Great Migrations

BMW Financial Services .NET Adoption Case Study

Note:
This article was originally published in .NET Developer Journal under the title "**Five Pillars of Visual Basic Retirement**" in November 2006.
http://dotnet.sys-con.com/read/299072.htm - See the http://archive.org version here: Five Pillars of Visual Basic Retirement @ .NET DEVELOPER'S JOURNAL

Introduction

In the past, Visual Basic (VB) upgrades were fairly painless and inexpensive because Microsoft made new versions of VB backward compatible, but things are different this time. An upgrade to .NET brings with it a radical shift in terms of architecture, design, deployment, features, and tools. The upgrade will be even more challenging if you decide to move from the forgiving VB compiler to the rigorous C# compiler. Confronted with declining vendor and community support and major migration challenges, BMW Financial Services in Dublin, Ohio set out to define a strategy that would allow us to adopt C# in an efficient and deliberate manner. Our objectives were to minimize disruption and costs and leverage the momentum of the platform change to move our capabilities forward. This article presents our strategy and some of the experiences we are encountering along the way.

"We knew that taking our business critical systems through such a huge transformation would be challenging. We needed an approach that not only minimized cost and ensured quality but would insulate our business from disruption. We built new architecture frameworks as a part of the project, and the Promula translation tools were an essential part of our strategy. These tools, tuned by Promula to our specifications, have given us the planning flexibility we needed to incorporate the migration into our release process. To this point we have migrated about 50% of our portfolio, with few of our business users even knowing we were doing it. We are on schedule and on budget to complete the migration by March 2008."

- Jeff Haskett,

The General Manager of Applications Development

Group for BMW Financial Services of North America.

On the Road to .NET

BMW Financial Services is committed to Microsoft technologies and has used Visual Basic (VB) as the mainstay of application development for over ten years. Our applications began taking shape in 1994, even as VB itself was still growing up. In early 2000, I was hired to manage the newly established system architecture team. We found ourselves facing a wide variety of architectures and coding styles. Although we are almost purely a VB shop, we had several different application frameworks and our code employed a number of different "standards" for common tasks. We also used over one hundred different third-party COM components. Our intent was to standardize our architectures and to implement more intelligent code reuse, but such changes are expensive and building a case purely on the basis of architecture goodness is near impossible.

By 2001, Microsoft had begun to promote its new, improved development platform: Microsoft's next generation toolset that would become .NET. We decided that this new platform should become a key component of our standardization strategy. As luck would have it, in early 2002 we found ourselves at cross roads: we were about to embark on a major CRM package implementation. The package warranty did not allow us to use stored procedures - the other mainstay of our application development. We would have to use the package's APIs. By now, .NET was entering the mainstream, particularly for developing middle tier web services. It was a match made in heaven: we needed a solid platform for integrating with the CRM package, and .NET fit the bill perfectly. By the end of the year, we had used C# (See Sidebar: Choosing a.NET Language) to design, build, and deploy a service-oriented middle tier that integrated our legacy applications with the CRM solution. More importantly, we had also educated management about the impending loss of VB support and got the go ahead to build additional .NET frameworks that would form the foundation for migrating to .NET.

We planned three application frameworks to help us migrate: one for desktop, one for web, and one for batch jobs. We fondly refer to these frameworks as DesktopCAFE, WebCAFE, (CAFÉ stands for Common Application Framework for the Enterprise), and BEEF (Batch Environment Execution Framework). Each of these frameworks leverages .NET and the Microsoft Application Blocks / Enterprise library to provide architecture support for quickly assembling robust, modular applications.

Sidebar:

Choosing a .NET Language

It was mid-2002, and our first major .NET system was on the critical path of a huge CRM implementation. One of the questions blocking progress was: what language to use? We had been a one-language shop from day one, and it worked for us. We absolutely wanted to standardize on one language rather than support many. We knew all .NET languages have their strengths and weaknesses; but, they are similar in structure and use similar tools; they compile to the same intermediate language, and they use the same runtime engine. We also knew language decisions can become mired in controversy, but we could not let the language decision languish. We conducted a brief language evaluation and looked at the pros and cons of the different options (VB.NET, C#, Managed C++, Jscript, etc.) Within a few days, we settled on C#, published our findings in a brief language selection statement, and got on with the work of adopting .NET. Some of the reasons why we chose C# are listed below:

