The History of Software Engineering

Grady Booch

The first computers were human (and for the most part, women)[1]. The term "digital" didn't enter circulation until around 1942, when George Stibitz took the ideas from another George (Boole) and applied them to electro-mechanical devices[2]. It took another decade for John Tukey to popularize the term "software"[3]. 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[4]. 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[5]. Even earlier, in the June 1965 issue of *Computers and Automation*, there appeared a classified ad seeking a "systems software engineer"[6].

All the data I have points to Margaret Hamilton as the person who first coined the term. Having worked on the SAGE program, she became the lead developer for Skylab and Apollo while working at the Draper Labs. According to an (unpublished) oral history, she began to use the term "software engineering" sometime in 1963 or 1964 to distinguish her work from the hardware engineering taking place on the nascent US space program[7].

Software Engineering vs. Computer Science

Grace Hopper suggested that programming is a practical art[8]; Edsger Dijkstra called the art of programming the art of organizing complexity[9]; Donald Knuth referred to programming as art because it produced objects of beauty[10]. I suspect that all of these observations are true, but what I like best is David Parnas' observation – much like Oettinger's – that there is a distinction between "computer science" and the other stuff that we do[11]. 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 engineer, one must consider static and dynamic forces of the physical 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[12], but the fundamentals of engineering hold true although, as we shall see, each age discovers some new truth about engineering software.

Turn of the 19th Century: Human Computers

Ada Lovelace was perhaps the first person to understand that programming was a thing unto itself[13]. 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*[14]. At the end of that century, we saw the first human computers, such as Annie Cannon[15], Henrietta Leavitt[16], and others, the so called "Harvard Computers[17]" 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[18]. 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-Word II: Birth of the Electronic Computer

Efficiency and the reduction of costs were then as they are now important to every industrial process, and so we saw people such as Frederick Taylor[19] and Frank and Lillian Gilbreth[20] (of *Cheaper By The Dozen*[21] fame) introduce time and motion studies. The Gilbreths also promoted the concept of process charts – the direct predecessor to flow charts – to codify industrial processes[22]. 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 Work Progress Administration was launched as part of Roosevelt's New Deal. Gertrude Blanche was put in charge of the Mathematical Tables Project, the predecessor to today's *Handbook of Mathematical Functions*[23]. 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 1941, W. J. Eckert published *Punched Card Methods in Scientific Computing* which turned out to be, in a manner of speaking, the first computing methodology or pattern language[24].

As the winds of war were gathering in Europe, George Stibitz applied 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[25]. 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 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)[26]. From this the ENIAC came into prominence, but most importantly, later yielded the *First Draft of a Report on the EDVAC*[27].

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 from a machine's hardware[28]. This lead 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 Konrad Zuse[29]; 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[30]. However, it took an engineer – most notably Tommy Flowers – to turn those theories into pragmatic solutions, and from this Colossus was born[31]. Dorthey du Boisson[32], a human computer, served as the primary operator of the Colossus. In her experience of leading a team of women who operated Colossus, she codified the ideas of workflow that eventually was programmed into the machine itself. In the United States, ENIAC[33] then later EDVAC[34] 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 Marylyn Wescoff[35]. 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-Word 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 Goldstein 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[36]. 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[37]. John Backus took Hopper's early work and went further, yielding FORTRAN, the high-level imperative language that would dominate scientific computing for years to come[38].

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)[39]. 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 Backus' work, created COBOL, another imperative language, focused on the needs of businesses[40]. With the introduction of IBM's System/360[41], 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[42] emerged – a predecessor to 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[43] seized on the business opportunity, and established England's first software services business, making 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: Software Engineering's Coming of Age

The rise of the Cold War between the United States and the Soviet Union generated another set of forces that pushed software engineering to come of age. Tom Kilburn and his work with Whirlwind[44] explored the possibilities of real-time programming, and that work led directly to the SAGE[45] 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: human-computer interfaces using CRT displays and light pens, the institutionalization of core memory, 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 most certainly the most important part.

