Lessons From A VFP Conversion Gone Bad

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.

There's the story of the operations research team doing studies on various military sorties during World War Two. One such study involved an analysis of Allied bombers returning from bombing missions over Europe. Each plane that returned was examined carefully for bullet holes and shrapnel damage. Maps of the location of all damage were constructed and overlaid, so as to identify where to fortify the planes with additional armor plating.

At the last minute, a second team of analysts was brought in to confirm the recommendations, and they surprisingly turned the recommendations on their head, suggesting that the planes that returned with holes were obviously capable of surviving damage to those areas, and no reinforcement was needed. They then explained that the studies had made a huge sampling error - the planes that needed to be modified were the ones that did not return!

However, it wasn't necessary to examine those planes (which was convenient, since the enemy wasn't likely to be cooperative in returning any planes that were still able to be examined.) Rather, it was obvious that the vulnerable locations on the non-returning planes were those places not damaged on the returning bombers.

Similarly, examining conversions that went well offers the same sampling error - to make your conversion a success, study the conversions that didn't go well, so that those vulnerabilities can be avoided.

Preliminaries

Before getting started with Fox stories, let's discuss the environment in which we're living.

Chicken Or Egg? Start with Code or Data?

Applications consist of data and code. What do you start with first? We're going to start with data, because the code lives to serve the data, not vice versa. Code may change. Data stays around forever.

So I'll start with stories that have to do with data.

Things Are The Way They Are...

...because they got that way. I first heard this at my first DevCon back in 1992, but later learned that the source was Jerry Weinberg years earlier, in his classic 1985 tome, “Secrets of Consulting”.

Converting an old application can be incredibly frustrating. It'd be shocking if you didn't shake your head thinking, “Why did they do THAT? What were they thinking?” This is counterproductive. So it's best to fix your mindset before doing anything else.

People don't wake up one morning and say to themselves, “Hey, I'm going to write a bunch of horrible code today!” They do the best they can with what they've got. But what they've got is

almost always imperfect, and so they usually end up with imperfect results. That imperfection may not be evident on day one, but eventually it will be.

Remind yourself, "They did the best they could with the knowledge and tools they had at the time."

The more things change, the more they change

An app from 20 years ago used different best practices than we use today. Zipping up your source code folder structure with today's date and copying it to external media was state of the art version control backup in 1995. If they automated that and it's been working fine ever since, serving their needs, well, there's a lot to be said for the 'If it's not broke" philosophy

It's tempting to go all evangelistic on them, "WHY AREN'T YOU USING GITHUB?", slamming the door and mouthing off to everyone in earshot like you're the only smart person in the office. Before you roll your eyes or bark at them, walk a mile in their shoes.

I ran into an app a couple of years ago that was virtually unreadable. Copyright dates that went back to the 80s, it hurt my head to go through it. Opening up any PRG and this is what I immediately saw:

- No blank lines to separate code segments.
- No indenting of logic structures.
- No comments.
- No spaces to separate components of expressions to help readability.
- Four letter abbreviations for every command and function.
- Variable name lengths minimized – most were one or two letters long, all were a maximum of five or six letters long.

You're shaking your head at this, just like I did, aren't you? It turns out, however, this was not the ravings of a lunatic. Rather, there were specific reasons for every one of these attributes.

The first and most important reason for all of this was that back when this application was first developed (remember the copyright dates), PRG files had a limitation of 64K, and the editor had the unfortunate behavior of just truncating a file that grew larger. A number of his PRGs were so complex (at the time) that he simply couldn't spare the extra characters that blank lines, spaces, complete command and function names, and the like required.

Second, back in those days, remember what the monitors looked like? 80X25 green screens, or maybe color. We were used to printing out our source code listings and poring over them, and it was easier to get a birds eye view of 6 or 8 pages, with few or no blank lines, rather than 16 or 18 pages that were replete with blank lines, indents, and so on.

As, yes, now you remember. We saw that style a lot 30 years ago, didn't we? But now you're arguing, things have changed. Who would keep code like that around now?

