Effective Delphi Class Engineering Part 6: To Talk Of Many Things

by David Baer

This article concludes my series on effective class engineering using Delphi. It's been a long haul, but I hope that you have found it to be a worthwhile effort which has made a difference in your development projects.

The previous articles all focused on a single major theme, which has now left a few smaller tidbits lying about unacknowledged. None of these topics is sufficiently large to require an entire article. So, we'll look at them here. You could say that our theme this time is *miscellany*.

The Big Event

Events are not restricted to components; they are marvels of flexibility that can be powerful assets for users of your class.

Even the most casual Delphi user will almost certainly have encountered and made use of events. A typical scenario is as follows: drop a button on a form, switch to the `Events` tab in the Object Inspector, double click the `OnClick` line, and you're in business. Delphi adds a method to your unit into which you may immediately place the code that responds to the click of the button.

But what has this to do with those of us writing classes that aren't components? To understand the answer, let's break the scenario I have described above into two essential pieces. On the one hand, we have the Object Inspector, double clicking, code generation, and so on, that is supplied by Delphi's IDE. But none of this would be of much use if there weren't something at the language level supporting it all, and that is where events, the second piece, come into play.

Events are essentially procedural types. Stated another way, an event *type* declaration specifies a type of procedure with a particular parameter list composition. For example, the most familiar event type of them all is declared as:

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

The `of object` designation specifies that the procedure is a class method, as distinct from a standalone procedure. Given this type definition, you may create a variable of that type:

```
MyNotifyEvent: TNotifyEvent;
```

You may assign a value to it in your program code:

```
procedure
  TSomeClass.MyNotifyHandler(
  Sender: TObject);
begin
  ...
end;
...
MyNotifyEvent :=
  MyNotifyHandler;
```

Although lacking the convenience of being able to assign the event handler code at design-time, users of our class may still be very much ahead of the game thanks to our thoughtfulness.

Events are not formally part of the Delphi object model; they're available to us courtesy of a language feature. But, let's not worry too much about formality here or we might deprive our class users of a major convenience.

When should you consider making events available for your class users? There's no simple answer to that. However, consider that, in languages without events, there are two typical alternatives. On the one hand, event handlers are very much akin to callback procedures (which were commonly employed prior to the widespread adoption of object oriented programming technologies). So, there's one answer: use them where you might be tempted to introduce some kind of callback method.

In OO languages, another alternative would be for your class user to create a descendant class and override one or more methods to introduce the 'in-response-to' processing. But, without doubt, this requires extra work on the part of our user, whereas an event can be tapped into with a single assignment statement. Remember that requiring a derived class in the first place implicitly requires that users of our class be class writers themselves.

Those who have studied the VCL to any degree will already know how easily events are fired. For those who haven't, let's take a brief look at an example:

```
MyEvent: TNotifyEvent;
...
if Assigned(MyEvent) then
  MyEvent(Self);
```

What's happening with this `Assigned` business? We're checking to see if a handler method has actually been assigned, and if one has, we call it. We cannot simply code:

```
if MyEvent <> nil then ...
```

Doing so would result in an objection by the compiler, which doesn't know that we are not attempting to call `MyEvent`.

Special Events

If you can't find an event type that fits your needs, it's not a problem: just define a new event type.

Recall that events are available courtesy of being able to declare procedure types in Object Pascal. While the workhorse `TNotifyEvent` will do nicely for all kinds of situations, and the VCL contains numerous other type definitions for event procedures, you may occasionally need something new that's not provided. This is not a problem; just define the type yourself.

Consider some of the specializations available:

- Passing a notification type into the handler routine.
- Having a signal back from the handler code that it took care of something (and your code doesn't need to do anything further).
- Passing in a new property setting in an `OnBeforeChange` type event.
- Allowing the handler code to supply some optional values to be used in preference to default values.

The possibilities are limitless. Listing 1 illustrates some of them, taken from examples in the VCL.

