Building TSmiley

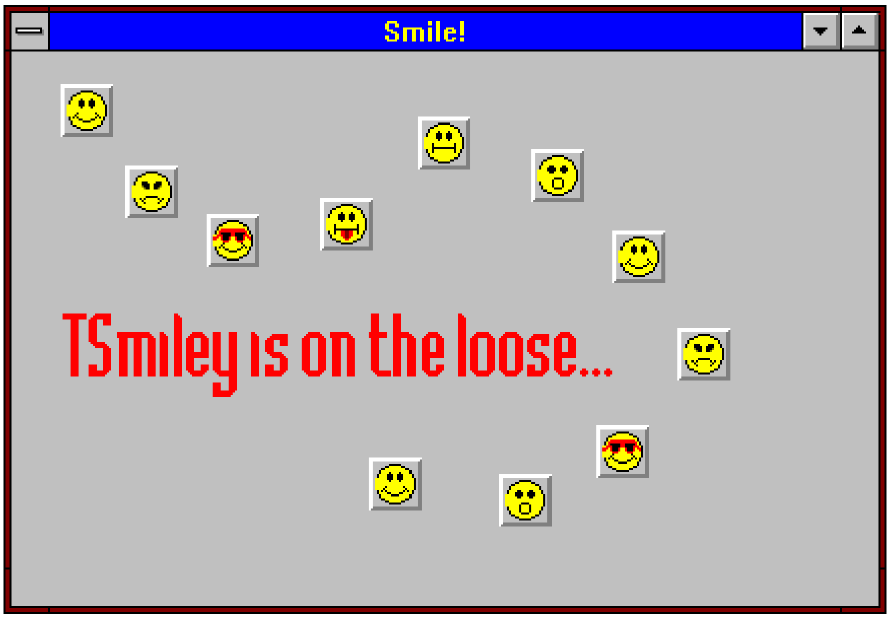
Get started with component building and have fun too!

by Nick Hodges

Delphi breaks new ground in many areas. One area that certainly attracts a lot of attention from developers is the ability to build, quickly and easily, native components. Bob Swart has been going into some detail about component building in his *Under Construction* column, but perhaps you've not been following it from the start and feel the need for a refresher on the basics of component construction. This article is for you!

I have built a simple Delphi component which demonstrates just how easy the process is. My little `Smiley` component changes faces from happy to sad to indifferent, among others, depending upon the whim and mood of the programmer. `Smiley` also doesn't like getting clicked on, demonstrating how easy it is to override default event handlers.

The resulting code also shows a number of techniques, including the use of Run Time Typing Information (RTTI) and properties, as well as the new event model itself. In addition, I built a basic Property Editor to make it easier for users to choose `Smiley`'s mood.



➤ *This example shows just how indispensable TSmiley is!*

Building The Framework

Though Delphi itself is a visual development tool, component building is done the old fashioned way – largely by typing code. Delphi does provide the user with a tool to build the basic framework for the component, but from there on code generation is up to the programmer. Selecting `File | New Component` brings up a dialog box which asks for a new component name, its base class and the palette page to which the component should be added. A parent class for the new component can be selected from any component currently registered with Delphi. Clicking the `Ok` button brings up the code editor with the skeleton code for the new component.

Initialization

Delphi's new object model uses a slightly different convention for initializing and destroying objects compared to OWL. Rather than the traditional `Init`, Object Pascal uses `Create` as the default constructor name. Another change is the fact that all classes in Delphi descend from the `TObject` class, so every constructor called `Create` will be overriding an ancestor method.

`Smiley`'s creation is declared as shown in `constructor TSmiley.Create` in Listing 1 (which shows all the code for the Smiley component, which is also included on the disk of course).

Delphi makes extended use of enumerated types, so in keeping with that, I created an enumerated type, `TMood`, to define the various moods that `Smiley` could take on. Since I will use the good old API call `LoadBitmap` to get the various faces out of a resource file, I also need an array of `PChar`s to use in passing the bitmap name to the API call. `Smiley` initializes itself with the `smHappy` face, but the user can easily change this with the Object Inspector. Once the `fMood` property is set, `Smiley` calls the `LoadBitmap` API call to get the selected bitmap from the resource file.

