Code Nation

Personal Computing and the Learn to Program Movement in America

Michael J. Halvorson

Code Nation explores the rise of software development as a social, cultural, and technical phenomenon in American history. The movement germinated in government and university labs during the 1950s, gained momentum through corporate and counterculture experiments in the 1960s and 1970s, and became a broad-based computer literacy movement in the 1980s. As personal computing came to the fore, learning to program was transformed by a groundswell of popular enthusiasm, exciting new platforms, and an array of commercial practices that have been further amplified by distributed computing and the Internet. The resulting society can be depicted as a “Code Nation”—a globally-connected world that is saturated with computer technology and enchanted by software and its creation.

Code Nation is a new history of personal computing that emphasizes the technical and business challenges that software developers faced when building applications for CP/M, MS-DOS, UNIX, Microsoft Windows, the Apple Macintosh, and other emerging platforms. It is a popular history of computing that explores the experiences of novice computer users, tinkerers, hackers, and power users, as well as the ideals and aspirations of leading computer scientists, engineers, educators, and entrepreneurs. Computer book and magazine publishers also played important, if overlooked, roles in the diffusion of new technical skills, and this book highlights their creative work and influence.

Code Nation offers a “behind-the-scenes” look at application and operating-system programming practices, the diversity of historic computer languages, the rise of user communities, early attempts to market PC software, and the origins of “enterprise” computing systems. Code samples and over 80 historic photographs support the text. The book concludes with an assessment of contemporary efforts to teach computational thinking to young people.
Code Nation
ACM Books

Editor in Chief
Sanjiva Prasad, Indian Institute of Technology (IIT) Delhi

Founding Editor
M. Tamer Özsu, University of Waterloo

ACM Books is a series of high-quality books for the computer science community, published by ACM in collaboration with Morgan & Claypool Publishers. ACM Books publications are widely distributed in both print and digital formats through booksellers and to libraries (and library consortia) and individual ACM members via the ACM Digital Library platform.

Computing and the National Science Foundation, 1950–2016: Building a Foundation for Modern Computing
Peter A. Freeman, Georgia Institute of Technology
W. Richards Adrion, University of Massachusetts Amherst
William Aspray, University of Colorado Boulder
2019

Providing Sound Foundations for Cryptography: On the work of Shafi Goldwasser and Silvio Micali
Oded Goldreich, Weizmann Institute of Science
2019

Concurrency: The Works of Leslie Lamport
Dahlia Malkhi, VMware Research and Calibra
2019

The Essentials of Modern Software Engineering: Free the Practices from the Method Prisons!
Ivar Jacobson, Ivar Jacobson International
Harold “Bud” Lawson, Lawson Konsult AB (deceased)
Pan-Wei Ng, DBS Singapore
Paul E. McMahon, PEM Systems
Michael Goedicke, Universität Duisburg-Essen
2019

Data Cleaning
Ihab F. Ilyas, University of Waterloo
Xu Chu, Georgia Institute of Technology
2019
Conversational UX Design: A Practitioner’s Guide to the Natural Conversation Framework
Robert J. Moore, IBM Research–Almaden
Raphael Arar, IBM Research–Almaden
2019

Heterogeneous Computing: Hardware and Software Perspectives
Mohamed Zahran, New York University
2019

Hardness of Approximation Between P and NP
Aviad Rubinstein, Stanford University
2019

The Handbook of Multimodal-Multisensor Interfaces, Volume 3: Language Processing, Software, Commercialization, and Emerging Directions
Editors: Sharon Oviatt, Monash University
Björn Schuller, Imperial College London and University of Augsburg
Philip R. Cohen, Monash University
Daniel Sonntag, German Research Center for Artificial Intelligence (DFKI)
Gerasimos Potamianos, University of Thessaly
Antonio Krüger, Saarland University and German Research Center for Artificial Intelligence (DFKI)
2019

Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker
Editor: Michael L. Brodie, Massachusetts Institute of Technology
2018

The Handbook of Multimodal-Multisensor Interfaces, Volume 2: Signal Processing, Architectures, and Detection of Emotion and Cognition
Editors: Sharon Oviatt, Monash University
Björn Schuller, University of Augsburg and Imperial College London
Philip R. Cohen, Monash University
Daniel Sonntag, German Research Center for Artificial Intelligence (DFKI)
Gerasimos Potamianos, University of Thessaly
Antonio Krüger, Saarland University and German Research Center for Artificial Intelligence (DFKI)
2018

Declarative Logic Programming: Theory, Systems, and Applications
Editors: Michael Kifer, Stony Brook University
Yanhong Annie Liu, Stony Brook University
2018
The Sparse Fourier Transform: Theory and Practice
Haitham Hassanieh, University of Illinois at Urbana-Champaign
2018

The Continuing Arms Race: Code-Reuse Attacks and Defenses
Editors: Per Larsen, Immunant, Inc.
Ahmad-Reza Sadeghi, Technische Universität Darmstadt
2018

Frontiers of Multimedia Research
Editor: Shih-Fu Chang, Columbia University
2018

Shared-Memory Parallelism Can Be Simple, Fast, and Scalable
Julian Shun, University of California, Berkeley
2017

Computational Prediction of Protein Complexes from Protein Interaction Networks
Sriganesh Srihari, The University of Queensland Institute for Molecular Bioscience
Chern Han Yong, Duke-National University of Singapore Medical School
Limsoon Wong, National University of Singapore
2017

The Handbook of Multimodal-Multisensor Interfaces, Volume 1: Foundations, User Modeling, and Common Modality Combinations
Editors: Sharon Oviatt, Inca Designs
Björn Schuller, University of Puebla and Imperial College London
Philip R. Cohen, Voicebox Technologies
Daniel Sonntag, German Research Center for Artificial Intelligence (DFKI)
Gerasimos Potamianos, University of Thessaly
Antonio Krüger, Saarland University and German Research Center for Artificial Intelligence (DFKI)
2017

Communities of Computing: Computer Science and Society in the ACM
Thomas J. Misa, Editor, University of Minnesota
2017

Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining
ChengXiang Zhai, University of Illinois at Urbana–Champaign
Sean Massung, University of Illinois at Urbana–Champaign
2016
An Architecture for Fast and General Data Processing on Large Clusters  
Matei Zaharia, Stanford University  
2016

Reactive Internet Programming: State Chart XML in Action  
Franck Barbier, University of Pau, France  
2016

Verified Functional Programming in Agda  
Aaron Stump, The University of Iowa  
2016

The VR Book: Human-Centered Design for Virtual Reality  
Jason Jerald, NextGen Interactions  
2016

Ada’s Legacy: Cultures of Computing from the Victorian to the Digital Age  
Robin Hammerman, Stevens Institute of Technology  
Andrew L. Russell, Stevens Institute of Technology  
2016

Edmund Berkeley and the Social Responsibility of Computer Professionals  
Bernadette Longo, New Jersey Institute of Technology  
2015

Candidate Multilinear Maps  
Sanjam Garg, University of California, Berkeley  
2015

Smarter Than Their Machines: Oral Histories of Pioneers in Interactive Computing  
John Cullinane, Northeastern University; Mossavar-Rahmani Center for Business and Government, John F. Kennedy School of Government, Harvard University  
2015

A Framework for Scientific Discovery through Video Games  
Seth Cooper, University of Washington  
2014

Trust Extension as a Mechanism for Secure Code Execution on Commodity Computers  
Bryan Jeffrey Parno, Microsoft Research  
2014

Embracing Interference in Wireless Systems  
Shyamnath Gollakota, University of Washington  
2014
Contents

Acknowledgments xiii

PART I LEARNING TO CODE 1

Chapter 1 How Important is Programming? 3
1.1 Programming Culture 5
1.2 Learning a Language 7
1.3 New Ways of Thinking 8
1.4 Equity and Access 13
1.5 Personal Connections 15
1.6 Manifestos of the Movement 17
1.7 A New History of Personal Computing 19

Chapter 2 Four Computing Mythologies 25
2.1 The NATO Conference on Software Engineering 27
2.2 The Complexity of Software 32
2.3 Systems are for Customers 35
2.4 The Counterculture Movement 39
2.5 Everything is Deeply Intertwined 45
2.6 The Birth of Computer Science 49
2.7 Computers for the People 54
2.8 Personal Computing 58

Chapter 3 FORTRAN, Logo, and the Tower of Babel 63
3.1 Solving Problems with Computers 65
3.2 The Tower of Babel 70
3.3 High-level Languages 75
3.4 Learning FORTRAN 78
3.5 Daniel McCracken’s Primers 83
3.6 Seymour Papert and Logo 87
3.7 Cynthia Solomon  92
3.8 Logo as a Model for Code Nation  93
3.9 How successful was Logo?  95

Chapter 4  Advocating Computer Literacy  99
4.1 Robert Albrecht and the Popularization of the Movement  100
4.2 I Speak BASIC  103
4.3 The B. F. Skinner Approach  108
4.4 Hold Me Closer Tiny BASIC  110
4.5 Arthur Luehrmann and the Computer Literacy Debate  112
4.6 A Blow to the Movement  120
4.7 Apple Computer’s Education Agenda  121
4.8 Applications over Languages  123

