Issue 018: Obsolescence

Graham Lee

March 2nd, 2020

Welcome to the eighteenth issue of *De Programmatica Ipsum*, dedicated to the subject of *Obsolescence*. In this edition:

- Graham pleads for longer living computer systems[1], and an end to the prevailing tech consumerism.
- Adrian writes about the upcoming, unavoidable, greatest revolution in computing[2]. No, really.
- In the Library section[3], Adrian explores Jean Sammet[4]'s 1969 opus *Programming Languages, History and Fundamentals*.

Enjoy this issue! Please subscribe to our free newsletter[5] to stay updated about new releases, or contribute[6] if you would like to support our work.

Cover photo by Konstantin Dyadyun[7] on Unsplash[8].

[1] https://deprogrammaticaipsum.com/the-twenty-year-computer/
[2] https://deprogrammaticaipsum.com/a-farewell-to-the-von-neumann-architecture/
[3] https://deprogrammaticaipsum.com/category/library/
[4] https://deprogrammaticaipsum.com/jean-sammet/
[5] https://deprogrammaticaipsum.com/newsletter/
[6] https://deprogrammaticaipsum.com/contribute/
[7] https://unsplash.com/@kostyadyadyun?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[8] https://unsplash.com/s/photos/obsolete?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

A Farewell To The Von Neumann Architecture

Adrian Kosmaczewski

March 2nd, 2020

Unbeknownst to most if not all full stack developers and Scrum masters out there, the computer industry has been fighting a raging war against the Von Neumann architecture for the past 70 years.

John Von Neumann was probably one of the greatest scientists of all time. His contributions to science and technology range from computers, weather prediction, and economics, to fluid dynamics, quantum logic, and set theory. He sat in a desk in the university of Princeton, next to Gödel, Ulam, and Einstein.

One of his papers is the famous "First Draft of a Report on the EDVAC[1]," an unfinished paper in which he describes the mechanisms behind one of the first computers. This is the origin of the term "Von Neumann Architecture," although it is now known that other scientists had come up with similar architectures before Von Neumann. But that is beside the point. The important thing to know is that whichever computer you are using to read this text, it is based in that eponymous architecture.

The Von Neumann architecture is the reason why most software developers argue that learning a second programming language requires substantially less investment than learning the first. All languages respond to the same underlying logic, because they ultimately all talk to the same kind of computers, regardless of their obvious syntactic differences. John Backus said it clearly[2] in his ACM Turing Award lecture in 1977:

> The differences between Fortran and Algol 68, although considerable, are less significant than the fact that both are based on the programming style of the

[1] https://en.wikipedia.org/wiki/First_Draft_of_a_Report_on_the_EDVAC
[2] https://dl.acm.org/doi/10.1145/1283920.1283933

von Neumann computer.

This is not the place for a lengthy discussion of the major characteristics of this architecture; suffice to say that they include the use of a bus to connect the CPU with memory, and the use of that same memory space for both data and instructions. The important bits rely in the ways that this architecture, however useful, has also reached its limits once and again during decades.

Patches

As all things created by humans, the Von Neumann architecture is imperfect. Its most well known problem is the "Von Neumann Bottleneck." To solve this issue, computer scientists have brought up the concept of caches, which begat cache invalidation as one of the hardest things to do[3] in computer science. Modern CPUs these days have various levels of caches, of various capacity, allowing CPUs to avoid expensive roundtrips to fetch information from memory.

The fact that programs are stored in the same medium as data is not without its problems. While it can make virtual machines possible, it can also enable buffer overruns, particularly if your chosen programming language does not perform the required (and nowadays taken from granted) checks. Viruses, another side effect of this architectural choice, span a whole industry of operating system makers who must place protections between "kernel" and "user spaces" so that harm can be contained.

Speaking about compiler checks, they have reached such level of sophistication that they are able to perform both garbage collection and buffer overrun checks during compilation.

At some point between the 80s and the 90s, many scientists and industry pundits sold the RISC architecture as another solution for the bottleneck; let us pipeline instructions issued from a different instruction set; a "reduced" one, instead of a "complex" one. One of the great things of RISC architectures is a reduced "instructions per watt" ratio, which makes it perfect for mobile devices; the ARM architecture is an example of this factor at play.

