When Good Software Goes Bad

The Surprising Durability of an Ephemeral Technology

Nathan Ensmenger
Indiana University

September 11, 2014

Software is Hard

In the very last published article of his long and distinguished career, the historian of computing Michael Mahoney asked a simple but profound question: “What makes the history of software hard?”[1] In his characteristically playful style, Mike was engaging both with an issue of central importance to historians — namely, how can we begin to come to grips with the formidable challenges of writing the history of software — but also one of great interest to practitioners. From the earliest decades of electronic computing to the present, the problem of software has loomed large in the software literature, both practical and historical. In stark comparison to computer hardware, which was with each successive generation becoming faster, smaller, and less expensive, software systems seemed to only grow ever more complex, costly, and error-prone. By the end of the 1960s, many in the computer industry were talking openly about a “software crisis” that threatened to undo all of the progress of the previous decades. The rhetoric of crisis would continue to define the industry for the next half-century, with specific claims about the nature and causes of the crisis being constantly adapted to new circumstances, actors, and agendas.

I have written extensively elsewhere about the history of the software crisis and its relationship to the larger history of computing, labor history, and the history of science and technology.[2] At the heart of my analysis was the notion of heterogeneous engineering as it has been developed by John Law and Bruno Latour. Software, I argued, is perhaps the ultimate heterogeneous technology. It straddles the boundaries between science and technology, art and engineering, the intellectual and the material. Unlike the computer itself, which is a physical artifact, a tangible “thing” that can readily be isolated, identified, and evaluated, software largely invisible, ethereal, and ephemeral. While certain forms of software, such as a sorting algorithm, can be generalized and formalized as mathematical abstractions, most remains local and specific, inextricably inter-

[1]Michael S. Mahoney. “What Makes the History of Software Hard”. In: *Annals of the History of Computing, IEEE* 30.3 (July 2008), pp. 8–18.

[2]Nathan Ensmenger. *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise.* MIT Press, 2010.

twined with specific technological configurations, organizational constraints and culture, informal industry standards, and government regulations. A better example of the messy complexity of the heterogeneous sociotechnical system can scarcely be imagined. All of which is simply an embellishment of the essential point the Mike Mahoney made so much more succinctly: The history of software is hard, Mike argued, because software itself is hard: hard to design, hard to develop, hard to use, hard to understand, and hard to maintain.

In this paper, I will explore the last of these issues, which is perhaps the most perplexing and paradoxical — but arguably the most significant — of the reasons why software is hard: namely, the challenges associated with software maintenance.

The problem of maintenance is a ubiquitous but neglected element of the history of technology. All complex technological systems eventually break down and require repair (some more so than others), and, in fact, as David Edgerton has suggested, maintenance is probably the central activity of most technological societies.[3] But maintenance is also low-status, difficult, tedious and risky.[4] Engineers and inventors do not like maintenance (and therefore generally do not do maintenance), and therefore historians of technology have largely ignored it.

Nowhere is this distaste for maintenance more apparent that in software development. Not only are software developers particularly adverse to working on other people's systems (the infamous "not-invented-here" syndrome), but software itself is, in theory, a technology that should never need fixing. Software does not break down or wear out, at least in the conventional sense. Once a software-based system is working, it will work forever (or at least until the underlying hardware breaks down – but that is someone else's problem). Any latent "bugs" that are subsequently revealed in the system, are considered flaws in the original design or implementation, not the result of the wear-and-tear of daily use, and in theory could be completely eliminated by rigorous development and testing methods. Occasionally a stray cosmic ray might flip an unexpected bit in a software system, causing an error, but generally speaking, software is a technology that can never be broken.

Except that software does get broken. All the time. And at great expense and inconvenience to its owners and users. In most large software projects maintenance represents the single most time consuming and expensive phase of development. From the early 1960s to the present, software maintenance costs have represented between 50% and 70% of all total expenditures on software development.[5] In 1995 firms in the United States alone spent $70 billion on maintenance of more than ten billion lines of legacy software code.[6] The total maintenance costs associated with the

[3]David Edgerton. *The shock of the old: technology and global history since 1900.* Oxford: Oxford University Press, 2007.

[4]Tine Kleif and Wendy Faulkner. ""I'm No Athlete [but] I Can Make This Thing Dance!" Men's Pleasures in Technology". In: *Science, Technology & Human Values* (2003).

[5]B. P. Lientz, E. B. Swanson, and G. E. Tompkins. "Characteristics of application software maintenance". In: *Commun. ACM* 21.6 (1978), pp. 466–471; Girish Parikh. "Software maintenance: penny wise, program foolish". In: *SIGSOFT Softw. Eng. Notes* 10.5 (1985), pp. 89–98; Gerardo Canfora and Aniello Cimitile. *Software Maintenance.* Tech. rep. University of Sannio, 2000; Ruchi Shukla and Arun Kumar Misra. "Estimating software maintenance effort: a neural network approach". In: *ISEC '08: Proceedings of the 1st conference on India software engineering conference.* Hyderabad, India: ACM, 2008, pp. 107–112.

[6]J Sutherland. "Business objects in corporate information systems". In: *ACM Computing Surveys (CSUR)* (1995).

Y2k Crisis has been estimated at more than $300 billion.[7] Most computer programmers begin their careers doing software maintenance, and many never do anything but. And despite decades of effort, innovation, and investment in software development methods and technologies, the problem never seems to get any better.

