

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]


**FROM "BLACK ART" TO INDUSTRIAL DISCIPLINE:
THE SOFTWARE CRISIS AND THE MANAGEMENT OF PROGRAMMERS**

Nathan L. Ensmenger

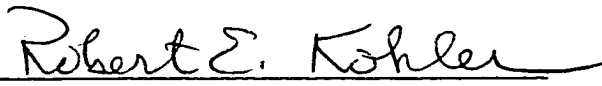
A DISSERTATION
In
History and Sociology of Science

Presented to the Faculties of the University of Pennsylvania in Partial
Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2001



Supervisor of Dissertation



Graduate Group Chairperson

UMI Number: 3015310

UMI[®]

UMI Microform 3015310

Copyright 2001 by Bell & Howell Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

For Deborah and the boys.

Acknowledgements

There are many people who contributed to the completion of this dissertation. It is only as I sit down to write these acknowledgements that I realize how indebted I am to my colleagues, advisors, and family.

Emily Thompson was a model advisor, and her thoughtful comments and conscientious editing greatly enhanced the quality of my writing. Rob Kohler has been a fount of ideas and inspiration throughout my graduate school experience, and this project originated in one of his seminars. Janet Tighe was, as always, gracious with her time, advice, and chocolate. Walter Licht influenced my interpretation of software labor history in many ways, not least through his comments on my chapters.

William Aspray deserves special recognition for his generous and unflagging support of this and other projects. His influence on the discipline of the history of computing extends far beyond his many publications, and includes as well his remarkable gift for encouraging young scholars.

There are many other historians who have been kind enough to read and comment on my work. Michael Mahoney, Bill Leslie, David Hounshell, and Bruce Seely have all been particularly stimulating influences on my research.

Financial support for this project has come from the History & Sociology of Science department, the School of Arts & Sciences, and the Charles Babbage Institute. The folks at the Babbage Institute have been extremely good to me, and Arthur Norberg and Jeffrey Yost deserve special thanks for their support and encouragement, as do Erwin and Adelle Tomash, who provided the funding for my fellowship.

Additional financial support also came from quite a different quarter. This project has the unique distinction of being probably the only humanities dissertation ever funded (albeit indirectly) by Silicon Valley venture capital. My former colleagues at E.J. Bell and Associates (now Crosstier.Com) - Ed Bell, David Stansell, and Jeff Trabaldo - provided me with enough short-term consulting work to support my very expensive graduate school habit. Thanks, guys - there is no way I could have done this without you.

It would be difficult to overstate the contributions of my colleagues in the History and Sociology of Science Department. My fellow dissertation students - Joshua Buhs, Tom Haigh, Erin McLeary, and Susan Miller - provided encouragement, editorial suggestions, and a much-needed sense of perspective and humor. Pat Johnson, Joyce Roselle, and Sybil Csigi made things happen. It would have been impossible to finish this project without their assistance.

Finally, let me acknowledge the long-suffering and seemingly inexhaustible support of my family. I thank my parents, Stephen and Elisabeth, for all the love and opportunities that they have provided me. My wife Deborah has put up with a great deal during the pursuit of my mad dream; she deserves much of the credit for its completion. Deborah, you truly are my ideal partner and “a prize beyond all jewels.” And, last but certainly not least, a special thanks to my sons, Asher and Tate, who made the first five minutes after I came home from work the best part of every day.

ABSTRACT

FROM “BLACK ART” TO INDUSTRIAL DISCIPLINE: THE SOFTWARE CRISIS AND THE MANAGEMENT OF PROGRAMMERS

Nathan L. Ensmenger

Professor Emily Thompson

For almost as long as there has been software, there has been a software crisis. Laments about the inability of software developers to produce products on time, within budget, and of acceptable quality and reliability have been a staple of industry literature from the early decades of commercial computing to the present. In an industry characterized by rapid change and innovation, the rhetoric of the crisis has proven remarkably persistent.

Rather than treating the software crisis as a well-defined and universally understood phenomenon, as it is often assumed to be in the industry and historical literature, this dissertation considers it as a socially constructed historical artifact. Ostensibly a debate about the “one best way” to manage a computer programming project, the software crisis was in reality a series of highly contested social negotiations about the role of electronic computing – and of computing professionals – in modern corporate and academic organizations.

This dissertation explores the history of the computer in the modern corporation as viewed from the perspective of computer programmers. It describes the culture and methods of programming practice as they developed in

the late 1950s and early 1960s, and traces the changes that occurred in these practices as the programming community transformed from a tightly knit family of "computer people" into a diverse and highly fragmented collection of programmer/coders, systems analysts, and information technology specialists. My argument is that the significant developments in software management that occurred in the 1950s and 1960s can best be understood as a jurisdictional struggle over control of the increasingly valuable occupational territory opened up by the electronic digital computer. Like any major new social or technological innovation, the computer could not simply be inserted, unchanged and unnoticed, into the well-established social, technological, and political systems that comprised the modern corporate organization. Just as the computer itself was gradually reconstructed, in response to a changing social and technical environment, from a scientific and military instrument into a mechanism for corporate communication and control, the modern business organization had to adapt itself to the presence of a powerful new technology.

Table of Contents

PROLOGUE: THE PERPETUAL SOFTWARE CRISIS	1
I. THE SOFTWARE CRISIS AS A CRISIS OF PROFESSIONAL IDENTITY	1
INTRODUCTION: INVENTING THE COMPUTER PROGRAMMER	12
I. INVENTING THE COMPUTER PROGRAMMER.....	12
II. THE LABOR CRISIS IN PROGRAMMING.....	27
III. PROGRAMMERS AS PROFESSIONALS.....	32
IV. PROGRAMMERS AND MANAGERS	38
V. ENGINEERING A SOLUTION	42
CHAPTER ONE: PROGRAMMING AS TECHNOLOGY AND PRACTICE	44
I. AUTOMATIC PROGRAMMERS	44
II. THE TOWER OF BABEL.....	51
III. NO SILVER BULLET	87
CHAPTER TWO: THE MONGOLIAN HORDE VERSUS THE SUPERPROGRAMMER	93
I. THE SOFTWARE CRISIS AS A PROBLEM OF PROGRAMMER MANAGEMENT.....	93
II. ARISTOCRACY, DEMOCRACY, AND SYSTEMS DESIGN.....	106
III. PROGRAMMERS, EVOLUTION, AND THE STRUGGLE FOR OCCUPATIONAL TERRITORY	146
CHAPTER THREE: THE PROFESSIONALIZATION OF PROGRAMMING	167
I. THE HUMBLE PROGRAMMER	167
II. THE DRIVE TOWARDS PROFESSIONALISM	171
III. COMPUTER SCIENCE AS THE KEY TO PROFESSIONALISM	176
IV. THE CERTIFIED PUBLIC PROGRAMMER.....	197
V. PROFESSIONAL ASSOCIATIONS.....	221
VI. THE LIMITS OF PROFESSIONALISM.....	243
EPILOGUE: NO SILVER BULLET	249
I. FROM EXHILARATION TO DISILLUSIONMENT	249
II. SOFTWARE'S CHRONIC CRISIS.....	254
BIBLIOGRAPHY	257

Table of Figures

ADVERTISEMENT FOR BENDIX COMPUTING, C. 1967.....	35
ADVERTISEMENT FROM DATAMATION MAGAZINE, C. 1968.....	56
GENEALOGY OF PROGRAMMING LANGUAGES, 1952-1970.....	80
COMMUNICATIONS PATTERNS IN THE CHIEF PROGRAMMING TEAM.....	123
THE DEVELOPMENT SUPPORT LIBRARY.....	129
CDP RECIPIENTS, 1961-1973.....	201
ASSOCIATION FOR COMPUTER MACHINERY MEMBERSHIP, 1961-1973.....	227

Prologue: The Perpetual Software Crisis

The shortage of programmers in the United States, or the “software crisis” as it has been dubbed by experts, is producing fallout among practicing professionals today.¹

Harry Leslie, *Datamation* (1967)

Demand [for capable developers] will continue to outstrip supply for the foreseeable future. Hence, more and more software will be behind schedule, over budget, underpowered, and of poor quality – and there's nothing we can do about it.²

“The Real Software Crisis,” *Byte Magazine* (1996)

I. The Software Crisis as a Crisis of Professional Identity

In the fall of 1968, a diverse group of academic computer scientists, corporate managers, and military officials gathered in Garmisch, Germany for the first-ever NATO Conference on Software Engineering. The conference was intended to address what many industry observers believed to be an impending “crisis” in software production. Large software development projects had acquired a reputation for being behind-schedule, over-budget, and bug-ridden. “We build software like the Wright brothers built airplanes,” complained one prominent participant: “build the whole thing, push it off the cliff, let it crash, and start over again.”³ The solution to the so-called “software crisis,” suggested the conference organizers, was for software developers to adopt “the types of theoretical

¹ Harry Leslie, “The Report Program Generator,” *Datamation* (26-28) 13, 6 (1967), 26.

² Bruce Webster, “The Real Software Crisis,” *Byte Magazine* 21, 1 (1996), 218.

³ R.M. Graham, quoted in Peter Naur, ed. *Software Engineering: Proceedings of the NATO Conferences* (New York: Petrocelli/Charter, 1976), 32.

foundations and practical disciplines that are traditional in the established branches of engineering."⁴ In the interest of efficient software manufacturing, the "black art" of programming had to make way for the "science" of software engineering.

By defining the software crisis in terms of the discipline of "software engineering," the NATO Conference set an agenda that influenced many of the technological, managerial, and professional developments in commercial computing for the next several decades. The general consensus among historians and practitioners alike is that the Garmisch meeting marked "a major cultural shift in the perception of programming. Software writing started to make the transition from being a craft for a long-haired programming priesthood to becoming a real engineering discipline. It was the transformation from an art to a science."⁵ The call to integrate "good engineering principles" into the software development process has been the rallying cry of software developers from the late 1960s to the present.⁶

Despite the widespread adoption of the language and ideology of software engineering, the vision of a rational "software factory" outlined at the 1968

⁴ Naur, et al., 7.

⁵ Martin Campbell-Kelly and William Aspray, *Computer: A History of the Information Machine* (New York: Basic Books, 1996), 201.

⁶ For example, see W. Saba, "Software Engineer, Magic, and Witchcraft," *IEEE Computer* 29, 9 (1996); Edward Yourdon, ed., *Classics in Software Engineering* (New York: Yourdon Press, 1979); Herbert Freeman and Phillip Lewis, *Software Engineering* (New York: Academic Press, 1980)

Garmisch Conference has never coalesced into reality. The rhetoric of crisis continues to dominate discussions about the health and future of the industry. A recent report on the software crisis by the National Software Alliance raised “serious concerns not only for the software industry, but, given the reliance of every industry on software, for the entire U.S. economy.”⁷ Software engineering has not yet been able to establish itself as a “real” engineering discipline, and many computer programmers still regard what they do as more of an art than a science. According to one recent article in *Scientific American*, “A quarter of a century later software engineering remains a term of aspiration. The vast majority of computer code is still handcrafted from raw programming languages by artisans using techniques they neither measure nor are able to repeat consistently.” The Y2K crisis is only the most recent manifestation of the software industry’s apparent predilection for apocalyptic rhetoric. In an industry characterized by rapid change and innovation, the rhetoric of crisis has proven remarkably consistent.

References to the chronic “software crisis” are in fact so ubiquitous that it is possible to lose sight of their historical origins and significance. Specific claims about the nature and extent of the crisis can be used, however, as a lens through

⁷ Quoted by Bob Bellinger, “Software Warning,” *Electronic Engineering Times* (November 3, 1997). The NSA was responding to a statement by Federal Reserve Chairman Alan Greenspan.

which to examine broader issues in the history of software. This dissertation explores the historical construction of the software crisis as a crisis of programming labor. It argues that many of the crucial innovations in modern software development – high-level programming languages, structured programming techniques, and software engineering methodologies, for example – reflect corporate concerns about the supply, training, and management of programmers.

Rather than treating the software crisis as a well-defined and universally understood phenomenon, as it is often assumed to be in the industry and historical literature, this dissertation will consider it as a socially constructed historical artifact. Like any major new social or technological innovation, the computer could not simply be inserted, unchanged and unnoticed, into the well-established social, technological, and political systems that comprised modern corporate and academic organizations. Just as the computer itself was gradually reconstructed, in response to a changing social and technical environment, from a scientific and military instrument into a mechanism for corporate control and communication, modern businesses and universities had to adapt themselves to the presence of a powerful new technology.

My research focuses on this process of mutual reconstruction as it occurred in commercial organizations in the 1950s and 1960s, for these were by far the

largest users of information technology in that period. As the computer transformed from a *tool to be managed* into a tool *for* management, computer users emerged as powerful “change-agents” (to use the management terminology of the era). As one Wharton MBA graduate warned his colleagues in a 1965 article, “As the EDP [electronic data processing] group takes on the role of a corporate service organization, able to cut across organizational lines, a revolution in the organizational power structure is bound to occur.”⁸ Faced with this new and powerful challenge to their occupational territory, traditional managers attempted to reassert their control over corporate data processing. The stridency of the crisis rhetoric used in this period reflects the increasingly contested nature of this struggle over the intellectual and occupational boundaries of the nascent programming profession.

Ostensibly a debate about the “one best way” to manage a computer programming project, the software crisis was in reality a series of highly contested social negotiations about the role of electronic computing – and of computing professionals – in modern corporate and academic organizations.

Because the labor crisis in programming has been so widely referred to and written about, it serves as an ideal launching pad for an exploration of other, less familiar issues in the history of software. Software represents a crucial link

⁸ John Golda, “The Effects of Computer Technology on the Traditional Role of Management,” (MBA thesis, Wharton School, University of Pennsylvania, 1965), 34

between the computer and its larger social and economic environment. In order to take full advantage of the promise of information technology, users had to adapt it to their own particular systems and processes. Software is what made the computer useful. It transformed the latent power of a general-purpose machine into a specific tool for solving actual real-world problems. As the historian Michael Mahoney has suggested, software applications are "what make the computer worth having; without them a computer is of no more utility or value than a television set without broadcasting."⁹ For many organizations, it was the availability of software that most determined the success or failure of their computerization efforts. As computer hardware became faster, more reliable, and less-expensive, the relative importance of software – and software developers - became even more pronounced.

This dissertation attempts to provides a connection between the history of software and a larger literature on labor and social history and the history of technology. It reexamines the perennial debate about programming training and management in terms of contemporary debates about socially constructed notions of "skill," "knowledge," and "productivity." It describes the culture and methods of programming practice as they developed in the late 1950s and early

⁹ Michael Mahoney, "Software: the self-programming machine," to appear in *Creating Modern Computing*, ed. A. Aker and F. Nebeker, (New York: Oxford U.P., forthcoming).

1960s, and traces the changes that occurred in these practices as the programming community transformed from a tightly knit family of "computer people" into a diverse and highly fragmented collection of programmer/coders, systems analysts, and information technology specialists. It focuses on the conflict between the craft centered practices of the early programmers and the "scientifically" oriented management techniques of their corporate managers. It argues that the skills and expertise that computer programmers possessed transcended traditional boundaries between business knowledge and technical expertise, and that computer programmers constituted a substantial challenge to established corporate hierarchies and power structures. It suggests that the continued persistence of a "software crisis" mentality among industrial and governmental managers, as well as the seemingly unrelenting quest of these managers to develop a software development methodology that would finally eliminate corporate dependence on the craft knowledge of individual programmers, can best be understood in light of this struggle over workplace authority.

It should be noted, however, that the study of labor processes presents serious methodological challenges to historians. Conventional interpretations of the software crisis are often based on the software management literature, which is typically biased towards the perspective of employers and managers. This

literature also tends to reflect an ideal rather than reality. The voice of the worker is rarely represented in the types of sources that historians are accustomed to dealing with. Very little is known about the experiences and attitudes of the typical software developer, or about the craft practices and "shop floor" activities of programmers.¹⁰ There is almost no secondary literature available on this subject.

In an attempt to counter the traditional bias towards management perspectives, I deliberately chose to construct my narrative around an eclectic collection of source material and perspectives. The ongoing debate about the software labor crisis has been passionate, contentious, and replete with ambiguities and self-contradictions. The fact that the community of software workers included both former theoretical physicists and Helen Gurley Brown's "Cosmo Girls" is not an incidental curiosity; it is an essential element of the labor history of software.¹¹ One of the goals of this dissertation is to convey the sense of excitement and drama experienced by early software workers.

¹⁰ Michael Mahoney has more fully described these difficulties in his "The History of Computing in the History of Technology," *Annals of the History of Computing* 10,2 (1988), 113-125.

¹¹ In 1968, at the height of the software crisis, Helen Gurley Brown, the controversial editor of *Cosmopolitan Magazine*, published an article encouraging her "Cosmo Girls" to become programmers. The article provoked a strong reaction on the part of male programmers with professional aspirations. See Lois Mandel, "The Computer Girls," *Cosmopolitan Magazine* (April, 1967), 52-56.

The **Introduction** describes the origins of the computer programming as an occupational and academic discipline, and explores the emergence of a so-called “software crisis” in the late 1950s and 1960s. It focuses on the conflict between the “art of programming” and the “science” of software engineering. It argues that by representing the art of programming as a complex blend of creative genius and technical acumen, “exacting and difficult enough to require real intellectual ability,” its practitioners were able lay claim to the professional status associated with similarly demanding and creative endeavors.¹² Although corporate managers were often frustrated by their dependence on the idiosyncratic techniques of individual programmers, there was little evidence or experience to suggest that they had any alternative. The psychological studies and aptitude tests that companies used to identify potential programmers reinforced conventional wisdom by suggesting that good programmers, like gifted chess players or musicians, were born, not made.

By the beginning of the 1960s, however, a changing social and technical environment prompted a re-evaluation of the nature and causes of the software crisis. **Chapter One** describes attempts to develop technical solutions to the software crisis. New “automatic programming” systems such as FORTRAN and

¹² B. Conway, *Business experience with electronic computers, a synthesis of what has been learned from electronic data processing installations*. (Controllers Institute Research Foundation, Inc.: New York, 1959), p. 81.

COBOL promised to "eliminate the middleman" by allowing users to program their computers directly, without the need for expensive programming talent.¹³

Although these technologies threatened the professional autonomy of programmers, they also served as a focus for productive theoretical research, and helped establish computer science as a legitimate academic discipline.

By the end of the 1960s the search for a "silver bullet" solution to the software crisis had turned away from programming languages and towards more comprehensive techniques for managing the programming process.

Chapter Two explores the changing relationship between software workers and their corporate employers. Faced with a new and powerful challenge to their occupational territory, traditional managers attempted to reassert their control over corporate data processing. They attempted to impose on software development lessons learned from traditional industrial manufacturing. Like most professional managers in this period, they assumed that the proper management of programming projects was simply a matter of identifying and implementing the "one best way" to develop software components.

Chapter Three focuses on the attempts of programmers to establish the institutional structures associated with professionalism: university curricula;

¹³ RAND Symposium, "On Programming Languages, Part II," *Datamation* 8, 11 (1962); Fred Gruenberger and Stanley Naftaly, eds., *Data Processing. Practically Speaking* (Los Angeles: Data Processing Digest, 1967), 85.

certification programs; professional associations; and codes of ethics. It argues that the professionalization of computer programming represented a potential solution to the software crisis that appealed to programmers and employers alike. The controversy that surrounded the various professional institutions that were established in this period, however, reveals the deep divisions that existed within the programming community about the nature of programming skill and the future of the programming professions.

The **Epilogue** relates the events of the 1960s to subsequent developments in software engineering. It suggests that thinking about the invention of this discipline as a series of interconnected social and political negotiations, rather than an isolated technical decision about the “one best way” to develop software components, provides an essential link between internal developments in information technology and their larger social and historical context.

Introduction: Inventing the Computer Programmer

Creativity is a major attribute of technically oriented people. Look for those who like intellectual challenge rather than interpersonal relations or managerial decision-making. Look for the chess player, the solver of mathematical puzzles.¹

"Selection of EDP Personnel," *Personnel Journal* (1965)

Another striking characteristic of programmers is their disinterest in people. Compared with other professional men, programmers dislike activities involving close personal interaction. They prefer to work with things rather than people.²

"Vocational Interests of Computer Programmers," *Journal of Applied Psychology* (1967)

I. *Inventing the Computer Programmer*

For almost as long as there has been software, there has been a software crisis.³ Laments about the inability of software developers to produce products on time, within budget, and of acceptable quality and reliability have been a staple of industry literature from the early decades of commercial computing to the present. The "software gap" of the 1950s led to "software turmoil" in the early 1960s and by the end of the decade had emerged as a full-blown "software

¹ Joseph O'Shields, "Selection of EDP Personnel," *Personnel Journal* 44, 9 (October 1965), 472.

² Dallis Perry and William Cannon, "Vocational Interests of Computer Programmers," *Journal of Applied Psychology* 51, 1 (1967).

³ The first known use of the word "software" appears in a 1959 article by the Princeton statistician John Tukey. See John Tukey, "The Teaching of Concrete Mathematics," *American Mathematical Monthly* 65, 1 (1958). By early 1962 Daniel McCracken was already lamenting the "software turmoil" that threatened to "set the software art back several years." ("The Software Turmoil: Nine Predictions for '62," *Datamation* 8, 1 [1962]). References to the "Gap in Programming Support" appear even earlier (Robert Patrick, "The Gap in Programming Support," *Datamation* 7, 5 [1961]).

crisis.” Despite numerous attempts to address this crisis, either by the introduction of new “automatic programming” technologies or the imposition of new managerial controls on the software development process, the rhetoric of crisis has continued to dominate discussions about the health and future of the software industry. In the words of one industry observer, by the middle of the 1980s “the software crisis [had] become less a turning point than a way of life.”⁴

This chapter explores the historical construction of the software crisis as a crisis of programming labor. It argues that many of the crucial innovations in modern software development – high-level programming languages, structured programming techniques, and software engineering methodologies, for example – reflect corporate concerns about the supply, training, and management of programmers. In their role as mediators between the technical system (the computer) and its social environment (existing structures and practices), computer programmers have always played a crucial role in defining what the computer is and what it could be used for. They also served as a focus for opposition to and criticism of the use of information technology. Much of the rhetoric of the software crisis, for example, has focused on the character and practices of programmers.

⁴ John Shore, “Why I Never Met a Programmer I Could Trust,” *Communications of the ACM* 31, 4 (1988).

Despite their obvious importance to the history of commercial computing, we know almost nothing about the background and practices of the average working computer programmer. In many respects computer programmers are the unsung heroes of the computer and information revolution. The Andrei Ershov has gone so far as to call them "the lynchpin of the second industrial revolution."⁵ And yet programmers have long complained about the low status of their profession and its lack of public recognition.⁶ In any case, by the end of the 1960s there were more than three hundred thousand of these generally anonymous programmers and systems analysts working in the United States alone. Clearly this large and influential group of computer users did not simply appear out of thin air to meet the burgeoning demands of corporate computer installations. In a complex process that took several decades, and which has probably still not yet been completed, the computer programmer had to be painstakingly invented.

It might seem odd to talk about the invention of an occupation. After all, invention is a word that usually reserved for machines, rather than people or groups of people. And yet historians have long suggested that technological innovators, including the designers of electronic computers, also invent the kind

⁵ A.P. Ershov, "Aesthetics and the Human Factor in Programming," *Communications of the ACM* 15, 7 (July, 1972), 503.

⁶ Daniel McCracken, "The Human Side of Computing," *Datamation* 7, 1 (1961), 9-10; C.J.A., "In defense of programmers," *Datamation* 13, 9 (1967); Datamation Editorial, "Editor's Readout: The Certified Public Programmer," *Datamation* 8, 3 (1962).

of people they expect to use their innovations.⁷ The two acts of invention are in fact inseparable: assumptions made about who will be using a technology, how, and for what purposes, inevitably influence its eventual design. This means that the invention of the user, like the invention of the technology itself, is often a highly contested social process involving conflict and negotiation. Much of the discourse about the software crisis, for example, focuses not so much on the software itself as on the character and practices of its user/programmers.

In the case of the electronic computer, this process of “inventing” the user was particularly controversial. The decades of the 1950s and 1960s were a period of rapid and fundamental changes in the technology and practice of computing. Whereas in the early 1950s electronic computers were generally regarded as interesting but extravagant scientific curiosities, by 1963 these devices and their associated peripherals formed the basis of a billion-dollar industry. By the end of the decade, more than 165,000 computers had been installed in the United States alone, and the computer and software industries employed several hundred thousand individuals world-wide.⁸ While the high-tech appeal of electronic computing appealed to many corporate executives, few had any idea how to effectively integrate this novel technology into their existing

⁷ See, for example, Thierry Bardini, *Bootstrapping: Douglas Englebart, Coevolution, and the Origins of Personal Computing* (Stanford, CA: Stanford University Press, 2000), 103.

⁸ Kenneth Flamm, *Creating the computer government, industry, and high technology* (Washington, D.C: Brookings Institute, 1988), 135.

social, political, and technological networks. The adoption of this expensive, unfamiliar, and often unreliable technology posed challenging problems, both social and technical, for even the most motivated corporate managers.

As the electronic computer moved out of the laboratory and into the marketplace, it became an increasingly valuable source of professional and institutional power and authority. By virtue of their control over this powerful new technology, computer specialists were often granted an unprecedented degree of independence and authority by high-level managers. The systems they developed often replaced, or at least substantially altered, the work of traditional white-collar employees. Departmental managers, not unsurprisingly, often resented the perceived impositions of the computer personnel, regarding them as threats to their position and status.⁹ They attempted to reassert control over operational decision-making by redefining programmers as narrow specialists and "mere technicians." The result was a highly-charged struggle over the proper place of the programmer in traditional occupational and professional hierarchies.

Over the course of the 1950s and 1960s, the identity of the computer programmer was continually invented and reinvented in response to a changing

⁹ T. Alexander, "Computers Can't Solve Everything," *Fortune* (October, 1969), 169; Thomas Whisler, "The Impact of Information Technology on Organizational Control," in *The Impact of Computers on Management*, Charles Myers (Ed.) (Cambridge, MA: MIT Press, 1967), 44.

social and technical environment. Corporate employers preferred programmers who were reliable, business-oriented, and who produced straightforward, easy-to-maintain software. Computer scientists privileged mathematical training and theoretical rigor. Others pursued professional agendas modeled after a wide range of traditional disciplines including medicine, engineering, and accounting.

All of the major technical innovations developed in this period reflected competing visions of what a programmer was and should be. The designers of the FORTRAN programming language, for example, had in mind a very different model of the programmer than did the COBOL committee. The former was designed to allow mathematically sophisticated users to program scientific algorithms; the latter for readability, ease-of maintenance, and machine independence. Embedded in each of these systems of technology and practice was a particular model of what the users of these inventions should look like. Was the idealized computer programmer a routinized laborer in a Taylorized “software factory” or a skilled, autonomous professional? Should programmers base their occupational identity on the model of the engineer/scientist or the certified public accountant? Should they emphasize craft technique or abstract knowledge? Did they need to be college educated or simply a vocational school graduate? Should they be male or female? The answers to each of these questions had significant implications for the role of electronic computing – and of

computing professionals – in modern corporate and academic organizations. It is no wonder, therefore, that they were not readily resolved in this, or for that matter any other, period in the history of computing.

Programmers as Clerical Workers

The first clear articulation of what a programmer was and should be was provided in the late 1940s by Herman Goldstine and John von Neumann in a series of volumes on "Planning and Coding of Problems for an Electronic Computing Instrument." These volumes, which served as the principal textbooks 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. These practices were themselves adapted from those used at the large manual computation projects at the nearby Aberdeen Proving Grounds. In these projects, the most senior women (by this point in time "computing" had become an almost exclusively feminine occupation), developed elaborate "plans of computation" that were carried out by their fellow human "computers." Since electronic computing was envisioned by the ENIAC developers as "nothing more than an automated form of hand

computation," it seemed natural that similar plans could be constructed for their electronic counterparts.¹⁰

Drawing on their experience with 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 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 often was male; the sixth task was to be carried out by "coders"—almost always female (at least in the ENIAC project). Coding was regarded as a "static" process by Goldstine and von Neumann, one that involved writing out steps of a computation in a form that could be read by the machine, such as punching cards, or in the case of ENIAC in plugging cables and setting switches. Thus there was a division of labor envisioned that gave the most skilled work to the

¹⁰ David Alan Grier, "The ENIAC, the verb 'to program' and the Emergence of Digital Computers," *Annals of the History of Computing* 18:1 (1996), 53.

high-status male scientists and the lowest skilled work to the low-status female coders.

The use of the word "coder" in this context is significant. "Coding" implied manual labor, and mechanical translation or rote transcription; "coders" obviously ranked low on the intellectual and professional hierarchy. It was not until later, that the now commonplace title of "programmer" was adopted. The verb "to program," with its military connotations of "to assemble" or "to organize," suggested a more thoughtful and system oriented activity.¹¹ Throughout the next several decades, however, 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 example, 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

¹¹ Ibid.

reducing the flow chart to the specific instructions, or coding, required by the UNIVAC to solve the problem.¹²

By differentiating between these two tasks, one clerical and the other analytical, the manual reinforced the Goldstine/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 the 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.

Over the course of the next decade, however, the line between coder and programmer became increasingly ambiguous. As the ENIAC managers and "coders" soon realized, for example, controlling the operation of an automatic computer was nothing like the process of hand computation, and the Moore School "girls" were therefore responsible for defining the first state-of-the-art methods of programming practice. Programming was a very imperfectly

¹² Sperry Rand Univac, "Introduction to Programming," *Programming for the UNIVAC, Part I* (typewritten manuscript, dated 11 June 1949). Hagley Archives, Box 372, Accession 1825. Emphasis is mine.

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 dynamic 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 to 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, "we learned to diagnose troubles as well as, if not better than, the engineers."¹³ The model of the programmer as "coder" – and programming as unskilled clerical labor – failed to correspond to the reality imposed by the technology. The professional and intellectual opportunities offered by a programming career became increasingly apparent to many of the male scientists and engineers on the ENIAC and other hardware projects. As a result, programming began to be "reinvented" as an legitimate and high-status discipline, at least within the confines of the computing community.

The "Black Art" of Programming

Although they continued to struggle to 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 1959 Price-Waterhouse report on

¹³ W. Barkley Fritz, "The Women of Eniac," *Annals of the History of Computing* 18, 3 (1996), 20.

“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.”¹⁴ In fact, the study’s authors suggested, “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 ... training 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.”¹⁵ Although Goldstine and von Neumann had envisioned a clear division of labor between “planners” and “coders,” in reality this boundary became increasingly indistinct. As Willis Ware indicated in a 1965 guest editorial in *Datamation*,

..it is clear that only a part - perhaps a small part, at that - of the programming process is involved with actually using a language for writing routines. Much of the programming process involves intellectual activity, mathematical investigation, discussions between people, etc. Very often, individuals who are trained as programmers actually do the early stages of the programming process but they may very well write no routines....One man who participated in the SAGE initial programming

¹⁴ 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.

¹⁵ *Ibid*, 89-90.

has estimated...that one half of the total programming man-hours...was for analysis and definition of the problem.¹⁶

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, that the idea that coding could be left to less experienced or lower-grade personnel was "erroneous," and that "the human element is crucial in programming."¹⁷

This new-found emphasis on programmer skill and creativity led to a subtle redefinition of the nature and causes of the software crisis. In the early part of the 1950s the solution to the "software turmoil" was believed to be the mass-production of new programmer trainees. By the beginning of the 1960s the problem had become the more subtle matter of programmer personnel selection: of "finding - and keeping - 'the right people'".¹⁸ Indeed, by the middle of the 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 often meant that a simple programming problem could quickly turn into a research excursion into algorithm theory and numerical

¹⁶ Willis Ware, "As I See It: A Guest Editorial," *Datamation* 11, 5 (1965), 27.

¹⁷ Conway, *Business experience with electronic computers*, 90.

¹⁸ Robert Gordon, "Personnel Selection," in Fred Gruenberger and Stanley Naftaly, eds., *Data Processing. Practically Speaking* (Los Angeles: Data Processing Digest, 1967), 87-88.

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...each problem required a unique beginning at square one, and the success of a program depended primarily on the programmer’s private techniques and inventions.”¹⁹

This reliance on individual creativity and clever “work-arounds” created the impression that programming was indeed more of an art than a science. This notion was reinforced by a series of aptitude tests and personality profiles that suggested that computer programmers, like chess masters or virtuoso musicians, were endowed with a uniquely creative ability. Great disparities were discovered between the productivity of individual programmers. One well-known IBM study determined that truly excellent programmers were twenty-six times more efficient than that produced by their merely average colleagues.²⁰

Despite the serious methodological flaws that compromised this particular study

¹⁹ Nick Metropolis, J. Howlett, and Gian-Carlo Rota, eds., *A history of computing in the twentieth century a collection of essays* (New York: Academic Press, 1980), p. 126.

²⁰ Hal Sackman, W.J. Erickson, and E.E. Grant, “Exploratory Experimental Studies Comparing Onling and Offline Programming Performance,” *Communications of the ACM* 11, 1 (1968).

(including a sample population of only twelve individuals), the 26:1 performance ratio quickly became part of the standard lore of the industry. Dr. E.E. David of Bell Telephone Laboratories spoke for many when he argued that large software projects could never be managed effectively, because "the vast range of programmer performance indicated earlier may mean that it is difficult to obtain better size-performance software using machine code written by an army of programmers of lesser average caliber."²¹

In this new model of the "art of programming," creative programmers were a highly valued commodity, and were often granted a great deal of personal authority and professional autonomy. One industry observer went so far as to argue that the "major managerial task is finding - and keeping - "the right people": with the right people, all problems vanish."²² Programmers were selected for their intellectual gifts and aptitudes, rather than their business knowledge or managerial savvy. "Look for those who like intellectual challenge rather than interpersonal relations or managerial decision-making. Look for the chess player, the solver of mathematical puzzles."²³ Skilled programmers were thought to be effectively irreplaceable, and were treated and compensated accordingly.

²¹ Peter Naur, Brian Randall, and J.N. Buxton, ed., *Software engineering Proceedings of the NATO conferences* (New York: Petrocelli/Carter, 1976), 33.

²² Gordon, "Personnel Selection," 88.

²³ Joseph O'Shields, "Selection of EDP Personnel," *Personnel Journal* 44, 9 (October 1965), 472.

II. The Labor Crisis in Programming

By the beginning of the 1960s developments occurred in both the technical and social environment of electronic computing that prompted new efforts to “reinvent” the programmer. Commercial manufacturers such as IBM began producing general purpose computers that were relatively reliable, affordable, and easy to program. They also provided the services and peripherals necessary to integrate the electronic computer into existing systems and processes. As the computer became more of a tool for business than a scientific instrument, the nature of its use – and of its primary user, the computer programmer – changed dramatically. The projects that business programmers worked on tended to be larger, more highly structured, and less mathematical than those involved in scientific computing. The needs of business demanded a whole new breed of programmers, and plenty of them.

As the market for commercial computers changed and expanded, the demand for qualified programmers increased accordingly. In 1962 the editors of *Datamation* declared that “first on anyone's checklist of professional problems is the manpower shortage of both trained and even untrained programmers, operators, logical designers and engineers in a variety of flavors.”²⁴ Five years later, “one of the prime areas of concern” to electronic data processing (EDP)

²⁴ *Datamation* Editorial, “Editor's Readout: A Long View of a Myopic Problem,” *Datamation* 8, 5 (1962), 21.

managers was still "the shortage of capable programmers," a shortage which had "profound implications, not only for the computer industry as it is now, but for how it can be in the future."²⁵ One widely quoted study from the late 1960s noted that although there were already 100,000 programmers working in the United States, there was an immediate need for at least 50,000 more.²⁶ "Competition for programmers has driven salaries up so fast," warned a contemporary article in *Fortune* magazine, "that programming has become probably the country's highest paid technological occupation...Even so, some companies can't find experienced programmers at any price."²⁷

Faced with an growing shortage of skilled programmers, employers were forced to lower their hiring standards. Many large software corporations in this period underwrote expensive internal training programs, "not because they want to do it, but because they have found it to be an absolute necessary adjunct to the operation of their business."²⁸ Vocational schools sprung up all over the country promising high salaries and dazzling career opportunities. An article in *Cosmopolitan Magazine* urged Helen Gurley Brown's "Cosmo Girls" to go out and become "computer girls" making \$15,000 a year as programmers. At one

²⁵ Richard Tanaka, "Fee or Free Software," *Datamation* 13, 10 (1967), 205-206.

²⁶ Quoted in Edward Markham, "Selecting a Private EDP School," *Datamation* 14, 5 (1968).

²⁷ Gene Bylinsky, "Help Wanted: 50,000 Programmers," *Fortune* (March, 1967), p. 141.

²⁸ James Saxon, "Programming Training: A Workable Approach," *Datamation* 9, 12 (1963), 48.

point the so-called “population problem” in software became so desperate that service bureaus in New York farmed out programming work to inmates at the nearby Sing-Sing prison, promising them permanent positions pending their release!²⁹

The influx of new programmer trainees and vocational school graduates into the software labor market failed to alleviate the acute shortage of programmers that plagued the industry, however. In fact, one study by the Association for Computing Machinery’s Special Interest Group on Computer Personnel Research (SIGCPR) warned of a growing *oversupply* of a certain undesirable species of software specialist: “The ranks of the computer world are being swelled by growing hordes of programmers, systems analysts and related personnel. Educational, performance and professional standards are virtually nonexistent and confusion grows rampant in selecting, training, and assigning people to do jobs.”³⁰ Employers and programmers alike were anxious to produce better standards for training and curriculum, but it was unclear to whom they should turn for guidance.

The obvious candidates for establishing standards for programming competency were the universities. Although computer science in the 1950s was not an established discipline, many of the larger research universities were

²⁹ News Brief, “First Programmer Class at Sing Sing Graduates,” *Datamation* 14, 6 (1968).

³⁰ H. Sackman, “Conference on Personnel Research,” *Datamation* 14, 7 (1968).

beginning to offer graduate training in computer-related specialties. But since academic computer scientists were struggling in this period to define a unique intellectual identity for their discipline, they focused on developing a theoretical basis for their discipline, rather than providing training in practical techniques.

As computing became more business-oriented the mismatch between university training and the needs of employers became even more apparent. Many corporations saw these university programs – most of which focused on formal logic and numerical analysis – as being increasingly out-of-touch with the needs of their business. As the computer scientist Richard Hamming pointed out in his 1968 Turing Award Lecture, "Their experience is that graduates in our programs seem to be mainly interested in playing games, making fancy programs that really do not work, writing trick programs, etc., and are unable to discipline their own efforts so that what they say they will do gets done on time and in practical form."³¹

Hamming was hardly the only member of the computing community to find fault with the increasingly theoretical focus of contemporary computer science. As early as 1958 a Bureau of Labor report on *The Effects of Electronic Computers on the Employment of Clerical Workers* had noted a growing sense of corporate

³¹ Richard Hamming, "One Man's View of Computer Science," chap. in *ACM Turing Award Lectures: The First Twenty Years, 1966-1985* (New York: ACM Press, 1987). The Turing Award was one of the first, and most prestigious, academic awards granted in computer science.

disillusionment with academic computer science: "Many employers no longer stress a strong background in mathematics for programming of business or other mass data if candidates can demonstrate an aptitude for the work. Companies have been filling most positions in this new occupation by selecting employees familiar with the subject matter and giving them training in programming work."³² Academic computer scientists sought to reinvent the programmer in the model of the research scientist; corporate employers resisted what they saw as "a sort of holier than thou academic intellectual sort of enterprise."³³ The tension between these competing visions of what a programmer should be only served to exacerbate the perceived shortage of "qualified" programmers.³⁴

In any case, the relatively small number of colleges and universities that did offer some form of practical programming experience were unable to provide trained programmers in anywhere near the quantities required by industry. As a result, aspiring software personnel often pursued alternative forms of vocational training. Some were recruited for in-house instruction programs provided by their employers. IBM provided programming training services to many of its clients. Others enrolled in the numerous private EDP training schools that began

³² William Paschell, *Automation and employment opportunities for office workers; a report on the effect of electronic computers on employment of clerical workers* (Bureau of Labor Statistics, 1958), 11.

³³ Rand Symposium, 1969. Charles Babbage Institute Archives, CBI 78, Box 3, Fld. 4.

³⁴ This seems to be as true in the 1990s as it was in the 1960s. See W. Gibbs, "Software's Chronic Crisis," *Scientific American* (September 1994), 86.

to appear in the mid-1960s. These schools were generally profit-oriented enterprises more interested in quantity than quality. For many of them the "only meaningful entrance requirements are a high school diploma, 18 years of age...and the ability to pay."³⁵ The more legitimate schools oriented their curricula towards the requirements of industry. The vocational schools suffered from many of the same problems that plagued the universities: a shortage of experienced instructors, the lack of established standards and curricula, and general uncertainty about what skills and aptitudes made for a qualified programmer. "Could you answer for me the question as to what in the eyes of industry constitutes a 'qualified' programmer?" pleaded one *Datamation* reader. "What education, experience, etc. are considered to satisfy the 'qualified' status?"³⁶ The problem was not only that the universities and vocational schools could not provide the type of educational experience that interested corporate employers; the real issue was that most employers were simply not at all sure what they were looking for.

III. Programmers as Professionals

Many software personnel were keenly aware of the crisis of labor and the tension it was producing in their industry and profession, as well as in their own

³⁵ Edward Markham, "EDP Schools - An Inside View," *Datamation* 14, 4 (1968), 22. The scandalous conditions of EDP schools were a frequent topic in the industry literature, and some companies imposed strict "no EDP school graduate" policies.

³⁶ John Callahan, "Letter to the editor," *Datamation* 7, 3 (1961), 7.

individual careers. Although computer specialists in general were appreciative of the short-term benefits of the software labor shortage (above average salaries and plentiful opportunities for occupational mobility), many believed that a continued crisis threatened the long-term stability and reputation of their industry and profession.

Concerns about the future of their occupation weighed heavily on the minds of many programmers. What was the appropriate career path for a software worker? "There is a tendency," suggested the ACM SIGCPR, "for programming to be a 'dead-end' profession for many individuals, who, no matter how good they are as programmers, will never make the transition into a supervisory slot. And, in too many instances this is the only road to advancement."³⁷ Whereas traditional engineers were often able (and in fact expected) to climb the corporate ladder into management positions, programmers were often denied this opportunity.³⁸ It was not clear to many corporate employers how the skills possessed by programmers would map onto the skills required for management. Part of the problem was the lack of a uniform programmer "profile." There was no "typical" programmer. The educational and occupational experience of programmers varied dramatically from individual to individual and workplace

³⁷ Datamation Report, "The Computer Personnel Research Group," *Datamation* 9, 1 (1963), 38.