1. We felt C# was more mature in terms of sample code, tools (such as NDoc and NUnit), and published books and articles.

2. Microsoft seemed to be using C# internally, so we figured it was more mature and tested.

3. We liked the strictness of C# in terms of strong typing, error handling, and case-sensitivity

4. We did not think VB.Net was that much easier than C#.

5. We liked that C# had been submitted to ECMA as a standard

6. We liked that C# was similar to C++ and Java.

7. We felt the C# community would be large, lively, and strong in the long term.

By the end of 2003, we had published the Batch Architecture Strategy, the WebCAFE adoption strategy and had also deployed a very basic version of DesktopCAFE. Also in 2003, our service-oriented middle tier grew at frightening pace. In our haste to provide an easy-to-use service framework for the CRM project, we put very little governance around the creation of new services. This was both good and bad. On the good side: the C# middle tier provided a ready alternative to writing more VB. On the not so good side, it was the only alternative; and we soon had a library of over 100 services of questionable purpose.

VB Retirement Strategy

Our VB Retirement Strategy was a small document (26 pages) with several parts. It presented the case for VB retirement. It described the as-is and the to-be states of our processes and architectures. And, it spelled out our guiding principles of .NET adoption. Most of the strategy was dedicated to describing the process improvements and other efforts that would be needed to ensure success.
One of the most important aspects of the strategy was the set of guiding principles that would steer our decisions about .NET adoption. These are listed below:

- The effort will be a gradual, across a 3-year timeline scheduled to complete prior to the Microsoft Visual Basic retirement date in March 2008.
- We will seek to minimize impact to business projects by aligning migration efforts with scheduled projects and avoiding code freezes.
- We will have a strong first year commitment to research and analysis so we can “get smarter” about the challenges up front and enjoy more predictable/efficient migrations in year two and three.
- We will attempt to leverage automated translation tools to the extent feasible
- We will do a straight port of business logic, but allow structural changes in cases where technical incompatibilities exist or where we can move to common frameworks.
- Architecture/Logic improvements can also occur if the cost of doing migration and project work together is lower than the cost of doing them separately.

The bulk of the strategy was dedicated to describing the five processes we would need in order to ensure a successful migration. Our mission would be to mature these five

processes and use them to power the migration. These five processes are described here:

- **Rearchitecting** - this is implementing non-functional improvements to our systems. Our definition of Rearchitecting is beyond the syntax changes and API replacements inherent in the VB-to-.Net translation. Rearchitecting is concerned with taking advantage of object-oriented principles, structured exception handling, and other .NET features. It also entails restructuring how the components and layers of the application interact with one another. New frameworks are a key aspect of the rearchitecting process. Another major aspect of rearchitecting is the implementation of transitional architectures that allows us to incrementally retire legacy architectures as we move to new ones.
- **Translation** - this is the most basic work of the migration: converting working VB projects to functionally equivalent, maintainable C# projects. Our existing code is the best specification we have for our systems: it is accurate, it is detailed, and it has been tested in use. Translation allows us to leverage this asset. The Translation process deals with defining detailed VB-to-C# conversion standards, tuning conversion tools, and developing refactoring techniques.
- **Retooling** - this is updating development processes and developer skills to work with .NET. The Retooling effort is concerned with overcoming the challenges that mixed language development presents for our configuration management, build, and deployment procedures. Retooling also involves training developers and enforcing conversion standards.
- **Testing** - this is making sure the other first three processes are working in a way that limits defects. Our organization has a streamlined quality culture: “we only test what has changed.” This model does not scale very well when “almost everything is changing”. We are still coming to grips with this, and we are taking steps to ensure that quality is built into the conversion process up front rather than expecting to test it at the end.
- **Managing** - this is the work of coordinating technical teams, synergizing migration work with other projects, ensuring continuous improvements, monitoring progress, managing the budget, and removing obstacles.

These five processes interact to reduce the cost and increase quality and speed of the .NET adoption effort. For example, an investment in Translation capability will result in higher

quality C# code that is more correct and ready for Rearchitecting. An investment in Retooling will result in more repeatable translations, builds, and deployments, as well as smoother adoption of new architectures and tools. Rearchitecting will yield more complete specifications for Translation and a solid target for migrated applications. Testing will ensure that other migration processes are working properly. Managing coordinates and balances these activities to make the most efficient use of program resources. Figure 1 shows how these processes relate to each other.

