What Makes the History of Software Hard

Michael S. Mahoney
Princeton University

History of commitments constrains choice. Narrow incentives and opportunities motivate choice.
—Rob Kling and Walt Scacchi[1]

We never have a clean slate. Whatever new we do must make it possible for people to make a transition from old tools and ideas to new.
—Bjarne Stroustrup[2]

Creating software for work in the world has meant translating into computational models the knowledge and practices of the people who have been doing that work without computers. What people know and do reflects their particular historical experience, which also shapes decisions about what can be automated and how. Software thus involves many histories and a variety of sources to be read in new ways.

This article originally appeared in L. Böszörmenyi, ed., MEDICHI 2007—Methodic and Didactic Challenges of the History of Informatics, *Austrian Computer Society (OCG), Klagenfurt, Austria, 178 pp. The Austrian Computer Society (OCG) has kindly granted permission for an edited version to be published by the* Annals.*[3]*

The history of software is the history of how various communities of practitioners have put their portion of the world into the computer. That has meant translating their experience and understanding of the world into computational models, which in turn has meant creating new ways of thinking about the world computationally and devising new tools for expressing that thinking in the form of working programs. It has also meant deciding, in each realm of practice, which aspects of experience can be computed and which cannot, and establishing a balance between them. Thus, the models and tools that constitute software reflect the histories of the communities that created them and cannot be understood without knowledge of those histories, which extend beyond computers and computing to encompass the full range of human activities. All software, even the most current, is in that sense ''legacy'' software. That is what makes the history of software hard, and it is why the history of software matters to the current practitioner.

Decentering the machine

Until recently, the history of computing has focused largely on the machines themselves, tracing the path by which the electronic digital computer emerged from a long history of efforts to mechanize calculation and then evolved into a variety of increasingly complex forms in response to the possibilities opened by the transistor, the integrated circuit, and magnetic and optical media of ever greater capacity. This focus has led to a familiar narrative (see Figure 1), which begins with the abacus and then follows a by now familiar sequence of devices—mechanical and electromechanical, analog and digital—all converging on ENIAC, or more broadly on the several electronic devices of a similar nature conceived and built at the time. From ENIAC the story moves to the EDVAC design and to its early realizations.[4] The mainframe era then begins, followed by the short reign of the minicomputer and then by the current age of the microcomputer in ever more powerful form. The sequence is commonly treated as a natural evolution of forms. As each new generation appears, the old disappears from the story, as if mainframes went the way of the dinosaur when minicomputers appeared, and they in turn followed the mainframe into obsolescence as the microcomputer opened a new era of personal computing.

It would be hard to know from most histories of this sort that IBM still derives a major share of its income from mainframe computers, especially as accounts of the Internet focus on the PC clients to the exclusion of the mainframe servers. Similarly, emphasis on the PC has hidden the ways in which with their growing power they have incorporated minicomputer architectures and systems and, with them, their mainframe legacy. Also lost from view with a focus on the visible PC is embedded computing, which accounts for the vast majority of the microprocessors at work today.

To the extent that the standard narrative covers software, the story follows the generations of machines, with an emphasis on systems software, beginning with programming languages and touching—in most cases, just touching—on operating systems, at least up to the appearance of time-sharing. With a nod toward Unix in the 1970s, the story moves quickly to personal computing software and the story of Microsoft, seldom probing deep enough to reveal the roots of that software in the earlier period. One may learn a bit about the beginnings of the data processing industry, since it is synonymous with IBM, but there is little if anything about the building of the first large systems in business, industry, and government. For this story, one must delve into the specialist literature, much of it still dominated by "pioneer" accounts. The software of the embedded systems underlying the world's technical, industrial, and financial infrastructure remains, of course, as invisible to the historical record as do the devices on which it runs.

Indeed, with respect to applications software, the standard narrative converges in another way. Just as it places all earlier calculating devices on one or more lines leading toward the electronic digital computer, as if they were somehow all headed in its direction, so too it pulls together the various contexts in which the devices were built, as if they constituted a growing demand for the invention of the computer and as if its appearance was a response to that demand. That way of telling the story is historically

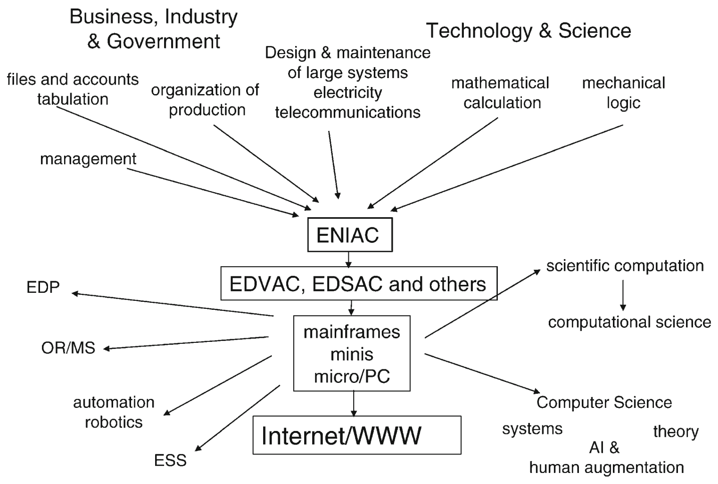

Figure 1. The convergent account.

