Software

Software

A Technical History

Kim W. Tracy
Rose-Hulman Institute of Technology, IN, USA

ACM Books #38

Software: A Technical History
Kim W. Tracy

books.acm.org
http://books.acm.org

ISBN: 978-1-4503-8724-8 hardcover
ISBN: 978-1-4503-8725-5 paperback
ISBN: 978-1-4503-8726-2 EPUB
ISBN: 978-1-4503-8727-9 eBook

Series ISSN: 2374-6769 print 2374-6777 electronic

DOIs:

10.1145/3477339	Book	10.1145/3477339.3477346	Chapter 6
10.1145/3477339.3477340	Preface	10.1145/3477339.3477347	Chapter 7
10.1145/3477339.3477341	Chapter 1	10.1145/3477339.3477348	Chapter 8
10.1145/3477339.3477342	Chapter 2	10.1145/3477339.3477349	Appendix A
10.1145/3477339.3477343	Chapter 3	10.1145/3477339.3477350	Bibliography
10.1145/3477339.3477344	Chapter 4	10.1145/3477339.3477351	Bios/Index
10.1145/3477339.3477345	Chapter 5		

A publication in the ACM Books series, #38
Editors in Chief: Sanjiva Prasad, *Indian Institute of Technology (IIT) Delhi, India*
Marta Kwiatkowksa, *University of Oxford, UK*
Charu Aggarwal, *IBM Corporation, USA*

This book was typeset in Arnhem Pro 10/14 and Flama using pdfTEX.

First Edition

10 9 8 7 6 5 4 3 2 1

To my sons, Robert and Michael

Contents

List of Figures

List of Tables

Preface

Software professionals and students are focused on creating *new* technologies involving software. As a result, many may view software history as not directly relevant to their work or studies. This text makes the argument that current and future software systems are influenced by the software of the past, the lessons learned in its creation, and even the historical software code base. A good example is computer security, where security professionals must include knowledge of any software that may be compromised and that often includes older software and knowledge of older protocols and vulnerabilities that may still be in the core of modern software systems.

Furthermore, legacy software systems are notoriously difficult to replace. As noted in Charette [2020] and as experienced by this author as a chief information officer, legacy systems take considerable effort and money to replace and tend to be built upon, rather than replaced. So, those working on systems for complex organizations are likely to have to deal with these existing software systems. Charette [2020] also cites examples such as the US Social Security Administration still running some 60 million lines of COBOL and 20 million lines of assembly code. The US Internal Revenue Service has original master tax file systems that were installed in the 1960s and have had additional layers of other software as well as many other software systems connected to them. This building of dependencies on legacy software further entrenches its use. Other systems used by the US government have software sub-systems that are about 50 years old such as in the Departments of Education and Transportation [Charette 2020]. Extreme events such as security breaches or system failures to adjust to new requirements are often required in order to get funding to replace and upgrade these systems. So, knowing about older software technologies has a *practical* element as well as an *understanding of lessons from software's history* element.

Software is a relatively recent creation, having taken shape as a distinct concept in the 1950s. Software has rapidly evolved and has developed its own related disciplines including computer science and software engineering, among others. Software continues to evolve rapidly and to become increasingly complex with many programs and many layers of software being commonly used in a single device. While dependent on physical computers to actually run, software domains now evolve largely independently of the computing hardware. Abstractions have been built that allow software's theoretical basis of what is computable and efficient to be analyzed independently of any specific hardware. Technology students today rarely get a complete picture of how software has changed over time. This text attempts to give that overview of how software has evolved in several important domains. Besides that overview, specific examples in these domains are provided to illustrate the issues and the solutions that have been developed for that domain.

A quote from historian Michael Mahoney (see Mahoney [2011, p. 65]), originally published in a 2005 paper "The Histories of Computing(s)" makes the point of the nascent state and importance of software history:

> But we [historians] remain largely ignorant about the origins and development of the dynamic processes running on those devices [computers], the processes that determine what we do with computers and how we think about what we do. The histories of computing will involve many aspects, but primarily they will be histories of software.

It is easy to get wrapped up in a specific story of software history and to get distracted by details, particularly if there is a reason to reminisce about that software. In covering every detail, it is easy to miss the forest for the trees. However, it's those stories that make the history of software interesting and relevant. This book attempts to strike a balance between the high-level "forest" of software and the lower-level "trees" of specific stories and examples. So, the book covers the broad breadth of software and chooses detailed examples. It is only with these detailed examples that software history not only comes alive but that students can develop an understanding of how software technology has evolved and responded to the demands placed on it over time. Even at that, most of these detailed examples cannot include every detail as a complete understanding of a very complex software system might take a lifetime.

In the last couple of decades, software has gotten attention as a distinct topic from computer history. In particular there are wide-scoping works on the software industry (such as CampbellKelly [2003] and Cortada [2012]) and the professionalization of the programmer community (such as Ensmenger [2010]). There's also been work on the history of computer science as a field and how theoretical computer science evolved (such as Mahoney [2011] and Priestley [2011]).

The intended audience for this book is students of technology who have some background in software development. They need not be expert programmers, but to get the most from this book they should be able to read and understand source code for various programming languages and have a basic understanding of how computers work.

The included exercises vary from those that are relatively easy to those that will take an extensive effort. Those taking extensive effort are identified as "Projects" and those that are easier to accomplish are identified as "Exercises."

Use of the Book

This book is intended to be used in an upper-division undergraduate course where students have some background in writing software. It need not be extensive, but students need to be able to read source code and have an ability to understand differing computing architectures. It is structured such that the first two chapters should be covered first. These two chapters introduce the overall issues and ways that the history of software is approached in this book. Chapters 3 to 8 are meant to stand by themselves, with Chapters 3 to 7 covering software topics that are foundational in nature, generally closer to the system level. It is anticipated that later editions or volumes will add chapters related to higher-level software history such as artificial intelligence, graphics, security, enterprise applications, among others. Chapter 8 summarizes the lessons learned from earlier chapters and is intended to solidify the goals of the course.

Exercises and Projects are included in every chapter. The intent is that "Exercises" are relatively straightforward problems that require only a paragraph or so to respond. It quickly became evident that many of the more interesting problems in software history require much more work and investigation and are included as "Projects." Projects may require a great deal of work and some could form the basis for undergraduate research projects, master's projects, or even doctoral theses in software history.

Additional resources are available online at software-history.net.

Acknowledgments

Undoubtedly, this work does not contain every important detail about software or its development. The intent is to cover the most important details for students of software technology. Certainly, entire books could be written on each of the chapters included here or even on single topics, and some have been written. I know that others will find details missing which they believe important and I welcome any feedback to that effect. While I've sought out information and advice

from many on the details included here, any errors are purely my own. For helping me improve the quality of the book, my thanks go to the editorial staff at ACM Books and Compuscript Ltd including but not limited to Bernadette Shade, Ashley Petrylak, Scott Delman, Barbara Ryan, Achi Dosanjh, and Karen Grace.

I would like to especially thank many of the people consulted for this work and the help provided by several archives. In particular, the Smithsonian's National Museum of American History, the Computer History Museum archives, the Charles Babbage Institute, the archives at the Massachusetts Institute of Technology, and the archives at Stanford University have provided a great deal of helpful archival sources. The Linda Hall Library and Hagley Library also held helpful resources. Individuals including Burton Grad, Peggy Kidwell, Paul McJones, Guy Fedorkow, Tom Misa, R. Arvid Nelsen, Greg Singleton, Jeff Ullman, John Hennessy, and Donald Knuth have also had helpful insights. Many others have reviewed various drafts including Babu Ranganathan, Peter Capek, Joy Stockwell, and numerous students.

I would also like to thank my many students at the Rose-Hulman Institute of Technology (RHIT) and at Michigan Technological University who have provided helpful insights, revisions, and interest in the topic. Several former RHIT students produced content for the software history website, including Brock Grinstead, Austin Gassert, and Leela Pakaneta. ACM's Special Interest Group for Computer Science Education (SIGCSE) is appreciated for providing a grant to support work on the software history website where additional materials are being housed, including source code and supporting documentation for several examples.

A number of institutions have allowed me to reuse images including ACM, IBM, AT&T, SRI International, Charles Babbage Institute, The Smithsonian National Museum of American History, NASA, NIST, US National Archives, US Navy, US Army, Rhode Island Computer Museum, the London School of Economics, Iowa State University, MIT, and UNISYS. These images and figures help students understand the environment and details in which software was built.

Kim W. Tracy
tracy@rose-hulman.edu or tracy@cs.stanford.edu
Terre Haute, Indiana
January 2021

1 Introduction to Software History

Software is a relatively recent technology, really only beginning in its own right in the 1950s. In the time since then, it's taken many forms, evolved immensely in the tools used, and in the purposes to which it has been put. Interestingly, software has become so varied and so complex that many current students of technology do not have a basic knowledge of the history of software. As a result, students and practitioners are sometimes repeating mistakes of the past and often relearning how to build successful systems. Additionally, some areas of software require a working knowledge of previously deployed software systems and their design decisions, such as software security. When reusing existing software, it is wise to evaluate the relevance of the techniques and assumptions that were used in building that original software. This book focuses on software as a technology and how it has evolved over time. We will look at the trends, important innovations, and events, as well as the ever-broadening world of software.

This chapter includes a definition of software and uses techniques from historians of technology on how to structure this history. Additionally, early computer hardware history is covered as a background. In Chapter 2, software is further defined and categorized into types. Key types of software (structured into *domains*) will then be covered as individual chapters, with the last chapter looking at the entire span of software technology and the lessons that can be learned from the history of software.

In this book, we focus on the technical basis of software, its types, and how it changes over time.

1.1 What is "Software"?

Software really didn't exist as a separate concept or term until the late 1950s. When higher-level languages (such as FORTRAN, LISP, and COBOL) were created, the notion of software being separate from the computer became more commonplace.

This book is focused on software as a separate entity, even though there is obviously (especially in the early days of computing) a close relationship to the specific computational device used. Before software was a separate concept from a particular computer, considerable effort was expended on how to use these devices to solve problems and how to program them (see Wilkes et al. [1951], Goldstine and von Neumann [1948], and Campbell-Kelly and Williams [1985]). These earlier efforts significantly impacted how we approach programming and the tools that we use to make those programs. Reusable concepts and designs influenced the design of later software and systems throughout the history of software. A working definition of software is given here as:

> *Software is the set of programs, concepts, tools, and methods used to produce a running system on computing devices.*

This definition deserves more explanation. *Programs* and *applications* are the manifestations that most people encounter as *software.* The first printed use of the word *software* is generally thought to have been by John Tukey in 1958.[1] The source code, libraries, executables, and comments make up a single program, but a program does not exist in isolation and usually includes a number of other programs to solve a problem or to run as a *system.* Additionally, a number of tools (many of which are software as well) such as compilers, assemblers, flowcharts, design tools, simulators, and architectural diagrams are used to create and make the software a running system. The *methods* used to produce software include analysis and design methodologies as well as evolving best practices of creating complex software systems. The *concepts* used to produce a software system are some of the most reusable components. We will see in later chapters that these concepts drive the evolution of software over time. A concept can be reused and built upon independently from the hardware or other project-specific details. As an example, while early database systems defined many database concepts and were implemented in many systems, very little of those early systems are reusable today. Those database concepts, however, have been expanded and have become a large subfield of software. Another example is the use of software patterns, where rather than encoding the details of an implementation the bare essentials are specified so that the pattern can be reused to address similar problems.

So, this rather broad definition of *software* is used throughout this book in order to explore how software has changed over time.

1. See Shapiro [2000] for more discussion on the origin of the term *software*. Tukey is also credited for coining the term *bit*, meaning binary digit.

As an example, the UNIX® operating system[2] is a set of many programs, with some that manage processes, some that manage the file system, and others that manage the system's memory. For a running UNIX system, many of these programs are required for the system to be usable as an operating system. UNIX was also created with the idea that a set of programming tools would be built to better support the UNIX programming environment over time. Those tools began with the DEC PDP-7 assembler, and quickly moved to using the recently developed C programming language and associated compiler. Eventually, UNIX included a broad set of programming tools including C, lex, yacc, sed, vi, awk, grep, and many others. UNIX was initially created by a small, highly collaborative group of developers without a lot of formal methodology. All of these parts are essential to understanding how UNIX was developed, and also to be able to learn from the successes and failures of the system.

Most of the focus of this text will be on concepts and programs, particularly those that are still used, are reusable in modern systems, or are embedded in modern systems.

1.2 Challenges of Software History

Software, as defined previously, is a non-tangible technology. That is, there are no physical artifacts in the same way as computational devices. As a result, it can be less obvious how early software functioned, how ideas were shared and changed, and how such software was created. Additionally, software has become extremely malleable. It changes as the needs change, better ideas are explored, and, unfortunately, rendering software easy to destroy or dispose of.[3] Often software, even though theoretically changeable, becomes so entrenched that there's a fear of "if it isn't broke, don't fix it." This can result in software lasting for decades with the details of such legacy code and how to fix it all but forgotten. The breadth of software is huge and constantly growing by being applied to new application areas. This book focuses on key underlying software systems such as operating systems, database management systems, and programming languages, but there is an extreme diversity of application and specialization that has developed. Many current software practitioners have a narrow perspective of software that focuses only on their specialty.

Current students of technology are generally taught computing history sporadically and even less is taught on software history. So, the challenge is understanding

2. UNIX is a registered trademark of the Open Group.

3. This is becoming somewhat less true as many copies of current software systems are being made as well as many versions being preserved in software change control systems, such as GIT or SVN.

this amorphous mass of constantly evolving and constantly expanding software. There are a number of reasons for current students and practitioners to be aware of software history:

- *Persistence.* Software has become more persistent than particular generations or models of computers. Old software can often run on new machines and computer hardware vendors have a vested interest in reducing the inertia of moving to a new machine. There is also a tendency not to want to rewrite a solution when the existing solution works just fine. As a result, there's a tremendous amount of software that was written a long time ago that is still being actively used. This sort of "lock-in" can lead to a resistance to change the software and hardware that is working.[4] Software and hardware vendors can benefit from this inertia to change and continue to charge for enhancements and support for very old software.
- *Concepts.* Concepts used to develop software are shared across many different kinds of software. These concepts evolve over time and many of these concepts continue to have relevance. Two examples are the concepts of *buffering* and *caching.* Buffering and caching have been used in many types of software and hardware to solve issues of differing speeds of operation and to make communication work more efficiently, ranging from CPU instructions, memory, paging, TCP sockets, data caching, and web caching. Web caching has expanded to also include content delivery networks (CDNs) that cache web content from around the globe in order to enhance the response time of web requests.
- *Specialization.* Many students and professionals, particularly in the last few decades, have become very specialized and have a deep but not necessarily broad view of software. Many students of technology and computing are no longer taught underlying topics such as assembler language, file systems, and other topics, yet these still form the basis for the systems we use today.
- *Trends and success factors.* Viewing the changes and trends in the software world can help to predict future trends and the direction of existing technologies. Furthermore, there are numerous failures and successes that are applicable to situations one experiences today.

4. As an example, there still exists a lot of COBOL code written in the 1960s and 1970s embedded in business software that is running today. An estimate from 2009 for COBOL's 50th anniversary puts the lines of running COBOL at over 250 billion lines of code, with Forrester noting that 32% of enterprises still use Cobol for development or maintenance. See Scott Colvey's article: "Cobol hits 50 and keeps counting," in *The Guardian*, April 9, 2009, https://www.theguardian.com/technology/2009/apr/09/cobol-internet-programming.

- *Dependence.* Current software depends on prior generations of software, both for concepts but also to run upon and with. The trajectory of future software depends on preexisting software. In particular, what is possible today is because of the preexisting software and other technology that already exists. Many systems are built on top of existing layers of software and are dependent on that software to remain stable.[5]

Software history is complex but is important to understanding the concepts and components of current systems. In particular, the environments and assumptions of earlier software systems can be very different from those of a newer system that is reusing or building upon the earlier software.

1.3 Modeling Software Technology Evolution

Software does not have a preexisting model for how its history should be organized. Because efforts are just beginning for the study of software history, there is no extant, single model for organizing the study of its history. Here it is argued that software is a technology, and a model is put forth for how to structure software history. The model presented here uses ideas from Arthur's [2009] work. Additional ideas used here come from Parayil [1999] and Constant [1980].[6]

1.3.1 Software As a Technology

Arthur [2009] notes[7] that technologies should have three defining characteristics:

1. A *linkage* to an underlying physical phenomenon that the technology leverages.
2. A *structure* such that the technology is built from other components.
3. These other *components* are, themselves, technologies.

Software and other technology components are then assembled into larger *subsystems* and *systems* that may include software from many different component technologies.

Software fits this model well. These three parts of the definition are addressed in turn as they apply to software.

5. That is, to remain stable in its interface or to have the interface be capable of being emulated by newer software. This is also true in reverse in a sense, where emulators are being built where older software can run again even though the hardware it was dependent upon is long gone.

6. Constant created a model for the Turbojet (based on Thomas Kuhn's [1962] earlier work) that contained many interesting concepts that are generally useful for software technologies, including *normal technology*, *traditions of practice*, and other useful concepts.

7. Arthur also notes several other definitions for technology, such as fulfilling a human need, that are not used here.

1.3.1.1 Linking Software to Physical Phenomena

Software leverages physical phenomena related to computation. The software must be connected to and run on some physical device(s) that can implement the various aspects of computation. If there's no physical device at all, that means there's no device for the software to run on. In particular, software runs on a computational device and may use other physical devices such as displays, sensors, storage devices, and output devices. Each of these pieces of hardware leverages different physical phenomena. However, software is often abstracted from these physical devices by other pieces of software such as the operating system, firmware, and compilers. The more portable the software is to other devices and hardware, the more useful it tends to be. As a result, software's tie to the specific physical phenomena that it exploits is loose as different hardware technologies can be used as a basis for computation. So, while software technology depends on physical phenomena, it is often abstracted from those phenomena. Often more important are the underlying concepts that software exploits in other layers of software. As an example, an application program written to read and write from a file does not worry about what kind of storage media is used to store that data; it uses the operating system's provided functions to read and write to and from the file. In fact, that underlying phenomenon may be different depending on where the file is stored (say a flash drive versus a disk drive versus a DVD versus paper tape). While the earliest computers did not specify an explicit programming interface such as an instruction set architecture with explicit opcodes (see Haigh et al. [2016] for how the ENIAC was programmed), it was quickly recognized that a specific instruction set (or "initial orders" as it was sometimes called) was helpful in creating a programming interface. As a result, there is an explicit abstraction from the hardware and the phenomena on which the hardware relies.

1.3.1.2 Software Is Built from Other Components

New software is often built from preexisting software components and technologies. When it's not created by reusing other software (say creating firmware for a device), it is still leveraging and using other technologies in order to produce something useful. So, typically new software systems will reuse other preexisting pieces of software along with newly built software in order to produce a running system. Software is often built as part of a *system*,[8] such as that defined in Hughes [2004, chapter 4, pp. 77–79]. As a system, software may be built with numerous subsystems and work with and interact with other non-software subsystems. So, a system such

8. As in Hughes [2004], systems are built to solve large problems and Hughes notes an emphasis on systems and the creation of a "systems age" that supplanted the machine age.

as an operating system is often broken down into component subsystems such as a process subsystem, memory subsystem, and a file management subsystem.

1.3.1.3 Software Components Are Technologies

The last point that these components are also technologies is a largely recursive point: most of the time, our components (or subsystems) are also software. When we are integrating hardware more directly, these hardware components are also technologies. This is also consistent with the concept of systems and subsystems as in Hughes [2004]. Software systems behave in a similar way to other technology systems by creating an interdependency between the components as well as being dependent on the social context for what is built and how it is built.

1.3.2 Software Domains

The concept of technology being composed of technology leads to the concept of *domains*. See Figure 1.1. The entirety of software technology could be considered a technology domain that is used by other areas of technology.[9] However, software technology itself is composed of domains of specific technology. This book is structured using these domains of software technology. Domains (see Arthur [2009, pp. 69–76]) are “any cluster of components drawn from in order to form devices or methods, along with its collection of practices and knowledge, its rules of combination, and its associated way of thinking.” So, for software technologies one such grouping would be operating systems, which has its own methods and collections of practices and knowledge. Furthermore, such a domain’s software history is interrelated to the extent that it provides a relatively cohesive evolutionary story. Another way to conceptualize a software domain is as the body of knowledge and expertise that someone working in that field should know to work effectively in that field.

1.3.3 Other Terms and Techniques

Besides segmenting software technology into domains, there are other important influences and terms used throughout this book.

Agendas [Mahoney 2011, pp. 165–169][10] have been used by historians to indicate a shared sense in the community of practitioners of the problems that need to be solved and the interrelationships between work done by different groups.

9. Before software was a separate concept, it was in the same technological domain as computing hardware and therefore part of the computer hardware domain until the 1950s. That is, it was highly dependent on a particular computer architecture and difficult to directly apply to a different computer.

10. See also the definition of *paradigms* in Kuhn [1962] as relates to “normal science.”

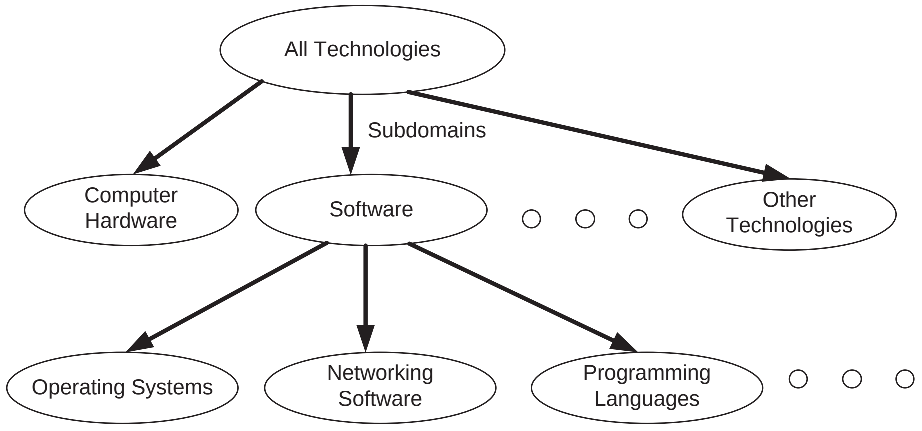


Figure 1.1 A hierarchy of technology domains and subdomains.

At any given point in time, there are *hot* problems in the area that are driven by need, funding, and the potential to solve. See Figure 1.2 for an example from Mahoney.[11] As an example, in the early 1950s it was clear that a more effective method was needed to program than writing assembler code or machine code. So, many groups were working to develop higher-level languages and compilers. At the same time, computers were becoming powerful enough and with enough memory to actually help in translating source code to machine code. There was also a need at the time to increase the productivity of programmers which would only become more important as computers became even more powerful. So, high-level agendas are used here to show some of the changes in focus of software over time. *Standard engineering* (see Arthur [2009, pp. 90–95] and Russell [2014]) is a term used by some historians to indicate (alternatively called *normal technology* in Constant [1980, p. 10]) the use of methods, devices, and principles that are known and widely accepted. Standard engineering in terms of software is using existing techniques, tools, and methods to solve new problems. The vast majority of software is of this standard engineering type. This book is mostly about when those tools, methods, and techniques are replaced with new concepts in software. Even in standard engineering, there are often new problems to be solved that result in new tools, methods, or techniques being developed that may eventually have a much larger impact. An example of standard engineering for software is the use of existing systems (database, operating systems, web servers, programming language, etc.) in order to write a new application program to allow online sales of a product. The techniques for solving this problem are well known. In the

11. From Michael Mahoney's papers at the Charles Babbage Institute.

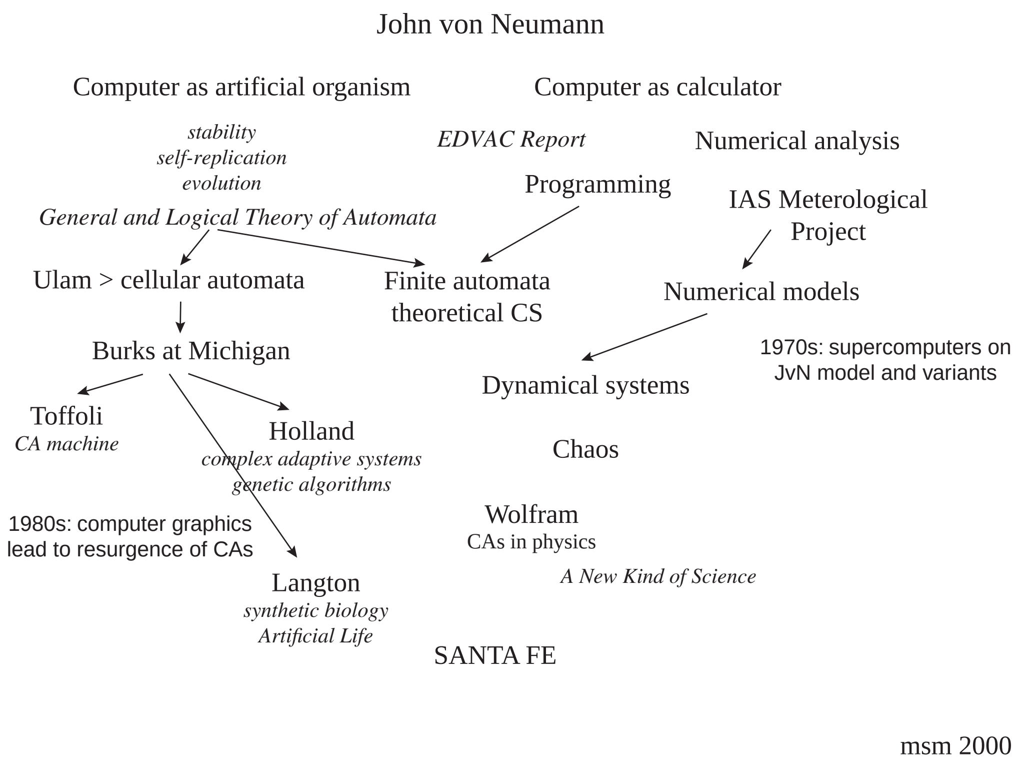


Figure 1.2 An example agenda from Michael Mahoney's papers, related to computing models. (Source: Image courtesy of the Charles Babbage Institute Archives, University of Minnesota Libraries, Minneapolis.)

course of writing this program, the programmer may notice a way to improve the compiler and invent a new technique or tool to help improve the way it operates. This change may be widely needed by others and eventually influence other systems.

Non-standard engineering, on the other hand, is creation of a new method, technique, tool, or system that influences the course of other software. There are many examples of where programmers are put in a situation where non-standard engineering must occur in order to solve the problem. A good example of this was the Semi-Automatic Ground Environment (SAGE) system in the 1950s. This system was built as part of the US government's air defense systems. One of its functions was to take input from a large number of radar stations and produce a single image for the North American Aerospace Defense Command (NORAD). Besides being a large software system to develop, it also had to operate a network of computers, integrate input from them, and display the information in graphical form. All of these tasks were unsolved problems at the time and at the early stage of development. The SAGE project had to solve these problems in order to produce a working system, making much of the project groundbreaking.

Events can often have a significant impact on the agendas and directions for the software community. An example event that had a significant impact is the 1988 Morris Internet Worm. At that time, only a small subset of commercial entities even had network firewalls, and there was not a large focus on network security outside of the military. After the Morris Worm brought down many hosts on the Internet, commercial companies (particularly those on the Internet at the time) began to take security seriously and to develop methods to protect their enterprise. Some software domains are more affected by the occurrence of specific events, such as security.

Influence diagrams (see Figure 1.3) are used to indicate the primary factors affecting change in a particular software technology domain or subdomain. These vary according to the domain and are intended as a mechanism to consider what is impacting the evolution of that domain. As an example, programmer tools have had direct influences from computer science in the theory of computation and the development of compiler technology and programming languages. Also, programmers' tools have been dependent on the sufficiency of the computational speed of the underlying computer, where techniques such as software development environments (SDE) weren't practical until computers had the computational capacity to run them. These tools were additionally dependent on the widespread availability of networking, graphical displays, and pointing devices such as mice. The evolution of programming methodologies to areas such as object-oriented design and

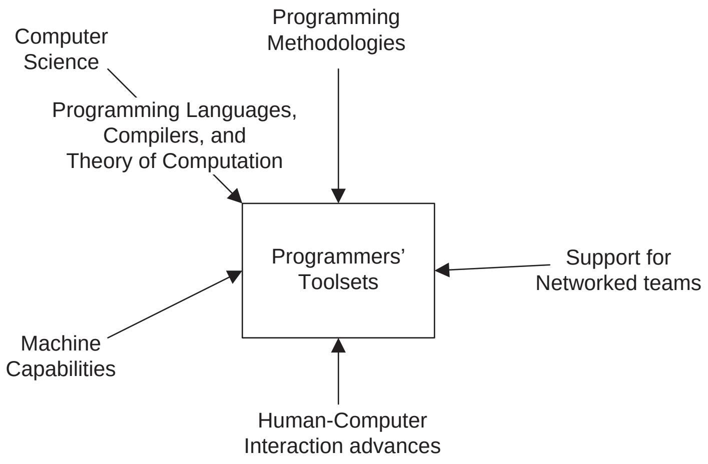

Computer Science
Programming Methodologies
Programming Languages, Compilers, and Theory of Computation
Programmers' Toolsets
Support for Networked teams
Machine Capabilities
Human-Computer Interaction advances

Figure 1.3 A sample influence diagram: Programmers' Toolsets.

Agile programming techniques contributed to redesigning how these tools worked and how teams used them. The intent of these diagrams is to help identify what influenced change in this area in the past and what might influence change in the future.

Environments are the circumstances and conditions under which software runs and is developed. This can include funding and social conditions for the development of software. However, it can also refer to the technical environment within which the system runs. This can have a great influence on the ability of the software to work at all, perform reliably, be secure, etc. An example of the funding environment that stimulated network software came from the US Advanced Research Project Agency (ARPA, becoming the US Defense Advanced Research Projects Agency [DARPA]) and the funding for Advanced Research Projects Agency Network (ARPANET) and Internet-related software. This funding was very influential in producing protocols such as TCP/IP, FTP, and others. The existence of ARPANET (and other networks including the Internet and UUCP) contributed to an environment that facilitated research and information sharing between those universities and research labs that were connected to it. ARPANET was justified and envisioned (see Licklider [1963] for Licklider's memorandum on "intergalactic computer networks") as a network that would facilitate information sharing and research. ARPANET did (along with additional ARPA funding and other networks) stimulate and support significant research in programming languages, artificial intelligence, electronic mail, early social networking as well as many other research areas.

The technical environment for software can include the operating system, configurations, the computer itself, and peripherals. An unsecured environment can lead to the software's security being circumvented. Events can cause the environment to change, such as the funding environment may change in response to an event. As an example, the 1988 Internet Worm caused federal funding to be issued to create the Computer Emergency Response Team (CERT) among other efforts to improve security.

1.3.4 How Software Technology Evolves

Software is constantly changing and programmers for a particular software system pick tools and reuse software based on many factors and influences. Those influences for what to use for a particular system range from the ease-of-use, fit to the software being developed, corporate standards, and many others. However, there are some common drivers to develop new techniques. These are certainly not the only ways software evolves but give a perspective from the history of technology point-of-view.

(1) *Functional failure*. Constant [1980, pp. 12–13] defines a functional failure to be when a technology fails under new or more stringent conditions. In terms of software, one might call this a *functional inadequacy.* As a result, new techniques are required in order to meet these new conditions. An example of this in software is when electronic business (sales over the Internet) became popular in the mid-1990s. In the rush to get sales on the Internet going, many systems were repurposed with a web front-end and still used the same back-end system. The problem was that the security and reliability conditions had changed for the back-end system and many systems failed as a result. As a small example, this author recalls a survey system that granted a gift certificate at the end of the survey to an online book seller. At the end of the survey, it gave an error message, so I reloaded the page. In another window, in came two gift certificates to the online book seller. Another reload of the error page, another gift certificate came in. It quickly became a way to print money at the survey giver's expense. New techniques had to be developed to better control the interchange between the back-end systems and users on the web (such as web application servers and securing eXtensible Markup Language [XML]).

(2) *Combination*. Arthur [2009] cites the combination of technologies as one of the major ways that technology evolves. Certainly, for software this has also been the case. An example of this sort of change is the introduction of Global Positioning System (GPS) hardware on many devices.[12] As a result, software has used the results from GPS to offer many location-based services through software such as real-time alerts of traffic problems in your vicinity. Entire areas of software and computing have been created to reflect the combination of software and computing technologies with another field, such as computational biology, computational chemistry, etc.

(3) *Technological co-evolution*. Constant [1980, pp. 13–15] defines the notion of technological co-evolution where multiple technologies are dependent on one another. For software, the evolution of computing hardware has been co-evolutionary with software. Faster computers have allowed the use of techniques that were not viable in the past, thereby releasing their use. A good example of this has been voice recognition. Many of the actual techniques used in voice recognition have been available for some time, but it has been difficult to deploy due to the amount of computing resources required. Other

12. GPS works with a series of satellites that provide location data to GPS receivers and was developed for military navigation systems but is useful anytime that location is informative for an application.

examples include the creation of graphic displays to allow a method for displaying graphics and the ability to use machine learning on large datasets. Software has also sometimes affected the evolution of hardware. The advent of recursion in programming languages influenced the design of later computer instruction sets such as the support of stacks and reentrant procedure call mechanisms.

(4) *Structural deepening.* Arthur [2009, pp. 131–143] also notes that structural deepening defines new technologies by creating more complex subdomains and making them more useful. This certainly happens with software technologies. For example, operating systems started off with a minimal set of features to ease working with the computer and its devices, but have since added many features such as virtual memory, multi-processing, multi-user, and support for myriad devices. As a result, operating systems have a lot of subdomains such as memory management that can be called software technology domains in their own right.

    Domain specialization is a form of structural deepening where a specific variant is developed for a specific use or market. Using operating systems as an example, real-time operating systems have been developed to be able to respond to real-world events in a predictable amount of time. Real-time operating systems often are preemptive and will force processes to terminate or give up the CPU in order to meet the time requirements.

(5) *Innovation and inventions.* Occasionally, new software domains and new software technologies are created by a new invention, or more commonly by improving an existing technology by finding an innovative use or making a small improvement. Even new software inventions are generally based on existing technologies and usually have a need or pressure to resolve a problem. As an example, the first database (Charles Bachman's Integrated Data Store) emerged in the early 1960s as the result of a need for such a system as well as being based on existing file systems and reporting systems. Many software systems have been driven by an economic or opportunistic drive to beat the competition or to enter new, potentially lucrative markets.

Another example of the application of innovation to software change is that there has been a desire and push to deploy computers to new sets of users, particularly as computers have become less expensive. This has driven a need to innovate how that software will be used by a wider variety of users, many of whom will have less technical knowledge. An innovation that fits this model is the

Internet browser[13] that expanded the use of the Internet dramatically to new sets of users. The proliferation of widespread wireless data networks along with smartphones has brought the Internet and data applications to an even broader set of users.

Timelines showing major changes and events will also be used to show how a particular domain has changed over time.

1.4 Computer Hardware History

The capabilities and power of computer hardware directly influence software's capabilities and power. This section provides a brief overview of computing hardware history, from a software capability point of view. Figure 1.4 shows how calculating devices evolved into basic computers with major milestones. It shows some of the influences that significantly impacted later stages of development. Figure 1.4 spans hundreds of years beginning with the development of methods of computation and mechanical calculators. Making numerical computation more efficient and more accurate is what drove many of the innovations in Figure 1.4.

In this text, the distinction of being before the von Neumann architecture is significant because how these machines were programmed was heavily intertwined with the design of the hardware. After the von Neumann architecture was implemented (first in the United Kingdom), the program as a concept became more distinct from the hardware and was *fed* into the computer, rather than being *part* of the computer. With that change, the notion of having a fixed instruction set in the computer that the program would use to encode its operations took hold.

1.4.1 Hardware Before the von Neumann Architecture

Figure 1.4 shows a high-level diagram of computing hardware before the Electronic Numerical Integrator and Computer (ENIAC) and the development of von Neumann architecture machines. For centuries, devices that could automate computation and eliminate the drudgery and inaccuracy of manual computation had been attempted and achieved. In Figure 1.4 some of the most important and influential achievements that led to the creation of fully electronic, fully programmable computers are included. For a more complete view of this history, please see other works such as Williams [1997].

Numbering systems were critical to the ability to make computation more efficient and accurate. Logarithms and, specifically, John Napier's Bones were very influential to early computational devices in the 16th and 17th centuries. With the concept of logarithms, devices such as slide rules could be built that leveraged

13. Attributed to Sir Tim Berners-Lee with the creation of the WorldWideWeb browser in 1990.

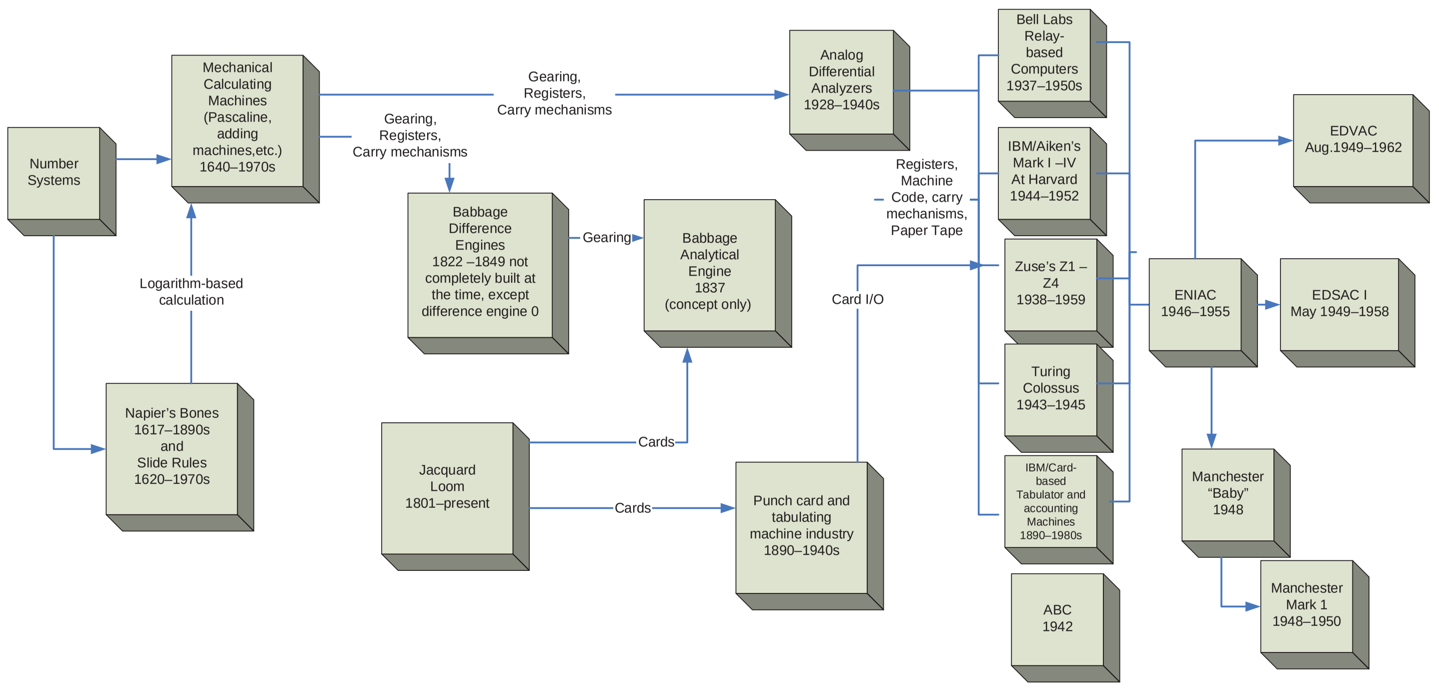

Number Systems
Mechanical Calculating Machines (Pascaline, adding machines,etc.) 1640–1970s
Logarithm-based calculation
Napier's Bones 1617–1890s and Slide Rules 1620–1970s
Gearing, Registers, Carry mechanisms
Gearing, Registers, Carry mechanisms
Babbage Difference Engines 1822 –1849 not completely built at the time, except difference engine 0
Gearing
Babbage Analytical Engine 1837 (concept only)
Jacquard Loom 1801–present
Cards
Cards
Punch card and tabulating machine industry 1890–1940s
Card I/O
Analog Differential Analyzers 1928–1940s
Registers, Machine Code, carry mechanisms, Paper Tape
Bell Labs Relay-based Computers 1937–1950s
IBM/Aiken's Mark I –IV At Harvard 1944–1952
Zuse's Z1 – Z4 1938–1959
Turing Colossus 1943–1945
IBM/Card-based Tabulator and accounting Machines 1890–1980s
ABC 1942
ENIAC 1946–1955
EDVAC Aug.1949–1962
EDSAC I May 1949–1958
Manchester "Baby" 1948
Manchester Mark 1 1948–1950

Figure 1.4 A high-level view of computer hardware before EDSAC and EDVAC.

the computational properties of logarithms. Many calculating devices were built in the 17th and 18th centuries such as Schickard's calculator, Pascal's calculating machine, and Leibniz's calculating machine, among others. Descriptions of these machines and their evolution is well described in Williams [1997]. Sophisticated gearing mechanisms and mechanisms designed to carry a one to the next higher decimal place for addition (and multiplication) were reused for larger scale computational devices in the 19th century, in particular Charles Babbage's difference engine. Babbage also designed (but did not get to function at the time) the Analytical Engine (around 1840), which had some programmability and contained a number of ideas that would later be implemented for electronic computers. The Analytical Engine had a number of features that increased the flexibility of the device: in particular, having programmability via cards (see http://www.fourmilab.ch/babbage/cards.html) and, interestingly, the ability to produce graphical output (in a plotter-like way). A simulator of the Analytical Engine and a compendium of other related material can be found at http://www.fourmilab.ch/babbage/contents.html. The Analytical Engine used many ideas that were rediscovered 100 years later, in particular separating data from the program control. While those building computers in the 1930s and 1940s were aware of Babbage's earlier work, the direct influence of his work on them is minimal, at best. Howard Aiken at Harvard University was aware of Babbage's work and it may have impacted his work on the Harvard Mark I to Mark IV computers.

The history of electronic, programmable computers really begins in the late 1930s with a number of competing efforts and are briefly covered here. These efforts included the work at AT&T Bell Telephone Laboratories on relay-based machines by George Stibitz, Howard Aiken's Mark I (IBM ASCC) computers at Harvard University, Konrad Zuse's Z1 to Z4 in Germany, and IBM's extensive work on tabulating machines. Turing also developed the concept of a universal Turing machine during the 1930s.

A brief description of each of these efforts in the late 1930s follows:

- *Turing machine concept*. Turing published a now famous paper in 1936 describing a theoretical model for a universal computing machine [Turing 1936]. This paper was in response to a problem proposed by mathematicians Hilbert[14] and Ackermann in 1928 known as the *Entscheidungsproblem* (decision problem), which was also worked on by Alonzo Church (best known for lambda calculus). The Turing machine concept has since become the foundation of computation theory. This paper possibly influenced von Neumann's

14. David Hilbert also proposed a set of unsolved mathematical problems in 1900 including #2 that led to Kurt Gödel's Second Incompleteness Theorem.

work and may have influenced Turing's thinking in the design of later computers, such as the National Physical Laboratory (NPL) Pilot ACE (Automatic Computing Engine). There is very little direct influence of this paper on early computers, but the paper had a large influence in proving what could and could not be computed by computers.

- *Bell Labs relay computers*. One of the primary drivers for AT&T at this time was to be able to automate telephone switching.[15] Telephone usage was growing rapidly and quickly overtaking the feasibility of using human operators for every call (see Figure 1.5). Figure 1.5 shows some of the difficulties in scaling manual telephone switching already occurring in 1929 with a large number of operators needed and messengers on skates collecting billing for each call. The practical limit on how many lines a single operator could manage was about 10,000, limited by how far an operator could reach (see Bell Laboratories [1977, p. 189]), which also caused increasing costs as the number of telephone lines and calls increased. George Stibitz used electromechanical relays[16] to design a series of machines beginning in 1938 with a machine called the Complex Number Calculator (CNC) designed to perform arithmetic operations on complex numbers. This machine was later called the Model 1. The CNC (or Model 1) used a teletype interface as in Figure 1.7. Bell Labs continued work on specialized computers (many for the military) into the 1950s. See Figure 1.6 for a photo of the AT&T Bell Labs CNC and Figure 1.9 for Stibitz's reconstruction of Model "K" (for "kitchen") model that Stibitz built to demonstrate the concept of using relays for addition. AT&T Bell Telephone Laboratories continued to develop a number of specialized computers (see Figure 1.8), particularly those related to telephone switching. See Irvine [2001] and Andrews [1982a, 1982b] for more on the Bell Labs relay computers.
- *Differential analyzers*. The Massachusetts Institute of Technology's Vannevar Bush and Harold Locke Hazen developed an analog machine designed to integrate differential equations in 1928 and went on to use the device for many years. Other devices for integration had been built and proposed by

15. In 1907, Theodore Vale as president of AT&T adopted the slogan of "One Policy, One System, Universal Service," which drove it to begin to develop a nationwide monopoly. In 1913, AT&T entered into an out-of-court agreement known as the Kingsbury Commitment, which was replaced by the 1921 Willis–Graham Act that established telephone companies as natural monopolies. There were already 18,522,767 telephone stations in the United States by 1927 [Daggett 1931] with 13,726,056 of them controlled by the Bell System.

16. A *relay* is an electrically operated switch, as relays had been successfully used in other parts of the Bell System. Claude Shannon (see Shannon [1940]) described how to use relays to design digital circuits that implement logical functions.

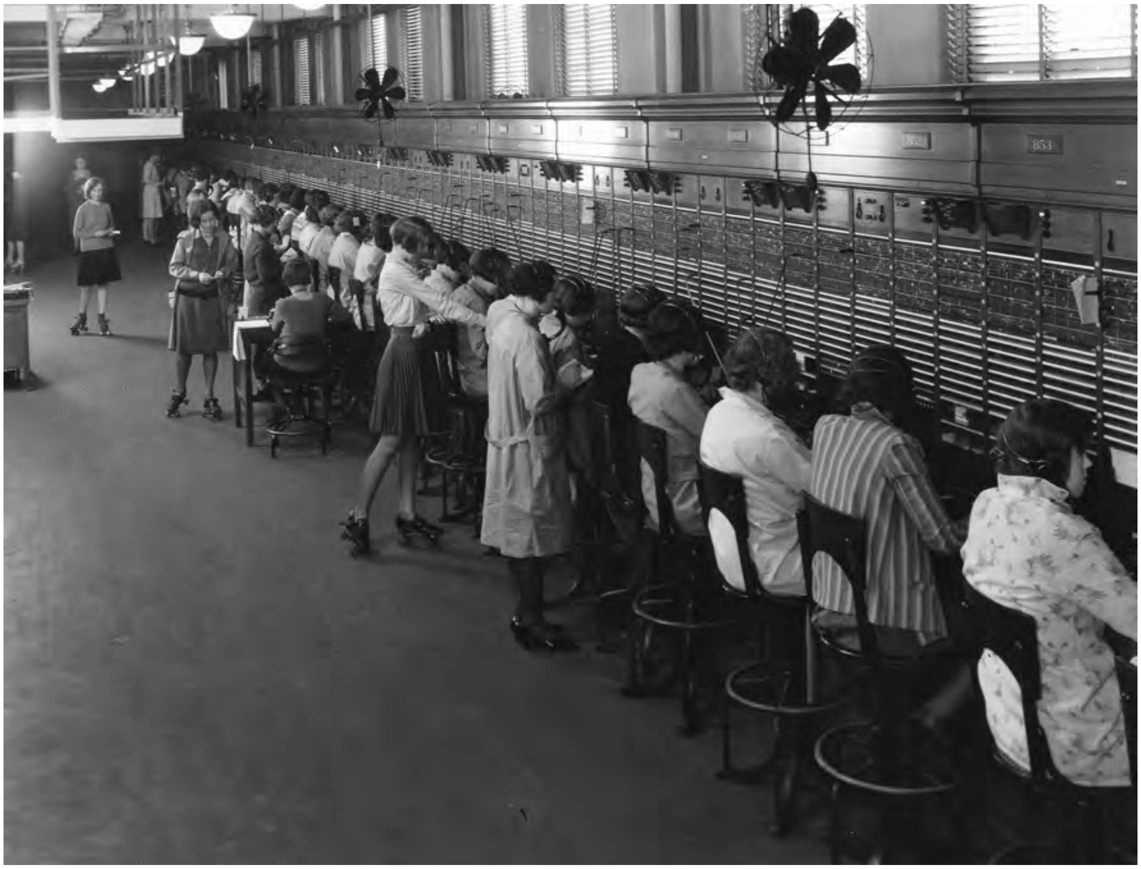


Figure 1.5 The Chicago Long-Distance (Toll) Office in 1929. (Source: Courtesy of AT&T Archives and History Center.)

Lord Kelvin in the 1870s. Numerous other differential analyzers were built around the world, including in Norway, Japan, Canada, and England, and continued to be used into the 1940s.

- *Harvard Mark I*. Aiken at Harvard University proposed a general-purpose electromechanical computer in 1937 that was designed, built, and funded by IBM in 1939. IBM named it the Automatic Sequence Controlled Calculator (ASCC) and it began work for the US Navy Bureau of Ships in 1944. The Mark I used tapes and was built using switches and relays. Aiken continued work on additional machines named the Harvard Mark II (see Figure 1.11), Harvard Mark III (or Aiken Dahlgren Electronic Calculator, ADEC), and Harvard Mark IV. See Cohen and Welch [1996] for more on Aiken and the Harvard computers and http://www-03.ibm.com/ibm/history/exhibits/markI/markI_reference.html for IBM's ASCC site. See Figure 1.14 for a photo of

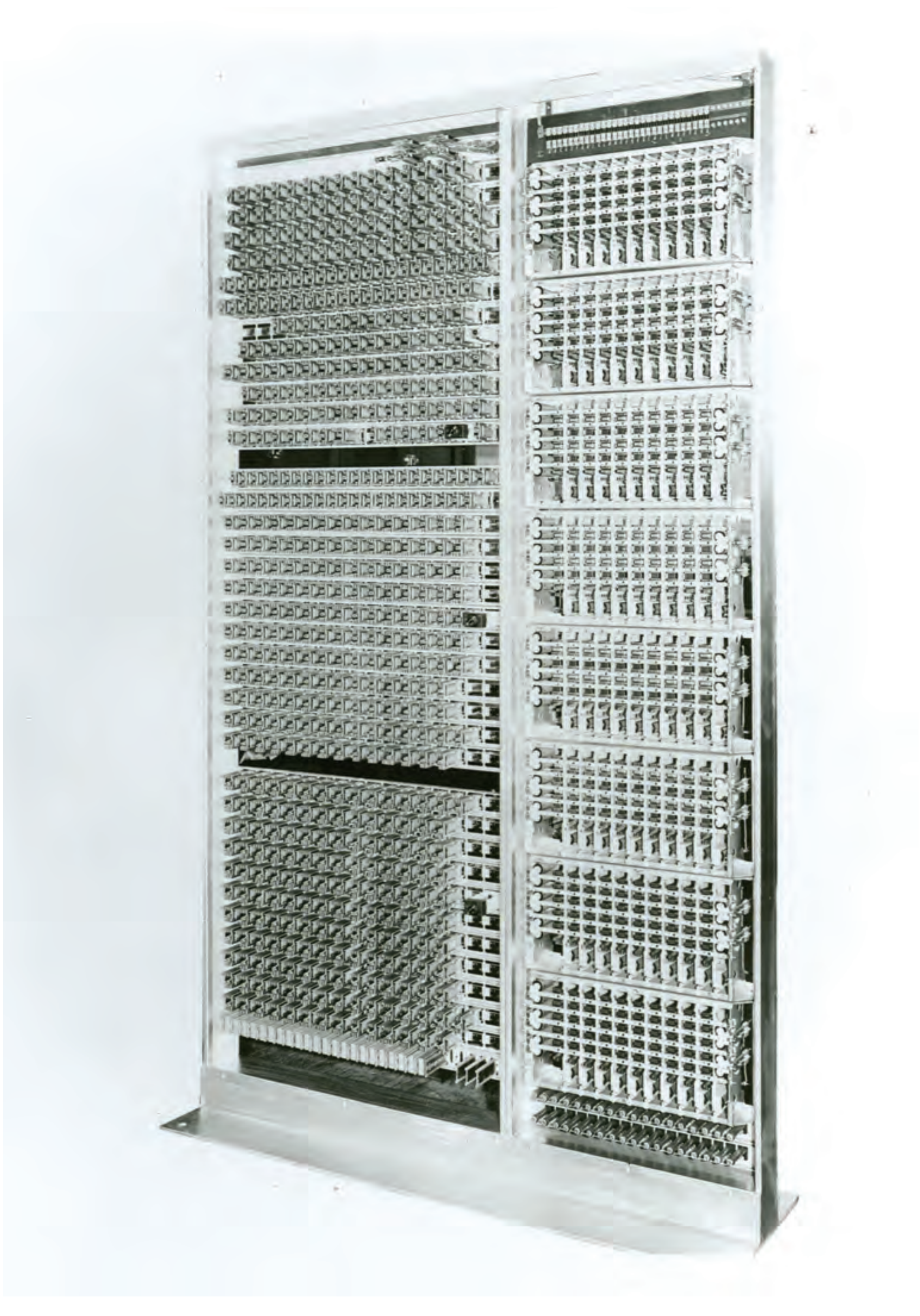


Figure 1.6 Bell Labs Complex Number Computer, which used relays for logical operations (1939). (Source: Courtesy of AT&T Archives and History Center.)

the ASCC,[17] Figure 1.13 for an example of one of Grace Hopper's tapes used on the Harvard Mark I (ASCC), and Figure 1.12 for a closer look at this tape and the "patches" used to correct the tape.

