

DPI

*De* Programmatica *Ipsium*

DE PROGRAMMATICA IPSUM

---

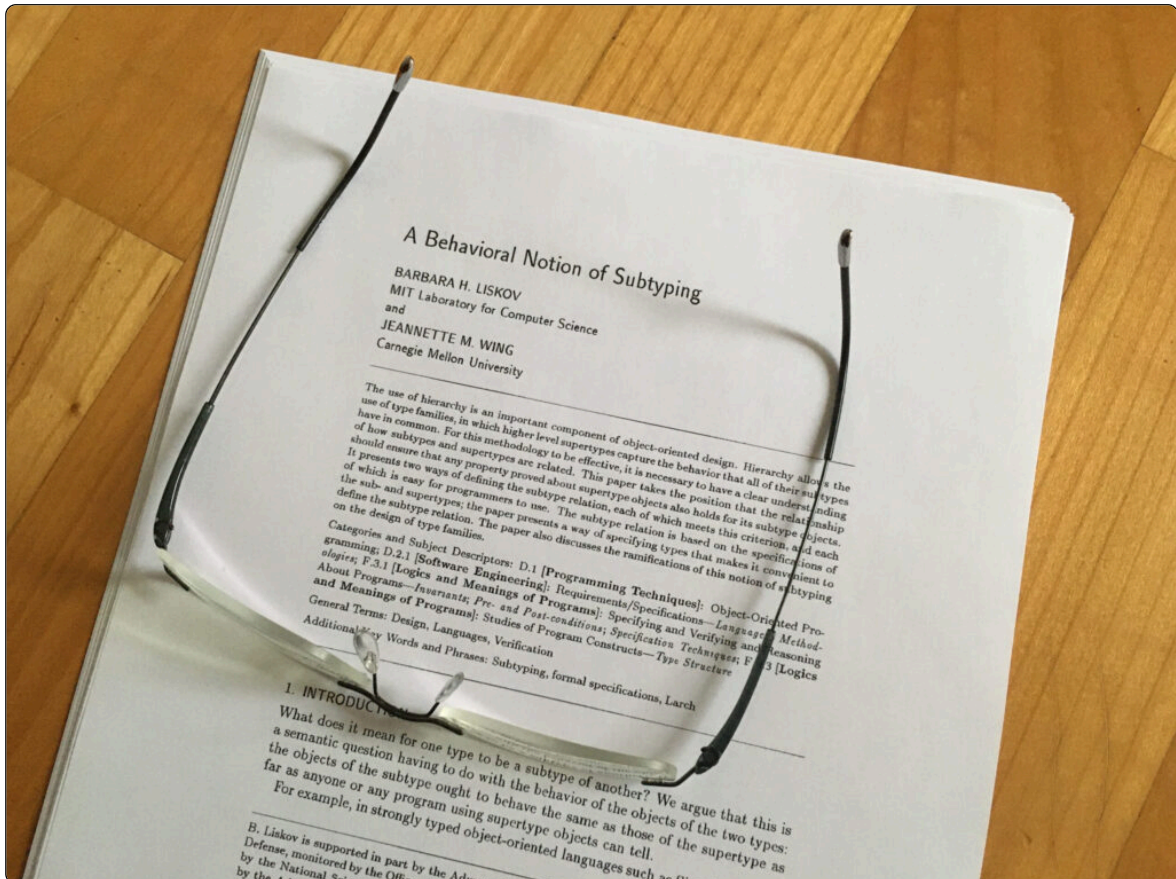
# Papers

# Table of Contents

Barbara Liskov .....	5
Barry Boehm .....	11
Edward Nash Yourdon .....	17
Sir Tony Hoare .....	23
A Review Of Research Around Programming Education From The 1960s To Today .....	29
J. C. R. Licklider & M. Mitchell Waldrop .....	43
Ulrich Drepper .....	51
Joanna Rutkowska .....	57
Loren Carpenter .....	65



# Barbara Liskov



By Graham Lee, February 3rd, 2020

Many developers will have heard of Barbara Liskov, through her appearance in Robert C. Martin's SOLID list of design principles. The abstract of her 1994 paper with Jeanette Wing, A Behavioral Notion of Subtyping<sup>1</sup>, makes the principle sound easy in, well, in principle:

*This paper takes the position that the relationship [between supertype and subtype] should ensure that any property proved about supertype objects also holds for its subtype objects.*

Then it becomes a little more opaque when written in the body of the paper:

*Let  $\phi(x)$  be a property provable about objects  $x$  of type  $T$ . Then  $\phi(y)$  should be true for objects  $y$  of type  $S$  where  $S$  is a subtype of  $T$ .*

People often stop here, which doesn't really give enough information to effectively apply the principle, though it certainly gives enough information to sagely quote an academic "principle" while trashing someone in a code review. Such treatments are unfair representations of a 31-page paper which gives two different descriptions of the relationship between supertype and subtype, along with a discussion of which is preferable and why. Computer programming is a complex activity, and sometimes its ideas need space to be conveyed well.

The first (and preferred, according to Liskov and Wing) formulation says this. Imagine that a type is defined with a Bertrand Meyer<sup>2</sup>-style contract, explaining the type's name, values, and its methods along with their preconditions and post-conditions. In addition to *invariants* – predicates that are always true whatever methods are applied to a type, there are *constraints*: predicates that restrict the historical relationship between states of a type instance. A constraint might be something like:

*for all stacks  $s$ ,  $s.size > 0$ .*

Now, no matter how much you call `s.pop()`, you're never going to get a stack of size  $-1$ .

The subtype property says that  $S$  is a subtype of  $T$  if these things hold, for all  $s$  that are members of the subtype  $S$ :

1. the invariants of  $S$  applied to  $s$  imply the invariants of  $T$  applied to ( $s$  treated as a member of  $T$ ).
2. there is a method on  $S$  with the same number of arguments as a corresponding method on  $T$ , where:

– each argument in the  $T$  version is a subtype of the corresponding argument in the  $S$  version; – the return value of the  $S$  version is a subtype of the return value of the

T version, or neither method returns a value; – the set of exceptions raised by the S version is a subset of the exceptions raised by the T version; – the preconditions of the T version applied to (s treated as a member of T) imply the preconditions of the S version; – the postconditions of the S version imply the postconditions of the T version applied to (s treated as a member of T).

3. the constraints of S applied to s imply the constraints of T applied to (s treated as a member of T).

Notice that the methods don't need to have the same name on both types! If you have a renaming operation such that “s treated as a member of T” maps all of the T method names onto the S method names, and otherwise the rules hold, then you still have a subtype. Also there's no mention of inheritance. One of the biggest reasons programmers using OOP get into a pickle is using inheritance for multiple purposes<sup>3</sup>. In a Smalltalk program, inheritance is simply a way to access method implementations from one class in another class. In Java, the compiler enforces properties that are an incomplete subset of the subtype rule, so inheritance is kinda code reuse and kinda subtyping, and hilarity ensues. Particularly if you use Java's arrays anywhere in a method signature.

One of the things that makes Liskov such an influential author is that she believes that academic software engineering needs to be applicable to practice to be of any value. As she said in *Barbara Liskov on Programming, Career, and the Future*<sup>4</sup>:

*In my research area of software systems, it's also very important to reduce ideas to practice. This means that the solutions I invent need to be “sufficient” to solve the problem; they should be as simple as possible, but the system has to really run, and it has to run with good enough performance.*

So she is also co-author, with John Guttag, of *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*<sup>5</sup> in which a practical, modular approach to designing software through specification is expounded. Yes, the subtype principle is given nine pages.

While this article has focussed on Barbara Liskov's contribution to type theory and program design, she's had a varied career working on programming languages,

## PAPERS

concurrent systems, error correction, and information security. People are still debating the principles outlined in the 1994 paper on subtypes, and in her words (the IEEE interview again):

*Actually, I think research does have an impact on industry. The problem is that the time lag can be quite long.*

So look out for more articles in *De Programmatica Ipsum* in the 2040s about how Barbara Liskov revolutionised distributed systems theory.

Cover photo by Adrian Kosmaczewski.

## REFERENCES

<sup>1</sup> <https://doi.org/10.1145/197320.197383>

<sup>2</sup> <https://deprogrammaticaipsum.com/bertrand-meyer/>

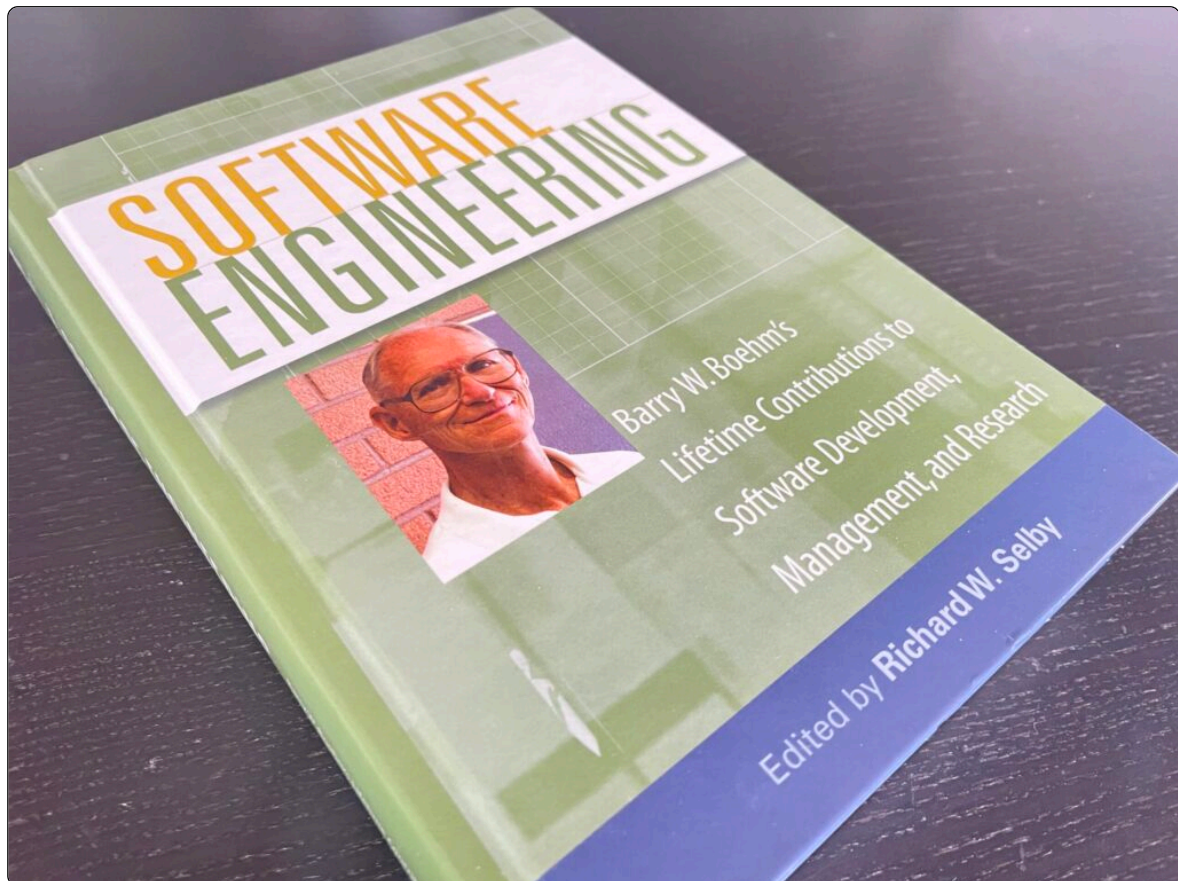
<sup>3</sup> <https://www.sicpers.info/2018/03/why-inheritance-never-made-any-sense/>

<sup>4</sup> <https://doi.ieeecomputersociety.org/10.1109/MDSO.2005.8>

<sup>5</sup> [https://openlibrary.org/books/OL7408379M/Program\\_Development\\_in\\_Java](https://openlibrary.org/books/OL7408379M/Program_Development_in_Java)



# Barry Boehm



By Adrian Kosmaczewski, December 5th, 2022

We have often talked about software economics in this magazine. For example, when we enumerated Eric Sink's<sup>1</sup> perspectives on the software business, discussed platforms<sup>2</sup> as a paradigm for economic analysis, or talked about how Brad Cox<sup>3</sup> advocated for an object-oriented economy. But there is a more extraordinary author about the subject, one we mentioned a few times in this magazine and who sadly passed away last August: Barry Boehm<sup>4</sup>.

Barry Boehm started his career in software engineering in 1955, and his experience led him to analyze the economic forces shaping software engineering in business, government, and society.

The 800-page book “Software Engineering: Barry W. Boehm’s Lifetime Contributions to Software Development, Management, and Research”<sup>5</sup> by Richard W. Selby, published by Wiley in 2007 together with the Computer Society, contains a compilation of 43 influential papers published between 1973 and 2006.

Barry Boehm wrote about software quality, economics, tooling, risk management, and project management in those papers. He was a pragmatist; he wanted to find valuable solutions to common problems and provide software project managers with tools to implement those solutions.

Among the various inventions of Barry Boehm, we will find the COCOMO software project estimation models he developed in the 1980s and 1990s. I mentioned them in a previous article<sup>6</sup>:

*In his 1981 paper titled “Software Engineering Economics”<sup>7</sup> he went as far as to propose a formula to evaluate the cost of software projects. You heard right; this is an algorithmic method for software cost estimation (you can rub your eyes, I will wait for you); that is, a mathematical formula that could have been implemented in Excel 35 years ago. And, lo and behold, SLOCs are at the basis of the whole equation.*

*This rather revolutionary contraption was called the COCOMO, or COncstruc-tive COst MOdel. An unfortunate name, distorted 5 years later by the Beach Boys<sup>8</sup> as the soundtrack of a Hollywood blockbuster. But quite an incredible precedent, mostly forgotten by younger developers, and followed by the release of COCOMO 2.0<sup>9</sup> (by the same Barry Boehm) in 1995.*

Some of the readers of this magazine might remember a social media website called “Ohloh” created by the same team<sup>10</sup> that brought us SourceForge and that offered services to open-source software developers. One of those services was, indeed, a cost calculation widget available on every project page.

Ohloh became Open Hub<sup>11</sup> in the meantime. However, thanks to the Internet Archive’s Wayback Machine, we can see what Ohloh looked like in 2010. Their “Codebase Cost” page<sup>12</sup> explained in detail that they were using COCOMO for such evaluations. For example, the PEAR framework<sup>13</sup>, providing reusable PHP components, was estimated to have cost 17’555’092 USD in June 2010<sup>14</sup>. Today, the Open Hub tells us that Mozilla Firefox<sup>15</sup> costs half a billion dollars<sup>16</sup>.

In an age where we are debating the “unpopular opinion”<sup>17</sup> of whether we should pay people to make open-source software, Barry Boehm gave us the formula to figure out how expensive software actually was, and never paid attention to it. The work done by Boehm in this area is only comparable to that of Steve McConnell<sup>18</sup> and his book about software estimation<sup>19</sup>.

Barry Boehm’s did not stop there. In the 2000s, he proposed Value-Based Software Engineering (VBSE) as a paradigm to drive software team efforts and cost calculations.

As explained by Kevin Sullivan in the foreword of chapter 9,

*VBSE rests on the simple but transformative idea that any expenditure of scarce and valuable resources on software or a software-intensive system or initiative should be seen and managed as an investment in a risky asset: a project leading to a product or service of uncertain value.*

Sounds about right: consider software a risky asset and apply rules inspired by financial management to understand costs, outcomes, and resources.

His obituary<sup>20</sup> says it all. A “legend” and a “giant” are the words used to describe his work and persona.

Barry Boehm was part of a generation of academics who oversaw the rise of computing in all areas of society. He watched them evolve from large rooms onto our desks and later into our pockets, transforming every aspect of our lives. But sadly, instead of a podcast or a blog, he poured his brain into PDF papers, which most full-stack engineers have never read.

## PAPERS

Not many remain from his generation, and whether we like it or not, we are their legacy. We should understand the work of people like Barry Boehm better.

