CLX Component Development

CHAPTER

13

IN THIS CHAPTER

The last three chapters have focused on creating custom components in Delphi. More precisely, Chapters 11, “VCL Component Building,” and 12, “Advanced VCL Component Building,” have focused on creating custom VCL components. However, as noted in Chapter 10, “Component Architecture: VCL and CLX,” there are two component class hierarchies in Delphi 6: the VCL and CLX. In this chapter, we change our focus slightly to that of creating custom CLX components. Fortunately, much of what you have learned in creating VCL components also applies to creating CLX components.

What Is CLX?

CLX, pronounced “clicks,” is an acronym for *Component Library for Cross-Platform*, and was first introduced in Borland’s new Linux RAD tool, Kylix. However, CLX isn’t just simply the VCL under Linux. That is, the CLX architecture is also available in Delphi 6, and therefore provides the foundation for creating native cross-platform applications using Delphi 6 and Kylix.

In Delphi, the VCL is typically associated with the components that appear on the Component Palette. This isn’t surprising because the vast majority of the components appearing on the palette are visual controls. However, CLX encompasses much more than a visual component hierarchy. Specifically, CLX is divided into four separate parts: BaseCLX, VisualCLX, DataCLX, and NetCLX.

BaseCLX, as it is called in Kylix, contains the base units and classes that are shared between Kylix and Delphi 6. For example, the `System`, `SysUtils`, and `Classes` units are part of BaseCLX. VisualCLX is similar to what most people consider the VCL. However, VisualCLX is based on the Qt widget library rather than the standard Windows controls defined in `User32.dll` or `ComCtl32.dll`. DataCLX contains the data access components and encompasses the new dbExpress technology. And finally, NetCLX contains the new cross-platform WebBroker technology.

If you are familiar with previous versions of Delphi, you will recognize that the units included in BaseCLX have been available in Delphi since version 1. As such, you could argue that these units are also part of the VCL. In fact, Borland recognized the confusion caused by calling these base units collectively as BaseCLX, and in Delphi 6 these base units are referred to as the RTL.

The point of all this is that even though these base units will be used in both VCL and CLX applications, a CLX application is typically defined as one built using the classes in VisualCLX.

In this chapter, we will be focusing on VisualCLX. In particular, we’ll be investigating how to extend the VisualCLX architecture by creating our own custom CLX components. As noted earlier, VisualCLX is based on the Qt widget library, which is produced by Troll Tech. Qt, pronounced “cute,” is a platform independent C++ class library of user interface (UI) widgets (or controls).To be precise, Qt currently supports Windows and the X Window System, and thus can be used on both Windows and Linux desktops. In fact, Qt is the most prevalent class library used for Linux GUI development. For instance, Qt is used in the development of the KDE Window Manager.

Other cross-platform class libraries are available, but Borland chose to build VisualCLX on top of Qt for several reasons. First, Qt classes look very much like VCL components. For example, properties are defined as get/set method pairs. Qt also incorporates the notion of events through a mechanism called a signal. Plus, the Qt graphics model is very similar to the one used in the VCL. And finally, the Qt library defines a wide variety of standard user interface controls, which are called widgets in the Qt nomenclature. As a result, the Borland engineers were able to wrap many of the existing Qt widgets with Object Pascal wrappers rather than create the required components from scratch.

The CLX Architecture

As suggested previously, VisualCLX consists of Object Pascal classes that wrap around existing functionality defined in the Qt classes. This is very similar to the way in which the VCL encapsulates the functionality of the Windows API and the Common Controls. One of the design goals in creating CLX was to make it as easy as possible to port existing VCL applications to the CLX architecture. As a result, the class hierarchy in CLX is very similar to the VCL as illustrated in Figures 13.1 and 13.2. The dark gray boxes in Figure 13.1 highlight the principal base classes in the VCL.

However, the class hierarchies aren’t identical. In particular, some new classes have been added and some classes have been moved to different branches from their VCL counterparts. Figure 13.2 highlights these differences with light gray boxes. For example, the CLX Timer component does not descend directly from `TComponent` as it does in the VCL. Instead, it descends from the new `THandleComponent`, which is a base class that should be used whenever a nonvisual component requires access to the handle of an underlying Qt control. Also, note how the CLX Label component is no longer a graphical control, but rather a descendant of the new `TFrameControl` class. The Qt library provides a wide variety of bordering options for controls, and the `TFrameControl` class provides a wrapper around that functionality.

As noted earlier, controls in the Qt library are called widgets. As a result, the `TWidgetControl` class is the CLX equivalent to the VCL’s `TWinControl`. Why change the classname? Switching to `Widget` puts the class in line with the base Qt classes, and removing `Win` further removes the dependency on the Windows controls in VisualCLX.

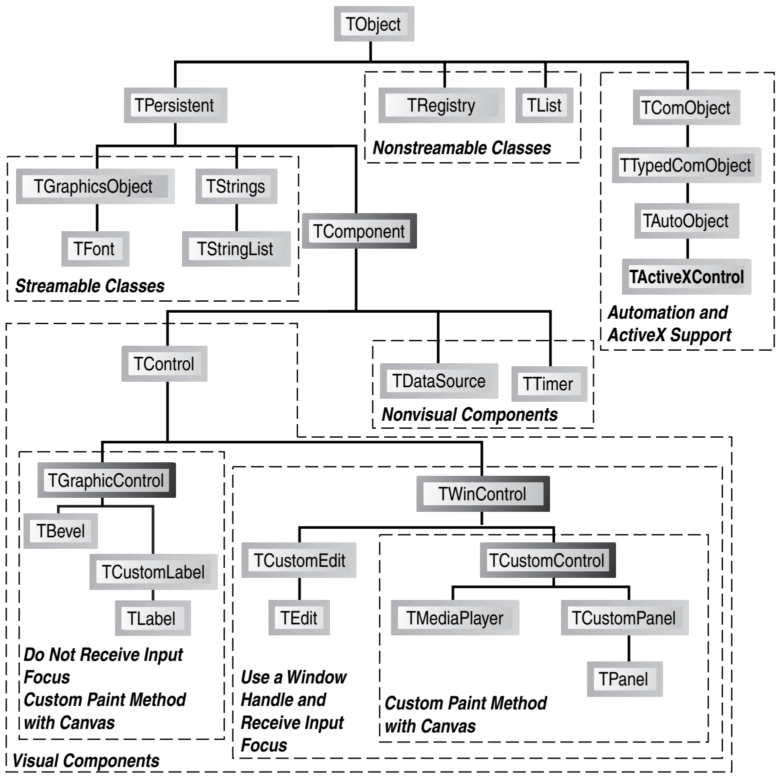


FIGURE 13.1
The VCL Base Class hierarchy.

Surprisingly, Borland also defined the `TWinControl` class in CLX as an alias for `TWidgetControl`. During the development of Kylix and CLX, one of the early ideas that was promoted was that a single source file could be used to define both a CLX component and a VCL component. Conditional directives would be used to specify a VCL `uses` clause when compiled under Windows and a CLX `uses` clause when compiled under Linux. However, this approach is only feasible for very simple components. In practice, there are usually enough significant changes between the implementations to warrant the creation of separate units.

NOTE

Creating a VCL component and a CLX component in a single source file is different from creating a CLX component (in a single source file) that can be used in both Delphi 6 and Kylix. This chapter illustrates how to do the latter.

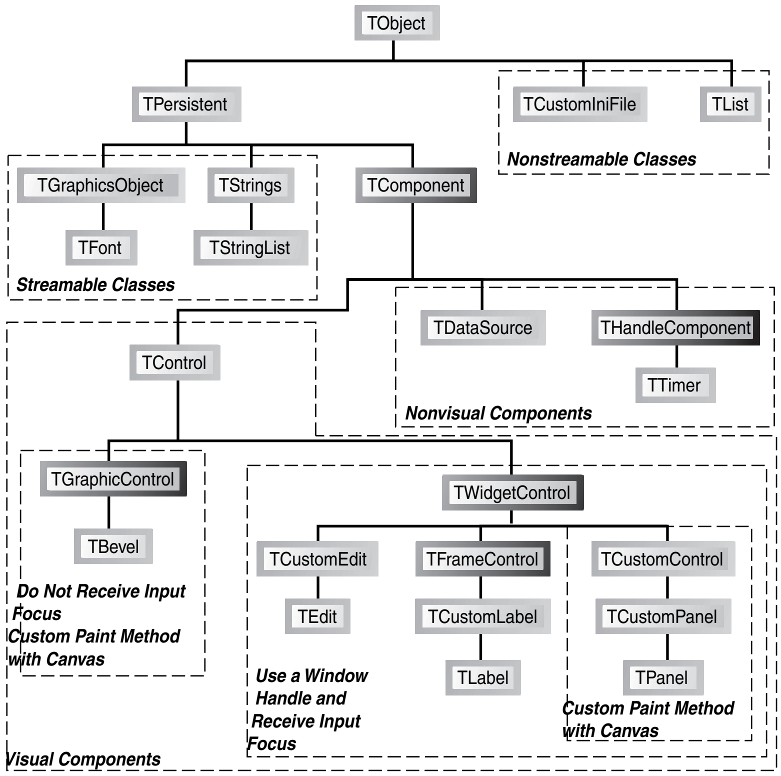


FIGURE 13.2
The CLX Base Class hierarchy.

Fortunately, the changes in the class hierarchy illustrated in Figure 13.2 should have little impact on application developers. That is, most of the VCL components that come with Delphi have VisualCLX equivalents such as `TEdit`, `TListBox`, `TComboBox`, and so on. Unfortunately, component writers aren't so lucky because they will be much more affected by changes to the class hierarchy.

NOTE

The Object Browser in Delphi 6 and Kylix is extremely helpful in learning the structure of the new class hierarchy. However, because of the `TWinControl` alias to `TWidgetControl`, you will actually see two identical class hierarchies in the Object Browser.

Fortunately, there are actually quite a few similarities between the VCL and CLX architectures that the figures don't illustrate. For instance, the `TCanvas` class is very similar in both architectures. Of course, the implementation encapsulated by the class is quite different. Under the VCL, the `TCanvas` class provides a wrapper around a Windows GDI device context, which is accessible through the `TCanvas.Handle` property. Under CLX, the `TCanvas` class provides a wrapper around a Qt painter, which is also accessible through the `TCanvas.Handle` property. As a result, you can use the `Handle` property to access any of the low-level GDI functions in a VCL application and the Qt graphics library functions in a CLX application.

The components in CLX were designed to ease porting an existing VCL application to a CLX application. As a result, the public and published interfaces to many of the components are nearly identical in both architectures. This means that events such as `OnClick`, `OnChange`, and `OnKeyPress` as well as their corresponding event dispatch methods—`Click()`, `Change()`, and `KeyPress()`—are implemented in both the VCL and CLX.

Porting Issues

CLX does indeed share many similarities with the VCL. However, many platform differences must be addressed, especially for component writers. You must address Win32 dependencies in your code. For example, any calls to the Win32 API (in `Windows.pas`) will need to be changed if the component is to operate in Kylix.

NOTE

Building a CLX component implies that you want to use the component in Kylix under Linux and possibly in Delphi 6 under Windows. If you only need to support Windows, create a VCL component and not a CLX component.

In addition, several runtime library (RTL) issues must be handled differently on Linux versus Windows such as case sensitivity for filenames and path delimiters under Linux. For some VCL components, it will simply be impossible to port to Linux. Consider a VCL component that provides a wrapper around the Messaging API (MAPI). Because MAPI doesn't exist under Linux, a different mechanism will need to be used.

