Issue 048: Evolution

Adrian Kosmaczewski

September 5th, 2022

Welcome to the forty-eighth issue of *De Programmatica Ipsum*, dedicated to the subject of *Evolution*, and closing the 4th year of publication of this magazine.

This month we have news to announce. From now on, this magazine will include a new section: *Vidéothèque*, where we will talk about important videos online related to the practice and activity of programming. This 48th edition is also the first one without the active participation of Graham, who for professional reasons has been constrained to step aside from contributing.

In this edition:

- We will analyze how Alan Perlis thought programming languages would evolve[1] from 1966 to today.
- In the Library section[2], we will talk about "Platform Revolution"[3] by Parker, Van Alstyne, & Choudary.
- In the new Vidéothèque section[4], we will talk about the time Bret Victor[5] took us on a trip to 1973.

We would also like to thank our patrons who generously contribute every month (or have contributed in the past) to our work, and help us run this magazine. Thank you so much! In alphabetical order: Adam Guest, Adrian Tineo Cabello, Christopher Nascone, Jean-Paul de Vooght, Patryk Matuszewski, Paul Hudson, and Roger Turner.

[1] https://deprogrammaticaipsum.com/alan-perlis-and-the-evolution-of-programming-languages/
[2] https://deprogrammaticaipsum.com/category/library/
[3] https://deprogrammaticaipsum.com/geoffrey-g-parker-marshall-w-van-alstyne-sangeet-paul-choudary/
[4] https://deprogrammaticaipsum.com/category/videotheque/
[5] https://deprogrammaticaipsum.com/bret-victor/

Enjoy this issue! Please subscribe to our free newsletter[6] to stay updated about new releases, share the articles on social media, or contribute[7] if you would like to support our work.

Cover photo by Eugene Zhyvchik[8] on Unsplash[9].

[6] https://deprogrammaticaipsum.com/newsletter/

[7] https://deprogrammaticaipsum.com/contribute/

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

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

Alan Perlis And The Evolution Of Programming Languages

Adrian Kosmaczewski

September 5th, 2022

Alan Jay Perlis[1] knew a thing or two about programming languages, both as an early pioneer of our industry and as one of the designers of ALGOL. The language that has inspired the one you, dear reader of this magazine, probably use every day to earn a living.

In his first-ever ACM Turing Award Lecture[2], "The Synthesis of Algorithmic Systems," in 1966, Dr. Perlis enumerated three ways programming languages evolve from one to the next.

> Successor languages come into being from a variety of causes:
>
> (a) The correction of an error or omission or superfluity in a given language exposes a natural redesign which yields a superior language.
>
> (b) The correction of an error or omission or superfluity in a given language requires a redesign to produce a useful language.
>
> (c) From any two existing languages a third can usually be created which (i) contains the facilities of both in an integrated form, and (it) requires a grammar and evaluation rules less complicated than the collective grammar and evaluation rules of both.

Perlis did not stop there. In 1982 he published his famous epigrams[3], which you have most probably read on Twitter. Those observations were natural memes, even if that name did

[1] https://en.wikipedia.org/wiki/Alan_Perlis
[2] https://amturing.acm.org/award_winners/perlis_0132439.cfm
[3] http://pu.inf.uni-tuebingen.de/users/klaeren/epigrams.html

not exist.

> 41. Some programming languages manage to absorb change, but withstand progress.

Let us look in detail at how each of these (three plus one) principles stated by Perlis can be applied to the evolution of programming languages of the last 56 years. Maybe much of this evolution was inspired by Peter Landin[4] and his 1966 (what a year, huh?) ISWIM[5] proposal for the following 700 languages[6]; let us not digress and identify some patterns of programming language evolution by looking backward in time.

Natural Redesign Towards A Superior Language

What do we understand by "superior"? As privileged observers from 2022, we can quickly identify some "modern" traits that make languages undoubtedly (well, at least from our point of view) superior. Let us mention a few: support for types (if at all) or of a more robust type system than its predecessor (although abusing this characteristic invariably leads to longer compilation times); essential IDE support; the generation of safer and (or) faster code; type inference facilities; a stronger community and ecosystem.

