Creating Commercial Components

(Borland VCL Framework)

•

Technical White Paper

View Contents

Date: September 15, 2000

Authors: Ray Konopka, Raize Software
Andrew Pharoah, ComponentSource
Chris Brooke, ComponentSource

www.componentsource.com

Email: publishers@componentsource.com

US Headquarters

ComponentSource
3391 Town Point Drive,
Suite 350,
Kennesaw, GA 30144-7083
USA

Tel: (770) 250 6100
Fax: (770) 250 6199
International: +1 (770) 250 6100

European Headquarters

ComponentSource
30 Greyfriars Road,
Reading,
Berkshire RG1 1PE
United Kingdom

Tel: 0118 958 1111
Fax: 0118 958 1111
International: +44 118 958 1111

Contents

Introduction

This white paper has been constructed to help component authors develop and enhance professional software components for delivery on the 'open market'. Information covered in the document is based on the knowledge and expertise of those component authors who have successfully established themselves in the component marketplace. The content is aimed at developers who wish to create components based on the Borland Visual Component Library (VCL) framework. In the following section, the business benefits of using components are discussed, and the functionality suitable for component development is identified.

Commercial Overview

The market for Software Components is expected to grow to around $4.4 billion by 2002, $1.0 billion from products and $3.4 billion from related services. (Source: PricewaterhouseCoopers)

Software applications are now created as a collection of software components. For example: Microsoft ® Office 2000. Increasingly application developers are employing component-based software development techniques, which enable them to reduce their time to market and improve their software quality. Software authors who are experts in a specific horizontal or vertical market sector are now "componentizing" their applications to meet the increasing demand for sophisticated business components. As such this represents a huge opportunity for you to unlock hidden revenues from years of research and development.

Why is buying a software component a good idea?

Everybody, software developers included, admit that they do something, (write a program or subroutine), better second time around. This is the essence of a "component", built and continuously improved by an expert or organization and encapsulating business logic or technical functionality. By buying a component a developer can add functionality to their application without sacrificing quality. Indeed quality should improve, as the component will have gone through several development iterations and include feedback from 1,000's of users.

What types of components will people buy?

Initially software components were used to provide technical functionality, such as SMTP for email or enhanced user interfaces. Developers are now requesting sophisticated components that solve real business issues from component authors.

What is helping make this happen now?

Inprise Corporation (formerly, Borland) created the Visual Component Library as the foundation for its Borland Delphi and Borland C++Builder RAD development tools. The success of these tools and the richness of the VCL's object-oriented architecture encourage the development of reusable components, which extend the VCL hierarchy.

The Visual Component Library is much more than a specification, it is an extensive class library with functionality that can be immediately incorporated into your own components through inheritance.

To find out how to create components from your existing applications or as part of your next software development project - read the remainder of this white paper. Many varied technical topics are covered and this paper provides a "best practice" guide to commercial component creation in a Borland VCL based environment.

Component Overview

Identifying A Component Candidate

How do I identify a component candidate? - Understanding how a component works and how functionality differs from applications is important when identifying a suitable component candidate. In this section we investigate existing applications for potential functionality, consider component reusability and finally discuss the importance of business knowledge and how this applies to the components you write.

a) Analyze Application Functionality

The key factor is being able to recognize functionality that is essentially self-sufficient. In other words, is the functionality loosely coupled to other parts of the application? Tightly coupled functionality is very difficult to componentize. Developers should look at the functions encapsulated in their own applications and others to assess the commercial viability of componentizing particular functions. Another characteristic of a component is that the business logic is separate from the data. However, this does not apply to single parameter data that is passed to a methods interface. If you are creating a component it's important to manipulate data in arbitrary collections or streams.

b) Component Reusability

An important factor worth considering is a products commercial viability. Market demand determines whether a component is commercially viable or should be used only within your own organization. Typical examples include components that are directly linked to hardware such as monitoring components for alarm systems. Unless the components can be sold separately from the hardware the ability to sell the product online is greatly reduced.

Components that can be integrated without any consultation will succeed in what's known as the 'Open Market'. This market allows components to be distributed without any consultation or tailoring service. All information regarding the product is supplied in online documentation such as demonstrations, evaluations, help files and sample code.

For more information on the open market browse to:
http://www.componentsource.com/services/cbdiopen_market.asp

c) Expert Functionality

Expertise and knowledge are the two areas you should focus on when writing a software component. If you are developing a component from scratch then consider the components already on the market and assess whether you could offer a different or superior solution. Where possible write components that are related to your core business area. It's likely that these functions will be more valuable than peripheral functionality designed to provide a basic solution. For example, if your core business provides insurance underwriting services then concentrate on these core functions first as opposed to peripheral components such as a basic user interface components for data presentation in a grid or as a chart of graph.

Designing Commercial VCL Components

This section presents an overview of several of the design decisions that need to be made after a candidate has been selected.

Component Types

There are two basic types of components that you can create-visual and nonvisual. Both types can encapsulate either technical or business knowledge. The differences between the two depend on the functionality that is implemented. For example if the component provides only a benefit to the developer e.g. a TCP/IP communication library then the component is categorized as technical. Business components, on the other hand, provide a benefit to both the developer and the end-user by encapsulating business knowledge. Typical examples include address formatting and credit card validation components. Both visual and nonvisual components have their benefits and in the following sections we will look at different examples of both. In addition, we'll also discuss the differences between client-side and server-side components.

a) Visual Components

Visual components present a pre-designed presentation interface to the user. Examples include data grids and charting components. In the Delphi and C++Builder arena, these components are implemented as descendant classes of the Visual Component Library (VCL). (More on this will be covered in the Developing VCL Components section.)

Visual VCL-based components can also be installed into the Component Palette, which is part of the Delphi and C++Builder Integrated Development Environments (IDE). The developer is then able to "drop" the visual component onto a form and manipulate the component's properties through the IDE's Object Inspector. In addition to manipulating properties, developers are also able to respond to events that occur within the component. The critical element here is that customizations to the visual and functional interfaces of the component can be made at design-time.

b) Non-Visual Components

In the VCL, visual and nonvisual components are both extensions to the VCL hierarchy. That is, both component types are implemented by creating descendants of VCL base classes. The direct ancestor of a component determines whether the component is visual or nonvisual. Specifically, a component that descends from TComponent directly is a nonvisual component. Most nonvisual components do not provide any graphical user interface to the user-they are simply wrappers around reusable functionality. An example of a nonvisual components is a Timer component that generates an event whenever the timer fires.

Like visual components, a nonvisual component can also be installed into the Component Palette and once this is done, the component can be dropped onto a form. Being a nonvisual component, the component appears as an icon on the form. The icon is not visible at runtime, but at design-time the icon provides access to the component's properties and events.

b) Client-Side versus Server-Side

Unlike other component architectures, there is very little distinction between client-side and server-side VCL-based components. However, this does not mean that all VCL-based components can be used in a server application. Indeed, it does not make much sense to use a visual component with a graphical user interface on a server that is locked away in a separate room-no one will be there to see the component. As a result, server-based applications are typically created using nonvisual components.