misleading in at least two ways. First, it makes the application of the computer in a wide range of areas unproblematic. If people have been waiting for the computer to appear as the desired solution to their problems, it is not surprising that they then make use of it when it appears, or indeed that they know how to make use of it. By the same token, it then becomes difficult to explain, for example, why IBM would have hesitated to enter the computer market in the late 1940s, or why the National Institutes of Health had to launch a program to persuade biologists that computers could be of use to them.[5] Second, the convergent account sets up a narrative of revolutionary impact, in which the computer is brought to bear on one area after another, in each case with radically transformative effect.

There is a lot of "revolution talk" in computing. It dates back at least to 1962 with the publication of Edmund C. Berkeley's *The Computer Revolution*, in which the author made the computer part of the "second industrial revolution" and fretted over what was to become of humans in an age of *Giant Brains, or Machines That Think* (the title of his 1949 book). One can hardly pick up a journal in computing today without encountering some sort of revolution in the making, usually proclaimed by someone with something to sell. Critical readers recognize most of it as hype based on future promise rather than present performance, and those with long memories recall the many revolutions that have been canceled or postponed owing to technical difficulties.

Historians have grown wary of the notion of revolutions, especially those that purport to erase the past or render it obsolete, as if societies could start from scratch. There have been episodes in which societies changed radically over a relatively short period, and "revolution" seems an appropriate designation for the process, for example, the Industrial Revolution in England in the late 18th and early 19th centuries. But even there one must be careful, because revolution talk tends to hide the continuities that tie the present to the past, or rather the continuities through which the past informs the present. Nothing is in fact unprecedented, if only because we use precedents to recognize, accommodate, and shape the new. In our personal encounters with new situations, we turn to our memory for similar situations as a guide to how to respond. History is our community memory, and it is as much embedded in our lives as our personal memories. Through language, customs, education, and institutions, it informs the way we think and the way we do things. It shapes our expectations and our actions; it is built into our practices.[6] As a senior colleague of mine once remarked to a group of scientists, the question is not whether one does history but whether one does it well. For example, the quest for a discipline of software engineering began with a search for historical forebears, and the subsequent literature is filled with appeals to the history of technology, some of them well founded, others not.[7]

Histories of computing

The main problem with the revolutionary or impact model is that it attributes agency to the computer as if it had a nature of its own that it could impose on subjects or activities to which it is applied, that is, as if the computer in and of itself had the power to transform those activities. We speak of "the impact of the computer on X," but to speak of "the computer" as if it were one thing is misleading. There is no one computer, but only computers. Even in the most general form of a finite Turing machine, the computer is a schema rather than a single device. What it can do depends on the finite state table and on the contents of the tape. In its most general, or universal, form it can do anything for which we can provide suitable instructions, that is, the table for a specific Turing machine. That is the source of its power: it is a protean machine. However, precisely because it can do anything, it can do nothing in and of itself. It does things only when we provide the programs that cause the universal machine to emulate particular machines of our design.

The history of computing, especially of software, should strive to preserve human agency by structuring its narratives around people facing choices and making decisions.

In the impact model, "the computer" happens to people, who are put in the position of responding to it. But computers do nothing without programs, and programs do not just happen by themselves. People design them for their own purposes. Hence, the history of computing, especially of software, should strive to preserve human agency by structuring its narratives around people facing choices and making decisions instead of around impersonal forces pushing people in a predetermined direction. Both the choices and the decisions are constrained by the limits and possibilities of the state of the art at the time, and the state of the art embodies its history to that point. Much more than the art of computing is involved.

When, then, the first electronic digital computers appeared, they had no inherent form but rather the potential to assume an almost infinite variety of forms. Moreover, there was nothing in the conceptual structure of the device that dictated how it might be programmed to take on those forms. What it would become depended on what people wanted to make it. The first machines continued to serve the purposes for which ENIAC had been created, namely numerical calculation. As the first general-purpose machines became commercially available in the early 1950s, they attracted the interest of a variety of communities of practitioners beyond science and engineering. These different groups saw different possibilities in the computer, and they had different experiences as they sought to realize those possibilities, often translating those experiences into demands on the computing community, which itself was only taking shape at the time.

It is important to emphasize, as the diagram in Figure 2 does, that what I would like to call the "communities of computing" had their own histories, that is, their own ways of doing things based on past experience. Only one community's history coincided with that of the computer itself, namely that of the "number crunchers."[8] The others, including the nonmathematical sciences, had been getting along without computers (and not missing them), creating devices specifically for their needs as in the case of the tabulator. The chart shows some of the main groups. Let me say a word about them.

