Lessons From A VFP Conversion Gone Bad - II

Whil Hentzen

There's plenty of work out there doing FoxPro 2.x to Visual FoxPro conversions (even FoxBase and FoxPro 1.0 to VFP), and that capability will continue to be in demand for another five years or more as companies try to leverage their investment in decades of existing code and data through 2020 and beyond.

I've been involved in a number of conversions, ranging from 500 to 3000 hours over the last few years, and have seen my share of successes - as well as a disaster or two. I'd like to share some my experiences.

Today, I'll take a different approach from most articles, which are often technical how-to descriptions. Everyone loves a good story, and everyone loves watching a train wreck (as long as they are out of danger themselves.) Thus, I'll tell a number of stories, each a miniature disaster, either from my own experience or from a peer. Finally, much like Aesop's fables, each story will conclude with a lesson.

Last issue's article evidently struck a chord with many of you. We've all shared a war story or two in the bar after a full day of conference sessions, but that's different than admitting one's goof-ups in print, to say nothing of developing a rough classification of areas prone to failure. And last article simply related data-related stories. So, we'll continue this issue with the other half of the chicken and egg quandary - code.

Before digging into the specifics, I want to review the multi-fold purpose of this article.

Some of these stories are just stories, offered to help you commiserate, so that you don't think you're the only one who has run into that situation, or, in some cases, the only developer who has pulled such a bonehead stunt themselves.) After all, software development is 90% technical expertise. The other 90% is having the right mental attitude.

Stories can also be instructional. Software development can sometimes be more art than science, and the broader your base of experience, the more likely you'll be able to solve a problem. That broad base of experience doesn't have to be learned in the first person; some of it can be supplied by talking to other developers. Some of you may remember the story tossed around at multiple conferences about the application that was running extremely slow. The story teller explained how he opened up the guilty form, and (I'm paraphrasing) found this line of code in the form's init:

```
select * from ITEMS into array AllItems
```

and then a few unrelated lines below, found this line of code:

```
select * from ITEMS into array AllItems
```

Naturally, further investigation found that the AllItems array was never used anywhere else in the form. The client was flabbergasted as how fast the form was the next time they used it.

Other stories provide a compendium of things to look out for when working on old code, so as to avoid land mines ahead of time. I'm not implying that this is the be-all and end-all checklist of danger zones, but they represent situations that I've run into time and time again.

Throughout, there is an underlying theme that cautions us that people can be unbelievable. The first draft of this article had a mean tone to it, born out of frustration and exasperation with situations that were at times close to inconceivable. (Say it with me, Princess Bride fans, "I don't think that word means what you think it means.")

To be sure, once in a while one runs into someone who really shouldn't be involved in software development. As the sign on my wall says, "If you don't know what you're doing, don't do it here." Yet even then, we need to remember that ofttimes the inept are not in their position due to personal malfeasance, but rather out of circumstances beyond their control - perhaps ignorance or desperation of management, who

simply needs a body, any body, to take over a sudden need in a project.

Thus, at the end, I'll sum up with a couple of stories that remind us that most of the problems we encounter with conversions are not technical ones, but people ones.

Finally, a reminder that you may not be learning anything new as much as being reminded about things that you knew once upon a time but haven't needed in a long time now. Some of the situations are things that we ourselves used to do, but have been deprecated, or we now simply know better. If they sound familiar, it's because we have simply forgotten that we used to do these things ourselves. "It seemed like a good idea at the time."

With those caveats, let's begin.

Warming Up – Tales From The Dark Side

These are the stories we tell at the bar after the formal conference sessions have ended. There are no lessons here, they're just silly.

I Don't Think That Word Means What You Think It Means

This one won't take long. I saw this in a comment one day:

```
* move all values into a 3 dimensional array
```

Quick, I ran back to my collection of "What's New In VFP n" books to find out when multi-dimensional arrays were added to the product. Not a single mention. So reading the rest of the code in the routine, I realized the developer meant

```
* move all values into a 3 column array
```

Nor Do I Think That Word Does What You Think It Does

I was debugging a particular bit of functionality in an app and did a search for a particular string that showed up in the error message being thrown. Found it in the middle of a PRG, and then began working backwards to figure out what was going on in the routine. At the very top of the program, I found this code segment (simplified for this example):

```
* This code was added after the plastics
* division was moved to a separate facility

lcOldLocation = facility.location
lcPrimaryLocation = ;
  FindLocation(lcOldLocation)

<some processing>

replace location with lcOldLocation, ;
  primary with lcPrimaryLocation
```

And the problem was that the value of lcOldLocation had changed. How could that be? Until I looked in the FindLocation function and found that it was changing the value of a similarly named variable, lcOldLocation. Naturally, that shouldn't happen, because the 'l' starting the variable name indicates that it's scope is local, right? Except that there wasn't a local declaration anywhere in the executing routine (or in the function.)