- *Zuse's Z1 to Z4*. In Germany, Konrad Zuse began construction of his Z1 machine in 1936, which was a programmable, digital mechanical device with no relays or vacuum tubes. In 1939, Zuse created the Z2, which did use relays for computation. In 1941, Zuse completed the Z3, which was used by the German Aircraft Research Institute to perform statistical analyses. The Z3 used 2,000 relays. The Z4 design was completed just before the end of World War II but was not used until after the war. See http://zuse.zib.de/ for details from the Konrad Zuse Internet Archive. Zuse continued to develop a line of computers after the end of World War II.
- *IBM tabulating equipment*. While not directly in the computer business at this time,[18] IBM had created a lot of the peripherals that would be reused in computing, in particular card punching and reading equipment. IBM had built a business out of these devices. Based on Herman Hollerith's machines that were used in the 1890 US census (see Truesdell [1965] and Figure 1.16), IBM developed tabulating equipment with the ability to also do calculations. These include subtraction (1928), multiplication (1931), the IBM 805 Test Scoring Machine (1937), and the IBM 801 (1934) that was able to clear bank checks. These sorting and tabulating machines were complex, electromechanical devices that performed many business functions and established the market for mechanization of business functions. In addition, IBM won the 1935 contract with the US government to support the implementation of

17. As described by IBM: "This undated view of the ASCC shows (at right) its three interpolators—the value tape mechanisms which automatically selected values required in interpolating processes—next to which are (from right to left) the functional counters, multiplying-dividing unit and storage counters."

18. IBM was actually heavily involved in exploring computing in the 1930s and 1940s, particularly with work at Harvard with the Mark I. The Mark I was built by IBM as noted above. In addition, IBM established the Watson Scientific Computing Laboratory in 1945 at Columbia University, clearly focused on computing and built the Selective Sequence Electronic Calculator (SSEC) that first operated in 1948. IBM had been working with Columbia University with the establishment of the Thomas J. Watson Astronomical Computing Bureau in 1937. IBM was also named the primary computer hardware vendor for the SAGE project in 1953. In 1952, IBM produced a very successful computer, the IBM 701, which propelled it into the computer business. IBM was able to leverage its embedded base of tabulating equipment to quickly enter the computer market.

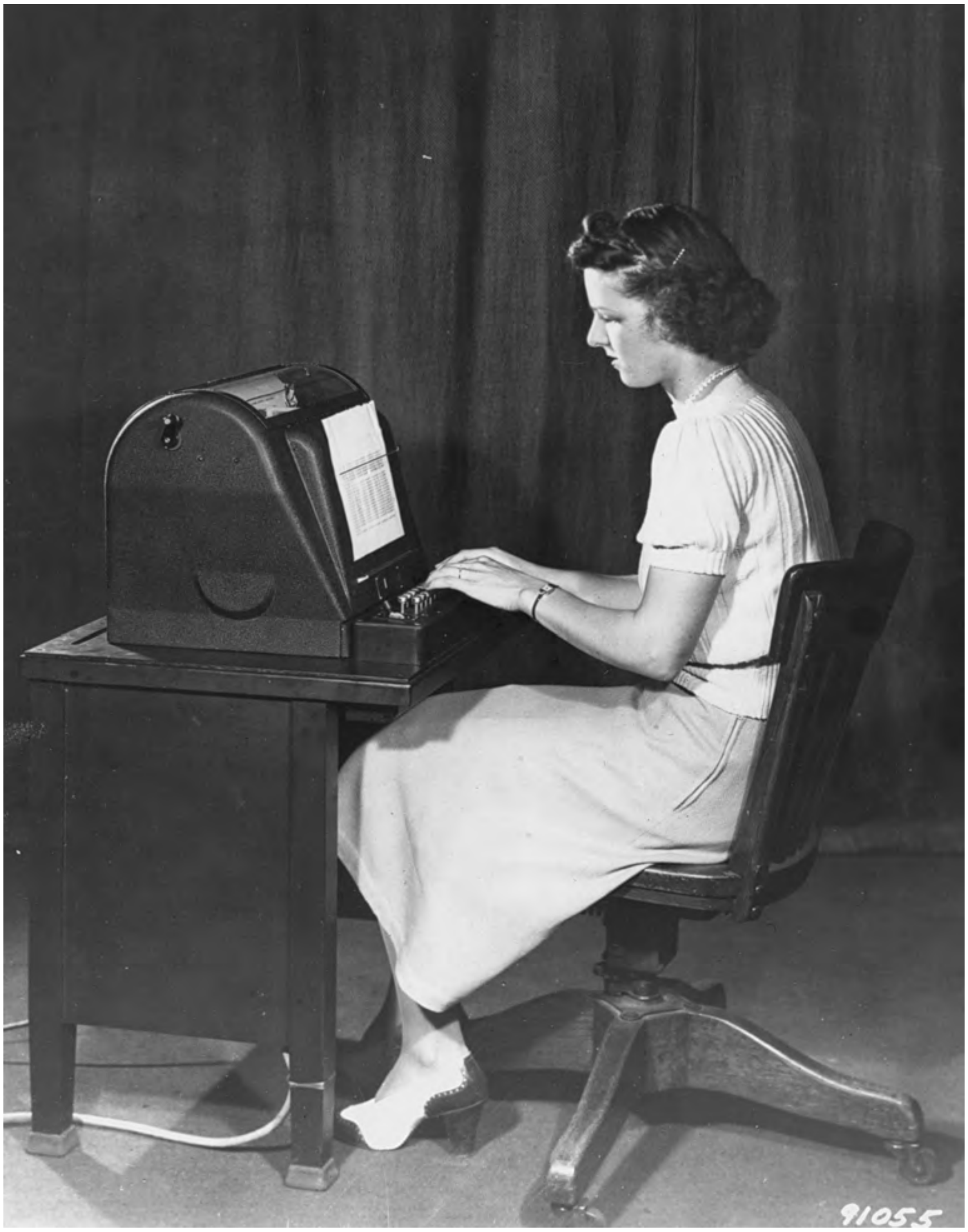


Figure 1.7 H. L. Marvin operating the Bell Labs Complex Number Computer teletype interface (1939). (Source: Courtesy of AT&T Archives and History Center.)

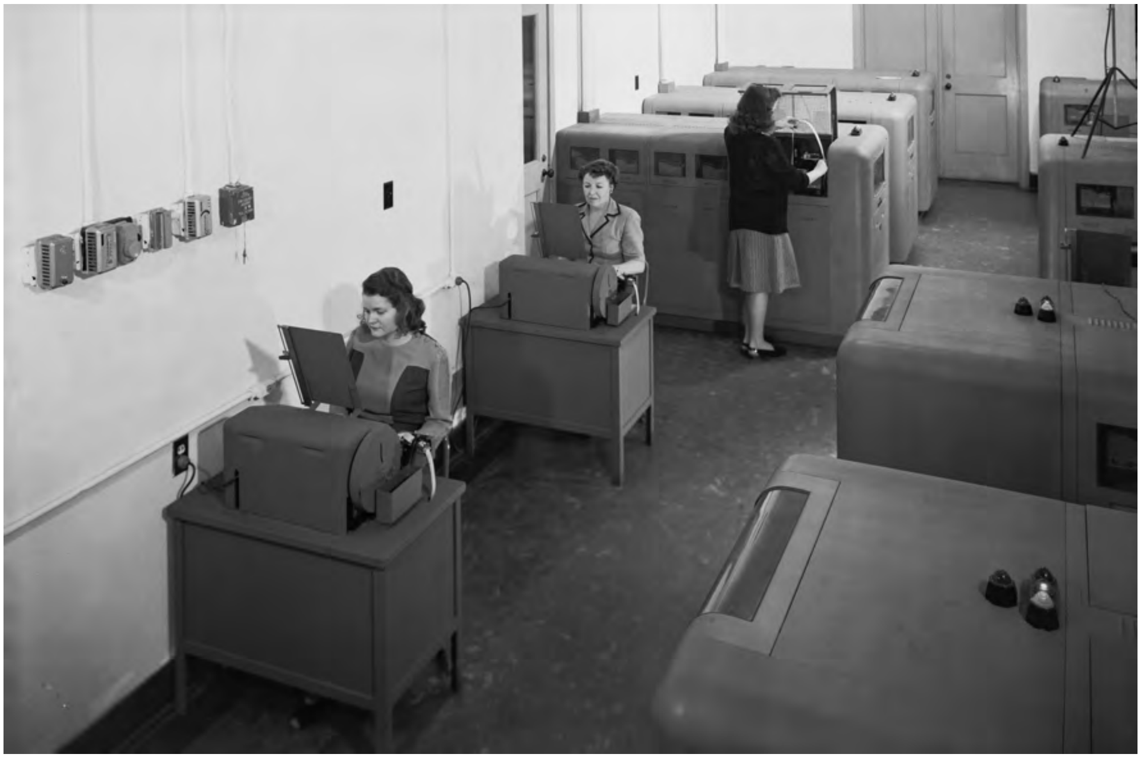


Figure 1.8 Bell Labs relay computers in use at Langley Research Center (1947) by female “computers” who performed mathematical computations for male staff. Note: the actual relay computer is not shown (these are the input/output devices). (Source: NASA, credit: NACA Langley.)

the Social Security Act. IBM was also working with the Columbia University Statistical Bureau and built a complex tabulator in 1931 that was called the “Columbia Machine.” Another example is the IBM Card-Programmed Electronic Calculator, announced in 1949, that was used in the space program and aided in the development of the Redstone rocket that was used in the Mercury space program (see Figure 1.17). IBM had a number of items that were readily reusable as computer input/output equipment, as was the case for computers such as the Atanasoff–Berry Computer (ABC) and the ENIAC, described below. See Yost [2011], Pugh [2009], and Maney et al. [2011] for more on IBM’s role. Some excellent resources showing the evolution of IBM’s early computers is Bashe et al. [1981, 1985]. An excellent resource covering the breadth of IBM’s business and technical history is Cortada [2019].

This work in the late 1930s was critical in preparing to build general-purpose computers in the 1940s. Besides developing basic computational devices, they also

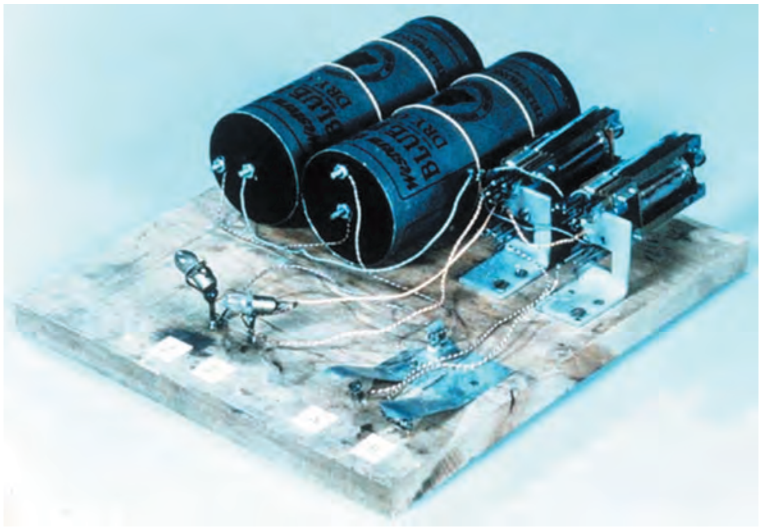


Figure 1.9 Stibitz reconstruction of Bell Labs "Kitchen" Model, original built in 1936. (Source: Courtesy of AT&T Archives and History Center.)

developed key components that would be helpful in later phases such as the use of paper tape and punched cards for input and output. In the 1940s, much of the work on computers was funded by and in support of the military and continued through World War II. These included the work by Turing and team at Bletchley Park on the code-breaker machines Bombe and Colossus, as well as work by Eckert and Mauchly on the ENIAC at the Moore School at the University of Pennsylvania. Atanasoff and Berry worked on the ABC at Iowa State University (then Iowa State College) before World War II. Atanasoff discontinued work on the ABC when he took a post supporting the war effort.

Figure 1.18 shows the early computers described in this chapter on a timeline beginning in the late 1930s into the early 1950s. Efforts from the United Kingdom are shown in blue, from the United States in green, and from Germany in red.[19] As indicated in Figure 1.18, computers were beginning to be sold to businesses.

This work from the early 1940s is briefly described below:

- *Bombe and Colossus*. This work was created to break German encryption algorithms during World War II. The original breakthrough came from Poland, and the creation of the Polish "bomba" to break the German Enigma encryption device. This work continued with the creation of the UK Bombe to continue to work on various versions of Enigma. The Bombe essentially worked

19. This diagram gives a rough timeline that is not meant to be precise (with a granularity of a year, so efforts that occurred in the same year are shown as occurring at the same time).

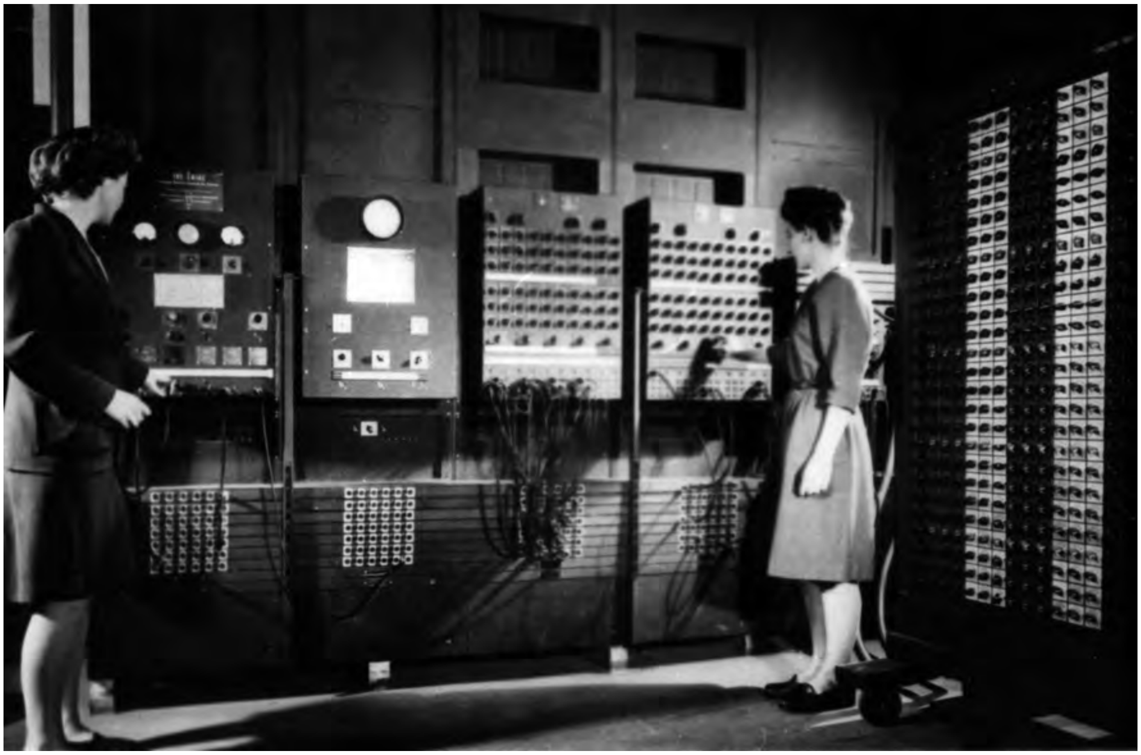


Figure 1.10 Betty Jennings (left) and Frances Bilas (right) operating the ENIAC's main control panel at the Moore School at the Univ. of Pennsylvania. (Source: US Army photo from the archives of the ARL Technical Library.)

by simulating a number of Enigma machines in order to find the current code used.[20] Other governments, including France and the United States, were also involved in this work. Colossus was a machine built by T. H. Flowers in order to break the Lorenz Schlüsselzusatz cipher in Bletchley Park. Colossus was a much more sophisticated code-breaking machine built specifically to process ciphertext.[21,22]

20. See a paper by Fredrich Bauer for more on Bombe, which is included in De Leeuw and Bergstra [2007, pp. 381–446].

21. See a paper by B. Jack Copeland in De Leeuw and Bergstra [2007, pp. 447–477] for a description of Colossus and how it impacted later computers including the Automatic Computing Engine (ACE) and the Manchester "Baby," among other UK computers.

22. Also see http://www.alanturing.net/ for an archive of Turing-related documents including the Bombe, Robinson, Colossus, ACE, and others. The Turing Digital Archive includes scans of many of Turing's papers and papers related to Turing's work at http://www.turingarchive.org/.

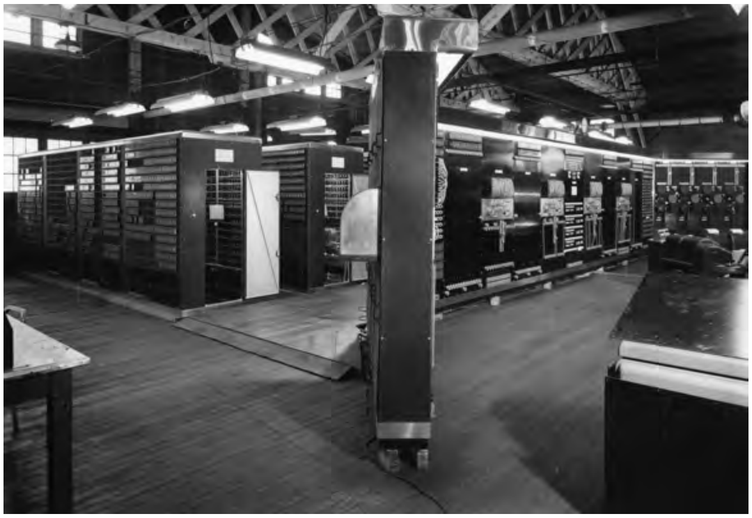


Figure 1.11 General view of the Harvard Mark II calculator frontispiece. (Source: Courtesy of Grace Murray Hopper Collection, Archives Center, National Museum of American History, Smithsonian Institution.)

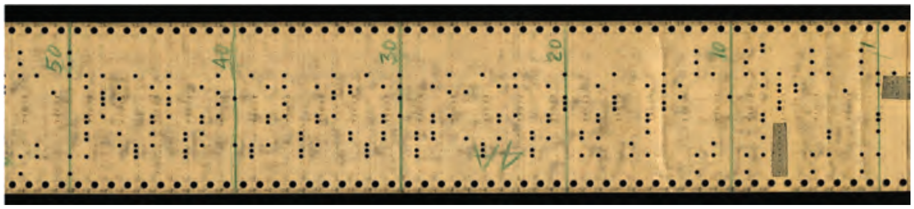


Figure 1.12 Mark I Problem L paper tape with "patches." (Source: Courtesy of Grace Murray Hopper Collection, Archives Center, National Museum of American History, Smithsonian Institution.)

- *Atanasoff–Berry Computer (ABC)*. John Atanasoff had developed a number of computing devices before the ABC, notably an analog calculator called the "laplaciometer." His proposal for funding of what would become the ABC came to the Iowa State College in March 1939 (see http://jva.cs.iastate.edu/img/Computing%20machine.pdf for a scan of the original proposal). The ABC was targeted at solving large systems of linear algebraic equations. Mauchly had visited Atanasoff in Iowa and was aware of his ideas. With the entry of the United States into World War II after Pearl Harbor, Atanasoff (and his graduate assistant, Berry) joined the war effort and work on the ABC was halted. Atanasoff had filed a patent for the ABC via Iowa State College and

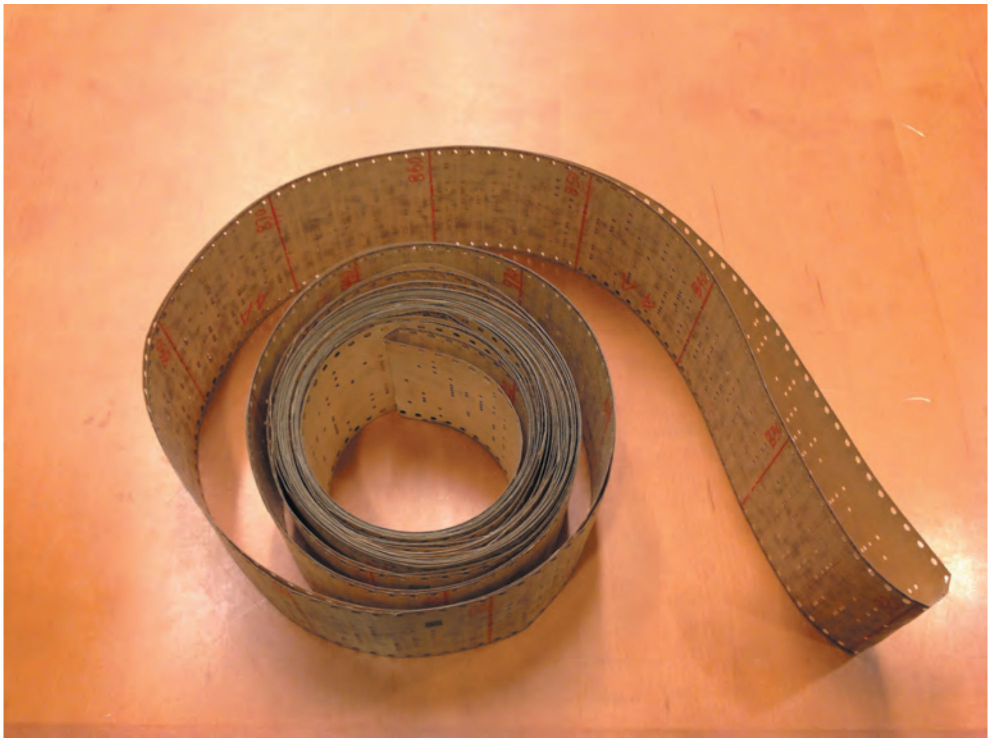


Figure 1.13 One of Grace Hopper's paper tapes used on the Harvard Mark I (tape 1 of 4 for Problem "L," Bessel function tables). (Source: Author's photograph of paper tape courtesy of the Grace Hopper Murray Collection, Archives Center, National Museum of American History, Smithsonian Institution.)

this patent became involved in a complex patent dispute with Eckert and Mauchly's ENIAC patents. The case was eventually resolved in ABC's favor in 1972.[23] See Figure 1.19, which shows the drum memory (based on capacitance, so it needed to be refreshed periodically) used by the ABC as well as a partial view of the tubes used for the accumulator and other circuits. The drum memory is the sole surviving component of the ABC computer and is shown in Figure 1.15.

- *Electronic Numerical Integrator and Computer (ENIAC)*. The ENIAC was built in response to the need for accurate firing tables to be used in World War II.

23. Note that this case really didn't resolve the technical dispute of who created the first general purpose computer. Certainly, Atanasoff's ideas were important and the ABC exhibited a number of firsts, but there are still arguments about which computer was the "first" to fully exhibit all the properties of being electronic, programmable, and digital.

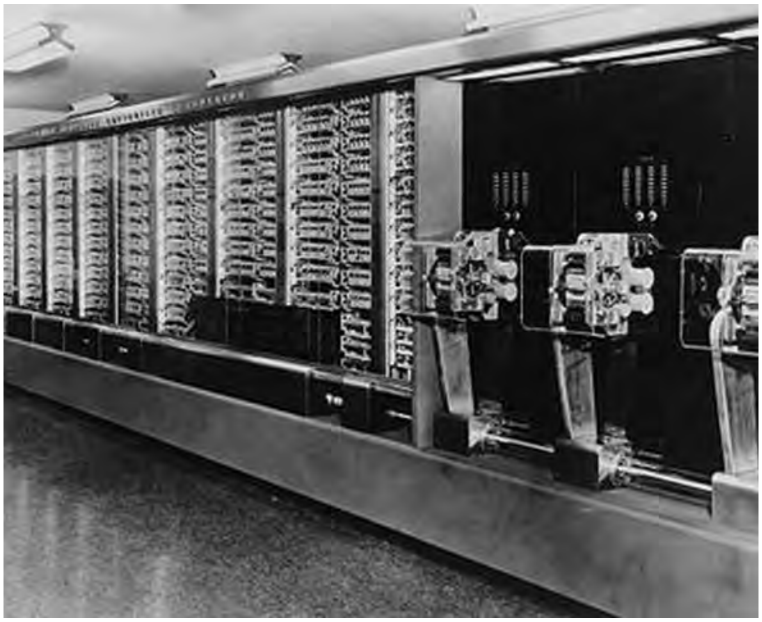


Figure 1.14 IBM ASCC (or Harvard Mark I, undated). (Source: Courtesy of International Business Machines Corporation, ©International Business Machines Corporation.)

The original contract to the Moore School of the University of Pennsylvania was in June 1943 for "research and development of an electronic numerical integrator and computer and delivery of a report thereon."[24] The ENIAC was a huge, vacuum tube-based, special-purpose computer that used IBM cards for input and output. It had 18,000 vacuum tubes and 1,500 relays and was the largest computing effort ever completed up to that time. As a result, the ENIAC became the turning point for the creation of a computing industry by Eckert and Mauchly. The experiences from ENIAC led to more advanced and general-purpose systems including the BINAC, EDVAC (see Figure 1.21), and the formation of the Eckert–Mauchly Computer Corporation (which was sold to Remington Rand in 1950). Please see Figure 1.10 for a photo of the ENIAC control panel and Figure 2.3 for an image showing how the ENIAC was programmed during its early use. Also see Figure 1.20[25] for an example of using punched cards with the ENIAC in 1946.

24. See Haigh et al. [2016] and the accompanying http://eniacinaction.com for more about ENIAC's story.

25. Hagley ID: 1985261_001_001_023, Box/folder number, Sperry Corporation, UNIVAC Division photographs and audiovisual materials (Accession 1985.261), Audiovisual Collections and Digital Initiatives Department, Hagley Museum and Library, Wilmington, DE 19807.

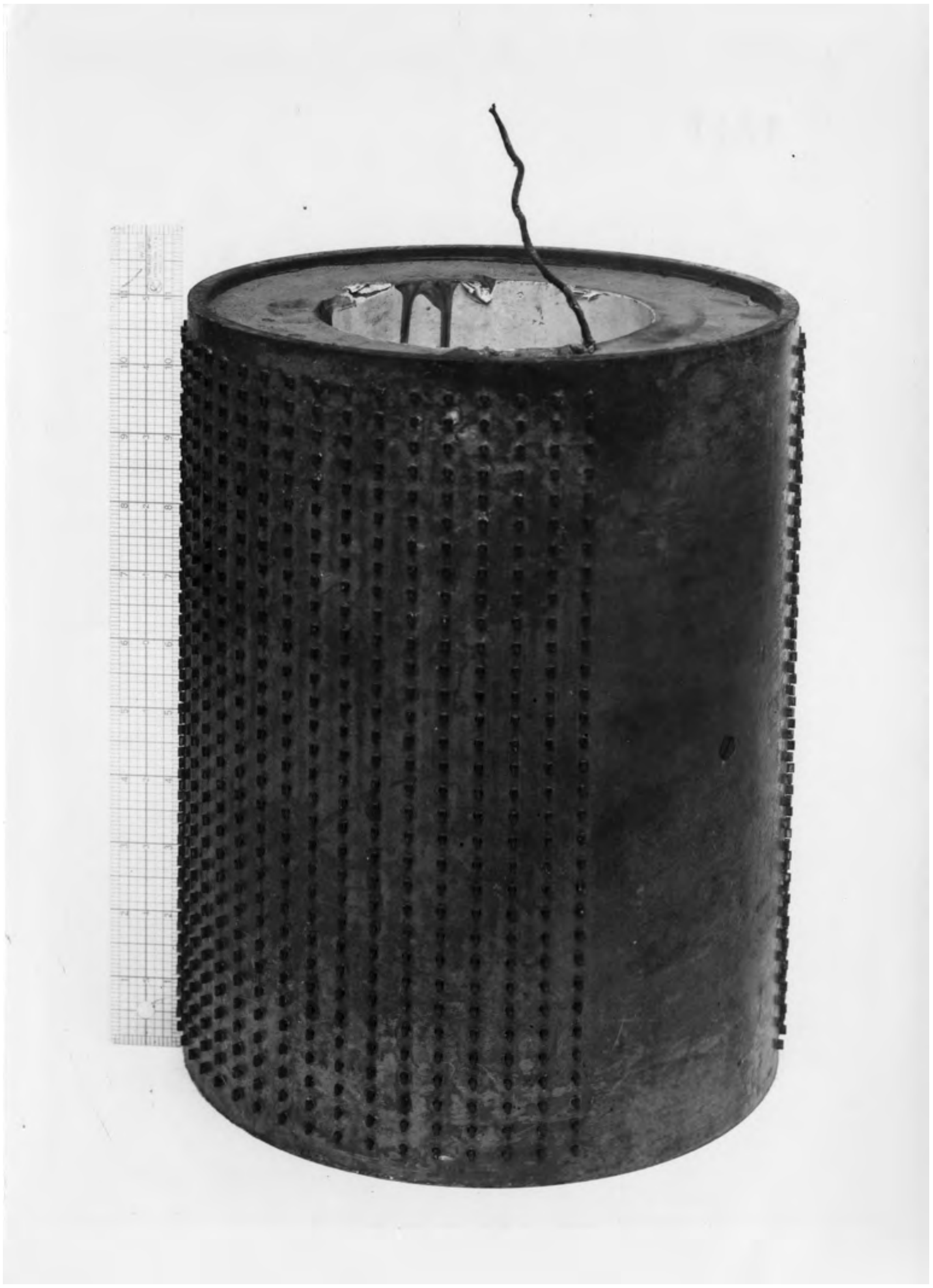


Figure 1.15 The Atanasoff–Berry Computer's memory drum—the sole surviving component. (Source: Iowa State Univ. Library Special Collections and Univ. Archives.)

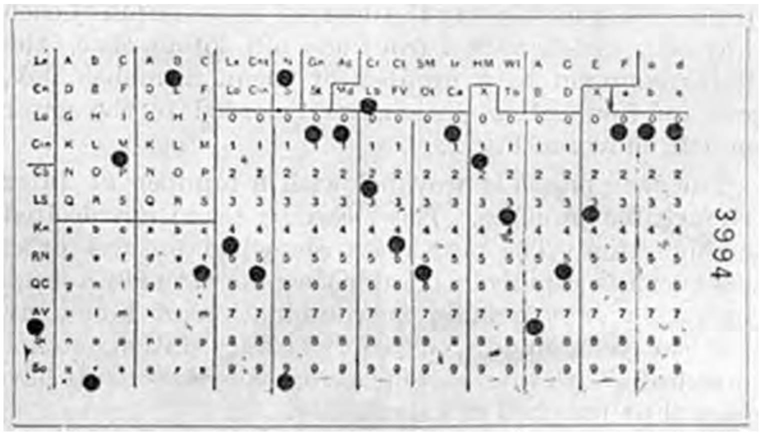


Figure 1.16 Herman Hollerith punched card as used in the 1890 US census. This card had only 22 columns and used round holes (IBM cards later were 80 columns and used rectangular holes). (Source: Courtesy of International Business Machines Corporation, ©International Business Machines Corporation.)

These machines in the early 1940s were influential and important but programming them was a challenge. Many were not really general purpose and programs were hardly a separate concept until machines began to be built with a common concept of computer design in the late 1940s. This common concept separated out the program as a separate component that was stored in memory and fed to a computational unit.

1.4.2 The von Neumann Architecture

John von Neumann produced a report called the *First Draft of a Report on the EDVAC* in 1945 that established what is now called the "von Neumann Architecture." This heavily influenced the design of later computers, in particular Wilkes' Electronic Delay Storage Automatic Calculator (EDSAC) at Cambridge University, Standards Eastern Automatic Computer (SEAC), and Standards Western Automatic Computer (SWAC) at the US National Bureau of Standards, Manchester "Baby" (followed by the Mark 1) at Manchester University, Turing's Pilot ACE, Binary Automatic Computer (BINAC), and the UNIVersal Automatic Computer I (UNIVAC I). This report included the notion of a stored-program concept. In this report (see von Neumann [1945]), he defines three primary components:

- *Arithmetic unit*: This unit would be responsible for executing arithmetic operations and was called the "central arithmetical" (CA) unit. This essentially corresponds to what became an arithmetic logic unit and eventually part of a central processing unit.

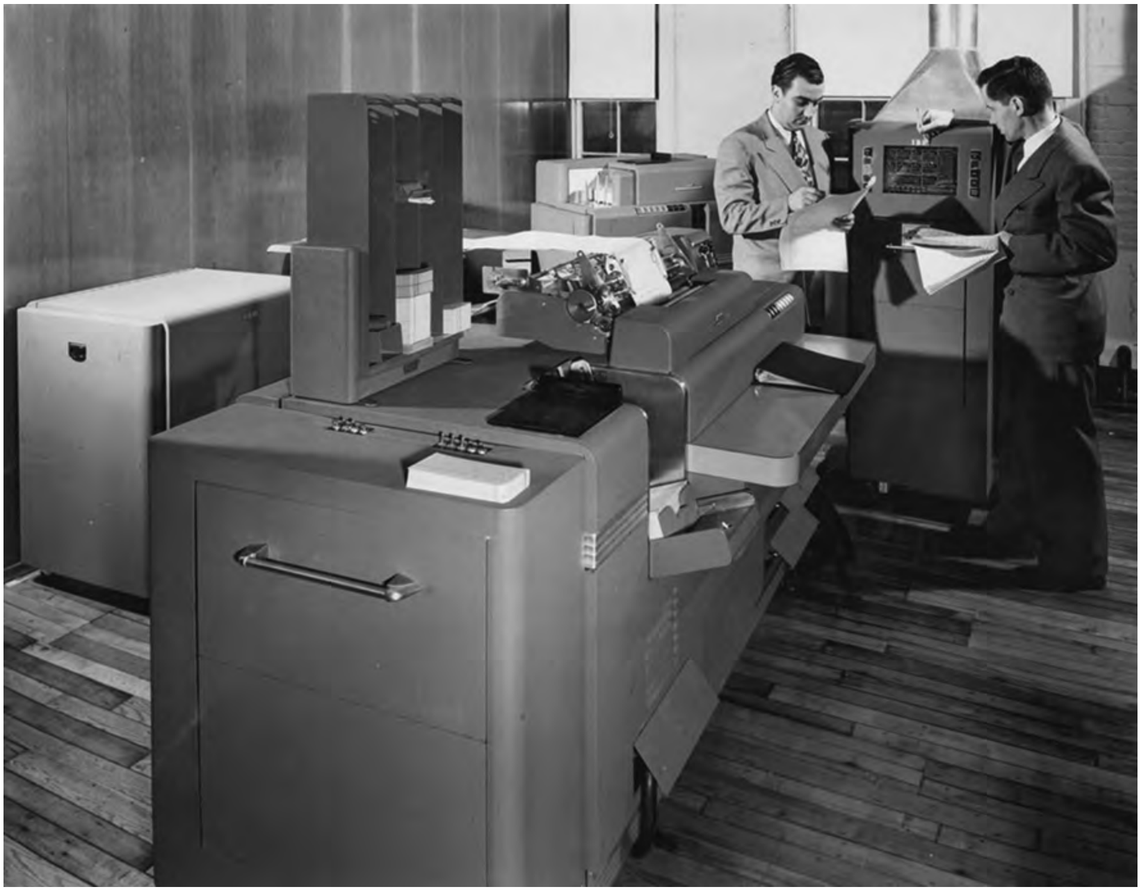


Figure 1.17 The IBM Card-Programmed Electronic Calculator was the first digital computer used by the US space program. (Source: Courtesy of International Business Machines Corporation, ©International Business Machines Corporation.)

- Central control (CC): This is the ability to store a program and to control what gets sent to the CA unit as well as input and output.
- Memory (M): The report notes that "considerable memory" will be needed.

The components, CA, CC, and M, were then used to define a logical structure to work together. Many of these ideas had been known and theorized (such as in Babbage's Analytical Engine) but this draft report synthesized the ideas into a logical framework (von Neumann was a mathematician) that then heavily influenced many later machines.

Computers beginning in the late 1940s more clearly separated the program from the machine. This was a critical turning point for computers becoming capable

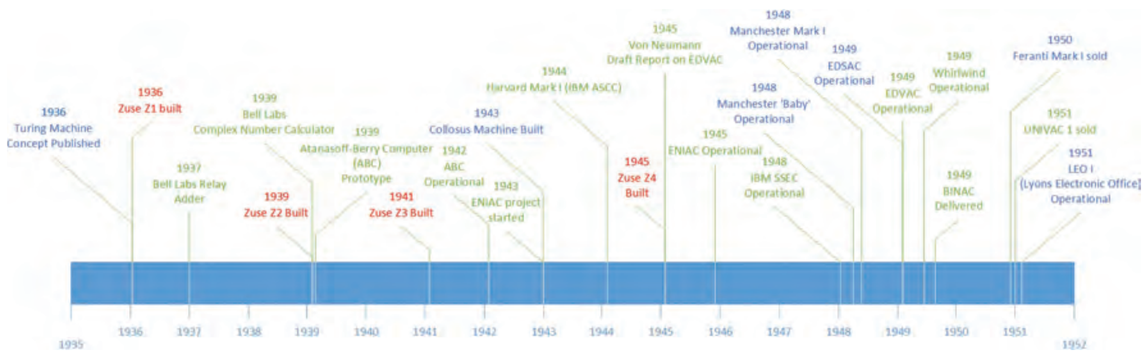


Figure 1.18 Early computer timeline showing selected major projects.

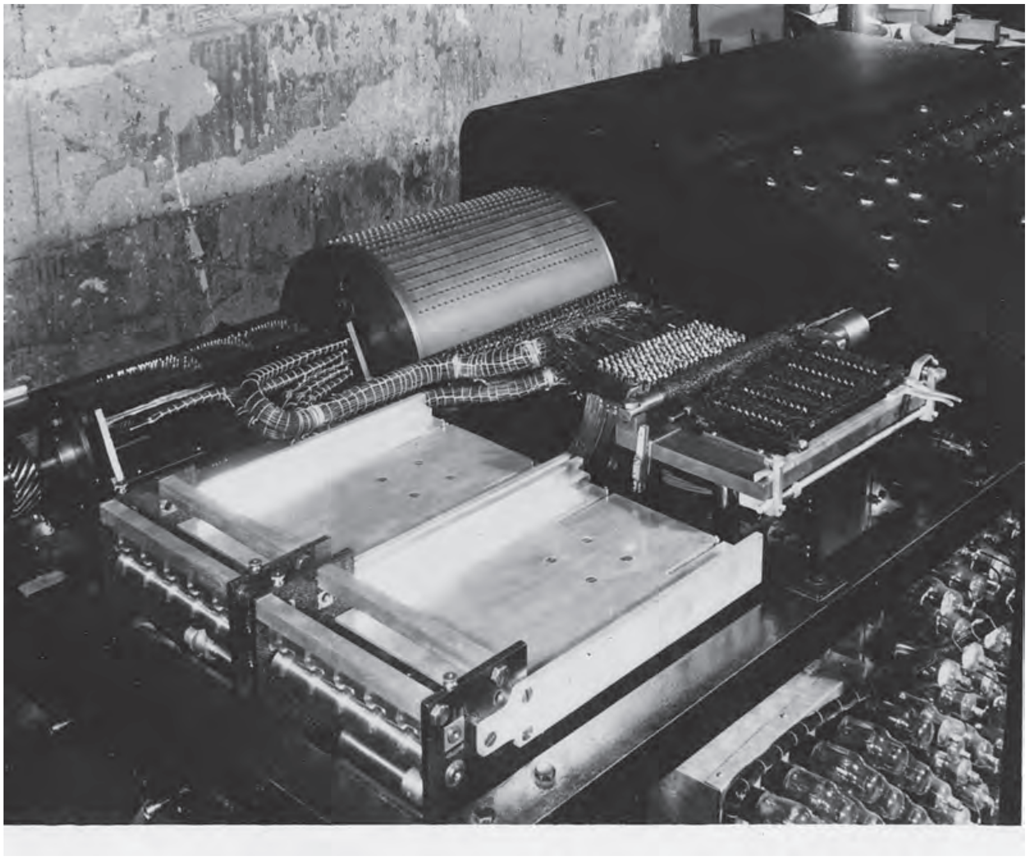


Figure 1.19 The Atanasoff–Berry Computer (ABC) with a drum memory (circa 1942). (Source: Iowa State Univ. Library Special Collections and Univ. Archives.)

Figure 1.20 Using IBM punched card equipment with the ENIAC (1946). Betty Jennings (left) and Frances Bilas (right). (Source: Photo from Hagley Library, photo courtesy of Unisys Corp.)

of being fully programmable and separating the notion of program from the hardware. Machines such as the BINAC, Pilot ACE, Manchester Baby, Cambridge EDSAC, and Eckert and Mauchly's BINAC[26] and UNIVAC were built on the von Neumann model. Even the ENIAC was retrofitted to a form of this architecture to be more easily programmed.

1.4.3 Computers After the von Neumann Architecture

The Manchester Small-Scale Experimental Machine, nicknamed "Baby" in 1948, was the first computer that used the von Neumann architecture. This was followed by the Manchester Mark 1 in 1949. The EDSAC was built at the University of Cambridge in 1949, again using the von Neumann architecture.

26. The BINAC is often cited as the first commercially available computer, though only one was sold to Northrup Aircraft Company in 1949.

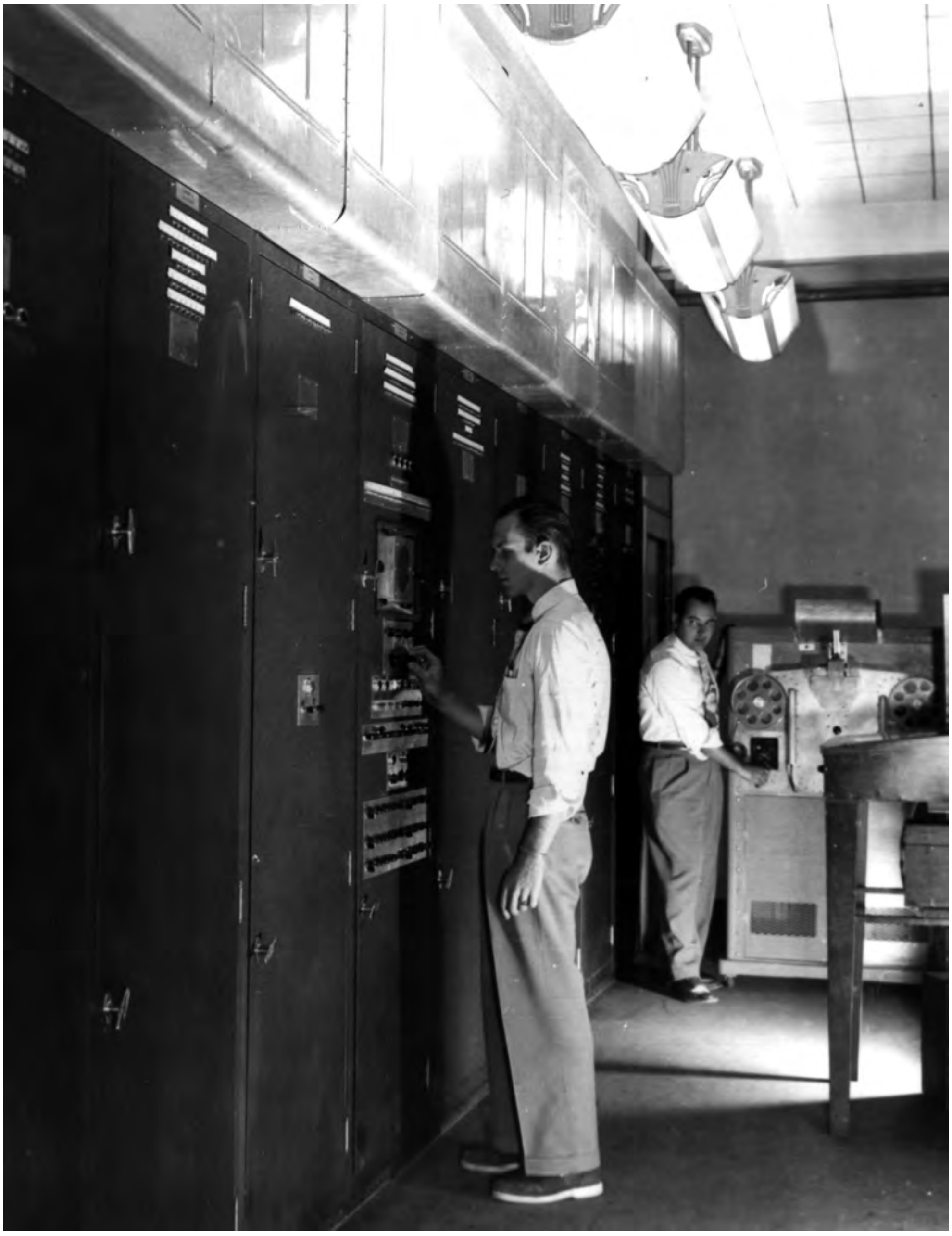


Figure 1.21 The EDVAC as installed in the US Army Ballistic Research Laboratory (BRL). Richard Bianco at the paper tape; unknown man at console. (Source: US Army Photo from the archives of the ARL Technical Library.)

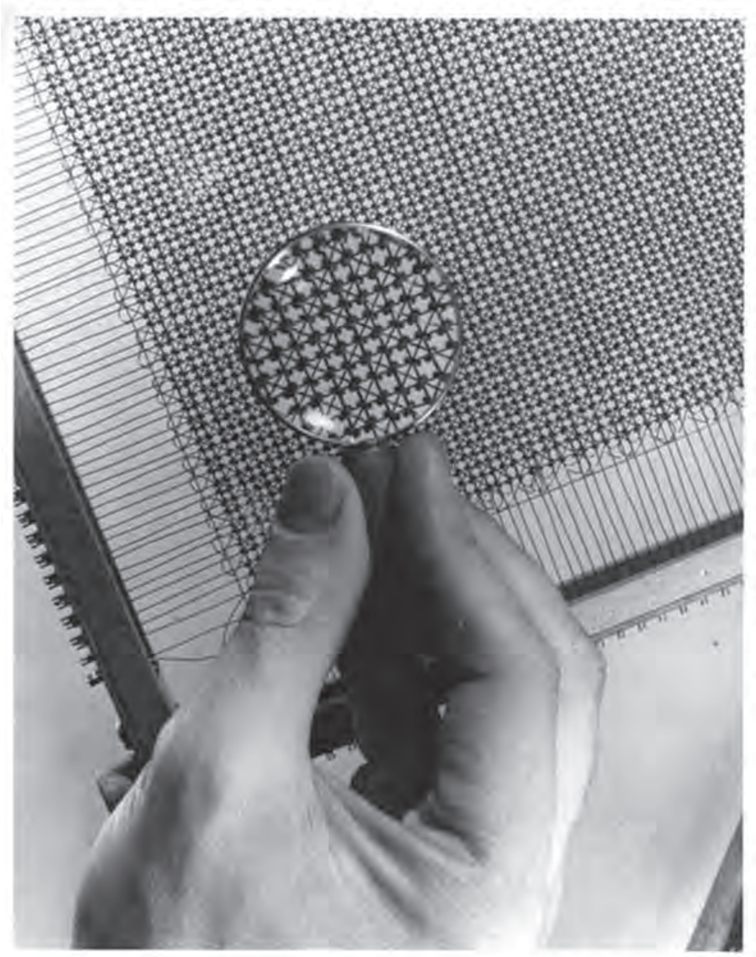


Figure 1.22 A closer view of Project Whirlwind's core memory. (Source: Courtesy MIT Museum.)

See Figure 1.23 for a high-level description of the changes in computer hardware since the EDSAC and EDVAC (see Figure 1.21) and the commercialization of computers. This description focuses mostly on "computer classes" as described in Bell [2008]. Commercialization of computers was initiated by sales of the UNIVAC, which was followed by a number of companies entering the market including IBM. Most of these manufacturers were building large machines intended for large companies and government installations. Through the 1950s and 1960s, the US mainframe market was divided between IBM and primarily seven other companies that became known as the "seven dwarfs." These "seven dwarfs" were UNIVAC, Control Data Corporation (CDC), National Cash Register (NCR), General Electric (GE), Honeywell, Burroughs, and Radio Corporation of America (RCA). Other companies

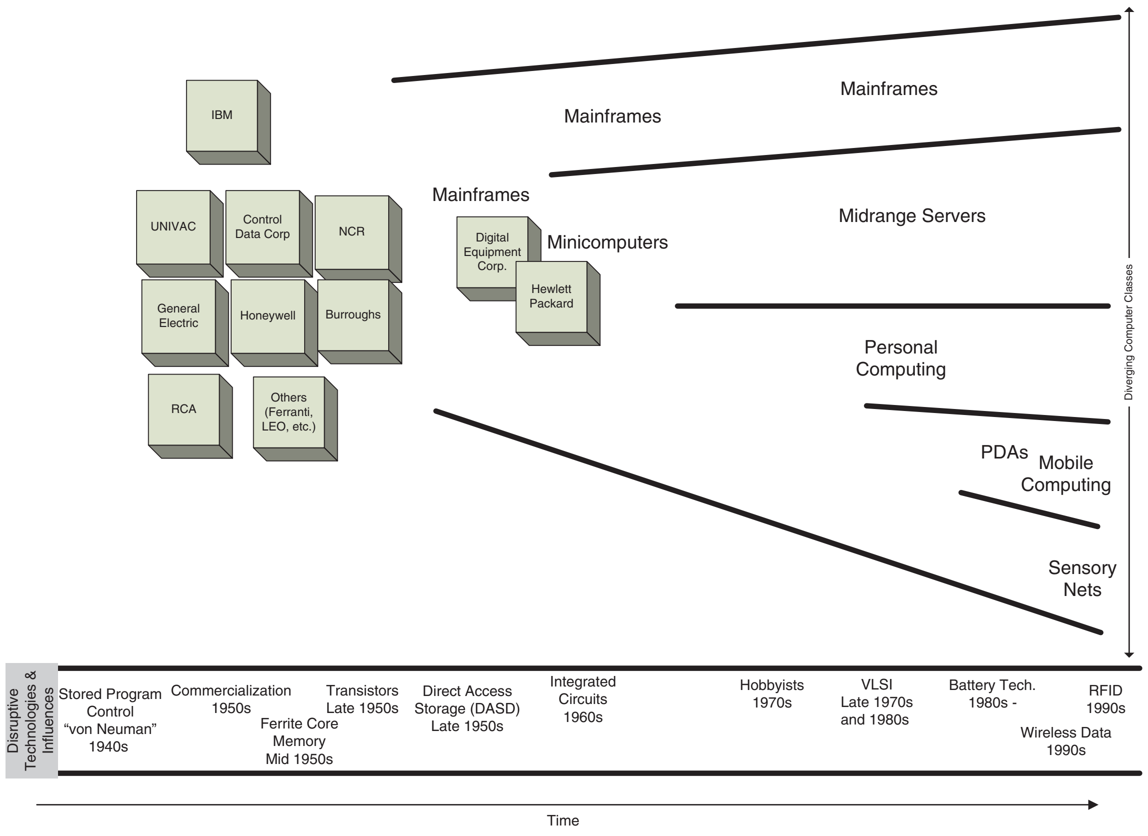


Figure 1.23 Disruptive technologies after the commercialization of computers stimulated division into different classes of computers.

also competed in the mainframe market such as Philips and Ferranti, among many others. As new classes of computers were developed, new competitors entered the market. An example of this is Digital Equipment Corporation and Hewlett Packard that entered the market primarily with minicomputers, along with many other companies. Similarly, many new companies entered the market with the personal computer (PC) class of computers, including Apple, Commodore, Compaq, Dell, and Gateway. At the bottom of Figure 1.23 is a set of events and changes that helped push forward to the next phase. These are as follows:

- *Stored program concept (1945)*. As described earlier, this helped usher forth a set of computers that clearly separated the program from the machine. This concept helped to make machines easier to program and was adopted by commercial computer companies. The ENIAC was also retrofitted to adhere more closely to this model.

