

DRAFT

Input regarding this document appreciated - submission information available at:
<http://www.borland.com/events/cdrom.html>.

The Visual Component Library in C++Builder

Introduction

This paper discusses the Visual Component Library (VCL)—the class library from which C++Builder applications are built. It discusses the VCL hierarchy and explains the purpose of the key levels within the hierarchy. It also discusses the purposes of the common properties, methods and events that appear at the different component levels in the VCL hierarchy.

This paper does not discuss individual components and classes that make up the VCL. Rather, the goal of the paper is to further the reader's understanding of how various components and classes are built. This paper assumes that you have a general working knowledge of C++Builder's component usage.

The Visual Component Library is actually a class library which consists of classes and components that you use to create both C++Builder and Delphi applications. This VCL is designed so that you can manipulate these classes within the visual environments of C++Builder and Delphi, *at design-time*, while you create your application, *as well as at run-time*. This differs from many other development environments, where the behavioral and visual characteristics of your applications are handled only *at run-time*. In C++Builder, you modify the behavioral and visual characteristics of your components as you develop your application visually—or modify component behavior and appearance at run-time.

You should have a good working knowledge of the VCL. The depth of knowledge that you require depends on how you intend on using the VCL. Therefore, you should realize that there are two types of C++Builder developers: Applications Developers and Component Writers.

Two Audiences: Applications Developers, Component Writers

The VCL for Applications Developers

Applications Developers create complete applications by interacting with the C++Builder visual environment (as mentioned earlier, this is a concept nonexistent in many other class libraries). These people use the VCL to create their user-interface and the other elements of their application: database connectivity, data validation, business rules, etc.

Applications Developers should know which properties, events, and methods each component makes available. Additionally, by understanding the VCL architecture, Applications Developers will be able to easily identify where they can improve their applications by extending components or creating new ones. Then they can maximize the capabilities of these components, and create better applications.

VCL vs OWL

There are several differences between the VCL and frameworks such as OWL. Many of these differences are presented in this paper. The most important difference merits further discussion now, however. The

VCL, like OWL, is a fully object oriented class library which encapsulates many complex tasks. However, the VCL is a powerful system designed to give the applications developer the ability to create robust applications rapidly, without the need to understand object oriented design concepts, or even the VCL hierarchy. In fact, the VCL streamlines applications development because it is designed to be manipulated in a visual environment. The applications developer can actually see the effects that changes made to components will have on their applications while they are designing the application, without having to write a line of code.

The VCL for Component Writers

Component Writers expand on the existing VCL, either by developing new components, or by increasing the functionality of existing ones. Many component writers make their components available for Applications Developers to use.

A Component Writer must understand the VCL in greater depth than the Application Developer. For example, Component Writers must know whether to write a new component or to extend an existing one when the need for a new characteristic arises. This requires more intimate knowledge of the VCL's inner workings. Although this paper can serve as an introduction to component writing, a component writer will have to refer to the C++Builder documentation (see the *Component Writer's Guide*) and third party references for details.

Empowering the Applications Developer (writing objects for design-time development)

One of the paradigm shifts that component writers should recognize is that they are creating objects (components) for a design-time environment and not objects which are solely restricted to the run-time development. What this means is that users of your component should be able to set various visual and behavioral attributes at design-time through the Object Inspector. Why is this so important? It greatly simplifies the development of many common programming tasks that applications developer would otherwise have to code. These tasks may relate to something visual or non-visual in the application. For example, playing an AVI file could be a pretty cumbersome task. In C++Builder, one simply drops a TMediaPlayer component onto the form, specifies an AVI file name for the FileName property and executes the line of code:

```
MediaPlayer1->Play();
```

Even non-visual tasks are greatly simplified. For example, in OWL, there is a TModule class which encapsulates the loading and unloading of DLL as well as other DLL related features. The same can be done in C++Builder. In C++Builder, however, the applications developer would be able to specify the DLL file name, and whether or not it is automatically loaded when the application runs without having to write a line of code. These are simple examples of how a visual development environment empowers applications developers by simplifying complex tasks thus allowing developers to rapidly create powerful applications.