One convention you'll normally want to observe in your custom event types is to make the first parameter the familiar `Sender: TObject`. It may seem that, in some cases, this information is essentially useless. However, remember that users of your class may wish to supply a single handler method that services multiple objects, possibly of differing class types. Without `Sender`, that may prove to be difficult.

Don't clutter up your custom event parameter lists with information readily available from another source. For example, if you have some sort of array object which maintains a `CurrentRow` property, an `OnRowChanging` event might seem thoroughly convenient having both the current and new row index passed in as a parameter, but only the new row index will be unavailable to the handler code. It can pick up the current index from the property.

```
TMovedEvent = procedure (Sender: TObject; FromIndex, ToIndex: Longint) of object;
TTVChangingEvent = procedure (Sender: TObject; Node: TTreeNode; var AllowChange:
  Boolean) of object;
TTVChangedEvent = procedure (Sender: TObject; Node: TTreeNode) of object;
TDrawCellEvent = procedure (Sender: TObject; ACol, ARow: Longint; Rect: TRect;
  State: TGridDrawState) of object;
```

➤ *Listing 1*

Next, let me offer some suggestions regarding event naming. First of all, do it the way it's done in the VCL as far as naming the types (`TWxyzEvent`) and corresponding properties (`OnWxyz`). I guarantee that you'll be annoyed with yourself if you don't learn and use this convention from the very start.

On the other hand, you might wish to avoid another (all too common) VCL practice. Many events are notifications that some object state is about to change or has just changed. What does the name `OnChange` tell you about the timing (ie, before or after)? Would `OnChanging` and `OnChanged` not be a bit more informative all around? Or how about `OnBeforeChange` and `OnAfterChange`? Either way, it's immediately clear at what point the event is being fired.

Even when you perceive that you'll only need one of the two timings (before versus after), if you settle for `OnChange` you will find that adding the second timing later on will leave you with an inconsistency that looks unprofessional. For many event types, you will sacrifice nothing by including a verb tense in the name and you'll enhance the ability to extend event functionality later.

There's one more issue worth mulling over. Delphi and the IDE always work with event handlers that are methods. This approach allows event response code to be delegated to a method of the form.

For a class (as opposed to a component), a class method may not be the best packaging for an event handler. Your class user may actually be better served with a standalone procedure instead of a method. Object Pascal allows us to declare both method and standalone procedure types, so the language doesn't force the method solution on us. It's definitely an option.

Your class user may not always have a form handy, and some cases could arise where there's no other convenient class with which to associate the event handler method either. From that perspective, a standalone procedure is a cleaner solution. Unfortunately, providing a standalone procedure for an event handler deviates considerably from normal Delphi practice. Therefore, it's not an obvious choice: convenience versus potential confusion.

Exceptions Are The Rule

Exceptions are class instances: you can often make a class or class family more usable by defining accompanying exception types.

Appropriate use of exceptions in programs is good practice in all manner of Delphi coding, not just in class writing. But whereas the developer writing general application code can be comfortable with raising an exception by creating an instance of `Exception`, as a class designer you shouldn't be so casual.

Exceptions are classes and, as such, inheritance is an option when declaring them. Indeed, if you wish to declare a new exception type, you must inherit directly or indirectly from `Exception`:

```
EMyClassError =
  class(Exception);
```

If you wish to provide additional information about an error condition (ie, more than just the error description text), you may define a class that has an appropriate constructor and properties. Listing 2 provides an example of just that. The example comes from some code I presented in a previous article on XML. Forgive me if that seems a bit lazy, but the fit was just too good to ignore here. The code takes error information returned from a Microsoft XML parsing service (via a COM interface), and

repackages the information as a Delphi exception, which provides read-only properties for the extra bits of error information.

One other motivation exists for supplying your own custom exceptions, and again it's an advantage to your class user. Most of the time code just needs to protect itself from an ungraceful or embarrassing failure due to some lower-level problem. Using a general-purpose `try..except` block will normally do the trick.

But there are times when we need to code responses to specific error conditions, ones that may be recoverable, for example. We can do this by using the `on` clause of the `except` block. But this option may be available only when the error conditions are adequately delineated via individual exception types.

