White Paper

embarcadero®

FireMonkey in Depth: Components and Styles

Marco Cantù, Delphi Product Manager

Embarcadero Technologies

July 2013

Americas 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
Level 2
100 Clarence Street
Sydney, NSW 2000
Australia

FireMonkey in Depth: Components and Styles

This paper offers a deeper look into some of the key elements of the FireMonkey architecture, namely components architecture and the role of styles in this user-interface library. We'll see how styles define controls, look at the different sections of the FireMonkey class hierarchy, see how you can customize individual components, as well as how to write your own. Understanding styles is key to working profitably in FireMonkey, and this is why this paper has such a specific focus.

STYLES IN VCL AND FIREMONKEY

In Delphi XE2, the Embarcadero R&D team added styles to the VCL library, letting you change the look-and-feel of the existing VCL controls. Styles in the VCL are limited to the user interface elements and are applied application-wide. It is a nice feature, but it is clearly an ancillary feature. In fact, it was added very late in the development of this library.

You might consider FireMonkey styles as a related feature. In fact, you can pick a style for an entire application and even change it at run time, changing the look-and-feel and not only adapting the controls to a different set of colors but also changing their graphical design. In it true that global styles in FireMonkey are a key feature in letting the library adapt the same control objects (like Button and Edit boxes) to different platforms and operating systems.

While this is certainly a key feature of FireMonkey, there is in fact more about style and adapting to a platform. Styles are at the heart of the library, are its foundation. Most end-user controls, like button or edit boxes are just... a style! Unlike on Windows or other operating systems, in FireMonkey buttons and edit boxes aren't primitive elements of the library, they are controls made of sub-elements: these composition rules (that is how sub-elements make up a control) are called styles and determine look-and-feel, behavior, properties, and just about everything in the life of a control.

So if you want to understand styles in FireMonkey, you should forget about the VCL analogy or any windows-related tool. You should rather think about HTML and Cascading Styles Sheets. That's where the idea of FireMonkey styles came from, as you can see in this core description:

> *"Controls have properties that exist to be styled. It's like the difference between hard-coding a font color in HTML, and using a style sheet."*

Understanding styles and understanding the internal structure of FireMonkey is basically the same thing: You cannot achieve one without the other, and vice verse. That's why this paper focuses on these two sides at once. We'll delve into the library structure and class hierarchy with the goal of understanding styles. But before we do this I'll introduce you to a FireMonkey demo that makes heavy use of styles: I won't explain all of the details, but we'll use this demo as a reference while we'll explore the internals.

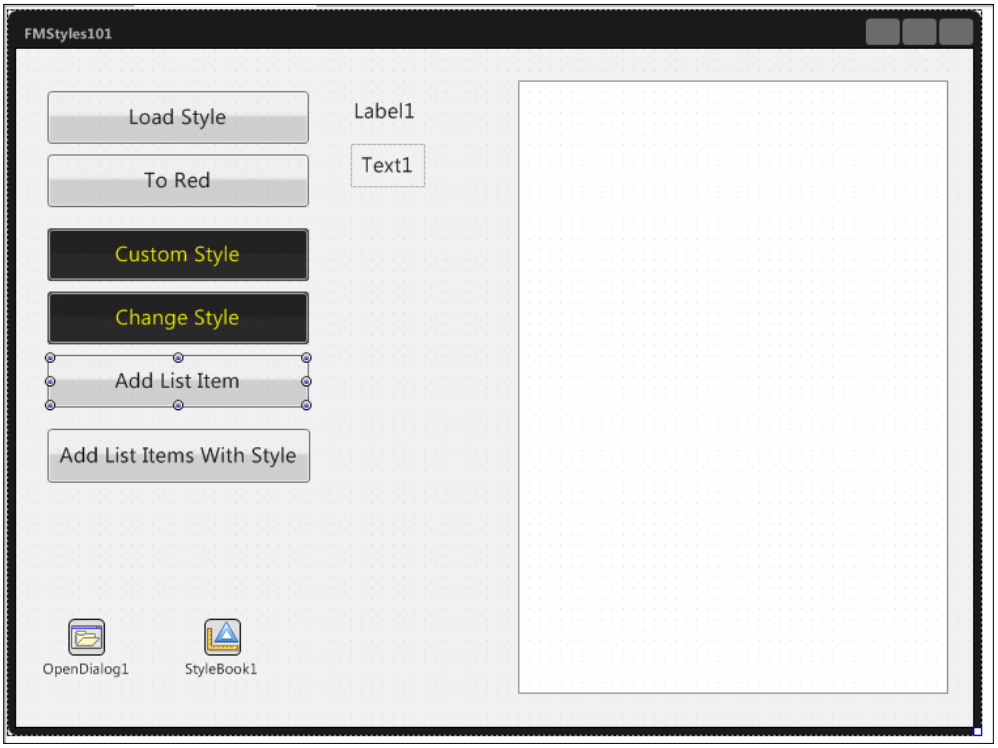
THE FMSTYLES101 DEMO

The introductory example about FireMonkey styles is meant to show you what you can do with styles, even without understanding all of the details. This demo will show you how styles can be used:

- The change the overall look-and-feel of an application, changing the global styles.
- See the difference behavior of a Label control compared to a Text control.
- How you can tweak the "hidden" properties of controls using styles.
- How you can keep properties of multiple controls "in-synch" with a shared style.
- How you can create dynamic sub-controls with a complex structure based on a style.

Notice there are two distinct versions of this demo (almost identical in code, but not in behavior). The first version is called "FMStyles101 Original", while the second is called "FMStyles101 PixelPerfect". I'll explain you in due time the actual differences.

The form of this application is rather simple, as it has a few buttons and a list box control. There are also some non-visual components, the most important being the StyleBook component. Here is the form at design time:



Here is the list of the controls that are part of the form, with the properties omitted (as we'll look at the relevant ones while progressing through the demo:

```
object FMStylesForm: TFMStylesForm
  Caption = 'FMStyles101'
  StyleBook = StyleBook1
  object Label1: TLabel
  object Text1: TText
  object ListBox1: TListBox
  object btnLoadStyle: TButton
  object btnToRed: TButton
  object btnCustomStyle: TButton
  object btnChangeStyle: TButton
  object btnListItem: TButton
  object btnListItemStyle: TButton
  object StyleBook1: TStyleBook
  object OpenDialog1: TOpenDialog
end
```

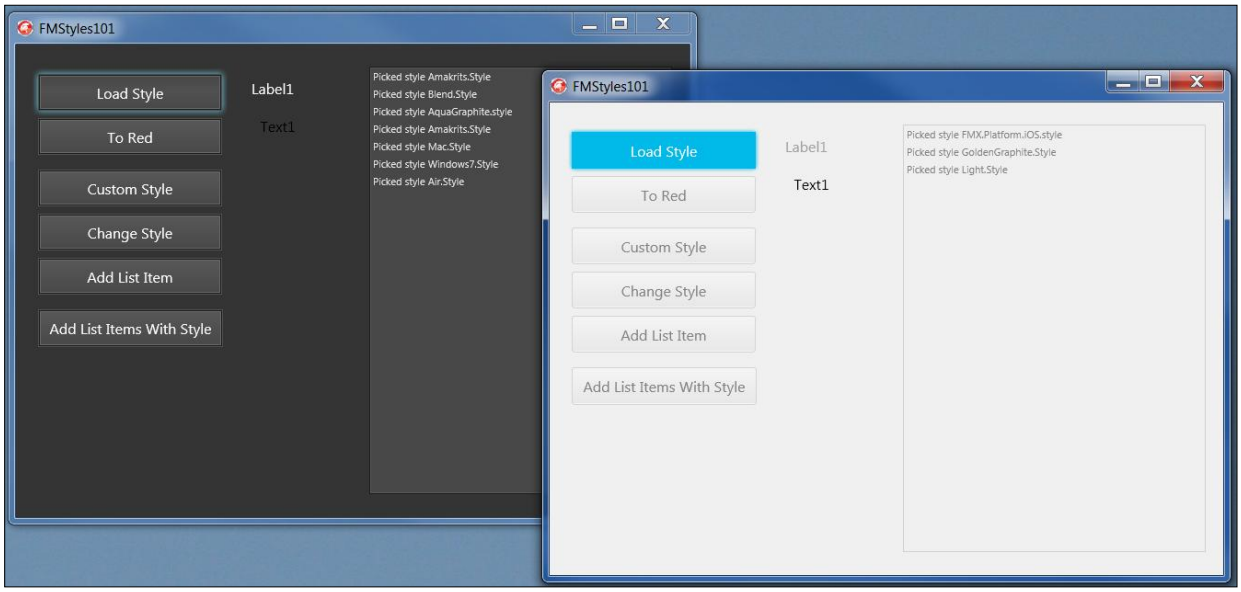
CHANGING THE GLOBAL STYLE

The first button, Load Style, can be used to change the global styles of the application. To accomplish this we need to add a StyleBook component to the form and connect it from the

form's StyleBook property, as show in the previous code listing. We also need an OpenDialog component. Here is the code of the button:

```
procedure TFMStylesForm.btnLoadStyleClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    StyleBook1.FileName := OpenDialog1.FileName;
    ListBox1.Items.Add ('Picked style ' +
      ExtractFileName (OpenDialog1.FileName));
  end;
end;
```

As you can see, when you select a new style (picking an external file, like those in the C:\Users\Public\Documents\RAD Studio\11.0\Styles folder) most of the user interface controls change accordingly:



Using a global style you can change the overall look-and-feel of applications, adapting it to Windows 7, Windows 8, and Mac OSX. Styles are also used for mobile iOS applications, as I'll explain later on. Technically the styles can be loaded from external files or from internal resources, and you can also use the *TStyleManager* global class rather than a *TStyleBook* component to operate on global styles application-wide.

