mobilize.NET

The ROI of Eradicating VB6

December 2019

Your App. New. Again.

Contents

www.mobilize.net **info@mobilize.net** **+1-425-609-8458**

Summary

Millions of lines of Visual Basic 6.0 code still run today, 19 years after the final version shipped. Much of that code started out as a small RAD application or prototype which grew (more by accretion than design) until it became large, unwieldy, and mission critical.

This paper address issues with continuing to use and support VB6 applications and examines alternatives for potentially disposing of them.

Among the many reasons VB6 is no longer a good thing to rely on are:

- A Windows update may block your app from running
- Built-in application security vulnerabilities
- 32 bit performance issues
- Potential to lose customers because you're a dinosaur
- Developers and resources are expensive and hard to find
- Can't do anything modern like take advantage of SaaS interfaces
- (i.e. can't integrate with Salesforce or other SaaS products, etc.)
- It's an obsolete technology out of support.

A brief history of Visual Basic

In the spring of 1964, as Lyndon Johnson was settling into his new-found presidency and the Beatles were stealing the hearts of American teenagers,

John Kemeny and Thomas Kurtz released the first implementation of BASIC: the Beginner's Allpurpose Symbolic Instruction Code.

Little did they know that they had created something that would come to be the most popular programming language in the world.

BASIC was simple and approachable and lightweight enough to run on the 8-bit microprocessors that powered the first wave of micro-computing in the 1980s: hobbyist boxes

like the Apple II and the TRS-80. When the IBM PC shipped, it contained a version of BASIC built right into the OS.

By the late 1980s microprocessors weren't just for hobbyists anymore and BASIC had established a firm foothold in the world of programming languages. Microsoft—which passionately wanted its newly minted Windows operating system to be successful—cobbled together the first "visual" way to create Windows programs with BASIC, not C or Assembly, as the underlying programming language.

This was Visual Basic 1.0, released in 1991, and before it was replaced by VB.NET 10 years later it had become the go to way to code for Windows for millions of people around the world.

Visual Basic 6.0[1]—the final version of "real" Visual Basic before it was replaced by VB.NET (an entirely different language sharing only the basic syntax of Visual Basic)—was wildly popular for three reasons:

1. When Visual Basic came along Microsoft Windows was just becoming widely adopted but still quite difficult to write applications for. Visual Basic drastically reduced the pain of writing traditional Windows applications which eventually led to millions of VB apps.

2. Based on the popular BASIC programming language, it allowed non-programmers to write code. Loosely typed, late-bound, and supporting questionable syntax like "On Error GoTo", Visual Basic mostly got out of the way of rapidly coding business apps. Suddenly everyone was writing Windows applications using VB.

3. VB's support for ActiveX libraries allowed a vast ecosystem of third-party components that could easily be dropped into code to perform all kinds of functions, making rapid applicationdevelopment even easier.

As we will see, many of those same reasons that made VB so popular make it increasingly a problem for IT today. This paper looks at issues around VB6 and different approaches to mitigate those issues.

COM is an obsolete technology that is famously difficult to understand.

ActiveX components are a security and compatibility nightmare.

ActiveX issues

The universe of ActiveX (i.e. COM-based components, formerly called OCXs) third-party components used so commonly by VB6 developers is a fundamental threat to the ongoing life of VB6 code. In the 90s component developers flourished.

Every copy of Visual Basic included a catalog showcasing hundreds of different vendors and their custom controls for everything from charting to calendar views to data access and visualization. Most of those vendors are merely a memory today and there is no one to support their control libraries for modern Windows versions.

---

1 In this document when we refer to "VB6" we intend this to include any and all applications written using the Visual Basic product family from the initial release of VB1 to its final version (6.0). We are NOT referring to anything written subsequently with VB.NET, which must be regarded as a completely different programming language sharing only some syntax elements with Visual Basic.

Early adopters of VB who wanted to support older and lower-cost PCs opted for 16-bit executables using 16-bit controls. Those applications are today—in many cases—trapped by their 16-bitness since for some time now no CPUs have been manufactured that can support any sort of 16-bit mode short of running a VM emulating a 16-bit processor.

Those applications can be modernized but in doing so will have to abandon those 16-bit components. More about components later. The ability of COM (and any ActiveX or OCX) to take control of any Windows service represents an on-going security vulnerability of unmeasurable risk. Modern OS designs insulate applications from other bits and pieces far more carefully than COM ever can—consider how web browsers run in sandboxes with little access to hardware or low-level services. In the days of C and C++ developers explicitly carved out chunks of memory to execute threads and lacking careful design and testing can and did bring systems to complete crashes—thus the infamous Blue Screen of Death (BSOD).