³⁸ Louis Kaufman and Richard Smith, "Let's Get Computer Personnel on the Management Team," *Training and Development Journal* (December, 1966), 25-29.

to workplace. There was a vast gulf, for example, "between the systems programmers - who must tame the beast the computer designers build - and the applications programmers - who must then train the tamed beast to perform for the users."³⁹ It was possible for two programmers sitting side by side – and managed by the same data processing manager and hired by the same personnel administrator - to be working on entirely different types of projects each requiring distinctly different sets of skills and experience.

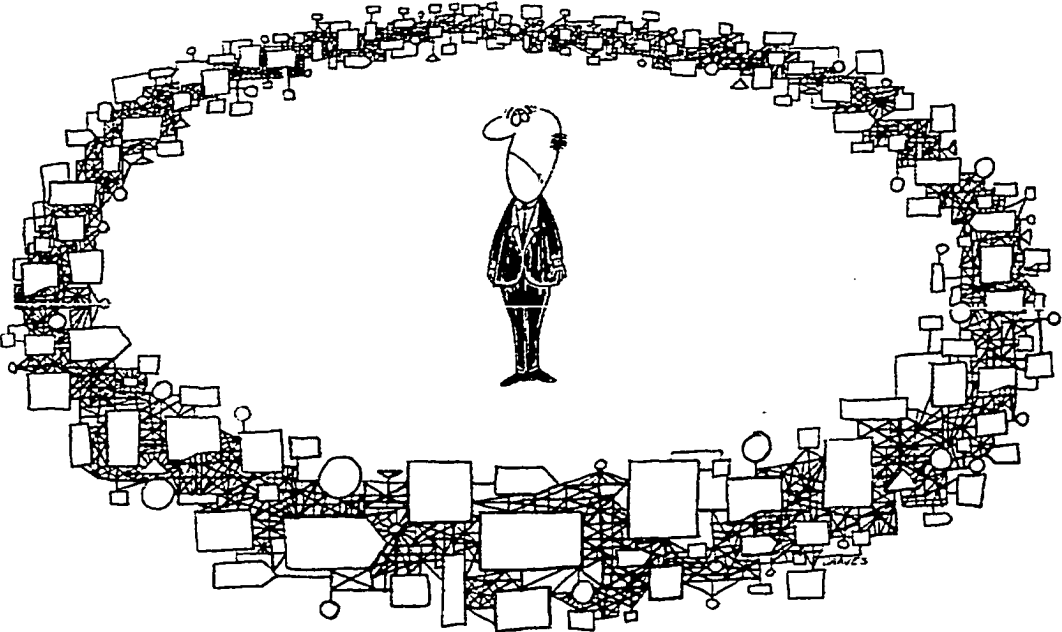
Many of the job advertisements in the late 1960s and early 1970s reflected the concerns that programmers had regarding their occupational future and longevity. "At Xerox, we look at programmers...and see managers."⁴⁰ "Working your way towards obsolescence? At MITRE professional growth is limited only by your ability."⁴¹ "Is your programming career in a closed loop? Create a loop exit for yourself at [the Bendix Corporation]."⁴² Like their counterparts in the 1990s, programmers in this period were worried about burning out by age forty. Corporations struggled to retain the employees in whom they had invested so much time and money in recruiting and training. The average annual

³⁹ Christopher Shaw, "Programming Schisms," *Datamation* 8, 9 (1962), 32.

⁴⁰ Xerox Corp., "At Xerox, we look at programmers and see managers (ad)," *Datamation* 14, 4 (1968).

⁴¹ Mitre Corp., "Are You Working Your Way Toward Obsolescence (ad)," *Datamation* 12, 6 (1966).

⁴² Bendix Computers, "Is Your Programming Career in a Closed Loop (ad)?" *Datamation* 8, 9 (1962).




is your programming career in a closed loop?

Have you programmed your career into a corner? Create a loop exit for yourself... apply for one of many openings in the area of **AUTOMATIC PROGRAMMING SYSTEMS, MONITORS and EXECUTIVE SYSTEMS, SCIENTIFIC APPLICATIONS and WRITERS** at Bendix Computer Division.

Bendix Computer has been a leading manufacturer of digital computing systems for 10 years... has long enjoyed a reputation for leadership. Growing acceptance of the Bendix G-20 and new military computer systems has created exceptional opportunities. The resulting combination of leadership and growth will help you out of that iterative loop... and into a new open-ended career.

Check it out for yourself. Call or write: Mr. William Keefer, Manager, Professional Staff Relations, Bendix Computer Division, 5630 Arbor Vitae Street, Los Angeles 45, California.

Bendix Computer Division



86

CIRCLE 83 ON READER CARD

DATA MATION

Figure 0.1: Advertisement for Bendix Computing, c. 1967.

turnover rate in the industry approached 25%, and at one edp installation turnover reached more than 10% *per month*. Poor management, long hours, and easy mobility "too often made an already mobile workforce absolutely liquid."⁴³ One problem was a labor market that provided plentiful opportunities for experienced developers: "Once a man is taught the skills, he may be hard to keep. Companies that use their computers for unromantic commercial purposes risk losing their programmers to more glamorous fields such as space exploration."⁴⁴ Managers attributed excessive employee turnover to the tight labor market, unscrupulous "body snatchers and other recruiting vultures,"⁴⁵ and the inherent fickleness of over-paid, prima donna programmers. However, a 1971 study of job satisfaction and computer specialists suggested that the majority of programmers valued the psychological benefits of their work – in terms of self-development, recognition, and responsibility – over its financial rewards.⁴⁶ What programmers disliked was the imposition of the "ultra-strict

⁴³ "EDP's Wailing Wall," *Datamation* 13, 7 (1967), 21. While this high level of turnover was no doubt disruptive, it hardly compares to that experienced in certain traditional manufacturing industries. During the Ford Motor Company's 'labor crisis' of 1914, annual employee turnover reached 380%. Turnover in the contemporary software industry still averages 19% (based on the 11th Annual Salary Survey, *Computerworld*, September 1, 1997).

⁴⁴ Bylinsky, "Help Wanted: 50,000 Programmers," 168.

⁴⁵ John Fike, "Vultures Indeed," *Datamation* 13, 5 (1967), 12.

⁴⁶ Enid Mumford, *Job Satisfaction: A study of computer specialists* (London: Longman Group Limited, 1972), 93.

industrial engineering and accounting type controls" aimed at limiting their professional autonomy.⁴⁷

Despite their concerns about the status and future of their profession, software developers in this period seemed to hold the position of power in the labor/management relationship. Programmers were able to vote with their feet on many crucial aspects of the terms and condition of their employment. Large government projects had difficulty attracting qualified programmers, in part because of salary considerations but mostly because they were seen as boring and rigid. As one contemporary organizational sociologist suggested, programmers appeared to be "one group of specialists whose work seems ideally structured to provide job satisfaction."⁴⁸ What is curious, however, is that programmers on the whole do not seem to have translated their monopoly of the software labor market into stable long-term career prospects. They were unable to establish many of the institutional structures and supports traditionally associated with the professions. Although starting salaries were high and individual programmers were able to move with relative ease horizontally throughout the industry, there were precious few opportunities for vertical

⁴⁷ Robert Head, "Controlling Programming Costs," *Datamation* 13, 7 (1967), 141.

⁴⁸ Mumford, *Job Satisfaction*, 175.

advancement.⁴⁹ Many programmers worried about becoming obsolete, and felt pressure to constantly upgrade their technical skills.⁵⁰ Most significantly, however, they faced the open hostility of managers. It was no secret that many corporate managers in this period were only too eager to impose new technologies and development methodologies that promised to eliminate what they saw as a dangerous dependency on programmer labor.⁵¹

IV. Programmers and Managers

By the end of the 1960s new development elevated the debate over programmer training and recruitment to the level of national crisis. In the first half of the decade innovations in transistor and integrated circuit technology had increased the memory size and processor speed of computers by a factor of ten, providing an effective performance improvement of almost a hundred. The falling cost of hardware allowed computers to be used for more and larger applications, which in turn required larger and more complex software. As the scale of software projects expanded, they became increasingly difficult to supervise and control. They also became much more expensive. Large software development projects acquired a reputation for being behind-schedule, over-

⁴⁹ James Jenks, "Starting Salaries of Engineers are Deceptively High," *Datamation* 13, 1 (1967), 13.

⁵⁰ *Datamation* Editorial, "Learning a Trade," *Datamation* 12, 10 (1966), 21.

⁵¹ Avner Porat and James Vaughan, "Computer Personnel: The New Theocracy - or Industrial Carpetbaggers," *Personnel Journal* 48, 6 (1968), 540-543.

budget, and bug-ridden. By 1968 the language of crisis dominated all discussions about the health and future of the industry.

The perception of that a software crisis was imminent focused unwelcome attention on programmers and their practices. Faced with rising software costs, and threatened by the unprecedented degree of autonomy that corporate executives seemed to grant to “computer people,” many corporate managers began to reevaluate their largely hands-off policies towards programmer management. Whereas in the 1950s computer programming was widely considered to be a uniquely creative activity – and therefore almost impossible to manage using conventional methods – by the end of the 1960s new perspectives on these problems began to appear in the industry literature. An influential report by the venerable management consulting firm of McKinsey & Company suggested that the real reason that most data processing installations were unprofitable is that “many otherwise effective top managements...have abdicated control to staff specialists - good technicians who have neither the operation experience to know the jobs that need doing nor the authority to get them done right.”⁵² The same qualities that had previously been thought essential indicators of programming ability, such as creativity and a mild degree of personal eccentricity, now began to be perceived as being merely

⁵² McKinsey & Company, “Unlocking the Computer's Profit Potential,” *Computers & Automation* (April 1969), 33.

unprofessional. As part of their rhetorical construction of the software crisis as a problem of programmer management, corporate managers often accused programmers of lacking professional standards and loyalties: "... too frequently these people [programmers], while exhibiting excellent technical skills, are non-professional in every other aspect of their work."⁵³ Many of the technological, managerial, and economic woes of the software industry became wrapped up in the crisis of software management.

There is no lack of evidence of pervasive management dissatisfaction with both programmers and the programming process. We have already described the enormous expenses incurred in the training, recruitment, and retention of software specialists. And since labor costs comprised almost the entire cost of any software development project, any increases in programmer efficiency or reductions in personnel immediately impacted the bottom line. In addition, software specialists had acquired a negative reputation in the eyes of corporate managers as being intractable and individualistic. According to one unflattering depiction, a programmer "doesn't want to be questioned, doesn't want to account accurately and in detail for his time...He doesn't want to be supervised...doesn't want to supervise. Says he wants responsibilities, but gripes

⁵³ Malcolm Gotterer, "The Impact of Professionalization Efforts on the Computer Manager," chap. in *Proceedings of 1971 ACM Annual Conference* (New York: Association for Computing Machinery, 1971), 368.

if they're assigned to him...The computer was acquired for him, not for operating results...It's not a pretty profile..."⁵⁴ A widely quoted psychological study that identified as a "striking characteristic of programmers...their disinterest in people," reinforced the managers' contention that programmers were insufficiently concerned with the larger interests of the company.⁵⁵ The apparent unwillingness of programmers to abandon the "black art of programming" for the "science" of software engineering was interpreted as a deliberate affront to managerial authority: "The technologists more closely identified with the digital computer have been the most arrogant in their willful disregard of the nature of the manager's job. These technicians have clothed themselves in the garb of the arcane wherever they could do so, thus alienating those whom they would serve."⁵⁶ The reinterpretation of the software crisis as a product of poor programming technique and insufficient managerial controls suggested that the software industry, like the more traditional manufacturing industries of the early twentieth century, was drastically in need of a managerial and technical overhaul.

⁵⁴ Datamation Editorial, "Checklist for Oblivion," *Datamation* 10, 9 (1964), 23.

⁵⁵ Perry and Cannon, "Vocational Interests of Computer Programmers."

⁵⁶ Datamation Editorial (1966), 22.

V. *Engineering a Solution*

There are numerous other ways in which the highly contested nature of the "invention" of the programmer was reflected in this period. The ongoing controversy about the content of computer science curricula, periodic scandals about the sharp practices of fly-by-night "EDP training schools," the seemingly endless debates about the relative merits of various programming languages - all point to a self-conscious concern on the part of programmers about their own ambiguous occupational identity. The great emphasis that contemporary observers of the software crisis place on issues of professional development argue for the historical significance of this struggle for status and autonomy.

In the late 1960s, in the wake of 1968 NATO Conference, a new model for situating the professional programmer was invented. Software engineering emerged as a compelling solution to the software crisis in part because it was flexible enough to appeal to a wide variety of computing practitioners. The ambiguity of concepts such as "professionalism," "engineering discipline," and "efficiency" allowed competing interests to participate in a shared discourse that nevertheless enabled them to pursue vastly different personal and professional agendas. Industry managers adopted a definition of "professionalism" that provided for educational and certification standards, a tightly disciplined workforce, and increased corporate loyalty. Computer manufacturers looked to

“engineering discipline” as means of countering charges of incompetence and cost-inefficiency. Academic computer scientists preferred a highly formalized approach to software engineering that was both intellectually respectable and theoretically rigorous. Working programmers tended to focus on the more personal aspects of professional accomplishment, including autonomy, status, and career longevity. The software engineering model seemed to offer something to everyone: standards, quality, academic respectability, status and autonomy.

By considering the software engineering movement in terms of a larger process of professional development, we can better understand why it succeeded (on a rhetorical level, at least, if not in actual practice) where other systems and methodologies have failed miserably. Thinking about the invention of a discipline as a series of interconnected social and political negotiations, rather than an isolated technical decision about the “one best way” to develop software components, provides an essential link between internal developments in information technology and their larger social and historical context.

Chapter One: Programming as Technology and Practice

However, in the present crisis, considerably increased attention is being given to the development of highly polished automatic programming languages and systems, especially along the lines of source language compilers. Very sophisticated assembly systems, debugging systems, operating systems, simulators and translators are being developed to meet an ever-mounting need for better tools.¹

"Trends in Programming Concepts," *Datamation* (1961)

Is a language really going to solve this problem? Do we really design languages for use by what we might call professional programmers or are we designing them for use by some sub-human species in order to get around training and having good programmers? Is a language ever going to get around the training and having good programmers?²

RAND Symposium on Programming Languages (1962)

I. Automatic Programmers

The first commercial electronic digital computers became available in the early 1950s. For a short period the focus of most manufacturers was on the development of innovative hardware. Most of the users of these early computers were large and technically sophisticated corporations and government agencies. In the middle of the decade, however, users and manufacturers alike became increasingly concerned with the rising cost of software development. By the beginning of the 1960s, the origins of "software turmoil" that would soon

¹ Ascher Opler, "Trends in Programming Concepts," *Datamation* 7, 1 (1961):13-14.

² RAND Symposium, "On Programming Languages, Part I," *Datamation* 8, 10 (1962): 29-30.

become a full-blown software crisis were readily apparent.³ As larger and more ambitious software projects were attempted, and the shortage of experienced programmers became more pronounced, industry managers began to look for ways to reduce costs by simplifying the programming process. A number of potential solutions were proposed: the use of aptitude tests and personnel profiles to identify the truly gifted “superprogrammers;” updated training standards and computer science curricula; and new management methods that would allow for the use of less-skilled laborers. The most popular and widely adopted solution, however, was the development of “automatic programming” technologies. These new tools promised to “eliminate the middleman” by allowing users to program their computers directly, without the need for expensive programming talent.⁴ The computer would program itself.

Over the course of the next several decades, hundreds of automatic programming systems and languages would be developed. Some of these were special purpose languages specifically designed for very limited problem domains. Others were academic exercises intended to explore new theories of computer science. Most of these systems, however, were designed explicitly as a means of addressing the burgeoning software crisis, either by eliminating the

³ See Daniel McCracken, “The Software Turmoil: Nine Predictions for '62,” *Datamation* 8, 1 (1962); Robert Patrick, “The Gap in Programming Support,” *Datamation* 7, 5 (1961).

⁴ RAND Symposium, “On Programming Languages, Part II,” *Datamation* 8, 11 (1962); Fred Gruenberger and Stanley Naftaly, eds., *Data Processing. Practically Speaking* (Los Angeles: Data Processing Digest, 1967), 85.

need for skilled programmers or as a "means of replacing the idiosyncratic 'artistic' ethos that has long governed software writing with a more efficient, cost-effective engineering mind-set."⁵ For many traditional managers in this period, the optimal solution to a troublesome manufacturing problem was automation. Automatic programmers would constitute a rational "software factory" that would reduce costs, improve efficiency, and eliminate quality problems.

Despite their associations with deskilling and routinization, automatic programming systems could also work to the benefit of occupational programmers and academic computer scientists. High-level programming promised to reduce the tedium associated with machine coding, and allowed programmers to focus on more system-oriented – and high status – tasks such as analysis and design. Language design and development served as a focus for productive theoretical research, and helped establish computer science as a legitimate academic discipline. And automatic programming systems never did succeed in eliminating the need for skilled programmers. In many ways, they contributed to the elevation of the profession, rather than the reverse, as was originally intended by some and feared by others.

⁵ David Morrison, "Software Crisis," *Defense* 21, 2 (1989), 72.

In order to understand why automatic programming languages were such an appealing solution to the software crisis, as well as why they apparently had so little effect on the outcome or severity of the crisis, it is essential to consider these languages as parts of larger social and technological systems. This chapter will describe the emergence of programming languages as a means of managing the complexity of the programming process. It will trace the development of several of the most prominent automatic programming languages, particularly FORTRAN and COBOL, and will situate these technologies in their appropriate historical context. Finally, it will explore the significance of these technologies as potential solutions to the ongoing software crisis of the late 1960s and early 1970s.

Assemblers, Compilers, and the Origins of the Sub-Routine

At the heart of all every “automatic programming” system was the notion that computer could be used, at least in certain limited situations, to generate the machine code required to run itself or other computers. This was an idea of great practical appeal: although programming was increasingly seen as legitimate and challenging intellectual activity, the actual coding of a program still involved tedious and painstaking clerical work. For example, the single instruction to “Add the short number in memory location 25,” when written out in the machine code understood by most computers, was stored as a binary number

such as 111000000000110010. This binary notation was obviously difficult for humans to remember and manipulate. As early as 1948, researchers at Cambridge University began working on a system to represent the same instruction in a more comprehensible format. The same instruction to "Add the short number in memory location 25," could be written out as A 25 S, where A stood for "add," 25 was the decimal address of the memory location, and S indicated that a "short" number was to be used.⁶ A Cambridge Ph.D. student named David Wheeler wrote a small program called "Initial Orders" that automatically translated this symbolic notation into the binary machine code required by the computer.

The focus of early attempts to develop automatic programming utilities was on eliminating the more unpleasant aspects of computer coding. Although in theory the actual process of programming was relatively straightforward, in practice it was quite difficult and time-consuming. A single error in any one of a thousand instructions could cause an entire program to fail. Simply getting a program to work properly often involved hours or days of laborious effort. As another Cambridge researcher, Maurice Wilkes would later vividly recall: "It had not occurred to me that there was going to be any difficulty about getting programs working. And it was with somewhat of a shock that I realized that for

⁶ This example comes from Martin Campbell-Kelly and William Aspray, *Computer: A History of the Information Machine* (New York: Basic Books, 1996), 182.

the rest of my life I was going to spend a good deal of my time finding mistakes that I had made in my programs.”⁷

These errors, or “bugs” as they soon came to be known, were often introduced in the process of transcribing or reusing code fragments. Wilkes and others soon realized that there was a great deal of code that was common to different programs – a set of instructions to calculate the sine function, for example. In addition to assigning his student David Wheeler to the development of the “Initial Orders” program, Wilkes set him to the task of assembling a library of such common “subroutines.” This method of reusing previously existing code became one of the most powerful techniques available for increasingly programmer efficiency. The publication in 1951 of the first textbook on the *Preparation of Programs for an Electronic Digital Computer* by Wilkes, Wheeler, and Cambridge colleague Stanley Gill helped disseminate these ideas throughout the nascent programming community.⁸

While Wilkes, Wheeler and Gill were refining their notions of a subroutine library, programmers in the United States were developing their own techniques for eliminating some of the tedium associated with coding. In 1949 John Mauchly of Univac created his “Short Order Code” for the BINAC computer.

⁷ H.S. Tropp, “ACM’s 20th Anniversary: 30 August 1967,” *Annals of the History of Computing* 9, 3 (1988), 269.

⁸ Maurice Wilkes, David Wheeler, and Stanley Gill, *Preparation of Programs for an Electronic Digital Computer* (Reading, MA: Addison-Wesley, 1951).

The Short Order Code allowed Mauchly to directly enter equations into the BINAC using a fairly conventional algebraic notation. The system did not actually produce program code, however: it was an interpretative system that merely called up predefined subroutines and displayed the result. Nevertheless, the Short Order Code represented a considerable improvement over the standard binary instruction set.

In 1951 Grace Hopper, another Univac employee, wrote the first automatic program "compiler." Although Hopper, like many other programmers, had benefited from the development of a subroutine library, she also perceived the limitations associated with its use. In order to be widely applicable, subroutines had to be written as generically as possible. They all started at line 0 and were numbered sequentially from there. They also used a standard set of register addresses. In order to make use of a subroutine, a programmer had to both copy the routine code exactly and make the necessary adjustments to the register addresses by adding an offset appropriate to the particular program at hand. And, as Hopper was later fond of asserting, programmers were both "lousy adders" and "lousy copyists!"⁹ The process of utilizing the subroutine code almost inevitably added to the number of errors that eventually had to be "debugged."

⁹ Richard Wexelblat, ed., *History of programming languages* (New York: Academic Press, 1981), 10.

To avoid the problems associated with manually copying and manipulating subroutine libraries, Hopper developed a system to automatically gather subroutine code and make the appropriate address adjustments. The system then “compiled” the subroutines into a complete machine program. Her A-0 compiler dramatically reduced the time required to put together a working application. In 1952 she extended the language to include a simpler mnemonic interface. For example, the mathematical statement $X + Y = Z$ could be written as `ADD 00X 00Y 00Z`. Multiplying Z by T to give W was `MUL 00Z 00T 00W`. The combination of an algebraic-language interface and a subroutine compiler became the basis for almost all modern programming languages. By the end of 1953 the A-2 compiler, as it was then known, was in use at the Army Map Service, the Air Controller, Livermore Laboratories, New York University, the Bureau of Ships, and the David Taylor Model Basin. Although it would take some time before automatic programming systems were universally adopted, by the middle of the 1950s the technology was well on its way to becoming an essential element of programming practice.

II. The Tower of Babel

Over the course of the next several decades, more than a thousand code assemblers, programming languages, and other automatic programming systems were developed in the United States alone. Understanding how these systems

were used, how and to whom they were marketed, and why there were so many of them is a crucial aspect of the history of the programming professions.

Automatic programming languages were the first, and perhaps the most popular, response to the burgeoning software crisis of the late 1950s and early 1960s. In many ways the entire history of computer programming – both social and technical – has been defined by the search for a “silver bullet” capable of slaying what Frederick Brooks famously referred to as the werewolf of “missed schedules, blown budgets, and flawed products.”¹⁰ The most obvious solution to what was often perceived to be a technical problem was, not surprisingly, the development of better technology.

Automatic programming languages were an appealing solution to the software crisis for a number of reasons. Computer manufacturers were interested in making software development as straightforward and inexpensive as possible. After all, as an early introduction to programming on the UNIVAC reminded pointedly, “the sale and acceptance of these machines is, to some extent, related to the ease with which they can be programmed. As a result, a great deal of research has been done, or is being done, to make programming

¹⁰ Frederick P. Brooks, “No Silver Bullet: Essence and Accidents of Software Engineering,” *IEEE Computer*, 20,4 (1987), 10-19.

simpler and more understandable..."¹¹ Advertisements for early automatic programming systems often made outrageous and unsubstantiated claims about the ability of their systems to simplify the programming process.¹² In many cases they were specifically marketed as a replacement for human programmers. Fred Gruenberger noted this tendency as early as 1962 in a widely disseminated transcript of a RAND Symposium on Programming Languages:

You know, I've never seen a hot dog language come out yet in the last 14 years - beginning with Mrs. Hopper's A-0 compiler...that didn't have tied to it the claim in its brochure that this one will eliminate all programmers. The last one we got was just three days ago from General Electric (making the same claim for the G-WIZ compiler) that this one will eliminate programmers. Managers can now do their own programming; engineers can do their own programming, etc. As always, the claim seems to be made that programmers are not needed anymore.¹³

Advertisements for these new automatic programming technologies, which appeared in management-oriented publications such as *Business Week* and the *Wall Street Journal* rather than *Datamation* or the *Communications of the ACM*, were clearly aimed at a pressing concern: the rising costs associated with finding and recruiting talented programming personnel. This perceived shortage of programmers was an issue that loomed large in the minds of many industry observers. "First on anyone's checklist of professional problems," declared a

¹¹ Sperry Rand Univac, *An Introduction to Programming the UNIVAC 1103A and 1105 Computing Systems* (1958) Hagley Archives, Box 372, Accession 1825.

¹² John Backus, quoted in Wexelblat, *History of programming languages*, 26.

¹³ RAND Symposium, "On Programming Languages: Part II," 25-26.

1962 *Datamation* editorial, "is the manpower shortage of both trained and even untrained programmers, operators, logical designers and engineers in a variety of flavors."¹⁴ The so-called "programmer problem" became an increasingly important feature of contemporary crisis rhetoric. "Competition for programmers has driven salaries up so fast," warned a 1967 article in *Fortune* magazine, "that programming has become probably the country's highest paid technological occupation ... Even so, some companies can't find experienced programmers at any price."¹⁵ Automatic programming systems held an obvious appeal for managers concerned with the rising costs of software development.

There is an interesting gender aspect to many of these developments in automatic programming technologies. As male programmers attempted to differentiate themselves from lower-status coders and key-punch operators, they carefully distanced themselves from any possible association with activities identified as "women's work." Many of the advertisements for automatic programming languages and other office automation technologies used women as a visual proxy for less expensive, more tractable labor.

Figure 1.2 shows one of a series of advertisements that presented an unambiguous appeal to gender associations: machines could not only replace their human female equivalents, but were an improvement on them. In its "Meet Susie Meyers" advertisements for its PL/1 programming language, the IBM Corporation asked its users an obviously rhetorical question: "Can a young girl with no previous programming experience find happiness handling both commercial and scientific applications, without resorting to an assembler

¹⁴ Editorial, "Editor's Readout: A Long View of a Myopic Problem," *Datamation* 8, 5 (1962), 21-22.

¹⁵ Gene Bylinsky, "Help Wanted: 50,000 Programmers," *Fortune* (March, 1967), 141.

language?" The answer, of course, was an enthusiastic "yes!" Although the advertisement promised a "brighter future for your programmers," (who would be free to "concentrate more on the job, less on the language") it also implied a low-cost solution to the labor crisis in software. The subtext of appeals like this was non-too-subtle: If pretty little Susie Meyers, with her spunky miniskirt and utter lack of programming experience, could develop software effectively in PL/1, so could just about anyone.



Our optical reader can do anything your keypunch operators do.

(Well, almost.)

It can't take maternity leave. Or suffer from morning sickness. Or complain of being tired all the time. But it can read. And gobble data at the rate of 2400 repetitions per hand-punched character a second. And compute while it reads. And reduce errors from a keypunch operator's cost of a thousand to an efficient one in a hundred thousand.

Our machine reads upper and lower case characters in intermixed, standard type fonts. It can handle intermixed sizes and weights of paper, including carbon-backed sheets.

An ordinary computer program tells our reader what to do... to add, subtract, edit, check or verify its results. Lets you forget format restrictions, leading and trailing zeros, skipped fields, and fixed record lengths. And our reader won't obsolete any of your present hardware because it speaks the same output language as your computer.

Our Electronic Retinal Computing Reader can replace all or almost all of your keypunch operators. At least that's what it is doing for American Airlines.

If you have a solving need application, it can do the same for you. Tell us your problem and we'll tell you how.

 **RECOGNITION EQUIPMENT** Incorporated

Figure 1.2: Advertisement from Datamation Magazine, c. 1968.

Despite the danger of automatic programming systems being used to eliminate their occupational monopoly, many programmers actively embraced this new technology. For those interesting in advancing the academic status of computer science, the design of programming languages provided an ideal forum for exploring the theoretical aspects of their discipline. More practically-oriented programmers saw programming languages as a means to distance themselves from the more tedious aspects of machine coding. Since “coding” had traditionally been an activity associated with low-status (and predominantly female) clerical workers, any tool that allowed programmers to focus more on design and analysis than on technical minutia was inherently desirable. As one review of the Report Program Generator (RPG) suggested, “Writing programs is fascinating and rewarding work but the functions of many programs are identical and rewriting these sections can not only become tiresome but also can get a programmer bogged down on one assignment and unable to move quickly into a new one.”¹⁶ Languages such as RPG would allow programmers to focus on more interesting and rewarding levels of analysis.

Whatever the motivation behind the development and adoption of any particular automatic programming system, by the middle of the 1950s there were a number of these systems being proposed by various manufacturers. Two of

¹⁶ Harry Leslie, “The Report Program Generator,” *Datamation* 13, 6 (1967), 26-28.

the most popular and significant were FORTRAN and COBOL, each developed by very different groups and intended for very different purposes.

FORTRAN

Although Grace Hopper's A-2 compiler was arguably the first modern automatic programming system, the first widely-used and -disseminated programming language was FORTRAN, developed in 1954-57 by a team of researchers at the IBM Corporation. As early as 1953 the mathematician and programmer John Backus had proposed to his IBM employers the development of a new, scientifically oriented programming language. This new system for mathematical FORMula TRANslation would be designed specifically for use with the soon-to-be-released IBM 704 scientific computer: it would "enable the IBM 704 to accept a concise formulation of a problem in terms of a mathematical notation and [would] produce automatically a high-speed 704 program for the solution of the problem." The result would be faster, more reliable, and less expensive software development. FORTRAN would not only "virtually eliminate programming and debugging," but would reduce operation time, double machine output, and provide a means of feasibly investigating complex mathematical models. In January 1954 Backus was given the go-ahead by his IBM superiors, and a completed FORTRAN compiler was released to all 704 installations in April 1957.

From the very beginning, development of the FORTRAN language was focused around a single overarching design objective: the creation of efficient machine code. Project leader John Backus was highly critical of existing automatic programming systems, which he saw as little more than mnemonic code assemblers or collections of subroutines. He also felt little regard for most contemporary human programmers, who he often derisively insisted on referring to as “coders.”¹⁷ In a 1980 article entitled “Programming in America in the 1950s - Some Personal Impressions,” Backus famously described programming in the 1950s as

... a black art, a private arcane matter involving only a small library of subroutines, and a primitive assembly program. Existing programs for similar problems were unreadable and hence could not be adapted to new uses. General programming principles were largely nonexistent. Thus each problem required a unique beginning at square one, and the success of a program depended primarily on the programmer's private techniques and inventions.¹⁸

A truly automatic programming language, believed Backus, would allow scientists and engineers to communicate directly with the computer, thus

¹⁷ When asked about the transformation of the “coder” into the programmer, Backus dismissively suggested that “it’s the same reason that janitors are now called “custodians.” “Programmer” was considered a higher class enterprise than “coder,” and things have a tendency to move in that direction.” This quote appears in Wexelblat, *History of programming languages*, 68.

¹⁸ Nick Metropolis, J. Howlett, and Gian-Carlo Rota, eds., *A history of computing in the twentieth century a collection of essays* (New York: Academic Press, 1980).

eliminating the need for inefficient and unreliable programmers.¹⁹ The only way that such a system would be widely adopted, however, was to ensure that the code it produced would be at least as efficient, in terms of size and performance, as that produced by its human counterparts.²⁰ Indeed, one of the primary objections raised against automatic programming languages in this period was their relative inefficiency: one of the higher-level languages used by SAGE developers produced programs that ran an order of magnitude slower than those hand-coded by a top-notch programmer.²¹ In an era when programming skill was considered to be a uniquely creative and innate ability, and when the state of contemporary hardware made performance considerations paramount, users were understandably skeptical of the value of automatically generated machine code. The focus of the FORTRAN developers was therefore on the construction of an efficient compiler, rather than on the design of the language.

In order to ensure that the object code produced by the FORTRAN compiler was as efficient as possible, several design compromises had to be made. FORTRAN was originally intended primarily for use on the IBM 704, and contained several device-specific instructions. Little thought was given to making FORTRAN machine-independent, and early implementations often

¹⁹ Jean Sammett, *Programming Languages: History and Fundamentals* (Englewood Cliffs, N.J: Prentice-Hall, 1969), 148.

²⁰ Wexelblat, *History of programming languages*, 28.

²¹ Sammett, *Programming Languages*, 144.

varied greatly from computer to computer, even those developed by the same manufacturer. The language was also designed solely for use in numerical computations, and was therefore difficult to use for applications requiring the manipulation of alphanumeric data. The first FORTRAN manual made clear this focus on mathematically problem-solving:

The FORTRAN language is intended to be capable of expressing any problem of numerical computation. In particular, it deals easily with problems containing large sets of formulae and many variables and it permits any variable to have up to three independent subscripts.

However, for problems in which machine words have a logical rather than numerical mean it is less satisfactory, and it may fail entirely to express some such problems. Nevertheless many logical operations not directly expressible in the FORTRAN language can be obtained by making use of provisions for incorporating library routines.²²

The power of the FORTRAN language for scientific computation can be clearly demonstrated by a simple real-world example. The mathematical expression described by the function

$$root = \frac{-(B/2) + \sqrt{(B/2)^2 - AC}}{A}$$

could be written in FORTRAN using the following syntax:

²² —, "The FORTRAN Automatic Coding System for the IBM 704 EDPM" (IBM Corporation, 1956). Cited in Sammett, *Programming Languages*, 150.

$$\text{ROOT} = \frac{-\text{B}/2.0 + \text{SQRTF}(\text{B}/2.0 ** 2 - \text{A} * \text{C})}{\text{A}}$$

Using such straightforward algorithmic expressions, a programmer could write extremely sophisticated programs with relatively little training and experience.²³

Although greeted initially with skepticism, the FORTRAN project was enormously successful in the long term. A report on FORTRAN usage written just one year after the first release of the language indicated that "over half [of the twenty-six 704 installations] used FORTRAN for more than half of their problems."²⁴ By the end of 1958, IBM produced FORTRAN systems for its 709 and 650 machines. As early as January 1961 Remington Rand Univac became the first non-IBM manufacturer to provide FORTRAN, and by 1963 a version of the FORTRAN compiler was available for almost every computer then in existence.²⁵ The language was updated substantially in 1958 and again in 1962. In 1962 FORTRAN became the first programming language to be standardized through the American Standards Association, which further established FORTRAN as an industry-wide standard.²⁶

²³ Example taken Saul Rosen, ed., *Programming Systems and Languages* (New York: McGraw-Hill, 1967), 30.

²⁴ John Backus, "Automatic Programming: Properties and Performance of FORTRAN Systems I and II," *Proceedings of Symposium on the Mechanization of the Thought Processes* (Middlesex, England: National Physical Laboratory Press, 1958).

²⁵ H. Oswald, "The Various FORTRANs," *Datamation* 10,8 (1964), 25-29; ———, "Survey of Programming Languages and Processors," *Communications of the ACM* 6,3 (1965), 93-99.

²⁶ USA Standard FORTRAN, United States of America Standards Institute, USAS X3.9-1966, New York, March 1966.

The academic community was an early and crucial supporter of FORTRAN, contributing directly to its growing popularity. The FORTRAN designers in general, and John Backus in particular, were regular participants in academic forums and conferences. Backus himself had delivered a paper at the seminal 1954 Symposium on Automatic Programming for Digital Computers hosted by the Office of Naval Research. One of his top priorities, after the compilation of the FORTRAN Programmer's Reference Manual (itself a model of scholarly elegance and simplicity), was to publish an academically-oriented article that would introduce the new language to the scientific community.²⁷

FORTRAN was appealing to scientists and other academics for a number of reasons. First of all, it was designed and developed by one of their own. John Backus spoke their language, published in their journals, and shared their disdain for coders and other "technicians." Secondly, FORTRAN was designed specifically to solve the kinds of problems that interested academics. Its use of algebraic expressions greatly simplified the process of defining mathematical problems in machine-readable syntax. Finally, and perhaps most significantly, FORTRAN provided them more direct access to the computer. Its introduction "caused a partial revolution in the way in which computer

²⁷ Backus, et al. "The FORTRAN automatic coding language," *Proceedings of the West Joint Computer Conference*, 1957. Backus would later become widely-known throughout the academic community as the co-developer of the Backus-Naur Form, the notational system used to describe most modern programming languages.

installations were run because it became not only possible but quite practical to have engineers, scientists, and other people actually programming their own problems without the intermediary of a professional programmer."²⁸ The use of FORTRAN actually became the centerpiece of an ongoing debate about "open" versus "closed" programming "shops." The closed shops allowed only professional programmers to have access to the computers; open shops made these machines directly available to their users.

The association of FORTRAN with scientific computing was a self-replicating phenomenon. Academics preferred FORTRAN to other languages because they believed it allowed them to do their work more effectively, and they therefore made FORTRAN the foundation of their computing curricula. Students learned the language in university courses and were therefore more effective at getting their work done in FORTRAN. A positive-feedback loop was established between FORTRAN and academia. A 1973 survey of more than 35,000 students taking college-level computing courses revealed that seventy percent were learning to program using FORTRAN. The next most widely used alternative, BASIC, was used by only thirteen percent, and less than three percent were exposed to business-oriented languages such as COBOL.²⁹

²⁸ Sammett, *Programming Languages*, 149.

²⁹ Daniel McCracken, "Is There FORTRAN In Your Future?," *Datamation* 19, 5 (1973), 236-237.

Throughout the 1960s and 1970s, FORTRAN was clearly the dominant language of scientific computation.

COBOL

On April 8, 1959, a group of computer manufacturers, users, and academics met at the University of Pennsylvania's Computing Center to discuss a proposal to develop "the specifications for a common business language for automatic digital computers."³⁰ The goal was to develop a programming language specifically aimed at the needs of the business data processing community. This new language would rely on simple English-like commands, would be easier to use and to understand than existing scientific languages, and would provide machine-independent compatibility: that is, the same program could be run on a wide variety of hardware with very little modification.

Although this proposal originated in the ElectroData Division of the Burroughs Corporation, from the very beginning it had broad industrial and governmental support. The Director of Data Systems for the Department of Defense readily agreed to sponsor a formal meeting on the proposal, and his enthusiastic support indicates a widespread contemporary interest in business-oriented programming:

³⁰ I.E. Block, *Report on Meeting Held at University of Pennsylvania Computing Center*, April 9, 1959.

The Department of Defense was pleased to undertake this project: in fact, we were embarrassed that the idea for such a common language had not had its origin in Defense since we would benefit so greatly from such a project, and at the same time one of the Air Force commands was in the process of developing one of the business languages AIMACO.³¹

The first meeting to discuss a common business language (CBL) was held at the Pentagon on May 28-29, 1959. Attending the meeting were fifteen officials from seven government organizations; fifteen representatives of the major computer manufacturers (including Burroughs, GE, Honeywell, IBM, NCR, Phillips, RAC, Remington-Rand Univac, Sylvania, and ICT); and eleven users and consultants (significantly, only one member of this last group was from a university). Despite the diversity of the participants, the meeting produced both consensus and a tangible plan of action. The group not only decided that CBL was necessary and desirable, but agreed on its basic characteristics: a problem-oriented, English-like syntax; a focus on ease of use rather than power or performance; and a machine-independent design. Three committees were established, under the auspices of a single Executive Committee of the Conference on Data Systems Languages (CODASYL), to suggest short-term, intermediate, and long-range solutions, respectively. As it turned out, it was the short-term committee that produced the most lasting and influential proposals.

³¹ Charles Phillips, *Report from the Committee on Data Systems Languages* (Oral presentation to the Association for Computing Machinery, Boston, MA, September 1, 1959). Cited in Wexelblat, *History of programming languages*, 200.

The original purpose of the Short-Range Committee was to evaluate the strengths and weaknesses of existing automatic compilers and to recommend a “short term composite approach (good for the next year or two) to a common business language for programming digital computers.”³² There were three existing compiler systems that the committee was particularly interested in considering: FLOW-MATIC, which had been developed for Remington-Rand Univac by Grace Hopper (as an outgrowth of her A-series algebraic and B-series business compilers), and which was actually in use by customers at the time; AIMACO, developed for the Air Force Air Material Command; and COMTRAN (soon to be renamed the Commercial Translator), a proposed IBM product that existed only as a specification document. Other manufacturers such as Sylvania and RCA were also working on the development of similar languages. Indeed, one of the primary goals of the Short-Range Committee was to “nip these projects in the bud” and to provide incentives for manufacturers to standardize on the CBL rather than to pursue their own independent agendas.³³ At the first meeting of one of the Short-Range Committee task groups, for example, most of

³² Charles Phillips, *Minutes, Meeting of the Executive Committee of the Conference on Data Systems Languages* (1959). Cited in Wexelblatt, *History of programming languages*, 202.

³³ Other languages considered were Autocoder III, SURGE, Fortran, RCA 501 Assembler, Report Generator, and APG-1 (Wexelblatt, 204).

the time was spent getting statements of commitment from the various manufacturers.³⁴

From the very beginning, the process of designing the CBL was characterized by a spirit of pragmatism and compromise. The Short-Range Committee, often referred to as the PDQ (pretty darn quick) Committee, took seriously their charge to work quickly to produce an interim solution. Remarkably enough, less than three months later the committee had produced a nearly-complete draft of a proposed CBL specification. In doing so the CBL designers borrowed freely from models provided by Remington-Rand Univac's FLOW-MATIC language and the IBM Commercial Translator. In a September report to the Executive Committee of CODASYL, the Short-Range Committee requested permission to continue development on the CBL specification, to be completed by December 1, 1959. Shortly thereafter, the name COBOL (an acronym for **CO**mmun **B**usiness **O**riented **L**anguage) was formally adopted. Working around the clock for the next several months, the PDQ group was able to produce their finished report just in time for their December deadline. It was approved by the CODASYL, and in January 1960 the official COBOL-60 specification was released by the Government Printing Office.

³⁴ *Task Group of Statement Language*, July 22, 1959.

