Writing Delphi Custom Components

CHAPTER

21

IN THIS CHAPTER

The ability to easily write custom components in Delphi 5 is a chief productivity advantage that you wield over other programmers. In most other environments, folks are stuck using the standard controls available through Windows or else have to use an entirely different set of complex controls that were developed by somebody else. Being able to incorporate your custom components into your Delphi applications means that you have complete control over the application's user interface. Custom controls give you the final say in your application's look and feel.

If your forte is component design, you will appreciate all the information this chapter has to offer. You will learn about all aspects of component design from concept to integration into the Delphi environment. You will also learn about the pitfalls of component design, as well as some tips and tricks to developing highly functional and extensible components.

Even if your primary interest is application development and not component design, you will get a great deal out of this chapter. Incorporating a custom component or two into your programs is an ideal way to spice up and enhance the productivity of your applications. Invariably, you will get caught in a situation while writing your application where, of all the components at your disposal, none is quite right for some particular task. That's where component design comes in. You will be able to tailor a component to meet your exact needs, and hopefully design it smart enough to use again and again in subsequent applications.

Component Building Basics

The following sections teach you the basic skills required to get you started in writing components. Then, we show you how to apply those skills by demonstrating how we designed some useful components.

Deciding Whether to Write a Component

Why go through the trouble of writing a custom control in the first place when it's probably less work to make do with an existing component or hack together something quick and dirty that "will do"? There are a number of reasons to write your own custom control:

- You want to design a new user-interface element that can be used in more than one application.
- You want to make your application more robust by separating its elements into logical object-oriented classes.
- You cannot find an existing Delphi component or ActiveX control that suits your needs for a particular situation.
- You recognize a market for a particular component, and you want to create a component to share with other Delphi developers for fun or profit.
- You want to increase your knowledge of Delphi, VCL internals, and the Win32 API.

One of the best ways to learn how to create custom components is from the people who invented them. Delphi's VCL source code is an invaluable resource for component writers, and it is highly recommended for anyone who is serious about creating custom components. The VCL source code is included in the Client Server and Professional versions of Delphi.

Writing custom components can seem like a pretty daunting task, but don't believe the hype. Writing a custom component is only as hard or as easy as you make it. Components can be tough to write, of course, but you also can create very useful components fairly easily.

Component Writing Steps

Assuming that you have already defined a problem and have a component-based solution, here are the important points in creating a component—from concept to deployment.

- First, you need an idea for a useful and hopefully unique component.
- Next, sit down and map out the algorithm for how the component will work.
- Start with the preliminaries—don't jump right into the component. Ask yourself, "What do I need up front to make this component work?"
- Try to break up the construction of your component into logical portions. This will not only modularize and simplify the creation of the component, but it also will help you to write cleaner, more organized code. Design your component with the thought that someone else might try to create a descendant component.
- Test your component in a test project first. You will be sorry if you immediately add it to the Component Palette.
- Finally, add the component and an optional bitmap to the Component Palette. After a little fine-tuning, it will be ready for you to drop into your Delphi applications.

There are six basic steps to writing your Delphi component.

1. Deciding on an ancestor class.
2. Creating the Component Unit.
3. Adding properties, methods, and events to your new component.
4. Testing your component.
5. Registering your component with the Delphi environment.
6. Creating a help file for your component.

In this chapter, we will discuss the first five steps; however, it is beyond the scope of this chapter to get into the topic of writing help files. However, this does not mean that this step is any less important than the others. We recommend that you look into some of the third-party tools available that simplify writing help files. Also, Borland provides information on how to do this in their online help. Look up "Providing Help for Your Component" in the online help for more information.

Deciding on an Ancestor Class

In Chapter 20, "Key Elements of the Visual Component Library," we discussed the VCL hierarchy and the special purposes of the different classes at the different hierarchical levels. We wrote about four basic components from which your components will descend: standard controls, custom controls, graphical controls, and non-visual components. For instance, if you need to simply extend the behavior of an existing Win32 control such as `TMemo`, you'll be extending a standard control. If you need to define an entirely new component class, you'll be dealing with a custom control. Graphical controls let you create components that have a visual effect, but don't take up Win32 resources. Finally, if you want to create a component that can be edited from Delphi's Object Inspector but doesn't necessarily have a visual characteristic, you'll be creating a non-visual component. Different VCL classes represent these different types of components. You might want to review Chapter 20 unless you're quite comfortable with these concepts. Table 21.1 gives you a quick reference.

Table 21.1 VCL Classes as Component-Based Classes

VCL Class	*Types of Custom Controls*
`TObject`	Although classes descending directly from `TObject` are not components, strictly speaking, they do merit mention. You will use `TObject` as a base class for many things that you don't need to work with at design time. A good example is the `TIniFile` object.
`TComponent`	A starting point for many non-visual components. Its forte is that it offers built-in streaming capability to load and save itself in the IDE at design time.
`TGraphicControl`	Use this class when you want to create a custom component that has no window handle. `TGraphicControl` descendants are drawn on their parent's client surface, so they are easier on resources.
`TWinControl`	This is the base class for all components that require a window handle. It provides you with common properties and events specific to windowed controls.
`TCustomControl`	This class descends from `TWinControl`. It introduces the concepts of a canvas and a `Paint()` method to give you greater control over the component's appearance. Use this class for most of your window-handled custom component needs.

VCL Class	*Types of Custom Controls*
`TCustomClassName`	The VCL contains several classes that do not publish all their properties; they leave it up to descendant classes to do. This allows component developers to create "custom" components from the same base class and to publish only the predefined properties required for each customized class.
`TComponentName`	An existing class such as `TEdit`, `TPanel`, or `TScrollBox`. Use an already established component as a base class for your class (such as `TEdit`) and custom components when you want to extend the behavior of a control rather than create a new one from scratch. Many of your custom components will fall into this category.

It is extremely important that you understand these various classes and also the capabilities of the existing components. The majority of the time, you'll find that an existing component already provides most of the functionality you require of your new component. Only by knowing the capabilities of existing components will you be able to decide from which component to derive your new component. We can't inject this knowledge into your brain from this book. What we can do is to tell you that you must make every effort to learn about each component and class within Delphi's VCL, and the only way to do that is to use it, even if only experimentally.

Creating a Component Unit

When you have decided on a component from which your new component will descend, you can go ahead and create a unit for your new component. We're going to go through the steps of designing a new component in the next several sections. Because we want to focus on the steps, and not on component functionality, this component will do nothing other than to illustrate these necessary steps.

The component is appropriately named `TddgWorthless`. `TddgWorthless` will descend from `TCustomControl` and will therefore have both a window handle and the capability to paint itself. This component will also inherit several properties, methods, and events already belonging to `TCustomControl`.

The easiest way to get started is to use the Component Expert, shown in Figure 21.1, to create a component unit.

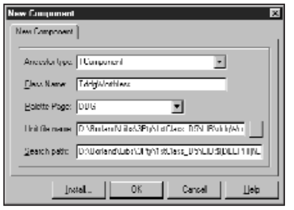

FIGURE 21.1
The Component Expert.

You invoke the Component Expert by selecting Component, New Component. In the Component Expert, you enter the component's ancestor class name, the component's class name, the palette page on which you want the component to appear, and the unit name for the component. When you select OK, Delphi automatically creates the component unit that has the component's type declaration and a register procedure. Listing 21.1 shows the unit created by Delphi.

LISTING 21.1 `Worthless.pas`, a Sample Delphi Component

```
unit Worthless;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
type
  TddgWorthless = class(TCustomControl)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('DDG', [TddgWorthless]);
end;
end.
```

You can see that at this point `TddgWorthless` is nothing more than a skeleton component. In the following sections, you'll add properties, methods, and events to `TddgWorthless`.

Creating Properties

Chapter 20 discusses the use and advantages of using properties with your components. This section shows you how to add the various types of properties to your components.

Types of Properties

Table 20.1 in Chapter 20 lists the various property types. We're going to add properties of each of these types to the `TddgWorthless` component to illustrate the differences between each type. Each different type of property is edited a bit differently from the Object Inspector. You will examine each of these types and how they are edited.

Adding Simple Properties to Components

Simple properties refer to numbers, strings, and characters. They can be edited directly by the user from within the Object Inspector and require no special access method. Listing 21.2 shows the `TddgWorthless` component with three simple properties.

LISTING 21.2 Simple Properties

```
TddgWorthless = class(TCustomControl)
  private
    // Internal Data Storage
    FIntegerProp: Integer;
    FStringProp: String;
    FCharProp: Char;
published
    // Simple property types
    property IntegerProp: Integer read FIntegerProp write FIntegerProp;
    property StringProp: String read FStringProp write FStringProp;
    property CharProp: Char read FCharProp write FCharProp;
  end;
```

You should already be familiar with the syntax used here because it was discussed previously in Chapter 20. Here, you have your internal data storage for the component declared in the `private` section. The properties that refer to these storage fields are declared in the `published` section, meaning that when you install the component in Delphi, you can edit the properties in the Object Inspector.

NOTE

When writing components, the convention is to make private field names begin with the letter *F*. For components and types in general, give the object or type a name starting with the letter *T*. Your code will be much more clear if you follow these simple conventions.

Adding Enumerated Properties to Components

You can edit user-defined enumerated properties and Boolean properties in the Object Inspector by double-clicking in the `Value` section or by selecting the property value from a drop-down list. An example of such a property is the `Align` property that exists on most visual components. To create an enumerated property, you must first define the enumerated type as follows:

```
TEnumProp = (epZero, epOne, epTwo, epThree);
```

You then define the internal storage field to hold the value specified by the user. Listing 21.3 shows two enumerated property types for the `TddgWorthless` component:

LISTING 21.3 Enumerated Properties

```
TddgWorthless = class(TCustomControl)
  private
    // Enumerated data types
    FEnumProp: TEnumProp;
    FBooleanProp: Boolean;
published
    property EnumProp: TEnumProp read FEnumProp write FEnumProp;
    property BooleanProp: Boolean read FBooleanProp write FBooleanProp;
  end;
```

We've excluded the other properties for illustrative purposes. If you were to install this component, its enumerated properties would appear in the Object Inspector, as shown in Figure 21.2.

Adding Set Properties to Components

Set properties, when edited in the Object Inspector, appear as a set in Pascal syntax. An easier way to edit them is to expand the properties in the Object Inspector. Each set item then works in the Object Inspector like a Boolean property. To create a set property for the `TddgWorthless` component, we must first define a set type as follows:

```
TSetPropOption = (poOne, poTwo, poThree, poFour, poFive);
TSetPropOptions = set of TSetPropOption;
```

FIGURE 21.2
The Object Inspector showing enumerated properties for `TddgWorthless`.

Here, you first define a range for the set by defining an enumerated type, `TSetPropOption`. Then you define the set `TSetPropOptions`.

You can now add a property of `TSetPropOptions` to the `TddgWorthless` component as follows:

```
TddgWorthless = class(TCustomControl)
 private
  FOptions: TSetPropOptions;
published
  property Options: TSetPropOptions read FOptions write FOptions;
end;
```

Figure 21.3 shows how this property looks when expanded in the Object Inspector.

FIGURE 21.3
The set property in the Object Inspector.

Adding Object Properties to Components

Properties can also be objects or other components. For example, the `TBrush` and `TPen` properties of a `TShape` component are also objects. When a property is an object, it can be expanded in the Object Inspector so its own properties can also be modified. Properties that are objects must be descendants of `TPersistent` so that their published properties can be streamed and displayed in the Object Inspector.

To define an object property for the `TddgWorthless` component, you must first define an object that will serve as this property's type. This object is shown in Listing 21.4.

LISTING 21.4 `TSomeObject` Definition

```
TSomeObject = class(TPersistent)
  private
    FProp1: Integer;
    FProp2: String;
  public
    procedure Assign(Source: TPersistent)
  published
    property Prop1: Integer read FProp1 write FProp1;
    property Prop2: String read FProp2 write FProp2;
  end;
```

The `TSomeObject` class descends directly from `TPersistent`, although it does not have to. As long as the object from which the new class descends is a descendant of `TPersistent`, it can be used as another object's property.

We've given this class two properties of its own: `Prop1` and `Prop2`, which are both simple property types. We've also added a procedure, `Assign()`, to `TSomeObject`, which we'll discuss momentarily.

Now you can add a field of the type `TSomeObject` to the `TddgWorthless` component. However, because this property is an object, it must be created. Otherwise, when the user places a `TddgWorthless` component on the form, there won't be an instance of `TSomeObject` that the user can edit. Therefore, it is necessary to override the `Create()` constructor for `TddgWorthless` to create an instance of `TSomeObject`. Listing 21.5 shows the declaration of `TddgWorthless` with its new object property.

LISTING 21.5 Adding Object Properties

```
TddgWorthless = class(TCustomControl)
private
  FSomeObject: TSomeObject;
  procedure SetSomeObject(Value: TSomeObject);
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
published
```

```
  property SomeObject: TSomeObject read FSomeObject write SetSomeObject;
end;
```

Notice that we've included the overridden `Create()` constructor and `Destroy()` destructor. Also, notice that we've declared a write access method, `SetSomeObject()`, for the `SomeObject` property. A write access method is often referred to as a *writer method* or *setter method.* Read access methods are called *reader* or *getter methods.* If you recall from Chapter 20, writer methods must have one parameter of the same type as the property to which they belong. By convention, the name of the writer method usually begins with `Set`.

We've defined the `TddgWorthless.Create()` constructor as follows:

```
constructor TddgWorthless.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FSomeObject := TSomeObject.Create;
end;
```

Here, we first call the inherited `Create()` constructor and then create the instance of the `TSomeObject` class. Because `Create()` is called both when the user drops the component on the form at design time and when the application is run, you can be assured that `FSomeObject` will always be valid.

You must also override the `Destroy()` destructor to free the object before you free the `TddgWorthless` component. The code to do this follows.

```
destructor TddgWorthless.Destroy;
begin
  FSomeObject.Free;
  inherited Destroy;
end;
```

Now that we've shown how to create the instance of `TSomeObject`, consider what would happen if at runtime the user executes the following code:

```
var
  MySomeObject: TSomeObject;
begin
  MySomeObject := TSomeObject.Create;
  ddgWorthless.SomeObjectj := MySomeObject;
end;
```

If the `TddgWorthless.SomeObject` property were defined without a writer method like the following, when the user assigns their own object to the `SomeObject` field, the previous instance that `FSomeObject` referred to would be lost:

```
property SomeObject: TSomeObject read FSomeObject write FSomeObject;
```

If you recall from Chapter 2, "The Object Pascal Language," object instances are really pointer references to the actual object. When you make an assignment, as shown in the preceding example, you refer the pointer to another object instance while the previous object instance still hangs around. When designing components, you want to avoid having to place conditions on your users when accessing properties. To prevent this pitfall, you foolproof your component by creating access methods for properties that are objects. These access methods can then ensure that no resources get lost when the user assigns new values to these properties. The access method for `SomeObject` does just that and is shown here:

```
procedure TddgWorthLess.SetSomeObject(Value: TSomeObject);
begin
  if Assigned(Value) then
    FSomeObject.Assign(Value);
end;
```

The `SetSomeObject()` method calls the `FSomeObject.Assign()`, passing it the new `TSomeObject` reference. `TSomeObject.Assign()` is implemented as follows:

```
procedure TSomeObject.Assign(Source: TPersistent);
begin
  if Source is TSomeObject then
  begin
    FProp1 := TSomeObject(Source).Prop1;
    FProp2 := TSomeObject(Source).Prop2;
    inherited Assign(Source);
  end;
end;
```

In `TSomeObject.Assign()`, you first ensure that the user has passed in a valid `TSomeObject` instance. If so, you then copy the property values from `Source` accordingly. This illustrates another technique you'll see throughout the VCL for assigning objects to other objects. If you have the VCL source code, you might take a look at the various `Assign()` methods such as `TBrush` and `TShape` to see how they are implemented. This would give you some ideas on how to implement them in your components.