Languages featuring such changes represent small evolutionary steps, welcome but not revolutionary changes. They are praised on Hacker News and adopted by the community as welcome improvements over existing languages. Some examples, roughly in chronological order:

- C (for B)
- Pascal (proposed by Wirth as a better ALGOL 60, even though Brian Kernighan[7] did not like it[8])
- D (for C++)
- Raku[9] (for Perl)
- Scala (for Java)
- Crystal[10] (for Ruby)
- Hack[11] (as an attempt by Facebook to bring PHP to another level)
- Carbon[12] (for C++, maybe?)

Some of these redesigns involve syntax changes (D, Pascal, Hack); some others with a recreation of their compilers or runtime models (Crystal, HipHop); in other cases, introducing a completely different paradigm (Scala) to an existing platform (the Java virtual machine) or even a new syntax but with backward compatibility (Carbon.)

Redesign Towards A Useful Language

Paraphrasing Bjarne Stroustrup, this is the realm of languages everyone complains about[13]. What do we understand by "useful"? In the opinion of this author, pragmatic languages.

[4] https://en.wikipedia.org/wiki/Peter_Landin
[5] https://en.wikipedia.org/wiki/ISWIM
[6] https://www.cs.cmu.edu/~crary/819-f09/Landin66.pdf
[7] https://deprogrammaticaipsum.com/brian-kernighan/
[8] https://www.lysator.liu.se/c/bwk-on-pascal.html
[9] https://lwn.net/Articles/802329/
[10] https://crystal-lang.org/
[11] https://en.wikipedia.org/wiki/Hack_(programming_language)
[12] https://9to5google.com/2022/07/19/carbon-programming-language-google-cpp/
[13] https://www.goodreads.com/quotes/226225-there-are-only-two-kinds-of-languages-the-ones-people

The ones that get shit done[14]. The languages we use every day. They are generally featuring the "industry" moniker, usually easier to understand by masses of engineers and usually (sadly) adopted by academia[15] at some point. Bundled with everything smart people need to get stuff done[16]: a package manager, a "batteries included" library of pre-built functions, a (not very) strict type system (which predictably yields a healthy equilibrium between correctness and productivity), and advanced IDE support.

And marketing. Lots of marketing. Usually, a big organization is behind, financing the evolution of the language through a foundation, committee, GitHub project, or some other similar mechanism.

They also feature more uncomplicated licensing conditions and might include some functional programming facilities over their predecessors; it is undoubtedly fancier to use a `map()` function than to use a `while`. Even though, according to Turing, both will get the job done.

Some examples? Classics:

- Delphi and Turbo Pascal (as useful Pascal compilers)
- C# (as a useful Java)
- Pharo (as a useful[17] Smalltalk)
- F# (as a useful OCaml)
- Kotlin (as another useful Java, but this time compatible with it)
- Elixir (as a useful Erlang)
- Scheme and Clojure (as a useful Lisp)
- TypeScript (as a useful JavaScript[18])

The astute reader will have realized by now that at least three languages designed by Anders Hejlsberg[19] appear in this list. As a software engineer concerned with solving practical problems for my customers, I have always favored useful languages in my work. But developers should be beware of Dr. Perlis' third aphorism because it is in this category where it hurts the most:

> 3. Syntactic sugar causes cancer of the semi-colons.

