Issue 023: Academia

Adrian Kosmaczewski

August 3rd, 2020

Welcome to the twenty-third issue of *De Programmatica Ipsum*, dedicated to the subject of *Academia*. In this edition:

- Graham explains the current state of software engineering in research[1], and how it could be improved.
- Adrian tells his own personal story[2] of wrong choices, lost time, and experiences learnt in university campuses.
- In the Library section[3], Graham talks about a foundational title[4] for NeXTSTEP and Mac OS X software development: "NeXTSTEP Programming Step One: Object-Oriented Applications" by Simson L. Garfinkel and Michael K. Mahoney.

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 Ross Sneddon[7] on Unsplash[8].

[1] https://deprogrammaticaipsum.com/on-research-software-engineering/
[2] https://deprogrammaticaipsum.com/teacher-leave-this-kid-alone/
[3] https://deprogrammaticaipsum.com/category/library/
[4] https://deprogrammaticaipsum.com/garfinkel-and-mahoney/
[5] https://deprogrammaticaipsum.com/newsletter/
[6] https://deprogrammaticaipsum.com/contribute/
[7] https://unsplash.com/@rosssneddon?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[8] https://unsplash.com/s/photos/academia?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

On Research Software Engineering

Graham Lee

August 3rd, 2020

Let's be plain upfront: academia dropped the ball on software engineering. Go back to the genesis of the field, and you see that computing was being advanced mostly by needs in the public sector, with the private sector playing a role. The first publicly demonstrated computer was produced by Konrad Zuse in 1941, and marketed by his company Zuse Apparatebau. But it was actually funded by the Nazi government in Germany, specifically the aerodynamic research institute, for its use in designing and flying cruise missiles.

The Z3 was the first Turing-complete machine, and Zuse went on to write the first computer chess program in the first high-level programming language Plankalkül, a language of his own devising and an unacknowledged forerunner to ALGOL. This was all part of his PhD thesis, but having failed to pay the submission fee to the University of Augsburg he did not obtain a degree.

World War 2 was, unsurprisingly, the big limitation on Zuse's success, as someone who worked on the Nazi war effort. Despite computing being a breakout field, its impact was not as obvious as something like rocketry and so Zuse was never recruited into the U.S. Operation Paperclip nor its Soviet counterpart, Osoaviakhim. A lot of his equipment was destroyed in bombing, and the terms of American occupation rule in West Germany meant that the country was neither permitted to develop electronic computing machinery nor was it financially capable of it. Zuse founded two post-war computer companies, both funded by ETH Zurich, and eventually delivered a computer to the technology institute in 1950. This was only the second computer ever to be sold, and the first commercial computer to work correctly: computer engineers were already moving fast and breaking things back then.

It was not only in continental Europe that early computing was a mix of military, academic, and commercial interests. Digital computers arose in both the UK and USA out of military applications: cryptanalysis in Britain and ballistics in the States. With the war over, people and material from the British project found their way to Manchester and Cambridge Uni-

versities. ENIAC, the first American general-purpose computer, found its way to the University of Pennsylvania. Meanwhile, IBM had independently invented a computer, which became the Harvard Mark I (and was used by the U.S. Navy, as well as a Manhattan Project mathematician by the name of John von Neumann). All of these anglophone projects came together at the Moore School Lectures[1]. John Von Neumann write up his notes from the lectures, and published an early draft in which he hadn't added the citations. Thus was born the "Von Neumann" architecture, definitely invented at least once by Turing (who Von Neumann had talked to about it) and once by Zuse (who he hadn't).

You wouldn't know now that these three strands of development in computing are related. Judging from today's computing scene, there's an academic pursuit called "Software Engineering" where people apply maths to things like design patterns and formal methods that most professional programmers discarded as Not Relevant or Too Hard decades ago. Professional programmers call themselves "engineers", but don't really do engineering so much as they take a guess at what they thought their customer wanted and then check back in a couple of weeks to see if they were correct. When the military needs computers, they buy them from Dell and and get software by Microsoft, IBM, or Google. Unless they're in a sanctioned country like Iran, when they buy them from a Dell reseller.

The three fields come together in one nexus: Carnegie-Mellon University's Software Engineering Institute. The SEI is where the military pays academia to appraise commercial software vendors.