CAUTION

Never make an assignment to a property in a property's writer method. For example, examine the following property declaration:

```
property SomeProp: integer read FSomeProp write SetSomeProp;
  ....
  procedure SetSomeProp(Value:integer);
  begin
    SomeProp := Value;  // This causes infinite recursion }
  end;
```

Because you are accessing the property itself (not the internal storage field), you cause the `SetSomeProp()` method to be called again, which results in a recursive loop. Eventually, the program will crash with a stack overflow. Always access the internal storage field in the writer method of a property.

Adding Array Properties to Components

Some properties lend themselves to being accessed as though they were arrays. That is, they contain a list of items that can be referenced with an index value. The actual items referenced can be of any object type. Examples of such properties are `TScreen.Fonts`, `TMemo.Lines`, and `TDBGrid.Columns`. Such properties require their own property editors. We will get into creating property editors later in the next chapter. Therefore, we will not go into detail on creating array properties that have a list of different object types until later. For now, we'll show a simple method for defining a property that can be indexed as though it were an array of items, yet contains no list at all.

We're going to put aside the `TddgWorthless` component for a moment and instead look at the `TddgPlanets` component. `TddgPlanets` contains two properties: `PlanetName` and `PlanetPosition`. `PlanetName` will be an array property that returns the name of the planet based on the value of an integer index. `PlanetPosition` won't use an integer index, but rather a string index. If this string is one of the planet names, the result will be the planet's position in the solar system.

For example, the following statement will display the string `"Neptune"` by using the `TddgPlanets.PlanetName` property:

```
ShowMessage(ddgPlanets.PlanetName[8]);
```

Compare the difference when the sentence `From the sun, Neptune is planet number: 8` is generated from the following statement:

```
ShowMessage('From the sun, Neptune is planet number: '+
  IntToStr(ddgPlanets.PlanetPosition['Neptune']));
```

Before we show you this component, we list some key characteristics of array properties that differ from the other properties we've mentioned.

- Array properties are declared with one or more index parameters. These indexes can be of any simple type. For example, the index may be an integer or a string, but not a record or a class.
- Both the `read` and `write` property access directives must be methods. They cannot be one of the component's fields.

- If the array property is indexed by multiple index values, that is, the property represents a multidimensional array, the access method must include parameters for each index in the same order as defined by the property.

Now we'll get to the actual component shown in Listing 21.6.

LISTING 21.6 Using TddgPlanets to Illustrate Array Properties

```
unit planets;

interface

uses
  Classes, SysUtils;

type

  TddgPlanets = class(TComponent)
  private
    // Array property access methods
    function GetPlanetName(const AIndex: Integer): String;
    function GetPlanetPosition(const APlanetName: String): Integer;
  public
    { Array property indexed by an integer value. This will be the default
      array property.  }
    property PlanetName[const AIndex: Integer]: String
         read GetPlanetName; default;
    // Array property index by a string value
    property PlanetPosition[const APlanetName: String]: Integer
         read GetPlanetPosition;
  end;

implementation

const
  // Declare a constant array containing planet names
  PlanetNames: array[1..9] of String[7] =
   ('Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn',
    'Uranus', 'Neptune', 'Pluto');

function TddgPlanets.GetPlanetName(const AIndex: Integer): String;
begin
  { Return the name of the planet specified by Index. If Index is
    out of the range, then raise an exception }
  if (AIndex < 0) or (AIndex > 9) then
    raise Exception.Create('Wrong Planet number, enter a number 1-9')
```

```
  else
    Result := PlanetNames[AIndex];
end;

function TddgPlanets.GetPlanetPosition(const APlanetName: String): Integer;
var
  i: integer;
begin
  Result := 0;
  i := 0;
  { Compare PName to each planet name and return the index of the
    appropriate position where PName appears in the constant array.
    Otherwise return zero. }
  repeat
    inc(i);
  until (i = 10) or (CompareStr(UpperCase(APlanetName),
        UpperCase(PlanetNames[i])) = 0);

  if i <> 10 then // A Planet name was found
    Result := i;
end;

end.
```

This component gives you an idea of how you would create an array property with both an integer and a string being used as an index. Notice how the value returned from reading the property's value is based on the function return value and not a value from a storage field, as is the case with the other properties. You can refer to the code's comments for additional explanation on this component.

Default Values

You can give a property a default value by assigning a value to the property in the component's constructor. Therefore, if we added the following statement to the constructor of the `TddgWorthless` component, its `FIntegerProp` property would always default to 100 when the component is first placed onto the form:

```
FIntegerProp := 100;
```

This is probably the best place to mention the `Default` and `NoDefault` directives for property declarations. If you've looked at Delphi's VCL source code, you've probably noticed that some property declarations contain the `Default` directive, as is the case with the `TComponent.FTag` property:

```
property Tag: Longint read FTag write FTag default 0;
```

Don't confuse this statement with the default value specified in the component's constructor that actually sets the property value. For example, change the declaration of the `IntegerProp` property for the `TddgWorthless` component to read as follows:

```
property IntegerProp: Integer read FIntegerProp write FIntegerProp default 100;
```

This statement does not set the value of the property to `100`. This only affects whether or not the property value is saved when you save a form containing the `TddgWorthless` component. If `IntegerProp`'s value is not `100`, the value will be saved to the DFM file. Otherwise, it does not get saved (because 100 is what the property value will be in a newly constructed object prior to reading its properties from the stream). It is recommended that you use the `Default` directive whenever possible because it may speed up the load time of your forms. It is important that you realize that the `Default` directive does not set the value of the property. You must do that in the component's constructor, as was shown previously.

The `NoDefault` directive is used to re-declare a property that specifies a default value, so that it will always be written to the stream regardless of its value. For example, you can re-declare your component to not specify a default value for the `Tag` property:

```
TSample = class(TComponent)
published
  property Tag NoDefault;
```

Note that you should never declare anything `NoDefault` unless you have a specific reason. An example of such a property is `TForm.PixelsPerInch`, which must always be stored so that scaling will work right at runtime. Also, string, floating point, and `int64` type properties cannot declare default values.

To change a property's default value, you re-declare it by using the new default value (but no reader or writer methods).

Default Array Properties

You can declare an array property so that it is the default property for the component to which it belongs. This allows the component user to use the object instance as though it were an array variable. For example, using the `TddgPlanets` component, we declared the `TddgPlanets.PlanetName` property with the `default` keyword. By doing this, the component user is not required to use the property name, `PlanetName`, in order to retrieve a value. One simply has to place the index next to the object identifier. Therefore, the following two lines of code will produce the same result:

```
ShowMessage(ddgPlanets.PlanetName[8]);
ShowMessage(ddgPlanets[8]);
```

Only one default array property can be declared for an object, and it cannot be overridden in descendants.

Creating Events

In Chapter 20, we introduced events and told you that events were special properties linked to code that gets executed whenever a particular action occurs. In this section, we're going to discuss events in more detail. We'll show you how events are generated and how you can define your own event properties for your custom components.

Where Do Events Come From?

The general definition of an event is basically any type of occurrence that might result from user interaction, the system, or from code logic. The event is linked to some code that responds to that occurrence. The linkage of the event to code that responds to an event is called an *event property* and is provided in the form of a method pointer. The method to which an event property points is called an *event handler*.

For example, when the user clicks the mouse button, a `WM_MOUSEDOWN` message is sent to the Win32 system. Win32 passes that message to the control for which the message was intended. This control can then respond to the message. The control can respond to this event by first checking to see whether there is any code to execute. It does this by checking to see whether the event property points to any code. If so, it executes that code, or rather, the event handler.

The `OnClick` event is just one of the standard event properties defined by Delphi. `OnClick` and other event properties each have a corresponding *event-dispatching method*. This method is typically a protected method of the component to which it belongs. This method performs the logic to determine whether the event property refers to any code provided by the user of the component. For the `OnClick` property, this would be the `Click()` method. Both the `OnClick` property and the `Click()` method are defined by `TControl` as follows:

```
TControl = class(TComponent)
private
  FOnClick: TNotifyEvent;
protected
  procedure Click; dynamic;
  property OnClick: TNotifyEvent read FOnClick write FOnClick;
end;
```

Here is the `TControl.Click()` method:

```
procedure TControl.Click;
begin
  if Assigned(FOnClick) then FOnClick(Self);
end;
```

One bit of essential information that you must understand is that event properties are nothing more than method pointers. Notice that the `FOnClick` property is defined to be a `TNotifyEvent`. `TNotifyEvent` is defined as follows:

```
TNotifyEvent = procedure(Sender: TObject) of object;
```

This says that `TNotifyEvent` is a procedure that takes one parameter, `Sender`, which is of the type `TObject`. The directive, `of object`, is what makes this procedure become a method. This means that an additional *implicit* parameter that you do not see in the parameter list also gets passed to this procedure. This is the `Self` parameter that refers to the object to which this method belongs. When the `Click()` method of a component is called, it checks to see if `FOnClick` actually points to a method and, if so, calls that method.

As a component writer, you write all the code that defines your event, your event property, and your dispatching methods. The component user will provide the event handler when they use your component. Your event-dispatching method will check to see whether the user has assigned any code to your event property and then execute it when code exists.

In Chapter 20, we discussed how event handlers are assigned to event properties either at runtime or at design time. In the following section, we show you how to create your own events, event properties, and dispatching methods.

Defining Event Properties

Before you define an event property, you need to determine whether you need a special event type. It helps to be familiar with the common event properties that exist in the Delphi VCL. Most of the time, you'll be able to have your component descend from one of the existing components and just use its event properties, or you might have to surface a protected event property. If you determine that none of the existing events meet your needs, you can define your own.

As an example, consider the following scenario. Suppose you want a component that contains an event that gets called every half-minute based on the system clock. That is, it gets invoked on the minute and on the half-minute. Well, you can certainly use a `TTimer` component to check the system time and then perform some action whenever the time is at the minute or half-minute. However, you might want to incorporate this code into your own component and then make that component available to your users so that all they have to do is add code to your `OnHalfMinute` event.

The `TddgHalfMinute` component shown in Listing 21.7 illustrates how you would design such a component. More importantly, it shows how you would go about creating your own event type.

LISTING 21.7 `TddgHalfMinute` Event Creation

```
unit halfmin;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls;

type
  { Define a procedure for the event handler. The event property will
    be of this procedure type. This type will take two parameters, the
    object that invoked the event and a TDateTime value to represent
    the time that the event occurred. For our component this will be
    every half-minute. }
  TTimeEvent = procedure(Sender: TObject; TheTime: TDateTime) of object;

  TddgHalfMinute = class(TComponent)
  private
    FTimer: TTimer;
    { Define a storage field to point to the user's event handler.
      The user's event handler must be of the procedural type
      TTimeEvent. }
    FOnHalfMinute: TTimeEvent;
    FOldSecond, FSecond: Word; // Variables used in the code
    { Define a procedure, FTimerTimer that will be assigned to
      FTimer.OnClick. This procedure must be of the type TNotifyEvent
      which is the type of TTimer.OnClick. }
    procedure FTimerTimer(Sender: TObject);
  protected
    { Define the dispatching method for the OnHalfMinute event. }
    procedure DoHalfMinute(TheTime: TDateTime); dynamic;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    // Define the actual property that will show in the Object Inspector
    property OnHalfMinute: TTimeEvent read FOnHalfMinute write FOnHalfMinute;
  end;

implementation

constructor TddgHalfMinute.Create(AOwner: TComponent);
{ The Create constructor, creates the TTimer instanced for FTimer. It
  then sets up the various properties of FTimer, including its OnTimer
```

continues

LISTING 21.7 Continued

```
  event handler which is TddgHalfMinute's FTimerTimer() method. Notice
  that FTimer.Enabled is set to true only if the component is running
  and not while the component is in design mode. }
begin
  inherited Create(AOwner);
  // If the component is in design mode, do not enable FTimer.
  if not (csDesigning in ComponentState) then
  begin
    FTimer := TTimer.Create(self);
    FTimer.Enabled := True;
    // Set up the other properties, including the FTimer.OnTimer event handler
    FTimer.Interval := 500;
    FTimer.OnTimer := FTimerTimer;
   end;
end;

destructor TddgHalfMinute.Destroy;
begin
  FTimer.Free;
  inherited Destroy;
end;

procedure TddgHalfMinute.FTimerTimer(Sender: TObject);
{ This method serves as the FTimer.OnTimer event handler and is assigned
  to FTimer.OnTimer at run-time in TddgHalfMinute's constructor.

  This method gets the system time, and then determines whether or not
  the time is on the minute, or on the half-minute. If either of these
  conditions are true, it calls the OnHalfMinute dispatching method,
  DoHalfMinute. }
var
  DT: TDateTime;
  Temp: Word;
begin
  DT := Now; // Get the system time.
  FOldSecond := FSecond; // Save the old second.
  // Get the time values, needed is the second value
  DecodeTime(DT, Temp, Temp, FSecond, Temp);

  { If not the same second when this method was last called, and if
    it is a half minute, call DoOnHalfMinute. }
  if FSecond <> FOldSecond then
    if ((FSecond = 30) or (FSecond = 0)) then
      DoHalfMinute(DT)
```

```
end;

procedure TddgHalfMinute.DoHalfMinute(TheTime: TDateTime);
{ This method is the dispatching method for the OnHalfMinute event.
  it checks to see if the user of the component has attached an
  event handler to OnHalfMinute and if so, calls that code. }
begin
  if Assigned(FOnHalfMinute) then
    FOnHalfMinute(Self, TheTime);
end;

end.
```

When creating your own events, you must determine what information you want to provide to users of your component as a parameter in the event handler. For example, when you create an event handler for the `TEdit.OnKeyPress` event, your event handler looks like the following code:

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
end;
```

Not only do you get a reference to the object that caused the event, but you also get a `Char` parameter specifying the key that was pressed. Deep in the Delphi VCL, this event occurred as a result of a `WM_CHAR` Win32 message that drags along some additional information relating to the key pressed. Delphi takes care of extracting the necessary data and making it available to component users as event handler parameters. One of the nice things about the whole scheme is that it enables component writers to take information that might be somewhat complex to understand and make it available to component users in a much more understandable and easy-to-use format.

Notice the `var` parameter in the preceding `Edit1KeyPress()` method. You might be wondering why this method was not declared as a function that returns a `Char` type instead of a procedure. Although method types can be functions, you should not declare events as functions because it will introduce ambiguity; when you refer to a method pointer that is a function, you can't know whether you're referring to the function result or to the function pointer value itself. By the way, there is one function event in the VCL that slipped past the developers from the Delphi 1 days and now it must remain. This event is the `TApplication.OnHelp` event.

Looking at Listing 21.7, you'll see that we've defined the procedure type `TOnHalfMinute` as this:

```
TTimeEvent = procedure(Sender: TObject; TheTime: TDateTime) of object;
```

This procedure type defines the procedure type for the `OnHalfMinute` event handler. Here, we decided that we want the user to have a reference to the object causing the event to occur and the `TDateTime` value of when the event occurred.

The `FOnHalfMinute` storage field is the reference to the user's event handler and is surfaced to the Object Inspector at design time through the `OnHalfMinute` property.

The basic functionality of the component uses a `TTimer` object to check the seconds value every half second. If the seconds value is 0 or 30, it invokes the `DoHalfMinute()` method, which is responsible for checking for the existence of an event handler and then calling it. Much of this is explained in the code's comments, which you should read over.

After installing this component to Delphi's Component Palette, you can place the component on the form and add the following event handler to the `OnHalfMinute` event:

```
procedure TForm1.ddgHalfMinuteHalfMinute(Sender: TObject; TheTime: TDateTime);
begin
  ShowMessage('The Time is '+TimeToStr(TheTime));
end;
```

This should illustrate how your newly defined event type becomes an event handler.

Creating Methods

Adding methods to components is no different than adding methods to other objects. However, there are a few guidelines that you should always take into account when designing components.

No Interdependencies!

One of the key goals behind creating components is to simplify the use of the component for the end user. Therefore, you will want to avoid any method interdependencies as much as possible. For example, you never want to force the user to have to call a particular method in order to use the component, and methods should not have to be called in any particular order. Also, methods called by the user should not place the component in a state that makes other events or methods invalid. Finally, you will want to give your methods meaningful names so that the user does not have to try to guess what a method does.

Method Exposure

Part of designing a component is to know what methods to make private, public, or protected. You must take into account not only users of your component, but also those who might use your component as an ancestor for yet another custom component. Table 21.2 will help you decide what goes where in your custom component.

Table 21.2 Private, Protected, Public, or Published?

Directive	*What Goes There?*
Private	Instance variables and methods that you do not want the descendant type to be able to access or modify. Typically, you will give access to some private instance variables through properties that have `read` and `write` directives set in such a way as to help prevent the users from shooting themselves in the foot. Therefore, you want to avoid giving access to any methods that are property-implementation methods.
Protected	Instance variables, methods, and properties that you want descendant classes to be able to access and modify—but not users of your class. It is a common practice to place properties in the protected section of a base class for descendant classes to publish at their discretion.
Public	Methods and properties that you want to have accessible to any user of your class. If you have properties that you want to be accessible at runtime, but not at design time, this is the place to put them.
Published	Properties that you want to be placed on the Object Inspector at design time. *Runtime Type Information* (RTTI) is generated for all properties in this section.

Constructors and Destructors

When creating a new component, you have the option of overriding the ancestor component's constructor and defining your own. You should keep a few precautions in mind when doing so.

Overriding Constructors

Always make sure to include the `override` directive when declaring a constructor on a `TComponent` descendant class. Here's an example:

```
TSomeComponent = class(TComponent)
private
  { Private declarations }
protected
  { Protected declarations }
public
  constructor Create(AOwner: TComponent); override;
published
  { Published declarations }
end;
```

NOTE

The `Create()` constructor is made virtual at the `TComponent` level. Non-component classes have static constructors that are invoked from within the constructor of `TComponent` classes. Therefore, if you are creating a non-component, descendant class such as the following, the constructor cannot be overridden because it is not virtual:

```
TMyObject = class(TPersistent)
```

You simply re-declare the constructor in this instance.

Although not adding the override directive is syntactically legal, it can cause problems when using your component. This is because when you use the component (both at design time and at runtime), the non-virtual constructor won't be called by code that creates the component through a class reference (such as the streaming system).

Also, be sure that you call the inherited constructor inside your constructor's code:

```
constructor TSomeComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  // Place your code here.
end;
```

Design-Time Behavior

Remember that your component's constructor is called whenever the component is created. This includes the component's design-time creation—when you place it on the form. You might want to prevent certain actions from occurring when the component is being designed. For example, in the `TddgHalfMinute` component, you created a `TTimer` component inside the component's constructor. Although it doesn't hurt to do this, it can be avoided by making sure that the `TTimer` is only created at runtime.

You can check the `ComponentState` property of a component to determine its current state. Table 21.3 lists the various component states, as shown in Delphi 5's online help.

Table 21.3 Component State Values

Flag	*Component State*
`csAncestor`	Set if the component was introduced in an ancestor form. Only set if `csDesigning` is also set.
`csDesigning`	Design mode, meaning it is in a form being manipulated by a form designer.
`csDestroying`	The component is about to be destroyed.

Flag	*Component State*
`csFixups`	Set if the component is linked to a component in another form that has not yet been loaded. This flag is cleared when all pending fixups are resolved.
`csLoading`	Loading from a filer object.
`csReading`	Reading its property values from a stream.
`csUpdating`	The component is being updated to reflect changes in an ancestor form. Only set if `csAncestor` is also set.
`csWriting`	Writing its property values to a stream.

You will mostly use the `csDesigning` state to determine whether your component is in design mode. You can do this with the following statement:

```
inherited Create(AOwner);
if  csDesigning in ComponentState then
  { Do your stuff }
```

You should note that the `csDesigning` state is uncertain until after the inherited constructor has been called and the component is being created with an owner. This is almost always the case in the IDE form designer.

Overriding Destructors

The general guideline to follow when overriding destructors is to make sure you call the inherited destructor only after you free up resources allocated by your component, not before. The following code illustrates this:

```
destructor TMyComponent.Destroy;
begin
  FTimer.Free;
  MyStrings.Free;
  inherited Destroy;
end;
```

TIP

As a rule of thumb, when you override constructors, you usually call the inherited constructor first, and when you override destructors, you usually call the inherited destructor last. This ensures that the class has been set up before you modify it and that all dependent resources have been cleaned up before you dispose of a class.

There are exceptions to this rule, but you generally should stick with it unless you have good reason not to.

Registering Your Component

Registering the component tells Delphi which component to place on the Component Palette. If you used the Component Expert to design your component, you don't have to do anything here because Delphi has already generated the code for you. However, if you are creating your component manually, you'll need to add the `Register()` procedure to your component's unit.

All you have to do is add the procedure `Register()` to the `interface` section of the component's unit.

The `Register` procedure simply calls the `RegisterComponents()` procedure for every component that you are registering in Delphi. The `RegisterComponents()` procedure takes two parameters: the name of the page on which to place the components, and an array of component types. Listing 21.8 shows how to do this.

LISTING 21.8 Registering Components

```
Unit MyComp;
interface
type
  TMyComp = class(TComponent)
  ...
  end;
  TOtherComp = class(TComponent)
  ...
  end;
procedure Register;
implementation
{ TMyComp methods }
{ TOtherCompMethods }
procedure Register;
begin
  RegisterComponents('DDG', [TMyComp, TOtherComp]);
end;
end.
```

The preceding code registers the components `TMyComp` and `TOtherComp` and places them on Delphi's Component Palette on a page labeled DDG.

The Component Palette

In Delphi 1 and 2, Delphi maintained a single component library file that stored all components, icons, and editors for design-time usage. Although it was sometimes convenient to have everything dealing with design in one file, it could easily get unwieldy when many components were placed in the component library. Additionally, the more components you added to the palette, the longer it would take to rebuild the component library when adding new components.

Thanks to packages, introduced with Delphi 3, you can split up your components into several design packages. Although it's slightly more complex to deal with multiple files, this solution is significantly more configurable, and the time required to rebuild a package after adding a component is a fraction of the time it took to rebuild the component library.

By default, new components are added to a package called `DCLUSR50`, but you can create and install new design packages using the File, New, Package menu item. The CD accompanying this book contains a prebuilt design package called `DdgDsgn50.dpk` which includes the components from this book. The runtime package is named `DdgStd50.dpk`.

If your design-time support involves anything more than a call to `RegisterComponents()` (like property editors or component editors or expert registrations), you should move the `Register()` procedure and the stuff it registers into a separate unit from your component. The reason for this is that if you compile your all-in-one unit into a runtime package, and your all-in-one unit's `Register` procedure refers to classes or procedures that exist only in design-time IDE packages, your runtime package is unusable. Design-time support should be packaged separately from runtime material.

Testing the Component

Although it's very exciting when you finally write a component and are in the testing stages, don't get carried away by trying to add your component to the Component Palette before it has been debugged sufficiently. You should do all preliminary testing with your component by creating a project that creates and uses a dynamic instance of the component. The reason for this is that your component lives inside the IDE when it is used at design time. If your component contains a bug that corrupts memory, for example, it might crash the IDE as well. Listing 21.9 depicts a unit for testing the `TddgExtendedMemo` component that will be created later in this chapter. This project can be found on the CD in the project `TestEMem.dpr`.

LISTING 21.9 Testing the `TddgExtendedMemo` Component

```
unit MainFrm;

interface

uses
```

continues

LISTING 21.9 Continued

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, exmemo, ExtCtrls;

type

  TMainForm = class(TForm)
    btnCreateMemo: TButton;
    btnGetRowCol: TButton;
    btnSetRowCol: TButton;
    edtColumn: TEdit;
    edtRow: TEdit;
    Panel1: TPanel;
    procedure btnCreateMemoClick(Sender: TObject);
    procedure btnGetRowColClick(Sender: TObject);
    procedure btnSetRowColClick(Sender: TObject);
  public
    EMemo: TddgExtendedMemo;  // Declare the component.
    procedure OnScroll(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnCreateMemoClick(Sender: TObject);
begin
  { Dynamically create the component. Make sure to make the appropriate
    property assignments so that the component can be used normally.
    These assignments depend on the component being tested }
  if not Assigned(EMemo) then
  begin
    EMemo := TddgExtendedMemo.Create(self);
    EMemo.Parent := Panel1;
    EMemo.ScrollBars := ssBoth;
    EMemo.WordWrap := True;
    EMemo.Align := alClient;
    // Assign event handlers to untested events.
    EMemo.OnVScroll := OnScroll;
    EMemo.OnHScroll := OnScroll;
  end;
end;
```

```
{ Write whatever methods are required to test the run-time behavior
  of the component. This includes methods to access each of the
  new properties and methods belonging to the component.

  Also, create event handlers for user-defined events so that you can
  test them. Since you're creating the component at run-time, you
  have to manually assign the event handlers as was done in the
  above Create() constructor.
}
procedure TMainForm.btnGetRowColClick(Sender: TObject);
begin
  if Assigned(EMemo) then
    ShowMessage(Format('Row: %d  Column: %d', [EMemo.Row, EMemo.Column]));
  EMemo.SetFocus;
end;

procedure TMainForm.btnSetRowColClick(Sender: TObject);
begin
  if Assigned(EMemo) then
  begin
    EMemo.Row := StrToInt(edtRow.Text);
    EMemo.Column := StrToInt(edtColumn.Text);
    EMemo.SetFocus;
  end;
end;

procedure TMainForm.OnScroll(Sender: TObject);
begin
  MessageBeep(0);
end;

end.
```

Keep in mind that even testing the component at design time doesn't mean that your component is foolproof. Some design-time behavior can still raise havoc with the Delphi IDE, such as not calling the inherited `Create()` constructor.

NOTE

You cannot assume that your component has been created and set up by the design-time environment. Your component must be fully usable after only the `Create()` constructor has executed. Therefore, you should not treat the `Loaded()` method as part of the component construction process. The `Loaded()` method is called only when the

continues

component is loaded from a stream—such as when it is placed in a form built at design time. `Loaded()` marks the end of the streaming process. If your component was simply created (not streamed), `Loaded()` is not called.

Providing a Component Icon

No custom component would be complete without its own icon for the Component Palette. To create one of these icons, use Delphi's Image Editor (or your favorite bitmap editor) to create a 24×24 bitmap on which you will draw the component's icon. This bitmap must be stored within a DCR file. A file with a `.dcr` extension is nothing more than a renamed RES file. Therefore, if you store your icon in a RES file, you can simply rename it to a DCR file.

TIP

Even if you have a 256 or higher color driver, save your Component Palette icon as a 16-color bitmap if you plan on releasing the component to others. Your 256-color bitmaps most likely will look awful on machines running 16-color drivers.

After you create the bitmap in the DCR file, give the bitmap the same name as the class name of your component—in ALL CAPS. Save the resource file as the same name as your component's unit with a `.dcr` extension. Therefore, if your component is named `TXYZComponent`, the bitmap name is `TXYZCOMPONENT`. If the component's unit name is `XYZCOMP.PAS`, name the resource file `XYZCOMP.DCR`. Place this file in the same directory as the unit, and when you recompile the unit, the bitmap automatically is linked into the component library.

Sample Components

The remaining sections of this chapter give some real examples of component creation. The components created here serve two primary purposes. First, they illustrate the techniques explained in the first part of this chapter. Secondly, you can actually use these components in your applications. You might even decide to extend their functionality to meet your needs.

Extending Win32 Component Wrapper Capabilities

In some cases, you might want to extend the functionality of existing components, especially those components that wrap the Win32 control classes. We're going to show you how to do this by creating two components that extend the behavior of the `TMemo` control and the `TListBox` control.

TddgExtendedMemo: Extending the TMemo Component

Although the `TMemo` component is quite robust, there are a few features it doesn't make available that would be useful. For starters, it's not capable of providing the caret position in terms of the row and column on which the caret sits. We'll extend the `TMemo` component to provide these as public properties.

Additionally, it is sometimes convenient to perform some action whenever the user touches the `TMemo`'s scrollbars. You'll create events to which the user can attach code whenever these scrolling events occur.

The source code for the `TddgExtendedMemo` component is shown in Listing 21.10.

