# *Effective Delphi Class Engineering* Part 5: You Are TEgg Man... I Am TWalrus

**by David Baer**

Last month we concluded right in the midst of an examination of polymorphism and inheritance. We're going to resume just where we left off.

As I write this introduction (having already finished the bulk of the piece), I'm startled by what has turned out to be a nearly total absence of code examples. It's not that I start any article with some particular balance of prose to code in mind, but this one turned out to be mostly words. If it's code you want, however, the last guideline will satisfy your craving, I promise.

Now, without further ado, let's dive right back into it.

## Abstract Methods

*Use abstract method declarations to 'nail down' method signatures.*

Suppose that you're producing a class for use solely as a base from which other classes will inherit. In other words, only instances of derived classes will ever be created, no instances of the base class itself. Now, suppose further that some service must be uniformly supplied by derived classes, but the parent class is in no position to supply a default service.

This is not an uncommon situation when designing a family of classes. What to do? One solution is to declare a virtual method with a skeleton implementation. The declaration is effectively an API definition for the class, even if the implementation is deferred to child classes.

The skeleton method implementation might consist of nothing between the method's `begin` and `end`. If you want to be more thorough, it might contain a single statement that raises an exception (with a message that the method should not have been called). This would protect against the method being erroneously called on an instance of the base class.

There's an easier way, however, and one that can be helpful to someone working on the code for the derived class. You may add the `abstract` designation to the declaration, like this:

```
procedure MyMethod; virtual;
  abstract;
```

This tells the compiler that the implementation is intentionally omitted, but an overridden method that does have an implementation will need to be supplied by a child class. Not only will the compiler not object to the missing method, it will cause an exception to be raised should an attempt be made to call the base class's method. Classes containing one or more abstract methods are sometimes called abstract classes.

You should avoid using abstract methods in classes for which instances may be created, however, even where the non-implemented method is never needed by instances of the class. The compiler helpfully generates warning diagnostics when an instance of a class that has an abstract method is created. This is not something you should impose upon the users of your class. If instances *will* be created, go the route of the skeleton implementation that raises the exception.

## Polymorphic Payoffs

*Contemplate how the polymorphic capabilities of TStream-derived classes give you leverage in writing your own code, then consider how you might provide similar benefits to users of your classes.*

If you've done much work using Delphi stream objects, you will possibly already be attuned to this guideline. Delphi streams are convenient, powerful and flexible. Especially flexible, in that anything that needs a `TStream` to work with will be equally content with an object of type `TFileStream`, `TMemoryStream`, `TStringStream`, or any other `TStream`-derived class.

This flexibility, if you've not realised it, comes to you courtesy of polymorphism. So ask yourself if you have an opportunity to offer the users of your classes similar benefits?

No doubt about it, inheritance and polymorphism give us class designers and coders a big boost in terms of code reuse. But there's plenty of wealth to share with our users. `TStream` classes are a great example of this. However, if you want a further lesson in how class implementers can benefit from polymorphism, there is an even better place to seek enlightenment.

## Poly-Powers

*Study TStrings and derived classes to truly understand the amazing power of polymorphism.*

We've now arrived at the heart of the matter. With the concept we're about to explore, once you truly 'get it', OO mastery is a downhill coast from there on out. But, I remember that it took quite a while in my own OO studies before this particular light switched on.

I'm convinced that much of the difficulty arose from poor examples used to illustrate the point. The first several OO texts I studied

always set the explanation up something like this:

You have a base class, TAnimal, and several derived classes: TDog, TCat, TCow, etc. TAnimal has a virtual method Speak (abstract or empty placeholder, doesn't matter). Each derived class supplies an implementation wherein a message is displayed: *Bow Wow*, *Meow*, etc. Now, we have an array of TAnimal objects, each element of which is assigned an object of one of the derived types. We iterate through the array, invoking Speak on each element, and experience all manner of animal commentary. Isn't that special?

Or, at least it might be if this sort of scenario was commonly encountered in real-life coding. But it's really not that common. I don't mean the business about silly animal classes (instead of something more realistic), but the need to hold a collection of sibling class instances in an array over which we iterate calling the same method for each entry. This kind of example just never made me react with more than a mild 'hmmm, interesting'.

I don't believe I understood the importance of virtual method capabilities until I dived into the code implementing TStringList and its base class TStrings. It was then the penny finally dropped. So, let's take a close look at these classes to see just how they do take advantage of virtual methods. This explanation is neither brief nor obvious, but I think your patience will be well rewarded.