Chapter 5  Four Million BASIC Programmers  127
5.1 Introducing David Ahl  128
5.2 A Proliferation of BASICS  134
5.3 IBM BASICA  135
5.4 Adventure Games  137
5.5 Structured Programming  141
5.6 Microsoft Press and *Learn BASIC Now*  145
5.7 Microsoft Game Shop  153
5.8 Visual Basic for Windows  156
5.9 Innovative Programming Primers  159

PART II  HOBBYIST AND HACKER CULTURES  167

Chapter 6  Power Users, Tinkerers, and Gurus  169
6.1 Computing Terminology  171
6.2 Tinkering with Personal Computers  174
6.3 Van Wolverton and Batch Files  176
6.4 The *DOS for Dummies* Phenomenon  183
6.5 The Economic Impact of Personal Computers  187
6.6 Cary Lu Introduces the Macintosh  188
6.7 The Waite Group’s Macintosh Primers  192
6.8 The Maturing Mac Platform  200

Chapter 7  Hackers and Cyberpunks  205
7.1 Bill Landreth and 1980s Hacker Culture  206
### Contents

#### Chapter 7

7.2 Jude Milhon: From Civil Rights Activist to Cyberpunk 211  
7.3 *Mondo 2000* and *The Cyberpunk Handbook* 217  
7.4 Cypherpunks and Cryptography 222

#### Chapter 8

**Computer Magazines and Historical Research** 227  
8.1 Magazines and a Popular Culture of Computing 230  
8.2 Letters from the Programming Community 235  
8.3 New PC Users 236  
8.4 Power Users 241  
8.5 Advanced Hobbyists 245  
8.6 Professional Programmers 248  
8.7 New Approaches to Historical Research 252

#### PART III

**PROFESSIONAL PROGRAMMING CULTURES** 255

#### Chapter 9

**Developing for MS-DOS: Authors and Entrepreneurs** 257  
9.1 New Platforms for Commercial Software 259  
9.2 *Inside the IBM PC* with Peter Norton 262  
9.3 Borland’s Turbo Pascal 270  
9.4 Ray Duncan’s *Advanced MS-DOS* 274  
9.5 *The MS-DOS Encyclopedia* 281  
9.6 MS-DOS Sample Code 283  
9.7 Technology Diffusion 285

#### Chapter 10

**C Programming Nation: From Tiny C to Microsoft Windows** 289  
10.1 The C Language 290  
10.2 Learning C on Personal Computers 293  
10.3 Academic and Professional Resources 296  
10.4 C Programming for the People 299  
10.5 Charles Petzold’s *Programming Windows* 306  
10.6 On Complexity 316

#### Chapter 11

**“Evangelism is sales done right”: PCs and Commercial Programming Culture** 321  
11.1 The Macintosh Way 325  
11.2 The West Coast Computer Faire 328  
11.3 COMDEX and the Trade Show Movement 332  
11.4 The Trouble with Self-taught Programmers 339  
11.5 Software Engineering for the People 342
11.6 Professional and Enterprise Development Systems 346
11.7 Commercialization 350

Afterword: Programming in the Internet Age 357

Author's Biography 375

Index 377
I would like to thank the many friends, colleagues, and supporting institutions that have helped me bring this book to you. As all historians know, writers and researchers have many debts, and only some of them are repaid in the acknowledgments. As I complete this project, I am especially aware of the creative people and research institutions that have supported the Code Nation project over the past 5 years. I am also deeply aware of the many teachers and mentors that I have had the privilege to work with in a long career related to computing and higher education. This book is dedicated to you all, with my heartfelt thanks.

At ACM Books, I would like to thank Thomas Misa, who initiated this project and encouraged me as I worked on the early chapters. Tamer Ozsu and Achi Dosanjh provided helpful guidance as the project took shape and made its way into the publishing system at the venerable Association for Computing Machinery (ACM). Also at the ACM, I would like to thank Barbara Ryan, for her help with permissions, and Bernadette Shade, for her support with print production and graphics. Kim Halvorson sized and edited many of the images included before they entered production. Finally, I am grateful to Karen Grace for her careful editorial work as this book made its way through the publishing system.

At Pacific Lutheran University (PLU), I would like to thank the Provost and President’s Offices for taking an interest in the Code Nation project and offering a sabbatical to support the writing of the first chapters. I am very grateful to the talented staff of the Mordvedt Library, who helped me locate many obscure books and magazines through interlibrary loans. My history, computer science, and innovation studies colleagues have been welcoming conversation partners throughout the project. Damian Alessandro contributed research to Chapter 2, and we shared many enjoyable chats about the early years of Apple Computer and counterculture activities in the San Francisco Bay Area. Michael Schleeter in the Department of Philosophy has been a great teaching partner, and I have benefited greatly from his wisdom about ethics and technology. Finally, Dale and Jolita Benson have been very
generous supporters of this book from its beginning, and I appreciate their many contributions to the Business and Economic History program at PLU.

At the University of Washington, Steven Pfaff in the Department of Sociology offered many helpful comments, especially ideas related to “technology enthusiasm” and using computing mythologies as a way to tease out the subtle relationships between technology initiatives and popular movements in the U.S. The staff of the Engineering Library helped me locate many obscure computer books and magazines.

At the Computer History Museum in Fremont, California, Sara Lott helped me to locate and attain permissions for many of the images included in this book. Sydney Olson welcomed me for several on-site visits to the museum’s fantastic archive and reading room, and she tracked down materials in the collection when I had no idea where to look.

At Code.org, I am grateful for the help of Hadi Partovi, Alice Steinglass, and Lian Swanson, who connected me with resources about the Hour of Code campaign and the group’s promising strategies for expanding access to computer science education in schools.

Searching for images and materials took me to numerous corporations, institutions, and databases in the U.S. I am deeply grateful to the following locations for sharing their resources with me: the Alabama Department of Archives and History, Apple Computer, the ACM, Code.org, Dartmouth College, DEC, Getty Images, Hewlett-Packard, IBM, John Wiley & Sons, Microsoft, Macworld, O'Reilly Media, PC/Computing, PC Magazine, PC World, Pearson, Penguin Random House, Springer Nature, University of Minnesota Libraries/Charles Babbage Institute, University of Washington Libraries, and Ziff Davis.

Oral history played an important role in this project, and I would like to thank the following people who shared interviews, email correspondence, photos, and/or printed materials with me: Renzhi Cao, Ray Duncan, Kevin Eagan, Lee Felsenstein, John Froshauer, Ken Goffman, Dan Gookin, Kim Halvorson, Dean Holmes, Alison Bailey Kennedy, André LaMothe, Dail Magee, Jr., Theresa Mannix, Robert M. McClure, Bart Nagel, Ted Nelson, Larry Osterman, Charles Petzold, Brian Randell, Jeffrey Richter, David Rygmyr, Megan Sheppard, Amy Stevenson, Patty Stonesifer, Mitchell Waite, Jim Warren, and Van Wolverton.

Finally, I reserve my deepest gratitude for my family, who supported me in untold ways as this project made its way from a collection of curious ideas to a completed book about the history of programming and personal computing. My sons Henry and Felix grew up hearing tales about early personal computers, and we have often chatted about software, music, and computer gaming. Kim has taken it all in stride because she was there for most of it, and often sees what other people miss. I am immeasurably grateful for my life with you.
PART

LEARNING TO CODE
How Important is Programming?

“How important is it for you to learn to program a computer?

Since the introduction of the first digital electronic computers in the 1940s, people have answered this question in surprisingly different ways.

During the first wave of commercial computing—in the 1950s and 1960s, when large and expensive mainframe computers filled entire rooms—the standard advice was that only a limited number of specialists would be needed to program computers using simple input devices like switches, punched cards, and paper tape. Even during the so-called “golden age” of corporate computing in America—the mid- to late 1960s—it was still unclear how many programming technicians would be needed to support the rapid computerization of the nation’s business, military, and commercial operations. For a while, some experts thought that well-designed computer systems might eventually program themselves, requiring only a handful of attentive managers to keep an eye on the machines.

By the late 1970s and early 1980s, however, the rapid emergence of personal computers (PCs), and continuing shortages of computer professionals, shifted popular thinking on the issue. When consumers began to adopt low-priced PCs like the Apple II (1977), the IBM PC (1981), and the Commodore 64 (1982) by the millions, it seemed obvious that ground-breaking changes were afoot. The “PC Revolution” opened up new frontiers, employed tens of thousands of people, and (according to some enthusiasts) demanded new approaches to computer literacy. As Ted Nelson, a prolific inventor and computing advocate wrote, “You can and must understand computers NOW!” On learning to program computers, Nelson energetically compared programming to another American obsession—driving an
automobile. “If you’ve never written a program, it’s like never having driven a car,” Nelson instructed. “You may get the general idea, but you may have little clear sense of the options, dangers, constraints, possibilities, difficulties, limitations, and complications.”

