
- If a routine is defined in the interface section and it accesses symbols defined in the implementation section, that routine cannot be inlined.
- If a routine marked with inline uses external symbols from other units, all of those units must be listed in the uses statement, otherwise the routine cannot be inlined.
- Procedures and functions used in conditional expressions in while-do and repeat-until statements cannot be expanded inline.
- Within a unit, the body for an inline function should be defined before calls to the function are made. Otherwise, the body of the function, which is not known to the compiler when it reaches the call site, cannot be expanded inline.

If you modify the implementation of an inlined routine, you will cause all units that use that function to be recompiled. This is different from traditional rebuild rules, where rebuilds were triggered only by changes in the interface section of a unit.

OPERATOR OVERLOADING

Delphi allows certain functions, or "operators" to be overloaded within record declarations. The name of the operator function maps to a symbolic representation in source code. For example, the Add operator maps to the + symbol. The compiler generates a call to the appropriate overload, matching the context (i.e. the return type, and type of parameters used in the call), to the signature of the operator function. The following table shows the Delphi operators that can be overloaded:

```
type
  TMyClass = class
    class operator Adicionar(a, b: TMyClass): TMyClass; // Add two classes
  of TMyClass type
    class operator Subtrair(a, b: TMyClass): TMyclass; // Subtraction of
the TMyClass type
    class operator Implicit(a: Integer): TMyClass;      // Implicit
conversion from integer to the class of TMyClass type
    class operator Implicit(a: TMyClass): Integer;      // Implicit
conversion from the TmyClass class into an integer
    class operator Explicit(a: Double): TMyClass;      // Explicit
conversion from a Double to a TMyClass class
  end;

// Method implementation example. Add
class operator TMyClass.Add(a, b: TMyClass): TMyClass;
begin
  // ...
end;

var
  x, y: TMyClass;
```

```
begin
  x := 12;      // Implicit conversion of an Integer, executes the Implicit
method
  y := x + x;   // Executes TMyClass.Add(a, b: TMyClass): TMyClass
  b := b + 100; // Executes TMyClass.Add(b, TMyClass.Implicit(100))
end;
```

No operators other than those listed in the table may be defined on a class or record.

Overloaded operator methods cannot be referred to by name in source code. To access a specific operator method of a specific class or record, you must use explicit typecasts on all of the operands. Operator identifiers are not included in the class or record's list of members.

No assumptions are made regarding the distributive or commutative properties of the operation. For binary operators, the first parameter is always the left operand, and the second parameter is always the right operand. Association is assumed to be left-to-right in the absence of explicit parentheses.

Resolution of operator methods is done over the union of accessible operators of the types used in the operation (note this includes inherited operators). For an operation involving two different types A and B, if type A has an implicit conversion to B, and B has an implicit conversion to A, an ambiguity will occur. Implicit conversions should be provided only where absolutely necessary, and reflexivity should be avoided. It is best to let type B implicitly convert itself to type A, and let type A have no knowledge of type B (or vice versa).

As a general rule, operators should not modify their operands. Instead, return a new value, constructed by performing the operation on the parameters.

Overloaded operators are used most often in records (i.e. value types).

CLASS HELPERS

A class helper is a type that - when associated with another class - introduces additional method names and properties which may be used in the context of the associated class (or its descendants). Class helpers are a way to extend a class without using inheritance. A class helper simply introduces a wider scope for the compiler to use when resolving identifiers. When you declare a class helper, you state the helper name, and the name of the class you are going to extend with the helper. You can use the class helper any place where you can legally use the extended class. The compiler's resolution scope then becomes the original class, plus the class helper.

Class helpers provide a way to extend a class, but they should not be viewed as a design tool to be used when developing new code. They should be used solely for their intended purpose, which is language and platform RTL binding. You can see an example below.

```
type
  TMyClass = class
    procedure MyProc;
    function  MyFunc: Integer;
  end;
  ...
```

```
procedure TMyClass.MyProc;
var X: Integer;
begin
    X := MyFunc;
end;

function TMyClass.MyFunc: Integer;
begin
    ...
end;

type
    TMyClassHelper = class helper for TMyClass
        procedure HelloWorld;
        function MyFunc: Integer;
    end;

    procedure TMyClassHelper.HelloWorld;
    begin
        writeln(Self.ClassName); // Self refers to the TMyClass class, not to
TMyClassHelper
    end;

    function TMyClassHelper.MyFunc: Integer;
    begin
        ...
    end;

var
    X: TMyClass;
begin
    X := TMyClass.Create;
    X.MyProc;      // Executes TMyClass.MyProc
    X.HelloWorld; // Executes TMyClassHelper.HelloWorld
    X.MyFunc;      // Executes TMyClassHelper.MyFunc
```

Note that the reference is always pointed to TMyClass. The compiler recognizes when it's appropriate to execute the call in TMyClassHelper.

STRICT PRIVATE AND STRICT PROTECTED

In Delphi you have two options for determining the visibility of a class' attributes: strict private and strict protected.

- **Strict private:** class attributes are visible only within the class in which it is declared. Those attributes can't be seen by methods declared in the same unit, or by those that are not part of the class.
- **Strict protected:** determines that class attributes and their descendents are visible.

RECORDS SUPPORT METHODS

Record data types in Delphi represent a mixed set of elements. Each element is called a field; the declaration of a record type specifies a name and type for each field.

Records in Delphi 2006 are even more powerful, bringing features supported only in class.

Here's a list of the new record features in Delphi 2006

- Constructors
- Operator overload
- Non-virtual methods declaration
- Static methods and properties

Let's check the following example. It's the implementation of a record with the new characteristics:

```
Type
  TMyRecord = Record
  Type
    TColorType = Integer;
  Var
    pRed : Integer;
  Class Var
    Blue : Integer;
  Procedure printRed();
  Constructor Create(Val : Integer);
  Property Red : TColorType Read pRed Write pRed;
  Class Property  pBlue : TColorType Read Blue Write Blue;
  End;

Constructor TMyRecord.Create(Val: Integer);
Begin
  Red := Val;
End;

Procedure TMyRecord.printRed;
Begin
  WriteLn('Red: ', Red);
End;
```

Now the record can use many of the functionalities that were exclusive to classes. However, records are not classes, meaning they still have many differences:

- Records do not support inheritance.
- Records may have variable parts; classes may not.
- Records are data types and, as so, can be copied. Classes cannot.
- Records have no destructors.
- Records do not support virtual methods.

CLASS ABSTRACT, CLASS SEALED, CLASS CONST, CLASS TYPE, CLASS VAR, CLASS PROPERTY

There are many ways for you to declare classes, types, variables and properties.

- Class abstract → defines an abstract class
- Class sealed → classes cannot be extended through inheritance - such a class cannot be used as a base class for some other (derived) class
- Class const → defines a class constant that can be accessed without having to instantiate the class
- Class type → defines a class type that can be accessed without having to instantiate the class
- Class var → defines a variable from the scope of the class which you can access without having to instantiate the class
- Class property → grants access to the property without requiring the class to be instantiated

NESTED CLASSES

Nested types are used throughout object-oriented programming in general. They allow you to keep conceptually related types together, and to avoid name collisions. The same syntax for declaring nested types may be used with the Delphi compiler. Class sample below:

```
type
  TOuterClass = class
    strict private
      myField: Integer;

    public
      type
        TInnerClass = class
          public
            myInnerField: Integer;
            procedure innerProc;
          end;

          procedure outerProc;
        end;

This is how the method is implemented:

procedure TOuterClass.TInnerClass.innerProc;
begin
  ...
end;
```

To access the method within the Nested class, see the next example:

```
var
  x: TOuterClass;
  y: TOuterClass.TInnerClass;

begin
  x := TOuterClass.Create;
  x.outerProc;
  ...
  y := TOuterClass.TInnerClass.Create;
```

FINAL METHODS

The Delphi compiler also supports the concept of a final virtual method. When the keyword `final` is applied to a virtual method, no descendent class can override that method. Use of the `final` keyword is an important design decision that can help document how the class is intended to be used. It can also give the compiler hints that allow it to optimize the code it produces.

STATIC CLASS METHOD

When these classes are declared as `Static` they do not need to be instantiated.

FOR ... IN

Delphi 2007 brought supports for-element-in-collection (collections, arrays, string expressions and set-type expressions) style iteration over containers.

Example: Iteration in an Array

```
var
  IArray1: array[0..9] of Integer = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
  I: Integer;
begin
  for I in IArray1 do
  begin
    // do something here...
  end;
```

Example: Iteration in a String

```
var
  C: Char;
  S1, S2: String;

  OS1, OS2: ShortString;
  AC: AnsiChar;

begin
```

```
S1 := 'New resources in Delphi 2009';
S2 := '';

for C in S1 do
    S2 := S2 + C;

if S1 = S2 then
    WriteLn('SUCCESS #1');
else
    WriteLn('FAIL #1');

OS1 := 'Migrating from Delphi 7 to Delphi 2009...';
OS2 := '';

for AC in OS1 do
    OS2 := OS2 + AC;

if OS1 = OS2 then
    WriteLn('SUCCESS #2');
else
    WriteLn('FAIL #2');

end.
```

GENERIC

Delphi 2009 supports *generics*, which were previously available only for Delphi .Net.