TStrings and derived classes are, if nothing else, extremely convenient to use. Their basic charter is to store and manage an unlimited number of character strings; but additional services, such as concatenating all the held strings via the Text property, exist as well. The most familiar member of the TStrings family is TStringList. Users of it will very likely be familiar with the Objects property as well, and we'll have a closer look at that momentarily.

Members of the TStrings family do some moderately sophisticated processing, and much of the code supplying these services is found in the TStrings base class itself. But TStrings defers one very critical responsibility to descendants: interacting with the particular physical storage mechanism used by the derived class.

For TStringList, the responsibility involves maintaining an internal list of string references, much like maintaining an array of strings in application code. But consider TMemoStrings, the class used to access the text content of a TMemo object. Here, the class makes use of the Windows-supplied buffer of the actual Window control. This is just what's called for, given the buffer will be used to hold the text for display in any case.

What's the trick? It's the straightforward (but, oh so clever) use of virtual abstract methods in TStrings for processes that interact with the physical storage mechanisms. Specifically, TStrings declares two protected methods (Get and GetCount) and three public methods (Clear, Delete and Insert) that are virtual abstract. These routines are responsible for getting the string data in and out of the storage used by the particular derived class. TStrings declines to provide a default implementation of the routines, that's left as a requirement for descendant classes.

However, since the five methods *are* declared, they are available to be called from the 40 (or so) other TStrings methods that are implemented. As a result, those methods (like Add and Exchange) can supply eminently useful services, even though the management of physical storage is left unspecified. That, my friends, is polymorphism delivering the real goods!

Before we leave this topic, we should also consider the Objects property of TStrings. There's a very different motive at play in this case. The designers of TStrings presumably wanted to allow descendant classes to optionally implement the capability, but unlike the case with physical string storage, did not want to require an implementation. So, in this case, we see virtual routines (notably GetObject and PutObject) supplied with skeletal 'placeholder' implementations. But the point is that there *is* an implementation in this case.

Classes derived from TStrings *must* provide implementations of the abstract methods if things are to work. But supplying an overridden GetObject, etc, is entirely optional. For a derivative class like TMemoStrings, for which an Objects property would rarely be useful, no overriding is needed, and no harm results from this absence.

There you have it. This has been a wordy explanation, but we have just dealt with what I believe is the most elusive concept required to truly understand software object orientation. If it still seems a little confusing, then re-read this section, and keep staring at the code in the TStrings class family. When you do understand it, you can honestly and proudly call yourself an object oriented engineer.

## Personality Plus

*When designing a class whose responsibilities are growing to the point where multiple 'personalities' start to emerge, consider factoring out common traits into a base class, and derive each individual personality from that base.*

A Swiss Army knife may be the perfect tool to take along when venturing off into the wilderness. But amid the comforts of home, you'll better appreciate an actual pair of scissors when mending a garment, a corkscrew at the dinner table, a screwdriver when installing a board in your PC, etc.

Classes are often no different. The more focused they are on a specific task, the more efficiently they can usually be employed. If you're sceptical about this, just consider our old friends, the various Delphi stream classes. Suppose that rather than a TFileStream, TMemoryStream, etc, we just had a TEveryThingStream.

At a minimum, method calls would require more parameters. But I'd also bet that what is easily done with the specific stream types would be a good deal messier with the 'universal' version.

Object oriented design can be immensely rewarding when you have a good command of the techniques. Just don't get carried away producing Swiss Army classes. Have even more fun producing adroit, purposeful, classes. Where those classes share common characteristics, the factoring out of common services into a shared base class is a potent technique.

## Inheritance As A Partitioning Tool

*Feel free to use inheritance as a construction technique even when you are just implementing one class.*

When faced with a complex design challenge, a good software engineer will normally look for ways to decompose the problem into smaller, more manageable, pieces and to compose the solution of elements that address the individual pieces of the problem decomposition. This is standard operating procedure for those who are accustomed to thinking in an object oriented fashion.

Try to picture the composition process as proceeding in one or both of two 'dimensions'. Use of multiple classes to affect a solution ('componentization' if you will) suggests a horizontal composition. Employing internal helper classes to produce a more complex class (aggregation) suggests a vertical composition. With inheritance, we have one addition 'dimension' along which the solution may be structured.

