The Computer Boys Take Over

Computers, Programmers, and the Politics of Technical Expertise

Nathan Ensmenger

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Sabon by Toppan Best-set Premedia Limited. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Ensmenger, Nathan, 1972–
The computer boys take over : computers, programmers, and the politics of technical expertise / Nathan Ensmenger.
p. cm.—(History of computing)
Includes bibliographical references and index.
ISBN 978-0-262-05093-7 (hardcover : alk. paper)
1. Computer programming. 2. Computer programmers. 3. Software engineering—History. 4. Computer software—Development—Social aspects. I. Title.
QA76.6.E58 2010
005.1—dc22

2009052638

10 9 8 7 6 5 4 3 2 1

2
The Black Art of Programming

When a programmer is good, he is very, very good. But when he is bad, he is horrid.
—IBM study on programmer performance, 1968

An Unexpected Revolution

One of the great myths of the computer revolution is that nobody saw it coming—particularly not the so-called computer experts. In one widely repeated but apocryphal anecdote, Thomas Watson, the legendary founder and longtime chair of the IBM Corporation, is said to have predicted as late as 1943 a total world market for "maybe five computers." The story of this wildly inaccurate forecast, alternatively attributed to Watson, the Harvard professor and computing pioneer Howard Aiken, or the Cambridge professor of computer science Douglas Hartree, among others, is generally mobilized as a kind of modern morality play, a cautionary tale about the dangers of underestimating the power and rapidity of technological progress.[1] Similar tales (similarly apocryphal) are told about a series of unimaginative computer industry executives—from Digital Equipment Corporation's Ken Olsen to Microsoft's Bill Gates—whose alleged lack of imagination prevented them from fully appreciating the transformative potential of computer technology. Such stories are a staple of popular histories of the electronic computer, which generally privilege dramatic change—sudden, unanticipated, and inexorable—over continuity.

In reality, many of the predictions made by contemporaries about the revolutionary potential of the electronic computer were, if anything, wildly optimistic. Almost before there were any computers—functional, modern, electronic digital stored-program computers—enthusiasts for the new technology were confidently anticipating its influence on contemporary society. As early as 1948 the cybernetician Norbert Wiener was predicting a "second industrial revolution" enabled by the electronic computer.[2] A year later, the computer consultant Edmund Berkeley, in his popular book *Giant Brains; or, Machines That Think*, described a near future in which computers radically transform a broad range of human cognitive and occupational activities, including business, law, education, and medicine.[3] Despite the fact that electronic computers were in this period little more than glorified calculating machines, the provocative image of the computer as a "giant" or "mechanical brain" quickly became established in the popular imagination. Within just a few years of the introduction of the first commercial electronic computers, even mass-market publications like *Time* and *Newsweek* were predicting the use of computers in wide variety of commercial and scientific applications. Indeed, as Stephen Schnaars and Sergio Carvalho have recently suggested, far from underestimating its potential, during the 1950s the press in the United States "fell in love" with computer technology.[4]

In the business literature in particular, the coming computer revolution was declared boldly, widely, and repeatedly.[5] The expectation was that electronic computers would soon become an integral part of the already large and thriving business machines industry. As *Fortune* magazine confidently predicted in a 1952 survey of the computer industry, "office robots" were poised to "eliminate the human element" in many clerical operations, enabling massive gains in productivity.[6] While these wild predictions might have been unsettling to U.S. office workers, they did suggest a rapidly growing market for computer technology. At the very least, the computer manufacturers were convinced that computers were the wave of the future; in the early 1950s, dozens of firms—among them such major players as IBM, GE, Burroughs, RCA, and NCR—invested heavily in this potential new growth market.

And grow the market did. In 1950 there were only 2 electronic computers in use in the United States. By 1955 there were 240. Five years later, there were 5,400. By 1965, the grand total had grown to almost 25,000, and by 1970, 75,000.[7] By the end of the 1960s, electronic computers and their associated peripherals formed the basis of a $20 billion industry—an industry growing at an average rate of more than 27 percent annually. Within two decades of the development of the first electronic digital computer, the computer industry in the United States had emerged from nothing to become one of the largest and fastest-growing sectors of the U.S. economy—a position that it would hold for the next several decades.[8]

Coevolving with this flourishing new information industry was a novel species of technical professional: the computer programmer. In 1945 there were no computer programmers, professional or otherwise; by 1967 industry observers were warning that although there were at least a hundred thousand programmers working in the United States, there was an immediate need for at least fifty thousand more.[9] "Competition for programmers," declared a contemporary article in *Fortune* magazine, "has driven salaries up so fast that programming has become probably the country's highest paid technological occupation. . . . Even so, some companies can't find experienced programmers at any price."[10]

Of all the unanticipated consequences of the invention of the electronic computer in the mid-1940s, the most surprising was the sudden rise to prominence of the computer programmer. While the computer revolution itself might not have been unforeseen, the role of the computer programmer in bringing about that revolution certainly was. In all of the pioneering computer projects of this period, for example, programming was considered, at best, an afterthought. It was generally assumed that coding the computer would be a relatively simple process of translation that could be assigned to low-level clerical personnel. It quickly became apparent that computer programming, as it came to be known, was anything but straightforward and simple. Skilled programmers developed a reputation for creativity and ingenuity, and programming was considered by many to be a uniquely intellectual activity, a black art that relied on individual ability and idiosyncratic style. By the beginning of the 1950s, however, programming had been identified as a key component of any successful computer installation. By the early 1960s, the "problem of programming" had eclipsed all other aspects of commercial computer development. As the electronic computer increasingly moved out of the laboratory and into the marketplace, the centrality of programming—and programmers—became even more apparent.

Originally envisioned as little more than glorified clerical workers, programmers quickly assumed a position of power within many organizations that was vastly disproportionate to their official position in the organizational hierarchy. Defined by their mastery of the highest of high technology, they were often derided for their adherence to artisanal practices. Although associated with the emerging academic discipline of computer science, they were never widely considered to be either scientists or engineers. Neither laborers nor professionals, they defy traditional occupational categorizations. The ranks of the elite programmers included both high school dropouts and ex-PhD physicists. Even to this day, their occupational expertise remains difficult to clearly define or delineate. For example, the term programmer, which was widely used as a generic catchall description of a computing specialist in the 1960s, encompasses such a wide range of occupational categories—from the narrow and highly technical coder to the elite and influential "systems man"—that it is more useful as a rhetorical device than as an analytic category.

The questions of what programming was—as an intellectual and occupational activity—and where it fit into traditional social, academic, and professional hierarchies, were actively negotiated during the decades of the 1950s and 1960s. Programmers were well aware of their tenuous professional position, and they struggled to prove that they possessed a unique set of skills and training that allowed them to lay claim to professional autonomy. This chapter traces the history of computer programming from its origins as low-status clerical work (often performed by women) into one of the highest-paid technical occupations of the late 1950s and early 1960s. The focus is on the emergence of the computer programmer as a highly valued, well-compensated, and largely autonomous technical expert.

The Origins of Computer Programming

In the eyes of a computer scientist, all computers are created equal. That is to say, any true computing machine can, by definition, compute anything that is computable. Or to state the case a little more clearly, any device worthy of the name computer can be programmed to perform any task that can be performed by any other computer. This means that in theory at least, all computers are functionally equivalent: any given computer is but a specific implementation of a more general abstraction known as a Universal Turing Machine.

It is the Platonic ideal of the Universal Turing Machine, and not the messy reality of actual physical computers, that is the true subject of modern theoretical computer science; it is only by treating the computer as an abstraction, a mathematical construct, that theoretical computer scientists lay claim to their field being a legitimate *scientific*, rather than merely a technical or engineering, discipline. The story of this remarkable self-construction and its consequences is the subject of chapter 5.

The idealized Universal Turing Machine is, of course, only a conceptual device, a convenient fiction concocted by the mathematician Alan

Turing in the late 1930s as a means of exploring a long-standing puzzle in theoretical mathematics known as the *Entscheidungsproblem*. In order to facilitate his exploration, Turing invented a new tool, an imaginary device capable of performing simple mechanical computations. Each Turing Machine, which consisted of only a long paper tape along with a mechanism for reading from and writing to that tape, contained a table of instructions that allowed it to perform a single computation. As a computing device, the Turing Machine is deceptively simple; as a conceptual abstraction, it is extraordinarily powerful. As it turns out, the table of instructions for any Turing Machine can be rewritten to contain the instructions for building any other Turing Machine. The implication, as articulated in the Church-Turing thesis, is that every Turing Machine is a Universal Turing Machine, and by extension, every computing machine is essentially equivalent.