The structure of the COBOL-60 specification reveals its mixed origins and commercial orientation. Although from the very beginning the COBOL designers were concerned with “business data processing,” there was never any attempt to provide a real definition of that phrase.³⁵ It was clearly intended that the language could be used by novice programmers and read by managers. For example, an instruction to compute an employee’s overtime pay might be written as follows:

```
MULTIPLY NUMBER-OVTIME-HRS BY OVTIME-PAY-RATE GIVING  
OVTIME- PAY-TOTAL
```

It was felt that this readability would result from the use of English language instructions, but no formal criteria or tests for readability were provided. In many cases compromises were made that allowed for conflicting interpretations of what made for “readable” computer code. For example, arithmetic formulas could either be written using a combination of arithmetic verbs – i.e. ADD, SUBTRACT, MULTIPLY, or DIVIDE – or as symbolic formulas. The use of arithmetic verbs was adapted directly from the FLOW-MATIC language, and reflected the belief that business data processing users could not – and should not – be forced to use formulas. The capability to write symbolic formulas was included (after much contentious debate) as a means of providing power and flexibility to more mathematically sophisticated programmers.

³⁵ Jean Sammett, quoted in Wexelblatt, *History of programming languages*, 219.

However, such traditional mathematical functions such as SINE and COSINE were deliberately excluded as being unnecessary to business data processing applications.

Another concession to the objective of readability was the inclusion of extraneous "noise words." These were words or phrases that were allowable but not necessary: for example, in the statement

READ *file1* RECORD INTO *variable1* AT END *goto procedure2*

the words RECORD and AT are syntactically superfluous. The statement would be equally valid written as

READ *file1* INTO *variable1* END *goto procedure2*.

The inclusion of the noise words RECORD and AT was perceived by the designers to enhance readability. Users had the option of including or excluding them according to individual preference or corporate policy.

In addition to designing COBOL to be "English-like" and readable, the committee was careful to make it as machine-independent as possible. Most contemporary programming systems were tied to a specific processor or product line. If the user wanted to replace or upgrade their computer, or switch to machines from a different manufacturer, they had to completely rewrite their software from scratch, typically an expensive, risky, and time-consuming operation. Users often became bound to outdated and inefficient hardware

systems simply because of the enormous costs associated with upgrading their software applications. This was especially true for commercial data processing operations, where computers were generally embedded in large, complex systems of people, procedures, and technology. A truly machine-independent language would allow corporations to reuse application code and thereby reduce programming and maintenance costs. It would also allow manufacturers to sell or lease more of their most recent (and profitable) computers.

The COBOL language was deliberately organized in such a way as to encourage portability from one machine to another. Every element of a COBOL application was assigned to one of four functional divisions: IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE. The IDENTIFICATION division provided a high-level description of the program, including its name, author, and creation date. The ENVIRONMENT division contained information about the specific hardware on which the program was to be compiled and run. The DATA division described the file and record layout of the data used or created by the rest of application. The PROCEDURE division included the algorithms and procedures that the user wished the computer to follow. Ideally, this rigid separation of functional divisions would allow a user to take a deck of cards from one machine to another without making significant alterations to anything but the ENVIRONMENT description. In reality, this degree of portability was

almost impossible to achieve in real-world applications in which performance was a primary consideration. For example, the most efficient method of laying out a file for a 24-bit computer was not necessarily optimal for a 36-bit machine. Nevertheless, machine independence "was a major, if not *the* major" design objective of the Short-Range Committee.³⁶ Achieving this objective proved difficult both technically and politically, and greatly influenced both the design of the COBOL specification and its subsequent reception within the computing community.

One of the greatest obstacles to achieving machine independence was the computer manufacturers themselves. Each manufacturer wanted to make sure that COBOL included only features that would run efficiently on their devices. For example, a number of users wanted the language to include the ability to read a file in reverse order. For those machines that had a basic machine command to read a tape backwards this was an easy feature to implement. Even those computers without this explicit capability could achieve the same functionality by backing the tape up two records and then reading forward one. Although this potential READ REVERSE command could therefore be logically implemented by everyone, it significantly penalized those devices without the basic machine capability. It was therefore not included in the final specification.

³⁶ Jean Sammett, in Wexelblatt, *History of programming languages*, 234.

There were other compromises that were made for the sake of machine independence. In order to maintain compatibility among different machines with different arithmetic capabilities, eighteen decimal digits were chosen as the maximum degree of precision supported. This particular degree of precision was chosen “for the simple reason that it was *disadvantageous* to every computer thought to be a potential candidate for having a COBOL compiler.”³⁷ No particular manufacturer would therefore have an inherent advantage in terms of performance. In a similar manner, provisions were made for the use of binary computers, despite the fact that such machines were generally not considered appropriate for business data processing. The decision to allow only a limited character set in statement definitions – using only those characters that were physically available on almost all data-entry machines – was a self-imposed constraint that had “an enormous influence on the syntax of the language” but was nevertheless considered essential to widespread industry adoption. The use of such a minimal character set also prevented the designers from using the sophisticated reference language techniques that had so enamored theoretical computer scientists of the ALGOL 58 specification.

This dedication to the ideal of portability set the Short-Term Committee at odds with some of their fellow members of CODASYL. In October 1959 the

³⁷ Wexelblat, *History of programming languages*, 231.

Intermediate-Range Committee passed a motion declaring that the FACT programming language – recently released by the Honeywell corporation - was a better language than that produced by the Short-Range Committee and should therefore form the basis of the CBL.³⁸ Although many members of the Short-Range Committee agreed that FACT was indeed a technically advanced and superior language, they rejected any solution that was tied to any particular manufacturer. In order to ensure that the CBL would be a truly *common* business language, elegance and efficiency had to be compromised for the sake of readability and machine independence. Despite the opposition of the Intermediate-Range Committee (and the Honeywell representatives), the Executive Committee of the CODASYL eventually agreed with the design priorities advocated by the PDQ group.

The first COBOL compilers were developed in 1960 by Remington-Rand Univac and RCA. In December of that year the two companies hosted a dramatic demonstration of the cross-platform compatibility of their individual compilers: the same COBOL program, with only the ENVIRONMENT division needing to be modified, was run successfully on machines from both manufacturers. Although this was a compelling demonstration of COBOL's potential, other manufacturers were slow to develop their own COBOL compilers. Honeywell

³⁸ *Minutes of the IRTF*, 1959.

and IBM, for example, were loathe to abandon their own independent business languages. Honeywell's FACT had been widely praised for its technical excellence, and the IBM Commercial Translator already had an established customer base.³⁹ By the end of 1960, however, the United States military had put the full weight of its prestige and purchasing power behind COBOL. The Department of Defense announced that it would not lease or purchase any new computer without a COBOL compiler unless its manufacturer could demonstrate that its performance would not be enhanced by the availability of COBOL.⁴⁰ No manufacturer ever attempted such a demonstration, and within a year COBOL was well on its way toward becoming an industry standard.

It is difficult to establish empirically how widely COBOL was adopted, but anecdotal evidence suggests that it is by far the most popular and widely used computer language *ever*.⁴¹ A recent study undertaken in response to the perceived Y2K crisis suggests that there are 70 billion lines of COBOL code currently in operation in the United States alone. Despite its obvious popularity, however, from the very beginning COBOL has faced severe criticism and opposition, particularly from within the computer science community. One 1977

³⁹ Robert Bemer, "A view on the history of COBOL," *Honeywell Computer Journal* 5, 3 (1971).

⁴⁰ Campbell-Kelly, Aspray, *Computer*, 192.

⁴¹ See Stanley Naftaly, "How to Pick a Programming Language," in Fred Gruenberger and Stanley Naftaly, eds., *Data Processing. Practically Speaking* (Los Angeles: Data Processing Digest, 1967), 98; "What's happening with COBOL," *Business Automation*, April 1968.

programming language textbook judged COBOL's programming features as fair and its implementation dependent features as poor and its overall writing as fair to poor. It also noted its "tortuously poor compactness and poor uniformity."⁴² The noted computer scientist Edsger Dijkstra wrote that "COBOL cripples the mind," and another of his colleagues called it "terrible" and "ugly."⁴³ Several notable textbooks on programming languages from the 1980s did not even include COBOL in the index.

There are a number of reasons why computer scientists have been so harsh in their evaluation of COBOL. Some of these objections are technical in nature, but most are aesthetic, historical, or political. Most of the technical criticisms have to do with COBOL's verbosity, its inclusion of superfluous "noise words," and its lack of certain features (such as protected module variables). Although many of these shortcomings were addressed in subsequent versions of the COBOL specification, the academic world continued to vilify the language. In a 1985 article on "The Relationship Between COBOL and Computer Science," the computer scientist Ben Shneiderman identified several explanations for this continued hostility: First of all, no academics were asked to participate on the initial design team. In fact, the COBOL developers apparently had little interest

⁴² Allen Tucker, *Programming Languages* (Reading, MA: Addison-Wesley, 1977).

⁴³ Cited in Ben Shneiderman, "The Relationship Between COBOL and Computer Science," *Annals of the History of Computing* 7, 4 (1985), 350.

in the academic or scientific aspects of their work. All of the articles included in a May 1962 Communications of the ACM issue devoted to COBOL were written by industry or government practitioners. Only four of the thirteen included even the most basic references to previous and related work: the lack of academic sensibilities was immediately apparent. Also noticeably lacking was any reference to the recently developed Backus-Naur Form notation that had already become popular as a meta-language for describing other programming languages. No attempt was made to produce a textbook describing the conceptual foundations of COBOL until 1963. Most significant, however, was the sense that the problem domain addressed by the COBOL designers, i.e. business data processing, was not theoretically sophisticated or interesting. One 1974 programming language textbook described COBOL as having “an orientation toward business data processing ... in which the problems are ... relatively simple algorithms couple with high-volume input-output (e.g. computing the payroll for a large organization.” Although this dismissive account hardly captures the complexities of many large-scale business applications, it does appear to accurately represent a prevailing attitude among computer scientists. COBOL was considered a “trade-school” language rather than a serious intellectual accomplishment.⁴⁴

⁴⁴ Shneiderman, “The Relationship Between COBOL and Computer Science,” 351.

Despite these objections, COBOL has proven remarkably successful. Certainly the support of the United States government had a great deal to do with its initial widespread adoption. But COBOL was attractive to users – business corporations in particular- for other reasons as well. The belief that “English-like” COBOL code could be read and understood by non-programmers was appealing to traditional managers who were worried about the dangers of “letting the 'computer boys' take over.”⁴⁵ It was also hoped that COBOL would achieve true machine independence - arguably the holy grail of language designers – and, of all its competitors, COBOL did perhaps come closest to achieving this ideal. Although COBOL has often been derided by critics as the inelegant result of “design by committee,” the broad inclusiveness of the CODASYL helped ensure that no one manufacturer’s hardware would be favored. Committee control over the language specification also prevented splintering: whereas numerous competing dialects of FORTRAN and ALGOL were developed, COBOL implementations remained relatively homogenous. The CODASYL structure also provided a mechanism for ongoing language maintenance with periodic “official” updates and releases.

⁴⁵ John Golda, “The Effects of Computer Technology on the Traditional Role of Management,” (MBA thesis, Wharton School, University of Pennsylvania, 1965), 34; Robert Gordon, “Personnel Selection” in Fred Gruenberger and Stanley Naftaly, eds., *Data Processing. Practically Speaking* (Los Angeles: Data Processing Digest, 1967), 85.

ALGOL, Pascal, ADA and Beyond ...

Although FORTRAN and COBOL were by far the most popular programming languages developed in the United States during this period, they were by no means the only ones to appear. Jean Sammet, editor of one of the first comprehensive treatments of the history of programming languages, has estimated that by 1981 there were a least one thousand programming languages in use nationwide. It would be impossible to even enumerate, much less describe, the history and development of each of these languages. Figure 1.3 contains a “genealogical” listing of some of the more widely-used programming languages developed prior to 1970. This section will focus on a few of the more historically significant alternatives to FORTRAN and COBOL.

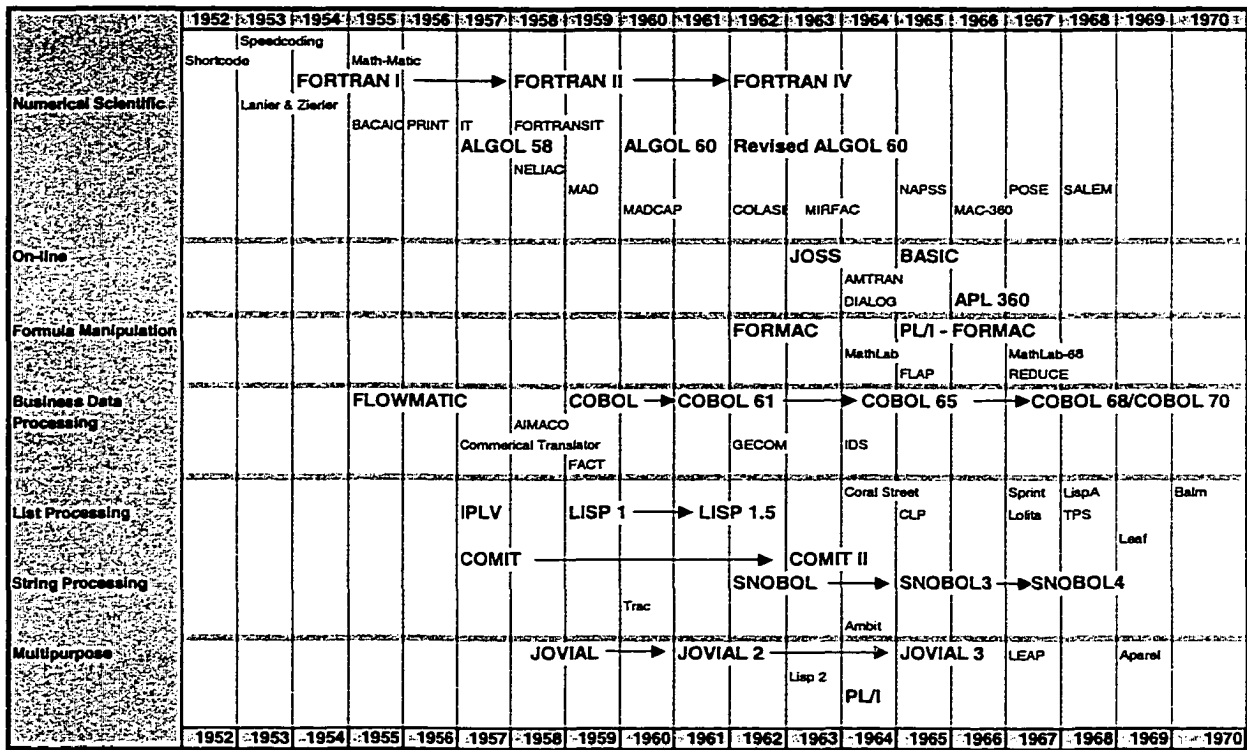


Figure 1.3: Genealogy of Programming Languages, 1952-1970.

More than a year before the Executive Committee of CODASYL convened to discuss the need for a common business-oriented programming language, an ad hoc committee of users, academics, and federal officials met to study the possibility of creating a universal programming language. This committee, which was brought together under the auspices of the Association for Computing Machinery (ACM), could not have been more different from the group organized by CODASYL. Whereas the fifteen member Executive Committee had contained only one university representative, the identically-sized ACM-sponsored committee was dominated by academics. At their first meeting, this committee decided to follow the model of FORTRAN in designing

an algebraic language. FORTRAN itself was not acceptable because of its association with IBM.

The ACM “universal language” project soon expanded into an international initiative. Europeans in particular were deeply interested in a language that would both transcend political boundaries and help avoid the domination of Europe by the IBM Corporation. During an eight-day meeting in Zurich, Switzerland, a rough specification for the new International Algebraic Language (IAL) was hashed out. Actually, three distinct versions of the IAL were created: reference, publication, and hardware. The reference language was the abstract representation of the language as envisioned by the Zurich committee. The publication and hardware languages would be isomorphic implementations of the abstract reference language. Since these specific implementations required careful attention to such messy details as character sets and delimiters (decimal points being standard in the United States and commas in Europe), they were left for a later and unspecified date. The reference language was released in 1958 under the more popular and less pretentious name ALGOL (ALGO`r`ithmic Language).

In many ways ALGOL was a remarkable achievement in the nascent discipline of computer science. ALGOL 58 was something of a work in progress; ALGOL 60, which was released shortly thereafter, is widely considered to be a

model of completeness and clarity. The ALGOL 60 version of the language was described using an elegant meta-language known as Backus Normal Form (BNF), developed specifically for that purpose. Backus Normal Form, which resembles the notation used by linguists and logicians to describe formal languages, has since become the standard technique for representing programming languages. The elegant sophistication of ALGOL 60 report appealed particularly to computer scientists. In the words of one well-respected admirer,

The language proved to be an object of stunning beauty ... Nicely organized, tantalizingly incomplete, slightly ambiguous, difficult to read, consistent in format, and brief, it was a perfect canvas for a language that possessed those same properties. Like the Bible, it was meant not merely to be read, but interpreted.⁴⁶

ALGOL 60 soon became the standard by which all subsequent language developments were measured and evaluated.

Despite its intellectual appeal and the enthusiasm in which it was greeted in academic and European circles, ALGOL was never widely adopted in the United States. Although many Americans recognized that ALGOL was an elegant synthesis, most saw language design as just one step in a lengthy process leading to language acceptance and use. In addition, in the United States there were already several strong competitors currently in development. IBM and its

⁴⁶ Alan Perlis, quoted in Wexelblat, *History of programming languages*, 60.

influential users group SHARE supported FORTRAN, and business data processors preferred COBOL. Even those installations that preferred ALGOL often used it only as a starting point for further development, more “as a rich set of guidelines for a language than a standard to be adhered to.”⁴⁷ Numerous dialects or spin-off languages emerged, most significantly JOVIAL, MAD, and NELIAC, developed at the System Development Corporation, the University of Michigan, and the Naval Electronics Laboratory, respectively. Although these languages benefited from ALGOL, they only detracted from its efforts to emerge as a standard. With a few noticeable exceptions – the ACM continued to use it as the language of choice in its publications, for example – ALGOL was generally regarded in the United States as an intellectual curiosity rather than a functional programming language.

The real question of historical interest, of course, is not so much why specific individual programming languages were created, but rather why *so many*. In the late 1940s and early 1950s there was no real programming community *per se*, only particular projects being developed at various institutions. Each project necessarily developed its own techniques for facilitating programming. By the middle of the 1950s, however, there were established mechanisms for communicating new research and development, and there were deliberate

⁴⁷ Perlis in Wexelblat, *History of programming languages*, 82.

attempts to promote industry-wide programming standards. Nevertheless, there were literally hundreds of languages developed in the decades of the 1950s and 1960s. FORTRAN and COBOL have emerged as important standards in the scientific and business communities, respectively, and yet new languages continued – and still continue – to be created. What can explain this curious Cambrian explosion in the evolutionary history of programming?

Some of the many divergent species of programming languages can be explained by looking at their functional characteristics. Although general purpose languages such as FORTRAN and COBOL were suitable for a wide variety of problem domains, certain applications required more specialized functions to perform most efficiently. The General Purpose Simulation System (GPSS) was designed specifically for the simulation of system elements in discrete numerical analysis, for example. APT was commissioned by the Aircraft Industries Association and the United States Air Force to be used primarily to control automatic milling machines. Other languages were designed not as much for specialized problem domains as for particular pedagogical purposes - in the case of BASIC, for example, the teaching of basic computer literacy. Some languages were known for their fast compilation times, others for the efficiency of their object code. Individual manufacturers produced languages that were optimized for their own hardware, or as part of a larger marketing strategy.

There were also less obviously utilitarian reasons for developing new programming languages, however. Many common objections raised against existing languages were more matters of style rather than substance. The rationale given for creating a new language often boiled down to a declaration that “this new language will be easier to use or better to read or write than any of its predecessors.” Since there were generally no standards for what was meant by “easier to use or better to read or write,” such declarations can only be considered statements of personal preference. As Jean Sammet has suggested, although lengthy arguments have been advanced on all sides of the major programming language controversies, “in the last analysis it almost always boils down to a question of personal style or taste.”⁴⁸

For the more academically-oriented programmers, designing a new language was a relatively easy way to attract grant money and publish articles. There have been numerous languages that have been rigorously described but never implemented. They served only to prove a theoretical point or to advance an individual’s career. In addition, many in the academic community seemed to be afflicted with what has often been referred to as the NIH (“not invented here”) syndrome: any language or technology that was designed by someone else could not possibly be as good as one that you invented yourself, and so a new version

⁴⁸ Jean Sammett, “Programming Languages History,” *Annals of the History of Computing* 13, 1 (1991), 49.

needed to be created to fill some ostensible personal or functional need. As Herbert Grosch lamented in 1961, filling these needs was personally satisfying but ultimately self-serving and divisive:

Pride shades easily into purism, the sin of the mathematicians. To be the leading authority, indeed the only authority, on ALGOL 61B mod 12, the version that permits black letter as well as Hebrew subscripts, is a satisfying thing indeed, and many of us have constructed comfortable private universes to explore.⁴⁹

One final and closely related reason for the proliferation of programming languages is that designing programming languages was fun. The adoption of meta-languages and the Backus Normal Form allowed for the rapid development and implementation of creative new languages and dialects. If programming was enjoyable, even more so was language design!⁵⁰

III. No Silver Bullet

In 1987 the computer scientist Frederick Brooks published an essay describing the major developments in automatic programming technologies that had occurred over the past several decades. As an accomplished academic and experienced industry manager, Brooks was a respected figure within the programming community. Using characteristically vivid language, his "No

⁴⁹ Herb Grosch, "Software in Sickness and Health," *Datamation* 7, 7 (1961), 32-33.

⁵⁰ *Ibid*, 33.

Silver Bullet: Essence and Accidents of Software Engineering” reflected upon the inability of these technologies to bring an end to the ongoing software crisis:

Of all the monsters that fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, one seeks bullets of silver that can magically lay them to rest.

The familiar software project, at least as seen by the nontechnical manager, has something of this character; it is usually innocent and straightforward, but is capable of becoming a monster of missed schedules, blown budgets, and flawed products. So we hear desperate cries for a --silver bullet--something to make software costs drop as rapidly as computer hardware costs do.

But, as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity.⁵¹

Brook’s article provoked an immediate reaction, both positive and negative.

The object-oriented programming (OOP) advocate Brad Cox insisted, for example, in his aptly titled “There is a Silver Bullet,” that new techniques in OOP promised to bring about “a software industrial revolution based on reusable and interchangeable parts that will alter the software universe as surely as the industrial revolution changed manufacturing.”⁵² Whatever they might have believed about the possibility of such a silver bullet being developed in the future, however, most programmers and managers agreed that none existed in

⁵¹ Brooks, ““No Silver Bullet.”

⁵² Brad Cox, “There is a Silver Bullet,” *Byte* 15, 10 (1990).

the present. In the late 1980s, almost three decades after the first high-level automatic programming systems were introduced, concern about the software crisis was greater than ever. The same year that Brooks published his "No Silver Bullet," the Department of Defense warned against the very real possibility of "software-induced catastrophic failure" disrupting its strategic weapons systems.⁵³ Two years later, Congress released a report entitled "Bugs in the Program Problems in Federal Government Computer Software Development and Regulation," initiating yet another full-blown attack on the fundamental causes of the software crisis.⁵⁴ Ironically, the Department of Defense decided that what was needed to deal with this most recent outbreak of crisis was yet another new programming language – in this case ADA, which was trumpeted as a means of "replacing the idiosyncratic 'artistic' ethos that has long governed software writing with a more efficient, cost-effective engineering mind-set."⁵⁵

Why have automatic programming languages and other technologies thus far failed to resolve – or apparently even mitigate – the seemingly perpetual software crisis? First of all, it is clear that many of these languages and systems

⁵³ Morrison, "Software Crisis," 72.

⁵⁴ The 33-page report, entitled "Bugs in the Program: Problems in Federal Government Computer Software Development and Regulation," was written by two staff members, James H. Paul and Gregory C. Simon, of the Subcommittee on Investigations and Oversight of the House Committee on Science, Space, and Technology. The content of the report was covered in *The Washington Post* (October 17, 1989), D1 and *Science* (November 10, 1989), 753 among many other publications. For example, see Gary Chapman, "Bugs in the program," *Communications of the ACM* 33,3 (1990), 251-252.

⁵⁵ Morrison, "Software Crisis," 72.

were not able to live up to their marketing hype. Even those systems that were more than a “complex, exception-ridden performer of clerical tasks which was difficult to use and inefficient,” (as John Backus characterized the programming tools of the early 1950s) could not eliminate the need for careful analysis and skilled programming. As Willis Ware characterized the situation in 1965,

We lament the cost of programming; we regret the time it takes. What we really are unhappy with is the total programming process, not programming (i.e. writing routines) per se. Nonetheless, people generally smear the details into one big blur; and the consequence is, we tend to conclude erroneously that all our problems will vanish if we can improve the language which stands between the machine and the programmer. T'aint necessarily so. All the programming language improvement in the world will not shorten the intellectual activity, the thinking, the analysis, that is inherent in the programming process. Another name for the programming process is “problem solving by machine;” perhaps it suggests more pointedly the inherent intellectual content of preparing large problems for machine handling.⁵⁶

Although programming languages could reduce the amount of clerical work associated with programming, and did help eliminate certain types of errors (mostly those associated with transcription errors or syntax mistakes), they also introduced new sources of error. In the late 1960s a heated controversy broke out in the programming community over the use of the “GOTO statement.”⁵⁷ At the heart of this debate was the question of professionalism: although high-level languages gave the impression that just anyone could program, many

⁵⁶ Willis Ware, “As I See It: A Guest Editorial,” *Datamation* 11, 5 (1965), 27.

⁵⁷ Edsger Dijkstra, “Go To Statement Considered Harmful,” *Communications of the ACM* 11, 3 (1968), 147-148.

programmers felt this was a misconception disastrous to both their profession and the industry in general. The debate over who the legitimate users of these languages should be, and consequently who the languages should be designed for, was by no means new. At the 1962 RAND Symposium on Programming Languages, Jack Little lamented the tendency of manufacturers to designing languages "for use by some sub-human species in order to get around training and having good programmers ..."⁵⁸ Dick Talmadge and Barry Gordon of IBM admitted to thinking in terms of an imaginary "Joe Accountant" user: the problem that IBM faced, according to Galler, was that "If you can design a language that Joe Accountant can learn easily, then you're still going to have problems because you're probably going to have a lousy language."⁵⁹ Fred Gruenberger of RAND later summed up the essence of the entire debate: "COBOL, in the hands of a master, is a beautiful tool - a very powerful tool. COBOL, as it's going to be handled by a low grade clerk somewhere, will be a miserable mess...Some guys are just not as smart as others. They can distort anything."⁶⁰

The designers and advocates of various automatic programming systems never succeeded in addressing the larger issues posed by the difficulties inherent

⁵⁸ RAND Symposium, "On Programming Languages, Part I," 29-30.

⁵⁹ RAND Symposium, "On Programming Languages, Part II," 27.

⁶⁰ Ibid, 28.

in the programming process. High-level languages were necessary but not sufficient: that is, the use of these languages became an essential component of software development, but could not in themselves ensure a successful development effort. Programming remained a highly skilled occupation, and programmers continued to defy traditional methods of job categorization and management. By the end of the 1960s the search for a “silver bullet” solution to the software crisis had turned away from programming languages and towards more comprehensive techniques for managing the programming process. Many of these new techniques involved the creation of new automatic programming technologies, but most revolved around more systemic solutions and new methods of programmer education, management, and professional development.

Chapter Two: The Mongolian Horde versus The Superprogrammer

Most experts agree that another barrier to the most desirable use of the computer is the immense culture and communication gap that divides managers from computer people. The computer people tend to be young, mobile, and quantitatively oriented, and look to their peers both for company and for approval ... Managers, on the other hand, are typically older and tend to regard computer people either as mere technicians or as threats to their position and status - in either case they resist their presence in the halls of power.¹

"Computers Can't Solve Everything," *Fortune Magazine* (1969)

I. **The Software Crisis as a Problem of Programmer Management**

In the collective memory of the programming community, the years between 1968 and 1972 mark a major turning point in the history of their industry and profession. It is during this period that the rhetoric of crisis became firmly entrenched in the vernacular of commercial computing. A series of highly public software disasters – the software related destruction of the Mariner I spacecraft, the IBM OS/360 debacle, the devastating criticism of contemporary EDP practices published by McKinsey and Company – lent credence to the popular belief that an industry-wide software crisis was imminent.² The 1968 Garmisch conference gave voice to widespread concerns that the production of software

¹ T. Alexander, "Computers Can't Solve Everything," *Fortune* (October, 1969), 169.

² The Mariner I incident involved a software problem that resulted in the destruction of multi-million dollar spacecraft. The OS/360 operating system, which cost the IBM Corporation half a billion dollars to develop – the single largest expenditure in company history, was delivered nine months late and riddled with errors. The 1968 McKinsey reports suggested that most corporate computer efforts were not only poorly managed but also unprofitable.

had become “a scare item for management...an unprofitable morass, costly and unending,” while at the same time establishing “software engineering” as the dominant paradigm for thinking about the future of the industry.³

In the wake of these events numerous attempts were made to realize this software engineering “revolution.” Many of these attempts involved not so much the development of new programming technologies as the imposition of new management methodologies. Indeed, many of the most significant innovations in software engineering to be developed in the immediate post-Garmisch era were as much managerial as they were technological or professional. This turn towards management solutions to the software crisis reflects a significant shift in contemporary attitudes towards programmers and other computer specialists. By reconstructing the software crisis as a problem of management technique rather than technological innovation, advocates of these new management-oriented approaches also relocated the focus of its solution, removing it from the domain of the computer specialist and placing it firmly in the hands of traditional managers.

This chapter explores the changing relationship between software workers and their corporate employers. My argument is that the significant developments in software management that occurred in this period can best be

³ Peter Naur, Brian Randall, and J.N. Buxton, ed., *Software Engineering: Proceedings of the NATO conferences* (New York: Petrocelli/Carter, 1976), 4.

understood as a jurisdictional struggle over control of the increasingly valuable occupational territory opened up by the electronic digital computer. Just as the computer itself was gradually reconstructed, in response to a changing social and technical environment, from a scientific and military instrument into a mechanism for corporate communication and control, the modern business organization had to adapt itself to the presence of a powerful new technology. As the computer transformed from a tool *to be* managed into a tool *for* management, computer users emerged as powerful "change-agents" (to use the management terminology of the era). Faced with this perceived challenge to their occupational territory, traditional managers attempted to reassert their control over corporate data processing. The managerial innovations of the late 1960s can only be understood in terms of this very real struggle for professional authority.

"Seat-of-the-Pants Management"

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 P. 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."⁴

That a technical manager in a conservative corporation should use such lofty language in reference to such a stereotypically mundane and prosaic occupation is striking but not unusual. Throughout the 1950s and early 1960s, computer programming was widely considered to be a uniquely creative activity - genuine "brain business,' often an agonizingly difficult intellectual effort" - and therefore almost impossible to manage using conventional methods.⁵ Anecdotal evidence seemed to indicate that "the past management techniques so successful in other disciplines do not work in programming development ... Nothing works except a flying-by-the-seat-of-the-pants approach."⁶ The general consensus was that computer programming was "the kind of work that is called creative [and] creative work just cannot be managed."⁷

The idea that computer programming was somehow an "exceptional" activity, unconstrained by the standard organizational hierarchy and controls, forestalled early attempts to automate its processes or to regulate its activities.

⁴ Frederick P. Brooks, *The Mythical Man-Month: Essays on Software Engineering* (New York: Addison-Wesley, 1975), 7.

⁵ Gene Bylinsky, "Help Wanted: 50,000 Programmers," *Fortune* 75, 3 (March, 1967), 141.

⁶ Charles Lecht, *The Management of Computer Programming Projects* (New York: American Management Association, 1967), 9.

⁷ Robert Gordon, "Review of Charles Lecht, *The Management of Computer Programmers*," *Datamation* 14, 4 (1968), 200-202.

We lament the cost of programming; we regret the time it takes. What we really are unhappy with is the total programming process, not programming (i.e. writing routines) per se ... *All the programming language improvement in the world will not shorten the intellectual activity, the thinking, the analysis, that is inherent in the programming process.*⁸

In the early decades of computing there were a number of reasons why software development projects were generally thought to be unsusceptible to traditional management approaches. First of all, even the most veteran computer users had only a few years' experience with these novel devices. In the early 1950s, the technology of electronic computing changed and developed with remarkable rapidity. The "best practice" guidelines that applied to one particular generation of equipment were quickly superceded by a different set of techniques and methodologies.⁹ Secondly, and perhaps more importantly, the performance and memory constraints of the first commercial computers demanded that programmers cultivate a series of idiosyncratic and highly individual craft techniques designed to overcome the limitations of primitive hardware. For example, contemporary memory devices were so slow and had such little capacity that programmers had to develop ingenious techniques to fit their programs into the available memory space. In order to coax every bit of speed out of a relatively slow storage device such as a rotating memory drum,

⁸ Willis Ware, "As I See It: A Guest Editorial," *Datamation* 11, 5 (1965), 27. Emphasis added.

⁹ Datamation Editorial, "The Facts of Life," *Datamation* 14, 3 (1968), 21.

programmers would carefully organize their coded instructions in such a way as to assure that the each instruction passed by the magnetic read head in just the right order and at just the right execution time. Only the best programmers could hope to develop applications that worked at acceptable levels of usability and performance. As IBM researcher John Backus famously characterized the situation, “programming in the 1950s was a black art, a private arcane matter ... each problem required a unique beginning at square one, and the success of a program depended primarily on the programmer's private techniques and inventions.”¹⁰

This reliance on individual creativity and clever “work-arounds” created the impression that programming was indeed more of an art than a science. This notion was reinforced by a series of aptitude tests and personality profiles that suggested that computer programmers, like chess masters or virtuoso musicians, were endowed with a uniquely creative ability. Great disparities were discovered between the productivity of individual programmers. Dr. E.E. David of Bell Telephone Laboratories spoke for many when he argued that large software projects could never be managed effectively, because “the vast range of programmer performance indicated earlier may mean that it is difficult to obtain

¹⁰ John Backus, “Programming in America in the 1950s - Some Personal Impressions,” in Nick Metropolis, et al., eds., *A history of computing in the twentieth century a collection of essays* (New York: Academic Press, 1980), 126.

better size-performance software using machine code written by an army of programmers of lesser than average caliber."¹¹

This focus on programmer performance contributed to the social construction of the software crisis as a problem of programmer personnel selection. In the 1950s the primary crisis facing the industry were quantifiable manpower shortages; by the 1960s the issue was more qualitative. The problem was not so much a lack of programmers per se; what the industry was really worried about was a shortage of experienced, *capable* developers. One industry observer went so far as to argue that the "major managerial task is finding - and keeping - 'the right people': with the right people, all problems vanish."¹² Programmers were selected for their intellectual gifts and aptitudes, rather than their business knowledge or managerial savvy. "Look for those who like intellectual challenge rather than interpersonal relations or managerial decision-making. Look for the chess player, the solver of mathematical puzzles."¹³ Skilled programmers were thought to be effectively irreplaceable, and were treated and compensated accordingly.

During this period, many corporate programmers enjoyed an unprecedented degree of personal authority and professional autonomy. "Experience shows

¹¹ Naur, et al., *Software Engineering*, 33.

¹² Robert Gordon, "Personnel Selection," in Fred Gruenberger and Stanley Naftaly, eds., *Data Processing. Practically Speaking* (Los Angeles: Data Processing Digest, 1967), 88.

¹³ Joseph O'Shields, "Selection of EDP Personnel," *Personnel Journal* 44, 9 (October 1965), 472.

that high quality individuals are the key to top grade programming," argued a 1959 Price-Waterhouse study on *Business Experience with Electronic Computers*:

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...A knowledge of business operations can usually be obtained by an adequate expenditure of time and effort. Innate ability, on the other hand, seems to have a great deal to do with a man's capacity to perform effectively in the fields of computer coding and systems design.¹⁴

Programmers were not only "encouraged to feel they are professionals," but they were included as active participants in all phases of application development, from design to implementation, in order to ensure their cooperation and enthusiasm.¹⁵ Systems analysts and programmers were "professionals doing work that is generally of a higher creative level than most work found in business today."¹⁶ As professionals with certain highly developed skills, "each practitioner has a 'personal monopoly' which manifests itself in the market place."¹⁷ For the time being, the power to control the computer rested with the individual programmer, rather than with the management bureaucracy.

By the beginning of the 1960s, however, developments occurred in both the technical and social environment of commercial computing that prompted a

¹⁴ 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), 81-83.

¹⁵ Conway, *Business experience with electronic computers*, 81.

¹⁶ Roger Guarino, "Managing Data Processing Professionals," *Personnel Journal* (December, 1969), 972-975.

¹⁷ *Ibid*, 972.

reevaluation of conventional methods of software production. In the first half of the decade innovations in transistor and integrated circuit technology increased the memory size and processor speed of computers by a factor of ten, providing an effective performance improvement of almost a hundred. The falling cost of hardware allowed computers to be used for more and larger applications, which in turn required larger and more complex software. As the scale of software projects expanded, they became increasingly difficult to supervise and control. They also became much more expensive. Large software development projects acquired a reputation for being behind-schedule, over-budget, and bug-ridden.

Freed from some of the constraints of earlier technology, the pressing problems for software developers now appeared to be more managerial than technical. New perspectives on these problems began to appear in the industry literature. "There is a vast amount of evidence to indicate that writing- a large part of programming is writing after all, albeit in a special language for a very restricted audience - can be planned, scheduled and controlled, nearly all of which has been flagrantly ignored by both programmers and their managers," argued Robert Gordon in a 1968 review article.¹⁸ The professional journals of this period are replete with exhortations towards better software development management: "Controlling Computer Programming"; "New Power for

¹⁸ Gordon, "Personnel Selection," 200.

Management"; "Managing the Programming Effort"; "The Management of Computer Programming Efforts."¹⁹ Although it was admittedly true "that programming a computer is more an art than a science, that in some of its aspects it is a creative process," this new perspective on software management suggested that "as a matter of fact, a modicum of intelligent effort can provide a very satisfactory degree of control."²⁰

One of the justifications often suggested for this changing attitude toward programming (and programmers) was basic economic efficiency. Towards the end of the 1960s the venerable management consulting firm of McKinsey & Company published a series of reports suggesting that the real reason that most data processing installations were unprofitable is that "many otherwise effective top managements...have abdicated control to staff specialists - good technicians who have neither the operation experience to know the jobs that need doing nor the authority to get them done right."²¹ These reports helped redefine contemporary understandings of the nature and causes of the software crisis, turning the focus of debate away from "finding and caring for good

¹⁹ C.I. Keelan, "Controlling Computer Programming," *Journal of Systems Management* (January, 1969); D. Herz, *New Power for Management* (New York: McGraw-Hill, 1969); Richard Canning, "Managing the Programming Effort," *EDP Analyzer* 6, 6 (1968), 1-15; Charles Lecht, *The Management of Computer Programming Projects* (New York: American Management Association, 1967).

²⁰ Keelan, "Controlling Computer Programming," 30.

²¹ McKinsey & Company, "Unlocking the Computer's Profit Potential," *Computers & Automation* (April 1969), 33.

programmers" and squarely towards the problem of programmer management.²² The solution to "unlocking the computer's profit potential," according to the McKinsey & Company, was to restore the proper balance between managers and programmers: "Only managers can manage the computer in the best interests of the business. The companies that take this lesson to heart today will be the computer profit leaders of tomorrow."²³ By reconstructing the software crisis as a problem of management technique rather than technological innovation, the McKinsey report also relocated the focus of its solution, removing it from the domain of the computer specialist and placing it firmly in the hands of traditional managers.

The 1968 Garmisch Conference irrevocably established software management as one of the central rhetorical cornerstones of all future software engineering discourse. The organizers of the conference called for the adoption by software manufacturers of the "types of theoretical foundations and practical disciplines" traditional in the established branches of engineering.²⁴ For a number of conference participants, the key word in this provocative manifesto was "discipline." In his widely quoted paper on "mass-produced software

²² Service Bureau Corporation, "Find and care for a good programmer, and keep him happy," *Datamation* 10, 7 (1964).

²³ McKinsey & Company, "Unlocking the Computer's Profit Potential," 33.

²⁴ Naur, et al., *Software Engineering*, 7.

components,” Douglas McIlroy forcefully articulated his plan for “industrializing” software production:

We undoubtedly produce software by backward techniques. We undoubtedly get the short end of the stick in confrontations with hardware people because they are the industrialists and we are the crofters. Software production today appears in the scale of industrialization somewhere below the more backward construction agencies. I think that its proper place is considerably higher, and would like to investigate the prospects for mass-production techniques in software.²⁵

Although McIlroy does not explicitly address issues of professional concern to occupational programmers, such as status, autonomy, and job satisfaction, his vision of a software “components factory” invokes familiar images of industrialization and proletarianization. According to his proposal, an elite corps of “software engineers” would serve as the Frederick Taylors of the software industry, carefully orchestrating every action of a highly-stratified programmer labor force. And like the engineers in more traditional manufacturing organizations, these software engineers would identify themselves more as corporate citizens than as independent professionals.

Not every proposed solution to the software crisis suggested at Garmisch was as blatantly management-oriented as McIlroy’s. Nevertheless, the theme of transformation from a craft-based “black art” of programming to the industrial discipline of software engineering dominated many of the presentations and

²⁵ Ibid.

discussion. The focus on management solutions reflected –and reinforced – a larger groundswell of popular opinion that extended far beyond the confines of the actual conference. The industry literature of the period is replete with examples of this changing attitude towards software management. Even those proposals that seemed to be most explicitly technical, such as those advocating structured programming techniques or high-level language developments, contained a strong managerial component. Most required a rigid division of labor and the adoption of tight management controls over worker autonomy. When a prominent adherent of object-oriented programming techniques spoke of “transforming programming from a solitary cut-to-fit craft, like the cottage industries of colonial America, into an organizational enterprise like manufacturing is today,” he was referring not so much to the adoption of a specific technology, but rather to the imposition of established and traditional forms of labor organization and workplace relationships.²⁶ The solutions to the “software crisis” most frequently recommended by managers - among them the elimination of rule-of-thumb methods (i.e. the “black art” of programming), the scientific selection and training of programmers, the development of new forms of management, and the efficient division of labor – were not fundamentally

²⁶ Brad Cox, “There is a Silver Bullet,” *Byte* 15, 10 (1990), 209.

different from the four principles of scientific management espoused by Frederick Taylor in an earlier era.²⁷

II. Aristocracy, Democracy, and Systems Design

It would be impossible to describe all of the numerous approaches to programmer management that were developed in this and subsequent periods. It is enough for the purposes of this dissertation to describe the defining characteristics of a few of the most prominent methodologies: the hierarchical system; the chief programmer team approach; and the adaptive programmer team (or “egoless” programming) model. The hierarchical systems approach, originally developed for large, government-sponsored programming projects at the System Development Corporation and IBM Federal Systems Division, resembles the highly stratified, top-down organizational structure familiar to most conventional corporate employees. The chief programmer team, although it was also developed at IBM Federal Systems, reflects an entirely different approach to programmer management oriented around the leadership of a single

²⁷ Taylor’s four principles of scientific management can easily be mapped on the software management literature of this and other periods. In brief, his four principles were: 1) develop a science for each element of work to replace traditional rule-of-thumb methods; 2) scientifically select, train, and develop the workers, rather than let them define their own work practices; 3) cooperate with the workers to insure adherence to the new scientific principles; 4) establish an equal division of the work and the responsibility between management and labor, with management taking over all the tasks for which they are better suited. See Frederick Winslow Taylor, *The Principles of Scientific Management* (New York: Harper Brothers, 1911).

managerially-minded "super-programmer." The adaptive team approach was popularized as "egoless" programming by the iconoclastic Gerald Weinberg in his 1971 classic *The Psychology of Computer Programming*.²⁸ Weinberg proposed an open, "democratic" style of management that emphasized teamwork and rotating leadership.

