Effective Delphi Class Engineering Part 3: Skyrocketing Property Values

by David Baer

Every so often I look across the river at the massed forces of the Java army and I can't help wondering if my career is on track. Maybe it's time for a change. Java has a lot going for it. Java books certainly command the lion's share of shelf space in the programming section of bookstores, not to mention the number of Java job opportunities (compared to most everything else). But, then I remember that the Java language doesn't have properties, and I quickly return to reality. Voluntarily give up properties? No way!

Properties are one of the most elegant and powerful elements of the Delphi object model. Casual Delphi users know them as those convenient devices that can be used for design-time value assignments in the object inspector. Their role in this context is highly important, playing a major part in the automatic component streaming services for components on forms and data modules.

But the utility of properties is hardly limited to just components. They're one of the class designer's most potent tools. We're going to look closely at class properties in this instalment and, if you're unfamiliar with them, you're in for some very pleasant surprises.

Basic Concepts

Don't declare public data members in your classes: provide properties instead.

The syntax of property declarations is quite straightforward. Properties offer a way to make what appear to be public class data members available to class users. They also have an array-like capability, which we'll explore further on. For now let's concentrate on the non-array property type.

Although properties act just like public class data items, they have an important quality in that your class may be much more in control of things than it would be with a public data member. Before we discuss why, take a look at the property declarations in Listing 1.

Here we see two properties declared. `MyProp1` is a publicly accessible `Integer` 'proxy' for the private `Integer` data member `FMyProp1`. `MyProp2` looks like a publicly accessible `Integer` to the class client as well, but it's actually accessed via two methods: function `GetMyProp2` for reading the value and procedure `SetMyProp2` for writing the value. Whether a field or method is specified, the 'get' designation is formally called the read specifier, and the 'set' is called the write specifier.

The example follows VCL convention, in which a field referenced in a read and/or write specifier is usually given the same name as the property itself, but prefixed with an `F` (for *field*). The read function is usually named `Get` followed by the property name, the write is named `Set` followed by the property name.

The get function returns a type matching that of the property, and the set procedure takes a single parameter of the same type as the property. The example doesn't show it, but you can freely intermix field and method specifiers. Designating as field-type read specifier and a method-type write specifier is very common practice.

If you examine the formal OP (Object Pascal) documentation on properties, you'll see there are other specifiers available in a declaration: `default` and `stored`. These are relevant for published properties of components *only*. In particular, you should avoid a very common mistake: the `default` specifier does *not* offer a convenient way to initialize a property. If you need an initial value for a property field, assign it that value in the class's constructor.

It's the availability of the read and write methods that make properties such a compellingly useful tool. We'll examine a number of aspects of these capabilities as we proceed, but let me offer two preview examples. A read method can return a calculated value on demand that otherwise might have to be continually kept accurate if, instead, a public field were used to provide the value. A write method can monitor value settings to ensure compliance with some constraint. So, the class client benefits from these services, yet referencing code may be simply and clearly written, as if plain variables were in use.

So, you may be thinking, 'Well, I can see that read and write specifiers are powerful, but I don't understand the guideline. If I have a field-type read and a field-type write specifier, what's the difference between that and just declaring a public data member instead? They're the same thing as far as the class client is concerned!'

They would be, except for one important nuance. Properties (with either type of specifier) are

➤ *Listing 1*

```
TMyClass = class
private
  FMyProp1: Integer;
  ...
  function GetMyProp2: Integer;
  procedure SetMyProp2(
    Value: Integer);
public
  property MyProp1: Integer
    read FMyProp1 write FMyProp1;
  property MyProp2: Integer
    read GetMyProp2
     write SetMyProp2;
end;
...
// examples of use in code
if MyObject.MyProp1 = 0 then
  MyObject.MyProp2 := 0;
```

not allowed to be used as var parameters in procedure calls, and their address may not be obtained with the @ (address-of) operator. Both of these operations would be incompatible with a property having a method-type specifier.