LISTING 21.10 `ExtMemo.pas`: The Source for the `TddgExtendedMemo` Component

```
unit ExMemo;

interface

uses
  Windows, Messages, Classes, StdCtrls;

type

  TddgExtendedMemo = class(TMemo)
  private
    FRow: Longint;
    FColumn: Longint;
    FOnHScroll: TNotifyEvent;
    FOnVScroll: TNotifyEvent;
    procedure WMHScroll(var Msg: TWMHScroll); message WM_HSCROLL;
    procedure WMVScroll(var Msg: TWMVScroll); message WM_VSCROLL;
    procedure SetRow(Value: Longint);
    procedure SetColumn(Value: Longint);
    function GetRow: Longint;
    function GetColumn: Longint;
  protected
    // Event dispatching methods
    procedure HScroll; dynamic;
    procedure VScroll; dynamic;
  public
    property Row: Longint read GetRow write SetRow;
    property Column: Longint read GetColumn write SetColumn;
  published
    property OnHScroll: TNotifyEvent read FOnHScroll write FOnHScroll;
    property OnVScroll: TNotifyEvent read FOnVScroll write FOnVScroll;
```

Listing 21.10 Continued

```
  end;


implementation

procedure TddgExtendedMemo.WMHScroll(var Msg: TWMHScroll);
begin
  inherited;
  HScroll;
end;

procedure TddgExtendedMemo.WMVScroll(var Msg: TWMVScroll);
begin
  inherited;
  VScroll;
end;

procedure TddgExtendedMemo.HScroll;
{ This is the OnHScroll event dispatch method. It checks to see
  if OnHScroll points to an event handler and calls it if it does. }
begin
  if Assigned(FOnHScroll) then
    FOnHScroll(self);
end;

procedure TddgExtendedMemo.VScroll;
{ This is the OnVScroll event dispatch method. It checks to see
  if OnVScroll points to an event handler and calls it if it does. }
begin
  if Assigned(FOnVScroll) then
    FOnVScroll(self);
end;

procedure TddgExtendedMemo.SetRow(Value: Longint);
{ The EM_LINEINDEX returns the character position of the first
  character in the line specified by wParam. The Value is used for
  wParam in this instance. Setting SelStart to this return value
  positions the caret on the line specified by Value. }
begin
  SelStart := Perform(EM_LINEINDEX, Value, 0);
  FRow := SelStart;
end;

function TddgExtendedMemo.GetRow: Longint;
{ The EM_LINEFROMCHAR returns the line in which the character specified
```

```
  by wParam sits. If -1 is passed as wParam, the line number at which
  the caret sits is returned. }
begin
  Result := Perform(EM_LINEFROMCHAR, -1, 0);
end;

procedure TddgExtendedMemo.SetColumn(Value: Longint);
begin
  { Get the length of the current line using the EM_LINELENGTH
    message. This message takes a character position as WParam.
    The length of the line in which that character sits is returned. }
  FColumn := Perform(EM_LINELENGTH, Perform(EM_LINEINDEX, GetRow, 0), 0);
  { If the FColumn is greater than the value passed in, then set
    FColumn to the value passed in }
  if FColumn > Value then
    FColumn := Value;
  // Now set SelStart to the newly specified position
  SelStart := Perform(EM_LINEINDEX, GetRow, 0) + FColumn;
end;

function TddgExtendedMemo.GetColumn: Longint;
begin
  { The EM_LINEINDEX message returns the line index of a specified
    character passed in as wParam. When wParam is -1 then it
    returns the index of the current line. Subtracting SelStart from this
    value returns the column position }
  Result := SelStart - Perform(EM_LINEINDEX, -1, 0);
end;

end.
```

First, we'll discuss adding the capability to provide row and column information to `TddgExtendedMemo`. Notice that we've added two private fields to the component, `FRow` and `FColumn`. These fields will hold the row and column of the `TddgExtendedMemo`'s caret position. Notice that we've also provided the `Row` and `Column` public properties. These properties are made public because there's really no use for them at design time. The `Row` and `Column` properties have both reader and writer access methods. For the `Row` property, these access methods are `GetRow()` and `SetRow()`. The `Column` access methods are `GetColumn()` and `SetColumn()`. For all practical purposes, you probably could do away with the `FRow` and `FColumn` storage fields because the values for `Row` and `Column` are provided through access methods. However, we've left them here because they offer the opportunity to extend this component.

The four access methods make use of various `EM_XXXX` messages. The code comments explain what is going on in each method and how these messages are used to provide `Row` and `Column` information for the component.

The TddgExtendedMemo component also provides two new events: OnHScroll and OnVScroll. The OnHScroll event occurs whenever the user clicks the horizontal scrollbar of the control. Likewise, the OnVScroll occurs when the user clicks the vertical scrollbar. To surface such events, you have to capture the WM_HSCROLL and WM_VSCROLL Win32 messages that are passed to the control whenever the user clicks either scrollbar. Thus, you've created the two message handlers: WMHScroll() and WMVScroll(). These two message handlers call the event-dispatching methods HScroll() and VScroll(). These methods are responsible for checking whether the component user has provided event handlers for the OnHScroll and OnVScroll events and then calling those event handlers. If you're wondering why we didn't just perform this check in the message handler methods, it's because you'll often want to be able to invoke an event handler as a result of a different action, such as when the user changes the caret position.

You can install and use the TddgExtendedMemo with your applications. You might even consider extending this component; for example, whenever the user changes the caret position, a WM_COMMAND message is sent to the control's owner. The HiWord(wParam) carries a notification code indicating the action that occurred. This code would have the value of EN_CHANGE, which stands for edit-notification message change. It is possible to have your component subclass its parent and capture this message in the parent's window procedure. It can then automatically update the FRow and FColumn fields. Subclassing is an altogether different and advanced topic that is discussed later.

TddgTabbedListBox: Extending the TListBox Component

VCL's TListbox component is merely an Object Pascal wrapper around the standard Win32 API LISTBOX control. Although it does do a fair job encapsulating most of that functionality, there is a little bit of room for improvement. This section takes you through the steps in creating a custom component based on TListbox.

The Idea

The idea for this component, like most, was born out of necessity. A list box was needed with the capability to use tab stops (which is supported in the Win32 API, but not in a TListbox), and a horizontal scrollbar was needed to view strings that were longer than the list box width (also supported by the API but not a TListbox). This component will be called a TddgTabListbox.

The plan for the TddgTabListbox component isn't terribly complex; we did this by creating a TListbox descendant component containing the correct field properties, overridden methods, and new methods to achieve the desired behavior.

The Code

The first step in creating a scrollable list box with tab stops is to include those window styles in the TddgTabListbox's style when the listbox window is created. The window styles needed are lbs_UseTabStops for tabs and ws_HScroll to allow a horizontal scrollbar. Whenever you add window styles to a descendant of TWinControl, do so by overriding the CreateParams() method, as shown in the following code:

```
procedure TddgTabListbox.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  Params.Style := Params.Style or lbs_UseTabStops or ws_HScroll;
end;
```

CreateParams()

Whenever you need to modify any of the parameters—such as the style or window class—that are passed to the CreateWindowEx() API function, you should do so in the CreateParams() method. CreateWindowEx() is the function used to create the window handle associated with a TWinControl descendant. By overriding CreateParams(), you can control the creation of a window on the API level.

CreateParams accepts one parameter of type TCreateParams, which follows:

```
TCreateParams = record
    Caption: PChar;
    Style: Longint;
    ExStyle: Longint;
    X, Y: Integer;
    Width, Height: Integer;
    WndParent: HWnd;
    Param: Pointer;
    WindowClass: TWndClass;
    WinClassName: array[0..63] of Char;
  end;
```

As a component writer, you will override CreateParams() frequently—whenever you need to control the creation of a component on the API level. Make sure that you call the inherited CreateParams() first in order to fill up the Params record for you.

To set the tab stops, the TddgTabListbox performs an lb_SetTabStops message, passing the number of tab stops and a pointer to an array of tabs as the wParam and lParam (these two variables will be stored in the class as FNumTabStops and FTabStops). The only catch is that listbox tab stops are handled in a unit of measure called *dialog box units*. Because dialog box units don't make sense for the Delphi programmer, you will surface tabs only in pixels. With the help of the PixDlg.pas unit shown in Listing 21.11, you can convert back and forth between dialog box units and screen pixels in both the X and Y planes.

LISTING 21.11 The Source Code for `PixDlg.pas`

```
unit Pixdlg;

interface

function DialogUnitsToPixelsX(DlgUnits: word): word;
function DialogUnitsToPixelsY(DlgUnits: word): word;
function PixelsToDialogUnitsX(PixUnits: word): word;
function PixelsToDialogUnitsY(PixUnits: word): word;

implementation
uses WinProcs;

function DialogUnitsToPixelsX(DlgUnits: word): word;
begin
  Result := (DlgUnits * LoWord(GetDialogBaseUnits)) div 4;
end;

function DialogUnitsToPixelsY(DlgUnits: word): word;
begin
  Result := (DlgUnits * HiWord(GetDialogBaseUnits)) div 8;
end;

function PixelsToDialogUnitsX(PixUnits: word): word;
begin
  Result := PixUnits * 4 div LoWord(GetDialogBaseUnits);
end;

function PixelsToDialogUnitsY(PixUnits: word): word;
begin
  Result := PixUnits * 8 div HiWord(GetDialogBaseUnits);
end;

end.
```

When you know the tab stops, you can calculate the extent of the horizontal scrollbar. The scrollbar should extend at least to the end of the longest string in the listbox. Luckily, the Win32 API provides a function called `GetTabbedTextExtent()` that retrieves just the information you need. When you know the length of the longest string, you can set the scrollbar range by performing the `lb_SetHorizontalExtent` message, passing the desired extent as the `wParam`.

You also need to write message handlers for some special Win32 messages. In particular, you need to handle the messages that control inserting and deleting, because you need to be able to measure the length of any new string or know when a long string has been deleted. The messages you're concerned with are `lb_AddString`, `lb_InsertString`, and `lb_DeleteString`. Listing 21.12 contains the source code for the `LbTab.pas` unit, which contains the `TddgTabListbox` component.

LISTING 21.12 `LbTab.pas`, the `TddgTabListBox`

```
unit Lbtab;

interface

uses
  SysUtils, Windows, Messages, Classes, Controls, StdCtrls;

type

  EddgTabListboxError = class(Exception);

  TddgTabListBox = class(TListBox)
  private
    FLongestString: Word;
    FNumTabStops: Word;
    FTabStops: PWord;
    FSizeAfterDel: Boolean;
    function GetLBStringLength(S: String): word;
    procedure FindLongestString;
    procedure SetScrollLength(S: String);
    procedure LBAddString(var Msg: TMessage); message lb_AddString;
    procedure LBInsertString(var Msg: TMessage); message lb_InsertString;
    procedure LBDeleteString(var Msg: TMessage); message lb_DeleteString;
  protected
    procedure CreateParams(var Params: TCreateParams); override;
  public
    constructor Create(AOwner: TComponent); override;
    procedure SetTabStops(A: array of word);
  published
    property SizeAfterDel: Boolean read FSizeAfterDel write FSizeAfterDel
➥default True;
  end;

implementation

uses PixDlg;

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

continues

LISTING 21.12 Continued

```
begin
  inherited Create(AOwner);
  FSizeAfterDel := True;
  { set tab stops to Windows defaults... }
  FNumTabStops := 1;
  GetMem(FTabStops, SizeOf(Word) * FNumTabStops);
  FTabStops^ := DialogUnitsToPixelsX(32);
end;

procedure TddgTabListBox.SetTabStops(A: array of word);
{ This procedure sets the listbox's tabstops to those specified
  in the open array of word, A.  New tabstops are in pixels, and must
  be in ascending order.  An exception will be raised if new tabs
  fail to set. }
var
  i: word;
  TempTab: word;
  TempBuf: PWord;
begin
  { Store new values in temps in case exception occurs in setting tabs }
  TempTab := High(A) + 1;      // Figure number of tabstops
  GetMem(TempBuf, SizeOf(A));  // Allocate new tabstops
  Move(A, TempBuf^, SizeOf(A));// copy new tabstops }
  { convert from pixels to dialog units, and... }
  for i := 0 to TempTab - 1 do
    A[i] := PixelsToDialogUnitsX(A[i]);
  { Send new tabstops to listbox.  Note that we must use dialog units. }
  if Perform(lb_SetTabStops, TempTab, Longint(@A)) = 0 then
  begin
    { if zero, then failed to set new tabstops, free temp
      tabstop buffer and raise an exception }
    FreeMem(TempBuf, SizeOf(Word) * TempTab);
    raise EddgTabListboxError.Create('Failed to set tabs.')
  end
  else begin
    { if nonzero, then new tabstops set okay, so
      Free previous tabstops }
    FreeMem(FTabStops, SizeOf(Word) * FNumTabStops);
    { copy values from temps... }
    FNumTabStops := TempTab;  // set number of tabstops
    FTabStops := TempBuf;     // set tabstop buffer
    FindLongestString;        // reset scrollbar
    Invalidate;               // repaint
  end;
```

```
end;

procedure TddgTabListBox.CreateParams(var Params: TCreateParams);
{ We must OR in the styles necessary for tabs and horizontal scrolling
  These styles will be used by the API CreateWindowEx() function. }
begin
  inherited CreateParams(Params);
  { lbs_UseTabStops style allows tabs in listbox
    ws_HScroll style allows horizontal scrollbar in listbox }
  Params.Style := Params.Style or lbs_UseTabStops or ws_HScroll;
end;

function TddgTabListBox.GetLBStringLength(S: String): word;
{ This function returns the length of the listbox string S in pixels }
var
  Size: Integer;
begin
  // Get the length of the text string
   Canvas.Font := Font;
  Result := LoWord(GetTabbedTextExtent(Canvas.Handle, PChar(S),
         StrLen(PChar(S)), FNumTabStops, FTabStops^));
  // Add a little bit of space to the end of the scrollbar extent for looks
  Size := Canvas.TextWidth('X');
  Inc(Result, Size);
end;

procedure TddgTabListBox.SetScrollLength(S: String);
{ This procedure resets the scrollbar extent if S is longer than the }
{ previous longest string                                            }
var
  Extent: Word;
begin
  Extent := GetLBStringLength(S);
  // If this turns out to be the longest string...
  if Extent > FLongestString then
  begin
    // reset longest string
    FLongestString := Extent;
    //reset scrollbar extent
    Perform(lb_SetHorizontalExtent, Extent, 0);
  end;
end;

procedure TddgTabListBox.LBInsertString(var Msg: TMessage);
{ This procedure is called in response to a lb_InsertString message.
  This message is sent to the listbox every time a string is inserted.
```

continues

LISTING 21.12 Continued

```
  Msg.lParam holds a pointer to the null-terminated string being
  inserted.  This will cause the scrollbar length to be adjusted if
  the new string is longer than any of the existing strings. }
begin
  inherited;
  SetScrollLength(PChar(Msg.lParam));
end;

procedure TddgTabListBox.LBAddString(var Msg: TMessage);
{ This procedure is called in response to a lb_AddString message.
  This message is sent to the listbox every time a string is added.
  Msg.lParam holds a pointer to the null-terminated string being
  added.  This Will cause the scrollbar length to be adjusted if the
  new string is longer than any of the existing strings.}
begin
  inherited;
  SetScrollLength(PChar(Msg.lParam));
end;

procedure TddgTabListBox.FindLongestString;
var
  i: word;
  Strg: String;
begin
  FLongestString := 0;
  { iterate through strings and look for new longest string }
  for i := 0 to Items.Count - 1 do
  begin
    Strg := Items[i];
    SetScrollLength(Strg);
  end;
end;

procedure TddgTabListBox.LBDeleteString(var Msg: TMessage);
{ This procedure is called in response to a lb_DeleteString message.
  This message is sent to the listbox everytime a string is deleted.
  Msg.wParam holds the index of the item being deleted.  Note that
  by setting the SizeAfterDel property to False, you can cause the
  scrollbar update to not occur.  This will improve performance
  if you're deleting often. }
var
  Str: String;
begin
  if FSizeAfterDel then
```

