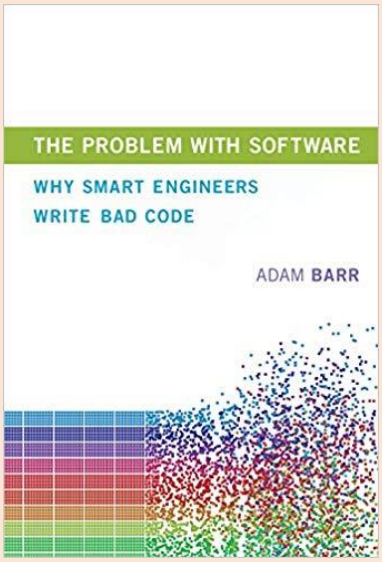
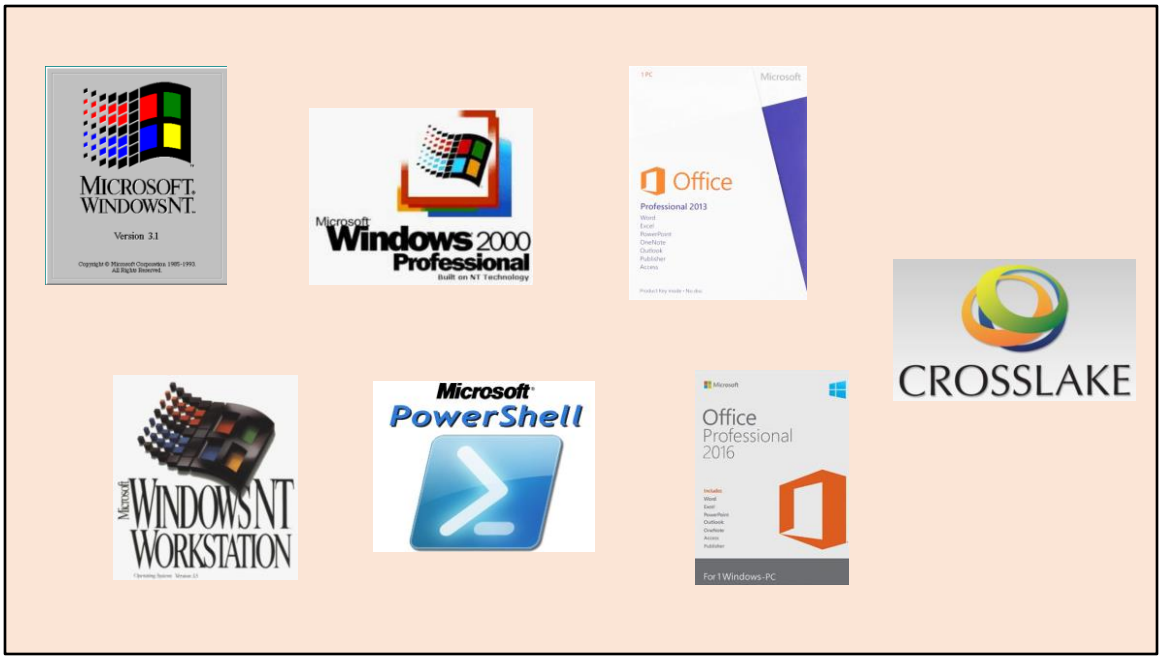


The subtitle is meant to avoid insulting software engineers, especially since I am a software engineer. I wanted to use something like “The 50-year quest to turn software engineering into real engineering”, but MIT Press does not like long subtitles.



My bio: I mostly worked for Microsoft, 23+ years a developer, dev lead, dev manager, engineering manager in Windows, Office, and a few other projects; also spent 5 years in Engineering Excellence as a trainer and consultant inside Microsoft, which is where I got interested in the process of software development; and also where I started to realize that lots of smart people had different ideas about how to develop software, and there was little empirical evidence available.

My first job out of college was at a startup writing laptop software for pharmaceutical salesmen; and I currently work as a consultant at Crosslake Technologies, primarily performing technical due diligence on potential acquisitions.

The term “software engineering” first appeared in mainstream usage in the title of a 1968 conference in Garmisch, Germany, organized by NATO. The attendees at the conference were both academics and industry people, and they all agreed that software engineering had problems. In 1969 they had a follow-up conference in Rome; it turned into a split between the academics and industry, a gap that has persisted to this day—fundamentally this is the problem with software.

I have heard it stated that Margaret Hamilton coined the term “software engineer”; she certainly was doing software engineering before 1968.

The Garmisch conference was notable for the **range of interests and experience** represented amongst its participants. In fact the **complete spectrum**, from the inhabitants of **ivory-towered academe** to people who were right **on the firing line**, being involved in the direction of really large scale software projects, was well covered. The vast majority of these participants found **commonality** in a widespread belief as to the **extent and seriousness of the problems** facing… "software engineering". This enabled a **very productive series of discussions**…to take place.

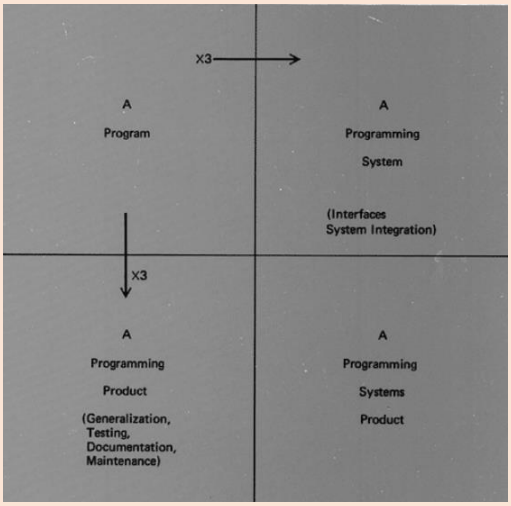
John Buxton and Bryan Randell, 1970

This is from Randell's page on the conferences at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html, which includes full copies of the reports. This (and the quote on the next page) are from the report on the 1969 conference (edited by Buxton and Randell), comparing the two.

