White Paper

Building User Interfaces with Delphi 2009

An Improved Visual Component Library Streamlines Modern Windows Application Development

By Marco Cantù

December 2008

INTRODUCTION: THE VISUAL COMPONENT LIBRARY

The Visual Component Library (VCL) is one of the cornerstones of Delphi and its architecture has significantly contributed to the success of the tool. Most of the "Delphi experience" relates to the VCL, and this white paper focuses on the development of the user interface of Delphi applications.

With four new components (BalloonHint, ButtonedEdit, CategoryPanelGroup, and LinkLabel) plus Ribbon support and countless small enhancements, the VCL has seen a significant update in Delphi 2009. Some of these updates are specific for Windows XP or Windows Vista and further enhance the high-quality support for Vista that's been part of the VCL since Delphi 2007.

CUSTOM HINTS AND BALLOON HINTS

The `Tcontrol` class introduces a new property, `CustomHint`, and its *parent* property, `ParentCustomHint`, to let child objects share the value defined by the parent control:

```
property CustomHint: TCustomHint
  read GetCustomHint write SetCustomHint;
property ParentCustomHint: Boolean
  read FParentCustomHint write SetParentCustomHint
  default True;
```

`CustomHint` lets you hook a custom hint object to any visual component that is an object of any class inheriting from `TCustomHint`. One such class is the new `TballoonHint`. It is a simple and adds little to what the base `TCustomHint` class already provides, but its architecture is more flexible than having only balloon hint support, as you can add your own custom hint classes and use them for any control.

You can the BalloonHint component out-of-the-box. Simply place this non-visual component in a form and hook it to the `CustomHint` property of a control to change the way the hint is displayed. You can see a BalloonHint component in Figure 1:

Figure 1 BalloonHint Component in Delphi 2009

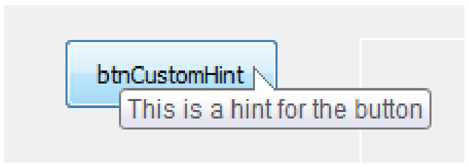
Here are the related settings from the DFM file of a button and its custom hint:

```
object btnCustomHint: TButton
  Hint = 'This is a hint for the button'
  CustomHint = BalloonHint1
  ShowHint = True
end
object BalloonHint1: TBalloonHint
```

```
  Images = ImageList1
end
```

The BalloonHint component uses the hint provided by the control to which it is hooked. As a user mouses over the button, the hint displays nicer than in the past as shown in Figure 2:

Figure 2 BalloonHint Display



Using the `ParentShowHint` and `ParentCustomHint` properties you can define this setting on a panel and have balloon hints active on each of the controls hosted by the panel. You might have noticed in the DFM listing previously that the BalloonHint component has an `Images` property, but no image is displayed. One way to set other runtime properties of the BalloonHint component, including the `Title` and the `ImageIndex`, (and have a nicer looking hint) is to manually invoke the hint.

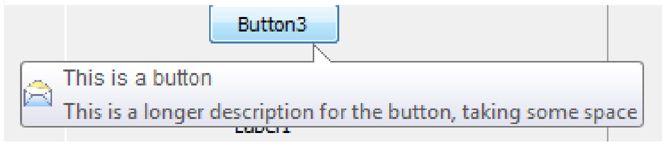
As this requires a lot of work, there is another easier way to set the title and the image index of the custom hint object connected with a control. Since the early days of Delphi, the `Hint` property allowed you to specify a short hint (used as hint) and a longer version (generally for a StatusBar message) separated by the pipe character (`|`). Using the custom hint association, the `Hint` property is now interpreted as follows:

```
title|message|imageindex
```

For example, I've customized a button as follows (the value of the hint is a single string) and the display is shown in Figure 3:

```
object Button3: TButton
  Hint =
    'This is a button|' +
    'This is a longer description for the button, ' +
    'taking some space|2'
  CustomHint = BalloonHint1
  Caption = 'Button3'
end
```

Figure 3 Custom Button


Button3
This is a button
This is a longer description for the button, taking some space

ENHANCEMENTS TO BUTTONS AND EDITS

Although some new features affect all controls, most of the Delphi 2009 improvements in the VCL are specific to individual controls. In this section I'll focus on the enhancements of some of the standard Windows controls, like buttons and edit boxes.

BUTTONS GET NEW FEATURES

You might think that the classic Windows push buttons are well-established, stable controls. That's actually not true. Since Windows XP, you can hook an image from an image list to a button, and have a graphical bitmap button without having to derive a custom owner-drawn control as Delphi did since the early days with the BitBtn (bitmap button) control. With Delphi 2009 you can now have the same graphical effect with a plain and standard `TButton`. Image list support comes through a series of properties you can use to determine which image to use in each of various states of the button. Here is the list of the new image-related properties of the `TCustomButton` class, listed with only their types:

```
property DisabledImageIndex: TImageIndex ...
property HotImageIndex: TImageIndex ...
property ImageAlignment: TImageAlignment ...
property ImageIndex: TImageIndex ...
property ImageMargins: TImageMargins ...
property Images: TCustomImageList ...
property PressedImageIndex: TImageIndex ...
property SelectedImageIndex: TImageIndex ...
```

Since this feature was introduced in the Win32 API in Windows XP, if your application needs to run on Windows 2000, you should use it with care or avoid using it altogether.

Similarly, if your program is meant to run on Vista, you can activate more new features, including the command link style used by many Vista dialogs and split-button styles that let you hook a drop-down menu to the button, which is activated by pressing the small drop-down arrow. The overall layout of the button is determined by the value of the new `Style` property of an enumerated type defined as a nested type of the `TCustomButton` class:

```
type
  TButtonStyle = (bsPushButton, bsCommandLink,
    bsSplitButton);
```