In addition to the platform issues described previously, some additional porting issues must be considered when migrating to CLX. For example, COM certainly isn't supported under Linux, but interfaces most certainly are. Owner-Draw techniques available in many VCL wrappers around Windows controls aren't recommended in CLX components. Owner-Draw capabilities have been deprecated in lieu of Qt Styles. Other VCL features that aren't supported under CLX include docking, bi-directional support, the input method editor, and Asian locale support.

One additional change that will certainly cause developers some problems is that the CLX versions of components are located in a different set of units from the VCL controls. For example, the `Controls.pas` unit in the VCL becomes the `QControls.pas` unit in CLX. The problem with this change is that if you are developing a CLX component or application under Delphi 6, the VCL units are still available. As a result, it is quite possible to inadvertently mix CLX and VCL units into your component units. In some cases, your component might run correctly under Windows. However, if you move the component over to Kylix, you will get compiler errors because the VCL units aren't available on Linux.

NOTE

Borland suggests that developers create their CLX components in Kylix on Linux to help prevent the misuse of VCL units in a CLX component. However, many developers will probably opt to develop under Delphi 6, with its new IDE enhancements and the comfort of Windows, and then test their components under Kylix.

Another issue that developers must contend with when writing CLX components (and applications for that matter) is case sensitivity under Linux. In particular because filenames and paths are case sensitive under Linux, the unit names that you specify on your `uses` clause of your own units must be the correct case. This requirement is needed in order for the Kylix compiler to be able to locate the units under Linux. Although Delphi is not case sensitive, this isn't the first time that Delphi requires an element to be case sensitive. The first situation involves the naming of the `Register()` procedure in a unit to be exported from a package.

No More Messages

Linux, or more appropriately XWindows, doesn't implement a messaging architecture like Windows does. As a result, there are no `wm_LButtonDown`, `wm_SetCursor`, or `wm_Char` messages passing around on Linux. When a CLX component is used under Linux, the underlying Qt classes handle the appropriate system events and provide the necessary hooks in order to respond to those events. The bottom line is that system events are handled by the Qt classes even on Windows. Therefore, a CLX component won't be able to hook into a Windows message.

As a result, VCL component message handlers such as `CMTextChanged()` have been replaced with dynamic methods—for example, `TextChanged()`. This will be highlighted in the following section. This also means that implementing certain behaviors, which are easily implemented using messages in the VCL, must be implemented quite differently under CLX.

Sample Components

In this section, we will take a detailed look into several VCL components that have been transformed into CLX components. The first one is a custom spinner component that involves several principle features including custom painting, keyboard handling, focus changes, mouse interactions, and even custom events.

The next three components are successive descendants of the base spinner component—each extending the previous component. The first descendant extends the base spinner by adding support for handling mouse events at design time and displaying custom cursors. The second spinner descendant adds support for displaying images from an ImageList. The final spinner component adds support for connecting the control to a field in a dataset.

NOTE

All the units presented in this chapter can be used in both Delphi 6 and Kylix.

The `TddgSpinner` Component

Figure 13.3 shows three instances of the `TddgSpinner` component being used in a CLX application. Unlike traditional spin-edits, this custom component displays the increment and decrement buttons that change the spinner's value at each end of the spinner rather than on top of one another at one end.

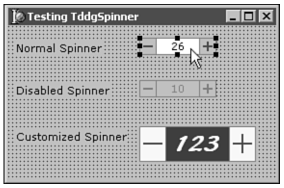


FIGURE 13.3
The `TddgSpinner` CLX component can be used to specify integer values.

Listing 13.1 shows the complete source code for the `QddgSpin.pas` unit, which implements the `TddgSpinner` component. This particular component started out as a custom spinner control that descended from the `TCustomControl` class in the VCL. However, the `TddgSpinner` class now descends from the CLX `TCustomControl` class, and as a result, can be used in both Windows and Linux.

Although classnames rarely change when migrating to CLX, unit names are typically prefixed with the letter Q to indicate their dependency on the Qt library via VisualCLX.

NOTE

Although commented out, each listing includes the original VCL-specific code. Comments that start with `VCL->CLX:` highlight specific issues involved in transforming the control from the VCL to CLX.

LISTING 13.1 `QddgSpin.pas`—Source Code for the `TddgSpinner` Component

```
unit QddgSpin;

interface

uses
  SysUtils, Classes, Types, Qt, QControls, QGraphics;
  (*
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  ImgList;
  *)

type
  TddgButtonType = ( btMinus, btPlus );
  TddgSpinnerEvent = procedure (Sender: TObject; NewValue: Integer;
                              var AllowChange: Boolean ) of object;

  TddgSpinner = class( TCustomControl )
  private
    // Instance Data for Component
    FValue: Integer;
    FIncrement: Integer;
    FButtonColor: TColor;
    FButtonWidth: Integer;
    FMinusBtnDown: Boolean;
    FPlusBtnDown: Boolean;

    // Method Pointers to Hold Custom Events
    FOnChange: TNotifyEvent;
    FOnChanging: TddgSpinnerEvent;

    (*
    // VCL->CLX:  These message handlers are not available in CLX

    // Window Message Handling Method
    procedure WMGetDlgCode( var Msg: TWMGetDlgCode );
      message wm_GetDlgCode;
```

LISTING 13.1 Continued

```
    // Component Message Handling Method
    procedure CMEnabledChanged( var Msg: TMessage );
      message cm_EnabledChanged;
    *)
  protected
    procedure Paint; override;
    procedure DrawButton( Button: TddgButtonType; Down: Boolean;
                          Bounds: TRect ); virtual;

    // Support Methods
    procedure DecValue( Amount: Integer ); virtual;
    procedure IncValue( Amount: Integer ); virtual;

    function CursorPosition: TPoint;
    function MouseOverButton( Btn: TddgButtonType ): Boolean;

    // VCL->CLX:  EnabledChanged replaces cm_EnabledChanged
    //            component message handler
    procedure EnabledChanged; override;

    // New Event Dispatch Methods
    procedure Change; dynamic;
    function CanChange( NewValue: Integer ): Boolean; dynamic;

    // Overridden Event Dispatch Methods
    procedure DoEnter; override;
    procedure DoExit; override;
    procedure KeyDown(var Key: Word; Shift: TShiftState); override;

    procedure MouseDown( Button: TMouseButton; Shift: TShiftState;
                         X, Y: Integer ); override;
    procedure MouseUp( Button: TMouseButton; Shift: TShiftState;
                       X, Y: Integer ); override;

    (*
    // VCL->CLX:  These following declarations have changed in CLX

    function DoMouseWheelDown( Shift: TShiftState;
                               MousePos: TPoint ): Boolean; override;

    function DoMouseWheelUp( Shift: TShiftState;
                             MousePos: TPoint ): Boolean; override;
    *)
```

LISTING 13.1 Continued

```
    function DoMouseWheelDown( Shift: TShiftState;
                        const MousePos: TPoint ): Boolean; override;

    function DoMouseWheelUp( Shift: TShiftState;
                        const MousePos: TPoint ): Boolean; override;


    // Access Methods for Properties
    procedure SetButtonColor( Value: TColor ); virtual;
    procedure SetButtonWidth( Value: Integer ); virtual;
    procedure SetValue( Value: Integer ); virtual;
  public
    // Don't forget to specify override for constructor
    constructor Create( AOwner: TComponent ); override;
  published
    // New Property Declarations
    property ButtonColor: TColor
      read FButtonColor
      write SetButtonColor
      default clBtnFace;

    property ButtonWidth: Integer
      read FButtonWidth
      write SetButtonWidth
      default 18;

    property Increment: Integer
      read FIncrement
      write FIncrement
      default 1;

    property Value: Integer
      read FValue
      write SetValue;

    // New Event Declarations

    property OnChange: TNotifyEvent
      read FOnChange
      write FOnChange;

    property OnChanging: TddgSpinnerEvent
      read FOnChanging
      write FOnChanging;
```

LISTING 13.1 Continued

```
    // Inherited Properties and Events
    property Color;
    (*
    property DragCursor;      // VCL->CLX:  Property not yet in CLX
    *)
    property DragMode;
    property Enabled;
    property Font;
    property Height default 18;
    property HelpContext;
    property Hint;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property TabOrder;
    property TabStop default True;
    property Visible;
    property Width default 80;

    property OnClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnEnter;
    property OnExit;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnStartDrag;
  end;


implementation


{==========================}
{== TddgSpinner Methods ==}
{==========================}

constructor TddgSpinner.Create( AOwner: TComponent );
begin
```

LISTING 13.1 Continued

```
  inherited Create( AOwner );

  // Initialize Instance Data
  FButtonColor := clBtnFace;
  FButtonWidth := 18;
  FValue := 0;
  FIncrement := 1;

  FMinusBtnDown := False;
  FPlusBtnDown := False;

  // Initializing inherited properties
  Width := 80;
  Height := 18;
  TabStop := True;

  // VCL->CLX:  TWidgetControl sets Color property to clNone
  Color := clWindow;

  // VCL->CLX:  InputKeys assignment replaces handling the
  //            wm_GetDlgCode message.
  InputKeys := InputKeys + [ ikArrows ];
end;


{== Property Access Methods ==}

procedure TddgSpinner.SetButtonColor( Value: TColor );
begin
  if FButtonColor <> Value then
  begin
    FButtonColor := Value;
    Invalidate;
  end;
end;

procedure TddgSpinner.SetButtonWidth( Value: Integer );
begin
  if FButtonWidth <> Value then
  begin
    FButtonWidth := Value;
    Invalidate;
  end;
end;
```

LISTING 13.1 Continued

```
procedure TddgSpinner.SetValue( Value: Integer );
begin
  if FValue <> Value then
  begin
    if CanChange( Value ) then
    begin
      FValue := Value;
      Invalidate;

      // Trigger Change event
      Change;
    end;
  end;
end;


{== Painting Related Methods ==}

procedure TddgSpinner.Paint;
var
  R: TRect;
  YOffset: Integer;
  S: string;
  XOffset: Integer;              // VCL->CLX:  Added for CLX support
begin
  inherited Paint;
  with Canvas do
  begin
    Font := Self.Font;
    Pen.Color := clBtnShadow;

    if Enabled then
      Brush.Color := Self.Color
    else
    begin
      Brush.Color := clBtnFace;
      Font.Color := clBtnShadow;
    end;

    // Display Value
    (*
    // VCL->CLX:  SetTextAlign not available in CLX
    SetTextAlign( Handle, ta_Center or ta_Top );    // GDI function
    *)
```

LISTING 13.1 Continued

```
    R := Rect( FButtonWidth - 1, 0,
               Width - FButtonWidth + 1, Height );
    Canvas.Rectangle( R.Left, R.Top, R.Right, R.Bottom );
    InflateRect( R, -1, -1 );


    S := IntToStr( FValue );
    YOffset := R.Top + ( R.Bottom - R.Top -
                         Canvas.TextHeight( S ) ) div 2;

    // VCL->CLX:  Calculate XOffset b/c no SetTextAlign function
    XOffset := R.Left + ( R.Right - R.Left -
                          Canvas.TextWidth( S ) ) div 2;

    (*
    // VCL->CLX:  Change TextRect call b/c no SetTextAlign function
    TextRect( R, Width div 2, YOffset, S );
    *)
    TextRect( R, XOffset, YOffset, S );

    DrawButton( btMinus, FMinusBtnDown,
                Rect( 0, 0, FButtonWidth, Height ) );
    DrawButton( btPlus, FPlusBtnDown,
                Rect( Width - FButtonWidth, 0, Width, Height ) );

    if Focused then
    begin
      Brush.Color := Self.Color;
      DrawFocusRect( R );
    end;
  end;
end; {= TddgSpinner.Paint =}


procedure TddgSpinner.DrawButton( Button: TddgButtonType;
                                  Down: Boolean; Bounds: TRect );
begin
  with Canvas do
  begin
    if Down then                            // Set background color
      Brush.Color := clBtnShadow
    else
      Brush.Color := FButtonColor;
    Pen.Color := clBtnShadow;
```