There are more "market driven" evolutions in this category; languages that mostly evolve their associated libraries or ecosystem to fulfill some novel role:

- R and Python have effectively replaced Fortran as the *de facto* language for scientific calculus.
- Python also reinvented itself as the language of AI.
- C# chose to evolve alone and gradually replace itself. The current iteration of the language is quite different from when it debuted in 2000: arguably, it was a Microsoft-owned clone of Java.
- Go became the language of choice for Cloud Native systems and cross-platform CLI tools; Ballerina[20], although initially intended to compete in this space, does not have nearly the same traction.
- Dart, a boring[21] but highly effective language, is slowly positioning itself in the mobile

[14] https://the.scapegoat.dev/why-i-love-php-and-javascript/
[15] https://deprogrammaticaipsum.com/teacher-leave-this-kid-alone/
[16] https://www.joelonsoftware.com/2007/06/05/smart-and-gets-things-done/
[17] https://medium.com/smalltalk-talk/behold-pharo-the-modern-smalltalk-38e132c46053
[18] https://deprogrammaticaipsum.com/innovationscript/
[19] https://en.wikipedia.org/wiki/Anders_Hejlsberg
[20] https://ballerina.io/
[21] https://akos.ma/blog/dart-is-boring/

app development market. With the current (in the opinion of this author, deserved) backlash against React Native, Dart has a real chance to grow beyond its current status.

From Many Existing To An Integrated And Less Complicated One

These languages usually mark historical milestones[22], representing significant shifts in the programming industry. Usually, there is one of these every 18 years, give or take, roughly following Proebsting's Law[23], named after University of Arizona computer science professor Todd A. Proebsting. Perhaps contradicting Dr. Perlis, history showed that sometimes there were more than just two languages involved in the evolutionary process. Normal, since we have many more languages today than in 1966.

- C++ (at least in its early forms 40 years ago) evolved from Simula and C;
- Objective-C, taking cues from Smalltalk and building upon C;
- Java, from[24] Objective-C, Simula, and Smalltalk;
- Rust, from[25] C, C++, OCaml, Haskell, and many others.
- Swift, from Objective-C, Haskell, Rust, and so many others.

Rust is arguably[26] targeting a higher goal today than any other language in the past 60 years, aiming to unify computer science with software engineering. Let us meet again in this magazine in one or two decades and see if it kept its promise.

The problem in this category is that, although Dr. Perlis sees the final result as "less complicated," these languages become extremely complicated[27] as they grow.

Absorbing Change But Withstanding Progress

Dr. Perlis' fourth category holds some venerable languages that have been able to pass the test of time, staying almost unscathed since version 1.0, almost begrudgingly adopting some fads (object-oriented programming, multiple CPU support, etc.) while remaining (stubbornly?) faithful to their origins: COBOL; Lisp; APL; Forth; PHP; BASIC; C; and Go.

These languages are withstanding progress, yes, but only to a certain degree. COBOL has had a new standard roughly every 20 years, and C got a new one every decade. Go has recently adopted generics, its most significant addition in 15 years. BASIC has yielded many offspring, yet its core has seldom changed. PHP features type annotations. Lisp will always be Paul Graham[28]'s preferred programming language. Nearly all of these languages have been safe bets for a long-sustained career in software development during (at least) the past 25 years.

Comes to mind Greenspun's Tenth Rule[29]:

> Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.

[22] https://thenewstack.io/rust-creator-graydon-hoare-recounts-the-history-of-compilers/
[23] https://proebsting.cs.arizona.edu/law.html
[24] https://cs.gmu.edu/~sean/stuff/java-objc.html
[25] https://www.infoq.com/news/2012/08/Interview-Rust/
[26] https://people.kernel.org/linusw/rust-in-perspective
[27] https://deprogrammaticaipsum.com/complex-vs-complicated/
[28] http://www.paulgraham.com/lisp.html
[29] https://en.wikipedia.org/wiki/Greenspun's_tenth_rule

Observations

The overall life expectancy of a programming language has dwindled in the past 56 years. A COBOL developer in the 1960s most probably retired in the 2000s, still writing COBOL. As a former professional VBScript, then C#, then Objective-C, later Swift, and finally Go developer, I can only see this trend accelerating. We should expect our favorite programming language to be replaced and removed from the market in a relatively shorter time every decade. To add insult to injury, new versions of the same programming language are sometimes incompatible with their previous ones.