Windows today is based on .NET which manages everything under the covers—developers using .NET do not address memory directly, instead relying on the .NET Framework to allocate memory, manage instantiated objects, and collect garbage. This is referred to as "managed code." COM—like the apps written natively to the Windows API in C++—are "unmanaged code." While managed and unmanaged code can intermingle—using concepts like COM Interop or PInvoke—moving forward best practices will dictate eliminating unmanaged code entirely as yet another area of potential defects or vulnerabilities.

Short version:

COM is bad.

.NET is good.

The component model was based on COM (Microsoft's Component Object Model)—a deeply complex system for inter-process communication. Today no one would dream of writing anything that relied on COM, instead using something simple like RESTful services or even SOAP/XML web services.

COM

Before the web, before the internet was ubiquitous, Microsoft created a method to let applications work together so that, for example, a Word document could contain an Excel spreadsheet.

This was Object Linking and Embedding (OLE) which later morphed into COM (Component Object Model) and eventually COM+ and DCOM. COM also spawned the interprocess communication methods used in OCX and ActiveX technologies.

Many VB6 applications continue to rely on COM in order to function—a common scenario is a reliance on a shared external library (DLL). What's wrong with COM?

- Doesn't always work on Windows 7/8/10, so legacy apps might not work when hardware is replaced or upgraded
- It's stuck in the 32-bit world (see below)
- COM libraries have to be registered with Windows, which can be troublesome, requires Administrator privileges, and can lead to so-called "DLL hell." .NET assemblies avoid all that nonsense, only need co-location with executable binaries to function correctly.

32 and 64 bitness

Desktop CPUs have evolved from 8 bits[2] (hobby computers in the 80s) to 16 bits (IBM PC) to 32 bits (Compaq 368) and now 64 bits.

Address space limitations have typically driven this evolution with the current addressable space of 2^64 or 9,223,372,036,854,775,807 (9 quintillion) is sufficiently large that there is no need to build 128-bit processors (the next generation of computers will probably be quantum computers).

It's important to note that VB6 as a programming language only understands 16- and 32-bitness; VB6 shipped in 1998 when 32-bit computers were at their peak (having been introduced on the X86 platform a full 13 years earlier) and the first Intel 64-bit processor was still three years away.

Does 32-bitness matter? In some ways, 32-bits is alive and well even in 2017 on Windows 10.

32-bit Hardware

The last Intel 32-bit processor (Pentium 4E) was introduced in February 2004. That same year Intel introduced the Xeon 64-bit CPU. What this means is that any 32-bit laptop or desktop computers are possibly as much as 15 years old (four lifetimes in computerese).

---

2 The 4-bit Intel 4004 was arguably the first commercially-available microprocessor but was never used in a general purpose desktop computer. By the introduction of VB1..0 in 1991 virtually all desktop computers being sold were 32-bits

Those physical machines, while it's possible that they could be running a contemporary OS like Windows 7 to Windows 10[3], they can still only have 4GB of addressable memory. In reality, they will more likely have something like 512MB of memory and an 80GB hard drive. Sure, they can be upgraded to 4GB of memory and a bigger hard disk, but they will still have serious limitations based on processor speed and addressable memory limits (4 GB).

Visual Basic 6.0 cannot create a 64-bit executable; it can only create a 32-bit executable. Given that all contemporary hardware is using a 64-bit CPU, how can a 32-bit executable run? The answer is WoW64, which is an emulation layer that Microsoft provides with current 64-bit Windows versions. Running 32-bit apps on a 64-bit processor is a little like buying a 2017 Corvette and disconnecting half the spark plugs; you're just not getting anything like the full performance available.

And since it's pretending to be a 32-bt machine when your app is running on WoW64, you're still stuck with the 2^32 (4.3 billion bytes) limit, with only 2GB for data and applications (the other 2GB are dedicated to the kernel). This will create performance bottlenecks for large data sets, I/O-intensive applications, and so forth.

In short: 32-bit apps are slow.

32-bit systems, whether hardware or software, are a dead-end street..

But there's more: increasingly, 32-bit bits and pieces just won't work with modern components.

As new technologies are rolled out, most of the time they assume they are working in a 64-bit world, so 32-bit compatibility is not a requirement.

Moving to 64-bits ensures you can continue to stay current; being stuck in a 32-bit world can limit your future growth..

Visual Basic 6 vs C#

Why upgrade from VB6 to C# (or VB.NET[4])? There are many reasons.

Let's drill down on a few of the most important:

Customer perception

You know that old smelly sofa that your great uncle Fred had in his living room?

