F# Out of Necessity

By Peter Santoro

Web/XML Team Lead

CT Department of Revenue Services (CT-DRS)

Abstract

The F# programming language has proven itself to be a very effective tool for dealing with the multiple challenges of tax processing. CT-DRS currently uses F# to process electronic tax filings 24/7, reduce agency costs, detect and/or correct processing errors in non-F# systems, and provide desktop productivity enhancement tools to our help desk and business users. F# has turned out to be a more productive language for developing correct software than C#, Python, and VB.NET.

Introduction

Developing and maintaining State government tax systems is admittedly not the most exciting problem domain, but it can be very challenging for software developers given the players involved and their expectations. The demand for tax software change requests is driven by a steady stream of legislative changes to tax law. Software that undergoes constant change tends to increase its entropy[1] and complexity. To make matters worse, the workload demands on the Web/XML team have steadily increased over time – without an increase in staff.

What follows is the story of why I decided to roll the software development dice with F#. Prior to discussing the current use of F# at CT-DRS, a discussion of tax processing and its challenges will give the reader more incite as to why F# was chosen out of necessity.

Overview of Tax Processing in the United States

In the United States, tax software processing adheres to the following standards: IRS MeF (Internal Revenue Service Modernized Efile), FTA (Federation of Tax Administrators[2]) and NACHA (Electronic Payments Association[3]). Tax forms are specified using XML schema[4]. States

[1]Software entropy is a measure of its disorder.

[2]Previously State XML standards were managed by TIGERS (ANSI ASC X12 Tax Information Interchange Task Group).

[3]Formally known as National Automated Clearing House Association.

[4]FTA State MeF XML schemas piggyback off the IRS Federal MeF XML schemas. The FTA State MeF XML schemas are then used as the starting point for all State-specific MeF XML schemas.

are required to get their XML schemas for new tax forms reviewed and approved by the FTA prior to production use. This review process is to assure that States are following the FTA Standards and that the Tax Software Processing Industry has no issues with supporting the given State's requirements.

Figure 1 depicts the high level MeF processing data-flow, wherein the taxpayer submits electronic tax forms and receives an associated acknowledgment. However, this MeF tax processing diagram is incomplete in that all accepted taxpayer submissions are also routed to the CT-DRS main tax processing system (ITAS) for further processing. MeF accepted taxpayer submissions are currently routed to the ITAS system via file export/import.