LISTING 13.1 Continued

```
    Rectangle( Bounds.Left, Bounds.Top,
               Bounds.Right, Bounds.Bottom );

    if Enabled then
    begin
      (*
      // VCL->CLX: clActiveCaption is set to
      //           clActiveHighlightedText in CLX.
      Pen.Color := clActiveCaption;
      Brush.Color := clActiveCaption;
      *)
      Pen.Color := clActiveBorder;
      Brush.Color := clActiveBorder;
    end
    else
    begin
      Pen.Color := clBtnShadow;
      Brush.Color := clBtnShadow;
    end;

    if Button = btMinus then               // Draw the Minus Button
    begin
      Rectangle( 4, Height div 2 - 1,
                 FButtonWidth - 4, Height div 2 + 1 );
    end
    else                                    // Draw the Plus Button
    begin
      Rectangle( Width - FButtonWidth + 4, Height div 2 - 1,
                 Width - 4, Height div 2 + 1 );
      Rectangle( Width - FButtonWidth div 2 - 1,
                 ( Height div 2 ) - (FButtonWidth div 2 - 4),
                 Width - FButtonWidth div 2 + 1,
                 ( Height div 2 ) + (FButtonWidth div 2 - 4)  );
    end;
    Pen.Color := clWindowText;
    Brush.Color := clWindow;
  end;
end; {= TddgSpinner.DrawButton =}


procedure TddgSpinner.DoEnter;
begin
  inherited DoEnter;
```

LISTING 13.1 Continued

```
  // Controls gets focus--update display to show focus border
  Repaint;
end;

procedure TddgSpinner.DoExit;
begin
  inherited DoExit;
  // Control lost focus--update display to remove focus border
  Repaint;
end;


// VCL->CLX:  EnabledChanged replaces cm_EnabledChanged handler

procedure TddgSpinner.EnabledChanged;
begin
  inherited;
  // Repaint the component so that it reflects the state change
  Repaint;
end;


{== Event Dispatch Methods ==}

{===================================================================
  TddgSpinner.CanChange

  This is the event dispatch method supporting the OnChanging
  event. Notice that this method is a function, rather than the
  common procedure variety. As a function, the Result variable is
  assigned a value before calling the user defined event handler.
===================================================================}

function TddgSpinner.CanChange( NewValue: Integer ): Boolean;
var
  AllowChange: Boolean;
begin
  AllowChange := True;
  if Assigned( FOnChanging ) then
    FOnChanging( Self, NewValue, AllowChange );
  Result := AllowChange;
end;
```

LISTING 13.1 Continued

```
procedure TddgSpinner.Change;
begin
  if Assigned( FOnChange ) then
    FOnChange( Self );
end;


// Notice that both DecValue and IncValue assign the new value to
// the Value property (not FValue), which indirectly calls SetValue

procedure TddgSpinner.DecValue( Amount: Integer );
begin
  Value := Value - Amount;
end;

procedure TddgSpinner.IncValue( Amount: Integer );
begin
  Value := Value + Amount;
end;


{== Keyboard Processing Methods ==}

(*
// VCL->CLX:  Replaced with InputKeys assignment in constructor

procedure TddgSpinner.WMGetDlgCode( var Msg: TWMGetDlgCode );
begin
  inherited;
  Msg.Result := dlgc_WantArrows;  // Control will handle arrow keys
end;
*)


procedure TddgSpinner.KeyDown( var Key: Word; Shift: TShiftState );
begin
  inherited KeyDown( Key, Shift );

  // VCL->CLX:  Key constants changed in CLX.
  //            vk_ prefix changed to Key_

  case Key of
    Key_Left, Key_Down:
      DecValue( FIncrement );
```

LISTING 13.1 Continued

```
    Key_Up, Key_Right:
      IncValue( FIncrement );
  end;
end;


{== Mouse Processing Methods ==}

function TddgSpinner.CursorPosition: TPoint;
begin
  GetCursorPos( Result );
  Result := ScreenToClient( Result );
end;


function TddgSpinner.MouseOverButton(Btn: TddgButtonType): Boolean;
var
  R: TRect;
begin
  // Get bounds of appropriate button
  if Btn = btMinus then
    R := Rect( 0, 0, FButtonWidth, Height )
  else
    R := Rect( Width - FButtonWidth, 0, Width, Height );

  // Is cursor position within bounding rectangle?
  Result := PtInRect( R, CursorPosition );
end;


procedure TddgSpinner.MouseDown( Button: TMouseButton;
                                Shift: TShiftState; X, Y: Integer);
begin
  inherited MouseDown( Button, Shift, X, Y );

  if not ( csDesigning in ComponentState ) then
    SetFocus;              // Move focus to Spinner only at runtime

  if ( Button = mbLeft ) and
     ( MouseOverButton(btMinus) or MouseOverButton(btPlus) ) then
  begin
    FMinusBtnDown := MouseOverButton( btMinus );
    FPlusBtnDown := MouseOverButton( btPlus );
```

LISTING 13.1 Continued

```
    Repaint;
  end;
end;


procedure TddgSpinner.MouseUp( Button: TMouseButton;
                              Shift: TShiftState; X, Y: Integer );
begin
  inherited MouseUp( Button, Shift, X, Y );

  if Button = mbLeft then
  begin
    if MouseOverButton( btPlus ) then
      IncValue( FIncrement )
    else if MouseOverButton( btMinus ) then
      DecValue( FIncrement );

    FMinusBtnDown := False;
    FPlusBtnDown := False;

    Repaint;
  end;
end;


function TddgSpinner.DoMouseWheelDown( Shift: TShiftState;
                                 const MousePos: TPoint ): Boolean;
begin
  inherited DoMouseWheelDown( Shift, MousePos );
  DecValue( FIncrement );
  Result := True;
end;


function TddgSpinner.DoMouseWheelUp( Shift: TShiftState;
                                 const MousePos: TPoint ): Boolean;
begin
  inherited DoMouseWheelUp( Shift, MousePos );
  IncValue( FIncrement );
  Result := True;
end;

end.
```

As you can see, the source code for the CLX version is very similar to the VCL edition. However, there are several important differences.

First, notice the inclusion of the Qt specific units: `Qt`, `QControls`, and `QGraphics`. `Types` is also a new unit that is shared between the VCL and CLX. Fortunately, the majority of the `TddgSpinner` CLX class declaration looks identical to what you would find in the VCL. That is, instance fields are declared the same way, as are method pointers to hold event handlers, as well as event dispatch methods.

The `CMEnabledChanged()` and `WMGetDlgCode()` message handling methods represent the first implementation change that we must handle in migrating to CLX. Specifically, the corresponding `cm_EnabledChanged` and `wm_GetDlgCode` messages don't exist in CLX. Therefore the functionality implemented in these message handlers must be moved elsewhere.

As noted earlier, in CLX, component messages such as `cm_EnabledChanged` have been replaced with appropriate dynamic methods. So instead of sending a `cm_EnabledChanged` message whenever the Enabled property is changed, the `TControl` class in CLX simply calls the `EnabledChanged()` method. Therefore, the code from the old `CMEnabledChanged()` method is simply moved to the overridden `EnabledChanged()` method.

A common task in component writing is to handle the arrow keys on the keyboard. For the `TddgSpinner` component, the arrow keys can be used to increment and decrement the value. In a VCL component, this behavior is accomplished by handling the `wm_GetDlgCode` message and specifying which keys your control will handle. As noted previously, the `wm_GetDlgCode` message doesn't exist for a CLX component. Thus a different approach must be taken. Fortunately, the `TWidgetControl` class defines the `InputKeys` property, which allows us to specify the keys we want to handle in the constructor of our component.

The constructor code also indicates another change between the VCL and CLX. That is, the `TWidgetControl` class sets the Color property, which is declared in the `TControl` class to be `clNone`. In the VCL, the `TWinControl` class simply uses the inherited Color value of `clWindow`. As a result, we need to set the Color property in the constructor to `clWindow` so that the spinner appears in the correct color.

After these constructor changes, there aren't too many other changes. As you can see, most event dispatch methods are also available under CLX. As a result, it is much easier to migrate to CLX if you are currently overriding event dispatch methods in your VCL components rather than handling specific Windows messages for the underlying window handle.

At the beginning of this chapter, it was noted that all the techniques you learned about VCL component building in the previous chapters also apply to creating CLX components. You will notice that property declarations, access methods, and even custom events are handled the same way in both the VCL and CLX.

More than any other component method, the `Paint()` method will probably require the most modifications when transforming a VCL component into a CLX component.

When transforming a VCL control into a CLX component, display methods, such as `Paint()`, will usually require the most modifications even though the `TCanvas` classes in both architectures have nearly identical interfaces.

Two display issues needed to be handled in transforming the `TddgSpinner` component. First, the VCL version of the `TddgSpinner` used the `SetTextAlign` GDI function to automatically center the text of the spinner in display area. However, under Linux, this API function doesn't exist. And even under Windows, this function wouldn't work because it expects a handle to a GDI device context, and CLX components don't have access to a device context   the `Canvas.Handle` property references a Qt Painter object.

Fortunately, most of the `TCanvas` methods do exist under both Windows and Linux. Therefore, we can circumvent this problem by calculating the center position manually.

The second display problem involves the `DrawButton()` method. In particular, the plus and minus symbols on the buttons are drawn using the `clActiveCaption` color in the VCL. Unfortunately, the `clActiveCaption` identifier is assigned to the `clActiveHighlightedText` value in the `QGraphics.pas` unit, which clearly isn't what we want.

NOTE

To perform any painting outside of your CLX component's `Paint()` method, you must first call the `Canvas.Start()` method and then call the `Canvas.Stop()` method when you are finished.

Not everything migrates as easily as you would have expected. The virtual key code constants defined in the VCL, such as `vk_Left`, aren't available in CLX. Instead, a completely new set of constants is used to determine which key was pressed. It turns out that the virtual key codes are part of the Windows API, and thus aren't available under Linux.

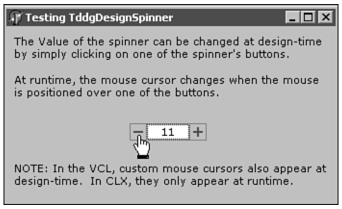
And that's it! We now have a fully functional custom CLX component that can be used in both Windows applications developed with Delphi 6 and Linux applications developed with Kylix. Of course, the most important aspect of this is that the same source code is used for both platforms.

Design-Time Enhancements

All things considered, migrating the `TddgSpinner` VCL component to CLX was fairly straightforward and not too tricky—although discovering the `InputKeys` property did take some effort.

However, as you shall see, once you start adding more functionality to our CLX components, the differences between the VCL and CLX will become evident.

Consider the source code displayed in Listing 13.2. This unit implements the `TddgDesignSpinner`, which is a descendant of `TddgSpinner`. Figure 13.4 illustrates how this component simply changes the mouse cursor whenever the mouse is positioned over one of the buttons. The descendant component also adds the ability to change the spinner value by clicking the plus or minus buttons directly on the form at design time as illustrated in Figure 13.5.