So now, here we are in the second half of the 1960s, with the confluence of three important events in the history of software[46]: 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 Hamilton coined the term "software engineering" and in which the 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[47]. 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, pass them on to programmers who would use their flowcharts to devise algorithms, who in turn would pass on their programs to keypunchers, then 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 ides of time sharing[48], 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*[49]. Dr. Brooks made the important insight that software engineering was not just a technical process, it was a very human process as well.

The economic rise after World War II, given a further boost by the Cold War, led inevitably to a counter-culture shift, as wonderfully described by John Markoff in *What The Dormouse Said*[50]. Quite literally, the introduction of personal computing was not only fueled by technical and social advances, it 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 "Fairytales"[51].

From the Sixties to the Eighties: Maturation

Software engineering was forced to mature. Larry Constantine[52] was perhaps the first to introduce the ideas of modular programming, with the ideas of coupling and cohesion applied as a mechanism for algorithmic decomposition. Edsger Dijkstra[53] took a more formal approach, giving us an important tool for the software engineering, the idea of structured programming.

Around the same time, there was important work by researchers such Robert Floyd[54] and Tony Hoare[55] who devised formal ways to express and reason about programs, a true attempt to connect computer science and software engineering. Niklaus Wirth[56] invented Pascal, an effort to explicitly support best practices in structured programming. Ole Dahl and Kristen Nygaard[57] had the outrageously wonderful idea that yielded the invention of Simula, a language that was object-oriented rather than algorithmic in nature.

Winston Royce[58] 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'

ideas of information hiding[59], Barbara Liskov's ideas of abstract data types[60], and Peter Chen's approaches to entity-relationship modeling[61], all of a sudden the field had a vibrant set of ideas whereby to expresses the artifacts and the processes of software development, leading to the first generation of software engineering methodologies: Doug Ross[62], Larry Constantine[63], Ed Yourdon[64], Tom Demarco[65], Chris Gane, Trish Sarson[66], and Michael Jackson[67] – to name just a few – developed methods for structured analysis and design that took over the field. Adding the work on Michael Fagan[68] (on software inspections), James Martin[69] (on information engineering), John Backus[70] (on functional programming), and Leslie Lamport[71] (on best practices for distributed computing), software engineering entered in its first golden age.

The Eighties and Onward: Golden Age

However, there was a sea change coming. Faced with growing problems of software quality, the rise of ultra-large software intensive systems, the globalization of software, and the shift from programs to distributed systems, new approaches were needed. Dahl and Nygaard's ideas of object-oriented programing gave rise to a completely new class of programming languages: Smalltalk[72], C with Classes[73], Ada[74], and many others. While structured methods were useful, they were not all together sufficient to these new languages, and thus was born the second golden age of software engineering.

Ada[75] – the 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[76] and Ed Yourdon[77] celebrated object-oriented approaches; others brought completely new ideas to the field: Stephen Mellor[78], Peter Coad[79], Rebecca Wirfs-Brock[80], to name a few. The Booch Method[81] grew out of this primordial soup of ideas, as did Jim Rumbaugh's OMT[82] and Ivar Jacobson's Objectory[83]. Sensing an opportunity to bring the market to some common best practices, the three of us united to produced what became the Unified Modeling Language[84]

(made an Object Management Group standard in 1987) and then the Unified Process[85].

Other aspects of software engineering come into play: Philippe Kruchten's 4+1 View Model of software architecture[86], Barry Boehm's work in software economics[87] together with his spiral model[88]; Vic Basili and his ideas on empirical software engineering[89], Capers Jones and software metrics[90], Harlan Mills and clean room software engineering[91], Donald Knuth's literate programming[92], and of course, Watts Humphrey and his Capability Maturity Model[93], to name a few. 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++[94] which later influenced the creation of Java[95]; Alan Cooper's Visual Basic which invigorated the Windows platform[96], Brad Cox's invention of Objective-C had a tremendous effect on NeXT and Apple. Furthermore, Brad's ideas surrounding component-based engineering[97] – another rise in software engineering's levels of abstraction – led directly to Microsoft's OLE and COM[98], which were the predecessors of today's microservice architecture[99].

The Nineties and the Millenium: Era of Disruptions

But another change was in the wind: the internet[100]. Suddenly we had a very rich, as of yet unexplored platform, wherein 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 norm. The Gang of Four[101] – 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[102] 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[103]. Mary Shaw furthered these concepts in her work on software architecture styles[104].