What we now call "data processing" has a continuous history as old as writing itself. Some of the oldest documents are inventories and financial records kept by professional scribes. As James Cortada and others have recently reminded us, IBM was in the business of data processing long before the appearance of the computer, as was Remington-Rand, and electronic data processing reflects that long prior experience.[9]

In one sense, management is as old as record-keeping. But, if one focuses on organization for the purposes of gathering and disseminating information, then the history of management in the modern age is rooted in the growth of bureaucracies and the transfer of their methods to commerce and industry. In *The Government Machine*, Jon Agar has recently traced the development of Britain's bureaucracy, conceived from the late 18th century as a machine, and has argued that the very notion of a general-purpose computer derives from that model rather than the other way around.[10]

The organization of production also has a long history but took a decisive turn during the Industrial Revolution. One can watch the subject taking new form in the papers of James Watt and Matthew Boulton, who designed factories as well as building steam engines for them, and in Charles Babbage's *Economy of Machinery and Manufactures* (1833). It reached a mature form with the work of Frederick W. Taylor and his colleagues, in particular the time-and-motion studies of Frank Gilbreth, and took its distinctively 20th-century shape in the factories of Henry Ford, who was pursuing automation long before the term was coined in the early years of computing.[11] The organization of large systems and networks, with its ties both to management and the organization of production, has roots going back at least to the appearance of the railroad, telegraph, and telephone in the 19th century and reached an advanced state in the electrical distribution systems built in the US and Europe at the turn of the 20th.[12]

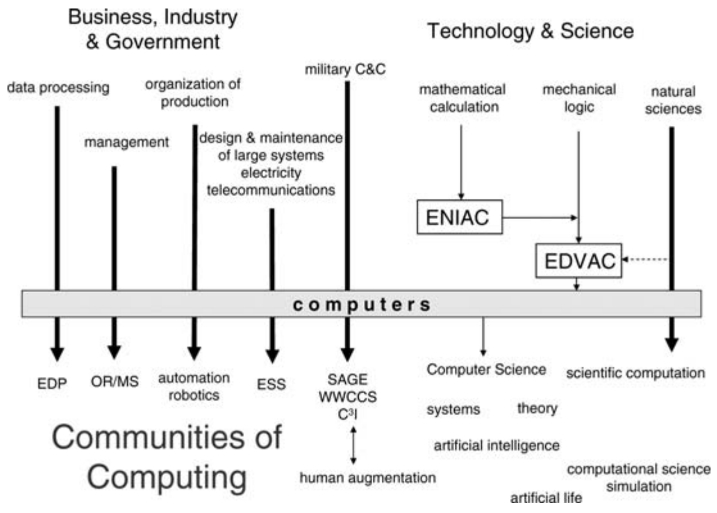

Figure 2. Communities of computing.

Systems of military command and control, including the logistics of supply, are as old as warfare and reached new levels of sophistication in the 19th and early 20th centuries. Indeed, in light of recent literature on the subject, we need a fuller and more subtle understanding than we have of the history of the community of practitioners in the military, to compensate for the visions of the future designed for it by researchers outside it. An important spin-off of this line of research was the field of human–computer interaction, aimed at the augmentation of human skills by the computer. As David Mindell has shown recently in *Between Human and Machines*, that field too had a history of its own before the computer.[13]

New fields emerged alongside those that predated the computer. Although there was at first some disagreement about whether computers were sufficiently different from early calculating machines to constitute a subject of study in themselves, the work of John von Neumann and Alan Turing pointed to the theoretical potential of the device. Subsequently, the need to develop systems both to make the computer easier to use and to keep it running efficiently opened a range of theoretical and technical issues that prompted the emergence of the new subject of computer science or informatics, as it is known in the non-Anglophonic world.[14]

Until recently, the history of computing in these fields has been written in terms of the machine and its impact (revolutionary, of

Few substantive analyses of software failures can be found, even in the software engineering classroom. Other branches of engineering study their mistakes in order to learn from them.

course) on them. The emphasis has lain on what the computer could do rather than on how the computer was made to do it. Yet, as just noted, all these fields and others consisted of established bodies of proven practice, developed without reference to computing. It was from the perspective of that practice that people in those fields viewed the possibilities offered by computers to enhance it.[15] What people did with computers reflected the histories of their own fields; computers had no useful history of their own to contribute. Hence, once one moves beyond the hardware and systems software in its narrowest sense, computing becomes part of those histories, and its history in turn reflects its encounter with them. Indeed, even systems software derived in many respects from earlier, non-computer models of corporate organization and management, as in the case of layered operating systems arranged in a hierarchy of authority and access or, for that matter, of the design of Cobol as a language for business data processing. In that sense, computing has not one but many histories. That is one thing that makes the history (or should I say histories?) of software hard. It is not primarily about computers.