Suppose you were assigned with the task of supplying a Whirring Whirling Widget. Now that sounds like a design challenge if ever there was one!

Even though there is only a single object type needed to fulfill the requirement, a perfectly viable approach could be to use inheritance. Start first by designing and implementing the fundamental widget-ness in a class that attempts no more than that.

Using that class as a base, derive another class that supplies the whirling. Then, once again, that class can be the base of the final class, in which you introduce the whirring.

In approaching the design this way, you will *not* enjoy the benefits of writing less code. There's only one final class and so there's no opportunity to take advantage of code re-use. In fact, in all likelihood, you'd have a tiny bit more code than you would in a single monolithic class.

But you still might accomplish the job faster by decomposing the problem in this natural way. For example, if everything were working just fine, except the whirling was a bit irregular, you'd know exactly where to look to diagnose and fix the problem.

Of course, if your users were later to approach you saying: 'Your widget's brilliant! Is there any possibility you could also build us a Whooping Whirling Widget and a Warbling Whirling Widget?', it'd be ages before that smug expression disappeared from your face.

## Dynamic Methods

*Dynamic methods look exotic and important, but the chances you'll ever actually need to use them in preference to virtual methods are slim.*

If you've spent any time studying classes in the lower reaches of the VCL hierarchy, you'll likely have encountered an alternative to `virtual` methods known as `dynamic` methods. Perhaps you've thought 'Hmm, the Borland engineers use these things all over the place in the VCL, but I don't use them anywhere. Maybe I'm unenlightened, maybe I'm missing out.' Relax, because you're neither.

Consider that Delphi first came on the scene at a time when Windows 3.x was the most common desktop OS (and there were a fair number of DOS machines still in service at the time, as well). The 16-bit addressing of the Windows of the time imposed some very serious difficulties where large data blocks were concerned. Specifically, addressing storage areas larger than 64Kb was a major hassle. As a result, all kinds of inventive devices were employed to conserve storage. Throw in the fact that RAM was incredibly expensive compared to what we enjoy today and there was an even greater motivation to conserve.

I'd be willing to bet that dynamic methods were one of those inventions. You'll hopefully remember from last month the fairly straightforward way in which virtual methods are implemented in Delphi. A table of virtual method addresses (the VMT) is maintained for each class. Furthermore, the VMT of a parent class becomes the first part of the VMT for a class inheriting from it. If one has a base class with many virtual methods, not only will the VMT of that class be large, so will the VMT of every class for which it's an ancestor.

Although VMTs offer a fast lookup mechanism, they can be wasteful of space in a situation like that just described. Eliminating this waste is the very purpose of dynamic methods. Without going into the fine details, suffice it to say that dynamic methods trade fast performance for conservation of storage.

Whereas the address of a virtual method will appear in every VMT of a class or its descendants, a dynamic method address will appear only in the class information of a class where the method first appears or is overridden. But the tradeoff is that the instruction path length for calling a dynamic method can be considerably longer. Thankfully, RAM is relatively inexpensive and abundant these days, so stick with virtual methods. The case for using dynamic methods is far more difficult to make now than it was in the dark ages of 16-bit Windows.

One final note: message handler methods (which are something we are going to look at in the final instalment of this series) have a lot in common with dynamic methods. Do not get them mixed up, however, because message handlers *are* something you will want at your disposal from time to time.

## Constructors Revisited

*Understand the unique personality of constructors to leverage their capabilities (or at very least, to keep yourself out of trouble).*

There's a special kind of method in Delphi (as well as most OO languages) called a *class* method. What makes a class method different from regular methods is that, with a regular method, one needs a valid instance of the class in order to invoke the method. A class method may be called with either a class type or object reference, as seen below.

```
StorageLength :=
  TMyClass.InstanceSize;
StorageLength :=
  MyObject.InstanceSize;
```

So it makes sense that constructors are a kind of class method, does it not? After all, how could we call a constructor to create an instance of a class if we needed a class instance to call the constructor?

As it turns out, that notion is half correct. Constructors in Object Pascal are unique and somewhat unusual hybrid methods. The compiler will generate different code depending on how the call is coded. To create a class instance, the constructor is appended to the class type name (or a class type reference). In this case, storage for the instance must first be allocated. This is accomplished by first calling the `NewInstance` method for the class (`NewInstance` being a class method itself, by the way), whose job it is to acquire the storage. The other notable behavior of the initial constructor call is that the address of the instance data block is returned like a function result.