What are Generics? 'Generics' is the defining term for generic types. It is a resource that allows you to predefine any type of data from arrays, collections, and other sorts of lists. Write code in a generic way and have it work with a specific type of data - classes or class methods. It's also possible to define types at runtime.

While working with Collections (data collection) you most often use some of the classes listed below:

- ArrayList
- HashTable
- Queue
- SortedList
- Stack

Items added to these classes are of TObject type, meaning you can only add a single type to the list. Whenever you need to read the items you must perform the type cast for each of them, leaving all processing for the compiler. This is nothing but additional effort for your application, or something that can potentially impact its performance.

A relevant portion of our code can be adapted to work with Generics. Below you see a sample class whose Key property should be a String and whose Value property should be an Integer. In this case you do not use Generics.

```
type
  TSIPair = class
  private
    FKey: String;
    FValue: Integer;
  public
    function GetKey: String;
    procedure SetKey(Key: String);
    function GetValue: Integer;
    procedure SetValue(Value: Integer);
    property Key: TKey read GetKey write SetKey;
    property Value: TValue read GetValue write SetValue;
  end;
```

Using Generics it's possible to define the Key and Value properties as being of any type. This is how you do so:

```
type
  TPair<TKey,TValue>= class    // declares the TPair type with 2 parameters
  private
    FKey: TKey;
    FValue: TValue;
  public
    function GetKey: TKey;
    procedure SetKey(Key: TKey);
    function GetValue: TValue;
    procedure SetValue(Value: TValue);
    property Key: TKey read GetKey write SetKey;
    property Value: TValue read GetValue write SetValue;
  end;
```

You can now use this class in many different ways:

```
type
  TSIPair = TPair<String,Integer>; // declares it with types String and Integer
  TSSPair = TPair<String,String>; // declares it with other types
  TISPair = TPair<Integer,String>;
  TIIPair = TPair<Integer,Integer>;
```

You can find many uses for generics. Examples are provided here as a simple, limited reference.

ANONYMOUS METHODS

As the name suggests, an anonymous method is a procedure or function that does not have a name associated with it. An anonymous method treats a block of code as an entity that can be assigned to a variable or used as a parameter to a method. In addition, an anonymous method can refer to variables and bind values to the variables in the context in which the method is defined. Anonymous methods can be defined and used with simple syntax. They are similar to the construct of closures defined in other languages.

Example:

```
function MakeAdder(y: Integer): TFuncOfInt;  
begin  
    Result := { START anonymous method } function(x: Integer)  
    begin  
        Result := x + y;  
    end; { END anonymous method }  
end;
```

The MakeAdder function returns a nameless function; that is, an Anonymous Method.

Note that MakeAdder returns a value of TFuncOfInt type. The type of the Anonymous Method is declared as a reference to the method.

Example: Executing the MakeAdder function

```
var  
    adder: TFuncOfInt;  
begin  
    adder := MakeAdder(20);  
    Writeln(adder(22)); // prints 42  
end.  
type  
    TFuncOfInt = reference to function(x: Integer): Integer;
```

The above statement indicates this Anonymous Method:

- Is a function
- Receives an Integer value
- Returns an Integer value

You can declare both procedures and functions as Anonymous Methods.

```
type  
    TSimpleProcedure = reference to procedure;  
    TSimpleFunction = reference to function(x: string): Integer;
```

Anonymous Methods offer much more than just a simple execution point in code:

- Binding in variables
- Ease to use and define methods
- Ease to parameterize the code

NEW \$POINTERMATH {ON – OFF } DIRECTIVE

The \$POINTERMATH directive enables mathematic operations with pointers.

NEW WARNINGS

When you get to compile your application with Delphi 2009 a couple of new warnings might appear in your IDE's message window. These messages regard the use of the new UnicodeString type. To be precise, four new warnings will be displayed to assist you in using Unicode.

DBEXPRESS

FRAMEWORK

One of the most significant changes in Delphi 2007, dbExpress architecture has been restructured, and you can now count on a framework that is totally written in Delphi. We have performed lab tests to simulate the most diverse situations with varied databases; in some of those tests the performance was improved by 100%.

DbExpress 4 is also a milestone for applications developed in Delphi that require database connection. The new architecture was designed to support Win32 and .NET, enabling the same drivers to be used in both platforms.

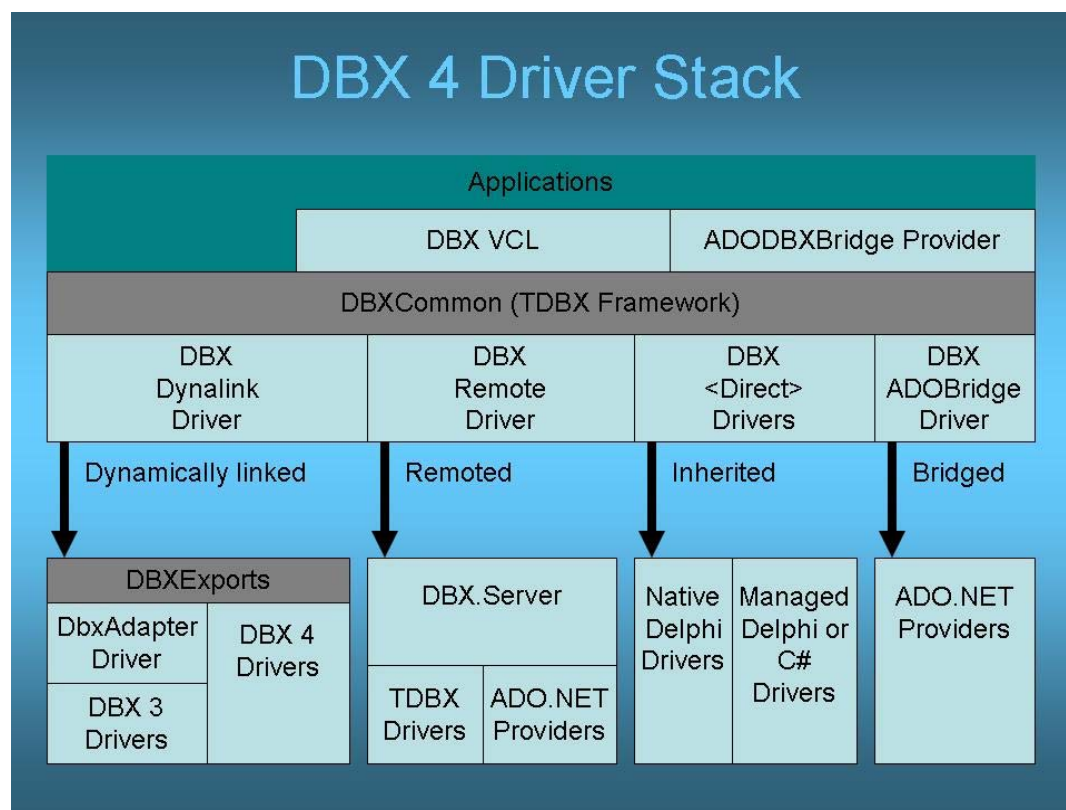


Figure 26 dbExpress 4 Architecture

Remember that applications developed in prior versions are 100% Delphi 2009 compatible.

The dbExpress Framework comes with a new group of classes that facilitate the task of accessing and otherwise handling databases. You can now find all the information regarding the database within the framework. Previously you'd have to use components `SQLConnection`, `SQLDataSet`, `SQLQuery`, and others instead.

The following example represents a console application that uses database connection resources, reads connections parameters, sends a query and displays its result – all in a single transaction.

```
program DBX4Example;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  DBXDynalink,
  Dialogs,
  DBXCommon;

var
  aConnName: string;
  aDBXConn: TDBXConnection;
  aDBXTrans : TDBXTransaction;
  aCmnd: TDBXCommand;
  aReader: TDBXReader;
  i, aColCount: integer;

begin
  aDBXConn := TDBXConnectionFactory.GetConnectionFactory.GetConnection(
    'EMPLOYEE', 'sysdba', 'masterkey');

  // Write the all connection parameters
  Writeln( '===== Connection Properties =====' );
  Writeln( aDBXConn.ConnectionProperties.Properties.Text );
  Writeln( '===== ' );
  Writeln( '' );

  if aDBXConn <> nil then
  begin
    aCmnd := aDBXConn.CreateCommand;

    // Start transaction
    aDBXTrans:= aDBXConn.BeginTransaction(TDBXIsolations.ReadCommitted);

    // Prepare and execute the SQL Statement
    aCmnd.Text := 'SELECT * FROM Country';
    aCmnd.Prepare;
    aReader := aCmnd.ExecuteQuery;

    aColCount := aReader.ColumnCount;
    Writeln( 'Results from Query: ' + aCmnd.Text );
```



```
Writeln( 'Number of Columns: ' + IntToStr(aColCount) );

while aReader.Next do
begin
  Writeln( aReader.Value['Country'].GetAnsiString );
end;

Writeln( '===== ' );
Writeln( ' ' );

end;
// Commit transaction
aDBXConn.CommitFreeAndNil(aDBXTrans);

Readln;
aReader.Free;
aCmnd.Free;
aDbxConn.Free;

end;

end.
```

DBEXPRESS METADATA

The new metadata support is used extensively by the Data Explorer pane of the Delphi IDE, but can also be used by any application. In short, you'll not only be able to browse the database structure but also to use classes and objects to modify it, rather than relying directly on the native database SQL commands for creating and modifying data structures. Not only will the code look more object-oriented, but it will be also easier to target different database servers with the same code, as dbExpress abstracts the metadata capabilities of each server.