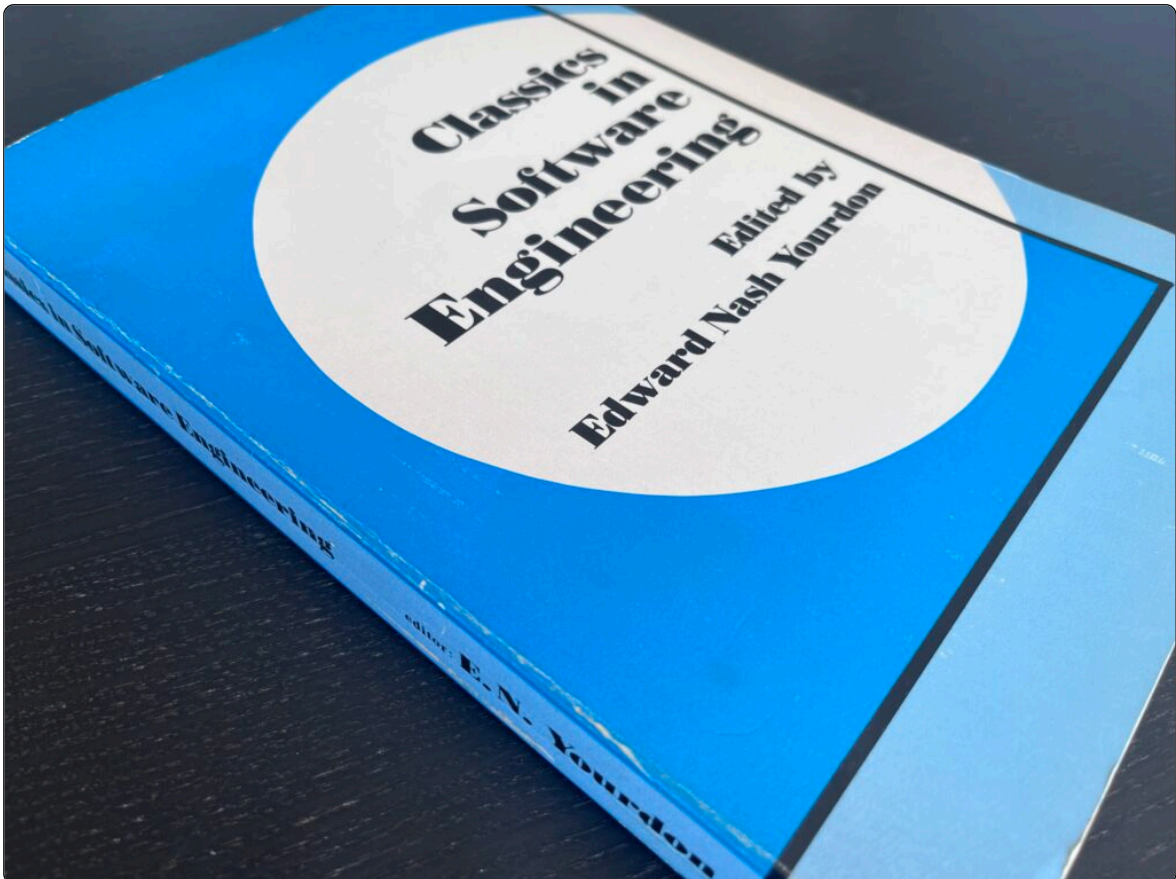
Cover photo by the author.

## REFERENCES

- <sup>1</sup> <https://deprogrammaticaipsum.com/eric-sink/>
- <sup>2</sup> <https://deprogrammaticaipsum.com/geoffrey-g-parker-marshall-w-van-alstyne-sangeet-paul-choudary/>
- <sup>3</sup> <https://deprogrammaticaipsum.com/brad-cox/>
- <sup>4</sup> [https://en.wikipedia.org/wiki/Barry\\_Boehm](https://en.wikipedia.org/wiki/Barry_Boehm)
- <sup>5</sup> <https://ieeexplore.ieee.org/book/5989528>
- <sup>6</sup> <https://deprogrammaticaipsum.com/the-impossible-dialogue/>
- <sup>7</sup> <https://staff.emu.edu.tr/alexanderchefranov/Documents/CMPE412/Boehm1981%20COCOMO.pdf>
- <sup>8</sup> [https://www.youtube.com/watch?v=fjWmbLS2\\_ec](https://www.youtube.com/watch?v=fjWmbLS2_ec)
- <sup>9</sup> <https://staff.emu.edu.tr/alexanderchefranov/Documents/CMPE412/Boehm1995%20COCOMO%202%20.pdf>
- <sup>10</sup> <https://www.linux.com/training-tutorials/ohloh-where-users-meets-developers/>
- <sup>11</sup> <https://www.openhub.net/>
- <sup>12</sup> [https://web.archive.org/web/20100626002648/http://www.ohloh.net/wiki/project\\_codebase\\_cost](https://web.archive.org/web/20100626002648/http://www.ohloh.net/wiki/project_codebase_cost)
- <sup>13</sup> <https://pear.php.net/>
- <sup>14</sup> <https://web.archive.org/web/20100621050651/http://www.ohloh.net/p/pear>
- <sup>15</sup> <https://www.mozilla.org/en-US/firefox/>
- <sup>16</sup> [https://www.openhub.net/p/firefox/estimated\\_cost](https://www.openhub.net/p/firefox/estimated_cost)
- <sup>17</sup> <https://www.youtube.com/watch?v=8mRVM3BUFPc>
- <sup>18</sup> <https://deprogrammaticaipsum.com/steve-mcconnell/>
- <sup>19</sup> <https://www.microsoftpressstore.com/store/software-estimation-demystifying-the-black-art-9780735635135>
- <sup>20</sup> <https://viterbischool.usc.edu/news/2022/09/barry-boehm-a-living-legend-in-systems-and-software-engineering-dies-at-87/>



# Edward Nash Yourdon



By Adrian Kosmaczewski, February 5th, 2024

Here is a confession. The first drafts of this issue of *De Programmatica Ipsum* were written under the name “*Structured Programming*”. Understandably enough, the news<sup>1</sup> of Niklaus Wirth’s passing triggered a prompt renaming and the choice of a somewhat narrower focus. However, Pascal’s rise in popularity during the 1970s and 1980s cannot be explained unless we elaborate on Structured Programming, and this month’s Library book is among the most important ones ever written about the subject.

The phrase “Structured Programming” implies the existence of an unstructured kind thereof. We have seen some examples of Unstructured Programming during the discussion of BASIC<sup>2</sup> a few months ago. The keyword that shows the lack of structure is GOTO, of course. Pascal showed up in history as the archnemesis of BASIC, and kept this role until the rise of Object-Oriented Programming<sup>3</sup> as the dominant paradigm in the 1990s, until the Delphi vs. Visual Basic debate was promptly drowned by the rise of Java and C#.

Let us analyze the genesis of Pascal, which occurred thanks to three major events that took place in the same *annus mirabilis* 1968.

Act one: in March, Edsger Dijkstra publishes his (in)famous “Go To Statement Considered Harmful”<sup>4</sup> article in the March 1968 edition of Communications of the ACM, probably the most cited paper ever in the history of computer programming. (This paper is, by the way, reprinted on page 29 of “Classics of Software Engineering”, this month’s Library book. But let us not digress.)

Act two: in October takes place the first, now legendary, NATO Software Engineering Conference<sup>5</sup> where the words “software crisis” fueled passionate discussions<sup>6</sup>:

*There was a considerable amount of debate on what some members chose to call the “software crisis” or the “software gap”. As will be seen from the quotations below, the conference members had widely differing views on the seriousness, or otherwise, of the situation, and on the extent of the problem areas.*

As the astute reader can imagine, the lack of structure was one of the major drivers of this “crisis”.

Act three: in December, the ALGOL committee (wrongly) chooses Adriaan van Wijngaarden<sup>7</sup>’s ideas as the basis for ALGOL 68<sup>8</sup>, instead of Niklaus Wirth’s much simpler proposal, historically referred to as ALGOL W<sup>9</sup>.

Niklaus Wirth promptly thanked the courtesy and left the ALGOL committee, taking with him the ideas of ALGOL W. He framed them with the family name of a famous philosopher of the seventeenth century<sup>10</sup>, thus creating a programming

language whose name<sup>11</sup> conveys seriousness, logic, and most importantly, clarity of thought<sup>12</sup>.

The first Pascal compiler (single-pass, top-down, recursive-descent based) appeared around 1970, after a failed first attempt in 1969, and the language slowly grew in popularity among teachers of computer science curricula.

Wirth told the story of Pascal in the second History of Programming Languages conference<sup>13</sup> of 1993:

*The primary innovation of Pascal was to incorporate a variety of data types and data structures, similar to ALGOL's introduction of a variety of statement structures. ALGOL offered only three basic data types: integers, real numbers, and truth values, and the array structure; Pascal introduced additional basic types and the possibility to define new basic types (enumerations, subranges), as well as new forms of structuring: record, set, and file (sequence), several of which had been present in COBOL.*

The major historical breakthrough of the language was, of course, the UCSD Pascal<sup>14</sup> system of the University of California San Diego in 1977, at the basis of various commercial implementations, including Apple's own version<sup>15</sup> for the Apple II.

“Structured Programming” was the big paradigm of the 1970s and 1980s, well before “Object Orientation” entered the collective psyche, and Pascal was poised to be its most important messenger. The GOTO keyword was to be erased from the minds of software developers forever, following Dijkstra's plea for sanity.

Yet... in an interesting twist of pragmatism, the first versions of Pascal *did include it*. Yes, you heard right; it was there, as a way to help software developers migrate their code from other languages (read, BASIC and FORTRAN) into this new world of procedures and functions. Again, let us read Niklaus Wirth's own words, telling us the reason in his HOPL II paper:

*The most vociferous criticism (of Pascal) came from Habermann, who correctly pointed out that Pascal was not the last word on language design. Apart from*

*taking issue with types and structures being merged into a single concept, (...) he reproached Pascal for retaining the much-cursed **goto** statement. In hindsight, one cannot but agree; at the time, its absence would have deterred too many people from trying to use Pascal. The bold step of proposing a **goto-less** language was taken ten years later by Pascal's successor Modula-2.*

Let us talk now about the actual book of this month: “Classics in Software Engineering”, a recollection of papers compiled and published in 1979 by Edward Nash Yourdon<sup>16</sup>, a computer consultant and engineer well known for his work in notations used in structured and object-oriented programming.

The GOTO keyword plays a central role in the book, almost to the point of obsession; apart from Dijkstra's paper previously mentioned in this page, there literally are four more articles with the phrase “go to” in its title:

1. One in favor<sup>17</sup>, by Martin Hopkins<sup>18</sup> of IBM, one of the inventors of the RISC architecture.
2. One against<sup>19</sup>, by William Wulf<sup>20</sup>.
3. A third one explaining how to translate “go to” programs to “while” programs<sup>21</sup>, by Edward Ashcroft<sup>22</sup> and Zohar Manna<sup>23</sup>.
4. Finally, Donald Knuth's own “Structured Programming with go to Statements”<sup>24</sup> paper.

“Classics in Software Engineering” proudly wears its name, featuring many more lectures and papers on the subject of Structured Programming by quite a few luminaries we have talked about in past issues of this magazine: Brian Kernighan<sup>25</sup>, Barry Boehm<sup>26</sup>, Tom DeMarco<sup>27</sup>, and Donald Knuth<sup>28</sup>. To top it all, it contains a copy of another classic article by Edsger Dijkstra: his own Turing Award<sup>29</sup> lecture, “The Humble Programmer”<sup>30</sup>, a timeless and enlightening lecture (or audio<sup>31</sup>, if you prefer) recommended to all software engineers.

And of course, the book features a contribution by Niklaus Wirth himself: “On the Composition of Well-Structured Programs”<sup>32</sup>, introduced by Yourdon on page 151 as follows:

*For example, after four pages of a delightful philosophical review of the dismal state of the art of programming, Wirth launches into examining a total of five sample programs – all involving an ALGOL-like pseudocode that will be somewhat unfamiliar to the average COBOL programmer, and all involving applications that he wouldn't care about. If you're a COBOL programmer, all I can say is, be patient! Spend the time to read through the examples and to study the code that Wirth presents; it really is worth the effort.*

I do not have much more to say, so I will just paraphrase and relay the same message to the audience of 2024: *if you are a JavaScript or Rust programmer, all I can say is, be patient! Spend the time to read this book; it really is worth the effort.*

If you want to read more from Niklaus Wirth himself, I can only recommend his classics, some of which are freely available online: his bestseller “Algorithms + Data Structures = Programs”<sup>33</sup> (1976); “Project Oberon: The Design of an Operating System, a Compiler, and a Computer”<sup>34</sup> (1992, revised in 2005 and 2013); and his lesser-known “Compiler Construction”<sup>35</sup> (1996). OK, here is one more, albeit not by Wirth, but about him: “The School of Niklaus Wirth”<sup>36</sup> (2000).

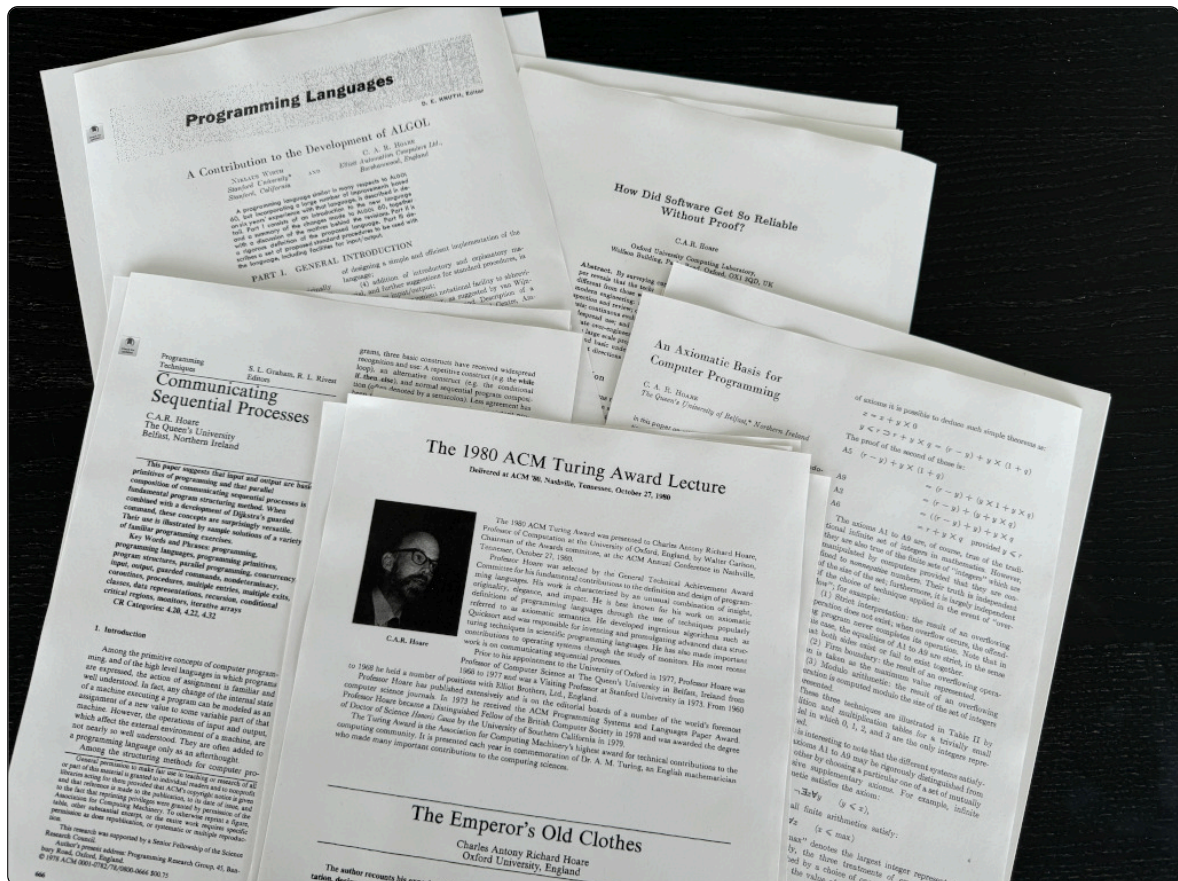
A final observation before closing this article. As much as I love (and have frequently mentioned in the pages of this magazine) Biancuzzi & Warden's wonderful book “Masterminds of Programming”<sup>37</sup>, I cannot but feel angry (dare I say, “betrayed”?) that it does not include an interview of Niklaus Wirth, who was evidently alive and very much active in 2009. This is, however, the only complaint that I have of an otherwise wonderful book that will have its own entry in this section in the future, and I want to think that the reasons for this omission were none other than scheduling conflicts or commercial deadlines.