Finally, the original developer offers the explanation: "I thought that naming the variable with 'l' would make it local. That's what happens with private variables."

Except that, no, that's not what happens with variables that begin with a 'p', they just appears to work that way from the way they had always structured their code.

Result: an entire application's structure predicated on the assumption that variables that were named with a leading 'l' were assumed to be local in scope. As you can imagine, hijinks ensued.

I STILL Don't Think That Word Does What You Think It Does

When modifying and extending an existing system, my rule of thumb in determining which style of coding to use depends on who the ultimate maintainer will be. If I'm simply contracted to write a module that they customer will maintain, I think my code should look like the existing code, so that they don't have to restart their brain to work on my contribution. On the other hand, if I'm taking the system over, then I'll use my own conventions, because I need to be able to read it later.

Last fall, I was working on a conversion that fell halfway in between. They wanted me to convert the entire system, start to finish, but then they were going to maintain it, with ongoing support from me.

Since I was bringing them out of the 2.0 dark ages, I thought it best to train them with somewhat more modern techniques, including using Hungarian notation. The first delivery of course was met with disdain for the naming convention, despite it's regular (albeit not consistent) use throughout the existing 25 year old codebase.

"We don't like using Hungarian, it causes too many problems."

"Such as?"

"What if you have to change the data type of the variable? Then you have to go back through and rename all of the variables."

My explanation that if you're changing the data type of a variable in mid-stream, you've got bigger issues than renaming to worry about. Because you'll have to go through all of the code that uses that variable and see what the ramifications will be anyway. For example, the following line

```
messagebox("There are no machines with " ;
  + lcStatus + " components.")
```

will crash if lcStatus is changed from character to numeric. They understood that, and did make the necessary changes to code to handle datatype changes, but, somehow, they felt that the little bit extra work to tweak the name as well, so that it was readable later, was too much work. Alas, I wasn't able to convince them to go the final step with renaming, and wrestled with misnamed variables throughout the app for the rest of the project. I can't count how many times I got burned when dealing with their security system, for example, - gnRights and gcSecurity were both numeric values.

PPPPPP

For those of you who haven't spent time in the miltary, this acronym politely expands to "Prior Planning Prevents Positively Poor Performance".

Ran into a system with a complicated series of class libraries, including one collection of forms named so that the hierarchy is obvious:

```
f_0
f_1
f_2
f_3
```

It took quite a while to grok what the differences were, and I still have a post-it attached to my monitor reminding me about some of the subtle nuances. Then a new developer at the company added a fifth, subclassing from f_3 due to a new set of requirements. Except that, working in somewhat of a vacuum, he named it

```
f_04
```

and created a number of forms based on it. Then someone pointed out that in every list of libraries (File Explorer, Project Manager, etc), this:

```
f_0
f_04
f_1
f_2
f_3
```

was the order displayed. He, as many programmers are wont to do, refused to accept culpability, and wouldn't do the grunt work of renaming. Everyone else on the team refused as well ("It's not MY fault") and the libraries have looked like that ever since.

Variable Scoping Is Fun, Fun, Fun

Another example of their misunderstanding didn't cause program errors as much as confusion. Rife through the system were statements like this found in the 'Run' button's click() method:

```
select * where type = m.ltype
```

But where does m.ltype come from? It's not in the click() method. It turns out that it's been defined as a global variable, and then the value of a control on the form is bound to that variable.

Well, For Instance...

20-some years ago, I had build a fairly simple application for a customer and then turned it over to their in-house Fox developer, one of those 'legend in his own minds' fellows. I spent a day walking him through how it worked, gave him a copy of my Programming VFP 3.0 book, taking pains to explain how functionality in forms was provided by code in the classes for those forms, and that was that.

Six or seven months later, he runs into me at a meeting and explains that he had recently had the need to make some modifications to the app, and, damn it, he could NOT find the code that did a lot of the basic add/edit/delete processes, so he had to essentially rewrite my app to make it work. He was extremely unhappy with me and made it clear to everyone he could bend an ear about what a lousy job I did.

And then a couple years later, another chance meeting where he announced that he cleverly discovered this other set of files called classes, and THAT'S where all the code was! Imagine.