What did I just say? The developer had been living with that code for 30 years. He knew it backwards and forwards. He could open a PRG file, spin to the third page, and find an expression embedded in the middle of a line a halfway down the page in an instant. What would the benefit be to him to modify hundreds of pages of listings that he knew intimately? None whatsoever.

Sure, these days we have a multitude of solutions, but at the time, given the tools, requirements, and environment he had to work in, his approach was the optimal solution, and it's not broke. No need to 'fix' it.

Sophistication Is as Sophistication Does

Let's face it, there were a LOT of dBASE and Fox programmers (hate to use the word 'developer') in the last 80s and early 90s who got into the business primarily because they liked to tinker. They weren't really very good at what they did, they didn't have the appropriate training or background or mindset, and they weren't necessarily working to ameliorate that situation. They were just more inclined to poke at things than most others, and thus got the reputation as the local "PC guru".

I remember sitting in a meeting with one such local guru who tossed around a cookie jar full of buzzwords, impressing everyone else in the meeting, such how an 'IDX file' worked to speed up his programs, and that he "didn't use those 'CDX files' because, you see, putting all the indexes in one file, where every operation had to wait to access the same file, actually slowed things down."

Accordingly, hacks 20 or 30 years ago got away with practices that wouldn't be accepted today. Their longtime customers also have their head in the sand, worrying about their own business, no longer aware that their developer is stuck in the past.

Sometimes Good Enough is Good Enough

Back in the 80s when every company seemed to be on the 'Quality' bandwagon, everyone was trying to define just what 'Quality' was. I heard a definition that, to this day, was the most succinct definition, yet was head and shoulders above

anything else I'd heard to that point, and have so far not heard a better one.

Quality is 'fitness' for use'.

In other words, quality isn't an attribute that stands alone, that belongs to an object or process, independent of what its surroundings. It only is relevant in respect to the bigger picture. You can look at a car and unequivocally state that it's blue or white or polka-dotted. You can't look at a flat head screw that's machined to 5/10,000s of spec and ascertain whether it's of good quality or not. You need to know what it's being used for.

That flat-head screw's quality level is overkill if you're using it to attach gutters to your garage, but it's not nearly good enough if you're using it in the compressor of a jet engine.

Similarly, you don't always need the most beautiful code or the most optimized algorithms – as long as the code helps the business make money, it's really good enough. And after a point, it's not about the system, it's about the people. If the developer – regardless of how much a hack they are – has kept their customer happy for two decades, that puts them way ahead of a large group of dilettantes whose own ego puts their personal desires ahead of the needs of their customers.

But sometimes they stayed that way

All this said, it's not uncommon to run across an app that was state of the art in 1995, developed with the period's best practices, and is even now working well. Except that the developers remained mired in 1995. They got good at where they were, but never moved on.

And frankly, if they're using antiquated software, it's understandable that they didn't keep up with modern practices. A system initially written in 1992 in FoxPro 2.6 for DOS and maintained over the years, doesn't particularly lend itself to many modern developments. Simply adding the ability to send an email can sometimes be regarded as either black magic or just short of a miracle.

To say nothing of software techniques. One of Fox's cornerstones, object-oriented construction, is completely foreign to Fox 2.6 developers, and, given the age of many of them, can be met with more than a little resistance if they're looking imminent retirement.

I was talking to a customer a while ago who was having difficulty with balancing the time required to test the betas I was delivering and their daily duties of current development and in liaising with the user base. “It seems I need a block of uninterrupted time to do my best programming.”

I agreed, adding that in 1995, Tom DeMarco posited that the single most critical factor in software development was “long blocks of uninterrupted time”, and the response was, “I don't read books.”

Lesson: Keep on learning. But be aware that not everyone shares your perspective.

Old Fashioned Data

Data can take many forms, and have any number of requirements in terms of its handling – backup, archival, portability, auditing, testability. While not part of the original specification, over time, applications may acquire these capabilities.

Or maybe they don't

In the beginning, say, back in the days of dual 180Kb floppy disks, we'd create the programs and DBFs in the same place, because that's all we had – drive B.

As soon as an application was built, certain utilities were added, namely indexing, reindexing, and packing of both tables and memo fields.

