Creating Component Suites: An Event List

by Paul Warren

I've learned some lessons about component writing lately. There is a big difference between writing an individual component and writing a component suite. By "component suite" I mean a group of components designed to work together for a common purpose, like the components on the Data Controls page of the Palette.

I have been working on a preventive maintenance manager, a very date intensive application. I purchased a calendar component suite which proved ineffective, so I started writing my own system. During development I found several areas where component suites require special attention.

First, components in suites generally need to interact with one another requiring special events or messages.

Secondly, they are likely to be used frequently and therefore influence the appearance of an application. Consideration needs to be given to allowing the user to 'borrow' your control's appearance and functionality in other areas.

Finally, components in a suite have to be simple to use. It's one thing to puzzle out how to use a single component, quite another to repeat this for several more.

In this article I'm going to examine the first issue: making components in a suite interact.

Events

Events are the real heart of the Delphi VCL. Without events components would be useless. There is perhaps one little quirk with events that needs to be fixed. If you hook an event in your code the event is no longer accessible to the user. Let's say you want to make your component react when the user clicks another component. You can do this by hooking the event. Listing 1 shows the code to do this.

At design time the user can assign a handler to the OnClick event. At runtime, though, your code re-assigns the event to the DoSomething handler and the user's code doesn't execute.

There are three possible solutions to this problem. First, you could use messages. SendMessage and PostMessage only work for descendants of TWinControl. Perform works for all TControl descendants but not non-visual components. All messages have a disadvantage in that the message is sent whether or not there are any components to respond.

Second, you could create a TComponentLink like TDataLink. This is probably a useful solution, but would involve a lot of coding. Finally, you could create an event list of some type. This is the route I chose.

Actually, there is a fourth method as well: declare an array[0..somenumber] of TNotifyEvent and assign your handler to one of the array members. This isn't a good solution though. Sooner or later a user is going to want to hook somenumber+1 events and they won't be able to.

TEventList

I was a big fan of Borland Pascal 7 collections. I used them a lot for many seemingly unrelated tasks. I really missed them for a while until I realized TList was nearly as useful. TCollection and TList both use untyped pointers to store data. Consequently, with a little imagination you can store pretty much anything. Even event handlers, well the pointers to them anyway.

To start, I created a TEventList descendant of TList. I added an AddEvent method and an Event property of type TNotifyEvent. To save the user from having to remember to dispose of any items created I added a destructor. Listing 2 shows the declaration.

While TNotifyEvent is a method pointer, you can't simply store it directly in a TList, at least not unless you like GPFs. It took a fair amount of experimentation to find the solution.

The answer is to wrap the TNotifyEvent in an object. TWrapper has one field, AEvent, of type TNotifyEvent. In the AddEvent method I create a new TWrapper, set its Event field to the passed Value parameter (type TNotifyEvent) and add the

➤ *Listing 1*

```
constructor TSomeComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FSourceComponent.OnClick := DoSomething;
end;
procedure TSomeComponent.DoSomething;
begin
  { do something useful }
end;
```

➤ *Listing 2*

```
TEventList = class(TList)
private
  function GetEvent(Index: Integer): TNotifyEvent;
public
  destructor Destroy; override;
  function AddEvent(Event: TNotifyEvent): integer;
  property Events[Index: Integer]: TNotifyEvent read GetEvent;
end;
```

object to the list. Listing 3 shows the TWrapper declaration and the AddEvent method.

In the destructor I dispose of the TWrappers one by one until the list is empty and then call the inherited destructor.

The only thing left is to access the events when required. To do this the GetEvent property access method returns the event based on the passed index into the list. See Listing 4.

Using TEventList

So what is TEventList really for? While the concept is simple it still seems a lot of work to allow more than one component to respond to a single event.

Well let's look at a trivial example. I've created a TSpeedButton descendant called THookedButton. It has two additional fields, FEventList and FHookEvent, and a write only property HookEvent, which has a property access method named SetHookEvent. Any component which assigns a method to a THookButton.HookEvent will receive notification when the overridden Click method is called. Listing 5 shows the code for THookedButton.

I have also created a TLabel descendant called THookLabel which can display a date, time or an upbeat message as its Caption depending on the setting of the MessageKind property. THookLabel also has a protected method UpdateLabel which simply sets the Caption when called. Listing 6 shows the complete THookLabel code.

The intention here is that when the user clicks the button each label will respond in a different and distinguishable way. So what, you ask! You can do that by calling UpdateLabel in the OnClick event handler within the main form.

Well consider this. There are many component suites available, both commercial and shareware or freeware that rely on interaction between various components. Will your users prefer the following instructions:

> *Place the button and three labels on a form, set the label* MessageKind *properties to the desired value and, in the button's* OnClick *event type:*

```
HookLabel1.UpdateLabel;
HookLabel2.UpdateLabel;
HookLabel3.UpdateLabel;
```

or do you think they will prefer these instead:

> *Place the button and three labels on a form, set the* MessageKind *property to the desired value and...*

...well that's it. No code. It just works.

