Issue 032: Modernism

Graham Lee

May 3rd, 2021

Welcome to the thirty-second issue of *De Programmatica Ipsum*, dedicated to the subject of *Modernism*. In this edition:

- Adrian bets on Rust to become the *modern* programming language[1] of the 2020s.
- Graham argues that convergence is a fad[2] in the long term.
- In the Library section[3], Adrian highlights the work of Eric Sink[4] and his book "The Business of Software".

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

Cover photo by Zhifei Zhou[7] on Unsplash[8].

[1] https://deprogrammaticaipsum.com/the-great-rewriting-in-rust/
[2] https://deprogrammaticaipsum.com/plus-ca-change/
[3] https://deprogrammaticaipsum.com/category/library/
[4] https://deprogrammaticaipsum.com/eric-sink/
[5] https://deprogrammaticaipsum.com/newsletter/
[6] https://deprogrammaticaipsum.com/contribute/
[7] https://unsplash.com/@phoebezzf?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[8] https://unsplash.com/s/photos/modernism?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

The Great Rewriting In Rust

Adrian Kosmaczewski

May 3rd, 2021

The book "Writing Secure Code, 2nd Edition"[1] written by David LeBlanc and Michael Howard, published by Microsoft Press in 2002, was once required reading at Microsoft, following Bill Gates' "Trustworthy Computing"[2] memo. The fifth chapter of said book is titled "Public Enemy #1: The Buffer Overrun[3]" and it starts with a very interesting historical perspective on the problem, referring to the Morris Worm[4] in 1986 as precedent, and even finding occurrences as far back as the 1960s.

Know Thy Enemy

The C programming language, often referred to as "portable assembly", provides no guards whatsoever against this enemy. Should you mistakenly (or purposedly) allocate `N` bytes of memory, but then write `N + k` (where `k` is an `unsigned int` strictly bigger than zero), you might as well be rewriting the very software you are running, for good or, as it is often the case, for bad.

Each generation of computer scientists has brought new ideas to the table to solve this problem, which arguably has cost the industry more than Sir Tony Hoare's Billion Dollar Mistake[5]. At the end of the 90s, the emergence of managed execution environments seemed to have provided a final blow to buffer overruns.

[1] https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223
[2] https://www.wired.com/2002/01/bill-gates-trustworthy-computing/
[3] https://en.wikipedia.org/wiki/Buffer_overflow
[4] https://en.wikipedia.org/wiki/Morris_worm
[5] https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/

Indeed, at the beginning of chapter 18 of the aforementioned book, the author (which one?) begins with an anecdote:

> While creating slides for two secure software papers at the November 2001 Microsoft Professional Developer's Conference, a friend told me that I would soon by (sic) out of a job because once managed code and the .NET Framework shipped, all security issues would go away.. This made me convert the SQL injection demonstration code from C++ to C# to make the point that he was wrong.

Ah, software developers.

Neither Java nor C# brought us fully secure software, even though millions of apps and billions of lines of code were rewritten, once and again, in those *modern* programming languages. Rewriting software is a widely derided activity: Joel[6] Spolsky[7], Steve Blank[8], and Peter Seibel[9] all explained its foolishness. It remains, however, a very popular sport in our industry.

Modern Languages

Looking backwards, each decade has had its chosen *modern* language, the one where the world has been rewritten into, once and again.

- 1960s: Fortran (because IBM!)
- 1970s: BASIC (because Byte Magazine!)
- 1980s: Pascal (because structured programming!)
- 1990s: C++ (because object orientation!)
- 2000s: Java (because World Wide Web!)
- 2010s: JavaScript (because... reasons!)

Let us play the game of predictions. What is the true *modern* programming language of the 2020s, that is, the one in which the world will be rewritten into?

It is not C#[10] or Java[11], who have earned the degree of "boring" even though they have solved the buffer overrun issue with runtime checks. Java is trying to make a comeback[12] these days, and even though for a short while there was an operating system written[13] in Java, people have become too wary of Oracle's legal team by now.

It is not an application language like F#[14], Dart[15], Swift[16], or Kotlin[17] either, which seem to be more interested in replacing the `null` keyword with optionals, thereby solving Sir Hoare's mistake.

