# So You've Inherited An Application – Now What? 51 Tips to Help You Survive

*Session Number 21*

*Whil Hentzen*
*Hentzenwerke*
*980 East Circle Drive*
*Milwaukee WI 53217 USA*
*Voice: 414.332.9876*
*Email: whil@hentzenwerke.com*

*There are thousands, no, tens of thousands, no, hundreds of thousands of Fox applications out there. Unfortunately, there aren't hundreds of thousands of Fox developers any longer, and more*

*likely, the original developer is one of those long gone. The application is now - potentially - yours. But this situation is often fraught with peril.*

*Along with those hundreds of thousands of applications come hundreds of thousands of different approaches to building them. As a result, there isn't a single recipe for approaching an inherited app. The best we can do is offer some ideas from which you can choose from those that make sense for your situation.*

*In this session, we'll first take a look at what to do when you're presented with "We have an application we'd like you to look at. Oh, and by the way, the original developer isn't here any longer." While there are technical challenges a-plenty with vintage applications, the bigger struggle will be the customer interface - determining what expectations should be, and then setting those expectations for your primary contact.*

*Once you've decided to take on the job, it's time to look at the technical side of things. We'll make some suggestions on how to quickly and efficiently grok a system that has landed on your shoulders, again using examples from real life situations whose names have been changed to protect the innocent.*

## 1. Before you take the job on

### *Inherit*

It is a dark and stormy night. The phone's bell shatters the peace and quiet. You jerk your head up from the keyboard, the indentations of the keys on your forehead not registering, your hand knocking over the can of Mountain Dew next to your mouse. "We've got this application that we'd like you to take a look at."

At this point, the field is wide open. You don't yet know how they got your name, the expensive Yellow Pages ad that you kept meaning to cancel, a Web search that landed on your home page, or maybe a referral through a friend of a friend of a friend. No matter, now, they have you on the line, and they want something. It could be anything.

"Our system works great; we just have a couple of modifications we need you to make."

"Can you tell us how much it will cost to add some new reports to our system?"

"Our old developer isn't available anymore and we have some changes we need made."

"We need to upgrade our system into Windows/SQL/The Internet."

"Our business has changed and we need to look at redoing our mish-mash of systems."

You talk to them for a few minutes, or more accurately, they talk to you for a few minutes, you interjecting the occasional "Uh-huh" or (hopefully) insightful question, and although you haven't made a commitment or signed an agreement, emotionally, this system is already becoming yours.

### *Diversity*

One size does not fit all. In the case of custom software maintenance, one size hardly fits anyone. Although there are a number of generally accepted practices in the Visual FoxPro community, there are many, many more folks who have either actively, passively or ignorantly eschewed the community and common practices, preferring to invent their own. And even with developers who stay the course, adhering to generally accepted principles whenever possible, there are, due to VFP's flexibility, a dozen ways to accomplish most any task. As a result, the number of approaches toward building an application in VFP are as varied as the number of developers who build those applications.

As a result, a road map showing you a single set of steps to follow isn't possible; what's relevant in one situation doesn't have any applicability to another. However, there are some similarities regardless of the specifics of the application, particularly in how you approach the customer. So the purpose of this article is to provide you with both a general philosophy of how to approach the inheritance of an application as well as a toolbox of mechanisms and devices from which you can pick and choose those tools that are appropriate for your situation.

## *One little change*

To them, the request isn't much. They've got this report that works just fine, you see. It's just a matter of copying that report, changing the order of items and subtotaling differently. So it should be a simple matter of making a copy of the report, changing the sort order of the cursor that feeds the report, and then rolling up your sleeves and doing a bit of work on the new subtotals. A couple of hours, maybe a half day, right? That's what another programmer told your contact it should take, although conveniently they're too busy to do the work themselves.

Sounds good. You haven't seen the code yet, but you'll take it on, because their description sounds like they know what they';re talking about, and thus, how hard could it be?

Ah yes, those five famous words.

## *How hard could it be?*

When you get inside the original report, well, that took a bit of work, as the menu option for the report doesn't just call an FRX. Instead, it references a memory variable which is set in a subroutine called from another program that doesn't have anything to do with this section of the application. So that's an hour of your life you'll never get back.

Anyway, back to the original report – it's constructed on the fly from a combination of tables and memory variables – so changing the order isn't a simple matter of changing the current index or creating a new index. You'll further see that the index used to sort the records in the controlling table is fragile. It wasn't constructed properly in the first place, and thus it's a fluke of their existing data that the report sorts correctly now.

"Oh, did we tell you that this report occasionally doesn't sort right? We've never been able to figure it out, but it's not that big a deal because it doesn't happen that often. I'm sure you can fix the sorting problem when you're doing the new report."

So you've now spent your estimated half day just researching how the current report works.

Another few hours and you've got the new report built with the new sort order and the new subtotals, and you've (cautiously) fixed the old report as well.

And you find out upon delivery of both reports that they wanted the new one called from a form in the system, not a menu option like you thought you heard them say. This was just one simple report, so it wasn't important to get any of this in writing, right? "Can't you just add a button to the form to call the report?"

Sure can! You open the form, plunk a button in the corner and add the call to the report. Only to have the report fail with a cascade of very unexpected errors referencing memory variables that have nothing to do with this report.

It turns out that the grouping expressions that the report uses reference a set of global variables that are changed when any form is opened in the system. This form stomps on several of those globals, which ruins your beautiful report. Those globals (globals? Who still uses globals?) were set up appropriately and behaved well when the report was called through the menu. It's going to take some re-engineering to make the report work from a form.

First, you're going to have to trace them down and determine under what conditions they're being changed, so you can either use them correctly, or replace them with properly scoped variables.

You're now well into Day Two of this "couple of hours" report. The $150 or $200 they thought they were going to spend on the report has ballooned to more than $750 and now they're rather unhappy. The system worked fine until you got involved, and the old developers never seemed to take this much time to do something (selective memory isn't restricted only to a teenager who brought the car back late) and just how did this simple change become so complex?

Clearly, you're in over your head. That's French for "you don't know what you're doing, do you?"

Multiply this scenario throughout an entire application, and suddenly that $5,000 they thought they were going to spend for a simple set of modifications is now $30,000 - and climbing.

## *Mission: Objectivity*

The person who contacted you most likely doesn't know what's inside the application, nor do they understand the process of software development. As a result, they have no ability to appreciate the potential difficulties involved in making even (what appear to him to be) the simplest of changes.

Thus, the goal for this article is to provide an objective source – a third party expert, so to speak – to help you to educate your customer. You don't want to go to him and simply announce, "Yup, it's a piece of crap and we're going to have to rebuild it from scratch" even if that's what you're thinking. You need to educate him about the process of software development – and that's much more than simply writing code and building EXEs – and then provide documentation about how their application varies from the norm. The size of the difference will help explain why their change requests are potentially going to be more work than would seem on the surface.

# 2. Sizing up the customer

## *Before the kiss*

After the initial conversation, you're going to want to size up the customer. This isn't a technical issue, it's a business issue. Do you want to work with them? As there are many types of developers and many types of applications, there are many types of customers. And of those many types, there are a few that you are better off steering clear of. The cheapskates, and the cheats are two that come to mind immediately. Depending on your personality, there may be others as well. For instance, some developers like a close working relationship with their contact, while others simply can't work with a contact who attempts to micro-manage every aspect of the project.