This is rather astute foresight on the part of OP's designers. Your class may begin life with a simple requirement for a field-type specifier, and then you may later discover that a method-type specifier is needed. By restricting var and @ usage for both specifier types, you're free to change your implementation at any time without breaking existing client code.

```
private
  function GetPixelColor(X, Y: Integer): TColor;
  procedure SetPixelColor(X, Y: Integer; Value: TColor);
public
  property PixelColor[X, Y: Integer]: TColor
    read GetPixelColor write SetPixelColor;
...
// examples of use in code
MyBitMap.PixelColor[I, J] = clBlack then
  MyBitMap.PixelColor[I, J] := clWhite;
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
private
  function GetItem(I: Integer): Pointer;
  function GetItemByName(const Name: String): Pointer;
  procedure SetItem(I: Integer; const Value: Pointer);
  procedure SetItemByName(const Name: String; const Value: Pointer);
public
  property Item[I: Integer]: Pointer read GetItem write SetItem;
  property ItemByName[const Name: String]: Pointer
    read GetItemByName write SetItemByName;
...
// examples of use in code
if MyList.ItemByName['Fred'] = nil then
  MyList.ItemByName['Ethel'] := nil;
```

Hands Off The Merchandise!

Use properties to provide read-only access to class data.

This could hardly be simpler or more obviously useful. If you omit a write specifier, a public property becomes a read-only property to class clients. Granted, you could provide a public function of the same name, and class clients would have a semantically equivalent read-only identifier for supplying the value. However, using properties for read-only items is preferable: it's consistent with their use for read-write items.

A more interesting proposition is the notion of write-only properties. If you provide a write specifier, but no read specifier, you have just this. Admittedly, the need for write-only properties is quite infrequent (but a good case might be made for passwords, for example).

Array Properties

Supply parameterized properties for array-like property value access.

OP extends the capability of properties by offering a capability formally known as array properties. I want to suggest also thinking of them as parameterized properties here, because I believe the designation 'array' is conceptually limiting. Certainly, array properties can offer array-like access to class information, but there's much more they can do.

Listing 2 shows the declaration of a read-write array property (omit the write specifier and we have a read-only property). There's nothing mysterious here, you should be able to deduce what's intended and required just by looking at the declaration.

There are several important things to note. First, array properties have one or more parameters that are often simply numeric indices in one or more dimensions. We'll see in the next item that there are more interesting parameter capabilities available than array-like subscripting. The other important point is that array properties may only be accessed using method-type specifiers. The simpler field-type specifier available to non-array properties is not an option in this case.

Array property read and write specifiers are straightforward. The read method requires a parameter list of the exact types (and in the same order as) the property parameter list. The write method requires that same parameter list, to which is appended a parameter for the value being assigned.

Array properties may be used in the obvious way: the class has an array of values which it wants to make public for read or read-and-write purposes, and we can use an array property to do this. Until Delphi 4, we had nothing but fixed length arrays available in the language. These were normally of little use to class designers due to lack of flexibility arising from the fixed length. As a result, class writers learned to rely on lists, which could be dynamically sized to accommodate their needs.

The TList class (in the Classes unit) offers an array property Items for easy, subscript-like access to items in the list. I recommend you study this class (be forewarned that it does not use the conventional property naming conventions I mentioned earlier), and not just to observe how it uses properties. TList is a utility class that you'll likely find yourselves frequently using as an internal helper class. As such, you cannot know it too intimately. With respect to array properties, you will often want to 'surface' the items in internal list objects as public properties for your class clients to access.

Values by Name

Use strings or other non-integer values as property parameters to give your class elegant ease-of-use.

Parameterized properties get really interesting where the parameters are something other than subscripts. In particular, use of a 'name' for the parameter can be very convenient to your class users. Listing 3 has an example.

Rather than require the user to look up a subscript based on the name in some table prior to accessing a property value, the get method provides the service itself. Note in the example that const is specified in both the property parameter list and the read and write method to optimize performance. The compiler demands consistency here (`const` specified in both the property and method parameters), just as it would between method declaration and implementation parameter lists.

Parameterized properties using non-subscript type parameters bring us to a point where we can consider an interesting possibility. When we think of array access, we ae usually dealing with non-sparse arrays. There's an entry there for each item in the low-to-high range in each dimension. But, with array properties, we needn't limit ourselves to this.

Consider instead the following scenario. We have a 'dictionary' lookup property, using a word (eg a name) as an association to a value. In an assignment to the property, if the name exists, we assign it the new value. If it does not exist, we add a new entry to the 'dictionary'. For retrieval, we return the value of an item when found, and either return some innocuous value when it's not found or raise an exception, as appropriate to the problem we're solving. Powerful stuff, is it not?