[6] https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/
[7] https://www.joelonsoftware.com/2000/11/20/netscape-goes-bonkers/
[8] https://steveblank.com/2011/01/25/startup-suicide-%E2%80%93-rewriting-the-code/
[9] http://codersatwork.com/
[10] https://docs.microsoft.com/en-us/dotnet/csharp/
[11] https://www.java.com/en/
[12] https://deprogrammaticaipsum.com/write-anywhere-run-once/
[13] https://en.wikipedia.org/wiki/JavaOS
[14] https://fsharp.org/
[15] https://dart.dev/
[16] https://swift.org/
[17] https://kotlinlang.org/

Is it then Go[18]? Apparently not, as it seems to be currently somewhat restricted to the creation of either Cloud[19] Native tools, or self-contained cross-platform command-line utilities.

Les Nouvelles Années Folles

After careful analysis, the language of the 2020s might as well be (and maybe already is) Rust[20], a language that achieves the rare feat of satisfying both high- and low-level programmers alike.

Rust shows many, if not all, signs of *modernity*:

It is open source, hosted on GitHub[21], and we all know that open always wins[22].

It features generics[23], and they are at the base of optionals[24] used to solve Sir Hoare's conundrums.

It does not feature inheritance, instead relying on composition[25].

Likewise, it does not rely on exceptions for error handling, proposing a handy `Result` generic type instead.

It does not have a garbage collector, instead controlling reference lifecycles and ownership during compilation.

It includes popular functional[26] programming constructs, like lambdas, map/filter/reduce, and even better, with a "lazy" behaviour inspired from Haskell[27].

It has a free book[28] and an online playground[29] to learn the language without having to install anything in your local machine.

Even though it uses type inference[30] to make code look like scripts, it has such a strong type system it can fight buffer overruns at compile time.

It can be installed in the terminal by piping a script[31] downloaded with `curl` from the trusty interwebz.

It includes a ready-to-use library of algorithms and abstractions, appealing to both system and app developers as well, in a "batteries included" kind of approach. And if you do not find what you need, just use its own package manager[32] called `cargo` and access to a large[33] array of contributed code; for example, an ORM[34], a GUI[35], or an image processing[36] library.

[18] https://golang.org/
[19] https://deprogrammaticaipsum.com/somebody-elses-computer-as-a-service/
[20] https://www.rust-lang.org/
[21] https://github.com/rust-lang
[22] https://deprogrammaticaipsum.com/open-always-wins/
[23] https://doc.rust-lang.org/rust-by-example/generics.html
[24] https://doc.rust-lang.org/std/option/
[25] https://riptutorial.com/rust/example/22917/inheritance-with-traits
[26] https://mmstick.gitbooks.io/rust-programming-phoronix-reader-how-to/content/chapter02.html
[27] https://www.haskell.org/
[28] https://doc.rust-lang.org/stable/book/
[29] https://play.rust-lang.org/
[30] https://deprogrammaticaipsum.com/the-truce-of-type-inference/
[31] https://www.rust-lang.org/tools/install
[32] https://doc.rust-lang.org/cargo/
[33] https://crates.io/
[34] https://diesel.rs/
[35] https://azul.rs/
[36] https://github.com/image-rs/image

It features built-in unit testing[37].

Its variables are immutable by default.

It has macros[38], an intelligent evolution of C macros mixed with C++ template metaprogramming.

Primitive arrays include the length as part of their type, and can be easily initialized in the same line.

Its compiler generates blazingly-fast code, and it can do cross-compilation. It can even generate standalone statically-linked binaries, ready for use in Docker containers. Heck, it can even generate WebAssembly[39] out-of-the-box. With a bit of luck, it might generate climate change-friendly[40] binaries.

It has its own annual developer conference, aptly named RustConf[41], a weekly newsletter[42], a dedicated Awesome Rust[43] page in GitHub, and lots of questions and answers on Stack Overflow[44].

It uses curly brackets, and anyway, the `rustfmt`[45] tool removes the risk of conflicts around code formatting style preferences.