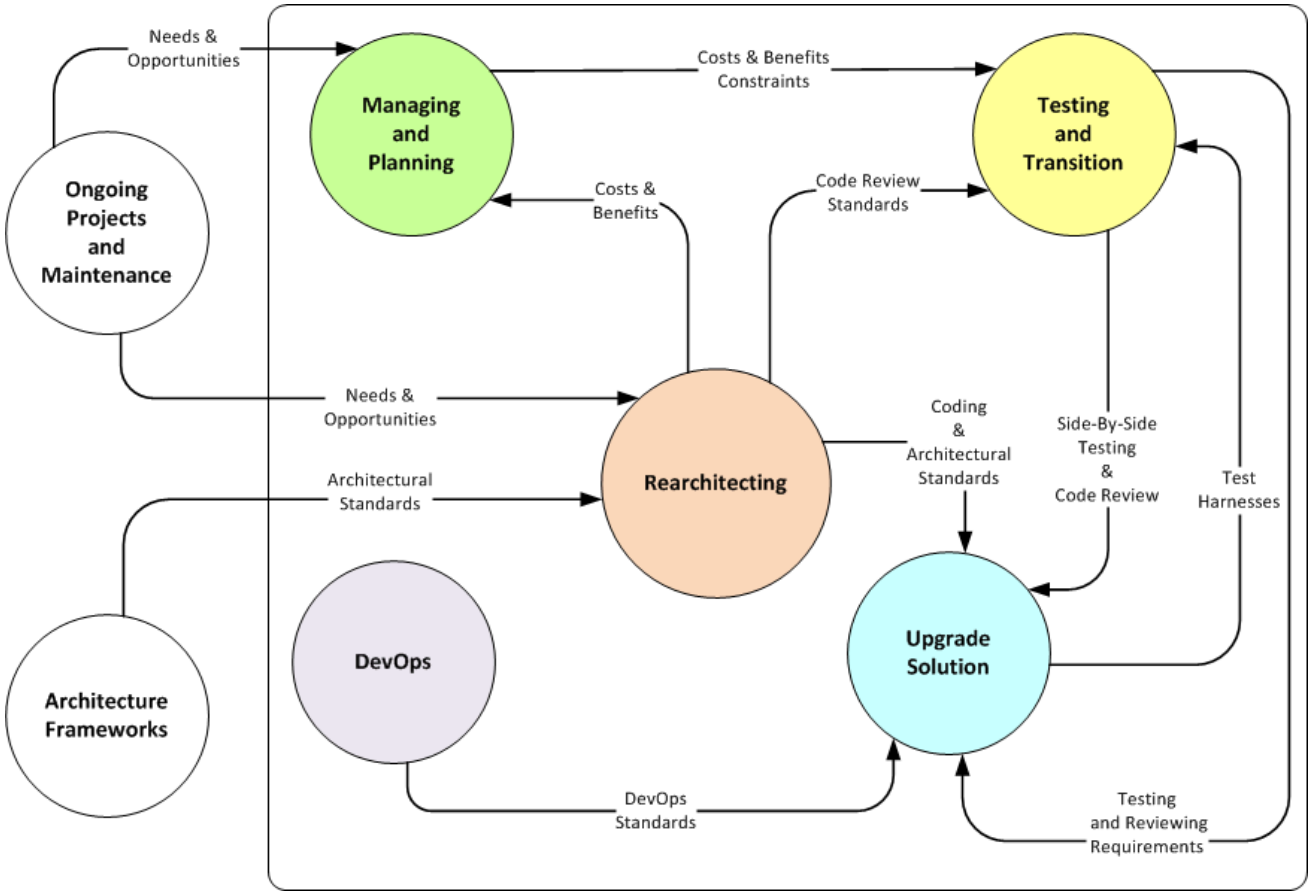

Figure 1: Relationships among the Five Pillars of Migration

Estimating the Cost of VB Retirement

In addition to agreeing on a strategy, 2004 also saw us reach the more important .NET adoption milestones:

- We estimated costs and established a program budget for the next three years.

- We got buyin from top-level management.
- We completed the implementation of WebCAFE and fleshed out the next versions of DesktopCAFE.