- *Early commercialization (1947–1951)*. The intent of the formation of the Eckert–Mauchly Computer Company, who then called their computer UNIVAC (Universal Automatic Computer), was to build commercial computers that could be sold to government and businesses. The first UNIVAC was installed in 1951 for the US Census Bureau by Remington Rand corporation. The Eckert–Mauchly Computer Company also produced the BINAC, which was delivered to Northrop Aircraft Company in 1949 (and is considered the first commercial computer sold). The BINAC was not a commercial success, but the UNIVAC I was a commercial success and eventually sold 46 units. As noted above, IBM and the "seven dwarfs" competed to serve the emerging mainframe market. Commercialization also occurred early in Great Britain with the Ferranti line based on Manchester University's machines and LEO (Lyons Electronic Office) based on Cambridge University's machines. Other early British computer companies included the British Tabulating Company LTD and the Elliott Brothers. See Lavington [1980] for more on early British computing.
- *Magnetic-core (ferrite) memory and transistors*. Magnetic-core memory (sometimes called "ferrite core" or just "core") was developed and first successfully deployed in the Massachusetts Institute of Technology's Whirlwind computer.[27] See Figure 1.22 that shows how each memory element was woven into the core. The transistor was created in 1947 at Bell Telephone Laboratories. Both of these inventions became critical to the expansion of a computer's power as well as significantly reducing its size. These both became widely used technologies in the mid-1950s and spurred the creation of affordable computers and widespread implementation in the late 1950s. This period of the late 1950s is when compilers began to appear as well as operating systems.
- *Direct access storage and integrated circuits*.[28] The creation of affordable disk drives and integrated circuits led to another dramatic increase in power as

27. Note that a patent for core memory was first filed in 1947 by amateur inventor Frederick Viehe. An Wang and Way-Dong Woo (at Harvard University) filed a core memory patent in 1949. RCA (1950) and MIT (1951) filed additional patents for core memory. MIT's Whirlwind project was the first to demonstrate its performance in a computer with a 32 by 32 "plane" of memory in August 1953. The Whirlwind project had a separate computer called the Memory Test Computer (MTC) to demonstrate and refine core memory.

28. Note that direct access storage was often called direct access storage devices (DASD), particularly in the context of IBM mainframe storage devices and included not only disk drives but also drums and other storage devices where information could be retrieved directly without reading through other information, such as in magnetic and paper tape. The first disk drive was the IBM

well as the ability to reduce usable computers to a size and cost that could be accommodated by a department or workgroup. A distinct new class of computers (called minicomputers) was created as a result. Companies such as Digital Equipment Corporation (as well as many others) created minicomputers, such as the PDP and VAX series. The PDP-8 was introduced in 1965 and is considered to be one of the first commercially successful minicomputers.

- *Hobbyists*. Hobbyists were highly influential in the early days of PCs in the mid- to late-1970s. Parts and components were cheap enough that an electronics hobbyist could afford to buy the parts and build their own computer. The development of microprocessors that were affordable (such as the MOS Technology 6502, Zilog Z80, and the Intel 8080) were key to making it affordable. Kits were developed such as by Heath Company (Heathkit brand) and Micro Instrumentation and Telemetry Systems (MITS) famously[29] created the Altair 8800 (based on Intel 8008 CPU and used the S-100 bus) both as a kit and an assembled unit.
- *VLSI*. Very large-scale integrated (VLSI) circuits were another level of miniaturization of the technology and again dramatically reduced costs. This led to other classes of computers such as personal digital assistants (PDAs), sensory networks, and smartphones.
- *Battery technology*. Battery technology has become cheap enough, powerful enough, small enough, and long-lasting enough to make mobile devices such as smartphones, tablets, and PDAs affordable and usable. The trends in battery technology are well documented in an article by Koomey et al. [2010], which has become known as "Koomey's Law." Koomey expressed the trend as "at a fixed computing load, the amount of battery you will need will fall by a factor of two every year and a half." This trend has been remarkably stable since the 1950s.
- *Wireless data*. The pervasiveness of wireless data networks were also a key factor in the widespread use of mobile computing devices.
- *RFID*. Radio frequency identification (RFID) has been an important technology for the deployment of sensors and sensory networks, though other

350 RAMAC disk storage unit announced in 1956, which would store 5 million 6-bit characters. RAMAC stands for "Random Access Method of Accounting and Control."

29. Note that it is for the MITS Altair (named "Altair" based on a planet from a *Star Trek* episode) that Micro-Soft (as Microsoft was named at that time, standing for "microcomputer software") created its first product, a BASIC compiler.

technologies (such as pervasive Wi-Fi and cellular data networks) are also being used for sensory networks.

1.5 Computer Hardware Trends and "Laws"

Bell's Law (after Gordon Bell while at Digital Equipment Corporation in 1972 [Bell 2008a]) states:

Roughly every decade a new, lower priced computer class forms based on a new programming platform, network, and interface resulting in new usage and the establishment of a new industry.

These classes of computers correspond to mainframes, minicomputers, PCs, PDAs, mobile devices, etc. Most of these classes of computers have continued but some of these classes became subsumed by others. Minicomputers have been supplanted by "midrange servers," though one could argue that this really is just a different name for the same class of computer. PDAs have almost entirely died as a class of computers, with mobile devices such as smartphones and tablets supplanting PDAs (and perhaps even PCs).

Bell argues that the creation of each of these new classes creates its own new industry with new programming and software environments. This has largely held true. One of the reasons for this is that a truly new class of computers allows an opportunity to introduce a new or better programming environment. The introduction of smartphones is a good case in point where it allowed the introduction of Apple's iOS, Google's Android, and RIM's Blackberry operating systems into the market, along with their programming environments which were different from prior classes of computers. New classes of computers have allowed a largely greenfield approach that is less burdened by the mass of previously developed software for other preexisting classes of computers. A good example of this effect was the creation of the PC class of computers. With PCs, new operating systems were developed such as CP/M and PC-DOS, as well as new programming language compilers (such as the BASIC compiler for the MITS Altair and popularization of the Pascal programming language with UCSD Pascal, Turbo Pascal and Macintosh Pascal). Even completely new types of tools such as spreadsheets (VisiCalc) were developed that took advantage of the special features of a PC-type machine (personalized and dedicated).

Over time, computing hardware has experienced an exponential increase in power while also seeing much cheaper prices. This fact alone has stimulated a lot of the churn in replacing systems as we are often able to deploy new systems that are not only more powerful, but cheaper—often to the extent of being able to replace them at less than the maintenance cost of the current system. This is the primary *observation*[30] that Gordon Moore made in 1965 (see Moore [1965]) in the form of integrated circuits:[31] It is amazing that this observation has held largely true since Gordon Moore made it in 1965, but technology is rapidly approaching physical limits where this trend may no longer apply.

the number of transistors per chip will double every two years.

This observation has also held true in the size and cost of memory as well as in disk storage cost. See Figure 1.24 for how this has applied to the number of transistors in microprocessors. Note that the transistor count scale is a logarithmic scale, making the exponential increase appear linear in Figure 1.24.

The impact on software of Moore's Law has been dramatic. The constant changes in computing devices has driven economic and expediency incentives to *reuse software* and to have software that is portable to a wide variety of computers. Additionally, the increased use of computers has often led to a shortage of time or programmers to create all the needed programs. The increased computing power has helped enable the creation of a number of abstract layers as performance became less of a constant concern. New computer lines were created with the ability to reuse software as a major design requirement. A famous example is the development of the IBM System/360 mainframe series of computers where binary compatibility was a design requirement. Later generations of IBM mainframes kept this ability to run old software; this was a major selling point. One of the factors in the design of MULTICS[32] and UNIX operating systems was the design for *portability of code*. This portability comes at some performance cost and the fact that computers were rapidly becoming more powerful made this cost easy to absorb. Moore's Law is also seen to have a negative effect on software in the creation of what has been disparagingly called *bloatware*. The argument is: because there is little incentive to be efficient, why not develop software in the quickest manner

30. While this is called a "law," it really is just an observation that fits the trend. Most of these "laws" are just observations.

31. Note that this is sometimes stated as doubling every 18 months, but that was attributed to another Intel executive.

32. MULTICS is an acronym for Multiplexed Information and Computing Service. UNIX is not an acronym but more of a pun on the MULTICS name. Portability was not a strong design factor for MULTICS but was a factor in choosing to use PL/I for most of the coding to allow more of the system to be independent from the hardware. At the same time, MULTICS was also using several hardware features not generally available, making it difficult to port.

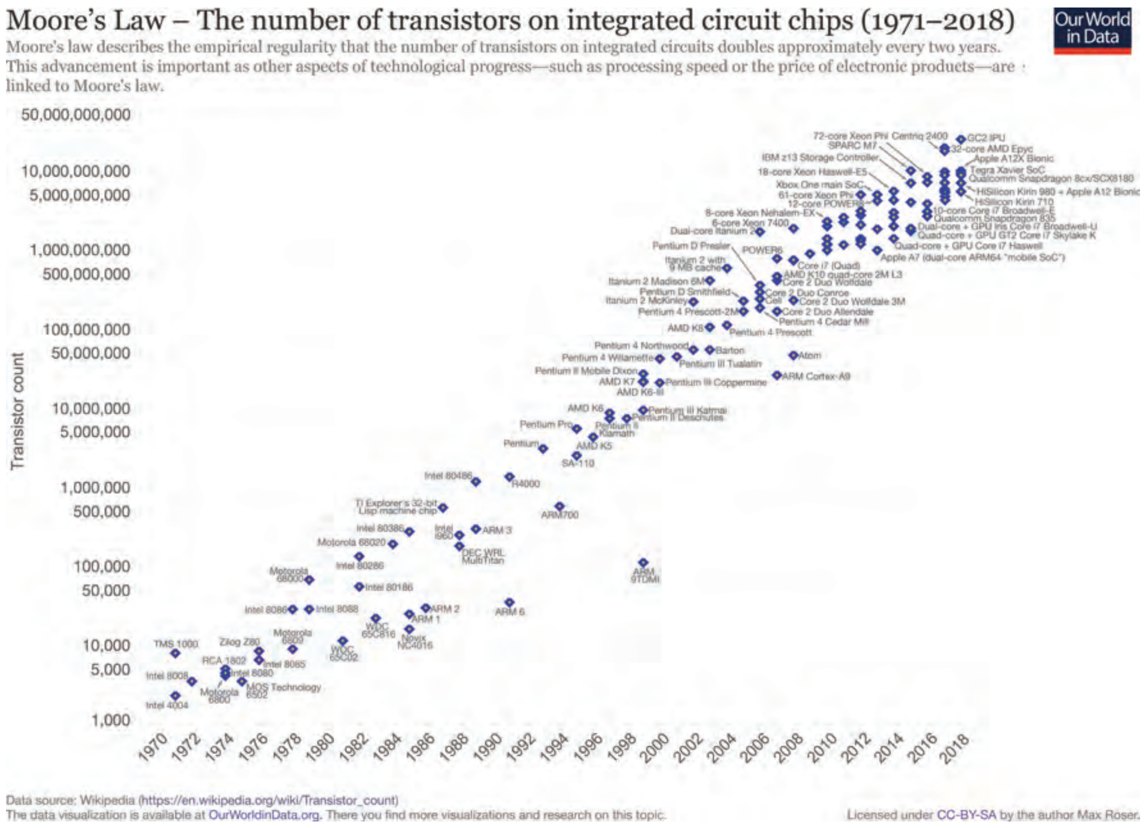


Figure 1.24 MOSFET transistor counts for microprocessors against dates of introduction from 1971 to 2018. (Source: https://ourworldindata.org/uploads/2019/05/Transistor-Count-over-time-to-2018.png licensed under CC-BY-SA by the author Max Rosen.)

and in effect waste processing and memory capacity? An example of this has been systems such as Microsoft Windows, where the current estimate is roughly 50 million lines of code in Windows 10 (according to https://code.org/loc and Microsoft's Facebook Windows page at https://www.facebook.com/windows/ in 2019). While lots of new functionality has been introduced to Windows, some would argue that the operating system is overly complex, leading to bugs as well as using more of the computer's processing power than it really needs to. Apple's iTunes has also been similarly criticized.[33]

33. See http://gizmodo.com/5335754/itunes-9-will-be-a-bloated-social-monster for an example of Apple iTunes 9 being criticized as "bloated."

1.6 Lessons Learned from Hardware Evolution Affecting Software

The evolution and changing of hardware has directly affected software in a number of ways. Some of these lessons that continue to affect how we develop software include:

- *Hardware advances have let software grow*. As Moore's Law has continued, software has been able to grow without having to be as concerned with performance or size. With early computers, the constraints of system performance and memory size restricted the problems that could be solved and how they were solved. If this hardware growth in performance and memory capacity do not continue to grow at the same rate, software may be forced to be more efficient and live within memory constraints in order to solve ever larger problems.
- *Hardware implementations of software have more limited lifespans*. There have been many examples of implementations of software in hardware and most of these have had very limited lifespans. Examples include the development of database machines and artificial intelligence machines that were designed to implement these systems in hardware. While databases and AI continue to be developed, these sorts of specialized computing devices come and go, often driven by a desire for better system performance.
- *Hardware stimulates software development*. New hardware devices and new hardware capabilities encourage the development of additional software to utilize those features. Besides just faster computing, the development of additional hardware capabilities (like graphical displays and sensors) and devices (like mice and network interfaces) have required software to leverage those new capabilities.
- *Abstracting software from the computational engine stimulates software development*. The ability to write software that isn't heavily dependent on a particular computer makes that software likely more valuable and useful. This makes it more worthwhile to write software that may be used by more people and companies. This abstraction of software has also made it more possible for that software not to need to change every time the hardware changes as well as allowing higher levels of software abstraction to be built on top of it.

1.7 Summary

This chapter introduces a definition of software that includes not only the program but the concepts and methods to build it. The definition assumes software is part of a running system and not just a theoretical construct. This chapter defined the terms and structures that will be used in Chapter 2 to build a specific model for software history and evolution.

This chapter gives a very brief outline of computer hardware history, focused on the computer itself. Specifics of other hardware (such as networking) will be included in other chapters as needed. This chapter also focused on early computer history. Other computers will be brought in as appropriate to later chapters.

Some of the key points about how software has been impacted by computing history are:

- The von Neumann architecture allowed for computers to be more easily programmed and separated the program from the machine. This eventually allowed for layers of software to be developed and software to become its own separate concept.
- New computer classes generate an opportunity for new programming environments and software. As an example, with the creation of the PC class, Microsoft created a BASIC compiler for the MITS Altair, and in winning a deal with IBM to produce the BASIC compiler (and the operating system) for the IBM PC, helped to create an industry for PC software. In Microsoft's case, they created new programming environments, leveraging the new PC class. Another example is the introduction of the Android and iOS operating systems for the mobile computing class. New computer classes are able to create new markets with much less of an embedded software base, and as a result, new ways of programming and new software systems can be adopted more quickly.
- Many efforts have been less influential (Zuse, Babbage, ABC, etc.) than they could have been, and in some cases, should have been. For example, Zuse developed a sophisticated programming language, Plankalkül, but it was not published for decades, losing the influence it could have had over the development of programming languages. Babbage's Analytical Engine had many interesting ideas that were similar to what was rediscovered in the development of programmable, electronic computers. Work on the ABC was dropped before it could be very influential.
- The speed of computation continues to increase while its unit cost decreases. This has led to the ability to create much more complex software systems (and some would argue *bloatware*). Additionally, this ability to do more with less cost has led software to new areas of application, often with the introduction of new classes of computation. For example, the creation of mobile apps

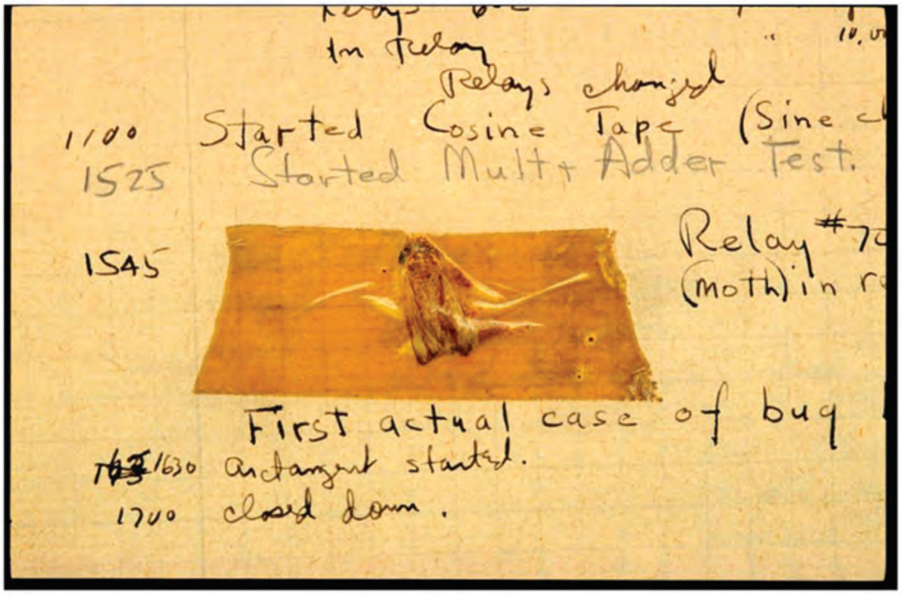
is an example of where having a new computing class allows for additional areas of application (using features characteristic of that class: e.g., mobility, always connected, and personalization). Another impact of the expansion of the use of software is the continued drive to reuse software. This drive to reuse is partially because there has often not been enough programmers to write all the programs needed at that time. So, drives to increase productivity for writing new software as well as to increase the usability of existing software through reuse and portability have been recurrent since the late 1950s. This has led to terms such as "software crisis" being coined to reflect the bottleneck that writing software is often perceived as being.

Lastly, a few models were introduced to begin to look at the larger scope of computer evolution, such as Bell's Law and Moore's Law.

1.8 Exercises and Projects

1.8.1 Exercises

1. Why does the definition of software in this text include "related methods and tools?" What does knowing about the methods and tools used to develop software tell us about the software?
2. Give an example of a specific physical phenomenon that software depends on in order to run. Can a different physical phenomenon be used? If so, give another example phenomenon. If not, explain why that's the only physical phenomenon that can be used.
3. A software engineering project might include both standard and non-standard engineering components. Give an example of a software engineering project where this would be appropriate.
4. Give an example of two kinds of software that have a technical co-evolution relationship. Explain how they have co-evolved.
5. Williams–Kilburn (or sometimes just called Williams) tubes were used for early memory storage, such as in the Manchester Baby. Explain what memory technology replaced them and why.
6. The notion of a program is different from that of software as defined here. Explain how a program can be machine dependent while the definition of software defines it as distinct from the machine.
7. The term "bug" is often ascribed to Grace Hopper having found a moth in a relay in the Harvard Mark series machines and pasting it in the logbook.



in relay

Relays changed

1100 Started Cosine Tape (Sine c

1525 Started Mult+ Adder Test.

1545 Relay #70 (moth) in r

First actual case of bug

~~143~~ 1630 Antangent started.

1700 closed down.

Figure 1.25 Bug entry in Harvard Mark II logbook held at the Smithsonian. (Source: Courtesy of Division of Medicine and Science, National Museum of American History, Smithsonian Institution.)

Find a paper or resource that refutes this origin of the term "bug" and explain the origin from your source's point of view. Hint: the logbook is at the Smithsonian National Museum of American History (see Figure 1.25).

8. Investigate the System Development Corporation (SDC; founded in 1955 in Santa Monica), which is considered one of the first software development companies. Find and explain at least one innovation that is attributed to SDC.

9. The IBM 704 computer has an interesting place in software history as being the machine on which FORTRAN and LISP were both developed. Explore why the IBM 704 had such an influence on software. What are the characteristics of the IBM 704's hardware as well as the market position that made it more likely to be the system that many used to develop innovative software systems in the mid-1950s? See Figure 1.26.

10. The Jacquard loom had a notion of "programmability." Find an example of the "program" for a Jacquard loom and document it. In your documentation explain the purpose of the program and each element of the cards used to implement the "program." An example of a mid-19th century woven

Figure 1.26 The IBM 704 is where FORTRAN, LISP, and the SHARE Operating System were developed (circa 1954). (Source: Courtesy of International Business Machines Corporation, ©International Business Machines Corporation.)

picture of Joseph-Marie Jacquard along with a picture of a Jacquard loom using punched cards is in Figure 1.27.

11. Konrad Zuse developed a language called Plankalkül (calculus of programs) at the same time as his early computers (Z1, Z2, Z3, and Z4) between 1942 and 1945. Plankalkül was designed but not implemented at that time; it was implemented much later. Find an example program in this language and explain how it works. See http://zuse.zib.de/ for an archive of his papers and examples of Plankalkül.

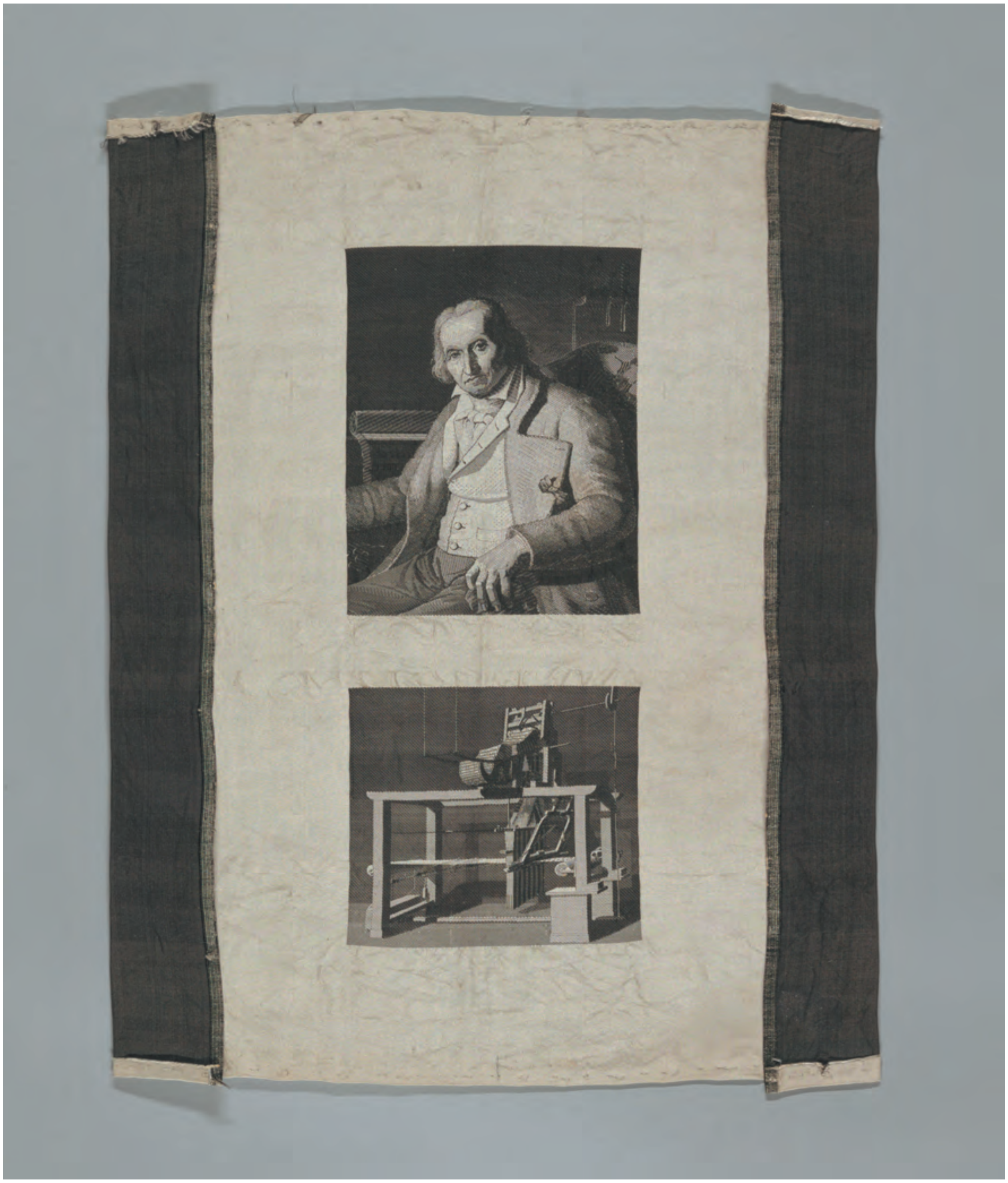


Figure 1.27 The Jacquard loom-produced silk woven picture of Jacquard and a loom using punch-card mechanism. (Source: Courtesy of Smithsonian Institution, bequest of Richard Cranch Greenleaf in memory of his mother, Adeline Emma Greenleaf.)

12. Compare the instruction set of the Burroughs 5000 with that of the IBM 704. Note the similarities and differences. Explain why such differences existed in these machine languages.

13. Ada, Countess of Lovelace, is often noted as the "first programmer," having worked with Charles Babbage on the Analytical Engine. In particular, she developed ways to use the Analytical Engine that used looping. Find an actual example in her writings where she noted this concept. Argue whether or not what she created was *software* by the definition given in this chapter. An excellent reference that explores Ada's contributions is Hammerman and Russell [2015].

14. The ENIAC used a method of patch cords to change the configuration of the "program" it was to run. Find an example of how these plugs were configured and explain how it implemented a particular function. Hint: There is a Java-based simulator for ENIAC and its operation is well-described in Haigh et al. [2016]. Explain how this configuration of the ENIAC is different from a modern program that would accomplish the same task.

15. Look up the US Patent number US2293127A (Computing Device). This patent was filed in 1937 by Howard Fishack, Loren Miller, and John Shively. Find out what other important patents this patent influenced and speculate why this patent was referenced.

16. Investigate whether Koomey's Law (for battery technology) is still holding for current battery technology. Are battery technologists worried about an end to Koomey's Law or do they see it lasting another decade or more? Create a graph showing Koomey's Law and project it into the future based on what battery technologists are predicting.

17. The ABC was credited to be the first electronic digital computer via the resolution of the *Honeywell v. Sperry Rand* lawsuit in 1973 and Atanasoff was declared the inventor of the computer. The ABC, however, was a special purpose computer that wasn't really programmable and not Turing-complete. The ABC did demonstrate a number of critical ideas such as using binary digits, using electronics to perform calculations, and having separated computation from memory. The ABC also had an arithmetic logic unit that was completely electronic. Determine the entire class of problems that can be solved using the ABC. Formally define what problems it can and cannot solve. Iowa State University has archives and documentation on the ABC. A good place to start is the website https://jva.cs.iastate.edu/history.php. The overall design of the ABC is in Figure 1.28 and a photo of Clifford Berry at the original ABC is in Figure 1.29.

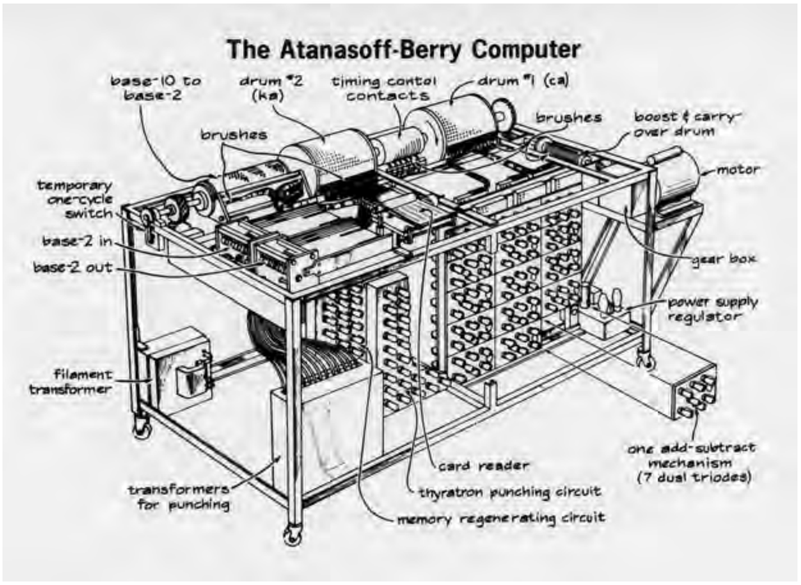


Figure 1.28 The ABC's overall design. (Source: Iowa State Univ. Library Special Collections and Univ. Archives.)

Figure 1.29 Clifford Berry with the ABC. (Source: Iowa State Univ. Library Special Collections and Univ. Archives.)

18. Bell's Law [Bell 2008b] predicted a new class of computing roughly every decade. Looking at recent types of computer classes, show whether new classes are still occurring every decade or significantly more or less frequently than every decade. Explain with examples and a timeline to justify your answer.

19. Corbató's[34] Law states that "The number of lines of code a programmer can write in a fixed period of time is the same, independent of the language used." Find any proof (research or formal studies) that indicate whether this was true or false. Is the "law" largely true or false? What was the motivation behind this law?
20. Bill Joy (co-founder of Sun Microsystems) formulated that the peak computer speed roughly doubles each year and thereby can be determined by a function of time. He expressed it as: $S = 2^{Y-1984}$ (where Y is the year). This formula has since become known as Joy's Law. Compare Joy's Law to Moore's Law and determine which has been more accurate since 2010.
21. Some have predicted that the traditional PC is being replaced by mobile devices such as smartphones and tablets. Show data for how the sales of desktop and laptop PCs are changing over time. Can you support the argument that they are being replaced by mobile devices? Why or why not?
22. Look up the US Patent number US2192612A (Multiplying Machine). This patent was filed in 1937 by IBM and refers to multiplying numbers using binary notation. Did this patent influence others to use binary? Is it referenced as prior art in later binary machines?
23. The Harvard Mark I used a method of plug boards and paper tape in order to be programmed. Grace Hopper was involved in programming this machine and produced plug diagrams as in Figure 1.30. This plug diagram is for Problem "L," generating Bessel function tables. Determine what this problem was trying to solve and describe what was the use of the problem's solution.
24. One of the key early decisions in computing hardware was the separation of the CPU from the memory (or the *mill* from the *store* in the Babbage Analytical Engine), with the memory storing instructions for the program and results. Explain why that made it easier to create computers that were more easily programmed for a wide variety of tasks. Be sure to compare it to a design that does not make this separation.
25. One argument is that because of Moore's Law being largely true the last few decades software *has not had* to evolve to become fundamentally more efficient. Give an example of a particular software system that would have had to have been more efficient if the hardware had not sped up nearly as quickly

34. Fernando Corbató at MIT and Project MAC will be mentioned again in connection with the CTSS and MULTICS operating systems.

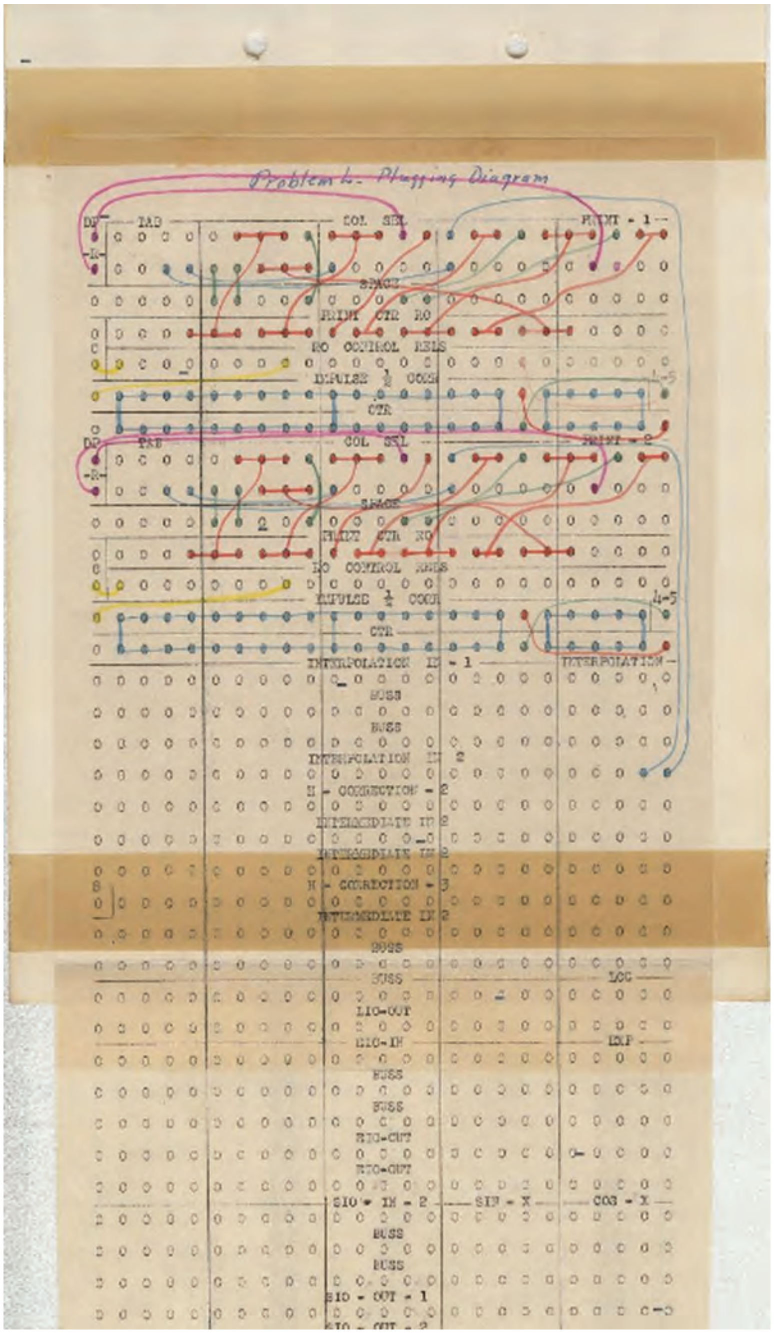


Figure 1.30 Grace Hopper's plug diagram for the Harvard Mark I, for Problem "L." (Source: Courtesy of Grace Murray Hopper Collection, Archives Center, National Museum of American History, Smithsonian Institution.)

as it did. That is, the example software system as designed would not have worked well or not worked at all because of the unavailability at the time of cheap, fast hardware. Explain why.

26. Consider what would happen if we had a large-scale move to a fundamentally different computing mechanism, such as quantum computing. By “large scale,” I mean that any new, meaningful software development or deployment will likely use the new technology. What would this do to software as a whole? Would you expect this to dramatically change most existing software or to have a marginal effect on most software? Explain how you would expect the existing software base to change and how future software might evolve differently.

27. Binary representations of integers and floating point took some time to standardize. One example of an alternative representation was the use of ones’ complement to represent negative integer numbers on UNIVAC 1100 computers. A side effect of using ones’ complement is that it is possible to have a “minus zero” representation. Explain how this could make programming such a machine more complex than using two’s complement.

1.8.2 Projects

1. Pick ten different computers from the late 1950s. Get the complete list of machine instructions for these and compare them. Identify unique instructions and explain why they were useful. Compare these to Intel’s Pentium microprocessor machine instruction set and explain how these old machine instructions were either subsumed, replaced, or no longer needed.

2. There have been many predictions that Moore’s Law is about to end. That is, processors’ transistor counts will not be able to continue to grow at the same rate as predicted by Moore’s Law. Investigate the factors that are limiting the growth of Moore’s Law and produce a report detailing the different mechanisms and techniques that may allow Moore’s Law to continue to predict growth in transistor density over the next few decades. Include in your report predictions on computing cost and whether that will continue to decline at the same rate. See resources such as Mack [2011], Schaller [1997], Mollick [2006], and Ceruzzi [2005].

3. Reliability for tube-based machines was a real challenge as tubes have a relatively high failure rate (compared to relays, transistors, and integrated circuits). As a result, many of these early tube-based machines were lucky to run for an entire day. Investigate the failure rate of tubes, transistors, ICs, and memory DIMMs. Based on this investigation, determine the probable

failure rates of machines using tubes, transistors, ICs, and VLSI for memory and processing. What is the point (i.e., how many elements) where computers based on each of these technologies becomes untenable (say, fails every 30 minutes or less without a way to recover) based on the current failure rates? Apply the same analysis to CPUs (i.e., How many CPUs will produce a machine with a failure every 30 minutes or sooner?).

4. Consider the tablet computer as a new "class" of computers and investigate how well that fits the model described by Bell [2008]. In particular, investigate the efforts to develop tablet computers such as the RAND tablet (1963), Xerox PARC's DynaBook [Kay 1972], the Linus Technologies Write-Top (1987), and the University of Illinois at Urbana-Champaign's winning entry in the PC of the Year 2000 competition (1987, sponsored by Apple Computer Corporation).[35] Given the long history of tablets, why have they become popular only recently? What other factors were involved in making them viable?
5. Investigate the origins of an accumulator. This usually manifests itself as a particular register that is identified as an *accumulator*. Track the history of accumulators through the earliest devices and into card tabulating accumulators (such as the IBM 514), and as a register of early computers. Explain how accumulators have changed in function over time and why they are not needed in the same way they once were in mechanical devices.
6. In Figure 1.31, Grace Hopper diagrams important computers and their influences on each other in a tree she presented in 1958. Compare her tree to those from more modern histories of computing noting those that are either not mentioned or rarely mentioned in these modern histories (such as Ceruzzi [2003], Ceruzzi and Haigh [2021], Mahoney [2011], and Williams [1997]). Computers that you may find are less referenced include those such as the BARK, RAYDAC, Elecom, among others on her diagram. Determine what contributions to computing these less referenced computers had and whether those contributions were critical. Also, note the computers that have since become more significant in more recent diagrams and speculate as to why she didn't include them in her diagram. This might include computers such as the Robinson machines and Colossus computers. Another detailed tree is at https://ftp.arl.army.mil/ftp/historic-computers/png/comp-tree.png where

35. See http://www.computerhistory.org/atchm/yesterdays-tomorrows-the-origins-of-the-tablet/ for a good starting article on tablets.

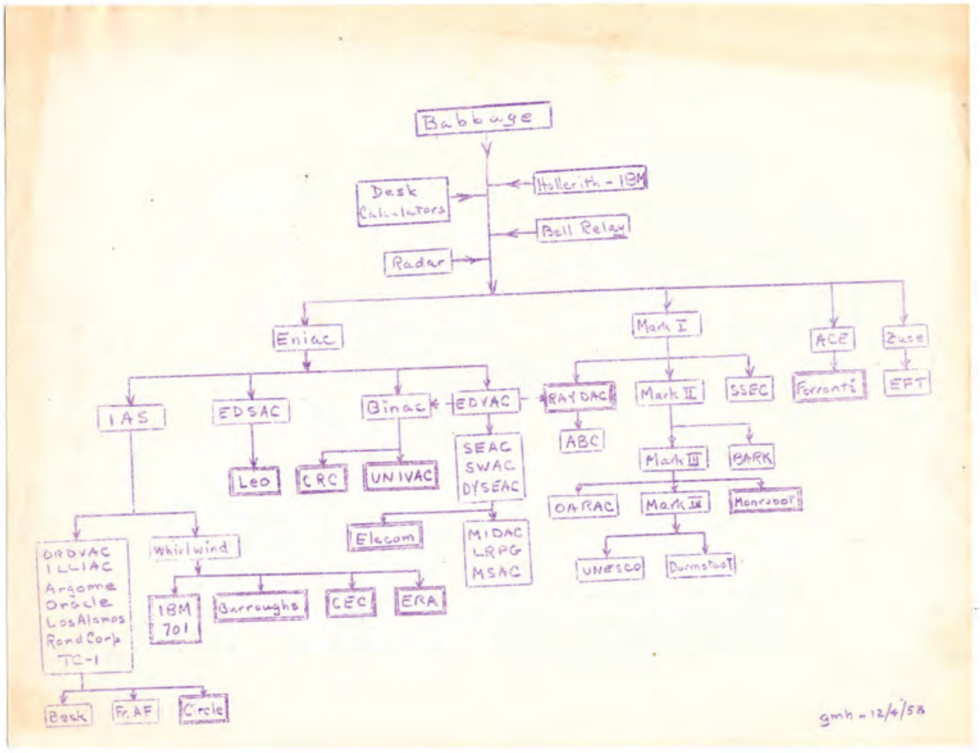


Figure 1.31 Grace Hopper's 1958 diagram showing computer influences. (Source: Courtesy of Grace Murray Hopper Collection, Archives Center, National Museum of American History, Smithsonian Institution.)

the decades are shown and influences of computers from the US Army's perspective until the mid-1960s.

7. Using only relays as switching devices, such as in the Bell Labs relay computers (see Figure 1.32), design and build a multiplier circuit that will take as input two 4-bit numbers and output the result. Refer to Shannon [1940] or other resources (there are several YouTube videos as well) on how to build logic circuits using relays and then assemble those gates into logic circuits to build the relay multiplier.

8. After Babbage's design of his Analytical Engine but before electronic computers were developed, there were several other efforts to design analytical computing devices such those by Ludgate, Torres, and Bush as described in Randell [1982]. Using Randell [1982], Haigh and Priestley [2016], Babbage

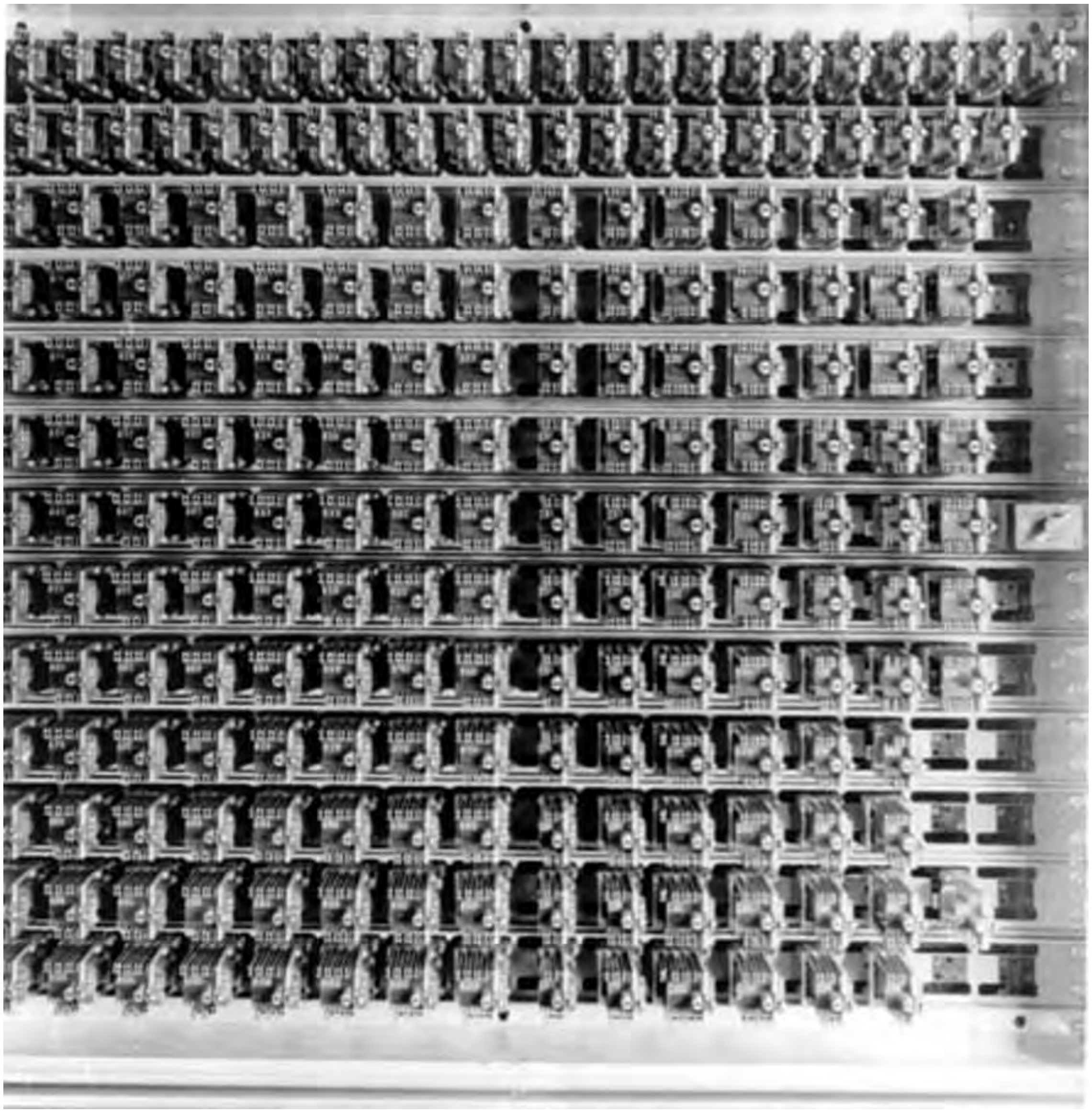


Figure 1.32 Stibitz's relay computer at Bell Labs (1940). (Source: Courtesy of AT&T Archives and History Center.)

[1864], and Babbage [1889] as starting points, detail how the concept of a program and programmable device evolved from Babbage's description of the Analytical Engine to digital devices such as the Colossus computers, ENIAC, and Manchester Mark 1. Explore the direct influence of Babbage's ideas and how those ideas influenced the concepts of program and programmability.

1.9 Further Readings and Online Resources

The history of technology, and in particular studies in how technology evolves, can be found in Constant [1980], Parayil [1999], and Arthur [2009]. A good book to understand the changes in programmer culture is Ensmenger [2010].

Computing hardware history has been studied and documented widely such as in Williams [1997], Mahoney [2011], Ceruzzi [1983, 2003] (updated as Ceruzzi and Haigh [2021]), Bruderer [2015], and Campbell-Kelly et al. [2013]. For more information on the ENIAC, see http://www.seas.upenn.edu/about-seas/eniac/ for the University of Pennsylvania's ENIAC site and Haigh et al. [2016]. Please see Hook et al. [2002] for a broad reference on the history of computing and telecommunications. See Lazou [1988] for descriptions of various supercomputers and techniques used to program them. The National Research Council [1966] provides a perspective into how universities affected computing and were beginning to use computing in the 1960s.

The evolution and growth of the software industry has been well documented in works such as Cortada [2012] and Campbell-Kelly [2003]. For details on Moore's Law see Mollick [2006] and Mack [2011].

There are many substantial history of computing resources available in archives and online, some of the more significant ones are as follows:

- *The Charles Babbage Institute (CBI)*. See http://www.cbi.umn.edu/ for their main site. The CBI houses a vast number of archives related to computing, as well as oral histories and online, digital collections. The CBI houses a large collection of resources in its physical archives of computing-related material including the papers of many pioneers of computing.
- *The Computer History Museum (CHM)*. See http://www.computerhistory.org/. CHM has a large museum in Mountain View, CA, as well as a large number of online resources including oral histories at: http://www.computerhistory.org/collections/oralhistories/. Affiliated with the CHM is the Software Preservation Group, which houses a number of important software systems at: http://www.softwarepreservation.org/projects/. The CHM also houses a physical archive of resources relating to computing history.
- *Bit Savers*. See http://www.bitsavers.org/ for a large repository of software and manuals. This repository is replicated at several mirror sites and contains complete scanned manuals as well as actual software.

- *The Smithsonian Institution*. The Smithsonian has a number of online resources available through http://www.smithsonian.com/ and their archives. They also house a number of oral histories. The Smithsonian archives contain the papers of several computing pioneers as well as physical artifacts related to computing.
- *Society for Industrial and Applied Mathematics (SIAM)*. SIAM has a number of oral histories focused on scientific computing and numerical analysis housed at http://history.siam.org/oralhistories.htm.
- *IEEE Global History Network*. The IEEE global history network houses a number of oral histories at: http://www.ieeeghn.org/wiki/index.php/Oral-History:IEEE_Oral_History_Collection. The IEEE Global History Network houses a number of other resources related to the history of computing, engineering, and technology.
- *UK National Archive for the History of Computing*. See http://www.chstm.manchester.ac.uk/research/nahc/ for this archive housed at the University of Manchester that holds a large number of items related to computing in the UK.

2

Software History Fundamentals

This chapter covers material about the nature of software history and how it can be difficult to capture and describe. This chapter also covers the general structure of software, types of software, and cultures and groups of software developers. Software has rapidly evolved to have a diverse breadth of types, and different types of software have different factors affecting their change over time.

For the earliest computing devices, such as the Harvard Mark I and the ENIAC, programming consisted of placing plugs in the right positions in order to compute the functions in the order you needed to solve a particular problem. The "software," which at this point wasn't even coined as a term, consisted of descriptions of how to configure the patch cables and how to configure the initial starting conditions. Even for later computers, there was no software at all that came with the computer—no compilers or operating system. You had to somehow key in machine instructions (in binary) either through a panel or read them in using a tape or cards. You might first input the command to read the tape through the panel and then read in the loader from a tape or cards. Then run the loader to read in your program, all in binary. So, your program was all in machine code, using the instructions available on that computer. Software at this stage consisted of paper forms encoding the instructions and the actual tapes. As we'll discuss in the coming chapters, assemblers, interpreters, and libraries of functions and procedures were developed in order to allow a level of programming using mnemonic codes. By the late 1950s, compilers and operating systems were being developed and deployed, allowing a level of programming and software more like we see today.