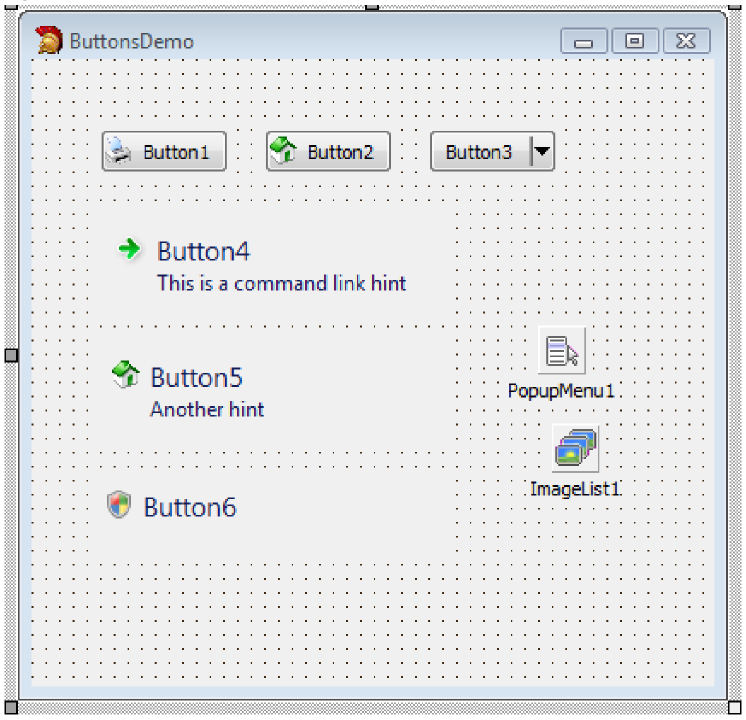
There are further properties you can use, depending on the selected style:

- With the split button style (in the API, the `BS_SPLITBUTTON` style value) you can use the `DropDownMenu` property (of type `TPopupMenu`) and customize it in the `OnDropDownClick` event.
- With the command link type (in the API, the `BS_COMMANDLINK` style value) you can use the default icon (a green arrow) or a specific image (as mentioned earlier) and provide more information about the action with the new `CommandLinkHint` string property.
- The `ElevationRequired` property, applicable both to a standard button and to a command link one, enables the display of the Windows shield to be used if the button leads

to a UAC-protected operation. The `ElevationRequired` property sends the `BCM_SETSHIELD` message to the button.

Using all of these new properties can affect the layout of your application quite radically, although you can obtain some of these user interface effects only if the application runs on Windows Vista (or later versions). These properties are not very complex to use, so rather than describing an example in detail, I'll simply list its key elements, after showing you the design-time form in Figure 4:

Figure 4 New Button Properties



This is the summary of the DFM file of the project:

```
object FormButtonsDemo: TFormButtonsDemo
  object Button1: TButton
    ImageIndex = 0
    Images = ImageList1
    PressedImageIndex = 1
  end
  object Button2: TButton
    ImageIndex = 1
    Images = ImageList1
    PressedImageIndex = 2
  end
  object Button3: TButton
    DropDownMenu = PopupMenu1
    Style = bsSplitButton
  end
  object Button4: TButton
    CommandLinkHint = 'This is a command link hint'
    Style = bsCommandLink
  end
  object Button5: TButton
    CommandLinkHint = 'Another hint'
    ImageIndex = 1
    Images = ImageList1
    Style = bsCommandLink
  end
  object Button6: TButton
    ElevationRequired = True
    Style = bsCommandLink
  end
  object ImageList1: TImageList...
  object PopupMenu1: TPopupMenu...
end
```

EDITS GET MANY NEW FEATURES

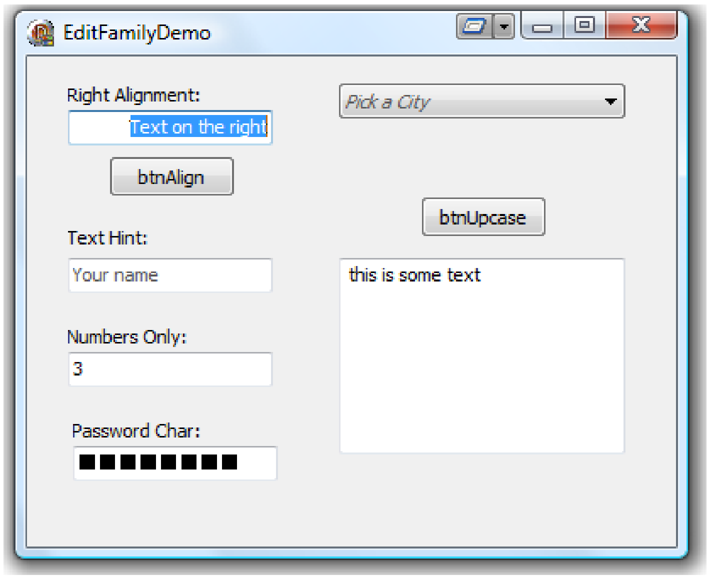
The Edit control is another standard and classic control of Windows that over the years got new features (particularly in Windows XP), some of which are now easily accessible using new properties of the `TEdit` class:

- The `Alignment` property enables the alignment of the text of the edit control, a feature that was previously available only for DBEdit controls (and implemented in native VCL code, as it wasn't available in early versions of the Win32 API). Setting the alignment activates the `ES_LEFT`, `ES_RIGHT`, or `ES_CENTER` Windows styles, eventually requiring the system to recreate the Edit window (so you should try to avoid changing this property at runtime once the Edit box has been displayed).
- The `NumbersOnly` property sets the `ES_NUMBER` style of the Edit control, which requires Windows XP or later. This applies an input filter that prevents user from typing non-digit keys, but still let's them paste non-numeric text (and lets the program freely set the `Text` property).
- The `TextHint` property supports in-place text hints displayed when the edit box is empty (again this requires Windows XP or later). The text hint could act as a replacement for a descriptive label, or reinforce one providing a call to action for the user.
- The `PasswordChar` property let's you set a custom password char (replacing the default asterisks, in Windows XP, or round dots, in Windows Vista) with a character or symbol of your own choice. This feature not only requires Windows XP or later but also a themed application.

These properties are also available in components that relate to the Edit control, such as LabeledEdit (a combination of an Edit and a Label) and the classic MaskEdit control of the VCL. Alternatively, the DBEdit control doesn't provide the new features of other edit controls. Actually, to be more precise, it inherits the new features from the base `TCustomEdit` class but doesn't expose them in published properties.

In Figure 5 , you can see some of these features (and others I'll explain later) in action:

Figure 5 Runtime Example of Edit Controls



On the left side of the form you can see four edit boxes using some of the new features. The first has its text right aligned, the second displays a text hint, the third allows only numeric input, and the fourth uses Unicode CodePoint 25A0 (Black Square) as its password character. (It is nice that you can use any Unicode symbol for the password character.)