Also note that this does not mean that nonvisual components can only be used in a server environment. On the contrary, nonvisual components are quite useful in GUI applications. The point is that visual components are typically used only in client-side application, while nonvisual components are used in both client-side and server-side applications.

Component Languages

Native-VCL components are created using either Borland Delphi or Borland C++Builder. Delphi uses the Object Pascal programming language, while C++Builder uses C++. It should be noted that the base VCL hierarchy is written in Object Pascal and is shared between both Delphi and C++Builder. Although native VCL components can be written in C++Builder, a C++ VCL component can only be used in C++Builder. However, a Delphi VCL component can be used in both Delphi and C++Builder. As a result, it is recommended that commercial VCL components be written in Delphi (Object Pascal) because of the much larger Delphi market.

Developing Commercial VCL Components

In this section, we will take a closer look at some of the details involved in developing custom VCL components.

The Component Kingdom

When you drop a component onto a form, the Delphi Form Designer creates an instance of the selected component. Actually, the Form Designer simply creates a new object from the corresponding component class. Components, however, are not defined by just any class declaration. All components ultimately trace their ancestry back to the TComponent class.

The TComponent class provides the basic functionality required for a component to behave like a component. For example, TComponent implements the Name and Tag properties that are inherited by all components. Furthermore, TComponent provides the methods and properties that allow components to be manipulated by the Form Designer. In other words, for a component to be manipulated at design-time, it must descend from TComponent.

Although all component classes ultimately descend from TComponent, you will rarely create a custom component that is a direct descendant of TComponent. This is because this base component class does not provide any visual representation. It is much more common to create new components that descend from an existing class somewhere further down the class hierarchy. However, the actual class from which you chose to inherit will depend on the type of component you wish to create.

The easiest way to create a custom component is to inherit from an existing component that comes close to being what you want, and alter its behavior. For example, to create a specialized edit field, don't create the entire component from scratch. Instead, define your new component to be a descendant of the TEdit class and make the necessary modifications.

Slightly more difficult to implement are graphic controls. Graphic components descend from TGraphicControl and have a visual representation, but do not receive the input focus. An example of a graphic component is TLabel. Since graphic controls cannot be focused, they do not need a window handle. Therefore, graphic components are system resource friendly. In fact, Windows does not even know about these types of components, because graphic controls do not supply a window class to Windows.

What if you want to build a component that needs the input focus? In this case, there are two predefined classes from which you can inherit: TWinControl and TCustomControl. TWinControl is used when creating a Delphi component based on an existing Windows control. For example, the standard edit field component, TEdit, is a descendant of TWinControl. To create a truly custom control, the TCustomControl class is used. Since TCustomControl is a direct descendant of TWinControl, it automatically inherits all of the necessary functionality. It differs from TWinControl by providing a Canvas property, which is used to draw the custom component. TWinControl does not provide a Canvas property because an existing Windows control should already know how to paint itself.

All of the component classes described above refer to visual components. However, not all components need to be visual. The Timer component is a common example of a nonvisual component. Creating a nonvisual component is one of those rare cases where the new component type descends directly from TComponent.

Table 1 summarizes the available parent classes and when to use each one as an ancestor for a new component class.

Inherit from...	**To create...**
an existing component	a modified version of a working component.
TGraphicControl	a graphical component that does not require input focus.
TWinControl	a component that requires a window handle or to create a wrapper for an existing windows custom control.
TCustomControl	an original component.
TComponent	a nonvisual component.

Table 1: VCL base classes for building components

A Component Skeleton

Since all components are descendants of TComponent, they all share a similar structure. The most obvious aspect of that structure is that all components are defined using a class declaration. The following skeleton class represents the basic elements of a Delphi component:

```
type
  TSkeletonComponent = class(TComponent)
```

```
    private
      FNewProp : TPropType; // Internal data holds property value
      FRunTimeProp : TPropType; // Internal data holds runtime prop value
      FOnNewEvent : TEventType; // Internal method pointer supports event
      function GetNewProp : TPropType; // Access Methods
      procedure SetNewProp( Value : TPropType );
      procedure SetRunTimeProp( Value : TPropType );
    public
      constructor Create( AOwner : TComponent ); override;
      destructor Destroy; override;
      procedure NewMethod; // New method implements behavior
      procedure NewEvent; // Event dispatch method for OnNewEvent
      property RunTimeProp : TPropType // Declare new runtime property
        read FRunTimeProp
        write SetRunTimeProp;
    published
      property NewProp : TPropType // Declare new design-time property
        read GetNewProp
        write SetNewProp;
      property Height; // Redeclare inherited property
      property Width;
      property OnNewEvent : TEventType // Declare new published event
        read FOnNewEvent
        write FOnNewEvent;
      property OnClick; // Redeclare inherited event
end;
```

Listing 1: A Component Skeleton

No matter how many features are utilized, a component is always described by its properties, methods, and events.

Properties

This section focuses on how properties are used in a component. Properties are the single most important aspect of any component, and their importance cannot be overstated. Properties define the primary interface through which users manipulate the component. Well-designed properties make components more usable.

Properties represent the most significant aspect of component design because of their ability to be modified at design-time using the Object Inspector. This is significant because it lets users customize components before executing the application. Since properties provide the primary interface to a component, generally components will have many more properties than methods. From the component user's point of view, it is better to err on the side of too many properties than too few.

Properties can be used to hide implementation details from the user as well as to create side effects. Side effects are quite common in component building. Any property that contributes to the visual representation of a component will most surely rely on a side effect whenever the value of the property is changed. For instance, changing the Height property of a TPanel not only updates the internal storage holding the Height value, but also causes the component to repaint itself. Repainting a component is a typical side effect.

Methods

Although properties provide the primary interface between the component and the user, they capture only the attributes of the component. Properties do not represent behaviors. For example, several components have the ability to cut, copy, and paste text to and from the clipboard. These actions are behaviors and are not invoked through properties. Instead, methods are created for each type of behavior.

a) Minimize Method Interdependencies

Note well: Component users will use your components in ways you never dreamed. Given this axiom of component building, you must design your components in such a way that you minimize the preconditions imposed on the component user. Here are a few things to avoid when creating new methods.

First, a component should always be created in a valid state. Do not force the user to execute a method before the component can be used. (If you do this, you won't have to worry about your component being used in an invalid state-it just won't be used!) Second, never impose an order in which methods must be

called. And finally, avoid creating methods that cause other methods or properties to become invalid.

This last guideline can be a little tricky to handle. There will certainly be times when a method will change the state of the component, which in turn may invalidate some other method or property. In circumstances such as these, the best approach is to correct the errors in the method before executing the method's main block of code. Of course, this assumes the error is correctable. If not, it might be a prime spot to raise an exception.

b) Exposing Behaviors

Methods can reside in either the public, protected, or private sections of a component declaration. Deciding on where a particular method should be located depends on how it will be used. Methods that are to be accessible to the user at runtime must be placed in the public section. Implementation-specific methods, however, should at least reside in the protected section. This prevents users of your component from accidentally executing one of the implementation-specific methods. Methods that should be accessible from descendant classes should also be located in the protected section.