2.1 Overview of Software History

Software systems can be very difficult to understand. The obstacles include not knowing the context in which the system was built, nor the technologies that were

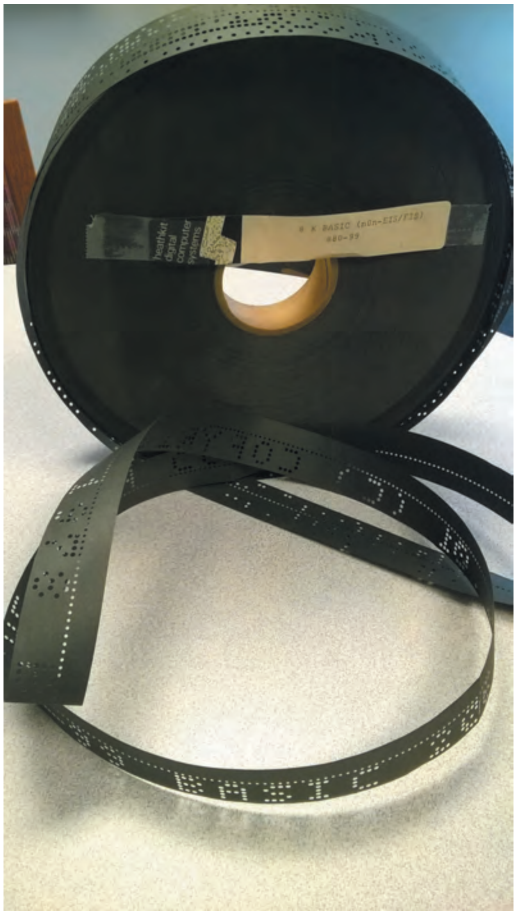
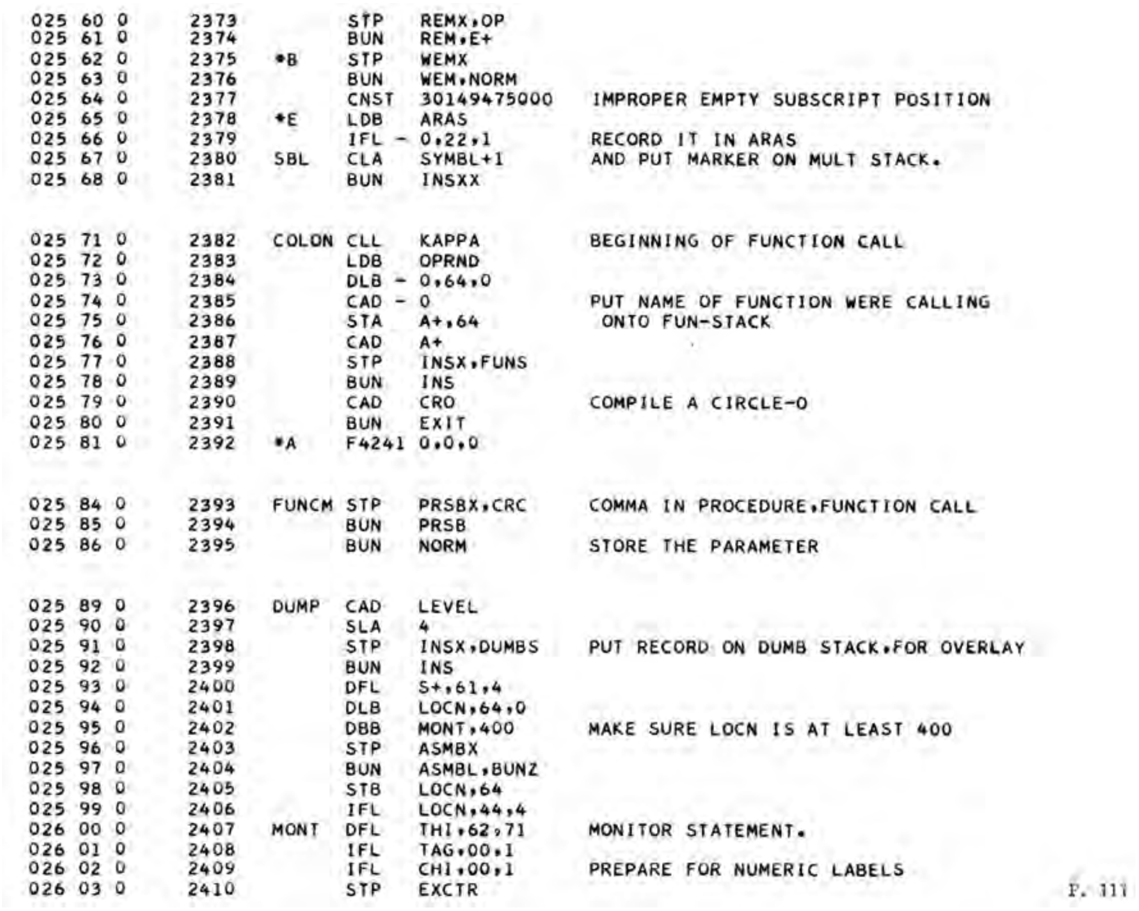


Figure 2.1 8K BASIC compiler paper tape for Heathkit H11 Computer (1978). (Source: Photo taken by author of personally owned tape.)

used, and not having documentation. Systems no longer in use can be even harder to understand as we often do not have a way of running the system to see how it behaves at runtime. Consider the BASIC compiler in Figure 2.1 from 1978 and designed for a 16-bit Heathkit H11 hobbyist computer. To load the tape, we'd need a suitable paper tape reader. We'd need to know how it was encoded. We'd need to know the assembly language for the Heathkit H11 on which it ran. And that's just to get it loaded, let alone understand the software, its design, and the context for which it was used. Ideally, we'd also have a working version of the H11 (and its operating system, HT-11) in order to actually run it in the way it was originally used. Consider this portion of code from Burroughs Corporation's BALGOL compiler in



```
025 60 0    2373          STP   REMX,OP
025 61 0    2374          BUN   REM,E+
025 62 0    2375   *B     STP   WEMX
025 63 0    2376          BUN   WEM,NORM
025 64 0    2377          CNST  30149475000    IMPROPER EMPTY SUBSCRIPT POSITION
025 65 0    2378   *E     LDB   ARAS
025 66 0    2379          IFL - 0,22,1         RECORD IT IN ARAS
025 67 0    2380   SBL    CLA   SYMBL+1        AND PUT MARKER ON MULT STACK.
025 68 0    2381          BUN   INSXX

025 71 0    2382   COLON  CLL   KAPPA          BEGINNING OF FUNCTION CALL
025 72 0    2383          LDB   OPRND
025 73 0    2384          DLB - 0,64,0
025 74 0    2385          CAD - 0              PUT NAME OF FUNCTION WERE CALLING
025 75 0    2386          STA   A+,64           ONTO FUN-STACK
025 76 0    2387          CAD   A+
025 77 0    2388          STP   INSX,FUNS
025 78 0    2389          BUN   INS
025 79 0    2390          CAD   CRO            COMPILE A CIRCLE-O
025 80 0    2391          BUN   EXIT
025 81 0    2392   *A     F4241 0,0,0

025 84 0    2393   FUNCM  STP   PRSBX,CRC      COMMA IN PROCEDURE,FUNCTION CALL
025 85 0    2394          BUN   PRSB
025 86 0    2395          BUN   NORM           STORE THE PARAMETER

025 89 0    2396   DUMP   CAD   LEVEL
025 90 0    2397          SLA   4
025 91 0    2398          STP   INSX,DUMBS     PUT RECORD ON DUMB STACK,FOR OVERLAY
025 92 0    2399          BUN   INS
025 93 0    2400          DFL   S+,61,4
025 94 0    2401          DLB   LOCN,64,0
025 95 0    2402          DBB   MONT,400       MAKE SURE LOCN IS AT LEAST 400
025 96 0    2403          STP   ASMBX
025 97 0    2404          BUN   ASMBL,BUNZ
025 98 0    2405          STB   LOCN,64
025 99 0    2406          IFL   LOCN,44,4
026 00 0    2407   MONT   DFL   THI,62,71      MONITOR STATEMENT.
026 01 0    2408          IFL   TAG,00,1
026 02 0    2409          IFL   CHI,00,1       PREPARE FOR NUMERIC LABELS
026 03 0    2410          STP   EXCTR                                            P. 111
```

Figure 2.2 Burroughs BALGOL 220 compiler excerpt. (Source: Courtesy of the Computer History Museum.)

Figure 2.2.[1] This is a page from the assembler code for a compiler written in 1961. This code is written in Burroughs Assembler language for the B220[2] computer.

The comments in this code are extremely helpful and show that this part of the code was used to recognize function calls (among other things) in programs being compiled. In order to understand this, one needs to also understand the

1. The complete listing can be found at http://archive.computerhistory.org/resources/text/Knuth_Don_X4100/PDF_index/KnuthDigitalArchive-Index.html. Note that this code is unusually well-documented. Donald Knuth documented this for Burroughs. In a personal discussion with Don Knuth, he noted this was no easy task, even at the time.

2. Originally called the Datatron 220, before Datatron was purchased by Burroughs. An emulator for this computer can be found at http://datatron.blogspot.com/2018/.

assembly language for the Datatron 220.[3] So "CLL" refers to the "clear location" instruction, "LDB" refers to the "Load Register B" instruction, "DLB" refers to the "Decrease Field Location, Load Register B" instruction, and so on. Furthermore, one would need to know the architecture of the machine and how operations were processed (registers, stacks, memory structure, etc.). As an example, one would need to understand memory overlays (as referenced in line 025-91 Figure 2.2). One might also need to understand the hardware configuration of the machine, interrupts, and peripherals to understand how it ran. In addition, there may be other bits of software that the system depends on such as the operating system that often provides system calls support but is code outside the program itself. As software has become more abstracted from the details of the computer(s) it runs on, the easier it is to understand a piece of software in isolation. However, software is always somewhat hardware-dependent[4] and also has an ever-increasing software environment on which it depends. This environment can include operating systems, compilers, networking, standards, formats, and other software systems.

Besides being difficult to interpret and understand, software is also vast. Millions of programmers continue to crank out programs of every type. With thousands of programming languages, thousands of different computers, and a huge variety of programs, it's impossible to cover every variety of software. There are also few constraints on the variability of software beyond those in the environment in which it must run. Furthermore, there are many environments in which software may run, further increasing the replication of similar solutions and applications. For example, there are a large variety of mobile devices, most of which run a variety of web browsers, each of which is a little bit different depending on the device, the operating system, and the version.

So, which software is important and which is not? Software that has had a big influence on the direction of other, later software or fulfills a critical niche obviously is. Other software has implemented a key concept or algorithm and has been influential to other software systems. Software that is widely used impacts many people and organizations. Systems that use *standard engineering*, as defined in the last chapter, are implementing previously known techniques and solutions. Even though there are a lot of software systems that are standard engineering (such as payroll systems), our focus will be on how software has changed. We

3. See http://www.bitsavers.org/pdf/burroughs/electrodata/B220/5006_Datatron_220_Instructions_1957.pdf for documentation of the machine language.

4. Though that dependence has become less and less direct as software has become more portable and often running in virtualized environments.

will therefore focus on those systems that use *non-standard engineering* and invent new techniques, concepts, or principles that are likely to influence, and often have influenced, past and future software systems.

2.2 Types of Software

This section describes the scope of software and describes a taxonomy for structuring the study of software.

For early machines, the programs developed were highly dependent on the particular computers for which they were developed. As a result, that software which was developed was not a general solution nor easily portable to a different computer. As machines became more powerful, it became helpful to use some of the extra power to aid in processing software, and specialized software such as operating systems was developed. As this sort of "structural deepening" or specialization occurred, we have a number of software specialties that have been created. A number of these have become technology domains in their own right. The structure of this book is around major and fundamental *domains* of software.[5]

The highest level of categorization of software is based on the type of computational device upon which it runs. While in Chapter 1 we mostly discussed those that use electronic computation using a von Neumann-type architecture, there are other computational devices that have been used or are in an experimental stage. This book only covers the first type (von Neumann). The other types have, so far, required and used different algorithms and different types of computation, but at some point they may be used as an abstract computing engine with an interpretive layer between existing software and the computational engine. Some of the types of these computational engines are listed here:

- *von Neumann.* This is the type described in Chapter 1 that almost all computers are now using. This is the model of a separate memory, computational component, and stored program that controls the running of the computer. Furthermore, this is almost entirely implemented as an electronic device using binary representation.[6] Most computers tend to be general purpose

5. Domains covered here were chosen by their relevance to current students of software. Some domains, such as scientific computing and embedded systems, are not yet covered here due to space and time but are influential and continue to be relevant.

6. Note that not all early computers used binary. ENIAC and others used decimal, notation, binary-coded decimal (BCD), and bi-quinary coded decimal. Bi-quinary coded decimal was used in the Colossus. Three-valued logic (trinary) has also been proposed and used in some computers.

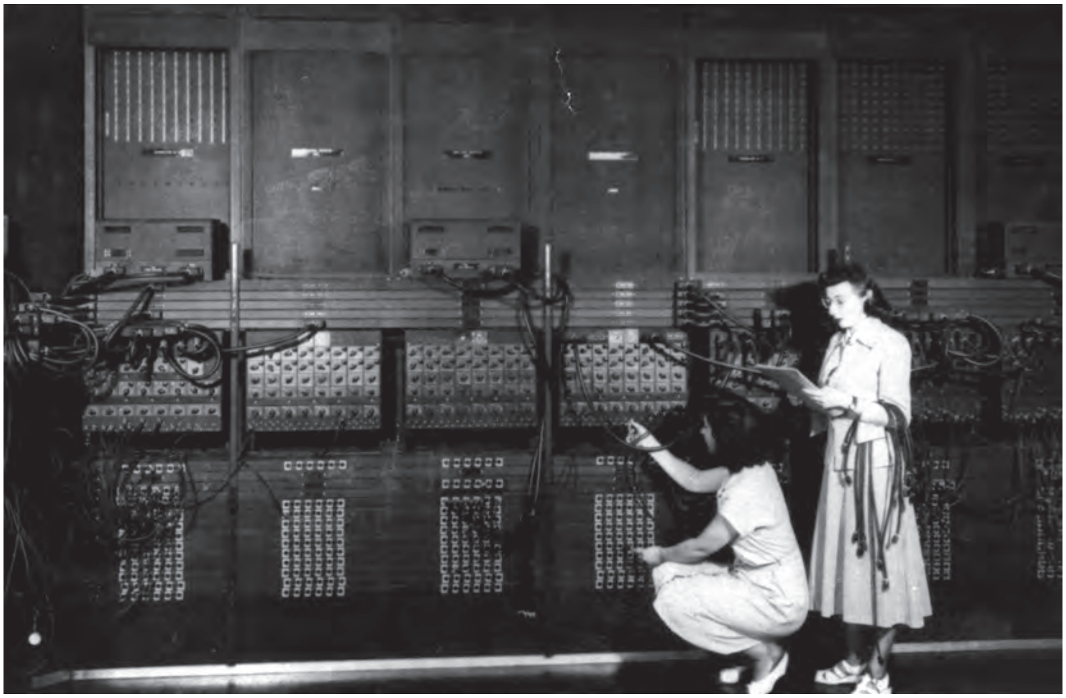


Figure 2.3 ENIAC programming before the von Neuman architecture: Standing: Marlyn Wescoff; crouching: Ruth Lichterman. (Source: US Army Photo from the archives of the ARL Technical Library.)

and are computationally complete.[7] All of the following chapters will assume this type of computational basis.

- *Pre–von Neumann.* Computers such as ENIAC and the Harvard Mark I used plug boards and did not really separate the program from the machine. They are included here for completeness. See Figure 2.3 for a photo of plugging in cables to program the ENIAC.
- *Biocomputers.* An emerging class of computation using systems of biologically derived molecules and processes as a basis for computation. They use these biocomputers to store, retrieve, and process data. Nanobiotechnology is an enabling technology that allows the design and assembly of biomolecular systems. DNA computing and peptide computing are some

7. That is, they are Turing complete and can perform every computation that a universal Turing machine can complete. Very simple computational machines can be Turing complete. As examples, systems such as the Minecraft game and Conway's Game of Life have been shown to be Turing complete.

methods being explored. Interestingly, scientists recently exhibited the ability to create a transistor-like device, dubbed a *transcriptor.*[8] It is unclear what software will eventually look like for biocomputers, but the field has been assigned the term *wetware.*

- *Quantum computation.* Quantum computation makes use of quantum-mechanical phenomena to perform operations on data. Quantum computers use what are called *qubits.* These qubits can be in what is called a *superposition of states*. So, a qubit's state is only set when it is observed. This yields a very different kind of algorithm and fundamental basis for building a computer. Small quantum computers have been built using a variety of different techniques. The D-Wave 2000Q quantum chip was introduced in 2017 and supports 2000 qubits. A quantum Turing machine has been built as a theoretical basis for this model of computation. See Nielsen and Chuang [2011] and Mermin [2007] for more information on quantum computation.
- *Optical (or photonic) computation.* Partially driven by the desire to make optical fiber communications all-optical, rather than having to translate back to electrical at every switching point, optical computation has been working to build optical switches, routers, and computers. So, optical computing uses optical fiber and then optical transistors to produce logic gates to build computers. So far, optical computing has not been cost effective. Software for optical computers would likely be the same (above the operating system) as in electronic computers. See Tucker [2010], McAulay [1991], and Karim and Awwal [1992] for more on optical computation. Some have argued that optical computing is dead, such as in Beeler [2009].
- *Analog computers.* Analog computers take many forms; there are mechanical, electro-mechanical, electrical, and digital–analog hybrid computational devices. Usually, analog computers deal with continuous functions rather than discrete values as binary computers do. There's also some who view them as analogous to their problem domain (hence "analogues") and look at them as modeling techniques (see Care [2010]). One of the more successful types was the differential analyzer machines such as those used at MIT in the 1920s and 1930s under Vannevar Bush's[9] group there. Many others were

8. Note the *transcriptor* was invented by a team in 2013 led by D. Endy at Stanford University. It was hailed as the last component needed to build biological computers. See http://openwetware.org/wiki/Endy_Lab for more on Endy's lab's work.

9. Vannevar Bush was very influential and is famous for having proposed the *memex* as a way to store and access a vast amount of information. He also founded Raytheon and was involved in the initial stages of the Manhattan Project.

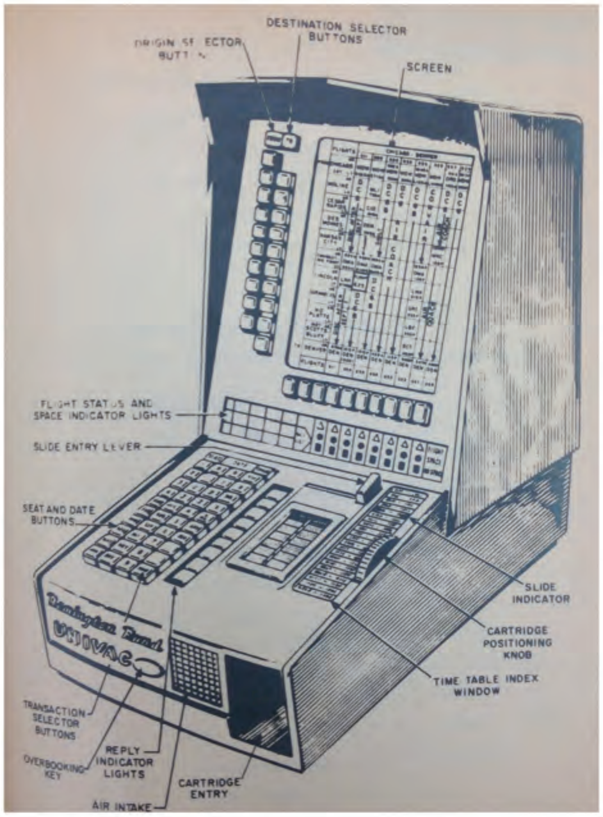


Figure 2.4 A proposed United Airlines reservation system agent terminal for UNIVAC in 1957 from Alan Perlis's papers. (Source: Image courtesy of the Charles Babbage Institute Archives, University of Minnesota Libraries, Minneapolis.)

built and used in the US and around the world, including at the University of Pennsylvania's Moore School (which would also build the ENIAC). These machines were built to solve differential equations. See Rekoff [1967] for a description of programming electronic differential analyzers to solve differential equations. One example of an analog computer out of the many in operation is the machine used by the Lewis Flight Propulsion Laboratory (now John R. Glenn Research Center) in 1949 as in Figure 2.5.

Focusing on software for electronic, von Neumann style computers, this book categorizes software by domain type. Using the notion of a technology domain, software is structured into domains as in Figure 2.6. As the highest level, software is usually driven by trying to create a functioning application to solve a real-world problem. As a result, many of the core technology domains listed in

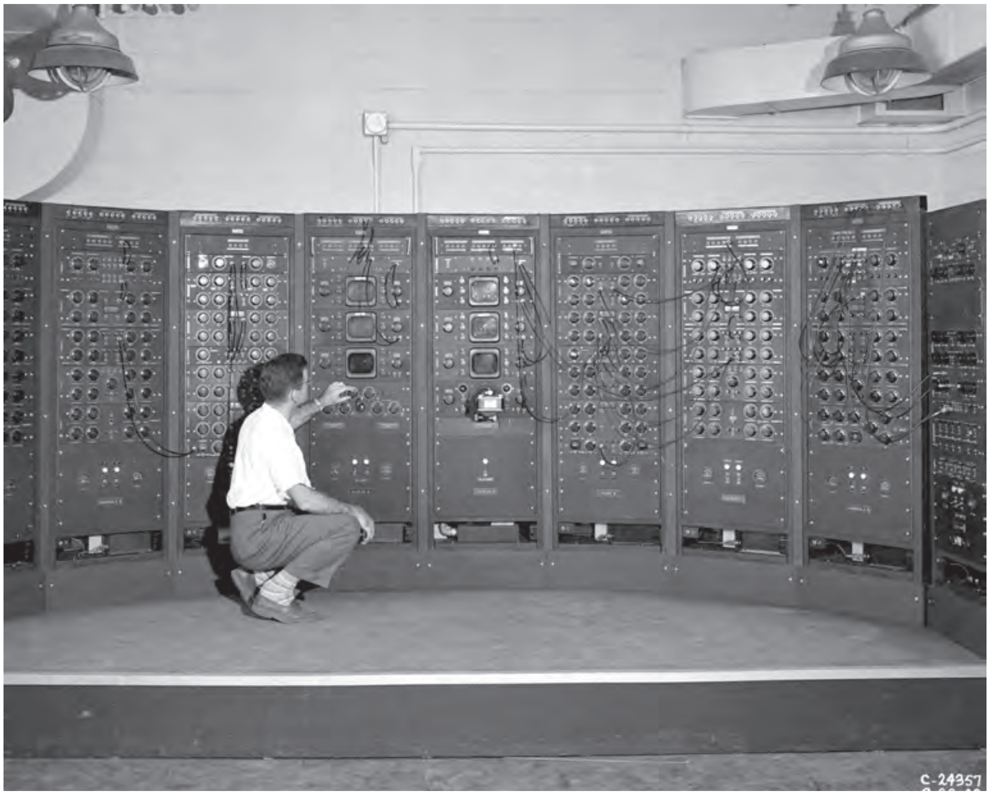


Figure 2.5 The Analog Computing Machine in the Fuel Systems Building of the Lewis Flight Propulsion Laboratory (1949). (Source: NASA; credit: NACA.)

this figure are pushed and refined by requirements and demands coming from varied applications. In addition, note that other domains exist in each high-level domain classification. In Figure 2.6, there are three main classifications of software domains[10]:

1. *Core domains.* The bulk of this book focuses on core domains. These are technologies that have evolved over time and are critical to the ability to build meaningful software applications. Many of these are collections of techniques, tools, concepts, and software components that have a natural relationship in trying to solve common problems or serve a common function. These domains have many possible subdomains that correspond to software that has a shared evolution (for example, *artificial intelligence* might have

10. Note that this model is introduced here as a way of structuring the history of software into parts that are logically related and tend to evolve and mature over time within the domain.

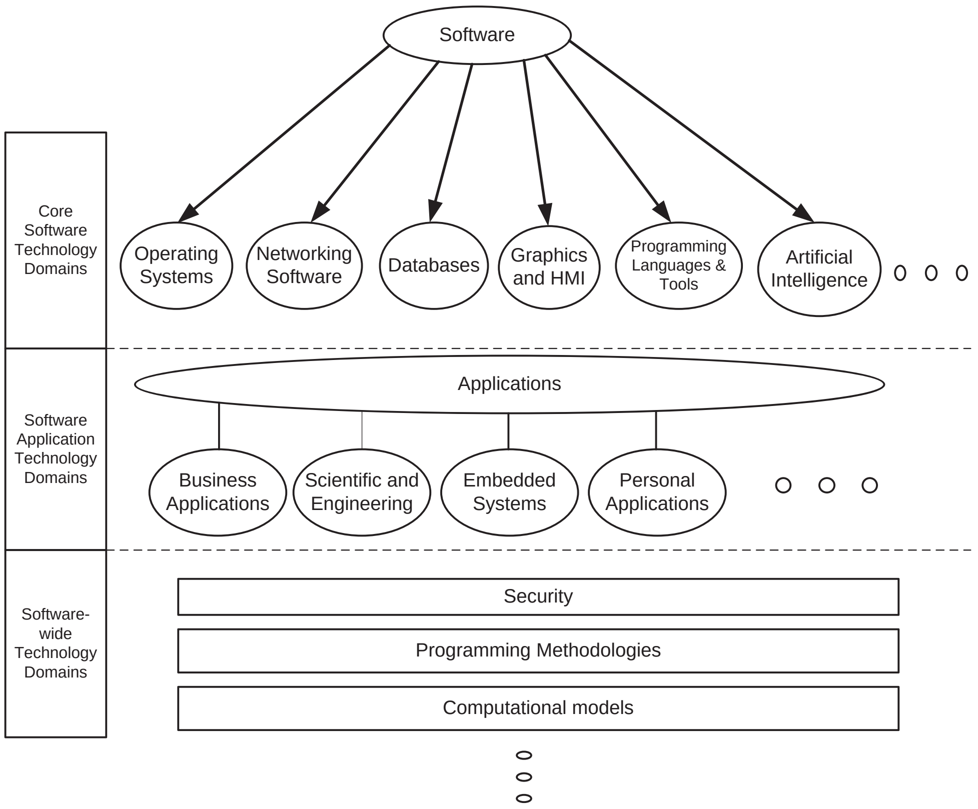


Figure 2.6 A software technology taxonomy.

subdomains corresponding to topics such as *machine learning*, *knowledge representation*, etc.).

2. *Application domains.* Software applications are generally built using components from the core domains and depend on the technology from those core domains to develop solutions and approaches in the application domain. Application software domains are where the bulk of software has been developed. See Figure 2.4 for an example of an early airline reservation system proposed for the UNIVAC.

3. *Software-wide domains.* There are some aspects of software technology that are pervasive to many core and application domains. Some of these are software security and the evolution of security technologies. Another one is software development methodologies and software engineering techniques. Such software-wide domains impact many other software technology domains. As an example, consider the impact of software security on

other domains. Networking software and operating system software are heavily impacted by the need to respond to security threats and have been re-architected to be much more secure. In software development methodologies, a methodology change such as an Agile software development methodology tends to impact the way many different types of software are developed.

While the intent of choosing domains and subdomains is to tell a cohesive story of how the domain has changed over time, these domains are also interrelated. Many domains are affected by what has happened in other domains. As an example, consider the time of the late 1950s when many new, much more powerful machines were being introduced. At this time the first compilers came out and the first operating systems. The ability to write in a high-level language (such as FORTRAN or ALGOL) as well as to have an operating system to make the machine more usable stimulated the use of the computer for a wide variety of applications. Especially in the early days of software, it was common to have the same set of people working on these different aspects, such as programming languages and operating systems. So, particularly when the domain is new, it is highly interrelated with other domains until it starts to have a life of its own and build its own complexity.

This textbook focuses on the core software domains. The intent is to cover the core topics and put in context more detailed topics, particularly those dealing with applications that are specific to industries or functional types. As an example, software gaming applications have become an industry of their own and have a lively and influential history covered only briefly in this text.

2.3 Cultures and Communities of Software

Throughout this book, there will be references to professional societies, user groups, research labs, technology companies, and government agencies. While the scope of this book is not to cover these communities directly,[11] they have had a profound influence on sharing ideas, software, and providing structure and motivation for advancing software.

2.3.1 Professional Societies

Professional societies have given a place to share ideas by organizing conferences and publications. Some of these are noted below as well as their relevance to software history. Many of these are currently active in preserving the history

11. See Ensmenger [2010] for an excellent portrayal of many of these communities and how they developed.

of the technologies they helped create (including software) through publications about that history and the collection of information before it disappears, such as collecting oral histories from technology pioneers.

- *ACM*.[12] ACM was founded in 1947 as an organization focused on computing. It remains one of the leading scientific and professional societies for computing. ACM has been extremely influential through its special interest groups (SIGs) that focus on a particular aspect of computing. As an example, SIGGRAPH (on computer graphics) has been extremely influential in sharing and publishing advances in computer graphics. ACM also awards the A. M. Turing Award[13] every year to those who have had a significant impact in computing. The A. M. Turing Award lectures provide a good source of insightful lectures. The A. M. Turing Award has become known as "the Nobel Prize of computing," and carries with it a substantial monetary prize ($1,000,000 as of this writing). Since 1966, the Turing Award (see https://amturing.acm.org/) has bestowed prizes for notable contributions to computing such as to Edgar (Ted) Codd in 1981 (relational database model), Ken Iverson in 1979 (APL programming language), Ivan Sutherland in 1988 (computer graphics), Fernando Corbató (time sharing operating systems) in 1990, and Frances Allen in 2006 (compiler optimization). Many of the contributions of Turing Award winners are discussed in the following chapters.
- *IEEE Computer Society and IEEE as a whole.* The IEEE[14] was formed in 1963 with the merger of the American Institute of Electrical Engineers (founded in 1884) and the Institute of Radio Engineers (founded in 1912), and was involved in computing before the 1963 merger. The AIEE had a Subcommittee on Large-Scale Computing, established in 1946, and the IRE had a Professional Group on Electronic Computers, established in 1951. After the creation of IEEE, these groups eventually became the IEEE Computer Society in 1971. IEEE is organized by technical societies such as the Computer Society, but also relevant to software are the IEEE Communications Society, the Computational Intelligence Society, the Information Theory Society, Robotics and Automation Society, Signal Processing Society, and the Systems, Man, and Cybernetics Society. The IEEE Computer Society continues to be the focal

12. ACM originally meant the "Association for Computing Machinery" but now prefers to go by just the acronym ACM.

13. See http://amturing.acm.org/ for information about the winners and the award. Also see http://amturing.acm.org/lectures.cfm for the A. M. Turing Award lectures.

14. Originally incorporated in 1963 as "The Institute for Electrical and Electronics Engineers," but now officially named just "IEEE."

point for most IEEE computer-related activity including publications and conferences. IEEE also has a number of computing-related awards such as the John von Neumann Medal that has been awarded since 1992 to people such as Donald Knuth in 1995 (contributions to computer science and programming), Douglas Englebart in 1999 (interactive, personal computing), and Fred Brooks in 1993 (computer architecture, software engineering, and education). These contributions are noted in later chapters.

- *American Federation of Information Processing Societies* (*AFIPS*). From 1951 until 1961, its precursor was called the National Joint Computer Committee and held the Western Joint Computer Conference (WJCC) and Eastern Joint Computer Conference (EJCC). AFIPS was formed in 1961 as a federation of societies (primarily, ACM, AIEE, and IRE) to replace the National Joint Computer Committee and held the Spring and Fall Joint Computer Conferences, and then the National Computer Conferences. AFIPS was dissolved in 1990 and some of its remaining functions were absorbed by the largest of its members, ACM and the IEEE Computer Society. AFIPS first published the *Annals of the History of Computing*, which is the most prestigious journal related to computing history. The *Annals of the History of Computing* moved to the IEEE Computer Society when AFIPS was dissolved. The conference proceedings (WJCC, EJCC, and NCC) provide a wealth of historical information.
- *Society for Industrial and Applied Mathematics (SIAM).* SIAM was formed in 1951 around the industrial uses of mathematics. SIAM is particularly influential in numerical computation and numerical analysis. SIAM publishes journals such as the *SIAM Journal on Numerical Analysis,* the *SIAM Journal on Computing,* and the *SIAM Journal on Scientific Computing.*
- *AITP and DPMA.* In 1949, the AITP was formed as the National Machine Accountants Association (NMAA) in Chicago as a society focused on information technology education for business professionals. In 1962, the name was changed to the Data Processing Management Association (DPMA), which it had until 1996, when it was changed to the Association of Information Technology Professionals (AITP). In 1962, they began offering certifications such as the Certificate in Data Processing (CDP).
- *Society for Information Management (SIM).* SIM began in 1968 and was called the Society for the Management of Information Systems (SMIS). In 1982, the name was changed to the Society for Information Management (SIM). SIM focuses on the managerial aspects of computing and management of information technology in support of an organization.

Other organizations also award significant prizes for computing, such as the Inamori Foundation in Japan that has awarded the Kyoto Prize since 1985 to several computing-related awardees.[15] The Kyoto Prize rotates over a number of fields other than computing. Awardees of the Kyoto Prize related to software have included Claude Shannon in 1985 (mathematical sciences), John McCarthy in 1988 (information science), Maurice Wilkes in 1992 (information science), C. A. R. (Tony) Hoare in 2000 (information science), and Alan Kay in 2004 (information science).

2.3.2 Other Influential Groups

Besides professional societies, a number of other groups have had influences over what software has been developed and how it was developed. This section describes a number of those other influential groups.

- *User groups.* Computer and software sharing groups have a long tradition of sharing information and sometimes of creating and disseminating software. These groups also can help users influence the software or hardware vendor by centralizing complaints and requirements. As an example, the SHARE IBM users' group[16] was responsible for the SHARE Operating System (SOS) as well as having been influential in the development of the first database management systems. Another example is the Digital Equipment Computer Users' Society (DECUS), which was influential from 1961 for Digital Equipment Corporation's (DEC) customers until the company merged with Compaq in 1998. These users' groups continue to be active but are less directly involved in creating and modifying the software itself. See the Independent Oracle Users Group (IOUG, http://www.ioug.org/) and the Hewlett Packard Corporation users' group (http://www.connect-community.org/) for some examples of current groups.
- *Open-source communities.* Open-source communities are focused on producing software that is shareable and free to its users. Richard Stallman started the GNU project in 1983 out of frustration with the legal limitations that were being placed on software. GNU is a recursive acronym that stands for "GNU's Not UNIX." The limitations that Stallman was concerned about include the increasing amount of proprietary software that limited the ability of the

15. The current amount of the prize is 100 million Japanese yen (the equivalent of more than $900,000) along with a 20-karat gold medal.

16. For other IBM users' groups see GUIDE (formed in 1956 as Guidance of Users of Integrated Data-Processing Equipment) and COMMON (see http://www.common.org/), which was formed in 1960 as a professional association of IBM technology users.

users of that software to change or even access the source code. The intent of the GNU project was to produce an entire suite of software that could be used for free by users. Open source has come to include many projects (including the Linux operating system) well beyond GNU. See Tozzi [2017], Raymond [1999], Moody [2001], and Kelty [2008] for more on the open-source culture and history.

- *Research labs.* University research labs as well as industrial research labs have often created technologies that have had significant influence in software. Xerox's Palo Alto Research Center (PARC) formed a computing group in 1970 that implemented many technologies including personal computing, windowing systems, and object-oriented programming languages (i.e., Smalltalk). See Hiltzik [1999] and Smith and Alexander [1999] for descriptions of Xerox PARC. Bell Telephone Laboratories had significant impact on computing in implementation of systems to support the phone system, as well as specific developments such as the UNIX operating system and the C and C++ programming languages. See Gehani [2003], Gertner [2012], and Bell Laboratories [1977] for more on Bell Laboratories. Labs such as SRI (originally, Stanford Research Institute) were very influential with a focus on government-funded research and produced many interesting developments such as the computer mouse with Doug Engelbart's group. Current university and corporate labs continue to produce software advances. There are a large number of examples that are noted in later chapters.
- *Computer and software companies.* Companies actually in the business of producing computers and software have a vested interest in producing software that sells (or helps sell) computers, consulting, or other devices. As a result, companies such as IBM, NCR, GE, DEC, Sperry-UNIVAC, HP, Sun Microsystems, Tandem, Microsoft, Oracle, Sybase, Novell, and many others have produced software and influenced their user bases. These companies have built a culture and community that often extends beyond the life of the company and its products. So, one can find employee groups and passionate users that restore old models of computers and preserve the history and culture of the company. Some of the larger companies will also support corporate archives to preserve their history, such as AT&T, HP, and IBM.
- *Government support.* Government support and software in support of military applications have also had a great deal of influence on software, though often only within the military community. Given the security constraints, some advances did not directly become known to the rest of the computer community until much later. Some projects such as those funded

by the Advanced Research Projects Agency (ARPA), later known as Defense Advanced Research Projects Agency (DARPA), had a great influence beyond the defense community, such as in funding Internet and expert systems research.

2.4 Environment

All software is written in the context of an *environment*. The environment includes the required context that the software needs in order to run. For example, an operating system depends on the hardware on which it runs. So, the hardware is part of the required environment on which the operating system depends. The hardware presents an interface for the types of commands that it accepts and how it can be programmed as part of this environment as well as particular limits and capabilities such as the amount of memory, how long it takes for commands and operations to complete, and particular peripherals that are available. Turning this around, if we have an operating system that can present an environment just like the hardware, then we have a virtual machine. Software applications are dependent on the environment presented by the operating system and may be interdependent with other applications in order to work correctly.

Software is dependent on its environment to run. To understand a piece of software, one has to understand its environment. Some software (such as the example in Figure 2.2) is heavily dependent on the hardware environment in which it runs and couldn't run in any other circumstance that doesn't account for that specific hardware. In this example, even small hardware changes, such as adding or removing memory, could require the program to be modified to still work. Understanding historical software depends on understanding the environment on which it was run.

Over the years, efforts have been made to make the environment more standardized so that software can be more easily ported to other environments. So, efforts to standardize the operating system interface that is presented to applications are intended to help reduce the variance in the environment presented by different operating systems. An example of this is the POSIX[17] standard, but there are many others. However, as one part of the environment may be standardized many others are not. There are also many more different places the software can run such as running on an embedded computer, a mobile device, a personal computer, and many other forms of computers, each with its own operating system and other software and operational constraints.

17. The Portable Operating System Interface standard is an IEEE standard designed to facilitate application portability across different UNIX operating system implementations.

For example, suppose you are running a game in your Internet browser. That game assumes a particular interface with the browser and may also depend on programming language details (such as a particular version of Java). There are many choices of browsers that could be used. There are many choices of devices (each with a different operating system). The browser may be running in a virtual machine, that is then running on top of another operating system that is running on a particular machine. Each of these layers has dependencies and configurations that make up the environment. Security and reliability of software is particularly sensitive to changes in the environment. For example, one could restrict the amount of memory available to the virtual machine in the game example until it starts behaving erratically. This could also lead to finding and compromising security vulnerabilities in that game or in the browser.

Besides the hardware and software environment, there are also assumptions and design decisions that went into designing any software system. These are not always documented or well known, particularly when the system is very old and assumptions have changed and the designers are no longer available to ask. Very often in older systems, system memory was a significant constraint. In the BALGOL compiler example in Figure 2.2, overlays[18] are a result of that assumption. Modern systems may assume that there are no real memory constraints and that their program will never run out of memory. When it does, it can lead to catastrophic failures of the system.

2.5 Influences on Software History

Looking at software as a set of technologies, we've discussed a number of large factors that cause software to evolve and to change. Some of those are in Figure 2.7. These influences include:

- *Computer science.* Computer science is about the study of computation and software, so it is no surprise that it has had a strong influence on software. Computer science helps understand the formality of computation, limits of what can be done, and building theoretical models that will stand the test of time. Computer science also has a deep influence from mathematics such as the building of formal computational, logic, and more specific models, such as formal security models. Computer science is *not* the same as software as we have defined it. Software refers to programs and related artifacts

18. Overlays are a technique used to partition a program into several parts because it cannot all fit in memory at once. Those overlays are then brought into memory when needed and overwriting the previous one. This resulted in programs being split into parts that were called *overlays*.

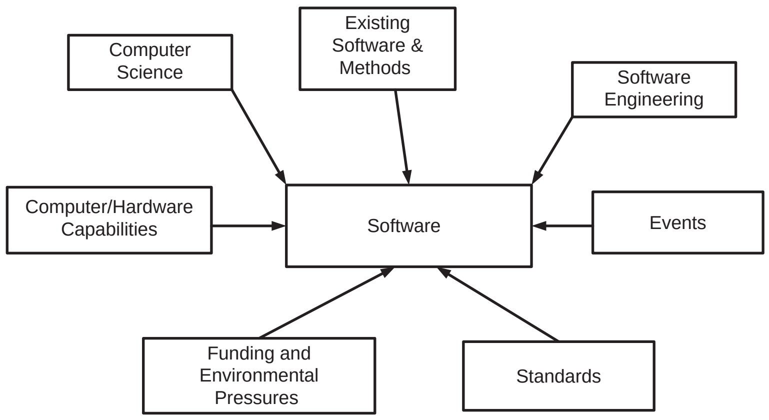

Computer Science
Existing Software & Methods
Software Engineering
Computer/Hardware Capabilities
Software
Events
Funding and Environmental Pressures
Standards

Figure 2.7 Major influences on software technology.

that describe those programs, while computer science refers to the concepts and study of software.

- *Computer and hardware capabilities.* The actual capabilities of the machine have had a liberating effect on software. When new hardware items are introduced, such as a graphical display, this has allowed graphics software to exploit that capability. Additionally, as memory and processing capabilities increased, this allowed software to grow and to tackle tougher problems without having to artificially subdivide the problem. Advancing speed and capabilities have allowed for techniques to be widely used that may have been impractical in the past, for example, supporting interpretive languages. Additionally, it is often argued that this increased speed and lowered cost of computing has allowed inefficient software to be built and supported software bloat.
- *Funding and environmental pressures.* Funding has helped shape the research agenda, particularly when it comes to projects like the Internet and security software that had been funded by the US government. Software and hardware companies have incentives to increase their revenue, which often results in software with new capabilities, or just software to make their hardware worth buying. Pressures such as adhering to laws and regulations, patching security vulnerabilities, and reducing liability can also drive the direction of system implementation and evolution of software systems.
- *Events.* Occasionally, an event can change the direction of software and encourage more development. A good example is the 1988 Morris Internet

Worm that affected a lot of systems[19] on the Internet. Before this, there were firewalls but only a few companies had deployed them. Many security projects were initiated shortly after 1988 as a result of the Internet Worm event.

- *Software engineering.* Software engineering affects the way in which software is developed. The "software crisis" of the 1960s spurred a lot of effort in creating software engineering tools and methods with the hope of building software more quickly and reliably. The "software crisis" was coined to indicate that programmers could not effectively keep up with all the new, much more powerful computers that were being deployed by many organizations wanting to use them for many applications.[20]
- *Existing software and methods.* What has already been developed influences future software in many ways. Besides providing inertia for change and a resistance to change what is working, there is a real cost to having to modify a base of software, train programmers and staff, as well as building new techniques and creating new concepts that depend on concepts that have been previously defined. As an example, it was easier to build new programming languages when one saw the success of FORTRAN and had techniques to formalize the writing of compilers.[21] Large software systems are often very difficult to replace, particularly if the services it provides need to remain up and running. An example of this type is an air traffic control system. These systems involve not only software subsystems but business processes, hardware subsystems, and an entrenched user base. Changing to a new system introduces uncertainties about the safety of the system, retraining for air traffic controllers, and rolling out the system in such a way that it can still interoperate with parts running on the old system.

  So, for the most part existing software and methods slow down changes in the software that might have occurred otherwise.

19. The 1988 Internet Worm is roughly estimated to have infected 10% of the computers on the Internet, which was about 6,000 of the 60,000 computers attached to the Internet.

20. Edsger Dijkstra stated the following in *The Humble Programmer*, his A. M. Turing Award lecture in 1972: *The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*

21. Here, I'm referring to techniques such as the Backus-Naur Form (BNF), compiler generators, and formal parsing techniques.

- *Standards.* Standards are especially critical in areas where interoperability including the interchange of data is important. An example is in networking software, which has to communicate with other networking software. Standards are also a two-edged sword. While they can stimulate change and encourage software development by creating a dependable base, they can also inhibit change by having an almost impossible to change installed base. As an example, consider the standards around TCP/IP protocols. These protocols have proved difficult to change and improve (such as moving to IPv6 from IPv4, which is taking decades to complete) as the protocols are now embedded in millions of devices and systems.

These are the major factors that contribute to the change of software. As we look at more-specific software domains in the coming chapters, we will investigate what the major changes are for each domain. Note that the influences vary in time and by domain.

2.5.1 Software Change Due to Invention

Software, as opposed to other technologies, tends to reuse and adapt preexisting software to interface to a new software technology. Most other technologies tend to dispose of the old technology and are eventually forced by economics to do "fork-lift upgrades."[22] Software (as a whole) has exhibited a resistance to such replacement and instead usually tends to persist for a long time and may eventually get re-written as time allows and economics encourages. This has been particularly true since software has become more portable to different physical computers and as computer vendors support backwards compatibility.

Many examples of this kind of inertia in the change of software have occurred and are very likely to reoccur with new inventions. One property of software is that interfaces can be built to do a transformation of the inputs and outputs such that there is no need to rewrite the underlying software. Using this property, it is often vastly less expensive to create the transformational interface rather than to rewrite the existing software base.

One example was the invention of the World Wide Web where there was a drive to move a large number of interactions to use the Web and to provide an interface to customers, businesses, and employees. Rather than rewriting all the software, interfaces were built to expose and integrate the technology such that the user

22. Some technologies have tended to stick with old, proven designs. In particular, aircraft designs are slow to change, and we are still flying B-52s, C-130s, and Cessnas that were designed in the 1950s, along with the Boeing 737s and 747s that were designed in the 1960s. US airlines have just recently retired the 747.

interface could be browser-based. Over time, much of the newly built software has been based on web technologies, while older software has gradually been changed, if at all.

Another example has been the support for networking protocols. Once there has been a large enough deployment of a particular protocol, it becomes economically unattractive to do a wholesale replacement of that protocol. Instead, we can develop an interface or encapsulation of that protocol and gradually phase it out over time. The Internet Protocol was designed to internetwork between networks that may have different underlying protocols. We continue to support IPv4 over IPv6 networks even today because it's possible to do so, as well as the economics makes it expensive to quickly eliminate IPv4.

2.5.2 Software Change Due to Functional Failure

As opposed to changes due to invention, changes to software due to a widespread functional failure are less smooth. While there have not been too many examples of a widespread functional failure of software, one example was the Year 2000 problem. This issue was enabled by the fact that older software did not have to be replaced because of hardware changes due to portability and vendors supporting backward compatibility. The Year 2000 problem (or "bug," depending on your perspective) was caused by software being written using only the last two years of the date (so "80" for "1980"). It really didn't matter for most software until dates after the year 2000 needed to be included. When much of that software was being written, it was difficult for programmers to envision that the software would run for 30 or 40 years, particularly because earlier software had not tended to last due to a lack of portability. For software that was critical, it had to be modified and a lot of older software systems were replaced rather than fixing their Year 2000 issues.

2.6 Summary

This chapter looks at software as a technology domain that is then composed of subdomains. We looked at the high-level categories of software that depend on the actual computational engine that is used (such as biocomputers, quantum computers, optical computers, etc.). We will focus our efforts on the vast majority of current software that is based on a von Neumann type of computational engine.

In Figure 2.6, we define the subdomains of software that we will study further in later chapters. These include areas such as operating systems, networking, programming languages, software methodologies and tools, and databases. These domains have a relatively cohesive story to be told, involve a common scope, and usually involve similar pressures and communities.

2.7 Exercises and Projects

2.7.1 Exercises

1. A key to making computers easier to build and to use simpler components was the use of binary. Early computers (such as the ENIAC) used decimal notations rather than binary. Describe how this shift occurred to binary and include Claude Shannon's MIT MS thesis and the development of components and theory that made that work. Explain in a few paragraphs the key developments and how early machines leveraged the transition to binary.
2. Biocomputers have recently gained some momentum. Explore why some have high hopes for biocomputers. Take a position on whether biocomputers will survive the next 20 years as a viable medium and defend it.
3. Optical computers have fallen out of favor in the last few decades. Determine if they are currently viewed as a promising area. How would optical computing affect software development?
4. Find a quantum computing algorithm for sorting (such as Lov Grover's algorithm described at http://en.wikipedia.org/wiki/Grover%27s_algorithm). Explain how this algorithm works on a quantum computing device.
5. USENIX (see https://www.usenix.org/) is a UNIX operating system users' group. Examine how this group has influenced the design and implementation of the UNIX operating system.
6. Define the environment for the *Angry Birds* mobile app running on an Android device. What is required of the environment to be able to effectively run Angry Birds? If you were only given the source code for the Angry Birds app, what more would you need to know to understand the code and whether it would run?
7. Amdahl Corporation produced a line of computers that were perfectly "plug compatible" with IBM mainframes and also had a users' group (the Amdahl Users' Group). Find information about this group and how it influenced Amdahl computers. Investigate the relationship of Amdahl Corporation's founder, Gene Amdahl, to IBM.
8. Investigate the 1968 NATO Software Engineering Conference held in Garmisch, Germany. How did this conference affect the creation of a *software engineering* discipline? When and where were the first software engineering undergraduate programs established? Why was this conference held?

9. Give an example of a technology for which software is a component. Describe why that technology is a technology (based on a physical phenomenon, built out of other components, and those components are also technologies).

10. The term *software* is used in a 1953 report by RAND Corporation.[23] Look up this use of *software* and explain how it has nothing to do with computer programming or software as we have defined it.

2.7.2 Projects

1. Investigate a software domain relating to *embedded systems.* Define the scope of this domain and determine if it is a subdomain of those described here. Is the domain of embedded systems cohesive enough to tell a story about its evolution or is it so fragmented as to not have a story that can be easily modeled in terms of influences and evolution?

2. Biocomputers' software is just being defined as of this writing. Investigate the different models being defined for biocomputing software. Is there an approach to bridge between traditional software and biocomputer software? What will be needed to make that bridge complete?

3. An interesting type of analog computer is based on the use of water and water flows to perform computations. An example of this type of machine is one built by New Zealand economist William Phillips called the MONIAC (Monetary National Income Analogue Computer), created in 1949. Investigate whether so-called "fluidic logic" could theoretically produce a general computer (i.e., could one build a NAND gate or other logic gates) and whether it could ever be Turing complete. The MONIAC (see Figure 2.8) was built to model flows of money and economic processes at the national level. See Reserve Bank Museum [2008, p. 10–12] for a short description of the MONIAC at the Reserve Bank of New Zealand Museum.[24] If it is possible to build a Turing complete water computer, show how those circuits could be built and programmed. Explain why this has not been pursued beyond small control systems.

4. Many analog computing devices have been built that use many different techniques in order to solve problems. One unique device is the FERMIAC (sometimes called Fermi's trolley or the Monte Carlo trolley). This device

23. See *A Survey of the Current Status of the Electronic Reliability Problem (U),* by R. R. Carhart, August 14, 1953, Report RM-1131, p. 69. https://www.rand.org/content/dam/rand/pubs/research_memoranda/2013/RM1131.pdf.

24. There is another one at the Science Museum in London.

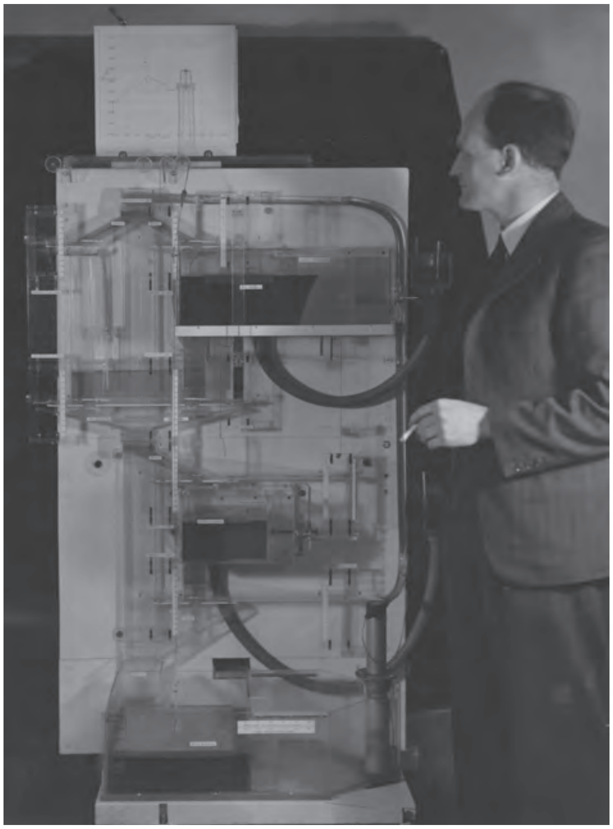


Figure 2.8 Professor A.W.H. (Bill) Phillips with the Phillips Machine, aka MONIAC. He was also known for the Phillips curve. Circa 1958–1967. (Source: Courtesy of LSE Library.)

was designed by Enrico Fermi to help with calculations needed for creating nuclear bombs using the Monte Carlo method before the ENIAC could be used for this purpose (See Haigh and Priestley [2016] for a description of how this was implemented on the ENIAC). Write a paper that shows how the FERMIAC was used to solve problems. Include a sample problem and indicate each step in order to arrive at a solution. Optionally, create a simulator that allows problems to be set up and solved as they would be on a FERMIAC device. See also Metropolis [1987].