Once again, a deliberate and successful attempt was made to attract an **equally wide range of participants**. The resulting conference bore **little resemblance** to its predecessor. The sense of **urgency** in the face of **common problems** was not so apparent as at Garmisch. Instead, a **lack of communication** between different sections of the participants became, in the editors' opinions at least, a dominant feature. Eventually the seriousness of this **communication gap**, and the realization that it was but a **reflection of the situation in the real world**, caused the gap itself to become a major topic of discussion. Just as the realization of the **full magnitude of the software crisis** was the main outcome of the meeting at Garmisch, it seems to the editors that the realization of the **significance and extent of the communication gap** is the most important outcome of the Rome conference.

John Buxton and Bryan Randell, 1970

This is from Randell's page on the conferences at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html, which includes full copies of the reports.



Fred Brooks (1931-) worked at IBM, including managing the development of the IBM System/360 computers, and later founded the computer science department at the University of North Caroline. This picture is from his Brooks's essay "The Tar Pit", from *The Mythical Man-Month* in 1975. It discusses the difference in complexity between what students work on in school or on their own (upper left quadrant) and what you work on in industry (bottom right). Software gets more complex in scope (top right) as it is created from the work of multiple teams, and more complex in time (bottom left) as the original developers are no longer working on it. Brooks claims each of these makes the software 3x more complicated, and the bottom right quadrant, where software is assembled from multiple teams and maintained by new teams, is 9x more complicated.

Even small companies have to worry about ongoing maintenance of software by somebody other than the original authors (bottom left).

It would be very nice if I could illustrate the various techniques with **small demonstration programs** and could conclude with "… and when faced with a program **a thousand times as large**, you compose it in the same way." This common educational device, however, would be **self-defeating** as one of my **central themes** will be that any two things that differ in some respect by a factor of already a hundred or more, **are utterly incomparable**."

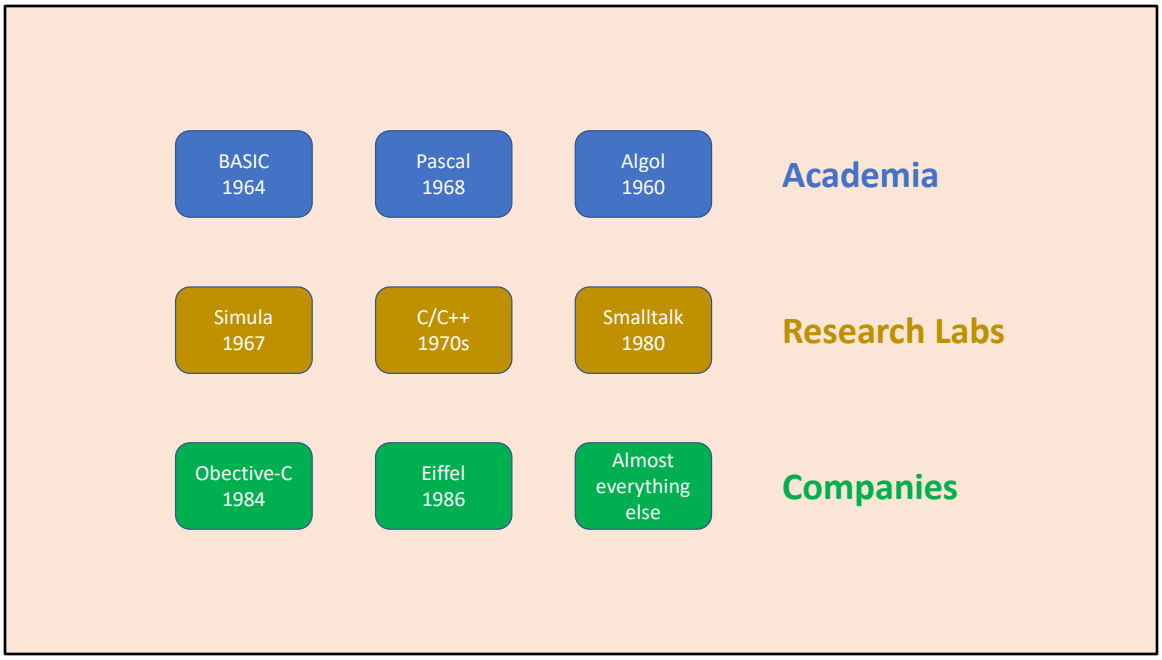
Edsger Dijkstra, 1972

Edsger Dijkstra (1930-2002) is a famous computer scientist and source of good quotes. He is saying that Brooks's 9x estimate for the top-left-to-bottom-right complexity increase is actually 100x.

"It is **characteristic** in software engineering that the problems to be solved by advanced practitioners require **sustained efforts** over months or years from many people, often in the **tens or hundreds**. This kind of mass problem-solving effort requires a **radically different** kind of **precision and scope** in techniques than are required for **individual problem solvers**."

Harlan Mills, 1980

Harlan Mills (1919-1996) was an IBM Research Fellow who also was on the faculty of various universities. He is also commenting on the difference between "programming in the small" and "programming in the large".



Although IBM was behind the creation of Fortran and PL/I, many of the first computer languages were developed at universities: BASIC was invented by two Dartmouth professors, Kemeny and Kurtz, Pascal by Wirth at the ETH in Zurich, and Algol by a committee of computer scientists. These were simpler days, where languages and the problems they solved were much less complex than they are today, and a better match to the capacity of a college professor.

After that we moved into an era where languages emerged from research labs, either private ones like the Norwegian lab that came up with Simula or more commonly research labs within larger hardware companies: C and C++ came from Bell Labs, and Smalltalk from Xerox PARC. These languages were invented, in various degrees, to support the company's business, but they were a by-product and not the end goal.

In the 1980s, there began to emerge languages designed by companies that were an end unto themselves: the company's business was the language, and the success of the company depended on programmers adopting a new language—always a difficult sell to programmers. Two of the first of these were Objective-C, invented by Brad Cox and Tom Love at a company called StepStone, and Eiffel, invented by Bertrand Meyer at a company called Eiffel Software.