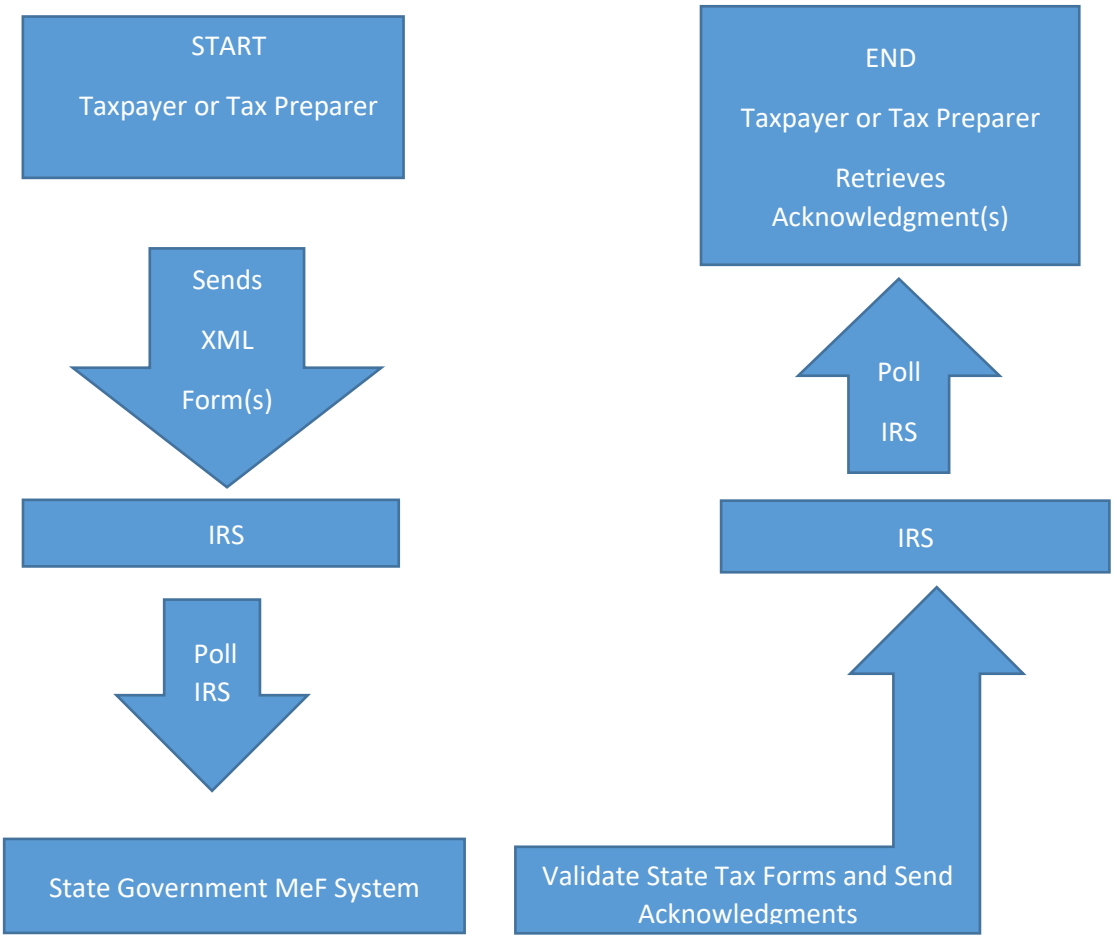

Figure 1 – MeF data-flow starts with the taxpayer sending of tax form(s) and ends with the retrieval of the associated State acknowledgment(s)[5].

[5]Acknowledgments may be either Accepted or Rejected, where Rejected acknowledgments include the associated reason(s) for the rejection.

Initial CT-DRS Vendor MeF Implementation

In late 2010, CT-DRS contracted with a MeF COTS[6] product vendor to implement the initial CT-DRS MeF software. The contract stipulated that the vendor was responsible for all first year support and maintenance. At the request of Agency management, the vendor was asked to deliver a single integrated solution which supported the retrieval/acknowledgment of taxpayer submissions and routing of taxpayer data to ITAS via file export. The thought process behind this decision was that it would make vendor support easier by isolating everything MeF in one place. In practice, this tight coupling actually complicated ongoing support and maintenance for the Agency.

The vendor's MeF COTS product is implemented as a Windows .NET service (one per licensed tax type) and uses a single Microsoft SQL Server instance for its data store. State customization is supported via in-process DLL plug-ins. Plug-ins can be written to support various processing extension points via OOP[7] inheritance and/or interfaces. The initial contract was for one tax type (individual income tax) with two tax forms. The original custom logic for CT-DRS was implemented using four C# plug-in DLLs.

Around this time, an edict from IT management mandated that all new non-ITAS software development had to be done using the VB.NET language. The thought process behind that decision was that VB programmers are cheap and easy to find. Warnings to management that VB.NET and VB are two different animals were ignored. As a result of this edict, the team now had to support MeF using both VB.NET and C#, which made it more difficult to find qualified staff.

After the first year of MeF processing, it was expected that CT-DRS personnel would be responsible for all State-related MeF development, maintenance, and support. Vendor fees for subsequent years covered the licensing and support for each MeF Windows Service deployed. Currently, CT-DRS processes the following MeF tax types: Corporation, Estates/Trusts, Individual Income, and Partnership.

[6]Commercial off-the-shelf.
[7]Object Oriented Programming.