5. The ENIAC did not originally separate out the program from the circuitry of the machine, but it was later retrofitted to allow a von Neumann-like architecture so that it could be more easily programmed. This was described in an Aberdeen Proving Ground Ballistic Research Laboratories report [Clippinger 1948] where the move was called from "local programming" to

"central programming." Analyze the new "orders," which are like machine instructions, and determine which ones were dependent on the ENIAC's prior architecture and which ones were more independent from the ENIAC's architecture. Compare the modified ENIAC's orders to the instruction set from the Small-Scale Experimental Machine (SSEM or the Manchester Baby, which had seven instructions) as described in Burton [2005].

2.8 Further Readings and Online Resources

This chapter builds a model for software to categorize the history of software into domains based on Arthur [2009] with some ideas building on those in Mahoney [2011], Kuhn [1962], and Constant [1980].

More can be found on communities and organizations that still exist such as for ACM (see http://www.acm.org/), IEEE (see http://www.ieee.org/index.html), SIAM (see http://www.siam.org/), SIM (see http://www.simnet.org/), and AITP (http://www.aitp.org/).

For information related to the use of history in the teaching of computing, see Lee [1996] and Impagliazzo et al. [1998]. Von Neumann [1958] discusses how the brain can be viewed as a computing machine and discusses issues and research directions.

3 Operating Systems

The history of operating systems has been closely aligned with hardware history until relatively recently with the proliferation of hypervisors and virtual machines.[1] The history of operating systems includes how they have evolved to become more portable and useful. The subject ranges from mainframe operating systems such as IBM's OS/360, to other influential systems such as MULTICS[2] and UNIX®,[3] to mobile device operating systems such as Apple's iOS and Google's Android. The drive to make computing hardware more effective and easier to use has led to the development of features like virtual memory, multi-programming, multi-processing, and time-sharing, and have contributed to a set of features that we expect of almost every operating system.

3.1 Operating Systems and Their Evolution

Operating systems can be defined, as in Silberschatz et al. [2012, p. 3], as a "program that manages the computer hardware." That is, operating systems have been built to make the computer hardware more efficient and easier to use.[4] More recently, one might argue that virtualized operating systems are less about managing computer hardware than providing an execution environment that eventually bridges to the hardware. Nonetheless, operating systems are built not as an end within themselves but as a way to make it easier to get more out of the computer and not

1. Virtual machines are an older concept as IBM's VM/370 was developed in 1972 (which was based on the earlier CP-67/CMS IBM System/360 operating system), but the proliferation of VMs occurred more recently in the late 1990s and 2000s.

2. See http://www.multicians.org/ for more on MULTICS and its history.

3. "UNIX" is a registered trademark of the Open Group, http://www.unix.org/.

4. A 1970 definition in Katzan [1970a, p. 8] states that "An operating system is an integrated set of control programs and processing programs designed to maximize the overall operating effectiveness of a computer system." At that time, the focus was on effective utilization of an expensive machine.

having to worry about all the hardware details in our programs, as early computers' programmers were forced to do. Operating systems have taken over many tasks that programmers previously had to include in their programs, including memory management, process management, synchronization, and file and device management. Operating systems have become a domain of software technology in their own right with many specialized types of operating systems and exhibit a structural deepening of the concepts and related technology.

Operating systems were not developed in a straightforward evolutionary process. Instead, they were developed amidst concern about how much processing capacity was used by the operating system versus how it really improved the throughput of the computer. There is some contention about what was the first operating system, but certainly one of the earliest was the monitor that General Motors Research Laboratories produced for the IBM 701 in 1953 (see Weizer [1981] and Rosin [1969]). General Motors Research created an operating system for the IBM 704, called General Motors/North American Monitor (see Patrick [1987] for a description of the systems developed for the IBM 701 and 704). Shortly thereafter, the SHARE IBM user group developed their own operating system for the IBM 704 (see Figure 3.1) called the SHARE Operating System (SOS). During this early time, it was common for companies using the computers to develop their own operating system rather than depending on the computer vendor.

These early operating systems did little more than to simplify the input, output, and transition between jobs. Each program would utilize the entire machine and in order to make efficient use of the expensive computer, the operating system would help reduce the time between these jobs to keep the machine as busy as possible. This generation of operating systems relates to the "First" generation in Table 3.1[5] and are often referred to as "Monitors" due to their relative simplicity. In the 1960's, operating system began to evolve into what we think of them as today. In the "Second" generation in Table 3.1, types of operating systems evolved into batch, transaction processing (such as the American Airlines SABRE system), real time, and time-sharing. The "Third" generation of operating systems contained many more complex modes of operation and the ability to support batch, time-sharing, and other modes of operation such as real-time. During this third generation, every computer manufacturer developed their own operating system with its own unique features. In the late 1960s, work began on a number of systems designed

5. These generations are modeled after [Weizer 1981] but are modified. His fourth generation reflected more of a move to firmware for operating system features. That has happened to a large extent, but the moves to portability and virtualization reversed some of that trend by abstracting the operating system from the hardware.

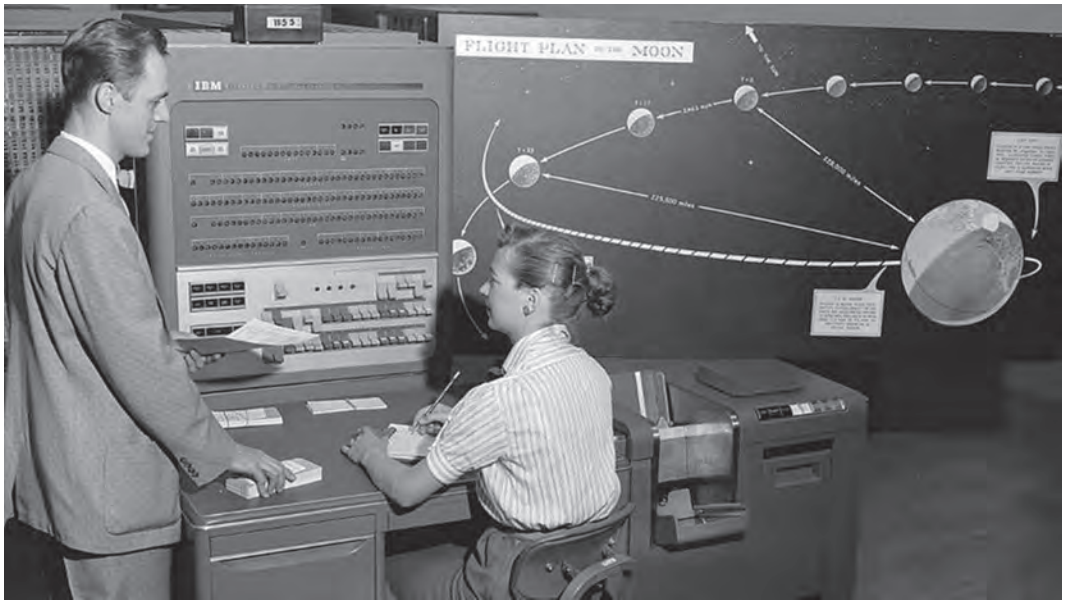


Figure 3.1 IBM 704 at NASA's Jet Propulsion Laboratory in 1959 with an unknown human "computer" seated at the console with a card reader/punch to the right. (Source: Courtesy of International Business Machines Corporation, ©International Business Machines Corporation.)

to be portable to multiple, different manufacturers of computers. Systems such as MULTICS (as part of project MAC at MIT) and others were developed with one of the primary design goals of being portable and easily modifiable (see Corbató and Vyssotsky [1965] for the reasoning behind their choice of the PL/I programming language). This "Fourth" generation of operating systems ended up coalescing the general-purpose operating system market to a few players. The "Fifth" generation is the move to virtualized operating systems. While IBM's VM operating system was developed in the early 1970s, the concept of virtualized computers and hypervisors was not widely adopted beyond IBM until the late 1990s. Please also see Figure 3.2 for a diagram of the interrelationship of these generations. Operating systems also are closest to the hardware and have a natural co-evolution with hardware. As a result, one of the strongest influential factors has been the availability of hardware to support the changes in operating systems. These influences on operating system change (see Figure 3.3) include hardware speed advances that enabled enough spare processing capacity to be able to run an operating system in addition to running a user's program. Another key hardware advance that profoundly affected operating systems was having reliable disk storage. This allowed for programs to

Table 3.1 **Operating system generations**

Generation	Timeframe	Description
First	1955–1962	*Simple monitors:* Batch system whose primary function is to provide job to job linkage with no multi-programming and works on a single computer type.
Second	1960–1968	*Single mode:* A single mode of operation (batch, time-sharing, real-time, etc.) was supported by the system. Designed to operate on a single computer type.
Third	1965–Present	*Complex multi-mode:* Has the ability to support multi-processing and multi-programmed computers and designed to operate on a family of computer systems, including the introduction of multi-threaded systems.
Fourth	1970–Present	*Portable:* Can be ported to multiple machines with different hardware architectures.
Fifth	1970–Present	*Virtual, hypervisors:* Built to host other operating systems and to abstract hardware for use by virtual machines.

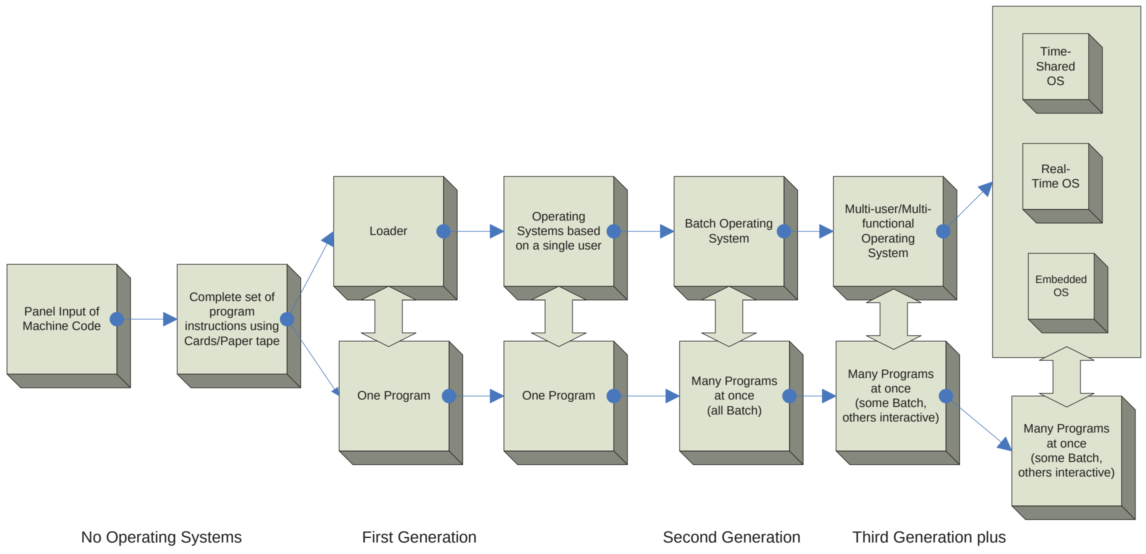

Panel Input of Machine Code
Complete set of program instructions using Cards/Paper tape
Loader
One Program
Operating Systems based on a single user
One Program
Batch Operating System
Many Programs at once (all Batch)
Multi-user/Multi-functional Operating System
Many Programs at once (some Batch, others interactive)
Time-Shared OS
Real-Time OS
Embedded OS
Many Programs at once (some Batch, others interactive)
No Operating Systems
First Generation
Second Generation
Third Generation plus

Figure 3.2 A high-level evolution of operating systems.

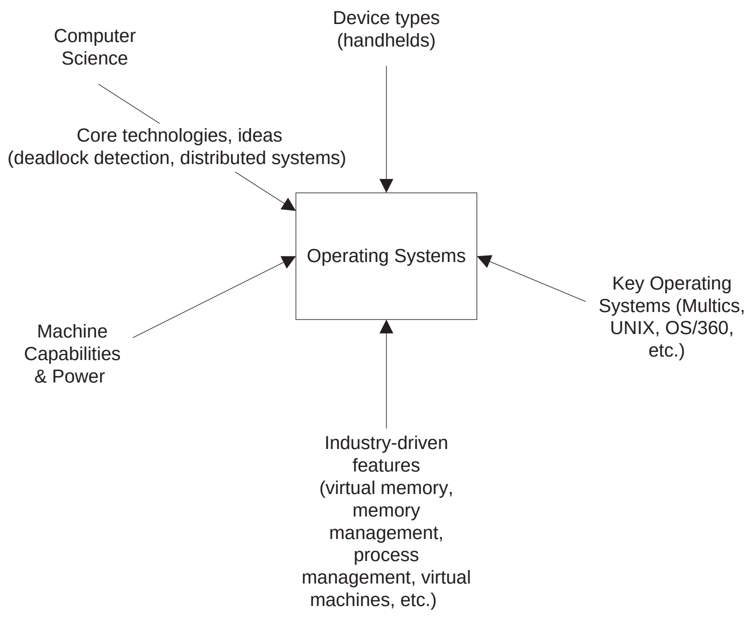

Computer Science
Device types (handhelds)
Core technologies, ideas (deadlock detection, distributed systems)
Operating Systems
Key Operating Systems (Multics, UNIX, OS/360, etc.)
Machine Capabilities & Power
Industry-driven features (virtual memory, memory management, process management, virtual machines, etc.)

Figure 3.3 Influences to changes in operating systems.

be efficiently brought in and out of memory and the feasibility of efficient virtual memory.

Academic computer science departments have had significant influence in many of the key concepts in operating systems, particularly in the development of core technologies such as deadlock detection, process synchronization, and distributed operating systems. They have also developed some key operating systems such as ATLAS, MULTICS, and THE that demonstrated new, groundbreaking features. Many of the features of operating systems were even more driven by those who were using them in industry and wanted to make efficient use of them. As a result, many features were developed and refined in industry and by computer manufacturers in order to make the machines more efficient, more useful, and easier to sell or buy.

Several key operating systems have influenced operating systems as a whole and will be covered in a later section of this chapter. These key operating systems not only produced an interesting system but have affected the later evolution of operating system software.

Lastly, operating systems have been affected by having different types (or "classes" as per Bell's Law) of computers. Whenever a new type of computer is introduced, it gives the opportunity to leverage the unique features of that new class of computer and to introduce a new operating system or to modify an existing one to take advantage of those features. An example would be the introduction of the Android operating system that was designed to work on smartphone-type devices (and has since expanded to other mobile devices). Apple's iOS builds off its success with MacOS and utilizes some of the same features. These new types of devices introduce an opportunity to re-think what the operating system should do for those kinds of devices.

Operating systems evolved over time from very simple monitors, which did little more than make the use of a computer more efficient, to protecting data and programs from one another, and to providing a major portion of the user interface. As computers became more powerful and less expensive, the ability to use some of those computing cycles for managing the resources of the machine became more palatable and feasible.

3.2 Operating Systems Scope

Operating systems have a significant number of types and many operating systems have come and gone over time. This section details some of the major types of operating systems, major features, as well as describes several operating systems that had a significant influence on later systems.

3.2.1 Operating System Types

Throughout most of this chapter, an "operating system" refers to a general-purpose operating system that is meant to be put to a wide variety of uses. However, there are a number of specialized operating systems that have been honed to specific purposes. Some of these are described below.

- *Resident monitors* are one of the simplest forms of an operating system. They are called "resident" as they always reside in memory and their primary task is to transfer control from one program to the next. The resident monitor would be started when the computer was booted, and it would then transfer control to the first program. When complete, the first program would then transfer control back to the resident monitor who could then start the next program, and so on. The concept of job control was introduced with monitors to have a way to instruct the monitor as to which job to do next via *control cards* that evolved into job control languages (such as IBM's JCL). Monitors would interpret these cards and then load the program into memory to be run. The monitor was made up of a control card interpreter, a method to transfer control and sequence the jobs, and a loader to bring the program into memory. Besides these components, device drivers were built to control the input and output devices. These device drivers were generally shared between the user program and the resident monitor.

- *Real-time operating systems* are built to have well-defined time constraints that must be met. Systems dealing with real-world time constraints are often built on real-time operating systems. Examples are phone systems, weapon systems, safety-critical systems (such as medical devices or life-support systems), and other device control systems (such as automobile or other embedded systems). There are two general categories of real-time operating systems. *Soft* real-time operating systems provide a way to identify critical tasks and to prioritize these over other tasks. Many general-purpose operating systems provide soft real-time features. *Hard* real-time systems are those where the tasks must be completed within specific time constraints or the system is considered to have failed. Generally, those time constraints are attached to external events. For example, picking up a landline phone handset may have a requirement to give a dial-tone within 30 milliseconds. If the system cannot always produce a dial tone in 30 milliseconds for every line, then it has failed. Examples of a hard real-time operating system are VxWorks (see http://www.windriver.com/), Intel's iRMX, and Duplex Multi Environment Real Time (DMERT, later named UNIX Real-Time Reliable,[6] see Wallace and Barnes [1984]).
- *Embedded operating systems* are those that run within a device to operate that device. They are sometimes real-time operating systems, but not necessarily. An example would be an operating system that runs on a cellular phone (non-smart).[7] The operating system will run the device but the user generally does not interact with the operating system directly. An *embedded system* is generally defined to be a system that is part of a larger system such that the existence of the system is not obvious to the user. So, the numerous systems (and operating systems within those) within an automobile are examples of embedded systems. Generally, embedded systems are tuned to perform some specific task. The operating system of such embedded systems can be heavily modified to meet the requirements of the device more specifically. For example, a network firewall will be running its own operating

6. Note that MERT, DMERT and UNIX RTR were not directly based on UNIX but were more influenced by systems such as Dijkstra's THE operating system. They did run a UNIX emulator to be able to run UNIX programs.

7. Smartphone operating systems such as iOS and Android have become more like general-purpose operating systems than an embedded one.

system, often a customized version of Linux to be more secure and to meet performance and real-time requirements.

- *Distributed operating systems* provide a single interface to a group of loosely coupled computers over the network so users access remote resources in the same way as local resources. This gives the distributed operating system the ability to allocate resources based on availability and requirements, rather than just location. So, the distributed operating system can move data and processes between the computers in such a way as to better meet the needs of the users. Bell Labs' Plan 9 and Inferno operating systems are examples of systems built to be distributed.
- *Network operating systems* are those that provide an environment where users can access remote resources on other computers on the network. More specifically, there have been several operating systems developed just to manage network resources in a client/server architecture such as Banyan VINES, Windows Server, and Novell NetWare.[8]
- *Secure operating systems* are those that are designed with security requirements built in and enforced by the operating system. Recently called *trusted operating systems* and evaluated using the Common Criteria (see http://www.commoncriteriaportal.org/), these types of systems are evaluated and given different levels of security assurance. Before the Common Criteria, the US Department of Defense used what was called the "Orange Book"–Trusted Computer System Evaluation Criteria (TCSEC) to assign security levels to operating systems. An example is Security-Enhanced Linux (SELinux) that can support mandatory access controls, which is one of the key criteria for secure operating systems.
- *Hypervisors* and *virtual machines* support multiple different operating systems on a single physical machine. The software that supports virtual machines has come to be called a *hypervisor.*[9] Hypervisors (such as VMware, Xen, or IBM VM/370) provide an interface that looks like a real machine to other operating systems. As such, hypervisors are operating systems themselves, managing the underlying hardware. Hypervisors also come in two basic types. Type 1 are those that run on the physical hardware without

8. Note that the term "network operating system" also sometimes refers to the operating systems used by network devices like routers, switches, and firewalls, and includes examples like Cisco's Internet Operating System (IOS), Juniper's JunOS, and Extreme Networks' ExtremeXOS.

9. A *hypervisor* is a supervisor of the supervisor, where *supervisor* is another term for an operating system.

any other operating system. VMware ESXi and Xen are Type 1. Type 2 are those that run on top of another operating system and then run its virtual machines within the hypervisor. VMware Fusion (which runs on MacOS) and Oracle's VirtualBox are examples of Type 2 hypervisors.[10]

These more specialized types of operating system give some idea of the diversification of operating systems, but does not cover all specializations.

3.2.2 Operating System Features

A brief background is provided of some operating systems features that are referenced in later sections. This section is not meant to take the place of a full course in operating systems and there are a large number of features not covered here. The focus in this section is on those features that have improved operating systems' ability to make the system easier to use, particularly for programmers.

3.2.2.1 Memory Management

One of the core functions for operating systems has been to manage the memory of the computer. Improvements in memory management have freed programmers from having to go to extraordinary means to make their programs fit within memory. While it was quite simple with a resident monitor as defined in Section 3.2, it also restricted the system to running a single program at a time and a program was bound by the physical memory in the machine.

One early concept that aided programmers was that of *relocatable code.* With relocatable code, the program would be placed in memory by the operating system and the program itself would not have to be concerned with physical (or, absolute) memory addresses. This concept of relocatable code became particularly important when the address space began to be used concurrently by multiple programs, such as being placed in different physical memory segments. *Multi-programming* (multiple programs running concurrently on the same computer) was introduced to significantly improve the job throughput of early machines and began as a way to use the CPU more effectively.

The way memory has been allocated to programs has changed significantly. Initially, with resident monitors a single program was allocated all of whatever was left of memory outside of what the resident monitor was taking. Many operating systems segmented memory into blocks and would bring in a program into a block

10. It's possible to nest hypervisors so that one could install a type 1 hypervisor, run operating systems on it, then run a type 2 hypervisor in one of the operating systems, and so on. However, performance will degrade quickly due to the overhead of the hypervisors and the consumption of memory.

in which it fit. This, however, led to a problem of memory fragmentation as the program would often be too small for the segment and waste the remaining portion. The opposite problem of a program being too big for its segment resulted in having to split a program (often in an artificial manner) into overlays. A program would then start with the first segment and then when it was ready bring in the second segment, and so on. The problems of wasted memory from fragmentation and of having to split your program into often artificial overlays were almost completely solved by the introduction of *paging.* Paging segmented memory into fixed size and relatively small parts, called “pages.” These pages could then be managed by the operating system and only active pages need stay in physical memory. So, pages could be put on a disk (or other “backing store”) and entire processes could be swapped in and out as needed. Needless to say, anytime a needed page had to be brought back into memory it took time, so it was desirable to make that as infrequent as possible. The concept of a “working set” of pages (see Glass [1998, pp. 250–271]) identified the pages and the number of pages required to keep the system performing well.

Virtual memory is the ability of a process to use more memory than the computer physically contains. Virtual memory separates logical memory from physical memory and the operating system is tasked with implementing that logical memory model in a physical memory. So, the *virtual address space*, or the memory locations that a program can reference, is separated from the *physical address space* that refers to the physical memory locations on a particular computer. Paging has become the most common manner in which this is done, though earlier systems provided virtual memory using a combination of segmentation and paging (such as MULTICS). The size of the address space became an issue as programs and their requirements for memory grew. So, the ability to offer a 32-bit byte-addressable space would be able to address four gigabytes of memory.[11] Performance of the virtual memory use was managed by the operating system, freeing programmers from having to constantly worry about their program size or physical location. Memory management support has relied a great deal on hardware support for memory management and most modern systems have significant hardware support for memory management.

11. While most computers are now byte-addressable, many earlier computers were “word” addressable. That is, rather than an address pointing to a byte, addresses would point to a “word.” Word-length varied and could be 36 bits (DEC PDP-10) or 60-bits (CDC 6600) or whatever the designers chose. So, the address used to reference these words could be a different size (like 18 bits in the PDP-10), making the addressable memory space $\textit{word size} * 2^{\textit{address size}}$ bits.

3.2.2.2 Process Management

Another key function of an operating system is managing the programs and processes to ensure the programs complete, perform sufficiently, and are cleaned up in order to make way for other programs and processes. The way this has been done in operating systems has changed from no real process management to support for multiple CPU cores and running thousands of concurrent processes in a single system.

Early operating systems only supported a batch mode of operation where either one job was done at a time or multiple jobs were loaded into memory and run in a multi-programming mode. The reason for this was primarily one of making efficient use of the expensive computing resources available at the time. The SAGE air defense system included a form of specialized time-sharing and interactivity as did some other real-time systems. In 1959, John McCarthy proposed in a memo that MIT develop its own time-sharing system based on the IBM 709.[12] As a result, a number of time-sharing systems were created that provided for an interactive user experience by giving each user a time-slice of processing at regular intervals, making it appear to be responding to everyone in a predictable and consistent amount of time. The MIT Compatible Time-Sharing System (CTSS) system was developed (and described below) as well as several other systems in the mid-1960s. This development gave programmers an interactive interface in which to build their programs.

3.2.2.3 Virtual Machines

Virtual machines provide a virtualized view of the underlying computer hardware that enables the running of multiple, even disparate, operating systems on the same device at the same time. This ability has enabled operating system makers and users to run different operating systems more easily and has enabled more flexibility in how computing hardware is used.

First developed for the IBM System/360-67 in 1966 as the CP-67 operating system and formally defined by Popek and Goldberg with formal requirements in Popek and Goldberg [1974], it is only since the mid- to late-1990s that machine virtualization has really been deployed on a wide scale. IBM's CP-67 was modified and deployed as VM/370 in 1972 where it became widely used as a mainframe operating system. One of the motivations at the time IBM deployed VM/370

12. This "Memorandum to P.M. Morse," dated January 1, 1959, details the hardware requirements and recognizes the impact of developing time sharing for general use. See http://www-formal.stanford.edu/jmc/history/timesharing-memo.html for a copy of this memo.

was to ease the migration from one operating system to another.[13] VM/370 provided a way to *virtualize* the hardware so that different operating systems could be installed on top of VM/370. This virtualization required another layer of software that provides the appearance of multiple dedicated sets of hardware to multiple operating systems on top of a single set of physical hardware. Today, we have a number of virtualization hypervisors ranging from VMware and the open-source XEN hypervisor.

3.2.3 General Operating System Technologies

Many of the features in the previous section are actually specific solutions that can be applied to more general problems. Some of the problems that operating systems have addressed that are more generally applicable to other systems include:

- Synchronization and deadlock: Operating systems have had to solve the problem of synchronizing access to devices and other shared resources and in the process have created mechanisms such as *semaphores.*[14] Semaphores are variables used to prevent processes from using the same resource at the same time incorrectly. In particular, the way semaphores were approached by Dijkstra (see Dijkstra [1965]) in the THE operating system [Dijkstra 1967] influenced the way others implemented process synchronization using semaphores.[15] When a cycle of processes is each waiting on another process in the cycle to release a resource such that none can proceed, a *deadlock* condition occurs. Detecting and preventing these deadlock conditions was of critical importance to operating systems as not only could a deadlock condition prevent the processes involved from completing, they eventually could starve the entire system of critical resources. As a result, many techniques have been developed to detect and prevent deadlock conditions.
- Buffering and caching: Issues of performance, particularly between input and output devices and the CPU have honed the techniques used for buffering of content and for caching the most-likely-to-be-used content. In particular, this was exacerbated by the different speeds that devices could

13. See presentation by Jim Elliott at a 2004 SHARE users' group, August 17, 2004, http://www.linuxvm.org/Present/SHARE103/S9140jea.pdf.

14. Note other techniques have also been used as hardware solutions such as the *testandset*() operation or using simple software locks.

15. In particular, the *wait*() and the *signal*() operations on semaphores were called *P* and *V* from the Dutch words "proberen" (to attempt) and "verhogen" (to increase), respectively.

communicate and how quickly the CPU could process information. This difference meant that either the CPU had to wait for the device to either deliver or receive data or that a means was needed to make input/output processing more independent of computation. This other means was *buffering.* The ability of a device to pre-load its information so that the CPU could process it or for the device to get its information without delaying the CPU drove the usage of many buffering mechanisms. Many of these mechanisms are implemented in hardware, such as the use of additional processors to manage the I/O[16] or the use of a direct memory access mechanism that allows a device to read and write directly into main memory. Caching of information has been critical to the performance of virtual memory (such as in the working set model) as well as in many other areas of the operating system.

- Scheduling and prioritization: Since the advent of multi-programming (and then, multi-user systems using time sharing), the ability to pick what to do next has been critical to operating systems. As a result, process scheduling and CPU scheduling have a long history of techniques. With only batch-oriented multi-programming systems, the issue is which job to run next and for how long (to completion, until it is waiting for I/O, etc.). With time-sharing systems, the issue is giving each user enough CPU capacity to be useful and to respond to the user within a reasonable amount of time. With real-time operating systems this scheduling needs to consider the time constraints the system must meet. As a result, operating systems have developed a set of techniques and algorithms that are useful in many other scheduling and prioritization problems.
- Fragmentation and re-use of a resource: Main memory is one example where fragmentation occurs and the resource needs to be re-used by many. Another example is the use of magnetic disk storage where files are erased and written, and the resource needs to be re-used efficiently.
- Virtualization and abstraction of a resource: The ability to abstract and generalize an interface has been a goal of operating systems from early on and can be seen in the desire to treat files and devices with the same interfaces (such as in UNIX). IBM's VM/370 (based on IBM's CP-40 and CP-70) product virtualized the physical device. We now see virtualization in networking, storage, and other technologies. These different sorts of technologies lend

16. See the CRAY 1 supercomputer (1977) that would use another computer as a front end as well as the CRAY 1S, which further separated the I/O processing from the rest of the computer. See http://archive.computerhistory.org/resources/text/Cray/Cray.Cray1.1977.102638650.pdf.

themselves to different degrees and kinds of virtualization. For example, disk drives are relatively easy to partition into smaller units, whereas tapes are impractical to virtualize due to their inability to be used by multiple users at the same time.

These sorts of problems have often been addressed first in operating systems due to their criticality in the running of the overall system. Since operating systems software was needed early in the history of software, these sorts of coordination, synchronization, and resource management problems were solved for operating systems first. As a result, other types of software were able to re-use and leverage these techniques from operating systems.

3.2.3.1 Migration of Operating System Features

Features in operating systems, over time, have often followed the various classes of computers. That is, a feature developed in an earlier class of computers (such as mainframes) will often eventually show up in a later class of computers (such as personal computers). The reasons for this are really two-fold:

1. *More power in the newer computer class.* For example, as personal computers became more powerful, they had a need to support multiple processors, multi-tasking, and even time-sharing types of features.
2. *More sophistication and maturity of the usage of the new class.* As the class of computers becomes more widely used and popular, there is a demand for more operating system features. Some of these features just take time to develop and aren't really worth developing until the market has grown enough and there is real demand for more advanced features in the operating system.

So, let's examine a feature like multi-processor support. In mainframes, we saw multi-processor support in the 1960s. This was followed by similar support in minicomputers in the late 1980s. We then saw multi-processor support for personal computers in the mid-1990s and for handheld and mobile computers in the late 2000s. The example of multi-processor support in the operating system is directly tied to the hardware advances for each type of device.

Another example less tied to hardware developments is the support for multiple simultaneous interactive users. We saw this in mainframes in the 1960s with efforts like Dartmouth Time-Sharing System (DTSS), CTSS,[17] and others. For

17. CTSS was termed "compatible" as it was meant to be compatible with IBM's FORTRAN Monitor System (FMS). See Larner [1987]. Interestingly, MIT also had an operating system called ITS, which stood for "Incompatible" Timesharing System, that was used by the MIT artificial intelligence lab during the same time period.

minicomputers this feature became popular in the 1970s. For desktop computers, multiple users were supported in the 1980s though this has never become a very popular way to use personal computers. For mobile and handheld devices, it is unclear that the support for multiple simultaneous interactive users will ever make sense (though one could imagine sharing that processing capacity with local sensors or other devices).

Other features have been so successful that new classes of computers have those features from the start. An example is virtual memory. While virtual memory took some time to perfect in mainframes and minicomputers, almost any new device or class of computer will support virtual memory.

Another way to look at this development of features is to look at a single, portable operating system over time. For example, when the UNIX operating system was developed it was a relatively simple operating system run only on minicomputers. Over time, as UNIX became more popular other features were developed including UNIX on mainframes, UNIX on personal computers, real-time UNIX, multi-processor support, fault-tolerant UNIX, and distributed versions of UNIX.

3.2.4 Influential Operating Systems

A few operating systems are described below. These are included as they were particularly influential in some way to later operating systems. Some of them were commercially successful. Many of them were not commercially successful but implemented ideas that were included in many systems.

- ATLAS—The ATLAS operating system developed at the University of Manchester (see Kilburn et al. [1961]) was one of the earliest systems to contain many of the features we expect from operating systems. In particular, it had a form of virtual memory that it used in order to greatly extend the address space beyond that contained in the core memory. At the time, the system had 16 KB words of core (with 48-bit words). Pages were defined as 512 words each. The system also used page replacement algorithms and kept track of access to pages in memory in order to know if the page had been used recently.
- THE—The THE multi-programming system was developed by a team led by Edsger Dijkstra at Eindhoven University of Technology.[18] THE was a batch system but did support multiple processes using a form of memory segmentation. The system was most noted for its clean design and its separation of the system into distinct layers that were sometimes called an onion-skin

18. Note that "THE" stood for "Technische Hogeschool Eindhoven," the name (in Dutch) of Eindhoven University of Technology at the time.

design. The system made use of semaphores for process synchronization. The layers were structured as follows:

- Layer 0: CPU allocation and interrupts.
- Layer 1: Allocating processes to memory.
- Layer 2: Communication to the system console.
- Layer 3: I/O management.
- Layer 4: User programs for compiling, execution of programs, and printing.

See Dijkstra [1968] for more on the THE operating system.

- MULTICS (Multiplexed Information and Computing Service) was a project started in 1964 as a cooperative effort between General Electric Company, Bell Telephone Laboratories, and Project MAC of MIT. MULTICS is a great example of a commercial failure that had a large impact on the development of operating systems. The intent was to develop a successor to the CTSS operating system (described below).[19]

  MULTICS's goals were not small and were summarized by Corbató and Vyssotsky in their paper initially describing the project as "The overall design goal of the MULTICS system is to create a computing system which is capable of comprehensively meeting almost all of the present and near-future requirements of a large computer service installation." The system was written in the PL/I programming language and designed to be largely machine-independent. It used virtual memory (using segmentation and paging), used a hierarchical file system, and was built to be a time-sharing system. MULTICS influenced operating system development both directly and indirectly, such as through UNIX. Bell Labs had hoped to obtain a usable system for use within Bell Labs, but when it became clear that wasn't going to happen anytime soon Bell Labs decided to drop out in 1969.[20]

19. See http://www.multicians.org/history.html for lots of material about MULTICS and its history.

20. The same people who had been working on MULTICS at Bell Labs started work on UNIX shortly thereafter as this quote from ftp://cm.bell-labs.com/who/dmr/trib/2.html states: "Over time, hope was replaced by frustration as the group effort initially failed to produce an economically useful system. Bell Labs withdrew from the effort in 1969 but a small band of users at Bell Labs Computing Science Research Center in Murray Hill – Ken Thompson, Dennis Ritchie, Doug McIlroy, and J. F. Ossanna – continued to seek the Holy Grail." McIlroy places some of the reason for Bell Labs' exit on the need for the newly formed computation centers to be cost-effective and they're the organization that then owned the computers, rather than Bell Labs Research (per Mahoney interview in 1989, https://www.princeton.edu/~hos/mike/transcripts/mcilroy.htm).

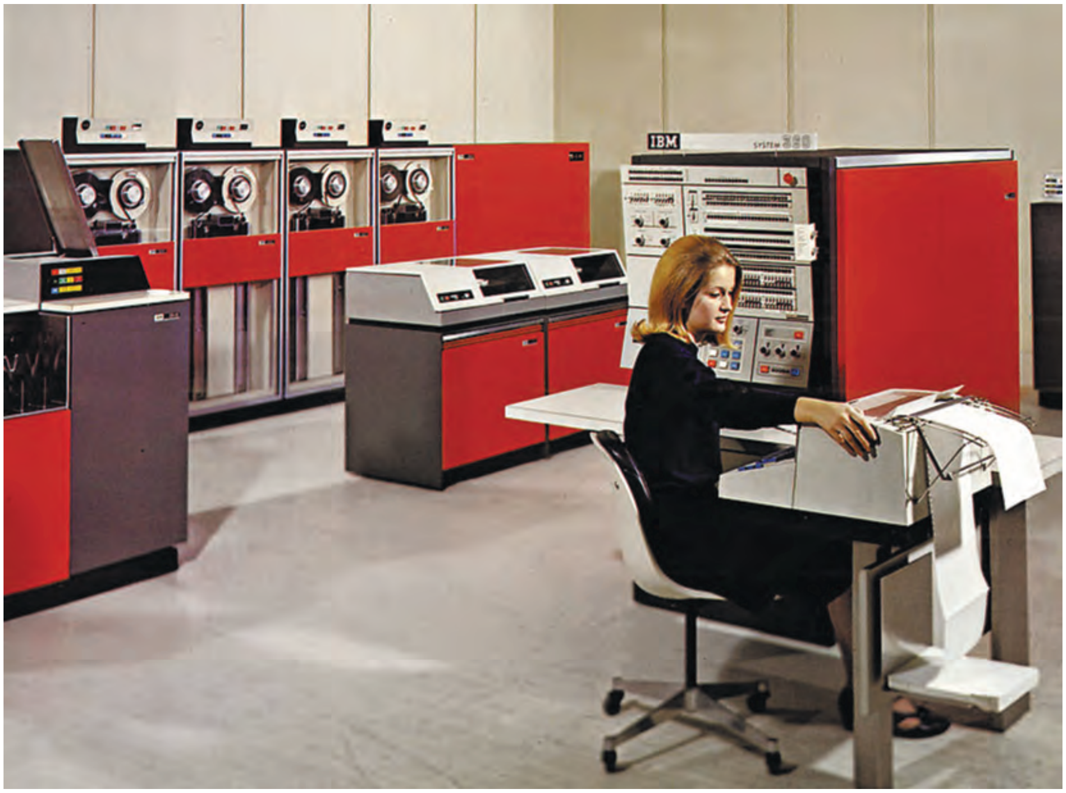


Figure 3.4 IBM System/360 model 40 with woman at console and assorted disk drives, tape drives, and card reader, circa 1964. (Source: Courtesy of International Business Machines Corporation, ©International Business Machines Corporation.)

- The IBM OS/360 (see Figure 3.4 for a picture of the console) was announced in 1964 to cover a family of different-sized computers and was a batch-oriented system. IBM did offer a separate time-sharing operating system (TSS/360, Time Sharing System for the IBM System 360), but it was never widely successful as an official product.[21] OS/360, TSS, and DOS/360 have been put in the public domain and can be found at http://www.ibiblio.org/jmaynard/. One can run these operating systems using the Hercules emulator (see http://www.hercules-390.eu/). OS/360 was viewed as something of a failure at the time, mostly for being over budget and late. Some potential users were

21. TSS is a good example of how difficult it is to discontinue software. AT&T was still running TSS well into the 1990s as it had been used for a particular product that had to be supported for a long time as part of the phone system.

also disappointed by its lack of virtual memory, a feature that had come to be expected by 1964. However, in the long run, the approach of developing a single operating system for a line of computers with application compatibility[22] was very successful for IBM and eventually solidified IBM's position as the leading provider of mainframe, business computing. See Boyer [2004] for IBM's view of the IBM 360 project. Fred Brooks, who managed the development of OS/360, has published many learnings from the System/360 project (and TSS) in Brooks [1975, 1986] and has had significant impact on the practice of software engineering. The CP-40 system introduced the groundbreaking idea of using a virtual machine on the S/360-40 in 1966, which was released externally as CP-67 in 1966. This system separated the control program (CP) that managed the resources of the machine from the individual user environment CMS (Cambridge Monitor System, later re-named the Conversational Monitor System).

- DTSS was an early time-sharing system that became operational in 1964.[23] It is noted as an early large-scale deployment (hundreds of simultaneous users) of time sharing. It is also noted as having been the platform on which the BASIC programming language was first created and deployed as an interactive programming environment. Dartmouth University has a page describing its history with links to simulators to run DTSS (see http://dtss.dartmouth.edu/) as well as an article in *Time* that discusses DTSS and BASIC (see McCracken [2014]). See Figure 3.5 for a photo of students using DTSS for a form of computing dating.
- CTSS was the system developed at MIT before MULTICS and was used as the initial system in developing MULTICS. It was one of the first time-sharing systems, initially demonstrated in 1961. As a result, it influenced many other time-sharing systems including MULTICS and UNIX. CTSS proposed an early email system in late 1964 or early 1965.[24] See Corbató et al. [1962] for an early description of CTSS. CTSS was influenced by other efforts at the time, such as other early time-sharing systems being developed. CTSS not only heavily influenced MULTICS but also UNIX as many of the Bell Labs UNIX developers had worked on MULTICS.

22. One can still run many OS/360 applications in a modern IBM system.

23. DTSS was inspired by a time-sharing system built at BBN on the DEC PDP-1, see Walden and Nickerson [2011].

24. See http://www.multicians.org/thvv/psn-39.pdf for a copy of the proposed CTSS mail command.

Figure 3.5 Members of Dartmouth's Glee Club using DTSS for an early form of computer dating with women from California. (Source: Adrian N. Bouchard/Dartmouth College, Courtesy of Dartmouth College Library.)

- The UNIX operating system was developed initially at Bell Telephone Laboratories in 1969 by some of the same people (see Figure 3.6) who had been working on MULTICS.[25] However, the influence of UNIX has been far beyond just that initial system. UNIX has a long and complex history and has influenced directly and indirectly a large number of systems such as Apple's MacOS (via NeXT NeXTSTEP and the Mach Kernel), Linux (via Minix), and Android (via Linux). A really nice diagram created by Éric Lévénez that shows these complex linkages can be found at http://www.levenez.com/unix/. It covers the history from 1969 up to 2019 (as of this writing). That diagram, even including many UNIX versions, is still not complete and many other

25. Some of the internal Bell Labs' story can be found at ftp://cm.bell-labs.com/who/dmr/trib/2u.html such as finding a little-used DEC PDP-7 to re-write the computer game "Space Travel" that had first been written on MULTICS.

Figure 3.6 UNIX and C pioneers, Dennis Ritchie (standing) and Ken Thompson (sitting) at a PDP-11/20 (undated, likely circa 1972). (Source: Courtesy of AT&T Archives and History Center.)

UNIX versions have been created that are not included in the diagram. So what made UNIX so influential? This can be explained by many factors, partially technical and partially because of the legal constraints placed on it. Bell Labs really had little interest in making money on UNIX and had released it to many universities in the 1970s. However, at the same time AT&T retained strict copyright and trademark restrictions on the source code as well as the name "UNIX." Technically, it was written in a portable language (C) and was relatively easy to port to a number of different machines and architectures. Also, it was relatively simple (compared to MULTICS) and technically appealing to programmers in its simplicity and tool-based approach. The legal restrictions were rigorously enforced by AT&T and, as a result, it was often easier to develop a new version of a UNIX-like system rather than trying to get official permission to use the Bell Labs code.[26] The development of Berkeley UNIX is an example of a major UNIX offshoot.

- RC 4000 Monitor (1969) was mostly known for its architecture that broke down an operating system into a group of interacting programs that used message passing between them to operate the system. The operating system kernel had functions of scheduling the CPU, initiation and control of programs, transfer of messages between program, and initiating data transfers (see Brinch-Hansen [1970] and http://brinch-hansen.net/papers/1969a.pdf).
- Digital Equipment Corporation's TOPS-20 Operating system began in 1969 as the TENEX (TEN-EXtended—it was written for the PDP-10 computer) operating system at BBN (Bolt, Beranek and Newman). TENEX was developed to support the research at BBN, particularly in artificial intelligence. Virtual memory using paging was the primary need that drove the development of TENEX at BBN. Dan Murphy provides a detailed history of TENEX and its move to DEC and renaming to TOPS-20 at http://tenex.opost.com/hbook.html.
- Personal Computer Operating Systems have also had a complex history. Before 1981 and the introduction of the IBM Personal Computer, most personal computers either came with Digital Research's CP/M (Control Program/Monitor) operating system, Apple's operating systems[27] (Apple DOS, ProDOS, or GS/OS for the Apple II), or none at all. CP/M was developed beginning in 1973 by Gary Kildall while at Intel (see Kildall [1980] for his description of how CP/M was developed). CP/M-86 was one of the choices for the initial operating system but licensing concerns delayed the availability of it on IBM PCs (as well as increased the cost), so most IBM PCs were shipped with Microsoft's PC-DOS.[28] CP/M's contribution to computing is noted in an IEEE milestone plaque placed in 2014 that states:

26. The release of *Lions' Commentary on UNIX* [Lions 1976a, 1976b] stimulated interest in UNIX as it contained the source code for UNIX Research Version 6.

27. The Apple I did not have an operating system.

28. Note that University of California–San Diego's p-System was another option for the IBM PC. PC-DOS was based on 86-DOS (or QDOS, Quick and Dirty Operating System) and originally purchased from Seattle Computer by Microsoft.

IEEE Milestone in Electrical Engineering and Computing
The CP/M Microcomputer Operating System, 1974

Dr. Gary A. Kildall demonstrated the first working prototype of CP/M (Control Program for Microcomputers) in Pacific Grove in 1974. Together with his invention of the BIOS (Basic Input Output System). Kildall's operating system allowed a microprocessor-based computer to communicate with a disk drive storage unit and provided an important foundation for the personal computer revolution.

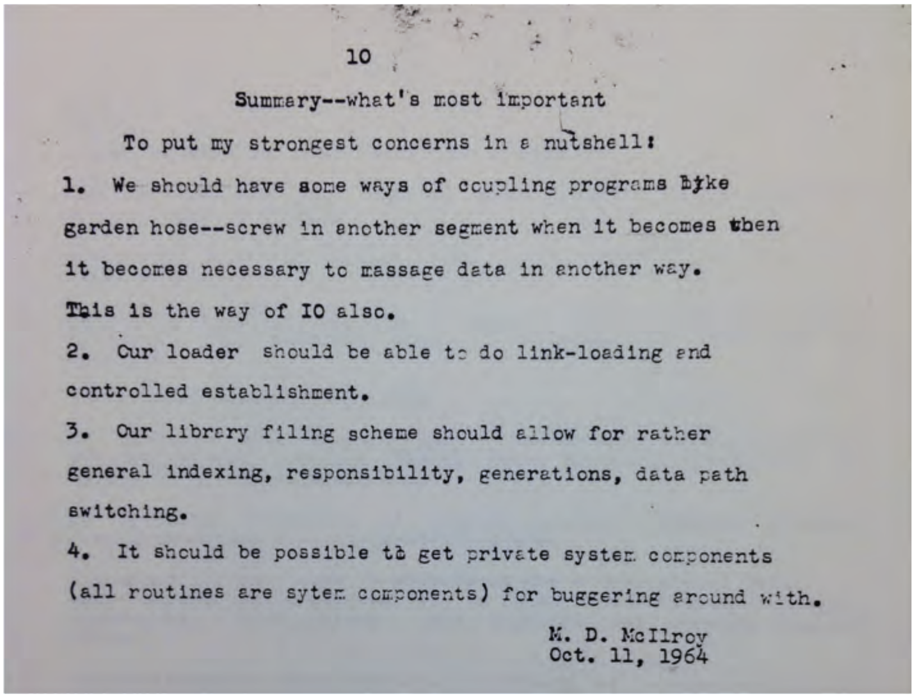
April 2014

Another excellent diagram created by Éric Lévénez for the evolution of Microsoft Windows (and related operating systems) is at http://www.levenez.com/windows/ and details the interrelationships between various versions (including embedded versions, automotive versions, server versions, and Windows mobile).

3.3 Operating Systems Case Study: Pipes in the UNIX System

Pipes or *pipelines* were introduced[29] into the UNIX operating system as a method for inter-process communication and gave a common mechanism that could be used by any program using the UNIX system. The concept is expressed in a 1964 memorandum by Doug McIlroy in Figure 3.7. Effectively, it is an elegant buffering mechanism. A pipe in UNIX is a type of file that does not have permanent storage (like a regular file would) but one that allows two processes to communicate. A pipeline is formed by a shell statement (or by otherwise using pipes to connect the output and input of processes) that connects two or more commands with the binary operation "|". As a simple example, one can use pipes in the Bourne command shell such as in Figure 3.8. In this example, the *who*, *cut*, and *less* commands are strung into a pipeline using the "|" character with the intent of printing out only the login names of everyone that is currently logged onto the system. Piping works

29. Doug McIlroy is credited with the introduction of software pipelining into the UNIX system and a posting by Dennis Ritchie (see http://cm.bell-labs.com/cm/cs/who/dmr/mdmpipe.html from Ritchie [1984]) shows the goals that Doug McIlroy expressed in 1964 for pipes.


10

Summary--what's most important

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes then it becomes necessary to massage data in another way. This is the way of IO also.

2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

4. It should be possible to get private system components (all routines are sytem components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964

Figure 3.7 Doug McIlroy's 1964 Bell Labs memorandum summary page with goals for pipes and operating systems. (Source: Image courtesy of the Charles Babbage Institute Archives, University of Minnesota Libraries, Minneapolis.)

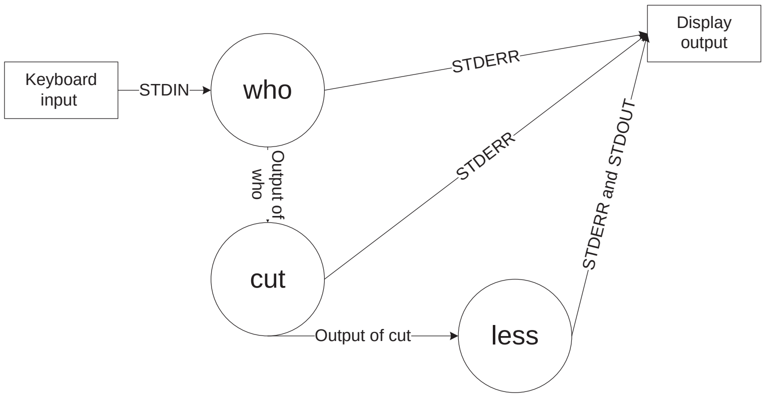


Figure 3.8 Pipes default operation example in a Bourne shell.

by taking the initial standard input (STDIN) from the terminal and then taking the standard output (STDOUT) from each command and feeding it the STDIN of the next one in the pipeline. Standard error (STDERR) feeds by default to the terminal display.

```
who | cut -f1 -d"␣" | less
```

So why was the introduction of pipes into UNIX important? They were really introduced to make it easier for the programmer to use the system. The origin of pipes goes back at least to 1964, as Dennis Ritchie refers to an October 11, 1964, memo of concerns from M. Doug McIlroy: “1. We should have some ways of connecting programs like garden hose—screw in another segment when it becomes necessary to massage data in another way. This is the way of IO also.”[30] Pipes have since been used in some form by many operating systems. Many of the tools developed for use on UNIX systems were developed with the idea of pipes in mind. As a result, there are many small tools that do one task very well and when linked together via pipes with other tools produce a very powerful and flexible way to accomplish large tasks. Tools such as *grep* (general regular expression parser), *sed* (stream editor), and *awk*[31] (text processing, interpretive programming language) are excellent examples of this “do one thing well” approach.

The introduction of pipes in the UNIX operating system was implemented in 1972[32] and this version is from 1979 (Bell Labs Research version 7). Pipes not only made shell programming more elegant, they were an integral part of the operating system and philosophy of UNIX. See Appendix—Source Code, Section A.1 for the source code listing of pipe.c.