Academia remained focused on the upper-left corner; it is no longer involved in topics that affect the bottom-right, such as creating new languages (or even making statements about languages that others create).

There is now a trend where languages are created by companies but not as ways to earn money, which at least avoids the incentive to over-hype the language, but they are still created to solve a specific problem without research into how broadly applicable they are.

The Lone Ranger used silver bullets to shoot his enemies, but the original use of the term was that silver bullets were required to kill werewolves.

"We hear desperate cries for a **silver bullet**, something to make software costs drop as rapidly as computer hardware costs do. But, as we look to the horizon of a decade hence, **we see no silver bullet**. There is no single development, in either **technology** or **management technique**, which by itself promises even one order of magnitude improvement **in productivity, in reliability, in simplicity**. … Not only are there **no silver bullets** now in view, the very nature of software makes it **unlikely that there will be any**."

Fred Brooks, 1986

Fred Brooks adopted the metaphor in his essay "No Silver Bullet".

- Structured programming
- Formal testing
- Object-oriented programming
- Design patterns
- Unit tests
- Test-Driven Development
- Agile
- Functional programming
- DevOps
- DevSecOps

This is a list of silver bullets over the years; there has been a trend away from academically-studied concepts:

- Structured programming had a lot of academic research behind it
- Formal testing a bit less so
- OO and design patterns not so much
- Unit tests/TDD and agile were where silver bullets became potential ways to earn money

"Every **Turing machine** is **reducible** into, or in a determined sense is **equivalent** to, a program written in a **language** which admits as **formation rules** only **composition** and **iteration**."

Corrado Böhm and Giuseppi Jacopini, 1966

From the early days: The Böhm-Jacopini theorem proved that any program could be represented without gotos—a contribution to structured programming from two academics.

"Program testing can be used to show the **presence** of bugs, but never to show their **absence**."

Edsger Dijkstra, 1969

"It is well known that you **cannot test reliability** into a software system."

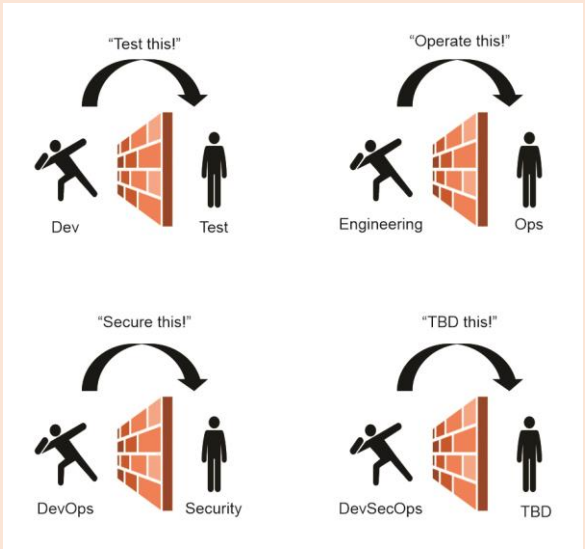
Harlan Mills, 1976

Later, you have an academic (Dijkstra) and an industry practitioner (Mills) agreeing on the problems with formal testing as a silver bullet. Despite this, of course, companies in the 1980s and 1990s spent a lot of time trying to prove correctness via formal testing.

"XP/Agile represents an **alternative** to software engineering. Further, XP/Agile challenges the **assumptions**, the **core values**, the **worldview** upon which software engineering is predicated…Object thinking focuses our attention on the **problem space** rather than the **solution space**. Object thinkers take the advice of Plato, David Parnas, Fred Brooks, Christopher Alexander, and many others by letting the **problem define its own solution**…This perspective shapes the software developer's thoughts **constantly and pervasively**."

[name withheld], 2004

This is what the silver bullets became by the 2000s—attempts to sell books and/or training, with no grounding in any systematic studies of real software development. The book this is taken from goes on in this vein for ~300 pages.



Note that the new ideas are still useful!

The move from separating dev/test, to engineering/ops, to DevOps/security, to DevSecOps and whatever else they will not consider when doing their work, is hard-earned wisdom and is an improvement, but they are not silver bullets.

Back in the 1970s, people studied programmers empirically.

Commenting style - Ben Shneiderman, 1980

Variable names – Larry Weissman, 1974

Indenting – Tom Love and Ben Shneiderman, 1977

GOTO - Max Sime, Thomas Green, and John Guest, 1973; Henry Lucas and Robert Kaplan, 1976

These were studied in controlled experiments back in the day.

"In preparing my retrospective and update of ***The Mythical Man-Month***, I was struck by **how few** of the propositions asserted in it have been **critiqued, proven, or disproven** by ongoing software engineering **research and experience**."

Fred Brooks, 1995

The original version of "The Mythical Man-Month" came out in 1975. This is from the introduction to the 20th anniversary edition.

"Most of these attack the central argument that **there is no magical solution**, and my clear opinion that there **cannot be one**. Most **agree** with the arguments in 'No Silver Bullet,' but then go on to assert that **there is indeed a silver bullet** for the software beast, which **the author has invented**."

Fred Brooks, 1995

In other words, his critics think they are the ones that know what is going on, and everybody else is clueless.

And yet the "silver bullets" did contribute valuable insights!

- GOTO is bad, testing can help, OO/design patterns/unit tests allow refactoring
- Agile at least recognizes reality and gets development teams closer to the customer
- Functional programming is useful in some situations
- DevOps and DevSecOps can allow the entire delivery pipeline to operate more efficiently

"Since there was no **mathematical rigor** to inhibit these discussions, some became quite **vehement**."

Harlan Mills, 1988

Mills was discussing his theory of "top down programming", which he attempted to prove mathematically, but discovered that people arguing against him were simply making common-sense arguments based on their personal experience.