Perhaps for that reason, the history of software has so far either remained close to the machine or has stepped back altogether to view software in the large and as an object of commerce. In the former instance, we know a great deal about the history of programming languages and a bit about the history of operating systems, databases, and other varieties of systems software. But historians have scarcely scratched the surface of applications software, the software that actually gets things done outside the world of the computer itself. In the second case, we have studies of the software industry, both commercial and personal; of programming as a labor process and nascent profession; or of software engineering and its origins in the (continuing) "software crisis." These were the perspectives from which a conference at the Heinz Nixdorf Forum in 2000 attempted to map the history of software. In retrospect, what the conference missed was software as model, software as experience, software as medium of thought and action, software as environment within which people work and live. It did not consider the question of how we have put the world into computers, and thus how we have come to view the world through a computational lens.[16]

Worlds of software

That process has not been easy or straightforward, as any programmer will testify. If it appears so, it is because so far we have concentrated on the success stories and told them in a way that masks the compromises between what was intended and what could be realized. Programming is where enthusiasm meets reality. The enduring experience of the communities of computing has been the huge gap between what we can imagine computers doing and what we can actually make them do. There have been (and continue to be) massive failures in software development, which have cost money, time, property, and even lives. That is what gave rise in the late 1960s to the notion of a "software crisis" and in the mid-1980s to the ACM Forum on System Reliability and Risks to the Public, maintained since its inception by Peter G. Neumann both online and in SIGSOFT's *Software Engineering Notes*.[17] Except for Frederick P. Brooks's famous *Mythical Man-Month*, we have few substantive analyses of such failures, even and especially in the software engineering classroom, where we might expect to find them. As software engineer/historian James Tomayko has pointed out with a nod to Henry Petroski, other branches of engineering study their mistakes to learn from them.[18] Software will look more human when we take seriously the difficulties of designing and building it and take critical account of some of its failures.

As already noted, responses to the "software crisis" included calls for a discipline of "software engineering" that would, in the words of a deliberately provocative definition, base "software manufacture … on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering."[19] (That is, of course, a historical program.) At first, practitioners sought a solution in the computer and looked to the improvement of their tools and to more effective project management. A great deal of effort went into developing high-level programming languages and diagnostic compilers for them. The languages were specifically designed to foster good programming practices (read: proper ways to think about programming). Subsequent research aimed at transforming and extending the languages into specification and design tools with similar diagnostic capabilities. At the same time, practitioners sought to bring the experience of industrial engineering to bear on software production, with an eye toward automating it in the form of a "software factory," a programming environment that would leave the programmer little choice but to do it right. Evident in the literature is the long history of the factory predating the computer.

Better tools did improve the programming. But, as Brooks has since pointed out, that is not where the real problems have lain, or rather problems at that level were only "accidental."[20] Almost from the start, studies showed that the bulk of the errors occurred at the beginning of projects, before programming ever began (or should have begun). The errors were rooted in failures to understand what was required, to specify completely and consistently how the system was supposed to behave, to anticipate what could go wrong and how to respond, and so on. As many as two-thirds of the errors uncovered during testing could be traced back to inadequate *design*; the longer they remained undetected, the more costly and difficult they were to correct.

Design is not primarily about computing as commonly understood, that is, about computers and programming. It is about modeling the world in the computer: translating a portion of the world into terms a computer can "understand." Here it may help to go back to the protean scheme to recall what computers do. They take sequences, or strings, of symbols and transform them into other strings. The symbols and the strings may have several levels of structure, from bits to bytes to groups of bytes to groups of groups of bytes, and one may think of the transformations as acting on particular levels. But in the end, computation is about rewriting strings of symbols.

Studies showed that the bulk of software errors occurred at the beginning of projects, before programming ever began.

The transformations themselves are strictly syntactical, or structural. They may have a semantics in the sense that certain symbols or sequences of symbols are transformed in certain ways, but even that semantics is syntactically defined. Any meaning the symbols may have is acquired and expressed at the interface between a computation and the world in which it is embedded. The symbols and their combinations express representations of the world, which have meaning to us, not to the computer. It is a matter of representations in and representations out. What characterizes the representations is that they are operative.[21] Through the computer, we can manipulate them, and they in turn can trigger actions in the world. What we can make computers do depends on how we can represent in the symbols of computation portions of the world of interest to us and how we can translate the resulting transformed representation into desired actions. A Boeing 777 has a variety of representations: its form, structure, flight dynamics, controls. They not only direct the design of the aircraft and the machining and assembly of its components, but they then interactively direct the control surfaces of the aircraft in flight. That is what I mean by "operative representation."

So putting a portion of the world into the computer means designing an operative representation of that portion of the world that captures what we take to be its essential features. That has proved, as I say, no easy task; on the contrary it has proved difficult, frustrating, and in some cases disastrous. It has most recently moved to a high-priority problem at the US National Science Foundation, which has undertaken a program aimed at exploring the "science of design."[22] Where that will go is anyone's guess at the moment, and I'm a poor prognosticator. What is clear is that historians of computing have inherited the problems to which it is addressed. If we want critical understanding of how various communities of computing have put their portion of the world into software, we must uncover the operative representations they have designed and constructed, and that may prove almost as difficult a task because of the nature of the historical sources in which they are embedded and expressed.