It does not feature a `goto` keyword, respecting Dijkstra's[46] commandment.

In short, Rust ticks many checkboxes in the *modern* category.

Ecosystem

Let us take a closer look at what the industry is doing with it.

Mozilla (the original creators of Rust) have been busy rewriting[47] parts of their flagship browser in Rust.

Linus Torvalds is convinced[48] Rust will take over the Linux Kernel. In the meantime, one can alias `cat` with `bat`[49], `ls` with `exa`[50], and run them inside alacritty[51], all written in Rust. And coreutils[52] in Rust is coming soon, and somebody is even rewriting LaTeX[53] in it, too.

Microsoft, almost 20 years after the "Trustworthy Computing" demo, says[54] memory safety issues (still) are 70% of all security bugs, and therefore argues[55] that Rust is the current best

[37] https://doc.rust-lang.org/rust-by-example/testing/unit_testing.html
[38] https://doc.rust-lang.org/book/ch19-06-macros.html
[39] https://www.rust-lang.org/what/wasm
[40] https://deprogrammaticaipsum.com/choosing-a-programming-language/
[41] https://rustconf.com/
[42] https://this-week-in-rust.org/
[43] https://github.com/awesome-rust-com/awesome-rust
[44] https://stackoverflow.com/questions/tagged/rust
[45] https://github.com/rust-lang/rustfmt
[46] https://en.wikipedia.org/wiki/Goto#Criticism
[47] https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/
[48] https://www.theregister.com/2020/06/30/hard_to_find_linux_maintainers_says_torvalds/
[49] https://github.com/sharkdp/bat
[50] https://the.exa.website/
[51] https://github.com/alacritty/alacritty
[52] https://github.com/uutils/coreutils
[53] https://tectonic-typesetting.github.io/en-US/
[54] https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/
[55] https://thenewstack.io/microsoft-rust-is-the-industrys-best-chance-at-safe-systems-programming/

chance for safe systems programming. Such is their interest in Rust that they are a member[56] of the recently[57] created Rust Foundation[58], and they promote[59] it as a solid alternative for apps running on Kubernetes.

Google is pushing Rust into Android[60] (for example in the Bluetooth[61] stack) but also in the Linux Kernel[62], and into Fuchsia[63], their new secure operating system.

Josh Triplett[64] from Intel says[65] that Rust is the "future of systems programming".

Amazon built[66] its new "serverless" AWS Firecracker[67] system using Rust.

Dropbox is rewriting[68] its sync engine with Rust.

Apple is hiring[69] Rust engineers.

JetBrains has a Rust plugin[70] for its IDEs (well, so does[71] Visual Studio Code, too).

IBM is teaching[72] Rust in their developer website.

Stack Overflow, after seeing Rust top the language charts[73] for 5 years in a row, published[74] a getting started guide for Rust.

Discord is switching[75] from Go to Rust.

Figma runs[76] a Rust-powered backend in production.

The Cloud Native world is steadily favoring Linkerd[77] (written in Rust) over Istio (written in Go) as the service mesh of choice.

"Rewrite it in Rust"[78] (RIIR for short) is a running meme[79] on the web, including its own Twitter account[80] and a dedicated GitHub project[81] gathering anecdotical experiences.

---

[56] https://www.computerweekly.com/blog/Open-Source-Insider/Microsoft-puts-pedal-to-metal-joins-Rust-Foundation

[57] https://www.techrepublic.com/index.php/recent/index.php/article/the-rust-programming-language-now-has-its-own-independent-foundation/

[58] https://foundation.rust-lang.org/

[59] https://msrc-blog.microsoft.com/2020/04/29/the-safety-boat-kubernetes-and-rust/

[60] https://security.googleblog.com/2021/04/rust-in-android-platform.html

[61] https://android.googlesource.com/platform/system/bt/+/master/gd/rust/

[62] https://security.googleblog.com/2021/04/rust-in-linux-kernel.html

[63] https://fuchsia.dev/fuchsia-src/development/languages/rust

[64] https://www.theregister.com/2020/07/13/rust_code_in_linux_kernel/

[65] https://hub.packtpub.com/rust-is-the-future-of-systems-programming-c-is-the-new-assembly-intel-principal-engineer-josh-triplett/