That type of misunderstanding happens more often than one would expect. I took over an system a while back where the application was based on one of the popular VFP frameworks. The framework had been extended substantially by another consultant, to the point where the entire system was class-based. Forms were collections of classes assembled together, with very little instance-specific code needed. I was, frankly, a little intimidated at how well the functionality had been thought through and abstracted, so that a few custom methods and the setting of a half dozen properties were all that was needed to implement a form. Every report form, for example, was assembled from a selection of

container widgets that provided functionality such as 'select one or all companies', 'select the span of years to query', 'include employees', 'include dependents', 'include retirees'. Some forms had a couple of custom controls, such as an option group for filtering the vehicle fleet being reported on.

Until it came time to start making modifications. Analyzing the operation of a report, with these standard widgets, required the tracing of code through three, four, five or even six levels of classes. Until I found why there was virtually no code in the report generation form itself. All instance-specific processing was handled in a giant case structure like so:

```
case ReportForm = 'RETIREE'
case ReportForm = 'DEBTS'
case ReportForm = 'LAYOFFS'
case ReportForm = 'PLANT ACQUISITION'
```

When I say 'giant', I mean tens of thousands of lines long, with hundreds of CASE statements.

And this instance-specific code was contained in one of the base class definitions.

Even better, all of the handling needed for the custom processing (such as the option group for vehicle filtering) was also done in the base class definition, in the CASE for that specific report.

Again, clearly didn't understand the use of class construction and subclassing. It took several months before my mind was trained to automatically look in the class defs for instance code particulars.

Belts and Suspenders

Working through the code in a system from a while back, I discovered that there were very few ELSE clauses in IF constructs, and absolutely no OTHERWISE clauses in any DO CASE structures.

When asked about the practice, suggesting that trapping for 'the case that will never happen' is a sound principle, I was told 'it never seems to be a problem for us'. Except that then for the next year I'd regularly hear from users about errors that would get thrown. They'd learned workarounds for those errors; when I started adding segments like

```
ELSE
  messagebox("Vehicle ID empty",
    "This will never happen")
```

into the branching constructs, well, what do you know?

Lesson: You know this one. Build an idiot proof system and the customer will hire a cleverer idiot.

The Hard-Coded Developer Login

More than one system I've run into (including my own, of course) has had a secret developer login. This login provides access to special developer-only features, such as opening a command window, view window, and the debugger. I've also seen this feature used in the code itself, like so:

```
if user=ME
  do SomethingSpecial.prg
endif
```

The problem comes when the developer doesn't provide the 'SomethingSpecial.prg', and thus when you run the system, it crashes.

Wisdom – Wish I'd Know That When...

Standards Are Good - That's Why We Have So Many Of Them

This one is (partly) my fault. If you run into the application in question, I apologize in advance.

I'd been brought in to overhaul and take over the maintenance of a large application being used by multiple departments in a mid-size company. It was one of those applications that had been developed 20 years ago, by a group of programmers who were consistent in their application of practices that they had developed inhouse. The app had then been maintained under the watchful eye of a senior programmer who ensured that those practices were followed.

Then came a change of IT management, and with it, a review of all systems to bring them up to modern standards. One of the decisions was to update the code as modifications were being introduced to follow 'more modern standards'. One of those more modern standards was a thorough combing of the code to make variable naming consistent. These changes included changing variable names to use common abbreviations consistently, adding 'm.' to all variable names, and converting to Hungarian scope and type identifiers. Thus, the panoply of versions for a last name - 'lastname', 'lname', 'namelast', 'last_name', and so on - would all be converted to 'namel'.

However, doing a wholesale review and refactor of the codebase would have been unreasonably time-consuming and fraught with peril, so they opted to make changes gradually. All new and modified code would heretofore follow the new standards, but existing code wouldn't be changed.

Thus, a year later, the codebase was rife with code like this:

```
replace tier with action+rank, ;
  startdt with startdt, ;
  enddt with enddt, ;
  reason with reason, ;
  effectdt with effectdt, ;
  code with m.lccode, ;
  status with m.lcstatus
```

You see which part of this one statement had been modified recently, don't you?

Another year into this project, and management decided that they didn't want to put anymore money into the app, and chose to instead focus their efforts into a all-hands-on-deck switch to a .NET application that should last through 2020. Last I heard (3 years later), they're still working on getting the first build to beta testing. I'm not one to throw stones; this article is nearly as late.

Lesson: A trite amorphism about not changing horses in mid-stream is tempting, but the bigger message is when asked to 'gradually' move code from one set of conventions to another, to more clearly define the benefits. While the resulting code was starting to 'look better', the juxtaposition of old and new was actually more jarring than had it all been left alone.