Remember this is a trivial example. I have personally tried to use components which required extensive and convoluted code in an event handler, often repeated for each component that is supposed to respond to the event. Much of this coding can be hidden from the user inside methods which hook

➤ *Listing 3*

```
type
  { object wrapper for TNotifyEvent }
  TWrapper = class(TObject)
    AEvent: TNotifyEvent;
  end;
function TEventList.AddEvent(Event: TNotifyEvent): integer;
var
  P: TWrapper;
begin
  { create a new wrapper }
  P := TWrapper.Create;
  { set its AEvent field := TNotifyEvent }
  P.AEvent := Event;
  { add to list and return its position }
  Result := Add(P);
end;
```

➤ *Listing 4*

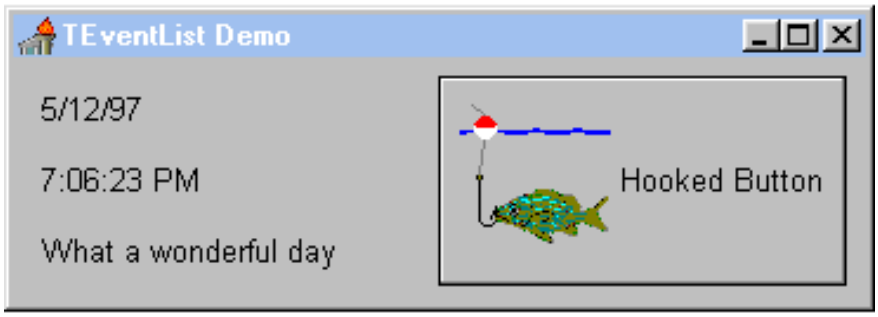
```
function TEventList.GetEvent(Index: Integer): TNotifyEvent;
var  P: TWrapper;
begin
  { set var P := desired Item }
  P := Items[Index];
  { return the TNotifyEvent }
  Result := P.AEvent;
end;
```

➤ *Listing 5*

```
unit Hookdbtn;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, Buttons, EList;
type
  THookedButton = class(TSpeedButton)
  private
    FEventList: TEventList;
    FHookEvent: TNotifyEvent;
    procedure SetHookEvent(Value: TNotifyEvent);
  protected
    procedure Click; override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    property HookEvent: TNotifyEvent write SetHookEvent;
  published
  end;
procedure Register;

implementation

constructor THookedButton.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FEventList := TEventList.Create;
end;
destructor THookedButton.Destroy;
begin
  FEventList.Free;
  inherited Destroy;
end;
procedure THookedButton.SetHookEvent(Value: TNotifyEvent);
begin
  FEventList.AddEvent(Value);
end;
procedure THookedButton.Click;
var i: integer;
begin
  inherited Click;
  for i := 0 to FEventList.Count-1 do begin
    FHookEvent := FEventList.Events[i];
    FHookEvent(Self);
  end;
end;
procedure Register;
begin
  RegisterComponents('Samples', [THookedButton]);
end;
end.
```



➤ *Left: Figure 1*

the appropriate event. Figure 1 shows the example in action.

Further Uses For TEventList

You may have noticed that, while I made no use of the fact that `TEventList` is indexed, you could in theory use the event index to trigger specific events. `AddEvent` returns the event index to the hooking component. Who knows, it could be useful.

Conclusion

Not only do you relieve your users of tedious and frustrating coding, you prevent mistakes. Twice now I have been so annoyed with component suites that I returned them to the vendor. Carefully planned components avoid this problem and one aspect of careful planning is reducing reliance on end user code in component execution.

Over the next few months, space permitting, I'll be back to examine the appearance issue and share some other useful tips for building component suites. Meanwhile, experiment with `TEventList` and possibly the `Perform` method and see how much nicer components can be when using them is virtually code free.

➤ *Listing 6*

```
unit Hooklbl;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, HookdBtn;
type
  TMessageKind = (mkDate, mkTime, mkUpbeat);
  THookLabel = class(TLabel)
  private
    FMessageKind: TMessageKind;
    FSource: THookedButton;
    procedure SetSource(Value: THookedButton);
  protected
  public
    constructor Create(AOwner: TComponent); override;
    procedure UpdateLabel(Sender: TObject);
  published
    property MessageKind: TMessageKind read FMessageKind write FMessageKind;
    property Source: THookedButton read FSource write SetSource;
  end;
procedure Register;
implementation
constructor THookLabel.Create(AOwner: TComponent);
var i: integer;
begin
  inherited Create(AOwner);
  { search for any THookedButtons }
  for i := 0 to TForm(AOwner).ComponentCount-1 do begin
    if TForm(AOwner).Components[i] is THookedButton then begin
      { if found set FSource }
      FSource := THookedButton(TForm(AOwner).Components[i]);
      { break since we only need one FSource }
      Break;
    end;
  end;
end;
procedure THookLabel.UpdateLabel(Sender: TObject);
begin
  case MessageKind of
    mkDate: Caption := DateToStr(Date);
    mkTime: Caption := TimeToStr(Time);
    mkUpbeat: Caption := 'What a wonderful day';
  end;
end;
procedure THookLabel.SetSource(Value: THookedButton);
begin
  FSource := Value;
  { if successful hook HookEvent }
  if (FSource <> nil) then
     FSource.HookEvent := UpdateLabel;
end;
{ register component on Samples page }
procedure Register;
begin
  RegisterComponents('Samples', [THookLabel]);
end;
end.
```

Paul Warren runs HomeGrown Software Development in Langley, British Columbia, Canada. Email Paul at hg_soft@uniserve.com or visit his web site:
http://users.uniserve.com/~hg_soft

[and be very careful if you want to try and sell him a component suite...! Editor]