In the real world, the appealingly egalitarian abstractions of the Church-Turing thesis quickly break down in the face of the temporal and spatial constraints of the physical universe. To implement even the simplest computations on an archetypal paper tape–based Turing Machine, for example, would require an enormous and prohibitive amount of resources. In fact, the figures involved quickly become absurdly Saganesque: the number of miles of paper tape required would be more than the total number of atoms in the universe, and the amount of time required would be more than all of known cosmological history. To the emerging discipline of theoretical computer scientists, perhaps, none of these practical realities were particularly significant. But to working computer engineers and programmers, such constraints were a daily reality, even in the era of electronic computing. Extracting acceptable performance and reliability out of the early electronic computers required an enormous degree of messy tinkering, local knowledge, and idiosyncratic technique. The developing tension between the messy tinkering of real-world computing and the clean abstractions of academically minded computer scientists would come to define one of the sharp divides within the ranks of the larger computing community. The struggle between theory and practice would become a major challenge for academics and practitioners alike, and would reflect itself in the structure of programming languages, professional societies, and academic curricula.

Conventional histories of computer programming tend to conflate programming as a vocational activity with computer science as an academic discipline. In many of these accounts, programming is represented as a subdiscipline of formal logic and mathematics, and its origins are identified in the writings of early computer theorists Alan Turing and John von Neumann. The development of the discipline is evaluated in terms of advances in programming languages, formal methods, and generally applicable theoretical research. This purely intellectual approach to the history of programming, however, conceals the essentially craftlike nature of early programming practice. The first computer programmers were not scientists or mathematicians; they were low-status, female clerical workers and desktop calculator operators. The origins of programming as a profession lie in the commercial traditions of machine-assisted, manual computation, not in the mainstream of theoretical mathematics.

The history of vocational computer programming begins, in the United States at least, with the construction of the ENIAC in summer 1945. Many historians have identified the ENIAC as the first true electronic computer. The question of "which was the first computer" is surprisingly difficult to answer. As Michael Williams suggests in a recent volume edited by Raul Rojas and Ulf Hashagen called *The First Computers* (note the crucial use of the plural), any particular claim to the priority of invention must necessarily be heavily qualified: if you add enough adjectives you can always claim your own favorite.[11] And indeed, the ENIAC has a strong claim to this title: not only was it digital, electronic, and programmable (and therefore looked a lot like a modern computer) but the ENIAC designers—John Mauchly and J. Presper Eckert—went on to form the first commercial computer company in the United States. The ENIAC and its commercial successor, the UNIVersal Automatic Computer (UNIVAC), were widely publicized as the first of the "giant brains" that presaged the coming computer age. But even the ENIAC had its precursors and competitors. For example, in the 1930s, a physicist at Iowa State University, John Atanasoff, had worked on an electronic computing device and had even described it to Mauchly. Others were working on similar devices. During the Second World War in particular, a number of government and military agencies, both in the United States and abroad, had developed electronic computing devices, many of which also have a plausible claim to being if not *the* first computer, then at least *a* first computer.

There are two major innovations in computing that the ENIAC embodied. The first was that it was electronic. Earlier computing devices, including tabulating machines, were either mechanical or electromechanical, meaning that they contained numerous moving parts. These moving parts were complicated to manufacture, difficult to maintain, and above all relatively slow. By replacing them with completely electronic components, Eckert and Mauchly were able to dramatically speed up the process of computation. Whereas the electromechanical Harvard Mark I (completed in 1943), which was of similar complexity to the ENIAC, could perform 2 or 3 additions per second, and a multiplication every six seconds, the ENIAC (completed just three years later) could perform 5,000 additions per second, or 333 multiplications. Although this extraordinary improvement in performance came at the price of increased cost and complexity—when completed the ENIAC weighed nearly thirty tons, occupied an entire room, and required more than eighteen thousand expensive and unreliable vacuum tubes—by the end of the 1940s it was clear that electronic computing was the wave of the future.

The second revolutionary feature of the ENIAC was its ability to be programmed. This meant that the machine could be reconfigured to perform different types of computation. In the case of the ENIAC the machine had to be physically wired, or "set up," as the process was called at the time, to compute specific functions—a complicated process that could take as long as two days.[12] Within a short time, however, the ENIAC was modified to allow it to be "programmed" automatically using punch cards.[13] In the meantime, the physicist and mathematician von Neumann had published his now-infamous *First Draft of a Report on the EDVAC*, which provided a description of the computer that was to be heir to the ENIAC.[14] This successor machine, which was called the Electronic Discrete Variable Automatic Computer (EDVAC), was the world's first stored-program computer. Unlike previous programmable machines, the EDVAC stored-program computer did not distinguish between data and instructions. This allowed it to modify its own instructions, which effectively allowed the computer to program itself. This not only allowed for greater flexibility in programming but also paved the way for the development of assemblers, compilers, and other programming tools. The concept of the stored-program computer was so significant that it has come to define the essence of the modern computer; today a device is only considered to be a true computer if it is a stored-program machine.

And this is what brings us back to the centrality of software to the history of computing: it was not so much the original invention of the electronic computer that launched the computer revolution but the later discovery that such computers could be made programmable. To be sure, prior to the electronic computer there were machines that could be controlled automatically. A Jacquard loom, for instance, used a series of steel cards, as many as twenty thousand at a time, to control the weaving of patterns on fabric.[15] Tabulating machines could also be programmed to a certain degree by rewiring their components. But the combination of speed and flexibility provided by the combination of an electronic digital computer and well-designed software was unprecedented. The electronic digital computer would eventually become a universal machine whose potential applications were limited only by the imagination of its programmers.

Therein lies the rub: the very aspect of electronic computing that made it so powerful and appealing was the aspect of least interest to its original designers. Computer programming began as little more than an afterthought in most of the pioneering wartime electronic computing projects, an offhand postscript to what was universally regarded as the much more pressing challenge of hardware development.

There were certainly legitimate reasons for privileging hardware over software; simply managing to keep the early electronic computers running without failure for more than a few minutes was an engineering challenge of heroic proportions. As was mentioned earlier, the core computational units of the ENIAC machine relied on more than eighteen thousand vacuum tubes, each of which had an average lifespan of just three thousand hours. This meant that statistically speaking, six of these tubes would fail every hour; or in other words, at least one tube failed every ten minutes. Figuring out how to control the rate of failure of vacuum tubes was one of the great contributions of the ENIAC's brilliant chief engineer, J. Presper Eckert. Similarly, the construction of mercury delay lines, which were an early form of short-term memory used in the Cambridge University EDSAC, the world's first working stored-program computer, required the precise coordination of acoustical waves moving at 1,450 meters per second. There is no question that overcoming the engineering challenges posed by the electronics of electronic computing was essential to the further development of computer technology.

But solving the programming hurdles was equally vital. Although in the decades after the ENIAC we have come to regard the electronic computer as an almost infinitely protean and useful machine, this is largely a reflection of the successes of software. In the immediate postwar period even programmable computers like the ENIAC were considered impressive but limited. It was not hard to imagine that the military and the government might have a need for a small number of such devices, yet few would have predicted how rapidly the commercial market for computers would expand over the course of the next decade.

"Glorified Clerical Workers"

The low priority given to programming was reflected in who was assigned to the task. Although the ENIAC was developed by academic researchers at the University of Pennsylvania's Moore School of Electrical Engineering, it was commissioned and funded by the Ballistics Research Laboratory (BRL) of the U.S. Army. Located at the nearby Aberdeen Proving Grounds, the BRL was responsible for the development of the complex firing tables required to accurately target long-range ballistic weaponry. Hundreds of these tables were required to account for the influence of highly variable atmospheric conditions (air density, temperature, etc.) on the trajectory of shells and bombs. Prior to the arrival of electronic computers, these tables were calculated and compiled by teams of human "computers" working eight-hour shifts, six days a week. From 1943 onward, essentially all of these computers were women, as were their immediate supervisors. The more senior women (those with college-level mathematical training) were responsible for developing the elaborate "plans of computation" that were carried out by their fellow computers.

In June 1945, six of the best human computers at Aberdeen were hired by the leaders of the top secret "Project X"—the U.S. Army's code name for the ENIAC project—to set up the ENIAC machine to produce ballistics tables. Their names were Kathleen McNulty, Frances Bilas, Betty Jean Jennings, Elizabeth Snyder Holberton, Ruth Lichterman, and Marlyn Wescoff. Collectively they were known as "the ENIAC girls."[16] Today the ENIAC girls are often considered the first computer programmers. In the 1940s, they were simply called coders.