The above is one of the reasons why this author believes that hyper-specialization[30], as demanded and supported by industry pundits, is a risk, a bet.

Another of Perlis' epigrams tells us that the evolution of programming languages is an unsolved problem:

> 73. It is not a language's weaknesses but its strengths that control the gradient of its change: Alas, a language never escapes its embryonic sac.

And if we believe Geoffrey James[31],

> Each language has its purpose, however humble. Each language expresses the Yin and Yang of software. Each language has its place within the Tao.
>
> But do not program in COBOL if you can avoid it.

Cover photo by GR Stocks[32] on Unsplash[33].

[30] https://deprogrammaticaipsum.com/specialization-is-for-insects/

[31] http://www.canonical.org/~kragen/tao-of-programming.html

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

[33] https://unsplash.com/s/photos/chess-king?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Bret Victor

Adrian Kosmaczewski

September 5th, 2022

Welcome to the first post of the new *Vidéothèque* series of *De Programmatica Ipsum*. In this new section, we would like to share with you what we consider the best videos available online about programming, the history of computing, software development, architecture, and other related fields. Video as a communication medium has grown unprecedentedly in the past 15 years; in this section, we will propose the most relevant among many classics.

The first video of this series will feature none other than Bret Victor[1] in one of his most memorable moments. We have already discussed Bret in a previous article about Smalltalk[2], where we named him one of the most influential figures in that galaxy. This video, however, shows that his light shines much brighter than that.

In a mind-twisting experiment, Bret Victor takes us back to July 1973. His demeanor, dress code, and even the mechanism by which he illustrates his words–an overhead projector[3], of all things–all take us back in time. Bret took attention to the slightest details, like the pens in his shirt pocket. Or even his haircut. And most of all, his words, like greeting peers as "programmers of automatic computing machines" in all seriousness.

Together with form, there is content.

His initial thesis (easily verifiable) is that people's thinking evolves slower than technology. He goes to show how every big evolution in programming (from binary to assembler, from assembler to FORTRAN, and then from language to language) is met with contempt or resistance, in the worst cases, hostility.

Bret goes on to explain four major ways programming could evolve, as seen from the point of view of a hypothetical researcher in the early seventies.

[1] https://en.wikipedia.org/wiki/Bret_Victor

[2] https://deprogrammaticaipsum.com/the-absolute-no-frills-quite-ignorant-very-incomplete-and-certainly-flawed-beginners-guide-to-smalltalk/

[3] https://en.wikipedia.org/wiki/Overhead_projector

Or, put another way, showing us what the future used to look like.

- From coding to directly manipulating data, explaining Ivan Sutherland[4]'s Sketchpad[5] (1962);
- From procedures ("how") to stating goals and constraints ("what",) showcasing Carl Hewitt[6]'s Planner[7] system (1969,) Alain Colmerauer[8]'s Prolog[9] (1972,) Ralph Griswold[10]'s SNOBOL[11] (1962,) Ken Thompson[12]'s regular expressions (1967,) and J. C. R. Licklider[13]'s "Intergalatic Computer Network"[14] as examples;
- From plain text to graphical and spatial representations of programs, following the steps of Douglas Engelbart[15]'s legendary NLS System[16] (1968,) T. O. Ellis[17]' GRAIL[18] project (1968,) and Larry Tesler[19]'s Smalltalk;
- From sequential to parallel programming, highlighting the limits of the Von Neumann architecture, the impact of the Intel 4004[20] microprocessor (1971,) and Carl Hewitt's Actor Model[21] (1973.)

Of course, the sad conclusion (or funny, depending on how you see it) of this video is that our industry has not fully evolved in any of these directions. Well, maybe the last one did happen (somehow, through multicore architectures and Erlang[22]) because as Herb Sutter said, the free lunch was over[23] around 2002. But as Bret mentions around the 16th minute[24], we still lack services that can figure out how to talk to each other by themselves, leading to a brittle world of interconnected microservices, breaking every so often and fueling a whole market of observability and management tools.

