When Good Software Goes Bad
The Surprising Durability of an Ephemeral Technology

Nathan Ensmenger • Indiana University

Even before there was a word for software, there was a problem of software maintenance. Maurice Wilkes, one of the first men ever to program a modern, stored-program digital computer, loved to recall in later life the "exact instant" in June 1949 when he "realized that a good part of the remainder of my life was going to be spent in finding errors in my own programs."[1] Technically the process Wilkes was describing is called debugging (fixing unintentional mistakes in design and/or construction) rather than maintenance (the repair or upkeep of an already functional system), but such fine distinctions are irrelevant to the generations of software developers who

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

would inevitably recapitulate Wilke's original epiphany: the delivery of working code was only the beginning of the life-cycle of any significant software application. A programmer could — and many did — spend the majority of their careers fixing, repairing, and extending their own (or, even worse, other people's) "legacy" systems. In this respect, according to the well-worn witticism, programming a computer was a little bit like sex: "One mistake and you have to support it for the rest of your life."

The idea that software systems would require constant maintenance is perplexing, almost paradoxical. After all, one of the defining features of software is its intangibility and ephemerality. The digital nature of software code means that, to the degree that it participates at all in material reality (for example, as a series of holes on a paper tape or dots and dashes of magnetic iron on a disc), it is in a form that can be readily disembodied and perfectly (and infinitely) replicated.

The virtuality of software (indeed, the terms have become almost synonymous) means that, in theory, software systems do not break down, or wear out, or require lubrication. Once a software-based system is working, it should work forever (or at least until the underlying hardware platform it runs on fails, and that is somebody else's problem). Even if there are "bugs" that are subsequently revealed in the system, these are considered flaws in the original design or implementation, not the result of the wear-and-tear of daily use. In principle, such bugs could be completely eliminated by rigorous development and testing methods. Occasionally a stray cosmic ray might flip an unexpected bit in a software application, causing an error, but generally speaking, software is a technology that should 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.[2] In 1995 firms in the United States alone spent $70 billion on maintenance of more than ten billion lines of legacy software code.[3] The total maintenance costs associated

[2] 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.

[3] J Sutherland. "Business objects in corporate information systems". In: *ACM Computing Surveys*

with the Y2k Crisis has been estimated at more than $300 billion.[4] 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.

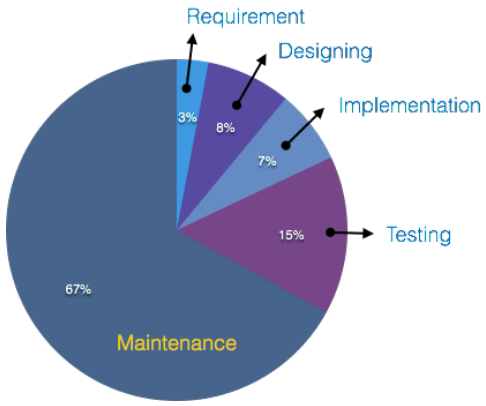

Figure 1: The Allocation of Software Effort

Errors, Enhancements, and Evolution

What, then, does maintenance mean in the context of software development? How does one repair an unbreakable technology?

Obviously, much hinges on the notion of what it means to for software to truly remain unbroken. Or, to flip the question around, on what it means for software to "work." This is a surprisingly difficult question to answer, and one which has plagued software developers since the earliest days of electronic computing.

The most straightforward 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,

(CSUR) (1995).

[4]Robert Mitchell. "Y2K: The good, the bad and the crazy". In: *Computerworld* (2009).

a somewhat arbitrary distinction, but one with critical implications for the budget, schedule, and perception of a project.[5] 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).[6] But more common were bugs that involved rare or edge cases that called into question basic questions about what 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.[7] 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.[8] One exhaustive study from the early 1980s estimates such emergency repairs to occupy at most one-fifth of all

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

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

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

[8] 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.

software maintenance workers.

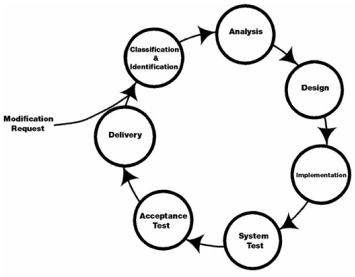

Figure 2: The Eternal Cycle of Software Maintenance

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."[9] 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."[10]

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

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

[10] 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.

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:

> 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.[11]

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.[12] 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.

It is interesting to note the high degree of sophistication of sociological analysis that can be found in the maintenance literature. 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

[11] 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.

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

Lehman's Law #1:

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

Figure 3: The First of Lehman's Eight Laws of Software Maintenance

bus" book.[13] 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."[14] 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."[15] Maintainable software was software that could

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

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

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

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.

The Dull & Dirty Work of Maintenance

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 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."[16] 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.[17]

Because maintenance does not involve design, it was (and is) generally considered a routine and low-status activity.[18] Most often the work is assigned to students, newly hired employees, or poor-performing programmers.[19] As one article on the "myths of

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

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

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

[19] 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".

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?"[20]

Since few organizations consider maintenance a strategic function most provide software maintenance workers with little in terms of training, oversight, or rewards.[21] Neither is maintenance taught in most universities.[22] 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.[23]

The Durability of the Digital

There are perhaps three important lessons to be learned from the history of software maintenance. One has to do with what Gerardo Con Diaz has called the "contested ontology" of software: that is to say, that debates about how software can be "broken" (from a technical, organizational, or legal point of view) are in fact often maneuvers in a larger conflict over the true nature of the thing itself. Is software art, science, or technology? Each perspective implies something different about what can go wrong with software, and how to fix it.

In: (1989), pp.

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

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

[22] 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.

[23] 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.

Another important lesson has to do with the supposed intangibility of software. It is often claimed that software is an essentially 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."[24] [57] Donald Knuth similarly argued that computer programs were a form of literature, "fun to write, and … a pleasure for other people to read."[25] 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.[26]

Figure 4: Obligatory Dilbert Cartoon

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."[27] 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,

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

[25] 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).

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

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

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."[28] 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."[29] 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.

And finally, the history of software maintenance might provide a new perspectives on the work of maintenance more generally. Because the distinction between design and implementation is *less* concrete in software development than most forms of technological construction (although, as was previously mentioned, not non-existent),

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

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

the relationship between these two facets of invention in certain respects becomes more clear.

For example, in the early 1990s the computer programmer Ward Cunningham introduced the metaphor of "technical debt" to describe the trade-off between planning and maintenance in software design and development. According to Ward,

> "Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite… The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise."[30]

Although the notion of technical debt was meant to highlight the importance of not ignoring the need for ongoing maintenance (therefore offloading on to some future developer), in software development as in high finance, the manipulation of debt has itself now become a business strategy. Companies can now deliberately accumulate "valuable" debt that can be traded, arbitraged, or off-loaded. Whether this works any better in software than in sub-prime mortgages is an open question, although the history of software suggests that the answer is negative…

[30] Ward Cunningham, "The WyCash Portfolio Management System," OOPSLA '92 Experience Report. [http://c2.com/doc/oopsla92.html]