This is the most relevant portion of the DFM file, describing the properties those four controls:

```
object edRightAlign: TEdit
  Alignment = taRightJustify
  Text = 'Text on the right'
end
object edTextHint: TEdit
  TextHint = 'Your name'
end
object edNumber: TEdit
  NumbersOnly = True
  Text = '3'
end
object edPassword: TEdit
  PasswordChar = #9632
  Text = 'password'
end
```

The button close to the first edit lets you switch the alignment property in a round robin fashion, by increasing the value of the enumeration and computing the modulus (the rest of the division) with the highest possible value:

```
procedure TFormEditFamily.btnAlignClick(Sender: TObject);
begin
  edRightAlign.Alignment := TAlignment (
    (Ord(edRightAlign.Alignment) + 1) mod
    (Ord(High(TAlignment)) + 1));
end;
```

THE NEW BUTTONEDEDIT CONTROL

A new control that extends the behavior of the Edit control is the ButtonedEdit component, which is a custom VCL control defined in the ExtCtrls unit. This is basically an edit box that can have small buttons on the left or right side, used to interact with the edit box itself. For example, you can add a Cancel button that empties the edit box, and a search or lookup button that validates the input or looks for some related information. (The Delphi IDE uses this component for the Search option of the Tools Palette.)

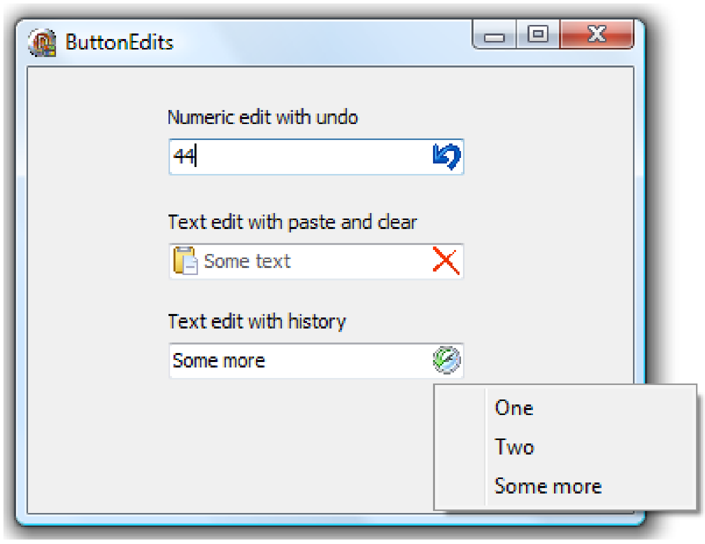
This component (which requires Windows XP or later) includes all of the new features of the Edit control including the modern-looking text hint. Setting up the buttons on the sides of the edit box is quite simple. The component has a `LeftButton` and a `RightButton` property, of type `TEditButton`, defined as:

```
type
  TEditButton = class(TPersistent)
  published
    property DisabledImageIndex: TImageIndex;
    property DropDownMenu: TPopupMenu;
    property Enabled: Boolean;
    property HotImageIndex: TImageIndex;
    property ImageIndex: TImageIndex;
    property PressedImageIndex: TImageIndex;
    property Visible: Boolean;
  end;
```

All of the image references are to the ImageList component that you can hook to the ButtonedEdit control. You can attach a method to the click either button using the `OnLeftButtonClick` and `OnRightButtonClick` events of the ButtonedEdit control. You can also attach a Popup menu to the buttons using the `DropDownMenu` property of the `TEditButton` class.

I've coded some usage scenarios to give you an idea of how to work with this component. These scenarios also show some of the other new features introduced for edit boxes. The main form of the example uses three ButtonedEdit controls, two with a single button and one with two buttons. The controls have also text hints and one of them has a drop-down menu attached. You can see the form at runtime (with the drop down menu active) in Figure 6:

Figure 6 Runtime Example of ButtonEdit



The first control is a numeric edit box with an undo button. The `edUndoRightButtonClick` event handler calls the `Undo` method of the ButtonedEdit control:

```
object edUndo: TButtonedEdit
  Images = ImageList1
  NumbersOnly = True
  RightButton.ImageIndex = 0
  RightButton.Visible = True
  TextHint = 'A number'
  OnRightButtonClick = edUndoRightButtonClick
end
```

The second edit control provides two buttons, one for pasting from the clipboard and the second to clear the edit box content (thus restoring the text hint):

```
object edClear: TButtonedEdit
  Images = ImageList1
  LeftButton.ImageIndex = 3
  LeftButton.Visible = True
  RightButton.ImageIndex = 1
  RightButton.Visible = True
  TextHint = 'Some text'
  OnLeftButtonClick = edClearLeftButtonClick
  OnRightButtonClick = edClearRightButtonClick
end
```

The third edit box has a history button, and keeps track of the text that is entered in the window, allowing a user to reselect it:

```
object edHistory: TButtonedEdit
  Images = ImageList1
  RightButton.DropDownMenu = PopupMenu1
```

```
    RightButton.ImageIndex = 2
    RightButton.Visible = True
    TextHint = 'Edit or pick'
    OnExit = edHistoryExit
  end
```

The component works by adding each new text to the popup menu as the user leaves the edit box, provided this text is not already in the menu:

```
procedure TFormButtonEdits.edHistoryExit(
  Sender: TObject);
begin
  if (edHistory.Text <> '') and
    (PopupMenu1.Items.Find (edHistory.Text) = nil) then
  begin
    PopupMenu1.Items.Add (NewItem (edHistory.Text, 0,
      False, True, RestoreText, 0, ''));
  end;
end;
```

The predefined menu items and each new menu item added dynamically are connected with the `RestoreText` event handler which takes the caption of the selected menu items, strips any hot key, and copies it to the edit box:

```
procedure TFormButtonEdits.RestoreText(Sender: TObject);
begin
  edHistory.Text := StripHotkey (
    (Sender as TMenuItem).Caption);
end;
```