This code (Section A.1) for pipes is fairly typical of UNIX code at the time. The intent was that it be readable and understandable. In fact, courses teaching UNIX internals within Bell Labs would use the source code to teach how UNIX operated along with books on UNIX design such as Bach [1986]. In order to read the code, there are a number of environmental factors to consider. First, UNIX had been re-written almost completely in the C programming language. At about this same

30. Note that this is the precise wording from the memo with the duplicate phrase “when it becomes. “IO” refers to Input/Output. See http://cm.bell-labs.com/cm/cs/who/dmr/mdmpipe.pdf.

31. Stands for “Aho, Weinberger, and Kernighan,” the authors of the language.

32. See http://cm.bell-labs.com/cm/cs/who/dmr/hist.html for more on the history of pipes in UNIX. As noted on this site, DTSS did something similar to pipes.

time, UNIX was being shared with a number of universities.[33] Additionally, there was an explicit project within Bell Labs to make UNIX portable.[34] All of these factors led to the need to make the code portable, readable, and consistent with the UNIX, tool-based mindset.

So, looking at the source code file for pipes (A.1, line 17), one sees that the only constant that is locally defined is the size of the pipe. The rest are all from system header files included as shown in A.1, lines 1 to 7. The remainder of the code is concerned with reading, writing, and locking the pipe to prevent concurrent access. Fairly consistently in UNIX source code, *fp* is used to refer to file pointer, with "i-nodes" (short for "index node") being used as the data structure to define files.

3.4 Lessons Learned from Operating System Software

A number of lessons have been learned from the creation and modification of operating systems that are generally applicable to other software systems. These include the following:

- *Sharing resources efficiently*: A fundamental aspect of operating systems is that they manage the resources of the machine with the expectation that this will increase the utility of the machine. As a result, one of the lessons that has been learned is how to manage those resources efficiently, particularly for shared resources such as memory and disk drives. This not only makes operating systems more efficient but has been used as models for how to share other computing resources such as for database transactions and other higher-level software systems. An example is how fragmentation of these shared resources, such as memory and disk storage, is managed so that the continued re-use of them doesn't result in pieces that are of sizes that aren't easily re-used.
- *Managing time and speed differences*: The necessity of an operating system to manage difference in timing and to synchronize work has resulted in a number of techniques that have general applicability in software systems. Operating systems have had to manage differences in speed between devices such as the difference in speed of RAM, disk drives, and other storage

33. UNIX was shared with universities with a number of restrictions on its use. This is part of what encouraged Richard Stallman, then at MIT, to start the GNU (GNU's Not UNIX) project.

34. Note that Wollongong University ported Research version 6 to an Interdata 7/32 computer in 1977. Bell Labs also ported a version to an Interdata 8/32 computer at about the same time. Princeton University also adapted UNIX to run on an IBM VM/370 in a virtual machine. See Johnson and Ritchie [1978] for more.

devices. This resulted in refinement of techniques such as caching that are used in many other kinds of software outside of operating systems.

- *Portability is important*: The ability to run the same operating system on different computers allowed operating systems to have longer lives and for functionality to be added in a more permanent manner than in the past. This involved adding the ability to abstract from the hardware to minimize the changes necessary to get the operating system to run on additional computer types. Why was this important? Because it allowed other software built on top of that operating system to not have to substantially change when a new computer was needed.
- *Platform stability engenders other advances*: Related to portability of operating systems is the resulting stability of the operating system platform and interfaces that can more effectively be maintained over a long period of time over different types and generations of computer equipment. The relative stability of IBM's System/360, UNIX, and Microsoft Windows across generations of computer hardware has helped enable other software to also be relatively stable as well freed up programming resources that might have otherwise been spent on porting software between different computers.
- *Virtualization is a key tool for abstraction*: Virtualization, which first emerged in the context of operating systems with IBM, has proven to be a very useful tool for the abstraction of many types of computing system components including storage, networking, and even data centers. This has led to the ability to build on those abstracted components so that the underlying hardware can more easily be replaced. It has also led to enhanced stability in those higher-level, virtualized abstractions that allow them to have a longer life cycle.
- *Ability to scale*: One of the first places where software has needed to scale to larger environments has been in the area of operating systems. As a result, the design of operating systems has evolved so that the same operating system can be run on computers that range in power from small devices to supercomputers. This was a key principle in the design of IBM's System/360 operating system and has become true with operating systems such as UNIX and Linux as they were ported to a number of different environments and devices. In order to scale to very large computers, operating systems had to be designed to support thousands of concurrent users, many different configurations of peripherals, and to work with many other systems over a network.

All of these lessons have been applied to other software systems.

3.5 Summary

Hundreds (and likely thousands) of operating systems have been written, and as systems become more powerful, the more work operating systems are allowed to perform. Clearly, this chapter is just the tip of the iceberg of information about operating systems history. Early operating systems had to be extremely modest in what they performed and not waste system resources. As systems became more powerful, operating systems allowed multiple programs to simultaneously use the computer (multi-programming), multiple simultaneous users (time sharing), and eventually to include additional sophisticated graphical user interfaces and application functionality, such as web browsers.

A number of operating systems have had long-lasting impact on operating system development, re-use of concepts, and re-use of source code including UNIX, MULTICS, CTSS, DTSS, THE, and many others. These are briefly described in this chapter with some of their impact. The features in operating systems have evolved to become much more sophisticated and many are now taken for granted as being part of almost any modern operating system (such as virtual memory, process synchronization).

Operating systems are not only software that makes the computer easier to use, but through their development, we've learned how to solve problems of synchronization, performance, and reliability that is also applicable to other software systems.

3.6 Exercises and Projects

3.6.1 Exercises

1. Real-time operating systems have some of the oldest roots. Examine the real-time features and requirements of the MIT Whirlwind I computer system. Identify what made them real-time requirements.
2. The SAGE (Semi-Automatic Ground Environment) system also was built with a number of real-time requirements. Identify five of the real-time requirements.
3. A feature called the "Fair Share Scheduler" was developed for the UNIX operating system (see Henry [1984] and Kay and Lauder [1988]). This feature allowed the processor capacity to be shared among groups and prevented one group from taking more than their established percentage. Investigate the origins of this feature and describe why it was an important development for UNIX at the time.

4. Before virtual memory was a standard feature of operating systems, a system problem was *thrashing.* Thrashing occurs when processes (or parts of processes such as segments or pages) are swapped in and out of memory to the extent that that's what the system is spending most of its time doing, rather than doing effective work in running processes. Peter Denning (see "Before Memory was Virtual" in Glass [1998, pp. 250–271]) describes how the problem of thrashing was alleviated by development of a "Working Set Model" of pages required in a multi-programmed system. Explain the working set model, how it alleviated thrashing, and how it could apply to other system problems such as storage.
5. Microkernels have been used several times in an attempt to simplify the architecture of operating systems. The Mach operating system developed at Carnegie Mellon University was a particularly influential system designed as a microkernel. Determine what modern operating systems were influenced by the Mach kernel.
6. TinyOS (http://www.tinyos.net/) is an open-source operating system designed for very small devices such as sensory networks. Investigate the current feature set and postulate what operating system features might be useful when those small devices have more processing capacity.
7. Plan 9 was a distributed operating system introduced by Bell Labs in 1992 and included a grid computing platform for using a set of distributed computers together to solve large problems. Compare Plan 9 to the GLOBUS Toolkit for grid computing. Explain why GLOBUS Toolkit was more widely used for grid computing than Plan 9.
8. Investigate the features for the Apple II operating system, Apple DOS 3.1 (June 1978), and compare its features to those available with CP/M at the time.
9. Berkeley System Distribution (BSD) UNIX was used as the basis for the first versions of the SUN Microsystems SunOS operating system. Explain why this version was chosen and used rather than some other version of UNIX or other operating system.
10. The Cray Research Cray 1 supercomputer used an operating system called COS, Cray Operating System. Describe the unique features of this operating system. COS (version 1.17) is available at https://archive.org/details/Cos1.17DiskImageForCray-1x-mp and the COS 1.0 (1978) manual at http://www.bitsavers.org/pdf/cray/COS/2240011E_Cray-OS_Ver_1.0_Reference_Jul78.pdf.

11. Investigate the personal computer operating system OS/2, originally a joint development between IBM and Microsoft. Determine how OS/2 was intended to be better than PC DOS and why it did not survive.
12. The Burroughs MCP operating system was the first to be written in a higher-level, system-programming language. Investigate why Burroughs decided to implement MCP in 1961 in ESPOL (Executive Systems Programming Language, an extension of ALGOL).
13. *Lions' Commentary on UNIX 6th Edition, with Source Code* [Lions 1976a, 1976b] was important in the popularization of the UNIX operating system and was often illegally copied. Explain how this book influenced the spread of UNIX. Find a copy of this book and the part of the UNIX source code (in slp.c) that has a comment of "You are not expected to understand this." Explain why this comment was reasonable according to Dennis Ritchie.
14. The Inferno operating system was developed in 1995 as a distributed operating system that followed Plan 9 and used a virtual machine (called *Dis*) and introduced a programming language called *Limbo*.[35] Inferno did not survive as a commercially viable operating system. Find out why it did not succeed and identify what succeeded instead.
15. An interesting story is the development of many different operating systems for the Control Data Corporation CDC 6600.[36] The CDC 6600 was one of the fastest machines of its time and supported multiple CPUs. Be sure to cover the Chippewa (aka COS), SIPROS, SCOPE, MACE, and Kronos operating systems for the 6600 and describe why there were so many operating systems for this machine. Wikipedia and a CDC document (at http://bitsavers.org/pdf/cdc/cyber/CDC_Operating_System_History_Mar76.pdf will give an excellent start along with the references in the footnote.
16. Investigate the type of operating system being used for the Apple Watch device (being called WatchOS, as of this writing). How is it different from the operating system being used for larger devices such as on iPhones and iPads? What operating system features are distinct from the other Apple platforms?

35. Inferno is now released as free software and maintained by Vita Nuova. The source code can be found at: https://code.google.com/p/inferno-os/.

36. The manuals for the hardware and the Chippewa, Kronos, and SIPROS operating systems can be found at http://www.bitsavers.org/pdf/cdc/cyber/cyber_70/. The SCOPE operating system manuals are at http://www.bitsavers.org/pdf/cdc/cyber/scope/. The CAL Timesharing System as well as source code are at http://www.mcjones.org/CalTSS/.

17. Find an example outside of operating systems where a shared re-usable resource is managed between many users (similar to memory or magnetic disk storage in operating systems software). Was it managed in a manner similar to the way operating systems manage it? Is there any evidence that they borrowed their techniques from what was learned in the development of operating systems software?
18. Making operating systems portable has reduced the variety of operating systems that are in mainstream use. Explain why this has been the case and whether the variety of operating systems is likely to increase or decrease in the future. Please be sure to justify your reasoning.
19. While the RC 4000 Monitor operating system (see Brinch-Hansen [1970, 1973]) was largely a commercial failure, explain how it influenced future operating systems and what modern-day operating systems still show its influence.
20. This chapter states that the stability of operating systems was a great boon to software. This was in contrast to when the hardware was changing drastically every few years as well as having very different operating systems provided by each computer manufacturer. Explain how rapidly changing hardware and operating systems limited the ability to create higher-level, stable software layers.
21. Find out the relationship between MIT AI Lab's ITS (Incompatible Timesharing System) and the GNU (GNU's Not UNIX) project. How did the culture and usage environment of ITS encourage projects like GNU?

3.6.2 Projects

1. In Hoare [1974], C.A.R. Hoare proposes using the monitor concept based on Brinch-Hansen's concept of monitor. Show how Hoare's monitor concept built on Brinch-Hansen's concept and how it influenced further operating system design. Another reference that may be helpful is Brinch-Hansen [2001]. Are there concepts in Hoare's paper that did not get adopted by the general operating system community? Explain why. What concepts were adopted and why?
2. Research the "PenPoint" Operating System developed by GO Corporation in the early 1990s. Compare this to more modern tablet and device operating systems (such as Apple iOS and Google Android) and determine which features of this operating system were influential.

3. The Berkeley Timesharing System (later called the Scientific Data Systems, SDS, Timesharing System) began development in 1964 at the University of California, Berkeley. Investigate the internals of the Berkeley Time-Sharing System and how it compared to contemporary time-sharing systems such as DTSS at Dartmouth and CTSS at MIT. Were there features of the system that were different from these other systems? How did this system affect the development of the TENEX operating system at BBN and UNIX at Bell Labs?
4. Using the emulator provided at https://github.com/pkimpel/retro-b5500, build a running version of Burroughs MCP operating system. Using that system, compile and run a small program in ALGOL. Prepare documentation and a presentation that describes your experiences with MCP and how to use it.
5. Many times, programs that were once considered applications have become part of the operating system. A classic example of this is the web browser where it was created as an application but is now (especially in personal computers and mobile devices) expected to be delivered as part of the operating system. Google's Chrome OS provides the Chrome browser as the interface to the user. Explore what other applications have been absorbed by the operating system and examine the pluses and minuses of each application's inclusion as part of the OS.
6. Download and install Bell Labs' Plan 9 operating system and install within a VM. See http://plan9.bell-labs.com/plan9/download.html for the software. Run the system and compare its features and performance to a version of Linux using the same size of virtual machine.
7. Download and install the Haiku operating system as a virtual machine. Investigate the features and determine the differences with other free operating systems such as Linux. Is there any compelling reason for anyone to use Haiku? When would such an operating system be useful?
8. Databases need to schedule transactions and also need to support sharing of the database. Examine the history of these techniques and explain how operating systems did (or didn't) influence database transaction scheduling and concurrency.
9. The use of *threads* in operating systems as a way to support "light-weight processes" has a long history and are now part of almost every modern operating system. Investigate the progression of threads from early implementations such as in Mesa (see Lampson and Redell [1980]) and the Thoth system

(see Cheriton [1982]) to how threads were implemented in UNIX (see Valhalia [1996]), Linux, Microsoft Windows, and other operating systems.

10. Examine the ability of operating systems to be *fault tolerant*. That is, having the ability to withstand failures and continue to operate. Determine where this sort of feature was first developed and how it became more sophisticated over time and used in different systems.
11. Use the Hercules emulator (see http://www.hercules-390.eu/) in order to run and install an OS/360 operating system. Compile and run a short program. Document and prepare a presentation that shows what had to be learned in order to accomplish this task.
12. Investigate the history of Microsoft's Windows CE for automotive embedded operating system. Map the evolution of its features and how those features exist (or not) in other Microsoft operating systems.
13. Investigate the "Direct Couple" extension to IBSYS for the IBM 7090. Document how the features of Direct Couple influenced features of IBM OS/360. See http://www.softwarepreservation.org/projects/os/dc.html for a starting point.
14. Time-sharing in operating systems went through a definition phase as do many new technologies. It was recognized by the early 1960s in many places as an idea worth pursuing. Investigate the many variants of time-sharing that were proposed before 1965. Use resources such as Trimble [1968] and Bullynck [2019] to find the various efforts that were undertaken for time-sharing. Determine the scope of the variants and produce a paper that shows what ideas won and lost in the effort to define time-sharing. Be sure to include early efforts not otherwise mentioned in this chapter such as the Ferranti Orion monitor system where time-sharing was proposed in 1961 (see Goodman [1961]) as well as university-led projects like CTSS, ITS, DTSS, and SDS.

3.7 Further Readings and Online Resources

Operating systems have a long and complex history and a good place to begin is in current operating systems textbooks such as Silberschatz et al. [2012] with a substantial section on operating system history. Bullynck [2019] provides a look at early systems and how they evolved through the 1950s and 1960s. In the last decade or so, many old operating systems have been open sourced, and the source code can be found online. These include old versions of UNIX, MULTICS, OS/360, and many others. Rosen [1967] includes a collection of papers describing several important early operating systems including CTSS, OS/360, MULTICS, and the ATLAS Supervisor. Additionally, simulators to run these old systems have been built and many are available online for free (such as http://www.hercules-390.eu/forOS/360 and https://code.google.com/p/retro-b5500/ for Burroughs's MCP). Binaries for some operating systems are available at BitSavers.org such as TENEX (see http://www.bitsavers.org/bits/BBN/Tenex/).

4

Programming Languages

Programming languages have a rich and highly interrelated history. This chapter deals with how compilers and interpreters evolved as well as concepts such as object-oriented programming. Programming language history is the most well-documented areas of software history, partially because programming languages are so central to the creation of software and the primary tool of programmers. Programming languages are the media that programmers use in order to implement algorithms and solve problems, making them key to the way programmers interact with computers. As a result, there have been a number of conferences devoted to the history of programming languages (see Wexelblat [1981], Bergin and Gibson [1996], and Hailpern [2007]).[1]

Since the late 1950s, thousands of programming languages have been developed, most of which are no longer in widespread use. However, a few of those early programming languages remain in widespread use. Languages such as FORTRAN, COBOL, and LISP were some of the earliest programming languages and they continue to be used. Other languages that were popular for a period of time have not remained in widespread use, such as BCPL, SNOBOL, PL/I, and ALGOL. Yet many of those languages have had a substantial impact on later programming languages. In this chapter, we will look at a number of languages and why they continued to be used or why they fell into disuse. Some readers of this text may not have heard of these influential languages as they are rarely covered in their courses, including modern courses on programming languages.

Figure 4.1 shows the logo used for the first ACM conference in 1978 for the History of Programming Languages. Some of the languages included there continue to survive in some form, while the others are no longer in use but were unique and/or influential.

1. A nice summary of the ACM History of Programming Languages (HOPL) conferences is in Bergin [2007]. Jean Sammet was also instrumental in the documenting of the early history of programming languages, such as in Sammet [1969] and also see Donald Knuth's [2003] summary of the early history.

Figure 4.1 Logo of the History of Programming Languages I Conference in 1978 with the languages represented. (Source: SIGPLAN Notices, Volume 13, Number 8, August 1978 ©ACM 1978, New York, NY.)

4.1 Definitions

What is and what is not a programming language is surprisingly not well-defined. Some people include any method that could be used to write a program including machine code and assembly language. In this book, we'll define a programming language along the same lines as Jean Sammet [1969]. The term *higher-level languages* will be considered equivalent to the term *programming languages* for the purpose of this book.

Programming languages will be defined to have the following characteristics:[2]

- *Machine code knowledge is not necessary.* Programming languages should be at a higher-level than machine code and abstract from the machine. Furthermore, a programmer using a programming language should not need to know anything about the machine-level instructions and architecture of the computer on which the programs are run.
- *Potential for conversion (i.e., porting) to other computers.* Similar to the need not to know the machine code of the computer you are using, a program written in a programming language should be portable to other computers.

2. These are the same four characteristics used in Sammet [1969]. However, here we won't be quite as rigid as Sammet.

- *Instruction explosion.* This requirement refers to the ability of a higher-level programming language to be translated into many more machine instructions. That is, it should be able to express programs more concisely than machine code. So, when a program is compiled in a programming language it should create many more machine instructions.
- *Problem-oriented notation.* This last requirement is the hardest to rigorously decide whether or not a given programming language meets. The idea is that it should be closer to human language so that someone reading the program can understand the problem being solved and the algorithm used to do it without a lot of detailed knowledge of the particular computer or machine code.

The first three of these requirements are trying to force a programming language to be more abstract than the machine language level of the computer. The last requirement is looking for another definition of "high-level" that makes a program more understandable by humans and closer to a problem domain. So, some early languages such as FORTRAN, COBOL, and LISP clearly met this definition, which other early languages such at IT (Internal Translator), A-0, A-1, A-2, IBM Speedcoding, and Univac Short Code are much less clear.[3]

4.1.1 Machine Language

Machine language[4] is the direct numerical language that the computer uses to operate. This is usually binary and specifies the hardware instruction to be used as well as the data for the instruction. As an example, the Microprocessor without Interlocked Pipeline Stages (MIPS) architecture has machine code instructions that are always 32 bits long and of the form of a 6-bit operation code (opcode) followed by arguments in the following forms depending on the type of instruction (R-type, I-type or J-type) as given in Table 4.1. MIPS, (see Patterson and Hennessy [2013]) being a relatively simple and regular machine instruction set, makes a good example. MIPS is designed for reduced instruction set computers (RISC) so its instruction set was designed to be simple. It has three formats for machine instructions: R-type (register instructions), I-type (Immediate), and J-type (jumps). These three

3. However, many of these early programming languages did have significant dependencies on the specific machine, where the word length might affect types such as INTEGER and REAL as well as the system libraries and often the language implementation itself was different on different machines.

4. Sometimes called a "first generation language" for programming; however, it is not a programming language by the definition for programming and high-level languages in Section 4.1.

Table 4.1 **MIPS machine code instruction types**

Instruction Type	opcode					
#bits	6	5	5	5	5	6
R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address/immediate		
J-type	op	target address				

formats[5] are all 32 bits long and their formats are specified in Table 4.1. Instruction sets and machine architectures vary widely. In MIPS machine language, *rs*, *rt*, and *rd* refer to registers values. *shamt* refers to a shift amount. *funct* is used to specify a specific operation in an R-type instruction.

A specific example of an instruction for a MIPS processor is an R-type instruction with an opcode of zero that will add the contents of register 1 (binary 00001) and 3 (binary 00011) and put the result in register 7 (binary 00111). Binary 100000 (as the "funct" equal to decimal 32) indicates addition for the R-type instruction.

```
000000 00001 00011 00111 00000 100000
```

Listing 4.1 A binary R-type instruction for a MIPS processor.

4.1.2 Assembly Language

An *assembler* generally takes code written with mnemonics representing machine instructions and translates that into machine code that is executable.[6] These mnemonics generally are one-to-one representations of the machine instructions (i.e., no code explosion) and form *assembly language.* So, to represent the MIPS add instruction one uses:

```
li   $t1, 1          # $t1 = 1  ("load immediate")
li   $t3, 3          # $t3 = 3
li   $t7, 7          # $t7 = 7
add  $t1,$t3,$t7     # Adds register 1 to register 3 = register 7
```

Listing 4.2 MIPS assembly instructions.

This corresponds to the MIPS machine code listed above, but I added the use of variables ($t1, $t3, and $t7) to store the values of the register numbers to be used in the *add* instruction. Assemblers added readability by the mnemonics (like *add* and *li*) but also in the use of memory location names and variable names.

5. Here, we only talk about the 32-bit MIPS format, not other varieties of MIPS.

6. These are sometimes called second generation languages, with machine language being 1st generation.

4.1.3 Compilers and Automatic Coding and Programming

The notion of a programming language compiler was not precisely defined during the early years of computers until FORTRAN was released for the IBM 704. There was a great deal of activity to create those initial compilers as is covered in Section 4.3. In particular, there was a drive towards creating *automatic coding*, often also called *automatic programming.* This drive was to make it possible for regular users of the machine to precisely define their problems such that they could be solved by the computer by automatic coding or programming. This eventually became called "compilers," but later efforts to make programming more automatic continued.[7] We will define *compilers* to be "the program(s) that transform source code written in a programming language into object or machine code."

Part of that drive for automatic coding came from the recognition that the timely creation of programs was becoming a bottleneck for effective use of computers. Coding directly in binary or even assembly was difficult to master and involved significant translation from an algorithmic level of problem solving. So, the desire was to allow more people to be able to produce programs. Another key driver came from the recognition that many of these solutions would be needed on many different types of computers and having to reprogram a known solution for each machine needing it was redundant effort that a portable, higher-level form of automatic coding would help solve. One way to show the change in programming languages to a higher level of abstraction is in the diagram in Figure 4.2. Machine code can be called first generation programming, with assembly language programming second generation, and third generation programming languages referring to general-purpose languages such as Java or C. Fourth generation can

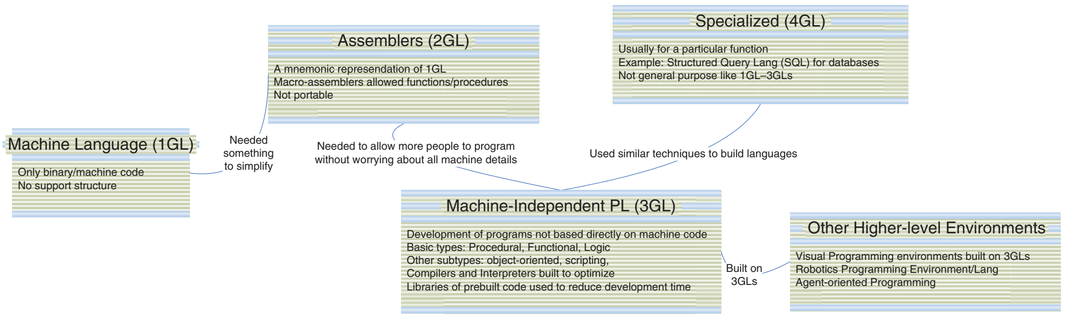

Machine Language (1GL)
Only binary/machine code
No support structure
Needed something to simplify
Assemblers (2GL)
A mnemonic represendation of 1GL
Macro-assemblers allowed functions/procedures
Not portable
Needed to allow more people to program without worrying about all machine details
Specialized (4GL)
Usually for a particular function
Example: Structured Query Lang (SQL) for databases
Not general purpose like 1GL–3GLs
Used similar techniques to build languages
Machine-Independent PL (3GL)
Development of programs not based directly on machine code
Basic types: Procedural, Functional, Logic
Other subtypes: object-oriented, scripting,
Compilers and Interpreters built to optimize
Libraries of prebuilt code used to reduce development time
Built on 3GLs
Other Higher-level Environments
Visual Programming environments built on 3GLs
Robotics Programming Environment/Lang
Agent-oriented Programming

Figure 4.2 Generations of programming languages and environments.

7. See Chapter 5 on programming environments and tools where efforts such as computer-aided software engineering (CASE) are discussed.

refer to higher-level languages that serve a given purpose, such as Structured Query Language (SQL). Additionally, there are other types of languages that take a higher-level abstraction such as agent-oriented programming to create intelligent agents. Additionally, Japan started a *Fifth Generation Computer Systems* project in 1982 (see Shapiro [1983]) that was meant to leverage massively parallel computing and the use of concurrent logic programming. This project's name loosely built on the existing four levels of programming levels of programming languages and three levels of hardware and was intended to produce a dramatic step forward. Another project that termed itself as "fifth generation" was the Kronos Research Group's Kronos computer that began in 1984 and focused on building a machine that would be well-suited to running the Modula-2 programming language.

4.1.4 Source Code, Object Code, Linkers, Loaders, Libraries, and Executables

Programs and programming languages are part of an overall system to produce running processes in a computer. In this section, we define those terms that are central to compiling a program. In Figure 4.3, we have the process beginning with source code, which is what most programmers will consider their *program,* which consists of the lines of code they have written in the programming language. The compiler's job is to translate this program into machine code. In general, the program does not contain all the functions and routines directly in the program itself,

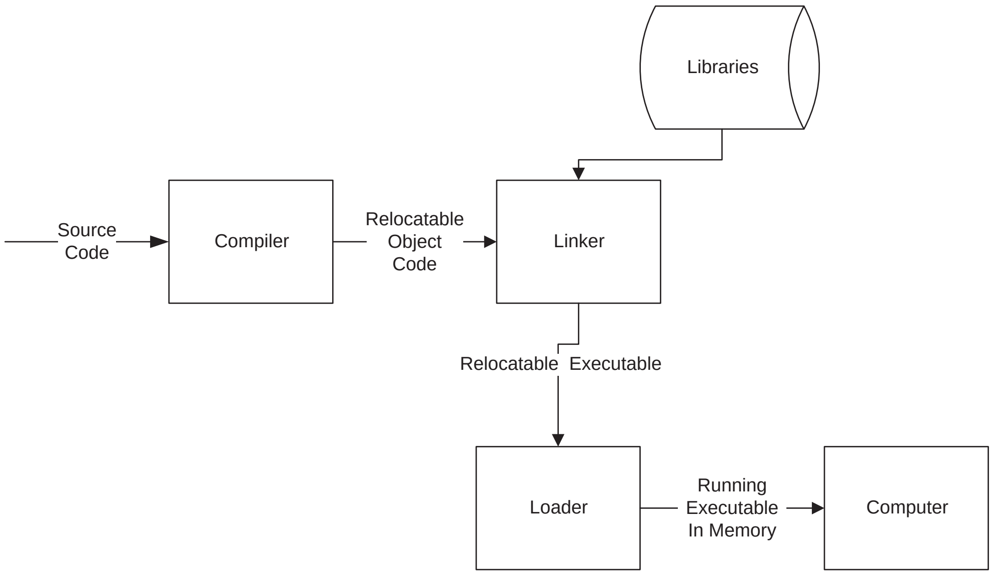


Figure 4.3 Flow of compiling a program to become a process running in memory.

but instead relies on libraries of prewritten and compiled routines that can be included in their program. The object code (also stored as a file) will be machine code that is *relocatable.* Being relocatable means that it can be placed somewhere in memory, and when it is the references to memory addresses will be modified so that they refer to the actual (or virtual memory) addresses the program needs to run. The *linker* is used to resolve the references to external libraries and system calls. The *loader* is the program used to load the relocatable executable to the computer's memory so that the operating system can execute the program. The loader will resolve the memory address references so that the program's executable can run.

4.2 Types of Programming Languages

There are many ways to categorize programming languages. One of the most common is to categorize them by *programming paradigm.* A programming paradigm is a fundamental way of building the structure and elements of computer programs. The four programming paradigms most commonly described are: imperative/procedural, logic, functional, and object-oriented.[8] In this section we will use these four programming paradigms. However, other dimensions are also valuable in looking at programming languages, such as how declarative or imperative the language is. A declarative language specifies *what* to solve, not *how* to solve it. An imperative language specifies how the program will be executed. For example, logic paradigm-based languages (such as Prolog) tend to be more declarative rather than a language such as C++, which is imperative.

Using programming paradigms certainly does not give a complete or even a necessarily clear taxonomy of programming languages. Many languages support multiple programming paradigms (such as C++ supports object-oriented and by supporting C programs, more purely procedural paradigms). It is not clear what precisely makes up a programming paradigm and when a new one should be declared as importantly distinct from the existing paradigms. Also, one can often use a programming language designed for one paradigm and use it in another paradigm. For example, a framework can be built in C++ to implement logic-based programming. With that said, paradigms still give a rough idea of the diversity of programming languages. Each of the next four sections describe these paradigms

8. Note that our focus here is on Turing-complete languages that can implement any program that could run on a Turing machine. So, we are not including languages that are less general in this section, such as data definition and query languages like SQL (for databases) that are not Turing complete.

and describe several of the more important examples from that paradigm and why each was important.

4.2.1 Imperative and Procedural Languages

The earliest programming languages were all imperative, meaning that a program would dictate *how* the problem is solved via a set of statements that change the program state. The program statements implement an algorithm to solve the problem at hand. Being a *procedural* programming language refers to programs being written subdivided into parts referred as procedures, methods, routines, or subroutines. Procedural programming is often associated with structured programming techniques but is really older than the advent of structured programming. Most programming languages tend to be of this category.

4.2.1.1 FORTRAN

FORTRAN (FORmula TRANslation), being one of the first programming languages, has been very influential.[9] It grew out of the desire to be able to express problems and algorithms in mathematical notation.[10] IBM assembled a team led by John Backus to ensure that the compiler would work correctly and be efficient enough to ensure acceptance. The original release of the initial version of FORTRAN was started in 1954 and was released for the IBM 704 in spring 1956. FORTRAN II was released in 1958 and included the ability to define subroutines. FORTRAN III was never fully developed, but FORTRAN IV was implemented in 1962 for the IBM 7090. Also in 1962, the American Standards Association (a precursor to the current American National Standards Institute, ANSI) began the process of developing a standard for FORTRAN using FORTRAN IV. FORTRAN has continued to evolve with other standards including FORTRAN 66, FORTRAN 77, FORTRAN 90, FORTRAN 95, FORTRAN 2003, FORTRAN 2008, and FORTRAN 2018. FORTRAN continues to be used widely, particularly among the scientific and numeric computing community.

```
C     Example program in FORTRAN 2
      DIMENSION X(20), RCIPX(20)
      SUMX=0.
      DO 5 I=1, 20
      IF (X(I)) 3,2,3
    2 RCIPX(I)=0.
```

9. See http://softwarepreservation.org/projects/FORTRAN/ for much more on the history of FORTRAN.

10. An influential effort in the movement toward expressing problems in mathematical notation (though not directly influencing FORTRAN) was the Internal Translator (IT) project at Purdue and then Carnegie Mellon (see Chipps et al. [1956]).

```
7        GO TO 5
8      3 RCIPX(I)=1./X(I)
9      5 SUMX=SUMX+RCIPX(I)
10       STOP
```

Listing 4.3 FORTRAN II example using an arithmetic *if* statement.

The code above is from FORTRAN II. In it, you'll note on line 5 an *if* statement of the form:

IF (expression) LINE1, LINE2, LINE3

Here, if the *expression* results in a negative value, then control is transferred to *LINE1*. If it's zero, then control is transferred to *LINE2* and if it's positive to *LINE3*. So, in this example, if the *X(I)* is either negative or positive we go to line 3 and if it's zero to line 2. This form of "arithmetic if" was a primary control structure for programming in FORTRAN II and was replaced with an IF–THEN structure in later versions of FORTRAN. Debugging other people's programs in FORTRAN II was very difficult.[11] FORTRAN was written with an assumption of being encoded on punched cards and used the columns on the card for formatting the FORTRAN source code. FORTRAN (for many versions) expected certain columns to be used for specific uses. The first column could contain a "C" to represent a comment (as in line 1 above). The next 4 columns would represent the line number ("2," "3," and "5" in the above) and the 5th column if filled with a "*" would allow a statement to continue to the next line.

4.2.1.2 COBOL

COBOL (COmmon Business-Oriented Language) was formed as a result of the Conference/Committee on Data Systems Languages (CODASYL). CODASYL was formed in 1959 to guide the development of a standard business programming language. The US government and industry partnered to form CODASYL out of an interest to create a portable programming language. The first ANSI standard for COBOL was adopted in 1968 with the first CODASYL version being defined in 1960. Three primary languages were considered as options for what would become COBOL: FLOW-MATIC, AIMACO, and COMTRAN. FLOW-MATIC was being developed at Sperry Rand with Grace Murray Hopper. FLOW-MATIC supported long variable names and had an English-like syntax.[12] COMTRAN was being developed at IBM

11. The author notes that he had the task early in his career of debugging a large FORTRAN II program without any comments and found it extremely difficult to debug and enhance.

12. B-0 (which was renamed FLOW-MATIC by Sperry Rand) was the business-oriented language that Grace Murray Hopper worked on while at Remington Rand for the UNIVAC. Named B-0 as

with Bob Bemer. Some features from COMTRAN were included in the eventual standard for COBOL.

CODASYL went on to develop other standards, most notably the CODASYL data model for network data model databases. The source code listing below is for a "Hello World" program ran on the Hercules OS/370 simulator on Linux. In that example, you can see the verbose nature of COBOL and some of the warnings. This example also includes Job Control Language (JCL) cards at the beginning and end of the COBOL listing.

```
//COBUCLG   JOB  (001),'COBOL BASE TEST',
//                 CLASS=A,MSGCLASS=A,MSGLEVEL=(1,1)
//BASETEST  EXEC  COBUCLG

//COB.SYSIN  DD  *

 00000* VALIDATION OF BASE COBOL INSTALL

 01000 IDENTIFICATION DIVISION.

 01100 PROGRAM-ID.  'HELLO'.

 02000 ENVIRONMENT DIVISION.

 02100 CONFIGURATION SECTION.
 02110 SOURCE-COMPUTER.   GNULINUX.
 02120 OBJECT-COMPUTER.   HERCULES.
 02200 SPECIAL-NAMES.
 02210       CONSOLE IS CONSL.

 03000 DATA DIVISION.

 04000 PROCEDURE DIVISION.
 04100 00-MAIN.

 04110       DISPLAY 'HELLO, WORLD' UPON CONSL.
 04900       STOP RUN.

//LKED.SYSLIB  DD  DSNAME=SYS1.COBLIB,DISP=SHR

//                DD  DSNAME=SYS1.LINKLIB,DISP=SHR
```

her earlier languages were A-0, A-1, and A-2. The design of B-0 was done on flowcharts written on drafting blueprint paper (as had been the case since the ENIAC and at Eckert–Mauchly Computer Corp) and filled hundreds of pages.

```
//GO.SYSPRINT DD SYSOUT=A

//
```

Listing 4.4 COBOL program with Job Control.

The output of that same COBOL program is given below, which generates some errors such as expecting cards to define the card punch device.

```
19.52.48 JOB    3  $HASP100 COBUCLG  ON READER1      COBOL BASE TEST
19.52.48 JOB    3  IEF677I WARNING MESSAGE(S) FOR JOB COBUCLG  ISSUED
19.52.48 JOB    3  $HASP373 COBUCLG  STARTED - INIT  1 - CLASS A - SYS
     BSP1
19.52.48 JOB    3  IEC130I SYSPUNCH DD STATEMENT MISSING
19.52.48 JOB    3  IEC130I SYSLIB   DD STATEMENT MISSING
19.52.48 JOB    3  IEC130I SYSPUNCH DD STATEMENT MISSING
19.52.48 JOB    3  IEFACTRT - Stepname  Procstep  Program   Retcode
19.52.48 JOB    3  COBUCLG    BASETEST  COB       IKFCBL00  RC= 0000
19.52.48 JOB    3  COBUCLG    BASETEST  LKED      IEWL      RC= 0000
19.52.48 JOB    3  +HELLO, WORLD
19.52.48 JOB    3  COBUCLG    BASETEST  GO        PGM=*.DD  RC= 0000
19.52.48 JOB    3  $HASP395 COBUCLG  ENDED
```

Listing 4.5 COBOL program output.

COBOL continues to evolve with numerous standards since its initial definition in 1959 to 1960 (called COBOL 60). Other standards include COBOL-65, COBOL-68, COBOL-74, COBOL-85, COBOL 2002, and COBOL 2014. In 2002, COBOL added the ability to support object-oriented programming. As a result of its large, embedded base and continued evolution, COBOL is still being used by many companies around the world.

4.2.1.3 ALGOL

In 1958 in Zurich, a committee met to create an international algebraic language using the emerging techniques for programming language design. Originally called IAL (International Algebraic Language), the ALGOL 58 version of ALGOL (ALGOrithmic Language) was created. In 1960, this committee met again to create a new version of ALGOL called ALGOL 60 that significantly revised the version from 1958 and became the basis for the ALGOL standard going forward. ALGOL never supplanted FORTRAN in the United States but did gain significant popularity in Europe through the 1960s and 1970s. ALGOL has probably had more influence on procedural languages than any other single language. ALGOL was defined by a precise syntax using Backus–Naur Form (BNF) and introduced notions of block structure and scope of names. It also included features such as dynamic storage allocation and recursive procedures.

Some of the languages directly or indirectly influenced by ALGOL include Pascal, Modula, C, Java, PL/I, B, ICON, and many others. ALGOL has had a lasting impact on procedural programming. The code sample below shows how readable ALGOL is as so many current-day languages borrowed from its syntax.

```
procedure Absmax(a) Size:(n, m) Result:(y) Subscripts:(i, k);
    value n, m; array a; integer n, m, i, k; real y;
comment The absolute greatest element of the matrix a,
    of size n by m  is transferred to y, and
    the subscripts of this element to i and k;
begin
    integer p, q;
    y := 0; i := k := 1;
    for p := 1 step 1 until n do
        for q := 1 step 1 until m do
            if abs(a[p, q]) > y then
                begin y := abs(a[p, q]);
                    i := p; k := q
                end
end Absmax
```

Listing 4.6 An ALGOL program to get the absolute greatest element of a matrix.

4.2.1.4 PL/I

PL/I (Programming Language One) is a language that was proposed by the SHARE users group (as New Programming Language) and developed by IBM in 1965. The intent was to develop a more modern language that could possibly take the place of both FORTRAN and COBOL. As a result, PL/I included a number of features to make it easier to interface with IBM file systems as well as advanced process synchronization mechanisms. Through the 1960s, PL/I compilers were often inefficient and unreliable. PL/I was a complex language and compilers were relatively difficult to build. The Project MAC group at MIT decided to adopt PL/I for the MULTICS operating system project. In addition, PL/I adopted many of the well-liked features from ALGOL including its block structure, recursion support, and many other features. The PL/I standard language semantics were expressed using formal operational semantics (see Wegner [1972]) using an abstract machine with translation rules. These were named the Vienna Definition Language (VDL), as VDL was designed by IBM in Vienna.

PL/I's syntax drew heavily from ALGOL as you can see in the code below.

```
/* Read in a line, which contains a string,
/* and then print every subsequent line
/* that contains that string. */

find_strings: procedure options (main);
   declare pattern character (100) varying;
   declare line    character (100) varying;
   declare line_no fixed binary;

   on endfile (sysin) stop;

   get edit (pattern) (L);
   line_no = 1;
   do forever;
      get edit (line) (L);
      if index(line, pattern) > 0 then
         put skip list (line_no, line);
      line_no = line_no + 1;
   end;
```

Listing 4.7 A very small program in PL/I.

4.2.1.5 C

The programming language C's development has been intimately tied with that of the UNIX operating system. The UNIX development team had recently been working on Multics, which was using PL/I to develop the Multics operating system. Prior to that, they had been working on the CTSS operating system at MIT that used the MAD programming language. While UNIX was originally developed using assembly language, UNIX was completely rewritten in C by 1972. UNIX was also built to be tuned to the needs of programmers, so an efficient compiler was needed. C was developed based on earlier experiences from the B programming language, BCPL, and MAD. The popularity of UNIX and Linux has spurred the popularity of C as well. It was standardized in 1989 by ANSI. Below is a simple program that generates a random number and asks a user to guess the value when given hints of "high" or "low."

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
/
```

```
/*   Guessing game with no parameters
 *      generates a random number between
 *      1 and 20 (inclusive) and then
 *      responds with "too high" or "too low" or
 *      "correct" and number of guesses taken /**/
#define DAYS 7

int main(void) {
        int guess;
        int guess_times = 0;
        /*
         * use the system time to set the randomizer seed value
         * use rand() function to get a random number
         * and then take modulo 20
         * add one to make it between 1 and 20
         */
        srand(time(NULL));
        int r = rand() % 20; r++;
        /*
         *Give a prompt to start the game.
         */
        printf("We\'re starting the guessing game! Enter your \
guesses for a number between 1 and 20 (inclusive).\n");
        /*
         * Loop forever until they guess
         * Note they may guess more than 20 times as they might
         *  not be paying attention to their previous guesses
         */
        while (0 < 1) {
                printf("What\'s your guess of the number?\n");
                /* get their guess */
                scanf("%d", &guess);
                /* how many times they have guessed  */
                guess_times++;
                /* if their guess is correct, then say so,
                    give them their count of guesses and exit */
                if (guess == r) {
                        printf("Correct! What\'s you\'re secret?\n");
                        printf("You made %d guesses.\n", guess_times);
                        return(0);
                }
                /* otherwise check if too high or too low
                 * and spit out hint */
                else if (guess > r) printf("Too High!\n");
                      else printf("Too Low!\n");
        }
        return 0;
}
```

Listing 4.8 A simple guessing game written in C.

4.2.2 Functional Programming Languages

Languages supporting the functional paradigm really began with LISP in 1958. LISP has since splintered into many variants, including MIT's Scheme along with Common LISP. Other, more recently developed languages such as ML (Meta Language) and Haskell are also functional in nature.

4.2.2.1 LISP Dialects

Functional programming languages began with John McCarthy and the definition of LISP. LISP's syntax is distinctly different from languages such as FORTRAN or ALGOL, being fully parenthesized prefix notation.[13] LISP stood for "LISt Processing" and was built to manipulate lists. Even LISP programs are lists of items that can be manipulated by other LISP programs. LISP was influenced by the Information Processing Language developed by Newell, Shaw, and Simon a couple of years earlier. LISP was built to handle symbolic (rather than numeric) information in order to be useful for artificial intelligence programming. LISP continues to enjoy usage in the artificial intelligence community, particularly in the United States.

LISP has many different variants including Common LISP, Scheme, and Racket.

The following LISP program is by John McCarthy and recovered from files in his archives held at Stanford University. It uses what he called "pure" LISP with only functional constructs.

```
;;; block3.lsp[f83,jmc] Block stacker with fn calls and opportunism
;;; The opportunism consists of moving a block to its final
    destination
;;; instead of the table during clear operations if this is possible.
```

13. A myth is that LISP was directly inspired by Alonzo Church's lambda calculus, but McCarthy clarified that in the History of Programming Language I conference in 1978 [Wexelblat 1981] and stated at that conference:

> Now, having borrowed this notation, one of the myths concerning LISP that people think up or invent for themselves becomes apparent, and that is that LISP is somehow a realization of the lambda calculus, or that was the intention. The truth is that I didn't understand the lambda calculus, really.

```
(defun build (structure s)
       (if (null structure)
           s
           (build (cdr structure)
                   (build1 structure (reverse (car structure)) 'table
    s))))
(defun build1 (st rtower location s)
       (if (null rtower)
           s
           (build1 st (cdr rtower) (car rtower)
                  (move st (car rtower) location s))))
(defun move (st block location s)
       (if (on block location (car s))
           s
           (immove block
                   location
                   (clear st block (clear st location s)))))
(defun immove (block location s)
       (cons (update
              (car s)
              (list block location))
             (cons (list block location) (cdr s))))
(defun clear (st block s)
       (if (or (null block) (eq block 'table))
           s
           (clear1 st block (find block (car s)) s)))
(defun update1 (s1 pair)
       (cond
        ((or (null s1) (and (null (car pair)) (null (cadr pair))))
         s1)
        ((eq (caar s1) (car pair))
         (cons (cdar s1) (update1 (cdr s1) pair)))
        ((eq (caar s1) (cadr pair))
         (cons (cons (car pair) (car s1))
               (update1 (cdr s1) (list (car pair) nil))))
        (t
         (cons (car s1) (update1 (cdr s1) pair)))))
(defun update (s1 pair)
       (update2 (if (eq (cadr pair) 'table)
                    (cons (list (car pair)) (update1 s1 (cons (car
    pair) nil)))
                    (update1 s1 pair))))
(defun update2 (s1) (cond
                     ((null s1) nil)
                     ((null (car s1)) (cdr s1))
```

```lisp
                     (t (cons (car s1) (update2 (cdr s1))))))
(defun find (b s1) (if (member b (car s1)) (car s1) (find b (cdr s1)))
    )
(defun clear1 (st b tower s)
       (if (eq b (car tower))
           s
           (clear1
            st
            b
            (cdr tower)
            (immove
             (car tower)
             ((lambda (w) (if (member w (car s)) (car w) 'table))
              (dest (car tower) st))
             s))))
(defun dest (b st) (dest2 b (dest1 b st)))
(defun dest1 (b st) (if (member b (car st)) (car st) (dest1 b (cdr st)
    )))
(defun dest2 (b tower) (if (eq (car tower) b)
                           (if (null (cdr tower)) 'table (cdr tower))
                           (dest2 b (cdr tower))))
(defun isclear (b st) (eq b (car (dest1 b st))))
(defun on (a b s1) (on1 a b (find a s1)))
(defun on1 (a b tower)
       (and (not (null tower))
            (or (and (eq (car tower) a)
                     (or (and (eq b 'table) (null (cdr tower)))
                         (and (not (null (cdr tower))) (eq (cadr tower
    ) b))))
                (on1 a b (cdr tower)))))

;;; tests
(setq t1 '((a b) (c)))
(setq t2 '((a b c)))
(setq s0 (cons t1 nil))
(setq tt0 '(b c))
(immove 'a 'c s0)
(move t2 'a 'c s0)
(immove 'a 'table s0)
(build1 t2 '(c) 'a s0)
(build1 t2 '(c b) 'table s0)
(build t2 s0)
(setq t3 '((a b c) (d e) (f)))
(setq t4 '((a b c d f) (e)))
(build t4 (cons t3 nil))
```

```
(setq t5 '((c b) (a d e) (f)))
(build t5 (cons t3 nil))
(setq t6 '((c b) (a d) (e f)))
(build t6 (cons t3 nil))
```

Listing 4.9 LISP program by John McCarthy for rules on moving blocks in a Blocks World.

4.2.3 Logic Programming Languages

Prolog[14] was originally developed circa 1973 and is considered a declarative language where one specifies formal logic statements that are then fed to a specialized theorem prover in order to generate conclusions. So, a *declarative* programming language specifies the problem to be solved rather than how to solve it. Other declarative logic programming languages have been developed based on concepts from Prolog including Gödel, ALF, and Datalog. Prolog was written to be an interpreter when a question is asked, and the Prolog theorem prover then tries to find an instance where it is true by instantiating variables. For example, in the code below are a number of Prolog predicates. Note that the program is composed of facts that end in a period with no ":-". Rules contain the ":-", which is read as "if."

In the example Prolog program given below, the program is run by asking "main" to be proven which will try to prove the items on the right-hand-side (*solve_canibal, reverse,* and *show_states*) to solve the missionaries and cannibals problem as a search problem. This particular problem was commonly used in artificial intelligence such as by Saul Amarel to show various ways of representing knowledge [Amarel 1968]. The missionaries and cannibals problem is a classic logic problem involving making sure the cannibals never outnumber the missionaries when they try to cross the river in a boat. There are three missionaries and three cannibals on one side of the river, and they wish to cross the river in a boat that holds one or two people. The problem is to cross the river and never violate the rule (i.e., never have more cannibals than missionaries on either side of the river or the boat). The set of successful states are put into the *Solution* list and then reversed to get the proper ordering once it is found. Prolog also is used mostly with recursive predicates.

```
% CANNIBAL.PL
% This program solves the cannibals and missionaries puzzle.

main :-
        solve_cannibal([state(3,3,l)],Solution),
        reverse(Solution,[],OrderedSolution),
```

14. Please see http://www.softwarepreservation.org/projects/prolog for more on Prolog history.

```
        show_states(OrderedSolution).

%
% solve_cannibal(+Sofar,-Solution)
%   searches for a Solution to the cannibals and missionaries
%   puzzle that extends the sequence of states in Sofar.
%

solve_cannibal([state(0,0,r)|PriorStates],
               [state(0,0,r)|PriorStates]).
solve_cannibal([state(M1,C1,l)|PriorStates],Solution) :-
        member([M,C],[[0,1],[1,0],[1,1],[0,2],[2,0]]),
          % One or two persons cross the river.
        M1 >= M,
        C1 >= C,
          % The number of persons crossing the river is
          % limited to the number on the left bank.
        M2 is M1 - M,
        C2 is C1 - C,
          % The number of persons remaining on the left bank
          % is decreased by the number that cross the river.
        member([M2,C2],[[3,_],[0,_],[N,N]]),
          % The missionaries are not outnumbered on either
          % bank after the crossing.
        not member(state(M2,C2,r),PriorStates),
          % No earlier state is repeated.
        solve_cannibal([state(M2,C2,r),state(M1,C1,l)|PriorStates],
    Solution).
solve_cannibal([state(M1,C1,r)|PriorStates],Solution) :-
        member([M,C],[[0,1],[1,0],[1,1],[0,2],[2,0]]),
          % One or two persons cross the river.
        3 - M1 >= M,
        3 - C1 >= C,
          % The number of persons crossing the river is
          % limited to the number on the right bank.
        M2 is M1 + M,
        C2 is C1 + C,
          % The number of persons remaining on the right bank
          % is decreased by the number that cross the river.
        member([M2,C2],[[3,_],[0,_],[N,N]]),
          % The missionaries are not outnumbered on either
          % bank after the crossing.
        not member(state(M2,C2,l),PriorStates),
          % No earlier state is repeated.
        solve_cannibal([state(M2,C2,l),state(M1,C1,r)
                    |PriorStates],Solution).

show_states([]).
show_states([state(M,C,Location)|LaterStates]) :-
          write_n_times('M',M),
          write_n_times('C',C),
          N is 6 - M - C,
          write_n_times('␣',N),
          draw_boat(Location),
          MM is 3 - M,
          CC is 3 - C,
          write_n_times('M',MM),
          write_n_times('C',CC),
          nl,
          show_states(LaterStates).

write_n_times(Item,0) :- !.
write_n_times(Item,N) :-
          write(Item),
          M is N - 1,
          write_n_times(Item,M).

draw_boat(l) :- write('␣(____)␣␣␣␣␣').
draw_boat(r) :- write('␣␣␣␣␣(____)␣').

member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).

reverse([],List,List).
reverse([X|Tail],SoFar,List) :-
          reverse(Tail,[X|SoFar],List).
```

Listing 4.10 The missionaries and cannibals problem in Prolog—a classic AI problem.

4.2.4 Object-Oriented Programming Languages

Major languages supporting an object-oriented programming paradigm were Simula I in 1964, followed by Simula 67 in 1967. Simula 67 introduced many of the concepts considered central to object-oriented programming including objects, classes, a class hierarchy, and inheritance. The next major language to take up an object orientation was Smalltalk in 1971. In the early 1980s, there was a push to move development towards an object-oriented approach. As a result, many languages started adding object-oriented extensions resulting in languages such as C++, Objective C, C with Classes, and Common LISP Object System (CLOS). In addition to the development of object-oriented programming languages, object-oriented system analysis and design began to change how systems were defined. Other technologies such as databases to store objects (sometimes called object bases) were developed to specifically support systems built with object-oriented programming languages. In the 2000s, enthusiasm waned somewhat as new development methodologies emerged (such as Agile) and object-oriented databases were reabsorbed back into largely relational databases.

4.2.4.1 Simula

The Simula I programming language was developed as a language to support simulations by Ole-Johan Dahl and Kristen Nygaard in the early 1960s in support of discrete event simulations. Most modern object-oriented languages take a similar approach to the object approach taken by the version developed in 1967 called Simula 67. An example of Simula 67 code from Dahl et al. [1970] includes classes, subclasses, and virtual functions in the following listing:[15]

```
Begin
   Class MyClass;
      Virtual: Procedure print Is Procedure print;
   Begin
   End;

   MyClass Class Char (char);
      Character char;
   Begin
      Procedure print;
        OutChar(char);
   End;

   MyClass Class Line (elements);
      Ref (MyClass) Array elements;
   Begin
      Procedure print;
      Begin
         Integer i;
         For i:= 1 Step 1 Until UpperBound (elements, 1) Do
            elements (i).print;
         OutImage;
      End;
   End;
```

15. See http://www.edelweb.fr/Simula/ for a scan of the "Common Base Language" by Dahl, Myhrhaug, and Nygaard, Norwegian Computing Center, Publication No. S-22, October 1970.