Labour Saver For Your Users

If you have one or more array properties, consider designating one of them as the default property.

If you followed my advice and studied the declaration of `TList`, you may have noticed the `Items` property was declared as `default`:

```
property Items[Index: Integer]:
  Pointer read Get write Put;
  default;
```

First of all, this is not to be confused with the default specifier I previously cautioned you against misinterpreting. The reserved word `default` has two uses in property declarations. In this second one (where `default` appears after the semicolon in the property declaration), it means that the class client code can reference property values using a shorthand notation.

By declaring a default property, you allow your class users to access that property with an object reference, followed by the parameters. The property name need not be given. Thus, if `MyList` has a reference to a `TList`, the two statements below are equivalent:

```
MyList.Items[0] :=
   MyList.Items[1];
MyList[0] := MyList[1];
```

One final thought. The `default` designation is only allowed for array properties. If you don't immediately understand the reason for this, consider what code referencing a non-array default property value would look like and pretend you're the compiler.

Consistency

Use common sense and try to be consistent when deciding to supply a parameterized function vs. a parameterized read-only property.

As suggested earlier, a non-array read-only property and a parameter-less function of the same name would be semantically equivalent to your class users in a practical sense. If you changed from one to the other your users would be unlikely to notice the difference.

This is not the case with read-only array properties versus parameterized functions. Clearly, the former require brackets delimiting the parameter list and the latter need parentheses. Of course, if your users make a mistake and code the wrong delimiters, it shouldn't take them too long to determine what's needed to correct the problem.

But this does bring us to what I've found to be an occasional dilemma. Some kinds of information clearly are appropriately made available via properties. The `Items` property of the `TList` is a perfect example. `Items` is a *property* of a list in the general sense of the word (ie, an attribute or current characteristic), and formalizing that quality by declaring it a `property` is entirely reasonable.

For other kinds of information that a class might provide, supplying it via properties can be contrary to the spirit of things. A class may well have access to data that allows it to supply information, but that information may not be appropriately considered an actual property (again, in the more general sense of the word).

Sooner or later, you'll probably run into a dilemma where the information is in a grey area. It's kind-of, sort-of, a property of the class, but it's not clearly so. Let me cite an example from a real experience.

I once designed a container class that was a kind of a list, but it had two 'views'. Items in the container could be filtered, and filtered items would not be seen by collaborating classes. Unfiltered items, then, were visible items in one view. The other view was the physical view, where all items were present and accessible.

Class client code usually dealt exclusively with one view or the other. So, I provided the properties `Items` and `VisibleItems`. There were occasions, however, where client code needed to map from one view to the other.

To accommodate that need, I supplied two mapping functions: `IndexOfVisibleIndex` and `VisibleIndexOfIndex`. Should these have been properties? I still don't know if I made the best choice. To me, this is one of those 'grey area' cases and arguments could be made for supplying this information via properties *or* functions.

So, although I cannot offer a nice, easily remembered guideline here, I would suggest that you at very least try to stick to a course that's consistent. If you vacillate, you'll likely end up with some frustrated class users.

Tough Love

Don't hesitate to incorporate appropriate safety measures in your set methods to assist your class users in staying out of trouble.

One of the great things about a number of modern programming languages is the seamless support for exceptions. OP is no slacker in this area. Exceptions can (and frequently should) be used liberally to protect your class users from their own errors. Certainly, one of the main venues in which exceptions are indispensable is in write specifiers. If the class user can assign a property value that will cause a class instance to malfunction, the class needs to guard against the possibility. Exceptions are usually the best way.

They are especially appropriate in the write methods of array properties. We'll often want to check that valid parameters (such as index subscripts) are being specified. This brings us to several design tradeoff decisions. How much protection is needed in relationship with the expected performance of class services? Bounds checking consumes CPU cycles and, if optimized performance is high on the list of requirements, we must be thoughtful about this.

A related concern has to do with the quality of the information returned in an exception message. If we have a subscripted (or indexed, if you prefer) property, this will frequently be there for providing access to items in some internal list (a `TList`, `TStringList`, etc). Should we do a check on the index in our class when the internal helper class will duplicate that test? The answer returns us to the issue of performance expectations versus class usability.