In the real world, however, no CPU is actually "100% RISC" or "100% CISC;" most are what Gordon Bell called "code museums" in his 1991 book *High Tech Ventures*. Code museums are CPUs built to "to hold programs created on earlier machines and to serve their present customers." (There's an interesting discussion about this in page 130 of the February 1990 edition of Dr Dobb's Journal, by Hal Hardenbergh.)

Another technique aimed to accelerate the execution of code are branch predictors, a concept developed by IBM in the 1950s. This consists in the CPU trying to "guess" which branch of the code is going to be executed next. This sounds like science fiction, and to a certain extent it is, but unfortunately it was at the root of the Spectre[4] security vulnerability, which affected virtually all CPUs.

Then there are the laws of physics. Herb Sutter's seminal "The Free Lunch Is Over[5]" article marks the end of the almost 40 years old reign of Moore's Law, thanks to the fact that our CPUs shrank down to the realm of atoms and molecules. This is why the advertised CPU speed of your computer did not increase after 2005, instead hovering around 3 GHz.

[3] https://martinfowler.com/bliki/TwoHardThings.html
[4] https://en.wikipedia.org/wiki/Spectre_(security_vulnerability)
[5] http://www.gotw.ca/publications/concurrency-ddj.htm

The solution found by the industry was to use multi-core CPUs, but it turned out that a program built for a single CPU would not necessarily go faster with many CPUs. Gene Amdahl correctly explained in 1967[6] that the theoretical execution speedup was limited by the internal structure of the code.

Somebody remembered that functional languages force developers to work with unmutable data structures, and that those are perfect choices for parallel programs. This brought back functional programming languages to the forefront, right at a time when Philip Wadler was complaining that nobody used them[7]. As a result nowadays all languages, including object-oriented ones, have lambdas: C++, PHP, C#, Java, pick yours. Operating systems started bundling runtime libraries[8] making it easier to execute code in parallel.

Ironically enough, two notorious side effects (no pun intended) of this renewed insterest in functional languages were: Electron apps and npm packages. After all, JavaScript was described by Crockford as Lisp in C's clothing[9] and the rest is history. Of course, since performance is hard (let us not forget that JavaScript has no integer types) somebody came up with WebAssembly[10] and somehow, the cycle is done.

Or is it?

Future

The future is not another patch around the venerable Von Neumann architecture. Arguably, the computer you are using to read this article is already obsolete, and IBM and Google are competing to see who is going to obsolete it first. It all revolves around Quantum Computing.

Quantum entanglement and Bell's Theorems show that there will be a new architecture for quantum computers. There are a few[11] proposals[12] already, and the meager knowledge of the author of these lines in quantum mechanics represents a major obstacle to be able to read any of those papers.

Yet, the press starts to warm up to the idea of quantum computers solving[13] the halting problem, with "God-like powers[14]" apparently. One can feel the excitement and the hype.

What we need, instead of hype[15], is a new generation of Hilberts, Russells, Gödels, Turings and Von Neumanns, to bring us to the next level, and let the 21st century of computing actually start. One day we will look back at the Von Neumann architecture and find it as peculiar as Babbage's Difference Engine.

But since Von Neumann developed a form of quantum logic, we might as well still be calling his name in the years to come.

---

[6] https://en.wikipedia.org/wiki/Amdahl%27s_law

[7] https://dl.acm.org/doi/10.1145/286385.286387

[8] https://en.wikipedia.org/wiki/Grand_Central_Dispatch

[9] https://www.crockford.com/javascript/javascript.html

[10] https://webassembly.org/

[11] https://arxiv.org/pdf/1702.02583.pdf

[12] https://ce-publications.et.tudelft.nl/publications/1548_a_heterogeneous_quantum_computer_architecture.pdf

[13] https://arxiv.org/pdf/2001.04383.pdf

[14] https://www.vice.com/en_us/article/xgqg9a/mathematicians-are-studying-planet-sized-quantum-computers-with-god-like-powers

[15] https://deprogrammaticaipsum.com/issue/issue-001-hype/

Cover photo by John Towner[16] on Unsplash[17].