Reading software as artifact

What are the sources of the history of software, and how do we read them? For perspective on the question, consider the larger subject of history of technology. Many years ago, when I started teaching the subject, I tried to follow my practice in history of science of assigning original sources, that is, the writings of the scientists themselves. So I looked for original sources of technology and had trouble finding them, until it occurred to me that I was looking in the wrong place, in a library and not in a museum—because the sources of technology are the artifacts themselves. As Derek J. de Solla Price so nicely put it, engineers and inventors "think with their fingertips." Henry Ford put it another way:

> There is an immense amount to be learned simply by tinkering with things. It is not possible to learn from books how everything is made—and a real mechanic ought to know how nearly everything is made. Machines are to a mechanic what books are to a writer. He gets ideas from them, and if he has any brains he will apply those ideas.[23]

What holds for mechanics holds too for historians of technology. We must tinker with the things to discover the ideas that underlay and informed them.

What would it mean to "read" an artifact of computing? It would look much like examples from other areas of the history of technology. We have a classic example in Tracy Kidder's *Soul of a New Machine*, the fascinating story of the designing of Data General's first 32-bit minicomputer, the Nova. The head of the project, Tom West, had managed to get access to one of DEC's new VAX machines and to pull the boards for close examination:

> Looking into the VAX, West had imagined he saw a diagram of DEC's corporate organization. He felt that the VAX was too complicated. He did not like, for instance, the system by which various parts of the machine communicated with each other; for his taste, there was too much protocol involved. He decided that VAX embodied flaws in DEC's corporate organization. The machine expressed that phenomenally successful company's cautious, bureaucratic style. Was this true? West said it didn't matter, it was a useful theory.[24]

Here we have an example for hardware of Conway's law, itself a suggestion to historians of what might be found by close attention to the artifacts of computing in genera1.[25] Operating systems surely offer another example, as do early database management systems (about which we know so little).

Programs are artifacts, and we must learn to read them as such. What makes it difficult is precisely that the representations are operative. Ultimately, it is their behavior rather than their structure (or the fit between structure and behavior) that interests us. We do not use computers by reading programs; we interact with programs running on computers. The primary source for the historian of software is the dynamic process, and, where it is still available, it requires special techniques of analysis. It is a dynamic artifact, a running process, expressed in a static artifact, a program, and the relation between the two is indeterminate. Christopher Langton describes the problem of their relationship:

> We need to separate the notion of a formal specification of a machine—that is, a specification of the logical structure of the machine—from the notion of a formal specification of a machine's behavior—that is, a specification of the sequence of transitions that the machine will undergo. In general, we cannot derive behaviours from structure, nor can we derive structure from behaviours.[26]

As historical artifact, software is most valuable in its dynamic form. The historian gains most from seeing how the software worked and from working with it. But, especially for early software, the dynamic artifacts have in large part already been lost to us. Even if we had the programs—and in many cases we do not—we lack the platforms on which they ran; we have neither the systems software nor the hardware for which it was written. We cannot experience the software as users experienced it and hence analyze that experience critically. Even where we have emulators, they fall short in essential ways. Although they convey some sense of programming by executing the code, what appears on the screen is merely a simulation of the physical device, omitting entirely the spatial, temporal, and social experience of writing programs and having them punched up and run, or of interacting with the system as a user.[27]

Hence, we are left with the static artifact, the program text, which may or may not correspond to the running version, as people discovered in many instances during the Y2K effort. We have some promising hints of how we might go about reading programs; Chapter 1 of Gerald Weinberg's *The Psychology of Computer Programming* and Brian Kernighan's and P.J. Plauger's *Elements of Programming Style* come first to mind.[28] Alan Kay offered a striking example in describing his first encounter with object-oriented programming. It was his first day as a graduate student at the University of Utah:

> Head whirling, I found my desk. On it was a pile of tapes and listings, and a note: "This is the ALGOL for the 1108. It doesn't work. Please make it work." The latest graduate student gets the latest dirty task.
>
> The documentation was incomprehensible. Supposedly, this was the Case-Western Reserve 1107 Algol—but it had been doctored to make a language called Simula; the documentation read like Norwegian transliterated into English, which in fact was what it was. There were uses of words like activity and process that didn't seem to coincide with normal English usage.
>
> Finally, another graduate student and I unrolled the listing 80 feet down the hall and crawled over it yelling discoveries to each other. The weirdest part was the storage allocator, which did not obey a stack discipline as was usual for Algol. A few days later, that provided the clue. What Simula was allocating were structures very much like instances of Sketchpad.[29]

Kay's experience suggests what faces the historian, and there is again some irony in it. It has been the common lament of management that programs are built by tinkering and that little of their design gets captured in written form, at least in a written form that would make it possible to determine how they work or why they work as they do rather than in other readily imaginable ways. Moreover, what programs do and what the documentation says they do are not always the same thing. Here, in a very real sense, the historian inherits the problems of software maintenance: the farther the program lies from its creators, the more difficult it is to discern its architecture and the design decisions that inform it.