In mid 2004, we tackled the daunting task of setting a budget for VB retirement. At the time we were actively maintaining over a million lines of VB code in thousands of files and hundreds of application components. In addition to the sheer volume of code, estimating was difficult because so many important factors were unknown. For example:

- How much of our VB code could or should be deleted or retired over the next three years?
- How much effort could be automated and what was the cost/benefit of automated conversion?
- How much effort would be saved by reusing the new architecture frameworks?
- How would the learning curve influence costs?
- How would the new languages and tools influence development costs?
- What would the testing effort cost?

Despite the uncertainty, we still needed to produce numbers for budgeting purposes. We made an assumption that some, yet to be identified, automated translation process could do most of the work and we would only estimate the effort needed to “fit, fix and finish” each module to the point where it would be ready for user acceptance testing. We qualified each form module as simple, medium, or complex and assigned relative effort based on the complexity. We assigned hours for class and BAS modules based on the number of lines. We identified obsolete components and excluded them from the calculations. Our development teams were able to qualify all 3500 files fairly quickly. Next we added some overhead for testing, translation tuning, and management, and used a spreadsheet to compute the total. It was certainly not perfect, but it was a number that we could explain and that we were comfortable with.

Our cost model made two important assumptions: First we would use new architecture components to replace legacy architecture components, and, here’s the kicker, the creation of these new components would be funded using separate funds; after all, it did not make

sense to keep build new systems in VB, so we had to build a new architecture with .NET applications anyway. Second, we would use a translation process to do a portion of the work; tuning the process would be funded by the migration program. Given a new architecture target and a tool to do most of the grunt work, the remaining costs over the life of the project would be something we call FF&F: Fit translated code to the new architectures, Fix any residual translation issues, and Finish the job with testing and deployment work.

Figure 2 shows how the effort is divided up among manual FF&F, translation, and architecture work. Notice that we assume the tuned translation process will do a lot of the migration - translating hundreds of thousands of lines of code. Also notice that the cost of adopting the new architecture components is partially handled by the architecture rollout and partially by the manual migration effort.