You Can't Know It All

I remember our friend Drew Speedie standing up in front of a user group, taking questions at the end of his presentation. Someone asked him about how to handle a problem with a report he was struggling with. Drew said, "I have no idea. I haven't written a report in five years. We've got a couple people on our team that specialize in that."

As I was writing the first draft of this article, I ran into a system where the developer had made great use of combo boxes. Needing to modify the contents of one of them, I opened one of the code windows for the form, and did a quick search for the name of the control, figuring I'd find the reference that populated it.

Oddly, no luck.

Hmmmm, maybe this form uses a parent class to populate the control. A little bit more searching. Still no luck. Ah, I then realized that the combo population must be abstracted through the use of a generic function call that passes the name of the control, assembled in pieces, somewhere in the base classes. So I searched for substrings of the name.

No joy.

This was getting a little out of hand. I should be busy getting my work done, not searching for something that I should have found in 15 seconds. And I'm running out of answers.

Until I accidentally clicked on the Properties window when the combo in question was highlighted, and saw all sorts of strings in bold in the Data tab. Oh, look, they're using Type = 2-Alias. Since I've been exclusively populating combos using Type = 5-Array for twenty years, I had totally forgotten that other people populated combos differently.

Lesson: It's OK, bunkie, you can't know it all. You can't even know most of it.

You've Never Seen It All

I was asked to extend a VFP application that had been in use for nearly ten years. It'd started out life as a standard out-of-the-box system based on a popular VFP framework. At one point, the development team morphed it into a custom n-tier architecture, and then a few years later, gave up on that attempt, going back to straight table access, but with business objects strewn here and there. If you're picturing Jeff Goldblum crawling out of the pod, half-man, half-fly, in the remake of The Fly, you'd be spot on. Except no hot female love interest.

Anyway, the application worked really well (despite itself.) It was simply difficult to follow the code, what with the multitude of class libraries, which may or may not be in play in any particular module of the app. It was pretty common to run into code segments that looks like this:

```
this.parent.mst_dataobj.l_cursor_name=lcCursorSubset
this.parent.mst_dataobj.l_databasename =
lcDataPath+'division\client'
this.parent.mst_dataobj.l_tablename='mastercheck'
this.parent.mst_dataobj.l_filter=lcFilterString
this.parent.mst_Dataobj.l_check_memo=lcCommentRaw
this.parent.mst_dataobj.l_openchecks
```

Read through that code carefully. You'll see that there's an obvious error in the last line - the l_openchecks property isn't being assigned a value. You may be wondering how that passed the compiler without throwing an error. I assure you, it didn't. How? One of those

```
on error *
```

statements we've all seen in an app that prevents error messages from bothering users? Nope. Let's explore further.

In another module, I came across a code segment that looked like this:

```
this.ohandle.mst_dataobj.sub_dataobj.opentables()
if
this.ohandle.mst_dataobj.sub_dataobj.tablesAreInitial
ized
  lcFactory = this.parent.caller
```

Clearly, the last line could use some improvement, as the lcFactory variable is being assigned an object (the caller of the parent) but it's named as a character variable. Again, this is not a mistake. Rather, it's another example of the same situation.

The lead developer who had taken over the dismantling of the n-tier architecture discovered that method calls do not need to have parens. Thus, these two statements worked identically:

```
this.opentables()
this.opentables
```

And he, in a perverse fit of ego, decided that he was no longer going to use parens, regardless of how difficult they made the code to maintain, simply because ***he didn't have to***.

Thus, the

```
this. parent.mst_dataobj.l_openchecks
```

statement was actually a call to the l_openchecks() method, and the

```
lcFactory = this.parent.caller
```

was actually assigning the return value of the caller() method to the lcFactory variable. Obnoxious, to be sure, and made followup work on the app just that much more difficult, particularly when some methods in a series had parents and others didn't.

Lesson: Just when you think you've seen it all, you still haven't.

Customers Say The Darndest Things

These don't really have 'lessons' per se as much as provide solace to you when you want to bang your head against a wall.

There Are So Many!!!

I was working through the conversion of a series of screens that made copious use of the following code structure in the when() method to restrict access to fields on the screen:

```
if <condition>
  return .f.
endif
```

The problem being, of course, is that there's no visual clue that the field isn't editable.

Being of the 'visual clue' school myself, I found this difficult to deal with, as it was never evident when a field could be edited or not; the user was simply prevented from tabbing or clicking into it, without explanation. The reasoning behind the technique was "This was easiest to do, and besides, our users know how to use screens."

Fortunately, during the conversion, a number of new employees were being hired, and they agreed that making the forms more visually educational would be a good idea for the new version.