But at least we have the program texts. Or do we? It is not clear, in part perhaps because for many of the communities of computing outside computer science we have only begun to look. When government and industry went looking in the late 1990s in anticipation of Y2K, the results were dismaying. Almost as dismaying was learning from contributors to the most recent History of Programming Languages Conference (HOPL III) that documentation for languages created in the 1980s and 1990s is incomplete.

Legacy software as history

What makes the history of software hard, then, is to a large extent what makes software engineering hard. In Brooks's terms, it is only accidentally about computers. Just as the design of software begins with an analysis of the activity to be automated, so too its history begins with the history of the activity to understand how its practice was translated into a computational model. The history does not end with the translation, but rather is embodied in the model and hence in the program that implements it. That is what gives "legacy software" a dual meaning. Legacy software is not just old code, but rather a continuing enactment, an operative representation, of the domain knowledge and practice embodied in it. That may explain the difficulties software engineers have experienced in upgrading and replacing older systems.

Early in 2007, a senior member of the programming language community sent me email expressing his concerns about the languages covered by the Third History of Programming Languages Conference (HOPL III) to take place in June.[30] "When I compare the three conferences," he wrote, "I see a movement from languages that received widespread use to languages used mostly within the research community." In addition to explaining why several languages one might have expected to see were not represented (the committee could not enlist a suitable author), I pointed out that the languages present could be taken as a sign of maturity of the field. The first two HOPLs covered what might be called the "heroic age" of programming languages, when individuals or small teams could start more or less from scratch and design a language to express their vision of computation and computing.

The languages of HOPL III follow a different pattern. They were developed in the context of a mature field and under a variety of technical and corporate constraints. With the exception perhaps of Statecharts, they establish no new paradigms but pursue the implications of existing ones. Haskell and Self take functional programming to its limits, Emerald grapples with the challenges of distributing objects over a network, Beta builds on Simula, C++ settles into a standard version, still incorporating C as a sublanguage. It's not clear that the missing languages, such as Java or Python, would change the situation, despite their widespread use. All of them have had to fit into existing systems, meet corporate demands, preserve compatibility with earlier versions, and so on.

Indeed, in retrospect, that was already the case with several of the languages at HOPL II. As Bjarne Stroustrup observed in explaining why C++ was based on C, "We never have a clean slate. Whatever new we do must make it possible for people to make a transition from old tools and ideas to new."[31] Any such transition involves an understanding of the old tools, of how they work, and of how people use them. It means taking the notion of legacy seriously, as something we continue to live with.

Let me close with another recent example. Toward the end of March 2007, H. Travis Christ, VP for Sales and Marketing for US Airways, wrote to the airline's customers to inform them about the progress of the merger of that company with America West, which required the merging of two reservations systems, undertaken by migrating the data of one of them to the other. It was an 18-month effort that constituted, in his words, "one of the biggest IT projects in aviation history."[32] On the one hand, the transfer had gone reasonably well. On the other hand, it had not, and the failures of the system's automated check-in kiosks at several major hubs coincided with a crippling ice storm on the East Coast to cause severe unhappiness among customers. "For those of you who like the details," he explained,

> When we transferred the seven million reservations from one system to the other, approximately 1.5 million of them didn't "sync up" correctly and our agents had to hand-process each reservation. Many systems that were otherwise ready to go became bogged down with these reservations. We've since whittled the number of "out of sync" reservations to a very small number.[32]

Christ anticipated the next question—or perhaps he had heard it in some version: "This all sounds very clunky. Why didn't you convert to a more modern system?" His answer revealed much about the current state of software engineering:

> Ugh. How much time do you have? The short version is this:
>
> Most airlines were built on "legacy" mainframe systems from the 60's and 70's. These systems are deeply embedded in everything from reservations, to flight operations, to airport operations, to accounting. They are very reliable, but very inflexible. As our business changes, it's as though we're fighting with one hand tied behind our back.
>
> You might respond: "So dummy, convert it to a 21st century system." We would like to do that and eventually will. Several technology companies are building more modern platforms, and we are in contact with them. In an industry where we lose money more often than we turn a profit, it's not easy to justify replacing a system that works with one that's very expensive, untried and carries additional risk. But stay tuned; we'll get there.[32]

In sum, the airlines are working with legacy systems built several decades ago that do the job well at the scale for which they were designed but are not adaptable to growth, change, and restructuring. That is as true of air traffic control as of reservations.[33] New systems are expensive and so far rest on promise rather than proven performance. Rather than facilitating flexible response to varying conditions in a crucial, yet fragile, sector of the American economy, current software technology adds to the burden and distracts from the task at hand.