LABEL CONTROLS VERSUS TEXT CONTROLS

The form has a couple of textual elements, but as you can see in the image above, they behave quite differently: One of them adapts to the styles, while the other ignores them. The text of the label gets white on a dark background, while the text of the text control remains black.

These is the initial definition of the two controls (extracted by copying the controls from the Delphi IDE and pasting them here in the document):

```
object Label1: TLabel
  Font.Size = 16.000000000000000000
  Height = 15.000000000000000000
  Position.X = 256.000000000000000000
  Position.Y = 40.000000000000000000
  Text = 'Label1'
  Width = 120.000000000000000000
end
object Text1: TText
  Color = claBlack
  Font.Size = 16.000000000000000000
  Height = 33.000000000000000000
  Position.X = 256.000000000000000000
  Position.Y = 72.000000000000000000
  Text = 'Text1'
  Width = 57.000000000000000000
end
```

The two controls are apparently very similar. Each of them has a size and a position, they both have a font size (given I customized it), they have some text, but only the Text object has a Color property. (More about this in the following section.)

How do these two controls differ? The fact is that for the first I've used a TLabel control, which is style-aware (which technically means it is a descendants of the TStyledControl class) while for the second I've used a TText control, which is style-unaware (which technically means it inherits from TShape or other TControl subclasses). Despite the strict similarity, these two controls come from different parts of the FireMonkey hierarchy and offer a classic example of the difference between styled and non-styled controls. Later on we'll figure out how these two classes are related, because they are indeed closely tied.

COLOR, COLOR OF MY TEXT

If we get back to the demo, we have the problem that the text color of the Text control remains black (while the labels adapts to the current style colors). Now that's quite easy to solve, as we can handle the OnClick event of the control and use the Color property of the control, executing code like this:

```
procedure TFMStylesForm.Text1Click(Sender: TObject);
begin
  if Text1.Color = claBlack then
    Text1.Color := claWhite
  else
    Text1.Color := claBlack;
end;
```

Now you might want to do the same for the label. While in the first version of FireMonkey this wasn't possible, we can now write some very similar code (this is executed as you press the To Red button:

```
  Label1.FontColor := claRed;
```

Sounds easy and works fine? Well, not exactly as the code above doesn't work. The label control has two different ways to pick a color for its text. One is to use whatever the style for the label indicates, the other is to pick the value from the FontColor property. Now, which one of the two options wins? This depends on yet another setting, the value of the ssFontColor element of the StyleSettings property. So to make the code above work, we need to add a second line of code:

```
Label1.StyledSettings := Label1.StyledSettings -
  [TStyledSetting.ssFontColor];
```

Make this experiment: pick a TLabel control at design time, change its FontColor, and look at how this StyleSettings property is modified (notice it is an set of enumerated values, hence the code above to disable one option… even if in the Object Inspector it might look like a series of Boolean values). Now, the specific property wins. If you change the style of the application, the value will be ignored. However, if you enable that StyleSettings again, whatever color the property refers to will be ignores, and the style will win.

For those with an experience in Delphi VCL development, this looks a lot like the ParentColor property. However, while the ParentColor property would let you pick the color of the parent object as default, the StyleSettings let you pick the standard graphical elements of the control from its style, rather than local settings.

I suggest you to test the application and runtime and see how it behaves. You might also want to add to it some code to turn on and off the StyleSettings.ssFontColor value at runtime (while the current code only disables it).

THE COLOR OF THE TEXT OF A BUTTON

We saw that we can change the color of the font of a Label control by picking a different style or by directly changing the FontColor property, but how do you change the color of the font of a Button? Honestly the Button control has that property, only not surfaced at design time. But as I anticipated, we could do that even if this control had no property we could directly use.

The fact is that inside both a Label and a Button there is a Text control (defined by the style structure, and here is where I want to guide you), and that Text control, as we have seen, has the ability of changing its color. I know you might be a bit surprised, but again that ultimately what using styles is about in FireMonkey.

How do we get access to that sub-element? The simplest technique is to call the FindStyleResource method of the main control. For example, we can turn the Button's text color to red by writing:

```
procedure TFMStylesForm.btnToRedClick(Sender: TObject);
var
  aText: TText;
begin
  aText := btnLoadStyle.FindStyleResource ('text') as TText;
  aText.Color := claRed;
end;
```

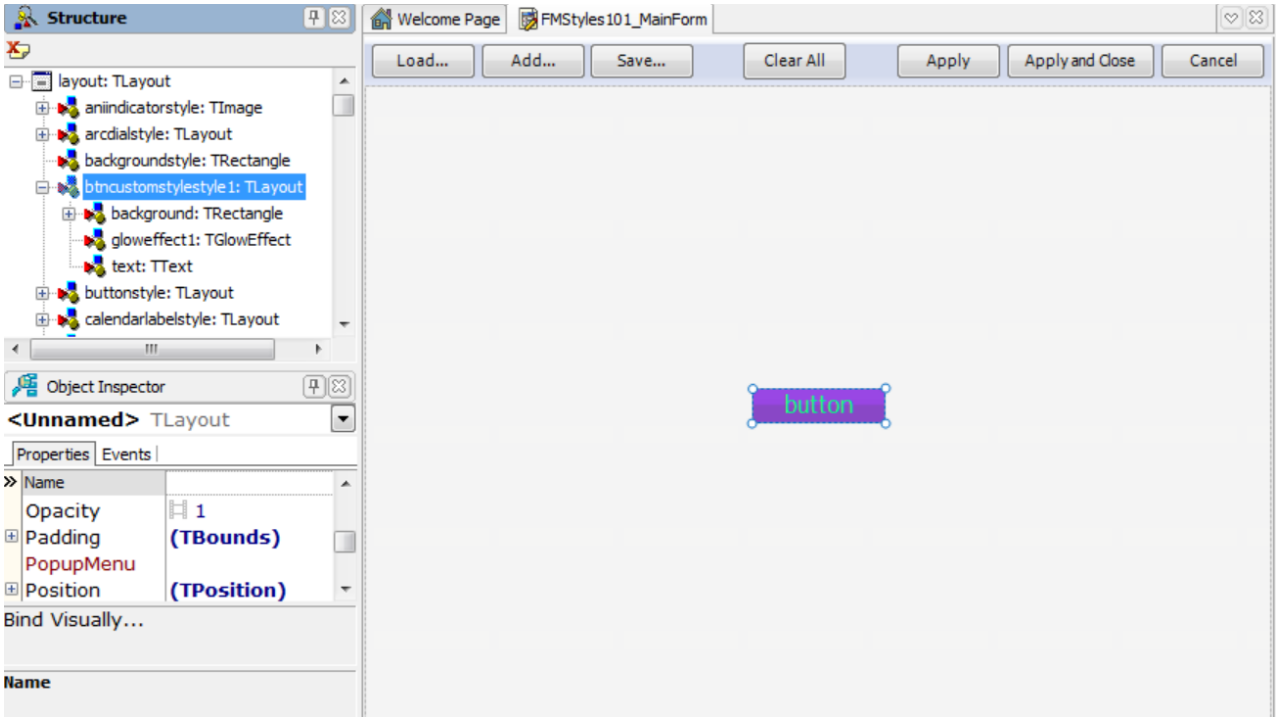
The control will change its color, but only temporarily: in fact, if you work on the button, its style will change depending on the status and this setting will be overridden. The same won't happen if we use code similar to what we have done for the TLabel (as we disabled the corresponding style setting).

SHARING THE STYLE AMONG BUTTONS

Another interesting feature related with styles, is the ability to share a custom style among a group of controls. To obtain this effect pick one of the two controls at design-time, select the Edit Custom Style command of the local menu: This will bring you to the Style Editor, connected with the StyleBook we have already placed in the form.

Note: This is where things change significantly if you use the two different versions of the demo. If the style in use is one of the newer bitmap styles (or pixel-perfect styles) its internal structure will be based on graphical elements and would be more difficult to manipulate. This is while I suggested you to start with the "Original" version of the demo, for which I selected a vector-based style in the StyleBook control.

This is the Style Editor, after you add a custom style for one of the button controls:



You can see a custom style has been added, called *btncustomstylestyle1* (the name is formed by the control name, *btnCustomStyle*, followed by *Style1*, but it is just a name and you can change it as you prefer). What you can see on the left is the list of the elements that make up the style, a concept we'll get back to later in more details. In practice, these are the elements of the style and here we can see that the Button (in case of a vector-based style) has indeed a text element (after a background and a glow effect). We can select it and change its color to, say, yellow. I've also changed the background to black and picked a larger font. Remember to choose the Apply button (or Apply and Close) to save your settings.