The image is from an episode of "Silicon Valley", where Richard breaks up with his girlfriend because she uses spaces in her code instead of tabs.

"**Coding style wars** are a waste of **valuable resources**, although the **confusion** caused by Hungarian probably **wastes more time**."

Unidentified Windows NT 3.1 developer, ~1992

The quote is from the book *Show Stopper!: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft*, by G. Pascal Zachary

Hungarian notation is a variable-naming convention that uses short prefixes at the front, such as szName with the "sz" prefix indicating a zero-terminated string. https://en.wikipedia.org/wiki/Hungarian_notation

In any case, the vast majority of Twitter users will not be able to make use of the open-source code, because code is typically hard to read even by those who have written it. Trust me.

This is from the May 2, 2022 article “Sorry Elon, ‘Open Source’ Algorithms Won’t Improve Twitter” by Cathy O'Neil in the Washington Post: https://www.washingtonpost.com/business/sorry-elon-open-source-algorithms-wont-improve-twitter/2022/05/02/362a6594-ca10-11ec-b7ee-74f09d827ca6_story.html

It is just assumed that code will be hard to read.

So, what happened to get us to this point? Why did academia and industry never get back in sync, despite the clear issues with software engineering?

"More interesting, however, was the coincidence that all of them had **learned to program** *before* they **studied programming formally** in school. That's a major change brought about by the personal computer. In my day, **I had not even *seen* a computer** before I went to work for IBM in 1956."

Gerald Weinberg, 1998

Gerald Weinberg (1933-2018) worked at IBM, including on Project Mercury, and then became a consultant and author. He is discussing a team he observed in the mid-1990s. He calls it a coincidence, but it's not; it's precisely the availability of personal computers that allowed people to teach themselves programming on their own, away from the guidance of experienced programmers.

This is also one of the main reasons that the number of female programmers began to drop around the same time.

This is Paul Allen and Bill Gates using a computer in high school. They were instrumental in popularizing BASIC, that is what Microsoft was founded for.

BASIC became the dominant language on the PC, and therefore everything, because those systems needed a language that could run on computers with 1K, 2K, 4K of memory. You had to have a small interpreted language.

The net effect was that everybody could teach themselves to program, and worse, they were teaching themselves to program in BASIC, at the time an unstructured language.

I had a similar experience to Gates and Allen in high school, although they were using BASIC about 10 years before me.

"It is practically **impossible** to teach good programming to students that have had a **prior exposure to BASIC**: as potential programmers they are **mentally mutilated** beyond hope of regeneration."

Edsger Dijkstra, 1975

This is not just "another Dijkstra quote"; he was attempting to steer programmers away from BASIC (and COBOL and Fortran) to structured languages such as Algol; unfortunately 1975 is the same year Microsoft was founded, and BASIC became the default language for personal computers because it could run in such small memory sizes.

I once had an argument with my sister, when she was taking a programming class her freshman year of college and I was still a self-taught BASIC high school programmer, about how things like local variables in functions really didn't make programs any clearer.

Dijsktra's essay was titled "How do we tell truths that might hurt?" and included this:

•FORTRAN --"the infantile disorder"--, by now nearly 20 years old, is hopelessly inadequate for whatever computer application you have in mind today: it is now too clumsy, too risky, and too expensive to use.
•PL/I --"the fatal disease"-- belongs more to the problem set than to the solution set.
•It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are

mentally mutilated beyond hope of regeneration.
•The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.
•APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.

"Our present programming courses are patterned along those of a **"course in French Dictionary."** We study the dictionary and learn what the meanings of **French** words are in **English** (that corresponds to **learning what PL/I or Fortran statements do to data**)…we then invite and exhort the graduates to **go forth and write French poetry**. Of course, the result is that **some people can write French poetry** and some not, but the **skills critical to writing poetry were not learned** in the course they just took in French dictionary."

Harlan Mills, 1972

This has not changed much since Mills wrote this 50 years ago.

There may actually be enough required learning as a developer that you need 4 years of college. But it's almost certainly not what top CS schools teach developers today. To be fair, a lot of schools do offer a course that discussed industry development (e.g. "CSC 326: Software Engineering" at NC State or "17-346 DevOps and Continuous Integration" at Carnegie-Mellon) but this is one course among many.

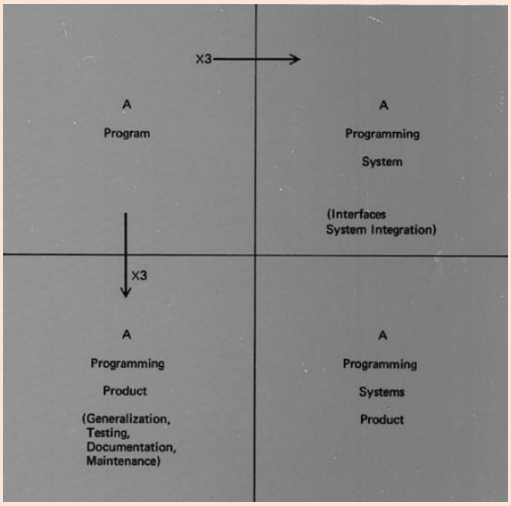
Yes, academic status is based on publication not teaching effectiveness, this is an old issue…but it is particularly bad in CS where the publications have so little to do with what industry does.

"If the **precision** and **scope** are not gained in **university education**, it is difficult to acquire them later, no matter how well **motivated** or **adept** a person might be at **individual, intuitive approaches** to problem solving."