Testing TddgDesignSpinner

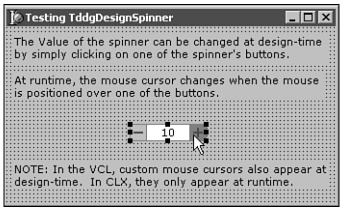
The Value of the spinner can be changed at design-time by simply clicking on one of the spinner's buttons.

At runtime, the mouse cursor changes when the mouse is positioned over one of the buttons.

11

NOTE: In the VCL, custom mouse cursors also appear at design-time. In CLX, they only appear at runtime.

FIGURE 13.4
The `TddgDesignSpinner` *displays a custom mouse cursor when the mouse is positioned over either button.*


Testing TddgDesignSpinner

The Value of the spinner can be changed at design-time by simply clicking on one of the spinner's buttons.

At runtime, the mouse cursor changes when the mouse is positioned over one of the buttons.

10

NOTE: In the VCL, custom mouse cursors also appear at design-time. In CLX, they only appear at runtime.

FIGURE 13.5
The `TddgDesignSpinner` *allows the Value property to be changed at design time by simply clicking on the component's buttons.*

LISTING 13.2 `QddgDsnSpin.pas`—Source Code for the `TddgDesignSpinner` Component

```
unit QddgDsnSpn;

interface

uses
  SysUtils, Classes, Qt, QddgSpin;
  (*
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  ddgSpin;
  *)
```

LISTING 13.2 Continued

```
type
  TddgDesignSpinner = class( TddgSpinner )
  private
    // VCL->CLX:  Custom cursor stored in QCursorH field
    FThumbCursor: QCursorH;

    (*
    // VCL->CLX:  Custom cursors and design-time interactions are
    //            handled differently under CLX. The following
    //            block is VCL-specific.

    FThumbCursor: HCursor;

    // Window Message Handling Method
    procedure WMSetCursor( var Msg : TWMSetCursor );
      message wm_SetCursor;

    // Component Message Handling Method
    procedure CMDesignHitTest( var Msg: TCMDesignHitTest );
      message cm_DesignHitTest;
    *)
  protected
    procedure Change; override;

    // VCL->CLX:  The following two methods are overridden for CLX
    procedure MouseMove( Shift: TShiftState;
                         X, Y: Integer ); override;
    function DesignEventQuery( Sender: QObjectH;
                               Event: QEventH ): Boolean; override;
  public
    constructor Create( AOwner: TComponent ); override;
    destructor Destroy; override;
  end;

implementation

(*
// VCL->CLX:  CLX does not support cursor resources
{$R DdgDsnSpn.res}                // Link in custom cursor resource
*)

uses
  Types, QControls, QForms;             // VCL->CLX:  Add CLX units
```

LISTING 13.2 Continued

```
// VCL->CLX:  Two arrays of bytes (one for the image and one for
//            the mask) are used to represent custom cursors in CLX

const
  Bits: array[0..32*4-1] of Byte = (
    $00, $30, $00, $00, $00, $48, $00, $00,
    $00, $48, $00, $00, $00, $48, $00, $00,
    $00, $48, $00, $00, $00, $4E, $00, $00,
    $00, $49, $C0, $00, $00, $49, $30, $00,
    $00, $49, $28, $00, $03, $49, $24, $00,
    $04, $C0, $24, $00, $04, $40, $04, $00,
    $02, $40, $04, $00, $02, $00, $04, $00,
    $01, $00, $04, $00, $01, $00, $04, $00,
    $00, $80, $08, $00, $00, $40, $08, $00,
    $00, $40, $08, $00, $00, $20, $10, $00,
    $00, $20, $10, $00, $00, $7F, $F8, $00,
    $00, $7F, $F8, $00, $00, $7F, $E8, $00,
    $00, $7F, $F8, $00, $00, $00, $00, $00,
    $00, $00, $00, $00, $00, $00, $00, $00,
    $00, $00, $00, $00, $00, $00, $00, $00,
    $00, $00, $00, $00, $00, $00, $00, $00 );

  Mask: array[0..32*4-1] of Byte = (
    $00, $30, $00, $00, $00, $78, $00, $00,
    $00, $78, $00, $00, $00, $78, $00, $00,
    $00, $78, $00, $00, $00, $7E, $00, $00,
    $00, $7F, $C0, $00, $00, $7F, $F0, $00,
    $00, $7F, $F8, $00, $03, $7F, $FC, $00,
    $07, $FF, $FC, $00, $07, $FF, $FC, $00,
    $03, $FF, $FC, $00, $03, $FF, $FC, $00,
    $01, $FF, $FC, $00, $01, $FF, $FC, $00,
    $00, $FF, $F8, $00, $00, $7F, $F8, $00,
    $00, $7F, $F8, $00, $00, $3F, $F0, $00,
    $00, $3F, $F0, $00, $00, $7F, $F8, $00,
    $00, $7F, $F8, $00, $00, $7F, $E8, $00,
    $00, $7F, $F8, $00, $00, $00, $00, $00,
    $00, $00, $00, $00, $00, $00, $00, $00,
    $00, $00, $00, $00, $00, $00, $00, $00,
    $00, $00, $00, $00, $00, $00, $00, $00 );


{==============================}
{== TddgDesignSpinner Methods ==}
{==============================}
```

LISTING 13.2 Continued

```
constructor TddgDesignSpinner.Create( AOwner: TComponent );
var
  BitsBitmap: QBitmapH;
  MaskBitmap: QBitmapH;
begin
  inherited Create( AOwner );

  (*
  // VCL->CLX:  No LoadCursor in CLX
  FThumbCursor := LoadCursor( HInstance, 'DdgDSNSPN_BTNCURSOR' );
  *)

  // VCL->CLX:  Byte arrays are used to create a custom cursor
  BitsBitmap := QBitmap_create( 32, 32, @Bits, False );
  MaskBitmap := QBitmap_create( 32, 32, @Mask, False );
  try
    FThumbCursor := QCursor_create( BitsBitmap, MaskBitmap, 8, 0 );
  finally
    QBitmap_destroy( BitsBitmap );
    QBitmap_destroy( MaskBitmap );
  end;
end;


destructor TddgDesignSpinner.Destroy;
begin
  (*
  VCL->CLX:  In CLX, use QCursor_Destroy instead of DestroyCursor
  DestroyCursor( FThumbCursor );       // Release GDI cursor object
  *)
  QCursor_Destroy( FThumbCursor );
  inherited Destroy;
end;


// If the mouse is over one of the buttons, then change cursor to
// the custom cursor that resides in the DdgDsnSpn.res
// resource file

(*
// VCL->CLX:  There is no wm_SetCursor in CLX
procedure TddgDesignSpinner.WMSetCursor( var Msg: TWMSetCursor );
begin
  if MouseOverButton( btMinus ) or MouseOverButton( btPlus ) then
    SetCursor( FThumbCursor )
```

LISTING 13.2 Continued

```
  else
    inherited;
end;
*)

// VCL->CLX:  Override MouseMove to handle displaying custom cursor

procedure TddgDesignSpinner.MouseMove( Shift: TShiftState;
                                      X, Y: Integer );
begin
  if MouseOverButton( btMinus ) or MouseOverButton( btPlus ) then
    QWidget_setCursor( Handle, FThumbCursor )
  else
    QWidget_UnsetCursor( Handle );
  inherited;
end;


(*
// VCL->CLX:  cm_DesignHitTest does not exist in CLX.  Instead,
//            override the DesignEventQuery method (see below).

procedure TddgDesignSpinner.CMDesignHitTest( var Msg:
                                            TCMDesignHitTest );
begin
  // Handling this component message allows the Value of the
  // spinner to be changed at design-time using the left mouse
  // button.  If the mouse is positioned over one of the buttons,
  // then set the Msg.Result value to 1. This instructs Delphi to
  // allow mouse events to "get through to" the component.

  if MouseOverButton( btMinus ) or MouseOverButton( btPlus ) then
    Msg.Result := 1
  else
    Msg.Result := 0;
end;
*)


function TddgDesignSpinner.DesignEventQuery( Sender: QObjectH;
                                         Event: QEventH ): Boolean;
var
  MousePos: TPoint;
begin
  Result := False;
```

Listing 13.2 Continued

```
  if ( Sender = Handle ) and
     ( QEvent_type(Event) in [QEventType_MouseButtonPress,
                              QEventType_MouseButtonRelease,
                              QEventType_MouseButtonDblClick]) then
  begin
    // Note: extracting MousePos is not actually needed in this
    //       example, but if you need to get the position of the
    //       mouse, this is how you do it.

    MousePos := Point( QMouseEvent_x( QMouseEventH( Event ) ),
                       QMouseEvent_y( QMouseEventH( Event ) ) );

    if MouseOverButton( btMinus ) or MouseOverButton( btPlus ) then
      Result := True
    else
      Result := False;
  end;
end;


procedure TddgDesignSpinner.Change;
var
  Form: TCustomForm;
begin
  inherited Change;

  // Force the Object Inspector to update what it shows for the
  // Value property of the spinner when changed via the mouse.

  if csDesigning in ComponentState then
  begin
    Form := GetParentForm( Self );

    (*
    // VCL->CLX:  Form.Designer replaced with DesignerHook in CLX
    if ( Form <> nil ) and ( Form.Designer <> nil ) then
      Form.Designer.Modified;
    *)

    if ( Form <> nil ) and ( Form.DesignerHook <> nil ) then
      Form.DesignerHook.Modified;
  end;
end;

end.
```

As you can see from the commented blocks of VCL-based code included in the source code, implementing these two features wasn't trivial because both features use messages in the VCL. As noted earlier, Linux doesn't use a message loop and thus CLX must use a different mechanism to implement these features.

First of all, specifying the mouse cursor to use in the control is much more complicated. In the VCL version, we simply attached a Windows resource file that included a custom cursor, and we then called the `LoadCursor` API function to get a reference to the cursor (a handle). This cursor handle is then used in handling the `wm_SetCursor` message, which Windows sends to any control that needs to have its mouse pointer updated.

Under CLX, this approach cannot be used. First, Qt doesn't support cursor resources. The `Qt.pas` unit defines several `QCursor_create()` methods—each providing a different way to construct a mouse cursor except from a cursor resource. You could specify one of the stock Qt cursors by passing an appropriate integer value to the `QCursor_create()` method. But, to create a custom cursor, you need to create two arrays of bytes that contain the bit layout for the cursor. The first array represents the black or white pixels, whereas the second array represents the mask, which determines which regions in the cursor are transparent.

Next, in order to display the mouse cursor at the appropriate time, we override the `MouseMove()` event dispatch method instead of handling the `wm_SetCursor` message. To change the cursor, the `QWidget_setCursor()` function is called whenever the mouse is positioned over either button. Otherwise, the `QWidget_UnsetCursor()` method is called.

In the VCL, handling the `cm_DesignHitTest` component message allows mouse events to be handled by a component at design-time. Unfortunately, this message doesn't exist in CLX. Instead, to accomplish the same features, we need to override the new `DesignEventQuery()` method. This method provides a way for component writers to become notified when the underlying Qt widget receives an input event at design-time. If the method returns `True`, the control should respond to the event. In our example, we are only concerned with mouse input events. Therefore, we must first determine whether the input event meets our criteria. If so, we must determine whether the mouse is positioned over one of the buttons.

