

# The History of Software Engineering

Grady Booch

**THE FIRST COMPUTERS** were human (and for the most part, women). The term “digital” didn’t enter circulation until around 1942, when George Stibitz took the ideas from another George (Boole) and applied them to electromechanical devices. It took another decade for John Tukey to popularize the term “software.” What, then, of the term “software engineering”?

## The Origins of the Term

Many suggest it came from the 1968 NATO Conference on Software Engineering, coined by Friedrich Bauer. Others have pointed to the 1966 letter by Anthony Oettinger in *Communications of the ACM*, wherein he used the term “software engineering” to make the distinction between computer science and the building of software-intensive systems.<sup>1</sup> Even earlier, in the June 1965 issue of *Computers and Automation*, there appeared a classified ad seeking a “systems software engineer.”

All the data I have points to Margaret Hamilton as the person who first coined the term. Having worked on the SAGE (Semi-automatic Ground Environment) program, she became the lead developer for *Skylab* and *Apollo* while working at the Draper Lab. According to an (unpublished) oral history, she began to use the term “software engineering” sometime in

1963 or 1964 to distinguish her work from the hardware engineering taking place in the nascent US space program.

## Software Engineering versus Computer Science

Grace Hopper suggested that programming is a practical art; Edsger Dijkstra called the art of programming the art of organizing complexity; Donald Knuth referred to programming as art because it produced objects of beauty. I suspect that all of these observations are true, but what I like best is David Parnas’s observation—much like Anthony Oettinger’s—that there is a distinction between “computer science” and the other stuff that we do. This is not unlike the distinction between chemical engineering and chemistry: both are valid; both have their particular sets of practices; both are very different things. Software engineering is, in my experience, equally an art and a science: it is the art of the practical.

Engineering in all fields is all about the resolution of forces. In civil engineering, one must consider static and dynamic forces of a physical nature and of human nature. In software engineering, one also must balance cost, schedule, complexity, functionality, performance, reliability, and security, as well as legal and ethical forces. Computing technology has certainly changed since the

time of Charles Babbage. However, the fundamentals of engineering hold true, although, as we shall see, each age discovers some new truth about engineering software.

## From the 19th to the 20th Century: Human Computers

Ada Lovelace was perhaps the first person to understand that programming was a thing unto itself. Around that same time, George Boole brought a new way of thinking to the mathematicians and philosophers of the world, as expressed in his classic book *The Laws of Thought*.<sup>2</sup> At the end of the 19th century, we saw the first human computers, such as Annie Cannon, Henrietta Leavitt, and others, the so-called “Harvard Computers” working for the astronomer Edward Pickering. The way these women organized their work was astonishingly similar to contemporary agile development practices; they too had a different way of thinking, very different for their time.

Around the start of the new century, as computational problems began to scale up and as mechanical aids to calculation became more reliable and economical, the process of computing underwent further regimentation. It was common to see large rooms filled with human computers (again, mostly women), all lined up in rows. Data would enter

one end; a computer would carry out one operation and then pass the result to the next computer. This was in effect the organic manifestation of what today we'd call a pipeline architecture.

### From the Great Depression to World War II: Birth of the Electronic Computer

Efficiency and the reduction of costs were then, as they are now, important to every industrial process. So, we saw people such as Frederick Taylor and Frank and Lillian Gilbreth (of *Cheaper by the Dozen*<sup>3</sup> fame) introduce time and motion studies. The Gilbreths also promoted the concept of process charts—the direct predecessor of flowcharts—to codify industrial processes. It did not take long for these same ideas in manufacturing to jump over to the problems of computing.

As the global Great Depression took hold, the Works Progress Administration was launched as part of President Roosevelt's New Deal. Gertrude Blanche was put in charge of the Mathematical Tables Project, the predecessor of today's *Handbook of Mathematical Functions*. This was a work relief project that employed hundreds of out-of-work mathematicians and computers (again, mostly women). Blanche's work developed best practices for human computing that were extremely sophisticated, including mechanisms for error checking, which influenced the way early punched-card computing evolved. In 1940, Wallace Eckert published *Punched Card Methods in Scientific Computing*,<sup>4</sup> which turned out to be, in a manner of speaking, the first computing methodology or pattern language.