The unit DBXMetaDataNames has been provided to read metadata. The dbExpress class TDBXMetaDataCommands provides a set of constants to read various types of metadata. Set the TDBXCommand.CommandType property to DBXCommandTypes.DBXMetadata and set TDBXCommand.Text to one of the constants in TDBXMetaDataCommands to acquire the designated metadata. TDBXCommand.ExecuteQuery returns a TDBXReader to access the metadata. The new classes in DBXMetaDataNames describe and provide access to this metadata's columns. Below a list of metadata you can read using dbExpress:

- Data types
- Tables
- Columns (from tables, views, etc.)
- Indexes
- Fields from those indexes
- Foreign key
- Fields from Foreign keys
- Stored Procedures
- Stored Procedures' parameters
- User list
- Catalogs

- Schemas
- Views
- Synonyms
- Stored Procedures' source code
- Packaged Stored Procedures
- Packaged Stored Procedures' source code
- Packaged Stored Procedures' parameters
- Roles
- Reserved words

Data Explorer includes support for creating SQL dialect sensitive CREATE, ALTER, and DROP statements. dbExpress also exposes a `DbxMetaDataProvider` class that surfaces this capability for applications. This slightly increases the size of application, since the metadata writers must be included. The ability to generically create tables is useful for many applications. The interface allows you to describe what a table and its columns look like and pass this description to the `TdbxMetaDataProvider.CreateTable` method.

Below you find an example on how to create tables, primary keys, foreign keys, and indexes using dbExpress Framework's classes.

```
var
  Provider: TDBXDataExpressMetaDataProvider;
  Country, State: TDBXMetaDataTable;
  IdCountryField,
  IdField: TDBXInt32Column;
  StrField : TDBXUnicodeVarCharColumn;
begin
  Provider := DBXGetMetaProvider(conn.DBXConnection);

  // Country
  Writeln('Creating Table - Country .....');
  Country := TDBXMetaDataTable.Create;
  Country.TableName := 'COUNTRY';

  IdCountryField := TDBXInt32Column.Create('COUNTRYID');
  IdCountryField.Nullable := false;
  IdCountryField.AutoIncrement := true;
  Country.AddColumn(IdCountryField);

  StrField := TDBXUnicodeVarCharColumn.Create('COUNTRYNAME', 50);
  StrField.Nullable := False;

  Country.AddColumn(StrField);

  Provider.CreateTable(Country);

  // Defines COUNTRYID as Primary Key
  AddPrimaryKey(Provider, Country.TableName, IdCountryField.ColumnName);

  // Defines Unique Index as COUNTRYNAME
  AddUniqueIndex(Provider, Country.TableName, StrField.ColumnName);

  // State
```

```
Writeln('Creating Table - State .....');
State := TDBXMetaDataTable.Create;
State.TableName := 'STATE';

IdField := TDBXInt32Column.Create('STATEID');
IdField.Nullable := false;
IdField.AutoIncrement := true;
State.AddColumn(IdField);

StrField := TDBXUnicodeVarCharColumn.Create('SHORTNAME', 2);
StrField.Nullable := False;
State.AddColumn(StrField);

StrField := TDBXUnicodeVarCharColumn.Create('STATENAME', 50);
StrField.Nullable := False;
State.AddColumn(StrField);

State.AddColumn(IdCountryField);

Provider.CreateTable(State);

// Defines STATEID as Primary Key
AddPrimaryKey(Provider, State.TableName, idField.ColumnName);

// Defines Unique Index as STATENAME
AddUniqueIndex(Provider, State.TableName, StrField.ColumnName);

AddForeignKey(Provider, State.TableName, IdCountryField.ColumnName,
               Country.TableName, IdCountryField.ColumnName);

FreeAndNil(Provider);
FreeAndNil(Country);
```

The source code for this example is available at <http://cc.embarcadero.com/Item/26210>.

DBEXPRESS DRIVERS

Support for the latest versions of databases: InterBase 2007/2009, MySQL 4.1. 5.0, Oracle 10g. Drivers for Oracle, InterBase and MySQL now come with Unicode support.

New concepts, called “Delegate Driver” and “Pools Connections”, are available in dbExpress and require simple parameter configuration.

You can also extend the dbExpress framework to write delegation drivers, which provide an extra layer between the application and the actual driver. Delegate drivers are useful for connection pooling, driver profiling, tracing, and auditing DBXTrace is a delegate driver used for tracing.

Below you see the log result generated by the Delegate in Delphi language. It’s easy to read, understand, and even execute operations once again.

Trace configuration. The following example captures events according to the TraceFlags parameter, saving the log file at c:\dbxtrace.txt. The dbExpress connection has a parameter to indicate the trace configuration; for instance: DelegateConnection= DBXTraceConnection

```
[DBXTraceConnection]
DriverName=DBXTrace
TraceFlags=EXECUTE;COMMAND;CONNECT
TraceDriver=true
TraceFile=c:\dbxtrace.txt
```

Generated log result:

```
Log Opened =====
{CONNECT      } ConnectionC1.Open;
{COMMAND      } CommandC1_1 := ConnectionC1.CreateCommand;
{COMMAND      } CommandC1_1.CommandType := 'Dbx.SQL';
{COMMAND      } CommandC1_1.CommandType := 'Dbx.SQL';
{COMMAND      } CommandC1_1.Text := 'select * from employee';
{PREPARE      } CommandC1_1.Prepare;
{COMMAND      } ReaderC1_1_1 := CommandC1_1.ExecuteQuery;
{COMMAND      } CommandC1_2 := ConnectionC1.CreateCommand;
{COMMAND      } CommandC1_2.CommandType := 'Dbx.Metadata';
{COMMAND      } CommandC1_2.Text := 'GetIndexes
"localhost:C:\CodeGear\InterBase\examples\database\employee.ib"."sysdba"."em
ployee" ';
{COMMAND      } ReaderC1_2_1 := CommandC1_2.ExecuteQuery;
{READER       } { ReaderC1_2_1 closed.  6 row(s) read }
{READER       } FreeAndNil(ReaderC1_2_1);
{COMMAND      } FreeAndNil(CommandC1_2);
```

You can also use connection pools with dbExpress natively. Below you see an *alias* (Pool_DelegateDemo) passing to DBXPoolConnection the control over the connection pool (where the maximum number of connections is set).

```
[DBXPoolConnection]
DriverName=DBXPool
MaxConnections=16
MinConnections=0
ConnectTimeout=0
```

```
[Pool_DelegateDemo]
DelegateConnection=DBXPoolConnection
DriverName=Interbase
DriverUnit=DBXDynalink
DriverPackageLoader=TDBXDynalinkDriverLoader
DriverPackage=DBXCommonDriver110.bpl
DriverAssemblyLoader=Borland.Data.TDBXDynalinkDriverLoader
DriverAssembly=Borland.Data.DbxCommonDriver,Version=11.0.5000.0,Culture=neut
ral,PublicKeyToken=a91a7c5705831a4f
Database=localhost:C:\CodeGear\InterBase\examples\database\employee.ib
RoleName=RoleName
User_Name=sysdba
Password=masterkey
ServerCharSet=
SQLDialect=3
```

```
BlobSize=-1  
CommitRetain=False  
WaitOnLocks=True  
ErrorResourceFile=  
Interbase TransIsolation=ReadCommitted  
Trim Char=False
```

DATA SNAP

Integration between DataSnap and dbExpress is surely among this version's most significant new features. Based on feedback received from our customers and user groups we have created the new DataSnap. As usual, creating a multilayer application seemed easy. However, considering its continued use and the impressive number of DataSnap applications available, many opportunities for improvement have been identified. In this chapter we explore the concepts surrounding the creation of multilayer applications with the new DataSnap.