Although it is possible to arrange these approaches into a roughly chronological order, it is not my intention to suggest that they represent any simple "evolution" towards increasing managerial control or economic efficiency. Each of these management methodologies represents separate but interrelated visions about how computer programming as an economic activity, and computer programmers as aspiring professionals, could best be integrated into the established social and technological systems of the traditional corporation. Each of these approaches built on, and responded to, the innovations and shortcomings of the others. They also reflected the backgrounds and aspirations of their advocates and developers. By studying carefully the salient features of each of these three methodologies, we will be better able to situate them in their particular social and historical context, and hence to understand more fully their contribution to contemporary debates about the nature and causes of the software crisis.

²⁸ Gerald Weinberg, *The Psychology of Computer Programming* (New York: Van Nostrand Reinhold, 1971).

Armies of Programmers

The first concerted attempts to manage software development projects using established management techniques occurred at the government- and military-sponsored SAGE (Semi-Automatic Ground Environment) air-defense project. The SAGE project was the heart of an ambitious “early warning” radar network intended to provide an immediate and centralized response to sneak attacks from enemy aircraft. The plan was to develop a series of computerized tracking and communications centers that would coordinate observation and response data from a widely dispersed system of interconnected perimeter warning stations. First authorized by Congress in 1954, by 1961 the SAGE system had cost more than \$61 billion to develop and operate, and required the services of over 200,000 employees. The software that connected the specially designed, real-time SAGE computers was the largest programming development then under way. The System Development Corporation (SDC), a RAND Corporation spin-off company responsible for developing this software, had to train and hire almost 2,000 programmers. In the space of a few short years the personnel management department at SDC effectively doubled the number of trained programmers in the United States.

In order to effectively organize an unprecedented number of software developers, SDC experimented with a number of different techniques for

managing the programming process. For the most part, however, SDC relied on a hierarchical structure that located most programmers at the lowest levels of a vast organizational pyramid built with layer upon layer of managers.²⁹ The top of this hierarchy was occupied by non-technical administrators. The middle layers were peopled by those EDP [electronic data processing] personnel who had exhibited a desire or aptitude for management. In other words, the managers in the SDC hierarchy were self-selected as being either uninterested or uncommitted to a long-term programming career. The management style in this hierarchical structure was generally autocratic. Managers made all of the important decisions. They assigned tasks, monitored the progress of subordinates, and determined when and what corrective actions needed to be taken.

This hierarchical approach to management was attractive to SDC executives for a number of reasons. First of all, it was a familiar model for government and military subcontractors. Secondly, it was often easier to justify billing for a large number of mediocre low-wage employees than a smaller number of excellent but expensive contractors. Finally, and perhaps most importantly, the "Mongolian horde" approach to software development corresponded nicely with contemporary constructions of the root causes of the burgeoning "software

²⁹ Claude Baum, *The Systems Builders: The Story of SDC* (Santa Monica, CA: System Development Corporation, 1981), 52.

turmoil.”³⁰ In the early part of the 1950s, the major problem facing the computer industry was believed to be programming training and recruitment. At the 1954 Conference on Training Personnel for the Computing Machine Field, E.P. Little of Wayne State University warned industry managers that “estimates of manpower needs for computer applications...[are] astounding compared to the facilities for training people for this work.”³¹ W.H. Wilson of General Motors observed “a universal feeling that there is a definite shortage of technically trained people in the computer field.”³² The widely perceived “gap in programming support” was thought to be a problem of quantity rather than quality.³³ It was only later that the emphasis shifted from programming training to programmer selection, from “where do we find programmers,” to “where do we find the *right* programmers.”

Faced with a shortage of experienced programmers, SDC embarked on an extensive programming of internal training and development. Most of their trainees had little or no experience with computers; in fact, many managers at SDC preferred it that way. Like many corporations in the 1950s, they believed that “It is much easier to teach our personnel to program than it is to teach

³⁰ Also known as the “Chinese army” approach, at least until the phrase became unpopular in the early 1950s.

³¹ Arvid W. Jacobson, ed., *Proceedings of the First Conference on Training Personnel for the Computing Machine Field held at Wayne University, Detroit, Michigan, June 22 and 23, 1954* (Detroit: Wayne University Press, 1955), 79.

³² Jacobson, *Proceedings of the First Conference on Training Personnel*, 21.

³³ Robert Patrick, “The Gap in Programming Support,” *Datamation* 7, 5 (1961), 37.

outside experienced programmers the details of our business."³⁴ In any case, in the period between 1956 and 1961 the company trained 7,000 programmers and systems analysts. At a time when all the computer manufacturers combined could only provide 2,500 student weeks of instruction annually, SDC devoted more than 10,000 student weeks to instructing its own personnel to program.³⁵

The apparent success that SDC achieved in mass-producing programming talent reinforced the notion that a hierarchical approach was the suitable model for large-scale software development. If large quantities of programmers could be produced on demand, then individual programmers were effectively anonymous and replaceable. A complex system like SAGE could be broken down into simple, modular components that could be easily understood by any programmer with the appropriate training and experience. The principles behind the approach were essentially those that had proven so successful in traditional manufacturing: replaceable parts, simple and repetitive tasks, and a strict division of labor.

The hierarchical model of software development was adopted by a number of other major software manufacturers, particularly those involved in similarly large military or government projects. It is not clear how direct was the

³⁴ Baum, *The Systems Builders*, 48.

³⁵ T.C. Rowan, "The Recruiting and Training of Programmers," *Datamation* 4, 3 (1958), 16-18.

connection between SDC and these other manufacturers. SDC certainly had a role in training a large number of programmers and EDP managers. "We trained the industry!" boasted SDC veterans: "Whatever company I visit, I meet two or three SDC alumni."³⁶ The labor historian Philip Kraft attributes much of what he refers to as the "routinization" of programming labor to the "degrading" influence of military-industrial organizations such as SDC. He describes the SDC "software factories" as "the first systematic, large-scale effort on the part of EDP users to transform the highly idiosyncratic, artisan-like occupation" of computer programming into "one which more closely resembled conventional industrial work."³⁷ He argues that SDC played a significant role in diffusing and popularizing the hierarchical approach to software engineering management.

Whether the claim that SDC policies and SDC personnel played a direct role in diffusing the hierarchical system of management throughout the computer industry was valid, similar top-down methodologies were widely adopted. In the IBM Federal Systems division, a multi-level organizational structure was used on all large government projects. IBM manager Philip Metzger provided a detailed description of the Federal Systems approach in his highly popular textbook *Managing a Programming Project*, which went through three editions

³⁶ Baum, *The Systems Builders*, 47.

³⁷ Philip Kraft, *Programmers and Managers: The Routinization of Computer Programming in the United States* (New York: Springer-Verlag, 1977), 39.

in the period between 1973 and 1996.³⁸ A 1974 article on "Issues in Programming Management" that appeared in the respected industry newsletter *EDP Analyzer* listed the hierarchical systems approach as one of the most commonly implemented software management methodologies.³⁹ Joel Aron, another IBM Federal Systems veteran, used the hierarchical model as the basis for his series of books on *The Program Development Process*.⁴⁰

The hierarchical approach to software development was attractive to managers because it corresponded nicely with the contemporary management theories. In the first half of the twentieth century, corporate management became a professional activity dominated by specialists and experts. As the historian Alfred Chandler has famously described it, "the existence of a managerial hierarchy is a defining characteristic of the modern business enterprise."⁴¹ These professional managers developed a shared culture and value system reinforced by an increasingly formalized program of training and education. They exerted a high degree of control over the work practices of their subordinates, "scientifically managing" all aspects of the business and

³⁸ Philip Metzger, *Managing a Programming Project* (Englewood Cliffs, N.J: Prentice-Hall, 1973).

³⁹ Richard Canning, "Issues in Programming Management," *EDP Analyzer* 12, 4 (1974), 1-14.

⁴⁰ Joel Aron, *The Individual Programmer, The Systems Programming Series* (Reading, MA: Addison-Wesley, 1983); Joel Aron, *Part II: The Programming Team, The Program Development Process* (Reading, MA: Addison-Wesley, 1983).

⁴¹ Alfred P. Chandler, *The Visible Hand: The Managerial Revolution in American Business* (Cambridge, MA: Belknap Press, 1977), 7.

manufacturing process. EDP managers assumed that the techniques and structures that appeared to work so efficiently in traditional industries would translate naturally into the software development department. It was only a matter of identifying and implementing the "one best way" to develop software components.

Embedded in the hierarchical model of management were a series of assumptions about the essential character of programming as an occupational activity. Implied in the suggestion that the structures and procedures of a traditional manufacturing organization could be seamlessly mapped on to the EDP department was a belief that the skills and experience required to program a computer were, in effect, not very different from those required to assemble an automobile. Managers could define, in the minutest detail, the specifications that the programmers would follow. The programmers, in turn, need only be trained to perform a very limited and specialized function. Individual programmers were looked upon as interchangeable units.⁴² They lacked a distinct professional identity. The path to advancement in the hierarchical system (if indeed there actually was one available to mere programmers) was through management. Certification programs were desirable in order to ensure a minimum levels of competence, but only as means for assuring a standard

⁴² Richard Canning, "Issues in Programming Management."

degree of performance and product.⁴³ Programmers were encouraged to be professionals only to the extent that being a professional meant self-discipline, a willingness to work long hours with no overtime pay, and loyalty to the corporation and obedience to supervisors.⁴⁴

The notion that programmers could be treated as unskilled clerical workers was reinforced by a series of technical developments intended to allow managers to mechanically translate high-level systems designs into the low-level machine code required by a computer. For example, one of the alleged advantages of the COBOL programming language frequently touted in the literature was its ability to be read, understood – and perhaps even written – by informed managers.⁴⁵ More than a fashionable management technique, the hierarchical organizational model was a philosophy about what programming was and where programmers stood in relation to other corporate professionals. It embodied – in a complex of interrelated cultural, technical, and political systems - a particular social construction of the nature and causes of the software crisis.

Despite the obvious appeal that the theory of hierarchical systems held for conventional managers, it rarely worked as intended in actual practice. Although managers would have preferred to think of programming as routine

⁴³ Richard Canning, "Professionalism: Coming or Not?," *EDP Analyzer* 14, 3 (1975), 1-12.

⁴⁴ Brian Rothery, *Installing and Managing a Computer* (London: Business Books, 1968), 80.

⁴⁵ Gordon, "Personnel Selection," 85.

clerical work and programmers as interchangeable laborers, experience suggested that in reality the situation was quite different. I have already described how, in the late 1950s and early 1960s, programming had acquired a reputation as being a uniquely creative activity requiring “real intellectual ability and above average personal characteristics.”⁴⁶ “To ‘teach’ the equipment, as is amply evident from experience to date, requires considerable skill, ingenuity, perseverance, organizing ability, etc. The human element is crucial in programming.”⁴⁷ Anecdotal evidence suggesting that skilled programmers were essential elements of software development was supported by numerous empirical studies produced by industrial psychologists and personnel experts.

The realization that computer programming was a more intellectually challenging activity than was originally anticipated threw a monkey wrench into the elaborate hierarchical systems that managers had constructed. Whereas the “software turmoil” of the 1950s was attributed largely to numerical shortages of programmers, the “programmer quality” problems of the 1960s demanded a subtly different construction of the root causes of the “software crisis.” The problem could still be defined as a management problem requiring a management-driven solution. What had changed was the prevailing conception

⁴⁶ Conway, *Business experience with electronic computers*, 81.

⁴⁷ *Ibid*, 81-82.

of what programmers were and what they did. A 1967 article in *Fortune*

Magazine laid out the issue in plain language for its executive readership:

The massive attack on systems software poses difficult management problems. On the one hand, a good programmer, like a writer or composer, works best independently. But the pressure to turn out operating systems and other programs within a limited time make it necessary to deploy huge task forces whose coordination becomes a monstrous task. The problem is further complicated by the fact that there is no "best way" to write either a systems or an application program, or any part of such program. Programming has nowhere near the discipline of physics, for example, so intuition plays a large part. Yet individual programmers differ in their creative and intuitive abilities.⁴⁸

Companies that implemented hierarchical systems methodologies also discovered that programmers were not content with the professional identity that these systems imposed upon them. Turnover rates in the industry reached crisis proportions, averaging close to 20 percent annually.⁴⁹ One large employer experienced a sustained turnover rate of 10% *per month*.⁵⁰ The problem, according to one SDC survey of termination interviews, was that programmers working in hierarchical organizations "did not foresee for themselves the opportunities they want for professional growth and development...or for promotion and advancement."⁵¹ The career aspirations of the programmers

⁴⁸ Bylinsky, "Help Wanted: 50,000 Programmers," 141.

⁴⁹ H.V. Reid, "Problems in Managing the Data Processing Department," *Journal of Systems Management* (May, 1970), 8; Richard Canning, "Managing Staff Retention and Turnover," *EDP Analyzer* 15, 8 (1977), 1-13.

⁵⁰ Datamation Editorial, "EDP's Wailing Wall," *Datamation* 13, 7 (1967), 21.

⁵¹ Baum, *The Systems Builders*, 52.

conflicted with the occupational role they had been assigned by the managers. Many preferred to pursue professional advancement *within* programming, rather than *away* from programming. In the hierarchical system, the higher an individual advanced, the more they worked as administrators rather than technologists.

Superprogrammer to the rescue...

The advocates of hierarchical management received their most devastating blow in the early 1970s, with the publication of Frederick P. Brooks' *The Mythical Man-Month*. Frederick Brooks was the project manager for the IBM Corporation's – and the world's - most ambitious software development project to date: the IBM OS/360 operating system. In the early half of the 1960s, IBM began to feel increasing competitive pressure from smaller computer manufacturers such as Control Data Corporation (CDC) and Honeywell. CDC had succeeded in challenging IBM's high-end mainframe business with a fast machine (the CDC 6600) with a superior price-performance ratio. In 1963 Honeywell announced their H-200 machine, which posed a serious threat to IBM's lucrative low-end 1401 computers. The H-200 was 30% cheaper, significantly more powerful, and offered 1401 software compatibility via its "Liberator" automatic program translator. In response to these changing market

conditions, IBM decided to integrate all of its products into a single, compatible series: the System 360 machines.

The IBM System/360 has been referred to as "the computer that IBM made, that made IBM."⁵² The System/360 systems solved a number of problems for IBM and its customers. It filled in the gaps in the IBM line of product offerings by providing an entire range of hardware and software compatible computers ranging from the low-end model 360/20 (intended to compete directly with the Honeywell H-200) to the model 360/90 supercomputer, which compared favorably to the CDC-6600. By making all of these machines software compatible (theoretically, at least) IBM provided an inexpensive upgrade path for its customers. The client could purchase just the amount of computing power that they needed, knowing that if their needs changed in the future they could simply transfer their existing applications and data to the next level of System/360 hardware. They could also make use of their existing peripherals, such as tape readers and printers, without requiring an expensive upgrade.

The System/360 was an enormously risky and expensive undertaking. The *Fortune* journalist Tom Wise referred to it as "IBM's \$5 Billion Gamble." He quoted one senior IBM manager as calling it the "we bet the company" project.⁵³ The riskiest and most expensive component of System/360 development was the

⁵² Martin Campbell-Kelly, in a 1991 lecture.

⁵³ Thomas Wise, "IBM's \$5,000,000,000 Gamble," *Fortune* (September 1966), 226.

OS/360 operating system. In the years between 1963 and 1966, over 5,000 staff years of effort went into the design, construction, and documentation of OS/360. When OS/360 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 – “the single largest expenditure in company history.”⁵⁴

Although the System/360 project turned out to be a tremendous success for IBM, sealing their position of leadership in the commercial computer industry for the next several decades, the OS/360 project was generally considered to be a financial and technological disaster. The costs of the OS/360 debacle were human as well as material:

The cost to IBM of the System/360 programming support is, in fact, best reckoned in terms of the toll it took on people: the managers who struggled to make and keep commitments to top management and to customers, and the programmers who worked long hours over a period of years, against obstacles of every sort, to deliver working programs of unprecedented complexity. Many in both groups left, victims of a variety of stresses ranging from technological to physical.⁵⁵

The highly-publicized failure of the OS/360 project served as a dramatic illustration of the shortcomings of the hierarchical management method. Techniques that had worked well on an application requiring 10,000 lines of code failed miserably when applied to a million code line project. Faced with serious

⁵⁴ Thomas Watson, Jr., quoted in Campbell-Kelly and Aspray, *Computer*, 199.

⁵⁵ Emerson Pugh, Lyle Johnson, and John Palmer, *IBM's 360 and Early 370 Systems* (Cambridge, MA: MIT Press, 1991), 336.

schedule slippages, quality problems, and unanticipated changes in scope, the OS/360 managers did what traditional manufacturing managers were accustomed to doing: they added more resources. The only noticeable result was that the project fell more and more behind schedule.

The Mythical Man-Month is OS/360 project leader Frederick Brooks's post-mortem analysis of the failures of traditional hierarchical management. It is one of the most widely-read and oft-quoted references on the practice of software engineering. The mythical man-month in the title refers to the commonly held notion that progress in software development projects occurs as a function of time spent times the number of workers allocated - the implication being that more workers equaled faster production. Brooks dismissed this assumption with now-famous Brooks's Law, one of the most memorable aphorisms in the lore of software development: *Adding manpower to a late software project makes it later*. Or to use one of Brooks's more earthy metaphors, "the bearing of a child takes nine months, no matter how many women are assigned."⁵⁶

The highly-quotable Brooks's Law was neither the only, nor even the most significant, of the insights provided in *The Mythical Man-Month*. Brooks did more than criticize existing methodologies; he provided an entirely new model for understanding software development management. Brooks was firmly

⁵⁶ Brooks, *The Mythical Man-Month*, 17.

convinced that there was a wide disparity in performance among individual programmers. He believed that small teams of sharp programmers were substantially more productive than much larger groups of merely mediocre performers. He also recognized, however, that even the best small team could only accomplish so much in any given period of time. The small team approach simply did not scale well onto larger projects. The problem of scalability was the heart of the “cruel dilemma” facing project managers: “For efficiency and conceptual integrity, one prefers a few good minds doing design and construction. Yet for large systems one wants a way to bring considerable manpower to bear, so that the product can make a timely appearance.”⁵⁷ And yet the Mongolian Horde model of throwing programming resources – so-called “man-months” – at projects was also obviously insufficient. What was needed was a way to apply the efficiency and elegance of the small team approach to the problems of large-project management.

Brooks proposed the adoption of what he called the “surgical team” model of software development. In doing so he borrowed heavily from the work of IBM manager and researcher Harlan Mills, who had earlier developed the “chief programmer team” concept.⁵⁸ In both versions of the chief programmer team

⁵⁷ Ibid, 31.

⁵⁸ The CPT concept was first introduced as one of two experimental “superprogrammer” projects by J.D. Aron in a paper given at the 1969 Rome Conference. The first

(CPT) approach, a single, expert programmer was responsible for all major design and implementation decisions involved with system development. The "chief programmer" (or surgeon), defined the program specifications, designed the program, coded it, tested it, and wrote the documentation. He was assisted in his tasks by an operating team of support staff. His immediate assistant (or copilot) was only slightly less expert than the chief programmer himself. He was the chief programmer's mirror and alter ego, serving not only as an emergency backup or stand-in, but also as an advisor, discussant, and evaluator. Although the assistant knew the program code intimately and may even have written some of it, it was the chief programmer who was ultimately responsible for it.

Other members of the Brooks's "surgical" team included an administrator, who handled schedules, money, personnel issues, and hardware resources; an editor, who provided the finishing touches to chief programmer's documentation; two secretaries who dealt with correspondence and filing; a program clerk who maintained all the technical records for the project; a "toolsmith" who built, constructed, and maintained the interactive tools used by the rest of the team for programming, debugging, and testing; a tester, who

experiment involved a 30 man-year project requiring 50,000 instructions. Harlan Mills attempted to complete the project himself (using a prototype 'surgical team') in only six months. The project eventually required about six man-years of effort to complete, and was considered a moderate success. The second experiment mentioned by Aron at the Rome conference turned out to be the famous New York Times project, which established the reputation of the chief programmer team approach when it was publicized by F.T. Baker in 1971.

served both as the chief programmer's adversary and assistant, and who developed test plans to challenge the integrity of the program design and devised test data for day-to-day debugging; and finally, the "language lawyer," who delighted in the mastery of the intricacies of a programming language. The language lawyer, unlike the chief programmer, was not involved in "big-picture" issues or system design; his responsibility was finding "neat and efficient ways to use the language to do difficult, obscure, or tricky things." Language lawyers were usually called in only for special, short-term assignments.⁵⁹

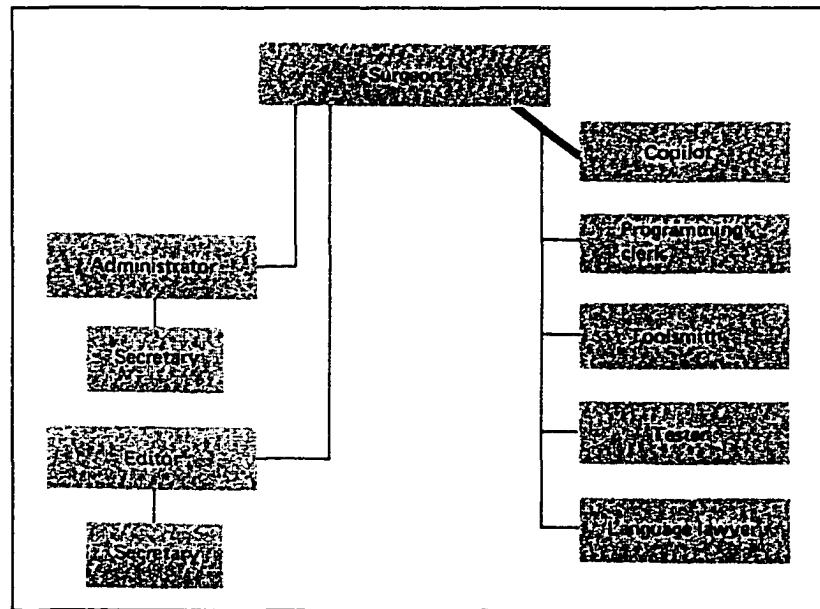


Figure 2.4: Communications Patterns in the Chief Programming Team

⁵⁹ Brooks, *The Mythical Man-Month*, 34-35.

The advantage to the chief programmer team approach, according to Mills and Brooks, was that it dramatically simplified communications between team members. Whereas a large, hierarchical organization of X number of employees could require as many as $(X^2-X)/2$ independent paths of communication, in the CPT model all essential information passed through the person of the chief programmer. Figure 2.4 illustrates the central role that that the chief programmer played in the organization of the programming team: all team members report to him directly, and did not communicate with each other directly.

By centralizing all decision-making in the person of the chief programmer, the CPT approach assured the maintenance of the program's structural integrity. Brooks compared the conceptual architecture of the typical large software project to the haphazard design of many European cathedrals; the patchwork structure of these cathedrals revealed an unpleasant lack of continuity, reflecting the different styles and techniques of different builders in different generations. Brooks preferred the architectural unity of the cathedral at Reims, which derived "as much from the integrity of design as from any particular excellences." This integrity was achieved only through the "self-abnegation of eight generations of builders," each of whom "sacrificed some of his ideas so that the whole might be of pure design."⁶⁰ Using wonderfully evocative Biblical language, Brooks

⁶⁰ Ibid, 42.

extolled the virtues of a unified conceptual design: "As the child delights in his mud pie, so the adult enjoys building things, especially things of his own design. I believe that this delight must be an image of God's delight in making things, a delight shown in the distinctiveness and newness of each leaf and each snowflake."⁶¹ Only the chief programmer team approach could guarantee such a degree of uncompromised architectural integrity.

The chief programmer team approach differed from hierarchical systems methodologies in a number of essential characteristics. Whereas the hierarchical model allowed for (and in fact encouraged) the use of novice programmers, the chief programmer team was built entirely around skilled, experienced professionals. This implied a radically different approach to professional development. Each member of the team was encouraged to develop within their own particular disciplinary competency; i.e. it wasn't necessary to become a surgeon to advance one's career. For example, an aspiring language lawyer could continue to focus on his technical specialty without feeling pressure to transfer into management. The chief programmer team approach embodied the belief that computer programming was a legitimate, respectable profession.

The chief programmer team also reflected changing contemporary notions about the nature of programming ability. The primary justification for using

⁶¹ Ibid, 7.

small teams of experienced programmers rather than large hordes of novices was the belief that one good programmer was worth at least ten of his average colleagues. In the person of the chief programmer, the innate technical abilities of the "superprogrammer" were merged with the organizational authority of the traditional manager. The chief programmer was both a technical genius and expert administrator. Programming aptitude could not be abstracted from its embodiment in particular individuals; skilled programmers were anything but "replaceable components" of an automated "software factory." In the elite "surgical team" model, the contributions of talented professionals far outweighed those provided by traditional management techniques or development methodologies.

Besides endowing computer programmers with considerable institutional power, *The Mythical Man-Month* reinforced the notion that programming was an exceptional activity, unlike any other engineering or manufacturing discipline. His suggestion that programming was akin to poetry strongly implied that programming was not an activity that could be readily systematized. What Brooks proposed was the adoption of useful tools and techniques, not some overarching methodology. As he later declared in a famous article entitled "No Silver Bullet," although the management of large

programming projects could be improved incrementally, there were no easy solutions to be derived from the lessons of traditional manufacturing.⁶²

Like the hierarchical systems model, the chief programmer team was intimately linked to specific techniques and technologies. Since all major decisions relating to both design and implementation had to be made by a single “superprogrammer”, the chief programmer team approach effectively demanded the adoption of top-down development techniques. Top-down programming was one of the foundational principles of the “structured programming” approach to software engineering advocated by many academic computer scientists in this period. The essence of top-down programming was the concept of abstraction: by proceeding step-by-step from general design goals to the specific implementation details, a systems architect could individually manage the otherwise unmanageable complexity of a large software development project. The use of top-down programming techniques enabled the authoritarian chief programmer to maintain the “architectural integrity” that Brooks believed was so central to the design of useful and beautiful software programs. The hey-day of the structured programming movement was coincident with the publication of *The Mythical Man-Month*, and the attractiveness of the “surgical team” approach

⁶² Frederick P. Brooks, “No Silver Bullet: Essence and Accidents of Software Engineering,” *IEEE Computer*, April, 1987.

to management was reinforced by, and helped reinforce, the popularity of structured programming as a development technology.

In addition to borrowing heavily from the established techniques and technologies of structured programming, the chief programmer model also help define technological innovations of its own. The development support library (DSL) was a system of documents and procedures that provided for the "isolation and delegation" of secretarial, clerical, and machine operations (see Figure 2.5 for a structural overview of the DSL).⁶³ Basically, the DSL was a set of technologies (including coding sheets, project notebooks, and computer control cards) that facilitated communications within the development team. The DSL was envisioned as a means of further centralizing control in the hands of the chief programmer:

The DSL permits a chief programmer to exercise a wider span of control over the programming, resulting in fewer programmers doing the same job. This reduces communications requirements and allows still more control in the programming. With structured programming, this span of detailed control over code can be greatly expanded beyond present practice; the DSL plays a crucial role in this expansion.⁶⁴

By providing a core set of public programs and documents that were highly visible to all members of the "surgical team," the DSL was supposed to

⁶³ F. Terry Baker and Harlan Mills, "Chief Programmer Teams," *Datamation* 19, 12 (1973), 198-199. In earlier accounts the DSL is referred to as the Programming Production Library (PPL).

⁶⁴ *Ibid*, 200.

discourage the “traditional ad hoc mystique” associated with conventional craft-oriented programming.⁶⁵ The chief programmer could read, understand, and validate all of the work done by his subordinates. The technology of the DSL was clearly intended to reinforce a conventional management agenda: the transfer of control over the work practices of programmers to the hands of the managerial “superprogrammer.” In language remarkably reminiscent of the “head versus hand” dialectic emphasized by Karl Marx and his disciples, one proponent of the chief programmer team approach described the DSL as having been “designed to separate the clerical and intellectual tasks of programming.”⁶⁶

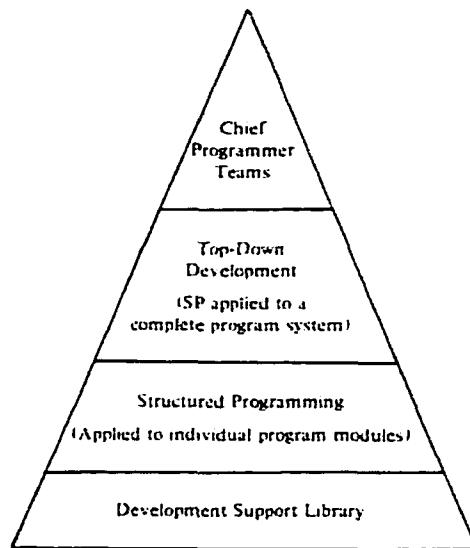


Figure 2.5: The Development Support Library

⁶⁵ Ibid, 201.

⁶⁶ Clement McGowan and John Kelly, *Top-Down Structured Programming Techniques* (New York: Petrocelli/Carter, 1975), 148.

Although the chief programmer team received much attention in the industry literature, it does not seem to have been widely or successfully implemented.⁶⁷ The original concept had been popularized by F.T. Baker in a series of articles documenting the successful implementation of the approach by IBM manager Harlan Mills. Mills had been the chief programmer in a team that developed a computerized information bank application for the *New York Times*. Mills claimed to accomplished in 22 months what a traditionally, hierarchically managed group would have required at least several more years of calendar time to develop. Baker's favorable reports on the *New York Times* project, which involved 83,000 lines of code and eleven man-years of effort, convinced many computer professionals of the scalability of the chief programmer team approach. The project was portrayed as having high productivity and low error rates, although questions later arose about the accuracy of Baker's assessment; Mills' system eventually proved unsatisfactory and was replaced with a less ambitious system.⁶⁸ For the time being, however, the *New York Times* system was considered to be proof-positive of the efficiency of the chief programmer approach.

⁶⁷ B.S. Barry and J.J. Naughton, "Chief Programmer Team Operations Description," *U.S. Air Force, Report No. RADC-TR-74-300*, v. 10 (of 15), 12-13.

⁶⁸ Stuart Shapiro, "Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering," *Annals of the History of Computing* 19, 1 (1997), 25.

Several objections to the chief programmer team approach were raised in the contemporary industry literature, however. The first is that it was difficult to find individuals with enough talent and energy to fulfill all of the functions required of the chief programmer.⁶⁹ The few who did exist were very expensive, and were not interested in working on small computers and mundane applications. A second problem was a perceived over-dependence on key individuals implied in the chief programmer team approach: "What happens if [our Superprogrammer] snaps up a more lucrative offer elsewhere? He'll likely take our back-up programmer with him, leaving us high-and-dry."⁷⁰ A number of observers suggested that the "surgical team" model led to excessive specialization.⁷¹ The computer scientist C.A.R. Hoare derided the small-team approach as a retreat towards "to the age of the master craftsman - more fashionably known as a chief programmer."⁷² There were widespread doubts about the ability of the small-team approach to scale-up to the needs of large development efforts.

⁶⁹ Barry Boehm, "Software Engineering," *IEEE Transactions on Computers* 25, 12 (1976), 349; Edward Yourdon, ed., *Classics in Software Engineering* (New York: Yourdon Press, 1979), 63.

⁷⁰ J.L. Ogdin, "The mongolian hordes versus superprogrammer," *Infosystems* (December 1973), 23.

⁷¹ Daniel Couger and R Zawacki, "What Motivates DP Professionals?," *Datamation* 24, 9 (1978), 116-123; Richard Canning, "Issues in Programming Management," *EDP Analyzer* 12, 4 (1974), 1-14.

⁷² Anthony Hoare, "Keynote Address: Software Engineering," *3rd International Conference on Software Engineering Proceedings* (1974), 1-4.

The most revealing criticisms of the chief programmer team system, however, had to do with the ways in which the presence of an elite administrator/programmer disrupted existing patterns of managerial authority: "The CPT [chief programmer team] perpetuates the prima donna image of the programmer. Instead of bringing the programmer into the organization's fold, it isolates and alienates him by encouraging the programmer to strive for a superhero image."⁷³ The chief programmer team allowed for little participation by non-technical administrators. A 1981 textbook on *Managing Software Development and Maintenance* corrected this perceived over-dependence on technical personnel by proposing a revised chief programmer team (RCPT) in which "the project leader is viewed as a leader rather than a 'super-programmer.'"⁷⁴

...whereas the chief programmer is an expert programmer, the project leader is an expert conceptualizer, designer, and project manager, but not necessarily a "super-programmer." Because he possesses both project management and technical skills and because his programming tasks have been reassigned to other team members, he is able to direct, oversee, and review all technical functions.⁷⁵

The RCPT approach was clearly intended to address a concern faced by many traditionally trained department-level managers; namely, that top executives had

⁷³ Carma McClure, *Managing Software Development and Maintenance* (New York: Van Nostrand Reinhold, 1981), 77.

⁷⁴ *Ibid*, 77-78.

⁷⁵ *Ibid*, 86.

“abdicated their responsibility and let the “computer boys” take over.”⁷⁶ As will be shown, it was this fear of the loss of control over valuable occupational territory that most determined contemporary reactions to proposed managerial solutions to the software crisis.

Computer Programming as a Human Activity

It is clear why the hierarchical system of management appealed to traditional managers: by treating software development as just another large-scale technological manufacturing project, it posed no threat to existing relationships of power and authority. It is equally apparent why so many elite programmers and academic computer scientists preferred the chief programmer team approach, which positioned them at the highest levels of both the technical *and* managerial pyramid. What is not so obvious, however, is which of these two organizational structures the average working programmer would have supported. Neither offered much in terms of professional opportunity. The hierarchical model unapologetically attempted to make their work as routine and mechanical as possible; the chief programmer team provided a real creative outlet for a single superprogrammer only. For moderately skilled programmers attempting to establish for themselves a legitimate professional identity that would provide them with autonomy and status, both models were equally

⁷⁶ John Golda, “The Effects of Computer Technology on the Traditional Role of Management,” (MBA thesis, Wharton School, University of Pennsylvania, 1965), 34

uninviting. What was needed was an alternative organizational model that could simultaneously support two seemingly contradictory agendas: increased managerial control over the "irrational" programming process, and ongoing support for the independent professional authority of programmers.

In 1969, the programmer and computing consultant Gerald Weinberg published *The Psychology of Computer Programming*. The book claimed to present the first detailed empirical study of computer programming as a complex human activity, and indeed, although Weinberg was neither a psychologist nor ethnographer, his observations appear to be remarkably accurate and insightful.⁷⁷ At the very least his work was well received by practitioners, whose personal experiences seem to have resonated with the anecdotes provided by Weinberg. *The Psychology of Computer Programming* has been widely cited as an accurate description of what really went on in actual programming projects.

Weinberg's book did more than simply describe existing attitudes and practices, however. It also proposed a new method for organizing and managing teams of software developers. The problem with existing hierarchical methods of software production, according to Weinberg, was that they encouraged programmers to become "detached" from the social environment - and overly

⁷⁷ At the time, Weinberg was serving as a programming instructor at the Institute for Advance Technology at SUNY Binghamton.

possessive of their software. When programmers invest so much of themselves in their programs, Weinberg suggested, they lose the ability to evaluate their creations objectively. The immediate result was bad software – and ultimately a software crisis. “Programmers, if left to their own devices, will ignore the most glaring errors in their output – errors that anyone else can see in an instant.”⁷⁸ The solution to the crisis provoked by “property-oriented” programming, argued Weinberg, was the adoption of the “egoless programming team”, in which every programmer is equal and where all of the code is “attached” to the team, rather than to the individual. By opening up the programming process to self-reflection and criticism, the egoless (or adaptive) programming model would increase efficiency, eliminate errors, and enhance communication – all without inhibiting the creative abilities of programmers.

Although egoless programming represented a relatively radical departure from traditional software development methodologies, it was predicated on fairly conventional notions about the nature of programming ability. There was little doubt, according to Weinberg, that the majority of people in programming were “detached” personality types who preferred to be left to themselves. This tendency towards detachment was reinforced “both by personal choice and because hiring policies for programmers are often directed toward finding such

⁷⁸ Weinberg, *The Psychology of Computer Programming*, 56.

people."⁷⁹ This detachment from people often led programmers to become excessively attached to their products. The "abominable practice" of attaching their names to their software (as in Jules' Own Version of the International Algebraic Language, better known as the JOVIAL programming language) offered evidence of the programmer's inability to disassociate themselves from their creations.⁸⁰ This proprietary sense of "ownership" on the part of the creator was not necessarily an unusual or even undesirable tendency - after all, artists "owned" paintings, authors "owned books," and architects "owned" buildings. In many cases these attributions led to the admiration and emulation of good workers by lesser ones. What was different about computer programs, however, was that they were "owned" exclusively by their creators. Good programs, unlike good literature, were never read by anyone other than the author. Thus, according to Weinberg, "the admiration of individual programmers cannot lead to an emulation of their work, but only to an affectation of their mannerisms."⁸¹ Junior programmers were unable to benefit from the wisdom and experience of

⁷⁹ Ibid, 53.

⁸⁰ The JOVIAL programming language was created for the United States Air Force in the late 1950s by the System Development Corporation (SDC). As it was to be a variant of the International Algebraic Language (eventually renamed ALGOL), it was suggested that it be called OVIAL (Our Own Version of the International Algebraic Language). Since OVIAL apparently had "a connotation relative to the birth process that did not seem acceptable to some people," the name was soon changed to JOVIAL. It was later decided that the "J" in JOVIAL would stand for Jules Schwartz, one of the programmers involved in the project. Hence, "Jules' Own Version of the International Algebraic Language."

⁸¹ Ibid.

their superiors. The only thing available to emulate was their mannerisms. The result was the perpetuation of bad work habits and personal eccentricities - "the same phenomenon we see in 'art colonies,' where everyone knows how to look like an artist, but few, if any, know how to paint like one."⁸²

Weinberg believed that the use of small, unstructured programming teams and regular code reviews would alleviate the problem of programmer "attachment." Each of the programmers in the group would be responsible for reading and reviewing all of the application code. Errors that were identified during the process were simply "facts to be exposed to investigation" with an eye towards future improvement, rather than personal attacks on an individual programmer.⁸³ By restructuring the social environment of the workplace, and thereby restructuring the value system of the programmers, the ideal of "egoless" programming would be achieved. The result would be an academic style, "peer review" system that would encourage high standards, open communication, and ongoing professional development. Junior programmers would be exposed to good examples of programming practice, and more senior developers could exchange subtle tricks and techniques. A piece of completed code would not be considered the product of an individual team member but

⁸² Ibid.

⁸³ Ibid, 57.

rather of the team as a whole. The openness of this process would also encourage the development of proper documentation.

There were a number of other salient features of the "egoless" (or adaptive) programming team that differed from conventional team-oriented approaches. The most unusual and significant was that all major design and implementation decisions were to be determined by consensus, rather than decree. There were no assigned team leaders, at least not in the conventional sense. Leadership shifted between team members based on the needs of the moment and the strength of the individual team members (hence the term "adaptive"). For example, if a particular phase of the project involved a lot of debugging, one of the team members especially skilled at debugging might assume the temporary role of team leader during that period. Even then, all of the important decisions would be made democratically. Work was assigned based on the strengths – and preferences – of the individual team members.

The democratic approach to software project management offered a number of advantages, according to Weinberg. It encouraged communication and flexibility. Schedule and design changes could be more readily accommodated, and resources could be allocated efficiently. Secondly, the lack of a formal hierarchy made the adaptive team significantly more robust than more structured alternatives. For example, the adaptive team could readily adjust to

the addition or removal of members. The success of the project would no longer hinge on the presence of any one particular individual. In an era in which the performance of programmers was believed to vary dramatically from programmer to programmer, and when turnover in the software industry averaged upwards of twenty-five percent annually, this was an appealing benefit. Last but not least, the social dynamics of the democratically-managed adaptive team appeared to correspond well with the actual experiences and expectations of the average working programmer.⁸⁴ Weinberg provided a great deal of anecdotal evidence suggesting that programmers worked best in environments in which they participated in all aspects of project development, from design to implementation to testing. By eliminating the things that caused programmers to become dissatisfied, turnover could be reduced significantly. The adaptive team approach to programming, argued Weinberg, was not only cost-effective and efficient – it kept the programmers happy. And, of course, happy programmers were productive programmers.

Like the chief programmer team and the hierarchical system of management, egoless programming constituted a solution to a specific conception of the burgeoning software crisis. The advocates of the adaptive team approach shared with many of their contemporaries certain basic assumptions about the nature of

⁸⁴ Ogdin, "The Mongolian Horde versus Superprogrammer," 23.

programming as a skill and activity: namely, that programming was an essentially creative undertaking; that individual programmers varied enormously in terms of style and productivity; and that current programming practices resembled craft more than they did science. They also believed that, despite these exceptional characteristics, software development was an activity that could, to a certain extent, be managed and controlled. What was unusual about the adaptive team solution was the degree to which it offered computer programmers a legitimate career path and an attractive professional identity.

In the hierarchical system of management, programmers were generally regarded as technicians rather than as professionals. The few programmers who did rise through the hierarchy did so by abandoning their technical interests in favor of managerial careers. The chief programmer team offered status and authority only to a small corps of elite "superprogrammers." All but the most talented individuals served as much less privileged support personnel. As will be seen, many programmers were extremely concerned with issues of professional development, both as they related to themselves as individuals and to their larger disciplinary community. The journal articles, job advertisements, and letters to the editor from this period show that many programmers were worried about becoming "dead-ended" in purely technical positions.

Hierarchical organizations and chief programmer teams did not offer them an attractive model of professionalization.