The use of the word coder in this context is significant. At this point in time the concept of a program, or of a programmer, had not yet been introduced into computing. Since electronic computing was then envisioned by the ENIAC developers as "nothing more than an automated form of hand computation," it seemed natural to assume that the primary role of the women of the ENIAC would be to develop the plans of computation that the electronic version of the human computer would follow.[17] In other words, they would code into machine language the higher-level mathematics developed by male scientists and engineers. Coding implied manual labor, and mechanical translation or rote transcription; coders were obviously low on the intellectual and professional status hierarchy. It was not until later that the now-commonplace title of programmer was widely adopted. The verb "to program," with its military connotations of "to assemble" or "to organize," suggested a more thoughtful and system-oriented activity.[18] Although by the mid-1950s the word programmer had become the preferred designation, for the next several decades programmers would struggle to distance themselves from the status (and gender) connotations suggested by coder.

The first clear articulation of what a programmer was and should be was provided in the late 1940s by Goldstine and von Neumann in a series of volumes titled *Planning and Coding of Problems for an Electronic Computing Instrument.* These volumes, which served as the principal (and perhaps only) textbooks available on the programming process at least until the early 1950s, outlined a clear division of labor in the programming process that seems to have been based on the practices used in programming the ENIAC. Goldstine and von Neumann spelled out a six-step programming process: (1) conceptualize the problem mathematically and physically, (2) select a numerical algorithm, (3) do a numerical analysis to determine the precision requirements and evaluate potential problems with approximation errors, (4) determine scale factors so that the mathematical expressions stay within the fixed range of the computer throughout the computation, (5) do the dynamic analysis to understand how the machine will execute jumps and substitutions during the course of a computation, and (6) do the static coding. The first five of these tasks were to be done by the "planner," who was typically the scientific user and overwhelmingly was frequently male; the sixth task was to be carried out by coders. Coding was regarded as a "static" process by Goldstine and von Neumann—one that involved writing out the steps of a computation in a form that could be read by the machine, such as punching cards, or in the case of the ENIAC, plugging in cables and setting up switches. Thus, there was a division of labor envisioned that gave the highest-skilled work to the high-status male scientists and the lowest-skilled work to the low-status female coders.

As the ENIAC managers and coders soon realized, however, controlling the operation of an automatic computer was nothing like the process of hand computation, and the Moore School women were therefore responsible for defining the first state-of-the-art methods of programming practice. Programming was an imperfectly understood activity in these early days, and much more of the work devolved on the coders than anticipated. To complete their coding, the coders would often have to revisit the underlying numerical analysis, and with their growing skills, some scientific users left many or all six of the programming stages to the coders. In order to debug their programs and distinguish hardware glitches from software errors, they developed an intimate knowledge of the ENIAC machinery. "Since we knew both the application and the machine," claimed ENIAC programmer Betty Jean Jennings, "as a result we could diagnose troubles almost down to the individual vacuum tube. Since we knew both the application and the machine, we learned to diagnose troubles as well as, if not better than, the engineers."[19] In a few cases, the local craft knowledge that these female programmers accumulated significantly affected the design of the ENIAC and subsequent computers. ENIAC programmer Betty Holberton recalled one particularly significant episode:

> In the fall of '46 when the new idea of wiring up the ENIAC with sort of semi-permanent wiring with instruction codes [emerged] . . . a number of us worked with Dr. von Neumann in setting up this code. . . . We felt we wouldn't need that many settings for all of the instructions. We sort of worked along for a while. But to my astonishment, he never mentioned a stop instruction. So I did coyly say, "Don't we need a stop instruction in this machine?" He said, "No we don't need a stop instruction. We have all these empty sockets here that just let it go to bed." And I went back home and I was really alarmed. After all, we had debugged the machine day and night for months just trying to get jobs on it.
>
> So the next week when I came up with some alterations in the code, I approached him again with the same question. He gave me the same answer. Well I really got red in the face. I was so excited and I really wanted to tell him off. And I said, "But Dr. von Neumann, we are programmers and we sometimes make mistakes." He nodded his head and the stop order went in.[20]

The deference with which Holberton proposes her tentative suggestion and von Neumann's initial patronizing dismissal are indicative of the status of the programmers relative to that of their scientific and engineering colleagues. Von Neumann's eventual acceptance reflects his recognition of the importance of local craft knowledge and an increasing acceptance of the value of programming expertise. Given that the programmers "were often able to point out to a technician which individual vacuum tube needed to be changed," they were able to interact much more with the computer engineers and technicians than was probably originally intended. This had the positive effect of convincing the ENIAC managers that programmers were essential to the success of the overall project and that well-informed, technically proficient, high-quality programmers were especially indispensable.

Thus, what was supposed to have been a low-level skill, a static activity, prepared these women coders well for careers as programmers, and indeed, those who did pursue professional careers in computing often became programmers and thrived at it. A few women, Grace Hopper and Betty Holberton of UNIVAC as well as Ida Rhodes and Gertrude Blanche of the National Bureau of Standards in particular, continued to serve as leaders in the programming profession. But despite the success of the ENIAC women in establishing a unique occupational niche for the programmer within the ENIAC community, programming continued to be perceived as marginal to the central business of computer development. By nature of their gender (female) and education (nonscientific and nonengineering), the early programmers remained isolated from their engineering and scientific managers. If software was admitted to be important, hardware was considered to be essential.

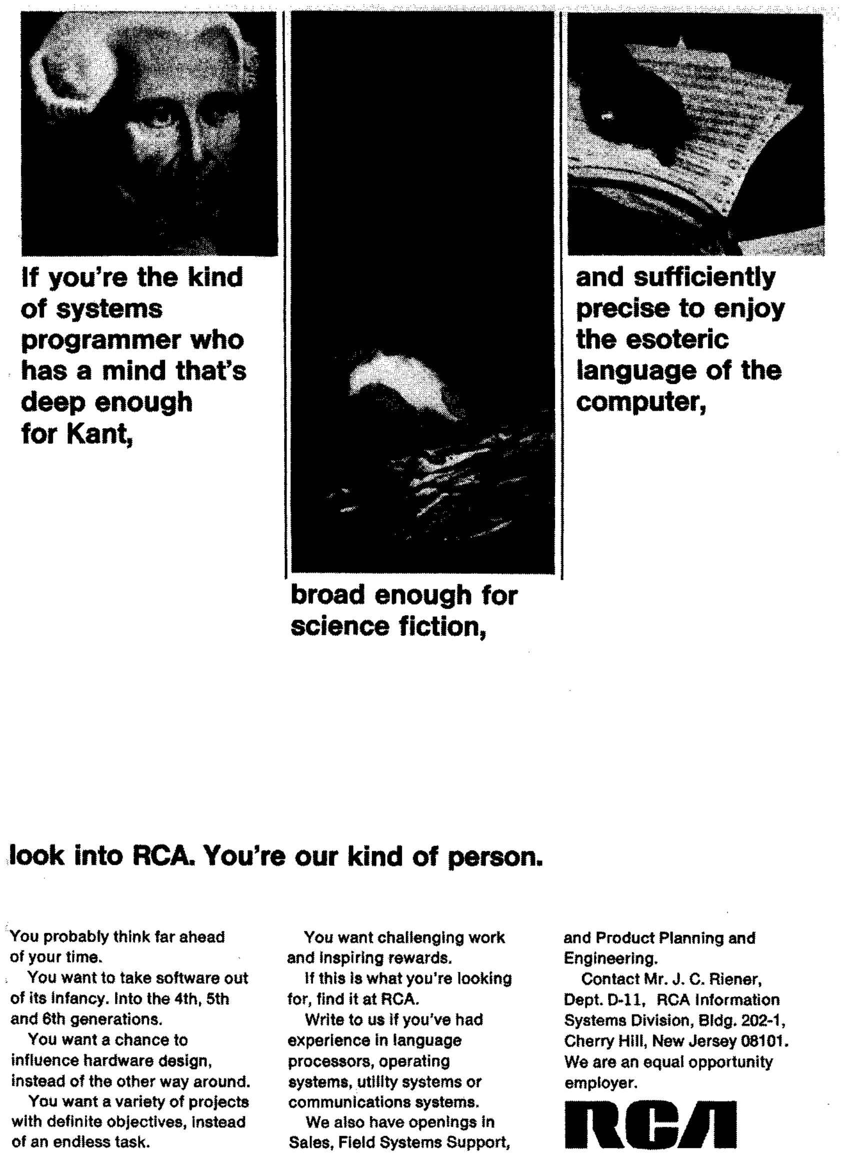
The conflation of programming and coding, and the association of both with low-status clerical labor, indicated the ways in which early software workers were gendered female. In the ENIAC project, of course, the programmers actually were women. In this respect programming inherited the gender identity of the human computing projects in which it originated. But the suggestion that coding was low-status clerical work also implied an additional association with female labor. As Margery Davies, Sharon Strom, and Elyce Rotella have described, clerical work had, by the second decade of the twentieth century, become largely feminized.[21] This was particularly true of clerical occupations that were characterized by the rigid division of labor and the introduction of new technologies. Some of these occupations carried over directly into the computer era: the job of keypunch operator, for example, had been thoroughly feminized long before it became associated with electronic data processing.[22] And although today we do not associate the work of keypunchers with the work of the computer programmer, in the 1950s and 1960s the differentiation between keypunch operators and other forms of computer work was not always clear. In any case, the historical pattern of the nineteenth and twentieth centuries has been that low-status occupations, with the exception of those requiring certain forms of physical strength, have often become feminized. In terms of the ENIAC, for example, the telephone switchboardlike appearance of the ENIAC programming cable-and-plug panels reinforced the notion that programmers were mere machine operators, that programming was more handicraft than science, more feminine than masculine, more mechanical than intellectual. The programmer/coder continued to occupy an uncertain position within the nascent association of computer professionals.