[16]https://unsplash.com/@heytowner?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

[17]https://unsplash.com/s/photos/architecture?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

The Twenty-Year Computer

Graham Lee

March 2nd, 2020

Computing needs to become significantly more long-term in its outlook, if we are to play our part in the coming move from a growth economy to a planetary survival civilisation. We need to move from a situation in which we expect people to buy (or lease) a new handheld computer every 1-2 years, even if their existing one works fine.

Why? The amount of pollution[1] and toxic mineral extraction involved in producing, using, and disposing of these computers (what, you thought that recycling electronic equipment[2] was actually a thing?) is vastly amplified by the demand for the new iPixel Galaxy Razr 14 Pro every year. Tim Apple has a meeting where fuck everything, we are doing five cameras[3], and then marketing comes up with reasons why your shitty three-camera phone makes you a social pariah who probably deserves to get green bubbles in your friends' messaging conversations.

Speaking of green, the vendors greenwash their new devices, explaining how their manufacturing processes reclaim more materials, produce less waste, and do not use chemicals that have already been banned in parts of their market anyway (like BFRs), or chemicals that you would never find in the component under consideration (like Mercury in LEDs). But make no mistake: recycling aluminium to make a new enclosure still takes more energy than continuing to use an existing enclosure, and that recycled aluminium enclosure is still packed into a plastic wrapper, placed into a plastic insert that is then glued into the box, for maximum viral unboxing video pleasure. The new computer may sip less electricity, but offset that against the energy required to *build* the thing and the picture is a lot less attractive.

[1] https://www.theguardian.com/environment/green-living-blog/2010/jun/09/carbon-footprint-mobile-phone

[2] https://www.vox.com/2017/11/8/16621512/where-does-my-smartphone-iphone-8-x-go-recycling-afterlife-toxic-waste-environment

[3] https://www.theonion.com/fuck-everything-were-doing-five-blades-1819584036

This needs to change, not just technologically, but socially. We need to produce computers that we can still use decades from now, and we need to congratulate not the yuppie with their brand new device, but the hipster with their ancient computer. They are the one who has not required a new computer to be manufactured.

What would a twenty-year computer look like? Probably a lot like the computer you are using right now. Look back 20 years. We already had both IPv4 and IPv6 network protocols. We had Windows NT, UNIX, and UNIX-like systems. We had 64-bit CPUs. We had 32-bit colour depth, CD-quality sound, and full-motion video. We had Wi-Fi networking.

Processors from the time had approximately the same (advertised) speeds as many CPUs available today, though used a lot more power to achieve the same computation rate (often because they ran at the advertised full speed at all time). Storage has peaked *and declined* in the intervening two decades, as cloud computing makes it possible to store much user data on centralised sytems more efficiently. And yes, we had that two decades ago, too[4].

We would probably want to diverge from today's computers in some ways, almost by definition as current machines are designed to be discarded and replaced within a small time. Believe it or not, a great example of the choices that go into a two-decade computer is a platform that only existed for under a decade. The following describes research undertaken with Steven Baker[5].

The Amiga, a computer produced by ex-Atari employees at a Silicon Valley startup and launched by Commodore Business Machines in 1985, is almost famously a non-future-proof system. Its impressive multimedia capabilities were implemented in custom chips that proved hard to update while maintaining backwards compatibility. The final model, 1993's CD32, featured the same "Paula" 8-bit sound chip as the original 1985 A1000. By this time, gamers came to expect higher sound quality, which games developers supplied by playing CD audio tracks during gameplay.

Nonetheless, there is much that the Amiga can teach us. There is still a vibrant, if small, community, using their Amigas today and developing new software. These are not twenty-year computers, these are *twenty-seven to thirty-five year* computers. How do they do it?

Compatible software upgrades. Three (at least!) different organisations supply updated versions of Amiga software that is compatible with existing applications and existing hardware. Much of the benefit of a computer is in the capabilities of its software, and you can upgrade the software without buying new hardware... as long as your vendor lets you.