Note also that the initialization of the `TBitmap` type differs from OWL as well. All Delphi objects are created on the heap and are automatically de-referenced. Since virtually everything points to the heap, there is no need to de-reference with the good old ^ any more as Delphi does it 'automagically' at compile time. Dedicated OWL programmers will have to break the old habit of initializing objects with `MyNewObject.Create;` and type

```
MyNewObject :=
  TNewObject.Create;
```

Properties

Object Pascal's new class model includes a powerful new feature: properties. Properties allow a programmer to create variables that appear to the component user as nothing more than ordinary variable fields. To the component builder, however, they combine both data and methods. They also have all the advantages but none of the drawbacks of private data. The data is protected from meddling inheritors, but is still accessible to less interloping ones. Very often a change in a variable will require a change in any number of other variables or program states, and properties allow for changing these variables with a simple assignment statement by the user.

For example, `Smiley` has a single simple property called `Mood`, which illustrates the value of properties. `Mood` can take on any of the `TMood` values, but of course each time `Mood` changes, the bitmap has to change as well. Properties allow the programmer to do all of the work behind the scenes, while on the surface the component user sees a simple variable change. If the program, either at run-time or design time, changes the value of the property `Mood`, Smiley calls the `SetMood` function automatically, which in turn performs the necessary work to change the bitmap to the required face. `Mood` is declared:

```
property Mood: TMood
  read FMood write SetMood;
```

Properties can be accessed by two different means. If the act of reading the current value of the property requires no further processing or calculation, the "reader" simply provides the current value of the private field (Delphi follows the convention of adding the letter F to the beginning of `private` property values). This is shown above, as the value of `Mood` is simply accessed through the `private` variable `FMood`. If there were a need to process or change something whenever a call was made for the current value of the property, then a function could be declared which takes no parameter and returns the

➤ *Listing 1*

```
unit Smiley;
{$R Smiley.res}
interface
uses
  WinProcs, Classes, Graphics, Controls, StdCtrls,
  Messages, ExtCtrls;
procedure Register;
type
  TMood = (smHappy, smSad, smShades, smTongue,
           smIndifferent, smOoh);
const
   MoodString : array[tMood] of PChar =
     ('smHappy', 'smSad', 'smShades', 'smTongue',
      'smIndifferent', 'smOoh');
   MaxHeight = 26;
   MaxWidth = 26;
type
  TSmiley = class(TImage)
  private
    { Private declarations }
    Face : TBitmap;
    FMood, OldMood : TMood;
    procedure SetBitmap;
    procedure SetMood(NewMood: TMood);
    procedure WMSize (var Message: TWMSize);
      message wm_paint;
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
    destructor Free;
    procedure Toggle;
    procedure MouseDown(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
    procedure MouseUp(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
  published
    property Mood: TMood read FMood write SetMood;
  end;

implementation
Uses
  Choose, DsgnIntf;

constructor TSmiley.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FMood := smHappy;
  Face := TBitmap.Create;
  {Old-fashioned API call}
  Face.Handle := LoadBitmap(hInstance, 'Happy');
  Self.Height := MaxHeight;
  Self.Width := MaxWidth;
  SetBitmap;
  OldMood := smHappy;
end; {Create}

destructor TSmiley.Free;
begin
  Face.Free; {Use Free rather than Destroy,
              as Free checks for a nil pointer first}
  inherited Free;
end; {Free}

procedure TSmiley.Toggle;
begin
  if fMood = smOoh then fMood := smHappy else
    Inc(fMood);  {Don't allow fMood to overflow}
  SetBitmap;
end; {Toggle}

procedure TSmiley.SetBitmap;
begin
  Face.Handle :=
    LoadBitmap(hInstance, MoodString[fMood]);
  Self.Picture.Graphic := Face as TGraphic;
  {Use RTTI to cast face as TGraphic, needed by TImage}
end; {SetBitmap}

procedure TSmiley.SetMood(NewMood: TMood);
begin
  FMood := NewMood;
  SetBitmap;
end; {SetMood}

{This method will respond to a mouse push on the Smiley
 by storing the old face for later use and giving the
 "Sad" face.  Smileys don't like to get clicked on!}
procedure TSmiley.MouseDown(Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  inherited MouseDown(Button, Shift, X, Y);
  OldMood := Mood;
  SetMood(smSad);
end; {MouseDown}

{Restores old face when the mouse comes back up}
procedure TSmiley.MouseUp(Button: TMouseButton; Shift:
TShiftState; X, Y: Integer);
begin
  inherited MouseUp(Button, Shift, X, Y);
  SetMood(OldMood);
end; {MouseUp}

{Keeps the user from sizing the Smiley at design time.
 You can use the 'csDesigning in ComponentState' to
 control what the user can do at design time}
procedure TSmiley.WMSize(var Message: TWMSize);
begin
  inherited;
  if (csDesigning in ComponentState) then begin
    Width := MaxWidth;
    Height := MaxHeight;
  end;
end; {WMSize}

procedure Register;
begin
  RegisterComponents('Custom', [TSmiley]);
  RegisterPropertyEditor(TypeInfo(TMood),
    TSmiley, 'Mood', TMoodProperty);
end; {Register}
end.
```