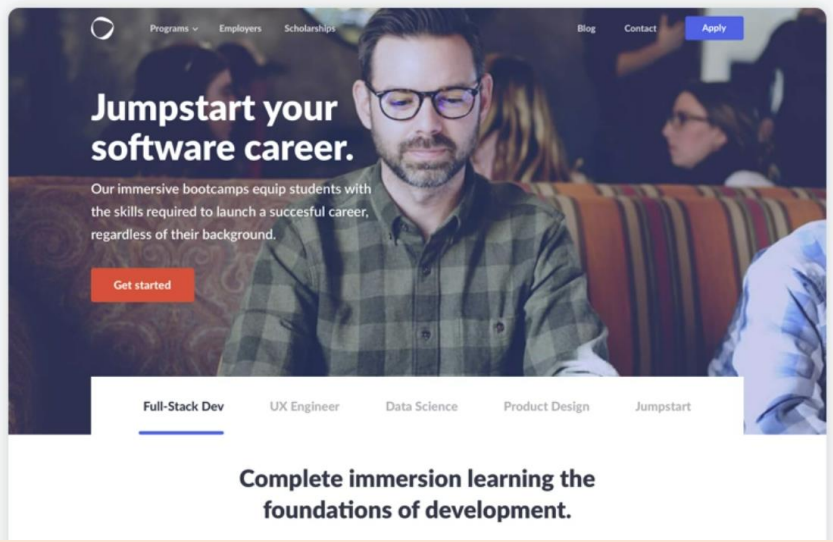
Harlan Mills, 1980

In university I wasn't "untaught" what I believed to be true about BASIC. I was taught algorithms, which is an orthogonal concept that generally doesn't require local variables, data abstraction, clean APIs, etc.

When I gave an earlier version of this talk, an attendee who was a recent graduate commented that they were told at school that they weren't being taught what they needed to know in industry.



The universities are still focused on the top-left quadrant—that is the scope and timeline that a class can cover, and what a professor and a few graduates students can't cover in their own projects.



The image is from: https://dribbble.com/shots/4556413-Marketing-Website-Coding-Bootcamp

Why do some people advise “Get a degree in something else and then go to coding camp”? Or even “Get a degree in CS and then go to coding camp”?

College mostly remain focused on the “programming in the small” topics, in particularly algorithms and specific languages, which has led to the rise of boot camps, which teach the specific skills that industry wants. And they cover (to varying degrees, it’s true) topics that matter in industry such as unit testing, defining APIs, application security, and using source code repositories.

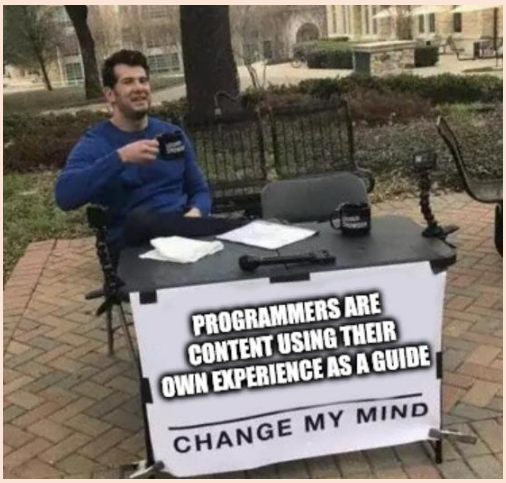
Coding camps do teach algorithms, but as a gate to get past in the interview, not something actually useful.

Given the hands-on nature and long hours of coding camps, I estimate a 3-month coding camp would give me about 50% of the programming time that I had in 4 years of majoring in Computer Science.

Things you may not know as much about as you would like

- What **programming language** to use
- What to look for in **code reviews**
- What to look for when **hiring**
- How **reliable** your software (or an open-source component) is
- Should we **extend** this software, or write something new?
- Is your software about to **fail**
- Is your software **obsolete**

These are the sorts of things you expect to be able to answer as a “real” engineer.



Programmers are too content with the current state—they may not know some of the things on the previous slide, but they are successful in their careers, so why mess with success.

Enrollment in college CS programs is growing rapidly, so they have no incentive to change.

"Experience is a **dear teacher**, but fools will learn at **no other**."

Benjamin Franklin

One of the most insightful quotes I heard at Microsoft was from a developer who used to work on packaged software such as Office, and had spent two years working on a large managed service in the late 2000s, gradually learning how different they were. After sharing wisdom on this, he said, "If I went back in time and told myself this, I wouldn't believe it".

"Another essential personality factor in programming is at least a small dose of ***humility***. Without humility, a programmer is **foredoomed** to the classic pattern of Greek drama: success leading to overconfidence (***hubris***) leading to blind **self-destruction**. Sophocles himself could not have invented a better plot (to reveal the **inadequacy of our powers**) than that of the programmer learning a few **simple techniques**, feeling that he is an **expert**, and then being **crushed** by the **irresistible power** of the computer."

Gerald Weinberg, 1971

Weinberg states things a bit dramatically, but the key is that you need to be aware of the gaps in your knowledge before you can begin to mitigate them.

The legend of Icarus: He made wings of feathers attached with wax, but flew to close to the sun, the wax melted, and he fell in the water and drowned. This is the classic story of hubris.

I thought I was the **hardest-working person** on the planet. I thought we were the **hardest-working industry**. That's what we tell ourselves. It's all **malarkey**. I've had this front-row seat over the last three years to **greatness**. It's a **humbling experience**…seeing just what it takes to **actually be that great**."

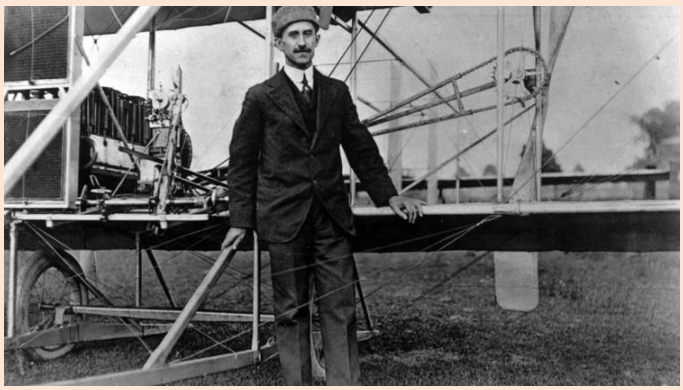
Alexis Ohanian, 2018

Ohanian was one of the co-founders of Reddit; he is observing his wife, the tennis player Serena Williams.