Cover photo by the author.

## REFERENCES

- <sup>1</sup> <https://lists.inf.ethz.ch/pipermail/oberon/2024/016856.html>
- <sup>2</sup> <https://deprogrammaticaipsum.com/programming-the-liberal-arts/>
- <sup>3</sup> <https://deprogrammaticaipsum.com/the-hype-cycle-of-oop/>
- <sup>4</sup> <https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf>
- <sup>5</sup> [https://en.wikipedia.org/wiki/NATO\\_Software\\_Engineering\\_Conferences](https://en.wikipedia.org/wiki/NATO_Software_Engineering_Conferences)
- <sup>6</sup> <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF>
- <sup>7</sup> [https://en.wikipedia.org/wiki/Adriaan\\_van\\_Wijngaarden](https://en.wikipedia.org/wiki/Adriaan_van_Wijngaarden)
- <sup>8</sup> [https://en.wikipedia.org/wiki/ALGOL\\_68](https://en.wikipedia.org/wiki/ALGOL_68)
- <sup>9</sup> [https://en.wikipedia.org/wiki/ALGOL\\_W](https://en.wikipedia.org/wiki/ALGOL_W)
- <sup>10</sup> [https://en.wikipedia.org/wiki/Blaise\\_Pascal](https://en.wikipedia.org/wiki/Blaise_Pascal)
- <sup>11</sup> [https://en.wikipedia.org/wiki/Pascal\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Pascal_(programming_language))
- <sup>12</sup> <https://en.wikipedia.org/wiki/Pens%C3%A9es>
- <sup>13</sup> [https://en.wikipedia.org/wiki/History\\_of\\_Programming\\_Languages\\_\(conference\)](https://en.wikipedia.org/wiki/History_of_Programming_Languages_(conference))
- <sup>14</sup> [https://en.wikipedia.org/wiki/UCSD\\_Pascal](https://en.wikipedia.org/wiki/UCSD_Pascal)
- <sup>15</sup> [https://en.wikipedia.org/wiki/Apple\\_Pascal](https://en.wikipedia.org/wiki/Apple_Pascal)
- <sup>16</sup> [https://en.wikipedia.org/wiki/Edward\\_Yourdon](https://en.wikipedia.org/wiki/Edward_Yourdon)
- <sup>17</sup> <https://www.cs.csustan.edu/~mmartin/teaching/CS4100F19/Lectures/HopkinsGOTO.pdf>
- <sup>18</sup> <https://www.legacy.com/us/obituaries/lohud/name/martin-hopkins-obituary?id=16857996>
- <sup>19</sup> <https://dl.acm.org/doi/10.1145/800194.805861>
- <sup>20</sup> [https://en.wikipedia.org/wiki/William\\_Wulf](https://en.wikipedia.org/wiki/William_Wulf)
- <sup>21</sup> <https://dl.acm.org/doi/abs/10.5555/1241515.1241521>
- <sup>22</sup> <https://dl.acm.org/profile/81100071527>
- <sup>23</sup> <https://engineering.stanford.edu/news/computer-science-pioneer-zohar-manna-dies-age-79>
- <sup>24</sup> <https://dl.acm.org/doi/10.1145/356635.356640>
- <sup>25</sup> <https://deprogrammaticaipsum.com/brian-kernighan/>
- <sup>26</sup> <https://deprogrammaticaipsum.com/barry-boehm/>
- <sup>27</sup> <https://deprogrammaticaipsum.com/tom-demarco-timothy-lister/>
- <sup>28</sup> <https://deprogrammaticaipsum.com/the-art-of-the-art-of-computer-programming/>
- <sup>29</sup> [https://amturing.acm.org/award\\_winners/dijkstra\\_1053701.cfm](https://amturing.acm.org/award_winners/dijkstra_1053701.cfm)
- <sup>30</sup> <https://dl.acm.org/doi/pdf/10.1145/355604.361591>
- <sup>31</sup> <https://www.youtube.com/watch?v=9hWv2ECXHng>
- <sup>32</sup> <https://dl.acm.org/doi/10.1145/356635.356639>
- <sup>33</sup> [https://en.wikipedia.org/wiki/Algorithms\\_%2B\\_Data\\_Structures\\_%3D\\_Programs](https://en.wikipedia.org/wiki/Algorithms_%2B_Data_Structures_%3D_Programs)
- <sup>34</sup> <https://people.inf.ethz.ch/wirth/ProjectOberon/PO.System.pdf>
- <sup>35</sup> <https://people.inf.ethz.ch/wirth/CompilerConstruction/CompilerConstruction1.pdf>
- <sup>36</sup> <https://shop.elsevier.com/books/the-school-of-niklaus-wirth/boszormenyi/978-0-08-057418-9>
- <sup>37</sup> <https://www.oreilly.com/library/view/masterminds-of-programming/9780596801670/>

# Sir Tony Hoare



By Adrian Kosmaczewski, August 5th, 2024

It would be unwise and useless to try to summarize in a thousand words the immense contributions of Sir Charles Antony Richard Hoare<sup>1</sup>, also known as Tony Hoare (I suppose we are all good friends in this industry) or, with a more Tolkien feeling, as C. A. R. Hoare. I will settle for “Sir Tony Hoare” in this article; familiar yet respectful enough.

I will try to focus here on the concurrency part, and its obvious contribution to the Go programming language. Suffice to say that when you have Quicksort<sup>2</sup>, communicating sequential processes<sup>3</sup>, a billion-dollar mistake<sup>4</sup>, the Hoare Triple<sup>5</sup>,

the Hoare program correctness logic, the dining philosophers problem, a knighthood, and a Turing Award in your résumé, you hardly need an introduction, least of all by me.

(To add more gravitas to our feeling of underachievement, let us remember that Sir Tony Hoare spoke Russian, studied with Andrey Kolmogorov<sup>6</sup> in the Soviet Union, helped implement ALGOL, taught computer science at Oxford, worked as researcher for Microsoft, and was named a Fellow of the Royal Society (actually, maybe I should add the “FRS” suffix to the name, too.) Did I mention that he is also thoroughly considered a kind human being?)

Two years before receiving the Turing Award, Sir Tony Hoare published a seminal paper called “Communicating Sequential Processes”<sup>7</sup> commonly referred to as “CSP”. The fact that there is a Wikipedia page<sup>8</sup> dedicated to the concept should give you some context about the “ground-breakingness” of this paper.

In short, and as Hoare himself explains in an interview<sup>9</sup>, the experience of a failed multiprocessing project took him to elaborate on the complete and absolute banishment of shared state as a *conditio sine qua non* for process communication; instead, he argued, processes should be independent, isolated, but able to communicate with one another in a formal way.

Yes, pretty much what the Go syntax proposes today. It even borrowed the arrow (prominently used all along Sir Hoare’s paper, representing the various exchanges among parties) as a syntactic element, even though the version we got in the language points to the left, instead of pointing to the right (which in my mind still looks odd, but maybe that is because I am a lefty.)

The question that appears in my mind as I read the paper is, why did Go only appear in 2009? Well, Erlang<sup>10</sup> implemented some of these ideas as back as 1986, so clearly somebody at Ericsson was busy reading important computer science papers. Actually, another programming language you might have not heard about, occam<sup>11</sup>, implemented these ideas even earlier, in 1983.

The ideas behind CSP were expanded in the 1985 book of the same name, a book that is freely available to download<sup>12</sup> today.

If you are interested in learning more about Sir Tony Hoare, check the 2021 book “Theories of Programming: The Life and Works of Tony Hoare”<sup>13</sup> by Cliff B. Jones and Jayadev Misra, and published by the Association for Computing Machinery. Sir Hoare’s book “Structured Programming” co-authored with Ole Dahl<sup>14</sup> and Edsger Dijkstra<sup>15</sup> is freely available on the Internet Archive<sup>16</sup>.

You can also read the July 2024 edition<sup>17</sup> of the Newsletter of the Formal Aspects of Computing Science (FACS) Specialist Group<sup>18</sup>, which was dedicated to Sir Tony Hoare in his 90th birthday, featuring articles from various experts who worked with him.

Alternatively, you can also peruse the various papers shown in the cover image of this article:

- “A contribution to the development of ALGOL”<sup>19</sup>, Communications of the ACM, volume 9, issue 6, pages 413-432, 1966, authored with none other than Niklaus Wirth<sup>20</sup> himself.
- “An axiomatic basis for computer programming”<sup>21</sup>, Communications of the ACM, volume 12, issue 10, pages 576-580, 1969.
- “Communicating sequential processes”<sup>22</sup>, Communications of the ACM, volume 21, issue 8, pages 666-677, 1978.
- “The emperor’s old clothes”<sup>23</sup>, Communications of the ACM, volume 24, issue 2, pages 75-83, 1980. This was Hoare’s Turing Award Lecture, delivered at ACM ’80 in Nashville, Tennessee, USA.
- How Did Software Get So Reliable Without Proof?<sup>24</sup>, FME ’96: Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, Co-Sponsored by IFIP WG 14.3, Oxford, UK, March 18–22, 1996. Proceedings edited by Marie-Claude Gaudel and Jim Woodcock, pages 1-17.

## Reading About Go

Satisfying the curiosity of those who came to this page looking for books about Go, here go some recommendations.

- We have already talked about Kernighan’s book<sup>25</sup> in this magazine.

## PAPERS

- The Go language website contains a short tour<sup>26</sup> that might be useful to newcomers. There is also a useful “Effective Go”<sup>27</sup> document you might want to check.
- For those looking to learn in a step-by-step basis, “Practical Go Lessons”<sup>28</sup> and “Learn Go with Tests”<sup>29</sup> will prove to be foundational.
- If you already know Go and instead would like to see how to use it in a real project, I can only recommend “Writing an interpreter in Go”<sup>30</sup> and “Writing A Compiler In Go”<sup>31</sup> by Thorsten Ball.
- There is a free chapter about Generics<sup>32</sup> available online, extracted from a classic book that recently got its 2nd edition<sup>33</sup>.
- And if all of this was not enough, there are more<sup>34</sup> and more<sup>35</sup> books for your reading pleasure.

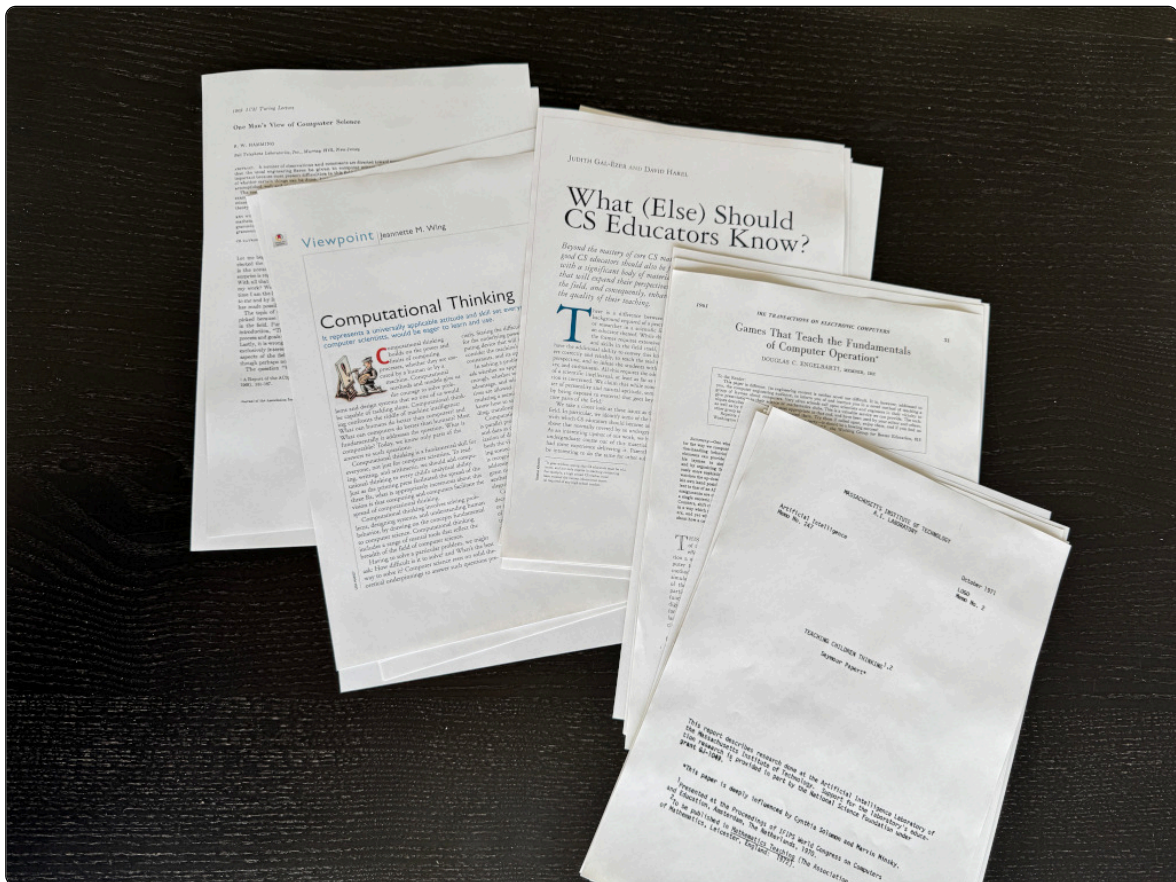
Cover photo by the author.

## REFERENCES

- <sup>1</sup> [https://en.wikipedia.org/wiki/Tony\\_Hoare](https://en.wikipedia.org/wiki/Tony_Hoare)
- <sup>2</sup> <https://en.wikipedia.org/wiki/Quicksort>
- <sup>3</sup> <https://www.youtube.com/watch?v=QUOlyIHmBrM>
- <sup>4</sup> <https://www.youtube.com/watch?v=ybrQvs4x0Ps>
- <sup>5</sup> <https://deprogrammaticaipsum.com/how-to-reason-about-mutable-state/>
- <sup>6</sup> [https://en.wikipedia.org/wiki/Andrey\\_Kolmogorov](https://en.wikipedia.org/wiki/Andrey_Kolmogorov)
- <sup>7</sup> <https://www.cs.cmu.edu/~crary/819-f09/Hoare78.pdf>
- <sup>8</sup> [https://en.wikipedia.org/wiki/Communicating\\_sequential\\_processes](https://en.wikipedia.org/wiki/Communicating_sequential_processes)
- <sup>9</sup> <https://www.youtube.com/watch?v=QUOlyIHmBrM>
- <sup>10</sup> [https://en.wikipedia.org/wiki/Erlang\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Erlang_(programming_language))
- <sup>11</sup> [https://en.wikipedia.org/wiki/Occam\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Occam_(programming_language))
- <sup>12</sup> <http://www.usingcsp.com/>
- <sup>13</sup> <https://dl.acm.org/doi/book/10.1145/3477355>
- <sup>14</sup> [https://en.wikipedia.org/wiki/Ole-Johan\\_Dahl](https://en.wikipedia.org/wiki/Ole-Johan_Dahl)
- <sup>15</sup> [https://en.wikipedia.org/wiki/Edsger\\_W.\\_Dijkstra](https://en.wikipedia.org/wiki/Edsger_W._Dijkstra)
- <sup>16</sup> [https://archive.org/details/Structured\\_Programming\\_\\_Dahl\\_Dijkstra\\_Hoare](https://archive.org/details/Structured_Programming__Dahl_Dijkstra_Hoare)
- <sup>17</sup> <https://www.bcs.org/media/1wrosvpv/facs-jul24.pdf>
- <sup>18</sup> <https://www.bcs.org/membership-and-registrations/member-communities/facs-formal-aspects-of-computing-science-group/newsletters/back-issues-of-facs-facts>
- <sup>19</sup> <https://dl.acm.org/doi/10.1145/365696.365702>
- <sup>20</sup> <https://deprogrammaticaipsum.com/niklaus-wirth/>
- <sup>21</sup> <https://dl.acm.org/doi/10.1145/363235.363259>
- <sup>22</sup> <https://dl.acm.org/doi/10.1145/359576.359585>
- <sup>23</sup> <https://dl.acm.org/doi/10.1145/358549.358561>
- <sup>24</sup> <https://www.cs.ox.ac.uk/publications/publication8320-abstract.html>
- <sup>25</sup> <https://deprogrammaticaipsum.com/brian-kernighan/>
- <sup>26</sup> <https://go.dev/tour/welcome/1>
- <sup>27</sup> [https://go.dev/doc/effective\\_go](https://go.dev/doc/effective_go)
- <sup>28</sup> <https://www.practical-go-lessons.com/>
- <sup>29</sup> <https://quii.gitbook.io/learn-go-with-tests>
- <sup>30</sup> <https://interpreterbook.com/>
- <sup>31</sup> <https://compilerbook.com/>
- <sup>32</sup> [https://learning-go-book.dev/chapter15\\_learningGo.pdf](https://learning-go-book.dev/chapter15_learningGo.pdf)
- <sup>33</sup> <https://www.oreilly.com/library/view/learning-go-2nd/9781098139285/>
- <sup>34</sup> <https://github.com/dariubs/GoBooks>
- <sup>35</sup> <https://go.dev/wiki/Books>



# A Review Of Research Around Programming Education From The 1960s To Today



By Adrian Kosmaczewski, December 2nd, 2024

The problem of teaching programming skills to new generations of software engineers is as old as the computers themselves. Each generation has tried to do it in a slightly different way, with various degrees of success. There is a lot of literature available online about the subject, and in this article we will point out papers and books that we found to be the most noteworthy. By no means this is an exhaustive list, but it features some interesting entries that might serve as a starting point for your own research.