value of the property. Delphi convention says that this function would be declared as:

```
function GetMood: TMood;
```

On the other hand, if the programmer wants to set a new value for Mood, this call would be made:

```
MySmiley.Mood := smSad;
```

Smiley calls the private method SetMood, the "writer," which takes the new mood, smSad, as its parameter. It performs the necessary steps to change the bitmap from its current state to the new state. Thus, a great deal of work can be hidden from the component user, allowing them to get a lot accomplished by making one simple assignment statement. Users of a component do not even need to know how this is done or what is going on in the background. They only need to know that all the housekeeping is being taken care of for them with the single assignment statement.

Notice also that the Mood property is declared in the published portion of the type declaration. Published properties are able to be displayed in the Object Inspector, so settings can be changed at design time. The Object Inspector recognizes published properties and will automatically add the new properties to its list, making them available at design time.

Events

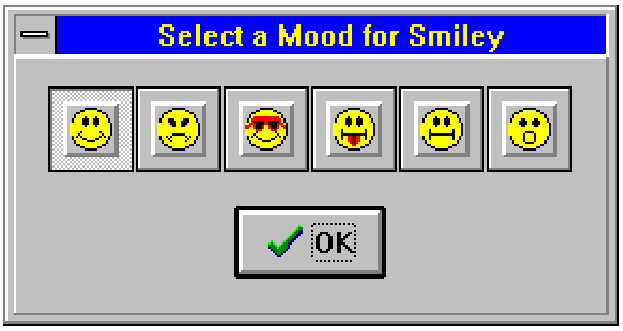
Almost all components will have some set of pre-defined events associated with them such as mouse clicks and key strokes. Delphi's event model allows the designer to assign methods – event handlers in Delphi parlance – to an event which will be called whenever that event occurs. Common events are OnClick, OnKeyDown and OnExit. Events are nothing more than pointers to an event handler. An event handler is a method defined by the programmer which will be called when that particular event fires. A programmer can assign event handlers to events at design time through the Object Inspector, or at run-time using assignment statements. Thus, the program behavior at run-time can be altered by changing the event handlers in the code.

To illustrate this, I created Smiley to have an aversion to being clicked on. In order to do this, I had to override the default MouseDown and MouseUp event handlers. Most Delphi components can react to a variety of events which are most normally described with the prefix On, such as OnClick, OnKeyPress, OnMouseDown, etc. At design time, the Object Inspector will allow the user to quickly create an event handler to deal with the occurrence of that event. As a general rule, VCL components have default handlers corresponding to their OnXXXX, named to match the event name. For example, the OnClick event has a default handler Click.