"In the past twenty-five years a **whole new data processing industry** has exploded into a critical role in business and government. Had this hardware development been spaced out over **125 years**, rather than just **25 years**, a **different history** would have resulted. For example, just imagine the **opportunity** for orderly industrial development with **five human generations** of university curriculum development, education, feedback for the **expansion of useful methodologies** and **pruning of less useful topics**."

Harlan Mills, 1976

It is now a few generations later, but the new crop of "old timers" (the ones who taught themselves BASIC in high school) are still around. So what does this mean for the software industry?



Many “old timers” are still around the industry, people who were self-taught in the BASIC days. It’s as if Orville Wright were still at Boeing.

Old-school people don’t appreciate the value of (and the time it takes to perform) code reviews, static analysis, writing unit tests, etc.

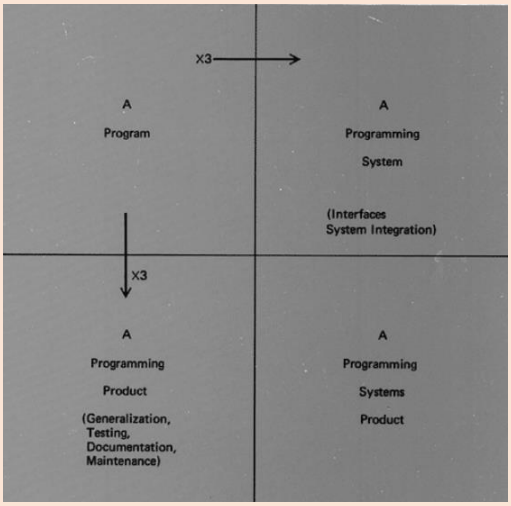
What they are focused on is performance.

"There is no doubt that the **grail of efficiency** leads to **abuse**. Programmers **waste enormous amounts of time** thinking about, or worrying about, the speed of **noncritical parts** of their programs, and these attempts at efficiency actually have a **strong negative impact** when **debugging** and **maintenance** are considered. We should forget about **small efficiencies**, say about 97% of the time: **premature optimization** is the **root** of all **evil**."

Donald Knuth, 1974

Donald Knuth (1938-) is a computer science professor possibly best-known for his books on algorithms, but also a keen observer of industry development.

Hero programmers love thinking about performance. Because it was critical back then!

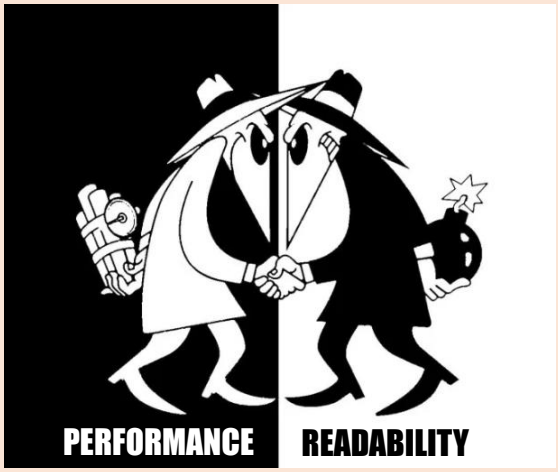


Also, it is easy to measure improvement locally, even on a small piece of code written by one person.

"**Hell** is **other people**."

Jean-Paul Sartre, 1944

The quote is from the play "No Exit". Performance can be measured and improved on your own. It's a perfect top-left quadrant problem. Worrying about maintainability is a bottom-right problem which requires interacting with others to figure out if your code can integrate with theirs and/or is maintainable.



Performance is the enemy of readability (despite some personal example you may have of a more performant algorithm that was also easier to read).

Most exploits are due to people skipping coding steps or checks for performance reasons. Or using a language that is inappropriate because they want things to run faster.

"**Optimization** is the **root** of all **evil**."

Adam Barr, 2022

Removing the "premature" from Knuth's quote. This doesn't mean you never do it—just like "Money is the root of all evil" doesn't mean you never use money.

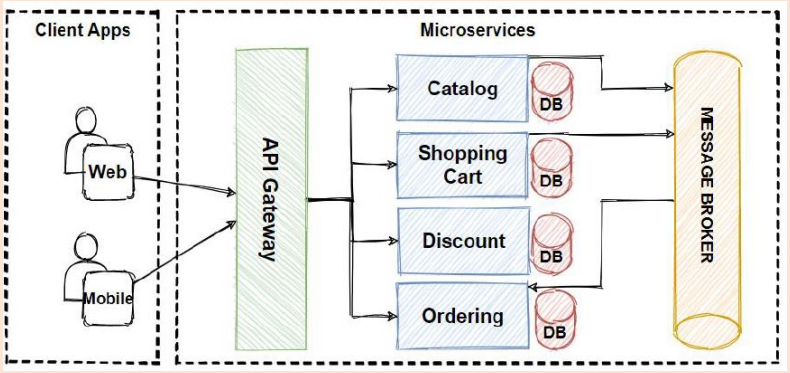
Can things just be like the happy early days when you were first programming and we could live in the top-left quadrant? In a word: No.

Microsoft made fun of IBM when working together in the late 1980s…but IBM had forgotten more about programming than Microsoft knew. Eventually Microsoft re-learned all of it. Except counting lines of code to measure productivity, that was a bad idea. But at least IBM was trying to measure productivity! And to be honest, Microsoft rewarded the “crank out lots of code” people also, it just wasn’t the only thing they looked for.

"The next generation of programmers will be **much more competent** than the first ones. **They will have to be**. Just as it was easier to get into **college** in the 'good old days,' it was also easier to get by as a **programmer** in the 'good old days.' For this **new generation**, a programmer will need to be capable of **a level of precision and productivity never dreamed of before**."

Harlan Mills, 1978

The languages and tools have changed, but what Mills and Brooks and Dijkstra were writing about in the 1970s are the same problems facing software today, but we have drifted further from empirical evidence.