## The Beginnings

Teaching people about computers has never been as complicated as it was in the early 1960s. To begin with, computers were relatively hard to come by according to our standards (and this is an understatement). However, many scientists (rightly) understood that the computer was a breakthrough machine in the history of mankind, and that knowledge about this invention had to be spread as much as possible.

Douglas Engelbart<sup>1</sup>, of “Mother of All Demos”<sup>2</sup> fame, came up with an interesting gamified approach to teach the inner workings of binary computers to “laymen”.

*The novel feature of the teaching method is that it makes use of human participants to simulate the function of logical elements that are typical of those used in digital computers. A group of such participants can be “wired” into a network that will function in a manner very similar to that of an actual digital network.*

(Douglas C. Engelbart, “Games That Teach the Fundamentals of Computer Operation”, IRE Transactions on Electronic Computers EC-10, no. 1 (March 1961): 31–41. <https://doi.org/10.1109/TEC.1961.5219149>.)

Of course this was not precisely teaching programming, but merely teaching about what computers are and how they work internally. You gotta start somewhere.

We have already remarked in this magazine<sup>3</sup> that 1968 was an *annus mirabilis* for computer science, similarly to how 1905 was one for Physics. It was the year of Engelbart’s “Mother of All Demos”, the year of Edsger Dijkstra’s “Go To Statement Considered Harmful” article, the year of the NATO Software Engineering

Conference and its definition of “software crisis”, and the year of ALGOL W starting to become Pascal<sup>4</sup> inside Wirth<sup>5</sup>’s brilliant mind (a language that, just 15 years later, would become a staple of programming curricula all over the world.)

That same year (seriously!) the report “Curriculum 68: Recommendations for Academic Programs in Computer Science”<sup>6</sup> appeared in the pages of Volume 11 of the Communications of the ACM. This was the first attempt at a formal definition of a study program for future generations of computer scientists and programmers.

The impact of this document on academia is hard to ignore. Richard Hamming<sup>7</sup> mentioned this landmark event during his 1968 (again!) Turing Award lecture:

*The topic of my Turing lecture, “One Man’s View of Computer Science,” was picked because “What is computer science?” is argued endlessly among people in the field. Furthermore, as the excellent Curriculum 68 report remarks in its introduction, “The Committee believes strongly that a continuing dialogue on the process and goals of education in computer science will be vital in the years to come.”*

(...)

*For example, let me make an arbitrary distinction between science and engineering by saying that science is concerned with what is possible while engineering is concerned with choosing, from among the many possible ways, one that meets a number of often poorly stated economic and practical objectives. We call the field “computer science” but I believe that it would be more accurately labeled “computer engineering” were not this too likely to be misunderstood.*

(Richard R. Hamming, “One Man’s View of Computer Science”, 1968 ACM Turing Award Lecture.)

In the decades that followed, the ACM Curriculum has been (understandably) updated many times, and this author has found online copies of the 1978<sup>8</sup>, 1991<sup>9</sup>, 2001<sup>10</sup>, 2005<sup>11</sup>, 2016<sup>12</sup>, and 2023<sup>13</sup> editions. These are massive documents, describing in detail the subjects, topics, and activities to be conducted in order to guide new students through the maze of computer technology.

## The Pioneer

Around 1970, Seymour Papert<sup>14</sup> applied the Constructivist theory<sup>15</sup> of Swiss child psychologist Jean Piaget<sup>16</sup> in order to come up with Logo<sup>17</sup>, a language geared towards teaching programming to young kids.

Logo was (still is, actually) quite a controversial topic in teaching circles. For some, the mere idea of moving a turtle on a screen (which was exactly what Logo allowed you to do) was too much of a simplification; for others, it was an opinionated approach that had no place in a classroom. In retrospect, the language did not survive to its hype.

The final report of the Didapro 7 / DidaSTIC<sup>18</sup> 2018 conference in Lausanne called “De 0 à 1 ou l’heure de l’informatique à l’école”<sup>19</sup> was edited by my friend Gabriel Parriaux and many others of his colleagues. In the abstract of the opening keynote by Professor Pierre Dillenbourg<sup>20</sup> of the École Polytechnique Fédérale de Lausanne<sup>21</sup>, we can read:

*In retrospect, Logo’s pedagogical potential fell victim to the level of expectation created by Papert’s speech, which was certainly brilliant and charismatic, but promised effects that no pedagogical approach could achieve. If you’re promised 1 million, you’ll be disappointed to receive half that.*

(Gabriel Parriaux, Jean-Philippe Pellet, Georges-Louis Baron, Éric Bruillard, et Vassilis Komis (Eds.), 2018, “De 0 à 1 ou l’heure de l’informatique à l’école”, actes du colloque Didapro 7 – DidaSTIC. Berne, Suisse: Peter Lang. <http://hdl.handle.net/20.500.12162/1438>. Translated from French to English by Adrian Kosmaczewski.)

Logo started a long tradition of education research at MIT, which eventually yielded the programming languages Scheme<sup>22</sup> and Scratch<sup>23</sup>, the latter a staple in programming labs during the first two decades of the 21st century, with a whole field of investigation to go with it. (We will come back to Scheme in a bit.)

The amount and impact of Seymour Papert’s research in the field of programming education is too big to summarize in a single section of a single article like this. Let me just point the interested reader to his 1980 book “Mindstorms: Children,

Computers, and Powerful Ideas”<sup>24</sup>, widely considered to be the hallmark in the field, and whose name was co-opted by Lego (hopefully with Papert’s blessing) for its famous line of construction sets<sup>25</sup>.

Let us close this section with a quote from 2001 that describes the towering impact of Piaget’s and Papert’s research, still felt more than half a century later:

*Psychologists and pedagogues like Piaget, Papert but also Dewey, Freynet, Freire and others from the open school movement can give us insights into: 1. How to rethink education, 2- imagine new environments, and 3- put new tools, media, and technologies at the service of the growing child. They remind us that learning, especially today, is much less about acquiring information or submitting to other people’s ideas or values, than it is about putting one’s own words to the world, or finding one’s own voice, and exchanging our ideas with others.*

(Edith Ackermann, “Piaget’s Constructivism, Papert’s Constructionism: What’s the Difference?,” January 2001.)

## The Personal Computer Era

The spread of the personal computer in the 1980s fundamentally changed the scenario for programming teachers. All of a sudden, schools of all levels and types could access, if they had the required monetary means, to a level of computing power that merely 10 years prior would have been unthinkable.

Even primary and high schools started experimenting with offering programming classes to their students, a fact that triggered a lot of controversy and research. Sadly, the conversation shifted around the most banal and useless of debates: which language is the best for teaching programming? (Insert rolling eye emoji here.)

*This perspective suggests that rather than arguing, as many currently are, over global questions such as which computer language is “best” for children, we would do better in asking: how can we organize learning experiences so that in the course of learning to program students are confronted with new ideas and*

*have opportunities to build them into their own understanding of the computer system and computational concepts?*

(Roy D. Pea and D. Midian Kurland, “On the Cognitive Effects of Learning Computer Programming”, *New Ideas in Psychology* 2, no. 2, January 1984, 137–68. [https://doi.org/10.1016/0732-118X\(84\)90018-7](https://doi.org/10.1016/0732-118X(84)90018-7).)

Donald Norman<sup>27</sup>, of “Design of Everyday Things”<sup>28</sup> fame, tried to shift back the conversation to the most important aspects: first, the usability of computers at the time (or rather, their lack thereof); second, to a much-needed distinction between levels of computer literacy.

*Let me quickly come to my main point: the difficulties that we mortals have with computers are unnecessary. If computers are not understandable, the few will dominate the masses, because the secret language of computation leaves out the uninitiated; those who understand make it hard for those who do not.*

(...)

*It is important to distinguish among four levels of computer literacy:*

- 1. Understanding general principles of computation. 2. Understanding how to use computers. 3. Understanding how to program computers. 4. Understanding the science of computation.*

(Donald A. Norman, “Worsening the Knowledge Gap\*: The Mystique of Computation Builds Unnecessary Barriers”, *Annals of the New York Academy of Sciences* 426, no. 1, November 1984, 220–33. <https://doi.org/10.1111/j.1749-6632.1984.tb16522.x>)

This paper by Donald Norman states as well the societal risk of increasing inequality between those countries that can afford computer equipment for their classrooms, and those that cannot. (As an anecdote, this was a fact witnessed by the author of these lines, who experienced a no-computing-whatsoever learning environment in Buenos Aires in 1990, and then a fully-equipped computer room filled with Apple Macintoshes in Geneva in 1991. The contrast could not have been harsher.)

# The Internet Era

By the mid-1990s, the rise of networking and the World Wide Web once again drove an acceleration of programming education. The future was online, and schools had to seize the day as quickly as possible. And to be able to face the new technological reality of the world, universities had to increase their professionalism; or at least, that is what Gal-Ezer and Harel thought:

*In fact, there is no clear agreement even on the name of the field. In European universities, the titles of many of the relevant departments revolve around the word “informatics,” whereas in the U.S. most departments are “computer science.” To avoid using the name of the machine in the title (a problem that prompted Dijkstra to quip that doing so is like referring to surgery as knife science), some use the word “computing” instead.*

(...)

*One of the main lessons we learned from teaching the material was that students must have an appropriate CS background. We cannot stress this statement enough. For example, one student in class was from electrical engineering, another’s sole connection to computing was via her use of computers in general education, and a third’s CS knowledge was 25 years old. These students simply did not fit in.*

(Judith Gal-Ezer and David Harel, “What (Else) Should CS Educators Know?” Communications of the ACM 41, no. 9, 1998.)

(Note: that last paragraph is infuriating and ageist. There, I said it.)

The publication of “Structure and Interpretation of Computer Programs”<sup>29</sup> (SICP) by Harold Abelson, Gerald Jay Sussman, and Julie Sussman in 1984 made Scheme<sup>30</sup> the go-to programming language for education for the following decade.

Schools and colleges all over the world adopted Scheme in the 1990s, but by the end of the decade the industry wanted more Java<sup>31</sup> developers, not Scheme developers, so Java schools<sup>32</sup> started producing Blub programmers<sup>33</sup>, functional programming be damned.

After all, even Philip Wadler<sup>34</sup> (of all people!) criticized Abelson and the Sussmans for their choice of programming language:

*This paper contrasts teaching in Scheme to teaching in KRC and Miranda, particularly with reference to Abelson and Sussman's text.*

(...)

*Some people may wish to dismiss many of the issues raised in this paper as being "just syntax". It is true that much debate over syntax is of little value. But it is also true that a good choice of notation can greatly aid learning and thought, and a poor choice can hinder it.*

(Philip Wadler, "A Critique of Abelson and Sussman or Why Calculating Is Better than Scheming", ACM SIGPLAN Notices 22, no. 3, March 1, 1987, 83–94.)

Let us speak a bit more about programming paradigms. The peak of the hype curve of Object-Oriented programming<sup>35</sup> happened right in the middle of the 1990s, and it had a strong impact in programming curricula. This debate burned quite a bit of research funding:

*It is a prevailing opinion that learning a programming language equals learning to program. In the call for papers for this workshop it is stated that "Switching to object-orientation is not just a matter of programming language". We suggest rephrasing and strengthening this statement: Learning to program is not just a matter of learning a programming language.*

(Jens Bennedsen and Michael E Caspersen, "Teaching Object-Oriented Programming", 2004.)

The research around teaching object-oriented programming to younger generations produced, among other highlights, the BlueJ System<sup>36</sup>:

*An environment for an object-oriented language does not make an object-oriented environment. The environment itself should reflect the paradigm of the language. In particular, the abstractions students work with should be classes and objects.*

(...)

*BlueJ is an integrated Java development environment specifically designed for introductory teaching. BlueJ is a full Java 2 environment: it is built on top of a standard Java SDK and thus uses a standard compiler and virtual machine. It presents, however, a unique front-end that offers a different interaction style than other environments.*

(Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg, “The BlueJ System and Its Pedagogy”, *Computer Science Education* 13, no. 4, December 2003, 249–68. <https://doi.org/10.1076/csed.13.4.249.17496>.)

What about on the other side of the fence? After teachers dropped Scheme because the industry wanted Java developers, should functional languages come back to the classroom? Surprisingly enough, the answer for some was a resounding no. (To be fair, Chakravarty and Keller propose to literally drop all programming paradigms, and instead to focus on, you know, teaching *thinking skills* instead of just teaching programming.)

*Let us start with a controversial thesis: We should not teach purely functional programming in freshman courses! In fact, we should not teach procedural, object-oriented or logic programming either. Instead, we should concentrate on teaching the elementary techniques of programming and the essential concepts of computing as a scientific discipline as well as foster analytic thinking and problem solving skills.*

(...)

*The central thesis of this article is that purely functional languages are ideally suited for introductory computing classes, but only if the focus is on general concepts rather than the specifics of functional programming.*

(Manuel M. T. Chakravarty and Gabriele Keller, “The Risks and Benefits of Teaching Purely Functional Programming in First Year”. *Journal of Functional Programming* 14, no. 1 (January 2004): 113–23. <https://doi.org/10.1017/S0956796803004805>.)

Sign of the times, the 2022 edition of SICP switched from Scheme to... JavaScript. I guess Douglas Crockford<sup>37</sup> must be proud.

Following the steps of Chakravarty and Keller, other researchers wanted out of the whole “what is the best programming language for education?” debate, and proposed a novel idea: how about teaching “computational thinking” to our younger generations? Particularly given the fact that most researchers knew the time of LLMs was just beyond the horizon; maybe we should focus on teaching kids how to think, in the good old ways of Papert and Piaget:

*Computational thinking confronts the riddle of machine intelligence: What can humans do better than computers? and What can computers do better than humans?*

(Jeannette M. Wing, “Computational Thinking”, 2006.)

Computational Thinking has had a major impact in programming education research during the past two decades. Noteworthy are the 2018 book “Hello World: How to be Human in the Age of the Machine”<sup>38</sup> by Hannah Fry<sup>39</sup>, which explores the subjects of computational thinking and its possible impact in society, and the 2019 book “Computational Thinking Education”<sup>40</sup> edited by Siu-Cheung Kong and Harold Abelson, and freely available in Open Access.

## Learn To Code!

The 2000s saw the rise of Jupyter notebooks<sup>41</sup>, Julia<sup>42</sup>, and R<sup>43</sup>, as implementations of Donald Knuth<sup>44</sup>’s concept of Literate Programming<sup>45</sup>.

These implementations of Literate Programming were not the first; commercial software packages like Mathematica<sup>46</sup> and Maple<sup>47</sup> had already explored the same idea starting in the 1980s. But the availability of such packages as Open-Source and Free Software implementations changed the game completely, and greatly contributed to their spread.

Needless to say, this distribution model had a non-negligible impact in the current developments around Machine Learning and Large Language Models. It also fueled a whole industry of “Learn to Code!” workshops, training courses, and

YouTube videos, still popular today despite the hiring freezes experienced by the software industry since 2022.