Question one: what happened?

We could point to a number of big events that introduced divisions between military, commercial, and academic computing. Arguably the academic field sewed the seeds of their own irrelevance with Curriculum '68[2], the Association for Computing Machinery's "recommendations for academic programs in computer science". While the ACM had been a cross-disciplinary organisation of computing enthusiasts, this National Science Foundation grant-supported effort brought together a committee of academic men to talk about how academic programs in computer science could be more, well, *academic*, damn it.

It's common for educational programs to have some sort of sop to industry, something about "training the workforce of the future" or "getting students ready for life after university". Curriculum 68, which is explicitly about defending and consolidating the legitimacy of "computer science" as a pursuit worthy of the name science, which means all those pesky programmers, coders and operators are definitively out of scope, being mere blue collar button pushers.

> Although programs based on the recommendations of the Curriculum Committee can contribute substantially to satisfying this demand [for "substantially increased numbers of persons to work in all areas of computing"], such programs will not cover the full breadth of the need for personnel. For example, these recommendations are not directed to the training of computer operators, coders, and other service personnel. Training for such positions, as well as for many programming positions, can probably be supplied best by applied technology programs, vocational institutes, or junior colleges. It is also likely that the majority of applications programmers in such areas as business data processing, scientific research, and engineering analysis will continue to be specialists

[1] https://en.wikipedia.org/wiki/Moore_School_Lectures
[2] https://dl.acm.org/doi/abs/10.1145/362929.362976

> educated in the related subject matter areas, although such students can undoubtedly profit by taking a number of computer science courses.

The ACM succeeded at the task of making computer science its own pursuit, by decoupling its relevance to the application of computing technology. "Software Engineering" tended toward the problem of assembling prefabricated units and hoping they approached the customer's need, while computer science retreated into the mathematics of algorithms and information theory. Programmers needed to know how many screen updates would be missed in sorting a list of ten mailboxes, and computer scientists told them how many instructions it would take on a theoretical processor writing out to a paper tape.

The AI winter was not kind to academic computing either. Having promised thinking machines just around the corner ever since Turing had thought about it a bit in the 1940s, it became abundantly clear that they weren't. In fact, it became evident that the sorts of problems you could pay a programmer to think hard about and write a reasonably efficient program to solve would require computers that hadn't been invented yet using novel techniques like neural networks and random forests. By the time the computers had been invented, they were no longer the preserve of the university.

While all of this was happening, commercially-owned pure research institutions were, at least in the US, taking on the job of actually inventing the future. Xerox PARC, the home of ethernet networking, laser printing, graphical user interfaces, the mouse, video conferencing, the tablet computer, object-oriented programming, and more, is well known. As is Bell Labs, the home of UNIX, statistical process control, transistor electronics, speech synthesis, the C and C++ programming languages, and more.

Reality is a bit muddier than this picture: UCLA and DARPA worked together to invent the internet (which was quickly appropriated by commercial service providers); Niklaus Wirth contributed many programming languages that people actually use (quickly appropriated by commercial developer tools vendors), and many operating systems and hardware designs that they actually don't, across his academic career. Mach, the kernel and operating system design used by NeXTSTEP, mkLinux, macOS/iOS/watchOS/tvOS, GNU/HURD, MachTen, OSF/1 and others, was a mash-up invented at Carnegie-Mellon University using bits from University of California, Berkeley. But by the twenty-first century academic computing was on the back foot, trying to find logical positivist things to say about Agile practices. Academia is now subservient to market forces after five decades of Friedman ideology, so computer science departments beg for handouts from commercial behemoths to become outsourced risk-takers (research salaries are much lower than Silicon Valley salaries), or push their academics to "spin out" their inventions into start-up companies to get into the orbit of those behemoths and get some of the sweet acqui-hire money.

Question two: what's happening now?

Meanwhile, the need for computing in academia itself only got greater. Meeting invitations moved from internal pigeon post to email. Papers moved from, well, paper to Word and LaTeX. High-Performance Computing—the application of supercomputers to various research problems, mostly but not exclusively in the sciences—both grew in application and shrank in innovation as the vendors learned to build machines from commoditised parts. Both the Apple PowerMac and the Sony Playstation 2 have appeared in the TOP 500 list of supercomputers. Mathematical proofs became so complex they could only be constructed by computer, and only tested by computer. Statistical analysis became the preserve of R, Python, and similar tools. Biology begat bioinformatics and computational biology. And so

on.