Throughout the next several decades programmers struggled to distance themselves from the status (and gender) connotations suggested by coder. An early manuscript version of the UNIVAC *Introduction to Programming* manual, for instance, highlighted the distinction between the managerial programmer and the technical coder: "In problem preparation, the detailed work may be accomplished by two individuals. The first, who may be called the 'programmer,' studies the problem, determines the appropriate method of solution, and prepares the flow chart. This person must be well versed in the particular field in which the problem lies, and should also be able to fully exploit the flexibility and versatility of the UNIVAC system. The second person, referred to as the 'coder,' need only be familiar with the technique of reducing the flow chart to the specific instructions, or coding, required by the UNIVAC to solve the problem."[23] By differentiating between these two tasks, one clerical and the other analytic, the manual reinforced the Goldstine and von Neumann model of the programmer. In this model the real business of programming was analysis: the actual coding aspect of programming was trivial and mechanical. "Problems must be thoroughly analyzed to determine the many factors that must be taken into consideration," suggested the same preliminary UNIVAC manual, but once this analysis had been completed, the "pattern of the [programming] solution would be readily apparent." Although this division of the programming process into two distinct and unequal phases did not survive into the published version of the UNIVAC documentation, its early inclusion highlighted the persistence of the programmer/coder distinction.

The Art of Programming

Although they continued to struggle with questions of status and identity, by the end of the 1950s computer programmers were generally considered to be anything but routine clerical workers. A Price Waterhouse report from 1959 titled *Business Experience with Electronic Computers* argued that "high quality individuals are the key to top grade programming. Why? Purely and simply because much of the work involved is exacting and difficult enough to require real intellectual ability and above average personal characteristics."[24] In fact, the study's authors observed that "the term 'programmer' is . . . unfortunate since it seems to indicate that the work is largely machine oriented when this is not at all the case. . . . [T]raining in systems analysis and design is as important to a programmer as training in machine coding techniques; it may well become increasingly important as systems get more complex and coding becomes more automatic."[25] Although Goldstine and von Neumann had envisioned a clear division of labor between planners and coders, in reality this boundary became increasingly indistinct. The clear implication of recent experience, in both scientific computation and business data processing, was that programmers should be given more responsibility for design and analysis, the idea that coding could be left to less-experienced or lower-grade personnel was "erroneous," and "the human element is crucial in programming."[26] Indeed, by the mid-1950s, a new model for programming had emerged that emphasized individual expertise and creativity. During this period computers remained a primarily scientific and military technology, and computer programming as a discipline retained a close association with the practice of mathematics. The limitations of early hardware devices usually meant that a simple programming problem could quickly turn into a research excursion into algorithm theory and numerical analysis. Computer programmers developed a reputation for creativity and ingenuity. Contemporary storage devices were so slow and had such little capacity that programmers had to develop great skill and craft knowledge to fit their programs into the available memory space. As John Backus (the IBM researcher best known as the inventor of the FORTRAN programming language) would later describe the situation, "Programming in the 1950s was a black art, a private arcane matter. . . . [E]ach problem required a unique beginning at square one, and the success of a program depended primarily on the programmer's private techniques and inventions."[27]

The notion that programming was a black art pervades the literature from this period. There are several reasons why programming was so difficult. To begin with, the programmer had to develop an algorithm suitable to the problem at hand. Since the primary purpose of the earliest computers was to produce solutions to complex mathematical functions that could not be solved analytically, these programs necessarily required skill in numerical analysis. Numerical analysis is the set of tools that mathematicians have developed to provide approximate solutions to otherwise-insoluble equations. This process of analysis was itself something of an art form: numerical solutions always involved a compromise between speed and accuracy—even when using the fastest computers. Choosing the right approximation required the programmer to balance acceptable error against the specific limitations of a given machine.



If you're the kind of systems programmer who has a mind that's deep enough for Kant,

broad enough for science fiction,

and sufficiently precise to enjoy the esoteric language of the computer,

look into RCA. You're our kind of person.

You probably think far ahead of your time.

You want to take software out of its infancy. Into the 4th, 5th and 6th generations.

You want a chance to influence hardware design, instead of the other way around.

You want a variety of projects with definite objectives, instead of an endless task.

You want challenging work and inspiring rewards.

If this is what you're looking for, find it at RCA.

Write to us if you've had experience in language processors, operating systems, utility systems or communications systems.

We also have openings in Sales, Field Systems Support, and Product Planning and Engineering.

Contact Mr. J. C. Riener, Dept. D-11, RCA Information Systems Division, Bldg. 202-1, Cherry Hill, New Jersey 08101. We are an equal opportunity employer.

RCA

Figure 2.1
RCA advertisement, 1962.

For problems that were not mathematical in nature, developing an appropriate algorithm could be even more challenging. This was a particular problem for the corporate users of computers. Even the simplest business activities can be difficult, if not impossible, to describe in terms of the limited instruction set understood by a computer. Programmers first had to thoroughly understand the activity in question, including all of its exceptional cases, imprecise terms, and potential errors. Not only was this process inherently difficult but it also frequently involved social and analytic skills foreign to the average programmer. "Because the background of the early programmers was acquired mainly in mathematics or other scientific fields, they were used to dealing with well-formulated problems and they delighted in a sophisticated approach to coding their solutions," noted the Price Waterhouse report. "When they applied their talents to the more sprawling problems of business, they often tended to underestimate the complexities and many of their solutions turned out to be oversimplifications. Most people connected with electronic computers in the early days will remember the one- or two-page flow charts which were supposed to cover the intricacies of the accounting aspects of a company's operations."[28] Most companies attempted to differentiate the more social and organizational processes essential to algorithm development, often referred to as system analysis, from the more technical procedures associated with programming. Inevitably the two would bleed into one another, however.[29]

Even after a suitable algorithm had been selected, the process of transforming that algorithm into a form that could be understood by a computer was challenging. Most electronic computers represented numbers internally in binary form, and so conversion routines from decimal to binary (and back) had to be developed. If the machine was a fixed-point machine, all of the numbers also scaled to stay within the bounds of the fixed-point arithmetic units. Since in a stored-program computer both programs and data were stored in the same memory, it was possible to confuse the two and create strange errors that were almost impossible to trace. Most of these machines had limited debugging capabilities (if any) and complicated mechanisms for accessing subroutine libraries. Programmers had to use obscure techniques to optimize for size rather than for legibility or ease of maintenance due to the limited amount of available memory. In order to coax every bit of speed out of a relatively slow storage device such as a rotating memory drum, programmers would carefully organize their coded instructions in such a way as to assure that each instruction passed by the magnetic read head in just the right order and at just the right execution time.[30] Only the best programmers could hope to develop applications that worked at acceptable levels of usability and performance. They had to cultivate a series of idiosyncratic and highly individual craft techniques designed to overcome the limitations of primitive hardware.[31]

In his memoir describing "Programming in America in the 1950s," John Backus offered an especially detailed example of the many ways in which a programmer project could run into problems:

> Some idea of the machine difficulties facing early programmers can be had by a brief survey of a few of the bizarre characteristics of the Selective Sequence Electronic Calculator (SSEC).
>
> This vast machine (circa 1948–1952) had a store of 150 words; instructions, constants, and tables of data were read from punched tapes the width of a punched card; the ends of an instruction tape were glued together to form a paper loop, which was then placed on one of 66 tape-reading stations. The SSEC could also punch intermediate data into tapes that could subsequently be read by a tape-reading station.
>
> One early problem strained the SSEC's capacity to the limit. The computation was divided into three phases; in the first phase a tape of many yards of intermediate results was punched out; during the second phase this tape was glued into a loop and mounted on a tape-reading station so that in the third phase it could be read many times.
>
> The problem ran successfully through many cycles of these three phases, but then a mysterious error began to appear and disappear regularly in the third phase. For a long time no one could account for it.
>
> Finally, the large pile of intermediate data tape was pulled from the bin below its reading station and a careful inspection revealed that it had been glued to form a Mobius strip rather than a simple loop. The result was that on every second revolution of the tape each number would be read in reverse order.[32]