```
  begin
    Str := Items[Msg.wParam]; // Get string to be deleted
    inherited;                // Delete string
    { Is deleted string the longest? }
    if GetLBStringLength(Str) = FLongestString then
      FindLongestString;
  end
  else
    inherited;
end;

end.
```

One particular point of interest in this component is the `SetTabStops()` method, which accepts an open array of `word` as a parameter. This enables users to pass in as many tab stops as they want. Here is an example:

```
ddgTabListboxInstance.SetTabStops([50, 75, 150, 300]);
```

If the text in the listbox extends beyond the viewable window, the horizontal scrollbar will appear automatically.

TddgRunButton: Creating Properties

If you wanted to run another executable program in 16-bit Windows, you could use the `WinExec()` API function. Although these functions still work in Win32, it is not the recommended approach. Now you should use the `CreateProcess()` or `ShellExecute()` functions to launch another application. `CreateProcess()` can be a somewhat daunting task when needed just for that purpose. Therefore, we've provided the `ProcessExecute()` method, which we'll show in a moment.

To illustrate the use of `ProcessExecute()`, we've created the component `TddgRunButton`. All that is required of the user is to click the button and the application executes.

The `TddgRunButton` component is an ideal example of creating properties, validating property values, and encapsulating complex operations. Additionally, we'll show you how to grab the application icon from an executable file and how to display it in the `TddgRunButton` at design time. One other thing; `TddgRunButton` descends from `TSpeedButton`. Because `TSpeedButton` contains certain properties that you don't want accessible at design time through the Object Inspector, we'll show you how you can hide (sort of) existing properties from the component user. Admittedly, this technique is not exactly the cleanest approach to use. Typically, you would create a component of your own if you want to take the purist approach—of which the authors are advocates. However, this is one of those instances where Borland, in all its infinite wisdom, did not provide an intermediate component in between `TSpeedButton` and `TCustomControl` (from which `TSpeedButton` descends), as Borland did with its other components. Therefore, the choice was either to roll our own component that pretty much duplicates the functionality you get from `TSpeedButton`, or borrow from `TSpeedButton`'s functionality and hide a few properties that aren't applicable for your needs. We opted for the latter, but only out of necessity. However, this should clue you in to practice careful forethought as to how component writers might want to extend your own components.

The code to `TddgRunButton` is shown in Listing 21.13.

LISTING 21.13 `RunBtn.pas`, the Source to the `TddgRunButton` Component

```
{
Copyright © 1999 by Delphi 5 Developer's Guide - Xavier Pacheco and Steve
Teixeira
}

unit RunBtn;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons;

type

  TCommandLine = type string;

  TddgRunButton = class(TSpeedButton)
  private
    FCommandLine: TCommandLine;
    // Hiding Properties from the Object Inspector
    FCaption: TCaption;
    FAllowAllUp: Boolean;
    FFont: TFont;
    FGroupIndex: Integer;
    FLayOut: TButtonLayout;
    procedure SetCommandLine(Value: TCommandLine);
  public
    constructor Create(AOwner: TComponent); override;
    procedure Click; override;
  published
    property CommandLine: TCommandLine read FCommandLine write SetCommandLine;
    // Read only properties are hidden
```

```
    property Caption: TCaption read FCaption;
    property AllowAllUp: Boolean read FAllowAllUp;
    property Font: TFont read FFont;
    property GroupIndex: Integer read FGroupIndex;
    property LayOut: TButtonLayOut read FLayOut;
  end;

implementation

uses ShellAPI;

const
  EXEExtension = '.EXE';

function ProcessExecute(CommandLine: TCommandLine; cShow: Word): Integer;
{ This method encapsulates the call to CreateProcess() which creates
  a new process and its primary thread. This is the method used in
  Win32 to execute another application, This method requires the use
  of the TStartInfo and TProcessInformation structures. These structures
  are not documented as part of the Delphi 5 online help but rather
  the Win32 help as STARTUPINFO and PROCESS_INFORMATION.

  The CommandLine parameter specifies the pathname of the file to
  execute.

  The cShow parameter specifies one of the SW_XXXX constants which
  specifies how to display the window. This value is assigned to the
  sShowWindow field of the TStartupInfo structure. }
var
  Rslt: LongBool;
  StartUpInfo: TStartUpInfo;  // documented as STARTUPINFO
  ProcessInfo: TProcessInformation; // documented as PROCESS_INFORMATION
begin
  { Clear the StartupInfo structure }
  FillChar(StartupInfo, SizeOf(TStartupInfo), 0);
  { Initialize the StartupInfo structure with required data.
    Here, we assign the SW_XXXX constant to the wShowWindow field
    of StartupInfo. When specifying a value to this field the
    STARTF_USESSHOWWINDOW flag must be set in the dwFlags field.
    Additional information on the TStartupInfo is provided in the Win32
    online help under STARTUPINFO. }
  with StartupInfo do
  begin
    cb := SizeOf(TStartupInfo); // Specify size of structure
    dwFlags := STARTF_USESHOWWINDOW or STARTF_FORCEONFEEDBACK;
    wShowWindow := cShow
```

continues

LISTING 21.13 Continued

```
  end;

  { Create the process by calling CreateProcess(). This function
    fills the ProcessInfo structure with information about the new
    process and its primary thread. Detailed information is provided
    in the Win32 online help for the TProcessInfo structure under
    PROCESS_INFORMATION. }
  Rslt := CreateProcess(PChar(CommandLine), nil, nil, nil, False,
    NORMAL_PRIORITY_CLASS, nil, nil, StartupInfo, ProcessInfo);
  { If Rslt is true, then the CreateProcess call was successful.
    Otherwise, GetLastError will return an error code representing the
    error which occurred. }
  if Rslt then
    with ProcessInfo do
    begin
      { Wait until the process is in idle. }
      WaitForInputIdle(hProcess, INFINITE);
      CloseHandle(hThread); // Free the hThread  handle
      CloseHandle(hProcess);// Free the hProcess handle
      Result := 0;          // Set Result to 0, meaning successful
    end
  else Result := GetLastError; // Set result to the error code.
end;

function IsExecutableFile(Value: TCommandLine): Boolean;
{ This method returns whether or not the Value represents a valid
  executable file by ensuring that its file extension is 'EXE' }
var
  Ext: String[4];
begin
  Ext := ExtractFileExt(Value);
  Result := (UpperCase(Ext) = EXEExtension);
end;

constructor TddgRunButton.Create(AOwner: TComponent);
{ The constructor sets the default height and width properties
  to 45x45 }
begin
  inherited Create(AOwner);
  Height := 45;
  Width  := 45;
end;

procedure TddgRunButton.SetCommandLine(Value: TCommandLine);
```

```
{ This write access method sets the FCommandLine field to Value, but
  only if Value represents a valid executable file name. It also
  set the icon for the TddgRunButton to the application icon of the
  file specified by Value. }
var
  Icon: TIcon;
begin
  { First check to see that Value *is* an executable file and that
    it actually exists where specified. }
  if not IsExecutableFile(Value) then
    Raise Exception.Create(Value+' is not an executable file.');
  if not FileExists(Value) then
    Raise Exception.Create('The file: '+Value+' cannot be found.');

  FCommandLine := Value;  // Store the Value in FCommandLine

  { Now draw the application icon for the file specified by Value
    on the TddgRunButton icon. This requires us to create a TIcon
    instance to which to load the icon. It is then copied from this
    TIcon instance to the TddgRunButton's Canvas.

    We must use the Win32 API function ExtractIcon() to retrieve the
    icon for the application. }
  Icon := TIcon.Create; // Create the TIcon instance
  try
    { Retrieve the icon from the application's file }
    Icon.Handle := ExtractIcon(hInstance, PChar(FCommandLine), 0);
    with Glyph do
    begin
      { Set the TddgRunButton properties so that the icon held by Icon
        can be copied onto it. }
      { First, clear the canvas. This is required in case another
        icon was previously drawn on the canvas }
      Canvas.Brush.Style := bsSolid;
      Canvas.FillRect(Canvas.ClipRect);
      { Set the Icon's width and height }
      Width := Icon.Width;
      Height := Icon.Height;
      Canvas.Draw(0, 0, Icon); // Draw the icon to TddgRunButton's Canvas
    end;
  finally
    Icon.Free; // Free the TIcon instance.
  end;
end;

procedure TddgRunButton.Click;
```

continues

LISTING 21.13 Continued

```
var
  WERetVal: Word;
begin
  inherited Click; // Call the inherited Click method
  { Execute the ProcessExecute method and check it's return value.
    if the return value is <> 0 then raise an exception because
    an error occurred. The error code is shown in the exception }
  WERetVal := ProcessExecute(FCommandLine, sw_ShowNormal);
  if WERetVal <> 0 then begin
    raise Exception.Create('Error executing program. Error Code:; '+
          IntToStr(WERetVal));
  end;
end;

end.
```

`TddgRunButton` has one property, `CommandLine`, which is defined to be of the type `String`. The private storage field for `CommandLine` is `FCommandLine`.

TIP

It is worth discussing the special definition of `TCommandLine`. Here is the syntax used:

```
TCommandLine = type string;
```

By defining `TCommandLine` as such, you tell the compiler to treat `TCommandLine` as a unique type that is still compatible with other string types. The new type will get its own runtime type information and therefore can have its own property editor. This same technique can be used with other types as well. Here is an example:

```
TMySpecialInt = type Integer;
```

We will show you how we use this to create a property editor for the `CommandLine` property in the next chapter. We do not show you this technique in this chapter because creating property editors is an advanced topic that we want to talk about in more depth.

The write access method for `CommandLine` is `SetCommandLine()`. We've provided two helper functions: `IsExecutableFile()` and `ProcessExecute()`.

`IsExecutableFile()` is a function that determines whether a filename passed to it is an executable file based on the file's extension.

Creating and Executing a Process

`ProcessExecute()` is a function that encapsulates the `CreateProcess()` Win32 API function that enables you to launch another application. The application to launch is specified by the `CommandLine` parameter, which holds the filename path. The second parameter contains one of the `SW_XXXX` constants that indicates how the process's main windows are to be displayed. Table 21.4 lists the various `SW_XXXX` constants and their meanings, as explained in the online help.

Table 21.4 `SW_XXXX` Constants

`SW_XXXX` *Constant*	*Meaning*
`SW_HIDE`	Hides the window. Another window will become active.
`SW_MAXIMIZE`	Displays the window as maximized.
`SW_MINIMIZE`	Minimizes the window.
`SW_RESTORE`	Displays a window at its size before it was maximized/minimized.
`SW_SHOW`	Displays a window at its current size/position.
`SW_SHOWDEFAULT`	Shows a window at the state specified by the `TStartupInfo` structure passed to `CreateProcess()`.
`SW_SHOWMAXIMIZED`	Activates/displays the window as maximized.
`SW_SHOWMINIMIZED`	Activates/displays the window as minimized.
`SW_SHOWMINNOACTIVE`	Displays the window as minimized but the currently active window remains active.
`SW_SHOWNA`	Display the window at its current state. The currently active window remains active.
`SW_SHOWNOACTIVATE`	Displays the window at the most recent size/position. The currently active window remains active.
`SW_SHOWNORMAL`	Activates/displays the window at its more recent size/position. This position is restored if the window was previously maximized/minimized.

`ProcessExecute()` is a handy utility function that you might want to keep around in a separate unit that may be shared by other applications.

TddgRunButton Methods

The `TddgRunButton.Create()` constructor simply sets a default size for itself after calling the inherited constructor.

The `SetCommandLine()` method, which is the writer access method for the `CommandLine` parameter, performs several tasks. First, it determines whether the value being assigned to `CommandLine` is a valid executable filename. If not, it raises an exception.

If the entry is valid, it is assigned to the FCommandLine field. SetCommandLine() then extracts the icon from the application file and draws it to TddgRunButton's canvas. The Win32 API function ExtractIcon() is used to do this. The technique used is explained in the commentary.

TddgRunButton.Click() is the event-dispatching method for the TSpeedButton.OnClick event. It is necessary to call the inherited Click() method that will invoke the OnClick event handler if assigned. After calling the inherited Click(), you call ProcessExecute() and examine its result value to determine whether the call was successful. If not, an exception is raised.

TddgButtonEdit—Container Components

Occasionally you might like to create a component that is composed of one or more other components. Delphi's TDBNavigator is a good example of such a component, as it consists of a TPanel and a number of TSpeedButton components. Specifically, this section illustrates this concept by creating a component that is a combination of a TEdit and a TSpeedButton component. We will call this component TddgButtonEdit.

Design Decisions

Considering that Object Pascal is based upon a single-inheritance object model, TddgButtonEdit will need to be a component in its own right, which must contain both a TEdit1 and a TSpeedButton. Furthermore, because it's necessary that this component contain windowed controls, it will need to be a windowed control itself. For these reasons, we chose to descend TddgButtonEdit from TWinControl. We created both the TEdit and TSpeedButton in TddgButtonEdit's constructor using the following code:

```
constructor TddgButtonEdit.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FEdit        := TEdit.Create(Self);
  FEdit.Parent := self;
  FEdit.Height := 21;

  FSpeedButton := TSpeedButton.Create(Self);
  FSpeedButton.Left := FEdit.Width;
  FSpeedButton.Height := 19; // two less then TEdit's Height
  FSpeedButton.Width  := 19;
  FSpeedButton.Caption := '...';
  FSpeedButton.Parent := Self;

  Width  := FEdit.Width+FSpeedButton.Width;
  Height := FEdit.Height;
end;
```

The challenge when creating a component that contains other components is surfacing the properties of the "inner" components from the container component. For example, the `TddgButtonEdit` will need a `Text` property. You also might want to be able to change the font for the text in the control; therefore, a `Font` property is needed. Finally, there needs to be an `OnClick` event for the button in the control. You wouldn't want to attempt to implement this yourself in the container component when it is already available from the inner components. The goal, then, is to surface the appropriate properties of the inner controls without rewriting the interfaces to these controls.

Surfacing Properties

This usually boils down to the simple but time-consuming task of writing reader and writer methods for each of the inner component properties you want to resurface through the container component. In the case of the `Text` property, for example, you might give the `TddgButtonEdit` a `Text` property with read and write methods:

```
TddgButtonEdit = class(TWinControl)
private
  FEdit: TEdit;
  protected
  procedure SetText(Value: String);
    function  GetText: String;
published
    property Text: String read GetText write SetText;
end;
```

The `SetText()` and `GetText()` methods directly access the `Text` property of the contained `TEdit` control, as shown here:

```
function TddgButtonEdit.GetText: String;
begin
  Result := FEdit.Text;
end;

procedure TddgButtonEdit.SetText(Value: String);
begin
  FEdit.Text := Value;
end;
```

Surfacing Events

In addition to properties, it's also quite likely that you might want to resurface events that exist in the inner components. For example, when the user clicks on the `TSpeedButton` control, you would want to surface its `OnClick` event. Resurfacing events is just as straightforward as resurfacing properties—after all, events are properties.

You need to first give the `TddgButtonEdit` its own `OnClick` event. For clarity, we named this event `OnButtonClick`. The read and write methods for this event simply redirect the assignment to the `OnClick` event of the internal `TSpeedButton`.

Listing 21.14 shows the `TddgButtonEdit` container component.

LISTING 21.14 `TddgButtonEdit`, a Container Component

```
unit ButtonEdit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons;

type
  TddgButtonEdit = class(TWinControl)
  private
    FSpeedButton: TSpeedButton;
    FEdit: TEdit;
  protected
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
    procedure SetText(Value: String);
    function  GetText: String;
    function GetFont: TFont;
    procedure SetFont(Value: TFont);
    function GetOnButtonClick: TNotifyEvent;
    procedure SetOnButtonClick(Value: TNotifyEvent);
  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;
  published
    property Text: String read GetText write SetText;
    property Font: TFont read GetFont write SetFont;
    property OnButtonClick: TNotifyEvent read GetOnButtonClick
        write SetOnButtonClick;
  end;

implementation

procedure TddgButtonEdit.WMSize(var Message: TWMSize);
begin
  inherited;
  FEdit.Width := Message.Width-FSpeedButton.Width;
  FSpeedButton.Left := FEdit.Width;
```

```
end;

constructor TddgButtonEdit.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FEdit        := TEdit.Create(Self);
  FEdit.Parent := self;
  FEdit.Height := 21;

  FSpeedButton := TSpeedButton.Create(Self);
  FSpeedButton.Left := FEdit.Width;
  FSpeedButton.Height := 19; // two less than TEdit's Height
  FSpeedButton.Width  := 19;
  FSpeedButton.Caption := '...';
  FSpeedButton.Parent := Self;

  Width  := FEdit.Width+FSpeedButton.Width;
  Height := FEdit.Height;
end;

destructor  TddgButtonEdit.Destroy;
begin
  FSpeedButton.Free;
  FEdit.Free;
  inherited Destroy;
end;

function TddgButtonEdit.GetText: String;
begin
  Result := FEdit.Text;
end;

procedure TddgButtonEdit.SetText(Value: String);
begin
  FEdit.Text := Value;
end;

function TddgButtonEdit.GetFont: TFont;
begin
  Result := FEdit.Font;
end;

procedure TddgButtonEdit.SetFont(Value: TFont);
begin
  if Assigned(FEdit.Font) then
    FEdit.Font.Assign(Value);
```

continues

LISTING 21.14 Continued

```
end;

function TddgButtonEdit.GetOnButtonClick: TNotifyEvent;
begin
  Result := FSpeedButton.OnClick;
end;

procedure TddgButtonEdit.SetOnButtonClick(Value: TNotifyEvent);
begin
  FSpeedButton.OnClick := Value;
end;

end.
```

TddgDigitalClock—Creating Component Events

`TddgDigitalClock` illustrates the process of creating and making available user-defined events. We will use the same technique discussed earlier when we discussed creating events with the `TddgHalfMinute` component.

`TddgDigitalClock` descends from `TPanel`. We decided that `TPanel` was an ideal component from which `TddgDigitalClock` could descend because `TPanel` has the `BevelXXXX` properties. This enables you to give the `TddgDigitalClock` a pleasing visual appearance. Also, you can use the `TPanel.Caption` property to display the system time.

`TddgDigitalClock` contains the following events to which the user can assign code:

`OnHour`	Occurs on the hour, every hour.
`OnHalfPast`	Occurs on the half-hour.
`OnMinute`	Occurs on the minute.
`OnHalfMinute`	Occurs every 30 seconds: on the minute and on the half minute.
`OnSecond`	Occurs on the second.

`TddgDigitalClock` uses a `TTimer` component internally. Its `OnTimer` event handler performs the logic to display the time information and to invoke the event-dispatching methods for the previously listed events accordingly. Listing 21.15 shows the source code for `DdgClock.pas`.

LISTING 21.15 `DdgClock.pas`: Source for the `TddgDigitalClock` Component

```
{
Copyright © 1999 by Delphi 5 Developer's Guide - Xavier Pacheco and Steve
Teixeira
}
```

```
{$IFDEF VER110}
{$OBJEXPORTALL ON}
{$ENDIF}

unit DDGClock;

interface

uses
  Windows, Messages, Controls, Forms, SysUtils, Classes, ExtCtrls;

type

  { Declare an event type which takes the sender of the event, and
    a TDateTime variable as parameters }
  TTimeEvent = procedure(Sender: TObject; DDGTime: TDateTime) of object;

  TddgDigitalClock = class(TPanel)
  private
    { Data fields }
    FHour,
    FMinute,
    FSecond: Word;
    FDateTime: TDateTime;
    FOldMinute,
    FOldSecond: Word;
    FTimer: TTimer;
    { Event handlers }
    FOnHour: TTimeEvent;       // Occurs on the hour
    FOnHalfPast: TTimeEvent;   // Occurs every half-hour
    FOnMinute: TTimeEvent;     // Occurs on the minute
    FOnSecond: TTimeEvent;     // Occurs every second
    FOnHalfMinute: TTimeEvent; // Occurs every 30 seconds
    { Define OnTimer event handler for internal TTimer, FTimer }
    procedure TimerProc(Sender: TObject);
  protected
    { Override the Paint methods }
    procedure Paint; override;

    { Define the various event dispatching methods }
    procedure DoHour(Tm: TDateTime); dynamic;
    procedure DoHalfPast(Tm: TDateTime); dynamic;
    procedure DoMinute(Tm: TDateTime); dynamic;
    procedure DoHalfMinute(Tm: TDateTime); dynamic;
    procedure DoSecond(Tm: TDateTime); dynamic;
```

continues

Listing 21.15 Continued

```
  public
    { Override the Create constructor and Destroy destructor }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    { Define event properties }
    property OnHour: TTimeEvent read FOnHour write FOnHour;
    property OnHalfPast: TTimeEvent read FOnHalfPast write FOnHalfPast;
    property OnMinute: TTimeEvent read FOnMinute write FOnMinute;
    property OnHalfMinute: TTimeEvent read FOnHalfMinute
             write FOnHalfMinute;
    property OnSecond: TTimeEvent read FOnSecond write FOnSecond;
  end;

implementation

constructor TddgDigitalClock.Create(AOwner: TComponent);
begin
  inherited Create(AOwner); // Call the inherited constructor
  Height := 25; // Set default width and height properties
  Width := 120;
  BevelInner := bvLowered; // Set Default bevel properties
  BevelOuter := bvLowered;
  { Set the inherited Caption property to an empty string }
  inherited Caption := '';
  { Create the TTimer instance and set both its Interval property and
    OnTime event handler. }
  FTimer:= TTimer.Create(self);
  FTimer.interval:= 200;
  FTimer.OnTimer:= TimerProc;
end;


destructor TddgDigitalClock.Destroy;
begin
  FTimer.Free; // Free the TTimer instance.
  inherited Destroy; // Call inherited Destroy method
end;

procedure TddgDigitalClock.Paint;
begin
  inherited Paint; // Call the inherited Paint method
  { Now set the inherited Caption property to current time. }
  inherited Caption := TimeToStr(FDateTime);
```

```
end;

procedure TddgDigitalClock.TimerProc(Sender: TObject);
var
  HSec: Word;
begin
  { Save the old minute and second for later use }
  FOldMinute := FMinute;
  FOldSecond := FSecond;
  FDateTime := Now; // Get the current time.
  { Extract the individual time elements }
  DecodeTime(FDateTime, FHour, FMinute, FSecond, Hsec);

  refresh; // Redraw the component so that the new time is displayed.

  { Now call the event handlers depending on the time }
  if FMinute = 0 then
    DoHour(FDateTime);
  if FMinute = 30 then
    DoHalfPast(FDateTime);
  if (FMinute <> FOldMinute) then
    DoMinute(FDateTime);
  if FSecond <> FOldSecond then
    if ((FSecond = 30) or (FSecond = 0)) then
      DoHalfMinute(FDateTime)
    else
       DoSecond(FDateTime);
end;

{ The event dispatching methods below determine if component user has
  attached event handlers to the various clock events and calls them
  if they exist }
procedure TddgDigitalClock.DoHour(Tm: TDateTime);
begin
  if Assigned(FOnHour) then
   TTimeEvent(FOnHour)(Self, Tm);
end;

procedure TddgDigitalClock.DoHalfPast(Tm: TDateTime);
begin
  if Assigned(FOnHalfPast) then
   TTimeEvent(FOnHalfPast)(Self, Tm);
end;

procedure TddgDigitalClock.DoMinute(Tm: TDateTime);
```

continues

Listing 21.15 Continued

```
begin
  if Assigned(FOnMinute) then
   TTimeEvent(FOnMinute)(Self, Tm);
end;

procedure TddgDigitalClock.DoHalfMinute(Tm: TDateTime);
begin
  if Assigned(FOnHalfMinute) then
   TTimeEvent(FOnHalfMinute)(Self, Tm);
end;

procedure TddgDigitalClock.DoSecond(Tm: TDateTime);
begin
  if Assigned(FOnSecond) then
   TTimeEvent(FOnSecond)(Self, Tm);
end;

end.
```

The logic behind this component is explained in the source commentary. The methods used are no different than those that were previously explained when we discussed creating events. `TddgDigitalClock` only adds more events and contains logic to determine when each event is invoked.

Adding Forms to the Component Palette

Adding forms to the Object Repository is a convenient way to give forms a starting point. But what if you develop a form that you reuse often that does not need to be inherited and does not require added functionality? Delphi 5 provides a way you can reuse your forms as components on the Component Palette. In fact, the `TFontDialog` and `TOpenDialog` components are examples of forms that are accessible from the Component Palette. Actually, these dialogs are not Delphi forms; these are dialogs provided by the `CommDlg.dll`. Nevertheless, the concept is the same.

To add forms to the Component Palette, you must wrap your form with a component to make it a separate, installable component. The process as described here uses a simple password dialog whose functionality will verify your password automatically. Although this is a very simple project, the purpose of this discussion is not to show you how to install a complex dialog as a component, but rather to show you the general method for adding dialog boxes to the Component Palette. The same method applies to dialog boxes of any complexity.

First, you must create the form that is going to be wrapped by the component. The form we used is defined in the file `PwDlg.pas`. This unit also shows a component wrapper for this form.

Listing 21.16 shows the unit defining the `TPasswordDlg` form and its wrapper component, `TddgPasswordDialog`.

LISTING 21.16 `PwDlg.pas`—`TPasswordDlg` Form and Its Component Wrapper `TddgPasswordDialog`

```
unit PwDlg;

interface

uses Windows, SysUtils, Classes, Graphics, Forms, Controls, StdCtrls,
  Buttons;

type

  TPasswordDlg = class(TForm)
    Label1: TLabel;
    Password: TEdit;
    OKBtn: TButton;
    CancelBtn: TButton;
  end;

  { Now declare the wrapper component. }
  TddgPasswordDialog = class(TComponent)
  private
    PassWordDlg: TPasswordDlg; // TPassWordDlg instance
    FPassWord: String;         // Place holder for the password
  public
    function Execute: Boolean; // Function to launch the dialog
  published
    property PassWord: String read FPassword write FPassword;
  end;

implementation
{$R *.DFM}

function TddgPasswordDialog.Execute: Boolean;
begin
  { Create a TPasswordDlg instance }
  PasswordDlg := TPasswordDlg.Create(Application);
  try
    Result := False;  // Initialize the result to false
```

continues

LISTING 21.16 Continued

```
    { Show the dialog and return true if the password
      is correct. }
    if PasswordDlg.ShowModal = mrOk then
      Result := PasswordDlg.Password.Text = FPassword;
  finally
    PasswordDlg.Free;  // Free instance of PasswordDlg
  end;
end;

end.
```

The `TddgPasswordDialog` is called a *wrapper* component because it wraps the form with a component that can be installed into Delphi 5's Component Palette.

`TddgPasswordDialog` descends directly from `TComponent`. You might recall from the last chapter that `TComponent` is the lowest-level class that can be manipulated by the Form Designer in the IDE. This class has two `private` variables: `PasswordDlg` of type `TPasswordDlg` and `FPassWord` of type `string`. `PasswordDlg` is the `TPasswordDlg` instance that this wrapper component displays. `FPassWord` is an *internal storage field* that holds a password string.

`FPassWord` gets its data through the property `PassWord`. Thus, `PassWord` doesn't actually store data; rather, it serves as an interface to the storage variable `FPassWord`.

`TddgPassWordDialog`'s `Execute()` function creates a `TPasswordDlg` instance and displays it as a modal dialog box. When the dialog box terminates, the string entered in the password `TEdit` control is compared against the string stored in `FPassword`.

The code here is contained within a `try..finally` construct. The `finally` portion ensures that the `TPasswordDlg` component is disposed of, regardless of any error that might occur.

After you have added `TddgPasswordDialog` to the Component Palette, you can create a project that uses it. As with any other component, you select `TddgPasswordDialog` from the Component Palette and place it on your form. The project created in the preceding section contains a `TddgPasswordDialog` and one button whose `OnClick` event handler does the following:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if ddgPasswordDialog.Execute then        // Launch the PasswordDialog
    ShowMessage('You got it!')            // Correct password
  else
    ShowMessage('Sorry, wrong answer!'); // Incorrect password
end;
```

The Object Inspector contains three properties for the `TddgPasswordDialog` component: `Name`, `Password`, and `Tag`. To use the component, you must set the `Password` property to some string value. When you run the project, `TddgPasswordDialog` prompts the user for a password and compares it against the password you entered for the `Password` property.

Component Packages

Delphi 3 introduced *packages*, which enable you to place portions of your application into separate modules that can be shared across multiple applications. Packages are similar to dynamic link libraries (DLLs) but differ in their usage. Packages are primarily used to store collections of components in a separate, shareable module (a Borland Package Library, or `.bpl` file). As you or other developers create Delphi applications, the packages you create can be used by the application at runtime instead of being directly linked at compile/link time. Because the code for these units resides in the `.bpl` file, rather than in your `.exe` or `.dll`, the size of your `.exe` or `.dll` can become very small.

Packages differ from DLLs in that they are specific to Delphi VCL; that is, applications written in other languages can't use packages created by Delphi (with the exception of CBuilder). One of the reasons behind packages is to get around a limitation of Delphi 1 and 2. In these prior versions of Delphi, the VCL added a minimum of 150KB to 200KB of code to every executable. Therefore, even if you were to separate a piece of your application into a DLL, both the DLL and the application would contain redundant code. This is especially a problem if you are providing a suite of applications on one machine. Packages enable you to reduce the footprint of your applications and provide a convenient way for you to distribute your component collections.

