Breaking Backwards Compatibility

Shôn Ellerton, January 16, 2020

Ah! The frustration when backwards compatibility gets broken when new updates to frameworks are released.

I've done quite a bit of coding in my career, particularly in the disciplines of telecommunications and wireless networking. Nothing terribly exotic. Nothing in the realm of being exotic or esoteric. My mainstay programming languages have been VB, VBA and SQL followed by a bit of C#, PHP, Python, Javascript, FORTRAN and even a little bit of COBOL and Pascal in my younger days. These don't include all the 'funny' little languages which few have heard of, an example being Deluge script created by Zoho, a SaaS CRM provider, which I used on a pilot project for digitising a paper-based data collection system using online tablets.

Technically, SQL isn't really a proper programming language, but I've done a massive amount of it, so I'll include it above. On VBA, many hardcore programmers tend to dismiss this as being a toy programming language; however, I've used a lot of it to create very cheap, effective and quite sophisticated thin-client SQL Server database applications using Microsoft Access.

One of the biggest unknowns with any platform, in my opinion, is its ability to survive in the years to come. Will it break in 10 years' time? Will it survive the next year or two? Will if break once the next framework of .NET is released? Will Java start deprecating more classes which are required for the platform to run? And so on.

The obvious question asked is how long the platform is required for. Is it to service a short-lived project or become part of a suite of software to futureproof the platform to support more projects arriving at the door in the foreseeable future? In my experience, most platforms are designed for with the long-term in mind, a strategy which usually costs many times more than designing for a single project. The gamble, here, of course, is that this strategy only pays off if future projects can make use of the platform; however, many times this does not happen.

Platforms which are resilient to time are often those that are written on frameworks supporting software which is *backward compatible*. A lot of the software I wrote many years ago using a combination of Microsoft Access and VBA served its purpose well, most of which was designed on the basis that it would be superseded by more robust web-based client/server models. I re-used much of the code I wrote for the development of more recent applications but became stuck a couple of years ago when needing to access some of the code in an older version of Access.

Microsoft committed the act of breaking backwards compatibility, a well-known issue for many developers. In this case, Microsoft deprecated support for Access Data Projects (.adp files) which I used in the past. They would not even run on a modern version of Office and therefore, I had to run an older version of Office to be able to access the code and, painstakingly convert, to the newer .accdb format. On the same token, others working in web design were suffering the same issues with technologies that rolled out updates for their frameworks; each time, having the whole platform re-tested to ensure that it works after the updates are committed.

These days, I have more involvement with mainframe technologies; having to ensure that the flow of data remains stable being exported from older operational data IBM mainframes to newer SQL Server-based data warehouses. The process uses third-party software to translate the COBOL journals containing IDMS data exported daily from the mainframe and then convert them to RDBMS SQL commands to maintain a replica of the mainframe database in SQL Server. Quite clever software!

Only yesterday did I have a very interesting conversation with someone at work who works directly on the mainframe software including the writing and administration of Java-based code that drives the middleware for the client

interface for the COBOL-based operational system. Prior to the conversation, he threw his hands up in the air and said that the Integer class has now been deprecated on Java's new release meaning a bit of a re-write will entail to keep the code operational. After a few rude words being exchanged on how everything went downhill with Java since Oracle's takeover, it was then discussed, along with one of his other colleagues, that some of the code will need to be re-written and, in future, that primitives should be used as much as possible instead of 'wrapper' classes and functions.

On hearing this, I asked the question how critical systems like those servicing banking systems, airline ticketing systems, ATMs, and many others survive in a world of ever-changing framework updates; some of which, deprecate functions and classes on what is, seemingly, the whim of the software vendor. I further asked the question if this could be one of the defining reasons that the massive ongoing work to modernise old COBOL-based mainframe systems to more modern technologies, sometimes on the Cloud, has not progressed very well in general or has proved to be incredibly expensive. A great article, *It's mainframes all the way down*, by Ruth Grace Wong discusses in more detail with some examples on how older systems based on mainframes are incredibly stable and cites examples of horrendous costs of attempting to convert them to more modern frameworks. It's worth a read.

Having worked in the mainframe industry for much of his life, he replied that the very ethos of mainframes is to ensure *complete* backwards compatibility. Whereas, outfits like Microsoft and Oracle (who own Java), have no qualms with releasing a new update for a framework requiring developers to update code relying on functions and classes that have been deprecated. For example, if one was to run an old piece of COBOL on a mainframe with the newest version installed, it should run as expected. Knowing a little bit about COBOL myself armed with the knowledge that remarkably few updates (less than 12) have been released since its birth in 1959, that the code (if written properly) is stable well into the future, and performance is often blindingly fast given the sheer core power of mainframe technology, it is, indeed a wonder. Although, saying that, I am aware that there have been a few issues with some of the COBOL releases, notably with COBOL-85, but, generally, it has been very stable, certainly more so than our current .NET environments.

Why would any organisation contemplate converting to newer systems?

After all, the enormity of breaking a critical operational system by not testing thoroughly every time a new framework update is too great.

What about security by obscurity? Hosting operational systems on commonplace technology is far riskier with respect to security because ‘too many people know about it’. As simple as that sounds, it’s true!

There are, of course, advantages to converting; for example, the ease to find personnel to maintain the system (COBOL programmers and mainframe gurus are getting increasingly hard to find) and the high cost of proprietary mainframes from the manufacturers like IBM or Unisys.

However, here’s the big question setting aside the potential difficulties of getting people with the right knowledge to look after them in the future.

Imagine having an ancient 40-year-old mainframe COBOL-based system which is still working and you wanted to make a similar system but decided to use a standard modern Cloud-based solution, after which being built, you leave for 20 years only installing any updates. Which system is liable to still be running?

It’s likely that future updates will break the latter without proper testing; however, mainframes, themselves require extensive (and expensive) maintenance as well and resources to look after them are scarce.

The answer is not always very clear.

Breaking backwards compatibility is very frustrating for those maintaining and developing systems. It is a credit to those who developed some of these older mainframe-based systems that they can still run flawlessly for 30 to 40 years or more!