With the Amiga platform, AROS[6], MorphOS[7] and Hyperion[8] offer post-Amiga environments with varying levels of source and binary compatibility. This extends to the firmware, with both AROS and Hyperion producing updated Kickstart ROMs.

Modular hardware upgrades. Need a new storage medium in your computer? Not just a bigger whatever-you-have, but a different thing? Throw your computer away, and buy a new one! Want a newer generation of CPU? Buy a new computer!

These are not answers that make sense in the Amiga land. Retrocomputing communities in general have put a lot of work into adapting modern hardware onto old connectors, so SD

[4] https://en.wikipedia.org/wiki/Sun_Ray
[5] https://twitter.com/srbaker
[6] https://aros.sourceforge.io
[7] https://morphos-team.net/
[8] https://amigaos.net/

card readers that can be used from SCSI and IDE busses, for example, are common. A more interesting development in the Amiga community is the invention of "vampire" accelerators like the Apollo[9] hardware, which mounts onto the pins on the CPU package. It draws power from the CPU's power lines, interfaces with the existing hardware using the data and address busses, but otherwise runs a different computer in the place where the Amiga's processor sits.

Or you can keep your chips and upgrade your motherboad[10], getting more connectors and modular extension capabilities on daughterboards.

Celebration of doing more with less. Go to an Amiga user group or watch one online[11] and you will see people talking about what they can do with their old Amigas, and how they get the most out of their ancient computers. The demo scene[12] also celebrates squeezing great music and visuals out of limited hardware.

Yes, there is new kit, like the X5000[13], but in these communities having the latest device is no more (or, in fairness, less) of a virtue signal than in continuing to use your 1985 Amiga 1000.

Co-opting external resources. Got a PC, or Mac? You can run your Amiga stuff on it in a fully-licensed way. If it is a G4 or G5 PPC Mac you can even run your post-Amiga OS natively. And go the other way, too: run your PC and Mac applications on an Amiga, with emulation. This may seem like a bit of a trivial point, of course one computer can emulate another. It is a question not of possibility, but of degree: it has been claimed that the FPGA reimplementations of the Amiga chipset represent the fastest classic Macintosh available.

Lack of predatory capital. All of the things described above are possible because there is nobody making it *impossible*. Since the collapse of Commodore Business Machines and their sale to Escom (which itself subsequently collapsed), there has been a complicated legal dance involving the ownership and licensing of the trademarks, software, and source code related to Commodore and Amiga.

None of which stops anyone who already owns a computer from using it for any purpose. With no signing checks in the ROM, you are free to use any operating software. There is no "app receipt validation" to stop you using packages that were not blessed by the mothership, either. No upgrade treadmill that says old computers cannot have new software. No concentration of seller power in a single app store that stops developers making packages for older computers.

Think about the computers you are using today, and whether you can still use them in twenty years. What would you need to change to make that happen? How do you produce that change?

And consider leaving a tip[14], so that when you are still using your computer in 2040 you can still read the latest *De Programmatica Ipsum* on it □