As the winds of war were gathering in Europe, George Stibitz applied

George Boole's ideas of binary logic to build the first digital adder made of electromechanical relays. He called this the K Model (the K representing the kitchen table on which he built it), and thus digital computing was born. The idea of building electromechanical mechanisms for computation spread rapidly, and it was not long thereafter that others realized that relays could be replaced by vacuum tubes, which were much, much faster. In the summer of 1944, a serendipitous meeting between John von Neumann (who

from a machine's hardware. This led to one of the first instances of abstraction in programming, the idea that one could devise a programming language at a level closer to human expression and further from the machine's hardware. Furthermore, as Hopper realized, one could use the computer itself to translate those higher-order expressions into machine language; the compiler was born.

In the lamentations of World War II, the computing world split into three pieces. In Germany, there was

Ada Lovelace was perhaps the first person to understand that programming was a thing unto itself.

at the time was working on the Manhattan Project) and Herman Goldstine (who was working at the Ballistic Research Laboratory) led to their connection with John Mauchly (a professor at the Moore School of Electrical Engineering). This caused ENIAC (Electronic Numerical Integrator and Computer) to come into prominence and, more important, later yielded the *First Draft of a Report on the EDVAC* (Electronic Discrete Variable Automatic Computer).<sup>5</sup>

And thus was born a new way of thinking: the concept of a programmable, electronic computer with its instructions stored in memory.

Grace Hopper, very much in the spirit of Ada Lovelace, then rediscovered the idea that software could be a thing unto itself, distinct

Konrad Zuse. In a different time and place, his work would have been the center of gravity of modern computing, for he invented the first high-order programming language as well as the first general-purpose stored computer.

In England, there was Bletchley Park, where Alan Turing laid the theoretical foundations for modern computer science. However, it took an engineer—most notably Tommy Flowers—to turn those theories into pragmatic solutions, and from this Colossus was born. Dorothy Du Boisson, a human computer, served as the primary operator of Colossus. In her experience of leading a team of women who operated Colossus, she codified the ideas of workflow that eventually were programmed into the machine itself.

In the US, ENIAC, then later EDVAC, dominated the scene. Initially, “programming” was carried out by wiring up plugboards, a task carried out by human computers (yet again, mostly women), such as Kay Antonelli, Betty Snyder, Frances Spence, Ruth Teitelbaum, and Marlyn Wescoff. The way they organized their work was reminiscent of the Harvard Computers and thus, in a manner of speaking, anticipated the structure of contemporary small development teams focused on continuous integration.

### Post World War II: Rise of Computing and Birth of Software Engineering

The technical and economic forces that would shape modern software engineering further coalesced in the economic rise at the end of World War II, where we began to see computing applied to problem domains beyond the needs of conflict. Herman Goldstine built on the ideas of the Gilbreths and, together with John von Neumann, invented a notation that eventually morphed into what today we call flowcharts. Maurice Wilkes, David Wheeler, and Stanley Gill invented the concept of subroutines, thus again raising computing’s levels of abstraction, and making manifest the pragmatics of algorithmic decomposition. John Backus took Grace Hopper’s early work and went further, yielding Fortran, the high-level imperative language that would dominate scientific computing for years to come.

The commercial world, now unleashed at the end of global conflict, turned to automatic aids to computing: opportunities for growth quickly outran the cost and reliability of human computers. The first computer put in commercial use was

the Lyons Electronic Office (LEO). John Pinkerton, LEO’s chief engineer, had the insight that software could be treated as a component unto itself. Realizing that many low-level programming tasks kept being written over and over again, he began to bundle these common routines into libraries, forming what today we’d call an operating system or framework, yet another rise in programming’s levels of abstraction.