[66] https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/

[67] https://firecracker-microvm.github.io/

[68] https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine

[69] https://news.ycombinator.com/item?id=24442134

[70] https://www.jetbrains.com/rust/

[71] https://marketplace.visualstudio.com/items?itemName=rust-lang.rust

[72] https://developer.ibm.com/devpractices/software-development/articles/os-know-rust/

[73] https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/

[74] https://stackoverflow.blog/2021/03/15/getting-started-with-rust/

[75] https://blog.discord.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f

[76] https://www.figma.com/blog/rust-in-production-at-figma/

[77] https://linkerd.io/

[78] https://unhandledexpression.com/rust/2017/07/10/why-you-should-actually-rewrite-it-in-rust.html

[79] https://transitiontech.ca/random/RIIR

[80] https://twitter.com/rustevangelism

[81] https://github.com/ansuz/RIIR/issues/14

Andrew Binstock mentioned[82] Rust in a Dr. Dobb's magazine article back in 2014, shortly before it shut down.

Eric Sink[83] is busy looking for ways for Rust code to be interoperable[84] with .NET.

And reaching the apotheosis of modernism, even JavaScript[85] and npm[86] are being rebuilt with Rust.

Even if not everybody[87] shares the same enthusiasm, clearly, something is going on here. Is Rust mainstream or hype[88]?

Coda

Rust will invariably solve some issues in *today's* programming, including[89] security-related trouble such as Heartbleed[90] or the `goto fail` fiasco[91] of 2014. But Rust will invariably introduce new issues, completely unforeseen as of now. And a new, *modern* programming language will appear in 2050 or 2060 solving those issues, and the rewrite cycle will begin all over again. And people will wonder how come anyone could get anything done in C.

In 2016 I mentioned in a talk[92] it was important to pay attention to LLVM[93] as the source of the great things to come.

> Everybody is raving about Swift, but in reality what I pay more attention to these days is LLVM itself. I think LLVM is the **most** important software project today, as measured in its long-term impact. Objective-C blocks, Rust & Swift (...) all of these things were born out or powered by LLVM.

Time will tell if this prediction was right. In the meantime, KubeCon + CloudNativeCon Europe 2021[94], the main conference of the Kubernetes world, opens up online the same day this article hits the press, featuring this time... a one-day special Cloud Native Rust Day[95].

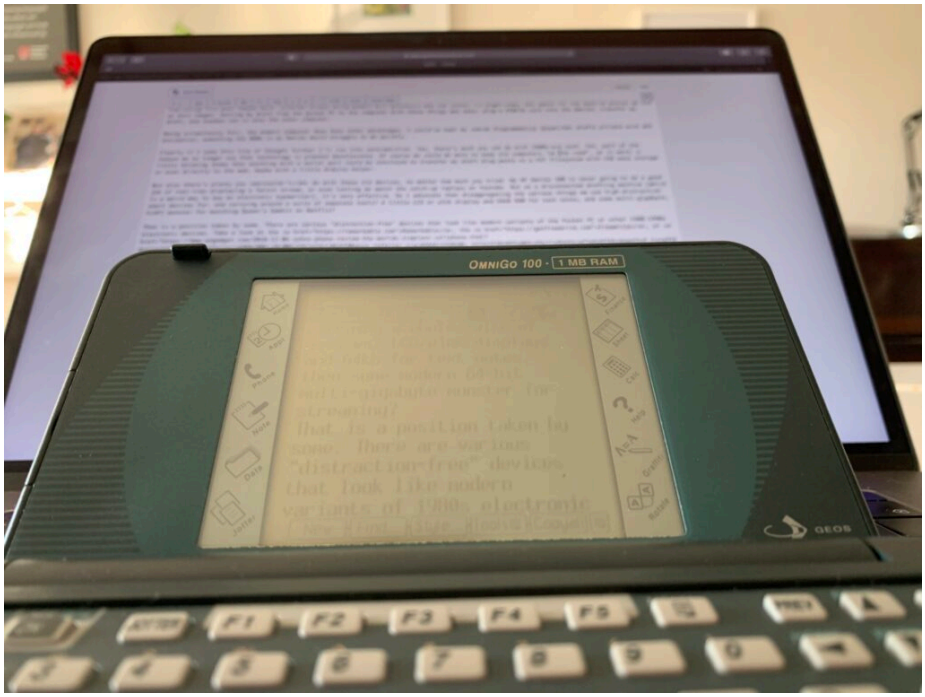
Cover photo by Jack Hamilton[96] on Unsplash[97].