Yet, here we are.

> I'm totally confident that in 40 years we won't be writing code in text files. Right? We've been shown the way.

(Minute 20:33[25]... and listen to another memorable quote at minute 26:07[26] that I will let you discover.)

Most of the code we put in production is written in linear text form, compiled or interpreted, and painfully validated, verified, and evaluated by various (human and automated) mechanisms. Except for some visual environments, mostly used to create GUIs, sometimes using

[4] https://en.wikipedia.org/wiki/Ivan_Sutherland
[5] https://en.wikipedia.org/wiki/Sketchpad
[6] https://en.wikipedia.org/wiki/Carl_Hewitt
[7] https://en.wikipedia.org/wiki/Planner_(programming_language)
[8] https://en.wikipedia.org/wiki/Alain_Colmerauer
[9] https://en.wikipedia.org/wiki/Prolog
[10] https://en.wikipedia.org/wiki/Ralph_Griswold
[11] https://en.wikipedia.org/wiki/SNOBOL
[12] https://en.wikipedia.org/wiki/Ken_Thompson
[13] https://en.wikipedia.org/wiki/J._C._R._Licklider
[14] https://en.wikipedia.org/wiki/Intergalactic_Computer_Network
[15] https://en.wikipedia.org/wiki/Douglas_Engelbart
[16] https://en.wikipedia.org/wiki/NLS_(computer_system)
[17] https://www.rand.org/pubs/authors/e/ellis_t_o.html
[18] https://www.rand.org/pubs/research_memoranda/RM5999.html
[19] https://en.wikipedia.org/wiki/Larry_Tesler
[20] https://en.wikipedia.org/wiki/Intel_4004
[21] https://en.wikipedia.org/wiki/Actor_model
[22] https://en.wikipedia.org/wiki/Erlang_(programming_language)
[23] http://www.gotw.ca/publications/concurrency-ddj.htm
[24] https://youtu.be/8pTEmbeENF4?t=960
[25] https://youtu.be/8pTEmbeENF4?t=1229
[26] https://youtu.be/8pTEmbeENF4?t=1567

the Cassowary linear constraint algorithm[27], most developers are still using their keyboards as a primary input method. GitHub Copilot just feeds that experience further; having an automated mechanism spitting unrelated code snippets stolen elsewhere will not, in the opinion of this author, change the *status quo* substantially in the long run.

In short, the experience of programming, the one you, dear reader, live day in and day out in your daily job, has not fundamentally evolved in the past 60 or even 70 years. Our command line environments even have the TTY (short for "teletype") moniker embedded in them. We are still coding... procedures... in text files... using sequential models.

We grew up, as programmers, learning only one way to do things, and considering it immutable. That is Bret's definition of a tragedy and the final conclusion of this stellar talk.

The video is available on YouTube[28] for your watching pleasure. And if this is not enough (I bet it will not be) just check the portfolio of Bret's work on his website[29].

Cover snapshot by the author.

[27] https://constraints.cs.washington.edu/solvers/cassowary-tochi.pdf
[28] https://www.youtube.com/watch?v=8pTEmbeENF4
[29] http://worrydream.com/

Geoffrey G. Parker, Marshall W. Van Alstyne, & Sangeet Paul Choudary

Adrian Kosmaczewski

September 5th, 2022

Let us talk about a book that would be usually featured in the "Business" section of your nearest bookstore. As such, it might have been overlooked by those inspecting the shelves of the "Computer" section. This book delves deeply into the economic fabric of the software industry and, for that reason, becomes a much-needed read by all software workers.

As Graham memorably said in a previous article[1],

> A *good programmer's library (I will let you decide whether that is a good library owned by a programmer, or a library belonging to a good programmer) includes essays, scholarly articles, videos, magazines, blog posts, podcast episodes, and more.*

Yes, software professionals should read more business books; in this author's experience, technologists and geeks tend to grossly misunderstand how the economic machine works, similar to how business-minded folks grossly misunderstand how computers and software work. The impossible dialogue[2] is pervasive and must be fought back at all costs.