A long time ago, when 'corporate mottos' and 'mission statements' were all the rage, I developed one for my company that simply stated, "We don't work with jerks." (The actual wording is somewhat stronger than that, but this is a family newspaper, right?) Every time I've attempted to ignore that rule, it's come back to bite me.

Trust your gut - do you want to work with these folks?

## *Quarter-inch holes*

I did a lot of work with business plans in the olden days, and one of the first sets of questions that put a business plan together involved determining what your product/service was. The line of questions went something like this: "What do you sell?", "What is it that your customers are buying?", "Who else could they buy it from?" and, eventually, "Why are they buying it from you?" (The classic example, of course, was posing the question to a manufacturer of drill bits – customers didn't buy 'quarter-inch drill bits', they bought 'the ability to make quarter-inch holes.')

Similarly, the customer might ask you to do one thing, but actually have something else in mind. Why the difference?

There are several reasons.

The first is that they just can't explain what they're really looking for. Unlike a term paper in Literature class, software developers can't get by with being vague. They have to know exactly, specifically what they're going to do. End-users, whose worlds are more often filled with multiple shades of grey, don't often appreciate this requirement. Thus they may not be able to do any better than "We need better statistics for our inventory turns." After you then turn this bland requirement into a series of reports with drill-down capability, they answer with "No, no,no, we need graphics! We need bar charts for month to month results! Lots of visuals! Our president likes pictures!!!!"

Second, they may have assumed that since they know more about their business than you do, they also know more about ***your*** business than you do. This results in their telling you specifically what they want changed, in other words, they've already translated their business need into a technical requirement.

Another is that they have a master plan in mind, but they're only going to tell you a little bit at a time. One reason for this approach is that they want to evaluate you on a small part of the project before going with the whole thing. They don't want to tip their hand about the entire scope quite yet. Thus, the piece that you're looking at might not be representative of the entire gig. Another reason is that they don't think you can handle it – that their business is so complex that they're assume that you can only absorb a bit at a time. (This could very well be true, but it behooves them to at least give you a big picture view first, so that you can develop some perspective of where this little piece fits in.)

Fourth is that they might not know. Working on a maintenance project is a study in discovery. You're discovering what the application does (we'll get to the 'Why is there no documentation?' issue later) and they're discovering what you can do. Frequently, particularly if you've kept up with the latest advances while they've stayed mired in an older version, your capabilities will appear wondrous and magical to them. As a result, you'll be able to offer suggestions and perform tasks that they only dreamed about, or may not have even know enough to dream about. "What if we provide the user with the ability to edit an email and then have the system send the email to the customer when the order ships?" "You can DO that?"

Finally, the technical issues may only be part of the bigger picture, the bigger picture involving business issues that your contact doesn't want to talk about for fear of exposing a lack of knowledge or another embarrassing situation. For example, the application may provide a critical function of a department, and the seemingly innocuous changes you're being asked to provide are actually imperative to the group.

As you sound out the customer, being able to distinguish the difference is critical. You have to be comfortable that you understand what they're really looking for. They're going to judge you based on what they have in mind, not what they told you.

## *Purpose*

In general, there are four categories of work that a customer is going to call about. First is to simply fix existing bugs. These could be problems that they've lived with for a while and just can't stand any longer, or that have recently cropped up.

If the customer hasn't  had another programmer change the application, then either the data or the environment is at fault, although it might take a bit of detective work to find that out. Famous is the story of the application that 'suddenly stopped working'; when asked, the customer insisted that they didn't touch a thing. Upon visiting the site, the developer 'happens' to notice that the operating system of the machine in question had been upgraded from Windows 3.1 to Windows 95. But other than that, they hadn't laid a finger on it.

Just as common is a request to change existing or add new functionality. New reports, changes in the business logic behind existing forms, modifications to data to support a new line of business, it's all fair game.

A third category is to move the application to a new paradigm. This may be upgrading from DOS to Windows (don't laugh – it happens a lot!), or to a new backend (such as from DBFs to a SQL database engine), or to the Web. In my experience, customers wanting the first and third conversions are pretty clear about what they want – their users are asking for a new interface or access to the system from the Web. As far as a change to the database engine; sometimes there's a sound technical rationale for it, but just as often, the impetus comes from illogical assumptions – such as FUD spread from an opposing developer, "You know that DBFs aren't any good for real applications – you need to go to SQL!"

Finally, some customers realize that their collection of hand-crafted applications developed piecemeal and cobbled together need to be thrown out and replaced. However, this particular situation needs to be handled with care. Particularity with VFP, as soon as you say 'Rewrite', someone is bound to ask whether or not VFP should be the target tool. As discussed later, let the customer bring up the 'R' word (rewrite) first.

## *Remorse*

Working on an existing application has more potential for disappointing the customer, since there are twice as many unknowns as with a new application. Not only do you have to learn what the application is supposed to do (in order to satisfy the customer's needs and wants) but then you have to figure out how the current application is doing it. And as mentioned before, it's a rare customer who fully appreciates the enormity of the task.

Thus, it's up to you to find out what will make them happy. This isn't just a technical question, in nearly all cases, you'll be able to make the application do what they're looking for. The question is whether you can do it in the time and budget that they've got allocated for the project. Since the work needed to do inside an existing app is even more ethereal than building a new app, it's highly unlikely that they've got an accurate number in mind. Thus, they'll either have to be willing open their checkbook again (and again, probably), or be willing to accept what you can do for the amount that they've got in hand.

## *Leafe*

I first heard this quote from Ed Leafe: "Better to be sad that you lost the job than to be sad that you got it." The flip side of "Remorse", this means that it's better to 'fess up at the beginning what you think the bad news, in terms of time or budget, or both, is going to be. If you go in with a low ball estimate, or if in the back of your mind, you're thinking that you're going to have to do more work than they're expecting, or if you think you're going to be able to talk them into a major upgrade or even a rewrite once you've got your hands in deep, you're bound to disappoint someone.

Accepting a fixed price job for a vaguely specified set of requirements is a guarantee for unhappiness either on your side or theirs.

## *Surrealism*

How many times have you wanted to ask a customer, "And on what planet is that thinking considered rational?"

Once in a while you'll run into a customer who is very clearly trying to skate something by you. I ran into a 'professional' (the profession is left to your imagination but I'm sure you won't need to imagine very hard) who pitched a "actual cost with a not to exceed price" proposal. In other words, he wanted to realize the benefit for efficient development, but didn't want to be exposed to any of the risk due to inexact or moving specifications. You need to make sure to watch out for these folks – trying to get something for nothing – and they're most definitely out there.

Another ruse I chuckle at is the 'develop this app for me for free and then I'll help you sell it to everyone else and you can make lots of money on those sales' brainstorm. Wish I had a nickel every time someone has 'thought up' that one.

But here I'm talking about folks who are so focused elsewhere that they don't realize how unrealistic their expectations are. It's your job to realize this, and then to attempt to bring them back to earth.

I once was talking to a potential customer about an add-on to his system that would perform inventory control and management. We got around to the part of the conversation where I asked what kind of budget he had in mind, and the response was dead silence. Fair enough, some people don't like to tip their hand first when talking about money. So I follow up with my standard line, "It's important to know what kind of ballpark we're talking about, you know, some people think they can get a complete custom app written for $500." He pauses for a second, and answers, "Oh, I wasn't wanting to spend anywhere near THAT much."