Now the button in the designer will have this internal code:

```
object btnCustomStyle: TButton
  Font.Size = 16.000000000000000000
  Height = 40.000000000000000000
  Position.X = 24.000000000000000000
```

```
    Position.Y = 136.000000000000000000
    StyleLookup = 'btnCustomStyleStyle1'
    TabOrder = 3
    Text = 'Custom Style'
    Width = 200.000000000000000000
  end
```

As you can see there is no reference to the colors, but the button will look purple and yellow already at design time. The fact is the information about the style is never saved inside the control, but the control has a reference to the name of the style used and the style itself is stored in the style book. The property in question is StyleLookup. In fact we can make the following button look exactly like this one by setting the property to the same value, but we can switch the style of the button with the following code:

```
procedure TFMStylesForm.btnChangeStyleClick(Sender: TObject);
begin
  if btnChangeStyle.StyleLookup = 'btnCustomStyleStyle1' then
    btnChangeStyle.StyleLookup := 'buttonstyle'
  else
    btnChangeStyle.StyleLookup := 'btnCustomStyleStyle1';
end;
```

Now that we have a custom style for two button, or actually for two of them, we can change some of their properties by modifying the re-applying the related style. But how do we change a style and, before that, what exactly is a style? In the past (until Delphi XE3), we could look at definition of a style by opening the text of the DFM file hosting the StyleBook control. In XE4, the style information is saved in a much more efficient binary format, so this isn't so simple to look into. However, as a workaround I published a small plug-in to the Delphi IDE (available on CodeCentral at cc.embarcadero.com/item/29428).

Note: This design-time package adds a TStringList editor to the Resources property of the FireMonkey StyleBook component, making it easy to edit the textual representation of a style directly.

Let's look at this small fragment of the StyleBook definition for this custom style:

```
  object TLayout
    StyleName = 'btnCustomStyleStyle1'
    DesignVisible = False
    Height = 24.000000000000000000
    Width = 91.000000000000000000
    object TRectangle
      StyleName = 'background'
      Align = alContents
```

```
      Fill.Color = claBlueviolet
      XRadius = 3.000000000000000000
      YRadius = 3.000000000000000000
      object TRectangle
        Align = alContents
        Fill.Kind = bkGradient
        Stroke.Color = xC84F4F4F
      end
      object TColorAnimation
        Trigger = 'IsMouseOver=true'
      end
      object TColorAnimation
        Trigger = 'IsMouseOver=false'
      end
      object TInnerGlowEffect
        GlowColor = xFF4F4848
        Trigger = 'IsPressed=true'
      end
      object TRectangle
        Align = alClient
        Fill.Kind = bkNone
        Stroke.Color = x96FCFCFC
      end
    end
    object TText
      StyleName = 'text'
      Align = alClient
      Color = claSpringgreen
      Font.Size = 16.000000000000000000
      Text = 'button'
    end
    object TGlowEffect
      GlowColor = x82005ACC
      Trigger = 'IsFocused=true'
    end
  end
```

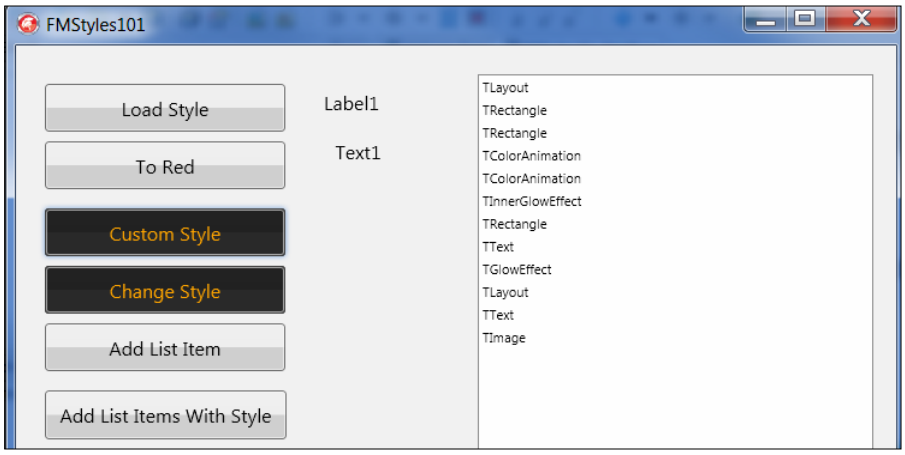
Surprised? The content of the Resource property of the StyleBook looks really like an XFM file, the FireMonkey versions of DFM files of the VCL. Because this is what a style actually is in practical terms! It is a collection of controls defined exactly like you do in a designer, the system can use to re-create those controls at run-time. And this is also the content of an external .style file.

Again, in XE4 these files are saved in an optimized format, to reduce the size and optimize loading time, a particular important issue for mobile platforms.

Let me explain a little more. As the StyleBook is loaded, it creates an internal controls structure, much like a form does when loading its textual description. This structure is not visible (it is kept in memory and has no visual container), but we can navigate it following the parent child relationship or, for simple scenarios, getting the top list of component owned by the StyleBook. This is what I've done in the following code, which enumerates the sub-elements of the style and changes the text color of the style, re-applying it to both buttons:

```
procedure TFMStylesForm.btnCustomStyleClick(Sender: TObject);
var
  child: TComponent;
begin
  for child in StyleBook1.Style do
  begin
    ListBox1.Items.Add (child.ClassName);
    if child is TText then
    begin
      if TText(child).StyleName = 'text' then
        TText(child).Fill.Color := claOrange;
    end;
  end;
  (Sender as TButton).UpdateStyle;
  btnChangeStyle.UpdateStyle;
end;
```

This code changes the button's text color (not much visible in the printed document) but also lists sub-elements of the style:



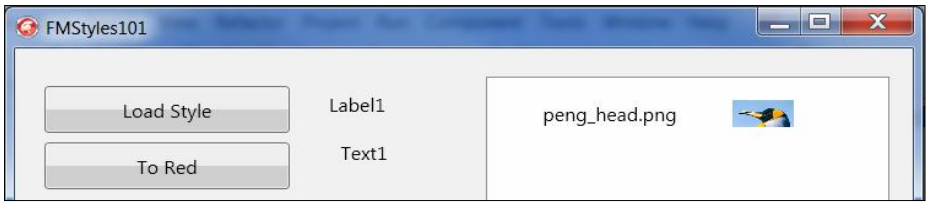
By modifying a style we affect all of the controls tied to that style at once. Notice how this is way more flexible that using the ParentXxx properties and component messages in the VCL to change the look-and-feel and multiple child controls at once.

ADDING STYLES LIST BOX ITEMS

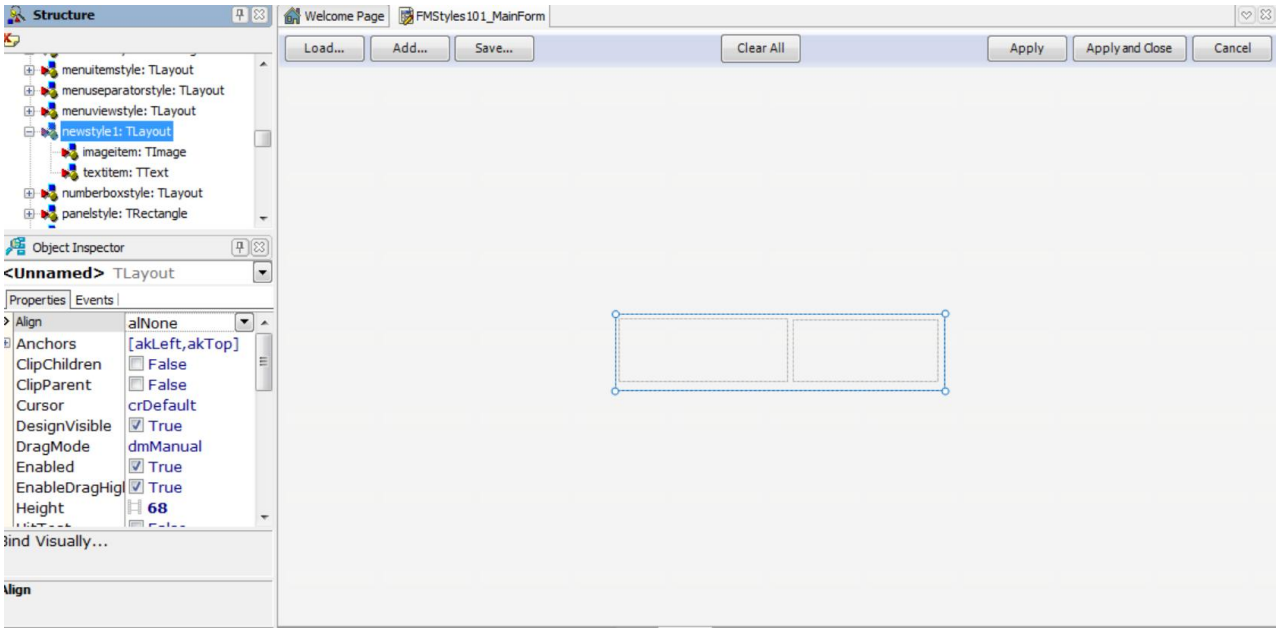
The last example of using styles I want to focus on, before we delve into the internals, is the definition of a custom style. There are many reasons for creating custom styles and one of them is to determine the structure of the sub-controls of an existing control. In the specific example, I want to add complex items to the list box, so each list box item has a text and an image sub-elements. This can be achieved by writing code like the following:

```
procedure TFMStylesForm.btnListItemClick(Sender: TObject);
var
  listItem: TListBoxItem;
  itemText: TText;
  itemImage: TImage;
begin
  // create a new custom listbox item
  listItem := TListBoxItem.Create(ListBox1);
  listItem.Parent := ListBox1;
  listItem.Height := 150;

  itemText := TText.Create (ListBox1);
  itemText.Parent := listItem;
  itemText.Position.X := 5;
  itemText.Position.Y := 5;
  itemText.Width := 190;
  itemText.Text := 'peng_head.png';
  itemText.Font.Size := 16;

  itemImage := TImage.Create(ListBox1);
  itemImage.Parent := listItem;
  itemImage.Position.X := 200;
  itemImage.Position.Y := 5;
  itemImage.Bitmap.LoadFromFile ('peng_head.png');
  listItem.Height := itemImage.Bitmap.Height;
end;
```