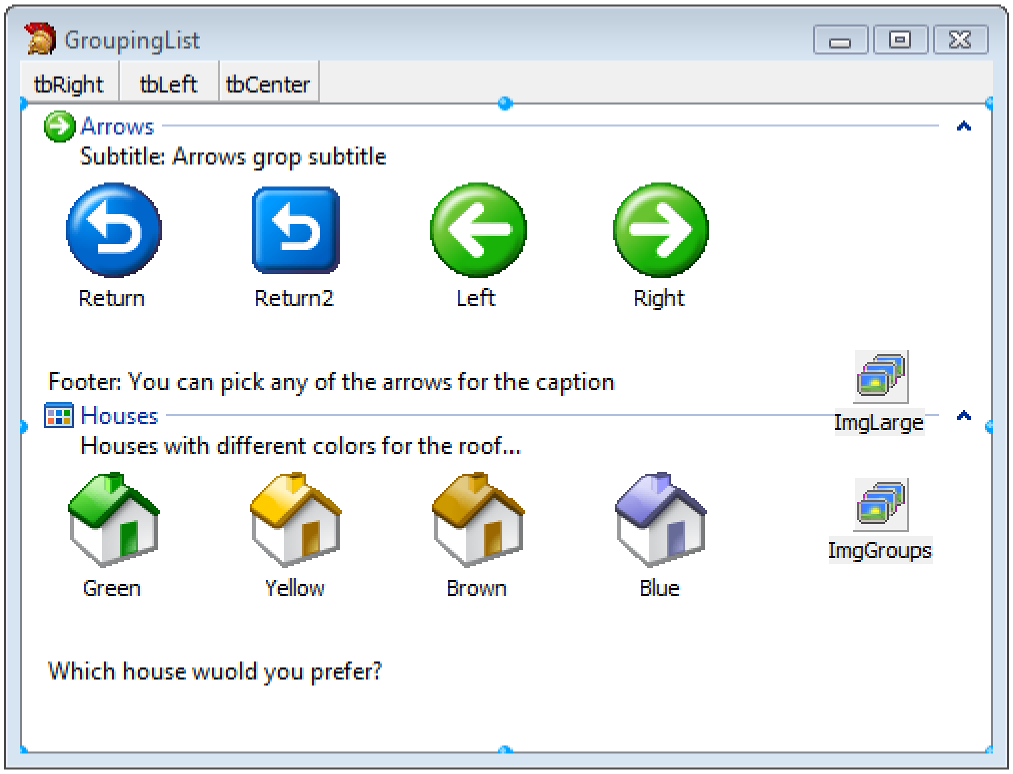
GROUPING IN A LISTVIEW

One common control worth exploring in some more detail is the ListView. In Delphi 2009, that ListView control has direct support for grouping. This feature requires Windows XP or Vista, with the latter providing extended features lacking in the former.

There are three new properties in the ListView control. The Boolean `GroupView` enables this new kind of display, the `GroupHeaderImages` refers to an ImageList containing the images for the group headers, and the `Groups` property is a collection of group definitions. Each group can have a main title (`Header`), a related icon (`TitleImage`), a longer description (`Subtitle`), a footer line (`Footer`), plus some more text elements and alignment properties for headers and footer. A set of options lets you set the group as collapsible, remove the header, hide the group, and so on.

You can see an example of grouping in a ListView in the main form of the following runtime example, displayed in Figure 7:

Figure 7 Runtime Example of ListView Control Enhancements



Subtitle: Arrows grop subtitle

Footer: You can pick any of the arrows for the caption

Houses with different colors for the roof...

Which house wuold you prefer?

This is the definition of the groups inside the ListView control (in DFM format), in which I've set a couple of extra descriptions that will show up only if you center the group headers:

```
object ListView1: TListView
  Groups = <
    item
      Header = 'Arrows'
      Footer = 'Footer: You can pick any of the arrows ' +
        'for the caption'
      GroupID = 0
      State = [lgsNormal, lgsCollapsible]
      HeaderAlign = taLeftJustify
      FooterAlign = taLeftJustify
      Subtitle = 'Subtitle: Arrow group subtitle'
      TopDescription = 'Top Descr: A group of arrows'
      TitleImage = 0
      SubsetTitle = 'Subset title...'
    end
    item
      Header = 'Houses'
      Footer = 'Which house would you prefer?'
      GroupID = 1
      State = [lgsNormal, lgsCollapsible]
      HeaderAlign = taLeftJustify
      FooterAlign = taLeftJustify
      Subtitle = 'Houses with different colors for ' +
        'the roof...'
      TitleImage = 1
      ExtendedImage = -1
    end>
  GroupHeaderImages = ImgGroups
  GroupView = True
end
```

The only code of the example is used to change the alignment of the header and footer of each group. This is the event handler of one of the three toolbar buttons:

```
procedure TFormGroupingList.tbRightClick(
  Sender: TObject);
var
  aGroup: TCollectionItem;
begin
  for aGroup in ListView1.Groups do
  begin
    (aGroup as TListGroup).HeaderAlign := taRightJustify;
    (aGroup as TListGroup).FooterAlign := taRightJustify;
  end;
end;
```

Besides grouping support, the ListView control has another unrelated new event, `OnItemChecked`, triggered when a user selects an item of the ListView.

THE NEW CATEGORYPANELGROUP CONTROL

Over the years, the so-called *Outlook Sidebar* family of components has seen the largest number of VCL controls available, mimicking the well-established interface that was originally introduced by the Microsoft email program.

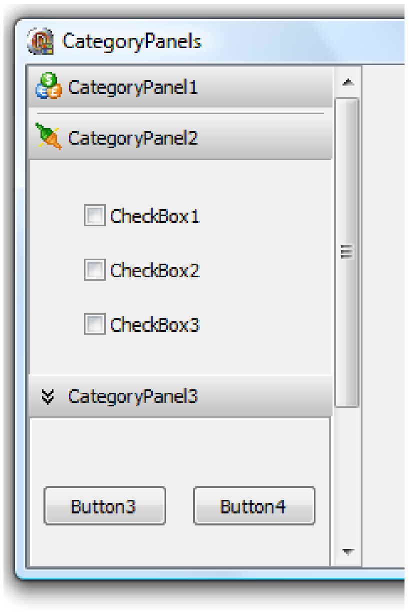
Styles have changed a lot from the original collection of large icons used for the various sections of the program, but the usage of a sidebar with options and commands continues. For the first time, Delphi 2009 offers a similar component out of the box.

The CategoryPanelGroup control is a visual container of CategoryPanel controls. You create these category panels using the shortcut menu of the CategoryPanelGroup at design time or calling its `CreatePanel` method at runtime. The individual CategoryPanels refer to the container using the `PanelGroup` property, while the grouping controls has a `Panel` property (a bare-bones `TList` of pointers) or a list of child controls, in the standard `Controls` property.

If you try adding any other control directly to the CategoryPanelGroup the IDE will show the error "*Only CategoryPanels can be inserted into a CategoryPanelGroup.*" Of course, once you've defined a few CategoryPanels you can add virtually any control you like to them.

Figure 8 shows the user interface of this control, taken from the CategoryPanels demo:

Figure 8 Category Panels Control



The grouping control and the individual panels have a plethora of properties which you can use to customize the user interface, managing headers with multiple images depending on their collapsed or expanded status, activate gradient backgrounds for the headers, change the font and the *Chevron* colors, and much more.