You see, he had in his mind he could call up and order a custom inventory application that would be assembled to his requirements for a couple hundred bucks. After all, DAC-Easy only cost $49 and it was a complete accounting application. He figured that three or four times that amount for a custom job would be plenty fair, right?

When I explained what 'custom software' really meant, he quickly decided that he needed to look elsewhere.

You need to outline the process of developing requirements, specifications, how the development process works, what will be involved in testing, and how to deploy. A five minute conversation about these steps will help open many people's eyes.

## *Difficulties*

Having a well-written platform of potential difficulties you can run into – before the fact – can be very helpful when communicating to a potential customer why you can't quote a fixed price for work on a system that you've never seen before. Here is a list of difficulties in working with unknown code that you can use as a starting point:

1. The code may not be documented inside. The following very simple function call

```
= GetMessage(m.stat, m.line, m.time)
```

has all sorts of questions:

- what does this function actually do?

- what values are in the parameters?

- what is the return value?

2. The code may not be written consistently.

For example, the above function actually writes data. It was initially a data retrieval function, and in other parts of the application, that's what it does. But then things grew, and it eventually morphed into a data reading and writing function. In this location, this function only writes data. But you'd never know from the name.

3. The code may not use industry standard conventions,

One simple example revolves around naming conventions. While there is some disagreement in the community, Hungarian notation is commonly used. It greatly helps readability, particularly in maintenance cases. In the GetMessage function, what types of values are being passed to the function? Logicals? Character? Numeric? No way of telling. To be sure, m.stat could contain a character string, but you don't really know. (In the application, it is actually an object. You can imagine the fun that this involved.) If the parm was named 'oStat', the name would be a clue that it probably should be an object.

I've been working with an app strewn with code like this:

```
if result.value = <something>
```

Coming into this code, you have no way to telling if result.value is a table and a field, a cursor and a field, or if it's an object and a property. Given that there are some two hundred tables in the application as well as a philosophy of letting the application clean up cursors upon close of forms, it's anybody's guess. If the code had been written

```
if oResult.value = <something>
```

it would have been instantly obvious.

4. The code may work only through coincidence, not because it was engineered that way. Many (most) developers work through their code until it appears to work, and then move on. Some will actually test their code. Very few will actually try to prove that their code works by using an independently developed test suite that handles boundary conditions, loads, and so on.

5. The code may have hidden business logic as well as hidden programming tricks.

The personal computer and the desktop application development tools that run on them lend themselves to being self-taught, unlike mainframe programmers who were taught discipline as a prerequisite to proper development techniques (such as deploying live without testing, or working on live code, both of which happen frequently in the desktop world). There is a body of generally accepted practices that many individual developers aren't aware of or don't' avail themselves of.

6. There may not be a reliable set of test data that can be used to ensure that the application performs the way it's supposed to.

## *No assumptions*

Just because the customer tells you something doesn't mean it's necessarily true. Not that they're lying to you - they just may not know. I once had a customer who asked for some new reports to their system. I asked if it was FoxPro for DOS or FoxPro for Windows. They said, "We're not sure." I'll wait until you've picked yourself up from the floor.... OK, now I'll explain. They were running a FoxPro for DOS application, inside the interaction version of FoxPro for Windows. Their previous developer told them they were running Fox for DOS, but had installed Fox/Windows without telling them, and they knew they were running Windows 3.1 for other applications, thus their confusion.

The application you're inheriting may be a DOS app ported into Windows, and then into VFP. Many is the app that is clearly running in VFP 7, say, but when you open the code, it's all FoxPro for Windows 2.x SCX code transported into VFP.

Or it could be a nicely written VFP application - just missing some of the source code.

Or a mish-mash. I worked on one Web-enabled application that used a combination of Web Connection, FoxWeb, and FoxISAPI - some of it working, some of it commented out - all in one system. They started with one framework, and when they ran into a problem, they stopped development and incorporated a new tool. Needless to say, it was a nightmare to dig through and figure what was going on in there.

You will need to open up the app and look inside yourself.

## *One assumption*

The one assumption you CAN make is that there will be no current documentation. There may be some doc, but it's not going to be thorough, and it's not going to be current. Useful systems are dynamic. They get changed, and even developers with the best of intentions never get around to updating the doc after making changes. But you knew that already.

## *Clueless*

You may run into a potential customer who sounds like they know what they're talking about. I recently had a long discussion with someone who was convinced they needed to convert their FoxPro application to MySQL. They didn't know what that meant, but it took 20 minutes to discover that what they really needed was a Web interface for the app, and that the back end really didn't matter. They had talked to a number of (purportedly knowledgeable) Web developers who all were adamant that the database had to be MySQL if the application was going to be on the Web, and the potential customer became fixated on that particular item.

## *!Slam*

No matter how bad the application is, you can't criticize it. You don't know who wrote the original application - it may well be the brainchild of the person you're talking to, or the slightly-addled but much-loved nephew of the president.

Even if you find out who the original developer was and how highly reviled they were, you still need to bite your tongue - someone at the company paid for the application, and you could be needing their approval for your work.

These rules still hold even if they disparage it themselves, you can't just assume you can jump on board. It's kind of like the fellow who makes insults his own wife - that doesn't give you license to do so as well.

# 3. Sizing up the system

## *Education*

Maintenance projects are rarely easy, 'just one small change' projects. So the goal is to educate the customer as to what it really involves. You need to set and/or agree on expectations. But it's difficult when there are so many unknowns.

You can never know for sure, but you can stack the deck in favor of success, where success is defined as 'both come out happy' with the results. You want to present what a system designed for easy maintenance looks like and in which ways their system varies from that description.

Remember that the customer fundamentally doesn't care about your problems and excuses and troubles in working with their application. They care about getting it done, and they don't understand why adding one little change is huge work for you, both because the change inside isn't small, nor do you know the ramifications of your change without solid testing.

You'll have to do some heroics, but you'll also have to educate them about practical realities. There are lots of unknowns and you simply don't know what you don't know yet.

## *Analogy*

The construction of a building analogy works well with customers. Draw the picture of being asked to do renovation on a hundred year old building that'd been updated a number of times, but nothing is labeled. No floor plans, no wiring diagrams, no nothing. You simply do not know how long it will take to do any type of major construction. What is in that wall? What is in that pipe in the wall? Water? Sewage? Noxious gas? Wires? Nothing? You probably don't want to rip it out without knowing, and since it's not labeled...

Same goes for software development.

## *May you?*

The first thing you want to do is make sure you have permission to work with the code. Can you even LOOK at it? Consider getting the permission in writing.

## *Can you?*

Once you have permission, the next step is to figure out if you can work with the code. First, do you have the source? All of the source? And nothing but the source? If they've got a project and a working EXE, you can do a quick check by trying to build an identical EXE to the one they're currently using.

Since the EXE might not be the only piece needed to deploy the app, check the project for excluded files, making sure you can find them as well. Then, find out how the app has been deployed, and see if there are other files that are part of the installation.

Finally, address that little issue of.... data. Where is the data? Ideally, the app can be configured to point to a relocatable data set, and has a set of clean test data that you can use, but that's not very likely. At any rate, you'll need to find the data set, and see how clean and useful it is.

## *MAIN2.PRG*

Just because the project builds an EXE identical to the one that's in production use doesn't mean you're home free. Just as rare as that set of clean and validated test data I mentioned earlier is a source code tree that doesn't contain any duplicate, unneeded or outdated code. Once you find the top level program or form for the project, look around for other files with similar names.