To give Smiley his aversion to mouse clicks, I overrode the default handlers for OnMouseDown and OnMouseUp. They are named, surprisingly, MouseDown and MouseUp and are declared as shown in Listing 1.

These event handlers are procedures of type TMouseEvent and are declared accordingly. The method contains all of the information about the mouse click, including the button used, its location and keyboard state. A programmer could easily use that information when handling the event.

In this case, Smiley is only interested that the event occurred. When the left mouse button goes down, Smiley first calls his old, default OnMouseDown handler to insure correct processing of the event. He then shows his displeasure by storing his old mood and displaying his sad face. When the mouse comes back up, Smiley returns his mood to the stored OldMood.



➤ *TSmiley's Mood property editor*

Property Editors

Just as Delphi allows new properties to be added to components, it allows for the creation of property editors for those new properties. Delphi provides basic property editors for all of the standard data types like strings, integers and enumerated types. Each of these property editors have the basic methods needed to maintain values of various types in the Object Inspector. The basic enumerated types editor can be seen by double-clicking on the property Cursor when a form has the focus in the Object Inspector. A drop-down box appears listing all of the types of cursors available. Some property editors, like the font editor, are a bit more complex. The font property editor provides a dialog box that lets you edit the entire multiple value property at once.

Smiley also provides a property editor for the Mood property. Double clicking on the Mood property will bring up a small dialog box which allows the user to select the desired mood for Smiley from a group of mutually exclusive speedbuttons.

To create a basic property editor such as the one Smiley uses took two steps (the code is in Listing 2 by the way). First, I needed to design a dialog box that would allow a user to make the choice of moods. The dialog box works like any other dialog box does. In this case, it provides six different speed buttons that have their AllowAllUp properties set to false. This causes them to function like glorified radio buttons, allowing only for a single choice to be made from the six. I use the Tag property to keep track of the currently selected button. The tag settings correspond to the position of the mood in the enumerated type TMood and the dialog has its own property to hold the current Mood setting.

Secondly, I created the property editor type by descending from TEnumProperty, Delphi's default property editor for an enumerated type. I had to override two methods, GetAttributes and Edit, to create the new property editor. GetAttributes is a function call that defines the characteristics of the editor. In this case, I merely set the descendant to have the property paDialog, meaning that the editor would provide a dialog to edit the property.

Overriding the Edit function tells the new property editor to produce the dialog box upon demand and also provides the small '...' box in the Object Inspector, which alerts the user that there is a dialog available to edit the property. Edit merely calls the dialog box and returns the value of the users selection. It uses a call to GetOrdValue to properly set the current value in the new dialog box and, once the user has made a selection, the procedure calls SetOrdValue to set the current value of the property. Both of these methods are descended from the parent property editor.

Property editors must be registered with Delphi just like components are and the call is RegisterPropertyEditor. Smiley registers his Mood editor like this:

```
RegisterPropertyEditor(
  TypeInfo(TMood), TSmiley,
  'Mood', TMoodProperty);
```

This call will alert Delphi that there is a property editor for the TMood type, that it is to be used for the TSmiley component, that the name of the property is Mood and that the property editor should be of type TMoodEditor. Now, when the user double clicks on the Mood selection or clicks the '...' button, a dialog box will appear, allowing the user to select the desired mood.

Odds And Ends

Smiley has a couple more interesting features hidden away which illustrate some of the finer points of Object Pascal and the VCL. One of these is the use of Run Time Typing Information (RTTI).

RTTI allows the programmer to check the type of a variable at run-time, as you might surmise. This often comes into play with event handlers, which can never be sure of what the type of their Sender is. Smiley illustrates the type checking that can go on in his Property Editor Dialog. When the Mood property of the dialog is changed, SetMood gets called (see Listing 2 for the full source code).

SetMood establishes the new value, but of course it must update whichever of the six speedbuttons is depressed. It does this by cycling through all the components on the form, a list of which is kept in the Components property of the form itself.