That's what a VB6 application looks like today compared to modern desktop or web applications. And your customers will recognize that your app is old and clunky and perhaps extend that negative impression to your company and value proposition as well.

Scarcity of developers

Visual Basic was hot 20 years ago; now it's considered a dead language. It's not taught in school and no one is writing books about it anymore. It's not that people can't learn it, but you have to be pretty dedicated and most younger developers will probably see a B6 job as a career-limiting move. As a point of comparison, search on Indeed.com for Visual Basic 6.0 jobs and you'll get 20-30 results. Search for C# developer and you'll get closer to 25K results.

Visual Basic also continues to plummet in popularity on developer indexes such as TIOBE. Basically, you need elderly developers who learned it back in the day and are still not quite ready to move into the assisted living facility. But those kinds of people are harder and harder to find. And they cost more.

No OOP

VB6 is BASIC, which is a procedural language graced with subroutines and classes. It's not an object-oriented programming language (OOP) and you can't really make it behave like one.

---

4 VB.NET (not to be confused with Visual Basic) is a .NET programming language functionally identical to C# but with Visual Basic syntax. "Functionally identical" is technically an overstatement, but the two implementations are so close as to make them interchangeable—each is strongly typed, object-oriented, and focused on supporting the full .NET Framework. C# was developed for programmers comfortable with the C language family (C++, Java, etc) while VB.NET was created for those coming from Visual Basic. Advocates for each continue to engage in a spirited debate over the relative merits of their favorite. For the purposes of this white paper, we will stay focused on C#, but everything we say here can equally be applied to VB.NET.

To apply modern design patterns on a VB6 app is a real struggle: one where you are constantly trying to work around the limitations of the language.

Many language shortcomings

A little research will show that many "serious" programmers have a litany of beefs about the original Visual Basic language. Some of the elements criticized frequently include On Error GoTo (as opposed to Try/Catch in C languages); variant data type; lack of unsigned number types; confusion between subroutines and functions; ability to create objects without declarations; and overuse of Global variables. All of these make it easy to do RAD (rapid application development) as they streamline things that are harder in C# but which can lead to code that is either buggier (see next bit) or harder to maintain.

Easy to write bad code

The very nature of VB simultaneously appealed to non-professional programmers (engineers, scientists, business people, Excel gurus) while repelling trained developers, who looked down at VB6 as only slightly better than a toy designed to create unmaintainable and buggy code. Among its many faults, VB6 makes it difficult or impossible to follow modern software engineering practices. For example, VB6 invites you to create business logic as part of an event handler on a discrete control, violating the concept of separation of concerns. Variables are created by simply using them—no need to explicitly declare them or their scope. As a result, far too many variables are global in typical VB6 code, creating a petri dish for runtime bugs. And the use of SUB means you can easily create spaghetti code by having subroutines call other subroutines which in turn call other subroutines and so on.

VB almost encourages developers to write code that breaks good software engineering practices.

Many controls no longer available

As mentioned above, a key element in the enormous popularity of VB6 was the vast ecosystem of external components that could be added to the toolbox in the IDE and dropped onto VB forms for "instant" functionality. The appeal of these components can't be underestimated: for a few hundred dollars or less you could drop in a chunk of pre-written, tested and supported code to perform a function you needed. The alternative was to try to write the functionality yourself, or do without. Unfortunately, many of those vendors are no longer

in business, so those controls have no counterpart for modern applications and are no longer supported.

Third-party controls stop working

Even though Microsoft continues to support the VB6 runtime on Windows 10, component vendors are under no obligation to update their controls. Because of this, you may find that your app no longer functions correctly with no warning and no fix.

Windows may shut you down without warning

On August 13, 2019, IT professionals all over the world noticed their VB-based applications had stopped working.

The monthly security patch released that day disabled remote procedure call (RPC) in the VB runtime, killing apps written in VB6, as well as apps using VBA and VB script. A vulnerability in RPC that Microsoft had discovered led to the decision to favor security over application continuity. Although they addressed the problem in the subsequent weeks, there's little doubt that Microsoft—faced with a similar decision—will put security ahead of keeping legacy applications running.

Microsoft has no obligation to ensure the continuity of anything out of support, which VB6 and related old technologies are now. It's not that Microsoft has any desire to damage old applications, but the reality is that vulnerabilities that are simply part of the platform are going to continue to be found and not all patches will allow for continued functionality.

Getting off VB6

You don't have to live with VB6 forever because there are perfectly valid alternatives to remove the risks and costs of having VB6.