If an instance variable is explicitly used rather than the class type name (or if the instance reference is implicitly `Self`), the compiler generates code that's just like a regular method call, in which the instance pointer is appended to the parameter list. This type of call will almost always be calling an ancestor constructor using the `inherited` keyword.

There's an interesting possibility arising from how all this works. If your constructor fully initializes the member data in the class, calling the constructor on an existing reference can be done to re-initialize the class. I'm not actually advocating that you ever code such a thing. For heaven's sake, create an `Initialize` method, even if it's a bit more work, so the poor maintenance programmer who might later encounter this won't think you're a madman. The point is that if you understand why you could do this, then you have a pretty good idea what's happening during the object creation process.

Object Pascal offers one more somewhat unique quality with respect to constructors: they may be virtual. But why would you ever even want a virtual constructor? If you want to create a class, surely you must know what kind of class you want to create, don't you? Well, the Delphi component streaming machinery that dynamically creates component instances doesn't know ahead of time. Thus, the language was given this somewhat unusual ability. However, the result is that you must be on your toes when declaring a constructor: where the base class's constructor is virtual, your declaration will need the `override` directive.

## Inherited Properties

*Take advantage of the capability that allows you to reference to hidden properties in base classes.*

Calling upon ancestor class services using inherited method calls is widely practised. It's not a Delphi-only technique, but it's something done in many OO environments. That you can access inherited properties in the same way is not as widely known. But the fact is you *can* do it, and the code is completely straightforward, eg:

```
Result := inherited MyProperty;
inherited MyProperty :=
  'Some Value';
```

The question is why or when would you need to do something like this? There's no standard answer, but the situation will probably involve inheritance from outside the class family (ie, the base class was not designed by your team). You may just need to slightly change the semantics, or you might need to actually change the type of the property. We needn't spend a lot of time on this guideline, because knowing thsat the capability exists is the main point here. But let me offer a simple case study where this is useful.

Suppose you're developing a set of custom Windows controls (yes, we're talking about components here, but never forget that components are classes). Let's say that you've got some special treatment of caption text: a capability to let the customer define synonyms for business terminology via an external dictionary. If you're inheriting your classes from VCL classes (eg `TLabel`), then the base class has a `Caption` property used for specifying the text to display.

At the same time, you don't want to impose unfamiliar property names on the users of your components. You'd like them to interact with the familiar `Caption` property in your class which can be externally manipulated, while the underlying label (or whatever) `Caption` gets set with the inherited reference. It's really very simple.

## Missing In Action

*There is no multiple inheritance in Object Pascal: get over it!*

Object Pascal, like a number of other perfectly respectable OO languages, limits class inheritance to a single parent class. Some languages, notably C++, offer the possibility of more than one parent class. Having followed the Delphi newsgroups over the years, I've observed that it's rare for any extended period of time to elapse without someone bemoaning the absence of multiple inheritance (MI) as a deficiency of Object Pascal.

In case MI is a new concept to you, let me just give you a hint about when it might be useful. If you have a non-trivial class family, classes at the end of the inheritance chain (final classes) are highly dependent upon their ancestry. If you want to graft additional behavior on to a final class, and that behavior is implemented in a class not in the ancestry, then you are stuck. You cannot just

blithely switch parent classes without breaking everything in sight. MI sometimes offers the way to accomplish the task.

But even in the C++ community, where use of MI is not uncommon, there are many who argue that its benefits are overshadowed by its shortcomings. I've no intention of resurrecting that debate here, other than noting that there seem to be a majority of Delphi practitioners who are perfectly content with things just as they are.

However, if you are a refugee from C++ who is having trouble adjusting, there is an alternative that is generally regarded as preferable to MI, and there is a second alternative which is even easier, which most people seem to overlook entirely.

The first alternative involves using Delphi interfaces. Although these would appear to have been introduced into the language to support Microsoft's COM technology, their application for non-COM purposes is now becoming routine practice. An explanation of how interfaces are a serviceable MI alternative is considerably beyond the scope of this discussion. But the topic has been well covered elsewhere. In fact, it was the subject of my very first article for *The Delphi Magazine*: Issue 38, August 1998 *[Available on our excellent value Collection 2000 back issues CD-ROM, along with all the other articles from Issues 1 to 60! Ed]*.