This picture is from Mehmet Ozkaya's article "Microservices Architecture" https://medium.com/design-microservices-architecture-with-patterns/microservices-architecture-2bec9da7d42a.

In particular, auto-scaling microservices solve a lot of problems (but not all). Clean separation, language-independent API that can be logged and inspected. And they remove any need for premature optimization because in the short-term you can spend money to handle load, and make code changes only if needed.

And eventually, client software will be written this way also!

You can (almost) make software “twice as strong”—the kind of thing that engineering can guarantee.

The image is from https://curio.io/stories/7Jxoimob79I26rametoH7N.

People may say, “I am in a hurry to get my startup/MVP version to market and I don’t have time for unit tests etc”.

I see this when doing due diligence on acquisitions. People have taken shortcuts to get going and then they get stuck with tech debt. The people who do it right from the start don’t notice the extra work required, it is just part of what they do each day.

At some point this will become the equivalent of checking in code that doesn’t compile—it is simply not acceptable. Test coverage results will be a part of demonstrating that code has been developed with real engineering practices, anything without it won’t be considered acceptable to take a dependency on.

Civil engineers don’t say “It’s just a little bridge to start, I don’t have to worry if it will fall down”.

"**Short cuts** makes **long delays**."

Peregrin Took, 3018 T.A.

This is my advice—you have to do all of those things that you may not want to do, that you didn't have to do when programming in the top-left quadrant: code reviews, unit tests, static analysis, etc.

You do have to worry about designing your API as cleanly as possible, not just from your point of view but from your caller's point of view.

Empirical studies of programmers are making a comeback!

Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. “Syntax Errors Just Aren’t Natural: **Improving Error Reporting** with Language Models”. Working Conference on Mining Software Repositories (MSR), 2014.

Baishakhi Ray, Daryl Posnett, Vladimir Filkov, Premkumar Devanbu. “A Large Scale Study of **Programming Languages and Code Quality** in Github”. ACM Foundations of Software Engineering (FSE), 2014.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. “Learning to **Generate Pseudo-Code from Source Code** Using Statistical Machine Translation”. International Conference on Automated Software Engineering (ASE), 2015.

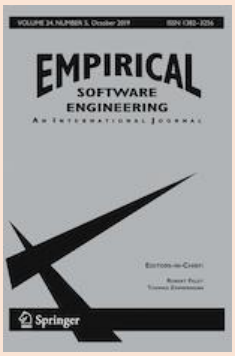
Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. “On the **‘Naturalness’ of Buggy Code**”. International Conference on Software Engineering (ICSE), 2016.

Sahil Bhatia and Rishabh Singh. “**Automated Correction for Syntax Errors** in Programming Assignments using Recurrent Neural Networks”. International Conference on Software Engineering (ICSE), 2018.

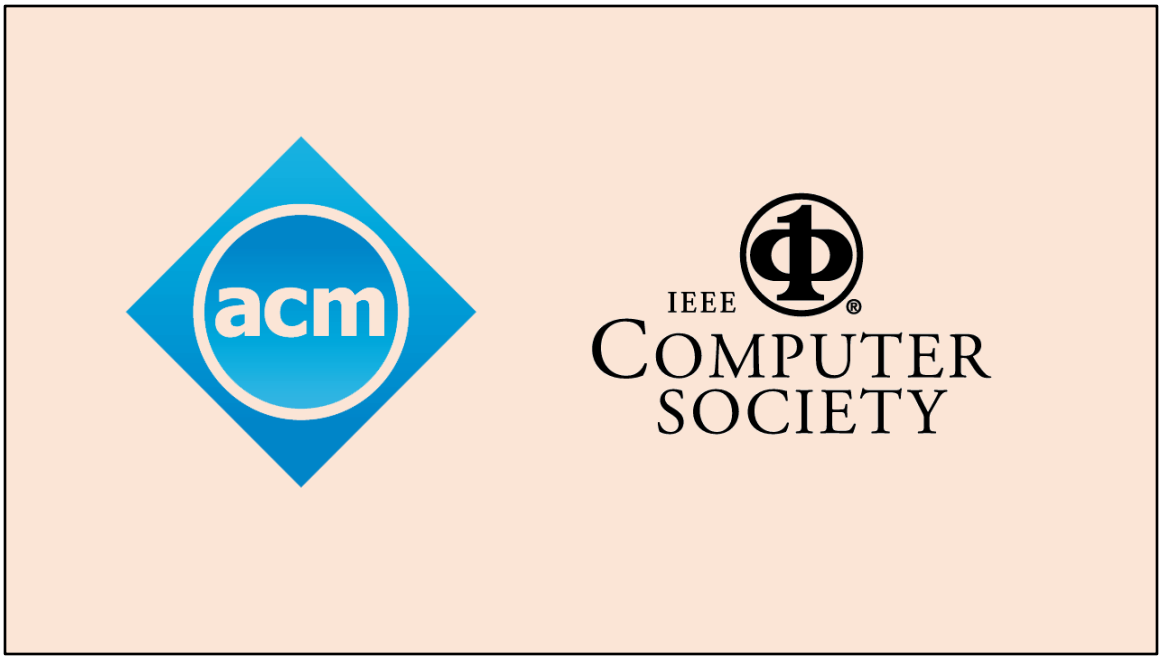
Casey Casalnuovo, Kevin Lee, Hulin Wang, Prem Devanbu, Emily Morgan. “Do Programmers Prefer **Predictable Expressions in Code**?”. Cognitive Science, 2020.

A sample of recent papers. Unfortunately this rarely crosses over to industry development. Even at Microsoft, which has a Research team that is heavily involved in empirical studies, the teams working on products do not pay attention to that.