Now that situation should be of common interest to computer people and to historians. Historians will want to know how it developed over several decades and why software systems have not kept pace with advances in hardware. That is, historians are interested in the legacy. Even as computer scientists wrestle with a solution to the problem the legacy poses, they must live with it. It is part of their history, and the better they understand it, the better they will be able to move on from it.

References and notes

1. R. Kling and W. Scacchi, "The Web of Computing," *Advances in Computers*, vol. 21, 1982, pp. 1-90; at p. 39 and pp. 56-60.
2. B. Stroustrup, "A History of C++," in T.M. Bergin, and R.G. Gibson, eds., *History of Programming Languages*, ACM Press, 1996, p. 750.
3. Another version of this article addressed to the implications of the subject for the humanities appeared as "The Histories of Computing(s)," *Interdisciplinary Science Reviews*, vol. 30, no. 2, 2005, pp. 119-135.

4. In England, of course, the story has a slightly different trajectory that includes Turing's ACE. Although pursuing his own ideas, Turing knew about the EDVAC report and took it into account. See *Alan Turing's Automatic Computer Engine: The Master Codebreaker's Struggle to Build the Modern Computer*, B. Jack Copeland, ed., Oxford Univ. Press, 2005. Germany offers yet another variant trajectory, originating with Konrad Zuse's machines.
5. On the first point, see Chaps. 3 and 4 of C.J. Bashe, et al., *IBM's Early Computers*, MIT Press, 1986; on the second, see the recent dissertation by J.A. November, ''Digitizing Life: The Introduction of Computers to Biology and Medicine,'' PhD dissertation, Princeton Univ., 2006.
6. For a subtle discussion of memory and history—personal, social, and scientific—see G.C. Bowker, *Memory Practices in the Sciences*, MIT Press, 2005.
7. For a discussion of the historical models of engineering to which software engineers have turned, see my ''Finding a History for Software Engineering,'' *IEEE Annals of the History of Computing*, vol. 26, no. 1, 2004, pp. 8-19; online version at http://www.princeton.edu/~mike/articles/finding/finding.html.
8. In *Calculating a Natural World: Scientists, Engineers, and Computers during the Rise of U.S. Cold War Research* (MIT Press, 2007), Atsushi Akera articulates the complex and evolving ''ecology of knowledge'' among institutions, occupations, organizations, knowledge, and artifacts and actors that constituted that community.
9. See, for example, J.W. Cortada, *Information Technology as Business History: Issues in the History and Management of Computers*, Greenwood Press, 1996, and *The Digital Hand: How Computers Changed the Work of American Manufacturing, Transportation, and Retail Industries*, Oxford Univ. Press, 2004; Thomas Haigh, ''The Chromium-Plated Tabulator: Institutionalizing an Electronic Revolution, 1954–1958,'' *IEEE Annals of the History of Computing*, vol. 23, no. 4, 2001, pp. 75-104, and ''Inventing Information Systems: The Systems Men and the Computer, 1950–1968,'' *The Business History Rev.*, vol. 75, no. 1, 2001, pp. 15-61.
10. J. Agar, *The Government Machine: A Revolutionary History of the Computer*, MIT Press, 2003.
11. I am telling an American story here. For the wider context, see inter alia J.A. Merkle, *Management and Ideology: The Legacy of the International Scientific Management Movement*, Univ. of California Press, Berkeley, 1980.
12. The seminal account is T.P. Hughes, *Networks of Power: Electrification in Western Society, 1880–1930*, Johns Hopkins Univ. Press, 1983.
13. D. Mindell, *Between Human and Machines: Feedback, Control, and Computing before Cybernetics*, Johns Hopkins Univ. Press, 2002.
14. For an overview, see M.S. Mahoney, ''The Structures of Computation,'' *The First Computers: History and Architectures*, Raul Rojas and Ulf Hashagen, eds., MIT Press, 2000, and ''Software: The Self-Programming Machine,'' *From 0 to 1: An Authoritative History of Modern Computing*, A. Akera and F. Nebeker, eds., Oxford Univ. Press, 2002.
15. For a detailed study of that process in the insurance industry, see J. Yates, *Structuring the Information Age: Life Insurance and Technology in the Twentieth Century*, Johns Hopkins Univ. Press, 2005.
16. U. Hashagen, R. Keil-Slawik, and A. Norberg, eds., *History of Computing: Software Issues*, Springer Verlag, 2002. In my own presentation (''Software as Science—Science as Software'', pp. 25-48) and the ensuing discussion, I missed an important question underlying critiques that I was ignoring the sciences that had contributed to computer science. I saw them—for example, psychology—as ancillary, as resources on which computer science has drawn, but not as constituents of computer science itself. In looking at them in that way, I missed seeing how their application to problems of computation brought out what was computational about them. If computer science is concerned with what can be computed, then it includes the computational aspects of the sciences, and its history includes the process by which those computational aspects were identified, elicited, and articulated.
17. The Forum was announced by Adele Goldberg in a President's Letter in the *Comm. ACM*, vol. 28, no. 2, 1985, pp. 131-133. The enabling resolution of the ACM Council in October 1984 opened with the statement that ''Contrary to the myth that computer systems are infallible, in fact computer systems can and do fail. Consequently the reliability of computer-based systems cannot be taken for granted.'' (Ibid., p. 131). For a compilation and commentary on the mishaps reported over the first ten years, see P.G. Neumann's *Computer-Related Risks*, ACM Press, 1995.
18. J.E. Tomayko, ''Software as Engineering,'' in *History of Computing*, Hashagen et al., eds., pp. 65-76. H. Petroski, *To Engineer is Human: The Role of Failure in Successful Design*, St. Martin's Press, 1985.
19. P. Naur and B. Randell, eds., *Software Engineering, Report on a Conference Sponsored by the NATO Science Committee*, Scientific Affairs Division, NATO, 1968, p. 13. The report was republished, together with the report on the second conference the following year, in *Software*