Access methods for properties should also be placed in the protected section and be made virtual. This allows descendant classes the ability to override the side-effect behavior of a property without having to redefine the property.

Events

Both properties and methods have a direct correlation with elements in the object model. Events, on the other hand, are implemented using a combination of object model features. From the component user's perspective, events are indications that a system event has occurred. For example, a key was pressed or the mouse pointer was moved. The user responds to these events by writing event handlers. Event handlers are typically methods located in the form containing the component that generated the event.

For example, the Button component has an event called OnClick. Whenever the user clicks the mouse on the button component, an OnClick event is generated. The user can respond to the click event by creating an event handler using the Object Inspector. The event handler is created as a method in the form containing the button. Inside this method, the user writes code to perform the necessary actions whenever the button is pressed. In essence, the user has modified the behavior of the button component. This is an important point. Events allow users to customize the behavior of components without having to create new ones.

The component developer (rather than the component user) has quite a different view of events. From this perspective, events are hooks into the component's normal processing. As described above, these hooks are used by component users to link custom code to a particular component.

Events are implemented using method pointers, which provide a way for one object to execute a method defined in another object. Method pointers provide the linking capability necessary to support event handling. However, method pointers are not manipulated directly. Instead, properties are defined for each event. By creating event properties, the Object Inspector can be used to connect custom code to individual components at design time.

The VCL Hierarchy

For the application developer, the Visual Component Library (VCL) is a framework consisting of a set of components that are used to construct applications. For the component writer, the VCL represents an extensible class hierarchy containing a vast amount of functionality that can be incorporated, through the mechanism of inheritance, into custom components. The VCL is extensible in that new components become part of this hierarchy and thus can serve as ancestors for still other components in the future.

The term Visual Component Library is a bit misleading because not all components are visual. The VCL includes non-visual components as well. The confusion is further compounded in that many classes that are not even components are considered to be part of the VCL.

Figure 1 displays the base classes that form the structure of the VCL. At the top is TObject, which is the ultimate ancestor for all classes in Object Pascal. Descending from it is TPersistent, which provides the necessary methods to create streamable objects. A streamable object is one that can, oddly enough, be

stored on a stream. A stream is an object that encapsulates a storage medium to store binary data (for example, memory or disk files). Because Delphi implements form files using streams, TComponent descends from TPersistent, giving all components the ability to be saved in a form file.

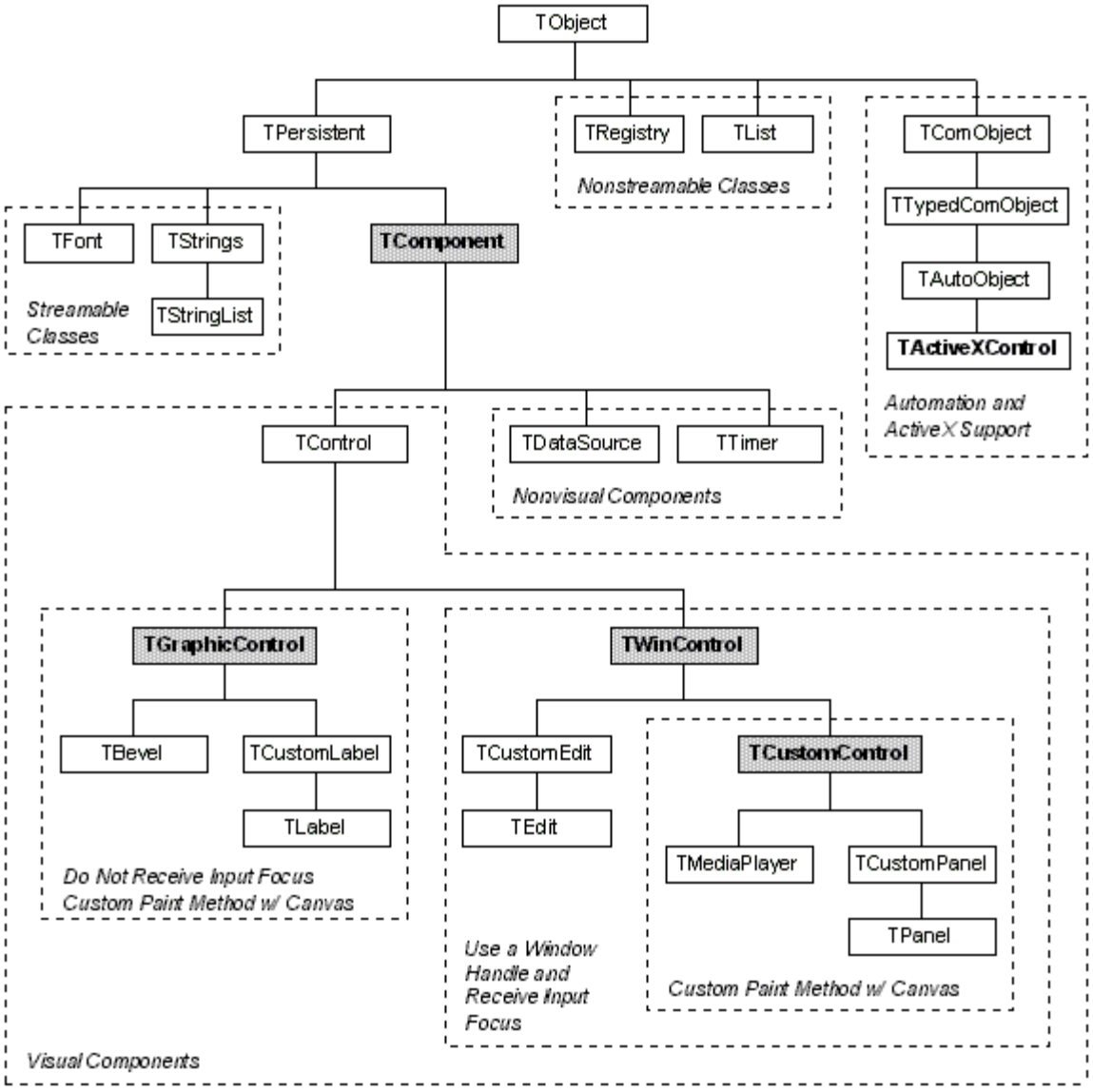

Figure 1: The base class structure of the Visual Component Library.

The TComponent class is essentially the top of the component hierarchy, and is the first of the four base classes used to create new components. Direct TComponent descendants are non-visual components. The TControl class implements the necessary code for visual components. Notice in Figure 1 that there are two basic types of visual controls: graphic controls and windowed controls-each one is represented by its own hierarchy descending from TGraphicControl or TWinControl, respectively. The main difference between these types of components is that graphic controls do not maintain a window handle, and thus cannot receive the input focus.