CONCEPTS

The new integration between DataSnap and dbExpress – which many are calling dbExpress remoting – brought great flexibility into the world of multi-layer application development. Before that, sending and receiving data by means of ClientDataSet was the practice. Working with remote functions was something that required the use of a Type Library, consequently making you dependent on COM (present in Remote Data Module). Developers requested that we remove the COM dependency. This is a main attribute of the new DataSnap: it does not depend on any COM technology. However, this technology was not disregarded and compatibility is maintained, allowing you to use it whenever you find appropriate.

DataSnap integration with dbExpress allows dynamic execution of server methods using, for example, SQLDataSet or the new SqlServerMethod component. The parameter and the result of the method are defined using the Param properties.

One way to execute the methods on the server is using the new SqlServerMethod component, this component inherits from CustomSQLDataSet, which means you can execute server side methods using a DataSet, input/output parameters will be represented by Params property.

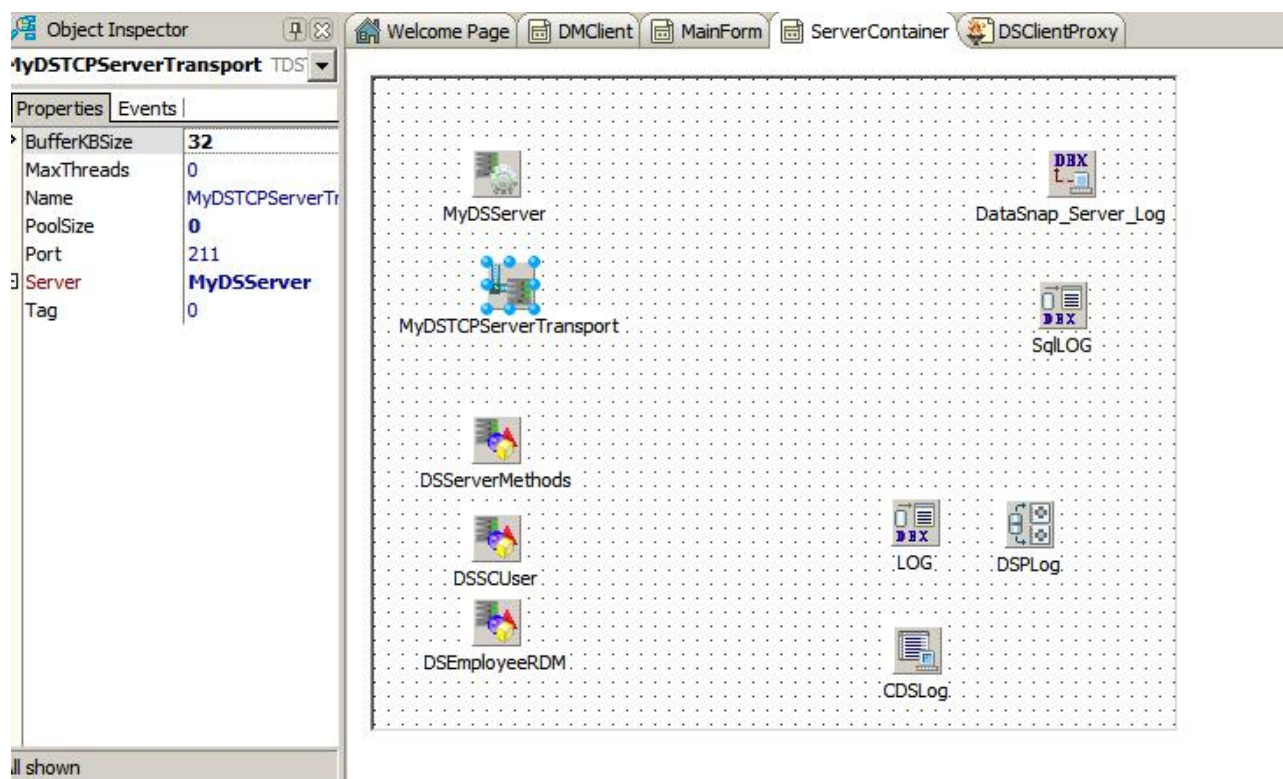
DATA SNAP SERVER – SERVER CONTAINER

DataSnap servers are defined by two components: DSServer and DSTCPTransport. Add these two components to your application and you have a DataSnap server. The form/datamodule which hosts these components can be called Server Container. From now on we will start using a new naming convention.

DSServer is your DataSnap server. When connected to DSTCPTransport – which is where you define the connection port, the maximum number of threads, etc. – it exposes your application as a server.

DSTCPTransport transports information by means of TCP, thus enabling you to extend and create new transport methods (e.g., HTTPS, SSL, to name a few). DSTCPTransport currently uses Indy's infrastructure for TCP connections.

If you use BSS, your application will still work in Delphi 2009. However, it's recommended that you migrate to the new DataSnap.



DATA SNAP SERVER – SERVER MODULE

You most probably have many classes that hold business rules which would otherwise be of better use in a multilayer application. Using Server Methods you can easily expose all public methods to the client side.

In order for a class to be made available as a Server Method, it must:

- Descend from TPersistent
- Have the {\$MethodInfo ON} directive. This directive allows dbExpress to obtain information about the class from RTTI.
- Be registered by means of the DSServerClass component.

Finally, we use the term Server Module to define the location of the providers, Server Methods, etc. You can create a Server Module selecting File → New → Other → Delphi Files → Server Module.

The Server Module is a DataModule that comes with the directive \$MethodInfo ON by default.

Each class you make available has an associated DSServerClass component. This component is responsible for registering the class and making it available to client applications. It's recommended that you keep your DSServerClass components in the Server Module. You can

have as many Server Modules as you wish, being then able to have a better organized application.

The various DSServerClass define the application lifetime, or the LifeCycle property:

- Server → One component instance is used per server (Singleton)
- Session → One component instance is used per DataSnapSession: (Statefull).
- Invocation → One component instance is used per invocation of a method (Stateless).

Additionally, DSServerClass comes with a few events that require you to use the OnGetClass event to get the class registered. Below you see an example of it, with TServerMethods working as a datamodule that holds many methods:

```
procedure TDMServerContainer.DSServerMethodsGetClass (DSServerClass :  
TDSServerClass; var PersistentClass : TPersistentClass);  
begin  
    PersistenClass := TServerMethods  
end;
```

Proceed the same way in case you have a Remote Data Module in your application.

DATASNAP CLIENT – DBEXPRESS

Since Delphi 2007 - when the dbExpress Framework was created - our goal was that of making it a supporting infrastructure for many other technologies, extending the existing multi-layer support and enabling Java, .NET, PHP, and other clients to connect to DataSnap servers.

DbxClient (originally created to allow Blackfish SQL connections) is now being used to connect the client to the DataSnap server, as well. This means that in order to connect to a DataSnap server you must use an SQLConnection, informing that the connection driver is DataSnap and providing the hostname (server) and port. Just as simple.

There are many ways for you to execute the server methods. Let's assume the server holds a class that contains the HelloWorld and GetEmployee methods.

```
function TDMServerDB.HelloWorld(IncommingMessage : WideString) : WideString;  
begin  
    Result := 'Hello World';  
end;  
  
function TDMServerDB.GetEmployee(ID : Integer) : TDBXReader;  
begin  
    SQLDataSet1.Close;  
    SQLDataSet1.Params[0].AsInteger := ID;  
    SQLDataSet1.Open;  
    Result := TDBXDataSetReader.Create(SQLDataSet1, False);  
end;
```

The HelloWorld method is quite simple. It sends a string and returns another. Method GetEmployee, in turn, sends an ID and receives a TDBXReader. In other words, it's a cursor that can be read client-side either as a DBXReader or as a ClientDataSet.

Still working with the HelloWorld method, you'll now execute it using the SQLServerMethod component. See below:

```
begin
  DMDataSnapClient.DSServerConnect.Open; // opens the connection by means of
  SQLConnection

  SMHelloWorld.SQLConnection := DMDataSnapClient.DSServerConnect;
  SMHelloWorld.Params[0].AsString := 'Message sent from Client';
  SMHelloWorld.ExecuteMethod;

  ShowMessage(SMHelloWorld.Params[1].AsString); // shows the result
End;
```

Let's look at the GetEmployee method. In this example the server method is executed through the dbExpress Framework. The same method could be executed through SqlServerMethod.

```
1 var
2   Command : TDBXCommand;
3   Reader   : TDBXReader;
4 begin
5   DMDataSnapClient.DSServerConnect.Open;
6   With DMDataSnapClient.DSServerConnect.DBXConnection do begin
7
8     Command := DMDataSnapClient.DSServerConnect.DBXConnection.CreateCommand;
9     Command.CommandType := TDBXCommandTypes.DSServerMethod;
10    Command.Text := TDSAdminMethods.GetServerMethodParameters;
11    Reader := Command.ExecuteQuery;
12
13    TDBXDataSetReader.CopyReaderToDataSet(Reader, ClientDataSet1);
14    ClientDataSet1.Open;
15 end;
```

Note that the execution is performed by TDBXCommand. The return of type DBXReader is copied to a ClientDataSet in line 13, thus allowing data to be visualized in the VCL.