As this anecdote suggests, writing programs under these constraints was a time-consuming and error-prone process. One the oldest-surviving computer programmers, a 126-line debugging tool developed for the Cambridge EDSAC machine (notable as being the first working stored-program computer in the world) was recently discovered to have contained more than twenty errors.[33] Because the author of the program, the mathematical physicist Maurice Wilkes, literally wrote the book on computer programming in the early 1950s (his 1951 *Preparation of Programs for an Electronic Digital Computer* is considered the first widely available textbook on programming), we can assume that this was not an unrepresentative example.[34] As Wilkes later recalled in his memoirs, early on in the life of the EDSAC, its programmers had "begun to realize that it was not so easy to get a program right as had at one time appeared." It was with some shock and dismay that he himself realized that "a good part of the remainder of my life was going to be spent in finding errors in my own programs."[35] The tedious process of identifying and removing these errors, known as "debugging," was time-consuming, difficult, and intellectually unfulfilling. As much as one-half of the budget of a large programming project could be spent on testing and debugging—activities that were perceived as being low-status and unpleasant.[36]

As will be described in subsequent chapters, improvements in computer hardware along with the development of compilers and other programming utilities would help alleviate some of the challenges associated with coding. But as many FORTRAN and COBOL programmers would soon realize, the dull and mechanical aspects of software development did not disappear with the advent of compilers and automatic programming languages. Nor did the intellectual challenges associated with analysis and design. Mistakes were inevitable, even from the most proficient of programmers. In one widely recited and tragic (and possibly apocryphal) example, a minor transcription error in control software for the Mariner I probe to Venus caused the spacecraft to veer off-course four minutes after takeoff, forcing NASA to destroy it remotely. The mistake that the programmer allegedly made was to replace the FORTRAN statement

```
DO 3 I = 1,3
```

with

```
DO 3 I = 1.3
```

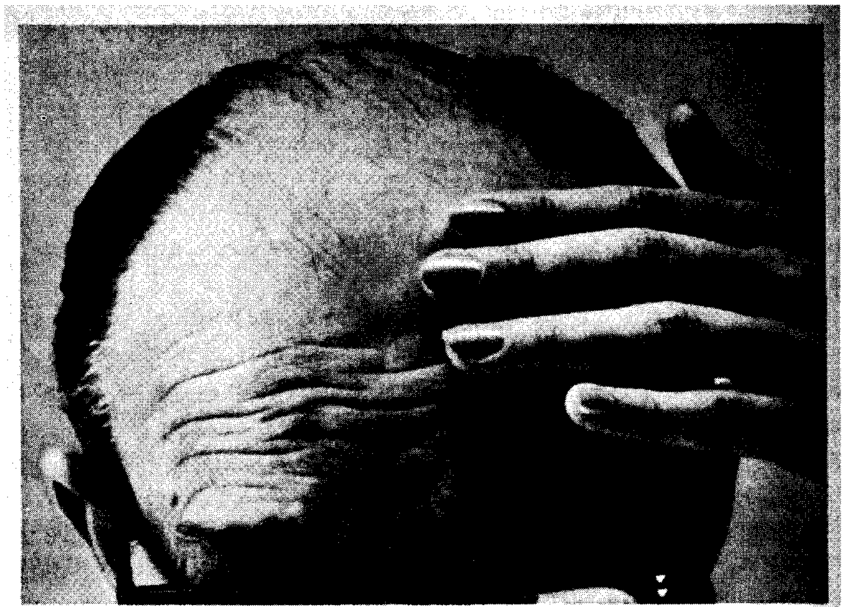
Instead of looping through a series of statements, as the code in the first version would have required, the latter form was interpreted by the FORTRAN compiler as the assignment of a variable. That the loss of the Mariner I could be caused by such a seemingly trivial error highlighted for many observers the central importance of employing only the most skilled programmers.[37] This perception holds true regardless of whether or not the Mariner I anecdote is factually accurate. During the late 1950s and 1960s such stories of software-related disaster were a staple of the popular press, and helped set the state for the emergence of a full-blown software crisis in the late 1960s.

Building Castles in the Air

In describing his experiences as the project manager of the single-largest and most expensive software development effort ever undertaken in the history of the IBM Corporation, the noted computer scientist Frederick Brooks provided a curiously literary portrayal of the computer programmer: "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."[38]

That a technical manager in a conservative corporation should use such lofty language in reference to such a seemingly prosaic occupation like programming is striking yet not unusual. But Brooks meant his literary metaphors to be taken seriously. Even more so than the poet, he argued, the programmer worked in the medium of the imagination, using words to bring to life grand conceptual structures. In fact, in the case of the programmers the relationship between words and reality was almost magical: "One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be." And like the magical incantation, the computer program demanded perfection: "If one character, one pause, of the incantation is not strictly in proper form, the magic doesn't work." This is what made programming so difficult, he suggested: "Human beings are not accustomed to being perfect, and few areas of human activity demand it. Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program."[39]

Like many of his contemporaries, Brooks was struggling to understand why software development projects seemed almost impossible to manage using conventional management techniques. In the late 1960s, Brooks had been the manager of the IBM OS/360 development project. The OS/360 operating system was the cornerstone of IBM's larger System/360 strategy, which consolidated IBM's computer product lines into a single range of hardware- and software-compatible machines. Although the System/360 turned out to be a tremendous success for IBM, it had almost been derailed by problems with the development of OS/360. In the years between 1963 and 1966, over five thousand staff years of effort went into the design, construction, and documentation of OS/360. When it was finally delivered in 1967, nine months late and riddled with errors, it had cost the IBM Corporation half a billion dollars—four times the original budget, and the single-largest expenditure in company history. And according to Brooks, the personal



Do you know what it means to find and care for a good programmer, and keep him happy?

Or maybe a couple of him (or her)?

They don't grow in labs.

They're an unusual breed.

Practically speaking, a profession.

It's not easy to tell who'll be a good programmer. Or who won't. Genius or plodder, once they're in the business, something within them comes out.

We bring it out.

Good programmers often come to us because we give them the company of so many others from whom they will learn.

And place them in one of our centers where they'll be happiest. And give them lots of training and experience.

And give them IBM computers to work with. The 1620, the 1401, the 7094, and soon the System/360. The latest. The small ones for small problems. The big ones for big problems.

And a team of technicians who can run them. And seldom waste a second.

They love to tackle matters like numerical control, multicomponent distillation, Type II structural steel frame specification, refinery operations forecasting, urban transportation planning, piping flexibility, rocket fuel evaluation, hydraulic network analysis, thermochemical equilibrium, supermarket chain and department store operations, high-rise apartment and industrial plant construction.

They get a thrill from Linear Programming and critical path scheduling (PERT) and generalized interrelated flow simulation (GIFS).

They get their triumphs from giving the computer a program with fewer steps. So it solves problems faster. Cheaper. In the data processing business, it's a *coup* when one company gets such a programmer.

Over 12,000 SBC customers know what we mean.

They know that the difference between computer services in this day and age of data processing machines is people.

Can SBC help you? How much does it cost? There's a way to find out quickly. Your Yellow Pages. "Data Processing Services." There we are.

SBC®

Service Bureau Corporation
Computing Sciences Division
425 Park Avenue
New York, N.Y. 10022

Figure 2.2
Service Bureau Corporation advertisement, 1964.

toll that OS/360 took on IBM's software personnel was perhaps even more significant.

The highly publicized failure of the OS/360 project served as a dramatic illustration of the shortcomings of the traditional management methods in software development. It was in *The Mythical Man-Month*, his postmortem analysis of the OS/360 disaster, that Brooks first compared programming to poetry. His larger point was that computer programming, as an inherently artistic activity, was resistant to most forms of industrial production. Take, for example, his own experience with OS/360: when faced with serious schedule slippages, quality problems, and unanticipated changes in scope, he and the other project leaders had done what traditional manufacturing managers were accustomed to doing, which was to add more resources. The only noticeable result was that the project fell more and more behind schedule.

After diagnosing the disease, Brooks proposed its cure. If skilled programmers were the sine qua non of quality software development, they must be elevated to the center of the production process. The remainder of *The Mythical Man-Month* is an attempt to figure out how to harness the power of highly artistic programmer/poets to the demands of industrial-strength software development. The development methodology that Brooks outlined was never widely adopted in industry, but his larger argument about the inherently creative nature of programming was. *The Mythical Man-Month* quickly became one of the most widely read and oft-quoted references on the practice of software development.