Here are three popular approaches to remediation with pros and cons:

Choice 1: Replace with off-the-shelf software

If you are an ISV and you sell your VB6 app, then obviously your app is custom to your needs. But if it's an internal (i.e. line of business) application, perhaps you don't need a custom app to solve the problem.

Platforms such as Microsoft Office or Salesforce can be extensively tailored to perform business functions that in the past required writing software.

Replace with off-the-shelf-software	
Pros	**Cons**
High quality engineered in	Lose competitive advantage of custom features
No in-house development resources needed	Perpetual dependency on an external vendor for mission-critical functionality
Rapid deployment	Per-seat costs will mount up over time
No large up-front investment	Requires retraining fo staff to new paradigm
	At the mercy of vendor for support and bug-fixing

Choice 2: Rewrite from scratch

When developers look at old VB apps, probably the first thought that comes into their minds is "rewrite."

And certainly this is a perfectly valid outcome for many legacy applications, but not always. As a company that has seen more VB6 legacy applications—large and small, ISV and line of business—we can attest that there are many examples where a rewrite is a terrible idea. Let's drill down. Consider that many VB6 apps were built around the concept of "forms over data" with some simple business rules enacted in the code. Those apps have little unique value that

couldn't easily be duplicated. Their primary function is CRUD plus reporting; that is create, read, update, and delete records in a database and present views of that data. Simple business rules like data validation are built into the code behind form objects. Small apps that follow this model with dreadful code (buggy, tangled, hard to follow) are best rewritten from scratch or replaced with some package that can do the same work. What kind of applications should be rewritten? Probably far fewer than most people would expect.

The reality is that writing software—any kind of software—is incredibly risky.

Numerous studies have shown failure rates of up to 70 percent for new software development, with typical problems including:

- **Time**: it takes much longer than planned
- **Budget**: it costs more than expected
- **Scope**: requirements and specs keep changing and increasing during the project
- **Defects**: more found and not found than expected
- **User acceptance**: upon delivery, users reject the app for issues such as performance or completeness
- **Outright failure**: after multiple setbacks, the project is abandoned and the costs written off.
- **Investment**: any prior development is lost—usually assets like custom business logic code that has been honed, refined, and debugged over many years.

Rewrite from scratch	
Pros	**Cons**
More fun to write new code than work with old code	Cost: writing code is expensive
Get to start with a clean sheet of paper	Investment: lose any prior investment
Can use any language or platform desired	Risk: see above
	Time: rewriting a large app can take years to complete

Choice 3: Migrate with automated tools

Migration using automated tools can be the perfect solution for moving many different kinds of VB6 apps forward. Using automation reduces the time and cost of creating a modern version of a VB6 app by as much as 90 percent, allowing you to get your new app into users' or customers' hands quickly and affordably.

Best-in-class migration tools use machine-assisted learning and AI algorithms to analyze code patterns, not just syntax, creating new code that recreates the intention of the original application. In doing so, it preserves business logic, rules, and data structures without introducing new errors.

Which apps benefit most from migration? Not all apps are good candidates for automated migration. Those that benefit the most have the following characteristics:

- Large and complex, with many forms
- Incorporating complex business logic that is critical or represents competitive advantage
- Has many users
- Is frequently updated and deployed to user community.

Here are examples of some VB6 apps that were excellent candidates for automated migration:

Example 1: Distributed call center app

National Systems sells call center software that Pizza Hut uses for home-based phone agents to take orders for pickup or delivery. Orders are routed randomly to available agents then directed to the most appropriate restaurant location. The original app was written in VB6 and, since the user base was not only widely distributed but also constantly changing and non-technical, deployments of new versions and support were expensive and problematic due to factors like varying versions of Windows and "DLL hell." The app required "five 9's" of reliability and needed to be replaced in such a way that the user base would require zero retraining and zero disruption. National Systems selected Mobilize.Net's WebMAP solution to migrate their VB6 app to a modern web architecture using KendoUI to duplicate the VB6 forms.

Example 2: Retail store app

Over the years this VB6 app was developed and sold to over 5000 retail outlets dealing in used (consigned) merchandise. The original VB app connected to various hardware devices like a cash drawer, Verifone credit card terminal, bar-code printer, and scanner. The company was successful but saw encroachment on their market position by a SaaS offering that could run in any browser. Given that, the company elected to make the jump from local instances with private databases running VB6 and Windows to a full,modern web architecture using WebMAP.

They achieved some major improvements as a result of the migration, including:

- Switching from license model to subscription model provides steady, predictable recurring income
- Hosting the application in Azure provides effortless resource elasticity to handle surges in demand
- Single multi-tenant database instance means the company can collect aggregated meta data for additional business value
- Customers can use any device to run their businesses, including tablets with card swipers
- Mobilize was able to write custom drivers to connect the web app with existing store hardware
- Business was able to respond to competitive pressure in months, not years, ensuring theirsurvival and ongoing success.