These are the settings of the panels above (from which I've removed details of the hosted controls):

```
object CategoryPanelGroup1: TCategoryPanelGroup
  VertScrollBar.Tracking = True
  HeaderFont.Charset = DEFAULT_CHARSET
  HeaderFont.Color = clWindowText
  HeaderFont.Name = 'Tahoma'
  Images = ImageList1
  object CategoryPanel1: TCategoryPanel
    Caption = 'CategoryPanel1'
    CollapsedImageIndex = 0
    ExpandedImageIndex = 0
    object Button1: TButton...
    object Button2: TButton...
  end
  object CategoryPanel2: TCategoryPanel
    Caption = 'CategoryPanel2'
    Collapsed = True
    CollapsedImageIndex = 2
    ExpandedImageIndex = 1
    object CheckBox1: TCheckBox...
    object CheckBox2: TCheckBox...
    object CheckBox3: TCheckBox...
  end
  object CategoryPanel3: TCategoryPanel
    Caption = 'CategoryPanel3'
    object GridPanel1: TGridPanel
      Align = alClient
      Caption = 'GridPanel1'
      ControlCollection = <...>
      ShowCaption = False
      object Button3: TButton...
      object Button4: TButton...
      object Button5: TButton...
      object Button6: TButton...
    end
  end
end
```

If we look at the header images, the first panel uses the same one for both states, the second uses two different images for the expanded and collapsed states, while the third has no custom images and uses the default *Chevron* symbol. The third CategoryPanel doesn't host its controls directly, but has a GridPanel (with 4 buttons) aligned to its entire surface. This is an example of how you can combine a CategoryPanel with panels providing custom positioning.

IMPROVED GRAPHICS SUPPORT

In the early days, Delphi graphic support was mostly limited to bitmaps. Over the years, there have been extensions to the image formats you could use in the Image component, including JPEG format support. In Delphi 2009, the support for multiple images has been extended to PNG and all formats can now be used with the Image control as well with the ImageList control.

Moreover, the ImageList control supports setting a specific color depth, although increasing its value clears all images from the current image list. There have also been enhancements in the ImageList editor and alpha channel support.

Additionally, the `TBitmap` class now supports the alpha channel, using the new `AlphaFormat` property, while `TGraphic` class has support for transparent images using the `SupportsPartialTransparency` property.

As there are many changes, I've picked a few worth underlining in the GraphicTest program. I'll start with the most significant change which is the native support for multiple formats, including PNG (which is new). The support for the formats comes from a set of units that define inherited `TGraphic` classes that you can selectively include in your application. Here are the units and the graphics classes they make available:

Format	*Unit*	*Class*
JPEG	jpeg.pas	TJPEGImage
GIF	GIFImg.pas	TGIFImage
PNG	pngimage.pas	TPngImage

By including the corresponding unit, you can directly load a file of those formats (plus the standard Bitmap, Icon, and Metafile formats) into an Image component. Because the format is determined by the file extension, you can easily load graphic files with different formats with simple code like:

```
procedure TFormGraphicsTest.btnLoadImageClick(
  Sender: TObject);
var
  strFilename: string;
begin
  case fImgNo of
    0: strFilename := 'adog.jpg';
    1: strFilename := 'Athene.png';
    2: strFilename := 'CodeGear.gif';
  end;
  Image1.Picture.LoadFromFile(strFileName);
  fImgNo := (fImgNo + 1) mod 3
end;
```

The program also has some code to create an empty bitmap in memory. A user can draw on this bitmap by moving the mouse over the image control. The bitmap can then be saved in the three different formats. For example, the code for saving the file in JPEG format looks like this:

```
var
  jpgImg: TJPEGImage;
begin
  jpgImg := TJPEGImage.Create;
  try
    jpgImg.Assign(Image1.Picture.Graphic);
    jpgImg.SaveToFile('test.jpg');
  finally
    jpgImg.Free;
  end;
```

To avoid repeating this code for the PNG and GIF formats, I've written a simple routine to take care of the various differences:

```
procedure SaveWithClass (graph: TGraphic;
  graphClass: TGraphicClass; const strFilename: string);
var
  grapImg: TGraphic;
begin
  grapImg := graphClass.Create;
  try
    grapImg.Assign(graph);
    grapImg.SaveToFile(strFilename);
  finally
    grapImg.Free;
  end;
end;
```

This works only with the default settings, though, as you'll need to work on the specific `TGraphic` descendant class to trigger its compression level and other specific options for the given format. In the demo program, the routine is called like this:

```
SaveWithClass (Image1.Picture.Graphic,
  TPngImage, 'test.png');
SaveWithClass (Image1.Picture.Graphic,
  TGIFImage, 'test.gif');
```

The support for multiple image formats doesn't relate exclusively to the Image component, but it has also been extended to the ImageList component. This means you now have PNG-based image lists. I've already used an ImageList in other demos where I've loaded PNG images from the GlyFX library licensed by CodeGear and included in Delphi (and installed, by default, in the \Program Files\Common Files\CodeGear Shared\Images\GlyFX folder).

Introducing the Fluent User Interface

The Fluent User Interface was invented by Microsoft, who is seeking a patent for it. This patent doesn't focus on the code behind the user interface (the ribbon controls used in Office 2007), but on the design of the user interface itself. Microsoft also refers to this user interface as "Microsoft Office Fluent UI."

If the Microsoft patent is granted, it will still apply even if the VCL implementation available in Delphi 2009 is a brand new version of the controls (in no way related with the code that Microsoft uses in Office and other applications, and that Microsoft doesn't license). That's why we have to look at the "legal side" of this component before looking at its use.

Unlike other guidelines, the Office Fluent UI Design Guidelines, describing how applications based on the Ribbon should work, are not public, but are "Microsoft's confidential information". Microsoft asks anyone that wants to use their Fluent User Interface to accept the terms of their Office UI license. This license is royalty-free, but there are guidelines and limitations related to what you can do. The most significant issue is that you are not allowed to create programs which compete directly with Microsoft Office.

For complete information, refer to the web site mentioned in the dialog box that appears while installing Delphi 2009:

```
http://msdn.microsoft.com/officeui
```

Once you agree with the license and register your application, you'll be able to download the 119-page PDF with the Office UI design guidelines.

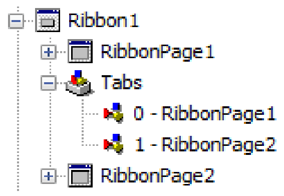
A Simple Ribbon

My first Ribbon example is a very bare-bones demo showing how the component works, but actually providing no real user interface. As we'll see in the next section, the only real way to create a complete Ribbon-based user interface is to use the Action Manager architecture along with it. It is technically possible to use the Ribbon component without Actions, but it is very clumsy and extremely limited... so after a very simple example I'll move in that direction.

We can, in fact, start some initial experiments with a plain Ribbon component, creating tabs and groups, and placing a couple of standard components into them. To follow my steps, create a new application and place a Ribbon component on its main form. Once you have that component in place you can use its shortcut menu (selecting it in the form or in the Structure pane) to add a new tab. The same menu will let you remove a tab or add the Application menu and Quick Access toolbar, as we'll see later on.