More and more software in research, and a growing realisation that the way software is treated by the research community not only isn't state of the art, but is actually holding research back. Pick an HPC simulation at random and you'll be lucky if you can get it to build at all, and even then only on the same cluster with exactly the same Fortran-77 compiler, MPI library, CPUs and GPUs as the researcher who last touched it. If it does build, you won't find any tests to tell you whether it will reproduce the same results that researcher obtained, or whether those results represent real science rather than buggy outcomes or compounded floating-point inaccuracies.

That's where I, my team, and people like us come in. You see, I may be an erudite essayist once per month over here, but most of the time I'm a Research Software Engineer[3]. Our task is to make sure good practices from commercial software engineering are being used in research, helping researchers achieve their goals through software. Often that means we write the software ourselves[4]. Sometimes it means promoting practices like Continuous Integration[5], or automated testing[6]. Often it means training academics[7] in programming languages and tools.

Our goal is to build a community of practice[8] across academia, improving the quality of research software everywhere.

Cover photo by Sidharth Bhatia[9] on Unsplash[10].

[3] https://www.rse.ac.uk

[4] https://github.com/OxfordRSE/

[5] https://sgibson91.github.io/blog/continuous-integration-fail-fast-fail-first/

[6] https://podcasts.ox.ac.uk/developing-better-code-automated-testing

[7] https://www.rse.ox.ac.uk/events/2020-06-17_code_management_course/

[8] https://society-rse.org/community/

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

[10] https://unsplash.com/s/photos/radcliffe-camera?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Teacher, Leave This Kid Alone

Adrian Kosmaczewski

August 3rd, 2020

Regular readers of this column already know[1] much of my personal story, including the fact that I am a self-taught software developer. They know that I started programming my Casio fx-180p[2] programmable calculator in the 1980s. They also know that in 1992 I bought my first PC, a venerable 380 SX tower with a whooping 128 MB hard disk and 2 MB or RAM, where I wrote my first[3] "Hello, World!" program in QBasic[4]. What they do not know is how I dropped out of college. Not once, but five times.

> I paid a hundred and twenty thousand dollars for someone to tell me go read Jean Austen. And then I didn't. – John Mulaney[5].

My family history somehow predestined me to go to university.

My maternal grandmother, Herta Schlerff, was born in Philippopolis[6] 3 days after a certain Neumann János Lajos[7] saw the light in Budapest; both were born on adjacent empires. She was among the first women[8] to graduate in the faculty of sciences in the University of Geneva, with a "Licence ès sciences mathématiques." Her diploma hangs proudly on the wall of my home office nowadays. She was also one of the few members of the statistical office of the League of Nations[9], that failed predecessor to our current United Nations. I can hardly put in words the pride that I feel when I think about her.

In my family, the historical connections to the software industry run deep. Both my uncle

[1] https://akos.ma/blog/being-a-developer-after-40/
[2] https://www.calculator.org/calculators/Casio_fx-180P.html
[3] https://deprogrammaticaipsum.com/memories-of-hello-world/
[4] https://en.wikipedia.org/wiki/QBasic
[5] https://youtu.be/aiqKK4ysI7g
[6] https://en.wikipedia.org/wiki/Plovdiv
[7] https://en.wikipedia.org/wiki/John_von_Neumann
[8] https://www.unige.ch/archives/files/9115/3866/3096/Diplomes_1926-1927.pdf
[9] https://en.wikipedia.org/wiki/League_of_Nations

and my mother worked for IBM back in the 1960s, precisely at the same time during which a certain Fred Brooks[10] brought the System/360[11] project to life. My home had plenty of old IBM manuals that my mother had kept from her days at Big Blue; sadly we left them behind as we moved to Europe. He had a degree from ETHZ. She did not go to college.

As far as I am concerned, it was not to be my call to simply graduate from college. I simply hated every single day I sat at a university hall. Crushed between my family's pressure and my cronic self-doubt, I forged for myself a tortuous path across colleges in two continents.