More often than not, due to the fragility of the DBF file structure, a data recovery utility that repaired busted DBF headers and other file corruption problems was also part of the utility suite.

Once hard disks arrived, folders were new and magical creatures, so it was common place to load the programs and DBFs in the root of the C drive, after all, it just just like drive B, only bigger. But then we got wise and created a directory for the application.

Migrating to that directory wasn't trivial, and it was easy to to confuse things by accidentally creating a table in the root, and then not being able to find it later in the directory. Thus so we'd make sure we kept things orderly by hard-coding the directory name in front of every file. Very professional.

Then at some point, the requirement to move the application to a different drive (TWO hard drives? Who woulda thunk?) reared it's head, so it was back to the drawing board, or editor, removing the drive designation. Now files simply had the directory hardcoded, such as “\db*.dbf”.

With this extended hard disk space, faster processors, and expanded capabilities of the language, applications kept on getting bigger. Data sets that became so large that keeping all of it in one set of tables was prohibitive, either because of processing time or simply because the files became too large. Now we had multiple sets of files, one for the current data, and others for archives, maybe on a year by year basis.

Other applications grew to the point that they used multiple directories for data, one directory for each client, company or other entity.

At some point after the dropping of the drive designation of part of a file specification, some developers began storing the directory location in a variable that was configured during installation, and so now the application was completely portable across drives and custom-named folders.

Applications were now being used in mission critical applications, either on the department level of Fortune 500 firms, or running major operations of small companies. Accordingly, the data was being treated more carefully. Auditing was incorporated, so that errors in the data could be traced back to their origin. Backup was added, so that catastrophic loss was prevented.

Even further, data sets were being treated more carefully. Some systems had the ability to switch between test and production data sets, and were structured to gracefully handle an empty or incomplete data set.

These features – archival, portability, auditing, backup, and testability – have become standard fare for modern applications. But they weren't standard in days of old. I've run into applications that were missing every one of these features. On the flip side, hard disks have become many times larger than the maximum DBF size, and the DBF and CDX structures have become almost impervious to damage, and thus have seen applications whose data utilities haven't been used in so long that the users aren't aware they're not functional any longer. Both situations, much to the chagrin of either myself, when assuming it was there, or the customer, when they realized they were missing functionality they thought was there, or that they all of a sudden needed.

Lesson: When working with an ancient application, go through a checklist of basic functionality, and ask the user how the absence of a feature should be handled.

The data

So this company had developed an order and inventory management application starting with dBASE III+. They migrated it to FoxBASE+ for speed gains, then to FoxPro/DOS, and a few years later, took the big leap into GUIs, investing in a near rewrite of the interface with FoxPro 2.6 for Windows. They were savvy developers, and didn't need to change their data structures during each of these migrations.

Over the next 15 years, they kept tweaking their system, incorporating more and more features, including modules for other departments, first a link to accounting, then sales, quickly followed by customer service, then the scrap department, and so on.

Around 2011, they decided that they needed to consider the future of their app. While they hadn't had any major problems yet, Windows 7 was making their lives more complicated, and they wanted to be prepared for what came next, five to ten years down the road.

They asked me to take a look at their system. Given they were three time zones and 3,000 miles away, it was impractical to hop in the car (or plane) for a quick review. Using the major miracle that is the Internet, they arranged to zip up their application so I could look at it here.

Fortunately, they had made their application portable, so they could literally just zip up the entire set of folders and subfolders the application resided in and send it to me. Unzipping in my "dev" folder, I was able to run the app without a hitch. So then it was time to size and scope the app, to give them some metrics on how big their system was.

As with many stories that begin with "Fortunately", there's often an "unfortunately" that follows. In this case, it was the data. There was a neatly labeled folder for "data", and, indeed, it was chock full of DBFs, FPTs, CDXs, and, regretfully, not an insignificant number of IDXs. As you'll recall, this app dated back to the late '80s, so it wasn't unexpected that no one had bothered to delete those IDXs.

The most recent was over 10 years old, so it was clear they weren't being used anymore, so I created a "legacy" folder and swept all of them into that folder. Started up the app, opened each form, and all was good.