Windowed components are further broken up into two categories. Direct descendants of TWinControl are typically wrappers around existing controls that are implemented within Windows and, therefore, already know how to paint themselves. The standard Windows controls, like the edit field, are direct descendants of TWinControl. For components that require a window handle but do not encapsulate an underlying Windows control that provides visualization (that is, the ability to repaint itself), the TCustomControl class is provided.

The TActiveXControl class was introduced in Delphi 3.0 and allows developers to create ActiveX controls. Technically, ActiveX controls are not part of the Visual Component Library, as illustrated by the

position of TActiveXControl in Figure 6. However, TActiveXControl descendants typically contain a native component reference. The component reference provides the control's functionality, while TActiveXControl provides the COM interface so that the component can function as an ActiveX control.

The four shaded classes back in Figure 1 represent the common base classes from which you will derive new custom components.

The Building Process

All of the material covered thus far is important to component building. Consider the previous material as tools of the trade. The remainder of this section will focus on the process of constructing a new component. Although the example component is simple, it provides an excellent means of demonstrating the steps involved in the process.

Regardless of complexity, the process described here can be used when developing any component. Figure 2 provides a graphical view of the steps involved in building a custom component. The process begins with some initial setup, namely the creation of a directory to hold the component unit and a test application. Components are built inside Delphi units. They are structured very much like a Delphi form unit with the class declaration for the component appearing in the interface section and the actual method definitions appearing in the implementation section. The fact that a component unit and a form unit are structured similarly is no coincidence. Remember that TForm is a descendant of TComponent, and when you create a new form in Delphi, you are in essence creating a new form component.

The next step is to create the component unit. As Figure 2 shows, this can be performed either manually or by using the Component Expert. The Component Expert actually does more than simply create the unit. It also generates placeholders for all of the basic elements required in a component unit. These placeholders include a partially filled uses clause, an empty class declaration, and a Register procedure. The dashed box in Figure 8 corresponds to the tasks that must be performed to produce the same output as the Component Expert.

TIP: *If you use the Component Expert to construct the component unit, it is highly recommended that you remove the Register procedure from the unit. This will be explained in detail in the section on Registration Units later in this document.*

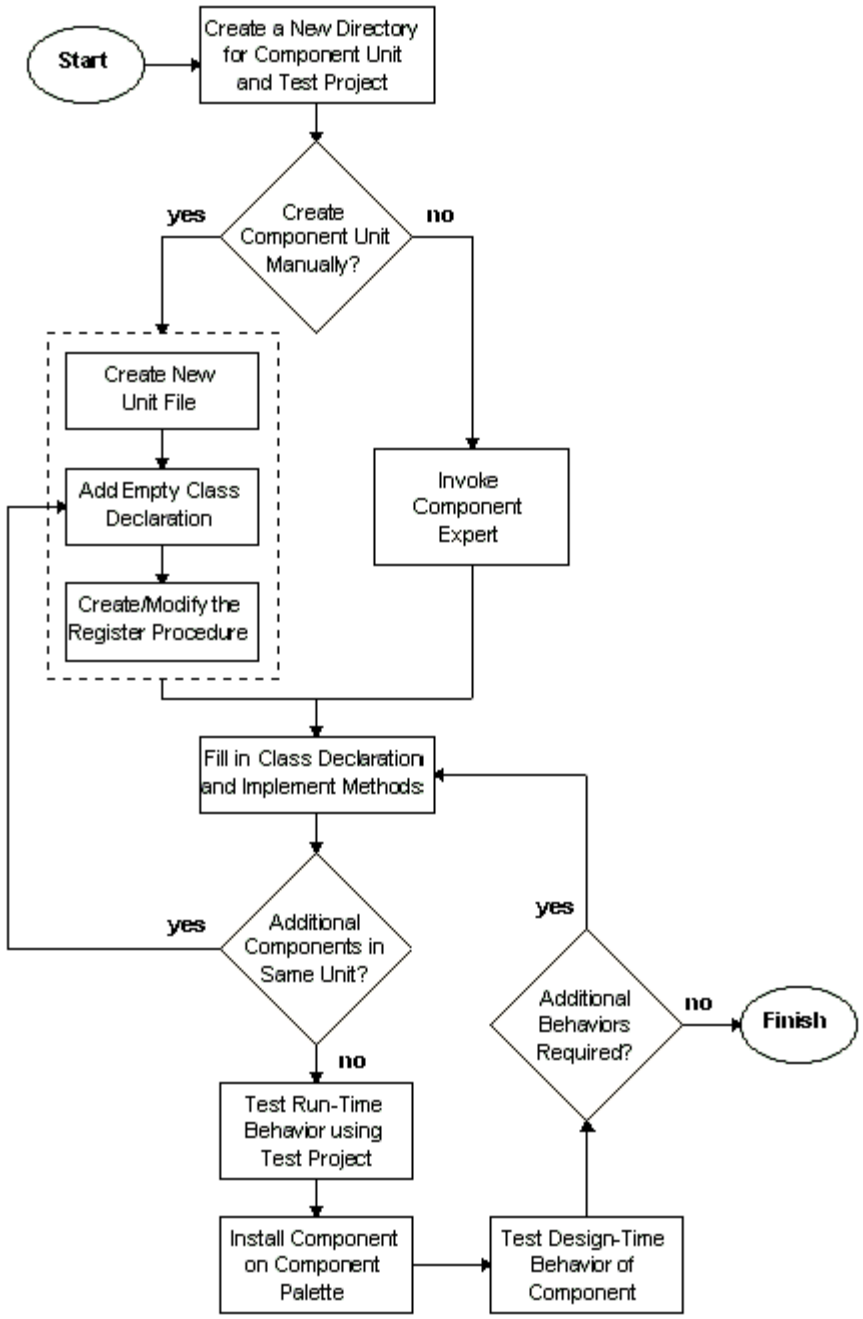

Figure 2: The process of building a custom component

After the basic elements of the unit are specified, either manually or by using the Component Expert, the next step is to fill out the class declaration and write the supporting methods. Once the coding tasks are completed, the component can be tested. Since components have two distinct interfaces, runtime and design-time, testing is performed in two steps. Runtime testing can commence as soon as coding is finished, but design-time testing can only occur once the component has been registered with Delphi and appears on the Component Palette.

Naming Conventions

The name you choose for your component class will dictate how the component will be referenced in the Delphi environment. If the class starts with a T, which it should by convention, Delphi strips off the first character and uses the remaining string as the name of the component. If the class name does not start with a T, then the entire class name is used. The resulting name is displayed as a hint when the cursor is positioned over the component in the component palette. Likewise, this same name with a numeric suffix (for example, Button4) is used as the default name for each component of this type that you drop on a form.

In addition to T, it has become commonplace to use an additional prefix when naming components. The prefix serves as an identifying string indicating the author of the component, for example, the author's initials. But the prefix does not have to represent a single person. It may be the name or abbreviation of a company, or even a product name. Table 10 shows some examples of names used in some component packages currently on the market.