If the main program is named 'main.prg', look for 'main2.prg', 'mainold.prg', and 'mainnew.prg'. I once worked on a system that had five 'main.prg' files scattered about in various subdirectories of the project in addition to the version in the root of the source code directory. And as Murphy would dictate, and the one that started up the system was not the version in the root, but a copy in the 'files' subdirectory. Worse, to take a look at it would give you no clue that it was the main program, as the first 70 or so lines had all been commented out by a previous developer who was clearly grasping at straws, trying to get a program with that new-fangled "READ EVENTS" to work correctly.

And even worse than that, it took a day to figure out how to get the application to run, as the source code was given to me without the explanation that a special configuration program and path command had to be run first, so that the correct 'main.prg' program would be executed. You can probably imagine that this wasn't the only workaround I ran into on this project.

## *Ancestor knowledge*

While FoxPro's legendary backwards compatibility makes life easy for developers, knowing that code you wrote years ago will still work in the most recent version of Fox, it can prove to be a double-edged sword when you are faced with maintaining such a system.

Do you have the legacy knowledge to handle old-style code, constructs like "@SAY", "KEYBOARD", ON KEY LABEL, RDLEVEL and PUSH MENU? You're more than likely to see code structures that you've not seen in a decade once you start looking at someone else's app. I know I pull out my old 2.6 books as well as the Hacker's Guide to VFP on a regular basis, and still scratch my head more often than I'd like to admit.

## *Ancestor developers*

There are 3 kinds of developers who wrote the system (and, no, they're not 'The Good, The Bad and the Ugly, although they well could be). They can make your life easier, or a lot harder, depending on which kind you've got and whether you can work with them.

The first kind are Gone. The flash-in-the-pan independent developer who has since moved on to bigger and better things (such as a challenging customer service career in the quick food business), or the employee who is no longer around; for all intents and purposes, they might as well have been hit by the proverbial beer truck. (That's how all developers meet their maker in Milwaukee - getting creamed by a beer truck. There is no other way.)

The second kind are still around, but for all practical purposes, unavailable. Their unavailability could be a physical constraint - they're committed full time to something else and simply can't squeeze in the hours any longer. Or it could be a political constraint - their new boss won't let them spend any time on something they did Back Then. Or it might even be an emotional constraint - the project went south at some point, and the developer is no longer interested in being involved. Whatever the reason, the best you're going to be able to get is get a couple of questions answered.

The third kind are still around, available, perhaps still involved in the project to some extent, but are unable to do what you're being asked to do, either due to a lack of time or a lack of knowledge. This is the best of all possible worlds, in that you can often get questions answered much more quickly than having to do the digging yourself. Yet, at the same time, your role in the project is assured. Occasionally you may find yourself at odds with the developer; they feel that you're there to take their job away. You can often gain their cooperation and trust by explaining that you're simply there to offer a second set of hands, to do the heavy lifting, and once that's done, your role is that of technology transfer, to turn things over to them and show them what they need to do to carry on. Smart developers will realize that this is an opportunity for them to learn from you, and perhaps even to garner some of the credit for taking over the project once you're gone.

## *Inventory*

Source code and data, of course, is just part of the picture. An important part, but not the only part, to be sure. Other pieces that should be on your checklist to ask for include the following:

- End user documentation. This includes the user manual, cheat sheets and help files.

- Functional specifications. These include, at a minimum, a description of the business and a glossary of company-specific terms, a list of menus, screens and reports, and a description of how each one works, and, finally, use cases.

- Technical specifications. These include database definitions (at a minimum, what does one row in each table represent and what are allowable values in each field, if not obvious?) and an object model.

## *Audit*

There are two goals to an audit. The first is to gather data for you to decide whether or not you want to take the gig.

The second is to gather data so that you can educate the customer about what's in there - and what's not.

## *Stack the deck*

While you're gathering data, you want to try to stack the deck in your favor. You're already beginning way behind the starting line. There is so much that can be unknown, the more data you have, the better off you'll be.

Write out a description of how you understand the application works, and produce numbers that describe the application - how many screens, reports, programs are in the system. How many lines of code, of those, how many lines are comments, how many are chunks of code that are simply commented out. Ted Roche has a number of papers on his website (www.tedroche.com) that provide various tools toward this end.

Being able to quantify their application helps them understand the size and scope, and just how big the project they're involved in is. The customer often doesn't' realize that the simple Sales Reporting system that they commissioned eleven years ago now comprises 32 forms, over 40 reports, with some 19,000 lines of code in over 200 files. THAT's why you're not going to be able to go in and 'make a couple of small changes' without considering the larger impact.

Producing numbers helps stack the deck in your favor when it comes time to talk with them.

## *WTF?*

These are places in the code where the previous developer worked their particular brand of magic, leaving you to scratch your head, asking, "What's This For?" While, technically, you don't have to actually put "WTF" in the code, it has been seen before.

As you're going through the code, count and document how many of these you run into. After you can show them several of these sections of 'magic code', they'll be much more appreciative of the struggle you've got in front of you.

## *Attention span*

"The key to iterative development is to keep the iteration time shorter than your attention span." I first heard those words from Alan Schwartz in 1992, and today, as I'm pushing middle age, they ring truer than ever. The larger a system is, the more opportunities there are to get distracted while you're investigating a particular issue.

If the environment is set up (or left alone) to lengthen the iteration time, the chances of distraction increase.

First, can you run the application interactively? I once worked on a system where the original developer had constructed an architecture that requires all class libraries to be converted to PRGs through a three step process. Then those PRGs needed to be built into an EXE that was placed in a separate testing directory. Iteration, then, felt more like submitting a deck of cards to the gods behind the glass window in a mainframe shop than PC-based RAD. The irony of the situation was that the original assumptions that 'required' this architecture had long gone away, yet we were stuck with this dinosaur of a project for years.

Next, does the program return the environment to the original state? On another project, the original developer took great pains to do so, with a pair of 'SET_SET' and "RESET_SET" routines called at the beginning and end of the system running. Except that somewhere along the way, someone snuck a hard-coded "SET EXACT ON" in a subroutine, and never set it back to the original state. Whenever the program terminated abnormally, EXACT was left ON, which threw any number of unexpected behaviors into the mix until VFP was restarted. Consider wrapping their program with a caller of your own that makes sure the environment is taken care of properly.

Finally, look at how easy it is to merge their application with your development environment. For example, I've always kept all my development work on a separate drive, with each customer having their own top-level directory on the drive. As a result, the first thing I've always done when inheriting a new application is to make sure it is portable, which occasionally causes conflicts with the idiots who hard-coded an installation directory on drive C.

I always fire up VFP in the same way, using an "Apps" add-on to the menu to set up the environment for a particular application. This keeps the number of shortcuts on the desktop or taskbar to a minimum. Other developers do the exact opposite, setting up shortcuts fore very customer and/or application on the desktop, with each shortcut using a custom configuration file that sets up the environment for that system. Whichever mechanism floats your boat, be sure to look at how you can incorporate this new application into your system. Having a dozen or more separate customer project directories floating around on your C drive, in the root as well as in Program Files, or, worse, needing a whole separate box, is just one more handicap in being able to iterate quickly and accurately.

## *Standards*

I mention 'standards' last not because they're unimportant, but because there are so many of them, and you're almost guaranteed not to like or agree with however the previous developer did things. Instead of picking nits (did they use 'm.' or not in front of variables?), look at the big picture. You want to get an idea of how consistent and professional the app was built.