In cases where you don't need to display the data, DBXReader can be read directly.

You might be wondering... If this is all dynamic now, wouldn't the compiler be able to detect when server methods change? The answer could be positive in case server methods were not represented by means of a client interface. SQLConnection includes an option called "Generate DataSnap Client Access", that generates a client-side unit with all the methods available at server-side. Each method in the client class contains an implementation to execute the server method.

See a sample class below:

```
TDSServerMethodsClient = class
private
  FDBXConnection: TDBXConnection;
  FGetServerDateTimeCommand: TDBXCommand;
  FExecuteJobCommand: TDBXCommand;
public
  constructor Create(ADBXConnection: TDBXConnection);
  destructor Destroy; override;
  function GetServerDateTime: TDateTime;
  function ExecuteJob(JobId: Integer): Integer;
end;
implementation

function TDSServerMethodsClient.GetServerDateTime: TDateTime;
begin
  if FGetServerDateTimeCommand = nil then
  begin
    FGetServerDateTimeCommand := FDBXConnection.CreateCommand;
    FGetServerDateTimeCommand.CommandType := TDBXCommandTypes.DSServerMethod;
    FGetServerDateTimeCommand.Text := 'TDSServerMethods.GetServerDateTime';
    FGetServerDateTimeCommand.Prepare;
  end;
  FGetServerDateTimeCommand.ExecuteUpdate;
  Result := FGetServerDateTimeCommand.Parameters[0].Value.AsDateTime;
end;

function TDSServerMethodsClient.ExecuteJob(JobId: Integer): Integer;
begin
  if FExecuteJobCommand = nil then
  begin
    FExecuteJobCommand := FDBXConnection.CreateCommand;
    FExecuteJobCommand.CommandType := TDBXCommandTypes.DSServerMethod;
    FExecuteJobCommand.Text := 'TDSServerMethods.ExecuteJob';
    FExecuteJobCommand.Prepare;
  end;
  FExecuteJobCommand.Parameters[0].Value.SetInt32(JobId);
  FExecuteJobCommand.ExecuteUpdate;

end;

constructor TDSServerMethodsClient.Create(ADBXConnection: TDBXConnection);
begin
  inherited Create;
  if ADBXConnection = nil then
    raise EInvalidOperation.Create('Connection cannot be nil. Make sure the
connection has been opened. ');
  FDBXConnection := ADBXConnection;
end;

destructor TDSServerMethodsClient.Destroy;
begin
  FreeAndNil(FGetServerDateTimeCommand);
  FreeAndNil(FExecuteJobCommand);
```

```

    inherited;
end;

```

And how do you access the DataSetProvider from the server, from a remote data module, or from a conventional data module? That's simple: client-side you use the new DSProviderConnection component, which is connected to your SQLConnection (DataSnap). The ServerClassName property indicates the class name (DataModule/RDM, usually) where the providers are located in the server. This way the ClientDataSet use DSProviderConnection as ProviderName.

The new DataSnap and dbExpress Framework provide greater flexibility, not limited by what's presented in this chapter. It's possible to dynamically access a list of methods along with their parameters, which translates into a great opportunity for developers to create components that control user access to the server, defining access rights specific to each of the server's methods and classes. You can take a look at a more complete DataSnap application sample visiting my blog at <http://blogs.embarcadero.com/andreanolanus>.

NEW TRANSLATION TOOL

The IDE-integrated translation tool is now standalone. This means the professional responsible for translating your project can now use the same tool you do, without having to install Delphi himself/herself. For each new language a translation project is generated. It's then much easier to edit language-specific DFM files.

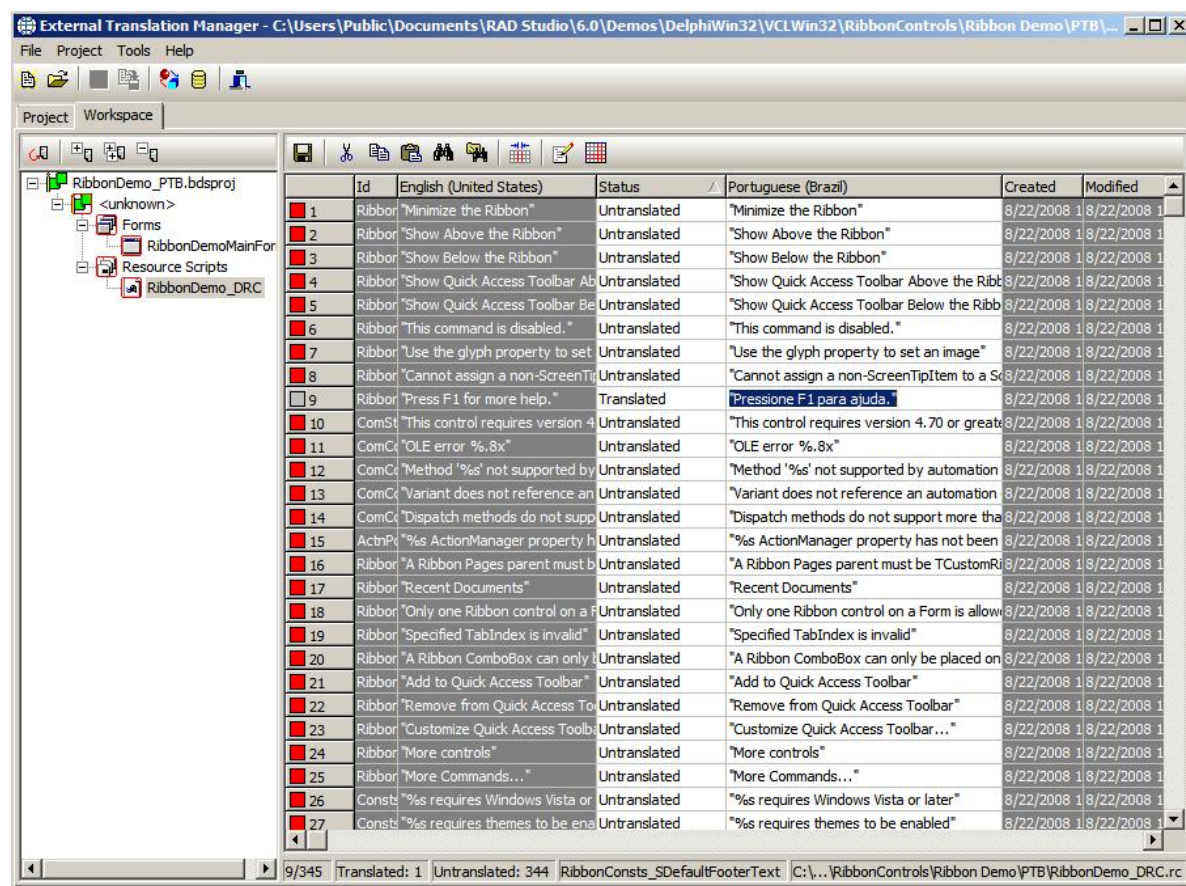


Figure 27 Translation Tool

UML MODELING

Any final product is expected to be first-class. Quality concerns are no different when it comes to software (especially considering they're a core element in supporting a company's growth efforts). Keep in mind that whenever you deliver low-quality software you are compromising your client's success.

Since Delphi 2006 you can use UML and all of its diagrams. In addition, you can also use LiveSource, which allows you to synchronize class diagrams and code.

Below you see a list of all diagrams available, along with their functionality:

- Use Case → it is a way to describe the interaction between a system and the real world. In this case, the actors (either persons or systems) represent the real world.
- Class Diagram → represents the classes of the system and the relationships they establish.
- Collaboration → used for modeling the dynamic aspects of a system or subsystem.
- Activity → allows you to represent dynamic situations by means of flows (using it you can represent the flow between different objects).
- Component → used in higher-level modeling, in cases where more complex structures are present. This diagram illustrates systems, embedded controls, etc.
- State → specifies the sequence of events of a given object.