The crisis of software maintenance, like all of the various iterations of the seemingly perpetual software crisis, has been interpreted and reinterpreted in the service of a variety of social, political, and technological agendas. At times it has been defined in terms of a failure of contemporary programming languages or software development methodologies; at others a lack of professionalism on the part of programmers; increasingly frequently, as a problem of programmer management. One of my favorites is a 1981 study by the National Science Foundation, which argued that the "software maintenance gap" represented a crisis of national security:

> If software practices continue to drift, in 20 years the U.S. will have a national inventory of unstructured, hard-to-maintain, impossible-to-replace programs written in Fortran and Cobol as the basis of its industrial and government activities. Conversely, the Soviets may very well have a set of well-structured, easily maintained and modifiable programs.[8]

What is not in dispute is the existence of a crisis. In fact, even before there was a word for what we today call software, there was perceived problem with software maintenance. Maurice Wilkes, one of the first people ever to program a modern, stored-program computer, famously recalled the moment, in June 1949, when he suddenly realized that "a good part of the remainder of my life was going to be spent in finding errors in my own programs."[9] Technically, what Wilkes was describing was not so much the process of maintaining computer programs but of debugging them (meaning the elimination of flaws in the original design or implementation, rather than the repair of accumulated errors), but the larger implication for the computing community is obvious: the delivery of a working application was only the beginning of the life-cycle of a software application. A programmer could – and many did – spend the majority of their career chasing down the bugs that gradually revealed themselves in the operation of a complex software-based system. In this respect, runs the well-worn joke, programming a computer was a little bit like sex: "One mistake and you have to support it for the rest of your life."

But even if we exclude the ongoing process of debugging software (which most of the estimates of software maintenance costs indeed do), maintenance still accounts for more than half of the overall cost of software development. This is true even of software that is considered effectively bug-free. What, then, does maintenance mean in this context? How does one repair an unbreakable technology?

In order to properly understand software maintenance, we must first come to grips with what we mean by software.

[7]Robert Mitchell. "Y2K: The good, the bad and the crazy". In: *Computerworld* (2009).
[8]Parikh, "Software maintenance: penny wise, program foolish".
[9]Maurice V. Wilkes. *Memoirs of a Computer Pioneer (History of Computing)*. 1985.

Software is/as System

Most of us today tend to think of software as a consumer good, a product, a pre-packaged application. You purchase a copy of Microsoft Word, or Call of Duty 4, you install it, and you make do with the functionality provided, whether or not it does exactly what you want or works entirely well as you might have hoped. You might lose the installation CD, or the manual, but you don't take your software back to the shop for regular repair.

Historically speaking, however, software was not something that was purchased off-the-shelf, nor was it a single application or product. Rather, it was a bundle of systems, services, and support.[10] To this day the vast majority of software is custom-produced for individual corporations in a process that resembles more the hiring of a management consulting firm than the purchase of a mass-market consumer good.[11][12]

In fact, it was not until more than a decade after the development of the first electronic computers that the statistician John Tukey first applied the word "software" to those elements of a typical computer installation that were not obviously "tubes, transistors, wires, tapes and the like."[13] But although the term itself might have been novel, the constitutive elements of Tukey's software – libraries, compilers, and systems utilities – were not. The first commercial electronic computers had only been available for a few years when the availability of useful and reliable "software" was identified as one of the critical bottlenecks hindering the expansion of the entire computing industry.[14]

It is important to note, however, that Tukey's software was not an end-user application, such as an accounting package or an engineering simulation program, but rather the collection of low-level tools used to construct and manage such applications. Today we would consider such tools to be part of an operating system or development platform. And although software code was generally provided for free by computer manufacturers – it was not until the very late 1960s that software was "unbundled" from the purchase of a computer – by itself it represented only a small component of a larger system of programmer services and support. Outside of this larger context of services, provided largely by expert consultants and specialist programmers, software as it was understood in the late 1950s was effectively useless.

It did not take long for industry observers and computing service providers to recognize the significance of this larger context. Just a few years after John Tukey introduced his preliminary definition of software, Bernard Galler, then head of the University of Michigan Computing Center (and later president of the ACM) broadened the term with an insightful emendation: for the

[10]T. Haigh. "Software in the 1960s as concept, service, and product". In: *Annals of the History of Computing, IEEE* 24.1 (2002), pp. 5–13.

[11]Martin Campbell-Kelly. *From airline reservations to Sonic the Hedgehog : a history of the software industry.* MIT Press Cambridge, Mass., 2003.

[12]For example, clients of the German software developer SAP, the 3rd largest software manufacturer in the world, are required to spend $2 million to $5 million per year on maintenance and support contracts that average 20% to 25% of the original software cost. Source: Computerworld `http://goo.gl/03ank2`

[13]John Tukey. "The Teaching of Concrete Mathematics". In: *American Mathematical Monthly* 65.1 (1958), pp. 1–9.

[14]Peter B. Laubach and Lawrence E. Thompson. "Electronic Computers: A Progress Report". In: *Harvard Business Review* Issue 233 (1955), p. 120.

user of a computer, “the total computing facility provided for his use, other than the hardware, is the software.”[15] The implication was that most users could not or did not distinguish between the elements of the software system: tools, applications, personnel, and procedures were all considered essential elements of the software experience. By the end of the decade the term had been expanded even further to include documentation, development methodologies, user training, and consulting services.

Software was an ever-expanding category that grew not only in size and scale but also in scope. As the nuts-and-bolts of computer hardware became faster, more reliable, and less-expensive – and therefore increasingly invisible to the end-user – the relative importance of software became even more pronounced. In effect, for most organizations, by the end of the 1960s, software *was* computer.