Components Types, Structure, and Concepts

Components are the building blocks that developers use to design the user-interface and to provide some non-visual capabilities to their applications. To an Application Developer, a component is an object most commonly dragged from the Component palette and placed onto a form. Once on the form, one can manipulate the component's properties and add code to the component's various events to give the component a specific behavior. To a Component Writer, components are C++ classes or classes written in Delphi. Some components encapsulate the behavior of elements provided by the system, such as the standard Windows 95 controls. Other objects introduce entirely new visual or non-visual elements, in which case the component's code makes up the entire behavior of the component.

The complexity of different components varies widely. Some might be simple while others might encapsulate an elaborate task. There is no limit to what a component can do or be made up of. You can have a very simple component like a **TLabel**, or a much more complex component which encapsulates the complete functionality of a spreadsheet.

Whatever it is that components do, they are all derived from a base class defined by the VCL already shipping with C++Builder. The base class from which a component descends depends on the capabilities that it is supposed to have.

Component Types

Components are really just special types of classes. There are three fundamental keys to understanding the VCL.

- **First**, you should know the special characteristics of the four basic component types: standard controls, custom controls, graphical controls and non-visual components.
- **Second**, you must understand the VCL structure with which components are built.
 - **Third**, you should be familiar with the VCL hierarchy and you should also know where the four component types previously mentioned fit into the VCL hierarchy.

The following paragraphs will discuss each of these keys to understanding the VCL.

As a component writer, there are four primary types of components that you will work with in C++Builder: standard controls, custom controls, graphical controls, and non-visual components. Although these component types are primarily of interest to component writers, it's not a bad idea for applications developers to be familiar with them. They are the foundations on which applications are built.

Standard Components

Some of the components provided by C++Builder encapsulate the behavior of the standard Windows controls: **TButton**, **TListBox** and **TEdit**, for example. You will find these components on the *Standard* page of the Component Palette. These components are Windows' common controls with component wrappers around them.

Each standard component looks and works like the Windows' common control that it encapsulates. The VCL wrapper simply makes the control available to you in the form of a C++Builder component—it doesn't define the common control's appearance or functionality, but rather, surfaces the ability to modify a control's appearance/functionality in the form of methods and properties. It's important to understand that the way you would modify the behavior/appearance of a component is through C++Builder's Object Inspector, not by writing code. The shipping standard components in C++Builder were shared with Delphi. You can examine how they wrap standard Windows controls in the file `STDCTRLS.PAS`.

If you want to use these standard components unchanged, there is no need to understand how the VCL wraps them. If, however, you want to extend or change one of these components, then you must understand how the Windows' common control is wrapped by the VCL into a C++Builder component. For example, the Windows class `LISTBOX` can display the list box items in multiple columns. This capability, however, isn't surfaced by C++Builder's **TListBox** component (which encapsulates the Windows `LISTBOX` class). (**TListBox** only displays items in a single column.) Surfacing this capability requires that you override the default creation of the **TListBox** component.

This example also serves to illustrate why it is important for Applications Developers to understand the VCL. Just knowing this tidbit of information helps you to identify where enhancements to the existing library of components can help make your life easier and more productive.

Custom components

Unlike standard components, custom components are controls that don't already have a method for displaying themselves, nor do they have a defined behavior. The Component Writer must provide code that tells the component how to draw itself and determines how the component behaves when the user interacts with it. Examples of existing custom components are the **TPanel** and **TStringGrid** components. It should be mentioned here that both standard and custom components are *windowed* controls. A "windowed control" has a window associated with it and, therefore, has a window handle. Windowed controls have three characteristics: they can receive input focus, they use system resources, and they can be parents to other controls. (parenthood is related to containership, discussed later in this paper.) An example of a component which can be a container is the **TPanel** component.

Graphical components

Graphical components are visual controls which cannot receive input focus from the user. They are *non-windowed* controls. Graphical components allow you to display something to the user without using up any system resources; they have less "overhead" than standard or custom components. Graphical components don't require a window handle—thus, they cannot receive focus. Some examples of graphical components are the **TLabel** and **TShape** components.

Graphical components cannot be containers of other components. This means that they cannot own other components which are placed on top of them.

Non-visual components

Non-visual components are components that do not appear on the form as controls at run-time. These components allow you to encapsulate some functionality of an entity within an object and allow you to manipulate how the component will behave through the Object Inspector. Using the Object Inspector, you can modify a non-visual component's properties and provide event handlers for its events. Examples of such components are the **TOpenDialog**, **TTable**, and **TTimer** components.