Then I started running the various reports, processes and utilities. Crash after crash. I found, incredibly, some of the routines written back then – employing the then current single index structure - were still in use. Some of the data files were static look-up tables and thus the IDXs were static as well. A couple other tables used unique indexes for reports, and while the data changed, the unique fields didn't see much activity, and again were unchanged.

Lesson: Don't just assume that ancient data structures are unneeded. Do a quick scan of the code, searching for references to those files. Then, don't just delete, archive off to a spare folder.

All the data

A customer with a system whose modules ran every department had a sizable data set, close to fifty gigabytes of DBFs.

The initial transfer of the system was done via a couple of thumb drives with the source, current

production environment and complete data set zipped, compressed, encrypted and spanned over both drives using proprietary programs. (Yes, the company was a little cautious about security.) As a result, installing on my network was a non-trivial process, involving multiple levels of decompression and decryption and installation of a few DLLs, then reconfiguring both the development and production environments, because their system was hard-coded to run on their network in several places. Once the installation was complete, doing a reinstall would require a complete wipe of my system of everything.

So, after several months of work, it was time to do an update of the data, due to a new table, some changed data structures and some test data that had been added to one group of tables. They had made it clear that they did a backup of the data each night and uploaded it to a remote server, and I could use that data whenever I needed a refresh.

Thus, I downloaded the previous night's zip file from the server (about 1.2 GB), delete the old data set, and unzipped the new zip file into the now empty folder. I continued development with the new data for a couple of weeks, and began gradually running into inexplicable problems. Some results were not the same as what the customer reported, other problems were simply errors thrown in modules that hadn't been changed, and thus shouldn't have had behavior changes.

Turns out that what they been calling "the data backup" should more accurately have been called "the partial data backup". It turned out that due to the size of the data, they didn't zip up the entire data set, just the files that had been changed that day. They never backed up the entire data set, other than when they backed up the network server itself. As a result, if they had to restore the entire data set, they had to dig out the network server backup, and then unzip every partial set on top of the intermittent set until they were caught up.

At this point, you'd think that if they had a problem, their fragmented backup strategy would only hurt themselves. But it turns out it was having an effect on me as well. I was using a partial data set. A whole batch of missing files was the problem.

And then here is the part that you tell in the bar late at night at a conference.

When I discovered what the problem was, I asked for the remaining data files. As best as I could tell, the rest would take maybe another gig and a half or two when compressed. Not that significant a difference from the gig-plus they were already backing up every night. Yet they continually refused, saying the work involved would be onerous, that it was unnecessary, repeatedly expressing puzzlement why I couldn't just to back to the original data set from the thumb drives.

As a result, for the next year, I dealt with system errors caused by missing data files that they wouldn't send me unless I asked for them one by one, until I had rebuilt the data set myself.

Lesson: Verify what their backups consist of, and keep your backups yourself, regardless of what they say and so.

And nothing but the data

Remember the company with the dBASE 3+ system and the IDXs from a few pages ago? There's more to that story.

Turns out that even the savviest of development teams runs out of steam at some point. This company's Achilles heel was keeping their system clear of cruft – such as the one-off tables that were created for a particular report that was used briefly, and then discarded. Indeed, they were good about actually removing the associated menu options from the system's menu. But the report files and the temporary data files needed for it weren't deleted at the same time, you know, just in case they were needed again some day.

Over twenty years of implementation of this philosophy meant that both the root folder of the system as well as the "data" folder were jam-packed full of data files that hadn't been used in years, in some cases, decades.

As you have likely seen yourself at some point, a folder with 1300 files, dates ranging from 1996 to 2015, is pretty difficult to parse in terms of which files are being used and which are unneeded.

So as part of the cleanup for the new system, I'd asked which tables were still being used. "Oh, yeah, we don't do a good job of deleting unused tables." and they shrugged. So I solved the problem myself:

First, I wrapped data access commands, such as USE, SELECT (SQL) and CREATE with scaffolding that identified when a table was touched, writing flags to a master list of DBFs in the folder. After running each menu option in the system, I had a list of many tables that were active.

Next, I did a search through the source code to see which tables were referenced. Naturally, this was not a sure-fire method, as macro substitution may have been used. Still, it provided

more data about which tables were definitely still current.