Errors, Enhancements, and Evolution

If we consider software not as an end-product, or a finished good, but as a heterogeneous system, with both technological and social components, we can understand why the problem of software maintenance was (is) so complex. It raises a fundamentally question – one that has plagued software developers since the earliest days of electronic computing – namely, what does it mean for software to work properly?

The most obvious answer to this question is that software “works” when it performs as expected, when the behavior of the system conforms to its original design or specification. In other words, the software works when it is free from flaws in its implementation, known colloquially as “bugs.” If these bugs are discovered prior to the official “release” of the software (often an ambiguously defined moment or milestone), then the work of fixing them is considered development; if they are discovered afterwards, this same work is defined as maintenance. This is, of course, a somewhat arbitrary distinction, but one with critical implications for the budget, schedule, and perception of a project.[16] Who does the work, who pays for it, who gets the credit (or blame), whether or not the development phase is considered a success or not — these are all affected by whether the work of debugging is categorized as development or maintenance.

Even more difficult to determine is what exactly constitutes a bug, and who gets to decide. Some bugs are obvious — the system crashes, or returns an obviously incorrect result (1 + 2 = 4). But most bugs are much more difficult to identify. Some unexpected behaviors are the result of the limitations of existing hardware, or of the limits of computation, or the result of deliberate design compromises. To provide an extremely simplified version of a very real issue that involves floating point computation, it is entirely possible for a computer to return the result 10/3 * 3 = 9.9999 and still be considered to be working (in the sense of being bug-free).[17] But more common were bugs that involved rare or edge cases that called into question basic questions about what

[15]Bernard Galler. “Definition of Software”. In: *Communications of the ACM* 5.1 (1961), p. 6.

[16]Girish Parikh. “Exploring the world of software maintenance: what is software maintenance?” In: *SIGSOFT Softw. Eng. Notes* 11.2 (1986), pp. 49–52.

[17]Donald MacKenzie. “Negotiating Arithmetic, Constructing Proof: The Sociology of Mathematics and Information Technology”. In: *SSS Social Studies of Science* (1993).

was the "correct" behavior of the system. The burden of finding such bugs almost always devolved upon the end users. They were almost necessarily only revealed *after* the software was integrated into its larger operational environment.

An additional complication is that although it is possible to determine during the planning and development phase whether a given software design contains flaws, it is impossible to demonstrate conclusively that it *does not*. Donald MacKenzie has written extensively about the software verification movement, which attempted to establish, either mathematically or using empirical testing regime, to "prove" that software was reliable.[18] The short version of his research is that this movement was a failure. All software has bugs; the question is simply whether of not they are known, and the degree to which they affect the general consensus on whether or not the software is "working."

In any case, although fixing bugs might seem the most obvious and significant form of software maintenance, in reality only a small percentage of maintenance efforts are devoted to fixing errors in implementation.[19] One exhaustive study from the early 1980s estimates such emergency repairs to occupy at most one-fifth of all software maintenance workers.

But if the real work of maintenance is not finding and correcting bugs, then what is it all about?

The majority of software maintenance involve what are vaguely referred to in the literature as "enhancements." These enhancements sometimes involved strictly technical measures – such as implementing performance optimizations – but most often what Richard Canning, one of the computer industry's most influential industry analysts, termed "responses to changes in the business environment."[20] This included the introduction of new functionality, as dictated by market, organizational, or legislative developments. Software maintenance defined in terms of enhancement therefore incorporated such apparently non-technical tasks as "understanding and documenting existing systems; extending existing functions; adding new functions; finding and correcting bugs; answering questions for users and operations staff; training new systems staff; rewriting, restructuring, converting and purging software; managing the software of an operational system; and many other activities that go into running a successful software system."[21]

While the notion that software is but one element in a fluid and permeable sociotechnical system might be familiar to us as historians of science and technology, it was profoundly uncomfortable to software developers. The idea that software technology had no fixed limits, no final end-state that could be unambiguously defined as success — in other words, that the work of software development was never done — was a real problem for project managers responsible for meeting deadlines, maintaining budgets, and satisfying requirements. In one of the earliest published papers on the problem of software maintenance, the computer scientist Meir Lehman described the situation thus:

[18]Donald MacKenzie. *Mechanizing Proof.* MIT Press, 2004.

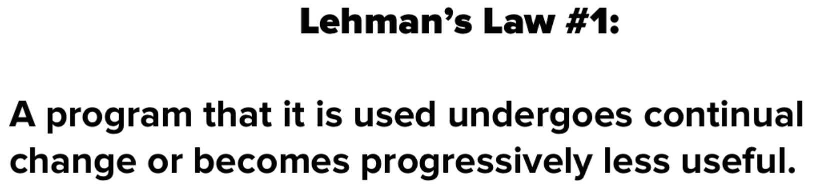
[19]David C. Rine. "A short overview of a history of software maintenance: as it pertains to reuse". In: *SIGSOFT Softw. Eng. Notes* 16.4 (1991), pp. 60–63.

[20]Richard Canning. "The Maintenance 'Iceberg'". In: *EDP Analyzer* (Oct. 1972).

[21]E. Burton Swanson. "The dimensions of maintenance". In: *ICSE '76: Proceedings of the 2nd international conference on Software engineering.* San Francisco, California, United States: IEEE Computer Society Press, 1976, pp. 492–497.