In doing so, I naturally took advantage of the methods available in VFP 9 in addition to when() and valid(), such as gotfocus(), lostfocus(), setfocus(), keypress(), interactivechange() and programmaticchange(). The first time they saw one of the new forms and how the new methods were used, their first comment was "Why do there have to be so many methods? Can't you just use when() and valid() for everything? We don't have time to learn all of those."

Something Old, Something New, Something Browsey

Most of us remember the pain and agony spent in coordinating BROWSES with the 2.x READ architecture. (Those of you who don't, well, you suffered just like the rest of us, you're just lucky to have a failing memory.)

I had spent a not-inconsiderable amount of time building a custom framework to replace the multitude of browses used in a system being converted from 2.5 to 9, replicating grid layouts, window positions, record placement, incremental search, and on-the-fly data entry.

Behind the scenes, I used all of the usual tricks, including co-opting their data dictionary, overlaying parameters to data-drive virtually all of the custom behavior, and even building a tool that allowed them to customize the column properties on the fly.

This all relied on a grid in a form, of course, and the resultant issues with returning more than one value from the selected record, of course. Different than simply SCATTERing the memvars in 2.x. But we made it work and there were benefits, such as multiple browses open and participating in the event loop, adding them to the Window menu, and so on.

Until the prime developer saw the code underneath, and went a little nuts at the abstraction used to data drive the entire process.

“That seems like a lot of work. Why can't you just keep the old browses?"

Along the same lines, the same folks resisted when seeing how forms were created. They kinda liked not having to compile SPRs, having never bought into the bastardization that is “customizing the SPR file” like some, but at the same time, I heard more than once,

"We like the control we have by using @SAY/GET. Not knowing what the final code is going to look like, that's just uncomfortable."

And even FURTHER along those same lines, finding that references to controls on the form involved structures like

```
thisform.txtFirstName.value
```

and, worse,

```
thisform.pgfMaster.pgMotors.txtID.value
```

well, you'd think their head was going to explode.

Lesson: They don't always want to invest in learning new things.

Do What I Meant, Not What I Said

Every user learns to work around the system they use, whether it be standing in line at their local coffee shop or printing multiple reports at the same time, instead of having to wait for one to enter the queue before closing the form and going onto the next.

As a result, when they come across inconsistent behavior, they learn to live with it, they even learn to expect the behavior.

I made the mistake of 'fixing' some of those errant behaviors in a system I was converting. For example, the user would be on Al's record, navigate down the table a bit to Gerry's, hit the browse button, view some data, and after closing the browse, the form would be back on Al's record, not Gerry's, as might be expected. In most cases, I didn't even realize that I was fixing problems like that; I was simply assuming that the behavior I would expect they wanted was what was currently happening.

And then I delivered the first build to the customer, for their users to start working with.

Naturally I was expecting a suite of bug reports, but to see dozens of complaints that the system wasn't working **incorrectly** like the previous version did, that was kind of a shocker.

Do What I Meant, Not What I Said, Redux

So the user said "Make it work just like the old version" except that they want changes here, and here, and, yes, over here, but other than that, it has to work just like the old one – except that the tricks to implement the older functionality ae no longer needed, or in some cases, available exactly the same way, since that system was 20 years old. Witness the following conversation, repeated several times over a six month span of a conversation project:

"We must be able to use JKEY to provide incremental searching in our lists."

"Well, we can't JKEY, because it doesn't work with VFP. But that's OK, because we can build the same mechanism with native VFP code."

"We don't want to write extra code, we just want to use JKEY. We've been using it, it works, and that's what we know."

"But...."

Lesson: Sometimes they REALLY don't want to invest in learning new things.

You Show Me Yours

Converting a medium size 2.5 system (80 screens or so), I build a set of form classes to handle the variety of forms they had in the old system. Some of their forms were extremely complex, with literally hundreds of controls and complex logic that dictated the interaction between them. Additionally, there was code that handled add/edit/delete rights according to both who was working on the form. Certain users had full access to the screen while others only had access to edit certain fields, and still others only had read-only privileges.

Furthermore, privileges depended on the type of record being displayed in the system. If a motor had shipped, it couldn't be deleted, and many fields are no longer editable.

As you can imagine, not only were there lots of twisty passages in the WHEN and VALID clauses, all nearly alike, the SHOW GETS for the screen was both enormous and convoluted. As requirements changed, they'd slap a band-aid on the code to get it working immediately, instead of considering a more maintainable approach.

If it sounds like I'm complaining or making fun, I'm not, we've all done that at one time or another. Those marketing folks who want the form NOW can be pretty pushy, right? "Things are the way they are because they got that way."