Hard deadlines dictated by internal/external partners
Extremely short development timelines (often 1-4 months)
Software errors increase financial risk, are highly visible internally, and can negatively impact public relations
Yearly requirement changes mandated by external partners and CT legislature are sometimes very significant
Late feedback and/or requirement changes from internal/external partners is very common
Sanctioned development efforts are often limited to business user change requests
Tax code complexity normally increases over time
Government hiring process and union rules make acquiring talent difficult
FTA approval of State XML schemas can be rescinded in the future
Current and prior two tax years must be supported
Insufficient computing resources to support realistic performance testing
Well-meant, but shortsighted technology edicts

Table 1 – Challenges of State MeF Tax Software Development

State Tax Software Development Challenges

Some of the challenges associated with State tax software development are listed in Table 1. Yearly requirement changes mandated by internal/external partners and CT legislators for the next tax year can be significant, often come very late in the year, and always come with hard deadlines. These factors often result in a compressed/rushed software development timeline. Another important factor to consider is that tax software processing errors can result in a public relations nightmare for government leaders. Together, these factors lead to an environment where sanctioned software development efforts are often limited to business user change requests. Unfortunately, this restriction on software development efforts has a nasty side effect, which needs further explanation.

Although many organizations are increasingly dependent upon software automation, they are rarely run like a well-oiled software product development company. This mismatch often leads to a huge disconnect between business operations and the needs that their software developers have in order to produce high quality software. Over time, this disconnect often results in the creation of a big ball of mud[8] system. The typical business solution to a big ball

[8]Big ball of mud systems appear to lack any coherent design and are very difficult to maintain. They are often the result of many requirement changes over time – without any commitment and effort to re-architect/refactor the software to better support future

of mud system is to purchase a new, more highly integrated, system that is marketed to support all business requirements in one neat package. Over time, these new shiny systems almost always turn into the organization's next big ball of mud system. It's an unfortunate fact that many business leaders simply consider software a necessary, but unwanted and evil expense – and accept no responsibility for their organization's software failures.

To complicate matters further, some CT-DRS business users still appear to erroneously believe that they have until mid-January of the new tax year to finalize changes to that year's tax forms, which is wishful thinking at best. IRS, State, and Tax Software Industry MeF testing for the next tax year normally starts at least two months prior to the start of the new tax year.

The FTA XML schema review/approval process, mentioned previously, is generally a good thing. However, once States get their XML schemas approved for production use, this approval can be rescinded by the FTA if a tax software vendor objects at a later date. This doesn't happen often, but it happened to CT-DRS in the early MeF years. A tax software vendor's implementation could not handle the nesting of elements at a certain point in a tax form – not because our XML schemas were invalid or didn't follow the FTA Standards. This resulted in CT-DRS having to modify its MeF XML schemas right before the start of the new tax year. Unfortunately, tax software vendor implementation limitations are like land mines in that they are hidden.

FTA approval of State XML schemas can also be rescinded for other reasons, too. Years ago, NACHA mandated electronic payment changes to FTA MeF and non-MeF XML schemas. In general, the FTA expected all States to update to the latest FTA schema versions to get these mandated changes. CT-DRS was granted an exception to this ruling for one legacy non-MeF system and was allowed to add the mandated changes to an older XML schema version. This exception was rescinded about six months later (after development and testing were completed), when other states asked for the same exception. The FTA mandated that all of these states needed to use a common, updated, older XML schema version.

Getting requirements late is always a problem for developers. Not getting critical requirements is even worse. For example, when it was time to support the Partnership tax type in MeF, all known business requirements were dutifully implemented on time. All appeared well, until a few tax software vendors started reporting IRS errors to the business users during late November testing. Eventually, this issue was brought to my team's attention and the IRS was contacted for assistance. The IRS individual said, "The issue is simple. The software vendor is trying to send a corporate return through the IRS partnership MeF system – that's why it's being rejected." This response was at first puzzling, until additional

change. Software malleability does not come for free.

communication with the IRS revealed the root cause of the problem. To make a long story short, here's what was learned. The IRS classifies Subchapter S Corporate (SCORP) tax filings as corporate filings. However, CT and a few other states, long ago merged SCORP and Partnership filings together, given their tax treatment similarities. When this information was brought to the attention of the MeF business users, their initial reaction was one of blank stares. Not surprisingly, many industry tax software developers were also not familiar with this issue. Eventually, one of the more experienced CT-DRS business users recalled the past merging of SCORP and Partnership tax forms. Satisfying this missing requirement meant that our Corporate MeF Windows Service would now have to support the processing of both Corporate and Partnership filings. Given that the existing MeF OOP code-base was designed/implemented around the vendor licensing scheme of supporting a single tax type per Windows service, it turned out to be an extremely difficult/risky task to undertake so late in the year. The task was completed on time, but only by working a lot of extra hours. This hardened my resolve to try and find a more efficient/safer way to deal with late breaking changes.