The `Change()` method must be overridden in `TddgDesignSpinner` so that the Object Inspector's display of the `Value` property can remain in-sync with the selected component. If this method isn't overridden, the Object Inspector won't be updated as the user clicks directly on the spinner's button on the Form Designer. As you can see, the only change is the reference of `Form.Designer` to `Form.DesignerHook`.

Component References and Image Lists

The next component once again extends the functionality of the spinner. In particular, the `TddgImgListSpinner` component descends from the `TddgDesignSpinner` and implements a component reference property to allow the user to connect the spinner to an ImageList. The images in the ImageList can then be displayed in place of the plus and minus default symbols as shown in Figure 13.6.

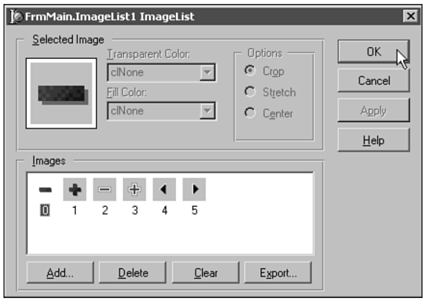


FIGURE 13.6
The `TddgImgListSpinner` *supports displaying images from an ImageList for each button.*

Listing 13.3 shows the complete source code for the `QddgILSpin.pas` unit, which implements the `TddgImgListSpinner` component. Unlike the `TddgDesignSpinner`, this component required very little changes in moving to CLX.

LISTING 13.3 `QddgILSpin.pas`—Source Code for the `TddgImgListSpinner` Component

```
unit QddgILSpin;

interface

uses
  Classes, Types, QddgSpin, QddgDsnSpn, QImgList;
  (*
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  ddgSpin, ddgDsnSpn, ImgList;
  *)

type
  TddgImgListSpinner = class( TddgDesignSpinner )
  private
    FImages: TCustomImageList;
    FImageIndexes: array[ 1..2 ] of Integer;
    FImageChangeLink: TChangeLink;

    // Internal Event Handlers
    procedure ImageListChange( Sender: TObject );
  protected
```

LISTING 13.3 Continued

```
    procedure Notification( AComponent : TComponent;
                            Operation : TOperation ); override;

    procedure DrawButton( Button: TddgButtonType; Down: Boolean;
                          Bounds: TRect ); override;
    procedure CalcCenterOffsets( Bounds: TRect; var L, T: Integer);

    procedure CheckMinSize;

    // Property Access Methods
    procedure SetImages( Value: TCustomImageList ); virtual;
    function GetImageIndex( PropIndex: Integer ): Integer; virtual;
    procedure SetImageIndex( PropIndex: Integer;
                             Value: Integer ); virtual;
  public
    constructor Create( AOwner: TComponent ); override;
    destructor Destroy; override;
  published
    property Images: TCustomImageList
      read FImages
      write SetImages;

    property ImageIndexMinus: Integer
      index 1
      read GetImageIndex
      write SetImageIndex;

    property ImageIndexPlus: Integer
      index 2
      read GetImageIndex
      write SetImageIndex;
  end;

implementation

uses
  QGraphics;                   // VCL->CLX:  Added for CLX support


{================================}
{== TddgImgListSpinner Methods ==}
{================================}

constructor TddgImgListSpinner.Create( AOwner: TComponent );
```

LISTING 13.3 Continued

```
begin
  inherited Create( AOwner );

  FImageChangeLink := TChangeLink.Create;
  FImageChangeLink.OnChange := ImageListChange;
  // NOTE: Since, the component user does not have direct access to
  // the change link, the user cannot assign custom event handlers.

  FImageIndexes[ 1 ] := -1;
  FImageIndexes[ 2 ] := -1;
end;


destructor TddgImgListSpinner.Destroy;
begin
  FImageChangeLink.Free;
  inherited Destroy;
end;


procedure TddgImgListSpinner.Notification( AComponent: TComponent;
                                           Operation: TOperation );
begin
  inherited Notification( AComponent, Operation );
  if ( Operation = opRemove ) and ( AComponent = FImages ) then
    SetImages( nil );             // Note the call to access method
end;


function TddgImgListSpinner.GetImageIndex( PropIndex:
                                           Integer ): Integer;
begin
  Result := FImageIndexes[ PropIndex ];
end;

procedure TddgImgListSpinner.SetImageIndex( PropIndex: Integer;
                                            Value: Integer );
begin
  if FImageIndexes[ PropIndex ] <> Value then
  begin
    FImageIndexes[ PropIndex ] := Value;
    Invalidate;
  end;
end;
```

LISTING 13.3 Continued

```
procedure TddgImgListSpinner.SetImages( Value: TCustomImageList );
begin
  if FImages <> nil then
    FImages.UnRegisterChanges( FImageChangeLink );

  FImages := Value;

  if FImages <> nil then
  begin
    FImages.RegisterChanges( FImageChangeLink );
    FImages.FreeNotification( Self );
    CheckMinSize;
  end;
  Invalidate;
end;

procedure TddgImgListSpinner.ImageListChange( Sender: TObject );
begin
  if Sender = Images then
  begin
    CheckMinSize;
    // Call Update instead of Invalidate to prevent flicker
    Update;
  end;
end;

procedure TddgImgListSpinner.CheckMinSize;
begin
  // Ensures button area will display entire image
  if FImages.Width > ButtonWidth then
    ButtonWidth := FImages.Width;
  if FImages.Height > Height then
    Height := FImages.Height;
end;


procedure TddgImgListSpinner.DrawButton( Button: TddgButtonType;
                                         Down: Boolean;
                                         Bounds: TRect );
var
  L, T: Integer;
begin
  with Canvas do
  begin
```

LISTING 13.3 Continued

```
    Brush.Color := ButtonColor;
    Pen.Color := clBtnShadow;
    Rectangle( Bounds.Left, Bounds.Top,
               Bounds.Right, Bounds.Bottom );

    if Button = btMinus then           // Draw the Minus (-) Button
    begin
      if ( Images <> nil ) and ( ImageIndexMinus <> -1 ) then
      begin
        (*
        // VCL->CLX:  DrawingStyle does not exist in CLX TImageList
        //            BkColor is used instead.
        if Down then
          FImages.DrawingStyle := dsSelected
        else
          FImages.DrawingStyle := dsNormal;
        *)
        if Down then
          FImages.BkColor := clBtnShadow
        else
          FImages.BkColor := clBtnFace;

        CalcCenterOffsets( Bounds, L, T );

        (*
        // VCL->CLX:  TImageList.Draw is different in CLX
        FImages.Draw( Canvas, L, T, ImageIndexMinus, Enabled );
        *)
        FImages.Draw( Canvas, L, T, ImageIndexMinus, itImage,
                      Enabled );
      end
      else
        inherited DrawButton( Button, Down, Bounds );
    end
    else                                // Draw the Plus (+) Button
    begin
      if ( Images <> nil ) and ( ImageIndexPlus <> -1 ) then
      begin
        (*
        // VCL->CLX:  DrawingStyle does not exist in CLX TImageList
        //            BkColor is used instead.
        if Down then
          FImages.DrawingStyle := dsSelected
        else
```

LISTING 13.3 Continued

```
          FImages.DrawingStyle := dsNormal;
        *)
        if Down then
          FImages.BkColor := clBtnShadow
        else
          FImages.BkColor := clBtnFace;

        CalcCenterOffsets( Bounds, L, T );

        (*
        // VCL->CLX:  TImageList.Draw is different in CLX
        FImages.Draw( Canvas, L, T, ImageIndexPlus, Enabled );
        *)
        FImages.Draw( Canvas, L, T, ImageIndexPlus, itImage,
                      Enabled );
      end
      else
        inherited DrawButton( Button, Down, Bounds );
    end;
  end;
end; {= TddgImgListSpinner.DrawButton =}


procedure TddgImgListSpinner.CalcCenterOffsets( Bounds: TRect;
                                               var L, T: Integer );
begin
  if FImages <> nil then
  begin
    L := Bounds.Left + ( Bounds.Right - Bounds.Left ) div 2 -
         ( FImages.Width div 2 );
    T := Bounds.Top + ( Bounds.Bottom - Bounds.Top ) div 2 -
         ( FImages.Height div 2 );
  end;
end;

end.
```

As usual, the `uses` clause of the unit needs to be changed to include the CLX specific units and to remove the VCL specific ones. In particular, notice that the `QImgList` unit replaces the `ImgList` unit. This is significant because under the VCL, the `TCustomImageList` component is a wrapper around the ImageList common control implemented in the `ComCtl32.dll`. Borland created a CLX version of the `TCustomImageList` component that uses the graphics primitives of Qt instead of the `ComCtl32.dll`.

The benefit of this is clearly visible in the class declaration. The declaration of the CLX version of `TddgImgListSpinner` is identical to the VCL version. Furthermore, the implementations of all but one of the component's methods are also identical.

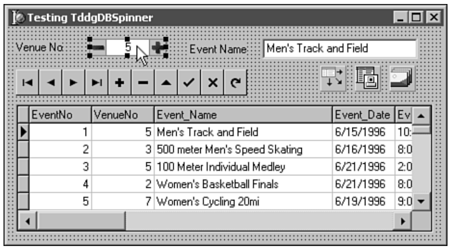
Of course, it is the single display method, the overridden `DrawButton()` method, that requires some tweaking. In particular, two issues need to be addressed. The first illustrates a key point in comparing classes that exist in both the VCL and CLX. That is, just because a VCL class has a corresponding class in CLX, it doesn't necessarily mean that all the functionality of the VCL class is also available in the CLX version.

In the case of the `TCustomImageList` class, the VCL version implements the `DrawingStyle` property, which is used by the VCL version of the `TddgImgListSpinner` to display the button's image differently when clicked. The `DrawingStyle` property doesn't exist in the CLX version, and therefore a different approach must be taken.

The second modification to the `DrawButton()` method results from the `TCustomImageList.Draw()` method being different between the two architectures.

Data-Aware CLX Components

In this fourth sample component, data awareness is added to the spinner component. That is, the `TddgDBSpinner` component can be connected to an integer field in a dataset through its `DataSource` and `DataField` properties. Figure 13.7 shows a `TddgDBSpinner` component connected to the `VenueNo` field of the `Events` dataset.



Venue No: 5 Event Name: Men's Track and Field

	EventNo	VenueNo	Event_Name	Event_Date	Ev
▶	1	5	Men's Track and Field	6/15/1996	10:
	2	3	500 meter Men's Speed Skating	6/16/1996	8:0
	3	5	100 Meter Individual Medley	6/21/1996	2:0
	4	2	Women's Basketball Finals	6/21/1996	8:0
	5	7	Women's Cycling 20mi	6/19/1996	9:0

FIGURE 13.7
The `TddgDBSpinner` *can be used to display and edit integer fields in a dataset.*

Listing 13.4 shows the source code for the `QddgDBSpin.pas` unit, which implements the `TddgDBSpinner` component, which in turn descends from `TddgImgListSpinner`.