The adaptive team approach, in comparison, offered promising career opportunities to wide range of software workers. The goal of the adaptive team was to foster a "family" atmosphere in which every member's contributions were important. Team members were anything but interchangeable units. Programmers could cultivate their technical skills and advance their careers without feeling pressure to transfer into administration. As one knowledgeable observer suggested, in the adaptive team approach "a good programmer does not get further and further away from programmers, as occurs in a hierarchical structure when he moves up the management ladder. Instead, he stays with programming and gravitates toward what he does best."⁸⁵

Judging from the response it received in the industry literature, *The Psychology of Computer Programming* appealed to a broad popular audience.⁸⁶ Weinberg's anecdotes about the real-life work habits of programmers rang true to many practitioners. His descriptions of the mischievous pranks that programmers played on their managers, for example, or of the social significance of a strategically located Coca Cola dispenser, captured for many of his readers the essential character of the programming profession. The book has remained

⁸⁵ Richard Canning, "Issues in Programming Management," *EDP Analyzer* 12, 4 (1974), 6

⁸⁶ J. Hirschfelder, *Computing Reviews* (1999).

in continuous publication since 1969, and was recently celebrated by the industry journal *Datamation* as "the best book on computer programming ever written."⁸⁷

Weinberg presented a romantic portrayal of software development that emphasized the quiet professionalism of skilled, dedicated programmer-craftsmen. Of the many models for "software engineering" that were suggested in the late 1960s and early 1970s, the "egoless" programmer was by far the most attractive to the average practitioner.

The popularity of egoless programming extended beyond the community of practitioners, however. Weinberg's theories about the efficiency of small "family" work-groups and "bottom-up" consensus decision-making resonated with certain popular contemporary management theories. In 1971, Antony Jay's *Corporation Man* provided an ethological analysis of "tribal behavior" in modern corporations that reinforced Weinberg's conclusion that six to ten member teams were a "natural" organizational unit.⁸⁸ Douglas MacGregor's *The Human Side of Enterprise* (1960) discriminated between the Theory X approach to management, which assumed that because of their innate distaste for regimented labor, most employees must be controlled and threatened before they would work hard enough, and the Theory Y belief that the expenditure of physical and mental effort in work is as natural as play or rest, and that the

⁸⁷ *Datamation* review cited on www.geraldweinberg.com.

⁸⁸ Antony Jay, *Corporation Man* (New York: Random House, 1971)

average man learns, under proper conditions, not only to accept but to seek responsibility.⁸⁹ For the supporters of Theory Y management, Weinberg's adaptive team represented an exemplary model of the participative problem solving approach.⁹⁰

The concept of egoless programming was rarely adopted *in toto*, however. In later descriptions of the chief programming team, Baker and Mills claimed that their system represented a form of egoless programming, in the sense that the code produced by the chief programmer was open for inspection by other members of the surgical team.⁹¹ In this case, the adaptive team terminology seems to have been adopted for public relations purposes only. The whole point of the chief programming team was to consolidate all aspects of design and implementation into the hands of a single "superprogrammer." It would have been impossible to maintain the level of "architectural integrity" desired by Brooks if the chief programmer were not heavily invested in his own individual conceptual structure.

Indeed, by the middle of the 1970s the language of egoless programming appears to have been almost entirely transformed and co-opted by conventional

⁸⁹ Douglas MacGregor, *The Human Side of Enterprise* (New York, McGraw-Hill, 1960)

⁹⁰ Ogdin, "The Mongolian Horde versus Superprogrammer," 23.

⁹¹ F. Terry Baker and Harlan Mills, "Chief Programmer Teams." In many of the later management-oriented texts, "egoless" programming meant that programmers should not be defensive about code-reviews, task assignments, and other management imposed structures.

managers. These managers picked up on the idea that requiring programmers to develop open, non-propriety code allowed for increased administrative oversight. To them, egoless programming meant that "all programmers were to adhere to rules that would make their products understandable to others and make the individual programmer replaceable."⁹² Weinberg's original intention that egoless programming would enable programmers to develop as autonomous professionals appears to have gone entirely by the wayside. One management consultant reminded his audience that managers should "stress the nonpunitive nature of the new approaches. Egoless programming is designed to help the programmer, not point out his faults..."⁹³ The not-so-subtle subtext of this reminder is that by this period egoless programming had acquired a reputation for being worker-hostile management jargon.

Although *The Psychology of Computer Programming* received a great deal of popular attention for its descriptive verisimilitude, it was less successful in its prescriptive capacity. Weinberg's recommendations do not appear to have been taken seriously by many academic or industry leaders. It may be that his adaptive teams did not scale well to large development efforts, and were used in nothing but small local projects. They may have proven inefficient or difficult to

⁹² Bo Sanden, "Programming masters break out of the managerial mold," *Computerworld* (June 16, 1986).

⁹³ Henry S. Lucas, "On the failure to implement structured programming and other techniques," chap. in *Proceedings of 1975 ACM Annual Conference* (New York: Association for Computing Machinery, 1975)

implement, although there is evidence that the use of informal, unstructured programming teams was standard practice in the industry. At least one author rejected the adaptive team approach because it failed to provide adequate mechanisms for formal managerial control.⁹⁴ It seems likely that this last objection was what ultimately proved fatal to Weinberg's proposal. The adaptive team approach reinforced the notion that programmers were independent professionals. It shifted organizational control and authority away from managers. It ceded valuable occupational territory to a group whose institutional power-base had not yet been firmly established. Weinberg's adaptive teams were unappealing to everyone but programmers, and programmers did not have the leverage to push through such an unpopular agenda.

III. Programmers, Evolution, and the Struggle for Occupational Territory

There are several possible interpretations of the dramatic turn towards managerial solutions to the software crisis that occurred in the late 1960s. The conventional wisdom espoused in most software engineering textbooks (and in the very few historical treatments of this era) is that this sea-change was driven solely by an economic imperative. In the internal language of the discipline, an "inversion in the hardware-software cost ratio curve" occurred in the mid-1960s

⁹⁴ McClure, *Managing Software Development and Maintenance*, 74-75

that clearly demanded a managerial response.⁹⁵ Put more simply, the cost of the actual computers went down at the same time that the cost of using them (developing and maintaining software) went up. By the middle of the decade the expenses associated with commercial data processing were dominated by software maintenance and programmer labor rather than equipment purchases. And since the management of labor fell under the traditional domain of the middle-level manager, these managers quickly developed a deep interest in the art of computer programming. According to this reductionist economic interpretation, the structure of the technology and the economy completely determined the course of future developments; the software crisis was (and is) as inevitable and uncomplicated as that.

An alternative explanation for these developments has been provided by the labor historians Philip Kraft and Joan Greenbaum. Building on the work of Harry Braverman and David Noble, Kraft and Greenbaum situate the history of programming in one of the grand conceptual structures of labor history: the ongoing struggle between labor and the forces of capital. In *Labor and Monopoly Capital: The Degradation of Work in the Twentieth Century*, Braverman argued that the basic social function of engineers and managers was

⁹⁵ Barry Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation* 19, 5 (1973). See also Michael Mahoney, "Software: the self-programming machine," to appear in *Creating Modern Computing*, ed. A. Aker and F. Nebeker, (New York: Oxford U.P, forthcoming).

to oversee the fragmentation, routinization, and mechanization of labor. Cloaked in the language of progress and efficiency, the process of routinization was characterized primarily as a means of disciplining and controlling a recalcitrant work force. The ultimate result was the deskilling and degradation of the worker. In his 1977 book *Programmers and Managers: The Routinization of Computer Programming in the United States*, Kraft described a similar process at work in the computer industry:

Programmers, systems analysts, and other software workers are experiencing efforts to break down, simplify, routinize, and standardize their own work so that it, too, can be done by machines rather than people...Elaborate efforts are being made to develop ways of gradually eliminating programmers, or at least reduce their average skill levels, required training, experience, and so on...Most of the people that we call programmers, in short, have been relegated largely to subsidiary and subordinate roles in the production process... While a few of them sit at the side of managers, counseling and providing expert's advice, most simply carry out what someone else has assigned them.⁹⁶

Kraft suggested that managers have generally been successful in imposing structures on programmers that have eliminated their creativity and autonomy. His analysis was remarkably comprehensive, covering such issues as training and education, structured programming techniques ("the software manager's answer to the conveyor belt"), the social organization of the workplace (aimed at reinforcing the fragmentation between "head" planning and "hand" labor), and careers, pay, and professionalism (encouraged by managers as a means of

⁹⁶ Kraft, *Programmers and Managers*, 26-28.

discouraging unions). Joan Greenbaum followed Kraft's conclusions and methodology closely in her 1979 *In the Name of Efficiency: Management Theory and Shopfloor Practice in Data-Processing Work*. More recently, she has defended their application of the Braverman deskilling hypothesis: "If we strip away the spin words used today like 'knowledge' worker, 'flexible' work, and 'high tech' work, and if we insert the word 'information system' for 'machinery,' we are still talking about management attempts to control and coordinate labor processes."⁹⁷

There is validity to both interpretations of the changing attitude of managers towards programmers that occurred in the late 1960s. Certainly there were numerous technical innovations in both hardware and software that prompted managerial responses. It is true that many of the larger software development projects in this period did run over budget and fall behind schedule. The cost of software development relative to hardware purchases did continue to climb, and the labor cost of programming did become a serious burden to many manufacturers and users. It is also true that some managers were interested, as Kraft and Greenbaum argue, in creating "software factories" where deskilled programmers cranked out mass-produced products that required little thought

⁹⁷ Joan Greenbaum, "On twenty-five years with Braverman's 'Labor and Monopoly Capital.' (Or, how did control and coordination of labor get into the software so quickly?)," *Monthly Review* 50, 8 (1999)

or creativity.⁹⁸ One 1969 guidebook for managers captured the essence of this adversarial approach to programmer management:

Can we make a final statement that describes the successful computer manager? We shall try. He is one whose grasp of the job is reflected in simple work units that are in the hand of simple programmers; not one who, with control lost, is held in contempt by clever programmers dangerously maintaining control on his behalf.⁹⁹

An uncritical reading of this and other similar management perspectives on the process of software development, with their confident claims about the value and efficacy of various performance metrics, development methodologies, and programming languages, might suggest that Kraft and Greebaum are correct in their assessments. In fact, many of these methodologies do indeed represent “elaborate efforts” that “are being made to develop ways of gradually eliminating programmers, or at least reduce their average skill levels, required training, experience, and so on.”¹⁰⁰ Their authors would be the first to admit it. A more critical reading of this literature, however, suggests that the claims of many management theorists represent imagined ideals more than current reality.

⁹⁸ See Douglas McIlroy on “‘Mass produced’ Software Components” in Naur, et al., *Software Engineering*. The Systems Development Corporation referred to their in-house programming methodology as the “Software Factory.”

⁹⁹ Rothery, *Installing and Managing a Computer*, 152. Compare this with the famous statement of Frederick W. Taylor: “Each man must learn how to give up his own particular way of doing things, adapt his methods to the many new standards and grow accustomed to receiving and obeying directions covering details, large and small, which in the past have been left to his individual judgement.” See Taylor, *Principles of Scientific Management*, 113.

¹⁰⁰ Kraft, *Programmers and Managers*, 26.

Writing in 1971, the occupational sociologist Enid Mumford actually lauded data processing as an "area where the philosophy of job reducers and job simplifiers - the followers of Taylor - has not been accepted."¹⁰¹ A quarter century after the Garmisch conference, a *Scientific American* reviewer still complained that "The vast majority of computer code is still handcrafted from raw programming languages by artisans using techniques they neither measure nor are able to repeat consistently"¹⁰² The widely held perception that, "Excellent developers, like excellent musicians and artists, are born, not made," hardly supports the conclusion that computer programmers were degraded and routinized laborers.¹⁰³

The fact that the software crisis has survived a half-century of supposed "silver bullet" solutions suggests that Kraft may have overlooked a crucial component of this history. What is missing from his analysis is the perspective on the software labor process provided by the many companies who recognized that computer programming was, at least to a certain extent, a creative and intellectually demanding occupation, and who, in their management of software personnel stressed "the importance of a judicious balance between control and

¹⁰¹ Enid Mumford, *Job Satisfaction: A study of computer specialists* (London: Longman Group Limited, 1972), 175.

¹⁰² W. Gibbs, "Software's Chronic Crisis," *Scientific American*, September 1994

¹⁰³ Bruce Webster, "The Real Software Crisis," *Byte Magazine* 21, 1 (1996)

individual freedom.”¹⁰⁴ Kraft implied that most corporations adopted a hierarchical system of management aimed at eliminating worker autonomy. He ignored the many alternative methodologies that were proposed and adopted in this period. Like his mentors Braverman and Noble, he overemphasized the willingness and ability of the managerial “class,” which he treats as a monolithic and homogenous category, rather than as the diverse group of individuals operating in very different social, political, and technical environments, to impose unilaterally their “routinization” agenda on the programming labor force. Many programmers were skilled workers who vigorously pursued their own professional advancement; it is clear that they were active participants in the struggle to develop the discipline of software engineering.

A more nuanced reading of the contemporary industry literature suggests that the key to understanding the managerial response to the software crisis has less to do with economic imperatives or dialectical materialism than with what the sociologist Andrew Abbott has described as the “jurisdictional struggles” that occur among groups of professionals struggling for control over a particular occupational territory. In *The Systems of Professions: An Essay on the Division of Expert Labor*, Abbott provides an “ecological” model for understanding professional change and development. His model can be summarized briefly as

¹⁰⁴ Robert Head, “Controlling Programming Costs,” *Datamation* 13, 7 (1967), 141

follows: 1) professions grow when occupational niches become available to them; they change when their particular territory becomes threatened; 2) the key events in professional development are struggles over jurisdictions; key environmental changes involve the creation or abolition of jurisdictions; 3) professional struggle occurs at three levels: the workplace, culture and public opinion, and legal and administrative rules. These levels are loosely coupled. Most shifts in jurisdiction start in the workplace, move to public opinion, and may end up in the legal sphere. 5) The most consequential struggles are over competence and theory - the core jurisdiction. Increasing abstractions allows for professional expansion, but over abstraction can dilute the core jurisdiction.¹⁰⁵

My argument is that this is just one of these jurisdictional struggles occurred on commercial computing in the late 1960s. The continued persistence of a "software crisis" mentality among industrial and government managers, as well as the seemingly unrelenting quest of these managers to develop a software development methodology that would finally eliminate corporate dependence on the craft knowledge of individual programmers, can best be understood in light of a struggle over workplace authority that took shape in the early decades of computing. In the 1950s and 1960s the electronic digital computer was

¹⁰⁵ Andrew Abbott, *The Systems of Professions: An Essay on the Division of Expert Labor* (Chicago: University of Chicago Press, 1988). My summary of Abbott borrows heavily from Paul DiMaggio's review in the *American Journal of Sociology* 95, 2. (Sep., 1989), 534-535.

introduced into the well-established technical and social systems of the modern business organization. As this technology became an increasingly important tool for corporate control and communication, existing networks of power and authority were uncomfortably disrupted. The conflicting needs and agendas of users, manufacturers, managers, and programmers all became wrapped up in highly public struggle for control over the occupational territory opened up by the technology of computing.

A New Theocracy or Industrial Carpetbaggers?

Prior to the invention of the electronic digital computer, information processing in the corporation had largely been handled by conventional clerical staffs and traditional office managers. There had been attempts by aspiring “systems managers” to leverage expertise in the technical and bureaucratic aspects of administration into a broader claim to authority over the design of elaborate custom information processing systems.¹⁰⁶ In certain cases, strong-willed executives were able to use information technology to consolidate control over lower-levels of the organizational hierarchy. For the most part, however, the

¹⁰⁶ Tom Haigh, “From Office Manager to Chief Information Officer: Managing Information Processing in American Corporations, 1917-1990.” Dissertation, University of Pennsylvania.

use of such technologies did not contribute to the rise of a class of technical professionals capable of challenging the power of traditional management.¹⁰⁷

As more and more corporations began to integrate electronic computers into their data processing operations, however, it became increasingly clear that this new technology threatened the stability of the established managerial hierarchy. Early commercial computers were large, expensive, and complex technologies that required a high-level of technical competence to operate effectively. Many non-technical managers who had adapted readily to other innovations in office technology such as complicated filing systems and tabulating machinery, were intimidated by computers – and by computer specialists. The high-tech appeal of electronic computing appealed to upper management, but few executives had any idea how to integrate this novel technology effectively into their existing social, political, and technological networks. Many of them granted their computer specialists an unprecedented degree of independence and authority.

The rising power of EDP professionals did not go unnoticed by other middle-level managers. In a 1967 essay on "The Impact of Information Technology on Organizational Control," management consultant Thomas Whisler warned his colleagues "it seems most unlikely that one can continue to hold title to the computer without assuming and using the effective power it

¹⁰⁷ JoAnne Yates, *Control Through Communication: The Rise of System in American Management* (Baltimore: Johns Hopkins University Press, 1989).

confers.”¹⁰⁸ A decade earlier, Whisler and his colleague Harold Leavitt had coined the term “information technology,” and had predicted that within thirty years the combination of management science and information technology would decimate the ranks of middle management and lead to the centralization of managerial control.¹⁰⁹ His 1967 article suggested that EDP specialists were the direct beneficiaries of such centralization, which occurred at the expense of traditional managers. He quoted one insurance executive who claimed that “There has actually been a lateral shift to the EDP manager of decision-making from other department managers whose departments have been computerized.” Another manager complained about the relative decline of managerial competence in relationship to computer expertise:

The supervisor...has been replaced as the person with superior technical knowledge to whom the subordinates can turn for help. This aspect of supervision has been transferred, at least temporarily, to the EDP manager and programmers or systems designers involved with the programming...underneath, the forward planning function of almost all department managers has transferred to the EDP manager.¹¹⁰

Information technology, argued Whisler, “tends to shift and scramble the power structure of organizations... The decision to locate computer

¹⁰⁸ Thomas Whisler, “The Impact of Information Technology on Organizational Control,” in *The Impact of Computers on Management*, Charles Myers (Ed.) (Cambridge, MA: MIT Press, 1967), 44.

¹⁰⁹ Harold Leavitt and Thomas Whisler, “Management in the 1980’s,” *Harvard Management Review* 36, 6 (1958).

¹¹⁰ Whisler, “The Impact of Information Technology on Organizational Control.”

responsibility in a specific part of an organization has strong implications for the relative authority and control that segment will subsequently achieve."¹¹¹

Whisler was hardly alone in his assessment of the impending danger of an organizational power shift. In her 1971 book, *How Computers Affect Management*, Rosemary Stewart described how computer specialists mobilized the mystery of their technology to "impinge directly on a manager's job and be a threat to his security or status."¹¹² In his 1969 *Computer Can't Solve Everything*, Thomas Alexander emphasized the cultural differences that existed between "computer people" and business managers: "Managers... are typically older and tend to regard computer people either as mere technicians or as threats to their position and status - in either case they resist their presence in the halls of power."¹¹³ Authors Porat and Vaughan listed several deprecating titles that managers used to describe their upstart rivals, including "the new theocracy," "prima donnas," "the new breed," "industrial carpetbaggers" and "other similarly unflattering titles."¹¹⁴

It is not difficult to understand why many managers came to fear and dislike computer programmers and other software specialists. In addition to the usual

¹¹¹ Ibid.

¹¹² Rosemary Stewart, *How Computers Affect Management* (Cambridge, MA: MIT Press, 1971), 196.

¹¹³ T. Alexander, "Computers Can't Solve Everything," *Fortune* (October, 1969), 169.

¹¹⁴ Avner Porat and James Vaughan, "Computer Personnel: The New Theocracy - or Industrial Carpetbaggers," *Personnel Journal* 48, 6 (1968), 540-543.

suspicion with which established professionals generally regarded unsolicited changes in the status quo, managers had particular reasons to resent EDP departments. The unprecedented degree of autonomy that corporate executives granted to "computer people" seemed a deliberate affront to the local authority of departmental managers. "All too often management adopts an attitude of blind faith (or at least hope) toward decisions of programmers," complained one management-oriented computer "textbook." As a result of the "inability or unwillingness of top management to clearly define the objectives of the computer department and how it will be utilized to the benefit of the rest of the organization," many operational managers "expect the worse and, therefore, begin to react defensively to the possibility of change."¹¹⁵ The adoption of computer technology threatened to bring about a revolution in organizational structure that carried with it tangible implications for the authority of managers: "What has not been predicted, to any large degree, is the extent to which political power would be obtained by this EDP group. Top management...have abdicated their responsibility and let the 'computer boys' take over."¹¹⁶

There were other reasons why traditional managers felt threatened by computers and computer specialists. The continuous gap between the demand

¹¹⁵ Ibid, 542.

¹¹⁶ Golda, "The Effects of Computer Technology on the Traditional Role of Management," 34.

and supply of qualified computer personnel had in recent years pushed up their salary level faster than those of other professionals and managers. It also provided them with considerable opportunities for horizontal mobility, either in pursuit of higher salaries or more challenging positions. These opportunities were often resented by other, less mobile employees. In the eyes of many non-technical managers, the personnel more closely identified with the digital computer "have been the most arrogant in their willful disregard of the nature of the manager's job. These technicians have clothed themselves in the garb of the arcane wherever they could do so, thus alienating those whom they would serve."¹¹⁷ Deserved or otherwise, computer programmers developed a reputation for being flighty, disloyal, and arrogant.

The "Cosa Nostra" of Data Processing

There is no doubt that by the end of the decade traditional corporate managers were extremely aware of the potential threat to their occupational territory posed by the rise of computer professionals. As Michael Rose described it in his 1969 book *Computers, Managers, and Society*,

[Local departmental managers] obviously tend to resist the change. For a start, it threatens to transform the concern as they know and like it...At the same time the local's unfamiliarity with and suspicion of theoretical notions leave him ill-equipped to appreciate the rationale and benefits of

¹¹⁷ Datamation Editorial, "The Thoughtless Information Technologist," *Datamation* 12, 8 (1966).

computerization. It all sounds like dangerously far-fetched nonsense divorced from the working world as he understands it. He is hardly likely to hit it off with the computer experts who arrive to procure the organizational transformation. Genuine skepticism of the relevance of the machine, reinforced by emotional factors, will drive him towards non-cooperation.¹¹⁸

In response to this perceived challenge to their authority, managers developed a number of inter-related responses intended to restore them to their proper role in the organizational hierarchy. The first was to define programming as an activity, and by definition programmers as professionals, in such a way as to assign it and them a subordinate role as mere technicians or service staff workers. I have already described some of the ways in which the rhetoric of management literature reinforced the notion that computer specialists were self-interested, narrow technicians rather than future-minded, bottom-line-oriented good corporate citizens. "People close to the machine can also lose perspective," argued one computer programming "textbook" for managers. "Some of the most enthusiastic have an unfortunate knack of behaving as if the computer were a toy. The term 'addictive' comes to mind..."¹¹⁹ Managers emphasized the youthfulness and inexperience of most programmers. The results of early aptitude tests and personality profiles – those that emphasized their "dislike for people" and "preference for...risky activities" – were widely cited as examples of

¹¹⁸ Michael Rose, *Computers, Managers, and Society* (Harmondsworth: Penguin, 1969), 207.

¹¹⁹ Michael Barnett, *Computer Programming in English* (New York: Harcourt, Brace & World, 1969), 5.

the "immaturity" of the computer professions. In fact, one of the earliest and most widely cited psychological profiles of programmers suggested that there was a negative correlation between programming ability and interpersonal skills.

¹²⁰ The perception that computer programmers were particularly anti-social, that they "preferred to work with things rather than people," reinforced the notion that programming was an inherently solitary activity, ill suited to traditional forms of corporate organization and management.

Another common strategy for deprecating computer professionals was to challenge their technical monopoly directly. If working with computers was in fact not all that difficult, then dedicated programming staffs were superfluous. One of the alleged advantages of the COBOL programming language frequently touted in the literature was its ability to be read, understood – and perhaps even written – by informed managers.¹²¹ The combination of new programming technology and stricter administrative controls promised to eliminate management's dangerous dependency on individual programmers: "The problems of finding personnel at a reasonable price, and the problem of control, are both solved by good standards. If you have a set of well-defined standards you do not need clever programmers, nor must you find yourself depending on

¹²⁰ Dallis Perry and William Cannon, "Vocational Interests of Computer Programmers," *Journal of Applied Psychology* 51, 1 (1967).

¹²¹ Robert Gordon - "Personnel Selection" in Fred Gruenberger and Stanley Naftaly, eds., *Data Processing. Practically Speaking* (Los Angeles: Data Processing Digest, 1967), 85.

them.”¹²² At the very least, managers could learn enough about computers to avoid being duped by the “garb of the arcane” in which many programmers frequently clothed themselves.¹²³ At West Point, cadets were taught enough about computers to prevent them from “being at the mercy of computers and computer specialists...we want them to be confident that they can properly control and supervise these potent new tools and evaluate the significance of results produced by them.”¹²⁴

The idea that the so-called “software crisis” could largely be attributed to mismanagement by technicians served a dual-purpose for traditional middle-level managers. First of all, it placed them solidly in the role of corporate champion. Many of the most prominent software engineering methodologies developed in the immediate post-Garmisch conference era were management-related or –driven. Secondly, this particular construction of the software crisis provided an unflattering image of the computer specialists vis-à-vis management. By representing programmers as short-sighted, self-serving technicians, managers reinforced the notion that they were ill-equipped to handle “big picture,” mission-critical responsibilities. After all, according to the McKinsey reports, “Only managers can manage the computer in the best

¹²² Rothery, *Installing and Managing a Computer*, 83.

¹²³ Datamation Editorial, “The Thoughtless Information Technologist,” *Datamation* 12, 8 (1966), 21-22.

¹²⁴ *Ibid.*

interests of the business."¹²⁵ And not just any managers would do: only those managers who had traditional business training and experience were acceptable, since "managers promoted from the programming and analysis ranks are singularly ill-adapted for management."¹²⁶

Experienced managers stressed the critical differences between "real-world problems" and "EDP's version of real-world problem."¹²⁷ The assumptions about programmers embedded in the infamous McKinsey reports – that they were narrowly-technical, inexperienced, and "poorly qualified to set the course of corporate computer effort" – resonated with many corporate managers.¹²⁸ They provided a convenient explanation for the burgeoning software crisis. Managers, had in effect, "abdicated their responsibility and let the 'computer boys' take over."¹²⁹ The fault was not entirely the manager's own, however. Calling electronic data processing "the biggest ripoff that has been perpetrated on business, industry, and government over the past 20 years," one author suggested that business executives have been actively prevented "from really bearing down on this situation by the self-proclaimed cloak of sophistication and mystique which falsely claims immunity from normal management methods.

¹²⁵ McKinsey & Company, "Unlocking the Computer's Profit Potential," 33.

¹²⁶ Ogden, "The mongolian hordes versus superprogrammer," 20.

¹²⁷ Larson, "EDP - A 20 Year Ripoff!", *Infosystems* (November 1974), 28.

¹²⁸ Datamation Editorial, "Trouble ... I Say Trouble, Trouble in DP City," *Datamation* 14, 7 (1968)

¹²⁹ Golda, "The Effects of Computer Technology on the Traditional Role of Management," 34.

They are still being held at bay by the computer people's major weapon - the snow job."¹³⁰ Computer department staffs, although "they may be superbly equipped, technically speaking, to respond to management's expectations," are "seldom strategically placed (or managerially trained) - to fully assess the economics of operations or to judge operational feasibility."¹³¹ Only the restorations of the proper balance between computer personnel and managers could save the software projects from a descent into "unprogrammed and devastating chaos."¹³²

In much of the management literature of this period, computer specialists were often cast as self-interested peddlers of "whiz bang" technologies. "In all too many cases the data processing technician does not really understand the problems of management and is merely looking for the application of his specialty."¹³³ In the words of one Fortune 500 data processing executive:

They [EDP personnel] don't exercise enough initiative in identifying problems and designing solutions for them...They are impatient with my lack of knowledge of their tools, techniques, and methodology - their mystique; and sometimes their impatience settles into arrogance...In sum, "These technologists just don't seem to understand what I need to make decisions."¹³⁴

¹³⁰ Harry Larson, "EDP - A 20 Year Ripoff!," 26.

¹³¹ D. Herz, *New Power for Management* (New York: McGraw-Hill, 1969), 169.

¹³² Robert Boguslaw and Warren Pelton, "Steps: A Management Game for Programming Supervisors," *Datamation* 5, 6 (1959), 13-16.

¹³³ W.R. Walker, "MIS Mysticism (letter to editor)," *Business Automation* 16, 7 (1969), 8.

¹³⁴ *Datamation* Editorial, "The Thoughtless Information Technologist," *Datamation* 12, 8 (1966), 21-22.

The 1969 book *New Power for Management* emphasized the myopic perspective of programmers: "For instance, a technician's dream may be a sophisticated computerized accounting system; but in practice such a system may well make no major contribution to profit."¹³⁵ Others attributed to them even more Machiavellian motives: "More often than not the systems designer approaches the user with a predisposition to utilize the latest equipment or software technology - for his resume - rather than the real benefit for the user."¹³⁶ Calling programmers the "Cosa Nostra" of the industry, the colorful former-programmer turned technology management consultant H.R.J. Grosch warned managers to "refuse to embark on grandiose or unworthy schemes, and refuse to let their recalcitrant charges waste skill, time and money on the fashionable idiocies of our racket."¹³⁷ Like many of his management-oriented colleagues, he argued that programmers needed to "accept reality, not to rebel against it." Many of the technological, managerial, and economic woes of the software industry became wrapped up in the problem of programmer management.

It is possible to overemphasize the degree to which traditional managers were hostile to programmers and other software specialists. Kraft and Greenbaum go too far in their portrayal of computer programmers as an

¹³⁵ Herz, *New Power for Management*, 169.

¹³⁶ H.L. Morgan and J.V. Soden, "Understanding MIS Failures," *Data Base* (Winter 1973), 159.

¹³⁷ Herb Grosch, "Programmers: The Industry's Cosa Nostra," *Datamation* 12, 10 (1966), 202.

oppressed and degraded labor force. Many data processing managers sincerely believed that the best way to “rationalize” software develop was to implement strict managerial controls on programming practices. A number of them had technical backgrounds and identified themselves more as programmers than as managers. Most of them would never characterize their relationship with software developers as openly and deliberately antagonistic. Nevertheless, it is clear that the introduction of computer technology into the corporate environment caused a realignment of established networks of power and authority. During this period of realignment many programmers actively pursued a professional agenda aimed at providing themselves with as much occupational territory as possible.

Chapter Three: The Professionalization of Programming

In one inquiry it was found that a successful team of computer specialists included an ex-farmer, a former tabulating machine operator, an ex-key punch operator, a girl who had done secretarial work, a musician and a graduate in mathematics. The last was considered the least competent.¹

H.A. Rhee, *Office Automation in Social Perspective* (1968)

In the development of professional standards, the computer field must be unrelenting in advocating stringent requirements for professional status, whether these include education, experience, examination, character tests, or what not ...²

"The Making of a Profession," *Communications of the ACM* (1961)

I. *The Humble Programmer*

The first computer programmers came from a wide variety of occupational and educational backgrounds. Some were former clerical workers or tabulating machine operators. Others were recruited from the ranks of the female "human computers" who had participated in wartime manual computation projects. Most, however, were erstwhile engineers and scientists recruited from military and scientific hardware development projects. For these well-educated computer "converts," it was not always clear where computer programming stood in relation to more traditional disciplines. In the early 1950s, the academic discipline that we know today as computer science existed only as a loose

¹ H.A. Rhee, *Office Automation in Social Perspective: The Progress and Social Implications of Electronic Data Processing* (Oxford: Basil Blackwell, 1968), 118.

² C.M. Sidlo, "The Making of a Profession (letter to editor)," *Communications of the ACM* 4, 8 (1961), 366.

association of institutions, individuals, and techniques. Although computers were increasingly used in this period as *instruments* of scientific production, their status as legitimate *objects* of scientific and professional scrutiny had not yet been established. Those scientists who left “respectable” disciplines for the uncharted waters of computer science faced self-doubt, professional uncertainty, and even ridicule. The physicist-turned-programmer Edsger Dijkstra recalled this difficult transformation in his 1972 Turing Award Lecture (revealingly entitled, “The Humble Programmer”):

...I had to make up my mind, either to stop programming and become a real, respectable theoretical physicist, or to carry my study of physics to formal completion only, with a minimum of effort, and to become...what? A programmer? But was that a respectable profession? After all what was programming? Where was the sound body of knowledge that could support it as an intellectually respectable discipline? I remember quite vividly how I envied my hardware colleagues, who, when asked about their professional competence, could at least point out that they knew everything about vacuum tubes, amplifiers and the rest, whereas I felt that, when faced with that question, I would stand empty-handed.³

Although many business programmers in this period did not share Dijkstra’s scientific and mathematical aspirations, most would have agreed with his

³ Edsger Dijkstra, “The Humble Programmer,” in *ACM Turing Award Lectures: The First Twenty Years, 1966-1985* (New York: ACM Press, 1987). When Dijkstra applied for a marriage license in his native Holland, some years later, he was rejected on the grounds that he had listed his occupation as ‘programmer,’ which was not a recognized, legitimate profession. Dijkstra swallowed his pride, as he tells the story, and resubmitted his application with his ‘second choice’ – theoretical physicist. Dijkstra would later become one of the strongest advocates for theoretical rigor and the use of engineering principles in computer science. His program of “structured programming” played a central role in debates over the so-called software crisis of the late 1960s.

assessment of the ambiguous status of the programming profession. Computer specialists had been working since the late 1950s to establish a unique intellectual and occupational identity. They did achieve some limited success: indeed, as the historian William Aspray has suggested, it is remarkable how rapidly computing acquired the *trappings* of professionalization in the United States: research laboratories and institutes, professional conferences, professional societies, and technical journals.⁴ Many of the structural elements of a computing profession were in place by the end of the 1950s. But the existence of professional institutions did not necessarily translate readily into widely recognized professional status. Academic computer scientists struggled to establish a legitimate – and independent – intellectual discipline based on a sound body of theoretical research. Systems analysts and business programmers worked to improve their standing within the organizational hierarchy by distancing themselves from computer operators and other “mere technicians.” Neither group was entirely successful. As one commentator suggested in a 1975 review of the computer professions, “true professional status for systems analysts and/or programmers...seems no closer today than it was ten years ago.

⁴ William Aspray, *The History of Computer Professionalization in America*, (unpublished manuscript). Emphasis mine.

It is not clear just what body of practitioners should rightly classify as professionals.”⁵

In many respects, the question of professionalism lies at the very heart of the historical construction of the software crisis. Much of the rhetoric of the crisis focused on the problem of programming labor, and on the training, recruitment, and management of programmers. Many of the themes developed in previous chapters – the development of new programming technologies, or of more “efficient” management methodologies - are closely tied to questions of professional status. If skilled programmers could be replaced by automated development tools, for example, or by more “scientific” management methodologies, then they could hardly have much claim to professional legitimacy. The question of what programming was - as an intellectual and occupational activity - and where it fit into traditional social, academic and professional hierarchies, was 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 will focus on the attempts of programmers to establish the institutional structures associated with professionalism: university curricula;

⁵ Richard Canning, “Professionalism: Coming or Not?,” *EDP Analyzer* 14, 3 (1975), 7-8.

recognized performance and safety standards; certification programs; professional associations; and codes of ethics. I argue that the professionalization of computer programming represented a potential solution to the software crisis that appealed to programmers and employers alike. The controversy that surrounded the various professional institutions that were established in this period, however, reveals the deep divisions that existed within the programming community about the nature of programming skill and the future of the programming professions.

II. The Drive Towards Professionalism

It is not difficult to understand why programmers in the 1950s and 1960s were so concerned with the process of professionalization. This was a period in which many white-collar occupations were attempting to achieve professional legitimacy. As sociologist Herbert Wilensky wrote in 1964, professionalism offered increased social status, greater autonomy, improved opportunities for advancement, and better pay.⁶ Belonging to a profession provided an individual with a "monopoly of competence," the control over a valuable skill that was readily transferable from organization to organization.⁷ Professionalism provided a means of excluding undesirables and competitors; it assured basic

⁶ Harold Wilensky, "The Professionalization of Everyone?," *American Journal of Sociology* 70, 2 (1964).

⁷ Magali Sarfatti Larson, *The Rise of Professionalism: A Sociological Analysis* (Berkeley: University of California Press, 1977).

standards of quality and reliability; it provided a certain degree of protection from the fluctuations of the labor market; and it was seen by many workers as a means of advancement into the middle class.⁸ Programmers in particular saw professionalism as means of distinguishing themselves from “coders” or other “mere technicians.”

The professionalization efforts of programmers were often encouraged by their corporate employers. Managers resisted the incursion of computer programmers and systems analysts into their traditional occupational territory by dismissing them as narrow technicians and self-serving careerists. As part of their rhetorical construction of the software crisis as a problem of programmer management, corporate managers often accused programmers of lacking professional standards and loyalties: “too frequently these people [programmers], while exhibiting excellent technical skills, are non-professional in every other aspect of their work.”⁹ Professionalism, or at least a certain form of corporate-friendly professionalism, was represented by managers as a means of reducing corporate dependence on the whims of individual programmers.¹⁰ It was also thought that professionalism might solve a number of other pressing

⁸ Robert Zussman, *Mechanics of the Middle Class: Work and Politics Among American Engineers* (Berkeley: University of California Press, 1985).

⁹ Malcolm Gotterer, “The Impact of Professionalization Efforts on the Computer Manager,” chap. in *Proceedings of 1971 ACM Annual Conference* (New York: Association for Computing Machinery, 1971), 368.

¹⁰ David Ross, “Certification and Accreditation,” *Datamation* 14, 9 (1968), 183, 186

management problems: it might motivate staff members to improve their capabilities; it could bring about more commonality of approaches; it could be used for hiring, promotions and raises, and it could help solve the perennial question of "who is qualified."¹¹

The professionalization of programming was attractive to potential employers, however, only if it encouraged good corporate citizenship. Because skilled programmers were the possessors of a "'personal monopoly' which manifests itself in the market place," increased professionalization might only exacerbate existing labor market shortages.¹² "Professionalism might well increase staff mobility and hence turnover," warned one contemporary observer, "and it probably would lead to higher salaries for the 'professionals.'"¹³ Programmers who were more loyal to their profession than to their employer, it was suggested, would use their so-called "professional prerogatives" to perpetuate an already dangerous degree of personal and professional autonomy.¹⁴

Despite these misgivings, corporate managers generally embraced the concept of professionalism. It appeared to provide a familiar solution to the increasingly complex problems of programmer management: "The concept of

¹¹ Canning, "Professionalism: Coming or Not?," 2.

¹² Roger Guarino, "Managing Data Processing Professionals," *Personnel Journal* (Dec., 1969), 972.

¹³ Canning, "Professionalism: Coming or Not?," 2.

¹⁴ Harry Larson, "EDP - A 20 Year Ripoff!," *Infosystems* (November 1974), 28-29.

professionalism," argued one personnel research journal from the early 1970s, "affords a business-like answer to the existing and future computer skills market...The professional's rewards are full utilization of his talents, the continuing challenge and stimulus of new EDP situations, and an invaluable broadening of his experience base."¹⁵ The rhetoric of professionalism was ideologically neutral, and appealed to a wide variety of individuals and interest groups.

In response to these various motivations to professionalize, programmers in the late 1950s and early 1960s worked to establish the institutional structures traditionally associated with the professions. These included the development of an academic infrastructure for supporting theoretical computer science research; support for industry-based certification and licensing programs; the establishment of professional societies and journals; the introduction of performance standards; and professional codes of ethics. Many of these institutional structures developed quite rapidly and were established on a provisional basis by the end of the 1950s.

The early adoption of the structures of professionalism conceals, however, the deep intellectual and ideological schisms that existed within the programming community. Although many practitioners agreed on the need for

¹⁵ Personnel Journal Editorial, "Professionalism Termed Key to Computer Personnel Situation," *Personnel Journal* 51, 2 (February, 1971), 156-157.

a programming profession, they disagreed sharply about what such a profession should look like. What was the purpose of the profession? Who should be allowed to participate? Who would control entry into the profession, and how? What body of abstract knowledge would be used to support its claims to legitimacy? By the beginning of the 1960s, clearly discernible factions had emerged within the nascent programming discipline. Science- and engineering-oriented programmers worked to develop a theoretical basis for their discipline. They joined associations like the Association for Computing Machinery (ACM) that published academic-style journals, imposed strict educational requirements for membership, and resisted certification and licensing programs. Business data processing personnel, on the other hand, pursued a more practice-centered professional agenda. If they joined any professional associations at all, it was the Data Processing Management Association (DPMA). They read journals like *Datamation*, which emphasized plain speech and practical relevance over theoretical rigor. The tension that existed between these two groups of aspiring professionals – the academic computer scientists and the business data processors – greatly influenced the character and fortunes of the various professional institutions that each faction supported.

III. Computer science as the key to professionalism

In September 1959, an article in the newly-founded Association for Computing Machinery (ACM) journal *Communications of the ACM* introduced to the world the outlines of a new intellectual and academic discipline: computer science. By this time there were numerous computer-related studies under way in a wide variety of academic departments at various research universities, including departments of mathematics, business and economics, library science, physics, and electrical engineering. In "The Role of the University in Computers, Data Processing, and Related Fields," Louis Fein argued that all of these activities should be consolidated into a single organizational entity. After submitting a range of possible names for this entity, including "information sciences," "intellitronics," and "synnoetics" (a term that Fein himself used on several occasions), he proposed the term "Computer Science."¹⁶ Other names for this new discipline (or its practitioners) had been suggested elsewhere in the contemporary literature: "Comptology," "Hypology," derived from the Greek root hypologi (meaning "to compute"), "Applied Epistemologist," and "Turingineer," among others. Computer science was the name that stuck.¹⁷

¹⁶ Louis Fein, "The Role of the University in Computers, Data Processing, and Related Fields," *Communications of the ACM* 2, 10 (1959), 7-14.

¹⁷ Quentin Correll, "Letters to the Editor," *Communications of the ACM* 1, 7 (1958), 2; P.A. Zaphyr, "The science of hypology (letter to editor)," *Communications of the ACM* 2, 1 (1959), 4; Editors of DATA-LINK, "What's in a Name? (letter to editor)," *Communications of the ACM* 1, 4 (1958), 6. A more complete treatment of this history

Establishing computer science as a legitimate *theoretical* discipline was clearly an essential component in the professionalization agenda of its practitioners. Within the status hierarchy of the university, theory ranked higher than practice, and was therefore desirable for its own sake. Outside of the academy, theoretical knowledge offered a potential key to professional advancement. It provided a means of distinguishing the competent professional from the mere technician. As the sociologist Andrew Abbot and others have suggested, the key to professional development is control over abstract knowledge: "Practical skill grows out of an abstract system of knowledge, and control of the occupation lies in control of the abstractions that generate the practical techniques ... Abstraction enables survival in the competitive system of professions."¹⁸ Aspiring computer professionals recognized the need to establish a more abstract and "scientific" approach to their discipline. In a 1961 letter to the editors of the *Communications of the ACM* on "The Making of a Profession," C.M. Sidlo suggested that "As a profession becomes mature it realizes that the science (not technology) needed by the profession must continually be extended to more basic content rather than restricted only to the obvious applied science. A profession is under an obligation to develop and base itself on a body of

can be found in Paul Ceruzzi, "Electronics Technology and Computer Science, 1940-1975: A Coevolution," *Annals of the History of Computing* 10, 4 (1989), 257-275.