Two other lasting developments of note took place in this era. First, Eric Raymond[105] 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[106], 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[107] which in turn morphed into micro-service 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: Java, JavaScript, Python, C++, C#, PHP, Swift, to name a few). Computing moved from the mainframe to the data center to the cloud, but coupled with microservices, the internet evolved to become the de facto computing platform, with company-specific ecosystems rising like walled cathedrals: Amazon[108], Google[109], Microsoft[110], Facebook[111], Salesforce[112], IBM[113] – really, every economically interesting company built their own fortress. This was now the age of the framework: long gone were the religious battles over operating systems, and now battles were fought along the lines of the veritable explosion of open source frameworks: Bootstrap, jQuery, Apache, NodeJS, MongoDB, Brew, Cocoa, Café, Flutter – truly a dizzyingly and 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[114] – 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[115] coined the term “scrum” in 1986 as an agile approach to product development, and later Ken Schwaber[116] and (independently) Jeff Sutherland and Jeff McKenna[117] codified those principles in the domain of software development. Around that same time, Kent Beck[118] introduced the concept of extreme programming while Ralph Johnson[119] further developed the idea of refactoring (which Martin Fowler[120] further codified in his book by the same name, *Refactoring*). In February, 2001, seventeen agilists met in Snowbird, Utah, and penned the *Agile Manifesto*[121]. The agile approach to software development entered the mainstream.

Software engineering had entered a new golden age. Git[122] and Github[123] emerged; Joel Spolsky gave us Stack Overflow[124]; Jeanettt Wing introduced the idea of computational thinking[125]; Andrew Shafer and Patrick Debois brought us the idea of DevOps[126]; the full stack developer became a thing; the Internet of Things[127] appeared in every imaginable corner of the world. Now, all of a sudden, literally everyone could learn how to code (and many did).

Efforts such as SWEBOK[128] (the Software Engineering Body of Knowledge, first released in 2004 and whose current version was released in 2014) and the Systems Engineering Body of Knowledge[129] by INCOSE exist as an attempt to codify software engineering best practices.

The Decade Ahead: Big Data and the New Season of Artificial Intelligence

But software engineering is about to undergo yet another change.

The foundations of artificial intelligence have been around for literally decades. Over the decades we’ve seen at least four seasons of AI, manifest by extreme rising and falling of fortunes[130]. What we have now feels different: the growth of big data, the abundance of raw computational power, and the presences 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[131] and Google's AlphaGo[132]. In many ways, we are just beginning to understand what is possible and where the limits of these connectionist models of computation live.

We as an industry have not yet built enough of these AI systems to fully understand how they may impact the software engineering process, as they most certainly will. What is the best lifecycle for systems whose components we teach and that learn, rather than program? How do we test them? Where does configuration management fit in when data for ground truth is perhaps more important that the neural network itself? How to be best architect systems with parts whose operation we literally cannot explain nor 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, augmented reality, virtual reality, and the spread of computing to every human, every device, literally every nook and cranny of the earth and beyond, 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[133] 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 high rise: the economics are different, the scale is different, 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 literally thousands more who have made software engineering what it is today, each by their own unique contributions. And so it will be for the future of software engineering. As I said in closing in my ICSE keynote in Florence in 2015, software is the invisible writing that whispers the stories of possibility to our hardware.

And you are the storytellers.

This essay is based on my ACM Learning Webinar of the same title broadcast on April 25, 2018. A recording is available at:

https://www.youtube.com/watch?v=QUz10Z1AfLc

---

[1] Grier, David. *When Computers Were Human*. Princeton, New Jersey: Princeton University Press, 2007.
[2] Ceruszzi, Paul. *Computing: A Concise History*. Cambridge, Massachusetts: MIT Press, 2012.
[3] Leonhardt, David. "John Tukey, 85, Statistician; Coined the World 'Software.'" *New York Times*, 28 July 2000.
[4] Naur, Peter and Randell, Brian. *Software Engineering: Report on a conference sponsored by the NATO Science Committee.* Brussels, Belgium: NATO Scientific Affairs Division, January 1969.
[5] Oettinger, Anthony. "President's Letter to the ACM Membership." *Communications of the ACM*, Vol. 9, No. 12, 1966.
[6] *Computers and Automation*. New York, New York: Edmund Berkeley and Associates, June 1965.
[7] NASA. *Margaret Hamilton, Apollo Software Engineer, Awarded Presidential Medal of Freedom*. 6 August 2017.
[8] Greenberger, Martin. *Management and the Computer of the Future*. Cambridge, Massachusetts: Sloan School of Management, 1962.
[9] Dijkstra, Edsgar. "Notes on Structured Programming." *EWD 249*. Eindhoven, Netherlands: Technological University Eindhoven, April 1970.
[10] Knuth, Donald. "Computer Programming as Art." *Communications of the ACM,* Vol. 17, No. 12, 1974.
[11] Parnas, David. "Software Engineering Programs Are Not Computer Science Programs." *IEEE Software*, November/December 1999.
[12] Swade, Doron. *The Difference Engine: Charles Babbage and the Quest to Build the First Computer*. London England: Penguin Books, 2002.
[13] Hollings, Christopher. *Ada Lovelace: The Making of a Computer Scientist*. Oxford, England: Bodleian Library, 2018.
[14] Boole, George. *An Investigation of the Laws of Thought*. London, England: Walton and Maberly, 1854.
[15] Cannon, Annie. *In the Footsteps of Columbus*. Boston, Massachusetts: Barata and Company, 1983.
[16] Johnson, George. *Miss Leavitt's Stars: The Untold Story of the Woman Who Discovered How To Measure The Universe*. New York City, New York: Norton and Company, 2006.
[17] Gelling, Natasha. "The Woman Who Mapped the Universe and Still Couldn't Get Any Respect." *Smithsonian*, 18 September 2013.
[18] Ceruzzi, Paul. *Computing: A Concise History*. Cambridge, Massachusetts: MIT Press, 2012.
[19] Kanigel, Robert. *The One Best Way: Frederick Winslow Taylor and the Enigma of Efficiency*. Cambridge, Massachusetts: MIT Press, 2005.