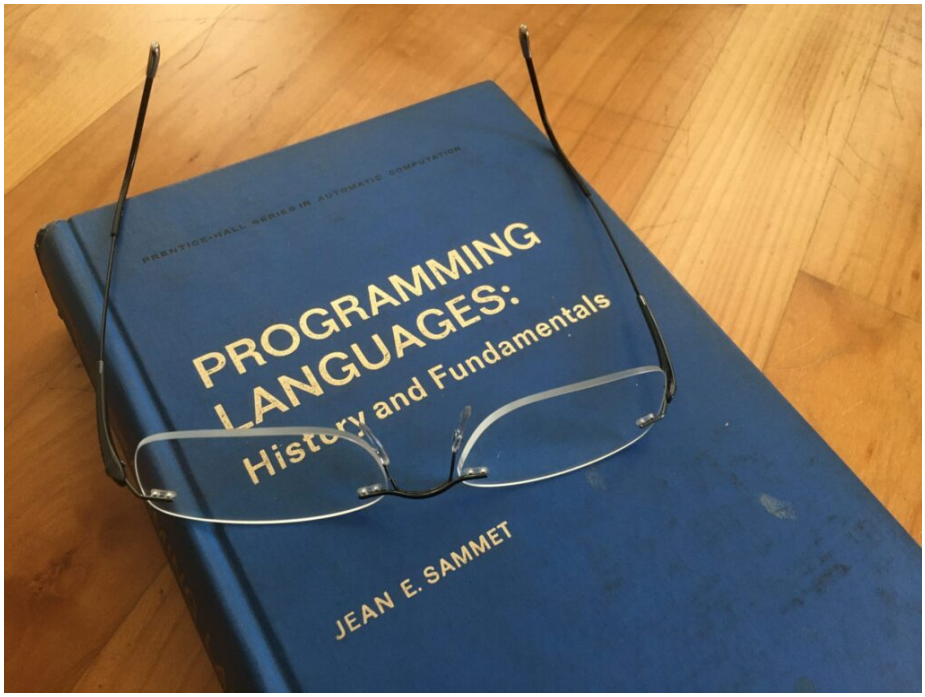
Cover photo by Alex Motoc[15] on Unsplash[16].

[9] https://apollo-accelerators.com/
[10] http://www.amigaclub.be/projects/amiga1200plus
[11] https://www.youtube.com/channel/UCYt9E2d_GCrPzquW-5MZwmQ
[12] https://www.youtube.com/watch?v=iD9xk3SDSYc
[13] http://a-eon.com/?page=x5000
[14] https://deprogrammaticaipsum.com/contribute/
[15] https://unsplash.com/@alexmotoc?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[16] https://unsplash.com/photos/P43VRz8fLWs

Jean Sammet

Adrian Kosmaczewski

March 2nd, 2020



From October 24th to 29th, 1927, twenty-nine scientists gathered in Brussels for the fifth Solvay Conference. Among the attendees, of which seventeen got a Nobel Prize before or after attending, were Erwin Schrödinger, Wolfgang Pauli, Werner Heisenberg, Paul Dirac, Louis de Broglie, Max Born, Niels Bohr, Max Planck, Marie Curie, Hendrik Lorentz, and Albert Einstein. One would think there might not have been such an assembly of brilliant thinkers since the Platonic Academy.

However pompous it might sound, in the field of programming we have our own "Solvay Conference." The Fourth ACM SIGPLAN History of Programming Languages Conference[1] (HOPL-IV) will take place from June 14th to 16th, 2020, at the Royal Geographic Society of London. The previous ones were held in 1978, 1993 and 2007; that is, not very often. The August 1993 edition of Dr. Dobb's Journal included an article by K.N. King about HOPL-II, which includes a guess of the timing of HOPL-III with surprising accuracy:

> Will there be a HOPL-III? Probably. When will it be held? Maybe 5 years from now, maybe 15.

The (jaw-dropping) speaker lineup in HOPL-I included Grace Hopper, John Backus, Peter Naur, John McCarthy, Kristen Nygaard, and Thomas Kurtz. In the second one, Alan Kay, Barbara Liskov, Dennis Ritchie, Guy Steele, Bjarne Stroustrup, and Niklaus Wirth. And for the third edition, Joe Armstrong, David Ungar, William Cook, Roberto Ierusalimschy, and

[1] https://hopl4.sigplan.org/

Simon Peyton Jones. Among the rumored participants to HOPL-IV, appear Don Syme[2], creator of F#, and Cleve Moler[3], creator of MATLAB.

There will be, however, a notorious absent in HOPL-IV. The mastermind and instigator behind the HOPL conferences, and chair of the previous three editions, one of the creators of the COBOL programming language, author of the first symbolic algebra system for computers (FORMAC), and author of the first book about the history of programming languages, Jean Sammet, passed away in 2017. This will be the first HOPL without her.

Among those many achievements, she wrote "Programming Languages: History and Fundamentals," published by Prentice-Hall in 1969. The description pamphlet of the book (available in Archive.org[4] at the time of this writing) boasts the incredible number of "120 programming languages" as one of the key selling points of this masterpiece in research and breadth.