> Large scale, widely used programs such as operating systems are never complete. They undergo a continuing evolutionary cycle of maintenance, augmentation and restructuring to keep pace with evolving usage and implementation technologies.[22]

By describing the "unending development of programs" in terms of the "the evolutionary processes[es] that governs the life cycle of any complex system," Lehman and his co-author Francis Parr were attempting to eliminate what they saw as the artificial distinction between development and maintenance. It was absurd to think of software as a technology that could simply be designed, constructed, and then be considered "finished." Rather, the software system "interacts and interplays with its environment in [a process of] mutual reinforcement" that only approaches, but never reaches, some intended functionality. As users lean to exploit the capabilities of the system, they "discover or invent new ways of using it," which encourages develops to modify of extend the system, which stimulates another round of user-driven innovation or process change, which in turn generates the demand for new features. In his later work, Lehman would define three types of software programs: S-programs, P-programs, and E-programs. S and P programs were limited in scope and precisely specified; E-programs were "real world" applications that were strongly linked to their operational environment.[23] In a series of eight "Lehman's Laws," he defined the life-cycle of E-programs as a continuous process of change, growth (in both size and complexity), and decay. Without an ongoing and rigorous strategy for software maintenance, E-programs were destined to be disasters.


Lehman's Law #1:

A program that it is used undergoes continual change or becomes progressively less useful.

Figure 1: The First of Lehman's Eight Laws of Software Maintenance

Perpetual Disappointment

It is no coincidence that the discovery that software development was an unending and evolutionary process — in other words, the software is a technology without telos — coincided with the emergence of a growing sense of dissatisfaction among the corporate managers who planned, supervised, and paid for software development projects. Beginning in the early 1960s, the Harvard Business Review had published a series of articles by the Harvard Business School professor

[22]M M Lehman and F N Parr. "Program evolution and its impact on software engineering". In: *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. 1976.

[23]M M Lehman. "Programs, life cycles, and laws of software evolution". In: *Proceedings of the IEEE*. 1980.

John Dearden on the "myths" and "mirages" of corporate computerization efforts.[24] "Many companies today are faced with serious problems in utilizing the capabilities of computers," Dearden argued, "and the situation appears to be getting worse rather than better."[25]In a widely-cited 1968 report the venerable consulting firm McKinsey and Company argued that, despite years of investment in "sophisticated hardware," "larger and increasingly costly computer staffs," and "complex and ingenious applications," the "computer efforts, in all but a few exceptional companies, are in real, if often unacknowledged, trouble."[26] A competing report from consultants at Touche Ross drew similar conclusions about the "fizzle" of the "Computer Revolution." Billions of dollars had already been spent on electronic computers by this period, the report's authors noted, but it was not necessary to "to scratch very far below the surface, in company after company … to find one story after another of runaway costs of EDP (Electronic Data Processing), of interdepartmental antagonisms it created, of wasted efforts, of misguided applications, of systems installed months and years behind schedule at double the originally anticipated costs, with marginal benefits after all the effort and expense."[27] A 1969 article in *Fortune* summarized the mood of the business press when it declared that one of the harsh lessons of recent business history had been that "computer can't solve everything." After "buying or leasing some 60,000 computers during the past fifteen years, businessmen are less and less able to state with assurance that it's all worth it."[28]

This widespread crisis of confidence in the "profit potential" of computerization only be understood in light of the rising costs of software development. The problem was not that the computers that these companies purchased did not "work," but rather there was no shared agreement on what "work" the organization wanted the computer to accomplish. Once the easy work of mechanizing routine clerical activities was complete, applying computer power to more complex administrative process proved extraordinarily difficult. Consider, for example, the software that had to be developed to "computerize" an accounting operation: this included not only the computer code that had to be written, but also an analysis of existing operations, the reorganization of procedures and personnel, the training of users, the construction of peripheral support tools and technologies, and the production of new manuals and other support materials.[29](%20) In an ideal world, the software development process could be readily partitioned into a series of discrete and sequential stages: analysis, design, development, testing, and release. In reality, these tasks were never so neatly delineated. Analysis, development, and testing were in fact iterative processes that resembled feedback loops or repeated cycles rather than a neatly ordered progression of task lines on a project manager's Gantt chart.

Most of the literature on the so-called "software crisis" of the late 1960s focuses on the rising cost of software in this period, and the resulting attempts by corporate managers, computer

[24]John Dearden. "How to Organize Information Systems". In: *Harvard Business Review* 43.2 (1965), pp. 65–73; John Dearden. "Myth of Real-Time Management Information". In: *Harvard Business Review* 44.3 (1966), pp. 123–132; John Dearden. "MIS is a mirage". In: *Harvard Business Review* 50.I1 (1972), pp. 90–99.

[25]Dearden, "How to Organize Information Systems".

[26]"Unlocking the Computer's Profit Potential." In: *McKinsey Quarterly* 5.2 (1968), p17–31.

[27]Arnold Ditri and Donald Wood. *The End of the Beginning – The Fizzle of the 'Computer Revolution'*. Touche Ross and Company. 1969.

[28]T. Alexander. "Computers Can't Solve Everything". In: *Fortune* 80.5 (1969), pp. 126–9, 168, 171.

[29]Andrew Friedman and Dominic Cornford. *Computer Systems Development: History, Organization and Implementation*. New York John Wiley & Sons, 1989.

