Software Quality and Paradox for Windows: Coupling and Cohesion

By Vince Kellen
2/26/1994
vjk@kellen.net
http://www.kellen.net

Programming in Paradox for Windows is radically different than programming in Paradox for DOS.

If you have started programming in ObjectPAL, you might have noticed this. The environment is so different. We've seen bright, talented, experienced Paradox programmers look dazed, confused, bemuddled and break down and cry when working with ObjectPAL. Well, maybe not cry, but I'm sure we've heard them curse (or cursed ourselves).

And with so much attention paid to learning ObjectPAL, we often forget that writing *quality* software in ObjectPAL is also a horse of a different color. The reason for this, as will be made clear shortly, is that the Paradox for Windows environment requires thinking about software quality issues in a new way.

Take the issue of coupling.

In my last article, I discussed coupling in the Paradox for DOS environment. Remember, coupling refers to type of connections between modules (or, in ObjectPAL, methods and procedures). The more ignorant a method or procedure can be about the application, the lower the level of coupling. If a method or procedure just needs one or two bits of information to do its task, *and it doesn't need to know a tittle about any other part of the application*, then the module exhibits low coupling. An example is in order. The custom method in Figure 1 is an example of a simple, yet loosely-coupled piece of code:

Figure 1. A loosely-coupled method

```
method push( var stackArray[] AnyType, const thingToPush
AnyType ) Logical
   return   stackArray.addFirst(thingToPush)
endmethod
```

This method takes an item and "pushes" it onto a "stack." A stack is a data structure that behaves just like a stack of cafeteria plates. You can only put a plate on the top of the stack. Likewise, you can only take a plate off of the top of the stack. Creating a stack in ObjectPAL is child's play: it requires, at the least, two methods (push() and pop()), a resizeable array and at least one item to push on the stack.

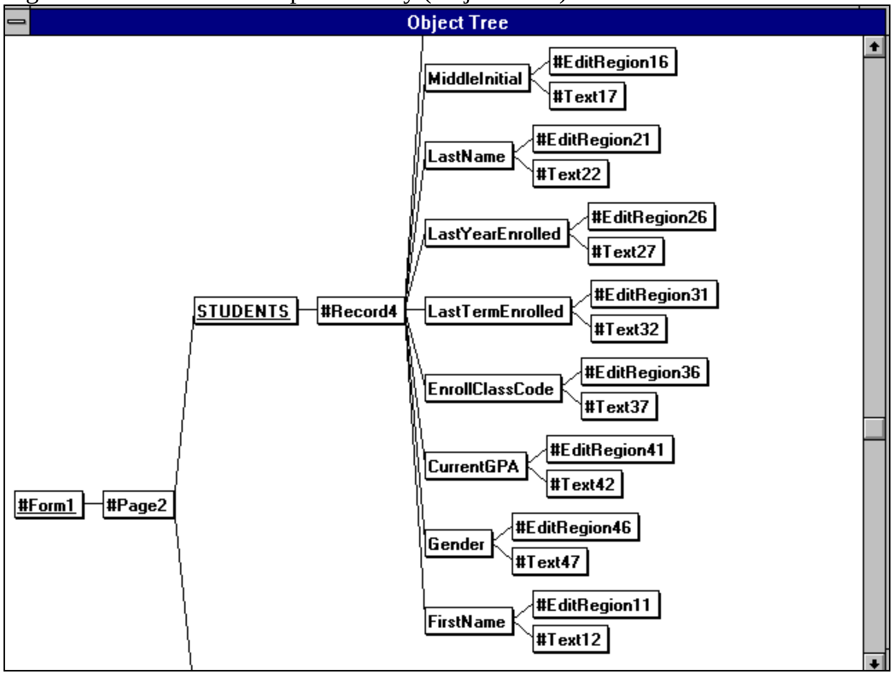
This method, push(), needs only two items, an array to hold a stack of items and an item to put on the top of a stack. This method requires no global variables, no tables, no text files, no UIObjects -- nothing else, to do its work. This is a loosely coupled method.

This is also a very trivial (although useful) example. In Paradox for Windows, one programs in the context of a form. A form has a containership of objects and this containership hierarchy has a lot to say about coupling.

Containership and Coupling

The containership hierarchy adds a new wrinkle to an old problem. Figure 2 shows a portion of a typical containership hierarchy. Objects to the right are lower in the hierarchy. Whenever you place code on an object, that code can automatically "see" procedures and variables. For example, if I declare a variable at the form object, any code placed in the STUDENTS multi-record object can use that variable. If the variable is declared higher in the object tree to a method or procedure, then the variable is global to that method or procedure. Global variables increase the level of coupling since that method or procedure needs to "know" about that variable to do its work.