Example 3: Complex line-of-business app

CFM Materials recycles serviceable parts from end-of-life jet aircraft engines, mostly from Boeing 737 models.

Starting with a small VB6 app to keep track of inventory, over two decades the app grew in scope and complexity until it became the ERP system that ran the entire business.

Recognizing that it was getting increasingly difficult to secure technical talent to maintain the application, the company elected to use the Visual Basic Upgrade Companion with Mobilize's assistance to migrate it to VB.NET and Windows Forms, running on the .NET Framework.

Once migrated, Mobilize migration engineers worked directly with the technical team at CFM Materials to bring them up to speed on the new code base so they will be able to take ownership and maintain it in the future.

Migration is not a rewrite

A migration doesn't do what a rewrite accomplishes, but that's not necessarily a bad thing.

A rewrite addresses the same business problem as the legacy app, but invariably gets additional requirements from the business owners.

A migration, on the other hand, perfectly preserves the existing functionality while moving the code base forward to a modern language and platform.

A rewrite is "let's start over."

A migration is "let's lift and shift."

Migrations preserve the accuracy of business logic and rules without introducing new defects.

In many ways migrations are superior to rewrites:

- A migration is that rare case in software development: a perfect specification, since the existing legacy application is the specification for the migrated app.
- A migration can be risk free by contracting with a vendor for a fixed-price, guaranteed delivery date project.
- Quality in a migrated application is simple to prove: it merely needs to pass a test suite based on the original application's functionality. When the new app functions identically to the legacy app, it's done.
- A migrated app reduces to zero or near-zero the number of logical defects in the code.
- A migrated app can be built in such a way as to have virtually no user impact, insofar as it will have the same look and feel as the legacy app.
- A migrated app can be deployed in a fraction of the time necessary to re-create the app via rewriting.

- Once migrated, the new code base can be refactored and enhanced to bring new capabilities to the application.

The Business Case for Migration

Examining a "typical" application lets us understand better the business casefor migrating a VB6 app compared to rewriting it.

The costs of rewriting

Let's assume a hypothetical "medium" sized application of around 200 KLOC[5] of VB6 code, including 150 VB forms and 40 separate components (DLLs or OCXs). This application uses a SQL database back end and is used by 3000 people.

What would it cost to rewrite this from scratch to a modern desktop application using C# (or VB.NET) and Windows Forms?

As of this writing, a knowledgeable journeyman-level C# programmer in the US will cost around $150k per year fully burdened (salary, office, benefits, taxes, overhead, and so on). While this can vary widely depending on location (California's Bay Area compared to Orlando, for example), for the purposes of analysis we'll just use this number. For your purposes you can use a more representative number, and enter it into our calculator here.

Over the years various academics and analysts have tried to calculate the productivity of software developers based on lines of code written[6]. Counting total finished lines of code divided by total developer time—which includes meetings, creating requirements and specifications, debugging, and surfing the Internet—ranges from 10 to 40 LOC/developer-day, with 10 being "average" corporate developers and 40 being "rock star" commercial ISV developers. Let's assume 20 lines of code per developer per day. If our developers work all year except for 2 weeks (10 work days) then each developer will average 5000 finished lines of code per year, at a cost of $30 per LOC ($150,000 / 5000). Your 200KLOC application will take 40 developer-years to complete.

Total cost?

$6,000,000.

5 "KLOC" = 1000 lines of code

6 Measuring developer productivity is an enormously controversial subject that you can read about on your own

Why so expensive?

For one thing, development—new software development—is much more than coding. Whenever a development team starts with a clean sheet of paper, even to rewrite an existing application, new requirements come out of the woodwork. And even the best programmers write bugs—a lot of bugs. Studies show that for each KLOC of new code, somewhere between 10 and 50 new bugs (defects)will be created, each of which must be identified (which in turn requires writing test cases), investigated, fixed, verified, and tracked. More studies show that all software includes defects not discovered prior to release, and these defects can be the most dangerous of all.

No new features

But what if you merely rewrote the existing app in a new language, sticking strictly to the existing set of features and postponing any new ones until after the rewrite was complete?

First of all, no one would ever do this. It's simply too tempting to take care of existing requests during the project. And that leads to feature creep.

But if they did—if they somehow managed to hold off all the demand for a change here and a change there—how much would it save?

Half. Half is a good number, so the cost per LOC could drop from $30 to $15.