[20] Gilbreth, Frank and Gilbreth, Lillian. *Applied Motion Study: A Collection of Papers on the Efficient Method to Industrial Preparedness*. New York City, New York: Sturgis and Walton, 1917.
[21] Gilbreth, Frank and Carey, Ernestine. *Cheaper By The Dozen*. New York City, New York: Thomas Crowell, 1948.
[22] Gilbreth, Frank and Gilbreth, Lilian. *Process Charts*. New York City, New York: American Society of Mechanical Engineers, 1921.
[23] Grier, David. "Gertrude Blanche of the Mathematical Tables Project." *IEEE Annals of the History of Computing*, Vol. 19, No. 4, October/December 1997.
[24] Eckert, Wallace. *Punched Card Methods in Scientific Computing*. New York City, New York: Columbia University Press, 1940.
[25] Stibitz, George and Larrivee, Jules. *Mathematics and Computers*. New York City, New York: McGraw-Hill, 1957.
[26] MacRae, Norman. *John von Neumann: The Scientific Genius Who Pioneered the Modern Computer, Game Theory, Nuclear Deterrence, And Much More*. New York City, New York: American Mathematical Society, 1999.
[27] von Neumann, John. *First Draft of a Report on the EDVAC*. Pittsburgh, Pennsylvania: Moore School of Electrical Engineering. 1945.
[28] Beyer, Kurt. *Grace Hopper and the Invention of the Information Age*. Cambridge, Massachusetts: MIT Press, 2012.
[29] Zuse, Konrad. *The Computer – My Life*. New York City, New York: Springer, 1993.
[30] McKay, Sinclair. *Bletchley Park: The Secret Archives*. London, England: Aurum Press, 2016.
[31] Copeland, Jack. *Colossus: The Secrets of Bletchley Park's Code-breaking Computers*. Oxford, England: Oxford University Press, 2010.
[32] Hicks, Mars. *Programmed Inequality: How Britian Discarded Woman Technologists and Lost Its Edge in Computing*. Cambridge, Massachusetts: MIT Press, 2018.
[33] McCartney, Scott. *ENIAC: The Triumphs and Tragedies of the World's First Computer*. New York City, New York: Walker and Company, 1999.
[34] Wilkes, Maurice. *Automatic Digital Computers*. New York City, New York: John Wiley and Sons, 1956.
[35] *Top Secret Rosies*. Director: LeAnn Erickson. *PBS*, 2011.
[36] Hartree, Douglas. *Calculating Instruments and Machines*. Urbana, Illinois: University of Illinois Press, 1949.
[37] Wheeler, David. "The Use of Sub-routines in Programmes." *Proceedings of the ACM National Meeting*, 1942.
[38] Sammett, Jean. *Programming Languages; History and Fundamentals*. New York City, New York: Prentice Hall, 1969.
[39] Ferry, Georgina. *A Computer Called LEO: Lyons Tea Shops and the World's First Office Computer*. London, England: Fourth Estate, 2003.
[40] Sammett, Jean. *Programming Languages; History and Fundamentals*. New York City, New York: Prentice Hall, 1969.
[41] Pugh, Emmerson. *IBM's 360 and Early 370 Systems*. Cambridge, Massachusetts: MIT Press, 2003.
[42] *SHARE*. https://www.share.org
[43] Lavington, Simon. "An Appreciation of Dina St Johnston, Founder of the UK's First Software House. *The Computer Journal*, Vol. 52 No. 3, 2009.
[44] Redmond, Kent. *Project Whirlwind: History of a Pioneer Computer*. Maynard, Massachusetts: Digital Press, 1980.
[45] Redmond, Kent. *From Whirlwind to MITRE: The R&D Story of the SAGE Air Defense Computer*. Cambridge, Massachusetts: MIT Press, 2000.
[46] Ornstein, Severo. *Computing in the Middle Ages: A View From the Trenches 1955-1983*. Bloomington, Indiana: Author House, 2002.
[47] Metropolis, Nicholas. *A History of Computing in the Twentieth Century*. New York City, New York: Academic Press, 1980.
[48] Pyke, Thomas. "Time-Shared Computer Systems." *Advances in Computers*. London, England: Elsevier, 1967.
[49] Brooks, Fred. *The Mythical Man Month: Essays on Software Engineering*. Boston, Massachusetts: Addison-Wesley, 1975.
[50] Markoff, John. *What the Dormouse Said*. London England: Penguin Books, 2005.
[51] Newell, Alan. "Fairytales." *AI Magazine*, Vol. 13, No. 4, 1922.