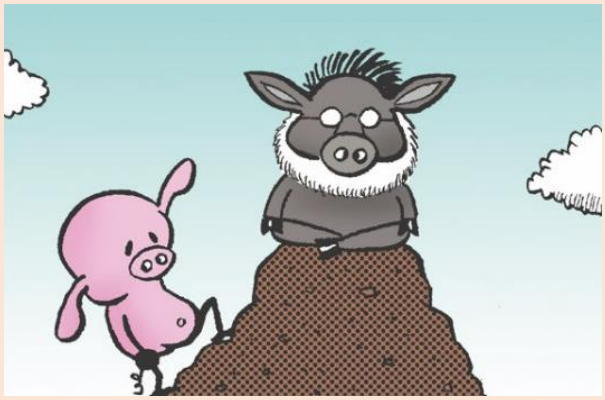
Conferences that focus on (or include) empirical studies:
ESEM (Empirical Software Engineering and Measurement): https://www.esem-conferences.org/
ICSE (IEEE International Conference on Software Engineering): https://conf.researchr.org/home/icse-2022
ICPC (IEEE International Conference on Program Comprehension): https://conf.researchr.org/home/icpc-2022
MSR (Mining Software Repositories): https://conf.researchr.org/home/msr-2022



Journals that focus on (or include) empirical studies:
"Empirical Software Engineering": https://www.springer.com/journal/10664
"Software: Evolution and Process": https://onlinelibrary.wiley.com/journal/20477481
"ACM Transactions on Software Engineering and Methodology":
https://dl.acm.org/journal/tosem
"IEEE Transactions on Software Engineering":
https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=32



I encourage people to get involved with these organizations.



Some advice! The image is from “Pearls Before Swine”, 6/26/20.

IF YOU ARE A **MANAGER**:

Make sure you **allow your employees the time** to following modern software development practices.

Reward them for doing it, not for “hero” behavior to fix issues because this was skipped earlier.

If you feel “I don’t have any wisdom to share with my team”—then try to **become wiser**.

IF YOU ARE A **SENIOR DEVELOPER** OR **ARCHITECT**:

Establish some **mandatory process** for the team, such as unit tests and static analysis.

Don’t believe every silver bullet, but don’t ignore them either. They are **valuable**, but they are **not magic**.

Read **empirical studies papers**, not trendy books on new processes.

And don’t be the grumpy old **performance-focused developer**.

IF YOU ARE A **DEVELOPER LOOKING TO IMPROVE**:

Contribute to open-source projects, ideally ones with a lot of contributors that have **established rules and processes**.

If you don't have time to contribute, **read the code**, trying to understand why the rules and process exist.

IF YOU ARE A **DEVELOPER RIGHT OUT OF COLLEGE**:

Thank you for attending an ACM talk!

You may feel an odd mix of **imposter syndrome** and **knowing more than your co-workers**, depending on your background.

But you likely have a lot to learn from working with **whatever development process** you land in, because it will be more **"bottom-right quadrant"** than what you have done before

IF YOU ARE **INTERVIEWING DEVELOPERS**:

Don't focus only on "**top-left quadrant" topics**. A coding question is fine, but don't worry too much on the **specific algorithm** the candidate comes up with.

Even if people have to come up with clever algorithms…it is much more important that the code be **readable and maintainable** than be "clever". If it's too clever, the next person to modify the code **will likely break it**.

Ideally, the candidate would have some significant experience working with **other people's code** to talk about; but at least talk about **testing and code maintainability.**

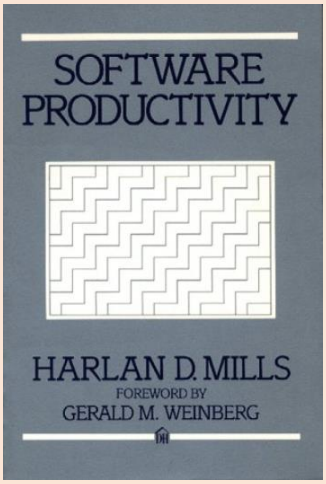
IF YOU ARE **TEACHING DEVELOPERS**:

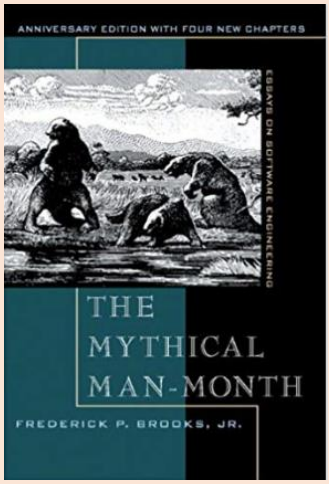
Have your students **read more code**—there is a lot of open source available.

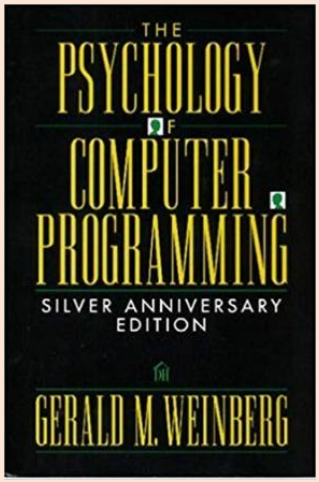
Much of software engineering training should be based on **studying code**, not **studying algorithms and language syntax**. You are going to produce code, you should study code. Good code, bad code, successful code, code that failed.

Look at **empirical studies** of industry developers; and ideally, conduct some of your own.

And don't ignore **coding camps**. They may become your competition.







And read these books! Dijkstra has some worthwhile books also (“A Discipline of Programming”, “Selected Writings on Computing”), but they mix interesting ideas about programming with proofs of basic algorithms, so there isn’t one particular one to recommend.


THANK YOU FOR LISTENING
ANY QUESTIONS
THANK YOU FOR LISTENING
ANY QUESTIONS
THANK YOU
ANY QUESTIONS?
THANK YOU FOR YOUR ATTENTION
PLEASE FEEL FREE TO ASK ANY QUESTIONS
THANK YOU FOR YOUR ATTENTION
ANY QUESTIONS? PLEASE ASK
THANK YOU FOR YOUR ATTENTION
ANY QUESTIONS?
@AdamDavidBarr