Sample Component	Prefix	Company	Product
TRzCheckList	Rz	Raize Software	Raize Components
TOvcEditor	Ovc	TurboPower Software	Orpheus
TwwTable	ww	Woll2Woll Software	InfoPower

Table 2: Samples of prefixes used in commercial component products.

Component prefixes also tend to be more practical. In order for Delphi to install a component, its component name must be unique. Therefore, if you purchase two component packages and each one contains a TVirtualListBox component, Delphi will only allow one of those components to be installed.

In addition to selecting the component name, a name for the unit file must also be selected. Do not underestimate the importance of this task. To avoid ambiguity, it is best to have unique file names. For example, it would not be wise to use the unit name of Buttons, because Delphi already has a unit of this name. As a result, like component names, unit names are commonly prefixed with an identifier string. Keeping the unit files unique is especially important when installing multiple components from many different sources.

Testing the Runtime Interface

Just as in application programming, once the implementation code for the component has been written, it's time to test it. Testing a component is a little different from testing a complete Delphi application. The difference stems from the fact that components have two separate, although similar, interfaces that need to be tested. When a component is used in an application, the runtime interface of the component determines the behavior of the component. However, when a component is dropped onto a form within the Delphi environment, your component's behavior is dictated by its design-time interface. Both of these interfaces need to be tested in order to create a quality component. It is quite possible to create a component that behaves properly at runtime but improperly at design-time, and vice-versa.

The runtime interface of a component is tested first. We start with this one for several reasons. First, it's much easier to test the runtime behavior of a component than its design-time behavior. Second, for most components, the design-time interface of a component is very similar to the runtime interface. Testing the runtime environment first helps to ensure that the component works properly at design-time. And third, the integrated debugger can be used for runtime debugging. This option is not easily available at design-time.

Creating the Test Application

To test a new component, we need to create a test application. This is a primary reason for creating a separate directory for each component unit. The directory contains the source file for the component and all the files associated with the test project. Keeping the test application around also makes it easier to maintain your components, because when it comes time to modify a component, you will have the test application already built.

So how will we test our new component when it doesn't appear on the component palette? Instead of using the Form Designer to drop a component onto the form and then use the Object Inspector to change some of its properties, we will dynamically create an instance of the component when the form of our application is created. Dynamically creating a component allows us to create the component without having to register it with Delphi. Registration is the process by which Delphi becomes aware of the components that reside on the component palette. Registration must be performed in order to test the design-time behavior of a component, but for testing the runtime behavior it isn't necessary.

In the same directory where your component unit exists, create a new project. All you will need to test a

component is a single form. The following steps summarize how to dynamically create a component:

1. Add the component unit to the form's uses clause.
2. Add a field in the form class that will be a reference to the component.
3. Create the component in the form's OnCreate event.
4. Set the Parent property of the component (typically to the form).
5. Set additional properties of the component as required.

Installing a Component on the Palette

In order to proceed to the next level of testing, the component must be installed onto the component palette. For a component to appear on the component palette, it must be linked into a design-time package.

a) Packages

A package is basically a collection of Delphi units wrapped inside a specially compiled dynamic link library. Delphi takes care of exporting the interface sections of the embedded units to other Delphi applications, including the IDE. This includes classes and components defined in the units. Thus, a Delphi package is much more than a simple collection of functions and procedures that you would find in a more traditional DLL.

Packages come in two flavors: runtime and design. Runtime packages operate like standard DLLs. That is, they provide functionality to multiple applications without binding the code directly to each application. For example, the core VCL now resides in a runtime package, VCL50.bpl (BPL stands for Borland Package Library). As a result, only one copy of the VCL resides in memory for all Delphi applications that use packages.

Design packages are used to install components, property editors, and component editors into the Delphi IDE. This is a significant departure from the Delphi 2 approach, which linked all component units into a single component library, CmpLib32.dcl. As a result, design-time packages can be dynamically loaded or released by the IDE.

b) The Component Resource

There is an additional task that is performed by the install process. When a component unit is installed, Delphi searches the directories listed in the search path for a file that has the same name as the unit but with a .dcr extension. DCR stands for Delphi Component Resource, and a DCR file is actually a Windows resource file. If a component resource file is found for a component being installed, its contents are also linked into the package. The main purpose of the component resource file is to specify the bitmaps that are used to represent the components on the component palette.

The Image Editor that ships with Delphi can be used to create a DCR file. However, I prefer to use Resource Workshop to create component resource files. Of course, any tool that can build a compiled (.RES) resource file will suffice. You could even use a text editor to create a .RC file that includes bitmaps that were created with Windows Paintbrush. The BRC resource compiler could then be used to build the .RES file. Just don't forget to change the extension of the file to .DCR.

Inside the component resource file, create a 24x24 pixel 16-color bitmap for each component defined in the corresponding unit. In order for Delphi to link each bitmap to its associated component, give each bitmap the same name as the component's class name. The only difference is that the bitmap name is in all upper case.

The lower left pixel in the bitmap serves as the transparent color indicator. All other pixels in the bitmap with this same color will appear transparent. If you do not create a component resource file for your new components, Delphi will use the bitmap of its ancestor if it has one, or it will use a default bitmap.

Testing the Design-Time Interface

Once the component is installed on the palette, it can be used in a Delphi application. Of course, it would be helpful to at least test its design-time behavior before doing so. Testing the design-time features of a component can become quite involved, especially for a component with a great many properties.

The key areas that need to be tested are behaviors that are designed to only execute at design-time-for example, custom painting. It is also important to test any property interactions that have been built into the new component.

Documenting Commercial VCL Components

Documentation Benefits

a) Reduction in Pre/Post Sales Support

Documentation for components sold in the open market is particular important as 'face to face' interaction does not take place between author and customer. Providing a comprehensive set of documentation will ensure that pre/post sales support is kept to a minimum. Providing pre sales documentation i.e. a thorough component specification prevents many of the refund situations common in traditional 'box product' channels.

Traditional channels sell product by providing marketing information but not the finer detail covered in help files and other technical documentation. Providing information such as help files and evaluations enables customers to make an 'informed' purchase decision. Documenting and publishing known issues such as Frequently Asked Questions (FAQ's) on a regular basis will also help reduce technical support after the sale.

b) The Confidence Factor

Components sold on the open market are 'Black Box' i.e. the source code is hidden. Because of this, trust is extremely important between customer and author. Therefore, provision of detailed product information such as evaluations, help files and white papers is essential for building confidence in potential customers.

Typical Documentation

It is absolutely essential that vendors provide documentation with commercial components. The most common type of component documentation is an extensive online help system. In addition, some vendors even provide options for printed manuals or PDF editions of manuals.

a) Online Documentation (HTML, HLP and PDF Files)

There are two ways in which a user can request context-sensitive help on a component within the Delphi design environment. First, the user can press F1 while the Object Inspector has the focus. In this case, Delphi displays the help associated with the currently highlighted property or event. The second way occurs when the user presses the F1 key when a form has the focus. In this instance, Delphi displays the help screen associated with the currently selected component.