Grace Hopper, Robert Bemer, Jean Sammet, and others, influenced by John Backus’s work, created Cobol, another imperative language, focused on the needs of businesses. With the introduction of IBM’s System/360, it was now possible to write software for more than one specific machine. IBM’s decision to unbundle software from hardware was a transformative event: now it was possible to develop software as a component that had individual economic value. Around this time, organizations such as SHARE emerged—a predecessor of today’s open source software movement—giving a platform for third parties to write software for hardware they themselves didn’t control. In the UK, Dina St. Johnson seized on the business opportunity and established England’s first software services business. This made manifest the idea that one could outsource software development to teams with particular computing skills a company with specific domain knowledge might not possess.

### Rise of the Cold War: Coming of Age

The rise of the Cold War between the US and the Soviet Union generated another set of forces that pushed software engineering to come of age. Tom Kilburn and his work with Whirlwind explored the possibilities of real-time

programming, and that work led directly to the SAGE system. Constructed as a defense against the Soviet threat of sending nuclear-armed bombers over the Arctic, SAGE led to a number of important innovations and issues, including

- human–computer interfaces using CRT displays and light pens,
- the institutionalization of core memory, and
- the problems associated with building very large software systems in a distributed environment.

Software development was no longer just a small part of bringing a computer to life; it was increasingly a very expensive part, and certainly the most important part.

So there we were, in the second half of the 1960s, with the confluence of three important events in the history of software:

- the rise of commercial software as a product unto itself,
- the complexities of defense systems such as SAGE, and
- the rise of human-critical software as demanded by the US space program.

This is the context in which Margaret Hamilton coined the term “software engineering” and in which NATO declared that there was a “software crisis.”

A sort of programming priesthood was the common form of software development at the time, and—in its time—it made a great deal of sense. In that era, the cost of a computer was greater than the cost of its programmers, and as such, computers would be kept apart in a climate-controlled room. Much like the

pipelined methods of the punched-card era, analysts would take requirements and pass them on to programmers, who would use their flowcharts to devise algorithms. These programmers in turn would pass their programs on to keypunchers. The resulting card decks would be given to the computer operators working in their sacred space.

It wasn't until the economics of computers changed with the rise of minicomputers and microcomputers, together with the realization of Christopher Strachey's idea of time sharing, that this model of development changed. This is also the context in which the basic principles of software project management came alive, as Fred Brooks so profoundly described in *The Mythical Man Month*.<sup>6</sup> Brooks made the important insight that software engineering was not just a technical process but also a very human process.

The economic rise after World War II, given a further boost by the Cold War, led inevitably to a counterculture shift, as wonderfully described by John Markoff in *What the Dormouse Said*.<sup>7</sup> The introduction of personal computing not only was fueled by technical and social advances but also changed the nature of software engineering. Now, programmers were more expensive than computers, and it was economically viable to put computers everywhere. This led to Allen Newell speaking of the enchanted world that computing made possible, as described in his wonderful essay "Fairytale."<sup>8</sup>

### From the Sixties to the Eighties: Maturation

Software engineering was forced to mature. Larry Constantine was perhaps the first to introduce the concept of modular programming, with

the ideas of coupling and cohesion applied as a mechanism for algorithmic decomposition. Edsger Dijkstra took a more formal approach, giving us an important tool for software engineering: the idea of structured programming.

Around the same time, there was important work by researchers such as Robert Floyd and Tony Hoare, who devised formal ways to express and reason about programs—a true attempt to connect computer science and software engineering. Niklaus

artifacts and the processes of software development.

This led to the first generation of software engineering methodologies. Doug Ross, Larry Constantine, Ed Yourdon, Tom DeMarco, Chris Gane, Trish Sarson, and Michael Jackson—to name just a few—developed methods for structured analysis and design that took over the field. Adding the work by Michael Fagan (on software inspections), James Martin (on information engineering), John Backus (on functional programming), and Leslie

The introduction of personal computing changed software engineering.

Wirth invented Pascal, an effort to explicitly support best practices in structured programming. Ole Dahl and Kristen Nygaard had the outrageously wonderful idea that yielded the invention of Simula, a language that was object-oriented rather than algorithmic in nature.