Performance and correctness are critical when writing tax processing software. Not only do taxpayers expect to receive their refunds in a timely manner, but they expect that their tax returns will be processed correctly. The failure to do either will likely make the nightly news.

Information Technology Management and *New* Programming Languages

Most large organizations, like CT-DRS, are extremely conservative in their use of technologies. It takes an act of God to obtain approval to use new technologies, including programming languages. Shortly after joining CT-DRS in 2008, I requested permission to use Python. After many months, the team was granted limited permission to use Python as a scripting tool to simplify some of our development tasks. Arguably the most important Python tool written was a MeF regression testing tool[9]. Yes, that's correct, Python was used to regression test critical MeF .NET applications, which enabled faster/safer refactoring.

In 2013, I attended an urgent, late Friday meeting, right before leaving for a scheduled long weekend with my family. Months earlier, a decision was made to let an expensive software license expire and use an internally developed application instead. Unbeknownst to the IT staff until that Friday, the replacement application lacked an important reporting feature used by our mail room. A replacement solution for this missing feature was needed in production by the following Wednesday. I volunteered for this task, with the proviso that the software be written in Python. Given that there were no other solutions available, permission was granted

[9]The core of this application relied on the excellent pytest package.

to use Python in production. After successful delivery of this Python application, the use of Python in the CT-DRS production environment grew slowly over time. Given that Python comes with "batteries included", it was especially useful when timely solutions were required.

Of course there was a downside to adding Python to the mix languages the team supported in production. We now had to support applications written using the following languages/run-times:

Classic ASP/VBScript, C#, Java, Javascript, VB6, VB.NET, and Python

Polyglot programming loses its appeal quickly, when one considers the extra work required to support each additional language in production with a small staff.

Web/XML Team Growing Pains

Working for a government agency, with limited funds and having to deal with very restrictive hiring practices, makes obtaining qualified staff extremely difficult. As a result, the WebXML team rarely consisted of more than two employees. However, this staffing limitation did not prevent management from continually increasing the team's workload. So to be perfectly clear here, the reference to team growing pains is not about increasing team size, but refers to the constantly increasing workload expected of the team.

In addition to the team's assigned work, fairly large refactoring tasks were also performed (see Table 2). Some of these tasks were in support of assigned work, but many were to improve performance/correctness and/or reduce the level of development effort going forward. Unfortunately, refactoring OOP source code can be incredibly difficult and risky, as most OOP language compilers and type systems are quite limited in this respect. Too much work is left up to the eyes of developers.

The team's most important, long term, assigned task was to undo the integration of the MeF retrieval/acknowledgment of taxpayer submissions and the file export process. As far as some of the business users were concerned, this was a simple matter of *moving* logic from one server to another. Unfortunately, the initial vendor C# implementation would not support such a simple plan.

Not only did the vendor implementation use what were now orphaned Open Source dependencies (e.g. Spring.NET), the file export logic was written to run inside the vendor's designed Windows service container, with all its constraints and dependencies. It would be unwise to use orphaned dependencies and support the vendor's execution constraints going forward. This of course meant that a rewrite was necessary. That's when I first thought that

there must be a better way to implement this critical logic than to somehow try to reuse the existing C# code base.