There is no doubt that in the formative years of commercial computing, there was widespread dissension within the programming community over the goals and direction of the programming profession. Computer scientists, corporate employers, and vocational programmers disagreed about the proper relationship between formal and idiosyncratic technique, local knowledge and generally applicable theory. What was largely agreed on, however, was that in the early 1960s, programming was "not yet a science, but an art that lacks standards, definitions, agreement on theories and approaches."[40] This popular perception of computer programming as a poorly understood, idiosyncratic, and creative process defined the discipline as it emerged in the 1950s, and continues to influence the culture and practice of programming even today. The notion that programming was an art served as both a resource and a source of much anxiety and discomfort for programmers.

For all of these reasons and more, programming in the 1950s acquired a reputation for being incomprehensible to all but a small set of extremely talented insiders. As John Backus would later describe it, "Each [programming] problem required a unique beginning at square one, and the success of a program depended primarily on the programmer's private techniques and invention."[41] Techniques developed for one application or installation could not be easily adapted for other purposes. There were few useful or widely applicable tools available to programmers, and certainly no science of programming. Programmers often worked in relative isolation, and had few opportunities for formal or even informal education. They generally perceived little value in the work going on at other firms or laboratories, as it was equally haphazard and idiosyncratic. They placed great emphasis on local knowledge and individual ability.

The widespread perception that programming was a black art pervades the industry and technical literature of the 1950s and 1960s.[42] Even today, more than a half century after the invention of the first electronic computers, the notion that computer programming still retains an essentially artistic character is still widely accepted.[43] Whether or not this is desirable is an entirely different question—one that remains a subject of considerable and contentious debate. What is important for the purposes of this book is the various ways in which the language of art, aesthetics, and craft is used throughout the history of computing to elevate, denigrate, or castigate programmers and other software specialists. By characterizing the work that they did as artistic, programmers could lay claim to the autonomy and authority that came with being an artist. If it were true, as one industry observer suggested in the late 1960s, that "generating software is 'brain business,' often an agonizingly difficult intellectual effort," then talented programmers were effectively irreplaceable, and should be treated and compensated accordingly.[44]

On the other hand, being artistic might also imply that one was not scientific or professional. One common usage of the word art, of course, is in reference to the visual, literary, or performing arts. In this context, describing programmers as artists implied that they were might be nonconformist, unreliable, or eccentric—not traits likely to endear them to straitlaced corporate managers. Although some programmers (and managers) did apply this meaning of the word art to programming—Brooks used a "programmers as poets" metaphor—for the most part the word was used in its more traditional association with craft technique and preindustrial forms of production.[45] When participants at the NATO

Conference on Software Engineering in 1968 portrayed computer programming as being "too artistic," they was using the word in this latter sense, as a rhetorical device for contrasting its "backward" craft sensibilities with "the types of theoretical foundations and practical disciplines" that they believed characterized "the established branches of engineering."[46] Note that the appeal here is to the tradition of the artisan or craftsperson, which is a masculine identity, rather than to the potentially effeminate artsy type.

For those computer programmers who also had academic aspirations, the word art was always used in opposition to science. For them the word suggested an undesirable lack of theoretical or mathematical rigor. They needed to distance the more artistic practices of programming from the more respectable discipline of computer science. This often brought these academically minded proto–computer scientists into conflict with working programmers, who had different professional and occupational agendas. The differences between these agendas would come to light in subsequent debates about programmer recruitment practices, programming language adoption, and academic curriculum.

50. Nathan Ensmenger, "Letting the 'Computer Boys' Take Over: Technology and the Politics of Organizational Transformation," *International Review of Social History* 48, no. S11 (2003): 153–180.

51. Harold Leavitt and Thomas Whisler, "Management in the 1980's," *Harvard Management Review* 36, no. 6 (1958): 41–48.

52. Whisler, "The Impact of Information Technology on Organizational Control."

53. Golda, "The Effects of Computer Technology on the Traditional Role of Management," 34.

54. Rosemary Stewart, *How Computers Affect Management* (Cambridge, MA: MIT Press, 1971), 196.

55. Thomas Alexander, "Computers Can't Solve Everything," *Fortune* 80, no. 5 (1969): 169.

56. McKinsey and Company, "Unlocking the Computer's Profit Potential," *Computers and Automation* 16, no. 7 (1969): 33.

57. Ibid., 33.

58. Harry Larson, "EDP: A 20 Year Ripoff!" *Infosystems* (1974): 26.

59. Ensmenger, "Letting the 'Computer Boys' Take Over."

60. Barry Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation* 19, no. 5 (1973): 48–59; Michael Mahoney, "Software: The Self-Programming Machine," in *From 0 to 1: An Authoritative History of Modern Computing*, ed. Atsushi Akera and Frederik Nebeker (New York: Oxford University Press, 2002).

61. Edsger Dijkstra, "The Humble Programmer," *Communications of the ACM* 15, no. 10 (1972): 873.

62. Martin Campbell-Kelly and William Aspray, *Computer: A History of the Information Machine* (New York: Basic Books, 1996), 201.

63. W. Saba, "Letter to the Editor," *IEEE Computer* 29, no. 9 (1996): 10; Edward Nash Yourdon, ed., *Classics in Software Engineering* (New York: Yourdon Press, 1979); Herbert Freeman and Phillip Lewis, *Software Engineering* (New York: Academic Press, 1980).

64. Frank Wagner, "Letter to the Editors," *Communications of the ACM* 33, no. 6 (1990): 628–629.

65. Ann Dooley, "100% over Budget," *Computerworld* 21, no. 7 (1987): 5.

66. David Morrison, "Software Crisis," *Defense* 21, no. 2 (1989): 72.

67. John Shore, "Why I Never Met a Programmer I Could Trust," *Communications of the ACM* 31, no. 4 (1988): 372.

Chapter 2

1. I. Bernard Cohen, *Howard Aiken: Portrait of a Computer Pioneer* (Cambridge, MA: MIT Press, 1999).

2. Norbert Wiener, *Cybernetics, or, Control and Communication in the Animal and the Machine* (Cambridge, MA: Technology Press, 1948).

3. Edmund Callis Berkeley, *Giant Brains; or, Machines That Think* (New York: Wiley, 1949).

4. Steven P. Schnaars and Sergio Carvalho, "Predicting the Market Evolution of Computers: Was the Revolution Really Unforeseen," *Technology in Society* 26, no. 1 (2004): 1–16.

5. Roddy Osborn, "GE and UNIVAC: Harnessing the High-Speed Computer," *Harvard Business Review* 32, no. 4 (1954): 99–107; M. L. Hurni, "Some Implications of the Use of Computers in Industry," *Accounting Review*, 29, no. 3 (1954): 447; John S. Coleman, "Computers as Tools for Management," *Management Science* 2, no. 2 (1956): 107.

6. "Office Robots," *Fortune*, 1952, 82–87, 112, 114, 116, 118.

7. Kenneth Flamm, *Creating the Computer: Government, Industry, and High Technology* (Washington, DC: Brookings Institution Press, 1988).

8. James W. Cortada, "Commercial Applications of the Digital Computer in American Corporations, 1945–1995," *IEEE Annals of the History of Computing* 18, no. 2 (Summer 1996): 18–29.

9. Bruce Gilchrist and Richard Weber, eds., *The State of the Computer Industry in the United States* (New York: American Federation of Information Processing Societies, 1972).

10. Gene Bylinsky, "Help Wanted: 50,000 Programmers," *Fortune* 75, no. 3 (1967): 141.

11. See Raul Rojas and Ulf Hashagen, eds., *The First Computers: History and Architectures* (Cambridge, MA: MIT Press, 2000).

12. Adele Goldstine, *A Report on the ENIAC (Electronic Numerical Integrator and Computer)* (technical report, Moore School of Electrical Engineering, University of Pennsylvania, June 1, 1946).

13. Richard F. Clippinger, *A Logical Coding System Applied to the ENIAC (Electronic Numerical Integrator and Computer)* (technical report, Ballistic Research Laboratories, Ordnance Department, Aberdeen Proving Ground, 1948).

14. John von Neumann, *First Draft of a Report on the EDVAC* (Philadelphia: Moore School of Electrical Engineering, University of Pennsylvania, June 30, 1945).

15. B. Randell, "The Origins of Computer Programming," *IEEE Annals of the History of Computing* 16, no. 4 (1994): 6–14.

16. W. Barkley Fritz, "The Women of Eniac," *IEEE Annals of the History of Computing* 18, no. 3 (1996): 13–23.

17. David Allan Grier, "The ENIAC, the Verb to Program, and the Emergence of Digital Computers," *IEEE Annals of the History of Computing* 18, no. 1 (1996): 53.