## Conclusion

This review is purposely short and left out plenty of important books, milestones, and papers that have marked the research of programming education. We cannot name them all, but here go some honorable mentions we must make:

- Regarding the rise of Python in the field of programming education:
  - “Python for Everybody”<sup>48</sup> by Charles Severance.
  - “Python Programming: An Introduction to Computer Science”<sup>49</sup> by John M. Zelle.
- “Teaching and Learning with Jupyter”<sup>50</sup> by Lorena Barba et al.
- “Lifelong Kindergarten: Cultivating Creativity through Projects, Passion, Peers, and Play”<sup>51</sup> by Mitchel Resnick, the creator of Scratch.
- “Inventive Minds: Marvin Minsky on Education”<sup>52</sup> by Cynthia Solomon and Xiao Xiao.
- Last but definitely not least, we cannot forget the work of Alan Kay and Adele Goldberg<sup>53</sup>, and in particular their outstanding contribution, namely Smalltalk, whose very name conveys the idea of teaching programming to younger generations in a playful yet sophisticated way.

We will end this short and opinionated summary with a quote by Turing Award winner John Hopcroft, one which we can only agree with:

*The potential of computer science, if fully explored and developed, will take us to a higher plane of knowledge about the world.*

(John E. Hopcroft, “Computer Science: The Emergence of a Discipline”, Turing Award Lecture, 1986.)

Cover photo by the author.

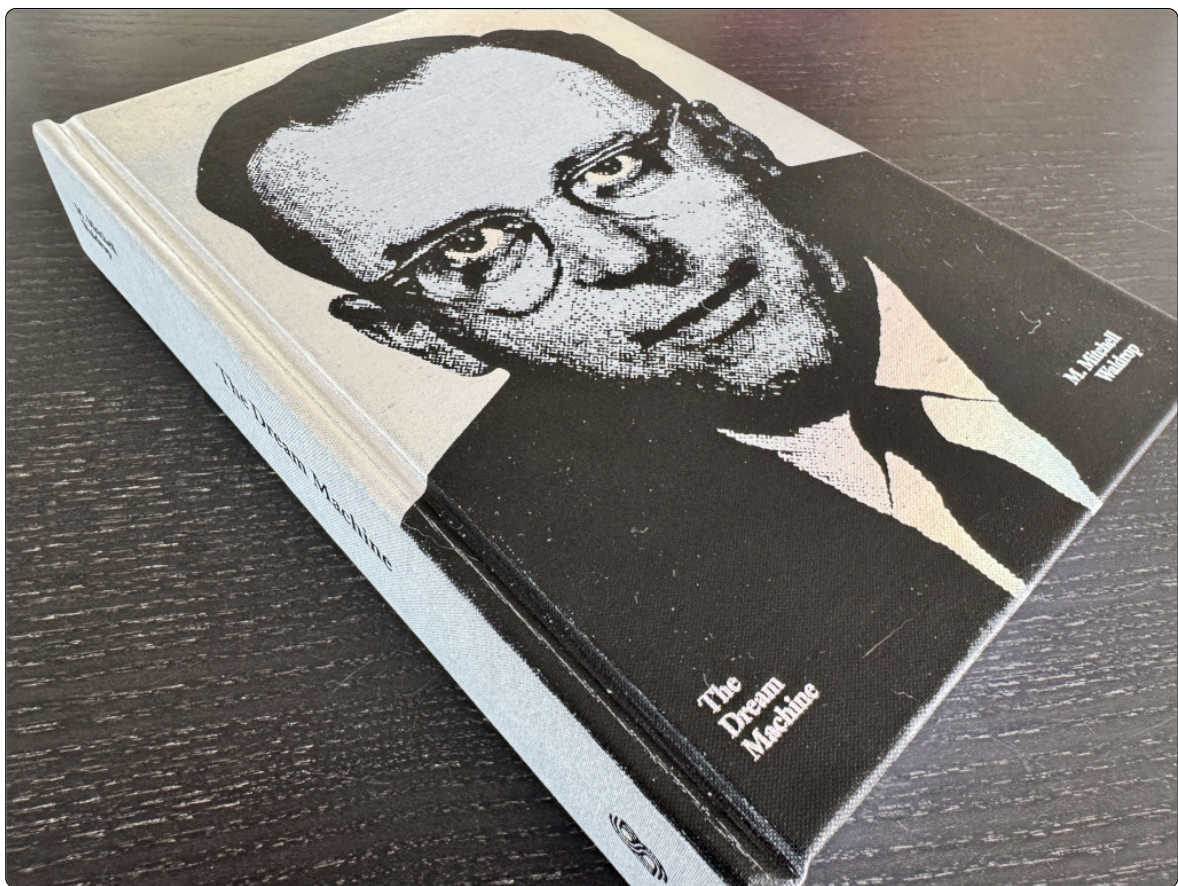
## REFERENCES

- <sup>1</sup> [https://en.wikipedia.org/wiki/Douglas\\_Engelbart](https://en.wikipedia.org/wiki/Douglas_Engelbart)
- <sup>2</sup> [https://en.wikipedia.org/wiki/The\\_Mother\\_of\\_All\\_Demos](https://en.wikipedia.org/wiki/The_Mother_of_All_Demos)
- <sup>3</sup> <https://deprogrammaticaipsum.com/edward-nash-yourdon/>
- <sup>4</sup> <https://deprogrammaticaipsum.com/lazarus-come-forth/>
- <sup>5</sup> <https://deprogrammaticaipsum.com/niklaus-wirth/>
- <sup>6</sup> <https://dl.acm.org/doi/10.1145/362929.362976>
- <sup>7</sup> [https://en.wikipedia.org/wiki/Richard\\_Hamming](https://en.wikipedia.org/wiki/Richard_Hamming)
- <sup>8</sup> <https://dl.acm.org/doi/10.1145/359080.359083>
- <sup>9</sup> <https://dl.acm.org/doi/book/10.1145/2594148>
- <sup>10</sup> <https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2001.pdf>
- <sup>11</sup> <https://dl.acm.org/doi/10.1145/1121341.1121482>
- <sup>12</sup> <https://ieeecs-media.computer.org/assets/pdf/ce2016-final-report.pdf>
- <sup>13</sup> <https://dl.acm.org/doi/book/10.1145/3664191>
- <sup>14</sup> [https://en.wikipedia.org/wiki/Seymour\\_Papert](https://en.wikipedia.org/wiki/Seymour_Papert)
- <sup>15</sup> [https://en.wikipedia.org/wiki/Constructivism\\_\(philosophy\\_of\\_education\)](https://en.wikipedia.org/wiki/Constructivism_(philosophy_of_education))
- <sup>16</sup> [https://en.wikipedia.org/wiki/Jean\\_Piaget](https://en.wikipedia.org/wiki/Jean_Piaget)
- <sup>17</sup> [https://en.wikipedia.org/wiki/Logo\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Logo_(programming_language))
- <sup>18</sup> <https://www.didapro.org/7/>
- <sup>19</sup> <https://orfee.hepl.ch/handle/20.500.12162/1438>
- <sup>20</sup> <https://people.epfl.ch/pierre.dillenbourg?lang=en>
- <sup>21</sup> <https://www.epfl.ch/en/>
- <sup>22</sup> [https://en.wikipedia.org/wiki/Scheme\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Scheme_(programming_language))
- <sup>23</sup> [https://en.wikipedia.org/wiki/Scratch\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Scratch_(programming_language))
- <sup>24</sup> [https://en.wikipedia.org/wiki/Mindstorms\\_\(book\)](https://en.wikipedia.org/wiki/Mindstorms_(book))
- <sup>25</sup> [https://en.wikipedia.org/wiki/Lego\\_Mindstorms](https://en.wikipedia.org/wiki/Lego_Mindstorms)
- <sup>26</sup> [https://doi.org/10.1016/0732-118X\(84\)90018-7](https://doi.org/10.1016/0732-118X(84)90018-7)
- <sup>27</sup> [https://en.wikipedia.org/wiki/Don\\_Norman](https://en.wikipedia.org/wiki/Don_Norman)
- <sup>28</sup> [https://en.wikipedia.org/wiki/The\\_Design\\_of\\_Everyday\\_Things](https://en.wikipedia.org/wiki/The_Design_of_Everyday_Things)
- <sup>29</sup> [https://en.wikipedia.org/wiki/Structure\\_and\\_Interpretation\\_of\\_Computer\\_Programs](https://en.wikipedia.org/wiki/Structure_and_Interpretation_of_Computer_Programs)
- <sup>30</sup> [https://en.wikipedia.org/wiki/Scheme\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Scheme_(programming_language))
- <sup>31</sup> <https://deprogrammaticaipsum.com/write-anywhere-run-once/>
- <sup>32</sup> <https://www.joelonsoftware.com/2005/12/29/the-perils-of-javaschools-2/>
- <sup>33</sup> <https://paulgraham.com/avg.html>
- <sup>34</sup> [https://en.wikipedia.org/wiki/Philip\\_Wadler](https://en.wikipedia.org/wiki/Philip_Wadler)
- <sup>35</sup> <https://deprogrammaticaipsum.com/the-hype-cycle-of-oop/>
- <sup>36</sup> <https://en.wikipedia.org/wiki/BlueJ>
- <sup>37</sup> <https://deprogrammaticaipsum.com/douglas-crockford/>
- <sup>38</sup> <https://www.waterstones.com/book/hello-world/hannah-fry/9781784163068>

- <sup>39</sup> [https://en.wikipedia.org/wiki/Hannah\\_Fry](https://en.wikipedia.org/wiki/Hannah_Fry)
- <sup>40</sup> <https://link.springer.com/book/10.1007/978-981-13-6528-7>
- <sup>41</sup> [https://en.wikipedia.org/wiki/Project\\_Jupyter#](https://en.wikipedia.org/wiki/Project_Jupyter#)
- <sup>42</sup> [https://en.wikipedia.org/wiki/Julia\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Julia_(programming_language))
- <sup>43</sup> [https://en.wikipedia.org/wiki/R\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/R_(programming_language))
- <sup>44</sup> <https://deprogrammaticaipsum.com/the-art-of-the-art-of-computer-programming/>
- <sup>45</sup> [https://en.wikipedia.org/wiki/Literate\\_programming](https://en.wikipedia.org/wiki/Literate_programming)
- <sup>46</sup> [https://en.wikipedia.org/wiki/Wolfram\\_Mathematica](https://en.wikipedia.org/wiki/Wolfram_Mathematica)
- <sup>47</sup> [https://en.wikipedia.org/wiki/Maple\\_\(software\)](https://en.wikipedia.org/wiki/Maple_(software))
- <sup>48</sup> <https://www.py4e.com/html3/>
- <sup>49</sup> <https://mcsp.wartburg.edu/zelle/python/>
- <sup>50</sup> <https://jupyter4edu.github.io/jupyter-edu-book/>
- <sup>51</sup> <https://direct.mit.edu/books/book/3134/Lifelong-KindergartenCultivating-Creativity>
- <sup>52</sup> <https://mitpress.mit.edu/9780262039093/inventive-minds/>
- <sup>53</sup> <https://deprogrammaticaipsum.com/adele-goldberg/>



# J. C. R. Licklider & M. Mitchell Waldrop



By Adrian Kosmaczewski, July 7th, 2025

The writings of Jorge Luis Borges twist our perception of time and space. In between articles about Shaw, Chesterton, Wilde, and Coleridge, his 1952 book “Otras Inquisiciones” included an unexpected gem: a short story called “El Tiempo y J. W. Dunne”. The question is, who was this John William Dunne<sup>1</sup> and what does he have to do with time? Well, his name might be forgotten by contem-

porary audiences, but Dunne was the author of one of the biggest bestsellers of the first half of the twentieth century.

Dunne, aviator, engineer, and philosopher according to Wikipedia, published in 1927 the influential book “An Experiment with Time”<sup>2</sup>, in which Dunne explains that not only precognitive dreams happen... but we can also analyze them and understand them, gaining some knowledge about the future in the process.

Or, to put it in a TikTok-friendly manner: *manifestation* is a thing.

More seriously; Borges says *à propos* of Dunne:

*Los teólogos definen la eternidad como la simultánea y lúcida posesión de todos los instantes del tiempo y la declaran uno de los atributos divinos. Dunne, asombrosamente, supone que ya es nuestra la eternidad y que los sueños de cada noche lo corroboran.*

Which translated in English would read as such:

*Theologians define eternity as the simultaneous and lucid possession of all moments in time and declare it one of the divine attributes. Dunne, astonishingly, assumes that eternity is already ours and that our dreams each night corroborate this.*

Did Joseph Carl Robnett Licklider<sup>3</sup> (1915-1990) read “An Experiment in Time”? Could it be that he had a series of dreams between 1960 and 1968, and that he quickly wrote them down in his diary before breakfast? We can only speculate. But we do know for a fact that those dreams begat a nothing short of extraordinary sequence of writings: “Man-Computer Symbiosis”<sup>4</sup> (1960); “Intergalactic Computer Network”<sup>5</sup> (1963); “Libraries of the Future”<sup>6</sup> (1965); and “The Computer as a Communication Device”<sup>7</sup>, this last one from a dream he shared with Bob Taylor, and published in 1968.

In all of these ~~manifestations~~ works, Lick (as he was commonly referred to, and you know that we like to sound cool and be familiar with our celebrities in this

magazine) explored and exposed the possibilities and the wonders of not only giving each person a computer, but also, and most important of all, of connecting all of those computers together in a common network.

Lick was the OG dream machine, all right. To such an extent, that the J. C. R. Licklider paper collection<sup>8</sup>, set up by MIT after Lick's death in 1990 with the help of his family, has a volume of 25 cubic feet (around 0.7 cubic meters), much of which has not been entirely reviewed. For the rest of us without access to the MIT vault, an *ad hoc* collection of Lick's related papers is available on the Internet Archive<sup>9</sup>.

Precisely, the story of Lick's dreams, their extent and impact, and how they became a reality (but sadly not whether he wrote them down before breakfast or not) is the subject of the monumental work of M. Mitchell Waldrop, "The Dream Machine"<sup>10</sup>, originally published in 2001<sup>11</sup>. It was re-released by Stripe Press in 2018 in a stunning volume that includes not only the original text, but also the contents of three of Lick writings enumerated above (that is, all except "Libraries of the Future").

*It was a vision that was downright Jeffersonian in its idealism, and perhaps in its naïveté as well. Nonetheless, Lick insisted, "the renewed hope I referred to is more than just a feeling in the air... It is a feeling one experiences at the console. The information revolution is bringing with it a key that may open the door to a new era of involvement and participation. The key is the self-motivating exhilaration that accompanies truly effective interaction with information and knowledge through a good console connected through a good network to a good computer".*

(Waldrop, page 401.)

When we say monumental, you had better believe it; the 500+ pages of this volume, laid out with astonishing detail (and a very small font size) summarize the history and evolution of computers from 1945 to 1990. Throughout these pages, Waldrop reveals that the backbone, the

axis, the arrow, the orientation, the mastermind of all that history was none other than Lick himself: he was the incarnation of the phrase “being at the right place at the right time”.

Speaking about personification: in a theme dear to this magazine, we cannot but acknowledge that Lick embodied the fight against the impossible dialogue<sup>12</sup> between engineering and business. He was a psychologist *and* a computer scientist. He understood the minds of humans and those of computers alike. Lick was the thread and the needle, shaping the path from Alan Turing, Claude Shannon, and John von Neumann<sup>13</sup>, to the World Wide Web<sup>14</sup>, Object-Oriented Programming<sup>15</sup>, and the home office<sup>16</sup>.

Why deny it, he was also the purse and the wallet along the path, having used his influence in BBN<sup>17</sup>, MIT, ARPA<sup>18</sup>, and anywhere he could, to fund those around him working on the bits and pieces required to bring his dream to life. Imagine the privilege of being able to sign checks with such nonchalance.

Paraphrasing some famous credit card advertising slogan: having ideas is great. Having cash to make them happen is priceless.

“The Dream Machine” appeared merely months after Michael Hiltzik’s “Dealers of Lightning”<sup>19</sup> hit the shelves; in many ways, Waldrop’s book is an extended version of Hiltzik’s, both backwards and forwards in time. To the extent that a full chapter in “The Dream Machine” (number eight, to be precise) is dedicated to Xerox PARC, giving a much better context to its creation than Hiltzik’s work can, albeit with slightly less detail (easy, since after all, Hiltzik’s subject is narrower in scope).