"Platform Revolution: How Networked Markets Are Transforming the Economy and How to Make Them Work for You[3]," written by Geoffrey G. Parker, Marshall W. Van Alstyne, &

[1] https://deprogrammaticaipsum.com/tim-peters/
[2] https://deprogrammaticaipsum.com/the-impossible-dialogue/
[3] https://wwnorton.com/books/Platform-Revolution/

Sangeet Paul Choudary and published in 2016, successfully bridges the two worlds. The central thesis of this short volume argues that the key to understanding the dramatic, exponential rise in popularity of some software ideas in our world (and the riches thereby generated) is embedded in the word "Platform." A word that is pervasive in our industry.

Software developers use, create, endorse, promote, bash, suffer, glorify, and belittle platforms day in, and day out. The technologies we use to develop applications are often referred to as "Platforms," including, but not limited to, programming languages and application frameworks. Mobile developers[4] publish apps on a *de facto* duopoly of "Platforms"[5], the so-called "App Stores" dictating the life and death of programmers' work without supervision or regulation. DevOps engineers nowadays choose Kubernetes[6] over Docker Swarm because of its "ecosystem," another fancy word closely related to "Platform."

So those are examples of platforms, but what is a "Platform" after all? How does it work? And why did they become so economically significant nowadays?

The core thesis of this book is that "Platform" is a new paradigm for economic analysis; not just a fancy word to describe a successful business model or a set of technologies, but rather a much larger idea: *the definitive economic architecture of our era.*

Let us apply the historical perspective; by 2016, we already had many examples of successful software-powered platforms. To name a few: Airbnb, Lyft, Uber, Twitter, Android Play Store, YouTube, iOS App Store, Facebook, WordPress, Hacker News, Amazon Web Services, PayPal, Craigslist, Java, GitHub, WhatsApp, eBay, Salesforce, IBM PC & "Wintel," Microsoft Office, Instagram, Linux, IBM's System/360, and so many more. Some new ones have emerged since the publication of this book; come to mind TikTok and Kubernetes while we are closely watching the much-hyped Metaverse to see how it unfolds.

These platforms have, in turn, generated waves of insanely wealthy psychopaths moguls: Adam Neumann, Mark Zuckerberg, Bill Gates, Jeff Bezos, Elon Musk, Mitch Kapor, Marc Andreessen, Steve Jobs, Larry Ellison, Jack Ma, Paul Graham, Satya Nadella, Tim Cook, Satoshi Nakamoto, and so many more men and just a tiny number of women, because such are the ways of the human world in the twenty-first century.

The basic building blocks of platforms are elementary to understand: the more people use and extend specific combinations of hardware and/or software, the faster the demand for said combination grows, which in turn feeds a virtuous circle that ultimately generates a nonlinear, "convex" growth in both size and intrinsic value.

The keyword here is *nonlinear*. Keep it in mind.

Nonlinear patterns were first formally analyzed by Jay W. Forrester[7] and then explained in detail by his disciple John D. Sterman[8], both from the MIT Sloan School of Business. (Sterman's groundbreaking masterpiece "Business Dynamics[9]" deserves an article of its own in this magazine.)

Put in Sterman's and Forrester's framework, a system measurable in some quantity (a "stock") with a feedback arrow pointing back to itself (a "flow") naturally displays

[4] https://deprogrammaticaipsum.com/issue-44-mobile/
[5] https://deprogrammaticaipsum.com/on-the-duopoly-of-mobile/
[6] https://deprogrammaticaipsum.com/antonomasia/
[7] https://en.wikipedia.org/wiki/Jay_Wright_Forrester
[8] https://en.wikipedia.org/wiki/John_Sterman
[9] https://jsterman.scripts.mit.edu/Business_Dynamics.html

exponential growth; the canonical example of such behavior being that of compound interest[10].