Naturally, this complexity could (and should) be broken into more manageable pieces in VFP. It would give a chance to finally re architect and fix some of the fragile spaghetti code issues that have grown over time as well as make it more easily maintainable for changes in the future.

I presented the first beta version of the form to them, at which point it was summarily rejected. The explanation?

"We worked hard to get our SHOW GETS working just right. We want to keep using them."

I ended up trashing most of the architecture developed for the form classes, and creating a single ShowGets() method that was called everywhere. As you recall, 2.x SHOW GETS

include explicit references to the screen's controls, like so:

```
SHOW GETS m.Type ENABLED
```

Those variables are also used elsewhere in the logic. While it's possible to bind VFP form controls to memory variables, there are reasons that you may choose not to do so. Personally, I find the practice makes the code be difficult to read and maintain. So, instead, I set up a series of translations at the beginning of all ShowGets() methods, like so:

```
m.Type = thisform.txtType.value
```

which meant that much of the code in the ShowGets() could remain essentially unchanged, a win for everyone.

In order to deal with maintenance issues, I then wrote a huge header for each form's ShowGets() that explained in detail the logic inside and the decisions made throughout the code, as a mini-tutorial for the next time I, or someone else, would have to venture into the code.

Lesson: The customer isn't always right, but they're always the customer.

Lesson Redux: They have a huge investment in current code and will go to almost any length to not throw it away.

Advice – The Start of a Checklist

I've thought about putting together a formal checklist of everything I'd like to look for when reviewing a new system. As the saying goes, "So many pedestrians, so little time", thus, alas, I haven't. But here's a start.

Globals Everywhere!

The developer created a variable in a routine. Later, he discovered that this variable was needed elsewhere. Instead of passing the variable as a parameter or making it a property of an appropriately scoped object, they do what they did back with dBASE III+: declared it global. There, THAT fixed it!

Except that he did it in the middle of a subroutine buried deep in the bowels of the system.

One of the tests I run when first examining a new application is to do a comprehensive search on the word 'public' in the app. If I see entries like this:

```
File name  Class.Method,Line Code
SomePr.prg searchVIN, 307    public name, type
```

(because why would you declare a global variable 300 lines into a supporting routine), I throw up a little in my mouth.

Lesson: Some programmers aren't as disciplined as others.

The Missing Procedure File

You'll always find calls to custom functions when investigating an application that needs to be converted, like so:

```
m.lcNewValue = SomeFunction(m.lcOldValue,
  m.llConversionFlag)
```

I've always used a naming convention to help identify where a function is located, perhaps something like this:

```
l_ - subroutines in standalone PRGs
x_ - standalone functions, application
specific
y_ - functions contain in the application's
procedure file
z_ - functions contained in my generic HW
procedure file
```

Unfortunately, I've never run into an app where the developers have used a naming convention, either through lack of forethought or discipline. So, faced with the desire to find out what SomeFunction does, it's time to undertake a spelunking adventure. Since pretty much every application I've worked on has generated this type of scenario, I've developed a rubric for dealing with it.

First, if the function is being called from a PRG file, I'll open up the Document View window to see if the function is defined somewhere in the PRG. The Document View window has several advantages over Find. First, it'll display only function/procedure definitions, not every call to that function. Second, if the function was defined more than once in the PRG file (yes, it happens), that becomes evident immediately. And third, double-clicking on the name moves you right to the function definition in the PRG, handy if the PRG is hundreds or thousands of lines long.

If that search yields nothing, or if the function isn't being called from a PRG file, the next place to look is the systems procedure file. When examining an application, I always lay out the structure of the application, including the target of the main program's SET PROCEDURE TO statement. So, again, with Document View open, I open MainProc.prg.

Now if that fails, I'll look in the PROGRAMS\ folder for the SomeFunction.prg file. If that produces no joy, finally, it's time to bring out Code References (or GoFish, if you're so inclined.)

Sound reasonable? Well, here's how it played out in one situation.

The application I was working on had tens of thousands of function calls, and there was no rhyme or reason for where the function was located - the framework's procedure file, the application's procedure file, the PRG that it was called from, or in the PROGRAMS\ folder that contained all of the standalone PRGs. It was quite a nuisance to have four places to look every time a function had to be examined.

However, it turned out that some functions couldn't be found in any of those four locations. There were two problems. The first is that some standalone PRGs weren't located in the PROGRAMS\ folder. A few were in the root, others were in the SCREENS\ folder, and still others in the REPORTS\ folder. The reasoning? "This PRG is only called from this one screen, so I put it in the SCREENS folder so I could find it easily.