If nothing else, you might want to define one general-purpose exception type for your class or class family, just to keep your user's options open. Thus, for class `TWonderWidget`, you could define `EWonderWidgetError`. Then, any place in the class code you want to raise an exception, do it using that exception type.

However, if there is an existing Delphi exception type that's a perfect fit for the problem situation, go ahead and use it. If your class supplies some kind of string to numeric conversion method, for example, use `EConvertError` to signal an error condition. In doing so, you will remain nicely consistent with the VCL.

Your Assignment, Mr Phelps

Go with the flow and supply an `Assign` *method for your class.*

You are probably aware that components routinely supply an assignment method for themselves or for one or more of their properties, straightforwardly named `Assign`. But `Assign` isn't just a component feature. In fact, it first appears in the VCL hierarchy in class `TPersistent`, which exists between `TObject` and `TComponent`.

You can supply an `Assign` method for your class even if it doesn't descend from `TPersistent` (although we'll see in a moment that `TPersistent` has a brilliant trick up its sleeve in this regard). But let's assume for now that our class is not a `TPersistent`. Even in this case, the availability of an `Assign` stays consistent with conventional VCL practice.

The most obvious use of an `Assign` method is where the 'from' object is of the same class type as the 'to' object. So, for a class of type `TMyClass`:

```
procedure Assign(
  Source: TMyClass);
```

But there's a more flexible way that can be just what's called for where more than one type of source object is a candidate for copying. Define `Source` as type `TObject`. Then, in the method code, you can use the `is` operator to determine what kind of assignment copying to do. If the type is not one you know how to deal with, then simply raise an exception.

But, really, there's a much better way.

Don't Dither, Be TPersistent

Consider using `TPersistent` *as a parent class rather than* `TObject` *to bolster the* `Assign` *capabilities.*

Now for that trick I promised to explain earlier. I've written about this before in *The Delphi Magazine*, because I think it represents the pinnacle of polymorphic elegance. The culmination of this trick is that you can effectively 'teach' a foreign class (one for which you may not even have source code) how to assign an instance of your class to an instance of its own class. Let's see how this is accomplished.

```
type
  EXmlDError = class(Exception);
  EXmlDParseError = class(EXmlDError)
    FErrorCode:   Integer;
    FReason:      String;
    FSrcText:     String;
    FLine:        Integer;
    FLinePos:     Integer;
  public
    constructor Create(ParseError: IXMLDOMParseError);
    property ErrorCode: Integer read FErrorCode;
    property Reason: String read FReason;
    property SrcText: String read FSrcText;
    property Line: Integer read FLine;
    property LinePos: Integer read FLinePos;
  end;
constructor EXmlDParseError.Create(ParseError: IXMLDOMParseError);
begin
  inherited Create('XML Parse Error');
  FErrorCode := ParseError.errorCode;
  FReason := ParseError.reason;
  FSrcText := ParseError.srcText;
  FLine := ParseError.line;
  FLinePos := ParseError.linePos;
end;
```

➤ *Listing 2*

We begin with the virtual method `Assign` in `TPersistent`, which has the single parameter `Source` of type `TPersistent`. A typical implementation will work in an `Assign` method override in a fashion similar to that described above. The code will run through a check of object types it's prepared to deal with. If the code can handle a type, it will, and we're done. If not, we continue checking for other types we can handle.

If we reach the end of the list of eligible types, the code must then call `inherited Assign`, and the process repeats. If we arrive at `Assign` in `TPersistent`, you'd think we're at the end of our options. But it's not *quite* time to give up and raise an exception. `Assign` first calls the `AssignTo` virtual method of the source object (it's a certainty that `AssignTo` is available to call, since the source object must be a `TPersistent` descendant).

If `AssignTo` *of the source class* knows how to copy itself to the target class, it does so. This is precisely how we can 'teach' the foreign class about the copying operation. The foreign class isn't actually performing the copy; it only looks like it is in the source code. Instead, we slyly insert our services into the processing, and we can do so courtesy of the marvellous insight that went into the design of `TPersistent`.