Component Structure

All components share a similar structure. Each component consists of elements that allow developers to manipulate its appearance and behavior via properties, methods and events. The following sections in this paper will discuss these common elements as well as talk about a few other characteristics of components which don't apply to all components.

Properties

Properties provide an extension of an object's data members. Unlike data members, properties do not store data: they provide other capabilities. For example, properties may use methods to read or write data to an object's data member to which the user has no access. This adds a certain level of protection as to how a given data member is assigned data. Properties also cause "side effects" to occur when the user makes a particular assignment to the property. Thus what appears as a simple data member assignment to the component user could trigger a complex operation to occur behind the scenes. More importantly, properties provide the mechanism behind C++Builder's visual development. That is, properties are accessible for modification at run-time through C++Builder's Object Inspector. This occurs whenever the declaration of the property appears in the **__published** section of a component's declaration.

Properties hide access to internal storage data members

There are two ways that properties provide access to internal storage data members of components: directly or through access methods. Examine the code below which illustrates this process.

```
class TSampleComponent : public TComponent
{
private:
    int FSize;
    void __fastcall SetSize(int ASize);
__published:
    __property int Size = {read=FSize, write=SetSize};
};
```

The code above is a simple “do nothing” component class, **TSampleComponent**. **TSampleComponent** descends from **TComponent**. What this means is that the **TSampleComponent** class will inherit the properties and methods already defined by **TComponent**.

TSampleComponent has a private data member **FSize** of type **int**. If this were a real component, **FSize** could serve any number of purposes. For example, it could specify the maximum characters for an edit control, or even a to specify a certain amount of memory to allocate. The key here, is that the user doesn’t directly access the **FSize** data member . Instead, a value is added to this data member by making an assignment to the **Size** property. This illustrates one of the main benefits to properties. The component user can give a value to **FSize** through the **Size** property either at run-time as shown below:

```
SampleComponent->Size = 15;
```

Or at design-time through the Object Inspector.

This illustrates how the property **Size** provides the access to the storage data member **FSize**.

The definition of the **Size** property specifies how a value is read from and written to the storage data member **FSize**. Although it is beyond the scope of this article to discuss the implementation of properties in regards to component design, we’ll briefly explain how this works. The **read** declaration in the property definition specifies how the property is used to read the value of an internal storage data member. For instance, the **Size** property has direct read access to **FSize**. Therefore, the following line of code:

```
MyVal = SampleComponent->Size;
```

is essentially the same as if the value was gotten directly from the **FSize** data member. The **write** declaration for **Size** shows that assignments made to the **Size** property result in a call to an *access method* which is responsible for assigning a value to the **FSize** storage data member. This access method is **SetSize()**. This is probably the most important aspect of properties. Access methods allow the component writer to perform whatever behind the scenes tasks are necessary when properties are assigned values. Property access methods can involve very complex operations.

Property-access methods

Access methods take a single parameter of the same type as that of the property. **Write** access methods also provide a method layer over assignments made to a component’s data members. Instead of the component user making the assignment to the data member directly, the property’s **write** access method will assign the value to the storage data member if the property refers to a particular storage data member. For example, examine the implementation of the **SetSize()** method below.

```

void __fastcall TSampleComponent::SetSize(int ASize)
{
    if ((ASize > 0) && (ASize <= 10)) {
        FSize = ASize;
    }
    else
        MessageDlg("Error: value must be between 1-10.", mtError,
            TMsgDlgButtons() << mbOK, 0);
}

```

The code in the **SetSize** first does some error checking to ensure the size specified is with a certain range. If the value is with the range the method assigns the new value to the internal storage data member, **FSize**. Otherwise, it displays an error message to the user. You can see how properties allow the component writer to perform error checking on property assignments at both run-time and at design-time. Overall, this makes for more robust applications as far as the applications developer is concerned.

Providing access to internal storage data members through property access methods offers the advantage that the Component Writer can modify the implementation of a class without modifying the interface. It is also possible to have access methods for the read access of a property. The read access method might, for example, return a type which is different than that of a properties storage data member. For instance, it could return the string representation of an integer storage data member. Another primary reason for **write** access methods is to cause some side-effect to occur as a result of an assignment to a property.

Properties cause side effects to occur

Side-effects are basically any action that occur within the access method of a property, other than just assigning a value to internal storage data members. Although the error checking illustrated in the above code could be considered as a side-effect, other side-effects can be much more complex. Consider, for example, the Top, Left, Height, and Width properties of most components. When you assign values to these properties, not only do you assign a value to some internal storage data member, you also cause the component to reposition and repaint itself with the new sizes. All this is done behind the scenes without any coding required by the component user. There is no limit to what side effects can do nor how sophisticated or complex they may be.