Why Use Packages?

There are several reasons why you might want to use packages. Three are discussed in the following sections.

Code Reduction

A primary reason behind using packages is to reduce the size of your applications and DLLs. Delphi already ships with several predefined packages that break up the VCL into logical groupings. In fact, you can choose to compile your application so that it assumes the existence of many of these Delphi packages.

A Smaller Distribution of Applications—Application Partitioning

You'll find that many applications are available over the Internet as full-blown applications, downloadable demos, or updates to existing applications. Consider the benefit of giving users the option of downloading smaller versions of the application when pieces of the application might already exist on their system, such as when they have a prior installation.

By partitioning your applications using packages, you also allow your users to obtain updates to only those parts of the application that they need. Note, however, that there are some versioning issues that you'll have to take into account. We'll cover versioning issues momentarily.

Component Containment

Probably one of the most common reasons for using packages is the distribution of third-party components. If you are a component vendor, you must know how to create packages. The reason for this is that certain design-time elements—such as component and property editors, wizards, and experts—are all provided by packages.

Why Not to Use Packages

You shouldn't use runtime packages unless you are sure that other applications will be using these packages. Otherwise, these packages will end up using more disk space than if you were just compiling the source code into your final executable. Why is this so? If you create a packaged application resulting in a code reduction from 200KB to roughly 30KB, it might seem like you've saved quite a bit of space. However, you still have to distribute your packages and possibly even the `Vcl50.dcp` package, which is roughly 2MB in size. You can see that this isn't quite the saving you had hoped for. Our point is that you should use packages to share code when that code will be used by multiple executables. Note that this only applies to runtime packages. If you are a component writer, you must provide a design package that contains the component you want to make available to the Delphi IDE.

Types of Packages

There are four types of packages available for you to create and use:

- Runtime package. Runtime packages contain code, components, and so on needed by an application at runtime. If you write an application that depends on a particular runtime package, the application won't run in the absence of that package.
- Design package. Design packages contain components, property/component editors, experts, and so on necessary for application design in the Delphi IDE. This type of package is used only by Delphi and is never distributed with your applications.
- Runtime and design package. A package that is both design- and runtime-enabled is typically used when there are no design-specific elements such as property/component editors and experts. You can create this type of package to simplify application development and deployment. However, if this package does contain design elements, its runtime use will carry the extra baggage of the design support in your deployed applications. We

recommend creating both a design and runtime package to separate design-specific elements when they are present.

- Neither runtime nor design package. This rare breed of package is intended to be used only by other packages and is not intended to be referenced directly by an application or used in the design environment. This implies that packages can use or include other packages.

Package Files

Table 21.5 lists and describes the package-specific files based on their file extensions.

Table 21.5 Package Files

File Extension	*File Type*	*Description*
`.dpk`	Package source file	This file is created when you invoke the Package Editor. You can think of this as you might think of the `.dpr` file for a Delphi project.
`.dcp`	Runtime/design package symbol file	This is the compiled version of the package that contains the symbol information for the package and its units. Additionally, there is header information required by the Delphi IDE.
`.dcu`	Compiled unit	A compiled version of a unit contained in a package. One `.dcu` file will be created for each unit contained in the package.
`.bpl`	Runtime/design package library	This is the runtime or design package, equivalent to a Windows DLL. If this is a runtime package, you will distribute the file along with your applications (if they are enabled for runtime packages). If this file represents a design package, you will distribute it along with its runtime partner to programmers that will use it to write programs. Note that if you aren't distributing source code, you must distribute the corresponding `.dcp` files.

Package-Enable Your Delphi 5 Applications

Package-enabling your Delphi applications is easy. Simply check the Build with Runtime Packages check box found in the Project, Options dialog on the Packages page. The next time you build your application after this option is selected, your application will be linked dynamically to runtime packages, instead of having units linked statically into your `.exe` or `.dll`. The result will be a much more svelte application (although bear in mind that you will have to deploy the necessary packages with your application).

Installing Packages into Delphi's IDE

Installing packages into the Delphi IDE is simple. You might need to do this if you obtain a third-party set of components. First, however, you need to place the package files in their appropriate location. Table 21.6 shows where package files are typically located.

Table 21.6 Package File Locations

Package File	*Location*
Runtime packages (`*.bpl`)	Runtime package files should be placed in the `\Windows\System\` directory (Windows 95/98) or `\WinNT\System32\` directory (Windows NT).
Design packages (`*.bpl`)	Because it is possible that you will obtain several packages from various vendors, design packages should be placed in a common directory where they can be properly managed. For example, create a `\PKG` directory off your `\Delphi 5\` directory and place design packages in that location.
Package symbol files (`*.dcp`)	You can place package symbol files in the same location as design package files (`*.bpl`).
Compiled units (`*.dcu`)	You must distribute compiled units if you are distributing design packages. We recommend keeping DCUs from third-party vendors in a directory similar to the `\Delphi 5\Lib` directory. For example, you can create the directory `\Delphi 5\3PrtyLib` in which third-party components' `*.dcu`s will reside. Your search path will have to point to this directory.

To install a package, you simply invoke the Packages page of the Project Options dialog by selecting Component, Install Packages from the Delphi 5 menu.

By selecting the Add button, you can select the specific `.bpl` file. Upon doing so, this file will become the selected file on the Project page. When you click OK, the new package is installed into the Delphi IDE. If this package contains components, you will see the new Component page on the Component Palette along with any newly installed components.

Designing Your Own Packages

Before creating a new package, you'll need to decide on a few things. First, you need to know what type of package you're going to create (runtime, design, and so on). This will be based on one or more of the scenarios that we present momentarily. Second, you need to know what you intend on naming your newly created package and where you want to store the package project. Keep in mind that the directory where your deployed package exists will probably not be the same as where you create your package. Finally, you need to know which units your package will contain and which other packages your new package will require.

The Package Editor

Packages are most commonly created using the Package Editor, which you invoke by selecting the Packages icon from the New Items dialog. (Select File, New from the Delphi main menu.) You'll notice that the Package Editor contains two folders: Contains and Requires.

The Contains Folder

In the Contains folder, you specify units that need to be compiled into your new package. There are a few rules for placing units into the Contains page of a package:

- The package must not be listed in the `contains` clause of another package or in the `uses` clause of a unit within another package.
- The units listed in the `contains` clause of a package, either directly or indirectly (they exist in `uses` clauses of units listed in the package's `contains` clause), cannot be listed in the package's `requires` clause. This is because these units are already bound to the package when it is compiled.
- You cannot list a unit in a package's `contains` clause if it is already listed in the `contains` clause of another package used by the same application.

The Requires Page

In the Requires page, you specify other packages that are required by the new package. This is similar to the `uses` clause of a Delphi unit. In most cases, any packages you create will have `VCL50`—the package that hosts Delphi's standard VCL components—in its `requires` clause. The typical arrangement here, for example, is that you place all your components into a runtime package. Then you create a design package that includes the runtime package in its `requires` clause. There are a few rules for placing packages on the Requires page of another package:

- Avoid circular references: `Package1` cannot have `Package1` in its `requires` clause, nor can it contain another package that has `Package1` in its `requires` clause.
- The chain of references must not refer back to a package previously referenced in the chain.

The Package Editor has a toolbar and context-sensitive menus. Refer to the Delphi 5 online help under "Package Editor" for an explanation of what these buttons do. We won't repeat that information here.

Package Design Scenarios

Earlier we said that you must know what type of package you want to create based on a particular scenario. In this section, we're going to present three possible scenarios in which you would use design and/or runtime packages.

Scenario 1—Design and Runtime Packages for Components

The design and runtime packages for components scenario is the case if you are a component writer and one or both of the following conditions apply:

- You want Delphi programmers to be able to compile/link your components right into their applications or to distribute them separately along with their applications.
- You have a component package, and you don't want to force your users to have to compile design features (component/property editors and so on) into their application code.

Given this scenario, you would create both a design and runtime package. Figure 21.4 depicts this arrangement. As the figure illustrates, the design package `(DDGDsgn50.dpk)` encompasses both the design features (property and component editors) and the runtime package (`DDGStd50.dpk`). The runtime package (`DDGStd50.dpk`) includes only your components. This arrangement is accomplished by listing the runtime package in the `requires` section of the design package, as shown in Figure 21.4.

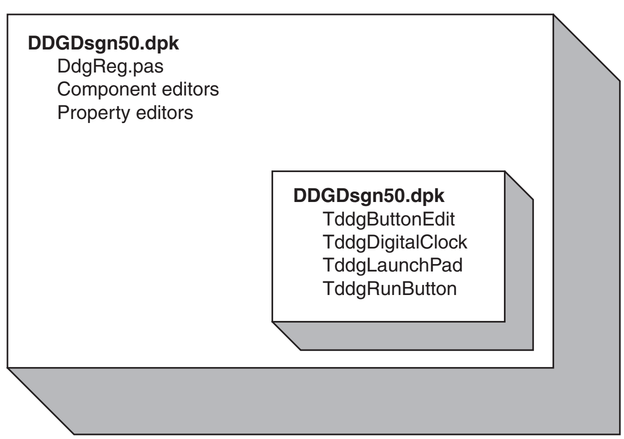

FIGURE 21.4
Design packages host design elements and runtime packages.

You must also apply the appropriate usage options for each package before compiling that package. You do this from the Package Options dialog. (You access the Package Options dialog by right-clicking within the Package Editor to invoke the local menu. Select Options to get to the dialog.) For the runtime package, `DDGStd50.dpk`, the usage option should be set to Runtime Only. This ensures that the package cannot be installed in the IDE as a design package (see the sidebar "Component Security" later in this chapter). For the design package, `DDGDsgn50.dpk`, the usage option Design Time Only should be selected. This enables users to install the package into the Delphi IDE, yet prevents them from using the package as a runtime package.

Adding the runtime package to the design package doesn't make the components contained in the runtime package available to the Delphi IDE yet. You must still register your components with the IDE. As you already know, whenever you create a component, Delphi automatically inserts a `Register()` procedure into the component unit, which in turn calls the `RegisterComponents()` procedure. `RegisterComponents()` is the procedure that actually registers your component with the Delphi IDE when you install the component. When working with packages, the recommended approach is to move the `Register()` procedure from the component unit into a separate registration unit. This registration unit registers all your components by calling `RegisterComponents()`. This not only makes it easier for you to manage the registration of your components, but it also prevents anyone from being able to install and use your runtime package illegally because the components won't be available to the Delphi IDE.

As an example, the components used in this book are hosted by the runtime package `DDGStd50.dpk`. The property editors, component editors, and registration unit (`DdgReg.pas`) for our components exist in the design package `DDGDsgn50.dpk`. `DDGDsgn50.dpk` also includes `DDGStd50.dpk` in its `requires` clause. Listing 21.17 shows what our registration unit looks like.

LISTING 21.17 Registration Unit for *Delphi 5 Developer's Guide* Components

```
unit DDGReg;

interface

procedure Register;

implementation

uses Classes, ExptIntf, DsgnIntf, TrayIcon, AppBars, ABExpt, Worthless,
  RunBtn, PwDlg, Planets, LbTab, HalfMin, DDGClock, ExMemo, MemView,
  Marquee, PlanetPE, RunBtnPE, CompEdit, DefProp, Wavez,
  WavezEd, LnchPad, LPadPE, Cards, ButtonEdit, Planet, DrwPnel;

procedure Register;
begin

  // Register the components.
```

continues

LISTING 21.17 Continued

```
  RegisterComponents('DDG',
  [ TddgTrayNotifyIcon, TddgDigitalClock, TddgHalfMinute, tddgButtonEdit,
    TddgExtendedMemo, TddgTabListbox, TddgRunButton, TddgLaunchPad,
    TddgMemView, TddgMarquee, TddgWaveFile, TddgCard, TddgPasswordDialog,
    TddgPlanet, TddgPlanets, TddgWorthLess, TddgDrawPanel,
    TComponentEditorSample, TDefinePropTest]);


  // Register any property editors.
  RegisterPropertyEditor(TypeInfo(TRunButtons), TddgLaunchPad, '',
    TRunButtonsProperty);
  RegisterPropertyEditor(TypeInfo(TWaveFileString), TddgWaveFile, 'WaveName',
    TWaveFileStringProperty);
  RegisterComponentEditor(TddgWaveFile, TWaveEditor);
  RegisterComponentEditor(TComponentEditorSample, TSampleEditor);
  RegisterPropertyEditor(TypeInfo(TPlanetName), TddgPlanet,
   'PlanetName', TPlanetNameProperty);
  RegisterPropertyEditor(TypeInfo(TCommandLine), TddgRunButton, '',
    TCommandLineProperty);

  // Register any custom modules, library experts.
  RegisterCustomModule(TAppBar, TCustomModule);
  RegisterLibraryExpert(TAppBarExpert.Create);

end;

end.
```

Component Security

It is possible for someone to register your components, even though he has only your runtime package. He would do this by creating his own registration unit in which he would register your components. He would then add this unit to a separate package that would also have your runtime package in the `requires` clause. After he installs this new package into the Delphi IDE, your components will appear on the Component Palette. However, it is still not possible to compile any applications using your components because the required `*.dcu` files for your component units will be missing.

Package Distribution

When distributing your packages to component writers without the source code, you must distribute both compiled packages, `DDGDsgn50.bpl` and `DDGStd50.bpl`, both `*.dcp` files, and any compiled units (`*.dcu`) necessary to compile your components. Programmers using your components who want their applications' runtime packages enabled must distribute the `DDGStd50.bpl` package along with their applications and any other runtime package that they might be using.

Scenario 2—Design Package Only for Components

The design package only for components scenario is when you want to distribute components that you don't want to be distributed in runtime packages. In this case, you will include the components, component editors, property editors, component registration unit, and so on in one package file.

Package Distribution

When distributing your package to component writers without the source code, you must distribute the compiled package, `DDGDsgn50.bpl`, the `DDGDsgn50.dcp` file, and any compiled units (`*.dcu`) necessary to compile your components. Programmers using your components must compile your components into their applications. They will not be distributing any of your components as runtime packages.

Scenario 3—Design Features Only (No Components) IDE Enhancements

The design features only (no components) IDE enhancements scenario is the case if you are providing enhancements to the Delphi IDE, such as experts. For this scenario, you will register your expert with the IDE in your registration unit. The distribution for this scenario is simple; you only have to distribute the compiled `*.bpl` file.

Scenario 4—Application Partitioning

The application partitioning scenario is the case if you want to partition your application into logical pieces, each of which can be distributed separately. There are several reasons why you might want to do this:

- This scenario is easier to maintain.
- Users can purchase only the needed functionality when they need it. Later, when they need added functionality, they can download the necessary package only, which will be much smaller than downloading the entire application.
- You can provide fixes (patches) to parts of the application more easily without requiring users to obtain a new version of the application altogether.

