Fixing things that can never be broken:
Software maintenance as heterogeneous engineering

Nathan Ensmenger
University of Pennsylvania
SHOT Conference, October 2008

In the very last published article of his long and distinguished career, the eminent 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. Since the earliest days of electronic computing, the problem of software has loomed large in the industry literature. 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.

This paper focuses on the problem of maintenance in the history of software. As all of the papers in this session argue, 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.[2] But maintenance is also low-status, difficult, and risky. Engineers and inventors don't like maintenance, and generally don't do maintenance, and therefore historians of technology have largely ignored it.

The problem of maintenance is particularly challenging for the both practitioners and historians of computing. To begin with, in theory software should never need maintenance. 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). Occasionally a stray cosmic ray might flip an unexpected bit in a software system, causing an error, but generally speaking, software can never be broken.

[1] Michael S. Mahoney. "What Makes the History of Software Hard". In: *Annals of the History of Computing, IEEE* 30.3 (2008). Pp. 8–18. ISSN: 1058-6180.

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

Except that software does get broken. All the time. At great expense and inconvenience to its users. In fact, from the early 1960s on, software maintenance has represented between 50% and 70% of all total expenditures on software.[3] By the end of the 1960s, many observers were talking openly about a looming software crisis facing the computer industry. The historical literature as thus interpreted this in terms of the problem of software development. There is a strong argument to be made that the software crisis was equally a problem of software maintenance. The rising cost of software maintenance, argued Richard Canning, an influential industry observer, in a 1972 article, which already devoured as much as half or two-third of programming resources, was just the "tip of the maintenance iceberg".[4]

The crisis of software maintenance, like all of the various iterations of the perpetual software crisis, has been framed and reframed many times in order to serve different agendas. At times it has been framed in terms of an ongoing programming problem; 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 software maintenance 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.[5]

But whatever its causes, the reality of the crisis has always been widely accepted. From the 1960s to the present, software maintenance has absorbed between one-half and two-thirds of all software-related resources.[6] This is an extraordinary figure.

[3] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. "Characteristics of application software maintenance". In: *Commun. ACM* 21.6 (1978). Pp. 466–471. ISSN: 0001-0782; Girish Parikh. "Software maintenance: penny wise, program foolish". In: *SIGSOFT Softw. Eng. Notes* 10.5 (1985). Pp. 89–98. ISSN: 0163-5948; 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.

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

[5] Parikh, "Software maintenance: penny wise, program foolish".

[6] Gerardo Canfora and Aniello Cimitile. *Software Maintenance*. Tech. rep. University of Sannio, 2000.

The problem of software maintenance has a long history. Even before there was a word for 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."[7] 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?

In order to properly understand software maintenance, we must first come to grips with software itself.

Software as Heterogeneous Technology

Most of us today tend to think of software as a consumer good, a product, a prepackaged 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.

But historically speaking, software is not something you purchase off-the-shelf, nor is it a single application or products. Rather, it is a bundle of systems, services, and support. (Even today this is true of the vast majority of software)

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

[7] Maurice V. Wilkes. *Memoirs of a Computer Pioneer (History of Computing)*. 1985.

elements of a typical computer installation that were not obviously "tubes, transistors, wires, tapes and the like."[8] 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.[9]

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 became a product that could be purchased separately from 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 user of a computer, "the total computing facility provided for his use, other than the hardware, is the software."[10] 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

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

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

[10] Bernard Galler. "Definition of Software". In: *Communications of the ACM* 5.1 (1961). P. 6.

most organizations, by the end of the 1960s, software had become the computer: software, rather than the computer, had become the focus of all discussion, debate, and dissension within the computing community.

It is the expansiveness of software that is the key to understanding the nature and causes of the software maintenance crisis that emerged in the late 1960s. Unlike hardware, which is almost by definition a tangible "thing" which can readily be isolated, identified, and evaluated, software is inextricably linked to a larger socio-technical system that includes machines (computers and their associated peripherals), people (users, designers, and developers), and processes (the corporate payroll system, for example). The sociologist John Law calls the development of such complex systems "heterogeneous engineering"; historically speaking, the development of heterogeneous systems is fraught with conflict, negotiation, disputes over professional authority, and the conflation of social, political, and technological agendas.[11]

Software 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 is that it performs as expected, that the behavior of the system conforms to its original design or specification. 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. But such techniques were always problematic, and never widely adopted. In any case only a small percentage of software maintenance is devoted to fixing such bugs in implementation.[12] One exhaustive study from the early 1980s estimates such emergency fixes to occupy at most one-fifth of all software maintenance workers.

The majority of software maintenance involve what are vaguely referred to in the literature as "enhancements." These enhancements sometimes involved strictly

[11] John Law. "Notes on the Theory of Actor-Network: Ordering, Strategy, and Heterogeneity". In: *Systems Practice* (1992). Pp. 379–393.

[12] 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. ISSN: 0163-5948.

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."[13] This included the introduction of new functionality, as dictated by market, organizational, or legislative develops, but also changes in the larger technological or organizational system in which the software was inextricably bound. Software maintenance also 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."[14]

By the early 1980s, the industry and technical literature had settled on a shared taxonomy for talking about software maintenance: There was corrective maintenance (bug fixes), perfective maintenance (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."[15] But software maintenance was the term that stuck..

The Challenge of Maintenance

Like all forms of maintenance, software maintenance is difficult, unpopular, and largely unrewarding. To begin with, maintenance required programmers to work on live systems, where mistakes and failures had real and immediate consequences. Because maintenance does not generally involve design, it is considered boring and low-status. And because of the unique nature of software – its intangibility – software systems are often coded before they are completely specified. Many programmers find it easy to "just start coding" than to develop design documents.

[13] Canning, op. cit.

[14] 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.

[15] Girish Parikh. "What is software maintenance really?: what is in a name?". In: *SIGSOFT Softw. Eng. Notes* 9.2 (1984). Pp. 114–116. ISSN: 0163-5948.

In software more than any other engineering discipline, the "not-invented-here" syndrome prevails. It is generally considered much simpler to redevelop systems from scratch than to untangle someone else's "spaghetti code." Most programs are poorly documented (if at all), and so most maintenance works involves intensive on-the-job learning.

But in working software systems, it is often impossible to isolate the artifact from its context. Despite the fact that the material costs associated with building software are low (in comparison with traditional, physical systems), the degree to which software is embedded in larger, heterogeneous systems makes starting from scratch almost impossible. In his highly regarded book *The Mythical Man-Month*, the computer scientist (and IBM program manager) Frederick Brooks famously likened programming to poetry, suggesting that "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."[16] To a degree, this is true – at least when the programmer is working on constructing a new system. But when charged with maintaining so-called "legacy" system, the programmer is working not with a blank slate, but a palimpsest. Computer code is indeed a kind of writing, and software development a form of literary production. But the ease with which computer code can be written, modified, and deleted belies the durability of the underlying document . 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."[17] 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 tangible.

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

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

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

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 substantially. 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.[18] 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.

The good news, of course, is that, at least in the case of software, the problem of ongoing maintenance necessitates the understanding and untangling of social structures, historical contingencies, and the accretion of change over time, then we as historians should be well-prepared to engage with it.

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