The code for these two short methods is in Listing 3. You can see the `AssignTo` capability in action in the VCL unit Controls.pas, where it's used in several different classes.

Type Information

RTTI (runtime type information) isn't restricted to components; if your class can benefit from it, use it.

One of the key enablers of Delphi's RAD capability is the ability of a class instance to discover type information about its properties and even access the values of those properties at runtime, based on property names. This is used, for instance, when creating forms from internal .DFM files (which live in the executable as Windows resource files).

But RTTI doesn't have to be limited to components, and it can occasionally be extraordinarily useful. I was recently able to put it to very good use in a class I wrote for an article in Issue 55 (March 2000). The class was a list class (a container class for objects) that that could be sorted on properties of the contained class.

The list class could not possibly know in advance what properties would be present in the contained objects. RTTI provided the magic. Forgive me for citing my previous work again (but it keeps me from having to dig too hard to find appropriate examples).

Using RTTI does involve a bit of processing overhead, so you may need to avoid it in performance-critical situations. It's also not a universal solution. RTTI is available only for properties that are declared as `published`.

That said, it also occasionally offers a means for some inspired innovation. The subject is much too big to examine here. At the moment, your best bet for learning about it is probably *Delphi In A Nutshell* by Ray Lischner (which I recommended in an earlier instalment as well).

RTTI isn't the easiest thing to learn to use (although several new functions in Delphi 5 made life a bit easier than in previous releases). But you can do some marvellous things with it. If you are serious about mastering Delphi class engineering, you owe it to yourself to learn how to work with RTTI.

Message Handling

Understand the message handling capabilities of Delphi objects; messages are one of the most flexible tools at your disposal.

The Object Pascal language offers a powerful option for declaring methods. Methods declared as follows are message handlers:

```
procedure Handle1234(
  Msg: TMessage); message 1234;
```

Such methods are used extensively throughout the VCL, to handle both Windows messages and other 'pseudo-Windows' messages that are synthesized by the VCL. In practice, the message number (it must be a constant for the compiler to accept it, by the way) would be a symbolic constant, and the method name would be the same, but use lower case and omit underscores. For example:

```
procedure CNKeyDown(
  var Message: TWMKeyDown);
  message CN_KEYDOWN;
```

A message handler must have a single parameter, and Delphi records are invariably used for it. The compiler really doesn't care about the size or composition of the record, but it does expect the record to start with the message number as a four-byte `Cardinal`. The record layout that follows the number is of concern only to the sender and recipient (that is, the actual code in the message handler).

Windows messages are not delivered into our programs in this fashion, but Delphi runtime facilities convert them to this format early on. An interesting side effect of this is that we uncouple messaging from its dependency on Window handles and convert it to an object-based messaging framework. This in turn allows non-WinControl objects (controls with no Windows handle) to participate in messaging.

But what does any of this have to do with class engineering, especially if our class has nothing to do with the Windows operating environment? The answer is that inter-object communication with messages is loose-coupling Nirvana. One object can send a message into another. If the recipient knows how to handle the message (ie, it has a message handler for that particular message), then that's all well and good. If it does not, then it's just a non-event for which some CPU cycles were wasted.

Actually, that's the major potential downside of message-based object communication. Method calls are much, much more efficient. Responding to an incoming message requires an object to look through a class table to see if it handles the message. If it finds the message number in that table, it calls the associated method. If not, it continues the search in the object's parent class, and so on.