Refactoring/Development Tasks In Approximate Implementation Order	**Reason(s) For Doing the Work**
Allow the processing and/or the IRS received dates to be changed via application configuration.	Tax processing is date driven and adequate testing of business logic and rules requires the ability to changes these dates.
Simplify the implementation of annual tax processing changes to the file export process logic.	Vendor supplied methodology was labor intensive and brittle.
Update APIs to support multiple tax forms, multiple tax years, and multiple schema versions per tax year.	Original vendor APIs only supported one tax year, one schema version per year, and only two tax forms. This is exactly what was contracted for, but hardly future proof.
Support multiple tax types per Windows service.	Required for CT-DRS SCORP tax filings.
Rewrite database layer to improve performance.	Database performance deteriorated over time. During peak processing periods the SQL Server database was 100% utilized.
Rewrite XML processing layer to improve performance.	Vendor supplied logic worked, but was not optimized.
Split MeF database into operational and historical data stores. Provide a yearly migration path of current tax data from the operation database to the historical database.	The number of historical data users was growing over time and this database activity should not be allowed to impact current year processing.
Develop new logic to support the new CT-1120CU tax form.	Required for tax year 2016.
Undo the integration of the MeF retrieval/acknowledgment of taxpayer submissions with the file export process.	This integration complicated ongoing maintenance for a number of teams and required precise communications between the teams – which did not always occur.
Develop new logic to support the completely revamped CT-10651120SI tax form.	Required for tax year 2018.
Reduce the complexity associated with the fact that the FTA defined tax year begin and end dates XML elements are optional.	These dates are mandatory for CT-DRS processing.

Table 2 – Significant MeF Software Maintenance Efforts to Combat the Big Ball of Mud Syndrome

F# to the Rescue

My initial exposure to functional programming was in the mid 1980's, while earning an M.S. Degree in Computer Science. Although I've written a significant amount of OOP code in a variety of languages over the years to earn a living, I haven't been a fan of the OOP paradigm for decades. For example, my C# code makes heavy use of sealed classes[10] as *modules* and static methods as mostly *pure functions*[11]. I find this "poor man's" way of functional programming in C# to be a simpler/less verbose way to improve program correctness than the alternatives, but admittedly it looks strange to most C# programmers. I often thought about using a functional programming language to tackle the ever-increasing workload at CT-DRS. Although I knew about F# shortly after its release in 2005, it wasn't until F# version 3.1 was announced in 2013 that I decided to give it a try.

My first F# program, written in late in 2013, was a command line tool to rename UNC[12] paths in desktop links. Such a tool was needed internally anyway, so why not try writing it in F#? Initially, learning to *dance* with the F# compiler, who was in the lead, was at times frustrating. I eventually got the hang of it and came to greatly appreciate the huge advantage of *dancing* with a smart compiler and an advanced type system that, if used properly, had my back.

After this initial small F# success, a multi-step plan was created to take advantage of F# in order to improve the quality, performance, and delivery of the team's software. The initial goals were to 1) break the integration of the file export process from the retrieval/acknowledgment of MeF submissions, 2) reduce ongoing software maintenance, and 3) reduce the languages/run-times the team had to support going forward. Given the risks associated with such a plan, I opted to do this experimental F# work at home and on my own time. Periodic system testing could be performed at work, in the background, supported by the MeF automated regression testing tools.

A summary of the NextGen plan is shown in Table 3. Given the team's other higher priority assignments (development and support of existing systems), earnest work on this plan didn't start until late 2014.

As usual, I decided to implement the most risky/critical NextGen application first (i.e. undo the MeF integration noted earlier). After all, the Python MeF regression test tool could be used to verify the F# actual results against the C# expected results. If this worked out well, the plan was to roll out less risky F# applications to production first. To ease the effort of having to support two completely different code-bases (i.e. C#/VB.NET vs. F#), the team's build and deployment scripts were updated to support both.

[10]Sealed classes prevent the use of OOP inheritance.
[11]Pure functions have no side effects and return the same result for a given input.
[12]Uniform Naming Convention.