Learning, as it is practised in the universities I have attended, simply does not work for me. I have many friends with PhDs and I know that many of them (not all) have loved the experience, and keep on piling postgraduate and postdoctoral degrees on top of each other. In my case, well.

- 1993: I entered the University of Geneva[12] to study physics; yes, the same one that my grandmother attended. I dropped out in 1996.
- 1998: I joined the Universidad de Buenos Aires[13] to learn business management. I lasted until 2000.
- 2001: Applied to the Universidad de Palermo[14] (also in Buenos Aires) to study marketing. I left after 3 months.
- 2002: Started studying computer science in EPFL[15]. I lost interest after 6 months.
- 2019: I wanted to learn some business-related thingy in ETHZ[16]. I could not stand it more than 3 months.

Of course, every time I had to bear the chorus of disappointed family members, repeating *ad nauseam* that my decision was wrong or misguided or even inappropriate. The thing is, I am not made for university campuses.

I just cannot avoid the feeling of losing my time in an overcrowded university hall. Listening to an almighty teacher trying to fill my brain with dogma and boredom, two or four hours a week, so that I could go and regurgitate it all in a term exam at the end. Rinse and repeat enough times and you have got yourself a paper hanging on the wall.

I wanted to make things. I should have gone to a more technical school, or applied to an apprenticeship. Heck, I wanted to be a musician when I was a kid, dammit.

In the meantime, however, I got a Degree of Master of Science in Information Technology of the University of Liverpool[17], that took me 3 years to complete. I got accepted into that program in 2005, solely thanks to my professional experience, as I had no bachelor degree whatsoever. It was a distance program, but got the privilege of receiving my diploma in person in 2008, wearing my gown, hood and mortar by Ede and Ravenscroft Ltd.[18], official robe-makers of the University of Liverpool and other institutions since 1689.

Yes, I have photos of me dressed like that, even a DVD. My mother cried so hard when I showed her the pictures and the video, it is still shocking to me to think that such a path might have been so important to her.

[10] https://en.wikipedia.org/wiki/Fred_Brooks
[11] https://en.wikipedia.org/wiki/IBM_System/360
[12] https://en.wikipedia.org/wiki/University_of_Geneva
[13] https://en.wikipedia.org/wiki/University_of_Buenos_Aires
[14] https://en.wikipedia.org/wiki/University_of_Palermo_(Buenos_Aires)
[15] https://en.wikipedia.org/wiki/%C3%89cole_Polytechnique_F%C3%A9d%C3%A9rale_de_Lausanne
[16] https://en.wikipedia.org/wiki/ETH_Zurich
[17] https://www.liverpool.ac.uk/
[18] https://www.edeandravenscroft.com/graduation-services/

Going to college is the hallmark event for all generations of 20th century-minded people. These days, based on my experience, I would not recommend anyone to go to university if they are not the PhD type. It turns out that a lot of people I met in university went on to finish their degrees just to please their parents, and then found jobs in other realms, doing what they actually cared about. Which, funny enough, in many cases was programming. In my case I just went directly there, to the thing I enjoyed the most.

I remember in 1993, while studying physics, we had a programming class. Our professor taught us Pascal, and we had to submit exercises written in Turbo Pascal[19]. Of course we had to pass exams, but oddly enough, written in paper. Wait for it: we had to write programs in Pascal in paper, and we would have points removed if we forgot semicolons, or made other syntax errors. Thankfully I have never again had the sad privilege of attending a more backwards, stupid, ridiculous teaching experience in my whole life. If there ever was a great way of making lots of people deeply hate programming, it was exactly that one. Kudos for a dreadful experience.

Speaking about learning experience, I enjoy online classes the most. In that sense, I am a proud member of the Internet generation. I enjoy forging my own path across books, papers, and references. I have a dead-tree book library that is certainly getting heavier and heavier to carry around, together with a more virtual collection of papers[20] and ebooks[21]. I like to learn at my own rhythm.

In a strong and direct way, writing a new article for this monthly publication is another mechanism of my own learning experience.