Manual Work

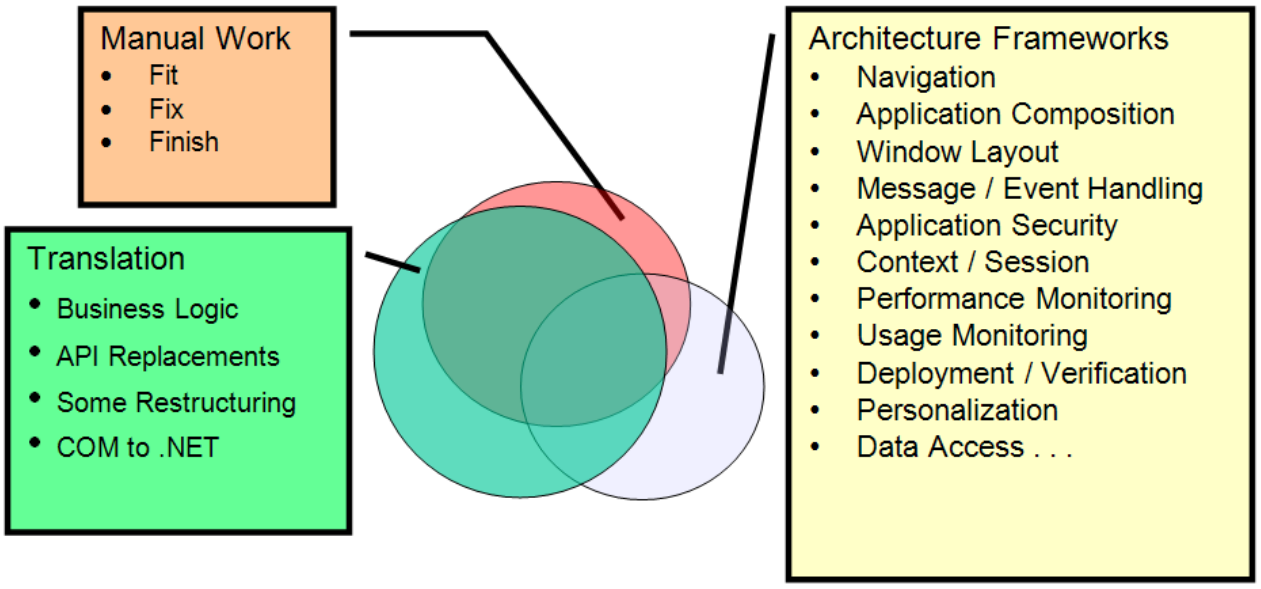

Figure 2. Three Types of Migration Work

Looking for Help

Our budget assumed that we would have an automated process to convert working VB code to working - or near working - C# code. We now needed to implement that process. In late 2004, we distributed an RFP to 10 different firms in order to validate the assumption, refine our approach, and see if someone could help. Our RFP was brief. It described “as-is” and “to-be” states at a high level. It described our IS organization including our team sizes,

our lack of system documentation, and our streamlined testing processes. It emphasized the dynamic, fast-paced demands of our bi-monthly release cycles and our need to minimize disruption to ongoing work. The first round RFP also included a proof of concept (POC) to translate a single VB project: 12 KLOC and four external references. The POC served two purposes: 1) it provided a sample of our code to the vendors, and 2) it provided a specific example basis for the vendors to demonstrate their conversion capabilities.

Almost all the vendors proposed a labor-intensive, CMM level-5 approach. Many of them planned to leverage the VB Upgrade Wizard and other off the shelf tools and a lot of manual changes to take the VB first to VB.NET and then to C#. They also proposed significant demands of our business resources to support testing. A few expected us to help them in configuring offsite, often offshore, development and test environments, including test versions of our infrastructure, applications, and data. In order to minimize code freezes, they proposed to do parallel development and merge our VB changes into their C# code before cut over. These engagements would be complex, and they would be expensive. Ultimately, consideration of these RFP responses led us to the conclusion that managing the migration internally and doing the work onsite was the best way to minimize disruption, avoid straining our light weight processes, develop our existing staff, and control cost and schedule.

One vendor proposed a particularly compelling service offering that complemented the onsite, internal approach. This vendor, Promula Development Corporation of Dublin, Ohio, specializes in automated software translation tools and services. Promula has been applying their translation technology and methodology in large-scale software conversions for over twenty years, and they were working on a VB6 to .NET migration methodology by translation. They proposed to work with us to develop a custom translation system built precisely to our specifications. Of course we were extremely skeptical, but their proof of concept results earned them a place in the second round of RFP evaluation.

The second round of evaluation began in early 2005 and was focused on the translation technology. We compared two translation service vendors and an in-house effort to leverage off-the-shelf translation tools. We put together a very difficult proof of concept. Vendors had 6 weeks to translate 16 VB projects - over 70 KLOC, and 50 references to external components. The in-house effort was a failure: using off the shelf tools left a

massive amount of work to be done by hand, and there was no way to make the off the shelf tools smarter.

On the other hand, the Promula results were very impressive: by the end of the POC, their product produced clean C# code directly from the VB6 code. Fifteen of the sixteen C# projects compiled straight from the translation. The tool also translated all 16 projects as a set, so it used native calls (rather than interop) for all the references within the translation set. The tool was even able to produce VB.NET output as an option.

2005 - The Year of Getting Smarter

2005 was declared “the year of getting smarter”. We are actively implementing the technology and process improvements needed for translation, rearchitecting, and retooling.

On the rearchitecting front, WebCAFE adoption began in February 2005 and continued opportunistically. DesktopCAFE version 2.0 went into production in August and we began a major wave of adoption beginning October. We also implemented an adapter architecture that allows legacy application modules to run seamlessly in the new framework so we can retire, rather than upgrade, the old framework.

On the translation front, we are actively defined conversion standards according to our preferences for transforming legacy coding patterns to .NET. We have published standards for source file and project structure, data access, messaging, error handling, application monitoring, and grids and Promula is configuring their translation system to meet these standards. We have many more conversion standards planned covering everything from UI components, to application security, to file-IO, to configuration. We have used the early versions of the tool to provide first cut C# code that we manually finished and deployed to production with only moderate effort.

On the retooling front, we completed design of configuration management process and tool changes needed to support repeatable standard translations, repeatable mixed-language builds, and automated code reviews. Developer training on .NET standards and frameworks is being done primarily by having developers use C# in our middle-tier services.