Did they use a framework? Was it a third party commercial took, the Visual FoxPro classes, or something homegrown? Did they understand how to use it, or are many of the methods simply overwritten with hastily written and poorly documented replacements? Do you have experience with the same tool?

Do they follow some sort of conventions in terms of naming variables, programs, routines, properties and methods? White space? Indentation? Yes, I've seen 300 and 400 line routines without a single blank line or indentation. Reads like a Russian novel.

Is the data normalized? Are the field names consistent across table, or do you see "nValue" in one table and "iValue" in another?

Do they use OOP, or are they still more comfortable with procedural methods? If they've used OOP, do they understand it? Have they documented their models?

Again, there is no right or wrong answer here. What you're looking for is consistency, or the lack thereof. They may be using an arcane approach or an unusual structure to accomplish a task, but if they use the same approach throughout the app, once you've learned dhow it works, the time required to use it is greatly reduced throughout the rest of the app. If they've approached each task with a different methodology, though, then you're going to become a good archaeologist, or you're going to become bald sooner than you had expected.

# 4. Meeting of agreement

## *Set Expectations*

Once you are done with your audit and you've decided you can, and want to, take on the job, it's time to meet with them. The main goal of this meeting is to agree on expectations. It's tempting to get into the technical aspects of the project, but your main goal is to make sure you both have the same understanding of what will make the customer happy. Once that's done properly, the rest, as they say, 'is just syntax'. If, on the other hand, you don't set expectations correctly, no amount of clever programming will save you from mismatched - and unmet - expectations.

## *Rewrite*

As mentioned before, let the customer bring up the 'R' word first. As soon as you say 'Rewrite', someone is bound to ask whether or not VFP should be the target tool.

As soon as 'rewrite' shows up in someone's budget request, someone else will start asking questions that you don't want raised. If you've worked with VFP, you know that the choice of tool, beyond being a religious matter, is one where logic rarely plays a part.

Every time I've mentioned the "R" word, management gets other IT people involved, and rare is the situation where they won't take on an opportunity to get rid of one of those "damn FP apps". Even if you're going to eventually rewrite the entire app, couch the project in terms of an upgrade or, perhaps, substantial modifications and feature enhancements. Never a 'rewrite'.

We have a neighbor who wanted to, upon closer examination of the house they'd just bought, tear it down and build a new one in its place. However, the Nazis on the village zoning commission wouldn't hear of it (perhaps you've heard of this happening where you live, eh?) As a result, our neighbors ended up gutting the house, tearing out everything except for the four outer walls. At one point during the 'remodeling' process, you could stand anywhere you wanted in the basement of this three story house, look up, and see blue sky. But since the four walls stayed where they were, it was considered a remodeling project and passed muster with the village.

You may well be able to employ the same technique.

## *Wisdom*

You build apps for a living. They don't. You have experience and knowledge that they don't, but that they're paying you for. As a result, you need to demonstrate not simply 'coding smarts' but development wisdom. In fact, they want you to demonstrate that they made the right choice.

When building a new app, I explain, up front, that there will be sometime during the testing process that the bug count will grow faster than it shrinks, that at some point, they'll be convinced that I'm an idiot who doesn't know what I'm doing, and they've made the biggest mistake in their life hiring me. At this, they look a bit concerned, and then I continue, comparing this to the point in the delivery room when their wife is screaming at them, "How'd you get me into this position! The pain! I hate you!" and then the baby is born and all is wonderful and those few tense moments are now forgotten. By explaining in advance that this is going to happen, it actually relieves them, they see that I've been down this road before and will not get rattled when the journey gets bumpy. It gives them confidence.

Same thing here - you want to reassure them that you know what you're doing, and that they can depend on you to roll with the punches. A few examples of things that you might discuss with them include:

- There will be discovery throughout application.

- If there's no doc (and we already discussed that there isn't any), there exists functionality that you don't know about and that you will likely break - how will customer react?

- If there is no allowance for a RAD environment, need extra time to discover, extra time to develop.

- You will like run into 'magic structures' - code segments that aren't documented and don't seem to make sense. Some programmers try to practice "job security through code obscurity" - if they are the only ones who understand the code, then they can't be gotten rid of. Of course, this isn't true; management is more than happy to do so anyway, thinking that if one guy wrote it, another guy can figure it out. And worse, some programmers find that while it's job security, it's also prison - if they're the only once who understand it, then they're never going anywhere else!

### *Compassion*

Remember, your contact probably doesn't know what's inside, nor does he understand the difficulties involved. Be kind.

## 5. Working with the system

You've come to an agreement, you've got a 'To Do' list, and you're ready to start. There is a lot of material out there that discusses various general techniques for handling older applications. Refactoring is one big body of knowledge, although the authoritative tome on the subject, Martin Fowler's Refactoring, is somewhat longer on concepts than detailed techniques, and sure as shootin' doesn't mention anything directly relevant to VFP. Thus, here are some VFP-specific ideas on where to go next that go above and beyond the material you'll find elsewhere.

### *Backup*

Begin a new (new to you, that is) project by creating a new directory in your development drive or directory for the new customer. (This step can be skipped if you've done work for them before, and have an existing folder for them.) Name this directory with the name of the customer, not the name of the application, because you'll be creating multiple project directories underneath.

Next, create an identical pair of directory structures, one with the original source code tree and the second with a copy. Name the first "project_orig" and the second simply 'project', where 'project' is the name of the application.

Third, create a folder named 'zips', and put the original copy of the source code (and any other files) that were sent to you there. You'll keep all files you send and receive in this folder.

Finally, create a folder named 'doc', which is where you'll keep all of your notes and correspondence for the project. Some folks prefer to keep separate 'doc' directories for each project. As I find documentation tends to span projects, I keep it all in one place, and simply name individual documents appropriately for the project.

You're ready to begin.

## *Cruft*

Every project accumulates cruft over its lifetime. If you're being called in to work on an existing application, it's highly likely the previous developer didn't clean it all up before leaving. As a result, you're going to be faced with cruft. Your first task should be to clean out any obvious unnecessary materials. You can do this by creating a new project, adding the main program to it, and doing a rebuild. Then create a new directory structure and copy the files in that project to the new directory.

You'll probably need to go back to the original project and add more files in; files that are indirectly referenced, for example, or those that are marked as excluded. And, of course, the couple that you just missed during the initial transfer.

Tamar Granor has written a couple of short routines that automate this process - copying the files used in a newly built project to a new directory so that the new directory has just those that are explicitly part of the project.

## *Making copies...*

Yes, I hope you've got Rob Schneider's image firmly ensconced in your mind. As you work with the project, resist the temptation to simply make undocumented changes to existing code. Undocumented means irreversible, and that spells trouble.

You want to make a copy of whatever you're changing, so that you can quickly and easily refer back to the original. For example, suppose you're working on the "SubmitCust" method in a form. First, make a duplicate of the method, calling it "SubmitCust_orig". Some folks go even further, incorporating a 'z' to the beginning of the name so that it lands at the bottom of the list of methods in the Properties Sheet. Your decision.

Next, as I'm a belt and suspenders kind of guy, I include a 'return .f.' as the first line in the method, so that it doesn't accidentally and unexpectedly get called somewhere in the application. Yes, it shouldn't happen, but you know how Murphy is.