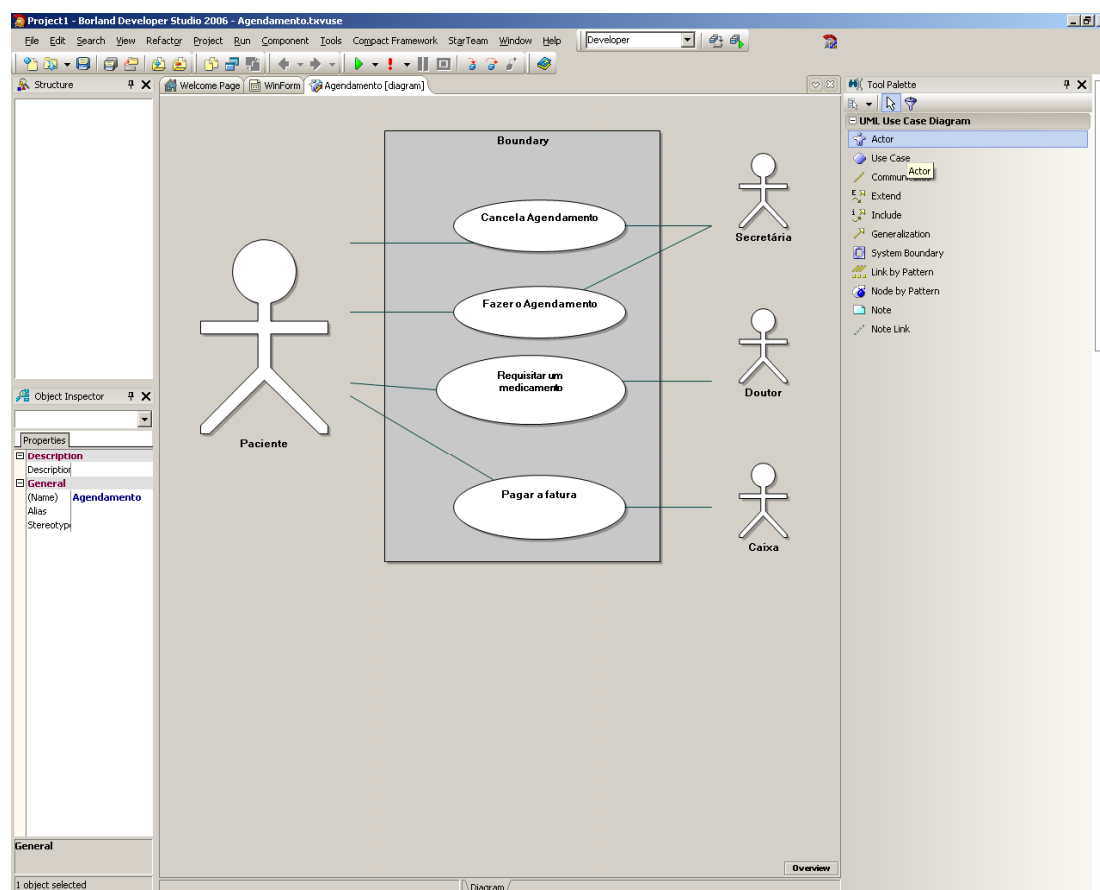


Figure 28 Use Case Diagram

Visualizing a class diagram makes it much easier for you to understand the classes in it (as compared with doing the same with code). Let's see an example in Delphi: the Buttons.pas unit has many components - TBitBtn, TSpeedButton, among others. Now imagine how hard it would be to decipher 1946 lines of code to learn which components are there and which relationships they established. Using reverse-engineering it's not a big deal... Let's check the following figure:

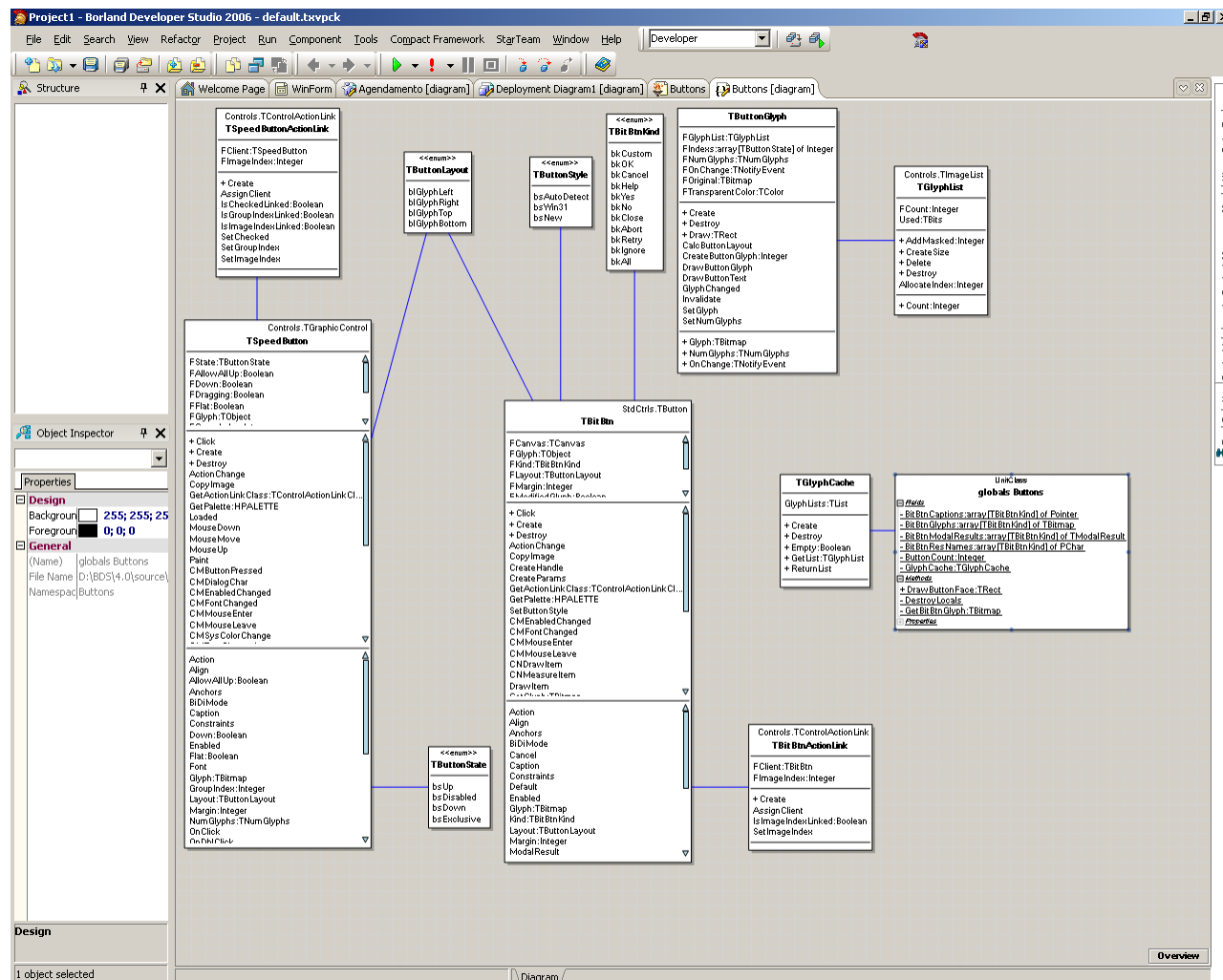


Figure 29 Class Diagram

Delphi has full reverse engineering capabilities. Embarcadero is committed to helping developers evolve legacy code; a commitment that is reinforced by our efforts toward continuously providing technology that allows applications to move on.

AUDITS

When it comes to quality people are always concerned about delivering high-performance software. Many even tell us they do not care about the way software is written, rather with its ability to work and meet their needs. In fact, this is a mistake with potential to impact you in a not so distant future. When you write unstructured code you're impairing your ability to extend the application in the future, in case you need to. It's then all pieced together, and your application does not grow in a structured manner. Delphi's audits and metrics help you locate flaws in your application while you're still developing it. They also help developers get used to writing standardized code.

How many times have you defined best-practice guidelines for coding, hoping it would prevent your staff from making mistakes of the kind that turns it into a handful of ineffective typed-matter no one understands?

Assuming your team can count on a best-practice manual, the second question that comes to mind is: How can we ensure the instructions therein are being followed?

The answer, again: code review. Now, think of a context where your project is coded in no less than thousands of lines. What you have here is a schedule catastrophe.

Using Delphi's code audit resources (QA Audits) you can set a definite group of best coding practices, making sure they're being followed by your projects. Clearly speaking, you detect flaws in your application before it gets to be run.