Types of properties

Properties can be of the standard data types. Property types also determine how they are edited in C++Builder's Object Inspector. The table below shows the different property types as they are defined in C++Builder's online help.

Property type	Object Inspector treatment
Simple	Numeric, character, and string properties appear in the Object Inspector as numbers, characters, and strings, respectively. The user can type and edit the value of the property directly.
Enumerated	Properties of enumerated types (including Boolean) display the value as defined in the source code. The user can cycle through the possible values by double-clicking the value column. There is also a drop-down list that shows all possible values of the enumerated type.
Set	Properties of set types appear in the Object Inspector looking like a set. By expanding the set, the user can treat each element of the set as a Boolean value: True if the element is included in the set or False if it's not included.
Object	Properties that are themselves objects often have their own property editors. However, if the object that is a property also has published properties, the Object Inspector allows the user to expand the list of object properties and edit them individually. Object properties must descend from TPersistent .
Array	Array properties must have their own property editors. The Object Inspector has no

built-in support for editing array properties.
--

For more information on properties, refer to the “Component Writer’s Guide” which ships with C++Builder.

Methods

Since components are really just objects, they can have methods. We will discuss some of the more commonly used methods later in this paper when we discuss the different levels of the VCL hierarchy.

Events

Events provide a means for a component to notify the user of some pre-defined occurrence within the component. Such an occurrence might be a button click or the pressing of a key on a keyboard.

Components contain special properties called events to which the component user assigns code. This code will be executed whenever a certain event occurs. For instance, if you look at the events page of a **TEdit** component, you’ll see such events as **OnChange**, **OnClick** and **OnDbClick**. These events are nothing more than pointers to methods.

When the user of a component assigns code to one of those events, the user’s code is referred to as an *event handler*. For example, by double clicking on the events page for a particular event causes C++Builder to generate a method and places you in the Code Editor where you can add your code for that method. An example of this is shown in the code snippet below, which is an **OnClick** event for a **TButton** component.

```
class TForm1 : public TForm
{
__published:      // IDE-managed Components
    TButton *Button1;
    void __fastcall Button1Click(TObject *Sender);
...
};

//-----
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    // Event code goes here.
}
```

What C++Builder does here is creates the method **Button1Click()**. Then, it assigns this method to the Button1’s **OnClick** event property. Now, when the application is running and the user clicks on the button, code executes behind the scenes that checks to see that **Button1.OnClick** actually refers to a method. If so, it executes that methods. Whatever code you place between the {} symbols will be executed every time the user clicks on the button.

Component Concepts

Containership

Some components in the VCL can own other components as well as be parents to other components. These two concepts have a different meaning as will be discussed in the following sections.

Ownership

All components may be owned by other components but not all components can own other components. A component's **Owner** property contains a reference to the component which owns it. The basic responsibility of the owner is one of resource management. The owner is responsible for freeing those components which it owns whenever it is destroyed. Typically, the form owns all components which appear on it, even if those components are placed on another component such as a **TPanel**. At design-time, the form automatically becomes the owner for components which you place on it. At run-time, when you create a component, you pass the owner as a parameter to the component's constructor. For instance, the code below shows how to create a **TButton** component at run-time and passes the form's implicit **this** variable to the **TButton's** constructor. The value of **this** will be assigned to the **TButton's Owner** property.

```
MyButton = new TButton(this);
```

When the form that now owns the **TButton** component gets freed, **MyButton** will also be freed. You can create a component without an owner by passing **nil** to the component's constructor, however, you must ensure that the component is freed when it is no longer needed. The code below shows you how to do this for a **TTable** component.

```
MyTable = new TTable(nil);  
/* Do stuff with MyTable */  
delete MyTable;
```

The **Components** property of a component is an array property which contains a list of the components which it owns. For instance, the code below shows how to loop through a form's components and then shows their class name.

```
for (int i = 0; i < ComponentCount; i++) {  
    ShowMessage(Components[i]->ClassName());  
}
```

Parenthood

Parenthood is a much different concept from ownership. It applies only to windowed components, which can be parents to other components. later, when we discuss the VCL hierarchy, you will see the level in the hierarchy which introduces windowed controls.