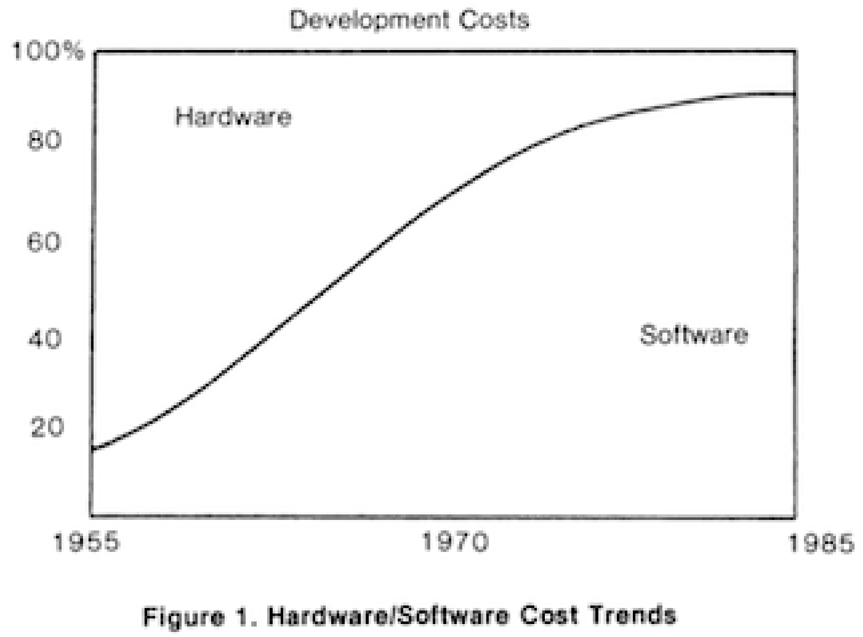


Figure 2: The infamous hardware/software inversion of the late 1960s

scientists, and aspiring "software engineers," to organize and rationalize the work of software development along the lines of industrial engineering. Recent research in the history of software has called into question some the fundamental assumptions that frame the conventional wisdom about the crisis, and have focused instead on its social, professional, and rhetorical construction. But even the most critical work in this area has taken for granted an idealized model of software production in which development is neatly separable from maintenance. In doing so, it has reified the assumption, still very common among practitioners, that software maintenance is an anomaly. Maintenance is only necessarily then the design and development process somehow goes awry. To the degree that maintenance is ever discussed in the historical literature, it is as a subsidiary adjunct to real work of software development.

Maintenance is a Misnomer

As we have already indicated, there is much evidence to suggest that software maintenance ought to be taken more seriously by historians of the software crisis. This is in part because of close reading of the industry literature makes clear that industry practitioners took the problem of maintenance seriously. Richard Canning, one of the most influential industry analysts of the early computer industry, was writing as early as 1965 about software maintenance. In widely-cited 1972 he declared that maintenance was already devouring as half or two-third of programming resources, and warned that this was just the "tip of the maintenance iceberg".[30] Barry Boehm, whose work on the "inversion" of the hardware-software cost relationship helped focus attention the software development, also noted the growing percentage of software spending associated with maintenance[31] By the late 1970s it was common knowledge that maintenance represented at least 40%, and as much as 75%, or all software expenditures.[32]

Even more interesting is the high degree of sophistication of sociological analysis that can be found in the maintenance literature. I have already described Meir Lehman's early and influential description of software evolution, which not only defines software in terms strongly reminiscent of a Hughesian sociotechnical system, but also describes a process of technological co-construction that would not be out of place in the SCOT "school bus" book.[33] In comparison with most of the authors who write about development in this period, who are often more prescriptive than descriptive (a reflection, in part, of their explicitly managerial agenda), the relatively small number of practitioners and academics who focus on maintenance generally paint a more detailed and nuanced portrait of the software lifecycle.

By the early 1980s, the industry and technical literature had settled on a shared taxonomy for talking about software that identified three different dimensions of software maintenance: corrective maintenance (largely focused on bug fixes), perfective maintenance (which included performance improvements), and adaptive maintenance (adaptions to the larger environment). Adaptive maintenance so dominated real-world maintenance that many observers pushed for an entirely new nomenclature: software maintenance was a misnomer, they argued: the process of adapting software to change would better be described as "software support", "software evolution", or (my personal favorite) "continuation engineering."[34] Unfortunately, software maintenance was the term that stuck.

In all of these new models for thinking about maintenance, the emphasis was on systematic and sustainable solutions. If the work of maintenance began only after the software system was released, it was already too late. "The main problem in doing maintenance," argued Norman Schneidewind in a 1987 survey of the state of software maintenance, "is that we cannot do maintenance on a system which was not designed for maintenance."[35] Maintainable software was software that could accommodate the inevitable evolution of the sociotechnical system. It should be expected, not lamented. The best programmers should be assigned to maintenance, and they should be granted access and influence in all of the stages of design and development.

At the heart of the proactive approach to maintenance was a recognition of the root causes of what makes software so hard: mapping a complex human cognitive or work process into machine-oriented algorithms involved communication, negotiation, and compromise. Devel-

[30]Canning, "The Maintenance 'Iceberg'".

[31]Barry W Boehm. "The high cost of software". In: *Practical Strategies for Developing Large Software Systems* (1975), pp. 3–15.

[32]Lientz, Swanson, and Tompkins, "Characteristics of application software maintenance".

[33]Wiebe Bijker, Thomas Hughes, and T. J Pinch, eds. *The Social Construction of Technological Systems*. The MIT Press Cambridge MA, 1987.