Figure 2. The Containership Hierarchy (Object Tree)



While this is obvious to anyone experienced with ObjectPAL, it is quite a shift from DOS PAL. In PAL, variables where only visible to other procedures in the context of the procedure calling tree (or stack). In Paradox for Windows, variables are visible in the context of the object tree. Variable and procedure visibility (or scope) are strictly determined by how you arrange the objects in your form, not how you construct a hierarchy of methods. In fact, variables declared in a parent method are not automatically visible to child methods, unless they are explicitly passed as arguments.

Paradox for Window's object tree controls coupling. No longer can we programmers create *an abstract hierarchy of procedures in our minds* to control the level of coupling. Instead, we create *a physical, visual, tangible form with our hands* to control the level of coupling.

Think about this for a moment. This means that a good portion of coupling is resolved with our eyes as we review the object tree to see what variables and procedures are visible to our methods. I have heard a lot of new ObjectPAL programmers complaining about the fact that they can't find their code; that they have a hard time figuring out the visibility rules. Perhaps they are used to establishing a calling tree, DOS PAL-style, in their minds and working from that. In Paradox for Windows, we have the object tree which can tell us where our methods and procedures are located. The object tree also tells us exactly what variables and procedures are visible to what objects.

Since this information is conveyed visually through a concrete metaphor (the object tree) and since our minds can absorb and organize visual information much better that abstract or linguistic information, I contend that the following is true:

1. It is easier to determine and manage coupling in ObjectPAL
2. ObjectPAL code can withstand higher levels of coupling than DOS PAL code

Controlling Coupling

Any variable declared in any object's Var window increases the level of coupling in your form. If I attach the variable NextNumber to the form's Var window, then the level of coupling in my form increases. Why? Because every object in the form can now see, and use or misuse that variable. Since our goal is to write loosely coupled methods and procedures, we want as many of our methods and procedures to be *blissfully ignorant* of as much as possible. If we can get by without declaring variable NextNumber, so much the better.

On the other hand, if we declare the variable NextNumber either above or below a method declaration, then only that method can see that variable. This decreases the level of coupling.

This leads to the first rule about writing loosely coupled code in ObjectPAL:

Never declare a variable in a VAR window unless some other object or some other methods within the object need to use the variable.

Some of you might be tempted to place a variable in a Var window, even though only one method uses the variable, so that the variable can retain a value between successive invocations of the method, as in incrementing a variable. Don't do this. You can enlarge a variable's life span (called persistence) by declaring the variable above the method or procedure declaration. When you do this, the variable retains its value between invocations of the method or procedure.

Also, using readEnvironmentString() and writeEnvironmentString() also increases coupling since strings placed in the environment are global variables. These two methods are more than just a coupling problem; they can bee a coupling evil since the only type of data you can place in the environment is a string. This means that you will most likely need to cast the variable in the environment to some other type. Since the data type is not documented, other programmers (or even you) can more easily misuse an environment string.

Using libraries as repositories for global variables increases coupling as well. If you think that you can use a dynarray to reduce coupling, you are again wrong. A dynarray can be viewed as a collection of variables with a common ancestor. Figure 3 shows a small dynarray used as a "single" variable. Figure 4 shows a sequence of variables which serve the same purpose of the dynarray.

Figure 3. Using a Dynarray for Global Variables
```
var
   myDyn dynArray[] AnyType
endvar

myDyn["UserName"] = "Vince Kellen"
```

```
myDyn["SecurityLevel"] = 9
myDyn["Title"] = "Vice President"
myDyn["Company"] = "Kallista, Inc."
```

Figure 4. Using a Series of Global Variables

```
var
   userName,
   title,
   company String
   securityLevel SmallInt
endvar

userName = "Vince Kellen"
securityLevel = 9
title = "Vice President"
company = "Kallista, Inc."
```

Is there any coupling difference between using a dynarray and a series of variables? Potentially, yes. Since variables are somewhat self documenting as to their name and their type, there is less confusion regarding how the variable ought to be used. Since dynarrays can have any type, the can cause more confusion than regular variables. On the other hand, a dynarray places variables into a group. You can then pass these variables around together, as a group or record structure, which can facilitate the *implementation* of global variables. This *does not*, however, reduce the coupling. The presence of global variables means that somewhere, out there, lurks methods or procedures which are more than just ignorant, perhaps even devious. Our mission, as programmers of quality software, is to keep as many methods and procedures as ignorant as possible.

To control coupling even further, avoid the use of dynarrays as global variables and limit the use of global variables whenever and wherever possible. Granted, global variables come in handy and are often easy to use and maintain. Other times, however, they are not as they introduce subtle errors into your application.