Parent components are responsible for the display of other components. They call the appropriate methods internally that cause the children components to draw themselves. The **Parent** property of a component refers to the component which is its parent. Also, a component's parent does not have to be its owner. Although the parent component is mainly responsible for the display of components, it also frees children components when it is destroyed.

Windowed components are controls which are visible user interface elements such as edit controls, list boxes and memo controls. In order for a windowed component to be displayed, it must be assigned a parent on which to display itself. This task is done automatically by C++Builder's design-time environment when you drop a component from the Component Palette onto your form. When creating a component at run-time, however, you must explicitly make this assignment, otherwise the component will not be displayed. An example of creating a **TEdit** component at run-time is shown below:


```
TEdit *myEdit;  
  
myEdit = new TEdit(this);  
myEdit->Parent = this;
```

Streamability

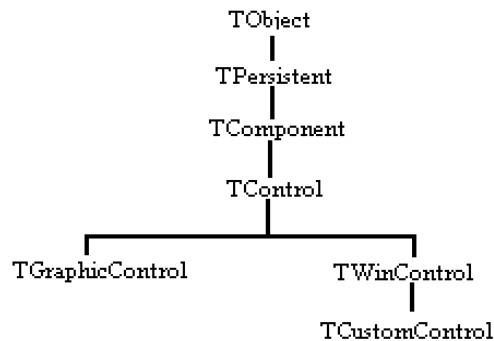
Another characteristic of a component is that it can be streamed. Streaming is a way to store a component and information regarding its properties' values to a file or to an area in memory. For example, the .DFM file created by C++Builder, is a resource file with information about a form and the components residing on the form. This information was streamed to that resource file.

Component Writers must understand the streaming mechanism of the VCL if they intend for their components to stream special data, which is not done automatically by the VCL.

The VCL Hierarchy

Overview

The figure below illustrates a sub-set of C++Builder's VCL hierarchy. The objects shown in the hierarchy the key classes from which components descend. Each object introduces a certain set of methods, events and properties and has a special purpose.



Earlier in this paper we discussed four types of components: standard controls, custom controls, graphical controls and non-visual components. The following paragraphs discusses how these different types of components relate to the objects shown in the hierarchy.

Non-visual components are descendants of **TComponent**. Whereas **TObject** is the base class from which all classes descend, **TComponent** is the base class from which all components descend. This paper will discuss the **TComponent** class in more detail shortly.

The **TGraphicControl** class provides the capability to have controls which are not windowed controls (they have no window handle). Therefore graphical controls descend from **TGraphicControl**.

TWinControl is the base class from which all windowed controls descend. It is at the **TWinControl** level that the window handle is introduced. Both standard controls and custom controls which are windowed and are therefore descendants of **TWinControl**.

Custom controls, however, are not likely to descend directly from **TWinControl** and will descend from **TCustomControl**.

TObject

TObject is the base class from which all other classes descend. Since all classes descend from **TObject**, every class inherits the methods which are defined by **TObject**. This gives all classes certain functionality. For example, every class can tell you its name, its type, and even its ancestry. You'll find **TObject** defined in the SYSDEFS.H header file.

Many of **TObject's** methods are static methods. This means that these methods can be called from a class type and don't require a class instance. Most of **TObjects** methods are used internally by the VCL.

All components must descend from **TComponent** or from a **TComponent** descendant. **TComponent**, being a descendant of **TObject**, inherits **TObjects** methods, data members and properties.

Objects which descend from objects higher than **TComponent** in the VCL hierarchy are non-component classes. Some useful non-component classes are **TStringList**, **TIniFile** and **TPrinter**. You can look up these classes in the online help if you're unfamiliar with them.

TPersistent

The **TPersistent** class descends directly from **TObject**. The special characteristic of **TPersistent** is that it is an abstract class that defines the methods that allow it to be streamed. **TPersistent** defines no special properties or events, but does define certain methods of use to the Component Writer. The table below shows these methods.

Method	Purpose
Assign	This public method allows a component to assign the data associated with another component to itself.
AssignTo	This protected method allows a component to override the implementation of the Assign method. TPersistent itself, raises an exception when this method is called. It is up to TPersistent descendant to override this method to define its implementation. The TClipboard class is an object that does this, for example.
DefineProperties	This protected method that allows component writers to define how the component stores extra or unpublished properties. By default, a component automatically stores published properties.