Another way to explain platform patterns is Metcalfe's classic network effects[11]. Robert Metcalfe, before calling "[12]Luddites" to members of the burgeoning "open sores" community and before suggesting[13] that Internet technologies "are not so much a threat" to personal privacy, gave his name to an often-mentioned law[14], used as an easy algorithm to explain the exponential growth of network effects. Simply put, as per Metcalfe's law, two phones are connected by one line; three by three; four by six; five by ten; *et cætera*. The rule is that the number of connections is always N squared minus N, all divided by two, N being the number of participants.

But apply these patterns of feedback and growth to economic markets, and you start to understand what platforms are, and what they may (and have) become. In essence, "Platform Revolution" explains such systems' technicalities, architectures, monetization, strategies, economics, and governance policies. The secret to the riches of the biggest moguls of our era, explained in a concise tome, ready for you to learn.

Even more important, the book addresses platform regulation, a topic dear to this magazine:

> In general, the historical record doesn't support the arguments of people who favor *no* regulation of business. In fact, it's difficult to identify any developed marketplace that has been completely free of intervention by government authorities.

(Chapter 11, page 237, emphasis by the authors.)

Yes, regulation is a needed and even desirable property of platforms. As this article hits the press, the Federal Supreme Court of Switzerland dictated that Uber must recognize its drivers as employees[15]. It effectively limits the platform's growth, much to the chagrin of free-market pundits, but controls its "externalities" in our society. Because, as Sterman said:

> We frequently talk about side effects as if they were a feature of reality. Not so. In reality, there are no side effects, there are just effects. When we take action, there are various effects. The effects we thought of in advance, or were beneficial, we call the main, or intended effects. The effects we didn't anticipate, the effects – which fed back to undercut our policy, the effects which harmed the system – these are the ones we claim to be side effects. Side effects are not a feature of reality but a sign that our understanding of the system is narrow and flawed. Unanticipated side effects arise because we too often act as if cause and effect were always closely linked in time and space. But in complex systems such as an urban center or a hamster (or a business, society, or ecosystem) cause and effect are often distant in time and space.

("Business Dynamics," John D. Sterman, 2000.)

As more and more developers dream of "starting their own company" (usually by clumsily thinking that yet another app in a crowded App Store might be a reasonable breakout,) or

[10]https://en.wikipedia.org/wiki/Compound_interest
[11]https://en.wikipedia.org/wiki/Network_effect
[12]https://web.archive.org/web/20070316025237/http://www.infoworld.com/articles/op/xml/99/06/21/990621opmetcalfe.html
[13]https://americanhistory.si.edu/comphist/montic/metcalfe.htm
[14]https://en.wikipedia.org/wiki/Metcalfe%27s_law
[15]https://www.unia.ch/fr/actualites/actualites/article/a/19137

while others argue about their favorite and hated platform on Reddit or Hacker News, it would be unwise to do neither without first reading "Platform Revolution." Human minds are, by design, tragically limited in their comprehension of exponential growth patterns. We cannot intuitively analyze such evolutions, and we will invariably underestimate their impact. We are wired to naturally understand phenomena evolving in a linear fashion; no wonder why linear algebra is taught in high school, while other kinds of algebræ are taught in college.

The fact is that our world is, by all standards, behaving exponentially in many critical vectors we observe: global temperatures are growing exponentially faster; wars are exponentially more deadly; pandemics expand exponentially quicker; and software platforms become exponentially popular in shorter amounts of time, sometimes even threatening privacy, democracy, and the livelihoods of millions of people around the world. We humans are not prepared for such patterns, and books like this are crucial to going beyond our shortcomings.

We must humbly prepare for a future that will, painfully and immediately, outgrow any capacity we may have left for surprise or reckoning. Paraphrasing Sterman, our understanding of the world is getting narrower and more flawed every day.

Cover picture by the author.