Audits verify whether the code conforms to the rules and parameters defined by your organization. Results show only the detected violations, in a categorized manner:

- Arrays and References
- Duplicated code
- Superfluous content
- Performance
- Branches and Loops
- Coding style
- Naming style
- Expressions
- Design flaws
- Possible Errors

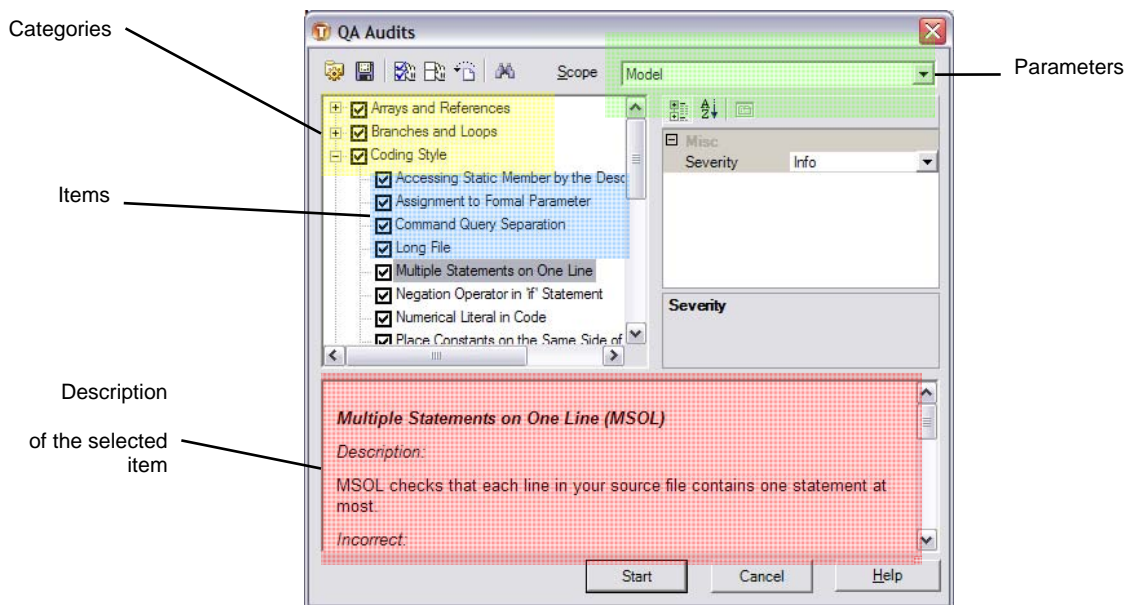


Figure 30 QA Audits

Each of the audit items includes a descriptive note explaining the correct and incorrect ways of using it. This helps developers understand how to use each specific audit item. The items can have their severity level set to Info, Warning, or Error. It's up to developers to define how relevant each item is.

LOOP BODY IS NEVER EXECUTED (LBNE)

Often times we come across execution routines that involve many loops, requiring us to debug it to make sure all loops are executed. Using the LBNE audit this kind of issue is detected. The following code is a simple example of something LBNE would detect. Even more complex cases which involve more conditions are also easily detected.

```
var
  x: Boolean;
begin
  x := false;
  while s do
  begin
    ....
  end;
```

INDEX OUT OF BOUNDS (IOB)

This is a common message for when you try to access an array position that does not exist. The snippet below generates this warning.

```
var
  nloops,
  i,
  j :integer;
  matriz : array of integer;
  somatorio : double;
begin
  for i := 0 to nloops do
  begin
    somatorio := 0;
    for j := 0 to High(matriz) do
      somatorio := somatorio + matriz[i];

  end;
```

Abbrevi...	Description	Severity	Resource	File	Line
MCS	Method 'Proc1' can be made static	Info	Proc1	c:\inetp...	8
IVNU	Iteration variable is not used in loop body	Warning	Proc1	c:\inetp...	25
PSIB	Place statement in block	Warning	Proc1	c:\inetp...	26
ONE	Operation has no effect	Warning	Proc1	c:\inetp...	26
IOB	Index may be out of array bounds	Warning	Proc1	c:\inetp...	26
IOB	Because another index variable was compared with length of this array	Warning	Proc1	c:\inetp...	25

Figure 31 Audits

The audit has located the error, informing you that the code in line 25 tries to access a variable that is not part of the 2nd loop. Drilling down a little:

In variable J's 'for' statement I'm trying to access one of the positions in array ARR, aiming at position I of the previous loop, while variable J's 'for' statement is the place where array ARR is being read.

METRICS

Metrics help you standardize the code. After all, who has never come across code with 10 constructors for a class, 10 ifs – the first enclosing the next ones, consecutively; methods with 20 parameters, and other practices that can only assist in making code incomprehensible? This is where metrics comes in handy, allowing you to define limits for the company to follow. An example: a class must not have more than 4 constructors, having 400 code lines tops, while topics are to be named in conformance with Pascal's convention, which rules the first letter of method names is always capitalized.

Each metric has its own limit which can be customized:

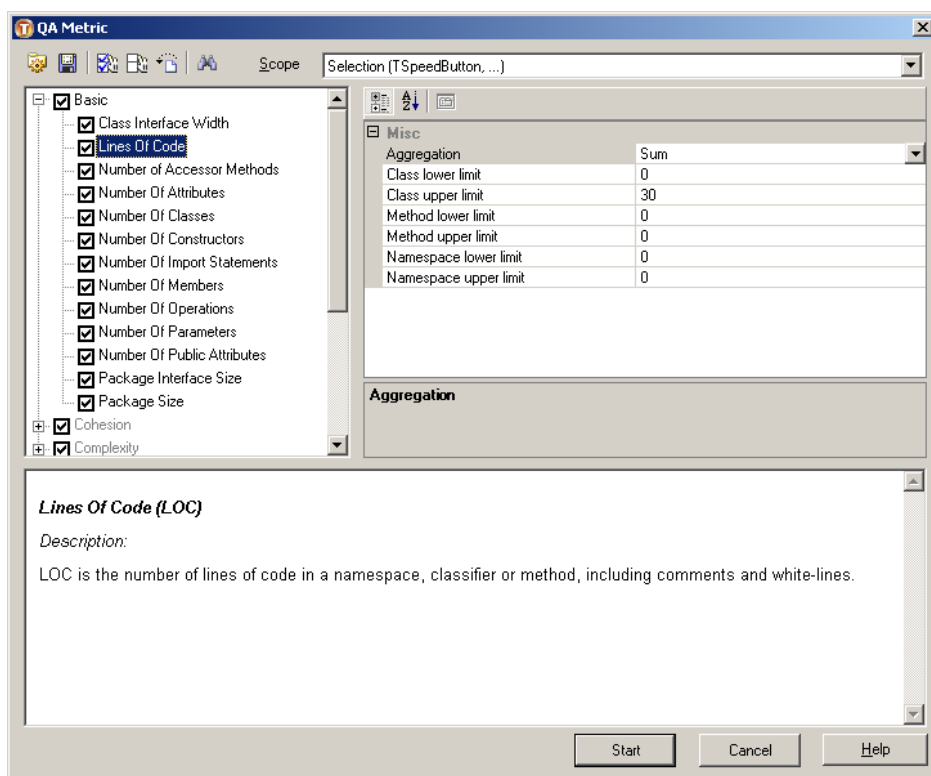


Figure 32 QA Metrics

After metrics are executed their results are analyzed with the assistance of a Kiviati chart. In a Kiviati chart the red circle represents the predefined limit. Points outside this boundary mean part of the code is breaking the metric rules.