Ted Nelson was not alone. By the late 1970s, scores of programming advocates recommended that people of all ages learn to code as a way of understanding what the world’s most intriguing devices were capable of. *Computer programming*—a process of formulating a problem for the computer to solve, writing instructions in a given computer language, loading instructions into the computer’s memory, running the program, and correcting errors—had emerged as a major late-night pastime and (for some) a promising profession. In response to the mandate of Nelson and others, a surge of interest in programming developed, and the number of people who could write at least elementary programs grew from several thousand in

1.1 Programming Culture

This book is about the rise of computer programmers and the emerging social, technical, and commercial worldview that I call *programming culture*, which took a distinctive form during the early decades of microcomputers and personal computing, *c.* 1970–1995. It is a popular history of coding that explores the experiences of novice computer users, tinkerers, hackers, and power users, as well as the ideals and aspirations of computer scientists, educators, engineers, and entrepreneurs. A central part of this story is the *learn-to-program movement*, which germinated in government and university labs during the 1950s, gained momentum through counterculture experiments in the 1960s and early 1970s, became a broad-based educational agenda in the late 1970s and early 1980s, and was transformed by commercialization practices in the 1990s and 2000s. The learn-to-program movement sought to make computers more understandable, imprint useful technical skills, establish shared values, build virtual communities, and offer economic opportunities for technology enthusiasts. The movement also supported user communities, schools, and emerging commercial industries, many of which benefited from the utility and connectivity provided by digital electronic computers.

The learn-to-program movement had its ups and downs, but eventually set the stage for 21st century expressions of computational literacy, such as the Hour of Code, YouTube and Lynda courseware, certification programs, coding boot camps, and university degrees in disciplines such as computer science, software engineering, information technology, artificial intelligence, and (most recently) human–computer interaction. As the title of this book suggests, the learn-to-program movement fostered a groundswell of popular support for computing culture in America, resulting in what I call a *Code Nation*—a globally-connected society that is saturated with computer technology and enchanted by software and its creation.

The learn-to-program movement (or more broadly, the software-maker movement) has inspired both disciples and critics. It has evolved over time and its advocates have traversed numerous professional boundaries and cultural institutions. The movement is historically distinct but also follows the patterns and rhythms of earlier socio-technical transformations, including the introduction of steam-powered technologies during the Industrial Revolution, the electrification of American businesses and homes, and the production of automobiles and “car culture” in the early 20th century.
Borrowing terminology from information science and the history of technology, the learn-to-program movement is identifiable as part of the “diffusion” and “domestication” phases that take place when a successful new technology is spread or “propagated” across society.\(^2\) Scholars from the field of business and economic history also recognize this transition as a key period in which a new discovery or invention is widely adopted and made useful for the general public, resulting in new consumer behaviors and potential changes in the way that a market or the broader economy functions.\(^3\) To achieve wide-spread diffusion, the movement often benefits from sustaining ideologies that strengthen the allegiance of followers and justify the time, resources, and commitment that are necessary for the movement’s success.

Beyond hopes for material gain, America’s expanding programming culture can also be viewed as a manifestation of the deep and abiding cultural tendency that many describe as “technological enthusiasm.”\(^4\) Technological enthusiasm is an upbeat, optimistic appraisal of new technical systems that not only stoke the engines of capitalism, but provide access to the ideals embedded in what is known as the American Project and the American Dream. The publishers of PC software systems readily participated in this vision, as each wave of entrepreneur–engineer strived to improve their software, best their rivals, and boost the productivity of corporations and the general public. By the 1980s, software creation had taken the form of a consensus ideology that united many Americans in a common, accessible dream of a better future through computing. As I will discuss in Chapter 2, this enthusiasm brought additional computing mythologies to the fore, and their collective use contributed to the positive view that American’s held about PCs and software in the years to come.

---


\(^3\) For a discussion of the phases that take place when a new consumer technology is introduced, see Joseph J. Corn, User Unfriendly: Consumer Struggles with Personal Technologies, from Clocks and Sewing Machines to Cars and Computers (Baltimore, MD: Johns Hopkins University Press, 2011). Also useful is Claude S. Fischer, America Calling: A Social History of the Telephone to 1940 (Berkeley: University of California Press, 1994); and the essay collection Does Technology Drive History? The Dilemma of Technological Determinism, eds. Merritt Roe Smith and Leo Marx (Cambridge, MA: The MIT Press, 1994).

1.2 Learning a Language

By the late 1960s, programming emerged from America’s research labs and government institutions to have a direct influence on universities, primary and secondary schools (K-12 in the U.S.), and the nation’s businesses. But what type of mental activity did programming entail? How should students take their first steps when learning to program a computer? In search of an analogy, some specialists suggested that learning to program was a bit like learning to read or speak in a foreign language. Utilizing the comparison, some educators pressed for the inclusion of computer languages in their school’s curriculum. Rather than taking a year or two of a spoken language, such as Spanish or German, a few innovative programs offered courses in computer language instruction, including FORTRAN, Logo, BASIC, and Pascal.

School administrators eager to provide practical job training (and to mollify prospective students and their parents) broadened the definition of “foreign language” to include instruction in computer languages, algorithms, and database management. The popular press advocated for coding instruction in newspapers and special reports, and computer book and magazine publishers released hundreds of titles to help students build simple applications for time-sharing systems and the first PCs.

No one argued that computer languages were the same as human languages, of course. But programming advocates pointed to the many parallels that they observed in the structure of spoken and computer grammars, and to the ways that basic logic and reasoning were gradually introduced to students. Instruction in programming seemed to permit access to the private world of a computer and its “brain” or central processing unit (CPU). Programming was also portrayed as a valuable exercise in logical thinking and problem solving. It was a mental activity that provided a conceptual introduction to how computers worked, as well as a deep dive into logic and syntax. For all these reasons, computer literacy advocates recommended that those who planned to use computers in the future should learn to code as soon as possible. “Even if you don’t write programs yourself,” Ted Nelson advised in 1974, “you may have to work with people who do.”

In the early years of the electronic computer revolution, it was the imposing image of the new machines that seemed to fascinate the public. The physicality of mainframe computers was reinforced by images of large devices whirring and blinking in popular films such as *Desk Set* (1957), *2001: A Space Odyssey* (1968), *Colossus: The Forbin Project* (1970), *Logan’s Run* (1976), and *War Games* (1983). As computers became more reliable and better understood, however, the focus of

---

popular attention turned away from computing machinery to software, the programs that ran on computers, and the coding experts who wrote code in high-level languages like FORTRAN, COBOL, BASIC, and C. The computer industry went through many transitions in the 1960s and 1970s, adding minicomputers and other special-purpose machines. Gradually, the attention of the computing community shifted from scientific and military systems to the application software that powered new types of businesses and helped them manage information.

By the 1980s and 1990s, it became apparent that there were not enough qualified programmers to design, build, and maintain the software systems needed in the U.S. as the country expanded its computational interests into new areas. Although the academic discipline of computer science had taken shape in America’s colleges and universities, these programs could not graduate enough scientists and engineers to satisfy the industry’s needs. The situation was much the same in the rest of the computerized world, in schools and markets stretching from Europe to Asia. Journalist Clive Thompson has written about it this way: “If you look at the history of the world, there are points in time when different professions become suddenly crucial, and their practitioners suddenly powerful. The world abruptly needs and rewards their particular set of skills.” Computer programmers suddenly became this influential group.

The “big bang” of software construction that took place in the 1970s created waves of demand for qualified programmers that continue to expand up to the present. Even in the Internet age—when learning to manage websites, write blogs, and use social media tools has taken on great importance—learning to code has not lost its appeal. As this book goes to press, the leaders of technology companies such as Amazon, Google, Facebook, Apple, and Microsoft regularly complain to Congress that the U.S. does not have enough qualified software developers to meet its needs. According to these advocates, a special exemption is needed in our national immigration policies to allow more foreign high-tech workers into the U.S. to satisfy the demand for software developers and associated fields, such as hardware engineering, artificial intelligence, data mining, computer security, user interface design, audio engineering, cloud computing, product testing and verification, technical writing, product support, project management, and related fields. Programmers have become the lifeblood of our technical society.

1.3 New Ways of Thinking

Calls to learn coding techniques abound now from the leaders of our digital economy. So, too, are warnings that if a group does not heed the call, they will miss

---

out on all or part of what the global digital economy has to offer. But where did this urgency to learn programming come from? What has motivated schools and non-profit organizations to devote so many resources to preparing instructions for a computer? When did programming literacy emerge as a national priority? And what were the early experiences of programmers as they tinkered with mainframes, minicomputers, and the first microcomputers? How is this story connected to the development of successful platforms such as CP/M, MS-DOS, Microsoft Windows, the Apple Macintosh, and Unix-based systems?

Whether past or present, the invitation to become a software maker is an invitation to join a distinctive community within our global society and economy. This computing subculture was founded by a small group of research scientists and academics, but it has expanded into a diverse assortment of hobbyists, students, gamers, artists, musicians, hackers, engineers, career professionals, and part-time workers. Although each of these groups is distinct in socio-economic terms, there is discernable common ground in their understanding of computers and computing technology. Computer programmers share a basic orientation to the world that is shared, despite the differences that they experience in relation to hardware and software systems, learning tools, and historical context.