[82] https://web.archive.org/web/20140110071058/https://www.drdobbs.com/jvm/the-rise-and-fall-of-languages-in-2013/240165192
[83] https://deprogrammaticaipsum.com/eric-sink/
[84] https://ericsink.com/entries/lousygrep.html
[85] https://deno.land/
[86] https://www.rust-lang.org/static/pdfs/Rust-npm-Whitepaper.pdf
[87] https://hackernoon.com/programming-in-rust-the-good-the-bad-the-ugly-d06f8d8b7738
[88] https://deprogrammaticaipsum.com/mainstream-is-the-new-hype/
[89] https://docs.rs/rustls/0.19.1/rustls/manual/_01_impl_vulnerabilities/index.html
[90] https://en.wikipedia.org/wiki/Heartbleed
[91] https://en.wikipedia.org/wiki/Unreachable_code#goto_fail_bug
[92] https://akos.ma/blog/being-a-developer-after-40/#9-llvm
[93] https://llvm.org
[94] https://events.linuxfoundation.org/kubecon-cloudnativecon-europe/
[95] https://events.linuxfoundation.org/cloud-native-rust-day/
[96] https://unsplash.com/@jacc?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[97] https://unsplash.com/s/photos/shuffle?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Plus Ça Change...

Graham Lee

May 3rd, 2021



Writing a draft for this post was as easy on an HP OmniGo 100[1] from 1995 as on any modern computer. Arguably easier, because I was not distracted by social media, notifications, or the possibility of a quick dip into another task. There is just me, an LCD screen, two AA batteries, and a really bad keyboard.

Publishing this post needed more. Internet[2] access using modern Wi-Fi protocols and the latest TLS algorithms, and space for the massive photos we use as hero images. Getting my draft from the pocket PC to the computer with those things was easy: plug a PCMCIA card into the OmniGo, transfer my draft, and sneaker net it onto the other computer.

Being scrupulously fair, the modern computer does have other advantages. I could have kept my *De Programmatica Ipsum* drafts private with AES encryption[3]; something the 8086 in my OmniGo would struggle to do quickly.

Clearly if I take this line of thought further I will run into contradiction. Yes, there is much you can do with 1980s-era tech. Yes, part of the reason we no longer use that technology is planned obsolescence[4]. Of course we could do more to keep old computers "on the road", as it were: a little thinking shows that anything with a serial port could be convinced to transfer

[1] https://www.hpmuseum.net/display_item.php?hw=200
[2] https://deprogrammaticaipsum.com/issue/issue-027-networking/
[3] https://deprogrammaticaipsum.com/issue/issue-003-security/
[4] https://deprogrammaticaipsum.com/issue/issue-018-obsolescence/

my draft blog posts to a FAT filesystem with USB mass storage or even directly to the web, maybe with a little Arduino helper.

But also there is plenty you *could not* do with these old devices, no matter how much you tried. My HP OmniGo 100 is never going to do a good job of real-time displaying a Twitch stream, or even letting me watch the catch-up replays on YouTube. But as a disconnected drafting machine (which is a weird way to say an electronic typewriter), it is very effective. Do I advocate then disaggregating the various things we use high-distraction smart devices for, and carrying around a suite of separate tools? A little LCD or eInk display and 64kB RAM for text notes, and some multi-gigabyte, hiDPI monster for watching "The Queen's Gambit"[5] on Netflix?

That is a position taken by some. There are various "distraction-free"[6] devices that look like modern variants of the Pocket PC or other 1980-1990s electronic devices. Take a look at the Remarkable[7], the Freewrite[8], or John's Phone[9]. Or if you want to go even more disconnected, how about a smart pen[10]?