Engineering: Concepts and Techniques: Proc. NATO Conferences, P. Naur, B. Randell, and J.N. Buxton, eds., Petrocelli, 1976. Randell has made both reports available for download; see http://homepages.cs.ncl.ac.uk/brian.randell/NATO/.

20. F.P. Brooks, ''No Silver Bullet—Essence and Accidents of Software Engineering,'' *Information Processing 1986*, H.J. Kugler, ed., Elsevier Science, 1986, pp. 1069-1076; reprinted in *Computer*, vol. 20, no. 4, 1987, pp. 10-19; and in the Anniversary Edition of *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1995, Chap. 16. Chap. 17, '''No Silver Bullet' Refired'' is a response to critics of the original article and a review of the silver bullets that have missed the mark over the intervening decade.
21. Symbolic algebra constitutes another example of operative representation; see M.S. Mahoney, ''The Beginnings of Algebraic Thought in the Seventeenth Century,'' *Descartes: Philosophy, Mathematics and Physics*, S. Gaukroger et al., eds., The Harvester Press, 1980, Chap.5.
22. See the project site at http://www.nsf.gov/funding/pgm_summ.jsp?pims_id=12766&org=CCF.
23. H. Ford, *My Life and Work*, Doubleday, 1922, pp. 23-24. For a reading of Ford's Model T, see my web document, ''Reading a Machine''; http://www.princeton.edu/~hos/h398/readmach/modelt.html.
24. T. Kidder, *The Soul of a New Machine*, Little, Brown, 1981, p. 26.
25. As originally stated, ''[O]rganizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations.'' M.E. Conway, ''How Do Committees Invent?'' *Datamation*, vol. 14, no. 4, 1968, pp. 28-31; online version at www.melconway.com/research/committees.html. For Conway's subsequent thoughts on the subject, see his webpage, www.melconway.com/law.
26. C.G. Langton, ''Artificial Life'' [1989], *The Philosophy of Artificial Life*, M.A. Boden, ed., Oxford Univ. Press, 1996, p. 47. On the philosophical implications of the distinction between program and process (and, third, semantics), see B. Cantwell Smith, *On the Origin of Objects*, MIT Press, 1998, pp. 32-42.
27. I thank an anonymous reviewer for directing my attention to this point.
28. G.M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, 1971; B. Kernighan and P.J. Plauger, *Elements of Programming Style*, McGraw-Hill, 1978.
29. A.C. Kay, ''The Early History of Smalltalk,'' *History of Programming Languages—II*, T.J. Bergin and R.G. Gibson, eds., ACM Press, 1996, p. 516.
30. The languages covered were AppleScript, BETA, C++, Emerald, Erlang, Haskell, High Performance Fortran (HPF), Lua, Modula-2/Oberon, Self, Statecharts, and ZPL. See http://research.ihost.com/hopl/HOPL-III.html. Among those conspicuously missing were Java, ML, Perl, and Python.
31. Indeed, it is striking how many of the languages of HOPL III were initially implemented in C or C++.
32. Email to author as customer from US Airways—Dividend Miles (dividendmiles@myusairways.com), 24 March 2007, Subject: Merger Update. A copy of the letter with slight different wording may be found at http://blog.fastcompany.com/archives/2007/03/14/usair_asks_fliers_can_we_get_a_hallelujah.html.
33. See, inter alia, R.N. Britcher, *The Limits of Software: People, Projects, and Perspectives*, Addison-Wesley, 1999.

Michael S. Mahoney is a professor of history in the Program in History of Science at Princeton University, where he earned a PhD in 1967. As historian of science, he has written on the development of the mathematical sciences from antiquity to 1700 and, as historian of computing, he has explored the emergence of the new technical disciplines of theoretical computer science and software engineering.

Readers may contact Mahoney at mike@princeton.edu.