Programmers often use global variables ought of habit. It is easier to use them then to devise a way not to use them. Just because its easier to use global variables doesn't make it right. I don't think that us programmers have given "global-less" programming a fair shot. We can always do more to conceive of solutions to problems that do not rely on global variables. We need to be spending more time making ObjectPAL code simpler and independent of any other part of the system, not more dependent. Enough said.

If you want to limit coupling even further, do the following:

1. Avoid the use of global variables in libraries or in the environment space.
2. Avoid the use of dynarrays when named variables will suffice.

Problems with the Object Tree

Despite the claim that the object tree is a superior method of depicting relationships between pieces of code, the object-tree approach in Paradox for Windows does have problems. Variables and procedures can be declared at any level in the tree: at the form, page, table frame or multi-record object (MRO), record or field. If variables are declared at each of these levels then the form begins to suffer from a large number of "semi-globals" or "regionals" as some people have called them.

Regional variables are variables which are declared at some object lower than the form. An excessive amount of regional variables can increase the coupling of the code in your forms. This does not mean that I would discourage the use of regionals. What it does mean is that if you have a form with lots of regionals, I would make sure that the following is true:

> Forms with a large number of regional variables and procedures are manageable as long as the regionals *are closely intertwined with clearly defined objects.*

Think about this for a moment. If your form contains several "smart" objects (although the term smart refers to a reusable object, perhaps it ought to be called "ignorant" object, in light of the fact that reusable objects, by definition, should be as ignorant of their environment as possible), then these smart objects are depicted in the object tree. And since these smart objects contain code with regional variables, your form can in fact contain a lot of regional variables without increasing coupling too much.

Understand that increasing coupling is not so much a bad thing as it is a necessary thing. Increasing coupling increases the risk that the application will have maintenance and defect problems. However, increasing coupling can sometimes let you do more sophisticated things. What we need to do is manage coupling appropriately.

No More Tables on the Workspace. Yeah!

In Paradox for DOS, tables on the workspace are a form of pernicious coupling. A huge amount of data can be shared between various procedures without any formal argument list, parameter passing or some other documentation in the code. Fortunately ObjectPAL automatically manages this aspect of coupling for us. We access tables through clearly defined objects: TCursors, Tables and UIObjects. Each of these object types have methods() which define (and document) how the data in the table is to be accessed.

No longer one module place tables on a workspace and leave them there for other modules to trip over. Instead, tables are bound to UIObjects and each UIObject has its own "interface" to the table. This interface is the various methods and properties which we can use to manipulate the object.

In addition, TCursors give us control over tables without introducing the problems of having tables open in a workspace, as in Paradox for DOS. TCursors are variables (objects) which fall within the confines of the object tree. The object tree then resolves the visibility of the TCursor.

Complexity and Procedure and Method Visibility

Interestingly enough, procedures have the same visibility as variables. Procedures declared on an object are visible to that object and all objects below it in the object tree. Methods, on the other hand are visible in two ways: automatically to the object it is attached to and all objects below it in the object tree; or manually visible to any object on any form, not just the current form.

Let's tackle these how the visibility of procedures affects our code quality. Since procedures are visible just like variables, then having an excessive number of regional procedures can increase complexity. Why? Because any object below the object which holds the procedure can use that procedure. This means that the lower level objects need to know more about their environment. Again, this is not a bad thing, and is very often used for good reason. What we need to be aware of is the fact that regional procedures require that programmers understand exactly what resources a given object has or needs to have at its disposal. This increases complexity.

This is not necessarily related to coupling. Coupling refers to data which must be exchanged between modules. Module visibility refers to resources which need to be shared between modules. The more resources modules need to share, the more complex your application becomes.

Method visibility is a more interesting case, because methods are automatically visible to lower level objects. These objects do not need to know where the method is located. The programmer simply invokes the method without any object declaration. Let's look at an example. Suppose the custom method in Figure 5 was placed in the STUDENTS MRO in Figure 2.

Figure 5. STUDENTS::colorIt() Custom method
```
method colorIt()
    active.color = Red
endmethod
```

To invoke this method from the LastTermEnrolled UIObject's arrive method, all you have to do invoke the method directly (Figure 6) from some built-in or custom method.