Step	Description of F# Work	Expected Benefits
1	Develop MeF File Processing As Separate Process From IRS MeF Communication Gateways	Decoupling these processes should eliminate communication/coordination efforts/problems and ease ongoing maintenance and support. New file processing logic should support auto-reconciliation to simplify support and troubleshooting.
2	Develop Replacements for Python User and Help Desk Tools	Built-in Python Tcl/Tk GUI libraries are somewhat outdated and the resulting GUIs are less familiar to Windows users. New GUIs should be more capable and easier to use.
3	Assist Production Control Team with Creating New Job Configurations	Ease future migration to the new NextGen jobs.
4	Develop Replacements for Existing Python Production Scheduled Jobs	Simplify or reduce job configurations for Production Control Team. Reduce future support and maintenance work by reducing the number of supported languages and associated run-times.
5	As Needed, Develop New Tools for Production Control	Monitor secure ftp bank transfers and alert on failures.
6	Develop Replacement for the Existing Production C# Address Standardization Windows Service	Simplify support and improve correctness/performance.
7	Develop Replacements for Existing VB.NET Production Scheduled Jobs	Utilize concurrency to reduce the number of jobs (which reduces licensing costs), improve performance, and simplify configurations for Production Control Team.
8	Develop Replacement for Existing MeF C# Plug-ins	Improve business rule performance (i.e. decrease service time) in order to speed turnaround times for taxpayers. Reduce future annual maintenance effort.
9	Develop Better Automated Testing Tools	Testing should be more thorough and take less time.
10	Migrate Steps 1, 7, and 8 to Production	Improve tax processing performance and correctness, reduce the number of production jobs and simplify their configuration, and reduce coordination between teams.

Table 3 – NextGen Migration Plan

The NextGen applications were marketed as significant enhancements that streamlined processing[13]. Although F# usage was not specifically discussed, I openly used snippets of F# source code to explain to business users and management the impact of various proposed changes. That's another great benefit of F# functional code vs. standard OOP code. F# source code can be much easier for non-programmers to comprehend. The F# type system, if used properly, also makes it much easier to answer "what's the impact of this change" questions. Just change the associated type(s), as proposed, and recompile. The F# compiler will dutifully show you, via errors and warnings, where you will need to make additional changes to accommodate the proposed change(s).

A layered architecture was used for all of the team's F# work. The bottom layer contains the core modules common to all application domains. Sitting above the core layer are the application specific domain modules, above which sit the associated applications/services.

After about 2.5 years of working part time at home, the initial production roll-out of low risk F# applications occurred in late 2017. Mind you, these applications were not simple ports to F#. Significant enhancements were used to win over management, business users, help desk personnel, and the production control team.

Desktop Windows Forms (WinForms) Tools in F#

Previous applications created for the MeF business users and help desk personnel were written in Python[14]. Rewriting these applications in F# would be a good low-risk way to try out F# in production and a great way to build user trust in the NextGen brand. To entice both user groups to make the switch, the NextGen brand of applications included significant improvements.

For the help desk personnel, this amounted to replacing command line tools with an F# GUI[15] version that offered an easier/quicker way to accomplish their work. The F# application also alerted users of possible issues, which helped to minimize mistakes. The NextGen help desk GUI application was much preferred over the previous command-line tools.

For the MeF business users, the NextGen replacement, not only needed to compete with the existing Python GUI application, but it needed to offer significant enhancements not supported by our MeF Vendor's GUI Tools. For example, the MeF Vendor's GUI tools were designed to

[13]Enhancements included: decoupling of the aforementioned problematic MeF integration, improving performance, reducing the number of scheduled production jobs, and reducing the number of supported languages/run-times to a single Microsoft supported .NET language.

[14]Python Tcl/Tk was used for GUI applications.

[15]MS Visual Studio's Windows Forms designer was used to generate C# WinForms code, which was translated by hand to F#.

be used locally and so required the business users to log into different servers in order to perform many of their tasks. The NextGen version needed to allow business users to perform the majority of their remote tasks directly from their desktop and minimize/correct any configuration errors. Not having to log into various servers to get their work done was a real time saver for the MeF business users and this was much appreciated.

Replace Python Production Applications with F# Alternatives

As noted earlier, Python was used in production for a variety of tasks. Although, Python worked well, it added to the number of languages/run-times the team had to support. Sometimes this effort was significant, like when all Python 2 applications were migrated to Python 3.