However, context-sensitive help is only available for the components, properties, and events that Delphi knows about. For example, if we drop a new custom component onto a form, select one of its properties in the Object Inspector, and then press F1, Delphi displays the Topics Found dialog box with a list of topics. Unfortunately, none of these topics will be correct because we have not yet instructed Delphi where to find help for the new component. In order to do this, we must first create the component help file.

Virtually any tool that can be used to create a Windows-based help file can be used to create a component help file. The key to creating a component help file is in using the correct format for your topic footnotes so that the WinHelp engine will locate the desired topic.

Context-sensitive component topic lookups are handled through the use of A-footnotes. When the F1 key is pressed, Delphi instructs WinHelp to search for an A-footnote that matches one of the patterns listed in Table 3. If more than one match is found, WinHelp displays the Topics Found dialog box listing all of the matches. Otherwise, the topic defining the A-footnote is displayed.

Format	Example	Use for

ClassName	TRzLauncher	Main topic page
ClassName_PropName	TRzLauncer_FileName	Component-specific property
ClassName_MethodName	TRzLauncher_Launch	Component-specific method
ClassName_EventName	TRzLauncher_OnFinished	Component-specific event
PropName_Property	Parameters_Property	General property
MethodName_Method	Execute_Method	General method
EventName_Event	OnClick_Event	General event

Table 3: A-footnote formats used by Delphi and C++Builder

When designing the help file for your product, it is important to understand how help is provided in other products. There is a lot to be said for consistency. And providing your online help in a similar format to what is standard in the industry is very helpful to the end-user.

Another documentation alternative that many vendors are moving towards is to provide Portable Document Format (PDF) versions of their manuals. The files can be distributed electronically with the product, or can be uploaded to the company's web server for pre-sales documentation. The benefit is that users have the opportunity to print out the chapters or sections that they are interested in.

b) Demonstrations

Developing a product demonstration can prove a valuable asset in the documentation you provide customers. Exposing component functions will help users understand the benefits of the product as a component-based solution. Demonstrations are compiled applications assembled with the component. They are not like evaluations that allow developers to use the component in a development environment. More information on evaluations is covered in the following topic.

The objective of a demonstration is to educate users on the functionality incorporated inside the component. The interface should demonstrate the main functions in a format that is understandable for all customers. Because of this it's important to remove industry jargon and acronyms that may confuse users. For data bound components, providing the option of entering a DSN (Data Source Name) could be of benefit. This allows users to connect to internal data sources in their own organisation and apply meaningful data in context with the component.

c) Evaluations

Component authors recognise evaluations will help secure a product sale. Once a customer is happy with a specification they often trial the component to check the component will actually provide the functionality they are looking for. Customers do not doubt component based development, but may have concerns with an 'independent' solution, because of this component evaluations are essential. Unlike applications, component evaluations add value and play a significant role in the pre sales process.

Writing an evaluation will require consideration into security. Producing a component that displays a reminder screen or setting time limits hidden in cryptic keys within the registry are just some of the techniques currently used. Setting a 5-10 day trial period for technical components and 10-30 days for complex business components is recommended. This gives the customer enough time to evaluate the product and make a decision whether to buy.

An ideal evaluation is the full retail version restricted by a security feature detailed above. This prevents users having to download the evaluation and retail component separately. ComponentSource has developed a license protection facility called C-LIC primarily designed to protect evaluations that can be unlocked into full retail products. C-LIC displays a reminder screen requesting the user to enter a license key provided when the full retail is purchased. More information on C-LIC is covered in the topic 'Deploying COM Components'.

However, because full versions of VCL-based components are usually distributed with full source code, this ideal approach to producing evaluations is rarely reached. In most cases, a vendor will produce a custom evaluation that will not include source code, and then when a user decides to purchase the full

version, a full, unrestricted version of the components are downloaded. The reason for this is that it is not wise to distribute the source code that contains the protection scheme that is used to create the evaluation version. This situation typically does not occur in other component architectures because the source code is rarely ever distributed.

d) Sample Code

Sample code is particularly useful when developers need to prototype and assess component functionality. A good technique is to provide the sample code used in the component demonstration. If possible, this should be provided in a basic, intermediate and advanced version. This will allow the developer to grasp how the demonstration was developed and it's stages of advancement throughout its development cycle.

Sample code usually is the final step that customers evaluate before making a decision whether to buy. Therefore its important to maintain a good perception by commenting all code and explaining exactly what happens and why. The quality of sample code will directly correlate to the quality of your final product. Because of this professionally written sample code using correct naming conventions, coding structures and error handling is essential. If the sample code is well structured then it can be reused in actual projects. This makes the whole process of integration far less complex and useful for developer's who need to rapidly assemble a component-based solution.

e) Readme Files

In this topic we list the various information that a Readme file should contain. Most installation scripts provide users with an opportunity to view a Readme file for last minute changes or errata information once installation is complete. These files should be written in a universal file format i.e. a text (TXT) file or HTML file. This prevents users having to own proprietary applications such as Microsoft Word to view the file. The following list provides an insight into the various information supplied in component Readme files.

Products Changes - this section is extremely important and should note all the functional changes that have been made in comparison to previous versions and any changes to documentation, installation etc.

Bug Fixing - bugs resolved from previous versions should be fully documented. Include the component version that contained the bug and a description of what has changed. This is particularly important if the component's interface has been changed.

System Requirements - Although compatibility information is supplied in our own sales documentation its worth reiterating this information in your Readme file. This should include information such as operating system for deployment, safety levels, threading standards etc.

Service Pack Installation - You should define any services packs that were applied when compiling the component. This often is the reason for components failing to run in a user's development environment.

Definitions of Component Filenames - Listing the filenames of all components (including dependencies) is particularly useful if the user is attempting to identify a problem. Although help and dependency files include this information, Readme files are often browsed as well.

Detailed Installation Notes - This should include information on how to de-install and update previous versions. A troubleshooting section should also be included defining solutions to common installation problems.

Notes on Sample Projects - Document any assumptions, known issues etc. If possible, describe each of the projects and the functions they expose. In addition to this defining a project's complexity i.e. basic, intermediate or advanced can also be of help.

Distribution Information - Particularly useful when a user creates an installation kit. Your component may reference many other dependencies, therefore detailing this information

will help the developer create a tailored installation kit and prevent many of the 'missing dependency' issues when testing.

Known Issues - You must document all known issues. If possible, also explain why the problem arises. If you do not provide this information then it's likely that unnecessary technical support issues will arise. Documenting known issues will demonstrate that you care and are focussed on providing a future solution

h) Compatibility

The following topic looks at the compatibility aspects of a software component. Publishing your product on www.componentsource.com will require a comprehensive specification of the component's compatibility. The product submission form that we ask you to complete covers the eight areas detailed below.

Operating System for Development - This section covers the different operating systems that your component can run on. The component may run on Windows 98 and Windows NT however this does not mean that it runs on Windows 2000. Unix is another example of an operating system that exists in many flavors such as Solaris, HPUX and Aix. Because of this, testing component functionality before labeling an operating system 'compatible' is essential. Any future technical issues that arise due to operating system compatibility will require support.