I think the real point to bear in mind is that convergence, like so many other things in computing, looks like progress on a short timescale but a fad over the eons. We had one computer (if we were lucky) in the 1980s, then we had digital watches, digital cameras, handheld GPS, satellite navigation...and then convergence. Convergence looked like progress. But it was a fad, and the new fad is single-purpose "distraction-free" devices. Which look so much like what came before, that I can write my article on a 1995 pocket PC in the same way that someone with one of the fancy new things would.

The same holds for other trends. Agile software development looked like an advance over the heavyweight, management-led methods that preceded it, but now people complain that Scrum is weighed down by bureaucracy[11]. Now DevOps[12] promises to deliver us from the doldrums by creating self-organising teams who value working toward a shared goal over satisfying a preconceived plan.

Once there were computers. Then there was batch processing, and renting out computer time. Then there was timesharing, with billing-on-demand. Then there were personal computers[13]. Then there was the cloud. *Plus ça change, plus c'est la même chose.*

The lesson in this is that while you can get ahead as a developer by jumping on the latest bandwagon, that is not how to succeed as an architect, or as a cybernetician. Step back from the fads, work out what your customer needs, and give that to them. It will be better, and some time soon it will be trendy again too.

Cover photo by the author.

---

[5] https://en.wikipedia.org/wiki/The_Queen%27s_Gambit_(miniseries)

[6] https://deprogrammaticaipsum.com/issue/issue-014-minimalism/

[7] https://remarkable.com

[8] https://getfreewrite.com

[9] https://www.engadget.com/2010-12-06-johns-phone-review-the-worlds-simplest-cellphone.html?guccounter=1&guce_referrer=aHR0cHM6Ly9kdWNrZHVja2dvLmNvbS8&guce_referrer_sig=AQAAAJtDdDaWv_jVCDzZE6BxM3Gq8BZuZQylCzOUzGzvqf1wGoDfA4udagxG1xF_CxrwfFK3ALNL78vosxYnChAS649KSIclaE7Z8bIf3Cy1un1ly8TLJiyw8ZLoq9k4HYlN4Zw1eB_8GmLFxEnFoZgzsdQ4mBqRgHPVtA-EigVdwo3t

[10] https://uk.livescribe.com

[11] https://www.sicpers.info/podcast/episode-28-fascinating/

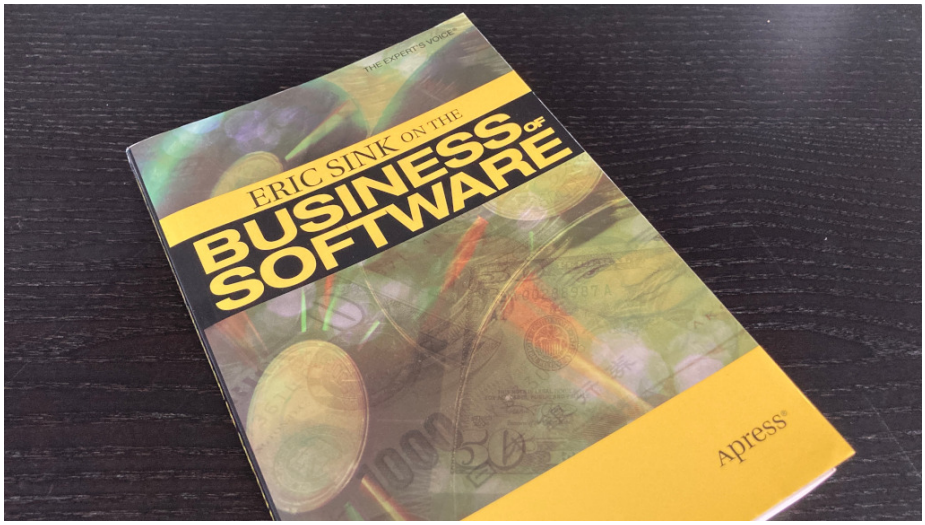
[12] https://deprogrammaticaipsum.com/issue/issue-016-devops/

[13] https://deprogrammaticaipsum.com/issue/issue-026-hardware/

Eric Sink

Adrian Kosmaczewski

May 3rd, 2021



Once upon a time, there was no GitHub, no iPhone, no AWS, no Android, no Google App Engine, no Stack Overflow, no Docker, no Kubernetes, no Rust, no Go, no Swift, no Kotlin, no Git, actually Subversion was barely starting to appear in the radar. Most importantly, there were no App Store yet.