Going offshore

The lure of lower-cost but high-quality software development from far-off locales like India or Ukraine has appealed to many companies who were either short of resources or simply looking to stretch their budgets.

Taking our last scenario (a pure rewrite with no new features) offshore could save another 50 percent[7]: driving the cost per LOC from $15 to $7.50 and the total project cost from $6M to $1.5M.

Net net on rewriting

Worst case: $6M. Best case: $1.5M.

7 Unfortunately, careful studies of offshore software projects simply don't bear out a 50 percent savings. The best studies show that, that, due to increased coordination and project management costs, total savings typically run in the range of 15 percent.

But wait: there's more.

Risk

The R word. Few endeavors by contemporary humans have a more abysmal success record than software development. In fact, "success" in software development is rarer than failure—"failure" being defined as a project that either fails to meet the specified budget, schedule, or requirements, or is abandoned altogether.

Software development is arguably the riskiest endeavor an organization can engage in.

As many as 70% of software development projects fall short of their goals or fail outright. This in spite of years of research into best engineering practices such as waterfall, Agile, team development, and more.

A business knows how much a new truck will cost. They know how much a new server will cost. But a new application? Statistically this is impossible to accurately predict.

The case for automated migration

As thousands of organizations have learned, the alternative to risky, expensive, time-consuming software rewrites is migration using AI-assisted automation:

- Automation reduces the time and effort to migrate legacy applications to new languages and platforms
- Migration tools—"software robots" if you will—preserve the investment in business logic, rules, and processes encoded in existing applications
- Automation prevents feature creep and allows for predictable budgets and schedules
- Automation doesn't introduce new defects like writing new code does
- Automation can move a legacy app to a new platform quickly and affordably, allowing for rapid re-deployment to users while developers begin the process of adding new functionality.

The economics of migration

Just as robotics can reduce the cost of assembling automobiles or mobile phones to a fraction of the cost of labor, automated migration tools can reduce the cost of modernizing legacy code to a fraction of the cost of using people.

How much can it save?

In most cases, between 80 and 95 percent compared to rewriting. Further, at least half of the total cost is verifying that the migrated application functions identically to the original legacy application.

Our hypothetical 200KLOC VB6 application could be migrated to a C#/.NET version for between $300K to $1.2M without the risk of rewriting.

And using near-shore (Costa Rica) resources—who happen to be the global leaders in this kind of project—companies can get a guaranteed delivery date at a fixed price. Removing all project risk.

Real world examples

A global 1000 enterprise migrated from VB6 to a modern web application—utilizing the Telerik KendoUI Javascript framework to perfectly emulate the look and feel of their VB application, thus no need to retrain thousands of users—for 20 percent of the cost of a rewrite (their original budget).

A vertical-market ISV was able to migrate their VB6 app to a modern web version (switching their revenue model from license to subscription) in less than one year, compared with the internal schedule of three years to achieve the same result with captive developers.

Typical migration process and issues

A typical migration project can be divided into four major stages:

4. Planning and analysis
5. Code modification and QA
6. Hosting, deployment, and operations
7. On-going support and maintenance

Stage 1: Planning the migration

Organizations with significant portfolios of in-house applications can benefit from an initial Portfolio Analysis, where the following outcomes are achieved:

- An inventory of all applications, including meta data such as programming language, ownership, platform, technical quality, security vulnerabilities, maintenance history, and business value
- Bucketing applications into potential outcomes including retire, replace with off the shelf software, improve, rewrite, or migrate.
- Relative priorities for each bucket category
- Time-frames, estimates, and expected technical issues for each..

The portfolio analysis is a scoping project at the department, division, or enterprise level that can provide senior management with invaluable 360-degree view of their in-house application portfolio.

This can help set priorities, staffing, and budgets for longer-term planning.