This results into a rather interesting output, something much harder to obtain in a VCL application:



Rather than this hard-coded and very un-flexible solution, we can opt for using styles. We can define the sub-elements of the list box items by defining a new style, a layout hosting these sub-controls. We have to do this in the Style Designer, as in the following image:



I've obtained this structure by dragging a Layout over the top node of the style, giving it a name, and dragging a Text element and an Image element to the surface of the new layout. Remember to change the StyleName property of the sub-elements to TextItem and ImageItem, as those are going to be referenced in the code.

This results in the following lines being added to the StyleBook data (again, extracted using the custom viewer mentioned earlier):

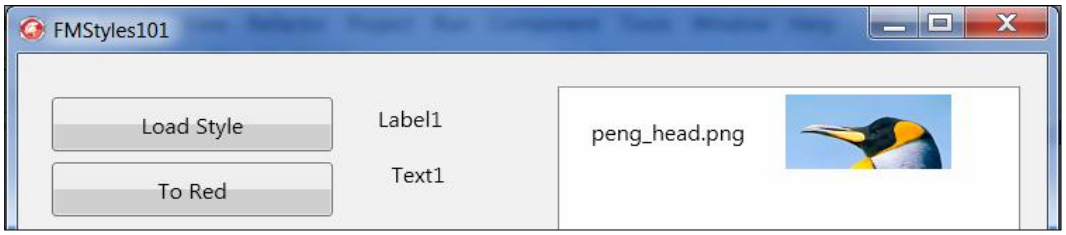
```
object TLayout
  StyleName = 'newstyle1'
  DesignVisible = False
  Height = 68.000000000000000000
  Position.X = 204.000000000000000000
  Position.Y = 313.000000000000000000
  Width = 290.000000000000000000
  object TText
    StyleName = 'TextItem'
    Color = claBlack
    Font.Size = 16.000000000000000000
    Height = 56.000000000000000000
    Position.X = 3.000000000000000000
    Position.Y = 4.000000000000000000
    Width = 149.000000000000000000
```

```
    end
    object TImage
      StyleName = 'ImageItem'
      Height = 55.000000000000000000
      Position.X = 156.000000000000000000
      Position.Y = 5.000000000000000000
      Width = 128.000000000000000000
    end
  end
```

Now in the code I can create a list box item and customize its structure by applying the style. This will cause the list box item to create the image and text sub-components that are present in the style, much like you create components by reading a DMF or FMX file while a form is created. At that point we can refer to the style and the style sub-elements by name, as we did earlier to access to the text of a Button or Label:

```
procedure TFMStylesForm.btnListItemStyleClick(Sender: TObject);
var
  listItem: TListBoxItem;
  itemText: TText;
  itemImage: TImage;
begin
  // create a new custom listbox item
  listItem := TListBoxItem.Create(ListBox1);
  listItem.Parent := ListBox1;

  // force the items style, creating sub-elements
  listItem.StyleLookup := 'newstyle1';

  // customize the text
  itemText := listItem.FindStyleResource ('TextItem') as TText;
  if Assigned (itemText) then
    itemText.Text := 'peng_head.png';

  // customize the image
  itemImage := (listItem.FindStyleResource('ImageItem') as TImage);
  if Assigned (itemImage) then
  begin
    itemImage.Bitmap.LoadFromFile ('peng_head.png');
    listItem.Height := itemImage.Bitmap.Height;
  end
  else
    ShowMessage ('Image binding element not found');
end;
```

The image added to the list is more balanced, as it was much easier to obtain it by designing the position and width of the control in the Style Editor, rather than writing the coordinates in the source code:

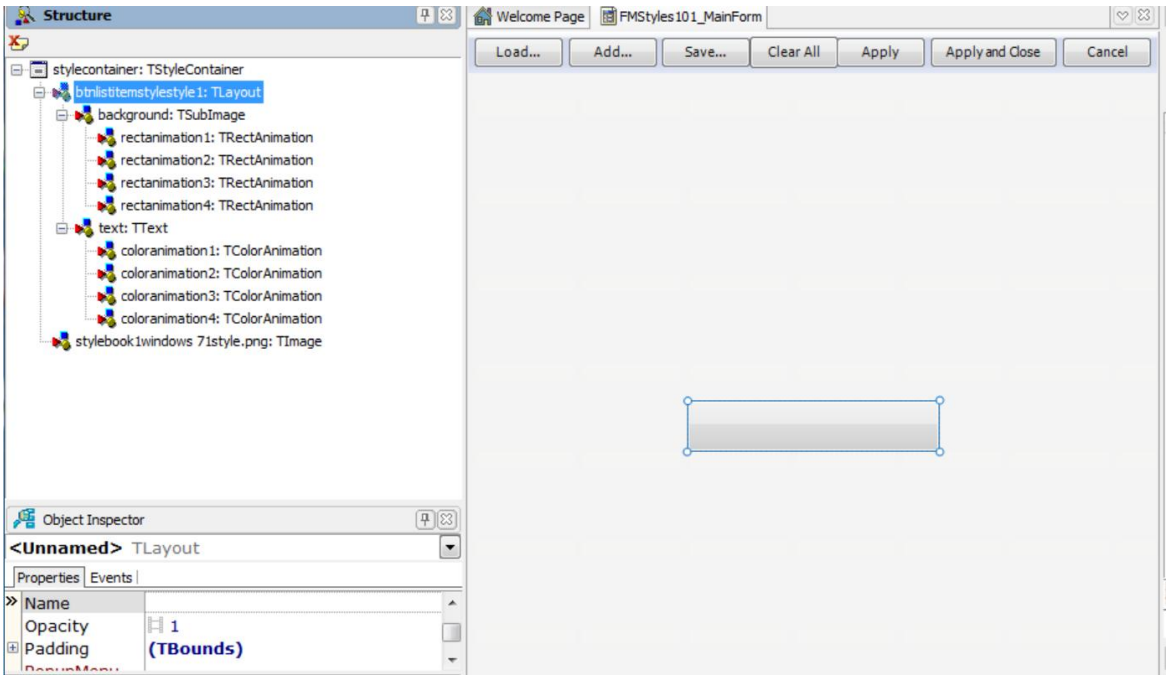


Now there is one more element to notice if we want to compare the two techniques we have used to customize the list box items. In the first case, the position, size and other elements of the controls were hard coded in the Pascal source code. In the second case, they were embedded in a resource part of the XFM file, so they were still somehow hard coded. But the program could as well load the style definitions from a resource file, maybe a file prepared by a graphic designer. Exactly as it happens when you build an HTML application and use a Cascading Style Sheet, as this is the metaphor FireMonkey styles are based on.