The second problem was even better. At certain points in the system, a specialized, complicated procedure was being run. It was similar to another process, except that it had different things to do throughout. So they used a second procedure file, like so:

```
lcOldProcFile = set('procedure')
set procedure to NewProc
<lots of code>
set procedure to &lcOldProcFile
```

The best part? The functions in the NewProc file were the same names as in the original procfile. So they could run processes and point to either the original or the substitute proc file, using the same function calls, but different things were being done. Very clever, a procedural version of subclassing. But without documentation, it was devilishly hard to figure out why.

Lessons: 1) do an inventory of all the places PRG files could be. 2) do a search on SET PROC TO in the entire application.

Scaffolding

A year ago I was tasked with converting a small but complex system written in 2.6. It was very sophisticated, and, interestingly, the primary motivation for moving to VFP was not 2.6's EOL but the need for capabilities that VFP had.

While the development team had become extremely adept with 2.6, they had not invested any time in VFP. They needed to come up to speed quickly. We forget that we've had years and years (and decades and decades) of experience. Trying to learn all of the new features of VFP, as well as develop an object-oriented mindset, is daunting.

One thing I've done for all customers new (or even new-ish) to VFP is incorporate a debugging scaffold into the application's class library hierarchy as well as each form and process in the system.

The scaffold incorporates a custom function that outputs strings to the Debug Output window (or, optionally, to a text file, should the Debug Output window not be available, such as during execution of a production EXE.)

The function is passed both a flag and one or more strings. The flag indicates whether the strings are output or not, and the strings are displayed one after another, due to the fortuitous architecture of 'debugout', unlike 'wait window' or 'messagebox', to accept multiple parameters. The following command displays

```
lcStatus is:ACTIVE:
```

when the glShow flag is true.

```
goApp.debugox(glShow, ;
  'frmJoin.tenderApp() lcStatus is:', ;
  lcStatus, ':')
```

You'll notice a couple of things in this command. First, I've identified the name of the form and the method that this command is located. It may be obvious where the command is when it was first added to the code, but a day, week or month later, not nearly as much.

Second, the leading and trailing semi-colons are to distinctively delineate the variable. In cases where the variable is blank, it may not be obvious, particularly after a few dozen scaffold commands have been added and you're trying to quickly scan through the output, looking for a particular piece of information.

There are four general levels to this scaffold.

The first level are statements at the beginning of every method in the framework as well as every custom method that runs code. Here's a simple example:

```
goApp.debugox(glShowMethod, ;
  'Starting frmMain.init()')
```

Depending on the complexity of your system, you could break this into multiple levels.

Running the program and watching what shows up in the Debug Output window can be extremely educational and helpful to the VFP developer coming up to speed. But at the same time, it can produce a lot of output, most of it unnecessary when trying to debug a particular process problem. Hence, the flag that can be set to false and suppress that output.

The second level of scaffolding is done within a single method. The llShowThis flag is set to true

when debugging what's happening in that method,

```
goApp.debugox(llShowThis, ;
  'lcStatus starts out as:', lcStatus, ':')
<some code>
goApp.debugox(llShowThis, ;
  'After 1st query,lcStatus value:', ;
  lcStatus, ':')
<some code>
goApp.debugox(llShowThis, ;
  'After 2nd query,lcStatus value:', ;
  lcStatus, ':')
```

The advantage of this mechanism is that the developer can trace what's going on in a very granular fashion, and then turn the whole output off with one keystroke, and then turn it back on again later if needed.

This second level can be extended to span multiple methods, even multiple forms, to trace the internals of a complex process. In that situation, the flag needs to be defined at an appropriate level in the system, generally in the main program.

Finally, once in a while, you may find that you need to do a dump of a particular area for a one-off problem. I'll earmark these with a leading 'HC' string and then the name of the object and method to indicate that that particular output is hard-coded, and where it's coming from.

```
goApp.debugox('HC frmCustAnalysis.tally()',;
  'Analyzing Account Number:', lcNoAccount,;
  ' UserID', lcIDuser, ':')
```

Lesson: Provide a debugging scaffold to help the development team get used to what happens when, and the interactions of various pieces of the system.

Peopleware

Finally, a reminder that by far, the majority of problems we face aren't technical issues, they're people issues. Our customers have their own needs and requirements, and they don't always match with ours. Here are a couple of ideas to help align theirs and ours when they get out of sync.

Rules Are Meant To Be Broken

I've only done a couple of completely new systems in the last five years, the rest have been extensions, upgrades or conversions. A fairly common scenario is where the customer, comfortable with the way they've done things, resists development processes that you use in your projects, such as formal issue tracking, version control, or parallel testing.

