How to set a component's default event handler

Why this article?

When you double click most Delphi components at design time the IDE automatically creates an empty event handler for the default event. Sometimes you need a different event to be used as the default. The purpose of this article is to explain how to do this. (There is some information on this topic in the Delphi 7 help file, but the example provided there is incorrect).

Let's make this a little more concrete by considering these three components:

```
type
  TCompA = class(TComponent)
  private
    fOnFoo: TNotifyEvent;
    fOnBar: TNotifyEvent;
    fOnClick: TNotifyEvent;
    fOnChange: TNotifyEvent;
    fOnChanged: TNotifyEvent;
    fOnCreate: TNotifyEvent;
  published
    property OnFoo: TNotifyEvent read fOnFoo write fOnFoo;
    property OnBar: TNotifyEvent read fOnBar write fOnBar;
    property OnChange: TNotifyEvent read fOnChange write fOnChange;
    property OnChanged: TNotifyEvent read fOnChanged write fOnChanged;
    property OnClick: TNotifyEvent read fOnClick write fOnClick;
    property OnCreate: TNotifyEvent read fOnCreate write fOnCreate;
  end;

  TCompB = class(TComponent)
  private
    fOnFoo: TNotifyEvent;
    fOnBar: TNotifyEvent;
    fOnChange: TNotifyEvent;
  published
    property OnFoo: TNotifyEvent read fOnFoo write fOnFoo;
    property OnBar: TNotifyEvent read fOnBar write fOnBar;
    property OnChange: TNotifyEvent read fOnChange write fOnChange;
  end;

  TCompC = class(TComponent)
  private
    fOnFoo: TNotifyEvent;
    fOnBar: TNotifyEvent;
  published
    property OnFoo: TNotifyEvent read fOnFoo write fOnFoo;
    property OnBar: TNotifyEvent read fOnBar write fOnBar;
  end;
```

Listing 1

Double clicking the components in the Delphi IDE creates the following event handlers:

- `TCompA`: *OnCreate*
- `TCompB`: *OnChange*
- `TCompC`: *OnBar*

Suppose we want the *OnFoo* event to be created in all cases? We'll find out how to do this in the course of this article.

Some background

Before we can solve our problem we need to examine how Delphi 7 decides which event to use as the default. (Things are a subtly different in Delphi 4 – as we'll see later).

When a component is double clicked, Delphi calls the *Edit* method of the registered component editor. The default component editor is *TDefaultEditor* and it is this editor that is responsible for creating the event handler. Let's examine how *TDefaultEditor.Edit* decides on the the default event.

Through a round about process, the *Edit* method ultimately calls the *TDefaultEditor.EditProperty* method once for each of the component's properties. Rather than pass a reference to the property itself to *EditProperty*, it is a reference to the associated *property editor* that is actually passed. *EditProperty* checks the name of each event property and records the property editor of the preferred event. These events, in order of preference, are:

- *OnCreate*
- *OnChange*
- *OnChanged*
- *OnClick*
- the first event alphabetically

This means that if *OnCreate* is present it is used as the default. If it is not present then *OnChange* is used, and so on. If none of the listed events is present then the event that is first alphabetically is used.

Overriding the default behaviour

From the discussion above it is clear that if we are to specify the default event ourselves we need to change the way that *TDefaultEditor.EditProperty* works. Luckily for us this method is virtual, so we can subclass *TDefaultEditor* and override *EditProperty*. The new class can be declared very simply as follows:

```
type
  TMyCompEditor = class(TDefaultEditor)
  protected
    procedure EditProperty(const PropertyEditor: IProperty;
      var Continue: Boolean); override;
  end;
```

Listing 2

The implementation is equally straightforward:

```
procedure TMyCompEditor.EditProperty(const PropertyEditor: IProperty;
  var Continue: Boolean);
begin
  // only call inherited method if required event name
  if CompareText(PropertyEditor.GetName, 'OnFoo') = 0 then
    inherited;
end;
```

Listing 3

Let's look at how this works. Recall that, in the base class, *TDefaultEditor.EditProperty* was called once for each property in the component. The method chose the default event from all those it was passed. Now, if *TDefaultEditor.EditProperty* was only called once then the default event must be the one whose property editor was passed in that single call.