Prior production data and the previously produced Python results were used to assist with developing/verifying these F# replacement applications. To convince the business users of the correctness of these new .NET NextGen applications, the team provided them with copies of the Python and NextGen output from numerous executions. This helped to reduce the fear of moving these NextGen replacement applications into production. All of these F# replacement applications went into production without a hitch. Again, this helped build user and management trust in the NextGen brand.

About this time, I also decided to replace the existing Python MeF automated regression testing tool, so that the team could drop Python support completely. Don't get me wrong, the pytest based tool worked great, but was slow[16] and required the creation/maintenance of test data – which itself had to be correct. A faster, more capable automated regression testing tool was needed that didn't require the creation/maintenance of regression test data. It should also support the comparison of a wider variety of actual and expected result data. The existing Python based MeF automated test tool only supported looking for differences in file-based data. Additional assurances were needed before moving any critical NextGen applications to production. This meant adding the ability to catch differences in actual versus expected results in database tables, too. Of course, this new regression testing tool was written in F#.

Replace C# Production Address Standardization Service with an F# Alternative

In order to reduce postage costs, CT-DRS uses a vendor product that creates an address in USPS[17] standard format from an address that may not be in USPS standard format. By using

[16]Pytest, by design, scans the file-system.
[17]United States Postal Service.

the USPS standard format, the Agency is able to obtain significant postage discounts. Initially this vendor product was only supported as a desktop application. Before CT-DRS hired me in 2008, other employees and consultants, with the vendor's permission (but no support), attempted to write a C# service wrapper around this product to allow it to be callable by remote applications. Unfortunately, after many months of effort that team never got the service to work correctly. Shortly after arriving at CT-DRS, I offered to assist with this effort. After looking at the source code, I recommended a rewrite, but management said no to that suggestion. I spent the next 4-5 weeks debugging, patching, and testing the code to get it to work reliably – which it did for a number of years – until we migrated it to a virtual machine. After migrating this Windows service to a VM on faster hardware, it would occasionally fail and negatively impact tax processing. Since the service lacked sufficient instrumentation, it was difficult to diagnose and fix these failures.

However, I was pretty certain that these occasional failures were caused by a mixture of mutable state and multi-threading logic. This seemed like the next good candidate to be replaced with F#. As with the other F# work, this would not be a simple port - the existing C# code was obviously broken. In this case, the concurrency model was switched from .NET 2.0 threads to agents[18] and error detection/logging/ instrumentation were enhanced.

During testing of this new F# service, a couple of minor logic errors were uncovered and fixed. F# is a great language that, if used correctly, can prevent many types of programming errors. Unfortunately, it is unlikely that any programming language will ever be capable of detecting/preventing all errors – especially logic errors. The new F# service has been in production for about a year now without a single reported hiccup. Again, this further solidified user and management trust in the NextGen brand.

Using F# to Decouple the Retrieval/Acknowledgment of MeF Taxpayer Submissions and File Export Processing

Most of the development of the F# file processing logic took place during 2016 -2017 and was working quite well by the beginning of the summer in 2017. One expected benefit of this work was a reduction the number of lines of code. F# did not disappoint. The F# code-base was significantly smaller (about 60% fewer lines) than the legacy C# code-base and was much easier to comprehend[19]. And this is noteworthy when you consider that the F# code-base includes some killer features (e.g. concurrent processing, fully automated detailed reconciliation) not found in the C# code-base.

[18]Agents were constructed using the F# MailboxProcessor type.
[19]It contained significantly less noise.

Things were going so well, that a decision was made to also replace the Windows Service C# plug-ins with F# plug-ins, as doing so would eliminate the need for the team to support C# going forward. This was another high risk task, but I was fairly confident that the new F# automated regression tool would catch most issues. Again, this was not a simple port of the existing C# plug-in code-base to F#. Goals for the NextGen version included 1) reduce the service time associated with processing taxpayer submissions and 2) reduce the team's annual maintenance efforts. Using F#, both goals were accomplished in record time. Although the business users were initially apprehensive about making any changes to the existing/working plug-ins; they liked the end result – which included about a 10-fold decrease in the service time to process taxpayer submissions. Honestly, given the magnitude of the changes that were made to the plug-in's internal design and implementation, it would have been unwise to attempt this work using the existing C# code-base and the (IMHO) inferior C# compiler and type system.