– What?

– Yes, there were no App Stores. No iPhone, no Android, no touchscreen smartphones.

– But were there smartphones?

– Oh, well, yes, but they were not as nearly as popular as cellphones. And they had plastic keyboards, you know.

– Wait, but, how did people get apps back then?

Well, there were several distributions channels, some of them going back to the eighties and seventies, like buying shrink-wrappped software in retail stores, and others relatively newer, like downloading from the web and paying for it using a credit card. The introduction of the concept of a curated app store, one of the last contributions from Steve Jobs to our industry, was clearly a revolutionary concept, and it worked particularly well for mobile applications, at least from Apple's stock price point of view. Although many desktop software developers consider that it did not work so well, like in the case of Mac software, for example.

In any case, the App Store is the third evolution in a long series of efforts for software vendors to be rewarded for their efforts of bringing software to us. And Eric Sink, in a book written in 2006, explained exactly just how to sell software in a world before the App Store.

Eric Sink is a seasoned software developer and entrepreneur, who worked in different projects mostly in the Microsoft Galaxy. He contributed to the Spyglass browser[1] (whose engine was

[1] https://en.wikipedia.org/wiki/Spyglass,_Inc.

the original rendering engine of the first editions of Microsoft Internet Explorer,) AbiWord[2] (yes, it used to be a commercial word processor before being open source) and the Vault[3] version control system (remember kids, this was way before Git or even Subversion existed.) Eric Sink not only did develop those products, he helped commercialize them, with various degrees of success.

And his book is precisely a series of interesting essays about blunders, mistakes, common misconceptions, hard truths, and fun anecdotes, all taken directly from his personal experience. Not only that, but he has absolutely no trouble in giving the actual figures and saying exactly how much did it cost to run his software business.

Eric Sink provides answers to many questions that still today are more relevant than ever for "small ISVs" or independent software vendors. Questions like what programming language should I use? What operating system should I write my application for? How to advertise my application? What trade shows should I attend? What kind of people should I hire? How should I keep accounting in my company? These questions are as valid today as they were sixteen years ago, even if the relevant answers are not the same.

While I was running my admiteddly small own business, this book was one of my secret weapons. I remember with intensity the parts where he recommends to always keep an eye on the amount of cash in hand; paying attention to that single variable helped me a lot to manage my activities, to stay out of debt, and to be able to invest and grow.

In an age and time where software developers are debating the relative merits of App Store versus website distribution, where the 30% cut of platform vendors really hurts, and where users are more than ever aware of the privacy issues brought by advertising, it is refreshing to go ten years back in time and remember how things used to be.

Maybe the answers for the future of our industry will require a bit of adjustment, the abandonment of some romantic views and the creation of new paradigms. Maybe it is time to find a mechanism that protects the privacy of users, that provides sustainable cash flows to businesses, and that will be profitable for platform vendors as well.

Maybe it is finally time for antitrust regulation. Maybe.

A final word about form: this book was born out of a blog, as a conglomerate of posts the author had published in the "The Business of Software" column of the Microsoft Software Developer Network (MSDN) website[4]. Most of those articles are still available at the author's blog[5]. The idea of creating a book out of a blog is not new; other famous examples of this style are for example Presentation Zen[6] by Garr Reynolds, Joel on Software[7] by Joel Spolsky, Being Geek[8] by Michael Lopp, and many, many others. The morale of the story is that if you have a good blog, you might as well have the germ of a good book in between those posts.

Cover photo by the author.

[2] http://abisource.org/
[3] http://sourcegear.com/
[4] https://msdn.microsoft.com/en-us/default.aspx
[5] http://ericsink.com/bos/Business_of_Software.html
[6] http://www.presentationzen.com/
[7] http://www.joelonsoftware.com/
[8] http://randsinrepose.com/books/