LISTING 13.4 `QddgDBSpin.pas`—Source Code for the `TddgDBSpinner` Component

```
unit QddgDBSpin;

interface

uses
  SysUtils, Classes, Qt, QddgILSpin, DB, QDBCtrls;
  (*
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  ddgILSpin, DB, DBCtrls;
  *)

type
  TddgDBSpinner = class( TddgImgListSpinner )
  private
    FDataLink: TFieldDataLink;        // Provides Access to Data

    // Internal Event Handlers for DataLink Events
    procedure DataChange( Sender: TObject );
    procedure UpdateData( Sender: TObject );
    procedure ActiveChange( Sender: TObject );

    (*
    // VCL->CLX:  Component Message handling methods not in CLX
    procedure CMExit( var Msg: TCMExit ); message cm_Exit;
    procedure CMDesignHitTest( var Msg: TCMDesignHitTest );
      message cm_DesignHitTest;
    *)
  protected
    procedure Notification( AComponent : TComponent;
                            Operation : TOperation ); override;
    procedure CheckFieldType( const Value: string ); virtual;

    // Overridden event dispatch methods
    procedure Change; override;
    procedure KeyPress( var Key : Char ); override;

    // VCL->CLX:  DoExit replaces CMExit
    procedure DoExit; override;
    // VCL->CLX:  DesignEventQuery replaces CMDesignHitTest
    function DesignEventQuery( Sender: QObjectH;
                               Event: QEventH ): Boolean; override;
```

LISTING 13.4 Continued

```
    // Overridden support methods
    procedure DecValue( Amount: Integer ); override;
    procedure IncValue( Amount: Integer ); override;

    // Property Access Methods
    function GetField: TField; virtual;
    function GetDataField: string; virtual;
    procedure SetDataField( const Value: string ); virtual;
    function GetDataSource: TDataSource; virtual;
    procedure SetDataSource( Value: TDataSource ); virtual;
    function GetReadOnly: Boolean; virtual;
    procedure SetReadOnly( Value: Boolean ); virtual;

    // Give Descendants Access to Field object and DataLink
    property Field: TField
      read GetField;

    property DataLink: TFieldDataLink
      read FDataLink;
  public
    constructor Create( AOwner: TComponent ); override;
    destructor Destroy; override;
  published
    property DataField: string
      read GetDataField
      write SetDataField;

    property DataSource: TDataSource
      read GetDataSource
      write SetDataSource;

    // This property controls the ReadOnly state of the DataLink
    property ReadOnly: Boolean
      read GetReadOnly
      write SetReadOnly
      default False;
  end;

type
  EInvalidFieldType = class( Exception );

resourcestring
  SInvalidFieldType = 'DataField can only be connected to ' +
                      'columns of type Integer, Smallint, Word, ' +
                      'and Float';
```

LISTING 13.4 Continued

```
implementation

uses
  Types;                          // VCL->CLX:  Added for CLS support


{===========================}
{== TddgDBSpinner Methods ==}
{===========================}

constructor TddgDBSpinner.Create( AOwner: TComponent );
begin
  inherited Create( AOwner );

  FDataLink := TFieldDataLink.Create;

  // To support the TField.FocusControl method, set the
  // FDataLink.Control property to point to the spinner.
  // The Control property requires a TWinControl component.
  FDataLink.Control := Self;

  // Assign Event Handlers
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
  FDataLink.OnActiveChange := ActiveChange;

  // NOTE: Since, the component user does not have direct access to
  // the data link, the user cannot assign custom event handlers.
end;


destructor TddgDBSpinner.Destroy;
begin
  FDataLink.Free;
  FDataLink := nil;
  inherited Destroy;
end;


procedure TddgDBSpinner.Notification( AComponent: TComponent;
                                      Operation: TOperation );
begin
  inherited Notification( AComponent, Operation );
  if ( Operation = opRemove ) and
     ( FDataLink <> nil ) and
```

LISTING 13.4 Continued

```
     ( AComponent = FDataLink.DataSource ) then
  begin
    DataSource := nil;            // Indirectly calls SetDataSource
  end;
end;


function TddgDBSpinner.GetField: TField;
begin
  Result := FDataLink.Field;
end;


function TddgDBSpinner.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

procedure TddgDBSpinner.SetDataField( const Value: string );
begin
  CheckFieldType( Value );
  FDataLink.FieldName := Value;
end;


function TddgDBSpinner.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TddgDBSpinner.SetDataSource( Value: TDataSource );
begin
  if FDatalink.DataSource <> Value then
  begin
    FDataLink.DataSource := Value;

    // FreeNotification must be called b/c DataSource may be
    // located on another form or data module.
    if Value <> nil then
      Value.FreeNotification( Self );
  end;
end;
```

LISTING 13.4 Continued

```
function TddgDBSpinner.GetReadOnly: Boolean;
begin
  Result := FDataLink.ReadOnly;
end;

procedure TddgDBSpinner.SetReadOnly( Value: Boolean );
begin
  FDataLink.ReadOnly := Value;
end;



procedure TddgDBSpinner.CheckFieldType( const Value: string );
var
  FieldType: TFieldType;
begin
  // Make sure the field type corresponding to the column
  // referenced by Value is either ftInteger, ftSmallInt, ftWord,
  // or ftFloat.  If it is not, an EInvalidFieldType exception is
  // raised.

  if ( Value <> '' ) and
     ( FDataLink <> nil ) and
     ( FDataLink.Dataset <> nil ) and
     ( FDataLink.Dataset.Active ) then
  begin
    FieldType := FDataLink.Dataset.FieldByName( Value ).DataType;
    if ( FieldType <> ftInteger ) and
       ( FieldType <> ftSmallInt ) and
       ( FieldType <> ftWord ) and
       ( FieldType <> ftFloat ) then
    begin
      raise EInvalidFieldType.Create( SInvalidFieldType );
    end;
  end;
end;


procedure TddgDBSpinner.Change;
begin
  // Tell the FDataLink that the data has changed
  if FDataLink <> nil then
    FDataLink.Modified;
  inherited Change;                     // Generates OnChange event
end;
```

LISTING 13.4 Continued

```
procedure TddgDBSpinner.KeyPress( var Key: Char );
begin
  inherited KeyPress( Key );

  if Key = #27 then
  begin
    FDataLink.Reset;                                 // Esc key pressed
    Key := #0;                // Set to #0 so Esc won't close dialog
  end;
end;


procedure TddgDBSpinner.DecValue( Amount: Integer );
begin
  if ReadOnly or not FDataLink.CanModify then
  begin
    // Prevent change if FDataLink is ReadOnly
    (*
    // VCL->CLX:  MessageBeep is a Windows API function
    MessageBeep( 0 )
    *)
    Beep;
  end
  else
  begin
    // Try to put Dataset in edit mode--only dec if in edit mode
    if FDataLink.Edit then
      inherited DecValue( Amount );
  end;
end;


procedure TddgDBSpinner.IncValue( Amount: Integer );
begin
  if ReadOnly or not FDataLink.CanModify then
  begin
    // Prevent change if FDataLink is ReadOnly
    (*
    // VCL->CLX:  MessageBeep is a Windows API function
    MessageBeep( 0 )
    *)
    Beep;
  end
  else
```

LISTING 13.4 Continued

```
  begin
    // Try to put Dataset in edit mode--only inc if in edit mode
    if FDataLink.Edit then
      inherited IncValue( Amount );
  end;
end;


{===================================================================
  TddgDBSpinner.DataChange

  This method gets called as a result of a number of
  different events:

  1. The underlying field value changes.  Occurs when changing the
     value of the column tied to this control and then move to a
     new column or a new record.
  2. The corresponding Dataset goes into Edit mode.
  3. The corresponding Dataset referenced by DataSource changes.
  4. The current cursor is scrolled to a new record in the table.
  5. The record is reset through a Cancel call.
  6. The DataField property changes to reference another column.
===================================================================}

procedure TddgDBSpinner.DataChange( Sender: TObject );
begin
  if FDataLink.Field <> nil then
    Value := FDataLink.Field.AsInteger;
end;


{===================================================================
  TddgDBSpinner.UpdateData

  This method gets called when the corresponding field value and
  the contents of the Spinner need to be synchronized.  Note that
  this method only gets called if this control was responsible for
  altering the data.
===================================================================}

procedure TddgDBSpinner.UpdateData( Sender: TObject );
begin
  FDataLink.Field.AsInteger := Value;
end;
```

Listing 13.4 Continued

```
{===================================================================
  TddgDBSpinner.ActiveChange

  This method gets called whenever the Active property of the
  attached Dataset changes.

  NOTE: You can use the FDataLink.Active property to determine
        the *new* state of the Dataset.
===================================================================}

procedure TddgDBSpinner.ActiveChange( Sender: TObject );
begin
  // If the Dataset is becoming Active, then check to make sure the
  // field type of the DataField property is a valid type.

  if ( FDataLink <> nil ) and FDataLink.Active then
    CheckFieldType( DataField );
end;


(*
// VCL->CLX:  CMExit replaced with DoExit (see below)

procedure TddgDBSpinner.CMExit( var Msg: TCMExit );
begin
  try   // Attempt to update the record if focus leaves the spinner
    FDataLink.UpdateRecord;
  except
    SetFocus;      // Keep the focus on the control if Update fails
    raise;                                  // Reraise the exception
  end;
  inherited;
end;
*)

procedure TddgDBSpinner.DoExit;
begin
  try   // Attempt to update the record if focus leaves the spinner
    FDataLink.UpdateRecord;
  except
    SetFocus;      // Keep the focus on the control if Update fails
    raise;                                  // Reraise the exception
  end;
  inherited;
end;
```

LISTING 13.4 Continued

```
(*
// VCL->CLX:  CMDesignHitTest replaced by DesignEventQuery

procedure TddgDBSpinner.CMDesignHitTest(var Msg: TCMDesignHitTest);
begin
  // Ancestor component allows Value to be changed at design-time.
  // This is not valid in a data-aware component because it would
  // put the connected dataset into edit mode.
  Msg.Result := 0;
end;
*)

function TddgDBSpinner.DesignEventQuery( Sender: QObjectH;
                                         Event: QEventH ): Boolean;
begin
  // Ancestor component allows Value to be changed at design-time.
  // This is not valid in a data-aware component because it would
  // put the connected dataset into edit mode.
  Result := False;
end;

end.
```

Fortunately, incorporating data awareness into a CLX component is nearly identical to the VCL implementation. That is, once you have a working nondata-aware CLX component, you simply need to embed a `TFieldDataLink` object into your CLX component and respond to the `DataChange` and `UpdateData` events. Of course, you will need to implement the `DataSource`, `DataField`, and `ReadOnly` properties, but this is no different from doing the same thing in a VCL component.

NOTE

Don't forget to change the `DBCtrls` unit to `QDBCtrls`. Although the `DB` unit is shared between the VCL and CLX, the `DBCtrls` unit isn't. Both `DBCtrls` and `QDBCtrls` define a `TFieldDataLink` class. Unfortunately, under Delphi 6, you won't receive any errors if you use the VCL version of the `TFieldDataLink` instead of the CLX version. In fact, the component might even operate correctly under Windows. However, when you try the component under Kylix, you will receive many syntax errors from the compiler.

However, one situation will require your attention. Many data-aware VCL components handle the `cm_Exit` component message in order to call the `UpdateRecord` method of the data link.

However, CLX doesn't implement the cm_Exit message, and therefore the DoExit() event dispatch method must be overridden instead.

The TddgDBSpinner is a direct descendant of TddgImgListSpinner, which in turn descends from TddgDesignSpinner. Recall that one of the features of the TddgDesignSpinner was to allow the user to change the value of the spinner using the mouse at design time. This feature is no longer useful in our data-aware component because if the user changes the value of the spinner, the associated dataset will be placed into edit mode. Unfortunately, at design-time there is no way to get out of edit mode once this happens. Therefore, the TddgDBSpinner overrides the DesignEventQuery() method and simply returns False to prevent mouse operations from being handled by the component at design-time.