I then presented this solution to the company, for implementation on the production system, at which point they told me that they have another part of their system, a set of batch files built over decades, located on a different part of the network, that touches this data set. As a result, while my work did identify which tables were used by the system in question, it didn't identify what else touched those files.

Lesson: Don't assume that your system is the only part of their IT infrastructure.

No, really, nothing but the data

The dBASE 3+ system had one more unexpected issue. They used a home-grown data dictionary for two groups of functions. The first was storing data about the tables, such as field descriptions, indexes, and so on, to support data utilities, such as indexing and packing. The other group of functions were user interface oriented – field formats and input masks, calls to generic validation routines, attributes such as "required" and "unique", and so on. No problem in either of these cases.

Turns out that their data dictionary was the electronic version of the Roach Motel – tables check in, but they never check out. Once a table's information was added to the data dictionary, it was never removed, even if the table itself was no longer in use. However, as a result of some preventative coding, a table didn't need to exist even if it was in the data dictionary, routines that spun through the metadata simply ignored missing tables. (Since this was a captive application, the developers didn't feel the need to warn about missing tables.)

As a result, it wasn't possible to determine what tables on disk were cruft simply by looking at the data dictionary. On the other hand, deleting tables on disk might result in an unstable system, even though the data dictionary wouldn't warn you that doing so was a Bad Thing.

Lesson: The data dictionary might be just like other dictionaries – full of spurious entries.

It Seemed Like A Good Idea

The reason that most of us are in business is because people want custom applications. They want things to work exactly the way THEY want. As a result, standardized features and mechanisms aren't always part of the package. Indeed, custom developers will come up with the most ingenious ways to solve a problem, even if there was an off-the-shelf solution out there.

Only Part of the Solution

You remember the dBASE 3+ system whose data dictionary had entries for tables that didn't have to exist. Fortunately, most systems aren't like that. More commonly, when a data dictionary routine encounters a table entry for which there is no file on disk, one of two things happens.

The first option is to warn the user that the table is missing, and to ask whether the table should be created or not.

The second option is to automatically create the table without prompting.

So far, so good.

But this is only the first part of the solution. What if the user answers 'no' to the "Table X is missing. Create it again?" question? Will the system handle a missing table gracefully?

Next, once you've got a new table, what next? It's empty, after all. Many home-grown systems assume there is data in every table. After all, once the table was created and populated, there wouldn't be any reason for the table to be emptied, so those types of safeguards were never written.

Some empty table problems cause an error to be thrown, such as the routine that counts how many records are in a table and positions the record pointer on the last record. "GO liRecNo" doesn't behave well when there aren't any records in the table.

Other empty table problems are more subtle. For example, a JOIN that assumes the existence of records in lookup tables will return an empty set if there are no lookup records, thus misleading the user into thinking there are no records at all for that query.

Lesson: Ask what behaviors should be occurring in the situations of missing and empty tables – even if "that would never happen."

DBFs Are Just Files

One of the downsides of xBASE applications has always been that the data files are simply files on disk, without any inherent protection from external tomfoolery. As Windows became more sophisticated, it was possible to lock down the folder with the data via rights, but that's not always done.

I consulted on an application for a bank that started losing index files on a regular basis. The system had been written in the pre-CDX days, and thus there were hundreds of IDX files in the data folder. Every week or so, a small batch of the IDXs disappeared.

As is often the case with these older applications, the system had been written by a highly skilled developer at the time. After a few

years, it had been handed down to another developer who didn't need to spend as much time on it, and thus wasn't as skilled. The third developer, now ten years later, was more of a part-time programmer than a developer, and was tasked with occasional maintenance. As a result, the code started looking like it had been poked with a sharp stick, and became increasingly difficult to follow.

When the IDX files began to disappear, they resorted to a number of temporary patches, but as the problem became endemic, they decided they had to fix it properly. I found there was no automated index recreation routine, just a page of notes that described which indexes belonged to which tables, and a standalone PRG that recreated them.

As the first step to fixing a problem is attempting to replicate it, I tracked down which IDX files were going MIA. It was truly a mystery, there seemed to be no pattern to which ones were disappearing. The same ones were vanishing each week, but one table's IDXs were never touched while another table had the same four go AWOL regularly.