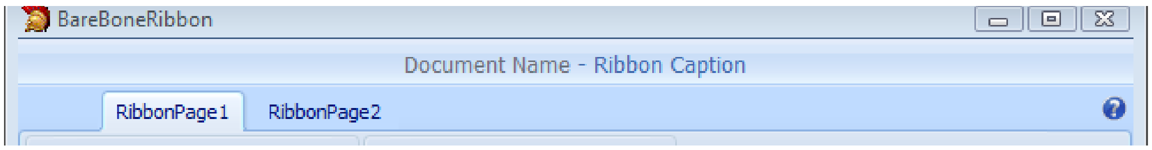
You can also work on the Ribbon Tabs by using the `Tabs` collection of the Ribbon component (technically a collection of `TRibbonTabItem` objects, each of which is connected with a `TRibbonPage`, a sort of panel) and the related AddItem command. This is available in the Structure view as shown in Figure 9Error! Reference source not found.:

Figure 9 Structure View of Ribbon Tabs



The header of a Ribbon with two tabs and pages looks like Figure 10Error! Reference source not found.:

Figure 10 Design Time View of a Ribbon with Two Tabs



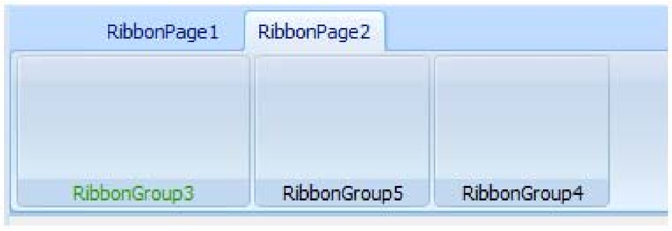
In this case I've kept on the (default) `ShowHelpButton` property that shows the question mark in the top right of the control; I've also kept on the `UseCustomFrame` property (something I'll cover later on). Here are a few other properties of the Ribbon control of the example:

```
object Ribbon1: TRibbon
  Width = 630
  Height = 145
  Caption = 'Ribbon Caption'
  DocumentName = 'Document Name'
  Tabs = <
    item
      Caption = 'RibbonPage1'
```

```
        Page = RibbonPage1
      end
      item
        Caption = 'RibbonPage2'
        Page = RibbonPage2
      end>
    StyleName = 'Ribbon - Luna'
    object RibbonPage1...
    object RibbonPage2...
  end
```

Once you have one or more Ribbon tabs, you can add Ribbon groups (or boxes) to them. Again, you can work with the shortcut menus of the components right in the form or in the Structure pane. Figure 11 shows how a Ribbon page with a few (empty) groups can look:

Figure 11 Runtime Display of a Ribbon Page with Multiple Groups

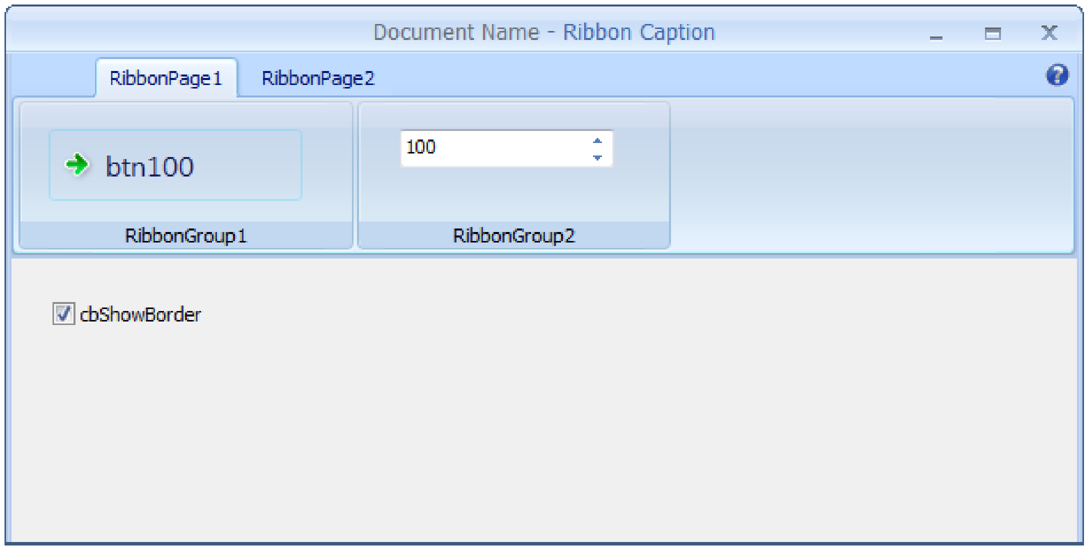


On a Ribbon page, you can add a group, remove a group, or reorder groups, through a simple specific dialog box (which tends to be easier to use, rather than dragging groups around the Ribbon page, hoping they'll stick in place).

What can you place in a group? You generally populate them with elements of various types, from commands to options that are connected with Actions of an ActionManager component. If you want to hack something together, most certainly diverging from the Ribbon UI specification, you can add plain buttons or special purpose RibbonSpinEdit controls to the groups, as I've done in this demo. Again, this is not the recommended approach, although the RibbonSpinEdit control itself does fit with the Ribbon UI specification.

You can see the first two populated pages of my demo at runtime in Figure 12:

Figure 12 Populated Groups on Ribbon Tabs



As you can see, this form is different than the usual one, because its caption and standard borders have been replaced by a special custom frame, painted by the Ribbon control itself. This is the default style for the Ribbon UI, with further graphical elements (like the Application menu) added, as we'll see later.

Actions and the Ribbon

Let's start creating an actual demo application based on the Ribbon and the Action Manager architecture. The first step, of course is to create a VCL application and add an ActionManager component to its main form. Next you can drop a Ribbon control onto the form. The control should automatically hook itself to the action manager; if not, use its `ActionManager` property.