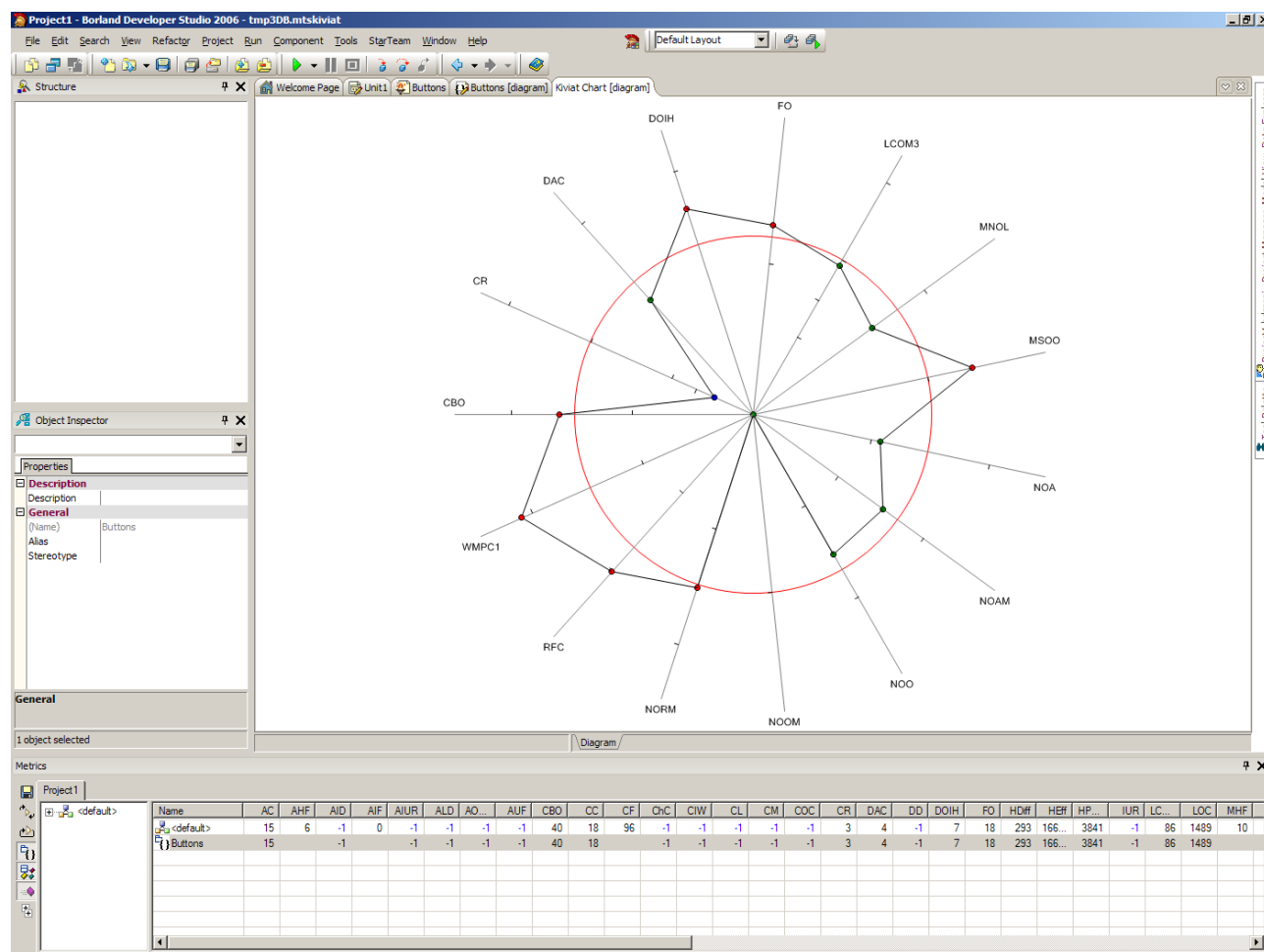


Figure 33 Metrics Analysis

You can analyze each of the classes separately. This way it's easier to identify violations of your metrics.

With audits and metrics developers are able to deliver higher-quality code/applications, just as good externally as they are in their core.

These are the things you are able to perform when your code is migrated from Delphi 7 to Delphi 2009.

DOCUMENTATION

Few things are as difficult as getting developers to document their applications. Developers develop, that's what they do best. Delphi can change this paradigm. With the assistance of Delphi's diagrams, developers, analysts, and architects learn how easy it can be to write code and document the entire application. It's as simple as getting into the diagram and... documenting. Taking a class diagram as an example, you simply click on the class, variable, method, and other class attributes and you're ready to document them.

The documentation is generated in HTML, separated as project overview, diagram view and documentation details.

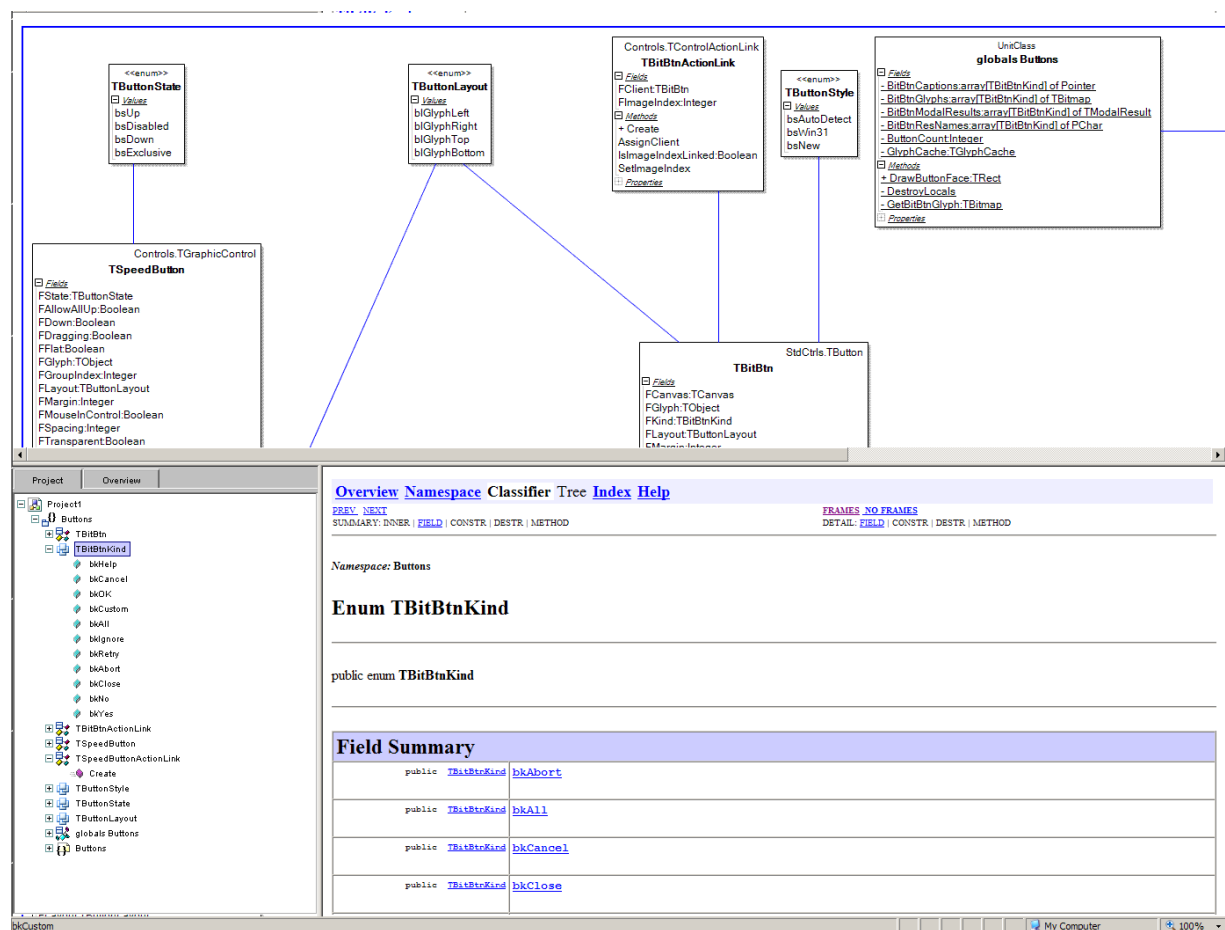


Figure 34 Thorough Documentation

THIRD-PARTY COMPONENTS

Many of the third-party components packed with Delphi were updated and now bring new resources and compatibility with existing applications. Additionally, if you upgrade your software to Delphi 2009 you're eligible to download the InfoPower Grid Essentials component pack. If you have any additional questions about some specific third-party component that you use, the Delphi product page has a list of many third components available today (<http://edn.embarcadero.com/article/38459>).

TEECHART 8

One of the components Delphi developers use the most, TeeChart, has been updated to version 8. It's now includes new resources aimed at working with Charts in Delphi.

RAVE REPORTS 7.6

Delphi 2009 comes with Rave Reports, the famous report generator, now update to version 7.6.

VCL FOR WEB

VCL for Web - previously called IntraWeb - allows you to create Web 2.0 applications, with transparent AJAX integration in many VCL components, and the newly added Silverlight support.

DELPHI 2009 – PROFESSIONAL, ENTERPRISE AND ARCHITECT

Delphi 2009 is offered in three editions: Professional, Enterprise and Architect. You can check the list of features available in each edition at: <http://www.embarcadero.com/products/delphi>.

The Architect edition includes the ER/Studio Developer Edition modeling tool. Database modeling is vital for developers working with applications that rely on databases to work.

ER/Studio supports the following databases: DB2 LUW V9, Hitachi HiRDB, IBM® DB/2®, Informix, InterBase®, Microsoft® Access, SQL Server, Visual FoxPro, MySQL, NCR Teradata, Oracle, PostgreSQL, Sybase, SQL Anywhere. Other databases can also be accessed through ODBC.

The tool provides resources as reverse-engineering, logical and physical modeling.

CONCLUSION

Delphi's IDE has been continuously improved over the past few years. Delphi 2009 is no exception. The new Project Manager, VCL components, RTL, generics, anonymous methods and the new DataSnap are sure to improve developers' productivity, allowing you not only to evolve existing applications but also develop new ones with next-generation technologies.

For additional detail drill down into what's new in Delphi 2009, please visit: <http://cc.embarcadero.com/coderage>.

ABOUT THE AUTHOR

In his role at Embarcadero, Andreano Lanusse is focused on helping make sure products are developed to meet customers' expectations. Responsible for defining market strategies for Latin America, he spends a great deal of time attending developer conferences, tradeshow, and user groups part of his duties as Latin Lead Evangelist, while also visiting strategic customers throughout Latin America. You can read more about Andreano Lanusse on his blog (<http://blogs.embarcadero.com/andreanolanusse>) and reach him at his e-mail address: andreano.lanusse@embarcadero.com.



Embarcadero Technologies, Inc. is a leading provider of award-winning tools for application developers and database professionals so they can design systems right, build them faster and run them better, regardless of their platform or programming language. Ninety of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero

products to increase productivity, reduce costs, simplify change management and compliance and accelerate innovation. The company's flagship tools include: Embarcadero® Change Manager™, CodeGear™ RAD Studio, DBArtisan®, Delphi®, ER/Studio®, JBuilder® and Rapid SQL®. Founded in 1993, Embarcadero is headquartered in San Francisco, with offices located around the world. Embarcadero is online at www.embarcadero.com.