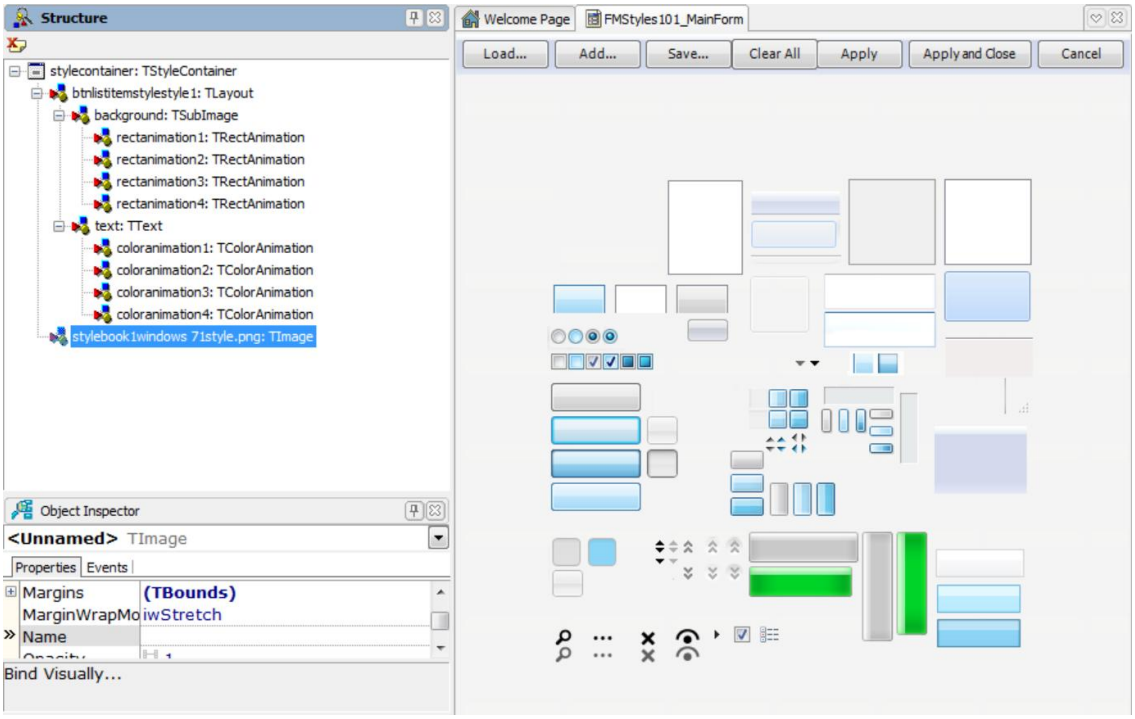
But what happens behind the scenes? After this rather long, but hopefully mind-opening demo, it is time to delve into the library itself.

What about Bitmap Styles?

In a separate version of this same demo, I've not loaded a custom vector-based style but kept the default platform style, which is a bitmap-based style. For such a style, the entire structure is based around a very large bitmap hosting all of the various graphical elements that make up controls, and some style elements, many of which refer to portions of the given bitmap. For example, if we open the style for a button, we'll see a very different structure:

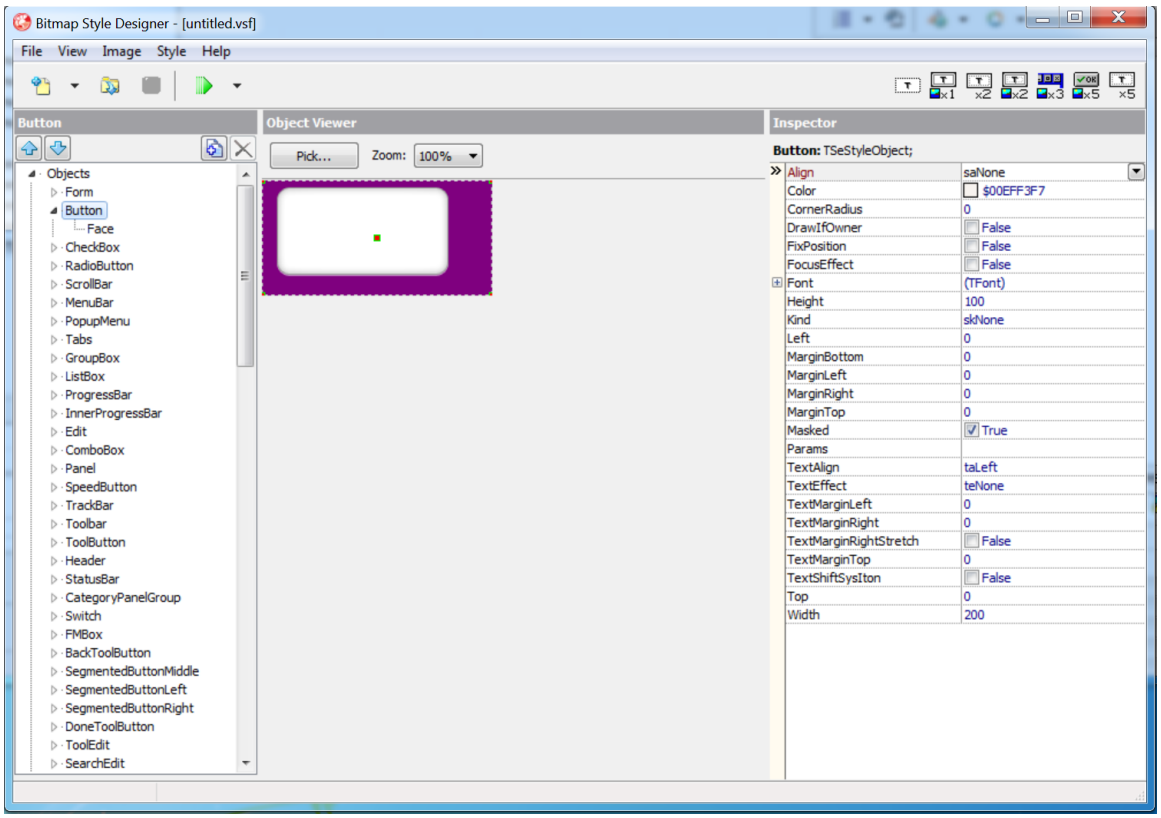


The TSubImage elements refers to a section of the entire style bitmap, shown below for this example:



To work on bitmap styles the recommended tool is not the embedded FireMonkey style editor, but the stand along Bitmap Styles editor you can find linked in the Tools menu. This is based on the concept of the single large PNG image for all of the style elements, plus specific

customizations for several controls and their graphical appearance. Here, as an example, is how you can customize a button in a “iOS-ready” style:



THE FIREMONKEY CLASS HIERARCHY

As it should be clear from this introductory demo, styles are a key element of the FireMonkey architecture and to understand them we need to have a look to the class hierarchy of the library and start looking into its internals. Good that, as it happens for the VCL, we have the full source code to guide us!

THE CORE COMPONENT AND THE CORE INTERFACES

At the top of the FireMonkey library there is a single class, which inherits from Delphi's RTL TComponent class. Everything component in FireMonkey is Delphi component, uses the same streaming mechanism, the ownership model, and just about everything you know about TComponent (and its base classes, TPersistent and TObject). There are two entities at the top of the FireMonkey hierarchy, a class and an interface:

This top FireMonkey class is called TFmxObject and it is used for creating, destroying and releasing object, for cloning, storing and loading them from a stream, and for manipulating child objects (like adding, removing, or searching them). The class has also support for features not found in the VCL, like custom resources and animations.

The top element the FireMonkey hierarchy is not a class, but an interface, called IControl. The reason why this is not a class, is that the library has two different branches for HD and 3D elements. In fact, the IControl interface is implemented by two different classes, TControl and TControl3D. Here we are focusing only on the HD model (the only one based on styles), but the internal structure of the library accounts for both. The IControl interface has methods and event for handling the input focus, mouse and keyboard events.

HOSTING SURFACES: FORMS, SCENES, AND VIEWPORTS

Although this is less key to our discussion on styles, another fundamental elements of the library is how hosting surfaces or root controls containers are managed. There are three interfaces:

- IRoot is the interface implemented by top-level containers and handles focused controls and active ones. It is implemented by all forms, TForm and TForm3D.
- IScene is the interface defining a container for 2D objects, and it is implemented by TForm but also by TCustomLayer3D, a 3D control you can use to host 2D elements.
- IViewport3D is the interface defining a container of 3D objects, and (symmetrically to the previous one) is implemented by TForm3D and by TViewport3D, a 2D controls that can host 3D elements.

Again, this is important in general to understand FireMonkey, but it not tied to styles. What is very important to notice anyway, is that a form is a FireMonkey controls container, but it isn't directly a control:

```
type
  TCommonCustomForm = class (TFmxObject, IRoot,
    IContainerObject, IAlignRoot)
```

This is completely different from the VCL, where a forms is a special TControl descendant, and where all windows (forms, edit boxes, buttons) are treated equally by the Windows operating system. This also explains why form in FireMonkey have quite limited features: it is often better to have a Layout as only child of a form and work on it instead. In this case, you can easily move that Layout and its children to a different host (like a tab) while hosting a form in a FireMonkey application is not a common operation. In other words, there is no nesting for forms. But an internal Layout can also add scaling and other standard behaviors.

There is also no KeyPreview property or OnKeyPress event at the form level, but you can override the KeyDown method (and other related methods). Again, forms have fewer events and properties and play a less relevant role in FireMonkey. They are there as a conduit between the library itself (which takes control of their internal surface) and the hosting operating system, but given this host can vary given the cross-platform nature of the library, forms offer only the basic common features.