Waldrop’s acknowledges this himself in page 419, speaking about the now legendary visit of Steve Jobs to PARC in December 1979:

*Nonetheless, Hiltzik’s book includes perhaps the most careful and complete reconstruction of the event to date, and it makes a number of key points. First, Jobs and his crew didn’t need a special presentation to learn about graphical user interfaces. That idea was already in the air by 1979, along with everything else PARC had done.*

In any case, both books complement each other wonderfully. (OK, OK, OK; if you want to know, in general I actually enjoyed more “The Dream Machine” than “Dealers of Lightning”, but both are great.)

No need to stop here: if you are interested in learning more about Lick, there is a short biography in chapter seven (titled “The Internet”) of “The Innovators: How a Group of Hackers, Geniuses, and Geeks Created the Digital Revolution”<sup>20</sup> by Walter Isaacson.

And tangentially related, even if more appropriate for the Vidéotheque section of this magazine, yet also distributed by Stripe Press, you might want to watch “We Are As Gods”<sup>21</sup>, a documentary about a certain Stewart Brand<sup>22</sup>... another person who, in between LSD trips, transcribed his premonitory dreams in the “Whole Earth Catalog”<sup>23</sup> for a whole generation to get inspiration from.

Bob Taylor<sup>24</sup> (1932-2017) said about Lick:

*Most of the significant advances in computer technology—including the work that my group did at Xerox PARC—were simply extrapolations of Lick’s vision. They were not really new visions of their own. So he was really the father of it all.*

Waldrop quotes another Bob in page 252: Robert Fano<sup>25</sup> (1917-2016) praising Lick in his own words:

*Second, says Fano, when Lick was presented with a miraculous, never-to-be-repeated opportunity to turn his vision into reality, he had the guts to go for it, and the skills to make it work. Lick had the power to spin his dreams so persuasively that Jack Ruina and company were willing to go along with him—and to trust him with the Pentagon’s money. Once he had that money in hand, moreover, Lick had the taste to recognize and cultivate good ideas wherever he found them. Indeed, the ideas he fostered in 1962 would ultimately lay the foundations for computing as we know it today.*

Maybe premonitory dreams are a thing, after all. It certainly is comforting to think that we can dream a better future for all of us, involving computers or not, and

then make it a reality—if we have the guts... and the cash, that is. Think about the possibilities.

(Or, to put it differently: imagine the negative impact DOGE will have in the US economy during the next 50 years. You have been warned.)

Let us finish this article with some of Waldrop's own closing words:

*Technology isn't destiny, no matter how inexorable its evolution may seem; the way its capabilities are used is as much a matter of cultural choice and historical accident as politics is, or fashion.*

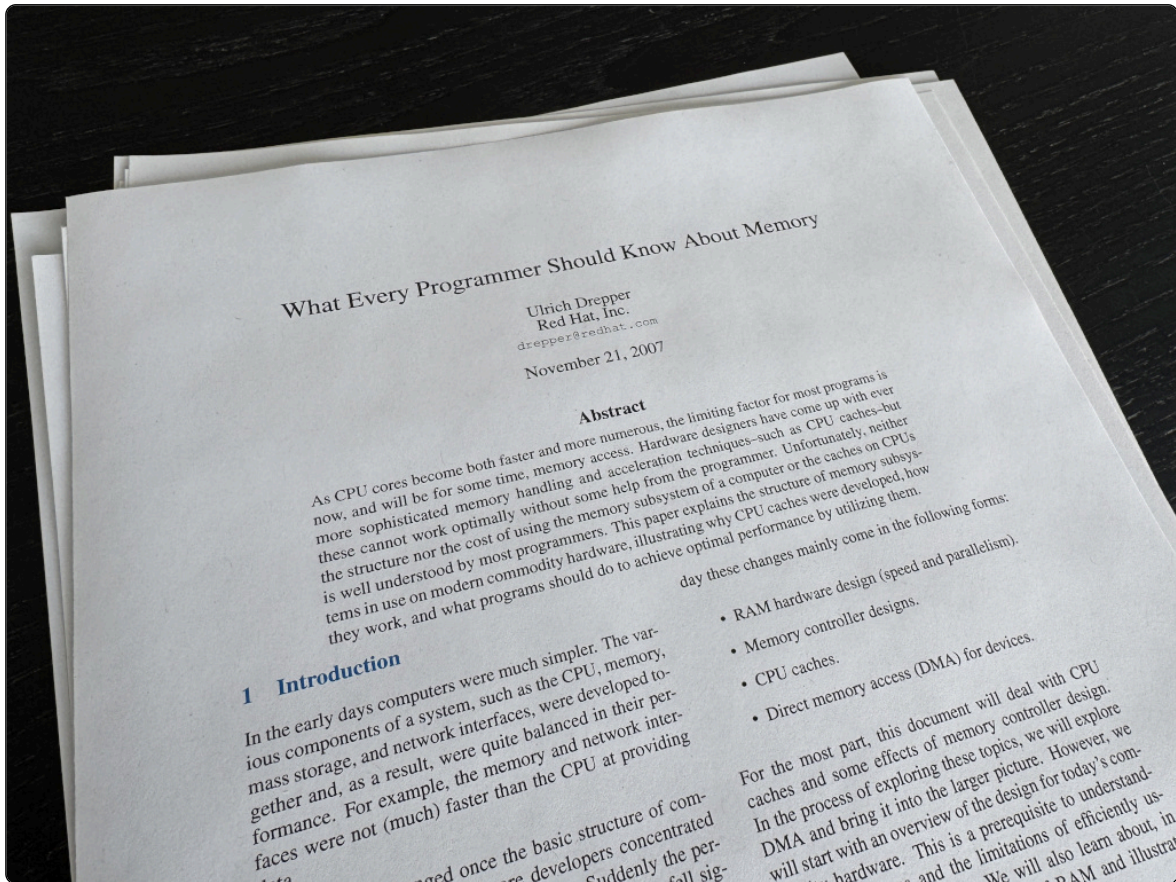
Cover photo by the author.

## REFERENCES

- <sup>1</sup> [https://en.wikipedia.org/wiki/J.\\_W.\\_Dunne](https://en.wikipedia.org/wiki/J._W._Dunne)
- <sup>2</sup> [https://en.wikipedia.org/wiki/An\\_Experiment\\_with\\_Time](https://en.wikipedia.org/wiki/An_Experiment_with_Time)
- <sup>3</sup> [https://en.wikipedia.org/wiki/J.\\_C.\\_R.\\_Licklider](https://en.wikipedia.org/wiki/J._C._R._Licklider)
- <sup>4</sup> [https://worrydream.com/refs/Licklider\\_1960\\_-\\_Man-Computer\\_Symbiosis.pdf](https://worrydream.com/refs/Licklider_1960_-_Man-Computer_Symbiosis.pdf)
- <sup>5</sup> [https://worrydream.com/refs/Licklider\\_1963\\_-\\_Members\\_and\\_Affiliates\\_of\\_the\\_Intergalactic\\_Computer\\_Network.pdf](https://worrydream.com/refs/Licklider_1963_-_Members_and_Affiliates_of_the_Intergalactic_Computer_Network.pdf)
- <sup>6</sup> [https://monoskop.org/images/1/14/Licklider\\_JCR\\_Libraries\\_of\\_the\\_Future.pdf](https://monoskop.org/images/1/14/Licklider_JCR_Libraries_of_the_Future.pdf)
- <sup>7</sup> [https://internetat50.com/references/Licklider\\_Taylor\\_The-Computer-As-A-Communications-Device.pdf](https://internetat50.com/references/Licklider_Taylor_The-Computer-As-A-Communications-Device.pdf)
- <sup>8</sup> <https://archivesspace.mit.edu/repositories/2/resources/992>
- <sup>9</sup> [https://archive.org/details/licklider\\_jcr/](https://archive.org/details/licklider_jcr/)
- <sup>10</sup> <https://press.stripe.com/the-dream-machine>
- <sup>11</sup> <https://www.wired.com/2001/10/the-dream-machine-j-c-r-licklider-and-the-revolution-that-made-computing-personal-by-m-mitchell-waldrop/>
- <sup>12</sup> <http://the-impossible-dialogue/>
- <sup>13</sup> <https://deprogrammaticaipsum.com/william-aspray/>
- <sup>14</sup> <https://deprogrammaticaipsum.com/from-hypertext-to-spas-to-hypertext/>
- <sup>15</sup> <https://deprogrammaticaipsum.com/the-hype-cycle-of-oop/>
- <sup>16</sup> <https://deprogrammaticaipsum.com/await-in-async-we-trust/>
- <sup>17</sup> [https://en.wikipedia.org/wiki/Raytheon\\_BBN](https://en.wikipedia.org/wiki/Raytheon_BBN)
- <sup>18</sup> <https://en.wikipedia.org/wiki/DARPA>
- <sup>19</sup> <https://deprogrammaticaipsum.com/michael-hiltzik/>
- <sup>20</sup> [https://en.wikipedia.org/wiki/The\\_Innovators\\_\(book\)](https://en.wikipedia.org/wiki/The_Innovators_(book))
- <sup>21</sup> <https://press.stripe.com/we-are-as-gods>
- <sup>22</sup> [https://en.wikipedia.org/wiki/Stewart\\_Brand](https://en.wikipedia.org/wiki/Stewart_Brand)
- <sup>23</sup> [https://en.wikipedia.org/wiki/Whole\\_Earth\\_Catalog](https://en.wikipedia.org/wiki/Whole_Earth_Catalog)
- <sup>24</sup> [https://en.wikipedia.org/wiki/Robert\\_Taylor\\_\(computer\\_scientist\)](https://en.wikipedia.org/wiki/Robert_Taylor_(computer_scientist))
- <sup>25</sup> [https://en.wikipedia.org/wiki/Robert\\_Fano](https://en.wikipedia.org/wiki/Robert_Fano)



# Ulrich Drepper



By Adrian Kosmaczewski, October 6th, 2025

This month's Vidéothèque movie<sup>1</sup> provides a (very) short and simple introduction to the subject of memory architecture. But this is not, by far, the minimum any software developer should know about memory segmentation and management for their daily work; let alone computer scientists, or developers working in native code for embedded platforms, or even mobile applications. This is where this month's Library choice shines in full: we are talking of the most comprehensive article you will ever read about the subject of computer memory, by far, and it remains as relevant as it was at the time of its publication 18 years ago.

We are talking about a paper titled “What Every Programmer Should Know About Memory”<sup>2</sup>, published in November 21st, 2007 by Ulrich Drepper<sup>3</sup>, who at the time of this writing is Distinguished Engineer<sup>4</sup> at Red Hat Research. Mr. Drepper is mostly known outside of research circles for being one of the main contributors (from 1995 to 2012) to the GNU C Library or glibc<sup>5</sup> project, one of the most important implementations of the C standard library in the world of Free and Open Source software.

(Just a quick disclaimer: although I am also working for Red Hat as I write these lines, I am completely unaffiliated with Mr. Drepper and I do not know him personally, at least not so far.)

The abstract of this paper gives a clear indication of the subject and target audience:

*This paper explains the structure of memory subsystems in use on modern commodity hardware, illustrating why CPU caches were developed, how they work, and what programs should do to achieve optimal performance by utilizing them.*

The author has not chosen the title of this paper randomly, either:

*The title of this paper is an homage to David Goldberg’s classic paper “What Every Computer Scientist Should Know About Floating-Point Arithmetic”. This paper is still not widely known, although it should be a prerequisite for anybody daring to touch a keyboard for serious programming.*

Faithful and attentive readers of *De Programmatica Ipsum* will surely remember that we mentioned Goldberg’s paper<sup>6</sup> in the article we published last year about floating-point arithmetic<sup>7</sup>.

As an aside, it is worth mentioning that the title prefix “What Every” is a common trait of some famous publications in our craft, just like the “... Considered Harmful” suffix; in the former case, suffice to mention the books “What Every Engineer Should Know about Software Engineering” by Philip A. Laplante, first published in 2007 and recently updated<sup>8</sup> in 2022, in a second edition co-authored with Mohamad Kassab. This book, in particular, is part of a series by CRC Press

about, precisely, “What Every” professional in a particular branch of engineering should know about some other subject.

In the same vein, we cannot omit a mention to “97 Things Every Programmer Should Know”<sup>9</sup>, edited by none other than Kevlin Henney<sup>10</sup>, and “97 Things Every Software Architect Should Know”<sup>11</sup>, this one edited by Richard Monson-Haefel<sup>12</sup>. Each of these two books are entirely worthy of their own entries in this section.

But, as usual, I digress. Let us return to Mr. Drepper’s paper, the subject of this month. This paper provides an explanation, with details, of how memory works under the hood, and how programmers can use this knowledge to write better software.

The paper is organized in 8 sections, starting from the hardware basis of memory and including SRAM, DRAM, bus latencies, explaining why DRAM is the dominant form of main memory (guess what: the main reason is... cost.) These explanations are sprinkled with very interesting bits and pieces of computer architecture history:

*Even though most computers for the last several decades have used the von Neumann architecture, experience has shown that it is of advantage to separate the caches used for code and for data. Intel has used separate code and data caches since 1993 and never looked back.*

The reader is then pushed into the realm of CPU caches, starting from the reason for their existence, then diving into cache levels (L1, L2, L3) and their historical evolution, and even showing how Intel and AMD implemented their caches differently. Because yes, that had a perceivable effect in the performance of the computers you could buy at the turn of the millennium.

On goes Mr. Drepper into virtual memory, hyper-threading, page tables, Memory Management Units (MMUs), Non-Uniform Memory Access (NUMA), Direct Memory Access (DMA), Direct Cache Access (DCA), and many other acronyms you might have probably already encountered, particularly when shopping for CPUs, motherboards, or other computer components. This is your chance to finally understand what this is all about.

Needless to say, this paper is particularly interesting for developers working with C, C++, Zig, or even Rust, where knowledge about memory layouts and performance can make or break a whole project. In particular, and thinking about those developers, Mr. Drepper provides an introduction to Valgrind<sup>13</sup> and its associated tooling: the Cachegrind<sup>14</sup> tracing profiler and the Massif<sup>15</sup> heap profiler. (If you are building applications with any of those “lower-level” programming languages, Valgrind is a must-have. Besides Cachegrind and Massif, it comes bundled with valuable tools like Callgrind, DHAT, Helgrind, DRD... each of these easily accessible with the `--tool` argument.)

At the risk of (yet another) spoiler alert, here are some major commandments developers should abide to after reading this paper: memory is thy new bottleneck, for CPUs have gotten faster, but memory did not; thou should know thy cache hierarchies; thou must remember that data locality is everything; threading does not automatically mean faster code; and remember that thou (and thy compiler) cannot beat physics.

This month’s Library paper, “What Every Programmer Should Know About Memory”, is available on the author’s website<sup>16</sup> and, needless to say, should be (another) mandatory reading for all of us. As explained by Peter Cordes<sup>17</sup> on a question answered on Stack Overflow<sup>18</sup>, this paper is still very much relevant, although many examples shown therein are, of course, based on well known CPU architectures from the 1990s and early 2000s.

If you are still interested about computer memory, in particular about its security aspects (and how security enforcement agencies can gain access to whatever you are doing in your computer these days), we must not forget to recommend “The Art of Memory Forensics”<sup>19</sup> by Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters, published in 2014. The authors of this masterpiece have also published Volatility<sup>20</sup>, a Python toolkit for memory forensics. Their book has chapters about Windows, Linux, and Mac memory architectures, including explanations of memory layout, including bits and pieces of C programming, the layout of data structures in memory... and so many other subjects that we would need another article in this section just for it.