Architecture of Product - The architecture of a component defines the type of system the component is compatible with. For instance, developers maintaining legacy systems may find it especially important to know a component is '16 bit' as opposed to '32 bit'. The fact a component can run in a '16 bit' environment such as Windows 3.1 may be the reason a customer decides to purchase. Another factor worth considering is consistency. If you label your component '32 bit' then do not check Microsoft Access 2.0 or Microsoft Visual Basic 3.0 as compatible as these are 16 bit products!

Tool Type - This section defines your software as an application tool, component, add-in etc. Again, selecting the tool type will be directly related to the containers the component can be used in.

Component Type - Defining the component type is particularly important when filtering our product catalog for co-branded sites. Your component will be receiving exposure in different filtered catalogs aimed at specific audiences - there is little point in marketing a VCL (a Delphi only component) to Microsoft Visual Basic audience. Consider each of the types listed - defining a component as a DLL does not define the component with a COM interface. In this particular case the component would not be published on a co-branded site such as Microsoft unless it was labeled as a COM Object/ActiveX DLL/In-Process Server.

Built Using - Stating the framework that a component is built with is especially useful when establishing if the component will run correctly in a design time environment. For example, a Microsoft Visual Basic 6.0 component may not work in a Visual Basic 5.0 environment due to the absence of MSVBVM60.dll. This does not mean the component is not compatible but provides the developer with information on the basic run-time libraries required for the component to run.

Compatible Containers - This section defines each development environment in which the component can be used. Mark only those environments that you have tested and can support your component in. Completing this section will make you eligible for different marketing initiatives and inclusion into catalogs targeted at specific audiences such as 'ActiveX' or 'Delphi' users.

General - This section includes options not easily categorized. Digital signatures, compatibility, scalability and support for apartment model threading are just some of the options that may require inclusion.

Year 2000 Compliance - If the component includes a date function or is anyway related to file storage then test for Year 2000 issues. Although components such as calendar controls are the obvious candidates the application tools can also be eligible for such tests. Tools that compile files such as help authoring tools and installation wizards must state Year 2000 compliance - either 'Yes', 'No' or 'Not Relevant'.

Deploying Commercial VCL Components

There are several issues that you must be aware of when it comes to deploying VCL components, especially in the commercial marketplace.

Component Installation

a) A Common Include File

Unlike other component models, it has become quite common (some would say mandatory) for vendors to include the source code for their components. If you plan on doing this, you should seriously consider using a common include file in all of your component units. The purpose of this file is to set compiler directives and conditional defines that govern how the components are compiled.

Using an include file is especially important when distributing the source code along with the compiled versions of your components. If the user decides to recompile your code, you cannot assume that the compiler settings on that machine are the same as the ones you used to develop the component. Therefore, by creating a common include file, no matter what the current compiler settings are, your component units will always be compiled with the correct settings.

For these same reasons, a common include file is essential for multi-person development teams. The include file guarantees that everyone builds the components using the same options.

b) Packaging

When you create a new custom component, it is very tempting to add the component to the DclUsr50 package so that it can be quickly installed into Delphi. This is fine for testing, but when it's time to distribute your components to other developers, you'll want to supply custom runtime and design packages for your components. To understand why, we need to take a closer look at the DclUsr50 package.

DclUsr50 is defined as a design-only package, meaning that it can be loaded by the IDE in order to install the components that reside within it. Note that design packages are only used by the IDE-they are never distributed with an application. When an application is "built with packages," the program is compiled so that it utilizes runtime packages. This is a subtle, but important point.

The predefined packages that ship with Delphi come in runtime/design pairs. For example, the runtime package containing the database related components is VclDb50. Its design counterpart is DclDb50. To distribute a database application that uses packages, you also need to distribute the VclDb50.dpl package. If the recipient of your application already has this package, then this step is not necessary. This works because there is only one version of the VclDb50 package.

So what does this have to do with DclUsr50? Actually, quite a bit. DclUsr50 does not have a runtime counterpart, which is not surprising because every developer will have a different set of components within that package. And without a runtime package, all of the components inside DclUsr50 must be statically linked into an application. For one or two components, this won't have much of an impact, but for an entire library of components, installing them into DclUsr50 defeats the purpose of runtime packages.

Yet, this is not the only reason to supply your own packages. Installing a pre-built package is much easier than adding component units to DclUsr50. In fact, the entire process can be automated.

c) Creating Custom Packages

There are four steps required to create a package. First, you must decide which type of package to create. There are actually four possible choices, but for component packaging, we can limit the selection to runtime-only versus design-only. Runtime-only packages contain only the units necessary to implement one or more components. A runtime package should never contain property editors, component editors, or other design-specific code. This type of functionality should be placed in a separate design package. I recommend creating the runtime package first because, as we'll see, doing so simplifies the construction of the design-time package.

Next, you must select a name for your package. This may sound like a trivial task, but it is very important to select a name wisely. Recall that packages are designed to be distributed to users. Therefore, package names should be unique even across different versions of the same package.

The third step in creating a package is to determine the units to be contained in the package. For runtime packages, you must specify the units that implement the components to be distributed in the package. For design packages, you must specify design-related units, which includes units that implement property editors, component editors, and registration units.

Any number of units can be added to a package. The only restriction is that you should not add a unit that is already contained in another package. Doing so prevents Delphi from being able to load both packages at the same time.

The fourth step is to determine which packages are required by the new package. As noted earlier, all components ultimately descend from TComponent, which is defined in the Classes unit that comes with the VCL. As a result, any new package that you create that contains a new component will require the VCL50 package, because it contains the Classes unit. Design packages often require their runtime counterparts, because the runtime version contains all the component units used by the editors and registration code contained in the design package.

d) Preventing Unauthorized Access

With the runtime packages created, we can now shift our attention to the design package. However, before we dive into details, we need to examine a problem that may arise with the way the Component Expert generates a component unit. The Component Expert automatically generates a Register procedure at the end of the unit. Under Delphi 1 and 2, having the Register procedure in the component unit poses no threat. In fact, it is quite useful because it allows individual components to be installed.

Unfortunately, under Delphi 3 and later, leaving the Register procedure in your component unit can have undesirable effects. Specifically, your components can be installed by others even without your component unit or package!

This situation arises when someone uses your component in a property editor or component editor, and then implicitly imports your unit into their design package. A unit is implicitly imported into a package when the runtime package containing the unit is not specified on the package's requires list.

When the property editor or component editor using your component is installed, your component will also be installed because Delphi finds the Register procedure defined in the component unit. This results in your component appearing on the component palette even though your component unit and package may not be present.

Fortunately, even though your component can be placed onto a form, the application cannot be compiled because the user does not have the component unit nor the package. But this is still certainly an undesirable effect.