THE TCONTROL CLASS

If these core FireMonkey classes are relevant, our specific focus related with styles starts with the TControl class (again, the TControl3D has no styles support and follows different rules). This class implements quite a few interfaces:

```
type
  TControl = class (TFmxObject, IControl, IContainerObject,
    IAlignRoot, IAlignableObject)
```

Needless to say this is an abstract class, one you don't create instances of. But it also also still quite high in the hierarchy, as you don't even inherit from it directly. It provides the interface and some of the actual implementation of the idea of “visual control” in FireMonkey. Its features include:

- **A parent / child mechanism for managing controls.** Differently from the VCL and from Windows, a child can be outside of the client area of its parent, but its coordinates will always related to the later. Beside properties like Parent, ChildrenCount, and Children, the class has clipping support with the CanClip, ClipChildren, and ClipParent properties. By default, clipping is disabled.
- **Size and positional properties**, like Position, Width, and Height, but also alignment, margins, padding and other properties affecting the actual position. The position is always relative to the parent control.
- **Properties related with the visual appearance** (and that are passed to child controls), like RotationAngle, RotationCenter, Scale, and Opacity. If you set the rotation or the opacity of a control, this equally affects its child controls.
- **Event and methods for mouse interaction**, hit testing, focus control and for keyboard interaction. You click on a control or type into one. One control has the focus, much like in Windows. Nothing unusual, for most parts.
- **Triggers for animations** (generally tied with events). This is something I don't want to focus on here.

- **Complete painting support**, through an internal Canvas property and many support objects like Fill and Stoke bitmaps. This is a relevant topic I'll focus next.

PAINTING ON A CONTROL CANVAS

Any type of control in FireMonkey supports direct painting. As we saw in the “Introduction to FireMonkey”, depending on the current platform the library will create a specific canvas to support Direct2D, OpenGL, or some other GPU platform. Suffice to say for our purpose here that the canvas painting code is completely platform independent. Not only, but painting in FireMonkey is also quite similar to its VCL counterpart. You can paint on a form Canvas or on control Canvas, which is generally preferred. You can also paint on an image and the like.

Traditional Delphi GDI painting uses Brush and Pen objects, with most drawing operations using both. For example, as you draw a rectangle its border will use the current pen while its interior will use the current brush.

FireMonkey uses a different model, that was already in Delphi's TCanvas2D class. You have a Fill object (more or less a Brush) and a Stroke object (more or less a Pen). All graphical operations are split, as you either draw the border or fill the interior, with methods like FillRect and DrawRect. If you want both effects, do two calls passing the same area and similar parameters. Finally, notice that you draw text with a FillText call, meaning you can use a brush with gradients and other effects for the text.

In general, you should paint only within OnPaint event handlers. If this is true also on Windows, or you screen image will go away, this is even more obvious in FireMonkey, since it has no effect at all.

As an example, I've added a a FireMonkey HD form a PaintBox (which is a low-level non-styled control) and a Label (a style control). Both support direct painting. This is the code for the OnPaint event of the PaintBox control, which is generally a very good choice for a standard custom painted surface.

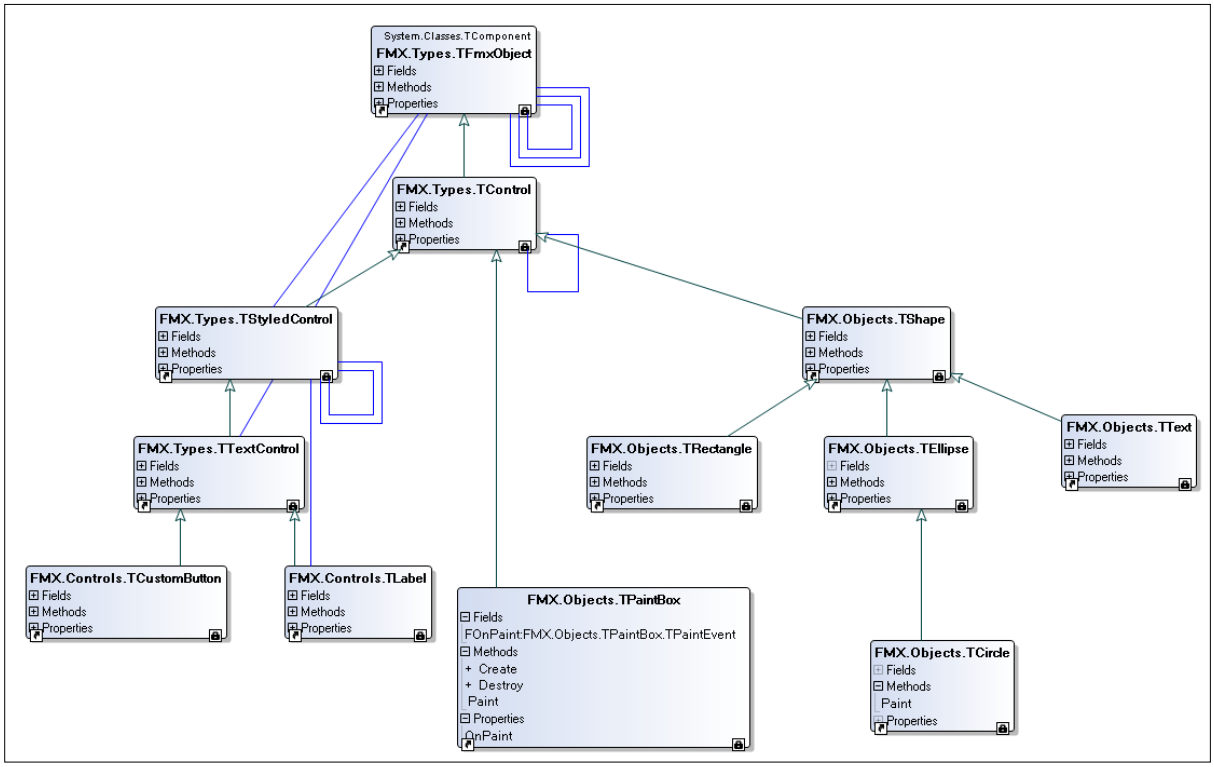
```
procedure TCanvasPaintForm.PaintBox1Paint(Sender: TObject; Canvas:
TCanvas);
begin
  PaintBox1.Canvas.Stroke.Color := claBlueviolet;
  PaintBox1.Canvas.Stroke.Kind := TBrushKind.bkSolid;
  PaintBox1.Canvas.DrawRect(
    RectF(100, 10, 280, 120),
    20, 20, [TCorner.crBottomLeft, TCorner.crTopRight], // rounded
    1); // opacity
```

```
    PaintBox1.Canvas.Fill.Kind := TBrushKind.bkGradient;
    PaintBox1.Canvas.Fill.Gradient.Color := claRoyalblue;
    PaintBox1.Canvas.Fill.Gradient.Color1 := claRed;
    PaintBox1.Canvas.FillRect(
      RectF(200, 10, 580, 120),
      0, 0, [], 0.6); // no rounded corners and partial opacity
  end;
```

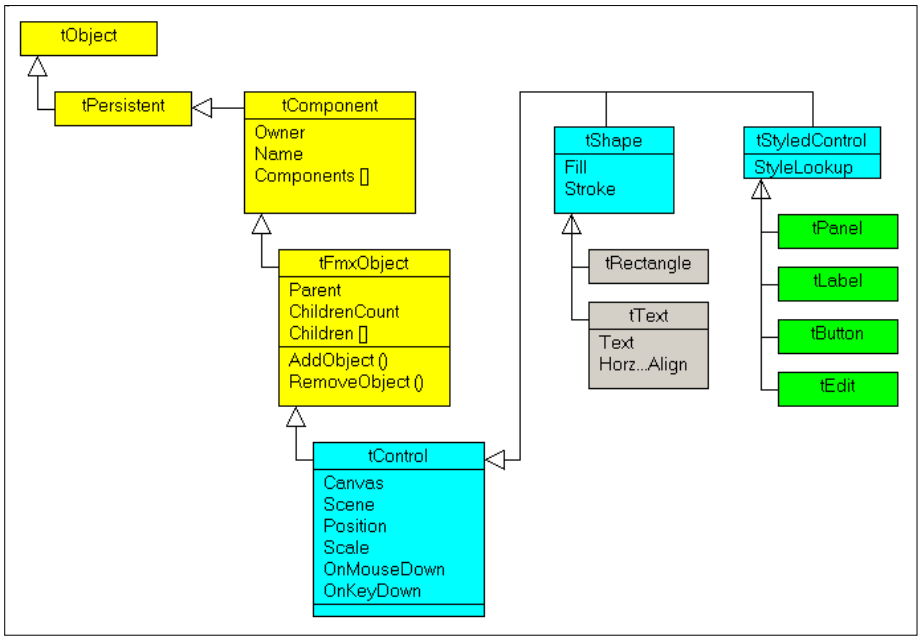
Notice that the two images partially overlap, but given the second has partial opacity, the first will show behind it. The label's OnPaint event has similar code, and again it is partially painted behind a button: Click on the button to change its opacity and the line will show even if behind it. Notice that painting on the label is far from optimal, as we are using hard-coded colors but we don't know which style the component will use at run time. In general, you should not paint on a control. If you want to have a rectangle covering a label use the corresponding primitive FireMonkey control, rather than painting directly. Only primitive controls should generally paint themselves.

The Hierarchy of Control Classes

As I've mentioned a few times by now, there are two primary branches under the TControl class in FireMonkey, styled controls (that is, the TStyledControl class) and everything else, with most of the other controls inheriting from the TShape class, but not all of them. The following class hierarchy, made with Delphi's UML designer, shows some of these classes:



Another representation, which is probably more clear, is the following I've borrowed from from French Delphi expert Felix Colibrì (see a larger version at http://www.felix-colibri.com/papers/firemonkey/firemonkey_architecture/firemonkey_architecture.html):



On the other hand I find the FireMonkey class hierarchies included in the Delphi help very confusing. Let's focus on shapes first and move to styled controls later on.