[52] Constantine, Larry. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. New York City, New York: Prentice Hall, 1979.
[53] Dijkstra, Edsgar. *A Discipline of Programming*. New York City, New York: Prentice Hall, 1976.
[54] Floyd, Robert. *The Language of Machines*. New York City, New York: W H Freeman, 1994.
[55] Daylight, Edgar. *The Dawn of Software Engineering: From Turing to Dijkstra*. Geel, Belgium: Lonely Scholar, 2012.
[56] Wirth, Niklaus. *Algorithms + Data Structures = Programs*. New York City, New York: Prentice Hall, 1976.
[57] Broy, Manfred and Denert, Ernst. *Software Pioneers*. New York City, New York: Springer, 2002.
[58] Royce, Winfred. "Managing the Development of Large Software Systems. *Proceedings of the IEEE Weston*, 1970.
[59] Hoffman, Daniel and Weiss, David. *Software Fundamentals: Collected Papers of David L. Parnas*. Boston, Massachusetts: Addision-Wesley, 2001.
[60] Liskov, Barbara. *Abstraction and Specification in Program Development*. Cambridge, Massachusetts: MIT Press, 1986.
[61] Chen, Peter. *The Entity-Relationship Model: A Basis for the Enterprise View of Data*. New York City, New York: Sagwan Press, 2018.
[62] Ross, Douglas. "Structured Analysis: A Language for Communicating Ideas." *IEEE Transactions on Software Engineering*, Vol. 3, No. 1, 1977.
[63] Constantine, Larry. *Constantine on Peopleware*. New York City, New York: Prentice Hall, 1995.
[64] Yourdon, Edward. *Modern Structured Analysis*. New York City, New York: Prentice Hall, 1988.
[65] DeMarco, Tom. *Structured Analysis and System Specification*. New York City, New York: 1979.
[66] Gane, Chris and Sarson, Trish. *Structured Systems Analysis*. New York City, New York: 1979.
[67] Jackson, Michael. *Software Requirements and Specifications: A Lexicon of Practice, Principles, and Prejudices*. New York City, New York: Addison-Wesley, 1995.
[68] Fagan, Michael. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems Journal* Vol. 15, No. 3, 1976.
[69] Martin, James. *Information Engineering*. New York City, New York: Prentice Hall, 1989.
[70] Backus, John. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." *Communications of the ACM*, Vol. 21, No. 8, 1978.
[71] Lamport, Leslie. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM*, Vol. 21, No. 7, 1978.
[72] Goldberg, Adele. *Smalltalk 80: The Language*. New York City, New York: Addison-Wesley, 1989.
[73] Stroustrup, Bjarne. "Evolving a Language In and For the Real World." *ACM History of Programming Language*, 2007.
[74] Sammet, Jean. "Why Ada is Not Just Another Programming Language." *Communications of the ACM*, Vol. 29, No. 8, 1986.
[75] Ichbiah, Jean. *Rationale for the Design of the Ada Programming Language*. Cambridge, England: Cambridge University Press, 1991.
[76] Martin, James and Odell, James. *Principles of Object-Oriented Analysis and Design*. New York City, New York: Prentice Hall, 1993.
[77] Yourdon, Edward. *Object-Oriented Systems Design*. New York City, New York: Prentice Hall, 1993.
[78] Shlaer, Sally and Mellor, Stephen. *Object-Oriented Systems Analysis: Modeling the World In Data*. New York City, New York: Prentice Hall, 1988.
[79] Coad, Peter. *Object-Oriented Design*. New York City, New York: Prentice Hall, 1991.
[80] Wirfs-Brock, Rebecca. *Designing Object-Oriented Software*. New York City, New York: Prentice Hall, 1990.
[81] Booch, Grady. *Object-Oriented Design with Applications*. New York City, New York: Benjamin-Cummings, 1990.
[82] Rumbaugh, James. *Object Oriented Modeling and Design*. New York City, New York: Prentice Hall, 1991.
[83] Jacobson, Ivar. *Object-Oriented Software Engineering: A Use Case Approach*. New York City, New York: 1992.
[84] Booch, Grady, Rumbaugh, James, and Jacobson, Ivar. *The Unified Modeling Language User Guide*. New York City, New York: Addison-Wesley, 2005.
[85] Kruchten, Philippe. *The Rational Unified Process*. New York City, New York: 2003.
[86] Kruchten, Philippe. "Architectural Blueprints: The 4+1 View Model of Software Architecture." *IEEE Software*, Vol. 12, No. 6, 1995.