[34]Girish Parikh. "What is software maintenance really?: what is in a name?" In: *SIGSOFT Softw. Eng. Notes* 9.2 (1984), pp. 114–116.

[35]N F Schneidewind. "The State of Software Maintenance". In: *Software Engineering, IEEE Transactions on* (1987).

oping large-scale software products involved ongoing (and often contentious) dialog between a variety of interested parties, including systems analysts, software architects, computer programmers, machine operators, corporate managers, and end users. This dialog could not be limited to a single moment in the development lifecycle, or isolated in the hands a few high-level architects. Despite the reputation of computer programmers for being anti-social, for "preferring machines over people," savvy software developers quickly realized that "communication with the computer [writing code] is only half of the problem … communication with other humans is just as important."[36] This was particularly true in the context of maintenance, where the biggest challenge facing programmers was reading — and comprehending — another programmer's code.[37]

In theory, a well-written computer program is self-documenting; that is to say, the computer code itself contains its own complete written specification. In practice, most computer programs were too arcane and idiosyncratic for even their original authors to fully understand. Despite efforts to cultivate good code commentary practices and other standardized documentary practices, reading and comprehending computer code remained notoriously difficult. In order to facilitate effective communication in large programming teams — communications between programmers and managers, between one programmer and another, and even between an individual programmer and his or her future self — programmers had to develop subsidiary technologies to document their designs, decisions, and intentions.

One such technology is the flowchart. Flowcharts did not originate with software development, but they are now intimately associated with it. Originally intended to serve as design specifications ("flowcharts are to programmers as blueprints are to engineers"), flowcharts were rarely sufficient for the task. Many programmers loathed and resented having to draw (and redraw) flowcharts, and the majority did not. Frederick Brooks dismissed flowcharts as an "obsolete nuisance" and denied that he had ever known "an experienced programmer who routinely made detailed flow charts before beginning to write programs."[38] Most programmers preferred to "code first, flowchart later" and produced their flowcharts only after they were done writing code. One of the first commercial software packages, Applied Data Research's Autoflow, was designed specifically to reverse-engineer a flowchart "specification" from already-written program code. In other words, the implementation of many software systems actually preceded their own design! Those flowcharts that were produced prior to development were either too simplistic to be useful, or so complex that they "more closely resemble[d] confusing road maps than the easily understood pictorial representations."[39] And flowcharts required their own form of maintenance: in any active software project, any changes to the program rendered is corresponding flowchart immediately obsolete. "Any resemblance between our flow charts and the present program is purely coincidental," Donald Knuth famously declared[40] Given that design

[36] *Are programmers paranoid?* Vol. Proceedings of the Tenth annual conference on SIGCPR. ACM Press, 1972, pp. 47–54; J.M. Yohe. "An Overview of Programming Practices". In: *ACM Computing Surveys* 6.4 (1974), pp. 221–245.

[37] Robert L Glass. "Documenting for software maintenance". In: *SIGDOC Asterisk Journal of Computer Documentation* (1979).

[38] Frederick Brooks. *The Mythical Man-Month*. Addison, 1982.

[39] Wayne LeBlanc. "Standardized flowcharts". In: *ACM SIGDOC Asterisk Journal of Computer Documentation* (1978).

[40] Donald E Knuth. "Computer-drawn flowcharts". In: *Communications of the ACM* (1963).

flowcharts were time-consuming to create and expensive to maintain, it was not wonder that most programmers did not bother with them.

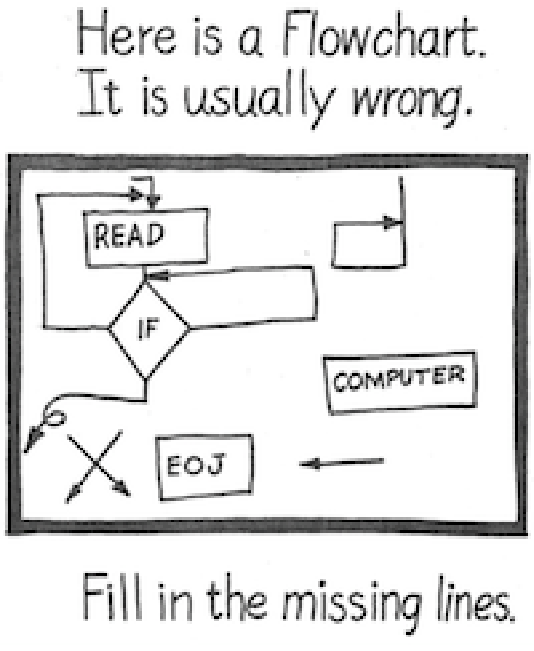


Figure 3: "The Programmer's Coloring Book," Datamation (September, 1963)

But if flowcharts failed to serve as design documents, they could be made to function as a communications technology. Despite efforts to cultivate good code commentary practices and other standardized documentary practices, reading and comprehending computer code remained notoriously difficult — even for the original author. In this context, the "expository" flowchart provided a form of visual documentation that facilitated understanding, memory, and conversation.[41] For those who viewed the flowchart as a design document, the fact that rarely corresponded to reality, or were produced only retrospective after the code was already written, was proof of their inherent insufficiency. The fact that they continued to be required by so many software development managers was a reflection of either unthinking adherence to tradition or bureaucratic incompetence. For those who believed flowcharts to be documentary or expository, however, none of these objections applied. If "flowcharts are primarily intended as tools for human communication," then it was possible for them to simultaneously beneficial *and* inaccurate, so long as they facilitated meaningful dialog between designers, users, and programmers.[42] And if the only flowcharts that could be considered definitely true to life were those created by machine and after the fact, then so be it. In fact, although Marty Goetz, the ADR product manager in charge of Autoflow, would later claim that Autoflow was popular because it allowed "strong programmers" to avoid the tedious work of drawing up a flowchart prior to writing their code,