Shapes or Primitive Controls

Shapes are a collection of controls that paint different graphical elements on screen. Classes include TRectangle, TText, TEllipse, but also TLine, TPie, TArc, and TPath.

Basically shapes override TControl.Paint and show some content on the screen. They have Fill and Stroke settings, initialize canvas in the base class, before the call to the Paint method, so descendant classes only need to override Paint and use the available Canvas. Here is an example, the painting of a Rectangle control:

```
procedure TRectangle.Paint;
var
  R: TRectF;
  Off: Single;
begin
  R := GetShapeRect;

  if Fill.Kind = TBrushKind.bkGrab then
    OffsetRect(R, Position.X, Position.Y);

  if Sides <> AllSides then
    ...
  else
  begin
    Canvas.FillRect(R, XRadius, YRadius, FCorners,
      AbsoluteOpacity, CornerType);
    Canvas.DrawRect(R, XRadius, YRadius, FCorners,
      AbsoluteOpacity, CornerType);
  end;
end;
```

Slightly different from shapes there are primitive controls like TPaintBox and TImage. However, primitive FireMonkey elements include also colors, effects, and other groups of classes. Basically primitive elements are all of the elements you can combine to create a styled control!

A Shapes Composition Example

As I mention, you should generally avoid direct painting and use core FireMonkey controls (or primitive controls) instead. As an example, we can obtain almost the same graphical effect of the

previous painting demo, without writing any painting code, but using and composing primitive controls. Here are the two forms (at design time the one with primitives and at run time the one based on painting code):

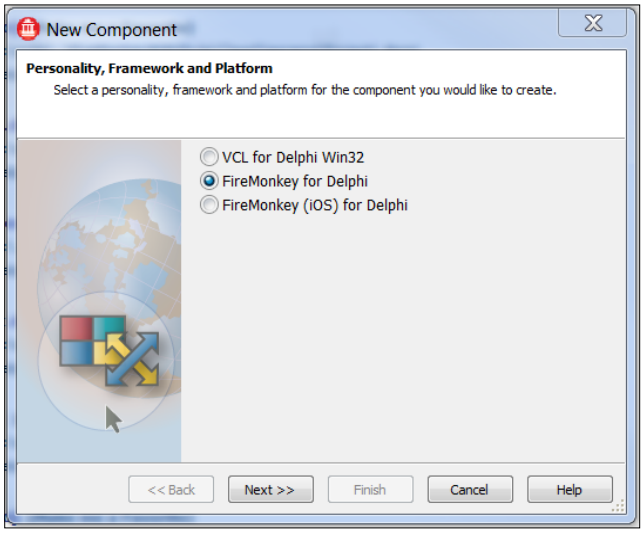
There is basically no code in the form above, as the same output is obtained by setting components properties. In fact, you read the code below you'll find the same values used by the previous painting code:

```
object Form1: TForm1
  Caption = 'Form1'
  ClientHeight = 407
  ClientWidth = 847
  object Layout1: TLayout
    Height = 193.000000000000000000
    Position.X = 16.000000000000000000
    Position.Y = 16.000000000000000000
    Width = 809.000000000000000000
    object Rectangle1: TRectangle
      Corners = [crTopRight, crBottomLeft]
      Height = 110.000000000000000000
```

```
      Position.X = 100.000000000000000000
      Position.Y = 10.000000000000000000
      Stroke.Color = claBlueviolet
      Width = 180.000000000000000000
    end
    object Rectangle2: TRectangle
      Fill.Kind = bkGradient
      Fill.Gradient.Points = <
        item
          Color = claRoyalblue
          Offset = 0.000000000000000000
        end
        item
          Color = claRed
          Offset = 1.000000000000000000
        end>
      Height = 113.000000000000000000
      Opacity = 0.600000023841857900
      Position.X = 232.000000000000000000
      Position.Y = 8.000000000000000000
      Width = 329.000000000000000000
    end
  end
  object Label1: TLabel
    Height = 177.000000000000000000
    Position.X = 16.000000000000000000
    Position.Y = 216.000000000000000000
    Text = 'Label1'
    Width = 809.000000000000000000
    object Rectangle3: TRectangle
      Corners = [crTopRight, crBottomLeft]
      Height = 110.000000000000000000
      Position.X = 100.000000000000000000
      Position.Y = 10.000000000000000000
      Stroke.Color = claBlueviolet
      Width = 180.000000000000000000
    end
    object Button1: TButton
      Height = 33.000000000000000000
      Position.X = 240.000000000000000000
      Position.Y = 56.000000000000000000
      TabOrder = 1
      Text = 'Button1'
      Width = 113.000000000000000000
    end
  end
end
```

A Custom Primitive Control

Now that you have a grasp of the core features and role of the primitive FireMonkey controls and of the TShape controls, we can see what it takes to write one. This is a reasonably simple adaptation of a LED control I wrote for the VCL many years ago. For writing a FireMonkey control you could use the New Component Wizard, which has specific support for the library. The form is below:



Wizard or not the code is relatively simple to write and extremely similar to that of a Delphi TGraphicControl descendant, with a custom paint method. This is the definition of the class for the new component:

```
type
  TLedStatus = (lsOn, lsOff);

  TCntLed = class(TShape)
  private
    fColor: TAlphaColor;
    fStatus: TLedStatus;
    procedure SetColor(const Value: TAlphaColor);
    procedure SetStatus(const Value: TLedStatus);
  protected
    procedure Paint; override;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property Status: TLedStatus
```

```
      read fStatus write SetStatus default lsOn;
    property Color: TAlphaColor
      read fColor write SetColor default claRed;
    property Position;
  end;
```

The controls has a color property (or type TAlphaColor) and a status property, which determines if the LED in the "on" or "off" state.

Notice you also need to re-surface the Position property, something that wasn't required in earlier versions of FireMonkey. The property, in fact, is there in the base class, but it isn't published (only public). If you omit it, the component won't remember its position.

The only significant method is Paint, which draws an external gray circle and an internal one with the chosen color (but only if the LED is "on"):

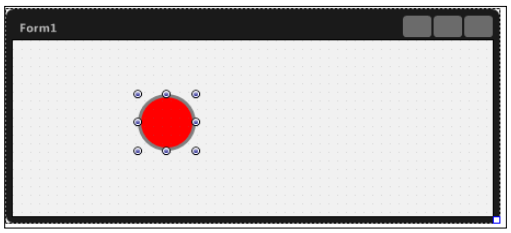
```
procedure TCntLed.Paint;
var
  Radius, XCenter, YCenter: Double;
begin
  // get the minimum between width
  // and height
  if Height > Width then
    Radius := Width / 2 - 2
  else
    Radius := Height / 2 - 2;
  // get the center
  XCenter := Width / 2;
  YCenter := Height / 2;
  // led border color (fixed)
  Canvas.Fill.Color := claGray;
  Canvas.FillEllipse (RectF(
    XCenter - Radius, YCenter - Radius,
    XCenter + Radius, YCenter + Radius),
    1.0);
  // led surface
  if fStatus = lsOn then
  begin
    Canvas.Fill.Color := fColor;
    Radius := Radius - 3;
    Canvas.FillEllipse (RectF(
      XCenter - Radius, YCenter - Radius,
      XCenter + Radius, YCenter + Radius),
    1.0);
  end;
```

```
end;
```

Finally, we need to add a Register function and compile the unit in a package, exactly like we would do for a VCL component:

```
procedure Register;
begin
  RegisterComponents ('CntFmx', [TCntLed]);
end;
```

If you install the component and place it on a new FireMonkey form, you'll see something like this:



Hurray, we've build one first FireMonkey control, and it wasn't very difficult. Now, this is a shape, what about styled controls?

STYLED CONTROLS

Style controls are all of the controls that descend from the TStyledControl and use styles to determine the control appearance. As we have seen styles are external resources, indicating the sub-elements that make up a control and their properties. As we have seen, the key property of a styled control is the StyleLookup property, while key methods include FindStyleResource (which you use to get to one of the sub-elements), GetStyleObject (with a simialr effect) and ApplyStyle.

All of the classic user interface controls, starting with Panel, Label, Button, and Edit are styles controls in FireMonkey. In fact, FireMonkey simulates the behavior of buttons or edit boxes by using styles. This makes it possible to adapt the control to the given platform.

If you go back to the picture with the structure of the style of a button (in the section “Sharing the Style Among Buttons”) a button is made of a background rectangle, with two internal rectangles (for the border and the focus) some animation and effects, plus a text element with the button caption. This is exactly what we mean when we say that “a styled control is a composition of primitive controls”. This composition is the “style”.