As a thought experiment, imagine that each subgroup within the programming collective can be conceived of as a concentric circle. In such a model of our programming culture, the entire assortment of circles would be drawn in close proximity to one another, and most of the circles would have points of intersection and overlap. A shared exposure to computational thinking is the overlap, even if the programming languages that people learn (and the tools they write programs with) change over time. Some computer programmers may take up professional work, and others will remain as hobbyists or late-night hackers. Still others may learn programming skills as part of a journey that leads to other types of fruitful work. Despite the differences, and there will be many, the entire set of circles is a model of our nation’s programming culture.7

---

Chapter 1  How Important is Programming?

The call to join ranks with computer programmers is not just an invitation to new ways of thinking (learning computational logic) and new consumer behaviors (buying software and a programming primer), it is also a call to new social relationships, to new ways of seeing and knowing, and to participating in new personal and professional networks. The programming circles that collectively shape America's technical identity are as much expressions of a distinct subculture as are the ideas and values of Impressionist artists and their admirers in Fin-di-siècle Paris or jazz musicians and their fans during the Swing Era in New York City.

As a social historian with interests in the history of technology, business, and education, I am curious about the experiences of today’s programmers and software creators, and where they received their training, inspiration, and cultural worldviews. (See Figure 1.2.) Although the Internet era has contributed much to the behaviors and identity of contemporary software developers, the core skills and thought patterns of modern programmers were influenced by even earlier commitments and achievements. These included the proliferation of high-level languages in the 1950s, the introduction of software engineering techniques in the 1960s, the idealism of educators, entrepreneurs, and authors in the 1970s and 1980s, and the diffusion of commercial programming techniques in the 1990s and 2000s.

My argument is that the learn-to-program movement gained momentum through each of these important transitions, as programmers, authors, and entrepreneurs created pathways through which Americans might learn programming skills and the fine-points of creating software for specific platforms. Computer book authors, magazine publishers, and influential programmer/educators played important, if overlooked, roles in the diffusion of these new skills. By establishing an ideological connection to the computer literacy movement, programmer/educators established a framework that made computer programming feel important, rewarding, and attached to the rituals of citizenship and corporate belonging. The learn-to-program movement took shape through the efforts of many unsung heroes, both women and men, and one of my goals with this book is to reacquaint historians and programmers with a cast of interesting actors and protagonists who have been left out of recent narratives. Part of this work involves using visual sources to

1.3 New Ways of Thinking

A middle school student learns computational thinking in a programming camp sponsored by the Tacoma/South Puget Sound MESA organization. (Photo: Joshua Wiersma/Pacific Lutheran University)

I will profile social reformers, writers, teachers, tinkerers, entrepreneurs, and hackers, as well as computer scientists, students, engineers, and the leaders of America’s computing societies, including the Association for Computer Machinery (ACM). Predictably, most of the programmers that we meet will be members of more than one social or professional group.

To get a sense for the magnitude of the sea change that took place, consider some basic demographics. In 1957, there were approximately 15,000 computer programmers employed in the U.S., a figure that accounts for approximately 80% of the world’s programmers active that year. The work of the first computing pioneers involved building and maintaining military systems, designing algorithms for scientific research, tracking census data, and implementing data-processing schemes for government bureaus and corporations.

In 2000, there were approximately 9 million professional programmers worldwide, with millions more who had been exposed to coding concepts as part of
Chapter 1  How Important is Programming?

Figure 1.3  Three men and two women gather for a meeting near an IBM 370 Model 138 Computer System in Berkeley, California. IBM's 1976 publicity photo emphasizes the value of teamwork and the extensive documentation that was prepared for programmers and administrators. (Courtesy of the Computer History Museum)

their school curriculum or other experiences. In addition to steady growth in military and scientific computing, the expanding digital economy has brought new opportunities for computer programmers in the fields of consumer software, video game programming, artificial intelligence, information publishing, digital communications, education, art, music, entertainment, medicine, and other areas that benefit from the use of computers.

The rising tide of opportunity for software developers has continued up to the present. In 2014, there were approximately 18.5 million software developers in the world, of which 11 million can be considered professional programmers and

1.4 Equity and Access

7.5 million can be considered as hobbyists. Many programmers create or maintain software as part of their regular employment, while others write code for non-profit organizations that they support, and still others program at school, for recreation, or as an aspect of their personal or professional development.

Equity and Access

Despite the bright economic outlook for software developers, there are still numerous challenges in bringing programming proficiencies to the general population. In reality, only a small subset of the people who use computers actually go on to learn something about computational thinking or software development. Our modern economy requires many important job skills and personal investments. Considering the costs and the effort required, does it really matter who learns to program and who does not?

In the book *Stuck in the Shallow End: Education, Race, and Computing*, Jane Margolis et al. argue the “who” that learns to use technology matters a great deal, and that America has suffered throughout its history from inequities in access to computing. Their research indicates that African-American and Latino children are much less likely to receive technology training in American schools than white or Asian children. When scholars analyze gender disparities and later professional outcomes, they find that only two out of ten information technology (IT) professionals are women in the current U.S. workforce.

Margolis and her contributors offer convincing evidence that the characteristics of programming culture matter tremendously to those who enter the subculture and to those who thrive in it (or recede from view). Understanding the long history of the learn-to-program movement and its cultural commitments and values reveals much about how people have interacted with computers in the past, and how we might expand computing opportunities in the future. Yasmin Kafai and Quinn Burke describe the challenge before us as working to better support “computational participation” in our schools and professional environments. In their important book, *Connected Code*, they recommend that thought leaders

---


shape technology-centered cultures carefully, ensuring that all participants feel welcomed and included.\footnote{Yasmin Kafai and Quinn Burke, \textit{Connected Code: Why Children Need to Learn Programming} (Cambridge, MA: The MIT Press, 2016).} (See Figure 1.4.)

One important outgrowth of this research has been the rise of not-for-profit organizations that teach young people how to program, including Code.org, Black Girls Code, Girls Who Code, Native Girls Code, and The Hidden Genius Project. At the high school level, many organizations focus on introducing programming concepts and preparing students to take the College Board’s AP Computer Science Principles examination. I evaluate the work of Code.org and the Hour of Code movement in the \textit{Afterword} for this book. As a preview, I note here that Code.org has completed over 720 million introductory programming sessions since the organization began in 2013, with 46% female and 48% underrepresented minorities currently...
1.5 Personal Connections

I have wanted to write this book for a long time because I am fascinated with software development. PCs were an important starting place for me during my teenage years, and I first learned to write computer code on early microcomputers and PCs. Like many people of my age and social context, my first experiments with home electronics took place in the family rec room during the late 1970s. My extended family bought a Tandy TRS-80 and an Atari video computer system, and the young people in our circles used them to play video games like Pong and Missile Command. A bit later, I experimented with an early IBM Personal Computer when it was released in August 1981, just weeks before I entered college at Pacific Lutheran University (PLU) in Tacoma, Washington. I took an introductory computer programming course and declared as a Computer Science major at PLU, deferring my interests in history and education for graduate school. I learned BASIC, Pascal, C, and assembly language programming on the university’s Digital Equipment Corporation (DEC) VAX 11-780 and DEC PDP-11 minicomputers. I also studied mathematics, data structures, algorithms, operating systems, digital logic, computer architecture, computer graphics, and networking theory. In 1985, I

---


graduated from university and I was hired at Microsoft Corporation to work in one of their two Bellevue (Washington) office buildings, just before the company moved to its better-known Redmond campus. I was employee #850 in the rapidly expanding organization (see Figure 1.5), arriving when the best-selling products were MS-DOS, Microsoft Word for MS-DOS, and a few popular programming languages and development tools.

During my job interview at Microsoft, I was shown a testing (beta) version of Microsoft Windows 1.0. It was not very impressive at the time, but the new graphical operating environment for IBM PCs and compatibles would eventually become an exciting platform for many users, programmers, and commercial software publishers. My first work was at Microsoft Press, the book publishing division of Microsoft, founded by Bill Gates in 1983 to provide technical support for computer enthusiasts who were frustrated by the poor quality of software manuals. In the early days of personal computing, product documentation was often little more than printouts assembled in a three-ring binder, and there was not much in the way of computer-based help or training for PC users. From these humble beginnings, a
major publishing industry took shape. It came to include bestselling magazines like PC Magazine, Macworld, and Compute!, as well as the computer book publishers Howard W. Sams, O’Reilly, Osborne McGraw-Hill, Que, Microsoft Press, Sybex, and IDG Books.

Our work at Microsoft Press was to help self-taught programmers and those who used Microsoft’s business applications to get the most out of their software. I edited books, worked with independent authors, attended industry trade shows, and (beginning in 1986) started writing do-it-yourself (DIY) computer books about using operating systems and programming languages. I was lucky that my university training required a healthy dose of the liberal arts along with my computing classes. Both fields of study prepared me to tackle substantial research and writing projects in the years to come, and they were valued in the book publishing division.

The learn-to-program movement was something that I saw first-hand while working with Microsoft’s customers and authors. In particular, there were fascinating people to learn from at computer industry trade shows, especially COMDEX and Macworld Expo. (See Figure 1.6.) In 1989, I co-authored the book Learn BASIC Now with my colleague and friend, David Rygmyr, and the book was carefully edited by Megan Sheppard and Dale Magee, Jr. (also employees of Microsoft Press). Our programming courseware included a full-featured version of the Microsoft QuickBASIC Interpreter for MS-DOS on three 5.25” disks, and Bill Gates wrote a Foreword to the book recalling his personal connection to Altair BASIC and his interest in using BASIC as a unifying language across computing platforms. (See Chapter 5.)