The second alternative seems obvious to me, although I've never seen it recommended in this context. Any Delphi object can send any other Delphi object a message. If the recipient object is equipped to handle the message, it will. Messages don't implement any kind of inheritance, of course. But there are situations where the method calls in an MI setting could be replaced with messages, and in doing so, the need for invoking services via those methods (and thus the need for MI) would disappear.

We'll look at Delphi messaging more closely in the final instalment of this series.

## Friendly Advice

*Compiler hints and warnings are your friends: heed their advice.*

Nowhere is it more important to heed compiler warning messages in Delphi than in those areas associated with inheritance. For example, it's easy to forget to provide the `override` keyword on a method declaration. Things would work very differently in this case and an oversight like this could cause some exceedingly perplexing behavior.

In my own code, I have a zero tolerance policy towards warnings and hints, and I know of more than a few seasoned pros that feel the same. Given that warnings and hints are almost always trivial to suppress with a tiny code modification or two, there's no excuse in my book to allow them to persist.

This is especially true of presumably professionally produced classes or components. If I'm evaluating a third-party component library and find that the code produces hints or warnings, I'm immediately sceptical about the overall quality of the product. This isn't to say that it's cause for automatic rejection. Unfortunately, the last time I needed to recompile a couple of Borland VCL units, I encountered... you guessed it.

Back to the subject of inheritance: one technique for eliminating a warning is noteworthy here. If you 're-declare' a virtual function without the `override` keyword, you're stating that the method is static. If you mistakenly code `virtual` when you meant `override`, you're stating that the method *is* virtual, but it's not a continuation of the virtual 'lineage' from class ancestors; with your new virtual declaration, you're starting a new branch of virtual methods.

Once in a while you intentionally need to reintroduce a method named the same as an ancestor method. The `reintroduce` keyword was added to Object Pascal (in Delphi 3 I believe) to allow you to tell the compiler: 'I'm doing this intentionally, so don't produce a warning in this case'.

Before we leave this subject, consider another error that's all too easy to make: specifying `overload` rather than `override`. Depending on who you ask, `overload` is either eminently useful or it's an evil enhancement to the language. Either way, it has nothing to do with inheritance, so, once again, heed the compiler warnings.

## The Grand Design

*To understand the power of inheritance, you'd have to look long and hard before finding a better role model than the VCL itself.*

The Delphi VCL is an extraordinary piece of software engineering with respect to what can be done with class inheritance (even if it won't compile without producing warnings and/or hints). You could benefit immensely from a study of the VCL, but there's a trick in how to approach it.

The VCL is vast, and it's more than a little intimidating when you first start exploring. When studying the VCL, one needs (especially at first) an action plan regarding what you're going to look at. Without it, you're at considerable risk of getting lost, if the intimidation factor doesn't get to you first.

Let me suggest an action plan for studying VCL inheritance: pick a familiar window control component, like the `TButton`. Peruse the class declarations between `TObject` and `TButton`. Stick only to those classes, don't allow yourself to be sidetracked. Even if you stay on course, you'll be visiting four separate units: `System`, `Classes`, `Controls` and `StdCtrls`.

You'll encounter some things dealing with component management that may make little sense, especially when you're low in the hierarchy in `TPersistent` and `TComponent`. Don't worry about it. What you want to be looking at on this trip is how the capabilities of the classes are introduced, one layer at a time. One of the things that make the VCL such a compelling subject for study is the number of levels employed. In the case of `TButton`, we have seven classes involved counting `TObject`

at the start of the inheritance chain and `TButton` at the end.

This isn't to suggest that many inheritance levels are something for which to strive. But consider just how much functionality there is in Delphi button, from the standpoint of being a component with streaming capabilities, the standpoint of being an encapsulated window control, and everything in between. There's an incredible amount going on here, even for a lowly button.

The VCL just *feels* right to me in this regard. Even after more than five years of working with it, I am still a little awestruck by how well it all fits together.

## Next Time

We're in the home stretch. After a month off (I need a break too!), the series will return with the concluding instalment in which we'll look at a number of diverse topics that don't merit an entire article, but which are nevertheless important class engineering considerations in Delphi.

David Baer is Senior Architectural Engineer at StarMine in San Francisco. The fact that this article, which achieved a world record high prose-to-code ratio, was written at the same time the Olympics were being held was purely coincidental. Contact him at dbaer@starmine.com