[41]T C Willoughby and A D Arnold. "Communicating with decision tables, flowcharts, and prose". In: *SIGMIS Database* (1972).
[42]Yohe, "An Overview of Programming Practices".

the Autoflow marketing literature from this period makes it clear that ADR viewed flowcharts as an expository technology. Autoflow "provides hardcopy communication medium for all project personnel," "assists management in educating and training junior personnel," and "allows management to … review and supervise program activity and quality." In other words, Autoflow provided all of the documents essential to long-term software maintenance.[43]

As I have argued elsewhere, flowcharts served as both boundary objects and what Kathryn Henderson has called "conscription devices."[44](%20) They facilitated communication, but also served as a form of implied contract between the various actors in the software development project. Having the client or end-user sign off on a flowchart helped protect the project manager and programmers against "feature creep." At the same time, the flowchart provided some guarantee to the client or manager that the programmers would build the system that they (the client or manager) had requested, rather than the one that they (the programmers) thought was best or most interesting. In this sense, they were a form of insurance against the costs of future maintenance.

The Dull & Dirty Work of Maintenance

But while the academic literature emphasized the skill, creativity, and experience required to perform software maintenance, this did not translate in changed attitudes or practices in the corporate environment. As one article on the "myths of maintenance," described it, neither the activity nor those who performed it received any respect:

- "Maintenance is all 3 A.M. fixes and frantic hysterics. It's nothing we can anticipate and it doesn't take up that much time anyway".
- "Any of my programmers can maintain any program".
- "You don't get anywhere doing maintenance".
- "Maintenance is the place to dump your trainees, your burnouts and Joe, the boss' nephew, who thinks that hexadecimal is a trendy new disco. How can they hurt anything there?"[45]

Like all forms of maintenance, software maintenance is difficult, unpopular, and professionally unrewarding. To begin with, maintenance often required programmers to work on live systems, where mistakes and failures had real and immediate consequences. Because in the context of software development maintenance was rarely planned or budgeted for in advance, the work almost always performed under high-stress, emergency conditions. In the early 1960s, for example, the development of the IBM OS/360 operating system turned into an four-year long marathon that absorbed more than 5,000 staff years of effort and cost the company more than half-a-billion

[43]Gerardo Con Diaz. "Embodied Software: Patents and the History of Software Development, 1946-1970". In: *Annals of the History of Computing* (forthcoming, 2015).

[44]Nathan Ensmenger. "The Multiple Meanings of a Flowchart". In: *Information & Culture* (forthcoming); Kathryn Henderson. "Flexible sketches and inflexible data bases: Visual communication, conscription devices, and boundary objects in design engineering". In: *Science, Technology & Human Values* (1991).

[45]B. Schwartz. "Eight Myths about Software Maintenance". In: *Datamation* 28.9 (1982), pp. 125–128.

dollars — making it, at that point, the single largest expenditure in IBM history. Much of the expense and delay associated with the project were incurred not in the initial design and development of the software, but were the result of a series of redesigns required by a constantly evolving socio-technical environment. In his classic 1975 post-mortem analysis of the OS/360 debacle, *The Mythical Man-Month*, project manager Frederick Brooks argued that the real costs of the failure were human rather than financial, and were "best reckoned in terms of the toll it took on people: the managers who struggled to make and keep commitments to top management and to customers, and the programmers who worked long hours over a period of years, against obstacles of every sort, to deliver working programs of unprecedented complexity."[46] Many of the developers in the OS/360 groups would later leave the company, victims of a variety of stresses ranging from technological to physical.[47]

Because maintenance does not involve design, it was (and is) generally considered a routine and low-status activity.[48] Most often the work is assigned to students, newly hired employees, or poor-performing programmers.[49] Since few organizations consider maintenance a strategic function most provide software maintenance workers with little in terms of training, oversight, or rewards.[50] Neither is maintenance taught in most universities.[51] The result is a poorly trained and unmotivated workforce, low levels of job satisfaction, and high levels of employee turnover — a fact which, given the high-level of tacit knowledge involved in software maintenance, only serves to further compound the situation.[52]

The Durability of the Digital

One of the defining feature of software is that it is essentially a literary technology: the way the software works is determined, to a greater or lesser degree, by how its code is written. In *The Mythical Man-Month* Frederick Brooks famously compared computer code to poetry, arguing the "The programmer, like the poet, works only slightly removed from pure-thought stuff. He builds his castles in the air, from air, creating by exertion of the imagination."[53](%20)Donald Knuth similarly argued that computer programs were a form of literature, "fun to write, and

[46]Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley New York, 1975.

[47]Emerson Pugh, Lyle Johnson, and John Palmer. *IBM's 360 and Early 370 Systems*. MIT Press Cambridge, MA, 1991.

[48]William E Perry. *Managing systems maintenance*. Prentice Hall, 1983.

[49]T M Pigoski. "Practical software maintenance: best practices for managing your software investment". In: (1996); E B Swanson and C M Beath. "Maintaining information systems in organizations". In: (1989), pp.

[50]Canfora and Cimitile, *Software Maintenance*.