Learn BASIC Now sold many copies and it was selected as a finalist for a national book award in the computer book “How To” category. Our self-study guide clearly intersected with the powerful demand for programming instruction, and the low-cost QuickBASIC Interpreter made the product relatively inexpensive for newcomers. Over the years, I wrote another 15 books about software development, mostly for self-taught programmers and those who wanted to learn the newest features of popular products like Microsoft Visual Basic or Microsoft Visual Studio. Through the books, I was actively connected to publishers, software development teams, user groups, academics, journalists, literary agents, and a wide range of computer users—many of whom would write or email us directly for help.

1.6 Manifestos of the Movement

Despite my positive interactions with new programmers, I gradually learned that I was only a small part of the third or fourth wave of technical writers who had spread the message about computational literacy and learning to code in the years since the
introduction of the first computers. *Preparation of Programs for an Electronic Digital Computer* was published in 1951 by Maurice Wilkes, David Wheeler, and Stanley Gill to instruct readers on how to formulate machine code for the revolutionary EDSAC computer at the University of Cambridge.\(^{14}\) Grace Mitchell, Daniel McCracken, and Elliott Organick also wrote creative programming primers for FORTRAN in the late 1950s and early 1960s, introducing non-specialists to programming.

In the era of time-sharing systems and early PCs, a new wave of programming advocates supported the movement. These were pioneers like Robert Albrecht and LeRoy Finkel, who participated in the People’s Computer Company and the Homebrew Computer Club in Menlo Park, California. From the beginning, these visionaries understood that not only did people need to *buy computers* and start programming, but they needed to *learn how to program* through books, materials, and social interaction. These computing innovators wrote fascinating programs and produced several best-selling computer titles, but they have largely been neglected in the history of computing. A new book by Joy Lisi Rankin, *A People’s History of Computing in the United States*, is an important exception to this lacuna, and Rankin demonstrates how Albrecht and his contemporaries inspired thousands of programmers to appreciate the benefits of BASIC.\(^{15}\)


Also important in the 1960s and 1970s were the pioneering efforts of the educational theorists Arthur Luehrmann, Seymour Papert, Cynthia Solomon, and Wally Feurzeig, all active in the computing hotbeds of Cambridge, Massachusetts and Greater Boston. Luehrmann coined the term “computer literacy” and encouraged students to learn structured programming with BASIC and Pascal. Papert, Solomon, and Feurzeig co-developed the Logo programming system at the Massachusetts Institute of Technology (MIT), and they wrote about its potential to teach computational thinking to children. Also, from the era of time-sharing systems, David Ahl, an early DEC employee, published tutorials that advocated for the use of computer games to teach programming concepts. My favorite of Ahl's titles is *101 BASIC Computer Games*, published by DEC in 1973. This book is filled with mimeographed program listings that Ahl received in the mail from BASIC users across the U.S. It was one of the first bestselling computer programming titles, selling tens of thousands of copies to novice computer users, hobbyists, academics, and working professionals.

Many of the earliest manifestos of the learn-to-program movement were sold out of VW vans and dusty boxes in computer clubs. However, this DIY world was also on the fringes of the professional software development community, which took its energy from debates within the nascent software engineering movement and the emerging discipline of computer science. The standard-bearers in this field created the computers, operating systems, and programming languages that would fuel the academic and commercial worlds of software development in the years to come. Readers learned about their important discoveries through conferences and influential computer books such as Donald Knuth, *The Art of Computer Programming* (1968 and later); Kathleen Jensen and Niklaus Wirth, *The Pascal User Manual and Report* (1971); Brian Kernighan and Dennis Ritchie, *The C Programming Language* (1978); and Rodnay Zaks, *Programming the Z80* (1979). Although these authors did not always publish programming primers, they helped experienced programmers understand the cadence of computer languages, taught people to devise data structures and algorithms, and explored the advanced features of operating systems and computer architecture. The introduction of professional and commercial programming practices is a crucial stage of the learn-to-program movement.
as the negative consequences that came to people who were denied the opportunity to code based on their location, gender, ethnicity, or economic circumstances. My emphasis is not on high-tech leadership strategies or the tactics that generated corporate wealth, but on the stories of lesser-known programmers, authors, academics, and entrepreneurs. Some were successful, and some have been mostly forgotten. But this is itself a lesson in the history of innovation, business, and technology.

To tell this tale, *Code Nation* presents a new history of personal computing in the U.S. I present a detailed analysis of early computer platforms, a discussion of important compilers and development tools, a “behind-the-scenes” look at application and operating-system programming, the origins of corporate and “enterprise” computing strategies, the rise of user’s guides and computer books, and early attempts to market and sell PC software. Writing a fresh history of personal computing involves significant challenges, in part because the most recent storytelling emphasizes the roles that famous “pioneers” and “founders” have played in narratives about Silicon Valley, the Greater Boston area, and the Pacific Northwest. There has been no shortage of popular books about Apple Computer, Microsoft, Amazon, Google, and Facebook—usually emphasizing the rise of the stereotypical “computer nerds” to positions of wealth and influence in the companies that benefited from personal computing and Internet-based technologies.\(^\text{16}\)

It is often difficult to move beyond these perspectives because of a curious lack of sources that document early personal computing and its broader impact on American society. Most of the earliest PC hardware and software companies have merged or gone out of business, leaving little in the way of historical materials to study. IBM is a noteworthy exception to this trend, recently releasing some of its materials to historians of computing.\(^\text{17}\) But Apple Computer’s corporate records have been carefully edited by their legal teams and are only partially available. Microsoft has also been reluctant to open its corporate archives to scholars and the general public. Beyond the personal narratives of former employees and product enthusiasts, how are historians to study the history of personal computing? What sources can we use to understand how corporate identities were shaped, hardware

---


\(^\text{17}\) See James W. Cortada, *IBM: The Rise and Fall and Reinvention of a Global Icon* (Cambridge, MA: The MIT Press, 2019), especially chapter 14. Cortada was well positioned to write this history because he is a former IBM employee as well as a professional historian.
and software products were created, and whether computing initiatives succeeded or failed? Just as important, how did the users of PCs experience new products and come to understand their features? Can we assess how regular people accepted, accommodated, or rejected the plans and proposals of industry elites?

_CODE Nation_proposes a publication-centered way of examining the early history of microcomputing and personal computing, from experiments with time-sharing systems, to the mail-order kits of early enthusiasts, to book and magazine publications for platforms like MS-DOS, the Apple Macintosh, Microsoft Windows, and Unix/Xenix. I evaluate the history of personal computing using hundreds of programming primers, textbooks, manuals, magazines, user’s guides, and trade show catalogs from the early 1950s to the late 1990s. These neglected sources have allowed me to explore the challenges presented by the first PC systems, the content of computer literacy debates, the methodology of early programming primers, the strategies of successful (and unsuccessful) entrepreneurs and corporations, and the way that computing has impacted the daily life of Americans. To support this analysis, I include technical descriptions of hardware and software systems, code snippets from historic programming languages, the biographies of little-known programmers and entrepreneurs, and a product-based assessment of early hardware and software systems. I also present over 80 historic photographs selected from relevant archives, museums, corporations, and private collections.

I have learned that printed materials related to computers and software—once a common feature of many offices, homes, and schools—have been discarded at an alarming rate. When discussing the issue of “disappearing sources” with a local college librarian, I learned that older computer books and magazines are especially vulnerable to being categorized as _ephemera_, or transitory sources of information about outdated methods or technologies. (See Figure 1.7.) With new computer books and periodicals arriving on a monthly basis, and shrinking budgets, how important is it to maintain an historic collection of FORTRAN, BASIC, and C primers? Especially in locations where shelf space is at a premium? My source’s questions are legitimate, of course. But the comment points out how vulnerable technical sources are to abandonment. “Often, they are simply recycled,” my informant conceded.

But, if we cannot study issues like computer literacy in the past, how can we hope to evaluate it in the present?

For the purpose of this study, I was able to find many older computer books and periodicals in private collections, as well as the technical libraries of larger public universities. For example, I have spent many weeks in the engineering library at the University of Washington in Seattle, which has a good collection. I also found many books, newsletters, and software packages in the Computer History Museum.
in Fremont, California. But like the chapbooks and “street literature” of earlier eras, historic computer books and materials can easily be lost if historians are not sensitive to the many treasures that they contain. In particular, they reveal the teaching strategies used to introduce new technical systems, and the opinions and practices of regular people who are learning new technologies. I hope that this publication-centered approach will be of interest to future historians of computing. There are still many fascinating sources that slumber in our nation’s technical collections.