Was it completely pointless? Of course not. Studying first and second year physics gave me a very strong maths background; this has helped me a lot in my career. As a counterexample of my programming class above, I can mention the classes of Algebra and Analysis. Regarding the latter, I still have the book[22] written by my Analysis teachers, Ernst Hairer[23] and Gerhard Wanner[24]. They taught us analysis following history, from the Renaissance to the 19th century. A fantastic idea, even though I had to take the exams three times until I passed. It was hard (oh yes) but very, very enjoyable.

(As an anecdote, I should add that one of my classmates back then was Sir Martin Hairer[25], 2014 Fields Medal winner. I am thankful to him for the immense patience he had while explaining me various subjects, countless times even. I'm not at all surprised he got that far.)

As for the aforementioned Algebra class, I must mention my teacher, Jean-Claude Hausmann[26]. In particular I remember him explaining the RSA algorithm to us, well before SSL certificates even existed. He was funny and witty, too, which is something many other teachers lacked. This class was, by all standards, a hallmark of my student years.

From my time in Buenos Aires, I must mention my teacher of economic history, although his name evaded my memory at the time of this writing. It was he who showed us Eric Hobsbawm, Amartya Sen, and Hannah Arendt. This course opened my eyes to history, and it was without any doubt the major trigger of my passion for software engineering history.

[19] https://en.wikipedia.org/wiki/Turbo_Pascal
[20] https://www.mendeley.com/download-desktop-new/
[21] https://calibre-ebook.com/
[22] https://link.springer.com/book/10.1007/978-0-387-77036-9
[23] https://en.wikipedia.org/wiki/Ernst_Hairer
[24] https://en.wikipedia.org/wiki/Gerhard_Wanner
[25] https://en.wikipedia.org/wiki/Martin_Hairer
[26] http://www.unige.ch/math/folks/hausmann/

Of course, not having a traditional academic path can definitely close[27] some doors, but in retrospect I can say it opened countless others. As my father once told me, "an academic degree is just another opportunity, nothing more, and nothing else."

When I look backwards, there are at least three things that have helped me to stay relevant, "marketable" one would say, during all these years.

The first thing was to learn a new programming language every year. Here is my current list, updated as of 2020:

- 1992: QBasic
- 1993: Turbo Pascal
- 1994: C
- 1995: Delphi
- 1996: JavaScript
- 1997: Java
- 1998: VBScript
- 1999: Transact-SQL
- 2000: C# & Prolog
- 2001: C++
- 2002: PHP
- 2003: Objective-C
- 2004: Visual Basic.NET
- 2005: Ruby
- 2006: LINQ
- 2007: Erlang
- 2008: Python
- 2009: Go
- 2010: Common Lisp
- 2011: Haskell
- 2012: Lua
- 2013: C++ 11
- 2014: Scala
- 2015: Swift
- 2016: Kotlin
- 2017: TypeScript & NASM
- 2018: F#
- 2019: Rust
- 2020: COBOL

Learning new programming languages did not only bring the obvious professional opportunities; it helped me find commonalities, pain points, and learn new and different toolkits. Usage of those programming languages has made me migrate from Windows (which I used from 1992 to 2006,) to the Mac (from 2002 to 2018,) and now to Linux, since 2014. I got out of my confort zone many times in my professional life, and I am still doing it. I migrated across galaxies[28] quite often. Last year I jumped from mobile app development to Kubernetes and cloud services. One of the best decisions I could make.

My second trick was to read. As much as I could, as often as I could, every year, I would read at least six books about programming. That is the magic number. An average of a book every

[27] https://deprogrammaticaipsum.com/tales-of-the-interview/
[28] https://deprogrammaticaipsum.com/the-developer-guide-to-migrate-across-galaxies/

2 months, with a strong proportion of them being about software history[29]. That is the big subject that I always look for in those books. How computers and programming languages came to be, who designed them, why, where. By the way, the HOPL IV conference[30] this year was postponed, but the papers are freely available online[31], and are a delight.

Last but not least, my third survival tip was to teach. Our friend Daniel Steinberg[32] once told me that teaching is the best way to learn. Maybe he borrowed this phrase from somebody else, but it does not matter; it is simply true. Explaining the same concepts over and over again helped me realize many things abour programming. One has to master the subject, and questions from students are the primary mechanism by which knowledge increases.