By the end of 2005 we had gotten much smarter about the perils and pitfalls of migration and we had become very efficient at tuning the translator. Many technical lessons and architecture standards were implemented in the translation tool (See Figure 3). In addition, we were confident that we could do migrations along with functional changes, so business and IS were able to plan how migration work would integrate with upcoming business projects.

Listing 1 is a small example of the type of code transformations we are making in adopting .NET. This specific example shows three interesting things:

- The replacement of VB On Error Goto with Structured try-catch exception handling,
- The replacement of a legacy ADO wrapper function (ExecProc) with a call to our standard ADO.NET data access component (DataMeister). Note that DataMeister is a slightly modified version of the SQLHelper application block.
- The translation requires stored proc parameters information to setup the SqlParameter array. The translation tool is able to lookup and integrate the names of stored proc parameters from metadata pulled from SQL server - a huge time saver.

Listing 1. Sample Translation produced by the Promula VB-to-C# Translation System

Original VB code

```
Public Function GetLessee(Acct As String) As String
    'Returns name of accountholder, if no last name then use company name.
    If gbErrorTrapOn Then On Error GoTo ErrorTrap
    Dim rs As ADODB.Recordset
    If ExecProc("cash", "cash_GetLessee", rs, -1, Acct, gsCountry) Then
        If Not rs.EOF Then
            If Len(Trim(rs!last_nme)) > 0 Then
                GetLessee = IIf(IsNull(rs!first_nme), "", Trim(rs!first_nme)) &
" " & _
                IIf(IsNull(rs!last_nme), "", Trim(rs!last_nme))
            Else
                GetLessee = IIf(IsNull(rs!company_nme), "",
Trim(rs!company_nme))
            End If
        End If
    Else
        GetLessee = "Unavailable"
    End If
    Exit Function
ErrorTrap:
        ShowError("GetLessee", Err.Number, Err.Description)
End Function
```

Generated C# code

```
public static string GetLessee(string Acct)
{
    string GetLessee = "";
    // Returns name of accountholder, if no last name then use company name.
    SqlParameter[] sqlParms = new SqlParameter[2];
    try
    {
        SqlDataReader rs = null;
        sqlParms[0] =
DesktopDataMeister.SetProcParameter("@account_no",ParameterDirection.Input,
false, Acct.ToString(), 10);
        sqlParms[1] =
DesktopDataMeister.SetProcParameter("@country_cde",ParameterDirection.Input,
false, Cash_Globals.gsCountry.ToString(), 3);
        rs = DesktopDataMeister.ExecuteReader(DesktopConfigurator.EDBServer,
        CommandType.StoredProcedure, "Cash.dbo.cash_GetLessee", sqlParms);
        if (rs.Read())
        {
            if ((rs["last_nme"].ToString().Trim()).Length > 0)
            {
                GetLessee = (VBNET.Information.IsDBNull(rs["first_nme"]) ? "" :
                rs["first_nme"].ToString().Trim()) + " "
                + (VBNET.Information.IsDBNull(rs["last_nme"]) ? "" :
                rs["last_nme"].ToString().Trim());
            }
            else
            {
                GetLessee = (VBNET.Information.IsDBNull(rs["company_nme"]) ? ""
:
                rs["company_nme"].ToString().Trim());
            }
        }
        else
        {
            GetLessee = "Unavailable";
        }
        return GetLessee;
    }
```

```
    catch (Exception exc)
    {
        ShowError("GetLessee", ABCFSException.MapExceptionToErrNumber(exc),
        exc.Message);
    }
}
```

2006 and Beyond - Running the Migration Factory

By early 2006 the new architecture frameworks were all firmly in production and ready for adoption by migrated legacy applications. Tuning the translator to match our architecture standards gradually leveled off as all significant issues were resolved and all enhancements were implemented. Running the translation has become a minor step in our release cycle. We translate a large pre-determined set of components for migration at the beginning of a release cycle, what we call the Cut-Over translation. The new C# code is then built and tested with the rest of the release with minimal manual intervention, and more importantly, no surprises.

To finish the program on target, we will have to upgrade 30-40 VB components with every release - that's almost 100,000 lines of code every two months! The next two years promise to bring a burst of evolution to our systems and organization.

About the Author

Mark Juras is president of Great Migrations LLC, a technology solutions provider that specializes in helping people migrate their software applications from one platform to another via translation from one programming language to another. At the time of writing this article, Mark was System Architecture Manager for BMW Financial Services of North America.