I begin Code Nation with a comparative analysis that examines computing in the 1960s and 1970s, emphasizing the era’s sense of crisis about how software was being created and its multilayered hopes for renewal. My survey presents four overlapping computing mythologies, each representing a different aspect of the period’s professional, cultural, and technical traditions. These narratives introduce early advocates for software engineering practices, countercultural idealists who
promoted widespread access to tools, creative scholars from the emerging discipline of computer science, and the designers of the first personal computers. In the 1980s and 1990s, American programmers drew on many of these motifs, creating a worldview that bundled hopes, anxieties, and dreams about the new platforms.
“Total learning expands when the range of spontaneous learning widens... and both liberty and discipline flower.”

“In recent years, I have talked to a number of top industry researchers and implementors who are reluctant to hire computer science graduates at any level. They prefer to take engineers or mathematicians, even history majors, and teach them programming.”

When it comes to social movements, the groups that strive toward a common goal with a shared sense of purpose are often the most successful. The learn-to-program movement of the 1970s and 1980s fits this pattern, as do many of the recent computer literacy initiatives, including Code.org’s Hour of Code. According to sociologists, the ideological beliefs that ground social movements act as a bulwark for striving organizations, strengthening the commitment of both leaders and members. Ideological beliefs also help adherents imagine a new world order, and they justify the relatively high levels of personal investment and resources that social movements require. Ideologies set the expectations of a movement’s believers, so that adherents can learn what the group is trying to accomplish and how they should propagate their beliefs. When the going gets tough, ideological commitments keep a social movement going.


This chapter explores four powerful ideologies that influenced America’s burgeoning computer industry in the 1960s and 1970s, each influencing the learn-to-program movement in its own way. Although Parts II and III of this book narrate how Americans learned to code in the 1980s and 1990s, it was only through an awareness of earlier successes and failures that the microcomputing and personal computing movements took shape. Like other scholars, I choose to use the term *foundation myths* to describe the ideologies that influenced the computer industry as it emerged from research settings to become a major contributor to the U.S. economy. *Foundation myths are socially-constructed memories that can carry important historical and cultural information.* They act as social markers, transmitting ideas, beliefs, and worldviews to community members and future generations. Foundation myths summarize historical debates and scientific commitments. They often work subtly, employing the language of metaphor or ritual. In more recent times, computer-related myths are used to celebrate heroic founders and to marginalize illicit behavior. You can often spot these myths when subtle descriptors are used in histories and popular accounts, such as “pioneer,” “entrepreneur,” “evangelist,” “guru,” “hacker,” and “cyberpunk.”

Among the many possibilities, I have chosen four myths about computer technology and computer programmers to begin this book. I will draw connections between these ideologies and the learn-to-program movement in the chapters that follow. The first mythology is a belief in an ongoing period of crisis in the computer industry related to the complexity of computer systems and the pitfalls of commercial software development. Strongly held beliefs about this “crisis” emerged in the 1960s among software development communities, and it set the expectation that most large software projects would arrive late, over budget, and in poor shape. The second mythology is that the computer industry works best when it is driven by popular, democratic impulses and shared community values. This counterculture narrative emerged in the 1970s when a group of West Coast technology enthusiasts argued that using and programming computers should be an enlightening, communal experience. These values strongly shaped some segments of the computer industry, including the microcomputer community in California and authors of programming tutorials in the 1970s and 1980s.

The third mythology relates to a belief about the commitments of professors and administrators in the emerging discipline of computer science. In the U.S., many computer professionals came to believe that computer scientists were occupied primarily with theoretical problems related to computational logic, algorithms, and engineering principles, rather than the practical skills needed to implement projects in the computing industry. Although some academics *did* assume an aloof posture in relation to business computing, this stereotype was largely inaccurate, and it had important consequences for how professional programmers were trained...
(or not trained) in the coming decades. Finally, there are several mythologies related to what is often called “the PC Revolution,” a phrase that tries to capture the excitement surrounding the creation of the first stand-alone microcomputers and personal computers (PCs) in the 1970s and 1980s. This term draws attention to vital energies in the American computer industry, but it also tends to lionize the experience of PC users and entrepreneurs over professionals working in other areas of digital computing. After gently nudging aside this rhetoric, Code Nation proceeds by exploring how the microcomputer movement actually did contribute in important ways to the development of programming culture and the commercial software industry in America. We will investigate how this upward trajectory took place in waves or stages—from the time-sharing systems of the 1960s and 1970s, to the bare-bones microcomputers and PCs of the late 1970s and early 1980s, to the powerful graphical user interface (GUI) workstations of the late 1980s and early 1990s, to the corporate and enterprise computing systems of the late 1990s and early 2000s. These stages involved fascinating operating system platforms, including CP/M, Apple DOS, MS-DOS, the Apple Macintosh, Microsoft Windows, Unix/Xenix, OS/2, and Windows NT Server.

By giving powerful computing mythologies their due, we acknowledge the importance of cultural memories in the history of business and technology, including the problems that people encountered in the past, and the aspirations of users and programmers in the future. The learn-to-program movement succeeded in part because it wove together each of these mythologies, creatively transforming past memories into a shared vision of progress and human belonging. The movement’s visionaries, authors, tinkers, and entrepreneurs deserve recognition for their contributions to the gradual computerization of society, a process with major cultural and economic consequences that is still underway.

We’ll begin with a technical problem and a story.

2.1 The North Atlantic Treaty Organization (NATO) Conference on Software Engineering

In October 1968, there was a sense of crisis in the air.

Although this year has been described as one of the most turbulent in the 20th century, the turning point was not related to political or military disruptions, but to a crisis in the nascent field of software engineering. In fact, executives in North America and Europe had been sounding the alarm since the mid-1960s. Now that powerful mainframe computers were transforming the world’s business and engineering systems, the software that drove these machines was taking on an oversized role in public life. Just weeks before Richard Nixon, Hubert Humphrey, and George Wallace faced off in the 1968 American Presidential Election, the world’s engineers were worrying about computers and software.
To list a few of the problems, there was a perpetual shortage of programmers to create software for the new systems. These programs were often massive, stretching to tens of thousands—even millions—of lines of code in computer languages like COBOL, FORTRAN, and ALGOL. The code configured American military systems and corporate data processing tools—programs like the banking, billing, and reservation systems that proliferated in the late 1960s. However, good software developers were hard to find. There was no clear procedure for locating, hiring, and training the specialists needed to build and maintain the required systems.

The growing complexity of software also required robust management techniques to ensure that projects were completed on time and on budget, but neither outcome was very common. To make matters worse, the growth of professional organizations like the Association for Computing Machinery (ACM) found it difficult to improve programmer productivity or software quality. Although revenue was pouring in to successful American hardware manufacturers like IBM, Burroughs, and Digital Equipment Corporation (DEC), the budding software industry seemed undisciplined in its workflows, ill-prepared to expand, and in a perpetual state of disorder.

Much has been written about the “software crisis” of the late 1960s, and some have argued that “software engineering” was the wrong metaphor to address the problems. But the dilemma was noted by many computer professionals around the world, from North America to Asia to Europe. The 1960s was a time of expansion, as organizations were drawn to utopian visions of mainframe and minicomputer technologies. But reassessments soon followed, and critics pointed to bloated software systems that were complex and error prone; programs designed for engineers with pocket protectors but not real people. The job performance of corporate software developers also came under fire. “When a computer programmer is good, he is very, very good,” concluded one IBM study published in 1968. “But when he is bad, he is horrid.”


The gender implications embedded in this statement are subtle, but important to catch. In 1968, approximately 88% of professional programmers in the U.S. were men. Although women made significant contributions to computing in the 1950s, programming work underwent a process of masculinization in Britain and America in the 1960s and 1970s. As part of this transition, the cultural stereotypes about programming being a male activity increased, and the era’s sources often associate programming with masculinity. The implications of the underrepresentation of women in computing will be examined in Chapters 3, 7, 8, and 10, which explore how Americans learned to code, and how new programmers negotiated for status in the communities that either welcomed or rejected them. Keep an eye on this complex issue; it surfaces in predictable but also unlikely settings. (See Figure 2.1.)

To address the global software crisis, the Science Committee of NATO sponsored a conference in October 1968, in the Garmisch-Partenkirchen district of West

---

Germany (Bavaria). The organizers planned to discuss the design and production of computer software, the experiences of software users, and the persistent problem of meeting software schedules and budgets. The term “Software Engineering” was chosen as a framing title for the meetings, hinting at a proposed solution to the dilemma: The computer industry should infuse programming with theory and practice from the disciplines of science and engineering to address their problems. It was high time to demand structured approaches to design, coding practices, and testing that would improve reliability. If this did not happen soon, the complexity of software, and its concomitant unpredictability, would likely stifle the electronic computer revolution.