CLX Design Editors

Design editors for CLX components are implemented in exactly the same way they are for VCL components. However, there are few changes that you must be aware of. The most significant is that the units that implement the base design-time functionality have been broken up and placed into new units. Specifically, the DsgnIntf unit has been renamed to DesignIntf. In most cases, you will also need to add the new DesignEditors unit to your uses clause. The DesignIntf unit defines the interfaces used by the Form Designer and Object Inspector. The DesignEditors unit implements the basic property editor and component editor classes.

Unfortunately, not all the design-time features of the VCL have made it over into CLX. For example, owner-draw property editors are only available in the VCL. As a result, CLX specific editors are implemented in the CLXEditors unit, whereas VCL specific editors are defined in the VCLEditors unit.

Figure 13.8 shows the TddgRadioGroupEditor, a custom component editor for the CLX TRadioGroup component, allowing a user to easily set the ItemIndex property. The TddgRadioGroupEditor is defined in the QddgRgpEdt.pas unit, which appears in Listing 13.5.

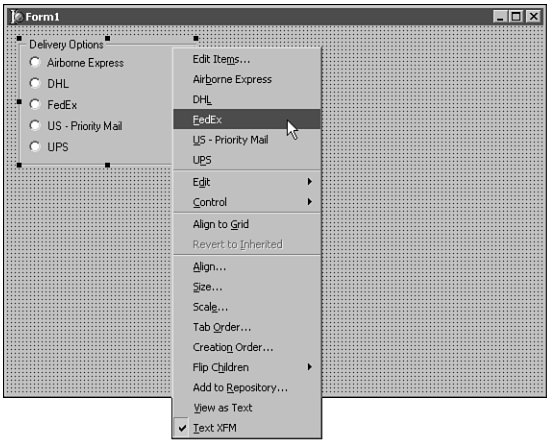


FIGURE 13.8

Selecting an item in the CLX RadioGroup is a snap with this custom component editor.

LISTING 13.5 `QddgRgpEdt.pas`—Source Code for the `TddgRadioGroupEditor` Component Editor

```
unit QddgRgpEdt;

interface

uses
  DesignIntf, DesignEditors, QExtCtrls, QDdgDsnEdt;

type
  TddgRadioGroupEditor = class( TddgDefaultEditor )
  protected
    function RadioGroup: TRadioGroup; virtual;
  public
    function GetVerbCount: Integer; override;
    function GetVerb( Index: Integer ) : string; override;
    procedure ExecuteVerb( Index: Integer ); override;
  end;


implementation

uses
  QControls;
```

LISTING 13.5 Continued

```
{==================================}
{== TddgRadioGroupEditor Methods ==}
{==================================}

function TddgRadioGroupEditor.RadioGroup: TRadioGroup;
begin
  // Helper function to provide quick access to component being
  // edited.  Also makes sure Component is a TRadioGroup
  Result := Component as TRadioGroup;
end;

function TddgRadioGroupEditor.GetVerbCount: Integer;
begin
  // Return the number of new menu items to display
  Result := RadioGroup.Items.Count + 1;
end;


function TddgRadioGroupEditor.GetVerb( Index: Integer ): string;
begin
  // Menu item caption for context menu
  if Index = 0 then
    Result := 'Edit Items...'
  else
    Result := RadioGroup.Items[ Index - 1 ];
end;


procedure TddgRadioGroupEditor.ExecuteVerb( Index: Integer );
begin
  if Index = 0 then
    EditPropertyByName( 'Items' )      // Defined in QDdgDsnEdt.pas
  else
  begin
    if RadioGroup.ItemIndex <> Index - 1 then
      RadioGroup.ItemIndex := Index - 1
    else
      RadioGroup.ItemIndex := -1;              // Uncheck all items
    Designer.Modified;
  end;
end;

end.
```

The techniques illustrated in the `TddgRadioGroupEditor` apply to both the VCL and CLX. In this example, the context menu of the `TRadioGroup` component is changed to reflect the items currently in the group. Selecting a group item's corresponding menu item causes the radio group's `ItemIndex` property to be set accordingly. If no items are in the group, only the Edit Items menu item is added.

If the user chooses this menu item, the string list editor is invoked on the Items property for the `TRadioGroup`. The `EditPropertyByName()` method isn't part of CLX or the VCL—it is defined in the `TddgDefaultEditor` class. This method can be used to invoke the currently registered property editor for any named property of a component within the context of a component editor. Listing 13.6 shows the source code for the `QddgDsnEdt.pas` unit, which implements the `TddgDefaultEditor` class.

LISTING 13.6 `QddgDsnEdt.pas`—Source Code for the `TddgDefaultEditor` Component Editor

```
unit QddgDsnEdt;

interface

uses
  Classes, DesignIntf, DesignEditors;

type
  TddgDefaultEditor = class( TDefaultEditor )
  private
    FPropName: string;
    FContinue: Boolean;
    FPropEditor: IProperty;
    procedure EnumPropertyEditors(const PropertyEditor: IProperty);
    procedure TestPropertyEditor( const PropertyEditor: IProperty;
                                  var Continue: Boolean );
  protected
    procedure EditPropertyByName( const APropName: string );
  end;


implementation

uses
  SysUtils, TypInfo;

{================================}
{== TddgDefaultEditor Methods ==}
{================================}
```

LISTING 13.6 Continued

```
procedure TddgDefaultEditor.EnumPropertyEditors( const
                                       PropertyEditor: IProperty );
begin
  if FContinue then
    TestPropertyEditor( PropertyEditor, FContinue );
end;


procedure TddgDefaultEditor.TestPropertyEditor( const
                                         PropertyEditor: IProperty;
                                         var Continue: Boolean );
begin
  if not Assigned( FPropEditor ) and
     ( CompareText( PropertyEditor.GetName, FPropName ) = 0 ) then
  begin
    Continue := False;
    FPropEditor := PropertyEditor;
  end;
end;


procedure TddgDefaultEditor.EditPropertyByName( const
                                               APropName: string );
var
  Components: IDesignerSelections;
begin
  Components := TDesignerSelections.Create;
  FContinue := True;
  FPropName := APropName;
  Components.Add( Component );
  FPropEditor := nil;
  try
    GetComponentProperties( Components, tkAny, Designer,
                            EnumPropertyEditors );
    if Assigned( FPropEditor ) then
      FPropEditor.Edit;
  finally
    FPropEditor := nil;
  end;
end;

end.
```

Packages

CLX components, like VCL components, need to be placed into a package in order to be installed into the Kylix or Delphi IDEs. However, it is important to note that a compiled Delphi 6 package containing a CLX component cannot be installed into Kylix. This is because packages under Windows are implemented as specially compiled DLLs, whereas packages under Linux are implemented as shared object (.so) files. Fortunately, the format and syntax of package source files under both platforms is identical.

However, the information that you need to provide in the packages will differ between Windows and Linux. For example, the `requires` clause for a Linux runtime package will usually specify the `baseclx` and `visualclx` packages. However, `baseclx` doesn't exist in Delphi 6. Under Windows, a runtime package containing CLX components will only require the `visualclx` package. Of course, as with VCL packages, CLX design packages will require the runtime packages containing your new custom CLX components.

Naming Conventions

The CLX components presented in this chapter are contained in the packages described in Tables 13.1 and 13.2. Table 13.1 shows the BPL files generated under Windows and also lists the packages required for each custom package. Table 13.2 shows the shared object files generated under Linux and likewise lists the required packages. The package source files are the same in both tables. As you can see, we've adopted a specific naming convention for package names.

TABLE 13.1 CLX Packages for Windows (Delphi 6)

Package Source	*Compiled Version*	*Requires*
`QddgSamples.dpk`	`QddgSamples60.bpl`	`visualclx`

`QddgSamples_Dsgn.dpk`	`QddgSamples_Dsgn60.bpl`	`visualclx`
		`designide`
		`QddgSamples`

`QddgDBSamples.dpk`	`QddgDBSamples60.bpl`	`visualclx`
		`dbrtl`
		`visualdbclx`
		`QddgSamples`

`QddgDBSamples_Dsgn.dpk`	`QddgDBSamples_Dsgn60.bpl`	`visualclx`
		`QddgSamples_Dsgn`
		`QddgDBSamples`

TABLE 13.2 CLX Packages for Linux (Kylix)

Package Source	*Compiled Version*	*Requires*
`QddgSamples.dpk`	`bplQddg6Samples.so.6`	`baseclx` `visualclx`

`QddgSamples_Dsgn.dpk`	`bplQddgSamples_Dsgn.so.6`	`baseclx`
		`visualclx`
		`designide`
		`QddgSamples`

`QddgDBSamples.dpk`	`bplQddgDBSamples.so.6`	`baseclx`
		`visualclx`
		`visualdbclx`
		`dataclx`
		`QddgSamples`

`QddgDBSamples_Dsgn.dpk`	`bplQddgDBSamples_Dsgn.so.6`	`baseclx`
		`visualclx`
		`QddgSamples_Dsgn`
		`QddgDBSamples`

CLX packages to be used under Windows typically incorporate the product version in the name. For example, `QddgSamples60.bpl` indicates that this file is for Delphi 6 and is a Borland Package Library as noted by the `.bpl` extension. Under Linux, Borland has chosen to follow traditional Linux practices in naming shared objects. For example, rather than use an extension to indicate the type of file, a `bpl` prefix is used to indicate a package. Borland will occasionally name some of its design packages with the `dcl` prefix. However, we discourage this practice because design packages are more clearly identified by the `_Dsgn` suffix. In addition, all packages (runtime and design) under Windows use a `.bpl` extension. Using a `bpl` prefix for all packages under Linux establishes a certain level of consistency. The suffix used for a Linux shared object is typically `.so` followed by a version number. The prefix and suffix used in generating a compiled package are controlled through the package options.

You will also note that the package source files don't specify a version number. In previous versions of Delphi, it was common practice to add a suffix to the package name to indicate which version of the VCL the package required. However, starting in Kylix and Delphi 6, Borland has added several new options controlling the names used when compiling package. In the Delphi 6 examples in Table 13.1, the new `{$LIBSUFFIX}` option is used to specify `60`. When Delphi compiles the package, the suffix is automatically added to the end of the `bpl` file. In the Kylix examples in Table 13.2, the `{$SOPREFIX}` directive specifies `bpl`, whereas the `{$SOVERSION}` directive specifies `6`.

Runtime Packages

Listings 13.7 and 13.8 show the source code for the nondata-aware and data-aware runtime packages containing the components presented in this chapter. Note the use of the conditional symbols MSWINDOWS and LINUX to specify the appropriate directives and to include the appropriate packages in the requires clause. MSWINDOWS is defined when compiling under Delphi 6, whereas LINUX is defined when compiling under Kylix.