In general, the more specific an exception message is, and the more pertinent the information included, the better it will be for your class users in debugging their code. In the case of an internal list, they shouldn't be required to know the implementation details of your class. If they need to diagnose an exception raised in an internal helper class, something is wrong.

So, what to do? My first inclination is usually to ignore the performance requirements and be especially generous in providing plenty of checks with nicely detailed exception messages. My motives aren't exactly unselfish. In my experience, the more information a user gets, the better the chance that your class will not be accused of having bugs.

Beyond that, it's stunning how fast ordinary business computers are today. Protection schemes like bounds checking can frequently be incorporated with no observable performance degradation. Furthermore, there are relatively few cases where performance surprises occur. If a class is performance-critical, you usually know that early in the design stage. If it's not identified as an issue, then don't worry about the overhead of your protection strategy.

Where you do have performance concerns that merit extensive optimization efforts, there are other solutions: conditional compilation (I recommend this if you like unreadable code), Delphi assertions, etc. The strategy may depend on whether your class users will have access to the source. If they do, you've got many ways to address the problem.

But the main point is that this is normally not a major concern to begin with. For the most part, don't worry about the overhead of protection strategies. When it comes to class users, no news is usually good news.

Cool Running

Property read and write methods both offer great opportunities for introducing optimizations in your class; take advantage of them.

Let's begin with write methods. Setting a single property value will, in some cases, have a pronounced effect due to collateral state changes in your class. An easy optimization (used often in the VCL) is to test the new value against the current one. If the new value equals the old, the write method can bow out then and there, and a lot of potential work can be avoided.

On the read side of things, we can employ a just-in-time allocation strategy for classes that may sometimes consume expensive resources, but those resources are not needed by all instances of the class. Again, the VCL uses this to good effect, and it's instructive to consider how it does.

Window controls require handles to be operational. Although these are less of a precious resource in the 32-bit world, they are still not free. But Delphi originated during the waning days of 16-bit Windows, when well-behaved programs needed to be conservative in doling out handles. Enter the `TWinControl.Handle` property read method to offer a helping hand! Since not all windows end up in a state where they require a handle (for a variety of reasons we don't need to get into) deferring the allocation until the handle is actually requested can keep 'expenses' down.

This method (edited slightly from its current form for sake of clarity) is shown in Listing 4: it could hardly be simpler.

Another strategy for reducing resource consumption is that of caching. You can use your property read method to draw from a pool of pre-allocated resources (objects, memory blocks, or whatever). You may want to allocate new ones if the current pool is exhausted or wrest ownership from another instance that is the least recently active. The best approach will depend upon the nature of the problem, and the possibilities are many.

Private Property

Non-public properties can be of use to you in your class implementation; feel free to use them as such.

This is an extension of the previous item, and although briefly stated, it's important enough to merit its own guideline. As you just saw, using properties with underlying access methods can allow simple, concise code to perform mighty deeds. This capability isn't restricted to public services. You may use non-public properties internally in your class to avail yourself of these benefits in the class's implementation code.

Labour Saver For You

Save yourself some coding by using indexed `Get` *and* `Set` *methods.*

Another way to save yourself some implementation effort is supplied in the form of 'shared' read and write specifiers. The compiler allows you to specify read and write specifiers that are used by multiple properties. The only requirement is that you supply a unique number using the index specifier in each property declaration. In generating code to call the read or write methods, the compiler inserts that number in the parameter list for you automatically. For a formal description of the syntax, refer to Delphi help, looking up 'index specifiers'.

But you can probably just examine the example given in Listing 5 to fully get the idea. In that, we have class TMyBars that maintains four internal lists: FooBars, CandyBars, SandBars and WineBars. Each list has a uniquely named *items* and *count* read-write property. Using index specifiers, we can get by with just four methods, rather than the sixteen that would otherwise be required.

Class Discrimination

For class type properties, use write methods to assign as much (or as little) of the 'from' object as you need.

Those of you who read Brian Long's *Delphi Clinic* in the April 2000 issue have a head start on this tip. Brian explained how property assignments often work for TStrings-derived property types.

When your class has a property which is a class type (ie, the property can be used as an object reference), there are two main choices of how you will want the assignment to work. The first way is where the property is being used to maintain a reference to an external object. You will see many examples of this in the VCL where components collaborate (a TDBEdit object has the DataSource property of type TDataSource, for example).