¹⁸ Andrew Abbott, *The Systems of Professions: An Essay on the Division of Expert Labor* (Chicago: University of Chicago Press, 1988).

knowledge rather than upon a body of applications.”¹⁹ The primary distinction between professionals and technicians, argued another practitioner, “is based on whether one has undergone a ‘prolonged course of specialized, intellectual instruction and study.’ This distinction has not yet materialized in the computer programming occupation because of the diverse backgrounds, lack of competence criteria and embryonic stages of applied computer science education.”²⁰ By the end of the 1960s, formal education was seen by an increasing number of data processing personnel as a necessary prerequisite to professional status. As the software crisis heated up in the late 1960s, university computer science programs served as a resource for practitioners in their struggle for professional legitimacy. They also served as a battleground in which various groups competed for control over occupational territory.

The lack of formal theories of software development offered a convenient explanation for the burgeoning software crisis. A 1968 *Datamation* editorial on “The Facts of Life” described the inadequacies of contemporary practices: “Our youthfulness means, for one thing, that we still lack a sound theoretical base for our work. Which means it’s hard to transfer learning from one particular experience to another. Which means in turn that it’s extremely difficult to select,

¹⁹ Sidlo, “The Making of a Profession,” 367.

²⁰ Malcolm Gotterer, “The Impact of Professionalization Efforts on the Computer Manager,” chap. in *Proceedings of 1971 ACM Annual Conference* (New York: Association for Computing Machinery, 1971), 371-372.

train, and develop good edp systems people, programmers and analysts especially."²¹ The phrase "software engineering" provided the organizing principle of the 1968 Garmisch Conference and was "deliberately chosen as being provocative, in implying a need for software manufacturing to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering."²² The subsequent adoption of the "software engineering" agenda by a broad spectrum of the computing community indicates the powerful appeal that a more "scientific" approach to software development held for many practitioners, managers, and manufacturers.

"A momentary aberration in the fields of mathematics and electrical engineering..."

Although the term "computer science" was not adopted until 1959, the origins of the discipline that it encompasses stretch back at least into the late 1940s. In fact, much of the mathematical theory that underpins computer science derives from work done as early as 1854.²³ Charles Babbage had provided a sophisticated description of the principles of digital computing in the 1830s. Alan Turing published his now-famous article on the computability of numbers

²¹ Datamation Editorial, "The Facts of Life," *Datamation* 14, 3 (1968)

²² Peter Naur, Brian Randall, and J.N. Buxton, ed., *Software engineering Proceedings of the NATO conferences* (New York: Petrocelli/Carter, 1976), 7.

²³ William Aspray, "The History of Computer Professionalism in America," (unpublished manuscript).

in 1937. George Stibitz of Bell Laboratories, Konrad Zuse at the Henschel Aircraft Company in Berlin, and Howard Aiken had all built working electro-mechanical digital computers by the mid-1930s. It was not until the immediate post-war period, however, that a community of researchers dedicated to the study of computer technology began to emerge.

By the end of the 1940s, five major research universities in the United States had experience with high-speed computing: Columbia, Harvard, University of Pennsylvania, MIT, and Princeton. Columbia had been working with the IBM Corporation since the late 1920s on projects related to scientific computing. In 1945 the university established the Watson Scientific Computing Laboratory on its campus. Howard Aiken of Harvard University used IBM and U.S. Navy resources to build his Mark I electro-mechanical computer, first installed at Harvard in 1944. The University of Pennsylvania collaborated with the Ballistics Research Laboratory of the Aberdeen Proving Grounds to develop the ENIAC machine. MIT was involved with several large computation projects in the 1940s, most famously the Project Whirlwind real-time flight simulator. Princeton University inherited a stored-program computer designed by the mathematician John von Neumann for the nearby Institute for Advanced Study.

Although each of these projects was developed for very different purposes in very different institutional contexts, they were united by their focus on

hardware. Theoretical questions took a back seat to pressing technical concerns; just getting these early machines to run without failure for more than a few minutes was an engineering challenge of heroic proportions. As one of the first general textbooks on electronic computing described it, "The outstanding problems involved in making machines are almost all technological rather than mathematical."²⁴ When the Moore School of Electrical Engineering at the University of Pennsylvania held its famous lecture series on the "Theory and Techniques for Design of Electronic Digital Computers" in 1947, the emphasis was much more on techniques than on theory.

By the beginning of the 1950s, a community of researchers had begun to form around the design and use of computers. The 1947 Moore School lectures had served as a seedbed for the nascent computing community. Many of the attendees would be leaders in the field for the next several decades.²⁵ Similar one-time conferences were held at Harvard in 1947 and 1949, at IBM in 1948, at Cambridge University in 1940, and Manchester University in 1951. Although most of these conferences focused on hardware development and engineering concerns, by the end of the 1940s there were indications that the discipline was becoming much more theoretically oriented. In 1945-46 John von Neumann

²⁴ B.V. Bowden, *Faster than Thought: A Symposium on Digital Computing Machines* (London: Sir Isacc Pitman & Sons, 1953).

²⁵ Martin Campbell-Kelly and Michael Williams, eds., *The Moore School Lectures: Theory and Techniques for the Design of Electronic Digital Computers* (Cambridge, MA: MIT Press: Charles Babbage Reprint Series, 1985).

circulated an informal "First Draft of a Report on the EDVAC." The EDVAC was the planned successor to the ENIAC computer then being developed at the University of Pennsylvania. Von Neumann had become involved with the project in 1945 as an outgrowth of his work on nuclear physics at Los Alamos. His report described the EDVAC in terms of its logical structure, using notation borrowed from neurophysiology. Ignoring most of the physical details of the EDVAC design, such as its vacuum tube circuitry, von Neumann focused instead on the main functional units of the computer: its arithmetic unit, its memory, input and output. By abstracting the logical design of the digital computer from any particular physical implementation, von Neumann took a crucial first step in the development of a modern theory of computation.²⁶ The outlines of a new discipline had begun to take shape.

Over the course of the next decade, organizational and administrative boundaries were drawn around computer science research. As early as 1947 courses in digital computing were being taught at MIT and Columbia. That same year Harvard offered a one-year master's degree program in applied mathematics "with special reference to computing machinery."²⁷ In 1948 this program was expanded to include Ph.D. students. In the 1950s an increasing

²⁶ Michael S. Mahoney, "Computer Science: The Search for a Mathematical Theory", in John Krige and Dominique Pestre (eds.), *Science in the 20th Century* (Amsterdam: Harwood Academic Publishers, 1997).

²⁷ I. Bernard Cohen, *Howard Aiken: Portrait of a Computer Pioneer* (Cambridge, MA: The MIT Press, 1999), 186.

number of universities began offering courses and degree programs in computer-related specialties. Most of these degrees were actually received from mathematics or electrical engineering departments; the first doctorate in "computer science" was not awarded until 1965.²⁸ By the 1964-65 academic year, however, there were approximately 4,300 undergraduates and 1,300 graduate students in computer science, data processing, information sciences and related programs. In 1966-67, these totals jumped to over 22,000 undergraduates and 5,000 graduates, roughly a five-fold increase.²⁹ By the end of the 1960s, computer science was well on its way towards becoming an independent departmental entity in many major research universities.

This move towards institutional independence and academic respectability was closely associated with, and predicated upon, the development of computer science theory. In his 1959 manifesto announcing the new discipline, Louis Fein had been careful to distance it from its hardware-oriented origins:

Too much emphasis has been placed on the computer equipment in university programs that include fields in the "computer sciences" ... Indeed an excellent integrated program in some selected fields of the computer sciences should be possible without any computing equipment at all, just as a first rate program in certain areas of physics can exist without a cyclotron.³⁰

²⁸ William Aspray, "Was Early Entry a Competitive Advantage?," *Annals of the History of Computing* (2000), 64.

²⁹ Thomas White, "The 70's: People," *Datamation* 16, 7 (1973), 42.

³⁰ White, "The 70's: People," 11.

A 1964 report from the ACM Curriculum Committee on Computer Science echoed this notion that computer science involved more than just the design and operation of computing equipment:

There are widespread misconceptions of the purpose of computer science despite the general acknowledgement that it is a distinctive subject. Computer science is not simply concerned with the design of computing devices - nor is it just the art of numerical calculation, as important as these topics are ... Computer science is concerned with *information* in much the same sense that physics is concerned with energy; it is devoted to the *representation, storage, manipulation, and presentation* of information in an environment permitting automatic information systems. As physics uses energy transforming devices, computer science uses information transforming devices.³¹

In a 1967 letter to the editors of *Science*, Herbert Simon, Allen Newell and Alan Perlis attempted to answer the question most commonly asked of the computer scientist: "Is there such a thing as computer science, and if there is, what is it?" Computer science, according to Simon and his colleagues, was quite simply the study of computers, just as astronomy was the study of stars and biology the study of life. The fact that the computer was an artificial rather than a natural phenomenon was irrelevant: as Simon would later argue in *The Sciences of the Artificial*, artifacts were perfectly legitimate objects of empirical research. "Computer science is the study of the phenomena surrounding

³¹ ACM Curriculum Committee, "An Undergraduate Program in Computer Science - Preliminary Recommendations," *Communications of the ACM* 8, 9 (1965), 544.

computers ... the computer is not just an instrument but a phenomenon as well, requiring description and explanation."³²

As the historian Paul Ceruzzi has suggested, by the end of the 1960s most computing theorists had adopted the definition of their discipline that has since come to dominate modern computer science, at least as it is practiced in the university: computer science is the study of algorithms.³³ Implied in this definition is the notion that the algorithm is as fundamental to computing as Newton's Laws of Motion are to physics. By founding their discipline on the algorithm rather than on engineering practices, computer scientists could claim fellowship with the sciences: computer science was science because it was concerned with discovering natural laws about algorithms. As Peter Wegener described in his 1970 "Three Computing Cultures," "The notion of a mechanical process and of an algorithm (a mechanical process which is guaranteed to terminate) are as fundamental and general as the concepts that underlie the empirical and mathematical sciences."³⁴ In his 1968 classic *Fundamental Algorithms*, computer scientist Donald Knuth attempted to situate "the art of programming" on a firm foundation of mathematical principles and theorems. That same year the Association for Computing Machinery released their

³² Herbert Simon, Allen Newell, and Alan Perlis, "Computer Science (letter to editor)," *Science* 157, 3795 (Sept. 22, 1967), 1373-1374.

³³ See Ceruzzi, "Electronics Technology and Computer Science."

³⁴ Peter Wegener, "Three Computer Cultures: Computer Technology, Computer Mathematics, and Computer Science," *Advances in Computers* 10 (1970), 9.

influential "Curriculum '68" guidelines, which encouraged university computer science departments to drop electronics and hardware courses in favor of mathematics and algorithms offerings.³⁵

Emerging as it did from the large hardware development projects of the 1940s, computer science was from the very beginning an interdisciplinary enterprise. This was both a strength and a weakness. All of the early recruits to the discipline had necessarily come from established departments. As William Aspray has suggested, computer science crossed virtually every academic boundary then established within the university, drawing content and people from mathematics, electrical engineering, psychology, and business.³⁶ Conflict between computer science and these older departments was inevitable. Some of the traditional disciplines felt threatened by the newcomer. At Harvard and Princeton, for example, undergraduate enrollments grew rapidly in computer science while they stagnated in other areas of applied science and engineering. At Penn and MIT, an increasing number of electrical engineering students chose to focus on computer-related subjects rather than on other areas of electrical engineering. As computer-related sub-fields began drawing resources and students from traditional disciplines, heated battles erupted over faculty slots,

³⁵ ACM Curriculum Committee, "Curriculum 68: Recommendations for Academic Programs in Computer Science," *Communications of the ACM* 11, 3 (1968), 151-157.

³⁶ Aspray, "Was Early Entry a Competitive Advantage?," 65.

graduate admissions, and courses. Its early success at attracting students and resources notwithstanding, computer science was repeatedly forced to defend its academic legitimacy.

One of the accusations frequently leveled against computer science was that is represented little more than a grab-bag of techniques, heuristics, and equipment. "Any science that needs the word 'science' in its name is by definition not a science," claimed one contemporary aphorism.³⁷ The discipline's close association with computer hardware was evoked to disparage its intellectual legitimacy:

The creation of computer science departments is analogous to creating new departments for the railroad, automobile, radio, airplane or television technologies. These industrial developments were all tremendous innovations embodied in machinery, as is the development of computers, but this is not enough for a discipline or a major academic field. A "discipline" is based upon a cohesive and consistent body of theory or theories along with a "bag of analytical" tools which are used to apply the theory. "Computer science" represents only a tool or technique without a body of cohesive and consistent theory. Therefore, it can hardly be classified as a discipline demanding a separate curriculum and an isolated program.³⁸

As Atsushi Akera has described in his study of early scientific computing efforts, many of the pioneering academic computing experts emerged out of the central computing facilities that provided computational services to other

³⁷ See Cerruzi, "Electronics Technology and Computer Science."

³⁸ Jack Carlson, "On determining CS education programs (letter to editor)," *Communications of the ACM* 9, 3 (1966), 135.

researchers. In order to take full advantage of expensive computing equipment, these facilities often served multiple departments (and in some cases, multiple universities). As personnel from these computing centers moved into newly founded computer science departments, they had difficulty shedding their image as service providers rather than legitimate researchers.³⁹ Computer science “is viewed by other disciplines as a rather easily mastered tool,” computer theorist David Parnas warned an ACM Curriculum Committee. “It is easy, in any field, to confuse the work of a technician with the work of a professional, but this is easier in computer science because a worker in another discipline will consider himself an ‘expert’ after learning to use a computer to process his data.”⁴⁰

The precarious status of computer science within the academic hierarchy was a subject of much discussion and concern within the computing community. In his 1966 presidential address to the ACM, Anthony Oettinger described an encounter with the Committee on Science and Public Policy of the National Academy of Sciences. The incident revealed what Oettinger called “numerous misconceptions about computer science within high councils of science and government,” among them that

³⁹ The best available source on this material is Atsushi Akeru, *Calculating a Natural World: Scientists, Engineers and Computers in the United States, 1937-1968* (Ph.D. dissertation, University of Pennsylvania, 1998).

⁴⁰ David Parnas, “On the preliminary report of C3S (letter to editor),” *Communications of the ACM* 9, 4 (1966), 242-243.

- 1)...the computer is just a tool and not a proper intellectual discipline;
- 2) Almost all creative computer designers and software inventors have been trained either as pure mathematicians or as experimental physicists. In other words, the really creative people in computers are those who were led into the field by a challenge of a problem in their own field;
- 3) There are not many good people in computer science as such...;
- 4) The training of faculty and students in computer usage can better be done by people in the various disciplines who have acquired computer experience, rather than by a separate cadre of computer scientists;
- 5) It is not the business of universities to train computer center managers or systems experts;
- 6) The [future potential?] of computers has been overrated, and when the current fad subsides, many universities will have ... badly overextended themselves with respect to both equipment and teaching/research commitments in computer science per se.
- 7) Computer science is not a coherent intellectual discipline but rather a heterogenous collection of bits and pieces from other disciplines...

Oettinger's report deftly summarized the most common objections raised against computer science by contemporary critics. Judging from the reaction that he provoked from the ACM membership, there was a real fear within the computer science community that their discipline was considered little more than a "momentary aberration in the fields of mathematics and electrical engineering."⁴¹ Oettinger himself later confessed to having doubts about

⁴¹ Robert Rosin, "Relative to the President's December Remarks," *Communications of the ACM* 10, 6 (1967), 342.

whether or not “computer science is a science.”⁴² Computer science was difficult to categorize, he suggested, because “on the one hand it has components of the purest mathematics and on the other hand the dirtiest of engineering.” Paul Ceruzzi has suggested that computer science as it existed in the 1950s and ‘60s was what Edward Layton referred to as an “engineering science,” meaning that it “took on the qualities of the sciences in their systematic organization, their reliance on experiment, and in the development of mathematical theory.”⁴³ Progress in the engineering sciences often occurred in the absence of any formal or useful theories. Although computer science would, by the end of the 1960s, adopt the algorithm as its principle subject of analysis, it had difficulty articulating any unifying themes or theories that would convince the academy that it was a truly scientific discipline rather than a miscellaneous collection of techniques applied to business, technology and science.

The response of the academic computer science community to accusations of insufficient theoretical rigor was understandable: they focused increasingly on those aspects of their discipline that most resembled traditional science and mathematics. Most of the model curricula proposed in this period emphasized algorithm theory and numerical analysis. Few included much in the way of

⁴² “I am not sure that computer science is a science” - Anthony Oettinger, “The Hardware-Software Complexity,” *Communications of the ACM* 10, 10 (1967), 604.

⁴³ Edward Layton cited in Ceruzzi (1989); Walter Vincenti cited in Ceruzzi (1989).

business data processing, or even practical programming courses, for that matter. While this might have been a good strategy from the point of view of academic program-building, it brought the academic computer scientists into conflict with their colleagues in industry.

"Cute programming tricks"

In 1968, Bell Laboratories research scientist Richard Hamming was awarded the Association for Computing Machinery's prestigious Turing Award. In his award lecture, entitled "One Man's View of Computer Science," Hamming addressed one of the most pressing concerns of his discipline: the relationship between theory and practice in computer science education. Although Hamming was a firm believer in the inclusion of advanced mathematics in the computer science curriculum, he criticized what he believed was an over-emphasis on theory. "At present there is a flavor of 'game-playing' about many courses in computer science. I hear repeatedly from friends who want to hire good software people that they have found the specialist in computer science is someone they do not want. Their experience is that graduates in our programs seem to be mainly interested in playing games, making fancy programs that really do not work, writing trick programs, etc., and are unable to discipline their own efforts so that what they say they will do gets done on time and in practical form." If the discipline were going to turn out "responsible, effective people

who meet the real needs of our society," Hamming suggested, computer science departments must abandon their love-affair with pure mathematics and embrace a hands-on engineering approach to computer science education. He criticized the ACM's recently released "Curriculum '68" report for its neglect of practical training and laboratory work.⁴⁴

Hamming was hardly the only member of the computing community to find fault with the increasingly theoretical focus of contemporary computer science. In the 1940s and 1950s, electronic computers were primarily a 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 often meant that a simple programming problem could quickly turn into a research excursion into algorithm theory and numerical analysis. For example, many of these machines did not have floating-point hardware, so programmers had to execute complicated calculations to ensure that the values of the variables would stay within the machine's fixed range throughout the course of the calculation. By the beginning of the 1960s, however, computer technology had become reliable and inexpensive enough to be adopted by a wide variety of commercial organizations. Most of these organizations used computers for business data processing rather than scientific analysis. The skills involved in

⁴⁴ Richard Hamming, "One Man's View of Computer Science," chap. in *ACM Turing Award Lectures: The First Twenty Years, 1966-1985* (New York: ACM Press, 1987).

programming business applications were very different from those required in scientific computing, and for many corporate employers it was not at all clear how a computer science education was relevant to software development.

As early as 1959 the consulting firm Price-Waterhouse had published a study on "Business Experience with Electronic Computing" that questioned the relevance of mathematics to real-world programming problems:

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...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.⁴⁵

In 1958, the Bureau of Labor noted the growing sense of corporate disillusionment with academic computer science: "Many employers no longer stress a strong background in mathematics for programming of business or other mass data if candidates can demonstrate an aptitude for the work. Companies have been filling most positions in this new occupation by selecting employees familiar with the subject matter and giving them training in programming

⁴⁵ 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), 82.

work."⁴⁶ A review of literature from the 1950s on the selection of computer programmers identified only those skills and characteristics that would have been assets in any white-collar occupation: the ability to think logically; to work under pressure, and to get along with people; a retentive memory, the desire to see a problem through to completion; and careful attention to detail. The only surprising result was that "majoring in mathematics was not found to be significantly related to performance as a programmer!"⁴⁷

During the course of the 1960s computer science managed to establish itself as an independent academic discipline. Nevertheless, many observers noticed that academic success did not necessarily translate into practical achievements. As the keynote speaker at a 1968 Conference on Personnel Research suggested, "we ought to help the programmer survive by proper education. But who can we look to for such education? Not the new departments of computer science in the universities...they are too busy teaching simon-pure courses in their struggle for academic recognition to pay serious time and attention to the applied work necessary to educate programmers and systems analysts for the real world."⁴⁸

The implication was that the professionalization strategies of academic computer

⁴⁶ William Paschell, *Automation and employment opportunities for office workers; a report on the effect of electronic computers on employment of clerical workers* (Bureau of Labor Statistics, 1958), 11.

⁴⁷ W.J. McNamara and J.L. Hughes, "A Review of Research on the Selection of Computer Programmers," *Personnel Psychology* 14, 1 (Spring, 1961), 41-42.

⁴⁸ Hal Sackman, "Conference on Personnel Research," *Datamation* 14, 7 (1968), 76.

scientists were very different from those of professional business programmers. The skills and abilities that were rewarded by university administrators were not necessarily valued within the corporate environment: "These four year computer science wonders are infinitely better equipped to design a new compiler that they are to manage a software development project. We don't need new compilers. We need on-time, on-budget, software development."⁴⁹

By the beginning of the 1970s the situation had worsened. "Possibly the most blatant failure of our industry has been its ineffective efforts at communicating with the academic community," argued one 1970 article on the so-called "The People Problem": "Ours is the first major industry in modern history to develop with only limited support from colleges and universities ... most colleges and universities still have not initiated degree programs leading to data processing careers. Those who do offer computer training frequently give the curriculum a scientific orientation, thus ignoring the additional skills needed by our industry."⁵⁰

Industrial employers became increasingly disgruntled with the products of the academic computer science departments. They began turning to other sources of educated practitioners. A 1972 *Datamation* survey of corporate data

⁴⁹ George DiNardo, "Software Management and the Impact of Improved Programming Technology," chap. in *Proceedings of 1975 ACM Annual Conference* (New York: Association for Computing Machinery, 1975), 288-289.

⁵⁰ J.A. McMurrer and J.R. Parish, "The People Problem," *Datamation* 16, 7 (1970), 57.

processing managers noticed "another attitude common to most of Datamation's wise men: the relative uselessness of departments of computer sciences ... and the people they are capable of turning out." For those people thinking about entering the field, the article recommended, "the consensus advice seems to be: stay out of computer sciences. Take a bachelor's degree in a technical subject, add a master's in business administration."⁵¹ Fred Gruenberger, himself a computer science educator, suggested:

Most programming managers in large corporations tell the same story repeatedly (although regrettably few people listen). Please, they say, give us well-educated MBAs, not Computer Science graduates ... While this may seem like a terribly narrow view, it has been repeatedly proven in both scientific and commercial data processing that programming can be taught to bright, well-motivated and well-educated people, but that company identification and a general feeling for "business" can almost never be taught."⁵²

Given the perceived lack of a close relationship between the needs of business and the output of the universities, the rise of computer science as an academic discipline contributed little to the professionalization of data processing. Employers look to other mechanisms for insuring the quality of their workforce, particularly professional certification exams.

⁵¹ Robert Forest, "EDP People: Review and Preview," *Datamation* 18, 6 (1972), 68.

⁵² Fred Gruenberger, "Problems and Priorities," *Datamation* 18, 3 (1972), 49.

IV. The Certified Public Programmer

In 1962 the editors of the data processing trade journal *Datamation* called for the creation of a new technical profession: the certified public programmer. The establishment of a rigorous certification program for computer personnel, they argued, would help resolve some of the "many problems" that were "embarrassingly prominent" in the software industry.⁵³ By defining clear standards of professional competency, an industry-wide certification program would serve several important purposes for the nascent programming profession. First, it would establish a shared body of abstract occupational knowledge - a "hard core of mutual understanding" - common across the entire professional community. Secondly, it would help raise the public's view of computing professionals "several impressive levels from its current stature of cautious bewilderment and misinterpretation to at least, confused respect."⁵⁴ Finally, and perhaps most significantly, it would enable computer professionals to erect entry barriers to their increasingly contested occupational territory: "With a mounting tide of inexperienced programmers, new-born consultants, and the untutored outer circle of controllers and accountants all assuming

⁵³ Editorial, "Editor's Readout: The Certified Public Programmer," *Datamation* 8, 3 (1962), 23-24.

⁵⁴ *Ibid.*

greater technical responsibility, a need for qualification of competence is clearly apparent."⁵⁵

The *Datamation* editorial coincided neatly with the announcement by the National Machine Accountants Association (NMAA) of their new Certificate in Data Processing (CDP) examination. The NMAA, which would later that year rename itself the Data Processing Management Association (DPMA), represented almost 16,000 data processing workers in the United States and Canada. The NMAA had been working since 1960 to develop the CDP exam, which represented the first attempt by a professional association to establish rigorous standards of professional accomplishment in the data processing field. According to their 1962 press release, the exam was intended to "emphasize a broad educational background as well as knowledge of the field of data processing," and to represent "a standard of knowledge for organizing, analyzing and solving problems for which data processing equipment is especially suitable." It was open to anyone, NMAA member or not, who had completed a prescribed course of academic study; who had at least three years direct work experience in punched card and/or computer installations; and had "high character qualifications."⁵⁶ The educational requirements were waived

⁵⁵ Ibid.

⁵⁶ In response to criticism from the many otherwise qualified programmers who did not have formal mathematical training or college-level degrees, the educational

through the 1965 examinations, however. The exam included 150 multiple choice questions on programming, numerical analysis, Boolean algebra, applications, elementary cost accounting, English, and basic mathematics (not including calculus). The first year it was offered, 1,048 applications took the CDP examination, 687 successfully.⁵⁷

The DPMA was not the only organization interested in establishing certification standards for data processing personnel. Employers and programmers alike were frustrated by the inability of standard selection mechanisms – such as a college-level mathematics degree - to tangibly assist in the recruitment and training of programmers.⁵⁸ “Could you answer for me the question as to what in the eyes of industry constitutes a 'qualified' programmer?” pleaded one *Datamation* reader: “What education, experience, etc. are considered to satisfy the 'qualified' status?”⁵⁹ Given this lack of agreement about what skills and educational background were appropriate for data processing personnel, certification programs promised to guarantee at least a basic level of competence. Employers viewed certification as means of screening potential

requirements were suspended until 1965. The other prerequisites – three year’s experience and “high character qualifications” – were so vague as to be almost meaningless, and seem to have been only selectively enforced.

⁵⁷ *Datamation* Report, “Certificate in Data Processing,” *Datamation* 9, 8 (1963), 59.

⁵⁸ John Hanke, William Boast, and John Fellers, “Education and Training of a Business Programmer,” *Journal of Data Management* 3, 6 (June, 1965), 38-53.

⁵⁹ John Callahan, “Letter to the editor,” *Datamation* 7, 3 (1961), 7.

employees, evaluating performance, and assuring uniform product and quality.⁶⁰ Programmers saw it as an indication of professional status, a means of assuring job security and achieving promotions, and an aid to finding and obtaining new positions.⁶¹ Furthermore, the certification of practitioners was considered one of the characteristic functions of any legitimate profession.⁶² The establishment of a successful certification program was thought to be the precondition for professional recognition.

For more than a decade the DPMA offered the only real certification option available to aspiring EDP professionals.⁶³ Although the CDP program was criticized by some as overly broad and superficial, by the end of 1965 almost seven thousand programmers had taken the exam, and the CDP appeared to be well towards becoming a widely-accepted industry certification standard. Some large firms such as State Farm Insurance, the Prudential Insurance Company of America, and the U.S. Army Corps of Engineers extended official recognition to

⁶⁰ Richard Canning, "The Question of Professionalism," *EDP Analyzer* 6, 12 (1968), 1.

⁶¹ Richard Canning, "The DPMA Certificate in Data Processing," *EDP Analyzer* 3, 7 (1965), 1-12.

⁶² Sidlo, "The Making of a Profession," 366.

⁶³ One of the few real competitors to the CDP was the Basic Programmers Knowledge Test (BPKT) developed at the University of Southern California in 1967. One of the criticisms that had been frequently leveled against the CDP was that it was broad rather than deep, and that its multiple choice format could only test for basic knowledge, not general competency. The BPKT was designed to test real-world programming proficiency, rather than aptitude or awareness. Although the early success of the BPKT prompted the DPMA to introduce its own version of a programming proficiency test, the Registered Business Programmers (RBP) examination, neither was widely embraced by the business community.

the CDP program, and the job specification for senior data processing jobs at a Washington, D.C. consulting firm called for both a college degree and a CDP.⁶⁴ The city of Milwaukee used the CDP as a means to assign pay-grades to data processing personnel.⁶⁵ In 1966, the DPMA mounted a major promotional campaign for the CDP program that included press releases, radio advertisements, and television "informercials."⁶⁶

In 1965, 6,951 individuals took the CDP examination, and another 4,000 persons completed CDP refresher courses conducted by local DPMA chapters.⁶⁷ By the end of 1975, 31,351 candidates had taken the CDP and 15,115 had been awarded the certificate.⁶⁸ Although it is difficult to find accurate employment information for software workers in this period, estimates from the Bureau of Labor indicate these 15,115 CDP recipients constituted approximately ten percent of the overall programming community.⁶⁹ Figure 3.6 shows, for the years between 1962 and 1973, the total number of candidates taking the exam, the total number of candidates who passed the exam, and the cumulative number of CDP holders.

⁶⁴ J.A. Guerrieri, "Certification: Evolution, Not Revolution," *Datamation* 14, 11 (1973), 101.

⁶⁵ DPMA Certificate Panel (1964) Charles Babbage Institute Archives, CBI 46, Box 1, Fld. 17.

⁶⁶ Charles Babbage Institute Archives, CBI 116, Box 1, Fld. 10.

⁶⁷ Jerome Geckle, "Letter to the editor," *Datamation* 11, 9 (1965), 12-13.

⁶⁸ Richard Canning, "Professionalism: Coming or Not?," *EDP Analyzer* 14, 3 (1975), 1-12.

⁶⁹ William C. Goodman, "The software and engineering industries: threatened by technological change?," *Monthly Labor Review*, Bureau of Labor Statistics, August 1996.

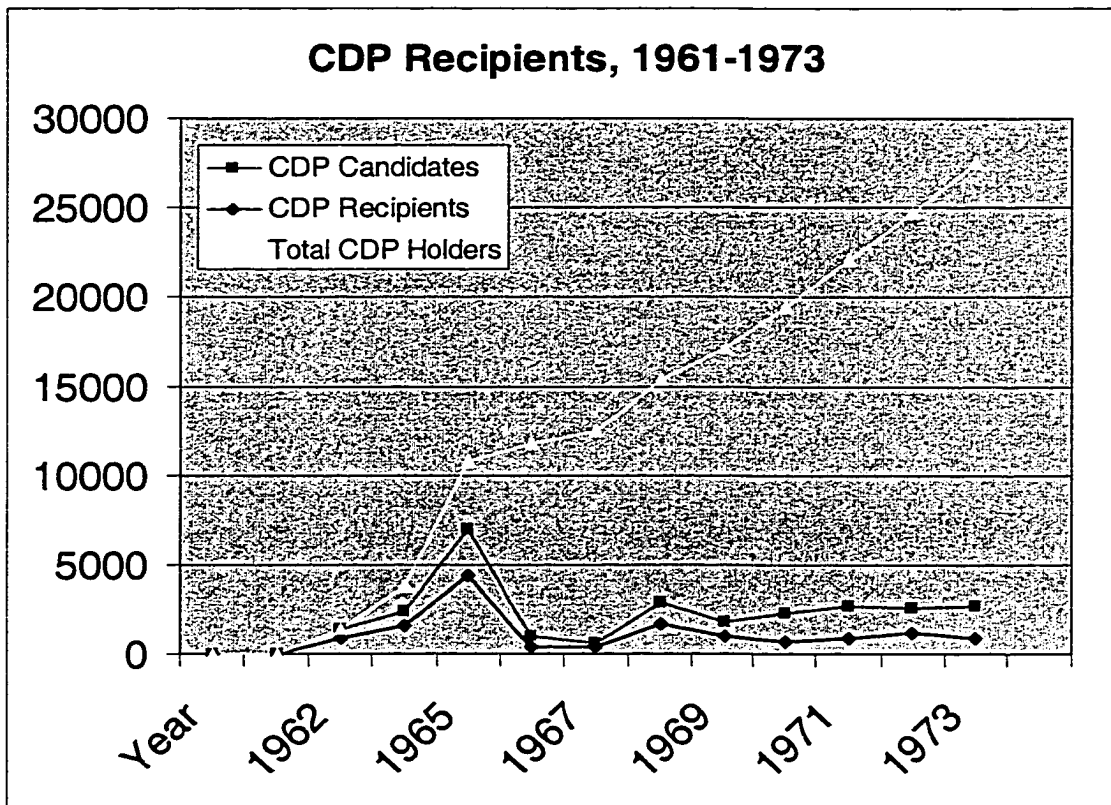


Figure 3.6: CDP Recipients, 1961-1973.

Figure 3.6 reveals the mixed fortunes and troubled history of the CDP examination. The striking early success of the program, which more than quintupled in size in its first three years, suggests that many programmers saw certification as an attractive professional strategy. This corresponds well with evidence from industry journals and other documentary sources. A survey of the 1963 candidates reveals a remarkable range of background, experience, and

education.⁷⁰ For the 1966 examination session, however, the education requirements outlined in the original 1962 program announcement were finally put into place. These requirements included specific courses in math, English, managerial accounting, statistics, and data processing systems. Whereas participation in the 1965 exam had jumped by more than three hundred percent from the previous year (possibly in anticipation of the imposition of these requirements), applications for the 1966 session dropped by almost eighty-five percent. Of the eighty-eight scheduled examination sites, twelve were dropped for lack of attendance. A major controversy erupted within the data processing community, particularly in DPMA-oriented publications such as *Datamation* and *Computerworld*.

Advocates of the academic requirements argued that such requirements not only elevated the status and legitimacy of the CDP, but were standard for most other professions, including law, medicine, engineering, and accounting. Opponents claimed that the specific course requirements were ambiguous, meaningless, and irrelevant. The DPMA Committee for Certification, which administered the CDP program, was flooded with letters from disgruntled applicants requesting special dispensation. Each case had to be individually

⁷⁰ CDP Advisory Council, Minutes of the Third Annual Meeting, Jan 17-18, 1964. Charles Babbage Institute Archives, CBI 88, Box 2, Fld. 3.

evaluated.⁷¹ In 1966 only 1,005 candidates were approved to sit for the exam. In 1967, this number dropped to 646. This posed not only financial difficulties for the DPMA, but also presented a grave threat to the perceived legitimacy of the entire CDP program. Faced with the imminent collapse of their membership support, the DPMA admitted that “the established eligibility requirements had unintentionally excluded some of the people for whom the CDP program was originally designed.”⁷² The Committee dropped the specific course requirements, providing a grandfather clause for those with three years experience prior to 1965, and requiring others to have only two years of post-secondary education. Applications for the 1968 exam session jumped back to almost three thousand.

Over the next several years, the CDP program struggled to regain its initial momentum. Annual enrollments dropped again briefly in 1969, then leveled off for the next several years at about 2,700. In an industry characterized by rapid expansion, this noticeable lack of growth represented a clear failure of the CDP program. With each year CDP holders came to represent a smaller and smaller percentage of the programming community. In 1970 the program faced yet another crisis: the announcement that a Bachelor’s degree would be required of

⁷¹ Charles Babbage Institute Archives, CBI 116, Box 1, Fld. 26.

⁷² —, “DPMA Revises CDP Test Requirements,” *Data Management* (August 1967), 34-35.

all CDP candidates, beginning with the 1972 examination. Once again a firestorm of debate broke out. The DPMA claimed that this new requirement merely reflected the changing realities of the labor market: since a college degree had already become a *de facto* requirement within the industry, requiring anything less for the CDP would severely undermine its legitimacy. The resulting controversy highlighted already existing tensions within the data processing community, and further divided the already fragmented DPMA Certification Council (many of whose own members could not satisfy the new degree requirement). Numerous observers called for the DPMA to relinquish control of the CDP examination to an independent certification authority. By the middle of the 1970s it became increasingly clear that the CDP program faced imminent dissolution.

In an attempt to restore momentum to their flagging certification initiative, the DPMA joined forces with seven other computing societies – the Association for Computing Machinery (ACM), the IEEE Computer Science Society, the Association for Computer Programmers and Analysts (ACPA), the Association for Educational Data Systems (AECS), the Automation One Association (A1A), the Canadian Information Processing Society (CIPS), and the Society of Certified Data Processors (SDCP) – to form the Institute for Certification of Computer Professionals (ICCP). The DPMA had always been extremely possessive of its

certification program, and its decision to relinquish control to an independent foundation reflects a growing sense of desperation about the future of the CDP.⁷³ The ICCP was charged with upgrading and expanding the CDP program, introducing new specialized examinations, and promoting professional development. In 1973 the ICCP took over responsibility for the CDP examinations. It also worked to develop a code of professional ethics to be adopted by its member organizations.

The ICCP failed to revive the CDP or to institute a meaningful certification program of its own. Because it represented such a wide variety of constituents, the ICCP was hindered by the same internal divisions that plagued the larger programming community. Rivalries among the constituent member societies, many of whom were only superficially committed to the concept of certification, doomed the organization to internal conflict and inactivity.⁷⁴ This failure of the various competing professional associations to cooperate crippled the ability of the ICCP to develop meaningful certification standards. No single program was able to reflect the diverse needs of the collective software community.

Furthermore, a series of highly critical assessments of the validity of the CDP

⁷³ Letter from R. Higgings, Charles Babbage Institute Archives, CBI 46, Box 2, Fld. 14.

⁷⁴ Paul Armer, "Editor's Readout: Suspense Won't Kill Us," *Datamation* 19, 6 (1973), 53.

examinations weakened popular and industry support.⁷⁵ The ICCP failed to present appealing alternative programs or examinations, and the organization languished during the 1970s.

In response to the inability of the professional associations to establish rigorous certification programs, the Society of Certified Data Processors (SCDP) adopted an approach to professional standards that circumvented the ICCP altogether: state licensing of computer professionals. The SCDP was a grass-roots organization of CDP-holders dedicated to improving the status and legitimacy of the CDP program. Founded by the self-proclaimed gadfly Kenniston W. Lord, the SCDP frequently challenged the wisdom and authority of associations such as the DPMA and ICCP. For many years, Lord and his fellow SCDP member Alan Taylor carried out a vituperative verbal campaign against the DPMA (and later the ICCP) in the pages of the weekly newspaper *Computerworld*.⁷⁶ Taylor, a popular columnist for *Computerworld*, accused the DPMA of running the CDP examinations as a profit-making enterprise rather than as an independent professional development program.⁷⁷ When the SCDP was denied formal representation in the ICCP in 1973, Lord proposed what was effectively a government takeover of responsibility for programmer certification. Unlike the

⁷⁵ R.N Reinstedt and Raymond Berger, "Certification: A Suggested Approach to Acceptance," *Datamation* 19, 11 (1973), 97-100.

⁷⁶ Charles Babbage Institute Archives, CBI 88, Box 22, Fld. 22.

⁷⁷ Charles Babbage Institute Archives, CBI 116, Box 11, Fld. 42.

certification programs voluntarily adopted by individuals and associations, however, government licensing would be mandatory. Since it is illegal to practice a licensed profession without the prior approval of the state, entry into that profession could be tightly controlled and monitored. Licensing would provide both control and protection, as well as a certain degree of public recognition and legitimacy.

In 1974, the SCDP developed a model licensing bill and submitted it to a number of state legislatures. According to its model legislation, no person in a state which passes the SCDP bill could “practice, continue to practice, offer or attempt to practice data processing or any branch thereof” without either 1) achieving a four-year degree in data processing and three years related experience or 2) successfully completing a certification examination and five years experience. The bill also provided a five-year window in which those with twelve years of experience could be “grandfathered” into the profession. Practitioners were granted a 24-month grace period in which to acquire the necessary qualifications. The legislation covered a wide variety of occupational activities and titles, including any that made use of the terms “data processing,” “data processing professional,” “computer professional,” or any of their derivatives. The state was given the power to revoke the certification of any

registrant who committed fraud, was proved guilty of negligence, or who violated the professional code of ethics.⁷⁸

The proposed SCDP legislation is notable as the only concerted attempt in this period to encourage government involvement in the programming labor market. In fact, the specter of externally imposed state regulation had often been raised as a primary justification for establishing certification programs in the first place: since self-regulation was considered to be one of the defining characteristics of a profession, surrendering control over this function to the state was essentially an admission of defeat. Observers warned that the lack of a solution from within the science would result in a solution imposed from without: "In several fields, the lack of professional and industrial standards has prompted the government to establish standards."⁷⁹ Ironically enough, even the defeat of the SCDP legislation proved humiliating to some practitioners, the state's unwillingness to legislate DP activities was perceived as a slight to the entire industry's importance and reputation.⁸⁰

Although the model SCDP legislation was adopted by none of the states to which it was submitted, the fact that it was proposed at all reveals one of the primary shortcomings of voluntary certification programs such as the CDP: the

⁷⁸ SCDP Draft Legislation (1974), Charles Babbage Institute Archives, CBI 116, Box 11, Fld. 42.

⁷⁹ David Ross, "Certification and Accreditation," *Datamation* 14, 9 (1968), 183; T.D.C. Kuch, "Unions or licensing? or both? or neither?," *Infosystems* (January 1973), 42-43.

⁸⁰ Charles Babbage Institute Archives, CBI 116, Box 11, Fld. 42.

lack of effective methods of enforcement. The inability, or unwillingness, of associations like the ACM and DPMA to self-regulate was widely criticized by industry observers. Neither group had ever taken action against a member accused of fraud or negligence, and both had reputations for being unwilling to take strong positions on issues of public interest or safety. Indeed, the DPMA was unable even to enforce the proper use of the trademark. Individuals and organizations who abused the CDP designation, either by claiming to have received a CDP when in reality they had not, or by instituting their own CDP programs, received only ineffective warning letters. No legal action appears to have been taken.⁸¹ According to SCDP president Kenniston W. Lord, the inability of the profession to regulate its own activities justified drastic action in regard to state licensing:

... one does not truly have a profession until one has the ability, legally, to challenge a practitioner and when proven guilty, to see that he is separated from the practice ... There are several sets of codes of ethics in existence, all reasonable thought out ... and all missing one key element – teeth. Or more specifically, leverage. Proved violations can lead to the removal of a certificate or the slapping of hands, but beyond that, nothing. This is one problem that the SDCP bill will solve.⁸²

The lack of ability and willingness of the DPMA to equip its certification program with “teeth” was not the only reason why the CDP failed to achieve widespread industry acceptance, however. The program had other

⁸¹ Charles Babbage Institute Archives, CBI 88, Box 18, Fld. 26.

