Think Piece

Software as History Embodied

Nathan Ensmenger, Editor

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 electronic computing's earliest days, the problem of software has loomed large in the industry literature. The history of software is hard, Mahoney argued, because software itself is hard: hard to design, develop, use, understand, and maintain.

It is this last challenge posed by software—the challenge of software maintenance—that I take up in this essay.

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—therefore, historians of technology have largely ignored it.

Unbreakable software?

Nowhere is this dislike of maintenance more apparent than in software development. Not only are software developers particularly averse to working on other people's systems (the infamous ''not invented here'' syndrome), but software itself is considered essentially unbreakable. Software does not wear out or break down in the traditional sense. Once a software-based system is working, it *should* work forever (or at least until the underlying hardware breaks down—but that's someone else's problem). Any latent ''bugs'' 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.

But despite the fact that software in theory never breaks down, in most large software projects maintenance represents the single most time-consuming and expensive phase of development. Since the early 1960s, software maintenance has been a continuous source of tension between computer programmers and users. In a 1972 article, the influential industry analyst Richard Canning argued that the rising cost of software maintenance, which by then already devoured as much as half or two-thirds of programming resources, was just the ''tip of the maintenance iceberg.''[3] Today, software maintenance is estimated to represent between 50% and 70% of all software expenditures.[4]

So if software is an artifact that, in theory, can never be broken, what is software maintenance? Although software does not wear out or break down in any traditional sense, what does ''break'' over time is the larger context of use. To borrow a concept from John Law and Donald MacKenzie, software is a heterogeneous technology. Unlike computer hardware, which is by definition a tangible ''thing'' that can be readily isolated, identified, and evaluated (and whose maintenance can be anticipated and accounted for), computer software has been inextricably linked to a larger sociotechnical system that includes machines (computers and their associated peripherals), people (users, designers, and developers), and processes (the corporate payroll system, for example). Software maintenance is therefore as much a social as a technological endeavor. Usually what needs to be ''fixed'' is the ongoing negotiation between the expectations of users, the larger context of use and operation, and the features of the software system in question.

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 software maintenance problem has historically been so complex. To begin with, it raises a fundamental question—one that has plagued software developers since the advent 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 system behavior conforms to its original design specification. But only a small percentage of software maintenance is devoted to fixing bugs in implementation.[5]

Most software maintenance involves what are vaguely referred to in the literature as ''enhancements.'' Enhancements sometimes involved strictly technical measures—such as implementing performance optimizations—but most often what Richard Canning termed ''responses to changes in the business environment.'' This included the introduction of new functionality, as dictated by market, organizational, or legislative developments, but also changes in the larger technological or organizational system in which the software was inextricably bound. Software maintenance also incorporated such

continued on p. 86

 1058-6180/09/$25.00

continued from p. 88

apparently nontechnical tasks as documentation, training, support, and management.[6] In the technical literature that emerged in the 1980s, this ''adaptive'' dimension so dominated the larger problem of maintenance that some observers pushed for the abandonment of the term *maintenance* altogether. The process of adapting software to change would better be described as ''software support,'' ''software evolution,'' or (my personal favorite) ''continuation engineering.''[7]

Imaginary yet tangible

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.''[8] To a degree, Brooks's fanciful metaphor is entirely accurate—at least when the programmer is working on constructing a new system. But when charged with maintaining a 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 Bjarne Stroustrup, 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.''[9] In this sense, software is less like a poem and more like a contract, a constitution, or a covenant. 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. Software is history, organization, and social relationships made tangible.

One of the remarkable implications of all 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 Cobol, which was first introduced in 1959.[10] All of this Cobol code needs to be actively maintained, modified, and expanded. Maintenance is central to the histories of software, computing, and technology. We need to know more about it, and we need to take it more seriously.

For practitioners, the tangled web of history embedded within legacy code is precisely what makes software maintenance so difficult; for historians, it represents only a remarkable opportunity.

References and notes

1. M.S. Mahoney, ''What Makes the History of Software Hard,'' *IEEE Annals of the History of Computing,* vol. 30, no. 3, 2004, pp. 8-18.
2. D. Edgerton, *The Shock of the Old: Technology and Global History since 1900,* Oxford Univ. Press, 2007.
3. R. Canning, ''The Maintenance 'Iceberg','' *EDP Analyzer,* vol. 10, no. 10, 1972, pp. 1-14.
4. G. Parikh, ''Maintenance: Penny Wise, Program Foolish,'' *SIGSOFT Software Eng. Notes,* vol. 10, no. 5, 1985.
5. D.C. Rine, ''A Short Overview of a History of Software Maintenance: As It Pertains to Reuse,'' *SIGSOFT Software Eng. Notes,* vol. 16, no. 4, 1991, pp. 60-63.
6. E. Burton Swanson, ''The Dimensions of Maintenance,'' *Proc. 2nd Int'l Conf. Software Eng.* (ICSE 76), IEEE Computer Soc. Press, 1976, pp. 492-497.
7. G. Parikh, ''What Is Software Maintenance Really? What Is In A Name?'' *SIGSOFT Software Eng. Notes,* vol. 9, no. 2, 1984, pp. 114-116.
8. F. Brooks, *The Mythical Man-Month: Essays on Software Engineering,* Addison-Wesley, 1975.
9. B. Stroustrup, ''A History of C++,'' *History of Programming Languages,* T.M. Bergin and R.G. Gibson, eds., ACM Press, 1996.
10. M. Swaine, ''Is Your Next Language COBOL?'' *Dr. Dobbs J.,* 18 Sept. 2008.

Contact department editor Nathan Ensmenger at annals-thinkpiece@computer.org.