Winston Royce then brought to us the idea of a formal software development process. Although he is much criticized for what we today call the waterfall process, his methodology was actually quite advanced: he spoke of iterative development, the importance of prototyping, and the value of artifacts beyond source code itself. Coupled with David Parnas's ideas of information hiding, Barbara Liskov's ideas of abstract data types, and Peter Chen's approaches to entity-relationship modeling, all of a sudden the field had a vibrant set of ideas whereby to express the

Lamport (on best practices for distributed computing), software engineering entered in its first golden age.

### The Eighties and Onward: Golden Age

However, a sea change was coming. Owing to the growing problems of software quality, the rise of ultralarge software-intensive systems, the globalization of software, and the shift from programs to distributed systems, new approaches were needed. Ole Dahl and Kristen Nygaard's ideas of object-oriented programming gave rise to a completely new class of programming languages: Smalltalk, C with Classes, Ada, and many others. Although structured methods were useful, they were not altogether sufficient for these new languages, and thus was born the second golden age of software engineering.

Ada—the US Department of Defense’s solution to the problem of the proliferation of programming languages and the changing nature of software itself—proved to be a catalyst for this era. Some of the structured-method pioneers pivoted. James Martin and Ed Yourdon celebrated object-oriented approaches; others brought completely new ideas to the field: Stephen Mellor, Peter Coad, and Rebecca Wirfs-Brock, to name a few. The Booch Method grew out of this primordial soup of ideas, as did Jim Rumbaugh’s OMT (object-modeling technique) and Ivar Jacobson’s Objectory. Sensing an opportunity to bring the market to some common best practices, the three of us united to produce what became the Unified Modeling Language (made an Object Management Group standard in 1987) and then the Unified Process.

Other aspects of software engineering came into play—for example,

- Philippe Kruchten’s 4+1 View Model of software architecture;
- Barry Boehm’s work in software economics, together with his spiral model;
- Vic Basili and his ideas on empirical software engineering;
- Capers Jones and software metrics;
- Harlan Mills and clean-room software engineering;
- Donald Knuth’s literate programming; and
- Watts Humphrey and his Capability Maturity Model.

Simultaneously, these software engineering concepts influenced the development of an entirely new generation of programming languages. Bjarne Stroustrup’s C with Classes grew up to become C++, which later

influenced the creation of Java. Alan Cooper’s Visual Basic invigorated the Windows platform. Brad Cox’s invention of Objective-C had a tremendous effect on NeXT and Apple. Furthermore, Cox’s ideas surrounding component-based engineering—another rise in software engineering’s levels of abstraction—led directly to Microsoft’s OLE (object linking and embedding) and COM (Component Object Model), which were the predecessors of today’s microservice architecture.

### **The Nineties and the Millennium: Era of Disruptions**

But another change was in the wind: the Internet. Suddenly we had a very rich, as of yet unexplored, platform. In it, distribution was the default, consumers were the new stakeholders, users were measured in the billions, and participants in this ecosystem were not necessarily reliable or trustworthy. We were no longer building programs; we were building systems, often made of parts that we no longer controlled.

By this time, there existed a relatively stable and economically very vibrant software engineering community. Independent companies existed to serve the needs of requirements analysis, design, development, testing, and configuration management. Continuous integration with incremental and iterative development was becoming the norm. The Gang of Four—Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides—gave us another bump up in software engineering levels of abstraction in the form of the design pattern. Institutionalized by the Hillside Group in 1993, patterns heavily influenced that generation of software development. Jim Coplien took the ideas of software

design patterns and applied them to organizational patterns. Mary Shaw and David Garlan furthered these concepts in their work on software architecture styles.

Two other lasting developments of note took place in this era. First, Eric Raymond evolved an important legal framework for open source, making it possible to scale the ideas first seen in the early days of computing, with SHARE. Kiran Karnik, working in India, established the first outsourcing contracts between General Electric and India, thus laying the foundation for a transformative economic shift in software development.