There seems to be no way to uniformly categorize the miscreants; I've had IT departments go along with new procedures because they see the value; they just hadn't gotten around to implementing themselves. At the same time, I've had other development staffs resist mightily, which amused me, because they were the first ones to complain when their users didn't follow the rules.

Bug tracking is the most critical, it seems to me, because it directly affects the quality of the end product. Losing track of open issues and not having a record of fixes is a sure way to keep lots of bugs lurking in the product.

I've had any number of customers try to avoid the formal bug tracking software, instead just calling or emailing with a bug request. I feel the optimal solution to a situation like this is one inspired by my kids, where behavior modification is best accomplished by implementing logical consequences (If you don't put your bike away, I'll put it away for you, and I'll lock it up, and you won't have it available when you want to use it next.)

So take the example of when a user won't enter a bug report. Their rationale is that they find it faster to email or call. However, the reason for the bug tracking system is to provide a single centralized location where issues can be completely entered, tracked, prioritized, and closed. Furthermore, it can be researched if that or a similar issue comes up again. The second reason is that when the system isn't used, pieces of data are forgotten to be noted (such as priority), so the form acts as a checklist of info to be collected.

So in this example, where they won't enter a bug report, I'll make sure that the logical consequences of said action (or inaction) take effect. When they ask about the status of an issue, I'll look it up in the system. When it's not there, it'll eventually come out that they didn't enter it into the system, but emailed it to me, or called me. Then I'll explain that "well, it must be around here somewhere, I'll have to look for it...." After not being able to find it a few times, they'll generally get the idea.

Lesson: It's not the process, stupid. Oh, wait, yes, it is.

See What You Made Me Do?

I've told the story many a time about the best bug report I've ever seen. It went like this:

"Tried to add record without pressing Add. Result: Record not added." Ah, yes, that was a surprise, wasn't it?

Fortunately, most of us learned long ago that submitting a bug report that goes like this:

```
Title: Cart ID problem
Steps to Reproduce:
1. Edit an Inmate
2. Add a new cart
3. Save
4. See what happens?
```

is a bad idea. The reason is that what happens for me may not be what happens for you. When I do it, I may see the Cart ID stay static, or it may disappear, or it may change to the wrong value. (Even more fundamentally, this assumes that I know what's supposed to happen to the cart ID when a new cart is added in the first place. I've had more than one customer tell me, "You don't have to understand what the system does, just make the code work.")

When you do it, something else may happen. You can't see what's happening on my screen, and all sorts of circumstances may prevent my system working the same as yours.

Thus, this is much preferred:

```
Steps to Reproduce:
1. Navigate to Inmate #200
2. Click Edit button
3. Click the [+] button next to the cart
listbox to add a new cart
4. The defaults of a blank description, qty 1,
and price 0.00 display.
5. Select 'deck of cards'
6. The price changes to 0.79.
7. Click Save button

What happened:
8. The cart listbox display the new row (deck
of cards) but the Cart ID shows '00000'.

Expected:
The Cart ID should be the next ID from
the SYSTEMKEY table.
```

So far, so good. What happens when a customer refuses to follow the instructions, and insists on delivering incomplete 'Steps to Reproduce', 'What Happened' and 'Expected' information? It's tempting to get emotional or belligerent (who, me?) I've had more than one customer insist, "It takes too long to enter all that information. Can't you just figure it out?"

Instead, the very best solution I've ever heard is Tamar Granor's even-keeled reply, "I'm going to need more information before I can help you.", explain what the missing data is, and leave it at that. Regardless of how many times it takes, keep nudging the customer to do the right thing. Eventually, they'll learn that it'll take less time if they do it right the first time.

Or you'll get fired, which will likely be a blessing in disguise. Do you really want to work with a customer who doesn't want to cooperate?

Although once in a while, you may end up working with someone even more stubborn. On one project, my contact never did accept that fact that writing out the information completely the first time would ultimately be more efficient. They kept thinking that the total time they spent would be less if they wrote all bug reports incompletely, gambling that I'd be able to figure out some without coming back to them. The time they saved would be greater than the extra time they spent when I kept coming back with requests for more information. Of course, the two year project took almost four to complete, but they were used to that.

Lesson: You don't have to be married to someone in order to not be able to read their mind.

Author Profile

Whil Hentzen is an independent software developer based in Milwaukee, Wisconsin (as opposed to Milwaukee, Minnesota, as many people think.) His writing has killed many trees over the years, but none since 2007. He has realized he really sort of misses it. You can reach him at whil@whilhentzen.com