Before adding any action to the ActionManager, add two ImageList controls and connect them to it. Adding standard actions, in fact, will automatically populate the image lists. Add one ImageList for the standard `Images` of the ActionManager (standard images are used for the Ribbon commands) and one for the `LargeImages` property (used by the Ribbon application menu and by large buttons in any Ribbon Group). You should have settings like these:

```
object RibbonEditorForm: TRibbonEditorForm
  Caption = 'RibbonEditor'
  Constraints.MinHeight = 300
  Constraints.MinWidth = 400
  object Ribbon1: TRibbon
    ActionManager = ActionManager1
    Caption = 'RibbonEditor'
    StyleName = 'Ribbon - Luna'
  end
  object ActionManager1: TActionManager
    LargeImages = listLarge
    Images = listStandard
```

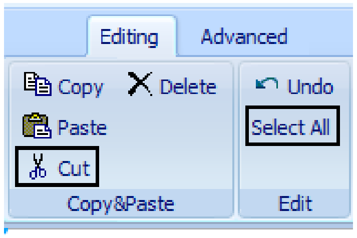
```
      StyleName = 'Ribbon - Luna'
    end
    object listStandard: TImageList...
    object listLarge: TImageList...
  end
```

My goal is to create a simple editor (not a full word processor as I don't mean to infringe on the Ribbon license, you know), so I basically need to place a RichEdit control aligned to the client area of the form and add a good amount of standard actions for editing (the six standard actions of the Edit category), rich edit support (the eight standard actions of the Format category), file support (the eight standard actions of the File category), and a few more (the Download action of the Internet category and the Font action of the Dialogs category).

Groups And Commands

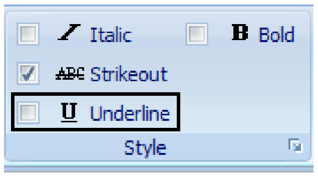
Now that I have all of these actions in place, I can create a Ribbon user interface for them. After creating two tabs and a few groups, I can drag actions into the groups. Figure 13 shows a couple of groups:

Figure 13 Groups in a Simple Editor



These groups host direct commands, so there is nothing specific to set. Another group has a set of non-exclusive options, like setting the text in bold and italic. For the action items of such a group, it is better to pick the `csCheckBox` value for the `CommandStyle` property (rather than the default `csButton`). The effect is to have a set of check boxes you can toggle both by selecting the check area or the icon and the text of the command. Figure 14 is an example from the demo:

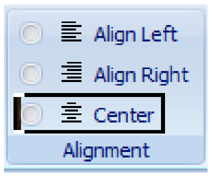
Figure 14 Example of Toggling Checkboxes



The only exception to dragging actions into groups is represented by the Font Dialog action, where I can hook a dialog action to one of the groups, using it's the group `DialogAction` property. This adds a small graphical element in the bottom right corner of the group, as in Figure 14.

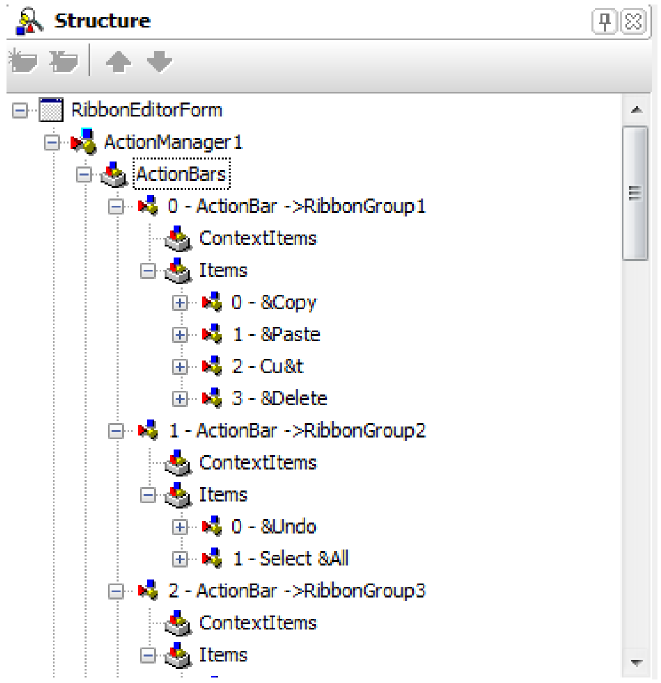
Another option is to have alternative options, represented by radio buttons, setting the `CommandStyle` property to `csRadioButton`, with the visual effect shown in Figure 15:

Figure 15 Alternative Options with Radio Buttons



As you select one the items of a Ribbon group, you'll see the various properties for the corresponding `TActionClientItem` object. But how are these objects managed? It turns out that the ActionManager component has a "toolbar" for each Ribbon group, as you can see in the ActionManager component editor. Even better, you can see the actual internal structure of these objects using the Structure view and expanding the ActionManager component collection, not those of the Ribbon! A small portion of it is shown in Figure 16:

Figure 16 Viewing Ribbon Groups in the Structure View



```
RibbonEditorForm
  ActionManager1
    ActionBars
      0 - ActionBar ->RibbonGroup1
        ContextItems
        Items
          0 - &Copy
          1 - &Paste
          2 - Cu&t
          3 - &Delete
      1 - ActionBar ->RibbonGroup2
        ContextItems
        Items
          0 - &Undo
          1 - Select &All
      2 - ActionBar ->RibbonGroup3
        ContextItems
        Items
```

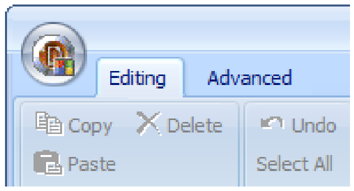
This means you can navigate among the elements in the various Ribbon groups in a less visual but more detailed way, selecting elements that are not visible, picking up small separators, and even adding new ActionClientItem objects. You can configure these new ActionClientItem objects by defining text elements and separators, picking actions, or connecting them to visual controls.

APPLICATION MENU

To complete our application, for which we hooked several custom actions but had to write no actual code, we should add two other relevant elements of the Ribbon user interface. Both are added using commands of the Ribbon component editor (that is, the shortcut menu that appears at design time when the component is selected) and can be added only if an ActionManager is connected to the Ribbon.

The first is the Application menu (the round control element in the top left corner of the Ribbon that replaces the traditional Windows application menu), shown in Figure 17:

Figure 17 Application Menu



This element features an empty drop-down menu. The idea is to use it for file-related operations, and add the standard File, Open, and Save actions to it. You could drag actions to this toolbar, but it's difficult, as it tends to close down. I find it easier to select it in the Structure view, add items, and hook each item to the corresponding action.

If the left of the Application menu is simply a list of file-oriented actions with large icons, the right side should host a list of recently used files. The Ribbon control has specific support for handling this "most recently used" (MRU) list. In this simplified demo I've decided to handle only the Load and Save As operations. Each of them adds an entry to the MRU list by calling a custom method that, in turn, invokes the `AddRecentItem` method of the Ribbon control. This operation adds a new entry at the top of the Recent Documents list, eventually deleting an existing entry referring to the same file name.