In this case, we're not interested in acquiring the contents of the 'from' object, our object just needs access to the other. The requirements for the property value assignment are straightforward enough. But this scenario does suggest that a notification protocol be put in place wherein your object is informed if destruction of the referenced object occurs. You don't want your object attempting access after the other is no longer in existence.

The other main case is where your class uses an internal object, and an assignment to the property requires copying some or all of the state of the 'from' object (or setting the internal object to a virgin state if the assigned value is nil). A VCL example of this case is the Font property of various controls. The font is always an internal helper class in this situation, and assignment to the property copies the 'from' settings into the internal TFont object.

Here you have great latitude in how much you wish to pull from the assigned object. Invoking the Assign method will be exactly what is needed in many cases, but you are not limited to that. In fact, if the 'from' class is the same type (or is even defined in the same unit), you are not even limited to copying public data: your property write method may access the most intimate of details of the 'from' instance. As the class designer this is completely your call.

A third possibility exists, although it is less common. In this scenario, the assignment effectively transfers ownership of the 'from' object to your object. The 'from' object remains an external one, but your object assumes the responsibility of freeing the assigned object when your object is destroyed. For this type of assignment, it would be necessary to ensure that the write method enacts whatever protocol is needed to transfer the ownership. An object having two owners is of course an Access Violation in waiting, so an existing owner

➤ *Listing 4*

```
function TWinControl.GetHandle:
  HWnd;
begin
  if FHandle = 0 then begin
    if Parent <> nil then
      Parent.HandleNeeded;
    CreateHandle;
  end;
  Result := FHandle;
end;
```

➤ *Listing 5*

```
TMyBars = class
private
  FooBars: TList;
  CandyBars: TList;
  SandBars: TList;
  WineBars: TList;
  function GetBarCount(Index: Integer): Integer;
  function GetBar(Item: Integer; Index: Integer): Pointer;
  procedure SetBarCount(Index: Integer; Value: Integer);
  procedure SetBar(Item: Integer; Index: Integer;
    Value: Pointer);
public
  property FooBarCount: Integer     index 0
    read GetBarCount write SetBarCount;
  property CandyBarCount: Integer   index 1
    read GetBarCount write SetBarCount;
  property SandBarCount: Integer    index 2
    read GetBarCount write SetBarCount;
  property WineBarCount: Integer    index 3
    read GetBarCount write SetBarCount;
  property FooBar[Item: Integer]: Pointer   index 0
    read GetBar write SetBar;
  property CandyBar[Item: Integer]: Pointer index 1
    read GetBar write SetBar;
  property SandBar[Item: Integer]: Pointer  index 2
    read GetBar write SetBar;
  property WineBar[Item: Integer]: Pointer  index 3
    read GetBar write SetBar;
end;
...
function TMyBars.GetBar(Item, Index: Integer): Pointer;
begin
  case Index of
    0: Result := FooBars[Item];
    1: Result := CandyBars[Item];
    2: Result := SandBars[Item];
  else
    Result:= WineBars[Item];
  end;
end;
function TMyBars.GetBarCount(Index: Integer): Integer;
begin
  case Index of
    0: Result := FooBars.Count;
    1: Result := CandyBars.Count;
    2: Result := SandBars.Count;
  else
    Result := WineBars.Count;
  end;
end;
procedure TMyBars.SetBar(Item, Index: Integer;
  Value: Pointer);
begin
  case Index of
    0: FooBars[Item] := Value;
    1: CandyBars[Item] := Value;
    2: SandBars[Item] := Value;
  else
     WineBars[Item] := Value;
  end;
end;
procedure TMyBars.SetBarCount(Index, Value: Integer);
begin
  case Index of
    0: FooBars.Count := Value;
    1: CandyBars.Count := Value;
    2: SandBars.Count := Value;
  else
     WineBars.Count := Value;
  end;
end;
```

needs to have its ownership revoked.

Next Time

After a month's rest, we'll bravely venture out of the shallow end of the pool, exploring the deep waters of inheritance and briefly encounter polymorphism.

David Baer is Senior Architectural Engineer at StarMine in San Francisco. Although a man of property, he almost never feels like appending 'Esq' to his signature. Contact him at dbaer@starmine.com