Our overridden method works by ensuring that the inherited method **is** only called once – for the event we want to be the default. In our descendant class, it is *TMyCompEditor.EditProperty* that is called for each property in the component. The method simply checks for a property editor whose property has the required name (`'OnFoo'`) and passes that property editor – and only that property editor – to the inherited method. And violá – *OnFoo* is the default event!

A reusable solution

In the code we developed in the previous section we hard-wired the name of the default event.

Our next job is to generalize the solution to make the code easier to re-use. Since the classes are distinguished only by the name of the default event they select, we will develop a base class that has an abstract method that descendants override to return the name of the required event.

Here is the declaration of the base class:

```
type
  TCompEditorBase = class(TDefaultEditor)
  protected
    function DefaultEventName: string; virtual; abstract;
      { override to return name of default event }
    procedure EditProperty(const PropertyEditor: IProperty;
      var Continue: Boolean); override;
      { records property editor of default event }
  end;
```

Listing 4

The implementation should come as no surprise – we simply replace the hard-wired name in the earlier class with a call to the abstract method:

```
procedure TCompEditorBase.EditProperty(
  const PropertyEditor: IProperty;
  var Continue: Boolean);
begin
  if CompareText(PropertyEditor.GetName, DefaultEventName) = 0 then
    inherited;
end;
```

Listing 5

We can now implement a component editor for a specific default event – our old friend *OnFoo* once more – simply by overriding the abstract *DefaultEventName* method as follows:

```
type
  TCompEditor = class(TCompEditorBase)
  protected
    function DefaultEventName: string; override;
  end;
  ...
function TCompEditor.DefaultEventName: string;
begin
  Result := 'OnFoo';
end;
```

Listing 6

Tidying up

To get these examples to compile we need to use the `Classes`, `SysUtils`, `DesignIntf` and `DesignEditors` units. Note that the last two units can only be used when integrating into the IDE – they can't be used in stand-alone applications.

We also need to register the component editor in our unit's Register procedure as follows:

```
procedure Register;
begin
  RegisterComponentEditor(TCompA, TCompEditor);
  // etc ...
end;
```

Listing 7

Delphi 4 differences

In Delphi 4 *TDefaultEditor.EditProperty* has a different signature. It is defined as:

```
procedure EditProperty(PropertyEditor: TPropertyEditor;
  var Continue, FreeEditor: Boolean);
```

Listing 8

We can deal with this using conditional compilation. Assuming the DELPHI6ANDUP symbol is defined when we are using Delphi 6 and above, we can implement *TCompEditorBase.EditProperty* as follows:

```
procedure TCompEditorBase.EditProperty(
  {$IFDEF DELPHI6ANDUP}
  const PropertyEditor: IProperty; var Continue: Boolean
  {$ELSE}
  PropertyEditor: TPropertyEditor; var Continue, FreeEditor: Boolean
  {$ENDIF}
);
begin
  if CompareText(PropertyEditor.GetName, DefaultEventName) = 0 then
    inherited;
end;
```

Listing 9

Additionally, we need to replace the `DesignIntf` and `DesignEditors` units with `DsgnIntf`:

```
uses
  Classes, SysUtils,
  {$IFDEF DELPHI6ANDUP}
  DesignIntf, DesignEditors;
  {$ELSE}
  DsgnIntf;
  {$ENDIF}
```

Listing 10

Summary

In this article we have reviewed how to change the event handler that is opened in the IDE when a component is double clicked.

We first examined how Delphi decides which events to use for this purpose and then developed a sub class of *TDefaultEditor* that can explicitly specify the default event. We then went on to generalize the solution by developing an abstract base class for component editors that change the default event.

The main discussion closed with a review of the units required to compile the new classes and looked at how to register the component editor.

Finally we discussed the changes that need to be made to compile the code under Delphi 4.

I hope this article was useful – please give me feedback using the *contact page*.

Demo code

If you would like to experiment with this code you can *download the demo code* that implements the example components and the component editors discussed in this article.