18. Ibid., 52.

19. W. Barkley Fritz, "The Women of Eniac," *IEEE Annals of the History of Computing* 18, no. 3 (1996): 20.

20. Henry S. Tropp, "ACM's 20th Anniversary: 30 August 1967," *Annals of the History of Computing* 9, no. 3 (1988): 269.

21. Margery W. Davies, *Woman's Place Is at the Typewriter: Office Work and Office Workers, 1870–1930* (Philadelphia: Temple University Press, 1982); Sharon Hartman Strom, *Beyond the Typewriter: Gender, Class, and the Origins of Modern American Office Work, 1900–1930* (Urbana: University of Illinois Press, 1992); Elyce J. Rotella, *From Home to Office: U.S. Women at Work, 1870–1930. Volume No. 25* (Ann Arbor, MI: UMI Research Press, 1981).

22. Thomas Haigh, "The Chromium-Plated Tabulator: Institutionalizing an Electronic Revolution, 1954–1958," *IEEE Annals of the History of Computing* 4, no. 23 (2001), 75–104.

23. Remington Rand UNIVAC, *Introduction to Programming: Programming for the UNIVAC, Part 1*, (1949), Hagley Museum Archives, Accession 1825, Box 372.

24. B. Conway, J. Gibbons, and D. E. Watts, *Business Experience with Electronic Computers: A Synthesis of What Has Been Learned from Electronic Data Processing Installations* (New York: Price Waterhouse, 1959), 81.

25. Ibid., 89–90.

26. Ibid., 90.

27. John Backus, "Programming in America in the 1950s: Some Personal Impressions," in *A History of Computing in the Twentieth Century: A Collection of Essays*, ed. N. Metropolis, J. Howlett, and Gian-Carlo Rota (New York: Academic Press,1980), 126.

28. Conway, Gibbons, and Watts, *Business Experience with Electronic Computers*.

29. Willis Ware, "As I See It: A Guest Editorial," *Datamation* 11, no. 5 (1965): 27–28.

30. George Trimble and Elmer Kubie, "Principles of Optimum Programming of the IBM Type 650," *IBM Applied Science Division Technical Newsletter* 8 (1954), 5–16.

31. J. N. Patterson Hume, "Development of Systems Software for the Ferut Computer at the University of Toronto, 1952 to 1955," *IEEE Annals of the History of Computing* 16, no. 2 (1994): 13–19.

32. Backus, "Programming in the 1950s."

33. Martin Campbell-Kelly, "The Airy Tape: An Early Chapter in the History of Debugging," *IEEE Annals of the History of Computing* 14, no. 4 (1992): 16–26.

34. Maurice Wilkes, David Wheeler, and Stanley Gill, *Preparation of Programs for an Electronic Digital Computer* (Reading, MA: Addison-Wesley, 1951).

35. Campbell-Kelly, "The Airy Tape."

36. Frederick P. Brooks, *The Mythical Man-Month: Essays on Software Engineering* (New York: Addison-Wesley, 1975), 20.

37. G. J. Meyers, *Software Reliability: Principles and Practices* (John Wiley and Sons, 1976).

38. Brooks, *The Mythical Man-Month*, 7.

39. Ibid., 7.

40. Bylinsky, "Help Wanted: 50,000 Programmers," 141.

41. John Backus, "Programming in America in the 1950s: Some Personal Impressions," In *A History of Computing in the Twentieth Century: A Collection of Essays*, ed. N. Metropolis, J. Howlett, and Gian-Carlo Rota (New York: Academic Press, 1980), 126.

42. George F. Weinwurm, ed., *On the Management of Computer Programmers* (London: Auerbach Publishers, 1970).

43. P. Mody, "Is Programming an Art?" *Software Engineering Notes* 17, no. 4 (1992): 19–21; Maurice Black, "The Art of Code" (PhD diss., University of Pennsylvania, 2002).

44. Bylinsky, "Help Wanted: 50,000 Programmers," 141.

45. Frederick Brooks, The Mythical Man-Month: Essays on Software Engineering (New York: Addison-Wesley, 1975), 7.

46. Brian Randall and J. N. Buxton, *Software Engineering: Proceedings of the NATO Conferences* (New York: Petrocelli/Carter, 1976).

Chapter 3

1. Brendan Gill and Andy Logan, "Talk of the Town," *New Yorker* 5 (January 1957): 18–19.

2. IBM Corporation, "Are You the Man to Command Electronic Giants?" *New York Times*, May 13, 1956, 157.

3. Gill and Logan, "Talk of the Town."

4. Ibid.

5. Mark I. Halpern, "Memoirs (Part 1)," *IEEE Annals of the History of Computing* 13, no. 1 (1991): 101–111.

6. Gene Bylinsky, "Help Wanted: 50,000 Programmers," *Fortune* 75, no. 3 (1967): 445–556.

7. Martin Campbell-Kelly and William Aspray, *Computer: A History of the Information Machine* (New York: Basic Books, 1996).

8. Bruce Webster, "The Real Software Crisis," *Byte Magazine* 21, no. 1 (1996): 218.

9. Bylinsky, "Help Wanted: 50,000 Programmers"; Stanley Englebardt, "Wanted: 500,000 Men to Feed Computers," *Popular Science*, January 1965, 106–109.

111. Gotterer, "The Impact of Professionalization Efforts on the Computer Manager," 368.

112. Louis Fein, "ACM Has a Crisis of Identity?" *Communications of the ACM* 10, no. 1 (1967): 1.

113. John Backus, cited in Richard Wexelblat, ed., *History of Programming Languages* (New York: Academic Press, 1981), 69.

114. Canning, "Professionalism: Coming or Not?" 2.

115. "Why Are Business Users Turned Off by ACM?" (1974) CBI 23, Box 1, Folder 3.

116. George Glaser, "Letter to W. Carlson," (July 15, 1974), CBI 23, "George Glaser Papers, 1960–1989," Box 1, Folder 3, Archives of the Charles Babbage Institute, University of Minnesota, Minneapolis.

Chapter 8

1. Ronald Graham, cited in Peter Naur, Brian Randall, and John Buxton, ed., *Software Engineering: Proceedings of the NATO Conferences* (New York: Petrocelli/Charter, 1976), 32.

2. Robert Glass, "Is There Really a Software Crisis?" *IEEE Software* 15, no. 1 (1998): 104–105.

3. Robert Bemer, *Computers and Crisis: How Computers Are Shaping Our Future* (New York: ACM Press, 1971).

4. Robert Gordon, "Review of Charles Lecht, *The Management of Computer Programmers*," *Datamation* 14, no. 4 (1968): 200–202.

5. Ibid., 7.

6. Martin Campbell-Kelly and William Aspray, *Computer: A History of the Information Machine* (New York: Basic Books, 1996), 210.

7. W. Saba, "Letter to the Editor," *IEEE Computer* 29, no. 9 (1996): 10; Edward Nash Yourdon, ed., *Classics in Software Engineering* (New York: Yourdon Press, 1979); Herbert Freeman and Phillip Lewis, *Software Engineering* (New York: Academic Press, 1980).

8. M. Douglas McIlroy, cited in ibid, 7.

9. Douglas McIroy, cited in Naur, Randall, and Buxton, *Software Engineering*, 7.

10. Ibid.

11. Brad Cox, "There Is a Silver Bullet," *Byte* 15, no. 10 (1990): 209.

12. Frederick Winslow Taylor, *The Principles of Scientific Management* (New York: Harper and Brothers, 1911).

13. Richard Canning, "Issues in Programming Management," *EDP Analyzer* 12, no. 4 (1974): 1–14.

14. Stuart Shapiro, "Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering," *IEEE Annals of the History of Computing* 19, no. 1 (1997): 25–54.

15. Gerald Weinberg, *The Psychology of Computer Programming* (New York: Van Nostrand Rheinhold, 1971).

16. Claude Baum, *The Systems Builders: The Story of SDC* (Santa Monica, CA: System Development Corporation, 1981), 52.

17. Ibid., 48.

18. Thomas C. Rowan, "The Recruiting and Training of Programmers," *Datamation* 4, no. 3 (1958): 16–18.

19. Baum, *The Systems Builders*, 47.

20. Philip Kraft, *Programmers and Managers: The Routinization of Computer Programming in the United States* (New York: Springer-Verlag, 1977), 39.

21. Philip Metzger, *Managing a Programming Project* (Englewood Cliffs, NJ: Prentice-Hall, 1973).

22. Richard Canning, "Issues in Programming Management," *EDP Analyzer* 12, no. 4 (1974): 1–14.

23. Joel Aron, *Part I: The Individual Programmer* (Reading, MA: Addison-Wesley, 1974); Joel Aron, *Program Development Process: The Programming Team* (Reading, MA: Addison-Wesley, 1983).