There are two ways to prevent the above scenario from occurring. The first is for property editor and component editor writers to avoid implicitly importing your component units. The second is to remove the Register procedures from your component units and place them in a registration unit. Because we have no control over editor writers, the second option is preferred.

e) Registration Units

As its name suggests, the purpose of a registration unit is to register components, property editors, and component editors with Delphi. Listing 2 shows the source code for the a sample registration unit.

```
unit SampleReg;
 interface
     procedure Register;
 implementation
 uses
     Forms, Classes, Controls, SysUtils, DsgnIntf,
     { Your component units are listed here };
 procedure Register;
 begin
     // Register Components
     { Register your components here by calling RegisterComponents }
     // Register Property Editors
     {Register your property editors here by calling RegisterPropertyEditor}
     // Register Component Editors
     {Register your comp. editors here by calling RegisterComponentEditor}
 end;
 end.
```

Listing 2: SampleReg.pas

Now that we have a registration unit, we can return our attention to the design package. The package is quite simple thanks to the registration unit and runtime packages created earlier. The contains clause lists all of the design-time specific units including the registration unit. The requires clause of a design package will specify the runtime packages created earlier.

f) What Files Should be Deployed

There are a variety of ways in which VCL-based components can be deployed. For instance, is it sufficient to distribute only the BPL files for each package?

The answer to this is no, because although deploying the BPL files is sufficient when distributing a compiled application to an end user, the BPL file alone is not sufficient to compile a new application that uses any of the components in the package. In this case, it is necessary to distribute the DCP files as well. Table 4 lists various file types associated with components and describes when these files should be deployed to other Delphi developers.

File Type	Extension	Reasons to Distribute
Runtime Package Library	*.bpl	These files are needed to support the use of the runtime package version of your components. These files are also often used to support design packages.
Runtime Package Symbols	*.dcp	These files are needed to compile a project that uses the corresponding runtime package library.
Design Package Library	*.bpl	These files are needed to install your components and design editors.
Design Package Symbols	*.dcp	Distribute these files if you want to allow others to use your design editors in their own packages.
Compiled Units	*.dcu	These files are needed to compile a project that does not use runtime packages. Do not distribute these files if you want to force developers to use your runtime packages.
Form Files	*.dfm	Same rules as Compiled Units
Source Files	*.pas	Source code is a convenience to developers. Distributing the source code in place of the compiled units is also a valid alternative.

Table 4: Determining which files to distribute.

Component Testing

How do I test a component? - Thorough testing is paramount to the success of a component being excepted in the open market. All evaluations and sample code should be tested in addition to the full retail product for functionality, installation and de-installation. An issue that should be approached with

care is the dependencies referenced by your component. Most installation tools require the selection of the original component's project file. This allows the wizard to analyze all references selected at the time the component was compiled. Absence of dependent files referenced by other dependent files is probably the most common installation issue. This is why testing on a clean machine, on all operating systems and all development environments is imperative. If this rigorous testing process is not followed then the likelihood of damaging a customers system is high. Therefore, to create a clean machine you must:

Format Hard Disk - If you only reinstall the operating system then static files that do not require registration may have already been installed. Therefore, without formatting the disk there is no guarantee that the installation will work on all machines.

Install Operating System - Make a note of any service packs applied as this must be included in the component's documentation i.e. the Readme file

Install Development Environment - Again, document any service pack installations. Always select the standard installation otherwise certain files may be missing causing erroneous errors when you test.

Test installation - Although we test the product installation thoroughly we recommend you also test the product to your best ability. This will ensure the swift progress of the component through our QA system.

Once the above steps are complete you can image the disk allowing you to re-clean your environment in minutes. Image applications take a snapshot of your clean system, with operating system and development environment installed. This prevents the long cycle of re-installing everything before testing can re-commence. A good practice is to allocate a hard disk per operating system per development environment. As several disks can be installed in one machine, imaging an environment provides an economical and effective solution.

Component Licensing

How do I license my component? - Unlike application licensing, few licensing utilities exist for component licensing. Those that currently exist often operate a 'two-phase unlock' process which requires manual intervention to generate the retail unlock key. However, this form of licensing does not suit 'mass market' adoption. The development of the C-LIC (common licensing) component enables authors to integrate a DLL providing a 'Try-Before-You-Buy' licensing solution.

a) The Common Licensing Problem

Components sold on the open market have are typically 'Black Box' architecture. This means that all functionality is encapsulated and cannot be adapted by the developer except through the public interface. Because of this, providing an evaluation that allows the developer to 'road test' a component is important when securing a sale. Nowadays, customers expect one download that runs in evaluation mode for a set number of days. Once this evaluation has expired, functionality is disabled until a license key is purchased and entered, unlocking the component into a full retail version. Often the best form of protection is to use a reminder/nag screen that launches each time the calling application runs the component. This prevents users without a license from releasing an application into a commercial environment.

b) C-LIC - The Common Licensing Solution

C-LIC is the ComponentSource license technology used to adapt a full retail product into an evaluation. However, please note the current version does not support copy protection. The C-LIC DLL can be integrated into a majority of languages that support the creation of software components. Its method of working is similar to that used in application software. C-LIC was developed to enable component authors to create a fully or part functioning evaluation protected by a 'nag' screen reminding users that the component is unlicensed. The nag screen allows customers to browse to the relevant product page and purchase the license key used to unlock the product into the full retail component. The license key is provided by ComponentSource and is generated by our own proprietary encryption.

C-LIC can also protect different levels of functionality. For example if your standard version has 10 functions and your professional version 20 functions then the purchase of a standard license will unlock

10 functions only - the other 10 functions will remain in evaluation mode. Please Note: C-LIC does not provide "copy protection".

Conclusion

Build VCL components and enter the component market now!

Many different companies of various sizes from around the world have already created new VCL based components and entered the "open market". For example:

- devSoft - IP*Works! for Internet connectivity via standard protocols
- LEAD Technologies - LEADTOOLS VCL components for imaging
- Multilizer - Multilizer VCL for multi-language software localization
- Raize Software - Raize Components for enhanced user interfaces
- Seagate Software - Crystal Reports VCLs for data reporting
- Steema Software - teeChart Pro VCL for charting and graphing data

Developer demand for components is currently outstripping supply - as a result an opportunity exists for experts to create components and enter the "open market" for components.

If you would like to learn more about the technical details regarding the construction of native VCL components, the following references will help:

Developing Custom Delphi 3 Components
Author: Ray Konopka
Publisher: Coriolis Group Books
ISBN: 1-57610-112-6
Status: Book is unfortunately out of print, but a PDF Edition of the book is available from Raize Software (http://www.raize.com).

Delphi Component Design
Author: Danny Thorpe
Publisher: Addison Wesley
ISBN: 0-201-46136-6
Status: Out of print

Delphi 5 Developer's Guide
Author: Steve Teixiera, et al
Publisher: Sams
ISBN: 0-67231-781-8

Component Writer's Guide
Included with the Borland Delphi and C++Builder Documentation

If you have any feedback on this white paper or questions about creating commercial software components email us on: publishers@componentsource.com

ComponentSource