Once inside a program or a method, again, resist the impulse to just make a couple of small changes. In one project, I had to change function calls to include a third parm. Instead of just changing

```
m.lnTemp = UpdateTemp(this.nOriginalTemp, this.cStatus)
```

to

```
m.lnTemp = UpdateTemp(this.nOriginalTemp, this.cStatus, this.lIsCelsius)
```

I did this:

```
*] 20070314 m.lnTemp = UpdateTemp(this.nOriginalTemp, this.cStatus)
m.lnTemp = UpdateTemp(this.nOriginalTemp, this.cStatus, this.lIsCelsius)
```

so that I could quickly see the before and after in case I had to roll back the change, instead of having to go back to the original source code archive.

## *Delicious Fox Wraps*

I've mentioned this before in passing - consider writing your own wrapper for the entire application, so that you can ensure that the environment is saved and restored properly. You can use your own wrapper do perform other functions as well, such as passing login parms to the program instead of having to login manually (saving those precious seconds in iteration) and configuring data sets automatically instead of being forced to choose one from within the program. Anything you can do in an automated fashion to save time and get to the spot inside the program where you're working will repay itself both in reliability as well as a reduction in distractions.

As many MAIN programs will do a CLEAR ALL near the beginning, you may need to save the original environment to a file (remember FoxPro 'VIEWS' before Visual FoxPro's Local and Remote Views?)

## *Write Once, Forget Often*

Every have the experience of being asked how something works, and having little to no recollection? For example, you've probably got a routine in your standard code library that handles conflict resolution when two users try to update the same record. Your routine can spin through each field, comparing the user's value with what's on disk, and then present the user with mismatches. Once you've written and field-tested this chunk of code, you're done. You put the routine away in your library, the internals never to be seen again. Until someone asks you about a nuance of GETFLDSTATE and you look at them blankly. "Whatsamatter, ya don't know?" they ask. "You must not be a very good developer if you don't know what order you pass parameters to GETFLDSTATE! Every developer should know that!"

Be prepared for numerous encounters like this, if not with someone else, with yourself - finding yourself having to look up how something works because you haven't actually used AtagInfo() interactively for years.

## *Rental Cars*

When you rent a car or take out a textbook, the first thing you do is write up an 'existing damage' report, so that if you bring the car or book back and it's not in pristine, 'just like new' shape, you've got documentation that the damage was already there, and you don't have to pay for the problem.

As you begin working with this app, consider starting a document that performs the same function - in particular, detailing things that don't work or work in an unexpected fashion. It's all too easy for a customer to have 'selective memory', insisting that a function that works one way in your app worked a completely different (and, naturally, much better) way in the code that they gave you.

## ***Instant Recall***

So you've got this application with dozens of forms, hundreds of methods and subroutines, and a less-than perfect architecture. It's tough trying to keep it all in your head, particularity when you don't have a complete map of the system. It's even harder to pick up from where you left off, and regain the mental map of what you have generated so far.

Most developers stop work on a piece when they've finished it. In one respect, that makes sense, it provides peace of mind and a sense of closure. On the other hand, doing so very cleanly shuts the door on the app, enabling you to put it away - and thus it's just that much harder to pick it up again tomorrow.

Writers have the same problem - how do you most effectively pick up a story that you're working on after you've put it down? For years I'd always put my writing away when I was done with a section or a chapter, feeling comfortable that I would be able to 'start fresh' next time. However, I frequently found myself having trouble getting in the groove again, regaining the map of the book that I had developed during my last time at the keyboard. Then I heard the following tip from another writer, and have since transferred it to writing software as well.

Instead of putting your writing away when you're at a logical breaking point, continue on somewhat further. Suppose you've finished a chapter in a book. Instead of calling it a day, archive off that chapter, and then begin the next chapter. Get a good running start on it, and then stop abruptly - use a timer or an alarm if necessary. The key is to stop in the middle of writing - in the middle of a section, or, better, in the middle of a paragraph, even in the middle of a sentence if possible. If you're truly on a roll, the next time you pick up that piece of writing, you'll be forced to read what you've just put down, and by having an obvious jumping off point, the mental image that you had created will come back to you more easily, and the words will start flowing.

The same technique works well for software development. Finish off a logical segment of code - a module or part of a routine. Archive off your work. Then continue on, sketching out the next piece of code, perhaps via comments, perhaps via p-code. Write out a bit of code, and then just stop.

When it's time to pick up again, you'll be forced to go through the routine or module that you were working on, and by the time you come to the jumping off point you left yourself with, you'll have grokked where you were and be able to carry on as if you'd never left.

## ***debugo is your friend***

I have found the 'debugout' command more useful than any other single tool in terms of learning what an application does. Yes, you can use the Coverage Analyzer and the Locals window and messageboxes and other tools, but DEBUGOUT does everything these do, plus more. And you can control how it works as well.

To use, place

```
debugout 'expression'
```

in your code. When your code runs, the value of 'expression' will be displayed in the Debug Output window. For example,

```
debugout m.lcStatus
```

will result in

```
Active
```

Showing up in the Debug Output window. You do not need to concatenate strings or use TRANSFORM() to merge values with disparate data types. Instead, just separate each value with a comma, like so:

```
debugout, m.lcStatus, m.lnTemperature, m.liRowNumber
```

There is a downside to simply separating values with commas, though; I'll get to that, and a fix for it, in a moment.

### ***debugo context***

This explanation is a very bad example in all but the most trivial cases. Why? Because while it's immediately obvious to you what 'Active' represents the first time you run it, five minutes later, when you're working on the next section of code, you'll forget what this means, and in what context it was used. So it's now become useless information. The reaction is to either delete it or ignore it. Both choices are bad.

Except for those few cases where you honest-to-goodness were doing a spot check and can delete it immediately after using it, if you needed to go through the effort of checking the value once, you're probably going to need to do it again. But as we discussed, you've already forgotten the context in which it was placed (and, most likely, the meaning of the value.) Imagine a routine full of debugo statements like these, resulting in a Debug Output window that looks like this:

```
Active
1
Active
71
Active
2
Inactive
71
Inactive
3
Inactive
71
Active
```

Perhaps you're smart enough to remember what each of these values means, by itself and in relation to the others. Me, I've got more important things to do than play mind games with myself. Much better to document the context along with the value, like so:

```
debugout "Entering GetTemp routine", m.lcStatus
debugout "Processing row number", m.li
debugout "Status upon return from CheckStatus routine", m.lcStatus
debugout "Temperature at end of GetTemp routine", m.lnTemp
```

### *debugo context redux*

If you're processing DEBUGOUG commands in a loop, even the output from the previous example can be hard to follow:

```
Entering GetTemp routine, Active
Processing row number", 1
Status upon return from CheckStatus routine, Active
Temperature at end of GetTemp routine, 71
Entering GetTemp routine, Active
Processing row number", 2
Status upon return from CheckStatus routine, Inactive
Temperature at end of GetTemp routine, 71
Entering GetTemp routine, Inactive
Processing row number", 3
Status upon return from CheckStatus routine, Inactive
Temperature at end of GetTemp routine, 71
```

Consider using a visual demarcation, like so:

```
debugo "Enter GetTemp Status:", m.lcStatus, ": in row:", m.li, ": ---------"
```

You'll notice several new features here. First, the string of hyphens at the end will serve as a visual break, identifying each group of DEBUGOUT statements. Second, since I like to see as much data as possible in a single screen, I try to double up on parameters, particularly when there isn't a reason to display them on their own line. Thus I identify the routine we're entering, the initial value of m.lcStatus, and which row number we're working on, all in one line.