TComponent

The **TComponent** class is a direct descendant of **TPersistent**. As we said earlier, all components are **TComponent** descendants. **TComponent's** special characteristics are that its properties are streamable and can be manipulated at design-time through the Object Inspector. **TComponent** can also own other components.

Certain non-visual components that descend from **TComponent** are also capable of being manipulated at design time. One such is the **TTimer** component. **TTimers** are not visual controls but still are available on the Component palette.

TControl

TControl defines properties, methods, and events common to visual components. **TControl** for example has the capability to display itself. Therefore, some of its properties have to do with its size and position: **Top**, **Left**, **Width** and **Height**. Other properties are **ClientRect**, **ClientWidth** **ClientHeight**.

TControl also introduces various properties having to do with its appearance and accessibility, such as: **Visible**, **Enabled** and **Color**. The **Font** property lets you specify a particular font for the control. Also you can set the text for the control through **TControl's** **Text** and **Caption** properties.

TControl contains several mouse and drag-drop events required of visual controls. Such events are: **OnClick**, **OnDblClick**, **OnMouseDown**, **OnMouseMove**, **OnMouseUp**, **OnDragOver**, **OnDragDrop**, and **OnEndDrag**. One interesting note about these events is that they are declared in the **protected** section of **TControl**. This is because **TControl** is most likely to be descended from rather than being used directly. Declaring **TControl's** properties and events in the **protected** section allows writers of descendant components to determine which properties and/or events to make public or published.

Another important characteristic of **TControl** is that it may have a parent. This parent must be a descendant of **TWinControl** since parent controls must be *windowed* controls. Since **TControl** introduces the concept of having a parent it introduces the **Parent** property which refers to its parent. Most of the C++Builder controls are descendants of either **TWinControl** or **TGraphicControl** both of which are discussed next.

TWinControl

The **TWinControl** class encapsulates a window—controls with a window handle. Certain descendants of **TWinControl** such as **TEdit**, **TListBox** and **TComboBox** encapsulate the standard Windows controls such as edit controls, list boxes, combo boxes respectively. Since these descendant components encapsulate the functionality of standard controls, you don't have to manipulate them through Win32 API functions. Instead you manipulate them through properties and methods provided by the various control components.

TWinControls have three basic characteristics: they have a window handle, they receive input focus, and they can be parents to other controls. Therefore, many of **TWinControl's** properties, methods and events have something to do with focus changing, keyboard events and the displaying of child controls. Methods of interest to component writers have something to do with window creation, focus control, message dispatching, and positioning.

TGraphicControl

TGraphicControls differ from **TWinControls** in that they do not have a window handle and cannot receive input focus. Also, they cannot be parents to other controls.

TGraphicControls are used when you want to display something on the form, without the functionality(ies) of a regular windowed controls. Two key advantages to this strategy are:

- **TGraphicControls** don't use up system resources since they don't require a window handle.
 - They are a bit faster at drawing than their **TWinControl** counterparts since their painting is not subject to the Windows message dispatching system. Instead, they piggyback on their parent's paint process.

TGraphicControls can respond to mouse events and therefore have mouse event handlers. Since **TGraphicControl** allows you to paint the itself, it contains a **TCanvas** property, **Canvas**, and a **Paint** method which **TGraphicControl** descendants must override.

TCustomControl

The standard controls which descend from **TWinControl** like **TEdit** and **TListbox** already have default drawing capabilities, provided by the encapsulated Windows control. What if you want to create a windowed component that provides its own custom painting? This is the purpose of **TCustomControl**.

TCustomControl is a descendant of **TWinControl** and is therefore a windowed control. This means that **TCustomControl** can also get input focus. Component writers create components that can descend from **TCustomControl**. Like the **TGraphicControl**, **TCustomControl** surfaces its **Canvas** property, which allows for custom drawing to its canvas. In fact, **TCustomControl's** provides a virtual **Paint** method which you override to perform your custom drawing.

Conclusion

Whether you plan use the Visual Component Library as an Applications Developer, or expand the existing library as a Component Writer, your understanding of the VCL will only make it easier for you to successfully accomplish either task. The VCL may seem very complex at first. However, by having a basic understanding of its hierarchy and by knowing the role played by key objects within the VCL, you will be able to effectively maximize your use of the VCL to create better applications and more powerful components.

Copyright © 1996. Borland International, Inc.