⁸² Kenneth W. Lord, quoted in R. Canning, “Professionalism – Coming or Not?”

shortcomings as well. From almost the very beginning the examinations had been tainted by accusations of fraud and incompetent administration. In 1966 several individuals reported receiving offers from an existing CDP holder to take their examinations for them for a fee.⁸³ A copy of the 1965 exam was stolen from a locked storage cabinet at California State College, and its disappearance was covered up by the DPMA Committee for Certification.⁸⁴ Complaints about testing conditions and locations were frequent and vociferous. For example, at one examination site at the University of Minnesota, the noise caused by nearby drama club rehearsal of a sword fight scene "was so severe as to shower the room with particles of plaster."⁸⁵ Other examinees suggested that poorly trained proctors ("the little old lady who passed out the papers") were not only unable to answer even basic questions about content and procedure, but in some cases switched rooms without notice, started sessions early for personal convenience, and misplaced completed examination booklets.⁸⁶ Although such administrative snafus were hardly unique to the CDP program, they undermined public confidence in the ability of the DPMA to adequately represent the profession.

⁸³ DPMA Board of Directors, 9th Meeting, March 11-12, 1966. Charles Babbage Institute Archives, CBI 88, Box 2, Fld. 7.

⁸⁴ DPMA Board of Directors, 12th meeting, 1967 Las Vegas. Charles Babbage Institute Archives, CBI 88, Box 2, Fld. 8.

⁸⁵ *Ibid.*

⁸⁶ Charles Babbage Institute Archives, CBI 116, Box 1, Fld. 9.

Another reason why the DPMA was unable to push through its certification initiative was a lack of support from other professional associations. A 1968 article on certification and accreditation in the *Communications of the ACM* failed to mention the CDP program. This conspicuous neglect of the most successful certification program then available reflects a growing tension between the two competing professional associations. The ACM recognized that a successful certification program required a strong controlling organization. The organization that controlled certification would effectively control the profession. Indeed, the 1959 proposal that launched the CDP program suggested that "The first association to undertake a Data Processor's Certificate is going to be the leading association in the data processing field."⁸⁷ Opposed to the idea that this controlling organization could be anything but the ACM, the Executive Council of the ACM worked to undermine the efforts of the DPMA at every occasion. In 1966 the Council considered a resolution, clearly aimed at the CDP, to "warn employers against relying on examinations designed for subprofessionals or professionals as providing an index of professional competence."⁸⁸ Later that year they established a Committee to Investigate the

⁸⁷ The Certificate and Undergraduate Program (1959), Charles Babbage Institute Archives, CBI 46, Box 1, Fld. 13.

⁸⁸ Notes on ACM (1966) Charles Babbage Institute Archives, CBI 46, Box 1, Fld. 3. An early draft of this document referred specifically throughout to the "DPMA certification program." Although the final version referred only to certification programs in the abstract, the target of its attacks was obviously the CDP.

Implications of the CDP. The first order of business for the Committee was the drafting of a strongly worded objection to the use of the word "professional" in association with the DPMA exam, and the wording of subsequent exam and program literature eliminated all references to such language: CDP therefore came to stand for "Certified Data Processor," rather than "Certified Data Professional."⁸⁹ Even this modest acronym was offensive to some professional groups. A member of a SHARE (an influential IBM users group) panel on certification was "disturbed to read [the] statement that many DPMA certificate holders are beginning to use the initials "CDP" in their titles." Such pretentious behavior, he suggested, "will quickly bring down upon DPMA the wrath of other professions. It is probably illegal in some states. I fail to see how it can conceivably benefit the cause of professionalism which DPMA and others of us are working toward."⁹⁰ Although the DPMA insisted that "many persons who use the CDP initials do so more to publicize the certification program," than to promote their own personal interests, pressure from competing associations forced them to abandon many of their more ambitious claims for the CDP program.⁹¹ A 1966 statement conceded that "it would be presumptuous at this early stage in the program to suggest that CDP represents the assurance of

⁸⁹ DPMA Board of Directors, 10th meeting, 1966. Charles Babbage Institute Archives, CBI 88, Box 2, Fld. 7.

⁹⁰ Letter from Jack Yarbrough, Charles Babbage Institute Archives, CBI 46, Box 1, Fld. 17.

⁹¹ Charles Babbage Institute Archives, CBI 46, Box 1, Fld. 16.

competence, or that the Certificate should be considered as a requirement for employment or promotion in the field.”⁹² It is no wonder that so many employers and practitioners lost confidence in the ability of the DPMA to successfully administer an industry-wide certification program.

An even more troublesome problem for the DPMA was resistance from their primary constituency to their proposed educational requirements. The original CDP announcement included a list of specific academic prerequisites, including college-level courses in math, English, managerial accounting, statistics, and data processing systems, as well as eight out of seventeen possible electives.⁹³ Many of the practicing EDP specialists who formed the core of the DPMA membership saw such requirements as being irrelevant, unattainable, or both. When the educational requirements were first enforced in 1966, applications dropped by more than eighty-five percent, never to recover.

The problem was not only that the new educational requirements were overly stringent for many aspiring EDP professionals; they were also entirely too specific. What exactly counted as a math, English, or managerial accounting course? Course titles and descriptions varied greatly by institution. Each application had to be evaluated individually to determine which courses legitimately counted towards the requirement. The Committee for Certification

⁹² Charles Babbage Institute Archives, CBI 46, Box 1, Fld. 3.

⁹³ Datamation Report, “Certificate in Data Processing,” 59.

was immediately overwhelmed with paperwork: complaints, transcripts, notes from faculty, requests for exemptions, et cetera. This was in addition to the massive efforts required to assure that each candidate had the requisite three years' work experience and "high character qualifications."⁹⁴ The situation quickly turned into an administrative nightmare for DPMA officials. The specific course prerequisites were soon replaced with a more straightforward, although no less controversial, two-year college requirement. When this prerequisite was modified to a four-year degree in 1972, opposition became even more vociferous. The head of the West Tennessee chapter of the DPMA wrote to complain that he, along with about 1/3 of his chapter's membership, had suddenly become ineligible to receive the CDP. A 1970 *Computerworld* survey indicated that many practitioners felt the new requirement "unduly harsh" and "ludicrous," believing that it would decimate the data processing staffs of many smaller departments. The always outspoken Herbert R. Grosch (himself a PhD astronomer and future ACM president) declared that "This policy is very ill-advised. What the hell is so hot about college - it turns out a bunch of knuckleheads - and a knucklehead PhD is no better than a knucklehead CDP."⁹⁵

⁹⁴ It is unclear exactly what was meant by this requirement. It does appear that certain candidates were eliminated on the basis of having misrepresented their qualifications or having committed fraud or other crimes, but no written standards for the "high character qualification" seem to have existed.

⁹⁵ *Computerworld* August 19, 1970. Charles Babbage Institute Archives, CBI 116, Box 1, Fld. 27.

Despite the strong negative reaction generated by these educational requirements, the DPMA leadership continued to insist on their necessity. Such requirements had always been considered an essential component of the DPMA's professionalization program: only by defining a "standard of knowledge for organizing, analyzing, and solving problems for which data processing equipment is especially suitable" could programmers ever hope to distinguish themselves from mere technicians or other "sub-professionals."⁹⁶ Like the academic computer scientists, business programmers recognized the need for a foundational body of abstract knowledge on which to construct their profession; they differed only about what that relevant foundation of knowledge should include. In insisting on strong educational standards, the DPMA was in complete accord with the conventional wisdom of the contemporary professionalization literature.⁹⁷ And by the end of the 1960s, it was true that many employers did prefer to hire college graduates – although not necessarily computer science or data processing graduates - for entry-level programming positions.⁹⁸ According to a study published in September 1968, by the Office of Education, U.S. Department of Health, Education and Welfare, 61% of 353 business data processing managers surveyed preferred that programmers have a

⁹⁶ Alex Orden, "The Emergence of a Profession," *Communications of the ACM* 10, 3 (1967), 145-146.

⁹⁷ Sidlo, "The Making of a Profession," 367.

⁹⁸ Edward Menkhaus, "EDP: Nice Work If You Can Get It," *Business Automation* (March 1969), 43.

college degree. Over 60% indicated that education background was a substantial factor in determining a programmer's chances for promotion.⁹⁹ As a recession hit the industry in the early part of the 1970s, this trend became even more pronounced.¹⁰⁰ An aspiring EDP school graduate, even with a CDP certificate, had little chance of breaking into data processing without a college degree. As one of these individuals lamented, "They told me 80% of all programmers don't have a college degree. Now everywhere I go I'm told they're sorry but they only want college people."¹⁰¹ Although the DPMA's decision to raise the educational requirements for the CDP was highly controversial, it was also probably justified.

Ultimately, however, the DPMA never managed to convince employers and practitioners of the relevance of their educational standards, nor, for that matter, of their certification exams. Neither group was convinced that a CDP meant much in terms of future performance. The DPMA Certification Council was not even able to pass a resolution requiring its own officials to possess the CDP.¹⁰² In 1971, the Certification Council decided to drop the baccalaureate degree requirement. Although this decision was a response to pressure from within the data processing community, it was widely regarded a sign of weakness rather

⁹⁹ Thomas White, "The 70's: People," *Datamation* 16, 7 (1973), 42-43.

¹⁰⁰ Robert Forest, "EDP People: Review and Preview," *Datamation* 18, 6 (1972), 68.

¹⁰¹ Edward Markham, "Selecting a Private EDP School," *Datamation* 14, 5 (1968), 33.

¹⁰² Executive Meeting Summary (1966). Charles Babbage Institute Archives, CBI 46, Box 1, Fld. 3.

than judicious concession.¹⁰³ As the director of the computing center at Virginia Tech wrote to the president of the local DPMA chapter, “the removal of the degree requirement has forced all of us to consider the attainment of the CDP not as an extension of our normal academic and work experience, but, as a matter of fact, something quite inferior to either one.”¹⁰⁴ His letter provides a stinging but accurate indictment of the failure of the CDP program to achieve widespread acceptance and legitimacy:

My experience indicates that people seek certification from their professional peer group for only two reasons. Either it is required by law or the individual feels that the mark of acceptance stamped upon him by his peer group is sufficiently important to be worthy of the extra effort to achieve that certification. Unfortunately, in the data processing profession, many, certainly most, of the people we recognize as outstanding professional achievers and accomplisners, do not hold the CDP.¹⁰⁵

One of the major criticisms leveled against the CDP examination by employers and data processing managers was that it tested “familiarity” rather than competence.¹⁰⁶ It was not clear to many observers what skills and abilities the CDP was actually intended to certify: “the present DPMA examination measures breadth of data processing experience but does not measure depth ... It certainly does not measure or qualify programming ability. It makes no pretense

¹⁰³ Charles Babbage Institute Archives, CBI 88, Box 18, Fld. 28

¹⁰⁴ *Ibid.*

¹⁰⁵ *Ibid.*

¹⁰⁶ Canning, “The DPMA Certificate in Data Processing.”

of being any measure of management skills."¹⁰⁷ The problem was a familiar one for the industry: although most employers in this period believed that only "competent" programmers were could develop quality software, no one agreed on what knowledge and abilities constituted "competence."¹⁰⁸ As Fred Gruenberger suggested at a 1975 RAND symposium on certification issues, "I have the fear that someone who has passed the certifying exams has either been certified in the wrong things (wrong to me, to be sure) or he has been tuned to pass the diagnostics, and in either case I distrust the whole affair."¹⁰⁹ His attitude reflects the ambivalence that many observers in this period felt about contemporary DP training and educational practices. If data processing was simply a "miscellaneous collection of techniques applied to business, technology and science," rather than a unique discipline requiring special knowledge and experience, then no certification exam could possibly test for the broad range of skills associated with "general business knowledge." "Given the choice between two people from the same school, one of whom has the CDP, but the other appears brighter," Gruenberger argued, "I'll take the brighter guy."¹¹⁰

¹⁰⁷ Canning, "The DPMA Certificate in Data Processing"; Jack Yarbrough, Charles Babbage Institute Archives, CBI 46, Box 1, Fld. 17.

¹⁰⁸ Milt Stone, "In Search of an Identity," *Datamation* 18, 3 (1972), 53-54.

¹⁰⁹ RAND Symposium, "Problems of the AFIPS Societies Revisited" (1975). Charles Babbage Institute Archives, CBI 78, Box 3, Fld. 7.

¹¹⁰ *Ibid.*

Although the DPMA revised and updated its examinations annually, and eventually introduced a Registered Business Programmers (RBP) exam intended specifically for programmers, it was never able to convince the industry of the relevance of its certification programs. One DP manager suggested that the CDP was at best “a minor plus for the person who can measure up to other standards,” but that it would never be considered a “real” qualification for employment.¹¹¹ Another warned of a “lack of confidence” in the validity of the CDP exam: “I do not expect to apply for a CDP or to use the possession of a CDP as a criterion for employment.”¹¹² Still another resented a perceived attempt on the part of the DPMA to foster a “closed shop” mentality, promising to “continue to regard the CDP holder with suspicion as to motive and qualification, the level of suspicion being in inverse proportion to the date of the certificate.”¹¹³ In the absence of a strong commitment to the CDP on the part of employers, many programmers saw little benefit in participating in the program. Those who did were increasingly self-selected from the lowest ranks of the labor pool, individuals for whom the CDP was a perceived substitute for experience and education.

¹¹¹ DPMA Certificate Panel (1964). Charles Babbage Institute Archives, CBI 46, Box 1, Fld. 17.

¹¹² Letter to Computerworld from Arthur Kaupe, March 1, 1972. Charles Babbage Institute Archives, CBI 116, Box 1, Fld. 33.

¹¹³ *Computerworld*. Charles Babbage Institute Archives, CBI 116, Box 1, Fld. 30.

V. Professional Associations

In the spring of 1975, on the eve of the annual National Computer Conference, a small group of the elite leaders of the computing community met in a nondescript conference room in a Quality Inn in Anaheim, California to discuss the future of the computing profession. Similar meetings had been convened every year for the previous two decades, always with the intent to address the most pressing issues facing the computing community. Although the specific composition of the group changed from year to year, the attendees always represented the highest levels of leadership in the discipline: award-winning computer scientists, successful business entrepreneurs, association presidents, prolific authors. The cumulative list of participants reads like a Who's Who of the computing industry: Gene Amdahl, Paul Armer, Herbert Bright, Howard Bromberg, Richard Canning, Herb Grosch, Fred Gruenberger, Richard Hamming, J.C.R. Licklider, Daniel D. McCracken, A.G. Oettinger, Seymour Papert, and Joseph Weizenbaum, among many others. This particular meeting included high-ranking representatives from all of the major professional societies: the Association for Computing Machinery (ACM), the Data Processing Management Association (DPMA), the IEEE Computer Society, and the Institute for the Certification of Computer Professionals (ICCP). These societies represented the largest and most influential constituent members of the umbrella

organization the American Federation of Information Processing Societies (AFIPS). On the agenda was a discussion of the role of AFIPS in the professional development of the discipline.

AFIPS had been founded in 1961 as a society of societies. The immediate goal was to provide an American representative to the upcoming International Federation of Information Processing (IFIP) conference. IFIP had been established several years earlier under the aegis of UNESCO (the United Nations Educational, Scientific, and Cultural Organization). Beginning in 1959, IFIP hosted an annual international conference on computing. Each member nation was allowed to send representatives from a *single* organization. Since the United States had no single organization that spoke for its computing community, AFIPS was created to represent three of the largest computer-related societies: the Association for Computing Machinery (ACM), the American Institute of Electrical Engineers (AIEE), and the Institute of Radio Engineers (IRE).¹¹⁴ It was hoped that AFIPS would eventually come to serve as the single national spokesman for computer interests in the United States.¹¹⁵

From the very beginning, AFIPS was a disappointment. AFIPS did represent the United States at the annual IFIP meeting. It was given control over the

¹¹⁴ The AIEE and IRE later merged into the IEEE.

¹¹⁵ Willis Ware, "AFIPS in Retrospect," *Annals of the History of Computing* 8, 3 (1986), 304.

lucrative Joint Computer Conferences, but beyond that, it proved incapable of serving as "the voice of the computing profession in America."¹¹⁶ It was crippled by a weak charter and a lack of tangible support from its founding societies. AFIPS was a society of societies, not a society of members, and was therefore dependent on and subservient to the interests of its constituent societies, rather than to the larger computing community. In addition, several obvious candidates for membership, including the Data Processing Management Association (DPMA) had been conspicuously excluded from participation, and the AFIPS voting structure made it obvious that additional members would be unwelcome.¹¹⁷ Even more limiting was a clause in the constitution, insisted on by the ACM as an essential precondition for its support, prohibiting AFIPS from placing itself "in direct competition with the activities of its member societies."¹¹⁸ Although the constitution was revised in 1969 to provide for stronger leadership and a more inclusive atmosphere, AFIPS continued to struggle for support and recognition. The DPMA did not join until 1974, for example, and even then without much enthusiasm. The 1975 gathering of the computing elite at the

¹¹⁶ ARPA survey, 1968 (reference in AFIPS constitution letter, Communications ACM, March 1969). The East and West Joint Computer Conferences were lucrative annual trade shows.

¹¹⁷ Bernard Galler, "The AFIPS Constitution (President's Letter to ACM Membership)," *Communications of the ACM* 12, 3 (1969), 188.

¹¹⁸ ———, "Reflections on a Quarter-Century: AFIPS Founders," *Annals of the History of Computing* 8, 3 (1986), 225-260.

Quality Inn in Anaheim represented one of the many attempts to reinvigorate interest in this ailing association.¹¹⁹

The transcripts of the 1975 meeting are revealing. The existence of a powerful professional association was obviously considered by the elite leadership of the computing community to be the cornerstone of a strong professional identity. After all, argued one ACM editorial, "Professions are organized, established and directed by professional societies; our Association should represent our profession."¹²⁰ Constant references were made to the role that the American Medical Association (AMA) played in the advancement of the medical profession. The AMA provided tangible benefits to its membership: it maintained standards and controlled access to the profession; it protected them from adverse legislation; it provided for public relations; it allowed for self-regulation. Not incidentally, it also served a valuable social function, providing status and privilege. In any case, strong national associations had served law and medicine quite nicely: who could doubt that they would do the same for the computing professions?¹²¹

¹¹⁹ In 1989, just two years after celebrating its twenty-fifth anniversary, AFIPS voted itself out of existence. The loss of control over the lucrative National Computing Conferences left it financially unstable and without any clear means of support. Few in the community mourned its passing.

¹²⁰ —, "Will you vote for an association name change to ACIS?," *Communications of the ACM* 8, 7 (1965), 424.

¹²¹ "Problems of the AFIPS Societies Revisited," Charles Babbage Institute Archives, CBI 78, Box 3, Fld. 7.

Despite a general agreement on the value of professional associations, at least within the confines of the Quality Inn conference room, the debate about the role and future of AFIPS was surprisingly contentious. Rivalries between the member societies, particularly ACM and the DPMA, were an endemic problem. Participants disagreed over membership qualifications, dues, voting privileges, and certification and licensing proposals. More important, however, was the lack of widespread popular support for these associations. A 1967 *Datamation* article indicated that "Less than 40% [of programmers] belong to any professional association. Probably less than 1% do anything in connection with an association that requires an extra effort on the individual's part."¹²² Even these low figures were probably inflated: a Wall Street Journal report from the next year revealed only that 13% of the data processing personnel surveyed belonged to any professional society.¹²³ These numbers correspond well with the low level of interest in the CDP certification program.¹²⁴ Although it is difficult to compile exact figures on association membership, it is clear that at best only a small percentage of the eligible population chose to participate in any professional society.

¹²² Richard Jones, "A time to assume responsibility," *Datamation* 13, 9 (1967), 160.

¹²³ "Survey on Use of Service Bureaus," *Wall Street Journal* (1969). Charles Babbage Institute Archives, CBI 80, Box 30, Fld. 29.

¹²⁴ See Figure 3.6 above.

If strong professional associations were widely perceived to be an important element of professional identity, why did groups like the ACM, DPMA, and AFIPS have such difficulty attracting and keeping members? AFIPS had some obvious structural problems that almost assured its ineffectiveness. Individuals could not directly join AFIPS; it was merely an umbrella organization for other associations, and possessed little real authority. But what about the ACM and the DPMA, the two largest relevant member societies? Both of these groups were established early, were relatively high-profile, and published their own widely distributed journals. Both were frequently mentioned as candidates for the position of *the* professional computing association. Yet neither was able to consolidate their control over any significant portion of the discipline's practitioners. The reasons behind their failure suggest the limitations of professional associations as an institutional solution to the software crisis.

The Association for Computing Machinery

On January 10, 1947, at the Symposium on Large-Scale Digital Calculating Machinery at the Harvard Computation Laboratory, Professor Samuel Caldwell of MIT proposed to a crowd of more than three hundred the formation of a new association of those interested in computing machinery. His proposal obviously landed on fertile soil: within six months a "Notice on the Organization of an Eastern Association for Computing Machinery" was circulating within the

computing community, and in September the first meeting of the Eastern Association for Computing Machinery was held at Columbia University. Seventy-eight individuals attended. Officers were elected, and an Executive Council appointed. A second meeting, held in December at the Aberdeen Proving Grounds in Maryland attracted three hundred participants. The next year the organization dropped the word "Eastern" from its title, and was thereafter known as the ACM.

During the 1950s the ACM grew steadily but not spectacularly. By 1951 there were 1113 members, including 43 in other countries; in 1956, the total had risen to 2305, and by 1959 had reached 5254. In the 1960s, membership grew somewhat more slowly, and there were a few periods during which the total number of members actually decreased. Overall, however, the ACM continued to expand at a rate of about 16% annually. By the end of 1969 there were 22,761 regular members. Figure 3.7 shows the annual membership statistics for the years 1947-1972.¹²⁵

¹²⁵ ACM 15 years. Charles Babbage Institute Archives, CBI 23, Box 1, Fld. 5.

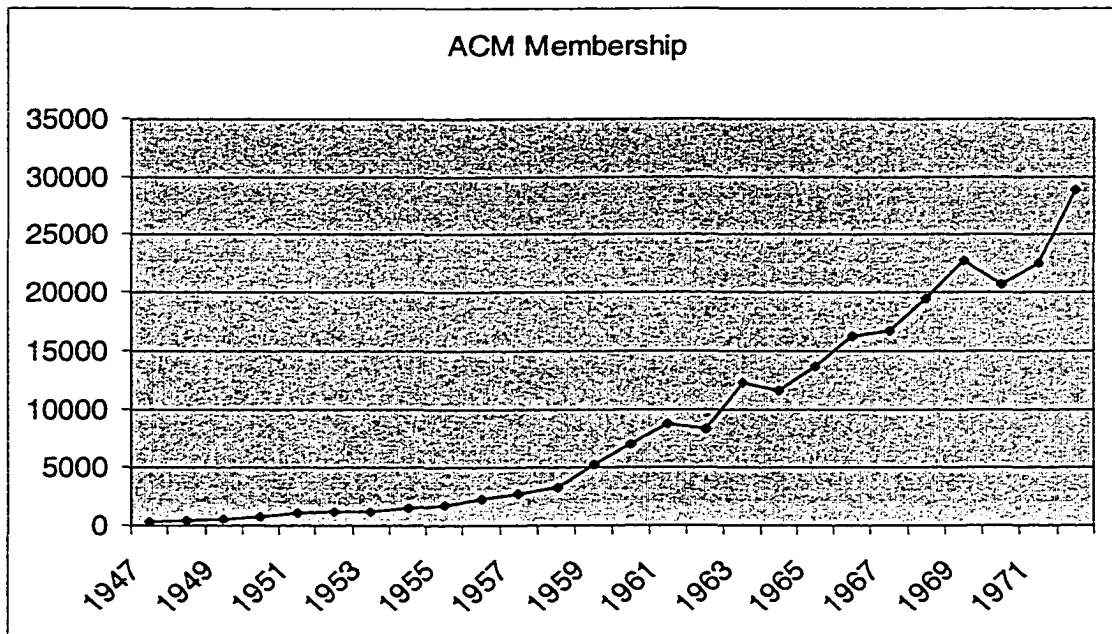


Figure 3.7: Association for Computer Machinery Membership, 1961-1973.

From its inception the ACM styled itself as an academically-oriented organization. Many of the original members either were or had been associated with a major university computation project, and most were university educated, a number at the graduate level. The focus of the organization's early activities were a series of national conferences, the first of which was co-sponsored by the Institute for Numerical Analysis at the University of California - Los Angeles. These meetings represented an outgrowth of an earlier series of university-sponsored conferences, and they retained an academic character. Many were low-budget affairs held at universities or research institutions, and frequently made use of dormitory facilities. The papers presented were usually technical,

and the proceedings were published. The ACM conferences never acquired the trade-show atmosphere that characterized other national meetings. In fact, deliberate efforts were made to distance the ACM from the influence of the commercial vendors, particularly IBM.¹²⁶ For many years the ACM resisted publishing its own journal, possibly because "some early ACM leaders saw the society as a declaration of independence from IBM, and, by extension, from all commercial considerations like the sale of publications and the solicitation of advertising."¹²⁷ Until 1953, when it began publishing the *Journal of the ACM*, the ACM exclusively supported the National Research Council's highly-technical journal *Mathematical Tables and Other Aids to Computation*. Even then, the primary contents of the *Journal* were theoretical papers, and the emphasis was on the dissemination of "information about computing machinery in the best scientific tradition."¹²⁸ Articles were peer-reviewed, and every attempt was made to maintain rigorous academic standards.

Throughout the 1950s and '60s the ACM continued to cultivate its relationship with the academic community. In 1954 it accepted an invitation to apply for membership in the American Association for the Advancement of

¹²⁶ Particularly the National Computer Conference, which became almost entirely commercial, resembling a trade show much more than an academic conference.

¹²⁷ Eric Weiss, "Publications in Computing: An Informal Review," *Communications of the ACM* 15, 7 (1972).

¹²⁸ Saul Gass, "ACM class structure (letter to editor)," *Communications of the ACM* 2, 5 (1959), 4.

Science. Since 1958 the ACM has been represented in the Mathematical Sciences Division of the National Academy of Sciences National Research Council. In 1962 it affiliated with the Conference Board of the Mathematical Sciences, which also consisted of the American Mathematical Society, the Mathematical Association of America, the Society for Industrial and Applied Mathematics, and the Institute of Mathematical Statistics. In 1966, the ACM established the prestigious Turing Award, the highest honor awarded in computer science. Almost half of the institutional members of the ACM were educational organizations, and after 1962 a thriving student membership program was developed.¹²⁹

The close association that the ACM maintained with the academic computer scientist proved a mixed blessing, however. Although the ACM was able to maintain a relatively high-profile within scientific and mathematical circles, it was often castigated by the business community. Many business programmers looked upon the ACM as “a sort of holier than thou academic intellectual sort of enterprise - not inclined to be messing around with the garbage that comptrollers worry about,” and the ACM leadership was characterized as “a bunch of guys with their heads in the clouds worrying about

¹²⁹ Charles Babbage Institute Archives, CBI 88, Box 22, Fld. 1; Charles Babbage Institute Archives, CBI 23, Box 1, Fld. 15.

Tchebysheff polynomials and things like that."¹³⁰ A 1963 *Datamation* article on "The Cost of Professionalism" warned that the members of the ACM had to "decide whether it's worth that much to belong to an organization which many feel has been dominated by - and catered pretty much to- Ph.D. mathematicians ... the Association tends to look down its nose at business data processing types while claiming to represent the whole, wide wonderful world of computing."¹³¹ A 1966 Diebold Group publication characterized the ACM as a group "whose interests are primarily academic and which is helpful to those with scholastic backgrounds, theoreticians of methodology, scientific programmers and software people." Although the ACM president immediately denied this characterization, calling it "too narrow," the popular perception that the ACM catered solely to academics was difficult to counter.¹³²

The ACM leadership was not entirely unaware of or unsympathetic to the needs of the business programmers. In his unsuccessful 1959 bid for the ACM presidency, Paul Armer urged the ACM membership to "THINK BIG," to "visualize ACM as the professional society unifying *all* computer users."¹³³ That same year, Herbert Grosch, an outspoken proponent of a strong, AMA-style

¹³⁰ Rand Symposium, 1969. Charles Babbage Institute Archives, CBI 78, Box 3, Fld. 4.

¹³¹ *Datamation* Editorial, "The Cost of Professionalism," *Datamation* 9, 10 (1963), 23.

¹³² Anthony Oettinger, "On ACM's Responsibility (President's Letter to ACM Membership) (1966)," *Communications of the ACM* 9, 8 (1966), 545-546.

¹³³ Paul Armer, "Thinking Big (letter to editor)," *Communications of the ACM* 2, 1 (1959), 2. Emphasis mine.

professional society, roundly criticized the ACM for its academic parochialism: "Information processing is as broad as our culture and as deep as interplanetary space. To allow narrow interests, pioneering though they might have been, to preempt the name, to relegate ninety percent of the field to 'an exercise left to the reader,' would be disastrous to the underlying unity of the new information sciences."¹³⁴ Several attempts were made during the next decade to make the ACM more relevant to the business community. In response to widespread criticism of the theoretical orientation of the *Journal of the ACM*, a new publication, the *Communications of the ACM*, was introduced in 1958. The main contents of the *Communications* were short articles, mostly unrefereed, on technical subjects such as applications, techniques, and standards.¹³⁵ In 1966 the Executive Committee announced a \$45,000 professional development program aimed at business data processing personnel. The program included short "skill upgrade" seminars offered at the national computer conferences, a traveling course series, and self-study materials.¹³⁶ There was even talk, in the mid-1960s,

¹³⁴ Herb Grosch, "Plus and Minus," *Datamation* 5, 6 (1959), 51.

¹³⁵ Robert Payne, "Reaction to Publication Proposal (letter to editor)," *Communications of the ACM* 8, 1 (1965), 71.

¹³⁶ Anthony Oettinger, "ACM sponsors professional development program (President's Letter to ACM Membership)," *Communications of the ACM* 9, 10 (1966), 712-713.

of a potential merger with the DPMA. In 1969, ACM president Bernard Galler announced a move towards "less formality, less science, and less academia."¹³⁷

Despite these short-lived efforts to reconcile with the business community, however, the conservative ACM leadership continued to pursue a largely academic agenda. As early as 1959 it was suggested that the ACM should impose stringent academic standards on its members, and in 1965 a four-year degree became a prerequisite for receiving full membership. Frequent battles arose over repeated attempts to change the name of the association to something more broadly relevant. In 1965 a proposal to change it to the Association for Computing and Information Science was rejected; a decade later the same issue was still being debated.¹³⁸ When Louis Fein suggested in 1967 that the ACM faced a "crisis of identity," ACM President Oettinger insisted vehemently that the "ACM has no crisis of identity." In doing so, he reaffirmed the association's commitment to a theoretical approach to computing: "Our science must, indeed, 'maintain as its sole abstract purpose of advancing truth and knowledge.'"¹³⁹

This commitment to abstract science was further reinforced the following year when the ACM Committee on Curriculum for Computer Science (C³S)

¹³⁷ Bernard Galler, "The Journal (President's Letter to ACM Membership)," *Communications of the ACM* 12, 2 (1969), 65-66.

¹³⁸ —, "Will you vote for an association name change to ACIS?," *Communications of the ACM* 8, 7 (1965); Vote on ACM name change (1978) Charles Babbage Institute Archives, CBI 43, Box 3, Fld. 10.

¹³⁹ Anthony Oettinger, "President's reply to Louis Fein," *Communications of the ACM* 10, 1 (1967), 1.

announced their Curriculum '68 guidelines for university computer science programs. Curriculum '68 advocated a rigorously theoretically approach to computer science that included little of interest to business practitioners.¹⁴⁰ Even when the ACM did recognize the growing importance of business data processing to the future of their discipline, the emphasis was always placed on research and education:

All of us, I am sure, have read non-ACM articles on business data processing and found them lacking. They suffer, I believe, from one basic fault: They fail to report fundamental research in the data processing field. The question of 'fundamentalness' is all-important ... In summary, this letter is intended to urge new emphasis on FUNDAMENTALISM in business data processing. This objective seems not only feasible but essential to me. It provides not only a technique for getting ACM into the business data processing business, but a technique (the same one) for getting the field of business data processing on a firm theoretical footing.¹⁴¹

There is little question that throughout the 1960s the ACM pursued a professionalization strategy that was heavily dependent on the authority and legitimacy of its academic accomplishments.

It was not until the 1970s that the ACM began to seriously reconsider its policy towards business-oriented practitioners. In 1974 the ACM Executive Council commissioned a series of studies on business programming as part of its

¹⁴⁰ Raymond Wishner, "Comment on Curriculum 68," *Communications of the ACM* 11, 10 (1968); Datamation Report, "Curriculum 68," *Datamation* 14, 5 (1968); Hamming (1968).

¹⁴¹ John Postley, "Letter to Editor," *Communications of the ACM* 3, 1 (1960), A6.

long-range planning report. In doing so the ACM was responding both to long-standing criticism and to a recent spate of anti-ACM editorials that had appeared in the industry newsletter *Computerworld*. "ACM had become not so much an industry professional group," declared one of these editorials, "as it was a home for members of educational institutions around the country to overwhelm us with their erudition on topics of vaguely moderate interest."¹⁴² The author noted that while most business data processing installations had standardized on the COBOL and FORTRAN programming languages, the ACM still supported ALGOL. He quoted ACM president Anthony Ralston to the effect that although only 25% of the ACM membership were academics, 10 out of 25 council members were.¹⁴³

The 1974 long-range report noted that of the 320 thousand software personnel then working in the United States, 85% dealt with business data processing (BDP). It admitted that while the ACM had a reputation for professionalism, "BDP people tend to be turned off by ACM's academically oriented leadership ... BDP professionals feel that academics don't understand what BDP needs, and they're right."¹⁴⁴ It concluded that any new ACM members were likely to come from BDP, and recommended the development of a new

¹⁴² "Why are business users turned off by ACM?" (1974). Charles Babbage Institute Archives, CBI 23, Box 1, Fld. 3.

¹⁴³ Ibid.

¹⁴⁴ Ibid.

publication aimed at a BDP audience. The report signaled to many in the ACM that the organization needed to broaden its membership and become more accommodating. The next few years witnessed a bitterly contested presidential election (the cornerstone of which was a debate over business data processing); yet another attempt to change the name of the ACM to something more broadly relevant; and efforts to reconcile with its business-oriented competitor, the Data Processing Management Association.

The Data Processing Management Association

The Data Processing Management Organization originated in 1949 as the National Machine Accountants Association (NMAA). The NMAA was founded as an association of accountants and tabulating machine managers. In 1952 it held its first convention in Minneapolis, MN. Ten years later, it represented almost 16,000 data processing workers in the United States and Canada. In 1962 the NMAA changed its name to the DPMA in hopes of expanding its membership beyond finance and accounting professionals, and to call attention to its new CDP certification program. The CDP program was only one of the DPMA's ambitious "Six Measures of Professionalism Program":

- 1) Professionals devote a portion of their efforts to helping newcomers acquire the knowledge required of practitioners in the field; therefore, a program of education for the beginner is one mark of professionalism.

- 2) Professionals devote a portion of their efforts to updating their own knowledge; therefore, a program for self-education is one mark of professionalism.
- 3) Professionals possess a minimum level of knowledge of the field; therefore, a standard way of measuring that knowledge is one mark of professionalism.
- 4) Professionals devote a portion of their efforts to contribute to the knowledge of their field; therefore, a program of continuing research is one mark of professionalism.
- 5) Professionals conduct themselves in a way that reflects credit on their profession or always act in the best interests of the general public and their profession; therefore, a code of ethics, accepted and practiced, is one mark of professionalism.
- 6) Professionals police their own ranks and invoke discipline among themselves to those who violate the established rules of ethical conduct; therefore, an effective means of policing and disciplining practitioners is one mark of professionalism.¹⁴⁵

In 1967 the DPMA released a report detailing its efforts to fulfill their professionalism agenda. *The Future of Data Processors* program worked with colleges and universities in curriculum development. National conferences, local chapter programs and seminars, and DPMA publications and home-study courses were all directed toward the self-education of individual members. The CDP program was obviously intended to establish a means of "measuring a minimum level of knowledge in the field." DPMA *Graduate Research Grants* encouraged contributions to the "knowledge of the field." The DPMA *Code of*

¹⁴⁵ DPMA report on "Six measures of professionalism." Charles Babbage Institute Archives, CBI 88, Box 21, Fld. 40.

Ethics dated back to the origins of the association, and was the first of such codes to be established for the computer-related professions. Finally, although the DPMA acknowledged that it had no existing mechanisms for determining and punishing misconduct, the report promised that the association would take a leading role in the development of an industry policing program. Of the DPMA's attempts to address these "six measures of professionalism," only the CDP program achieved even moderate industry acceptance; nevertheless, simply by articulating a clear professional agenda the DPMA claimed for itself a leadership role in the computing community.

From the very beginning the DPMA made efforts to reach a broad spectrum of data processing personnel. In 1964 the national leadership made specific efforts to include programmers within its membership.¹⁴⁶ The structure of the organization, which included strong regional chapters, allowed for diversity and local control. Each region had a representative on the Executive Council who served with several executive officers and implemented policy decisions from the International Board of Directors. In addition, the DPMA's official publication, the *Data Management Journal*, encouraged submissions on a much wider range of subjects than did the ACM's *Journal or Communications*. The DPMA also maintained a close association with the editors of *Datamation*,

¹⁴⁶ Local Chapter CDP publicity (1964). Charles Babbage Institute Archives, CBI 46, Box 1, Fld. 8.

another widely-read industry journal that focused on issues of timely concern and practical relevance.

The DPMA's inclusive approach to professional development brought it into conflict with competing societies, particularly the ACM. The differences between two organizations mirrored the larger tensions that existed within the computing community: academic computer scientists versus the business data processors; theory versus practice. I have already shown how this tension affected the adoption of the DPMA's CDP program: the ACM's obvious lack of support helped to undermine the program's legitimacy and prevented its widespread adoption. This opposition was based on grounds both philosophical – many in the ACM believed that the CDP examinations were superficial and irrelevant – and institutional, since control over an industry-wide certification program would have granted the DPMA considerable political authority.¹⁴⁷ Despite several half-hearted attempts to explore an ACM-DPMA merger, or at least to establish an inter-association liaison, the two groups rarely communicated.¹⁴⁸ When AFIPS was established in the early 1960s, the NMAA and other industry-

¹⁴⁷ Letter re: four year degree requirement (1970) CBI 116.1.27; Misc. Correspondence re: CDP exams (1973) CBI 46.2.14

¹⁴⁸ Notes on ACM/DPMA merger (1964) CBI 88.22.2; Correspondence re: ACM/DPMA liaison (1966) CBI 88.22.1; Discussion of DPMA/ACM merger (1970) CBI 88.22.3

oriented groups were treated with dismissive contempt, and the DPMA resisted AFIPS affiliation until the mid-1970s.¹⁴⁹

Professional societies...or technician associations?

The persistent conflict between the ACM and DPMA reflected a much larger tension that existed within the computing community. As early as 1959 the outlines of a battle between academically-oriented computer scientists and business programmers had taken shape around the issue of professionalism.¹⁵⁰ Although both groups agreed on the desirability of establishing institutional and occupational boundaries around the nascent computer-related professions, they disagreed sharply about what form these professional structures should take. Observers noted a deepening "programming schism" developing within the industry, a "growing breach between the scientific and engineering computation boys who talk ALGOL and FORTRAN...and the business data processing boys who talk English and write programs in COBOL."¹⁵¹ Individuals who believed that the key to professional status was the development of formal theories of computer science resisted "sub-professional" certification programs and tended to join the ACM; business data processors who were skeptical of "cute

¹⁴⁹ RAND Symposium, "Problems of the AFIPS Societies Revisited," 1975. Charles Babbage Institute Archives, CBI 78, Box 3, Fld. 7. At a meeting arranged by AFIPS officials, DPMA representatives were kept waiting, without explanation or apology, for over an hour.

¹⁵⁰ RAND Symposium, "Is It Overhaul or Trade-in Time? Part II," *Datamation* 5, 5 (1959).

¹⁵¹ Christopher Shaw, "Programming Schisms," *Datamation* 8, 9 (1962), 32.

mathematical tricks" either supported the DPMA or ignored the professional societies altogether.

It is clear that the turf battles that raged between the ACM and the DPMA during the 1950s and '60s helped undermine popular support for both organizations. In response to extensive *Datamation* coverage of a 1959 RAND symposium on "the perennial professional society question," one reader commented that he "hadn't laughed so hard in a decade. Are these guys kidding? You won't solve this problem by self-interested conversation about it, nor is it solved by founding another organization."¹⁵² In a 1985 retrospective on the troubled history of AFIPS, Harry Tropp suggested that "the question of turf seems to have been there from the beginning. It shows up in the [1950s] Rand Symposium ... There were the hardware and software types and then there were the users. We had the east coast/west coast turf problems. What I am hearing today is a whole new evolution of different turfs as this information processing society explodes."¹⁵³ The fact that the DPMA refused affiliation with AFIPS until the mid-1970s – largely because of the perception that the latter organization was dominated by the ACM – was a major factor in its perpetual ineffectiveness and

¹⁵² Wolf Flywheel, "Letter to the editor (on professionalism)," *Datamation* 5, 5 (1959), 2.

¹⁵³ AFIPS Presidents discussion (1985). Charles Babbage Institute Archives, CBI 114, Box 1, Fld. 4.

eventually dissolution.¹⁵⁴ Many observers were dismayed by the pettiness of the ACM-DPMA debates, which they believed detracted from the overall goal of establishing a legitimate professional identity:

I couldn't care less who publishes some abstract scientific paper! What I want to know is how do we pull together a hundred thousand warm bodies that are working on the outskirts of the computer business, give them a high priced executive director, lots of advertising, a whole series of technical journals; in other words, organize a real rip-snorting profession? Whenever somebody starts worrying about which journal what paper should be published in, we get bogged down in an academic cross-fire we've been in for ten years.¹⁵⁵

As damaging as these inter-associational rivalries were to the influence and reputation of the ACM and DPMA, what really hurt them was the lack of support that they received from industry practitioners. Neither organization was able to clearly establish its relevance to the needs of either workers or their managers. "Neither organization ... has done much for the industry or for society as a whole," argued one 1965 *Datamation* editorial. "We think the time is ripe to more clearly define larger, more important long-range goals which distinguish a professional society from a technician's association."¹⁵⁶ Employers looked to the professional associations to provide a supply of reliable, capable programmers. As was clear from the impassioned debates about structure and

¹⁵⁴ AFIPS was dissolved in 1987, just two years after celebrating its 25th anniversary.