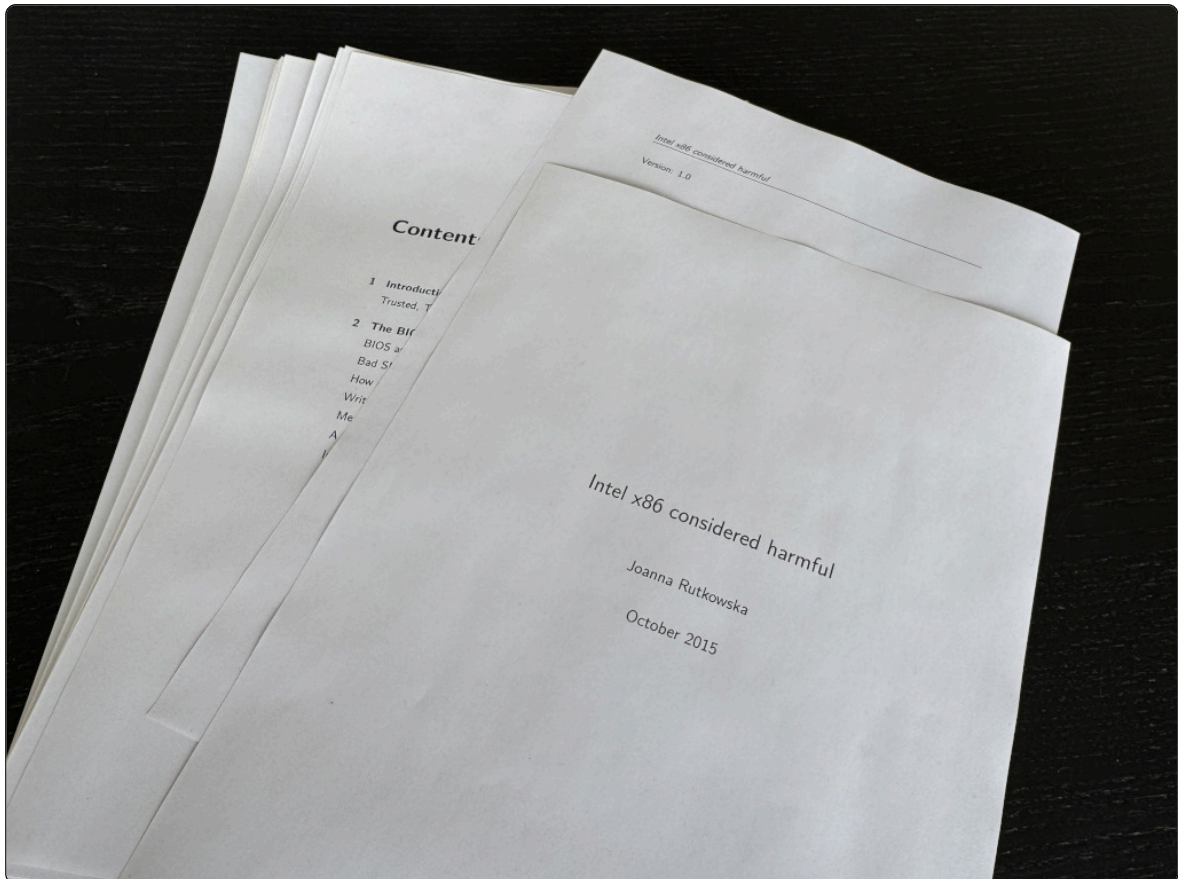
Cover photo by the author.

## REFERENCES

- <sup>1</sup> <https://deprogrammaticaipsum.com/ryan-baker/>
- <sup>2</sup> <https://www.akkadia.org/drepper/cpumemory.pdf>
- <sup>3</sup> <https://www.akkadia.org/drepper/>
- <sup>4</sup> [https://research.redhat.com/blog/project\\_member/ulrich-drepper/](https://research.redhat.com/blog/project_member/ulrich-drepper/)
- <sup>5</sup> <https://en.wikipedia.org/wiki/Glibc>
- <sup>6</sup> <https://deprogrammaticaipsum.com/the-smartest-concept-in-computer-science/#:~:text=David%20Goldberg>
- <sup>7</sup> <https://deprogrammaticaipsum.com/the-smartest-concept-in-computer-science/>
- <sup>8</sup> <https://www.taylorfrancis.com/books/mono/10.1201/9781003218647/every-engineer-know-software-engineering-phillip-laplante-mohamad-kassab>
- <sup>9</sup> <https://www.oreilly.com/library/view/97-things-every/9780596809515/>
- <sup>10</sup> [https://en.wikipedia.org/wiki/Kevlin\\_Henney](https://en.wikipedia.org/wiki/Kevlin_Henney)
- <sup>11</sup> <https://www.oreilly.com/library/view/97-things-every/9780596800611/>
- <sup>12</sup> [https://nofluffjuststuff.com/conference/speaker/richard\\_monson-haefel](https://nofluffjuststuff.com/conference/speaker/richard_monson-haefel)
- <sup>13</sup> <https://valgrind.org/>
- <sup>14</sup> <https://valgrind.org/docs/manual/cg-manual.html>
- <sup>15</sup> <https://valgrind.org/docs/manual/ms-manual.html>
- <sup>16</sup> <https://www.akkadia.org/drepper/cpumemory.pdf>
- <sup>17</sup> <https://stackoverflow.com/users/224132/peter-cordes>
- <sup>18</sup> <https://stackoverflow.com/a/47714514/133764>
- <sup>19</sup> <https://memoryanalysis.net/amf/>
- <sup>20</sup> <https://volatilityfoundation.org/>



# Joanna Rutkowska



By Adrian Kosmaczewski, December 1st, 2025

In a key scene of the 2012 blockbuster James Bond film “Skyfall”<sup>1</sup>, MI6 quarter-master Q, played by Ben Whishaw, realizes too late<sup>2</sup> that plugging a cable into the laptop of a notoriously skilled terrorist like Raoul Silva (one of Javier Bardem’s most remarkable roles) was a terrible idea. After a few seconds of connection, the laptop infects the systems of MI6, releasing all physical doors and disabling all security guards, prompting Silva to escape and wreak havoc through the London Underground. A message appears on the laptop screen, taunting Q, reading “Not such a clever boy”.

It seems to us like Q was not aware of Joanna Rutkowska<sup>3</sup>'s work before getting the job of quartermaster, let alone plugging that cable. Maybe it is time for MI6 to review their hiring processes.

Unbeknownst to most filmgoers is the fact that Ms Rutkowska is at the origin of the name "Evil maid attack"<sup>4</sup>, physically perpetrated against or through computing devices, allowing attackers to steal information or potentially even injecting malware, to be remotely activated later on.

(Kids: never plug your devices to those wall-mounted USB sockets you find on hotels and such. You have been warned. And no, we are not kidding.)

The noughties<sup>5</sup> were a troublesome, yet revealing, decade for computer security, dictated by a global political turmoil that would trigger an even more complicated decade right after (and let us not talk about the current one, shall we).

In terms of computer security, after learning from Microsoft itself in 2002 that most software was vulnerable<sup>6</sup> to exploits of a gazillion kinds, that network security<sup>7</sup> was almost non-existent, and that the infrastructure of our modern world is at risk of collapsing<sup>8</sup> every minute, the wake-up call has been violent, and the response, at best, sloppy and dull. Yes, we are now in 2025, and we have Rust, but until the great rewriting<sup>9</sup> is over, we will have to deal with a world run by quite deficient software, at best. While we wait for our app to compile, the state of the world keeps degrading.

It is not like we did not know any better. Just like Cassandra<sup>10</sup>, the daughter of Priam, himself king of Troy, who was cursed with the gift of prophecy and the inability of convincing anyone of her predictions, we have had many experts warning the industry of impending disasters, to no avail. Suffice to mention Window Snyder and Kate Moussouris<sup>11</sup>, Bruce Schneier<sup>12</sup>, and even our dear friend Anastasiia Vixentael<sup>13</sup>.

It is mandatory for us to add Ms Rutkowska to this illustrious list of pioneers and experts. During the infamous decade of the 2000s she published quite an impressive series of papers<sup>14</sup> about various research subjects, ranging from general security<sup>15</sup> topics, to low-level Windows kernel exploits<sup>16</sup>, to memory forensics<sup>17</sup>, and to USB security<sup>18</sup>. She is also the founder and main developer behind Qubes

OS<sup>19</sup>, described as a “reasonably secure operating system”, and praised by none less than Edward Snowden<sup>20</sup> himself.

Needless to say, Ms Rutkowska has reached the status of a living legend in the field. And today we are going to focus our attention in one of those papers: “Intel x86 considered harmful”<sup>21</sup>, published in October 2015.

(Spoiler alert: no, this is not about the Pentium FDIV bug<sup>22</sup> of 1994, already quite the damning record for Intel, if you ask me. This is much worse; sensible readers beware.)

It is, surprisingly, quite a readable paper for a software person like me, without an extended knowledge of hardware minutia. Her text starts with a very thoughtful and philosophical discussion about what the words “trust” and “trustworthy” mean; a fascinating topic we dedicated a whole issue<sup>23</sup> to last April. Why and how do we consider a particular system trustworthy? Ms Rutkowska’s point of view is overwhelming and obliterating, in a quote we have shared previously<sup>24</sup> in this magazine:

*The word “trusted” is a sneaky and confusing term: many people get a warm fuzzy feeling when they read it, and it is treated as a good thing. In fact the opposite is true. Anything that is “trusted” is a potentially lethal enemy of any secure system. (...)*

*The Operating System’s kernel, drivers, networking- and storage-subsystems are typically considered trusted in most contemporary mainstream operating systems such as Windows, Mac OSX and Linux; with Qubes OS being a notable exception. This means the architects of these systems assumed none of these components could ever get compromised or else the security of the whole OS would be devastated. (...) Quite an assumption indeed!*

Well, there you go Q, you clearly trusted your MI6 systems too much. (Well, to his defense, we can say the movie was filmed in 2012 and this paper was released 3 years later.)

She argues in her paper that modern Intel x86 platforms cannot be considered trustworthy due to their complex firmware, a series of never ending and persistent vulnerabilities in the boot process, and even worse, the introduction of opaque hardware components like the Intel Management Engine<sup>25</sup>.

(Those readers with good memory will also remember the curious fact that researchers discovered<sup>26</sup> in 2017 that the Intel Management Engine had an embedded version of Minix 3<sup>27</sup> running inside of it, a fact never fully disclosed by Intel themselves. Minix, for those unaware, is a famous “Unix-like” operating system originally written in the 1980s by professor Andrew Tanenbaum<sup>28</sup> to teach his operating systems class in the Vrije Universiteit Amsterdam<sup>29</sup>. But as usual, I digress.)

The paper then dives into security aspects of the boot process of Intel CPUs, including BIOS, UEFI, SMM, and how insecure the whole ensemble is at the end, even providing some recommendations about how to secure your Intel systems to the maximum possible extent (which, she argues, it is not much). She gave a conference talk precisely around the subject of “reasonably trustworthy x86 laptops”<sup>30</sup> in December 2015, right after the publication of this paper, also worth a watch.

Next comes a discussion about peripherals and their vulnerabilities, including network devices (again, Q, read this paper, please). And right after, a chapter dedicated to the Intel Management Engine, a topic she had been studying<sup>31</sup> and lecturing<sup>32</sup> about for quite a while at the time of the publication of this paper.

*Intel ME is very much similar to the previously discussed SMM. Like SMM it is running all the time when the platform is running (but unlike SMM can also run when the platform is shut down!). Like SMM it is more privileged than any system software running on the platform, and like SMM it can access (read or write) any of the host memory, unconstrained by anything.*

Yes, the mere fact of plugging your laptop to the main electricity socket is enough for the Intel Management Engine to kick off. And no, it cannot be disabled, leading to what Ms Rutkowska calls the “zombification” of general-purpose operating systems, and to the *de facto* existence of what she calls “an ideal rootkiting infrastructure”:

*When reading through the “ME Book” it is quite obvious that Intel believes that 1) ME, which includes its own custom OS and some critical applications, can be made substantially more secure than any other general purpose system software written by others, and 2) ultimately all security-sensitive computing tasks should be moved away from the general purpose OSes, such as Windows, to the ME, the only-one-believed-to-be-secure-island-of-trust...*

*There is another problem associated with Intel ME: namely it is just a perfect infrastructure for implanting targeted, extremely hard (or even impossible) to detect rootkits (targeting “the usual suspects”).*

You get the idea. I can only recommend diving into this fascinating text (particularly if you land the job of quartermaster at MI6) and, well, maybe choosing another architecture than x86 for your next laptop, if all else fails. Because according to Ms Rutkowska, the situation is not really different with AMD processors:

*And it seems AMD has an equivalent of Intel ME also, just disguised as Platform Security Processor (PSP).*

To finish this joyful edition of *De Programmatica Ipsum* dedicated to the topic of “Considered Harmful”, we can recommend Dijkstra’s own “Go To Statement Considered Harmful”<sup>33</sup>; then, continue with “‘Stored Program Concept’ Considered Harmful: History and Historiography”<sup>34</sup> by Hutchinson et al. Let us not forget about “Debunking the ‘Expensive Procedure Call’ Myth, Or Procedure Call Implementations Considered Harmful, Or Lambda: The Ultimate GOTO”<sup>35</sup> by Guy Steele Jr. Also, “Recursive Make Considered Harmful”<sup>36</sup> by Peter Miller, “Prototyping Considered Dangerous”<sup>37</sup> by Michael Atwood et al., “Global Variable Considered Harmful”<sup>38</sup> by Wulf and Shaw, “GOFBI Considered Harmful”<sup>39</sup> by Drew McDermott, and “Electron considered harmful”<sup>40</sup> by Drew DeVault.

After reading these pieces, however, you will have the eerie feeling that pretty much anything related to computers should be considered harmful. And you might be right. *Le sigh.*

Cover photo by the author.

## REFERENCES

- <sup>1</sup> <https://en.wikipedia.org/wiki/Skyfall>
- <sup>2</sup> <https://www.youtube.com/watch?v=aApTVqeGJMw>
- <sup>3</sup> [https://en.wikipedia.org/wiki/Joanna\\_Rutkowska](https://en.wikipedia.org/wiki/Joanna_Rutkowska)
- <sup>4</sup> [https://en.wikipedia.org/wiki/Evil\\_maid\\_attack](https://en.wikipedia.org/wiki/Evil_maid_attack)
- <sup>5</sup> <https://en.wikipedia.org/wiki/2000s>
- <sup>6</sup> <https://deprogrammaticaipsum.com/microsofts-writings-on-security/>
- <sup>7</sup> <https://deprogrammaticaipsum.com/sniffing-packets/>
- <sup>8</sup> <https://deprogrammaticaipsum.com/david-rice/>
- <sup>9</sup> <https://deprogrammaticaipsum.com/the-great-rewriting-in-rust/>
- <sup>10</sup> [https://en.wikipedia.org/wiki/Cassandra\\_\(metaphor\)](https://en.wikipedia.org/wiki/Cassandra_(metaphor))
- <sup>11</sup> <https://deprogrammaticaipsum.com/on-modern-security-culture/>
- <sup>12</sup> [https://en.wikipedia.org/wiki/Bruce\\_Schneier](https://en.wikipedia.org/wiki/Bruce_Schneier)
- <sup>13</sup> <https://deprogrammaticaipsum.com/secure-development-is-dead-long-live-secure-development/>
- <sup>14</sup> <https://blog.invisiblethings.org/papers/>
- <sup>15</sup> <https://theinvisiblethings.blogspot.com/2008/09/three-approaches-to-computer-security.html>
- <sup>16</sup> <https://blackhat.com/presentations/bh-usa-07/Rutkowska/Presentation/bh-usa-07-rutkowska.pdf>
- <sup>17</sup> <https://docs.huihoo.com/blackhat/dc-2007/bh-dc-07-rutkowska-beyond-the-cpu-defeating-hardware-based-ram-acquisition-tools.pdf>
- <sup>18</sup> <https://blog.invisiblethings.org/2011/05/31/usb-security-challenges.html>
- <sup>19</sup> <https://www.qubes-os.org/>
- <sup>20</sup> [https://en.wikipedia.org/wiki/Edward\\_Snowden](https://en.wikipedia.org/wiki/Edward_Snowden)
- <sup>21</sup> [https://blog.invisiblethings.org/papers/2015/x86\\_harmful.pdf](https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf)
- <sup>22</sup> [https://en.wikipedia.org/wiki/Pentium\\_FDIV\\_bug](https://en.wikipedia.org/wiki/Pentium_FDIV_bug)
- <sup>23</sup> <https://deprogrammaticaipsum.com/issue/issue-079-trust/>
- <sup>24</sup> <https://deprogrammaticaipsum.com/who-do-you-trust/>
- <sup>25</sup> [https://en.wikipedia.org/wiki/Intel\\_Management\\_Engine](https://en.wikipedia.org/wiki/Intel_Management_Engine)
- <sup>26</sup> [https://troopers.de/downloads/troopers17/TR17\\_ME11\\_Static.pdf](https://troopers.de/downloads/troopers17/TR17_ME11_Static.pdf)
- <sup>27</sup> [https://en.wikipedia.org/wiki/Minix\\_3](https://en.wikipedia.org/wiki/Minix_3)
- <sup>28</sup> [https://en.wikipedia.org/wiki/Andrew\\_S.\\_Tanenbaum](https://en.wikipedia.org/wiki/Andrew_S._Tanenbaum)
- <sup>29</sup> [https://en.wikipedia.org/wiki/Vrije\\_Universiteit\\_Amsterdam](https://en.wikipedia.org/wiki/Vrije_Universiteit_Amsterdam)
- <sup>30</sup> <https://www.youtube.com/watch?v=rcwngbUrZNg>
- <sup>31</sup> <https://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf>
- <sup>32</sup> <https://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20slides.pdf>
- <sup>33</sup> <https://dl.acm.org/doi/10.1145/362929.362947>
- <sup>34</sup> [http://link.springer.com/10.1007/978-3-642-39053-1\\_28](http://link.springer.com/10.1007/978-3-642-39053-1_28)