LISTING 13.7 QddgSamples.dpk—Source Code for the NonData-Aware CLX Runtime Package

```
package QddgSamples;

{$R *.res}
{$ALIGN 8}
{$ASSERTIONS ON}
{$BOOLEVAL OFF}
{$DEBUGINFO ON}
{$EXTENDEDSYNTAX ON}
{$IMPORTEDDATA ON}
{$IOCHECKS ON}
{$LOCALSYMBOLS ON}
{$LONGSTRINGS ON}
{$OPENSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$REFERENCEINFO OFF}
{$SAFEDIVIDE OFF}
{$STACKFRAMES OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST ON}
{$MINENUMSIZE 1}
{$IMAGEBASE $400000}
{$DESCRIPTION 'DDG: CLX Components'}
 b
{$IFDEF MSWINDOWS}
{$LIBSUFFIX '60'}
{$ENDIF}

{$IFDEF LINUX}
{$SOPREFIX 'bpl'}
{$SOVERSION '6'}
{$ENDIF}
```

LISTING 13.7 Continued

```
{$RUNONLY}
{$IMPLICITBUILD OFF}

requires
  {$IFDEF LINUX}
  baseclx,
  {$ENDIF}
  visualclx;

contains
  QddgSpin in 'QddgSpin.pas',
  QddgDsnSpn in 'QddgDsnSpn.pas',
  QddgILSpin in 'QddgILSpin.pas';

end.
```

NOTE

When specifying platform specific blocks of code, use separate `{$IDFEF}..{$ENDIF}` blocks for each platform as illustrated in the package source files. In particular, you want to avoid constructs such as the following:

```
{$IFDEF MSWINDOWS}
// Windows specific code here
{$ELSE}
// Linux specific code here
{$ENDIF}
```

If Borland ever decides to support another platform, the preceding construct will cause the Linux specific code to be used as long as the platform isn't Windows.

LISTING 13.8 `QddgDBSamples.dpk`—Source Code for the Data-Aware CLX Runtime Package

```
package QddgDBSamples;

{$R *.res}
{$ALIGN 8}
{$ASSERTIONS ON}
{$BOOLEVAL OFF}
{$DEBUGINFO ON}
{$EXTENDEDSYNTAX ON}
{$IMPORTEDDATA ON}
{$IOCHECKS ON}
```

LISTING 13.8 Continued

```
{$LOCALSYMBOLS ON}
{$LONGSTRINGS ON}
{$OPENSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$REFERENCEINFO OFF}
{$SAFEDIVIDE OFF}
{$STACKFRAMES OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST ON}
{$MINENUMSIZE 1}
{$IMAGEBASE $400000}
{$DESCRIPTION 'DDG: CLX Components (Data-Aware)'}

{$IFDEF MSWINDOWS}
{$LIBSUFFIX '60'}
{$ENDIF}

{$IFDEF LINUX}
{$SOPREFIX 'bpl'}
{$SOVERSION '6'}
{$ENDIF}

{$RUNONLY}
{$IMPLICITBUILD OFF}

requires
  {$IFDEF MSWINDOWS}
  dbrtl,
  {$ENDIF}

  {$IFDEF LINUX}
  baseclx,
  dataclx,
  {$ENDIF}

  visualclx,
  visualdbclx,
  QddgSamples;

contains
  QddgDBSpin in 'QddgDBSpin.pas';

end.
```

Design-Time Packages

Although it is possible to put your custom components into a combination runtime/design package, this approach isn't recommended. In fact, this approach only works if you don't have any design editors included in your package. If you do, your package will require the `designide` package, which cannot be redistributed.

The solution is to create separate design packages that handle registering the components contained in your runtime packages. Listings 13.9 and 13.10 show the source code for the non-data-aware and data-aware design packages, respectively. Again note the use of the conditional symbols `MSWINDOWS` and `LINUX` to specify the appropriate directives and to include the appropriate packages in the `requires` clause.

LISTING 13.9 `QddgSamples_Dsgn.dpk`—Source Code for the Nondata-Aware CLX Design-Time Package

```
package QddgSamples_Dsgn;

{$R *.res}
{$R 'QddgSamples_Reg.dcr'}
{$ALIGN 8}
{$ASSERTIONS OFF}
{$BOOLEVAL OFF}
{$DEBUGINFO OFF}
{$EXTENDEDSYNTAX ON}
{$IMPORTEDDATA ON}
{$IOCHECKS ON}
{$LOCALSYMBOLS OFF}
{$LONGSTRINGS ON}
{$OPENSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$REFERENCEINFO OFF}
{$SAFEDIVIDE OFF}
{$STACKFRAMES OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST ON}
{$MINENUMSIZE 1}
{$IMAGEBASE $400000}
{$DESCRIPTION 'DDG: CLX Components'}

{$IFDEF MSWINDOWS}
{$LIBSUFFIX '60'}
{$ENDIF}
```

LISTING 13.9 Continued

```
{$IFDEF LINUX}
{$SOPREFIX 'bpl'}
{$SOVERSION '6'}
{$ENDIF}

{$DESIGNONLY}
{$IMPLICITBUILD OFF}

requires
  {$IFDEF LINUX}
  baseclx,
  {$ENDIF}
  visualclx,
  designide,
  QddgSamples;

contains
  QddgSamples_Reg in 'QddgSamples_Reg.pas',
  QddgDsnEdt in 'QddgDsnEdt.pas',
  QddgRgpEdt in 'QddgRgpEdt.pas';

end.
```

NOTE

In order to support both Kylix and Delphi 6 with the same package source files, the names specified in the requires and contains clauses must match the case of the actual filename. For example, in the VCL it is common to specify the `DesignIDE` package in mixed case. However, under Linux, the `designide` package uses all lowercase letters. If mixed case is used in the source file, Kylix won't be able to locate the `designide.dcp` file because `DesignIDE.dcp` is different from `designide.dcp` on Linux.

LISTING 13.10 `QddgDBSamples_Dsgn.dpk`—Source Code for the Data-Aware CLX Design-Time Package

```
package QddgDBSamples_Dsgn;

{$R *.res}
{$ALIGN 8}
{$ASSERTIONS OFF}
{$BOOLEVAL OFF}
{$DEBUGINFO OFF}
```

LISTING 13.10 Continued

```
{$EXTENDEDSYNTAX ON}
{$IMPORTEDDATA ON}
{$IOCHECKS ON}
{$LOCALSYMBOLS OFF}
{$LONGSTRINGS ON}
{$OPENSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$REFERENCEINFO OFF}
{$SAFEDIVIDE OFF}
{$STACKFRAMES OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST ON}
{$MINENUMSIZE 1}
{$IMAGEBASE $400000}
{$DESCRIPTION 'DDG: CLX Components (Data-Aware)'}

{$IFDEF MSWINDOWS}
{$LIBSUFFIX '60'}
{$ENDIF}

{$IFDEF LINUX}
{$SOPREFIX 'bpl'}
{$SOVERSION '6'}
{$ENDIF}

{$DESIGNONLY}
{$IMPLICITBUILD OFF}

requires
  {$IFDEF LINUX}
  baseclx,
  {$ENDIF}
  visualclx,
  QddgSamples_Dsgn,
  QddgDBSamples;

contains
  QddgDBSamples_Reg in 'QddgDBSamples_Reg.pas';

end.
```

Registration Units

As you can see from the source listings for the design packages, registration units are used to handle registering all components. As is customary when creating VCL components, these registration units (`QddgSamples_Reg` and `QddgDBSamples_Reg`) are only contained within a design package. Listing 13.11 shows the source code for the `QddgSamples_Reg.pas` unit, which is responsible for registering the nondata-aware components and the `TddgRadioGroupEditor` component editor.

LISTING 13.11 `QddgSamples_Reg.pas`—Registration Unit for Nondata-Aware CLX Sample Components

```
{==================================================================
  QddgSamples_Reg Unit

  Registration Unit for all non-data-aware DDG-CLX components.

  Copyright © 2001 by Ray Konopka
==================================================================}

unit QddgSamples_Reg;

interface

procedure Register;

implementation

uses
  Classes, DesignIntf, DesignEditors, QExtCtrls,
  QddgSpin, QddgDsnSpn, QddgILSpin,
  QddgRgpEdt;


{=======================}
{== Register Procedure ==}
{=======================}

procedure Register;
begin
  {== Register Components ==}

  RegisterComponents( 'DDG-CLX',
                      [ TddgSpinner,
                         TddgDesignSpinner,
                        TddgImgListSpinner ] );
```

LISTING 13.11 Continued

```
  {== Register Component Editors ==}

  RegisterComponentEditor( TRadioGroup, TddgRadioGroupEditor );
end;

end.
```

Component Bitmaps

In order to identify your newly created custom CLX component on the Component Palette, you should create a component bitmap, which is a 16-color bitmap that is 24x24 pixels in size. The online help for Kylix and Delphi suggest that you create a separate resource file for each component unit.

However, the Package Editor searches for a matching `.dcr` file whenever you add a unit to a package. Unfortunately, the Package Editor does this for both runtime and design packages, and linking palette bitmaps into a runtime package is pointless because the bitmaps will go unused and simply waste space.

Therefore, instead of creating separate `.dcr` files for each component unit, simply create a single `.dcr` file containing all the component bitmaps. Fortunately, resources in Kylix are the same as those used in Delphi. That is, even though Kylix generates native Linux executables, the format used to attach resources is the Win32 resource format. As a result, we can use any resource editor that can create Windows `.res` files and then simply rename the file with a `.dcr` extension. For example, Figure 13.9 shows the `QddgSamples_Reg.dcr` file being edited in the Image Editor.

Notice that the name of the resource file is the same as the registration unit. As a result, when the registration unit is added to the design package, the component resource file is also added. Furthermore, because the registration unit isn't used in the runtime packages, the component bitmaps won't be linked into them.

As a final comment, don't underestimate the importance of well-designed bitmaps to represent your components. The component bitmaps are the first impression users will have of your components. Unprofessional bitmaps give the impression of unprofessional components. If you build components for the commercial market, you might want to get a professional graphics artist to design the bitmaps.

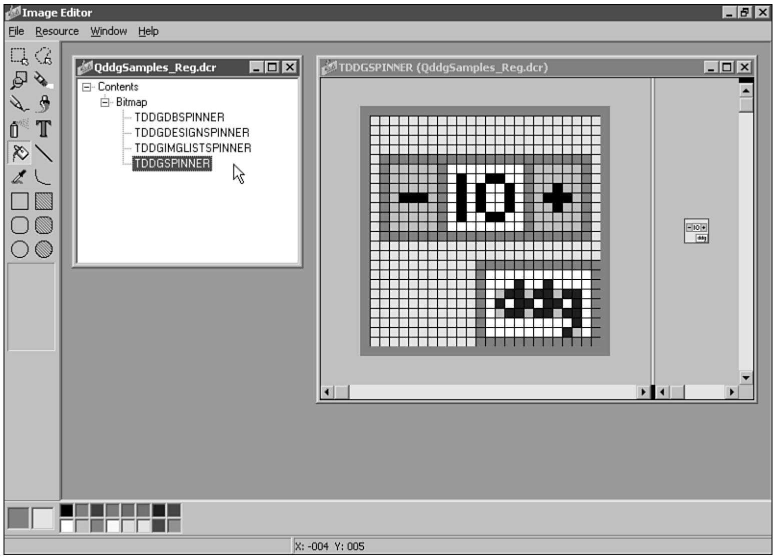


FIGURE 13.9
The Image Editor can be used to create DCR files for CLX components.

Summary

You can do several things in your current VCL-based components to aid in porting them to CLX in the future. First, use existing VCL wrappers wherever possible. For example, use the `TCanvas` methods instead of calling GDI functions directly. Override existing event dispatch methods such as `MouseDown()` instead of handling the `wm_LButtonDown` window message. Linux doesn't use messages; therefore, the `wm_LButtonDown` message doesn't even exist under Linux. Another helpful technique is to create your own abstraction classes to help isolate platform dependent code.

Although CLX was modeled after the VCL, migrating your existing VCL components to CLX will definitely require some effort. Platform specific calls such as calls to the Win32 API or to libc must be eliminated or at least wrapped within platform conditional compilation directives. However, it is indeed possible to create a custom CLX component using a single source file that will operate under both Delphi/Windows and Kylix/Linux.