The procedure then checks the type of each of the components using the new reserved word is, which returns True if the variable is of the specified type and False if it is not. If it finds one that is a speedbutton, then it knows that it is safe

Listing 2

```
unit Choose;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, Buttons, ExtCtrls,
  StdCtrls, Smiley, DsgnIntF, TypInfo;
type
  TChooseDlg = class(TForm)
    BitBtn1: TBitBtn;
    Panel1: TPanel;
    SpeedButton1: TSpeedButton;
    SpeedButton2: TSpeedButton;
    SpeedButton3: TSpeedButton;
    SpeedButton4: TSpeedButton;
    SpeedButton5: TSpeedButton;
    SpeedButton6: TSpeedButton;
    procedure SpeedButton1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    FMood      : TMood;
    procedure SetMood(NewMood: TMood);
  public
    property Mood: TMood read FMood write SetMood;
  end;
  TMoodProperty = class( TEnumProperty )
    function GetAttributes:
      TPropertyAttributes; override;
    procedure Edit; override;
  end;
var
  ChooseDlg: TChooseDlg;
implementation
{$R *.DFM}
procedure TChooseDlg.SpeedButton1Click(Sender: TObject);
begin
  FMood := TMood((Sender as TSpeedButton).Tag);
end; {SpeedButton1Click}
procedure TChooseDlg.FormCreate(Sender: TObject);
begin
  SpeedButton1.Down := True;
end; {FormCreate}
procedure TChooseDlg.SetMood(NewMood: TMood);
var
  Counter: Integer;
begin
  FMood := NewMood;
  for Counter := 0 to ComponentCount - 1 do begin
    if (Components[Counter] is TSpeedButton) then begin
      if TSpeedButton(Components[Counter]).Tag =
        Ord(NewMood) then
        TSpeedButton(Components[Counter]).Down := True;
    end;
  end;
end;  {SetMood}
function TMoodProperty.GetAttributes:
  TPropertyAttributes;
begin
  Result := [paDialog];
end;  {GetAttributes}
procedure TMoodProperty.Edit;
var
  ChooseDlg: TChooseDlg;
begin
  ChooseDlg := TChooseDlg.Create(Application);
  try
    ChooseDlg.Mood := TMood(GetOrdValue);
    ChooseDlg.ShowModal;
    SetOrdValue(Ord(ChooseDlg.Mood))
  finally
    ChooseDlg.Free
  end;
end; {Edit}
end.
```

to typecast it as a speedbutton and make the correct changes to the dialog setting. In this case, SetMood makes sure that the correct, current, setting for the Mood property is selected when the dialog box is opened.

Sometimes, a component designer may want to distinguish between the behavior of a component at run-time and at design-time. Since Delphi is constantly parsing a program's code in the background, allowing for components to function at design time, there may be times when a designer doesn't want a component to exhibit run-time behavior. This can be accomplished by using the ComponentState set:

```
TComponentState = set of
  (csLoading, csReading,
   csWriting, csDestroying,
   csDesigning);
```

This set includes csDesigning, which very handily allows a component to determine whether it's being used in run-time or design time (that is, within the Delphi environment).

In Smiley's case, I have overridden the response to the wm_paint message, causing Smiley not to be sizable at design-time. A simple check for csDesigning in ComponentState makes the determination. After calling the inherited response, Smiley resets its original size if the component is in design mode. See the TSmiley.WMSize in Listing 1.

Component Bitmaps

In order to have a custom bitmap show up on the palette, a component should have a corresponding .DCR file, which is really a resource file with a different extension, containing a single bitmap that is 24x24 pixels. You can use the Image Editor to create a .DCR file.

Conclusion

When you've installed Smiley (it'll go onto the Custom palette page by default) it'll only be two mouse clicks away from being a fun addition to any project!

Smiley covers many of the techniques that a skilled component builder will employ. He shows us basic component building, event handling, properties, property editors and more.

It is left as an exercise for the reader to figure out exactly what one would do with the Smiley component. I suppose it might be considered "truly useless", except for illustrating the finer points of component construction!

Nick Hodges, an experienced Delphi and Pascal developer, can be contacted via CompuServe on 71563,2250