TTEXT AND TLABEL, AGAIN

To better understand this relationship among shapes and styled controls, we can look at the Label control, which has a Text element inside it. This is the definition of the class (reduced to its core elements):

```
type
  TLabel = class(TTextControl)
  protected
    procedure ApplyStyle; override;
    procedure SetText(const Value: string); override;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property Font;
    property FontColor;
    property Text;
    property StyleLookup;
    property StyleSettings;
  end;
```

Not much, as you can see. In fact the bulk of the code is in the base TTextControl class. This is a class representing a control with a text element. Again, the key elements of the class are:

```
type
  TTextControl = class(TStyledControl)
  private
    FTextSettings: TTextSettings;
    FStyledSettings: TStyledSettings;
    FTextObject: TControl;
    FText: string;
  protected
    procedure ApplyStyle; override;
    procedure FreeStyle; override;
    procedure SetText(const Value: string); virtual;
    function FindTextObject: TFmxObject; virtual;
    property TextObject: TControl read FTextObject;
    procedure SetTextSettings(const Value: TTextSettings); virtual;
    procedure SetStyledSettings(
      const Value: TStyledSettings); virtual;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    property Text: string read GetText write SetText
      stored IsTextStored;
```

```
    property TextSettings: TTextSettings
      read GetTextSettings write SetTextSettings;
    property StyledSettings: TStyledSettings read GetStyledSettings
      write SetStyledSettings stored StyledSettingsStored nodefault;
    property Font: TFont read GetFont write SetFont;
    property FontColor: TAlphaColor read GetFontColor
      write SetFontColor default TAlphaColorRec.Black;
  end;
```

While the component has a reference to a text element (technically a FTextObject of class TFmxObject), you might be surprise to figure out this sub-control is not created in the control constructor. Instead, is it hooked up in the key ApplyStyle method (by calling FindTextObject) listed below in its initial portion:

```
function TTextControl.FindTextObject: TFmxObject;
begin
  Result := FindStyleResource('text');
end;

procedure TTextControl.ApplyStyle;
var
  S: TFmxObject;
  NewT : string;
begin
  FIsDisableDesign := True;
  inherited;
  { from text }
  S := FindTextObject;
  if Assigned(S) and (S is TControl) then
  begin
    FTextObject := TControl(S);
    if Supports(S, ITextSettings, FITextSettings) then
      FDefaultTextSettings.Assign(FITextSettings.TextSettings)
    else
      FDefaultTextSettings.Assign(nil);
  end
  else
    FDefaultTextSettings.Assign(nil);
  ...
```

The TLabel class has a modified version of the ApplyStyle method, but the meaning is similar. As you can see, once the style is applied, the control connects its FTextObject field to a sub-elements called “text”, which is generally (but not necessarily) a TText object. This “text” object should support the ITextSettings interface for updating the graphical elements of the text,

mapping it to some of the control properties (for example the text color we discussed at length earlier on).

In several methods, for example DoChanged, the local copy of the text (the FText value) is copied to the internal "text" object, but there are likely ways around the automatic updates that could get you in a dis-aligned situation. Here is a key code snippet of TTextControl.DoChanged:

```
if Assigned(FTextObject) then
begin
  // set text value
  if FTextObject is TText then
  begin
    TText(FTextObject).Text := TextStr;
  end;
  if FTextObject is TTextControl then
    TTextControl(FTextObject).Text := TextStr;

  // Effects
  FTextObject.UpdateEffects;
  UpdateEffects;
  FTextObject.Repaint;
end
```

Here you can see how is the "text" object is either a TText shape or a TTextControl object the string with the text is copied directly to those sub-controls ties to the main controls style. Of course this code is triggered as you change the value of the Text property of the main control.

There is still a pending question. Where is the code to create the sub-element of the TLabel? The answer is, it doesn't exist. It is only by applying a proper style (with the rules above) that the sub-element is created! But which style will a control use? This is determined by a virtual function of the TStyledControl class you can override in subclasses:

```
function TStyledControl.GetDefaultStyleLookupName: string;
begin
  Result := ClassName + 'style';
  Delete(Result, 1, 1); // just remove T
end;
```

As you can see the rule is quite simple: The style for TButton will be "Buttonstyle", that for TLabel, "Labelstyle". The actual value is surfaced in the read-only DefaultStyleLookupName property of styled controls.

A Custom Styled Control

Following the same model, we can create out own first styled control, getting fully into the FireMonkey components model. I want to build a LED control again, but this time I won't be painting its surface but refer to at least one circle defined in the style and map the control's properties to this internal object. This is the class definition, which has the same visible properties of the previous version:

```
type
  TLedStatus = (lsOn, lsOff);

  TCntStyledLed = class(TStyledControl)
  private
    fLedCircle: TCircle;
    fColor: TAlphaColor;
    fStatus: TLedStatus;
    procedure SetColor(const Value: TAlphaColor);
    procedure SetStatus(const Value: TLedStatus);
  public
    constructor Create(AOwner: TComponent); override;
    procedure ApplyStyleLookup; override;
  published
    property Status: TLedStatus
      read fStatus write SetStatus default lsOn;
    property Color: TAlphaColor
      read fColor write SetColor default claRed;
    property Position;
  end;
```

The key differences are the fLedCircle field and the ApplyStyleLookup method, which assign that field:

```
procedure TCntStyledLed.ApplyStyleLookup;
var
  S: TFmxObject;
begin
  inherited;

  S := FindStyleResource('led');
  if (S <> nil) and (S is TCircle) then
  begin
    fLedCircle := TCircle(S);
    fLedCircle.Fill.Color := fColor;
    fLedCircle.Visible := fStatus = lsOn;
  end;
```

```
end;
```

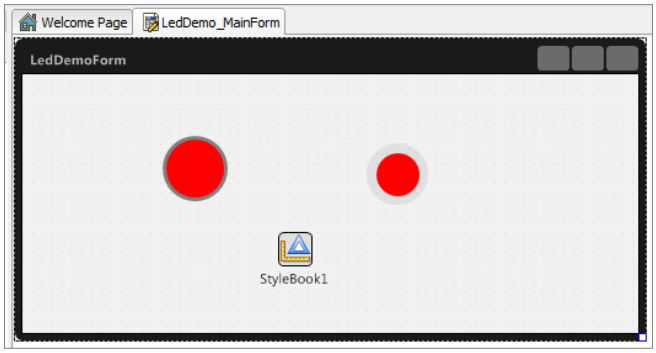
Now in case we need to change one of the properties, we also have to copy the value to the internal control:

```
procedure TCntStyledLed.SetColor(const Value: TAlphaColor);
begin
  fColor := Value;
  if Assigned (fLedCircle) then
    fLedCircle.Fill.Color := fColor;
end;
```

Now to test this component, besides placing it in a form, we need a proper style for it, like the following you can find in the StyleBook component of the demo:

```
object TLayout
  StyleName = 'cntstyledledstyle'
  Height = 50.000000000000000000
  Position.X = 196.000000000000000000
  Position.Y = 308.000000000000000000
  Width = 50.000000000000000000
  object TCircle
    StyleName = 'Circle1'
    Fill.Color = claMidnightblue
    Height = 50.000000000000000000
    Stroke.Kind = bkNone
    Width = 50.000000000000000000
    object TCircle
      StyleName = 'led'
      Fill.Color = claCrimson
      Height = 41.000000000000000000
      Position.X = 8.000000000000000000
      Position.Y = 5.000000000000000000
      Stroke.Kind = bkNone
      Width = 35.000000000000000000
    end
  end
end
```

Notice that the external circle is not compulsory, while the internal one is, and its style must be called “led”, just like the entire style must be called “cntstyledledstyle” if you want it to be applied automatically. This is how the two controls (the painted one and the styled one) look in a demo form at design time:



We could add many other elements to the LED control style, like animations and glowing effects, but this is enough as an introduction to the development of custom styled controls.

CONCLUSION

In this paper we have seen how styles play a key role in FireMonkey and they are not just used for decorating the user interface Styles in FireMonkey are the key element of the entire development on the platforms. We've explored a detailed introductory demo, delved into the library source code, trying to fully understand the architecture of the FireMonkey library. We've also looked at the basics for building a couple of simple custom controls.

Hopefully this paper has given you a better understanding of FireMonkey, helping you see the power of styles and how they can really let developers adapt the same user interface controls to very different operating systems and user interface rules. There is much more to styles that we have seen in this paper: I've only scratched the surface of bitmap based styles and haven't really focused on the ability of using styles to connect to native platforms controls, as FireMonkey does on iOS.

FireMonkey is a cross-platform library that using its style architecture delivers single source for multiple devices but also a very large degree of platform fidelity, rather than pushing a common abstract UI on all platforms. It results in a great opportunity for building the user interface of your applications in a high-level and fast design surface, while writing your code only once. Add to this robust and simple data access and visual data-binding capabilities, and you'll have a unique tool for building your desktop and mobile applications.

About The Author

Marco Cantù is Delphi Product Manager at Embarcadero Technologies. Before joining the company he was the author of the best-selling Mastering Delphi series and the self-published *Delphi Handbooks* series on the several versions of Delphi (from 2007 to XE).

Marco is a frequent conference speaker, author of countless articles on Delphi. You can read Marco's blog at *http://blog.marcocantu.com*, follow him on Twitter as *@marcocantu*, and contact him on *marco.cantu@embarcadero.com*.

About Embarcadero Technologies

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. Founded in 1993, Embarcadero is headquartered in San Francisco, with offices located around the world. Embarcadero is online at www.embarcadero.com.