24. Canning, "Issues in Programming Management."

25. Richard Canning, "Professionalism: Coming or Not?" *EDP Analyzer* 14, no. 3 (1976): 1–12.

26. Brian Rothery, *Installing and Managing a Computer* (London: Business Books, 1968), 80.

27. Robert Gordon, "Personnel Selection," in *Data Processing: Practically Speaking*, ed. Stanley Naftaly and Fred Gruenberger (Los Angeles: Data Processing Digest, 1967): 85.

28. B. Conway, J. Gibbons, and D. E. Watts, *Business Experience with Electronic Computers: A Synthesis of What Has Been Learned from Electronic Data Processing Installations* (New York: Price Waterhouse, 1959), 81.

29. Ibid., 81–82.

30. Gene Bylinsky, "Help Wanted: 50,000 Programmers," *Fortune* 75, no. 3 (1967): 141.

31. H. V. Reid, "Problems in Managing the Data Processing Department," *Journal of Systems Management* 21, no. 5 (1970): 8–11; Richard Canning, "Managing Staff Retention and Turnover," *EDP Analyzer* 15, no. 8 (1977): 1–13.

32. Editorial, "EDP's Wailing Wall," *Datamation* 13, no. 7 (1967): 21.

33. Baum, *The Systems Builders*, 52.

34. Martin Campbell-Kelly, cited in Campbell-Kelly and Aspray, *Computer*, 144.

35. Thomas Wise, "IBM's $5,000,000,000 Gamble," *Time*, September 1966, 226.

36. Thomas Watson Jr., cited in Campbell-Kelly and Aspray, *Computer*, 199.

37. Frederick Brooks, cited in Campbell-Kelly and Aspray, *Computer*, 200; Emerson Pugh, Lyle Johnson, and John Palmer, *IBM's 360 and Early 370 Systems* (Cambridge, MA: MIT Press, 1991).

38. Frederick P. Brooks, *The Mythical Man-Month: Essays on Software Engineering* (New York: Addison-Wesley, 1975), 17.

39. Ibid., 31.

40. Ibid., 34–35.

41. Ibid., 42, 7.

42. Frederick P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer* 20, no. 4 (1987), 10–19.

43. F. Terry Baker and Harlan Mills, "Chief Programmer Teams," *Datamation* 19, no. 12 (1973): 198–199.

44. Ibid., 200.

45. Ibid., 201.

46. Clement McGowan and John Kelly, *Top-down Structured Programming Techniques* (New York: Petrocelli/Carter, 1975), 148.

47. Barbara Barry and John Naughton, *Structured Programming Series. Volume X. Chief Programmer Team Operations Description* (Gaithersburg, MD.: IBM Ferderal Systems, 1975), 12–13.

48. Stuart Shapiro, "Splitting the Difference," 25.

49. Barry Boehm, "Software Engineering," *IEEE Transactions on Computers*, no. 12 (1976): 349; Yourdon, *Classics in Software Engineering*, 63.

50. J. L. Ogdin, "The Mongolian Hordes versus Superprogrammer," *Infosystems* 19, no. 12 (1973): 23.

51. Daniel Couger and Robert Zawacki, "What Motivates DP Professionals?" *Datamation* 24, no. 9 (1978): 116–123; Canning, "Issues in Programming Management."

52. C. Anthony Hoare, "Software Engineering: A Keynote Address." In *Proceedings of the 3rd International Conference on Software Engineering* (Piscataway, NJ: IEEE Press, 1978): 1–4.

53. Carma McClure, *Managing Software Development and Maintenance* (New York: Van Nostrand Rheinhold, 1981), 77.

54. Ibid., 77–78, 86.

55. John Golda, "The Effects of Computer Technology on the Traditional Role of Management" (master's thesis, Wharton School of Business, University of Pennsylvania, 1965), 34.

56. Weinberg, *The Psychology of Computer Programming*, 56.

57. Ibid., 53.

58. Ibid.

59. Ibid.

60. Ibid.

61. Ogdin, "The Mongolian Hordes versus Superprogrammer," 23.

62. Canning, "Issues in Programming Management," 6.

63. Rudolph Hirsch, "Programming Performance: Monitoring, Maximization, and Prediction," in *Special Interest Group on Computer Personnel Research Annual Conference* (New York: ACM Press, 1972), 36–46.

64. Steve McConnell, *Code Complete: A Practical Handbook of Software Construction* (Redmond, WA: Microsoft Press, 1993), 287; Girish Parikh, *Programmer Productivity: Achieving an Urgent Priority* (Reston, VA: Reston Publishing, 1984), 209; Edward Yourdon, *Writings of the Revolution: Selected Readings on Software Engineering* (New York, Prentice Hall, 1986), 288.

65. Anthony Jay, *Corporation Man* (New York: Random House, 1971).

66. Douglas McGregor, *The Human Side of Enterprise* (New York: McGraw-Hill, 1960).

67. Ogdin, "The Mongolian Hordes versus Superprogrammer," 23.

68. Bo Sanden, "Programming Masters Break Out of the Managerial Mold," *Computerworld* (1986): 73.

69. Henry Lucas, "*On the Failure to Implement Structured Programming and Other Techniques*," in *Proceedings of 1975 ACM Annual Conference* (New York: ACM Press, 1975), 291–293.

70. McClure, *Managing Software Development and Maintenance*, 74–75.

71. Edsger Dijkstra, cited in Eloina Paleaz, "A Gift from Pandora's Box: The Software Crisis" (PhD diss., University of Edinburgh, 1988), 175.

72. Donald MacKenzie, "A View from the Sonnenbichl: On the Historical Sociology of Software and System Dependability," in *History of Computing: Software Issues*, ed. Ulf Hashagen, Reinhard Keil-Slawik, and Arthur L. Norberg (Berlin: Springer-Verlag, 2002): 97–122.

73. Friedrich L. Bauer, "Software Engineering: A Conference Report," *Datamation* 15, no. 10 (1969).

74. John N. Buxton, cited in Paleaz, "A Gift from Pandora's Box," 185.

75. Douglas Ross, cited in Paleaz, "A Gift from Pandora's Box," 182.

76. Campbell-Kelly and Aspray, *Computer*, 201.

77. Michael Mahoney, "The Roots of Software Engineering," *CWI Quarterly* 3, no. 4 (1980): 325–334.

78. Christopher Strachey, cited in Randall and Buxton, *Software Engineering*, 147.

79. Ibid.

80. Ibid.

81. Ann Dooley, "100% Over Budget," *Computerworld* (1987): 5.

82. Gary Chapman, "Bugs in the Program," *Communications of the ACM* 33, no. 3 (1990): 251–252.

83. David Morrison, "Software Crisis," *Defense* 21, no. 2 (1989): 72.

84. William Wayt Gibbs, "Software's Chronic Crisis," *Scientific American* 271, no. 3 (1994): 86.

Chapter 9

1. J. Jimms, "Could Y2K cause a global recession?" *Fortune* 138, no. 7 (1998): 172–176.

2. Fred Kaplan, "Military on Year 2000 alert," *Boston Globe* (June 21, 1998): A1.

3. David Edgerton, *The Shock of the Old: Technology and Global History since 1900* (Oxford: Oxford University Press, 2007).

4. B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM* 21, no. 6 (1978): 466–471; Girish Parikh, "Software Maintenance: Penny Wise, Program Foolish," *SIGSOFT Software Engineering Notes* 10, no. 5 (1985): 89–98; 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), 107–112.

5. Richard Canning, "The Maintenance 'Iceberg,'" *EDP Analyzer* 10, no. 10 (1972): 1–13.

6. Gerardo Canfora and Aniello Cimitile, *Software Maintenance* (Technical report, University of Sannio, 2000).

7. Maurice Wilkes, *Memoirs of a Computer Pioneer* (Cambridge: MIT Press, 1985).

8. David Rine, "A Short Overview of a History of Software Maintenance: As It Pertains to Reuse," *SIGSOFT Software Engineering Notes* 16, no. 4 (1991): 60–63.

9. Canning, "The Maintenance 'Iceberg'" (1972).

10. E. Burton Swanson, "The Dimensions of Maintenance," in *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering* (San Francisco: IEEE Computer Society Press, 1976), 492–497.

11. Girish Parikh, "What Is Software Maintenance Really? What Is in a Name?" *SIGSOFT Software Engineering Notes* 9, no. 2 (1984): 114–116.

12. Frederick Brooks, *The Mythical Man-Month: Essays on Software Engineering* (New York: Addison-Wesley, 1975), 7.