With the Internet well in place and organizations beginning to embrace its possibilities, mobile devices hit the scene, and the world changed yet again. The foundation laid by Brad Cox for component-based engineering morphed into service-based architectures, which in turn morphed into microservice architectures, evolving as the Web’s technical infrastructure grew in fits and starts. New programming languages came and went (and still do), but only a handful still dominate—for example, Java, JavaScript, Python, C++, C#, PHP, and Swift. Computing moved from the mainframe to the datacenter to the cloud, but coupled with microservices, the Internet evolved to become the de facto computing platform. Company-specific ecosystems rose like walled cathedrals: Amazon, Google, Microsoft, Facebook, Salesforce, IBM—really, every economically interesting company built its own fortress.

This was now the age of the framework. Long gone were the religious battles over operating systems. Now, battles were fought



along the lines of the veritable explosion of open source frameworks: Bootstrap, jQuery, Apache, NodeJS, MongoDB, Brew, Cocoa, Caffe, Flutter—truly a dizzying, ever-growing collection.

Today, we no longer build just programs or monolithic systems; we build apps that live on the edge and interact with these distributed systems. Agile methods—in various personality-led variations—have flowered and have become the dominant method, in name if not necessarily perfectly in practice. Hirotaka Takeuchi and Ikujiro Nonaka coined the term “Scrum” in 1986 as an agile approach to product development. Later, Ken Schwaber and (independently) Jeff Sutherland and Jeff McKenna codified those principles in the domain of software development. Around that same time, Kent Beck introduced the concept of Extreme Programming, while Ralph Johnson further developed the idea of refactoring (which Martin Fowler further codified in his book *Refactoring: Improving the Design of Existing Code*). In February 2001, 17 agilists met in Snowbird, Utah, and penned the Agile Manifesto. The agile approach to software development entered the mainstream.

Software engineering had entered another golden age. Git and GitHub emerged; Joel Spolsky gave us Stack Overflow; Jeannette Wing introduced the idea of computational thinking; Andrew Shafer and Patrick Debois brought us the idea of DevOps; the full stack developer became a thing; the Internet of Things appeared in every imaginable corner of the world. Now, all of a sudden, everyone could learn how to code (and many did).

Artifacts such as SWEBOK (Software Engineering Body of

Knowledge, first released in 2004 and whose current version was released in 2014)<sup>9</sup> and the Systems Engineering Body of Knowledge by INCOSE<sup>10</sup> exist as an attempt to codify software engineering best practices.

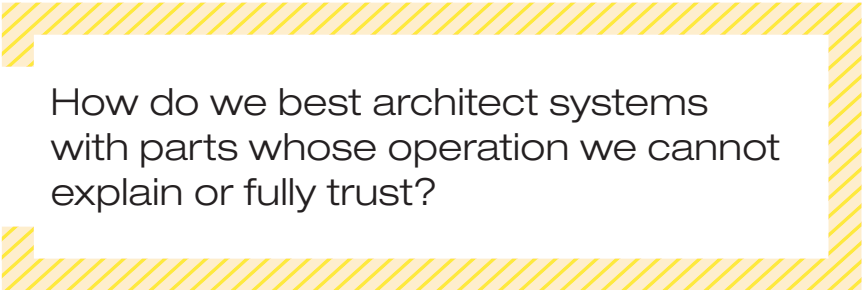
### The Decade Ahead: Big Data and the New Season of AI

But software engineering is about to undergo yet another change.

The foundations of AI have been around for decades. Over the decades, we’ve seen at least four seasons of AI, manifested by the extreme rising and falling of fortunes. What we have now feels different. The growth of big data, the abundance of raw computational power,

We as an industry have not yet built enough of these AI systems to fully understand how they might impact the software engineering process, as they most certainly will. What is the best lifecycle for systems whose components we teach, rather than program? How do we test them? Where does configuration management fit in when data for ground truth is perhaps more important than the neural network itself? How do we best architect systems with parts whose operation we cannot explain or fully trust?