In the meantime I had the honor and the pleasure to guide many friends and acquaintances into becoming professional programmers, with many of them making a living out of that knowledge ever since. Although it is not the usual kind of achievements one features in a resumé, it is probably the thing I am the proudest of in my life.

I suppose I should finish this piece with some kind of advice for younger generations; what I can say is that, thankfully, we are living in the 21st century. Today there are many more options to classical University degrees than there were back in the 1990s. Online education is a reality, and it is not only affordable, but also (in general) of excellent quality. Some of you, like me, will enjoy it more than the classic campus life. Some of you, however, can learn easier in a classroom with a teacher, and that is perfectly fine. Know thyself, and find what works best for you.

We must stop considering a university degree as a requirement for working in our industry. It is not, it has not been so in a long time; we live in an unprecedented age of information at our fingertips, and we are undergoing a revolution in teaching. I look forward to hearing from the next Jean Piagets[33] and Seymour Paperts[34] of the world, to see how we can rethink[35] and redefine teaching, learning, and the corresponding assessment thereof.

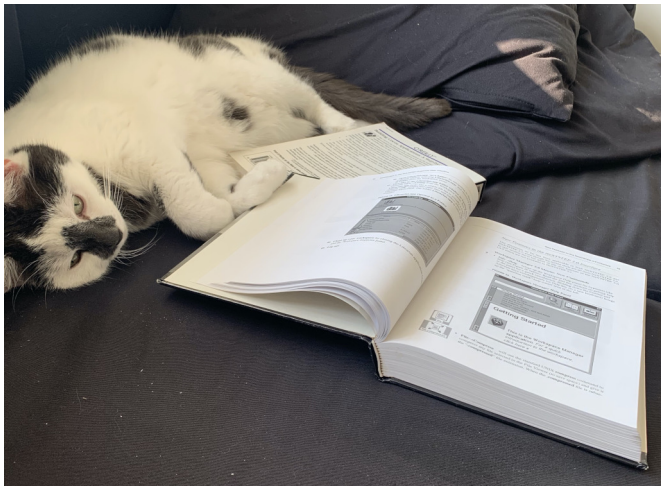
Cover photo by ian dooley[36] on Unsplash[37].

[29] https://deprogrammaticaipsum.com/history-is-a-fiction/
[30] https://hopl4.sigplan.org/
[31] https://dl.acm.org/toc/pacmpl/2020/4/HOPL
[32] https://deprogrammaticaipsum.com/at-least/
[33] https://en.wikipedia.org/wiki/Jean_Piaget
[34] https://en.wikipedia.org/wiki/Seymour_Papert
[35] https://inroads.acm.org/article.cfm?aid=3381026
[36] https://unsplash.com/@sadswim?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText
[37] https://unsplash.com/s/photos/freedom?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Simson L. Garfinkel & Michael K. Mahoney

Graham Lee

August 3rd, 2020



Let's start at the end. The last sentence in "NeXTSTEP Programming Step One: Object-Oriented Applications" by Simson L. Garfinkel and Michael K. Mahoney looks like this:

> Go out and write a killer app!

This is slightly punchier than the way the same authors signed off in "Building Cocoa Applications: A Step-by-Step Guide":

> Now go out and write a killer application!

I've still yet to do either (whatever your opinion on their products, the only real killer app authors are probably Dan Bricklin[1], John Warnock[2], Larry Page, Sergey Brin, and Mark Zuckerberg). But I have read to the end of many books on programming. It's quite an uplifting activity in the Objective-C community. The final sentence in the first edition of Aaron Hillegass's "Cocoa Programming for Mac OS X" is "Thanks for reading my book!", but it comes after this paragraph:

> Finally, try to be nice. Help beginners. Give away useful applications and their source code. Answer questions in a kind manner. It is a relatively small community, and few good deeds go forever unrewarded.

Having explored the backs of these books, let's go back to the beginning. When NeXTSTEP was launched, Object-Oriented Programming was new, and hard. The Smalltalk team already knew, as we previously saw[3], that it would be harder to teach existing programmers how to understand objects, than how to teach non-programmers how to use objects to program. The way Garfinkel and Mahoney describe it is almost paradoxical: "there's a steep curve to climb when learning to program in this easy-to-program environment".