The 5-day conference was attended by 50 people from 11 countries, with a large contingent representing the U.S. All 50 registered conference participants were men, with a few women serving in support roles. Analyzing the conference materials, Janet Abbate sees the absence of the American Grace Murray Hopper as the most striking in terms of gender exclusion. In 1968, Hopper served as the director of programming languages and standards for the U.S. Navy, and she was an established luminary in the computing industry.6

The proceedings show that the attendees were mostly computing professionals who managed software teams or worked regularly with the users of software systems. There were manufacturers’ representatives in attendance, as well as academics speaking for interested university faculties.7 As Nathan Ensmenger wrote, the meeting marked a major shift in general perceptions about software construction.8 After Garmisch, there was pressure on software developers to eschew craft and artisan approaches to programming and to adopt established engineering principles. The wild and woolly days of coding by trial and error were over. (Or so the organizers hoped.) Although some came to question the value of software engineering as a framing term, the methodology would have an important impact on programming culture in the coming decades.9

6. Abbate, Recoding Gender, 102–103.
Let’s take a look at the problems software developers wrestled with in the 1960s. At that time, the dominant paradigm for software development revolved around a solitary (male) computer genius enigmatically holding court with a small team of assistants. The consensus at Garmisch was that this scenario needed to be replaced with a cohort of systematically-trained engineers, responsible to management, who practiced structural thinking and followed orders. (A visual model of this hierarchy and approach can be seen in Figure 2.2.) As Nathan Ensmenger summarized, “Software engineering promised to bring control and predictability to the traditionally undisciplined practices of software development.”

Historian Stuart Shapiro analyzed the legacy of this “engineering movement” as it gained momentum in the 1970s and 1980s. According to Shapiro, the program was less about specific technical procedures and more about finding ways to regulate and standardize growing project complexity, budgets, and the new cadre of software engineers who had until recently attracted little notice. New approaches to programming took center stage in the push to make software development outcomes more reliable. These strategies included the use of structured programming techniques, adding language features to promote reliability, measuring software performance (metrics), and using integrated development environments (IDEs).

Most of these ideas would make their way into personal computing, too, although it would take a decade or more for the engineering practices to take hold.

Computer programming is sometimes envisioned as an individual task, but by the late 1960s, commercial software was typically constructed in groups. For example, in mainframe computing environments like the IBM System/360, it was necessary to hire an army of analysts, technicians, and software developers to build and maintain non-trivial systems. Productivity gains associated with the “division of labor” principle simply did not work when dividing up the tasks of a large software project. The new approach had to involve teaming. Grace Hopper subtly introduced this concept when she created the world’s first computer language compiler in 1952, known as the A-0 system, which translated symbolic codes into machine language. Hopper completed the first draft of her work and then immediately shared it with associates to see if they could make improvements, a strategy that she also followed during the creation of FLOW-MATIC, the predecessor to COBOL.

By the 1960s, many engineering groups were using structure and clearly defined roles to bring predictability to their projects. The team that created the DEC PDP-6 computer was led by C. Gordon Bell, the man wearing a suit jacket in this 1964 photo.


Sitting L–R: Lydia Lowe (McKalip)—Secretary to C. Gordon Bell, Bill Colburn—PDP-6 Project Engineer, Ken Senior—Field Service Technician, Ken Fitzgerald—Mechanical Engineer, Norman Hurst—Programmer, Harris Hyman, Operating Systems Programmer. (Courtesy the Computer History Museum and DEC)

The Complexity of Software

An underlying current of the 1968 NATO conference was that building software entailed a level of complexity that few fully recognized when digital electronic computers hit the scene in the 1950s.
But what made computer software so complex?

Let us start with a formal definition. Computer software is one part of a computer system that consists of data and computer instructions. Computer software is distinct from computer hardware, or the physical components of a computing system, such as the processors, circuits, boards, panels, monitors, switches, and other electronic devices in a machine.

A basic understanding of what software consists of changes over time, so it is useful to visualize a list of items that has taken shape in evolving contexts. Modern software includes a wealth of program types (operating system components, device drivers, application software, games, programming tools, malware), as well as supporting libraries, data, images, sound recordings, videos, email messages, Facebook posts, Tweets, and all manner of digital media. A software release typically consists of a bundled collection of items with a particular purpose, including a setup program, executable files, and hundreds (or thousands) of installed components, digital media files, documentation, and other resources.

From a business point of view, software may be considered a commercial product with economic value and utility, such as the popular applications GarageBand for iOS, Adobe Photoshop, or Microsoft Office. Software may also be distributed for free, such as open-source software or freeware. In these many contexts, one piece of software is distinct from another on a computer system, in the marketplace, or (under certain conditions) in copyright law. As recent historians have also discovered, each piece of software has its own history and impact, its creators and users. Software carries cultural memories and a society’s hopes and fears.

In the 1960s, most software programs were supplied for free with the expensive computer systems that organizations purchased or rented from hardware manufacturers. In the U.S., IBM was the leading computer manufacturer by a large margin, followed by successful electronics firms like Burroughs, UNIVAC, NCR, Control Data, Honeywell, General Electric, and RCA. Corporations used mainframe computers for complex calculations, resource planning, bulk data processing, and transaction processing, including managing shipping, payroll, and employee records. In these many contexts, organizations needed to adapt the free software that they were given to match the needs of their customers. They needed to hire and train programmers and technicians to accomplish this work.

As computers grew and took on more tasks, the ailments plaguing software could often be traced back to one principle cause—system complexity. The complexity of software was engendered by programming’s abstract nature and the scientific principle that a program constitutes a digital (discrete state) system based on
mathematical logic rather than an analog system based on continuous functions.\textsuperscript{12} As software systems were being constructed with growing sophistication, project designers needed to consider numerous interrelated factors in their solutions, including the organization’s list of requirements for a system (clearly stated or not), operating constraints related to hardware and software platforms, technical conditions within the computer itself (including memory resources), and the wide range of possible inputs and outputs that a program might encounter as it completed its work.

Real-world computer systems were designed so that they used only a prescribed set of resources, such as memory and processor time. From the point of view of the programmer, additional complexity arrived in the selection of programming languages, data structures, algorithms, flow control mechanisms, error handling structures, and the use of inherited source files and legacy code from other projects.

Individual computer programmers also brought their own tastes and psychological experiences to a project, as well as diverse training and educational experiences. All of these variables made the precise functionality of programs hard to predict, in the same way that storms and atmospheric conditions are challenging to forecast. The intricate balancing act was magnified in myriad ways as the responsibility for building new systems was distributed among team members who had different abilities and often coded in different locations and contexts.

In the late 1960s, anxious managers noted that the complexity of large systems created engineering problems with no easy solutions or mechanisms for assessment and control. As E.E. David of Bell Laboratories pointed out at the Garmisch conference,

> Production of large software has become a \textit{scare item} for management. By reputation it is often an unprofitable morass, \textit{costly and unending}. This reputation is perhaps deserved. No less a person than T. J. Watson [Jr., Chairman and CEO of IBM] said that OS/360 cost IBM over 50 million dollars a year during its preparation, and at least 5000 man-years’ investment.\textsuperscript{13}

In David’s telling example, the software development project for OS/360 (IBM’s operating system for the 360 series of computers) was famously late and over budget, problems blamed on poor management practices and unwieldy development teams. The project became the subject of Frederick Brooks’s well-known guidebook on managing software projects, \textit{The Mythical Man-Month} (1975), which we

\textsuperscript{12} Shapiro, “Software Engineering,” 20.

\textsuperscript{13} Naur and Randell, \textit{Software Engineering Report}, 13. The italic formatting is mine.
2.3 Systems are for Customers

Who were these early software systems designed for, and what percentage of the population did they actually represent? Playing back conversations from the 1960s, it is sometimes hard to tell. Here is one fascinating transcript from the 1968 conference that we have been using as a touchstone. It involves six prominent leaders from the global computer industry, including two computer manufacturers, three academics, and one of the conference’s organizers.¹⁴ (See Figure 2.3.) The exchange focused on the users of computer software and the extent to which customers

---

¹⁴. For the original transcript, see Naur and Randell, *Software Engineering Report*, 24–25. The speakers were Professor J. N. P. Hume (University of Toronto), J. D. Babcock (Allen-Babcock Computing, New York, NY), Professor J. Berghuis (Philips’ Computer Industrie [a Dutch computer manufacturer], Netherlands), J. W. Smith (Scientific Data Systems, El Segundo, CA), Dr. M. Paul (Leibniz-Rechenzentrum [a computing research center], Munich), and Professor A. J. Perlis (Carnegie Mellon, Pittsburgh, PA).
should be included in the design of new software — an emphasis which appears to be lacking in the first systems. A cautious professor J. N. P. Hume begins the dialog. (The italic is mine.)

J. N. P. Hume [University of Toronto]: One must be careful to avoid over-reacting to individual users. It is impossible to keep all of the users happy. You must identify and then concentrate on the requirements common to a majority of users, even if this means driving a few users with special requirements away. Particularly in a university environment you take certain liberties with people’s freedom in order to have the majority happy.

J. D. Babcock [Allen-Babcock Computing, New York, NY]: In our experience the users are very brilliant people, especially if they are your customers and depend on you for their livelihood. We find that every design phase we go through we base strictly on the users’ reactions to the previous system. The users are the people who do our design, once we get started.

J. Berghuis [Dutch professor and industry consultant]: Users are interested in systems requirements and buy systems in that way. But that implies that they are able to say what they want. Most of the users aren’t able to. One of the greatest difficulties will be out of our field as soon as the users realize what kind of problems they have.