```

   Ref (MyClass) myitem;
   Ref (MyClass) Array myitems (1 : 3);

   ! Main program;
   myitems (1):- New Char ('z');
   myitems (2):- New Char ('y');
   myitems (3):- New Char ('y');
   myitem:- New Line (myitems);
   myitem.print;
End;
```

Listing 4.11 Classes in Simula 67.

It should be noted that SIMULA 67 shares many of the same features as ALGOL and the intent was to make SIMULA "ALGOL-based." For an excellent study of how SIMULA came about, see Holmevik [1994].

4.2.4.2 Smalltalk

Smalltalk is an object-oriented programming language that was developed in the 1970s by Alan Kay and his team at Xerox Palo Alto Research Center (PARC). It stemmed from work in "human–computer symbiosis" with Doug Englebart developing new ideas on how computers could augment a human's ability to get work done.[16] Below is a Smalltalk 71 program from Kay [1993], defining a factorial function and a membership function:

```
to 'factorial' 0 is 1
to 'factorial' :n do 'n*factorial n-1'

to 'fact' :n do 'to 'fact' n do factorial n. ^ fact n'

to :e 'is-member-of' [] do F
to :e 'is-member-of' :group
     do'if e = firstof group then T else e is-member-of rest of group'
```

Listing 4.12 A factorial example in Smalltalk.

Note how the functions are recursive here and with objects being used as variables (with a ":" in front). An important aspect of Smalltalk is that the programming environment is integral to using Smalltalk and is modified by the objects that are created. Squeak and Pharo are modern, open-source variants that also have integrated development environments.

16. See Kay [1993] for a good history of how work in Smalltalk tied to and was influenced by other earlier work.

4.2.4.3 Java

Java was developed as a pure object-oriented language, as opposed to a language like C++ that was developed as an extension to a non-object-oriented language, C. Jim Gosling, Mike Sheridan, and Patrick Naughton at Sun Microsystems started Java in 1991 as a language called Oak, which was being developed for set-top boxes. Eventually, Java 1.0 was released in 1995. Java used the Java Virtual Machine (aka JVM) and rather than compiling to machine code, compiled Java to "byte code," designed to run on the JVM. So, Java was built to have extremely portable code that would port to any device that would run the JVM. This was branded as "Write Once, Run Anywhere." Java became one of the most popular programming languages, partially due to the language itself and partially due to the timing of Java's release. Being released in 1995, web programmers were searching for a programming environment that could be used on a variety of devices. When the JVM was included in Internet browsers, this made Java a natural choice for developing web-based applications.

Today, Java remains one of the most popular programming languages and is most commonly taught in high schools and as the core language in many university computer science programs.

Below is a small excerpt of an Oak program, which shows the influence of earlier procedural languages as well as using the object-oriented concepts in SIMULA 67. See FirstPerson [1994].

```
Class Foo {
        int x;
        float y;
        Foo()   { x=0; y=0.0; }
        Foo(int a)  { x=a; y=0.0; }
        Foo(float a) { x=0; y=a; }
        Foo(int a, float b) { x=a; y=b; }
}
Foo obj1 = new Foo();
Foo obj2 = new Foo(4);
Foo obj3 = new Foo(4.0);
Foo obj4 = new Foo(4, 4.0);
```

Listing 4.13 A trivial example in Oak (precursor to Java).

This excerpt from the Oak specification shows how similar Oak was to what eventually became Java. The example shows the different constructors that are used based on the parameters used to create the new object.

4.3 Prehistory of Programming Languages and Compilers

The history of the development of compilers and compiler technology is worthy of a book in its own right. Here, we will briefly discuss the background required to understand later sections as well as the influence compilers had on the development of programming languages.

4.3.1 Automatic Coding and Programming

It had been recognized since at least the days of Babbage that machine translation from human-readable algorithms to machine-executable instructions was desirable. A quote from Babbage written in his journal on July 9, 1836, reads (see Randell [1973] and Knuth [2003, p. 5]):

> This day I had for the first time a general but very indistinct conception of the possibility of making an engine work out *algebraic* developments. I mean without any reference to the *value* of the letters. My notion is that as the cards (Jacquards) of the Calc. engine direct a series of operations and then recommence with the first so it might perhaps be possible to cause the same cards to punch others equivalent to any given number of repetitions. But there hole [*sic*] might perhaps be small pieces of formulae previously made by the first cards.

While this quote from Babbage shows that the hopes for an easier way to program a computer had existed for some time, when actual computers were developed the means and mechanism for doing this were by no means obvious or easy. The effort to develop programming languages and compilers was substantial. A few of those efforts are described here.[17] Significantly, these developments span more than a decade and involve significant efforts from many large organizations. Also significant, these efforts involved many different regions of the world.

Much of the effort was in how to specify algorithms in an unambiguous manner that could be automatically translated. Efforts such as Zuse's Plankalkül were way ahead of their time and not implemented when conceived, and, in Zuse's case, not influential at the time.[18] Doing so took over a decade from 1945 to the late 1950s to culminate in the development of FORTRAN. After this period, the number and variety of languages exploded into a large variety of languages including LISP and the effort to develop COBOL as a standard business-oriented programming language.

17. This section is derived from Knuth [2003, p. 76], where he also gives subjective ratings of each of these as well as detailed descriptions of them and their importance.

18. Zuse's work was quite extensive but wasn't officially published until 1972.

Many of these early efforts are not compilers in the sense defined in Section 4.1. Instead, they represent incremental improvements. The first compiler in the sense of translating algebraic statements into machine language was Glennie's AUTOCODE (see Knuth [2003, p. 34–35] and Glennie [1952]).[19] Other efforts at around the same time (such as Hopper's A-0 and Backus's IBM 701 Speedcoding System) were more focused on assembling the parts such as preexisting code for functions and procedures and producing an executable more quickly. However, Glennie's AUTOCODE appears to have had little influence at the time on other efforts.

One of the earliest efforts to have a long-lasting impact on programming was Goldstine and von Neumann's Flow Diagrams (see Goldstine and von Neumann [1948]). In reviewing many early programs from the 1950s, one will see a pervasive use of flow diagrams and flowcharts to design these programs. Goldstine and von Neumann's flow diagrams set a common framework for designing programs.

Early programming languages were not only struggling to define what a compiler should do but also what would make a useful programming language. Knuth [2003] rates these early languages and compilers by several factors including readability, control structures, data structures, and machine independence. These early languages varied widely by those criteria and in their influence and application. Another interesting example that was influential was Perlis's Internal Translator (IT) programming language in 1956 which produced a successful compiler that also influenced the development of ALGOL. Many of these early languages would link together libraries of prebuilt modules and had more of the feel of a linker than actual compiler.

An interesting source of effort in compilers was the compiler effort at Remington Rand. While Remington Rand's compiler effort (led by Grace Murray Hopper) put a tremendous amount of effort into their compilers, many of those efforts had little long-lasting impact. One such effort was the development of the MATH-MATIC programming language (aka Algebraic Translator 3, AT-3) led by Katz at Remington Rand, circa 1956. The A-0, A-1, A-2, and A-3 (aka ARITH-MATIC) compilers were earlier compilers also by Hopper's group at Remington Rand that were really more like macro-expanders that were more like macro-assemblers than compilers as we think of them today. By the time of MATH-MATIC, other efforts such as FORTRAN were taking hold as efficient compilers. Another line of compilers at Remington Rand did have a long-lasting influence. This was the business-oriented

19. Note that Alick E. Glennie's AUTOCODE is generally prefaced with "Glennie's" as another slightly later AUTOCODE (1954) was by Brooker. Both AUTOCODEs were built for the Manchester Mark I computer.

compiler FLOW-MATIC (aka Business Language Version 0, B-0), which was one of the primary influences on COBOL and the CODASYL standard for COBOL.

A key factor for progress in automatic coding and compilers was to develop compilers that were able to produce executable machine code that was close to what could be achieved by an excellent assembly language programmer. Because computers were so expensive and machine time so expensive, any significant degradation of performance by a compiler would significantly reduce its acceptance. At the same time, the desire to increase the number of available programmers and to make them more efficient drove the desire to build usable compilers. As a result, there were a number of concurrent efforts to build efficient and usable compilers in the mid- to late-1950s, many of which are well-described and analyzed in Knuth [2003].

Compiler development continued after this initial period with many successful programming languages being developed outside of computer manufacturers. After the definition of ALGOL in 1958, a number of variants were supported that became popular in their own right. Two examples of these were the MAD (Michigan Algorithm Decoder) and JOVIAL (Jules Own Version of the International Algebraic Language). MAD was developed in 1959 and was widely used for writing operating systems such as CTSS and the Michigan Terminal System (MTS). JOVIAL was developed in 1959 at the System Development Corporation. JOVIAL was extensively used for real-time applications in the defense industry and eventually was standardized in 1973 with the standard MIL-STD-1589.

Many examples of the quest for automatic coding and being able to generate systems more easily without highly technical knowledge continue to this day. Some more modern-day examples are using computer-aided software engineering (CASE) tools where part of the goal was to generate working systems from system requirements, or using techniques such as model-driven development. While these techniques have proven useful for many systems, they are often oversold and have significant limitations on increases in productivity or the system types for which they are applicable.

4.4 Influences on Programming Language Change

The changes that have occurred to programming languages involves a broad set of factors, some of which are in Figure 4.4. One of the factors driving change in programming languages is the need to solve particular problems better, that is, application needs. The drive to create languages such as LISP or SNOBOL came from needing the ability to process lists and strings. For LISP, it was also a need to address problems in artificial intelligence. Computer science has been an enabler for creating the science around formal language and compiler design that

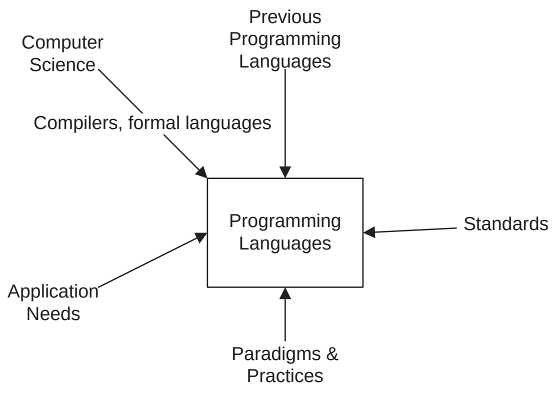


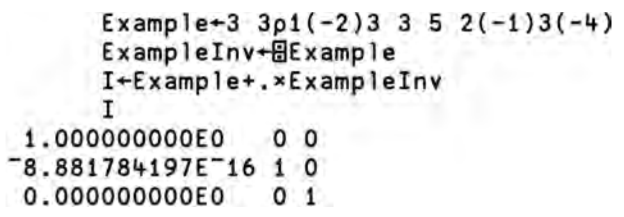
Figure 4.4 Influences on programming language change.

has made compiler development easier. Clearly, previous programming languages have had significant influence on future languages. ALGOL's influence shows up in a wide number of programming languages. Programming language standards have helped programs become more portable. Paradigms and practices have also influenced new programming language design. A clear example was that with the advent of object-oriented programming, object-oriented programming was added to a number of existing languages such as C (to C++ and Objective C), COBOL, and LISP.

Another influence has been the support of the vendor community in supporting a given language. With PL/I, IBM supported PL/I for many years and without that support it is not likely that it would have survived as long as it did. With Java, the support from Sun Microsystems helped to entrench Java as a programming language.

4.5 Case Study: APL

Ken Iverson's APL\360 for the IBM/360 is best described as a revolutionary language that took a very different approach to programming and programming languages. Initially described in Iverson's 1962 book (see Iverson [1962]), APL\360 was conceived of first as a mathematical notation by Iverson in 1956. Eventually, this matrix-oriented notation became known as "Iverson's Notation." The source code for APL\360 is available through the Computer History Museum at https://computerhistory.org/blog/the-apl-programming-language-source-code/. The system is written entirely in 360 Assembly language and took control of the entire machine to provide an interactive environment through its included timesharing operating system. See also the implementation of APL\360 using the Hercules emulator at http://wotho.ethz.ch/mvt4apl-2.00/.



```
      Example←3 3⍴1(-2)3 3 5 2(-1)3(-4)
      ExampleInv←⌹Example
      I←Example+.×ExampleInv
      I
 1.000000000E0    0 0
¯8.881784197E¯16 1 0
 0.000000000E0    0 1
```

Figure 4.5 A simple example of the use of APL to invert a 3×3 matrix.

A simple example of the use of APL is in Figure 4.5. This example inverts a three-by-three matrix. In the first line, the matrix elements (1, −2, 3, 3, 5, 2, −1, 3, and 4) are reshaped by the ρ character into an array with dimensions of 3×3. Operations are performed from right to left by default. So, one would read the first line of create a matrix called *Example* that is reshaped to be a 3×3 matrix with the elements of 1, −2, 3, 3, 5, 2, −1, 3, and 4. The second line applies the inverse operator (a domino-like symbol with two dots) to the *Example* matrix and stores the result into *ExampleInv*. Just to check the result the two matrices are multiplied together to see if we get the identity matrix. The line with just *I* is printing that result and we can see a bit of a round-off error in the first element of the second row (it should be zero).

APL was notoriously difficult to understand and the example (from https://computerhistory.org/blog/the-apl-programming-language-source-code/) is a version of John H. Conway's "Game of Life," which is a two-dimensional array of cells that each live or die based on a set of simple rules. However, it has an appeal to those desiring a concise, precise mathematical notation to describe what the program should produce. This ability to be very concise makes the language powerful in the sense that a few lines of code can do a lot of calculations compared to other programing languages. See also the Dyalog APL version of the Game of Life as described at https://tryapl.org and in the video at https://www.youtube.com/watch?v=a9xAKttWgP4fmt=18.

Even with APL's unusual syntax and use of a different character set, APL gained a steady level of acceptance and is still used in modern variants such as programming languages J and K. Even though APL was designed initially at IBM, other companies also developed their own versions of APL such as the 1973 version developed at Xerox (see http://www.livingcomputermuseum.org/Online-Systems/User-Documentation/CP-V-(Sigma-9)/7_APL_Manual.aspx for the manual).

4.6 Lessons Learned from Programming Languages

A number of lessons have been learned from the creation and modification of programming languages that are generally applicable to other software systems.

Programming languages come and go based on a wide variety of factors. However, we can learn from the languages that have stood the test of time and those that haven't. Even highly influential languages have often not survived the test of time.

Some of the reasons that programming languages have failed to have widespread use include the following reasons:

- *Usage.*

  Some languages never get used enough to even have a realistic chance to survive. Often this is because other entrenched languages are sufficiently filling the need and the cost of retraining and moving the existing codebase is too high.

- *They don't evolve.*

  Successful programming languages like FORTRAN, COBOL, and LISP have all evolved to incorporate new features such as object-oriented programming. Having a base of support so the language can grow over time is critical.

- *Too niche.*

  Some programming languages fulfill too narrow of a need to get broad acceptance. A good example is a language such as APL that has a loyal but small following.

- *Too complex.*

  A number of programming languages had complex definitions and as a result their implementations were complex. Languages such as PL/I and Ada were complex and implementing compilers was difficult at the time, slowing down the availability of compilers for the language. When these compilers were also buggy, this also slowed acceptance.

- *Loss of support.*

  When the primary backers of a language no longer support enhancement or use of the language, the language's use will take a serious blow. For example, when Ada was no longer required to win US government contracts, its use dropped significantly. There's still a smaller community that uses Ada for reliable, real-time systems and has developed an Ada 2012 language standard.

Languages are more likely to succeed when the new language fits an emerging need such as Java's use in web browsers and its adoption in the Microsoft Internet Explorer browser. When companies push and support a language, it can clearly help its acceptance. Many of these reasons also apply to the continued usage of other software systems.

Some more general lessons learned from programming language history include:

- *Even widely used programming languages can decline.*

  While programming languages can become very entrenched, their popularity does change over time and even relatively popular languages that enjoy lots of hype and adoption have declined in use over time. PL/I and Ada are good examples of this. Other languages were more entrenched earlier and have continued to have widespread use, such as FORTRAN and COBOL. Even FORTRAN and COBOL are gradually declining in use, especially for new systems, though there are still some pockets where they are popular.

- *Formalism of compiler theory stimulated growth.*

  Breakthroughs in parsing theory allowed the development of tools that made compiler development easier and helped to stimulate a proliferation of programming languages (see Knuth [1965]). In general, a formal breakthrough in an area can stimulate the development of software in that area.

- *Performance key to acceptance.*

  Particularly for early programming languages such as FORTRAN, being able to produce executable programs that were close to the performance of hand-coded assembly was critical to the acceptance of FORTRAN. Performance is often a key factor in the acceptance of software systems, which we've seen in operating systems and will see in databases as well as other software systems. Software systems where the system is viewed as overhead and independent of the application programs are often expected to minimize the amount of processing that they use. These systems often have a broad effect on the performance of the system as a whole or on a large number of user-level application programs.

4.7 Exercises and Projects

4.7.1 Exercises

1. Short Code (originally developed for the BINAC computer and then for the UNIVAC I) was not a compiler but could be considered an interpreter. Find out more about Short Code and explain why it is an interpreter.
2. The MAD programming language was widely used in the late 1950s and early 1960s but did not attain widespread usage after that. Explain why this language did not survive and what the programmers who would have been

predisposed to using MAD (perhaps by having used it on CTSS) used instead in the late 1960s and beyond.

3. The Pascal programming language was used for many years as a teaching language, and in many universities was the primary programming language of instruction. Investigate what programming language replaced Pascal for those universities that were using Pascal as their primary programming language. What did they replace it with and why?
4. Brian Kernighan wrote a short paper criticizing the Pascal programming language (see Kernighan [1981]). Find a copy of that paper and determine if Kernighan's criticisms were justified. Could any of the criticisms also be made about the C programming language (which Kernighan is known for)?
5. PL/I was used for many years by many companies. Why was PL/I used? Why was its use discontinued?
6. RPG (Report Program Generator) and its later versions RPG II, RPG III, and RPG IV have retained a following of programmers, particularly on some IBM systems. Explain RPG's origins and why it continues to have a following.
7. The SNOBOL programming language was originally designed as a string-manipulation language that evolved into a more general-purpose language by SNOBOL4. The language was taught in a number of universities into the 1970s. However, very few people use SNOBOL today. Explain what made SNOBOL innovative at the time and why SNOBOL is no longer in use.
8. Investigate the development of SAP's (Systemanalyse und Programmentwicklung) special-purpose language called ABAP (Advanced Business Application Programming). Why did SAP use its own language rather than an already established programming language? Is there any relationship or influence with COBOL? If so, show what features are similar between ABAP and COBOL. If not, explain the differing approaches between COBOL and ABAP.
9. Find and read a copy of C.A.R. Hoare's paper "Record Handling." This was part of a series of lectures delivered at the NATO Summer School, Villard-de-Lans, September 1966, and published by Academic Press. Show how the ideas in this paper influenced the object-oriented ideas in Simula 67.
10. Compare IBM 370 machine code to that of the MIPS processor. IBM System/370 is considered a complex instruction set computer (CISC) while MIPS is considered a reduced instruction set computer (RISC). Compare the number of instructions, the complexity of instructions, and the types of instructions.

11. Command shell programming languages such as UNIX's Bourne shell (sh), Korn Shell (ksh), and csh have developed a number of features that allow the development of complex programs using them. Why did command shells introduce general-purpose programming features? What are their limitations? When might they be better than a compiled programming language?
12. The AWK programming language (named after Al Aho, Peter Weinberger, and Brian Kernighan) was designed to do text processing and operates on a stream of textual data. Considering the context of being developed as part of the UNIX toolset, why was a language like AWK useful?
13. Look up the agent-oriented programming paradigm. Is this distinct enough from other paradigms to justify being named a "programming paradigm?" Justify your position.
14. Some paradigms are proposed to be independent of the programming language, such as *event-driven programming* where the program is designed around events happening external to the program (such as a user clicking a mouse or system interrupts). Should a programming paradigm have a language implementation to be a "programming paradigm?" Defend your position.
15. Railroad diagrams are an alternative method used to describe the grammar of programming languages. They are often used to describe context-free languages instead of Backus–Naur Form (BNF). Find out how they came about and explain if they are still used.
16. An important finding was the development of efficient left-to-right parsing algorithms to allow parsers to be built from a Backus–Naur Form defined grammar. This allowed tools to be built for creating parsers for LR(k) grammars. See Knuth [1965]. Describe how this development as well as the tools to build compilers that followed (such as the UNIX operating system's *lex* and *yacc* and later tools such as *flex* and *bison*) made it easier to develop compilers and catalyzed the development of programming languages.
17. Using the http://hopl.info/ website, find information about the Internal Translator (IT) programming language developed at Purdue University in 1955 for the Burroughs Datatron 205. Follow the trail of influences to find the newest programming language(s) that can claim an influence from IT. Follow the "evolution of" links as well as any links where IT "influenced" a later language. Explain what language IT influenced the most to have the most long-lasting impact. That is, what highly influential language did IT influence?

18. Edsgar Dijkstra [1975] is quoted as having said that the APL programming language “creates a new generation of coding bums.” Find the full quote, determine what he meant by it, and evaluate whether you agree or not with his statement.
19. While most programming languages die a very slow death and continue with some limited usage, some completely die and have no real usage. Examples of languages that are completely dead (in this author’s view) include B, BCPL, SNOBOL, JOVIAL, Concurrent C, and ALGOL. Examples that continue with some limited usage are APL, ADA, and PL/I.

    Choose a language whose usage has completely died out (either from the above list or another one) and describe what factors influenced the dramatic decline of your example’s usage.
20. Give an example of a programming language that was not highly commercially successful at the time but ended up having an important impact on later programming language(s). Note why it failed to be widely used and successful at the time, what the impact of the language was, and why that impact was then successful in a later programming language(s).

4.7.2 Projects

1. One could argue, based on programmer efficiency in assembly language, that a CISC such as IBM 360 Assembly would be more productive for the programmer than a RISC such as MIPS. Summarize and find arguments to this effect in the time that RISC architectures were proposed and built. Was there any evidence to show that a CISC architecture could be more productive for an assembly language programmer? How might a similar argument be used for more complex higher-level general-purpose programming languages versus less complex high-level languages?
2. The IBM 650 computer used a drum-based memory and had an assembly program called SOAP II (Symbolic Optimal Assembly Program) that took into account the rotation of the drum in how it placed instructions on it to help optimize the assembly. This program is described and contained in IBM [1957], also at http://www.bitsavers.org/pdf/ibm/650/24-4000-0_SOAPII.pdf. In a private discussion with Donald Knuth, he described the implementation of SOAP II as elegant and more like “music” in how it was written. Using IBM [1957] and Andree [1958], explain why this code is so elegant in comparison to other assembly language programs from that time. A useful comparison would be with the IT, a compiler written at Purdue and Carnegie

Institute of Technology for the same machine, the IBM 650 (see http://www.bitsavers.org/pdf/ibm/650/CarnegieInternalTranslator.pdf for the code for the IT). Explain how the elegance of programming in assembly language as exhibited by the 650's SOAP II implementation can translate to elegance in the programming of higher-level languages.

3. Bauer and Samelson's US Patent number 3,047,228 specifies a formula-controlled computer and provides an interesting approach of how to build and program a computer. Find and read the patent to determine if any of the ideas have survived in modern computers or compilers. If not, explain which ideas are still viable and which are not viable and why.

4. Investigate the inclusion of recursive procedures within programming languages. Describe the negotiations between the various computing communities and the different approaches that were tried, and which survived. A good place to start is by reading van den Hove [2014] and Daylight [2011] for ALGOL and Sammet [1969, pp. 589–602] and Wexelblat [1981, pp. 173–197] for histories of LISP. Daylight [2011] states that John McCarthy wanted to include recursion in FORTRAN but was unable to do so, so McCarthy invented LISP as a result. Write a paper describing the different forms of recursion considered up until 1965 and the controversy over including recursion at all (such as in ALGOL 60).[20]

5. Refer to the diagram at http://www.levenez.com/lang/lang.pdf that shows the relationship and evolution of major programming languages. In that diagram, the Ruby programming language begins in 1993 with influences from all of the following programming languages: Python, CLU, Eiffel 2, Smalltalk 80, Common LISP, and Perl. Investigate the history of Ruby and find out what features and concepts were used from those languages and explain why the developers of Ruby thought it necessary to develop Ruby.

6. Investigate the worldwide development and use of programming languages for artificial intelligence. Explain why LISP was so widely used in the United States while other languages such as Prolog were used in Europe and Japan.

7. PL/I's use of formal operational semantics to define the meaning of the programming language resulted in a complex specification written in another language called Vienna Definition Language (VDL) [Wegner 1972] that many found hard to understand and use. Find out if the use of VDL impacted the

20. C.A.R. Hoare, in an oral history interview with ACM for ACM Turing awardees, noted that the mention of recursion in the ALGOL report gave him the inspiration to complete the definition of Quicksort using recursion.

ability to produce working versions of PL/I compilers and whether it made it easier or harder to create PL/I compilers. Give evidence from contemporary sources in the 1970s.

8. An early description of "automatic coding" is a lecture given by Glennie at the University of Cambridge in early 1953 (see Glennie [1952]). Obtain a copy of these lecture notes (the Computer History Museum has a scanned copy at https://archive.computerhistory.org/resources/text/Fortran/102653981.05.01.acc.pdf) and compare the description of automatic coding to what it became a few decades later.

9. Packed decimal was a number format that allowed the precise specification of decimal numbers so that round-off errors could be avoided (rather than using a floating-point representation). This notation stores two digits in an 8-bit byte, so each digit is stored in 4-bits (termed a *nibble*); hence, two numbers are "packed" into a byte. Is this format still in use? If so, why is it still in use? What programming languages support(ed) a packed decimal representation? What computer manufacturers supported this representation in hardware? Are there benefits other than precision to this notation?

10. Investigate the Forth programming language's history. Charles Moore created Forth in 1968 on the IBM 1130, but in the 1980s Forth became popular as a development language for microcomputers. Describe how the features of the Forth programming language evolved over time and its use of stack-based notation including reverse Polish notation for its syntax. How did Forth influence other languages? Why is Forth no longer widely used?

11. The notion of *programming paradigms* is relatively fuzzy. Research and investigate the creation of new programming paradigms. In particular, focus on the development of object-oriented paradigm and how it evolved from the days of Simula, through Smalltalk, and for C++ and Java. Develop a paper that describes the evolution of the object-oriented paradigm from previously existing paradigms. Include information about the promises made of the object-oriented paradigm for improving software development time and reuse and detail how those promises were met or not.

12. Consider the development of "programming languages" for items other than computing. For example, investigate the difficulty in specification and in creating implementable and unambiguous designs that are to be printed on 3D printers. Describe any existing languages, problems in specification, and problems in translating it to something that the 3D printer can produce.

13. Consider the problem of typesetting. Similar to writing a program, one specifies the constraints and requirements of a document that is then "compiled" into a printed page. Look at examples such as troff, $\mathrm{T_EX}$, Metafont, and MetaPost. Investigate the issues that occurred in translation, ambiguity, and determination of the language to be used. Compare these issues to those in the creation of programming languages.

14. The APL\360 source code (in IBM 360 Assembly language) has been released to the public (see http://www.computerhistory.org/atchm/the-apl-programming-language-source-code/). As noted by the Computer History Museum on that page, this (amazingly short—only 37,567 lines in 90 files) program took complete control of the IBM 360 and built its own time-sharing environment. Take the source code and get it to operate in either an IBM 360 emulator or an IBM Virtual Machine environment.[21] Also, run a version of the K programming language, which has many elements of APL and is considered a descendant of APL. Write a program in both APL\360 and in K to compute the inverse of an $N \times N$ matrix. Compare the notation and ease of use of the two languages.

4.8 Further Readings and Online Resources

The history of programming languages is very well-documented and there are many resources that are available. A good starting point are the ACM History of Programming Languages (HOPL) conferences I (1978), II (1993), and III (2007) with proceedings Wexelblat [1981], Bergin and Gibson [1996], and Hailpern [2007], respectively. Those conferences contain papers describing the history of individual languages, usually by one of the principals involved in its development. Sammet [1969] is an early programming languages and history textbook that does a good job synthesizing the history to that point. Sammet [1972] is a short article that condenses early programming history. An excellent website that shows the influences of programming languages as well as many references for each programming language is the http://hopl.info/ site maintained by Diarmuid Pigott.

An excellent collection of papers on computer languages by Donald Knuth are in Knuth [2003]. This collection includes a description of the early development of programming languages from Plankalkül, Flow Diagrams, Short Code, FORTRAN and 16 other early programming languages. This collection also includes many other papers with early developments of ALGOL and SOL. Chapter 20 (p. 439) of this text is "A History of Writing Compilers," expanded from a talk given in 1962.

21. Also see Jürgen Winkelmann's version at http://wotho.ethz.ch/mvt4apl-2.00/ where he has Windows, Mac, and Linux versions.

5

Programming Environments, Tools, and Methodologies

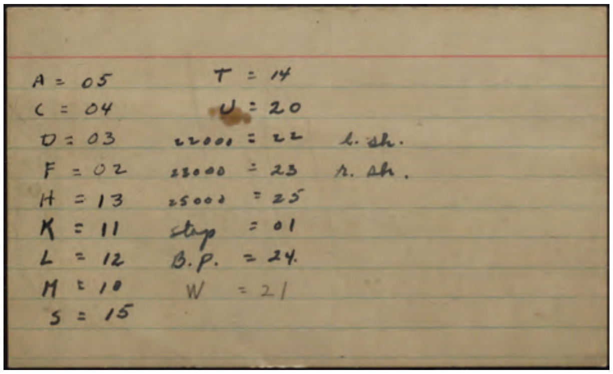
Beyond programming languages, programmers have built environments and programming toolsets to help make programming more efficient. These have changed dramatically over time along with the advent of software engineering and development methodologies. Initially, most of the support processes were based on paper processes, reference cards, and forms. It was quickly recognized that the use of the computer to help automate rote tasks was desirable.

Programming environments and the tools we use to build systems have changed radically—from the use of switches on the backplane to the use of modern software development environments. While our productivity has gone up tremendously, many positive features of those historical programming environments (such as the care that had to be taken when using punched cards with batch processing) are worth re-envisioning in today's programming toolset.

5.1 Early Programming Environments and Tools

As more computers were sold and deployed to companies and enterprises in the mid-1950s, many organizations developed paper-based tools to help in the design and implementation of software. Even before that time, ENIAC programmers had used a form of flowchart to help define how their program would operate (see Haigh et al. [2016] for a detailed description of ENIAC flowcharts). Flowcharts continued to be heavily used, such as the example in Figure 5.3. In that example from the Eckert–Mauchly Computer Corporation, they used blueprint drawing paper to create their flowcharts.

In the decades that followed many organizations developed their own form to help document what was being done as well as to communicate to others involved



A = 05	T = 14	
C = 04	U = 20	
D = 03	22000 = 22	l. sh.
F = 02	23000 = 23	r. sh.
H = 13	25000 = 25	
K = 11	stop = 01	
L = 12	B.P. = 24.	
M = 10	W = 21	
S = 15		

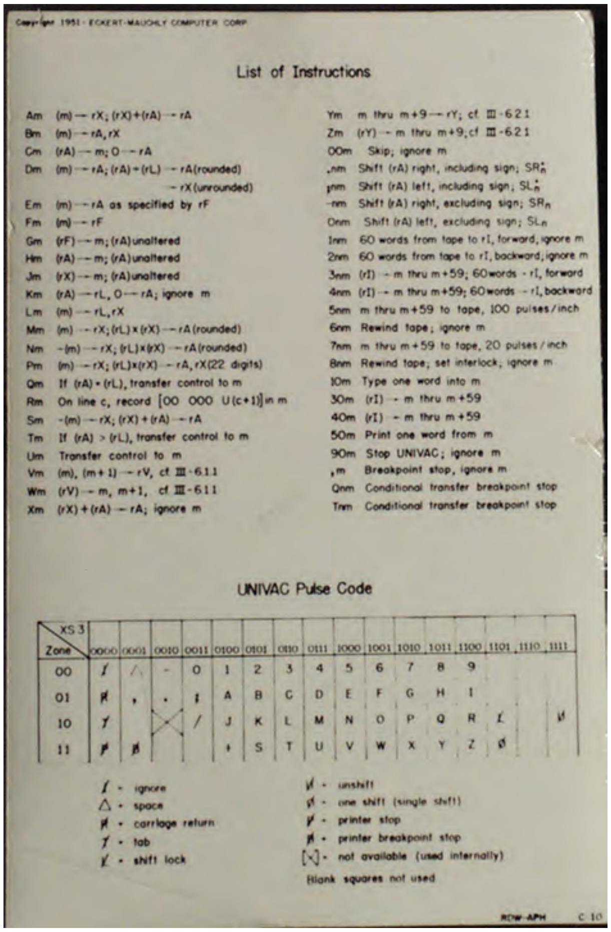
Figure 5.1 Grace Hopper's BINAC code card (labeled as "first"), circa 1949. (Source: Courtesy of Grace Murray Hopper Collection, Archives Center, National Museum of American History, Smithsonian Institution.)

in the process. As an example, see Figure 5.4 for an example batch job setup form for the IBM 701. It was used to communicate between the programmer and the computer operator any special instructions needed to set up the job and anything else that the operator needed to do to support the running of the job. This 1956 form also includes a handwritten request to change the binary deck to "octonary" or octal, base 8 numeral system. DUET is also checked on this form and refers to coding the program using a floating-point interpretive system that permitted intermingling of machine code with interpretive code (originally called DUAL[1]). Coding forms have been used since the earliest days of computing, such as this form (see Figure 5.5) for the BINAC in 1949 that shows a portion of the program used to perform matrix multiplication by segmenting the matrix into smaller matrices that the BINAC could handle. Another example form (see Figure 5.6) for the IBM 704 shows a form developed by the SHARE user group specifically designed for the IBM 704 SHARE assembly language code (SAP). In this 1956 programming classroom example, the comments describe what the assembly code is designed to do.

Besides forms, programmers made extensive use of paper quick reference and code cards in order to have a handy reference for the details they needed. One example is in Figure 5.1 where this card for BINAC codes was labeled as the "first code card" for BINAC developed at the Eckert–Mauchly Computer Corporation. Another example in Figure 5.2 is a quick reference card for UNIVAC I's instruction set.

Most computers in the 1950s and 1960s used either paper tape or punched cards for input. IBM heavily used punched cards and had built its business on the

1. See "The North American 701 Monitor," by Owen Mock, published in the National Computer Conference proceedings, 1987, pp. 791–795.



Copyright 1951 · ECKERT-MAUCHLY COMPUTER CORP

List of Instructions

Am	(m) → rX; (rX)+(rA) → rA
Bm	(m) → rA, rX
Cm	(rA) → m; 0 → rA
Dm	(m) → rA; (rA) ÷ (rL) → rA(rounded) → rX(unrounded)
Em	(m) → rA as specified by rF
Fm	(m) → rF
Gm	(rF) → m; (rA) unaltered
Hm	(rA) → m; (rA) unaltered
Jm	(rX) → m; (rA) unaltered
Km	(rA) → rL, 0 → rA; ignore m
Lm	(m) → rL, rX
Mm	(m) → rX; (rL) x (rX) → rA (rounded)
Nm	-(m) → rX; (rL) x (rX) → rA (rounded)
Pm	(m) → rX; (rL) x (rX) → rA, rX (22 digits)
Qm	If (rA) = (rL), transfer control to m
Rm	On line c, record [00 000 U(c+1)] in m
Sm	-(m) → rX; (rX) + (rA) → rA
Tm	If (rA) > (rL), transfer control to m
Um	Transfer control to m
Vm	(m), (m+1) → rV, cf III-611
Wm	(rV) → m, m+1, cf III-611
Xm	(rX) + (rA) → rA; ignore m
Ym	m thru m+9 → rY; cf III-621
Zm	(rY) → m thru m+9; cf III-621
00m	Skip; ignore m
.nm	Shift (rA) right, including sign; SR_n^*
;nm	Shift (rA) left, including sign; SL_n^*
-nm	Shift (rA) right, excluding sign; SR_n
0nm	Shift (rA) left, excluding sign; SL_n
1nm	60 words from tape to rI, forward; ignore m
2nm	60 words from tape to rI, backward; ignore m
3nm	(rI) → m thru m+59; 60 words → rI, forward
4nm	(rI) → m thru m+59; 60 words → rI, backward
5nm	m thru m+59 to tape, 100 pulses/inch
6nm	Rewind tape; ignore m
7nm	m thru m+59 to tape, 20 pulses/inch
8nm	Rewind tape; set interlock; ignore m
10m	Type one word into m
30m	(rI) → m thru m+59
40m	(rI) → m thru m+59
50m	Print one word from m
90m	Stop UNIVAC; ignore m
,m	Breakpoint stop, ignore m
Qnm	Conditional transfer breakpoint stop
Tnm	Conditional transfer breakpoint stop

UNIVAC Pulse Code

Zone \ XS 3	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	Ɨ	△	-	0	1	2	3	4	5	6	7	8	9			
01	Ꝁ	,	.	;	A	B	C	D	E	F	G	H	I			
10	Ŧ		[×]	/	J	K	L	M	N	O	P	Q	R	Ł		Ʉ
11	Ᵽ	ꞵ			+	S	T	U	V	W	X	Y	Z	ꞩ		

Ɨ - ignore	Ʉ - unshift
△ - space	ꞩ - one shift (single shift)
Ꝁ - carriage return	Ᵽ - printer stop
Ŧ - tab	ꞵ - printer breakpoint stop
Ł - shift lock	[×] - not available (used internally)
	Blank squares not used

RCW-APH C 10

Figure 5.2 Grace Hopper's UNIVAC quick reference card, 1951. (Source: Courtesy of Grace Murray Hopper Collection, Archives Center, National Museum of American History, Smithsonian Institution.)

tabulation of data and had produced many products using cards before they were used as input and output for computers. So, when IBM did enter the computer business, they leveraged their use of punched cards and many other manufacturers also used punched cards. Other computers would use paper tape and leveraged the preexisting industry in punched tape. A typical IBM punched card was an 80-character card that was meant to hold a single line of a program or data. Programming languages at the time were also designed with punched cards in mind. The FORTRAN standard also used the 80-character format with specific meaning for particular positions in the 80 columns on the card. Here, one can see that each

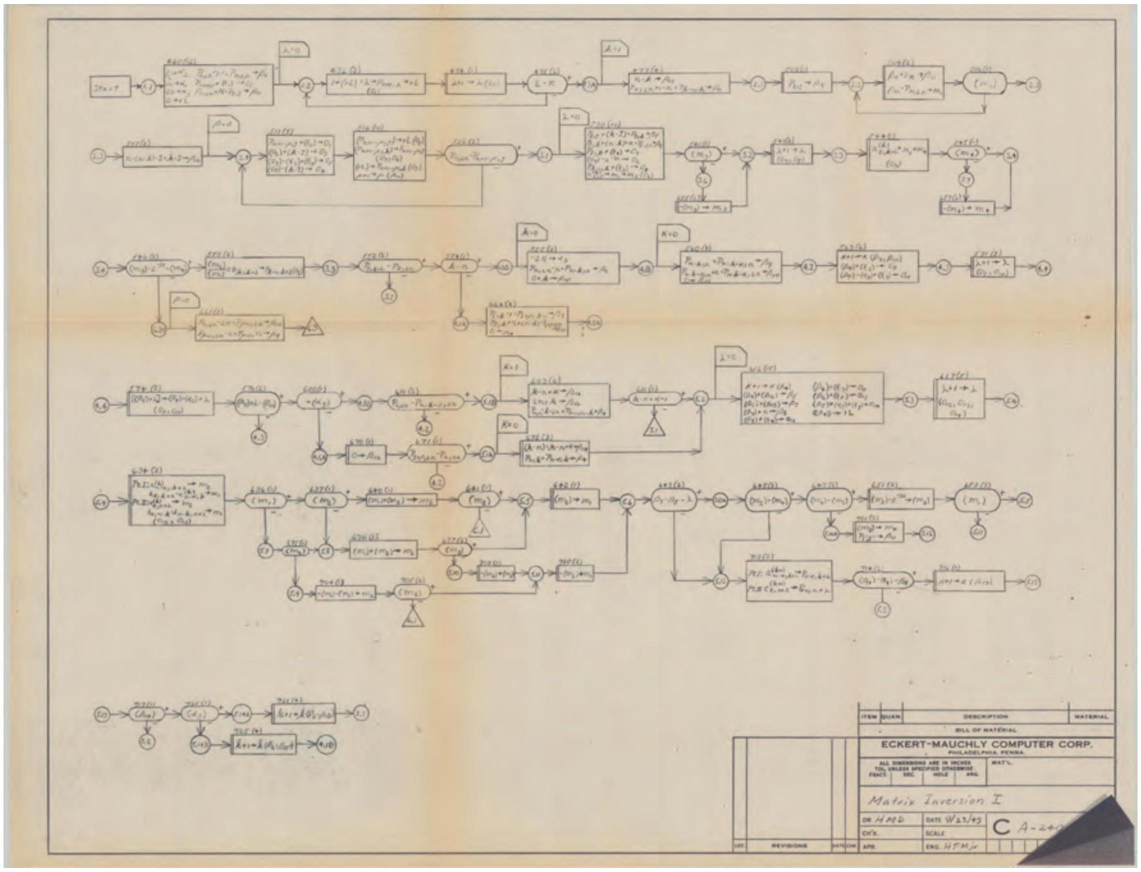
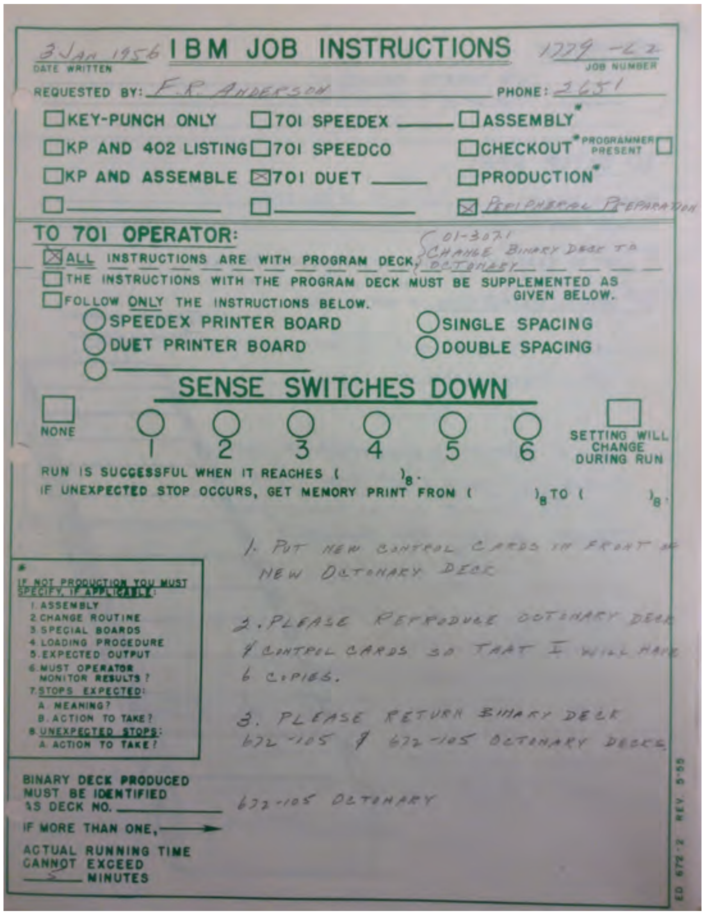


Figure 5.3 1949 Flowchart from Eckert–Mauchly Corp for Matrix Inversion Program on BINAC computer. (Source: Courtesy of Grace Murray Hopper Collection, Archives Center, National Museum of American History, Smithsonian Institution.)

character is represented by a number of punched holes in the column beneath it (see Figure 5.17). Cards were punched by a card punch and read into a card reader such as in Figure 5.7. Programmers would either punch their own cards or would fill out coding sheets (as in Figure 5.6) that were sent to a set of employees known as "coders" or as "data entry" or even as "key punch operators." In large organizations, these employees would punch the cards for programmers and then return the card deck to the programmers. So, a programmer would fill out the coding form, get the cards punched by the key punch operators, and then fill out job instructions (such as on Figure 5.4). This all led to long lead times before a programmer would receive any output from their program, which could end up being a simple syntax


3 JAN 1956 **IBM JOB INSTRUCTIONS** 1779-62
DATE WRITTEN — JOB NUMBER

REQUESTED BY: F.R. ANDERSON PHONE: 2651

☐ KEY-PUNCH ONLY	☐ 701 SPEEDEX ____	☐ ASSEMBLY*
☐ KP AND 402 LISTING	☐ 701 SPEEDCO	☐ CHECKOUT* PROGRAMMER PRESENT ☐
☐ KP AND ASSEMBLE	☒ 701 DUET ____	☐ PRODUCTION*
☐ ____	☐ ____	☒ PERIPHERAL PREPARATION

TO 701 OPERATOR:

☒ ALL INSTRUCTIONS ARE WITH PROGRAM DECK. { 01-3071 CHANGE BINARY DECK TO OCTONARY

☐ THE INSTRUCTIONS WITH THE PROGRAM DECK MUST BE SUPPLEMENTED AS GIVEN BELOW.

☐ FOLLOW ONLY THE INSTRUCTIONS BELOW.

○ SPEEDEX PRINTER BOARD ○ SINGLE SPACING
○ DUET PRINTER BOARD ○ DOUBLE SPACING
○ ____

SENSE SWITCHES DOWN

☐ NONE ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ☐ SETTING WILL CHANGE DURING RUN

RUN IS SUCCESSFUL WHEN IT REACHES ($)_8$.
IF UNEXPECTED STOP OCCURS, GET MEMORY PRINT FROM ($)_8$ TO ($)_8$.

1. PUT NEW CONTROL CARDS IN FRONT OF NEW OCTONARY DECK

2. PLEASE REPRODUCE OCTONARY DECK & CONTROL CARDS SO THAT I WILL HAVE 6 COPIES.

3. PLEASE RETURN BINARY DECK 672-105 & 672-105 OCTONARY DECKS

672-105 OCTONARY

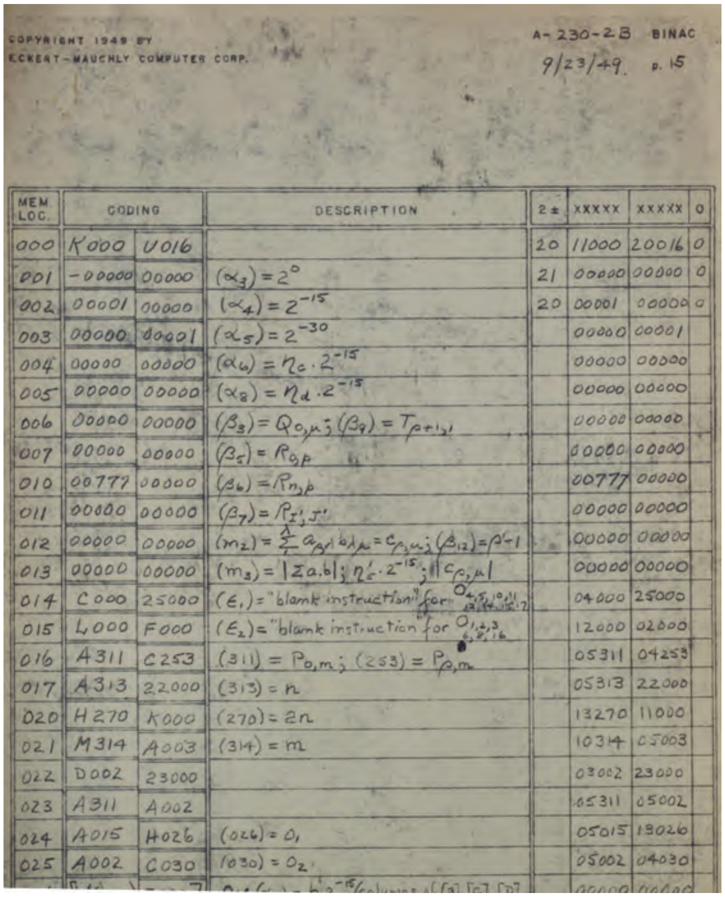
* IF NOT PRODUCTION YOU MUST SPECIFY, IF APPLICABLE:
1. ASSEMBLY
2. CHANGE ROUTINE
3. SPECIAL BOARDS
4. LOADING PROCEDURE
5. EXPECTED OUTPUT
6. MUST OPERATOR MONITOR RESULTS?
7. STOPS EXPECTED:
   A. MEANING?
   B. ACTION TO TAKE?
8. UNEXPECTED STOPS:
   A. ACTION TO TAKE?

BINARY DECK PRODUCED MUST BE IDENTIFIED AS DECK NO. ____
IF MORE THAN ONE, →
ACTUAL RUNNING TIME CANNOT EXCEED 5 MINUTES

ED 672-2 REV. 5-55

Figure 5.4 An IBM 701 setup form from 1956, Florence Anderson papers. (Source: Image Courtesy of the Charles Babbage Institute Archives, University of Minnesota Libraries, Minneapolis.)

error. As a result, programmers of this era were extremely careful to check their program, cards, and associated coding and instruction forms before submitting their job to the computer. Depending on the turnaround time at the particular organization's data center, this wait could be extensive. As an example, a number of universities had no computers and would send their batch jobs to another university, often resulting in a turnaround time of several days or a week. All this paper led to extensive efforts to organize and store programs such as in Figure 5.8 where the US National Archives Record Service Warehouse was already filling large warehouses in 1959 with boxes of punched cards. A common story from programmers



COPYRIGHT 1949 BY
ECKERT-MAUCHLY COMPUTER CORP.

A-230-2B BINAC
9/23/49 p. 15

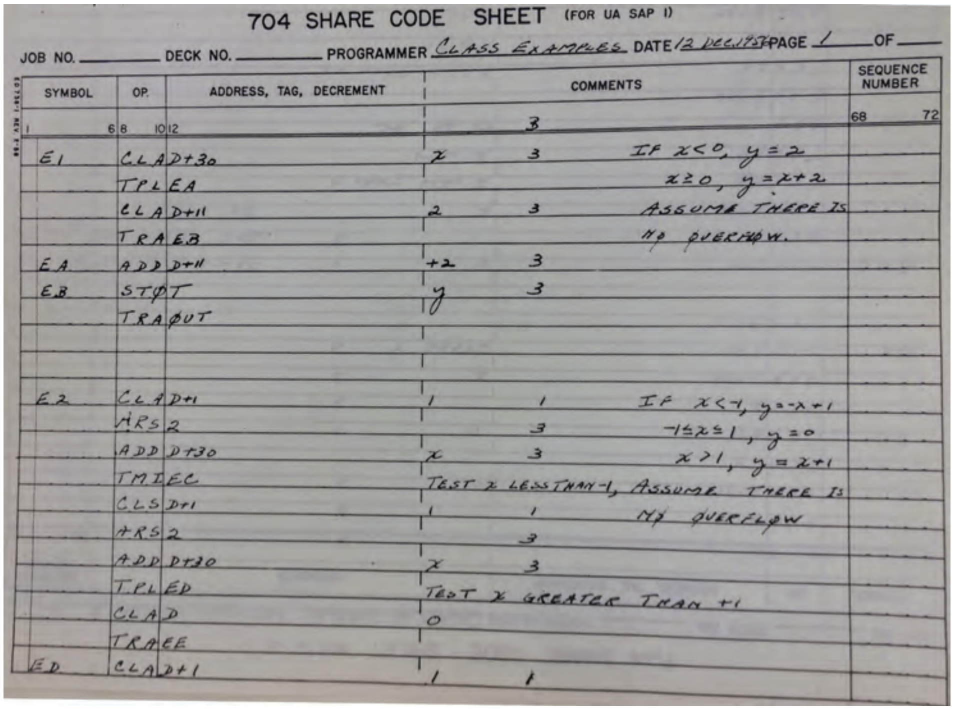
MEM. LOC.	CODING		DESCRIPTION	2 ±	XXXXX	XXXXX	0
000	K000	U016		20	11000	20016	0
001	-00000	00000	$(\alpha_3) = 2^0$	21	00000	00000	0
002	00001	00000	$(\alpha_4) = 2^{-15}$	20	00001	00000	0
003	00000	00001	$(\alpha_5) = 2^{-30}$		00000	00001	
004	00000	00000	$(\alpha_6) = \eta_c \cdot 2^{-15}$		00000	00000	
005	00000	00000	$(\alpha_8) = \eta_d \cdot 2^{-15}$		00000	00000	
006	00000	00000	$(\beta_3) = Q_{0,\mu}; (\beta_9) = T_{\rho+1,1}$		00000	00000	
007	00000	00000	$(\beta_5) = R_{q,p}$		00000	00000	
010	00777	00000	$(\beta_6) = R_{n,p}$		00777	00000	
011	00000	00000	$(\beta_7) = R_{I',J'}$		00000	00000	
012	00000	00000	$(m_2) = \sum_1^{\lambda} a_{\rho\lambda} b_{\lambda\mu} = c_{\rho,\mu}; (\beta_{12}) = \rho' - 1$		00000	00000	
013	00000	00000	$(m_3) = \lvert \Sigma a.b \rvert; \eta_c' \cdot 2^{-15}; \lvert c_{\rho,\mu} \rvert$		00000	00000	
014	C000	25000	(ϵ_1) = "blank instruction" for $O_{4,5,10,11,13,14,15,17}$		04000	25000	
015	L000	F000	(ϵ_2) = "blank instruction" for $O_{1,2,3,6,8,16}$		12000	02000	
016	A311	C253	$(311) = P_{0,m}; (253) = P_{\rho,m}$		05311	04253	
017	A313	22000	$(313) = n$		05313	22000	
020	H270	K000	$(270) = 2n$		13270	11000	
021	M314	A003	$(314) = m$		10314	05003	
022	D002	23000			03002	23000	
023	A311	A002			05311	05002	
024	A015	H026	$(026) = O_1$		05015	13026	
025	A002	C030	$(030) = O_2$		05002	04030	

Figure 5.5 1949 Coding form from Eckert–Mauchly Corp. for matrix multiplication program on BINAC computer. (Source: Courtesy of Grace Murray Hopper Collection, Archives Center, National Museum of American History, Smithsonian Institution.)

that used punched cards was of dropping a box of punched cards, resulting in them getting out of order and requiring a long time to re-order.

5.2 Evolution of Programmer Tools Over Time

Programmers are always looking for ways to build better and more efficient systems and programs. As computers became more powerful, more and more tools could be developed to help aid the programming process. This section looks at the high-level changes that have occurred to these tools. Many factors have contributed to the changes in the programmer interface that have enhanced and



704 SHARE CODE SHEET (FOR UA SAP I)

JOB NO. ______ DECK NO. ______ PROGRAMMER CLASS EXAMPLES DATE 12 DEC.1957 PAGE 1 OF ______

SYMBOL	OP.	ADDRESS, TAG, DECREMENT	COMMENTS	SEQUENCE NUMBER
1 6	8 10	12	3	68 72
E1	CLA	D+30	x 3 IF $x<0$, $y=2$	
	TPL	EA	$x \geq 0$, $y=x+2$	
	CLA	D+11	2 3 ASSUME THERE IS	
	TRA	EB	NO OVERFLOW.	
EA	ADD	D+11	+2 3	
EB	STO	T	y 3	
	TRA	OUT		
E2	CLA	D+1	1 1 IF $x<-1$, $y=-x+1$	
	ARS	2	3 $-1 \leq x \leq 1$, $y=0$	
	ADD	D+30	x 3 $x>1$, $y=x+1$	
	TMI	EC	TEST x LESS THAN -1, ASSUME THERE IS	
	CLS	D+1	1 1 NO OVERFLOW	
	ARS	2	3	
	ADD	D+30	x 3	
	TPL	ED	TEST x GREATER THAN +1	
	CLA	D	0	
	TRA	EE		
ED	CLA	D+1	1 1	

Figure 5.6 An IBM 704 coding form, 1956, Florence Anderson papers. (Source: Image courtesy of the Charles Babbage Institute Archives, University of Minnesota Libraries, Minneapolis.)

changed the way that programmers use the computer. These are summarized in Figure 5.9. The first row of Figure 5.9 shows how the mode of interaction has changed over time. Initially, the mode of interaction was direct and only a single program could be run at a time. Programmers used patch panels and the direct keying of machine code programs, often with very little, if any, operating system to help. As computers became more commonplace in the late 1950s, businesses and organizations with expensive computers wanted to keep them as busy as possible to maximize their investment. This led to systems based on a batch model of interaction where the focus was on maximizing the computer throughput of jobs. Operating systems were developed to support this model as noted in Chapter 3. Jobs for these were often submitted as decks of cards or otherwise queued to be run when time was available on the batch computer. With this process, programmers were encouraged to make every run of their program count. Usually, programmers carefully designed and error checked their programs before ever submitting

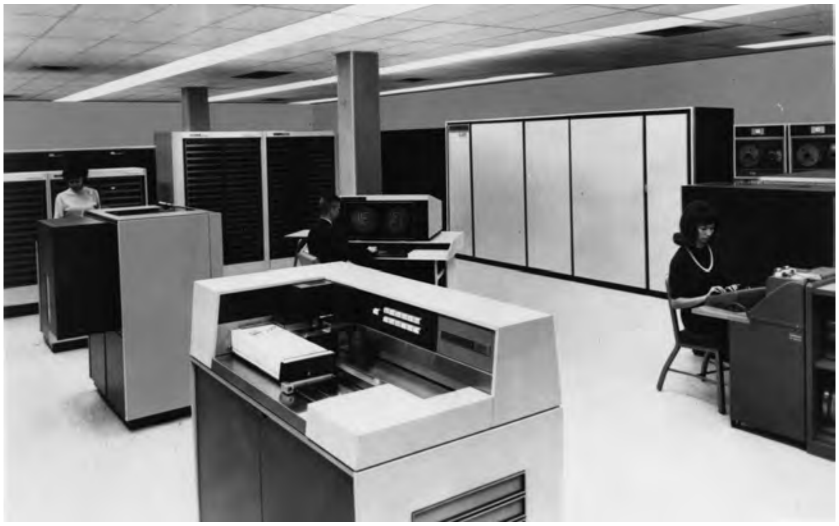


Figure 5.7 A Control Data Corporation 6400 with a 405 card reader in foreground, 1965. (Source: Image courtesy of the Charles Babbage Institute Archives, University of Minnesota Libraries, Minneapolis.)

them to be run. Interactive, usually time-sharing–based operating systems were developed that changed the way that programmers approached programming. As noted in Chapter 3, these time-sharing systems were developed in the mid- to late-1960s.

The change was in the fact that programmers could get the results from their program immediately rather than having to wait for the job to be submitted via batch and the results returned, often days or weeks later. So, this allowed a number of programmers to simultaneously run their programs and let the computer's compilers check the draft program's syntax. In the late 1980s the precise computer one was working on became less critical as networks of computers worked in a distributed computing scheme. This allowed programs to be compiled and run from any of the distributed set of computers. Programming tools were also developed to support this distributed computing environment to help synchronize programmers' efforts, such as source code (or version) control systems that could be used by a distributed team of programmers to work on the same set of code.

The second row of Figure 5.9 refers to the programming methodology. In the early 1950s, there really wasn't a methodology for building systems, but as more and larger software systems were developed it became clear that a method was needed to help many programmers work together. Large projects such as

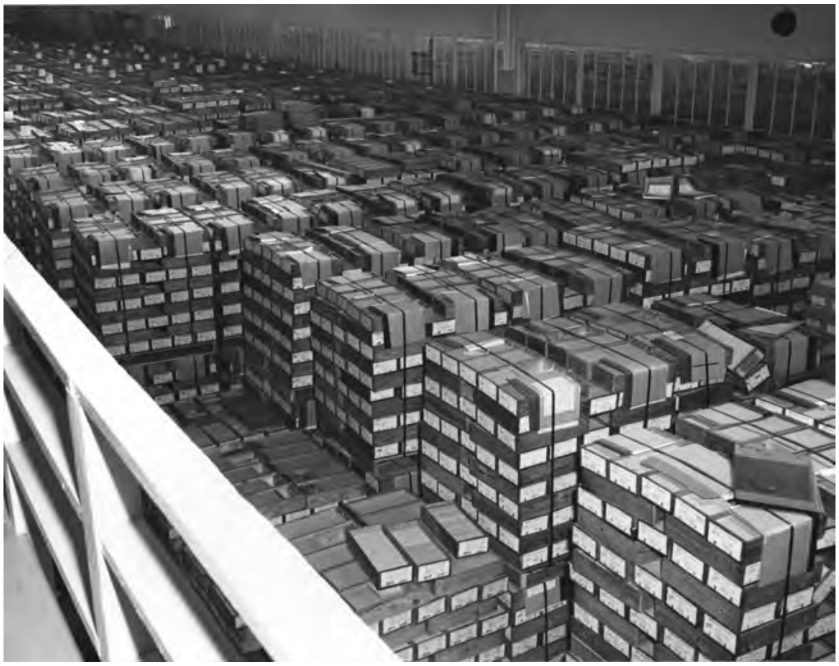


Figure 5.8 IBM card cartons (2000 cards each) in Federal Records Center, Alexandria, VA, November 4, 1959. (Source: US National Archives, https://catalog.archives.gov/id/12169529.)

SAGE, Apollo, and SAFEGUARD[2] developed methodologies that were based on a waterfall-type methodology. This methodology was based on having one phase (such as requirements) be completed and then followed by each successive phase, in turn. This worked reasonably well for very large projects where other parts (such as hardware) were being managed in a similar fashion. Structured programming and related methodologies also became popular during this same time period. These methodologies were focused on designing methodologies that would help build systems that met defined requirements and were maintainable. The next big shift in methodologies came with object-oriented programming's popularity in the late 1980s. With this shift, the hope was that by taking an object-oriented approach, programs would be more maintainable, easier to build, and easier to build components that could be re-used. Structured systems analysis and design was gradually replaced by object-oriented analysis and design. Later in this cycle, frustration with

2. The SAFEGUARD Program was a United States anti-ballistic missile system designed to protect the Minutemen Intercontinental Ballistic Missiles (ICBM) and has its origins dating back to the mid-1950s at Bell Telephone Laboratories.

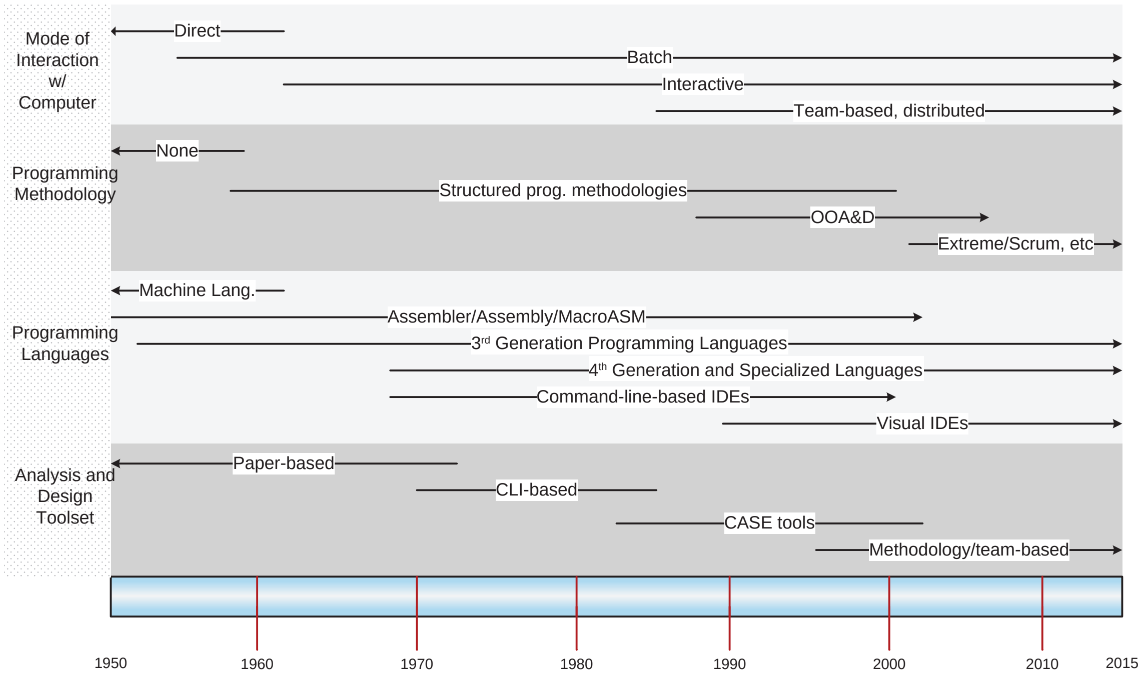


Figure 5.9 Evolution of programmer tools over time.

heavyweight methodologies that were burdensome, especially for small projects, led to the development of new methodologies with a focus on responsiveness to customers and changing requirements. These resulted in methodologies techniques often labeled "agile" being grouped into methodologies such as "Extreme" or "Scrum."

Programming languages have also had a significant impact on programmer productivity. It was recognized early that programming in machine language was not only difficult but time-consuming and very specialized. This resulted in the need for highly trained programmers who understood the details of a particular computer. Assemblers were developed to make the generation of machine code slightly easier, but it was recognized that a form of "automatic programming" that would allow less–highly trained users to build their own programs would be of great value. So, in the mid- to late-1950s, building a reliable and efficient compiler was a priority for companies like IBM and Remington Rand. IBM's FORTRAN and CODASYL's COBOL made it clear that higher-level languages were possible and worth pursuing to help increase programmer productivity. While they did not result in users always being able to write their own programs, they did result in a productivity increase for professional programmers. After this time, a large variety of programming languages, as well as more specialized programming languages, were developed, as described in Chapter 4.

Toolsets were built around compilers that included editors, debuggers, and associated tools into what could (in retrospect) be called *command-line-based integrated development environments* (IDEs). These toolsets, such as the Programmer's Workbench (PWB), developed alongside the UNIX operating system, solidified the notion that programmers did desire a set of tools to be more productive. These evolved into graphical IDEs that are called Visual IDEs in the diagram, which include toolsets such as Eclipse, Microsoft's Visual Studio, among many others.

The tools used to specify the requirements, the design, and documentation have also evolved. These began as diagrams such as flowcharts. Interestingly, the ENIAC programs used a form of flowcharts and this technique was shared with many involved in computing at the time. Flowcharts evolved into many other diagrams used to specify many different aspects of the system such as file formats, runtime behavior, and architecture of the system. Many organizations and companies had a number of different types of forms used to specify these different aspects of the system. As one might expect, these diagrams were supportive of the programming methodology and usually closely related to the methodology being used. Some command-line interface (CLI) tools were developed to help structure these requirements and designs. In the mid- to late-1980s, Computer-aided software engineering (CASE) tools were being developed with the focus on supporting the methodology as well as keeping all the system aspects consistent with one another. CASE tools evolved into having some parts that were called upperCASE (to support the system definition and design) and others that were called lowerCASE (to support the automatic creation of a running system). Such CASE tools were used for large projects, particularly those using structured analysis and design techniques. As newer Agile-related methodologies emerged such as Extreme and Scrum, team-based tools that supported a distributed team with changing requirements became more the norm.

5.3 Large Projects and the Software "Crisis"

A number of projects in the late 1950s and into the 1960s made it clear that software was difficult to develop and that there were not enough programmers to fulfill the needs of all the burgeoning projects requiring software. Projects such as SAGE (see Section 5.5 and Sackman [1967]) used a high percentage of the number of available programmers and contributed to the types of methodologies and programming techniques that would be used. Companies such as the System Development Corporation (SDC) were formed in order to help build the SAGE system. SAGE, being a real-time system and employing many programmers, had to solve new problems. SAGE programmers had a shared experience and methodology and used that

Figure 5.10 IBM System 360 Model 91 being used at NASA Goddard Flight Center in Greenbelt, MD, January 16, 1968. (Source: Courtesy of International Business Machines Corporation, ©1968 International Business Machines Corporation.)

experience when working on other projects. As more large projects involving software were attempted such as Apollo (see Figure 5.10), SABRE[3] and SAFEGUARD, it became clear that it was difficult to deliver such complex projects and that the supply of qualified programmers was low compared to the demand. Edsger Dijkstra put it well in his Turing Award lecture in 1972, "The Humble Programmer," where he states the major cause of the *software crisis* as:

> the major cause is … that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming had become an equally gigantic problem.

This concern led to the October 1968 NATO Conference on Software Engineering in Garmisch, Germany. At the conference, it was openly acknowledged that a crisis existed and the four major points from highlights of the report from that conference[4] were (verbatim):

3. Semi-Automated Business Research Environment (SABRE) was developed to automate reservations for American Airlines; started in 1957 leveraging IBM's SAGE experience and was initially deployed in 1960.

4. Available online at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

- the problems of achieving sufficient reliability in the data systems which are becoming increasingly integrated into the central activities of modern society
- the difficulties of meeting schedules and specifications on large software projects
- the education of software (or data systems) engineers
- the highly controversial question of whether software should be priced separately from hardware.

So, the conference was worried about software reliability, building large software systems reliably, having enough programmers, and, finally, the emerging trend to sell software separately from the hardware. While this conference had a relatively small number of attendees (around 50), it was highly influential and resulted in an increased focus on adding engineering principles to software production. The first two items have become large areas of emphasis for software engineering.

Another large project also in the same time frame was the IBM/360 System project mentioned in Chapter 3. Fred Brooks (project manager for the IBM/360) wrote an influential book in 1975 called *The Mythical Man-Month* [Brooks 1975], where he made the persuasive argument that one could not simply add programmers to a project and expect it to complete earlier. From the experience on OS/360, he relayed many practices that were widely adopted by the software engineering community.

5.4 Reflections on Programming Tools and Environments

Over time, and by design, programming environments and tools have become more abstract and have developed layers of abstraction. This has allowed larger and more complex problems to be solved and more complex systems to be successfully built. At the same time, there are many programmers that are only taught at higher levels of abstraction. Most of the time these programmers can be extremely effective, but if an unexpected interaction between the layers of abstraction occurs, these programmers are not able to fix it. As systems as a whole become more complex, they become more difficult to diagnose and fix with complex hierarchies of abstraction and of technologies. In Figure 5.11, there are several layers of abstraction including hardware, a virtual machine hypervisor, various virtual machines, and then further abstractions build on those. So, one could develop a mobile app that works in the simulator using that device's model, but if there's an issue where the simulator is different than the actual device, then it may not

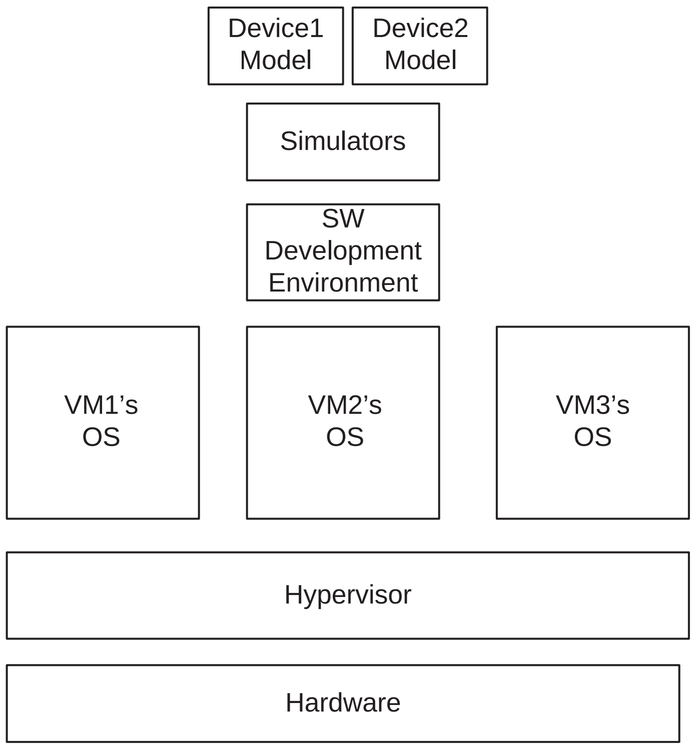


Figure 5.11 An example layering of abstractions allowing programmers to be functional at a higher level of abstraction.

work properly. Such full-stack problems can be difficult to solve, particularly for programmers who are only aware of the level in which they wrote the program. Furthermore, complex systems integrate a number of these environments, each with its own levels of abstraction to manage complexity. Stability in abstraction levels has been helpful in making progress. For example, for many years programmers would have to contend with very different operating systems and to re-write their software to deal with a new operating system interface. With the consolidation to a few commonplace operating systems, this has allowed programmers to have a relatively stable interface to an operating system such as UNIX, Linux, or Windows.

With Figure 5.12, we can see a number of major factors that have changed the programmers' toolset. One of those is computer science, which has provided techniques to enable tools such as programming language parsers and compilers to be created more easily. Early on, large, government-run projects such as SAGE heavily influenced how projects were managed and the resulting development methodologies. Development methodologies have also stimulated the

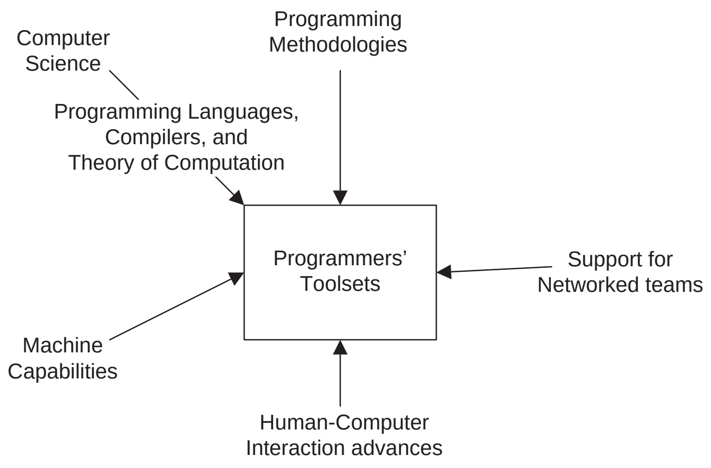

Computer Science
Programming Methodologies
Programming Languages, Compilers, and Theory of Computation
Programmers' Toolsets
Support for Networked teams
Machine Capabilities
Human-Computer Interaction advances

Figure 5.12 Influences on programmer tools.

development of programming tools such as the development of CASE tools. CASE tools were driven by the underlying methodology, such as structured systems analysis and design. As networking support became more prevalent in operating systems and bandwidth became less of a concern, distributed systems and programming teams became more common. Supporting these teams required more tools to better enable communication and synchronization of work across time zones and cultures. As computing power increased, there was sufficient power to support programmers' tools without impacting other work as significantly. Human computer interfaces and graphical user interfaces enabled programmers to build more sophisticated and graphical programming environments such as IDEs.

5.5 Case Study: SAGE

As noted earlier in this chapter, the SAGE (Semi-Automatic Ground Environment) Project (see Sackman [1967]) was an early, real-time system with a large software component.[5] This large project was built to help the North American Aerospace Defense Command (NORAD) respond to a possible Soviet air attack. This huge project was one of the first large projects to computerize a defense system. By its nature, it was a real-time system that had to detect and respond to events as they occurred. SAGE was a multi-billion-dollar project that consumed a large percentage of the available programmers in the United States but also gave them a

5. Also see a US government promotional video on the SAGE project at https://www.youtube.com/watch?v=vzf88oM9egk and a 1983 issue of *Annals of the History of Computing* including Everett et al. [1983] and Jacobs [1983].

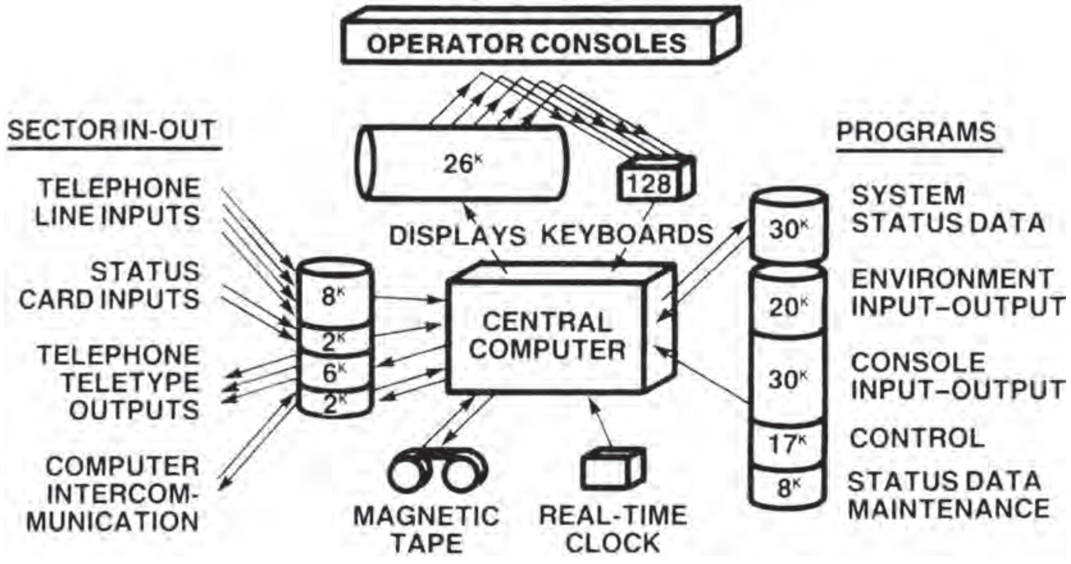


Figure 5.13 SAGE AN/FSQ-7 programs overview. (Source: ACM, Eastern Computer Conference Proceedings, 1957, p. 152, figure 9.)

shared experience, methodology, and approach to large computerized systems. This shared experience led to some commonality in approach with later real-time large systems, particularly with the involvement of one large consulting company, Systems Development Corporation, that used the experience on other projects. The SAGE project was started in 1953, when the USAF chose the SAGE concept and began development of a prototype by MIT's Lincoln Laboratory. It was initially built using IBM's AN/FSQ-7 computer. This prototype was called the Experimental SAGE Subsector. The first operational site was the New York Air Defense Sector in June 1958. SAGE was initially conceived as tracking Soviet aircraft, and with the emerging threat of intercontinental ballistic missiles (ICBMs), other systems had to be added to deal with the ICBM threat.

The AN/FSQ-7 computer was quite advanced for its time and the system included many advanced features, leveraging the work done on Whirlwind computers at the MIT Lincoln Lab. SAGE was built as a redundant system and to operate 24×7. The AN/FSQ-7 was physically very large: weighing 113 tons, consuming 1,500 kilowatts of power, and occupying an entire floor. In terms of processing capacity, it was very small by today's standards: 58,000 vacuum tubes, 69,000 (began with 8,000) words of memory, and 12 magnetic drums, each with a capacity of 150,000 words. The operational real-time program grew to around 100,000 instructions and was partitioned into 40 subprograms. See Figure 5.13 and Figure 5.14.

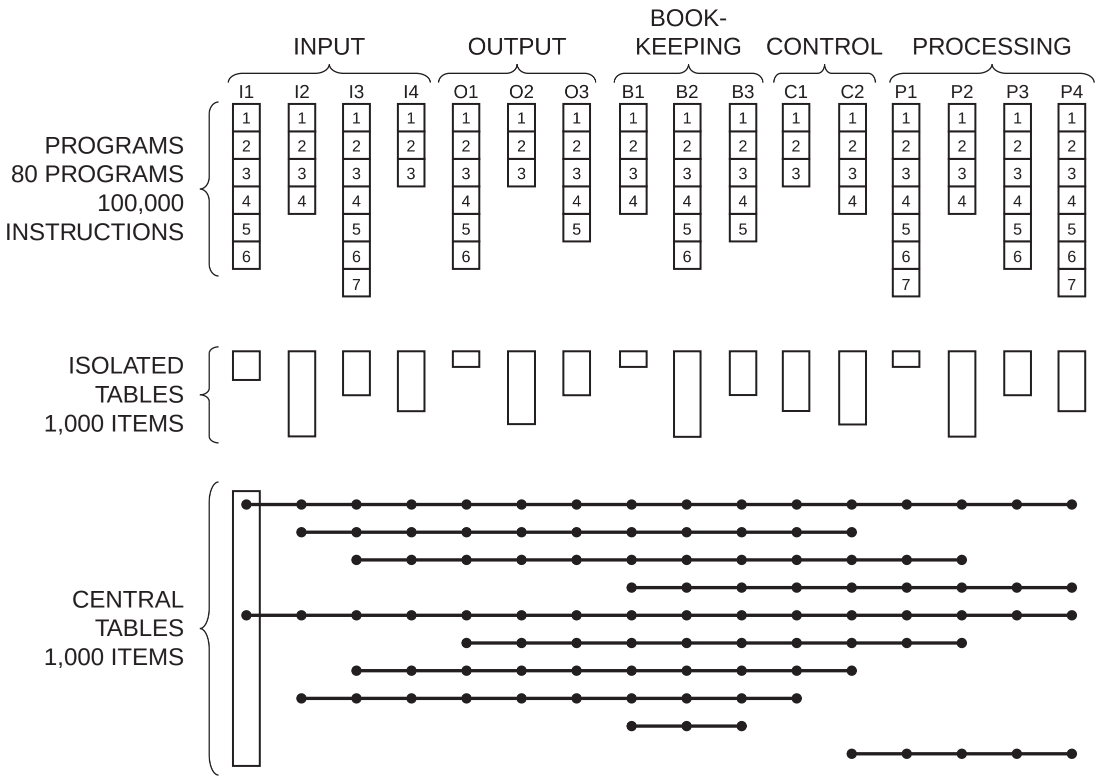


Figure 5.14 SAGE static program organization from Everett et al. [1983]. (Source: ACM, Eastern Computer Conference Proceedings, 1957, p. 152, figure 11.)

The methodology used was largely a waterfall-type model, with one phase feeding into the next. The time it took a program from the time of initial development until it became operational was between 13 and 52 months. Besides being a technical challenge, SAGE was viewed as a system that included people and computerized resources. The concern was that the operation of the entire system (including people) behaved as it should.

SAGE ended up hiring a large number of programmers. The techniques used there were then brought with those programmers when they went to other projects and companies. As a result, the effects of SAGE were long-lasting and reflected in many other large projects.

5.6 Case Study: GNU Emacs

The Emacs text editor has a long history dating back to 1976. Originally standing for Editor MACroS, it was written in support of the TECO editor on the Incompatible

Figure 5.15 GNU (GNU's Not UNIX) project logo. (Source: Aurélio A. Heckert, GNU Logo, 2003, https://www.gnu.org/graphics/heckert_gnu.html.)

Timesharing System (ITS) being used by the Artificial Intelligence Lab at MIT.[6] The idea of WYSIWYG (What You See Is What You Get) editors was starting to become popular and Richard Stallman had seen an editor called "E" (see Samuel [1980]) at the Stanford Artificial Intelligence lab in the early 1970s that was a full-screen editor. Stallman had previously modified the TECO editor to allow the entire file to edited at once in a single buffer as well as adding WYSIWYG functionality. Also in 1976, Bill Joy while at University of California–Berkeley wrote the *vi* ("visual") editor that became part of the UNIX operating system standard toolset. *vi* was based on features from the *Bravo* WYSIWYG editor produced in 1974 by Xerox PARC for the Xerox Alto personal computer and considered the first WYSIWYG document preparation system. As a result, UNIX programmers after this time were often very partial to either Emacs or *vi*, but not both, setting up a semi-religious dichotomy of editor preference.

Richard Stallman began development of GNU (a recursive acronym for GNU's Not UNIX) Emacs in 1984 in order to produce a free software alternative to proprietary versions of Emacs. This development was critical to the beginnings of the "free software" movement that produced the set of GNU tools (for the project logo see Figure 5.15) which, along with the Linux operating system, gave a complete toolset to programmers that was non-proprietary and free to modify. As part of

6. TECO (Text Editor & Corrector) was originally developed in 1962 for Digital Equipment Corporation computers and was written for the PDP-1 by Dan Murphy while he was a student at Massachusetts Institute of Technology.

this desire to make Emacs and other software free to use, Stallman included with Emacs the notice below that he required be carried with any distribution of the source code and was the beginning of what Stallman calls a "copy-left" protection notice (see Listing 5.1), which preserves the rights to re-use and modify the code.[7] By requiring the inclusion of this notice, it propagates the ability to freely modify and use the source code.