Until one day I sorted the list of missing files and realized there WAS a pattern. Turns out they used a naming convention for the IDX files whereby the first three letters of the first fields in the index expression were used to create the IDX name. So, for example, the IDX file that indexed on a person's title and the branch they worked at had a six letter name. And the IDX file that indexed on a person's shift, the theatre they worked in (a sub-group of employees), and the administrative code had an eight letter IDX name. Seriously.

And not so long ago, they had hired a new administrative assistant who had taken it upon herself to police the IT infrastructure, including deleting files that appeared to be inappropriate. And her hire date was when the IDX files started disappearing.

Problem solved.

Lesson: There is no lesson, it's just a great story. Or maybe the lesson is "when you least expect it, expect it."

The Mostly Not Needed Hardcoded Path

As mentioned earlier, the proper use of paths is now a foregone conclusion, but it wasn't always that way. There are a lot of applications out there that assume that data will be located in C:\DBDATA or somewhere similar, and thus have that path hard-coded throughout the code.

As the developer's skill level evolved, it wasn't uncommon for them to start refactoring the application to use a variable to hold the dataset path, initializing that variable during startup, perhaps via a config file.

However, sometimes this approach was only done partway – new code used the path variable, and as old code was modified, the path variable replaced the hard-coded path, but a full scale search and destroy effort on the hard-coded path was never implemented.

As a result, there are still places in the code that refer to the original "C:\DBDATA" location, and that code has never been discovered, because their installation of the application still has data in that folder. But when you install a version of the application on your machine, resulting in the data landing in H:\DEV\CUST_A\DBDATA.

Lesson: The first time you run into a hard-coded path, do a search for that string throughout the entire application.

If One Is Good, Then Two Must Be Better

One bit of fallout from the gradual implementation of paths for datasets can be mistakes made while coming up that learning curve. As a result, it's not uncommon for the same data file to be found both in the root folder of the application as well as the data folder.

If the table is a lookup or minor entity that is rarely or never updated, then having duplicate versions won't be much of a problem.

If it gets changed all the time, a problem will surface quickly and require repair.

However, if it's updated intermittently, it may not be obvious that the application is pulling data out of one version or the other at different times. Say it's a lookup table with a list of codes for transactions, and those codes get updated a couple of times a year for new types of transactions.

Reports run against the old version of the table may result in the most recent transactions (using the new codes) being left off the report, and no one will be aware until it's obvious that there is a LOT of missing data.

Lesson: Do a complete sweep of the data tables in the application folders, looking for duplicates.

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

Without external inputs, the way we react to stimuli is conditioned by our past experiences. These external inputs are collectively called 'context'. The way we interpret a word is based on our history with that word, unless we are given a different context.

As someone with a long history in manufacturing, the words 'raw material', 'part', 'assembly' and 'component' have very specific meanings in the food chain. Thus, when I hard someone refer to a transmission provided by a supplier as a 'raw material', I was understandably confused – generally, a transmission would be called a component.

Similarly, much confusion abounded when a customer used the term 'part' for what turned out to be a combination of parts – an assembly. That required a significant reworking of the data structures.

So when a customer mentioned they had an auditing feature in their system that tracked all changes to the data, I assumed that, based on my experience with other auditing systems, 'changes' included adds and deletes. Turns out, not so much.

When they wrote their auditing mechanism, their only concern was just modifications to existing entries in certain tables. They didn't really care about new records or records that were deleted – every record had fields for 'added by' and 'deleted by' that provided that tracking capability. No, their auditing mechanism only recorded actual changes to contents in a discrete field. I ended up reworking the new auditing mechanism to satisfy their interpretation of the word 'changes'.

Lesson: Spell out what the words mean. Definite the exact functionality.

Conclusions

These may seem like trivial lessons. As you read through them, 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. So consider this article as a checklist. Airline pilots, even the most experienced, still use a checklist every time they get in a plane. You should too.

I've compiled a similar group of stories, together with corresponding lessons, on code. Based on the feedback I get on this article, I'll put them together in a future article.

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