Finally, I use colons as delimiters before and after each value, so that it's immediately obvious if a parameter has an empty value. When you're cruising through dozens of lines of DEBUGOUT results, a line like this (produced without using delimiters):

```
Enter GetTemp Status in row 4-----
```

looks fine. However, if you look back to the line of code that generated this output, you'll see that there is no displayed value for m.lcStatus. If you delimit the values, you'll see output like so:

```
Enter GetTemp Status:: in row:4:-------
```

and now it's immediately obvious that the value for m.lcStatus is empty, which is probably a clue that something is amiss. This is particularly helpful for a line that has a value at the very end; without a closing delimiter, the output just tails off into nothingness, with no clue at all that something is missing.

If you're just using DEBUGOUT occasionally, you can get away with this level of commenting. However, in a large application with lots of methods and long routines, you can end up with hundreds of lines of output from multiple methods (and even multiple forms), and the next time you come to visit that code, you'll have no idea where the output is coming from.

For this reason,  it's a good idea to document which form and method you're in when you enter a method. Enter 'instrumenting'.

## *Instrumenting*

"Instrumenting" refers to the use of a device for recording, measuring, or controlling, especially such a device functioning as part of a control system. In the context of a software application, instrumenting can be used to determine which parts of your application are being used, how, and when.

The Coverage Analyzer does this, but it generates so much information that it's easy to lose sight of the specific details you're looking for - indeed, you spend so much time looking for one piece of information that you forget why you're looking for it. Remember the adage about attention spans?

You can instrument your applications to the extent that you need simply by including a line that calls a function that records execution plus another other relevant data at the beginning of each active method in your form, like so

```
oApp.instrument(this.name+'init', <second parm>)
```

But that means you need to write (and debug) this 'instrumenting' function. DEBUGOUT offers you a quick way to do the same thing.

The following line

```
debugout "frmTempAnalysis.GetTemp -----------------------------------------"
```

identifies which form you're in, which method, and provides a visual delimiter that you're at the beginning of the method.

## *Debugging Scaffold*

You typically don't want to provide a complete form and method trace in every DEBUGOUT statement you use because you end up with a lot of extraneous text that becomes distracting:

```
frmTempAnalysis.GetTemp -----------------------------------------
frmTempAnalysis.GetTemp Status:Active: in row:4:
frmTempAnalysis.GetTemp Temp:71:
frmTempAnalysis.GetTemp Temp After Update:71:
frmTempAnalysis.GetTemp Status After Update:Inactive:
frmTempAnalysis.GetTemp New Temp:74:
```

So you'll probably use one 'instrumenting' DEBUGOUT statement at the beginning of the method and other types of DEBUGOUT statements elsewhere:

```
frmTempAnalysis.GetTemp -----------------------------------------
Status:Active: in row:4:
Temp:71:
Temp After Update:71:
Status After Update:Inactive:
New Temp:74:
```

As your methods get more complex, you may find that you want to display certain instances of DEBUGOUT in some situations and other instances in other cases. You can use flags to turn sets of DEBUGOUT statements on and off like so:

```
if m.llshowTempDebugout
  debugout "Status upon return from CheckStatus routine", m.lcStatus
  debugout "Temperature at end of GetTemp routine", m.lnTemp
endif
```

Of course, bracketing your DEBUGOUT statements can become onerous after a while, particularly if you've got single statements throughout a routine. Enter debugox.

## *debugox*

Since every cool VFP add-on contains an 'x' in its name, my wrapper for DEBUGOUT has an 'x' in it as well. Abbreviating it to 'debugo' plus an 'x' makes it easy to type as well. DEBUGOX is a wrapper for DEBUGOUT that allows you to pass one or more parameters; the first of which is a flag that indicates whether or not the DEBUGOUT statement is to be executed. Typically, this flag is a form property or, in the cases of PRGs, a local variable. DEBUGOX can either be a form-level method or a subroutine that's part of the applications library.

The call to DEBUGOX, in this example, a form level method, looks like this:

```
thisform.debugox(thisform.lShowMethodEntry, m.lcStatus, m.lnTemp)
```

The init() method of this form initializes the property, lShowMethodEntry, so that it's consistently applied throughout the form.

DEBUGOX() looks like this:

```
lpara() m.tlShowThisInDebug, m.tu1, m.tu2, m.tu3...
if pcount() = 0
  debugout "No parms passed to debugox"
  return
endif
if m.tlShowThisInDebug
  do case
  case pcount() = 2
    debugo m.tu1
  case pcount() = 3
    debugo m.tu1, m.tu2
  case pcount() = 4
    * more code
  otherwise
    * yet more code
  endcase
endif
```

You can then pass a variety of parms in the first position to control which DEBUGOUT statements are displayed in the Debug Output window for a given scenario. Initialize them all at the same place in the form's init() method.

Be sure to include the debugox() call after any the parameter statement of any form-level methods!

## *HC*

After you've gotten in the habit of using "debugox" and documenting the form name and current method in your DEBUGOUT statements, it'll seem absolutely foreign not to do it all the time.

But once in a while you'll need to do a quick debugging session, and you know you'll be pulling out your DEBUGOUT statements as soon as you solve the puzzle.

In order to make those DEBUGOUT statements stand out from all the others in your routines, I preface them with a "HC:" string. They stand out in the Debug Output window, and this technique also makes for easy searching when it's time to get rid of them.

## *Permanent Debugox*

You can right-click on the Debug Output window and save the contents to a file. If you've followed the tips here, you'll have a permanent record of what is going on in your app, together with form and method documentation. How handy is that?

## *Data, schmata*

Up until now, you've probably been working with a chunk of data that came with the source code. This causes problems when it comes to test and prove that your code actually does what you say it does. You know what would be great? Having a clean set of data that has been proven to work. As ideal as this would be, I've never run across an app where that existed. So you'll have to create it if it. Here are some parameters that will help you create the penultimate data set to work with.

First, where is the data now? Is it all in one place? Have you identified what each table represents, and how they're related to each other?

Second, can the data be moved? Can you point to a different data set simply by changing the value of a variable or property? If not, now is the time to make this change. Being able to flip from one data set to another is a huge benefit when it comes time to test.

Third, do you have the meta-data needed to create an empty data set? You'll probably want to identify the lookup tables as opposed to the customer data, and populate the lookups along with the creation of empty tables.

Finally, make a copy of the empty data set and create your test data. Ideally, you'll want to create a series of data groups inside the data that prove various use cases. Critical is the ability to roll back to a known point after code you're testing has just trashed half of your test data. That's why the ability to flip between multiple data sets is so important.

## *Error Conditions*

"Steinbach's Guideline for Systems Programming: Never test for an error condition you don't know how to handle."

## *Steinbach's Corollary*

"If you don't know what you're doing, don't do it here."

## *Spinning Wheels*

Ever have one of those days where you set out to make one simple change, and end up at the end of the day further behind than you were when you woke up? The change seems trivial at first, but upon reflection, you realize that you're going to have to do a bit of digging to see what that last parm in the function call means. But that function isn't in the common library like most everything else. So you go up the calling chain, thinking it was popped into memory via a call in a higher level program. No luck; you've traced back to the main program and still haven't seen a reference to it.

So you decide to bite the bullet and let Code References do its thing, despite its lack of alacrity. Ten minutes later, you're left with several calls to the function, but you don't see where it's defined. You go back to the common library and look again.