The `OnAccept` events of the `FileOpen1` and `FileSaveAs1` actions have the following (similar) code, which calls the custom `AddToMru` method listed below them:

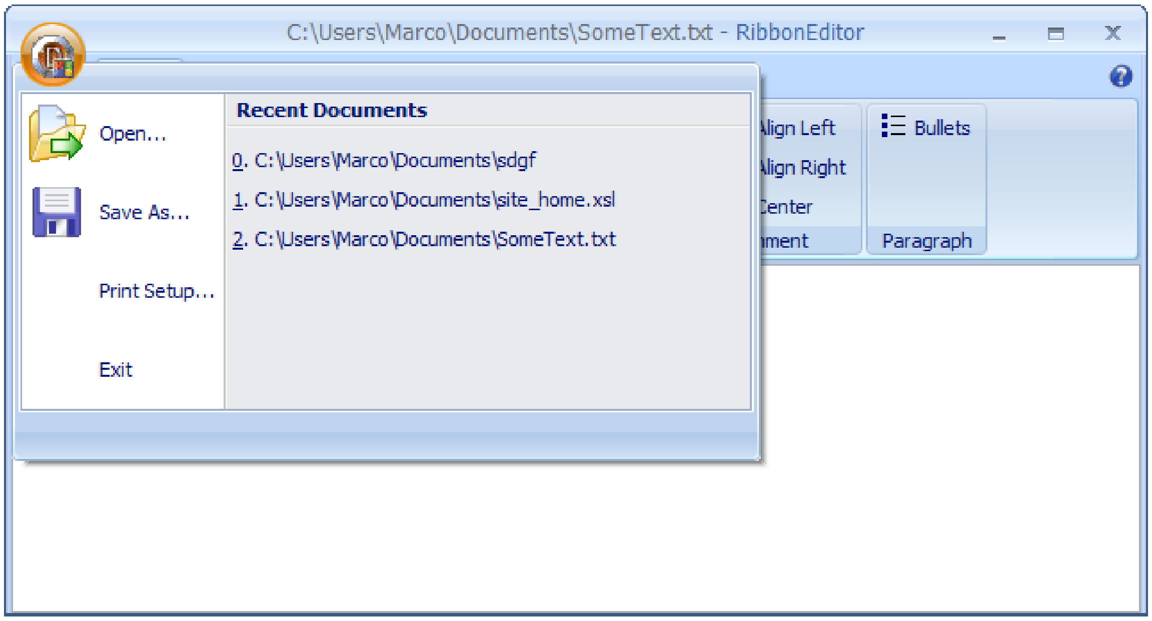
```
procedure TRibbonEditorForm.FileOpen1Accept(
  Sender: TObject);
begin
  RichEdit1.Lines.Clear;
  RichEdit1.Lines.LoadFromFile(FileOpen1.Dialog.FileName);
  Ribbon1.DocumentName := FileOpen1.Dialog.FileName;
  AddToMru(FileOpen1.Dialog.FileName);
end;

procedure TRibbonEditorForm.FileSaveAs1Accept(
  Sender: TObject);
begin
  RichEdit1.Lines.SaveToFile(FileSaveAs1.Dialog.FileName);
  Ribbon1.DocumentName := FileSaveAs1.Dialog.FileName;
  AddToMru(FileSaveAs1.Dialog.FileName);
end;

procedure TRibbonEditorForm.AddToMru(
  const strFilename: string);
begin
  Ribbon1.AddRecentItem(strFilename);
end;
```

When one of the MRU list items is selected, the Ribbon control triggers an `OnRecentItemClick` event handler, which I've coded in a quite naïve way, as it doesn't check if the file is already active in the editor. Also, this information is not saved between sessions. All I wanted to show is how you can manually populate the most recently used list, obtaining an effect such as what is shown in Figure 18:

Figure 18 Most Recently Used (MRU) List


C:\Users\Marco\Documents\SomeText.txt - RibbonEditor
Open...
Save As...
Print Setup...
Exit
Recent Documents
0. C:\Users\Marco\Documents\sdgf
1. C:\Users\Marco\Documents\site_home.xsl
2. C:\Users\Marco\Documents\SomeText.txt
lign Left
lign Right
Center
nment
Bullets
Paragraph

This is the event handler for the MRU list selection:

```
procedure TRibbonEditorForm.Ribbon1RecentItemClick(
  Sender: TObject; FileName: string; Index: Integer);
begin
  RichEdit1.Lines.Clear;
  RichEdit1.Lines.LoadFromFile(FileName);
  Ribbon1.DocumentName := FileName;
end;
```

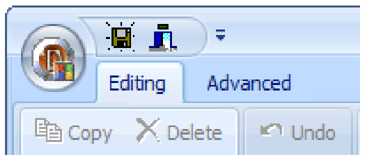
The right page of the Application Menu can also be used to display buttons, by setting the `CommandType` property of the `ApplicationMenu` object of the Ribbon control to `ctCommands` (rather than the default `ctRecent`). In this case, any item added to the `RecentItems` collection appears as a button. This is demonstrated by the Application Menu demo that ships with Delphi 2009, an interesting example as it is similar to the way Office 2007 works.

Quick Access Toolbar

The second graphical element of the Ribbon is its Quick Access Toolbar, a toolbar with operations that automatically managed by the system. This is added on the right of the round Application menu selector, in this case showing a couple of actions (Save As and Exit).

Next to the actions there is also the customize drop-down button that lets a user add extra commands to this toolbar, something quite powerful but that you might want to disable, shown in Figure 19:

Figure 19 Customize drop-down Menu



With these steps I have built a very simple but somewhat complete Ribbon-based editor. Delphi 2009 ships with another example that includes extra features and a nicer looking user interface, but no management of the most recently used files.

CONCLUSION

In this paper I've covered the relevant extensions that Delphi 2009 provides to the VCL library for the development of modern-looking user interfaces. With Delphi 2007 already providing significant support for Windows Vista, Delphi 2009 offers enhancements to several standard controls, from button to edit boxes, so that they'll easily adapt to the latest features provided by the Windows API.

Delphi 2009 offers a ready-to-use Ribbon component, which is compliant with the Microsoft's Office User Interface design guidelines, and leverages the Action Manager architecture that's been in Delphi for some time. Considering the extended graphical support, and the availability of many third-party extensions, you can see why the VCL is considered the best library for building the UI for native Windows applications.

ABOUT THE AUTHOR

This white paper has been written for Embarcadero Technologies by Marco Cantù, author of the best-selling series Mastering Delphi. The content has been extracted from his latest book "*Delphi 2009 Handbook*", http://www.marcocantu.com/dh2009. You can read about Marco on his blog (http://blog.marcocantu.com) and reach him at his email address: marco.cantu@gmail.com.