In this scenario, you will provide only the `*.bpl` files required by your application. This scenario is similar to the last with the difference being that instead of providing a package for the Delphi IDE, you will be providing a package for your own application. When partitioning your applications as such, you must pay attention to the issues regarding package versioning that we discuss in the next section.

Package Versioning

Package versioning is a topic that is not well understood. You can think of package versioning in much the same way as you think of unit versioning. That is, any package that you provide for your application must be compiled using the same Delphi version used to compile the application. Therefore, you cannot provide a package written in Delphi 5 to be used by an application written in Delphi 4. The Inprise developers refer to the version of a package as a *code base*. So a package written in Delphi 5 has a code base of 5.0. This concept should influence the naming convention that you use for your package files.

Package Compiler Directives

There are some specific compiler directives that you can insert into the source code of your packages. Some of these directives are specific to units that are being packaged; others are specific to the package file. These directives are listed and described in Tables 21.7 and 21.8.

Table 21.7 Compiler Directives for Units Being Packaged

Directive	*Meaning*
`{$G}` or `{IMPORTEDDATA OFF}`	Use this when you want to prevent the unit from being packaged—when you want it to be linked directly to the application. Contrast this to the `{$WEAKPACKAGEUNIT}` directive, which allows a unit to be included in a package but whose code gets statically linked to the application.
`{$DENYPACKAGEUNIT}`	Same as `{$G}`.
`{$WEAKPACKAGEUNIT}`	See the section "More on the `{$WEAKPACKAGEUNIT}` Directive."

Table 21.8 Compiler Directives for the Package `.dpk` File

Directive	*Meaning*
`{$DESIGNONLY ON}`	Compiles package as a design-time only package.
`{$RUNONLY ON}`	Compiles package as a runtime only package.
`{$IMPLICITBUILD OFF}`	Prevents the package from being rebuilt later. Use this option when the package is not changed frequently.

More on the {$WEAKPACKAGEUNIT} Directive

The concept of a weak package is simple. Basically, it is used where your package may be referencing libraries (DLLs) that may not be present. For example, `Vcl40` makes calls to the core

Win32 API included with the Windows operating system. Many of these calls exist in DLLs that aren't present on every machine. These calls are exposed by units that contain the `{$WEAKPACKAGEUNIT}` directive. By including this directive, you keep the unit's source code in the package but place it into the DCP file, rather than in the BPL file (think of a DCP as a DCU and a BPL as a DLL). Therefore, any references to functions of these weakly packaged units get statically linked to the application, rather than dynamically referenced through the package.

The `{$WEAKPACKAGEUNIT}` directive is one that you will rarely use, if at all. It was created out of necessity by the Delphi developers to handle a specific situation. The problem exists if there are two components, each in a separate package that reference the same interface unit of a DLL. When an application uses both of the components, this causes two instances of the DLL to be loaded, which raises havoc with initialization and global variable referencing. The solution is to include the interface unit in one of the standard Delphi packages, such as `Vcl50.bpl`. However, this raises the other problem for specialized DLLs that may not be present, such as `PENWIN.DLL`. If `Vcl50.bpl` contains the interface unit for a DLL that isn't present, it will render `Vcl50.bpl`, and Delphi for that matter, unusable. The Delphi developers addressed this by allowing `Vcl50.bpl` to contain the interface unit in a single package, but to make it statically linked when used and not dynamically loaded whenever `Vcl50` is used with the Delphi IDE.

As stated, you'll most likely never have to use this directive, unless you anticipate a similar scenario that the Delphi developers faced or if you want to make certain that a particular unit is included with a package but statically linked to the using application. A reason for the latter might be for optimization purposes. Note that any units that are weakly packaged cannot have global variables or code in their initialization/finalization sections. You must also distribute any `*.dcu` files for weakly packaged units along with your packages.

Package-Naming Conventions

Earlier we said that the package-versioning issue should influence how you name your packages. There isn't a set rule as to how you name your packages, but we suggest using a naming convention that incorporates the code base into the package's name. For example, the components for this book are contained in a runtime package whose name contains the `50` qualifier for Delphi 5 (`DDGStd50.dpk`). The same goes for the design package (`DDGDsgn50.dpk`). A previous version of the package would be `DdgStd40.dpk`. By using such a convention, you will prevent any confusion for your package users as to which version of the package they have and as to which version of the Delphi compiler applies to them. Note that our package name starts with a three-character author/company identifier, followed by `Std` to indicate a runtime package and `Dsgn` to signify a design package. You can follow whatever naming convention you like. Just be consistent and use the recommended inclusion of the Delphi version into your package name.

Add-In Packages

Add-in packages allow you to partition your applications into pieces or modules and to distribute those modules separately from the main application. This scheme is especially attractive because it allows you to extend the functionality of your application without having to recompile/redesign the entire application. This requires careful architectural design planning, however. It is beyond the scope of this book to go into such design issues. For a more detailed discussion of add-in packages and how they relate to application frameworks and design patterns, you will find articles at `http://www.xapware.com`.

Our example is a simple illustration of this technique. We will show how to add a form to an application without having to rewrite the application entirely. You can obtain a more elaborate example from the URL mentioned in the preceding paragraph.

Generating Add-In Forms

In Chapter 4, "Application Frameworks and Design Concepts," you learned about application frameworks. We developed an application whose forms were descendants of a base class (`TChildForm`). We'll use this same application to illustrate how you can create a shell application that knows only of the `TChildForm` class but can work with any descendent of that class. The descendants will be provided in thorough add-in packages.

NOTE

If you installed the forms used in the application framework demo from Chapter 4 to your Object Repository, you will have to remove them from the Repository before loading the project from this application.

The application is partitioned into three logical pieces: the main application (`ChildTest.exe`), the `TChildForm` package (`AIChildForm50.bpl`), and the concrete `TChildForm` descendant classes, each residing in its own package.

The main application is basically the same as that from Chapter 4 with some modification. The package `AIChildForm50.bpl` contains the *abstract* `TChildForm` class. The other packages contain descendant `TChildForm` classes or *concrete* `TChildForms`. We will refer to these packages as the *abstract package* and *concrete packages,* respectively.

The main application uses the abstract package (`AIChildForm50.bpl`). Each concrete package also uses the abstract package. For this to work properly, the main application must be compiled with runtime packages, including the `AIChildForm50.dcp` package. Likewise, each concrete package must require the `AIChildForm50.dcp` package. We will not list the `TChildForm` source or the concrete descendants to `TChildForm`, because they are not much different from those shown in Chapter 4. The only difference is that each `TChildForm` descendant unit must include `initialization` and `finalization` blocks that look like this:

```
initialization
  RegisterClass(TCF2Form);
finalization
  UnRegisterClass(TCF2Form);
```

The call to `RegisterClass()` is necessary to make the `TChildForm` descendant class available to the main application's streaming system when the main application loads its package. This is similar to how `RegisterComponents()` makes components available to the Delphi IDE. When the package is unloaded, the call to `UnRegisterClass()` is required to remove the registered class. Note that `RegisterClass()` only makes the class available to the main application, however. The main application still does not know of the class name. So how does the main application create an instance of a class whose class name is unknown? Isn't the intent of this exercise to make these forms available to the main application without having to hard code their class names into the main application's source? Listing 21.18 shows the source code to the main application's main form where we will highlight how we accomplish add-in forms with add-in packages.

LISTING 21.18 The Main Form of the Main Application Using Add-In Packages

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, ChildFrm, Menus;

const
  { Child form registration location in the Windows Registry. }
  cCFRegLocation = 'Software\Delphi 5 Developer''s Guide';
  cCFRegSection  = 'ChildForms';  // Module initialization data section

  FMainCaption   = 'Delphi 5 Developer''s Guide Child Form Demo';

type

  TChildFormClass = class of TChildForm;

  TMainForm = class(TForm)
    pnlMain: TPanel;
```

continues

LISTING 21.18 Continued

```
    Splitter1: TSplitter;
    pnlParent: TPanel;
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    mmiHelp: TMenuItem;
    mmiForms: TMenuItem;
    procedure mmiExitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    // reference to the child form.
    FChildForm: TChildForm;
    // a list of available child forms used to build a menu.
    FChildFormList: TStringList;
    // Index to the Close Form menu which shifts position.
    FCloseFormIndex: Integer;
    // Handle to the currently loaded package.
    FCurrentModuleHandle: HModule;
    // method to create menus for available child forms.
    procedure CreateChildFormMenus;
    // Handler to load a child form and its package.
    procedure LoadChildFormOnClick(Sender: TObject);
    // Handler to unload a child form and its package.
    procedure CloseFormOnClick(Sender: TObject);
    // Method to retrieve the classname for a TChildForm descendant
    function GetChildFormClassName(const AModuleName: String): String;
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation
uses Registry;

{$R *.DFM}

function RemoveExt(const AFileName: String): String;
{ Helper function to remove the extension from a filename. }
begin
  if Pos('.', AFileName) <> 0 then
    Result := Copy(AFileName, 1, Pos('.', AFileName)-1)
```

```
  else
    Result := AFileName;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  FChildFormList := TStringList.Create;
  CreateChildFormMenus;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  FChildFormList.Free;
  // Unload any loaded child forms.
  if FCurrentModuleHandle <> 0 then
    CloseFormOnClick(nil);
end;

procedure TMainForm.CreateChildFormMenus;
{ All available child forms are registered in the Windows Registry.
  Here, we use this information to create menu items for loading each of the
  child forms. }
var
  IniFile: TRegIniFile;
  MenuItem: TMenuItem;
  i: integer;
begin
  inherited;

  { Retrieve a list of all child forms and build a menu based on the
    entries in the registry. }
  IniFile := TRegIniFile.Create(cCFRegLocation);
  try
    IniFile.ReadSectionValues(cCFRegSection, FChildFormList);
  finally
    IniFile.Free;
  end;

  { Add Menu items for each module. NOTE THE mmMain.AutoHotKeys property must
    be set to maAutomatic }
```

continues

LISTING 21.18 Continued

```
  for i := 0 to FChildFormList.Count - 1 do
  begin
    MenuItem := TMenuItem.Create(mmMain);
    MenuItem.Caption := FChildFormList.Names[i];
    MenuItem.OnClick := LoadChildFormOnClick;
    mmiForms.Add(MenuItem);
  end;

  // Create Separator
  MenuItem := TMenuItem.Create(mmMain);
  MenuItem.Caption := '-';
  mmiForms.Add(MenuItem);

  // Create Close Module menu item
  MenuItem := TMenuItem.Create(mmMain);
  MenuItem.Caption := '&Close Form';
  MenuItem.OnClick := CloseFormOnClick;
  MenuItem.Enabled := False;
  mmiForms.Add(MenuItem);

  { Save a reference to the index of the menu item required to
    close a child form. This will be referred to in another method. }
  FCloseFormIndex := MenuItem.MenuIndex;
end;

procedure TMainForm.LoadChildFormOnClick(Sender: TObject);
var
  ChildFormClassName: String;
  ChildFormClass: TChildFormClass;
  ChildFormName: String;
  ChildFormPackage: String;
begin

  // The menu caption represents the module name.
  ChildFormName := (Sender as TMenuItem).Caption;
  // Get the actual Package filename.
  ChildFormPackage := FChildFormList.Values[ChildFormName];

  // Unload any previously loaded packages.
  if FCurrentModuleHandle <> 0 then
    CloseFormOnClick(nil);

  try
    // Load the specified package
```

```
    FCurrentModuleHandle := LoadPackage(ChildFormPackage);

    // Return the classname that needs to be created
    ChildFormClassName := GetChildFormClassName(ChildFormPackage);

    { Create an instance of the class using the FindClass() procedure. Note,
      this requires that the class already be registered with the streaming
      system using RegisterClass(). This is done in the child form
      initialization section for each child form package. }
    ChildFormClass := TChildFormClass(FindClass(ChildFormClassName));
    FChildForm := ChildFormClass.Create(self, pnlParent);
    Caption := FChildForm.GetCaption;
    FChildForm.Show;

    mmiForms[FCloseFormIndex].Enabled := True;
  except
    on E: Exception do
    begin
      CloseFormOnClick(nil);
      raise;
    end;
  end;
end;

function TMainForm.GetChildFormClassName(const AModuleName: String): String;
{ The Actual class name of the TChildForm implementation resides in the
  registry. This method retrieves that class name. }
var
  IniFile: TRegIniFile;
begin
  IniFile := TRegIniFile.Create(cCFRegLocation);
  try
    Result := IniFile.ReadString(RemoveExt(AModuleName), 'ClassName',
      EmptyStr);
  finally
    IniFile.Free;
  end;
end;

procedure TMainForm.CloseFormOnClick(Sender: TObject);
begin
  if FCurrentModuleHandle <> 0 then
  begin
    if FChildForm <> nil then
    begin
      FChildForm.Free;
```

continues

LISTING 21.18 Continued

```
      FChildForm := nil;
    end;

    // Unregister any classes provided by the module
    UnRegisterModuleClasses(FCurrentModuleHandle);
    // Unload the child form package
    UnloadPackage(FCurrentModuleHandle);

    FCurrentModuleHandle := 0;
    mmiForms[FCloseFormIndex].Enabled := False;
    Caption := FMainCaption;
  end;
end;

end.
```

The application's logic is actually very simple. It uses the system registry to determine which packages are available, the menu captions to use when building menus for loading each package, and the class name of the form contained in each package.

NOTE

We've included a file called `D5DG.Reg` on which you can double-click in Windows Explorer. This imports the registry settings in order for the add-in package demo to run properly.

The `LoadChildFormOnClick()` event handler is where most of the work is performed. After determining the package filename, the method loads the package using the `LoadPackage()` function. The `LoadPackage()` function is basically the same thing as `LoadLibrary()` for DLLs. The method then determines the class name for the form contained in the loaded package.

To create a class, you need a class reference such as `TButton` or `TForm1`. However, this main application does not have the hard-coded class name of the concrete `TChildForms`. This is why we retrieve the class name from the system registry. The main application can pass this class name to the `FindClass()` function to return a class reference for the specified class that already has been registered with the streaming system. Remember that we did this in the initialization section of the concrete form's unit, which is called when the package is loaded. We then create the class with these lines:

```
ChildFormClass := TChildFormClass(FindClass(ChildFormClassName));
FChildForm := ChildFormClass.Create(self, pnlParent);
```

The variable `ChildFormClass` is a predeclared class reference to `TChildForm` and can refer to a class reference for a `TChildForm` descendant.

The `CloseFormOnClick()` event handler simply closes the child form and unloads its package. The rest of the code basically is set up to create the package menus and to read the information from the system registry.

Further study on this technique will enable you to create very extensible and loosely coupled application frameworks.

Summary

Knowing how components work is fundamental to understanding Delphi, and you will work with many more custom components later in the book. Now that you can see what happens behind the scenes, components will no longer be such a mystery. The next chapter goes beyond component creation into more advanced component-building techniques.