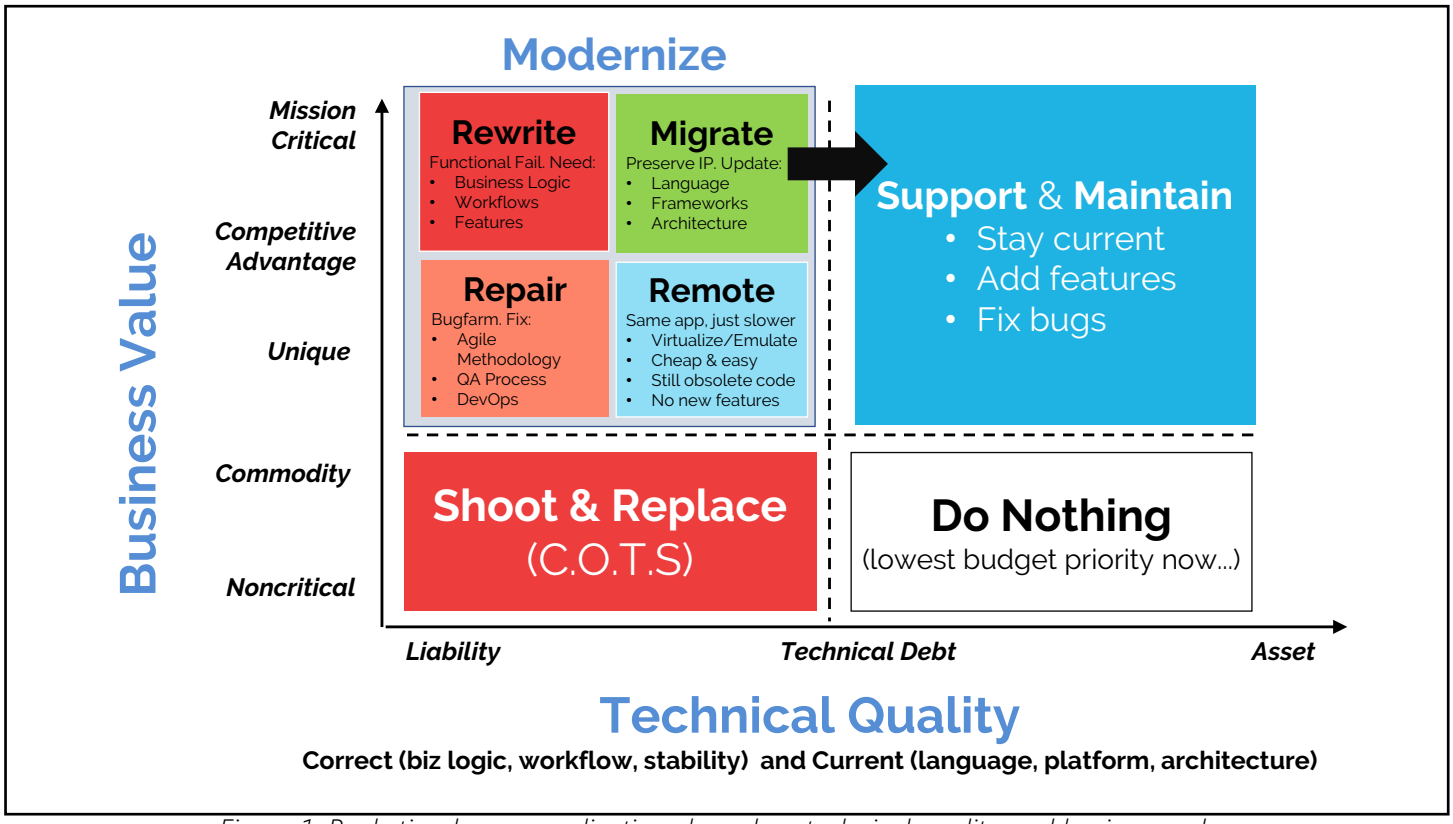

Figure 1: Bucketing legacy applications based on technical quality and business value

At a more granular level, a Migration Blueprint is a deep dive into a single application's code to deliver more technical analysis and planning:

- Migration engineers come on-site to analyze application code & development resources
- Identifies all migration issues and proposes appropriate remediation strategies
- Presents detailed schedule, milestones, and resource requirements for migration project
- Delivers proof-of-concept on actual code for quality and approach evaluation by your technical team
- Identifies unique application-specific adaptations for reduced migration effort

- Optional fixed-price, fixed-date proposal for delivery of migrated code.

Stage 2: Code modification and QA

Following the analysis and planning phase, the next step is to actually begin modifying code. Using highly-automated tooling (VBUC, WebMAP, etc), a semantic analysis of all the code in the application is performed creating an abstract symbol table (AST) which represents all the interactions and patterns in the application source code. This AST can further be used to identify additional application-specific adaptations that can increase the extent of the automation in the migration process.

A code-generator produces new code from the AST in the target language and platform.

Because of the nature of the tooling, business logic is preserved as is and (when desired) the look and feel of the original application. Generated code is completely native with no dependencies on binary runtimes. Some C# helper classes are provided (in source code form) to ease the transition from the old platform to the new one for your internal development team.