[1] https://en.wikipedia.org/wiki/Dan_Bricklin
[2] https://en.wikipedia.org/wiki/John_Warnock
[3] https://deprogrammaticaipsum.com/adele-goldberg/

NeXT wasn't really a big platform, they maybe sold a total of 50k workstations and a large minority of them went to the same three-letter agency in the US federal government. So there wasn't a big market for third-party books about programming their system. In the beginning, NeXT had a training program, and a great set of manuals (a combination of in-house material and copies of the FSF documentation[4]), and a trade journal. Developer advocates would write letters to Dr. Dobb's Journal of Computer Calisthenics and Orthodontia, showing how much shorter code for NeXT was than examples in articles about other platforms.

An early "third-party publication" about writing NeXT applications turned out to be something else: the author, "Ann Weintz", was actually a NeXT employee called Tony who'd published the book under a pseudonym. This book talked up the platform's C heritage, trying to convince people that NeXTSTEP was more familiar than a first glance suggested. By the way, continuing the theme of interesting messages in books about Cocoa, the first page in this book is a call to home-school your children (at least in the second edition). The last is a collection of bullet points about low-energy refrigerators and political corruption.

It's into this environment that Garfinkel and Mahoney launched their book. What they wanted, and undoubtedly achieved, was to make it clear that the O-O features of NeXTSTEP made it *different*, and that those differences *simplified* application development, once you got your head around them. There are enough objects in here to make the live environment and object network construction in Interface Builder useful, but not enough to tie you in knots wondering whether your Dog isa Rectangle. When I say "object network construction", it was common in NeXT apps to create many of the objects in Interface Builder and hook them together there, using it as a dependency injection tool. The final step would be to generate the headers and source files, and type out the implementations of the methods.

NeXTSTEP Programming Step One showed developers—including me—enough of the APIs to make applications, but not so much that it was a slog to get through. It first introduced the environment and tools, showing you how to make an app using Interface Builder (Project Builder wasn't introduced until NeXTSTEP 3) and without, to demonstrate how much code is being abstracted away by the environment. Then it walked through building a four-function calculator, a mathematical word processor (MathPaper), and a graphing application (GraphPaper), introducing concepts like pasteboards, saving and loading documents, and custom graphics along the way.

Am I over-egging things? Not according to contemporary reviews[5]. "NeXTSTEP Programming is so thorough that it can be used as a reference book." "This is a well-written and illustrated book, and serves both as a primer for novice NeXTSTEP users as well as a handy reference for already initiated NeXTSTEP application developers." "I believe "NeXTSTEP Programming, Step One" will rapidly become the classic textbook for learning to program in what may soon become the standard GUI and object-oriented environ- ment for the 1990s."Programming book a must for all developers." "an unqualified success."

This 1993 textbook was so good that it when, a decade later, Apple wanted some third-party books for Mac developers, they turned to the authors again. With Mac OS X being new for both users and developers, Apple wanted to kick-start an ecosystem of training material to give people confidence in adopting the novel system and tools. They partnered with O'Reilly on a combination of first and third-party publications, with Building Cocoa Applications by Garfinkel and Mahoney being one of the guides to using the Apple APIs (alongside the

[4] https://deprogrammaticaipsum.com/the-community/
[5] http://simson.net/ref/1993/NextstepReviews.pdf

Apple-penned "Learning Cocoa"). Partnering with O'Reilly turned these into professional-looking manuals with a familiar theme, O'Reilly's Dover Pictorial Archive animal etchings on the covers. O'Reilly had already used big cats for their Java series so turned to dogs for the Apple Books.

Apart from the screenshots and the newer APIs, this was the same book as Step One. There's an introduction to the graphical environment and tools. There's an example application using Project Builder and Interface Builder, and an example that doesn't use Interface Builder. There's Calculator, MathPaper, and GraphPaper. As we've seen, they even ended with a more prolix version of the same sentence.

If you're in the business of making developer tools, get yourself a copy of this book (it's still effectively "in print", on O'Reilly's learning platform[6]). Find out why it was the best book for learning how to develop for two platforms by two vendors. And make sure the on-ramp for your technology is at least this good. If you need help, I can recommend a couple of names.

Go out and write killer docs!

Cover photo by the author.

[6] https://www.oreilly.com/library/view/building-cocoa-applications/0596002351/