Around the end of summer in 2017, development work began for the next tax year. This was a good opportunity to compare the refactoring efforts required for the next tax year between the C# and F# code-bases. This comparison had already been done for the previous tax year changes, but now that the F# code-base was feature complete, it would be a more accurate comparison. Needless to say, refactoring the much smaller F# code-base required significantly less work and testing than the C# code-base. Why less testing? If the F# type system is used properly, a wide variety of errors can be eliminated – including making illegal states un-representable.

During 2018, the new F# automated testing tool was aggressively used to test/debug the remaining F# NextGen applications not yet in production. Along this journey, testing uncovered some discrepancies between what the experimental F# code-base produced vs. what the production C# code-base produced. Surprisingly, in most of these cases, it was discovered that the C# code-base was in error and required production patching. Fortunately, these production issues were minor and didn't impact actual tax processing. However, it did increase the confidence in the correctness of both the new F# automated testing tool and the remaining experimental F# applications.

Around the end of the summer in 2018, development work began for the next tax year, which included a major rewrite to our Partnership tax form. At this time, the team was informed that the migration of the final MeF NextGen applications was scheduled for mid-November. This was great news and meant that work to maintain both the C#/VB.NET and F# source code repositories was coming to an end. The decision was made to go forward with F# only for the next tax year.

Supporting the new Partnership tax form was a difficult task, because the business requirements were very late and changed frequently. A few years ago, the team was tasked with rewriting our largest corporate form using C# and was given about six months to do that work. Those requirements changed often too, but we got them early. This time, the work needed to be done in less than half that time – and at the same time changes to other tax forms had to be implemented. Again, F# did not disappoint. Let me be perfectly clear here: there is no way that all the required work to support tax year 2018 correctly could have been accomplished in C#, in the allotted time-frame. Let me say that again differently: if the CT-DRS MeF code-base wasn't rewritten in F#, CT-DRS would not have been able to process the new Partnership tax forms on time. Period. Now that would have made the nightly news for sure.

In mid-November, 2018, the last (and most critical) of the NextGen applications were migrated to production after business user testing and approval. Within a couple of days, we hit a small issue that was fixed in less than a day. No tax filings were harmed during this event. Given that this issue was diagnosed and resolved very quickly, I believe it actually added to the credibility of the NextGen brand. Note that as good as the automated regression test tools are, they cannot catch everything – especially at the interface between different systems, when one system holds state that the other doesn't have access to. Since then, small enhancements to the instrumentation and logging were made to ease ongoing support.

The team currently supports two legacy non-F# applications[20] that are to be obsoleted in the near future. Otherwise, the team relies exclusively on F# now for all software development.

It's now late April 2019 and MeF tax processing has gone incredibly smooth this year (i.e. zero hiccups with better performance). So much so, that I've had time to help other development teams resolve some of their issues. Of course, F# is used for that work, too.

[20]One written in VB.NET and the other written in Python.

The current CT-DRS Bureau Chief of Operations and my former boss recently sent me the following in an email after mid-April peak tax processing:

> Peter,
>
> Just wanted to say thanks for another successful filing season.
>
> It's amazing how much more efficient the process has become, and it is a direct result of your hard work.
>
> Thanks again for doing what you do and the pride you take in your job and doing what's best for our department and the taxpayers.

I owe CT-DRS management thanks for trusting my judgment over the years. In addition, I'd like to thank Don Syme, the F# team at Microsoft, and all the F# open source contributors for creating a truly pragmatic productivity enhancement tool for .NET software development.

If you're not using F# for .NET development, you are working harder and longer than you need to. Put the burden on a smart compiler and a more advanced type system to reduce the effort of creating correct software faster.