During the code modification phase, some predictable issues must be resolved on virtually all projects:

- VB6 overuse of Variant data types can create runtime errors. The migration tools will attempt to translate Var into discrete types based on analyzing the usage. Types which are bound at runtime may need further investigation.
- Some visual elements may not migrate perfectly to the new platform. Particularly when migrating from desktop to web, the available graphic libraries are reduced. Depending on what was intended, the HTML <canvas> can possibly be substituted.
- All references in the project must be available when the migrated project is compiled and built. Components must be local or available through COM Interop; otherwise some new code must be written to replace the prior functionality.
- Some considerations may be necessary to account for assumptions about type sizes; for example, in VB an integer is 16 bits and always signed; in C# an integer can be signed or unsigned and can be 64 bits. Assuming bit offsets based on type sizes can break migrated code.
- Data access methods can be replaced by .NET native classes like ADO.NET; however frequently as part of a migration from VB6 the underlying database technology should be upgraded. For example, migrating a Microsoft Access database to SQL Server.

- A number of issues are due to methods and properties that are still available but have changed their behavior from VB6 to .NET. The VBUC will flag these with discrete comments but they all need to be reviewed and verified that no defect results from the difference

Continuous migration

During a migration project the application owner may want or need to continue to make changes to the legacy application and publish those updates to the user community. Due to the nature of Mobilize.Net's tooling continuous migration is possible; changes to the original app are diffed compared to the migrated code and new migrated pieces can be integrated into the migration code repository. Specific details and best practices' recommendations on this level of integration are available at www.mobilize.net/continuousmigration

Stage 3: Stay current

Now that the legacy code has been replaced with modern code and a modern framework like .NET or HTML with Javascript, the development goals should be to avoid accumulating new technical debt.

Areas of post-migration focus include:

- Architectural improvements: coming from a procedural language like Visual Basic, the code can now be refactored to be truly object-oriented. Creation of classes, strong typing, and separation of concerns are all good next steps.

- Security improvements: applications can be vectors for malware into an organization. Analyzing your new app code for security flaws such as SQL injection or buffer overflow attacks can keep your organization and data safe.

- Performance enhancements: whether you moved from VB (desktop) to .NET (also desktop) or all the way to HTML (web architecture) performance can be an issue. Profiling your app to understand bottlenecks and potential performance improvements can enhance the overall user experience.

- DevOps: especially in the case where you are now deploying your application as a web app, integrating hosting and operations into your overall development process (aka DevOps) can have big payoffs.

- Stability: once a test harness and automated test suite has been prepared as part of the initial migration process, that suite should be re-run after any software releases or

dependency changes. This provides a complete regression analysis to ensure no new problems arise as a result of on-going work.

- Features: with the new platform that your app runs on, there are likely to be capabilities that were not available for the legacy code. Adding features to take advantage of new OS or framework classes can improve user satisfaction and productivity.

Costa Rica vs India: costs and issues

Many organizations lack sufficient internal resources to handle migration projects while still pursuing existing higher-priority development projects. In those cases, companies frequently turn to companies like Mobilize or India-based systems integrators (SIs) for project fulfillment.

The use of off-shore or near-shore resources for software development by US and European companies is a well-established trend that has been extensively discussed and documented. The basic value equation of moving away from on-shore development is exchanging reduced hourly labor rates for increased degrees of complexity and oversight.

- Mobilize.Net has one of the largest software engineering centers in Costa Rica. Companies that choose to get their migration project performed by Mobilize.Net get many advantages over projects performed in India or Eastern Europe, including:
- Mobilize.Net is an "English first" company—all employees are fluent in English and conduct all customer communications in that language.
- Costa Rica is on US Central Standard Time year-round (no daylight savings time). Calls, Skype, and email can happen during US business hours, not in the middle of the night.
- Our engineering center uses modern technologies like Visual Studio 2019, Team Foundation Server, Github, and Agile development practices. A technical representative will be happy to demonstrate the detailed project portal that all customers have access to for their project.
- Costa Rica has arguably the most stable economy and government in Latin America. Constitutionally they have no military, instead focusing those resources on creating one of the most literate and educated populations in the region.
- Mobilize.Net's technical team is led by managers whose educational backgrounds include Princeton, Oxford, and the University of Florida among others.

Conclusion

If you've read this far you know there are real consequences to continuing to rely on VB6 applications. And you know the future continuity of anything built with VB6 —and the associated implications of that—is unpredictable and risky to count on.

Why not start today exploring your path to freedom from VB6?

Mobilize.Net the world's authority on VB6 and—how to get off it—is here to help.

Headquarters
10500 NE 8th St. Ste. 1775
Bellevue, WA 98004
+1.425.609.8458

Engineering Center - Torre La Sabana
300 Metros Oeste del Edificio del ICE Americas
San José, Costa Rica 10108
+506.2519.1000