[87] Boehm, Barry. *Software Engineering Economics*. New York City, New York: Prentice Hall, 1981.
[88] Boehm, Barry. "A Spiral Model of Software Development and Enhancements." *ACM SIGSOFT*, Vol. 11, No. 4, 1986.
[89] Boehm, Barry and Rombach, Hans. *Foundations of Empirical Software Engineering: The Legacy of Victor Basili*. New York City, New York: Springer, 2005.
[90] Jones, Capers. *The Economics of Software Quality*. New York City, New York: Addison-Wesley, 2011.
[91] Mills, Harlan. *Software Productivity*. New York City, New York: Dorset House, 1988.
[92] Knuth, Donald. *Literate Programming*. Palo Alto, California: Center for the Study of Language and Information, 1992.
[93] Humphrey, Watts. *A Discipline for Software Engineering*. New York City, New York: Addison-Wesley, 1995.
[94] Stroustrup, Bjarne. *C++: The Programming Language*. New York City, New York: Addison-Wesley, 1991.
[95] Gosling, James. *The Java Programming Language*. New York City, New York: Addision-Wesley, 1996.
[96] Cooper, Alan. *The Inmates are Running the Asylum*. New York City, New York: Sams-Pearson Education, 2004.
[97] Cox, Brad. *Object-Oriented Programming: An Evolutionary Approach*. New York City, New York, Addison-Wesley, 1991.
[98] *Automation Programmer's Reference*. Bellevue, Washington: Microsoft Press, 1997.
[99] Newman, Sam. *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, California: O'Reilly Media, 2015.
[100] Hafner, Katie. *Where Wizards Stay Up Late*. New York City, New York: Simon and Schuster, 1998.
[101] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. New York City, New York: Addison-Wesley, 1994.
[102] *The Hillside Group.* https://hillside.net
[103] Coplien, James. *Organizational Patterns of Agile Software Development*. New York City, New York: Prentice Hall, 2004.
[104] Shaw, Mary and Garlan, David. *Software Architecture: Perspectives on an Emerging Discipline*. New York City, New York: Pearson, 1996.
[105] Raymond, Eric. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, California: O'Reilly Media, 2001.
[106] Karnik, Kiran. *The Coalition of Competitors*. Delhi, India: Harper Collins India, 2012.
[107] Daigneau, Robert. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. New York City, New York: Addison-Wesley, 2011.
[108] Wittig, Michael. *Amazon Web Services in Action*. New York City, New York: Manning Publications, 2015.
[109] Geewax, JJ. *Google Cloud Platform in Action*. New York City, New York: Manning Publications, 2018.
[110] Gurhrie, Scott. *Building Cloud Apps with Microsoft Azure*. Bellevue, Washington: Microsoft Press, 2014.
[111] Hyatt, Michael. *Platform: Get Noticed in a Noisy World*. Nashville, Tennessee: Thomas Nelson, 2012.
[112] Ouelette, Jason. *Development with the Force.com Platform*. New York City, New York: Addison-Wesley, 2013.
[113] Iyengar, Ashok. *IBM Cloud Platform Primer*. Boise, Idaho: MC Press, 2015.
[114] Ashmore, Sondra and Runyan, Kristin. *Introduction to Agile Methods*. New York City, New York: Addison-Wesley, 2014.
[115] Takeuchi, Hirotaka and Nonaka, Ikujiro. "The New New Product Development Game." *Harvard Business Review*, January 1986.
[116] Schwaber, Ken. *Agile Project Management with Scrum*. Bellevue, Washington, Microsoft Press, 2004.
[117] Sutherland, Jeff and Sutherland, JJ. *Scrum: The Art of Doing Twice the Work in Half the Time*. Danavers, Massachusetts: Currency, 2014.
[118] Beck, Kent. *Extreme Programming Explained*. New York City, New York: Addison-Wesley, 2004.
[119] Johnson, Ralph. "Living and Working with Aging Software." *University Of Illinois at Urbana-Champaign*, 1994.
[120] Fowler, Martin, Beck, Kent, Brant, John, Opdyke, William, Roberts, Don, and Gamma, Erich. *Refactoring: Improving the Design of Existing Code*. New York City, New York: Addison-Wesley, 1999.
[121] *The Agile Alliance*. https://agilealliance.org
[122] Chacon, Scott and Straub, Ben. *Pro Git*. New York City, New York: Apress, 2014.
[123] *GitHub*. https://github.com

[124] *StackOverflow*. https://stackoverflow.com
[125] Wing, Jeannette. "Computational Thinking." *Communications of the ACM*, Vol. 49, No. 3, 2006.
[126] Kim, Gene, Debois, Patrick, Willis, John, Humble, Jez, and Allspaw, John. *DevOps Handbook*. Portland, Oregon: IT Revolution Press, 2016.
[127] Grengard, Samuel. *The Internet of Things*. Cambridge, Massachusetts: MIT Press, 2015.
[128] *Guide to the Software Engineering Body of Knowledge*, Washington, DC: IEEE Computer Society Press, 2014.
[129] *INCOSE Systems Engineering Handbook*. New York City, New York: Wiley, 2015.
[130] Knight, Will. "AI Winter Isn't Coming." *MIT Technology Review*, 7 December 2016.
[131] Kelly, John and Hamm, Steve. *Smart Machines: IBM's Watson and the Era of Cognitive Computing*. New York City, New York: Columbia University Press, 2013.
[132] *AlphaGo*. https://deepmind.com/blog/alphago-zero-learning-scratch
[133] Harari, Yuval. *Sapiens: A Brief History of Humankind*. New York City, New York: Harper Collins, 2015.