This will be the challenge of the next generation of women and men who keep software engineering vibrant. Add to this mix the growth of quantum computing,



How do we best architect systems with parts whose operation we cannot explain or fully trust?

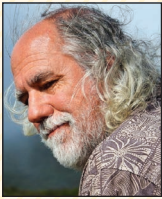
and the presence of these walled cathedrals have given rise to economic forces that have made first statistical approaches and now neural networks viable. Most of these modern advances have been in what I call “signal AI”: the use of neural networks and gradient descent to do complex pattern matching in images, video, and audio signals. The early outcomes are impressive, as evidenced in IBM’s Watson and Google’s AlphaGo. In many ways, we are just beginning to understand what is possible and where the limits of these connectionist models of computation live.

augmented reality, virtual reality, and the spread of computing to every human, every device, and every nook and cranny of the earth and beyond, and this makes for a tremendously exciting time to be in computing.

In the history of computing, we have seen the progression of systems from mathematical, to symbolic, to what Yuval Harari calls “imagined realities.” Some software is like building a doghouse: you just do it, without any blueprints, and if you fail, you can always get another dog. Other software is like building a house or a



## ABOUT THE AUTHOR



**GRADY BOOCH** is an IBM Fellow and one of UML's original authors. Contact him at [egrady@booch.com](mailto:egrady@booch.com) and on Twitter as [@egrady\\_booch](https://twitter.com/egrady_booch).

high-rise: the economics are different, the scale is different, and the cost of failure is higher. Much of modern software engineering is like renovating a city: there is room for radical innovation, but you are constrained by the past as well as the cultural, social, ethical, and moral context of everyone else in the city.

One thing I do know. No matter the medium or the technology or the domain, the fundamentals of sound software engineering will always apply: craft sound abstractions; maintain a clear separation of concerns; strive for a balanced distribution of responsibilities; seek simplicity. The pendulum will continue to swing—symbolic to connectionist to quantum models of computation; intentional architecture or emergent architecture; edge or cloud computing—but the fundamentals will stand.

**I** have named a few dozen women and men who have shaped software engineering, but please know that there are thousands more who have made software engineering what it is today, each by his or her own unique contributions. And so it will be for the future of software engineering. As I said in closing in my

keynote at the 2015 International Conference on Software Engineering in Florence, software is the invisible writing that whispers the stories of possibility to our hardware.

And you are the storytellers. 

### Acknowledgments

This essay is based on my ACM Learning Webinar of the same title, broadcast on 25 April 2018. A recording is available at <https://www.youtube.com/watch?v=QUz10Z1AfLc>. A more extensive bibliography is available as a Web Extra at <https://extras.computer.org/extra/mso2018050108s1.pdf>.

### References

1. A. Oettinger, "President's Letter to the ACM Membership," *Comm. ACM*, vol. 9, no. 8, 1966.
2. G. Boole, *The Laws of Thought*, Walton and Maberly, 1854.
3. F.B. Gilbreth Jr. and E. Gilbreth Carey, *Cheaper by the Dozen*, Thomas Y. Crowell, 1948.
4. W.J. Eckert, *Punched Card Methods in Scientific Computing*, Thomas J. Watson Astronomical Computing Bureau, Columbia Univ., 1940.
5. J. von Neumann, *First Draft of a Report on the EDVAC*, US Army Ordinance Department and Univ. Pennsylvania Moore School of Electrical Eng., 1945.
6. F. Brooks, *The Mythical Man Month: Essays on Software Engineering*, Addison-Wesley, 1975.
7. J. Markoff, *What the Dormouse Said*, Penguin, 2005.
8. A. Newell, "Fairytails," Carnegie Mellon Univ., 1976; <http://repository.cmu.edu/cgi/viewcontent.cgi?article=3321&context=compsci>.
9. *The Guide to the Software Engineering Body of Knowledge*, IEEE Computer Soc., 2004.
10. *Guide to the Systems Engineering Body of Knowledge (SEBoK)*, Int'l Council Systems Eng., 2012.

myCS

Read your subscriptions through the myCS publications portal at

<http://mycs.computer.org>