➤ *Listing 3*

```
procedure TPersistent.Assign(Source: TPersistent);
begin
  if Source <> nil then
    Source.AssignTo(Self)
  else
    AssignError(nil);
end;
procedure TPersistent.AssignError(Source: TPersistent);
var
  SourceName: string;
begin
  if Source <> nil then
    SourceName := Source.ClassName
  else
    SourceName := 'nil';
  raise EConvertError.CreateResFmt(@SAssignError, [SourceName, ClassName]);
end;
procedure TPersistent.AssignTo(Dest: TPersistent);
begin
  Dest.AssignError(Self);
end;
```

Of course, all of this is done by low-level facilities and we needn't get involved in the fussy details. All we need to do is provide a handler method, or make use of an alternative. If an object does not have a handler for a message, the message eventually ends up being passed to a virtual method of `TObject`:

```
procedure DefaultHandler(
  var Message); virtual;
```

If we wished, for example, to handle a range of message numbers with the same one piece of code, this is a way to do it. Another possibility is that, with `DefaultHandler`, we are not restricted to constant message numbers. We can test incoming message numbers against variables. For examples of `DefaultHandler` being overridden, once again, refer to the Controls.pas unit in the Delphi source.

So, we've a couple of ways to process incoming messages. What about the other side of the communication: how do we send one? `TObject` is, once again, at your service. The `TObject` method `Dispatch` is our answer:

```
procedure Dispatch(
  var Message); virtual;
```

Although `Dispatch` is a virtual method, I cannot imagine why anyone would ever need or want to override it (so just don't, OK?). To send a message to an object, simply invoke its `Dispatch` method passing a record as the parameter:

```
SomeObject.Dispatch(MyMessage);
```

That's all there is to it.

One caution is: if you do include message handler methods in your class, you should avoid calling them directly. It's not obvious, but calls to message handler routines are not polymorphic. They are compiled using early binding. If you need to invoke processing via both messages and calls, put the main processing code in a method you can call, and call that method from the message handler as well.

All The Rest

Remember that all those other principles of good coding practice still apply when you're writing class code.

I have received a number of suggestions during the course of writing this series regarding additional points to make. I've rejected a number of them because, while they involve sound advice on writing effective code, they had little or nothing to do with class design and coding. That was my criterion for including a topic: it had to directly apply to the programming of classes.

It would have been inappropriate, for example, to recommend: 'Avoid the use of `with` statements; they make debugging difficult, invite obscure bugs, and are deeply evil'. It doesn't matter that this advice is profoundly true (which, of course, it is). It just has nothing specifically to do with class engineering.

Nevertheless, I think it does no harm to offer a reminder that the rules don't change just because you're doing things in an OO fashion. The fact that OO can increase the reliability of your code doesn't mean you're allowed to get sloppy as compensation.

One item that's invariably mentioned in any list of good programming practices applies in spades for class code: use of meaningful names. I'm referring in particular to names used for public methods and properties.

Ease of use has to be regarded as one of the more important metrics of class quality. To achieve it, meaningful method and property names are unquestionably a requirement. So, spend the time it takes to get this right. Self-documentation begins with good names. Consider that the best external documentation may be that which isn't needed in the first place.

Closing Credits

So, we've reached the end at last. When I first got the go-ahead on this series from Our Esteemed Editor, I was quite excited. I thought: 'This'll be great. I won't need to constantly fret about finding new topics, there will be little research required, and the articles will be easy to write'. I was right about the first two, but far off the mark about the third.

The task was certainly made easier, however, by several individuals who contributed advice and encouragement. Thanks to Dave Heard, William Tai and Gert Kello for their constructive criticism. Very special thanks to Bob Linfield for making sure I didn't talk too fast during the tricky bits. And special thanks, also, to Will Signal, who can spot a grammatical error at 50 paces, for finding... ahem... one or two.

Lastly, there's one final person to thank: our old friend Anonymous, the author of countless gems of language abuse and object model misuse. We needn't worry about hurt feelings here, because Anonymous rarely bothers to read software books or magazines. Encountering these stinkers has been a source of continual frustration throughout my Delphi career, but it has also been the inspiration for most of the guidelines presented in this series. At the end of the day, that suggests one final guideline: *when life hands you lemons, publish!*

David Baer is Senior Architectural Engineer at StarMine in San Francisco. Just because he loathes 'with' statements doesn't mean he's not a 'with it' kind of guy. You can contact him at dbaer@starmine.com