Figure 6. LASTTERMENROLLED::arrive() Built-In Method
```
method arrive(var eventInfo MoveEvent)
    self.colorIt()
endmethod
```

The custom method colorIt() is available to the arrive() built-in method without any object qualification. This is a good thing since the arrive() built-in method can be ignorant about where colorIt() is located. However, any other form can attach to the form which holds this code and can invoke the colorIt() method by knowing where it is located. This means that these "invader" methods need to know the internals to this form. These knavish methods are not so ignorant and having them around might not be such a good thing. However, it is not always a bad thing, either. In fact, you may need to exploit the fact that you can get at methods on other forms by explicitly knowing where the method is located. Just remember that when you do so, your code is a bit more fragile, because if the target method (or "prey" method) moves to another location within the containership tree, your "invader" method (or "predator" method) might lose sight of its quarry and break.

If you want to write good, clean, ignorant code (and I mean ignorant as a complement), then observe the following rule:

Avoid needlessly writing code which refers to custom methods by referencing a complete or partial containership path. Use ObjectPal's automatic method and procedure scoping rules to your advantage.

Cohesion: Is the JD Simple?

Modules with high cohesion are modules that have simple, easy to understand job descriptions. These modules perform a single task, or a groups of tasks which are highly related.

Paradox for Windows sort of forces high cohesion in one key area: event programming. Since Paradox has 28 built-in methods corresponding to at least 28 events, each built-in method handles a very specific and concrete task. The setFocus() built-in method makes sure the object is visually "dressed up" so the user can see that object is the insertion point for text or some action.

This is quite good for programmers, because it forces decomposition of the programming problem into a series of small, discrete tasks. However, the price we pay for this high cohesion is the hard work we need to expend to understand the Paradox event model. Knowing what snippet of code (meaning highly-cohesive snippet of code) to place into what built-in method requires a significant investment of time in learning all about Paradox's events.

Although having several, very cohesive built-in methods is nice, there is some unevenness in the gang of 28. Several mouse events are a bit too cohesive (if there is such a thing). Do we really need separate events for mouseRightDown, mouseRightUp, mouseRightDouble? Couldn't these be rolled into the corresponding regular mouse events (with some sort of flag set in the eventInfo packet) without losing much cohesion? Probably so.

The action built-in method is a bit more problematic. Through this built-in method flow about 140 different events grouped into five categories: DataAction, EditAction, FieldAction, MoveAction, and SelectAction. And since the action event is critical for many serious applications, it tends to get cluttered with gobs of code. Could the action built-in method be split into five built-in methods so that we can gain some more cohesion? Probably so.

Once you begin to create groups of custom methods and custom procedures, however, you are on your own. You will have to make sure that the modules you write are highly cohesive. In order to do that, you need to make sure that the module doesn't try and do too much.

If a method is charged with displaying a dialog box to prompt the user for data, convert the user input into appropriate selection criteria, query the data tables, display the answer table by opening another form and then letting the user print a report from after the form closes, that module is not very cohesive. It is responsible for 5 distinct tasks. Each of these tasks can be broken down into smaller, more manageable methods.

Summary

Paradox for Windows throws a few wrinkles into the coupling/cohesion problem. First, the object tree ensures that methods are coupled via global and regional variables. This coupling is determined when the form is actually created and is resolved *visually*. Rearranging the form can potentially rearrange the coupling relationships. Second, the object tree lets us resolve coupling relationships visually with our eyes, which means that our ObjectPal code can perhaps withstand higher levels of coupling.

Third, since ObjectPal is, for the most part, strongly typed, coupling is more explicit. Fourth, the event model and the built-in methods enforce a certain level of cohesion. Since the event stream is broken down into 28 discrete events or event groups, the code we write in response to a single event is usually very narrow in scope.

Fourth, although the object tree gives us a visual, concrete metaphor to resolve relationships between our code, it is not yet visually rich enough to locate things quickly and easily. Although you can see what objects have code attached, you can't quickly and easily inspect multiple object's methods or procedures simultaneously. Instead, you have to "pick up" and "turn over" each object, one at a time, (by pinning the methods dialog and clicking on each object) to see what code is crawling around underneath.

Finally, despite the improvements managing coupling and cohesion with the object tree and the event model, some problem areas still exist. The action() built-in method is not a very cohesive one. And any custom methods or procedure we write still require efforts on our part to ensure high cohesion. Object based programming is not a cure for bad programming habits. It can coax some better habits out of us, but it can't prevent us from writing bad code.

Hopefully, as Paradox for Windows and ObjectPAL matures, the programming environment will introduce new *visual* ways to enhance maintainability and comprehensibility to our applications. As these visual metaphors replace the abstract metaphors we have carried around inside our heads, our ability to reliably write and maintain complex code will increase. After all, visual cues (processed by our eyes and visual cortex in our brain) let us organize many more complex relationships than we can through either mathematical or linguistic symbols.

After all, this is supposed to be visual programming.