Of all the programming languages described in the book, only a few might resonate in the minds of programmers in 2020: ALGOL, BASIC, COBOL, FORTRAN, LISP, PL/I, and SIMULA. Yes, they appear all in uppercase, neither because computers did not handle lowercase letters back then, nor because people were shouting[5] across mainframe rooms. Henceforth these lines will feature programming language names only in uppercase, just for the sake of nostalgia.

Ms Sammet's book is a milestone representing the end of the first era of programming, at a pivotal moment. Many things were about to happen in computing in 1969. The Xerox Palo Alto Research Center was going to open its doors in July 1970. The first commercially available microprocessor, the Intel 4004, and the first version of UNIX, would not be available until November 1971. The world's first home video game console, the Magnavox Odyssey, would be released in 1972. We would have to wait until 1973 for the first C programming language compiler. BASIC was just a teaching language mostly used in Dartmouth College. Microsoft and Apple would start operations in 1975 and 1976, respectively.

There are two experiences in our industry bringing us close to time travel; one is playing with virtual machines running older software. The other is, without any doubt, reading this book. We learn about those early efforts in terms that feel outdated by today's standards. SIMULA, arguably the first object-oriented programming language, is described as an extension to ALGOL built around a "collection of programs called *processes* conceptually operating in parallel." LISP is explained in detail, as the "only higher level language (...) in which the internal representation of the program is defined to be exactly the same as that of the data," and whose "most lasting contribution" is the term and technique of "Garbage Collection."

The evaluation of new programming languages nowadays floats around questions that were simply nonexistent half a century ago. Case in point: the book contains few, if any, information about type systems, let alone a mention about concepts such as type inference[6] or IDEs; Turbo Pascal was more than 15 years away. Instead, Ms Sammet describes "Control Languages for On-Line and Operating Systems" instead, describing IBM's Job Control Language (JCL) for the System/360, arguably one of the ancestors of current shell scripting languages.

[2] https://fsharp.org/history/hopl-draft-1.pdf

[3] https://blogs.mathworks.com/cleve/2018/03/21/matlab-history-modern-matlab-part-1/

[4] https://archive.org/details/TNM_Programming_Languages_history_and_fundamental_20171115_0058/mode/2up

[5] http://blog.josephwilk.net/rhetorical-programming/why-are-you-shouting-programmer.html

[6] https://deprogrammaticaipsum.com/the-truce-of-type-inference/

We also find languages used to provide graphical output, such as GRAF (GRaphic Additions to FORTRAN,) an ancestor of the DOT[7] language. PL/I receives a fairly large treatment: on one side, it was one of the most commercially significant programming languages of the era. And maybe too, because, well, Ms Sammet was an IBM employee, after all.

As an anecdote, Jean Sammet was an accomplished mathematician and created of FORMAC[8], one of the first symbolic algebraic manipulation systems, direct ancestor of Maple, GNU Octave, Mathematica, R, and Matlab. In the pages of her book she described FORMAC in detail, comparing it with other similar languages such as COLASL, MATHMATIC, and... MATHLAB and UNICODE. Paraphrasing Obi-Wan Kenobi, these last two ones are not the languages you are looking for. Phil Karlton was right in pointing out that naming things was hard[9].

And in case you are left wondering about the value of learning about such "old" technology, here is the economic argument. It turns out that a lot of the logic managing our world runs in mainframe computers. Porting and/or adapting those applications to the "cloud native" world is big money, as shown by companies by Raincode[10] ("PL/I in Kubernetes!") or NetCOBOL[11] ("COBOL in .NET!"). And, if all else fails, know that the salaries of experts in these languages are skyrocketing these days. Just sayin'.

Cover photo by the author.

[7] https://en.wikipedia.org/wiki/DOT_(graph_description_language)
[8] https://dl.acm.org/doi/10.1145/155360.155372
[9] https://www.martinfowler.com/bliki/TwoHardThings.html
[10] https://www.raincode.com/
[11] https://www.gtsoftware.com/products/netcobol/netcobol-for-net/