It's now been an hour and a half, looking for this function, and you're no closer now than you were when you popped open your first Diet Coke. Well, that's not completely true, you do know where the function is NOT.

So then you dig out all references to "SET PROCEDURE TO", and look at every function in each of those files. No luck. Then it dawns on you - and you do a search for SET PROC TO - and you realize as several more lines show up in Code References that you must have found the answer. Except that you're again wrong.

To bring a long story to an end, the answer is that the name of procedure file that contains this function is used in direct reference to the function, like so:

```
m.lcTempProcFile = AdditionalFunctions.prg
set procedure to (m.lcTempProcFile)
```

so the file never shows up in any listings. And, of course, AdditionalFunctions.prg is buried in the "Other" folder of the source code tree.

Fortunately, this story has a happy ending. But what about those days where you call it quits at 7:30 pm, and never found the source of that function? How do you deal with those situations?

First thing, you have to realize when you're heading down into the rabbit hole. At some point, you have to stop and realize that this isn't a quick fix after all. Your own personality and experience will tell you when that is, but for me, it's usually between the second and third "Aw shit!". At that point, I back off and take the hands off the keyboard and mouse, and ask myself what's happening.

Next is to go back to the original problem and reexamine it, making sure you haven't jumped to a conclusion too soon. Do you really have to find the function, or was that just what came to mind first, for example.

Then document the steps you've taken. I find that actually writing them down helps me validate that I didn't skip any, or make false assumptions. And sometimes an 'aha' moment (like searching for "SET PROC TO" in addition to "SET PROCEDURE TO") appears as well.

Documenting the steps serves an additional purpose - explain to your customer why you spent an entire day (or week, or month?) and didn't get anything useful (from their point of view) done. It's more documentation - more evidence - for your case when you explain why a simple fix isn't always so simple.

And, of course, if you did something goofy, like search for the function but misspell it, well, writing it down may serve as the mental kick in the pants to be more careful next time.

## *Refactoring*

As I mentioned earlier, the ideas presented so far have been specific to VFP, which means they're above and beyond the general body of knowledge that's passed between the large body of VB, Java and C++ developers out there.

But we really should revisit refactoring, because you'll find yourself doing this when working with an inherited application. When you can't refactor completely, consider trying to convert existing code into black boxes that can be blocked off from the rest of the system. You just need to understand what's going in and what's coming out.

Nancy Folsom wrote a great article on Refactoring that's part of the 2006 Great Lakes conference notes.

## *Bug Reporting*

One of my favorite t-shirts is from the Chicago Morgue, carrying the legend "Our day begins when your day ends." Maybe not one you'd wear when you're delivering Girl Scout cookies with your 7 year old daughter, but completely appropriate at the tail end of a project.

When you are done with your part of the gig - your coding is done and you're ready to turn over the app to your customer for beta testing, realize that while you think you're almost done, your customer has the exact opposite point of view - his work is just starting.

And your job isn't done - and I don't mean just in terms of fixing problems that your customer finds. Part of your job as the developer is to make your customer's life easier by providing a mechanism to track issues that he finds. I call these "AFIs" - an acronym for "Application Feedback Incident" - because not all feedback is a bug. Some issues that your customer will raise are indeed defects - performance that doesn't match the specification. But other issues are misunderstandings on what the system is supposed to do. Other issues are merely questions. And even others are requests for more work in disguise - enhancement requests, or "ER's" for short.

Providing a Web-based bug reporting system can help your customer in a number of ways. First, since the back end is a database (gasp!), you'll be able to track and report on issues raised. Enabling the customer to enter every bit of feedback makes sure that you don't miss something - it's all too easy to take a phone call but then forget to address it later. And seeing their feedback in black and white - together with dates indicating when the issue was raised, when it was reported fixed (or handled), and when the fix was accepted - gives them great peace of mind. And that leads to a happy customer, which was the original goal of this whole article.

It's important that your bug reporting system forces the customer to answer the big five questions:

- General description

- Steps to reproduce

- What happened?

- What did you expect to happen?

- What do you want us to do about it?

This last question isn't the flip retort that it might seem to be at the beginning. You want to encourage the customer to enter issues that they come across, even if they're just casual observations. However, you want to make sure you understand which issues need to be fixed immediately and which are can be put off until later.

It's also even more crucial to find out 'What happened?' and 'What did you expect to happen?' with inherited systems, because with an existing system, they're much more likely to say "It was working great until you broke it!"

## *Another beer*

Let's wrap up with a couple of "Can you top this?" stories.

A colleague relates the tale of taking over an application where the previous developer wasn't aware of form properties. As a result, in the init() of each form, the developer created a series of global variables that were named after the form and used only within that form. All of these variables were then destroyed upon closure of the form. The technique worked, of course, but, well, gosh, you know.

A long time ago I inherited an app from a self-taught developer who was cleverer than he was wise. He had discovered Fox's flexibility in calling chains, and had taken great pains to make use of every permutation he could. As you know, you can

```
set procedure to <file>
```

and then functions in that <file> are available elsewhere in your application, just as if they were native Fox functions. You can also explicitly reference a function in another file without using 'set procedure to' via a call like so:

```
do <some_function> in <another file>
```

You are not limited to just one procedure file, of course, due to the magical 'additive' keyword. So you can 'set procedure to' in one place, and then, later, say in the above-reference 'another_file',

```
set procedure to <a_third_file> additive
```

so that the functions in 'a_third_file' are also available without explicit reference to where they are located.

But, of course, it gets better. What if there are identically named functions in several procedure files? There's a whole slew of rules that determine which function takes precedence, although many developers aren't familiar with them, because it would be bad practice and poor architecture to construct an application like that.

This developer was intimately familiar with all of the nuances of the calling chain and program hierarchy rules.

This application (it did logistics planning and graphing) involved some 250 programs and close to a thousand functions. Imagine the trouble you could cause in application of this size if you were so inclined.

We never did discover to what use these convolutions were undertaken. I say 'we' because I had an intern trace the program flow of this application; after two weeks of work, the dining room table was covered with sheets of paper, listing programs and functions, with circles and arrows pointing every which way, and a paragraph on the back of each explaining what each one was. Stop me if you've heard this one before.

A third application I took over came from a so-called 'professional development house'. It was a Web application where all of the logic was contained in a single 25,000 line PRG that was compiled to an EXE. There were many admirable aspects to this system, such as pages and pages of functions with names like "write_value2" (with no corresponding "write_value" function). The most amazing feature was that every reference to a Web resource (such as a graphic or an HTML page) was a fully-expanded, hard-coded reference to the live site. Yes, instead of a link like

```
<a href="cust_add.html">Add New Customer</a>
```

the reference looked like this:

```
<a href="http://www.example.com/cust_add.html">Add New Customer</a>
```

To add insult to injury, the folder structure of the live application was actually spread across two domains, so it wasn't simply a matter of replacing the domains with relative references. When asked how they tested the application, the lead developer said that they had a program that did a search and replace for the live domains, replacing with a matching pair of test domains on the same site. And in the cases where that didn't work, they just delayed testing until evening or weekends when actual users wouldn't be using the site, and they then would deploy a new EXE to the live site and test there.

Professional developers.

## Summing up

### *Compassion Redux*

No one sets out one day saying 'Ha! Today I'm going to build a piece of crap." Well, almost no one. Sure, every once in a while a malcontent or someone is forced into a bad situation, and they do the minimum. But usually, "things are the way they are because they got that way". Good decisions at the time, given limited time, knowledge, resources.