[51]Donna M. Kaminski. "An analysis of advanced C.S. students' experience with software maintenance". In: *CSC '88: Proceedings of the 1988 ACM sixteenth annual conference on Computer science*. Atlanta, Georgia, United States: ACM, 1988, pp. 546–550.

[52]Gary M Bronstein and Robert I Okamoto. "I'm OK, You're OK, Maintenance is OK". in: *Computerworld* 15.2 (1981), pp. 20–24; Pankaj Bhatt, Gautam Shroff, and Arun K. Misra. "Dynamics of software maintenance". In: *SIGSOFT Softw. Eng. Notes* 29.5 (2004), pp. 1–5.

[53]Brooks, *The Mythical Man-Month: Essays on Software Engineering*.

… a pleasure for other people to read."[54] Both men were referring both to specific character of computer programming — that is, that to code requires one to write — and to the larger creative and aesthetic dimensions of the end product. Such literary metaphors were and are common among software developers.[55]

But to argue that software can be written is also to suggest that it can also be readily rewritten. As Brooks argued of his code-poetry, "Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures."[56] If this were true, then all software was contingent and transitional, subject to constant renegotiation and redesign. Whereas conventional engineers and architects had to plan carefully before committing their ideas to manufacture, computer programmers faced no material constraints on their creativity. While this allowed programmers an unprecedented degree of flexibility and autonomy ("build first and draw up the specification afterwards," was a frequent rallying cry in the software industry), it also created unrealistic expectations on the part of the ultimate end-users of their software applications. "The computer is the Proteus of machines," argued Seymour Papert, repeating a commonly held-perception, "Its essence is its universality."[57] But the universality of the computer, made possible by its literary nature of its software, meant that computer/software system was perpetually a work in progress, with new features being requested, functionality demanded, and new bugs introduced. Changing the software was as simple as (or even identical to) rewriting the design specification — or so it seemed to many non-programmers.

But in working software systems, it is often impossible to isolate the literary artifact from its material embodiment in a larger sociotechnical context. Despite the fact that the material costs associated with building software are small in comparison with traditional, physical systems, the degree to which software is embedded in larger, heterogeneous networks makes starting from scratch almost impossible.

While it might be true that when a programmer is working alone, or beginning work on an entirely new project, that he or she has almost complete creative control over "the media of creation," this would have been a rare occurrence — at least in the corporate context. When charged with maintaining a "legacy" system (a category that encompasses most active software projects), the programmer is working not with a blank slate, but a palimpsest. Because software is a tangible record, not only of the intentions of the original designer, but of the social, technological, and organization context in which it was developed, it cannot be easily modified. "We never have a clean slate," argued Barjne Stroudstroup, the creator of the widely used C++ programming language, "Whatever new we do must make it possible for people to make a transition from old tools and ideas to new."[58] In this sense, software is less like a poem and more like a contract, a constitution, or a covenant. Software is history, organization, and social relationships made

[54]Donald Knuth. *Literate Programming*. Center for the Study of Language and Information Stanford, CA, 1992; Donald E Knuth. "Computer programming as an art". In: *Communications of the ACM* (1974).

[55]Maurice Black. "The Art of Code". PhD thesis. University of Pennsylvania, 2002; Wendy Hui Kyong Chun. *Programmed Visions*. MIT Press, 2011, pp.

[56]Brooks, *The Mythical Man-Month: Essays on Software Engineering*, pp.

[57]S Papert. "Mindstorms: Children, computers, and powerful ideas". In: (1980).

[58]Bjarne Stroustrup. "A History of C++". In: *History of Programming Languages*. Ed. by T.M. Bergin and R.G. Gibson. ACM Press, 1996.

tangible.

One very concrete example of this can be found in the history of the ALGOL programming language. ALGOL (ALGOrithmic Language) was developed in the mid-1950s by an international collective of computer scientists, in part for use as a reference language in academic publications. In 1960 the ALGOL maintenance group decided to publish a "Taschenbuch" of algorithms to be published by Springer as a guidebook for researchers. But doing so would effectively "freeze" the language specification, prohibiting any future development. The reasons for technical, but institutional. In order to be useful for the Taschenbuch, the language had to remain static; but to remain relevant — or to simply continue to function on the machines on which it was currently implemented, the ALGOL specification needed to be flexible. In this case, the relationship between the two texts — one electronic, the other traditional — is particularly apparent.[59]

One of the remarkable implications of all of this is that the software industry, which many consider to be one of the fastest-moving and most innovative industries in the world, is perhaps the industry most constrained by its own history. As one observer recently noted, today there are still more than 240 million lines of computer code written in the programming language COBOL, which was first introduced in 1959 – and which was derided, even at its origins, as being backward looking and technically inferior. And yet 90% of the world's financial transactions are processed by applications written in COBOL, as is 75% of all business data processing. Five out of eight large corporations rely on COBOL code, many of them for mission critical applications. 70% of Merrill Lynch applications are coded in COBOL. The total value of active COBOL applications — many of them developed prior to the 1980s — is as high as $2 trillion.[60] All of this COBOL code needs to actively maintained, modified, and expanded. Maintenance is a central issue in the history of software, the history of computing, and the history of technology. We need to know more about it, and we need to take it more seriously.

[59]Peter Naur. "ALGOL 60 Maintenance". In: *ALGOL Bull.* 11 (1960), pp. 1–4.

[60]Michael Swaine. "Is Your Next Language COBOL?". In: *Dr. Dobbs Journal* (2008).