<sup>35</sup> [https://ia802904.us.archive.org/17/items/bitsavers\\_mitaaimAI\\_1976403/AIM-443\\_text.pdf](https://ia802904.us.archive.org/17/items/bitsavers_mitaaimAI_1976403/AIM-443_text.pdf)

<sup>36</sup> <https://aegis.sourceforge.net/auug97.pdf>

<sup>37</sup> [https://link.springer.com/chapter/10.1007/978-1-5041-2896-4\\_30](https://link.springer.com/chapter/10.1007/978-1-5041-2896-4_30)

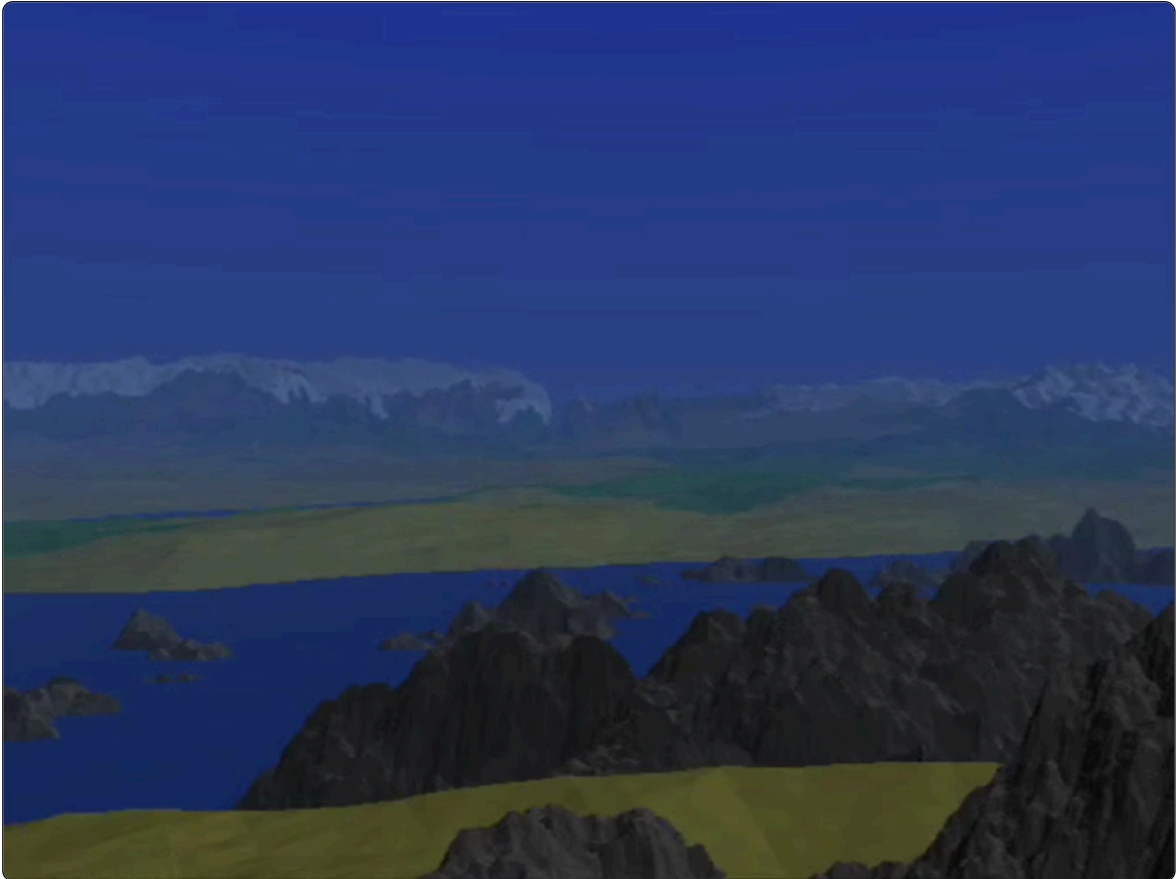
<sup>38</sup> <https://dl.acm.org/doi/10.1145/953353.953355>

<sup>39</sup> <https://www.cs.yale.edu/homes/dvm/papers/nogofai.pdf>

<sup>40</sup> <https://drewdevault.com/2016/11/24/Electron-considered-harmful.html>



# Loren Carpenter



By Adrian Kosmaczewski, April 6th, 2026

The year is 2026, and we take computer-generated movies for granted: Pixar, Illumination, DreamWorks, and a myriad of smaller studios delight us every year with more and more technical and storytelling prowess. Heck, we even have “artificial intelligence” systems that can generate whole movies out of a single “prompt” consisting of a certain amount of words that make a certain sense in a particular context. 50 years ago, however, the prospect of a computer generating images was the subject of intense scientific research.

The challenge was not small; the computers of that age were vastly underpowered compared to those of today, and the kind of images we could dream of required unknown algorithms and data structures, not to mention a non-existent, monumental amount of computing power. The latter problem would eventually and gradually be solved by Moore's Law; for the former part of the equation, that is, the algorithms and the data structures, we can thank the early research of this month's Vidéothèque subject, Loren Carpenter<sup>1</sup>.

Not to be confused with his namesake and contemporary director extraordinaire, John Carpenter<sup>2</sup> (nor, for that matter, with Karen and Richard Carpenter<sup>3</sup>, also active in the same era), the recently deceased Loren Carpenter<sup>4</sup> (he passed away last December 21st) paved the way for computer-generated movies as we know them today.

On July 14, 1980, at the SIGGRAPH<sup>5</sup> conference (a legendary focal point of research in the field of CGI), attendees became the privileged spectators of the first computer-generated movie of all time, "Vol Libre"<sup>6</sup>, a French title that translates as "Free Flight". This movie was a demonstration of the content of a paper titled "Computer rendering of fractal curves and surfaces"<sup>7</sup>, presented by Loren Carpenter at the same occasion.

It is important to set some context first, particularly for younger members of the audience: by the time Carpenter came into the scene, the "Star Wars" (1977) Death Star and trench run sequence<sup>8</sup> made by Larry Cuba<sup>9</sup> on a Digital Equipment Corporation PDP-11<sup>10</sup>, was considered the pinnacle of computer animation<sup>11</sup>.

The description of "Vol Libre" on Vimeo<sup>12</sup>, written by the author himself, says it all:

*I made this film in 1979-80 to accompany a SIGGRAPH paper on how to synthesize fractal geometry with a computer. It is the world's first fractal movie. It utilizes 8-10 different fractal generating algorithms. I used an antialiased version of this software to create the fractal planet in the Genesis Sequence of Star Trek 2, the Wrath of Khan. These frames were computed on a VAX-11/780 at about 20-40 minutes each.*

20 to 40 minutes per frame. Let that sink in. Do you know how fast your iPhone renders a full 4K movie on iMovie<sup>13</sup> these days?

In all honesty, “Vol Libre” looks dated to our modern eyes, slow, and clumsy. It lacks the polish and the finesse of a recent Pixar movie. But to the SIGGRAPH audience of 1980, after picking up their jaws from the floor, this short film represented actual proof that computer-generated movies were not only a theory but a reality (albeit a rudimentary one at the time). History<sup>14</sup> in the making; it is both wonderful and humbling to realize that a whole new industry debuted in a humble conference hall, not even 46 years ago.

*The audience erupted. The entire hall was on their feet and hollering. They wanted to see it again. “There had never been anything like it,” recalled Ed Catmull. Loren was beaming.*

(Quote from “Droidmaker” on “Vol Libre, an amazing CG film from 1980”<sup>15</sup> by Jason Kottke.)

Loren Carpenter was almost immediately hired by the aforementioned Ed Catmull<sup>16</sup> (recipient of the ACM Turing Award 2019<sup>17</sup>) to join an experimental unit of Lucasfilm called “Graphics Group”, dedicated to exploring the use of computers in moviemaking.

*“There was strategy in this,” said Loren, “because I knew that Ed and Alvy were going to be in the front row of the room when I was giving this talk.” Everyone at Siggraph knew about Ed and Alvy and the aggregation at Lucasfilm. They were already rock stars. Ed and Alvy walked up to Loren Carpenter after the film and asked if he could start in October.*

(Another quote from “Droidmaker” relayed on the same Jason Kottke blog post.)

George Lucas, probably rather skeptical of the whole concept or blissfully unaware of Moore’s Law, would later decide to spin off this unit a few years later, selling it to a certain Steve Jobs<sup>18</sup> (who had just been sidelined at Apple), becoming Pixar<sup>19</sup> in the process.

(Ironically enough, and following the steps of Larry Cuba, the “Star Wars” prequels and sequels would include an insane amount of computer-generated imagery, including a creepy recreation of a young Carrie Fisher<sup>20</sup> in the otherwise magnificent “Rogue One”<sup>21</sup>, to the outcry and indignation of most of its fanbase and critics alike. The fact that BB-8<sup>22</sup> and Baby Yoda<sup>23</sup> were instead filmed as physical props on set says a lot about how the audiences reacted to CGI. But I digress, again.)

The rest, as they say, is history. As he explained above, Loren Carpenter would reuse the same engine created for “Vol Libre” in the final terraforming scene of “Star Trek II: The Wrath of Khan”<sup>24</sup> released in 1982 and arguably the best movie in the Star Trek franchise. Later would come “The Adventures of André & Wally B.”<sup>25</sup> in 1984, “Luxo Jr.”<sup>26</sup> in 1986, “Red’s Dream”<sup>27</sup> in 1987, and finally “Tin Toy”<sup>28</sup> in 1988. This last one would become the first computer-generated movie to win the Academy Award for the “Most Disturbing”<sup>29</sup> Baby Ever Shown in a Movie Picture” category, and it foreshadowed a fully-fledged franchise known as “Toy Story”<sup>30</sup> since 1995.

Meanwhile, the work of Loren Carpenter set the basis not only for the algorithms and data structures required to create a movie inside the memory chips of a computer but also to streamline the industrial processes used by Pixar, and later by all of its competitors, to produce movies. Among the research papers<sup>31</sup> authored or co-authored by Carpenter, we must mention “Computer Rendering of Stochastic Models”<sup>32</sup>, “Volume rendering”<sup>33</sup>, “The Reyes image rendering architecture”<sup>34</sup>, “Distributed ray tracing”<sup>35</sup>, and “The A -buffer, an antialiased hidden surface method”<sup>36</sup>.

Next time you go to a theater to watch a computer-generated movie, pay attention: we not only have standard-length feature films released simultaneously in various locales, but the characters are actually moving their lips as if they spoke those words in those other languages than English. The labels on things and locations reflect local cultural brands and expressions, different for each region of the world. The animation is buttery smooth, well over the standard 24 frames per second of celluloid film stock productions. And they can even be rendered in 3-D, if needed.

The magic is complete, down to the smallest details, and Loren Carpenter had a lot to do with that.

Watch the second of this month's Vidéotheque movies, "Vol Libre" by Loren Carpenter, on Vimeo<sup>37</sup> or on YouTube<sup>38</sup>. After watching this, do not miss this gem: "Loren Carpenter Experiment at SIGGRAPH '91"<sup>39</sup>. You will thank me later.

Cover snapshot chosen by the author.

## REFERENCES

- <sup>1</sup> [https://en.wikipedia.org/wiki/Loren\\_Carpenter](https://en.wikipedia.org/wiki/Loren_Carpenter)
- <sup>2</sup> [https://en.wikipedia.org/wiki/John\\_Carpenter](https://en.wikipedia.org/wiki/John_Carpenter)
- <sup>3</sup> [https://en.wikipedia.org/wiki/The\\_Carpenters](https://en.wikipedia.org/wiki/The_Carpenters)
- <sup>4</sup> <https://history.siggraph.org/person/loren-c-carpenter/>
- <sup>5</sup> <https://en.wikipedia.org/wiki/SIGGRAPH>
- <sup>6</sup> <https://vimeo.com/5810737>
- <sup>7</sup> <https://dl.acm.org/doi/10.1145/965105.807478>
- <sup>8</sup> <https://www.youtube.com/watch?v=m8aYL2l5quU>
- <sup>9</sup> <https://www.evl.uic.edu/events/2093>
- <sup>10</sup> <https://en.wikipedia.org/wiki/PDP-11>
- <sup>11</sup> [https://en.wikipedia.org/wiki/Timeline\\_of\\_computer\\_animation](https://en.wikipedia.org/wiki/Timeline_of_computer_animation)
- <sup>12</sup> <https://vimeo.com/5810737>
- <sup>13</sup> <https://apps.apple.com/us/app/imovie/id377298193>
- <sup>14</sup> <https://www.historyofcg.com/pages/vol-libre/>
- <sup>15</sup> <https://kottke.org/09/07/vol-libre-an-amazing-cg-film-from-1980>
- <sup>16</sup> [https://en.wikipedia.org/wiki/Edwin\\_Catmull](https://en.wikipedia.org/wiki/Edwin_Catmull)
- <sup>17</sup> <https://awards.acm.org/about/2019-turing>
- <sup>18</sup> <https://deprogrammaticaipsum.com/steve-jobs/>
- <sup>19</sup> <https://en.wikipedia.org/wiki/Pixar>
- <sup>20</sup> [https://en.wikipedia.org/wiki/Carrie\\_Fisher](https://en.wikipedia.org/wiki/Carrie_Fisher)
- <sup>21</sup> [https://en.wikipedia.org/wiki/Rogue\\_One](https://en.wikipedia.org/wiki/Rogue_One)
- <sup>22</sup> <https://en.wikipedia.org/wiki/BB-8>
- <sup>23</sup> <https://en.wikipedia.org/wiki/Grogu>
- <sup>24</sup> [https://en.wikipedia.org/wiki/Star\\_Trek\\_II:\\_The\\_Wrath\\_of\\_Khan](https://en.wikipedia.org/wiki/Star_Trek_II:_The_Wrath_of_Khan)
- <sup>25</sup> [https://en.wikipedia.org/wiki/The\\_Adventures\\_of\\_Andr%C3%A9\\_%26\\_Wally\\_B](https://en.wikipedia.org/wiki/The_Adventures_of_Andr%C3%A9_%26_Wally_B).
- <sup>26</sup> [https://en.wikipedia.org/wiki/Luxo\\_Jr](https://en.wikipedia.org/wiki/Luxo_Jr).
- <sup>27</sup> [https://en.wikipedia.org/wiki/Red%27s\\_Dream](https://en.wikipedia.org/wiki/Red%27s_Dream)
- <sup>28</sup> <https://www.youtube.com/watch?v=DWi2WTqD59A>
- <sup>29</sup> <https://www.youtube.com/watch?v=DWi2WTqD59A>
- <sup>30</sup> [https://en.wikipedia.org/wiki/Toy\\_Story](https://en.wikipedia.org/wiki/Toy_Story)
- <sup>31</sup> [https://web.archive.org/web/20161204140529/https://graphics.pixar.com/library/indexAuthorLoren\\_Carpenter.html](https://web.archive.org/web/20161204140529/https://graphics.pixar.com/library/indexAuthorLoren_Carpenter.html)
- <sup>32</sup> <https://dl.acm.org/doi/10.1145/358523.358553>
- <sup>33</sup> <https://dl.acm.org/doi/10.1145/54852.378484>
- <sup>34</sup> <https://dl.acm.org/doi/10.1145/37401.37414>
- <sup>35</sup> <https://dl.acm.org/doi/10.1145/800031.808590>
- <sup>36</sup> <https://dl.acm.org/doi/10.1145/800031.808585>
- <sup>37</sup> <https://vimeo.com/5810737>
- <sup>38</sup> <https://www.youtube.com/watch?v=eSC5-rWKvEY>

<sup>39</sup> <https://vimeo.com/78043173>