J. W. Smith [Scientific Data Systems, El Segundo, CA]: Many of the people who design software refer to users as ‘they’, ‘them’. They are some odd breed of cats living there in the outer world, knowing nothing, to whom nothing is owed. Most of the designers of manufacturers’ software are designing, I think, for their own benefit — they are literally playing games. They have no conception of validating their design before sending it out, or even evaluating the design in the light of potential use. The real problem is training the people to do the design. Most designers of software are damn well incompetent, one way or another.

M. Paul [Leibniz-Rechenzentrum, Munich]: The customer often does not know what he needs, and is sometimes cut off from knowing what is or what might be available.

Alan Perlis [Conference organizer, Carnegie Mellon University, Pittsburgh, PA]: Almost all users require much less from a large operating system than is provided.

Imagine a heated conversation among influential professors and experts who think that they each know best, and you’ve got the gist of this exchange. But notice how divided the industry leaders are about computer users and the design of software.
in the 1960s. Each person is wondering: who are these systems really designed for? How much functionality should be provided to users?

Professor Hume outlined a realist position: You cannot make all of the software users happy, but if you try to satisfy a majority of them, you will probably find a good balance. Hume believed that it was legitimate to take away the freedoms of users on occasion to protect the system or satisfy the needs of the majority. We are still wrestling with the implications of this statement for security and civil liberties in electronic environments.

Bristling at the regimenting tone of this statement, James Babcock of Allen-Babcock Computing rushed to the defense of software users, whom he styled as “brilliant.”

Allen-Babcock Computing was founded in 1964 in Los Angeles by James Babcock and Michael Jane Allen Babcock to profit from the rapidly expanding market for computer time-sharing services. Between 1965 and 1966, the company participated in the development of Conversational Programming System (CPS), a time-sharing product that ran under IBM’s popular OS/360. In 1969, Allen-Babcock held a 3% share of the time-sharing services market—a lucrative position in this developing field.¹⁵ Babcock's direct experience with “external” computer users like this is unusual but important to catch. He had regular contact with paying customers in a competitive marketplace outside of academic or corporate contexts, where “control” over the system typically trumped the preferences of users. In Babcock’s opinion, if computer users do not directly enjoy the benefits of a new system, it is of little value.

But can computer users really be trusted? Professor Berghuis replied to Babcock that in his opinion customers had the right to make their selections, but most users do not know how to do it. If they only knew what computers were capable of, Berghuis mused, then software construction would be so much easier. Here, Berghuis puts the burden on users to know what they need, rather than on software designers to assess a customer’s requirements.

J. W. Smith of Scientific Data Systems (SDS), El Segunda, California, supported James Babcock’s position and he also spoke as an advocate for users.

SDS was a computer company established in 1961 by Max Palevsky and Robert Beck, veterans of the engineering firms Packard Bell and Bendix. This firm was an early adopter of integrated circuit technology and also a maker of time-sharing systems. (They sold their successful operation to Xerox in 1969.) Accordingly,

Mr. Smith also knew something about customers, and he quickly pointed out that most users were treated *disdainfully* by system designers, who seemingly devised software systems for themselves. Their overdesigned programs felt to him like “games”—i.e., eccentric, inward-facing diversions, representing the tricks of show-offs rather than any real attempt to satisfy the needs of users.

Our final comment comes from Professor Alan Perlis (1922–1990), who offered a new point—that software systems often contain *too much* functionality. This was another voice of concern about system complexity, which Perlis knew would translate into rising costs and overdesigned systems. Perlis would certainly have known. He was a savvy computer scientist and administrator who organized one of the first academic computer science programs in the U.S. He also believed that computer programming should be taught widely in schools, and we will see later what contributions he made to the learn-to-program movement. For starters, Perlis designed the Internal Translator (IT) programming language and co-designed Algorithmic Language (ALGOL) with a committee of 13 computer scientists. He advocated for the rights of both programmers and users throughout his distinguished career.

Here is the point. By the late 1960s, computers and software systems had radically changed—and so had users and customers. During this era, software was becoming highly complex, and it was rapidly incorporating support for advanced features such as multitasking and time-sharing. Software was also becoming “unbundled” from hardware sales, and as the process moved forward commercial programs took on the attributes of modern consumer products. These features included improved design concepts, compelling lists of features, market-based pricing, product reliability, and customer support from the software makers and third parties.

The decoupling of software services began in the mid-1960s and gained momentum when IBM announced in June 1969, that it would begin pricing software separately from hardware during the coming year. In less than a decade, software sales was re-organized into discrete channels, including lines for original equipment manufacturers (OEMs), retail stores, mail order customers, foreign translation markets, custom licensing products, and more. By the 1980s, boxed and shrink-wrapped software packages with attractive designs became the norm for the emerging PC industry, and these goods were manufactured in facilities that were governed by the best practices of supply-chain management, warehousing, fulfillment services, and accounting. In short, PC software markets were built on the firm foundation

of mainframe and minicomputer sales and support. Contrary to popular opinion, the corporate software products industry remained very strong in the U.S. for decades.\footnote{Not until 1998, when Microsoft overtook it, did IBM cease to be the world's largest software supplier. See Martin Campbell-Kelly, \textit{From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry} (Cambridge, MA: The MIT Press, 2003), 174.}

The software crisis of the late 1960s created an important mythology about computing in Europe and the U.S., because it called into question the reliability of software systems and the development processes connected to them. Worries about complexity would remain in the software industry for decades, resurfacing again in the era of GUIs and enterprise computing in the 1980s and 1990s. To address the issue, software managers introduced engineering principles and encouraged their employees to work in teams that were efficient, on time, and under budget—desirables that became an obsession for later programming contexts. In a subtler way, the software crisis also elevated a new group in the history of computing—\textit{users}, who through market influence and creative action would change how software products were designed and used. Over time, these customers would help to make technology companies among the most valuable corporations on earth.

\subsection*{2.4 The Counterculture Movement}

In the 1950s and 1960s, the centers of mainframe computing research in the U.S. were to be found in the headquarters of IBM in upstate New York and in the academic labs of nearby Cambridge, Massachusetts. By the late 1960s and early 1970s, however, a relatively compact region of California between San Jose and San Francisco became a crucible not only for political protests and a thriving counterculture but also a new set of computing paradigms that would deeply influence the technical world.\footnote{A number of excellent books have explored the myths and mythmaking of Silicon Valley entrepreneurs and software developers, including (by date of publication), Steven Levy, \textit{Hackers: Heroes of the Computer Revolution} (Garden City, NY: Anchor Press/Doubleday, 1984; Revised edition, Sebastopol, CA: O'Reilly, 2010); Theodore Roszak, \textit{From Satori to Silicon Valley: San Francisco and the American Counterculture} (San Francisco, CA: Don't Call It Frisco Press, 1986); John Markoff, \textit{What the Dormouse Said: How the Sixties Counter-culture Shaped the Personal Computer Industry} (New York: Penguin Books, 2005); Fred Turner, \textit{From Counterculture to Cybertulture: Stewart Brand, the Whole Earth Network, and the Rise of Digital Utopianism} (Chicago: University of Chicago Press, 2006); Michael Swaine and Paul Freiberger, \textit{Fire in the Valley: The Birth and Death of the Personal Computer}, Third Edition (Dallas, TX: The Pragmatic Bookshelf, 2014); Walter Isaacson, \textit{The Innovators: How a Group of Hackers, Geniuses, and Geeks Created the Digital Revolution} (New York: Simon & Schuster, 2014); Clive Thompson, \textit{Coders: The Making of a New Tribe and the
A seminal text in the communication of counterculture values was Theodore Roszak’s *The Making of a Counter Culture* (1969), which criticized the dominant industrialized cultures of Europe and America and suggested new ideals for disaffected citizens, students, and intellectuals. Roszak rejected what he called *technocracy* in modern societies, the oppressive regimes of corporate and technological expertise that seemingly dominated society and regimented social and intellectual life. His work echoed themes from other works of the period, including C. Wright Mills’ *The Power Elite* (1956), Leo Marx’s *The Machine in the Garden* (1964), Jacques Ellul’s *The Technological Society* (1964), and Lewis Mumford’s *The Myth of the Machine* (1967). Technology had its merits, these texts argued, but in the era of cold wars, nuclear weapons, and the expanding military-industrial complex, technology could also become a force of dehumanization. To reject this mindset required a transformation of consciousness, a mode of transcendence stimulated by new types of knowledge and collaborative styles of living.

If this social and political protest seems like a rejection of the mainframe computing culture that we have just surveyed, it was—at least in part. Countercultural intellectuals came to view most of the scientists who worked on government and military projects as Big Brother loving bureaucrats who were supporting the wrong team. The fact that many of the employees who worked on these projects also had concerns about the ethics of the military-industrial complex was beside the point, at least for a while.

In his description of the counterculture movement, Fred Turner has identified two groups that envisioned the transformation of consciousness as the essential task for healing American society in the 1960s. The first group withdrew from society and formed egalitarian communes in places like northern California, Colorado, and New Mexico. These communes could be in rural or urban areas, but they were unified in their rejection of middle-class, Cold War America and its presumed values. The second group focused on mind-expanding experiences including sexuality, psychedelic drugs, music, and alternative spiritualities. These countercultural experimenters often remained in society but developed a similar utopian outlook to those who choose to live in the communes.  

---