¹⁵⁵ RAND Symposium, "Is It Overhaul or Trade-in Time? Part II."

¹⁵⁶ *Datamation* Editorial, "Professional Societies ... or Technician Associations?," *Datamation* 11, 8 (1965), 23.

relevance of computer science curricula, however, it was far from obvious to many managers that formal educational programs contributed much to the production of "professional" programmers. The ACM's continued devotion to theoretical computer science made it seem out-of-touch with the practical demands of business. The DPMA's CDP program, although it was much more oriented to business data processing, failed to achieve widespread industry acceptance. As a result, it also was not able to guarantee the kind of standardized labor force in which corporations were interested. Employers saw little value in either organization.

VI. The Limits of Professionalism

In his 1968 monograph on *Office Automation in Social Perspective*, the Oxford sociologist H.A. Rhee noted that "The computer elite are beginning to erect collective defenses against the lay world. They are beginning to develop a sense of professional identity and values." But the process of establishing professional attitudes and controls and a professional conscience and solidarity, Rhee suggested, had "not yet advanced very far."¹⁵⁷ He could just have easily been describing the computing professions as they existed a decade earlier or a decade afterward. By 1968 computing had acquired many of the trappings of professionalism: academic computer science departments, certification programs,

¹⁵⁷ Rhee, *Office Automation in Social Perspective*, 118.

professional associations. And yet most computing practitioners were not widely regarded as “professionals,” at least not in the eyes of the general public. In 1967, for example, the U.S. Civil Service Commission declared data processing personnel to be “non-exempt” employees, officially categorizing programmers as technicians rather than professionals. Although this decision did not affect the lives or practices of programmers, it represented a symbolic defeat for professional associations such as the ACM, who lobbied hard to have it overturned.¹⁵⁸

The inability of programmers and other data processing personnel to successfully professionalize raises some perplexing questions for the historian: given the apparent interest in professionalization on the part of both employers and practitioners, why were these efforts so ineffective? As was described earlier, industrial employers in the 1960s complained not as much about technical incompetence as a general lack of professionalism among programmers. “It was his distressing lack of professional attributes that most often undermines his work and destroys his management's confidence,” declared Malcolm Gotterer. “Too frequently these people, while exhibiting excellent technical skills, are non-professional in every other aspect of their work.”¹⁵⁹

¹⁵⁸ Minutes of the Annual Meeting of the Certification Advisory Council (1967). Charles Babbage Institute Archives, CBI 116, Box 1, Fld. 13.

¹⁵⁹ Gotterer, “The Impact of Professionalization Efforts on the Computer Manager,” 368.

Increased professionalism would presumably address the most frequent complaints leveled against data processing personnel: an over-reliance on idiosyncratic craft techniques; an arrogant disregard for proper lines of authority; shoddy workmanship; a lack of commitment to the best interests of the organization. On the surface, the professionalization of programming appeared to be an ideal solution to many of the most deleterious symptoms of the burgeoning software crisis.

There are a number of explanations for the failure of most professionalization programs. Internal rivalries within the computing community undermined the effectiveness of groups such as the ACM and DPMA. No single organization could meet the needs of a diverse community of "computer people" that included everyone from Ph.D. mathematicians to high-school dropout keypunch operators. As Louis Fein suggested in his discussion of the ACM's "crisis of identity," "It is not clear ... that an organization can play simultaneously the role of a profession, of an industry, and of a science ... I cannot see that ACM members, or IEEE Computer Group members, or DPMA members, or Simulations Councils, Inc. members, are members of a profession. They are practitioners or scientists or engineers or programmers-members of a

technical society."¹⁶⁰ As the programming community broke down into competing factions – theoretical vs. practical; certified vs. un-certified; ACM vs. DPMA – its members lost the leverage necessary to push through any particular professionalization agenda.

In addition to internal rivalries, the aspiring computing professions also faced external opposition. For many corporate managers, professionalism was a potentially dangerous doubled-edged weapon: on the one hand, "Professionalism might motivate staff members to improve their capabilities, it could bring about more commonality of approaches, it could be used for hiring, promotions and raises, and it could help determine 'who is qualified.'" On the other hand, "professionalism might well increase staff mobility and hence turnover, and it probably would lead to higher salaries for the 'professionals.'"¹⁶¹ Computer personnel were often seen as dangerously disruptive to the traditional corporate establishment. The last thing traditional managers wanted was to provide data processing personnel with additional occupational authority. Professionalism was therefore encouraged only to the extent that it provided a standardized, tractable workforce; professionalization efforts that encouraged

¹⁶⁰ Louis Fein, "ACM Has a Crisis of Identity?," *Communications of the ACM* 10, 1 (1967).

¹⁶¹ Richard Canning, "Professionalism: Coming or Not?," *EDP Analyzer* 14, 3 (1976), 2.

elitism, protectionism, or anything that smacked of unionism were seen as counter-productive.

Perhaps the most important reason that programmers and other data processing personnel failed to professionalize, however, was that the professional institutions that were set up in the 1950s and 1960s failed to convince employers of their relevance to the needs of business. A 1974 *Computerworld* survey indicated that "no technical society has ever captured and held the attention of professionals in BDP [business data processing]."¹⁶² Employers looked to professional institutions as a means of supplying their demand for competent, trustworthy employees. As we have seen, although computer science programs in the 1960s thrived in the universities, in the business world they were often seen as overly theoretical and irrelevant. Likewise, the DPMA's CDP program failed to establish itself as a reliable mechanism for predicting programmer performance or ability. Neither the ACM nor the DPMA offered much to employers in terms of improving the supply or quality of the programming workforce.

Given this lack of active support from employers, the professional associations had little to offer most data processing practitioners. Neither a computer science education nor professional certification could ensure

¹⁶² "Just So Programs," *Computerworld* (May 15, 1974). Charles Babbage Institute Archives, CBI 23, Box 1, Fld. 3.

employment or advancement. In response to a 1974 *Computerworld* article on "Why Business Users Are Turned Off by ACM," AFIPS president George Glaser suggested that "The general lack of success of ACM in attracting business data processing professionals to its membership has relatively little to do with the nature and extent of the services it offers them. It is, rather, more attributable to a lack of interest on the part of these 'professionals' in any professional society."¹⁶³ Glaser's comment can be read either as an indictment of the apathy of the average computing practitioner or of the policies of the ACM; either way, it suggests the strained relationship that existed between the two communities. Many working programmers saw little value in belonging to either the ACM or the DPMA, and support for both organizations, as well as for professional institutions in general, languished during the late 1960s and early 1970s.

¹⁶³ George Glaser, "Letter to W. Carlson." Charles Babbage Institute Archives, CBI 23, Box 1, Fld. 3.

Epilogue: No Silver Bullet

A quarter of a century later software engineering remains a term of aspiration. The vast majority of computer code is still handcrafted from raw programming languages by artisans using techniques they neither measure nor are able to repeat consistently...¹

“Software’s Chronic Crisis,” *Scientific American* (1994)

I. *From Exhilaration to Disillusionment*

The 1968 NATO Conference on Software Engineering was, according to contemporary accounts, an exhilarating experience for many participants. The public acknowledgement of a perceived software crisis was a cathartic moment for the industry. As one prominent computer scientist described it, “The general admission of the software failure in this group of responsible people is the most refreshing experience that I have had in a number of years, because the admission of shortcomings is the primary condition for improvement.”² Despite the general recognition of impending crisis, the spirit of the conference was “positive, even liberatory.”³ Attendees rallied behind the organizers’ call for “a switch from home-made software to manufactured software, from tinkering to

¹ W. Gibbs, “Software’s Chronic Crisis,” *Scientific American*, September 1994.

² Dijkstra interview, cited in Eloina Palaez, “A Gift From Pandora’s Box: The Software Crisis,” (Ph.D. dissertation, University of Edinburgh, 1988).

³ Donald MacKenzie, “A View from the Sonnenbichl: On the Historical Sociology of Software and System Dependability,” in *Mapping the History of Computing: Software Issues*, U. Hashagen, R. Keil-Slawik, A. Norberg, eds. (New York: Springer-Verlag, forthcoming), 79.

engineering.”⁴ Software engineering emerged as the dominant rhetorical paradigm for discussing the future of software development. By adopting the “types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering,” computer programming could be successfully transformed from a black art into an industrial discipline. Software workers from a wide variety of disciplines and backgrounds adopted the rhetoric of software engineering as a shared discourse within which to discuss their mutual professional aspirations.

In order to capitalize on the enthusiasm generated in the wake of the Garmisch meeting, the NATO Science Committee quickly organized a second conference to be held the following year in Rome, Italy. The 1969 Rome Conference was intended to have an explicitly practical focus: the goal was to develop specific techniques of software engineering. As with the Garmisch meeting, a deliberate and successful attempt was made to attract a wide range of participants. The resulting conference, however, bore little resemblance to its predecessor. Whereas the Garmisch participants had coalesced around a shared sense of urgency, the Rome conference was characterized by conflict. According to the same observer who had referred glowingly to the Garmisch conference as a “most refreshing experience,” the discussions at the Rome meeting were

⁴ F.L. Bauer, “Software Engineering: A Conference Report,” *Datamation* 15, 10 (1969).

"sterile," the various groups of attendees "never clicked," and "most participants" left feeling "an enormous sense of disillusionment."⁵ A prolonged debate about the establishment of an international software engineering institute proved so acrimonious and divisive that it was omitted from the conference proceedings: "All I remember is that it ended up being a lot of time wasted, and no argument ever turned up to make something happen – which is probably just as well..."⁶

Why was the Rome conference considered such a disappointment relative to Garmisch? Many of the same participants had attended both meetings: there had been no significant changes in terms of demographic makeup or organizational structure. Neither were there any major new issues or technologies introduced or discussed. Many of the Rome presentations covered material that had previously been addressed, albeit at a less detailed and technical level, at Garmisch. And yet while the Garmisch conference is widely considered to have marked a pivotal moment in the history of software development - "a major cultural shift in the perception of programming" - the Rome conference seems to have been deliberately forgotten.⁷

⁵ J. Buxton, quoted in Paleaz.

⁶ D. Ross, quoted in Paleaz.

⁷ Martin Campbell-Kelly and William Aspray, *Computer: A History of the Information Machine* (New York: Basic Books, 1996), 201.

One obvious difference between the two events is that the earlier conference had encouraged participants to focus their attention on a commonly perceived but vaguely defined emergency, while the latter forced them to deal with specific controversial issues. Software engineering had emerged as a compelling solution to the software crisis in part because it was flexible enough to appeal to a wide variety of computing practitioners. The ambiguity of concepts such as “professionalism,” “engineering discipline,” and “efficiency” allowed competing interests to participate in a shared discourse that nevertheless enabled them to pursue vastly different personal and professional agendas. Industry managers adopted a definition of “professionalism” that provided for educational and certification standards, a tightly disciplined workforce, and increased corporate loyalty. Computer manufacturers looked to “engineering discipline” as means of countering charges of incompetence and cost-inefficiency. Academic computer scientists preferred a highly formalized approach to software engineering that was both intellectually respectable and theoretically rigorous. Working programmers tended to focus on the more personal aspects of professional accomplishment, including autonomy, status, and career longevity. The software engineering model seemed to offer something to everyone: standards, quality, academic respectability, status and autonomy.

The rhetorical flexibility that had served the consensus-seeking Garmisch participants proved unwieldy when it came to establishing specific standards and practices, however. The Rome conference illuminated in sharp relief the vast differences that existed between competing visions for the software engineering discipline. Unlike the conflict between workers and managers described in the previous chapter, these divisions were largely internal to the programming community. The primary split was between academic computer scientists and commercial software developers. The industry programmers resented being invited to Rome "like a lot of monkeys to be looked at by theoreticians;" the theoreticians complained of feeling isolated, of "not being allowed to say anything."⁸ As the editors of the conference proceedings have suggested, the "lack of communication between different sections of the participants" became the "dominant feature" of the meeting.⁹ "The seriousness of this communications gap," and the realization that it "was but a reflection of the situation in the real world," caused the gap itself to become a major topic of discussion.¹⁰ It was to remain an issue of central concern to the programming community for the next several decades.

⁸ Christopher Strachey, quoted in Peter Naur, Brian Randall, and J.N. Buxton, ed., *Software engineering Proceedings of the NATO conferences* (New York: Petrocelli/Carter, 1976), 147.

⁹ Naur, et al., 145.

¹⁰ *Ibid.*

II. *Software's Chronic Crisis*

In an 1996 article entitled "Software's Chronic Crisis," William Gibbs noted in a *Scientific American* article that a quarter of a century after the Garmisch conference, "software engineering remains a term of aspiration," rather an fully realized discipline.¹¹ Indeed, in the years after 1968 the rhetoric of the software crisis became even more heated. In 1987 the editors of *Computerworld* complained that "the average software project is often one year behind plan and 100% over budget."¹² In 1989 the House Committee on Science, Space and Technology released a report highly critical of the "shoot-from-the-hip" practices of the software industry. Among other things, the report called for a professional certification program for programmers.¹³ Later that same year the Pentagon launched a broad campaign to "lick its software problems" that included funds for a Software Engineering Institute and the widespread adoption of the ADA programming language.¹⁴ ADA was touted by Department of Defense officials "a means of replacing the idiosyncratic 'artistic' ethos that has long governed software writing with a more efficient, cost-effective

¹¹ W. Gibbs, "Software's Chronic Crisis," *Scientific American*, September 1994.

¹² Ann Dooley, "100% Over Budget," *Computerworld*, July 8, 1987

¹³ The 33-page report, entitled "Bugs in the Program: Problems in Federal Government Computer Software Development and Regulation," was written by two staff members, James H. Paul and Gregory C. Simon, of the Subcommittee on Investigations and Oversight of the House Committee on Science, Space, and Technology.

¹⁴ David Morrison, "Software Crisis," *Defense* 21, 2 (1989)

engineering mind-set."¹⁵ The list of critical reports, denunciations of current practices, and proposed "silver-bullet" solutions goes on and on. And yet, in the words of one industry observer, by the mid-1980s "the software crisis has become less a turning point than a way of life."¹⁶

The continued existence of a four decade long crisis in one of the largest and fastest growing sectors of the American economy reveals the highly contested nature of computer technology. Historians of technology have long argued that all technologies are, at least to a certain degree, socially constructed. That is simply to say that the physical design of an artifact is inextricably influenced by its larger environment. In the 1950s and 1960s the electronic digital computer was introduced into the well-established technical and social systems of the modern business organization. Like all new technologies, the computer both took its shape from - and helped to shape - its social, cultural, and technological context. As the computer became an increasingly important part of the modern corporate organization, control over its use and identity became increasingly contested. The conflicting needs and agendas of users, manufacturers, managers, and programmers all became wrapped up in highly public struggle for control over the professional territory opened up by the technology of computing.

¹⁵ Morrison, "Software Crisis,"

¹⁶ John Shore, "Why I Never Met a Programmer I Could Trust," *Communications of the ACM* 31, 4 (1988).

Thinking about the software crisis – and invention of the discipline of software engineering - as a series of interconnected social and political negotiations, rather than an isolated technical decision about the “one best way” to develop software components, provides an essential link between internal developments in information technology and their larger social and historical context. It can help explain why, in an industry characterized by rapid change and innovation, the rhetoric of crisis has proven so remarkably persistent.

Bibliography

- . "Editor's Readout: A Long View of a Myopic Problem." *Datamation* 8, 5 (1962): 21-22.
- . "What's Happening with COBOL?." *Business Automation* (April, 1968).
- . "DPMA Revises CDP Test Requirements." *Data Management* (August 1967): 34-35.
- . "Reflections on a Quarter-Century: AFIPS Founders." *Annals of the History of Computing* 8, 3 (1986): 225-260.
- . "Will you vote for an association name change to ACIS?." *Communications of the ACM* 8, 7 (1965): 424-426.
- ACM Curriculum Committee. "An Undergraduate Program in Computer Science - Preliminary Recommendations." *Communications of the ACM* 8, 9 (1965): 543-552.
- ACM Curriculum Committee. "Curriculum 68: Recommendations for Academic Programs in Computer Science." *Communications of the ACM* 11, 3 (1968): 151-197.
- Abbott, Andrew. *The Systems of Professions: An Essay on the Division of Expert Labor*. Chicago: University of Chicago Press, 1988.
- Akera, Atsushi. "Calculating a Natural World: Scientists, Engineers and Computers in the United States, 1937-1968." Ph.D. diss., University of Pennsylvania. 1998.
- Alexander, T. "Computers Can't Solve Everything." *Fortune* (October, 1969).
- Armer, Paul. "Editor's Readout: Suspense Won't Kill Us." *Datamation* 19, 6 (1973): 53.
- Armer, Paul. "Thinking Big (letter to editor)." *Communications of the ACM* 2, 1 (1959): 2-4.

- Aron, Joel. Part I: *The Individual Programmer. The Systems Programming Series*. Reading, MA: Addison-Wesley, 1974.
- Aron, Joel. *Part II: The Programming Team. The Program Development Process*. Reading, MA: Addison-Wesley, 1983.
- Aspray, William. "The Supply of Information Technology Workers, Higher Education, and Computing Research: A History of Policy and Practice in the United States." In *The International History of Information Technology Policy*, Richard Coopey. Oxford: Oxford University Press.
- Aspray, William. "Was Early Entry a Competitive Advantage?." *Annals of the History of Computing* (2000): 42-87.
- Aspray, William. "The History of Computer Professionalism in America." (Unpublished Manuscript) (2000).
- Aspray, William, and Arthur Burks, eds. *Papers of John Von Neumann on Computing and Computer Research*. Cambridge, MA: MIT Press, 1987.
- Backus, John. "Programming in America in the 1950s - Some Personal Impressions." In *A history of computing in the twentieth century: a collection of essays*, Metropolis, Nick; Howlett, J.; Rota, Gian-Carlo. 125-135. New York: Academic Press, 1980.
- Backus, John. "Automatic Programming: Properties and Performance of FORTRAN Systems I and II." *Proceedings of Symposium on the Mechanization of the Thought Processes*, Middlesex, England, National Physical Laboratory Press. 1958.
- Backus, John. "The FORTRAN automatic coding language." *Proceedings of the West Joint Computer Conferences*, 1957.
- Baker, F. Terry, and Harlan Mills. "Chief Programmer Teams." *Datamation* 19, 12 (1973): 58.
- Bardini, Thierry. *Bootstrapping: Douglas Englebart, Coevolution, and the Origins of Personal Computing*. Stanford, CA: Stanford University Press, 2000.

Barnett, Michael. *Computer Programming in English*. New York: Harcourt, Brace & World, 1969.

Barry, B.S., and Naughton, J.J., "Chief Programmer Team Operations Description." *U.S. Air Force, Report No. RADC-TR-74-300*.

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

Bemer, Robert. "A view on the history of COBOL." *Honeywell Computer Journal* 5, 3 (1971): 130-135.

Bendix Computer Division. "Is Your Programming Career in a Closed Loop?." *Datamation* 8, 9 (1962): 86.

Beniger, James R. *The Control Revolution: Technological and Economic Origins of the Information Society*. Cambridge: Harvard University Press, 1986.

Block, I.E. "Report on Meeting Held at University of Pennsylvania Computing Center.," April 9, 1959.

Boehm, Barry. "Software Engineering." *IEEE Transactions on Computers* C-25, 12 (1976): 1226-41.

Boehm, Barry. "Software and Its Impact: A Quantitative Assessment." *Datamation* 19, 5 (1973): 48-59.

Boguslaw, Robert, and Warren Pelton. "Steps: A Management Game for Programming Supervisors." *Datamation* 5, 6 (1959): 13-16.

Bowden, B.V. *Faster than Thought: A Symposium on Digital Computing Machines*. London: Sir Isacc Pitman & Sons, 1953.

Brooks, Frederick P. "No Silver Bullet: Essence and Accidents of Software Engineering." *IEEE Computer*, April, 1987.

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

- Bylinsky, Gene. "Help Wanted: 50,000 Programmers." *Fortune* 75, 3 (March, 1967).
- Callahan, John. "Letter to the editor." *Datamation* 7, 3 (1961): 7.
- Campbell-Kelly, Martin, and William Aspray. *Computer: A History of the Information Machine*. New York: Basic Books, 1996.
- Campbell-Kelly, Martin, and Michael Williams, eds. *The Moore School Lectures: Theory and Techniques for the Design of Electronic Digital Computers*. Cambridge, MA: MIT Press: Charles Babbage Reprint Series, 1985.
- Canning, Richard. "Managing the Programming Effort." *EDP Analyzer* 6, 6 (1968): 1-15.
- Canning, Richard. "Issues in Programming Management." *EDP Analyzer* 12, 4 (1974): 1-14.
- Canning, Richard. "Professionalism: Coming or Not?." *EDP Analyzer* 14, 3 (1976): 1-12.
- Canning, Richard. "Managing Staff Retention and Turnover." *EDP Analyzer* 15, 8 (1977): 1-13.
- Canning, Richard. "The DPMA Certificate in Data Processing." *EDP Analyzer* 3, 7 (1965): 1-12.
- Canning, Richard. "The Question of Professionalism." *EDP Analyzer* 6, 12 (1968): 1-13.
- Carlson, Jack. "On determining CS education programs (letter to editor)." *Communications of the ACM* 9, 3 (1966): 135.
- Ceruzzi, Paul. "Electronics Technology and Computer Science, 1940-1975: A Coevolution." *Annals of the History of Computing* 10, 4 (1989): 257-275.
- Chandler, Alfred P. *The Visible Hand: The Managerial Revolution in American Business*. Cambridge, MA: Belknap Press, 1977.

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

Conway, B., 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.

Correll, Quentin. "Letters to the Editor." *Communications of the ACM* 1, 7 (1958): 2.

Couger, Daniel, and R Zawacki. "What Motivates DP Professionals?." *Datamation* 24, 9 (1978): 116-123.

Cox, Brad. "There is a Silver Bullet." *Byte* 15, 10 (1990): 209.

Datamation Editorial. "Editor's Readout: The Certified Public Programmer." *Datamation* 8, 3 (1962): 23-24.

Datamation Editorial. "EDP's Wailing Wall." *Datamation* 13, 7 (1967): 21.

Datamation Editorial. "Learning a Trade." *Datamation* 12, 10 (1966): 21.

Datamation Editorial. "The Thoughtless Information Technologist." *Datamation* 12, 8 (1966): 21-22.

Datamation Editorial. "Checklist for Oblivion." *Datamation* 10, 9 (1964): 23.

Datamation Editorial. "The Facts of Life." *Datamation* 14, 3 (1968): 21.

Datamation Editorial. "The Cost of Professionalism." *Datamation* 9, 10 (1963): 23.

Datamation Editorial. "Professional Societies.or Technician Associations?." *Datamation* 11, 8 (1965): 23.

Datamation News Brief. "First Programmer Class at Sing Sing Graduates." *Datamation* 14, 6 (1968): 97-98.

Datamation Report. "DP Certification Program Announced by NMAA." *Datamation* 8, 3 (1962): 25.

- Datamation Report. "Certificate in Data Processing." *Datamation* 9, 8 (1963): 59.
- DiNardo, George. "Software Management and the Impact of Improved Programming Technology." Chap. in *Proceedings of 1975 ACM Annual Conference*. 288-290. New York: Association for Computing Machinery, 1975.
- Dijkstra, Edsger. "The Humble Programmer." Chap. in *ACM Turing Award Lectures: The First Twenty Years, 1966-1985*. New York: ACM Press, 1987.
- Dijkstra, Edsger. "Go To Statement Considered Harmful." *Communications of the ACM* 11, 3 (1968): 147-48.
- Editors of DATA-LINK. "What's in a Name? (letter to editor)." *Communications of the ACM* 1, 4 (1958): 6.
- Fein, Louis. "The Role of the University in Computers, Data Processing, and Related Fields." *Communications of the ACM* 2, 10 (1959): 7-14.
- Fein, Louis. "ACM Has a Crisis of Identity?." *Communications of the ACM* 10, 1 (1967): 1.
- Fike, John. "Vultures Indeed." *Datamation* 13, 5 (1967): 12.
- Flamm, Kenneth. *Creating the computer government, industry, and high technology*. Washington, D.C: Brookings Institute, 1988.
- Flywheel, Wolf. "Letter to the editor (on professionalism)." *Datamation* 5, 5 (1959): 2.
- Forest, Robert. "EDP People: Review and Preview." *Datamation* 18, 6 (1972): 65-67.
- Fritz, W. Barkley. "The Women of Eniac." *Annals of the History of Computing* 18, 3 (1996): 13-23.
- Fulkerson, L. "Should there be a CS Undergraduate Program? (letter to editor)." *Communications of the ACM* 10, 3 (1967): 148.

- Galler, Bernard. "The AFIPS Constitution (President's Letter to ACM Membership)." *Communications of the ACM* 12, 3 (1969): 188.
- Galler, Bernard. "The Journal (President's Letter to ACM Membership)." *Communications of the ACM* 12, 2 (1969): 65-66.
- Gass, Saul. "ACM class structure (letter to editor)." *Communications of the ACM* 2, 5 (1959): 4.
- Geckle, Jerome. "Letter to the editor." *Datamation* 11, 9 (1965): 12-13.
- Gibbs, W. "Software's Chronic Crisis." *Scientific American*, September 1994, 86.
- Golda, John. "The Effects of Computer Technology on the Traditional Role of Management." MBA thesis, Wharton School, University of Pennsylvania. 1965.
- Goldstine, Herman. *The Computer from Pascal to von Neumann*. Princeton: Princeton University Press, 1972.
- Goodman, William, "The software and engineering industries: threatened by technological change?." Bureau of Labor Statistics. *Monthly Labor Review* (August 1996).
- Gordon, Robert. "Review of Charles Lecht, The Management of Computer Programmers." *Datamation* 14, 4 (1968): 200-202.
- Gotterer, Malcolm. "The Impact of Professionalization Efforts on the Computer Manager." Chap. in *Proceedings of 1971 ACM Annual Conference*. 367-375. New York: Association for Computing Machinery, 1971.
- Greenbaum, Joan. *In the Name of Efficiency: Management Theory and Shopfloor Practice in Data-Processing Work*. Philadelphia: Temple University Press, 1979.
- Greenbaum, Joan. "On twenty-five years with Braverman's 'Labor and Monopoly Capital.' (Or, how did control and coordination of labor get into the software so quickly?)." *Monthly Review* 50, 8 (1999).

- Greenwood, Frank. "Education for Systems Analysis: Part One." *Systems & Procedures Journal* (Jan-Feb, 1966): 13-15.
- Grier, David Allan. "The ENIAC, the verb to program and the Emergence of Digital Computers." *Annals of the History of Computing* 18, 1 (1996).
- Grosch, Herb. "Programmers: The Industry's Cosa Nostra." *Datamation* 12, 10 (1966): 202.
- Grosch, Herb. "Computer People and their Culture." *Datamation* 7, 10 (1961): 51-52.
- Grosch, Herb. "Software in Sickness and Health." *Datamation* 7, 7 (1961): 32-33.
- Grosch, Herb. "Plus and Minus." *Datamation* 5, 6 (1959): 51.
- Gruenberger, Fred. "Problems and Priorities." *Datamation* 18, 3 (1972): 47-50.
- Gruenberger, Fred, and Stanley Naftaly, eds. *Data Processing. Practically Speaking*. Los Angeles: Data Processing Digest, 1967.
- Guarino, Roger. "Managing Data Processing Professionals." *Personnel Journal* (Dec., 1969): 972-975.
- Guerrieri, J.A. "Certification: Evolution, Not Revolution." *Datamation* 14, 11 (1973): 101.
- Hamming, Richard. "One Man's View of Computer Science." Chap. in *ACM Turing Award Lectures: The First Twenty Years, 1966-1985*. 207-218. New York: ACM Press, 1987.
- Hanke, John, William Boast, and John Fellers. "Education and Training of a Business Programmer." *Journal of Data Management* 3, 6 (June, 1965): 38-53.
- Hashagen, U., A. Norberg, and R. Keil-Slawik, eds. *Mapping the History of Computing: Software Issues*. New York: Springer-Verlag, 2001.
- Head, Robert. "Controlling Programming Costs." *Datamation* 13, 7 (1967): 141-142.

- Herz, D. *New Power for Management*. New York: McGraw-Hill, 1969.
- Hoare, Anthony. "Keynote Address: Software Engineering." *3rd International Conference on Software Engineering Proceedings*, 1974.
- Jacobson, Arvid, ed. *Proceedings of the First Conference on Training Personnel for the Computing Machine Field*. Detroit: Wayne University Press, 1955.
- Jay, Anthony. *Corporation Man*. New York: Random House, 1971.
- Jenks, James. "Starting Salaries of Engineers are Deceptively High." *Datamation* 13, 1 (1967): 13.
- Jones, Richard. "A time to assume responsibility." *Datamation* 13, 9 (1967): 160.
- Kaufman, Louis, and Richard Smith. "Let's Get Computer Personnel on the Management Team." *Training and Development Journal* (December, 1966): 25-29.
- Keelan, C.I. "Controlling Computer Programming." *Journal of Systems Management* (Jan., 1969): 30-33.
- Kraft, Philip. *Programmers and Managers: The Routinization of Computer Programming in the United States*. New York: Springer-Verlag, 1977.
- Kuch, T.D.C. "Unions or licensing? or both? or neither?." *Infosystems* (January 1973): 42-43.
- Larson, Harry. "EDP - A 20 Year Ripoff!." *Infosystems* (November 1974): 26-30.
- Larson, Magali Sarfatti. *The Rise of Professionalism: A Sociological Analysis*. Berkeley: University of California Press, 1977.
- Layton, Edwin. *The Revolt of the Engineers: Social Responsibility and the American Engineering Profession*. Baltimore: Johns Hopkins University Press, 1971.
- Leavitt, Harold, and Thomas Whisler. "Management in the 1980's." *Harvard Management Review* 36, 6 (1958): 41-48.

- Lecht, Charles. *The Management of Computer Programming Projects*. New York: American Management Association, 1967.
- Ledbetter, William. "Programming Aptitude: How Significant is It?." *Personnel Journal* 54, 3 (March, 1975): 165-166, 175.
- Leslie, Harry. "The Report Program Generator." *Datamation* (26-28) 13, 6 (1967).
- Libellator. "Programming Personalities in Europe." *Datamation* 12, 9 (1966): 28-29.
- Lucas, Henry S. "On the failure to implement structured programming and other techniques." Chap. in *Proceedings of 1975 ACM Annual Conference*. 291-293. New York: Association for Computing Machinery, 1975.
- Mahoney, Michael. "The History of Computing in the History of Technology." *Annals of the History of Computing* 10 (1988).
- Mahoney, Michael. "Computer Science: The Search for a Mathematical Theory." In *Science in the 20th Century*, Krige, John; Pestre, Dominique. Amsterdam: Harwood Academic Publishers, 1997.
- Mahoney, Michael. "Software: the self-programming machine." In *Creating Modern Computing*, Akera, Atsushi; Nebeker, F.
- Markham, Edward. "Selecting a Private EDP School." *Datamation* 14, 5 (1968): 33-40.
- Markham, Edward. "EDP Schools - An Inside View." *Datamation* 14, 4 (1968): 22-27.
- McClure, Carma. *Managing Software Development and Maintenance*. New York: Van Nostrand Rheinhold, 1981.
- McCracken, Daniel. "The Human Side of Computing." *Datamation* 7, 1 (1961): 9-11.
- McCracken, Daniel. "The Software Turmoil: Nine Predictions for '62." *Datamation* 8, 1 (1962): 21-22.

McCracken, Daniel. "Is There FORTRAN In Your Future?." *Datamation* 19, 5 (1973): 236-237.

McGowan, Clement, and John Kelly. *Top-Down Structured Programming Techniques*. New York: Petrocelli/Carter, 1975.

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

McKinsey & Company. "Unlocking the Computer's Profit Potential." *Computers & Automation* (April 1969): 24-33.

McMurrer, J.A., and J.R. Parish. "The People Problem." *Datamation* 16, 7 (1970): 57-59.

McNamara, W.J., and J.L. Hughes. "A Review of Research on the Selection of Computer Programmers." *Personnel Psychology* 14, 1 (Spring, 1961): 39-51.

Menkhaus, Edward. "EDP: Nice Work If You Can Get It." *Business Automation* (March 1969): 41-45, 74.

Metropolis, Nick, J. Howlett, and Gian-Carlo Rota, eds. *A history of computing in the twentieth century a collection of essays*. New York: Academic Press, 1980.

Metzger, Philip. *Managing a Programming Project*. Englewood Cliffs, N.J: Prentice-Hall, 1973.

Mitre Corporation. "Are You Working Your Way Toward Obsolescence." *Datamation* 12, 6 (1966): 99.

Morgan, H.L., and J.V. Soden. "Understanding MIS Failures." *Data Base* (Winter 1973): 157-171.

Morrison, David. "Software Crisis." *Defense* 21, 2 (1989): 72.

Mumford, Enid. *Job Satisfaction: A study of computer specialists*. London: Longman Group Limited, 1972.

- Myers, Charles, ed. *The Impact of Computers on Management*. Cambridge, MA: MIT Press, 1967.
- Naur, Peter, Brian Randall, and J.N. Buxton, ed. *Software engineering Proceedings of the NATO conferences*. New York: Petrocelli/Carter, 1976.
- Noble, David F. *Forces of Production: A Social History of Industrial Automation*. New York: Alfred A. Knopf, 1984.
- Noble, David F. *America By Design: Science, Technology and the Rise of Corporate Capitalism*. Oxford: Oxford University Press, 1977.
- O'Shields, Joseph. "Selection of EDP Personnel." *Personnel Journal* 44, 9 (October 1965): 472-474.
- Oettinger, Anthony. "The Hardware-Software Complexity." *Communications of the ACM* 10, 10 (1967): 606-606.
- Oettinger, Anthony. "On ACM's Responsibility (President's Letter to ACM Membership) (1966)." *Communications of the ACM* 9, 8 (1966): 545-546.
- Oettinger, Anthony. "ACM sponsors professional development program (President's Letter to ACM Membership)." *Communications of the ACM* 9, 10 (1966): 712-713.
- Oettinger, Anthony. "President's reply to Louis Fein." *Communications of the ACM* 10, 1 (1967): 1,61.
- Ogdin, J.L. "The mongolian hordes versus superprogrammer." *Infosystems* (December 1973): 20-23.
- Opler, Ascher. "Trends in Programming Concepts." *Datamation* 7, 1 (1961): 13-15.
- Orden, Alex. "The Emergence of a Profession." *Communications of the ACM* 10, 3 (1967): 145-147.
- Overton, Scott. "Programmer/Analyst: The Merger of Diverse Skills." *Personnel Journal* 52, 7 (July, 1972): 511-513.

Parnas, David. "On the preliminary report of C3S (letter to editor)." *Communications of the ACM* 9, 4 (1966): 242-243.

Paschell, William. *Automation and employment opportunities for office workers; a report on the effect of electronic computers on employment of clerical workers*. Washington, D.C: Bureau of Labor Statistics, 1958.

Patrick, Robert. "The Gap in Programming Support." *Datamation* 7, 5 (1961): 37.

Paul. James, and Simon. Gregory, "Bugs in the Program: Problems in Federal Government Computer Software Development and Regulation." *Subcommittee on Investigations and Oversight of the House Committee on Science, Space, and Technology* (1989).

Payne, Robert. "Reaction to Publication Proposal (letter to editor)." *Communications of the ACM* 8, 1 (1965): 71.

Personnel Journal Editorial. "Professionalism Termed Key to Computer Personnel Situation." *Personnel Journal* 51, 2 (February, 1971): 156-157.

Philips, Charles. "Report from the Committee on Data Systems Languages." *Association for Computing Machinery*, Boston, September 1, 1959.

Phillips, Charles. "Report from the Committee on Data Systems Languages.," Boston, MA, Association for Computing Machinery. September 1, 1959. Cited in Wexelblatt, p. 200.

Porat, Avner, and James Vaughan. "Computer Personnel: The New Theocracy - or Industrial Carpetbaggers." *Personnel Journal* 48, 6 (1968): 540-543.

Postley, John. "Letter to Editor." *Communications of the ACM* 3, 1 (1960): A6.

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

RAND Symposium. "On Programming Languages, Part I." *Datamation* 8, 10 (1962): 25-32.

RAND Symposium. "On Programming Languages, Part II." *Datamation* 8, 11 (1962): 23-30.

- RAND Symposium. "Defining the Problem, Part II." *Datamation* 11, 9 (1965): 66-73.
- RAND Symposium. "Is It Overhaul or Trade-in Time? Part II." *Datamation* 5, 5 (1959): 17-26,44-45.
- Reid, H.V. "Problems in Managing the Data Processing Department." *Journal of Systems Management* (May, 1970): 8-11.
- Reinstedt, R.N, and Raymond Berger. "Certification: A Suggested Approach to Acceptance." *Datamation* 19, 11 (1973): 97-100.
- Rhee, H.A. *Office Automation in Social Perspective: The Progress and Social Implications of Electronic Data Processing*. Oxford: Basil Blackwell, 1968.
- Rings, N. "Programmers and Longevity." *Datamation* 12, 12 (1966).
- Romberg, Bernhard. "Managing software, It can be done." *Infosystems* (December 1973): 42-43.
- Rose, Michael. *Computers, Managers, and Society*. Harmondsworth: Penguin, 1969.
- Rosen, Saul, ed. *Programming Systems and Languages*. New York: McGraw-Hill, 1967.
- Rosin, Robert. "Relative to the President's December Remarks." *Communications of the ACM* 10, 6 (1967): 342.
- Ross, David. "Certification and Accreditation." *Datamation* 14, 9 (1968): 183-184.
- Rothery, Brian. *Installing and Managing a Computer*. London: Business Books, 1968.
- Rowan, T.C. "The Recruiting and Training of Programmers." *Datamation* 4, 3 (1958): 16-18.
- Sackman, Hal. "Conference on Personnel Research." *Datamation* 14, 7 (1968): 74-76, 81.

Sackman, Hal, W.J. Erickson, and E.E. Grant. "Exploratory Experimental Studies Comparing Onling and Offline Programming Performance." *Communications of the ACM* 11, 1 (1968): 3-11.

Sammett, Jean. *Programming Languages: History and Fundamentals*. Englewood Cliffs, N.J: Prentice-Hall, 1969.

Sanden, Bo. "Programming masters break out of the managerial mold." *Computerworld*, June 16, 1986.

Saxon, James. "Programming Training: A Workable Approach." *Datamation* 9, 12 (1963): 48-50.

Shapiro, Stuart. "Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering." *Annals of the History of Computing* 19, 1 (1997): 20-54.

Shaw, Christopher. "Programming Schisms." *Datamation* 8, 9 (1962): 32.

Shneiderman, Ben. "The Relationship Between COBOL and Computer Science." *Annals of the History of Computing* 7, 4 (1985): 348-352.

Sidlo, C.M. "The Making of a Profession (letter to editor)." *Communications of the ACM* 4, 8 (1961): 366-367.

Simon, Herbert, Allen Newell, and Alan Perlis. "Computer Science (letter to editor)." *Science* 157, 3795 (Sept. 22, 1967): 1373-1374.

Sperry Rand Corp. "Programming for the UNIVAC System.," January, 1953. Box 372, Accession 1825, Hagley Achives, Sperry-Univac collection.

Stewart, Rosemary. *How Computers Affect Management*. Cambridge, MA: MIT Press, 1971.

Stone, Milt. "In Search of an Identity." *Datamation* 18, 3 (1972): 52-59.

Tanaka, Richard. "Fee or Free Software." *Datamation* 13, 10 (1967): 205-206.

-
- Taylor, Frederick Winslow. *The Principles of Scientific Management*. New York: W.W. Norton & Company, 1911.
- Tropp, H.S. "ACM's 20th Anniversary: 30 August 1967." *Annals of the History of Computing* 9, 3 (1988): 269.
- Tucker, Allan. *Programming Languages*. Reading, MA: Addison-Wesley, 1977.
- United States Department of Labor, "Manpower Development and Outlook in the Computing and Accounting Machines Industry." (May, 1968) Industry Manpower Surveys. reprinted from Area Trends in Employment and Unemployment, April 1968.
- Vincenti, Walter. *What engineers know and how they know it analytical studies from aeronautical history*. Baltimore: Johns Hopkins University Press, 1990.
- Walker, W.R. "MIS Mysticism (letter to editor)." *Business Automation* 16, 7 (1969): 8.
- Ware, Willis. "As I See It: A Guest Editorial." *Datamation* 11, 5 (1965): 27-28.
- Ware, Willis. "AFIPS in Retrospect." *Annals of the History of Computing* 8, 3 (1986): 303-311.
- Webster, Bruce. "The Real Software Crisis." *Byte Magazine* 21, 1 (1996): 218.
- Wegener, Peter. "Three Computer Cultures: Computer Technology, Computer Mathematics, and Computer Science." *Advances in Computers* 10 (1970): 7-78.
- Weinberg, Gerald. *The Psychology of Computer Programming*. New York: Van Nostrand Rheinhold, 1971.
- Weiss, Eric. "Publications in Computing: An Informal Review." *Communications of the ACM* 15, 7 (1972): 492-297.
- Wesoff, Jay. "The Systems People Blues." *Datamation* 14, 6 (1968): 10-11.

Wexelblat, Richard, ed. *History of programming languages*. New York: Academic Press, 1981.

White, Thomas. "The 70's: People." *Datamation* 16, 7 (1973): 40-46.

Wilensky, Harold. "The Professionalization of Everyone?." *American Journal of Sociology* 70, 2 (1964): 137-158.

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

Williams, Michael. *A History of Computing Technology*. Washington, D.C: IEEE Computer Society Press, 1997.

Wishner, Raymond. "Comment on Curriculum 68." *Communications of the ACM* 11, 10 (1968): 658.

Xerox Corporation. "At Xerox, we look at programmers and see managers (ad)." *Datamation* 14, 4 (1968).

Yates, JoAnne. *Control Through Communication: The Rise of System in American Management*. Baltimore: Johns Hopkins University Press, 1989.

Yourdon, Edward, ed. *Classics in Software Engineering*. New York: Yourdon Press, 1979.

Zaphyr, P.A. "The science of hypology (letter to editor)." *Communications of the ACM* 2, 1 (1959): 4.

Zussman, Robert. *Mechanics of the Middle Class: Work and Politics Among American Engineers*. Berkeley: University of California Press, 1985.