```
GNU Emacs copying permission notice Copyright (C) 1985 Richard M. Stallman
     Verbatim copies of this document, including its copyright notice,
     may be distributed by anyone in any manner.
     Distribution with modifications is not permitted.

GNU Emacs is distributed in the hope that it will be useful,
but without any warranty.  No author or distributor
accepts responsibility to anyone for the consequences of using it
or for whether it serves any particular purpose or works at all,
unless he says so in writing.

Everyone is granted permission to copy, modify and redistribute
GNU Emacs under the following conditions:

   Permission is granted to anyone to make or distribute verbatim copies
   of GNU Emacs source code as received, in any medium, provided that all
   copyright notices and permission and nonwarranty notices are preserved,
   and that the distributor grants the recipient permission
   for further redistribution as permitted by this document,
   and gives him and points out to him an exact copy of this document
   to inform him of his rights.

   Permission is granted to distribute modified versions
   of GNU Emacs source code, or of portions of it,
   under the above conditions, provided also that all
   changed files carry prominent notices stating who last changed them
   and that all the GNU-Emacs-derived material, including everything
   packaged together with it and not independently usable, is
   distributed under the conditions stated in this document.

   Permission is granted to distribute GNU Emacs in
   compiled or executable form under the same conditions applying
   for source code, provided that either
    A.  it is accompanied by the corresponding machine-readable
      source code, or
    B.  it is accompanied by a written offer, with no time limit,
      to give anyone a machine-readable copy of the corresponding
      source code in return for reimbursement of the cost of distribution.
      This written offer must permit verbatim duplication by anyone.
    C.  it is distributed by someone who received only the
      executable form, and is accompanied by a copy of the
      written offer of source code which he received along with it.

In other words, you are welcome to use, share and improve GNU Emacs
You are forbidden to forbid anyone else to use, share and improve
what you give them.   Help stamp out software-hoarding!
```

Listing 5.1 Stallman's rights-preserving notice included with Emacs source code.

7. The version included in these listings is from Emacs 16.56 released on July 15, 1985, with source code from that release extracted from https://github.com/larsbrinkhoff/emacs-16.56/blob/master/src/emacs.c.

In Listing 5.2 is part of the source code for the *emacs.c* file written in C. In this file, you'll see a number of references to the LISP programming language (such as in lines 26, 93, and 125) and the use of *DEFUN*, which comes from the Common LISP programming language. Emacs included a number of features that made it easier to edit LISP source code as well as having been first created for AI programmers.

```
 1 /* Fully extensible Emacs, running on Unix, intended for GNU.
 2    Copyright (C) 1985 Richard M. Stallman.
 3 This file is part of GNU Emacs.
 4 GNU Emacs is distributed in the hope that it will be useful,
 5 but without any warranty.  No author or distributor
 6 accepts responsibility to anyone for the consequences of using it
 7 or for whether it serves any particular purpose or works at all,
 8 unless he says so in writing.
 9 Everyone is granted permission to copy, modify and redistribute
10 GNU Emacs, but only under the conditions described in the
11 document "GNU Emacs copying permission notice".   An exact copy
12 of the document is supposed to have been given to you along with
13 GNU Emacs so that you can know how you may redistribute it all.
14 It should be in a file named COPYING.  Among other things, the
15 copyright notice and this notice must be preserved on all copies.  */
16 .
17 .
18 .
19 . // Part of file extracted by this text's author
20 .
21 .
22 .
23 DEFUN ("kill—emacs", Fkill_emacs, Skill_emacs, 0, 1, "P",
24   "Exit␣the␣Emacs␣job␣and␣kill␣it.␣␣Arg␣means␣no␣query.")
25   (arg)
26      Lisp_Object arg;
27 {
28   Lisp_Object answer;
29   int modbufcount;
30 
31   if (feof (stdin))
32     arg = Qt;
33   if (NULL (arg) && (modbufcount = ModExist())
34       && (answer = Fyes_or_no_p (format1 (
35 "%d␣modified␣buffer%s␣exist%s,␣do␣you␣really␣want␣to␣exit?␣",
36                                      modbufcount, modbufcount == 1 ? "" : "s",
37                                      modbufcount == 1 ? "s" : "")),
38           NULL (answer)))
39     return Qnil;
40 
41 #ifdef subprocesses
42   if (NULL (arg) && count_active_processes()
43       && (answer = Fyes_or_no_p (format1 (
44 "Subprocesses␣are␣executing;␣kill␣them␣and␣exit?␣")),
45           NULL (answer)))
46     return Qnil;
47 
48   kill_buffer_processes (Qnil);
49 #endif
50 
51   Fdo_auto_save (Qt);
52   fflush (stdout);
53   RstDsp ();
54   exit (0);
55 }
56 
57 DEFUN ("dump–emacs", Fdump_emacs, Sdump_emacs, 2, 2,
58   "FDump␣as␣file:␣\nfWith␣symbols␣from␣file:␣",
  "Dump␣current␣state␣of␣Emacs␣into␣executable␣file␣FILENAME.\n\
Take␣symbols␣from␣SYMFILE␣(presumably␣the␣file␣you␣executed␣to␣run␣Emacs).")
  (intoname, symname)
     Lisp_Object intoname, symname;
{
  register unsigned char *a_name = 0;
  extern int my_edata;
  Lisp_Object tem;
  extern _start ();

  CHECK_STRING (intoname, 0);
  intoname = Fexpand_file_name (intoname, Qnil);
  if (!NULL (symname))
    {
      CHECK_STRING (symname, 0);
      if (XSTRING (symname)->size)
        {
          symname = Fexpand_file_name (symname, Qnil);
          a_name = XSTRING (symname)->data;
        }
    }

  tem = Vpurify_flag;
  Vpurify_flag = Qnil;

  fflush (stdout);
  malloc_init (&my_edata);      /* Tell malloc where start of impure now is */
  unexec (XSTRING (intoname)->data, a_name, &my_edata, 0, _start);

  Vpurify_flag = tem;

  return Qnil;
}

Lisp_Object
decode_env_path (evarname, defalt)
     char *evarname, *defalt;
{
  register char *path, *p;
  extern char *index ();

  Lisp_Object lpath;

  path = (char *) getenv (evarname);
  if (!path)
    path = defalt;
  lpath = Qnil;
  while (1)
    {
      p = index (path, ':');
      if (!p) p = path + strlen (path);
      lpath = Fcons (p - path ? make_string (path, p - path) : Qnil,
                     lpath);
      if (*p)
        path = p + 1;
      else
        break;
    }
  return Fnreverse (lpath);
}

syms_of_emacs ()
{
  defsubr (&Sdump_emacs);
  defsubr (&Skill_emacs);

  DefLispVar ("command-line-args", &Vcommand_line_args,
    "Args passed by shell to Emacs, as a list of strings.");

  DefLispVar ("system-type", &Vsystem_type,
    "Symbol indicating type of operating system you are using.");
  Vsystem_type = intern (SYSTEM_TYPE);

  DefBoolVar ("noninteractive", &noninteractive,
    "Non-nil means Emacs is running without interactive terminal.");
}
```

Listing 5.2 Source code excerpt of Emacs from file *emacs.c*.

Emacs and its ability to edit the whole file at once (such as being able to do global replaces) in a WYSIWYG manner forms the basis of how modern text editors work. GNU Emacs was a reaction to the inability to freely use and modify proprietary software and was a stimulus for the free software and open-source software movements.

5.7 Case Study: AUTOFLOW

One of the first software patents was issued in 1970 to Martin Goetz for a program that would analyze source code and produce a flowchart as output.[8] With the AUTOFLOW patent application (US Patent number 3,533,086), he included the assembly language source code for the entire program written for an RCA 501 computer. With this software tool and the associated patent issued to Applied Data Research, AUTOFLOW became the first stand-alone commercial software product. RCA and other mainframe manufacturers did not license AUTOFLOW, so Goetz decided to sell it directly to RCA mainframe customers and it was eventually ported to several other mainframe systems, becoming a commercial success and proving that the stand-alone software model was feasible.

Flowcharts as a method for describing the design of computer programs had been popular since the ENIAC [Haigh et al. 2016] and continued to have a wide variety of uses in describing programs, among other things [Ensmenger 2016]. Being so ingrained in the way programs were designed at the time, AUTOFLOW was a helpful program to document already existing programs by using flowcharts.

The patent for AUTOFLOW uses the AUTOFLOW program itself to document the code for AUTOFLOW. This intriguing self-application of the program to itself lives on as a test of many programming tools. Programming language compilers

8. The first US software patent was Martin Goetz's earlier patent for a sorting program, issued on April 23, 1968, US Patent number 3,380,029. An even earlier British patent (GB1039141A) for *A Computer Arranged for the Automatic Solution of Linear Programming Problems* was issued to British Petroleum Company on August 17, 1966, and was concerned with using the simplex algorithm on a Ferranti Mercury computer in a memory efficient manner.

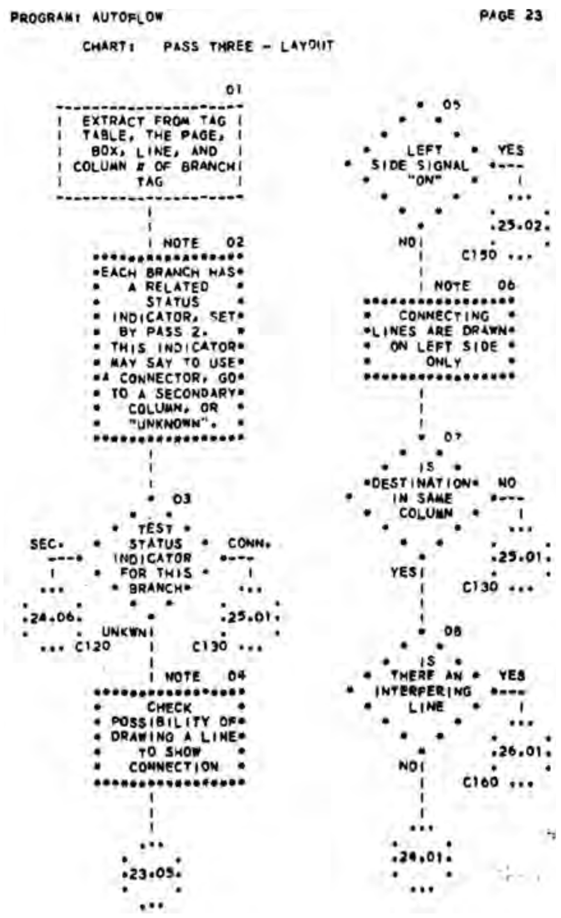

PROGRAM: AUTOFLOW
PAGE 23
CHART: PASS THREE - LAYOUT
01
EXTRACT FROM TAG TABLE, THE PAGE, BOX, LINE, AND COLUMN # OF BRANCH TAG
NOTE 02
EACH BRANCH HAS A RELATED STATUS INDICATOR, SET BY PASS 2. THIS INDICATOR MAY SAY TO USE A CONNECTOR, GO TO A SECONDARY COLUMN, OR "UNKNOWN".
03
TEST STATUS INDICATOR FOR THIS BRANCH
SEC.
CONN.
24.06.
25.01.
UNKWN
C120
C130
NOTE 04
CHECK POSSIBILITY OF DRAWING A LINE TO SHOW CONNECTION
23.05.
05
LEFT SIDE SIGNAL "ON"
YES
25.02.
NO
C150
NOTE 06
CONNECTING LINES ARE DRAWN ON LEFT SIDE ONLY
07
IS DESTINATION IN SAME COLUMN
NO
25.01.
YES
C130
08
IS THERE AN INTERFERING LINE
YES
26.01.
NO
C160
24.01.

Figure 5.16 AUTOFLOW example output. (Source: Martin A. Goetz, US Patent 3,380,029.)

are often tested to see if you can write a compiler for the language in the same language. An example of the program flow from the AUTOFLOW patent as produced by the AUTOFLOW documentation program is in Figure 5.16. This example shows part of the program's processing of the code to produce the layout of the output. It uses character (rather than graphics) representations of the flowchart boxes and connections to other parts of the flowchart. Processing boxes are rectangular, decision boxes are diamond-shaped, and connections to other flowchart diagrams are round. The RCA 501 assembly language source code included in the patent uses the comment fields of each command as well as REMARK (RMK) assembly language instructions to document the code and to use as input to the program.

The AUTOFLOW product was an early example of the demand for programming tools and an example of the popularity of tools that would help automate the creation of program documentation. Software patents, on the other hand, have had a turbulent history with results such as the US Supreme Court deciding in 1981 that "a claim drawn to subject matter otherwise statutory does not become non-statutory simply because it uses a mathematical formula, computer program, or digital computer." By the early 1990s, the US laws more clearly allowed software patents and added that the "practical application of a computer-related invention is statutory subject matter." This resulted in a number of software patents being issued such as the "One Click" patent (US Patent 5960411A, Method and System for Placing a Purchase Order Via a Communications Network) issued to Amazon on September 20, 1999. The One Click patent was viewed as controversial due to the simple nature of the patent and many considered it simply a business method rather than a novel use of software.

5.8 Lessons Learned from Programming Tools and Environments

Programming languages' toolsets and methodologies continue to change and evolve as we struggle to build systems in a predictable manner. Along with programming languages themselves, these environments and methodologies are central to how we build systems and as a result a great deal of effort has been put into them.

Some of the specific lessons learned from programming tools, environments, and programming methodologies include:

- *Engineering methods from other disciplines have had mixed results.*

  With the software crisis and since the beginning of software engineering in the 1960s, there have been attempts to re-use a lot of the engineering techniques that have been successful in other areas of engineering such as building interchangeable components, having standard engineering models, and other techniques re-tooled to try to use in software. These have had mixed results. The issue with software is that the level of change is very high, with programming languages, toolsets, and underlying hardware changing very rapidly. This has made it difficult to build long-lasting, re-usable software components. However, we have been able to use general architecture and design concepts practices, much of which has been borrowed from other engineering disciplines and architecture.[9] The idea of design patterns has produced a reliable way to propagate use of good software design. Software architectural patterns and reference architectures have also been a way to re-use ideas and approaches from prior successful software systems.

9. From architecture, Christopher Alexander (see Alexander et al. [1977]) has been influential in the idea of creating design patterns as well as agile software development techniques.

CASE techniques had limited success with particular classes of applications, but their limitations proved to hamper their utility and resulted in their decline in use. Many other techniques to automatically build systems from formal specifications, methods, and designs are promising and is an area of research that many have high hopes will be able to produce large-scale, provably correct systems.

Another issue with software that makes it difficult to apply techniques from other engineering disciplines is that software is often faced with novel problems to be solved that have no grounding in physics. After all, why would an organization give high salaries to programmers that are not solving new problems? That is, new algorithms are often needed to solve a new problem and the way it is solved by software is not constrained by forces in the real world (as a bridge would be). This freedom from physical constraints can enable myriad ways to solve the new problem, each of which has its own pluses and minuses.

- *Very large systems have stressed tools and methodologies.*

  Very large software projects have not only reinforced the need for methodologies but have also made clear the weaknesses in our methodologies. Many of these large projects were often way over budget and took much longer than expected because of the difficulties and unpredictable nature of software. These large projects have also stressed software in other ways such as maintainability, performance, availability, reliability, and security. Large projects have also required large numbers of people to participate and helped to design and clarify how to use systems such as source code control systems and methodologies and design practices that separate system development into individual tasks that can be more reliably integrated into a large system.

- *Some methodology is very helpful (as compared to **no** methodology).*

  We've also learned that some software development methodology is helpful for any project involving multiple people. Even with Agile and eXtreme programming methods, there is some structure that helps the project have a better chance of success. Too often organizations will apply a methodology without trying to tune it to the particular project(s) using it and increase the frustration of the teams by including work that the team cannot see as valuable.

- *Programmer toolsets and methodologies change often.*

  Programmer toolsets have changed often and tend to be an object of tinkering by programmers interested in improving their own efficiency. IDEs have often been focused on a particular programming language, which tended to reduce their useful lifespan. IDEs that have allowed modules or plug-ins to deal with specific programming languages have (so far) tended to have a longer useful lifespan. Other toolsets, such as UNIX-like programming commands and tools, have tended to have a longer lifespan due to the propagation and use of UNIX and Linux operating systems.

We've also seen methodologies change dramatically over time with the desire to build systems better. Much of that drive has been to build more reliable systems or to better reflect the customer's needs. With the ENIAC (and for many years after), programming designs were done as flowcharts, often on blueprint paper. Flowcharting continued to be a technique taught to programmers even in the 1980s. Eventually, we found that flowcharts were less than optimal and may actually cause bad program design by encouraging the ability to jump (i.e., GOTO) to any other point in the program. Along with structured analysis and design, flowcharts became less popular. An interesting example of this was the programming language B-0 (a precursor to COBOL)[10] whose implementation was completely defined by flowcharts. Still, flowcharts have some utility if one is trying to reverse-engineer and understand how an old system works, particularly if it is written in assembly language or otherwise allows GOTO statements or arbitrary jumps.

Given the rate of past change in methodologies and programmer toolsets, it's very likely that we'll continue to see change, particularly to improve system reliability, predictable development, and verifiable adherence to user requirements.

5.9 Exercises and Projects

5.9.1 Exercises

1. Investigate the problems that programmers had with paper tape and the lengths they went to repair and maintain them. Explain why paper tape was used for such a long period of time (until at least into the 1980s).
2. Punched cards, such as Hollerith cards, were very widely used into the 1980s. Investigate the efforts to standardize punched card formats. Why were IBM

10. The complete flowcharts are included in Jean Sammet's papers held at the Charles Babbage Institute archives and are almost 100 pages of flowcharts to define the B-0 language for the UNIVAC.

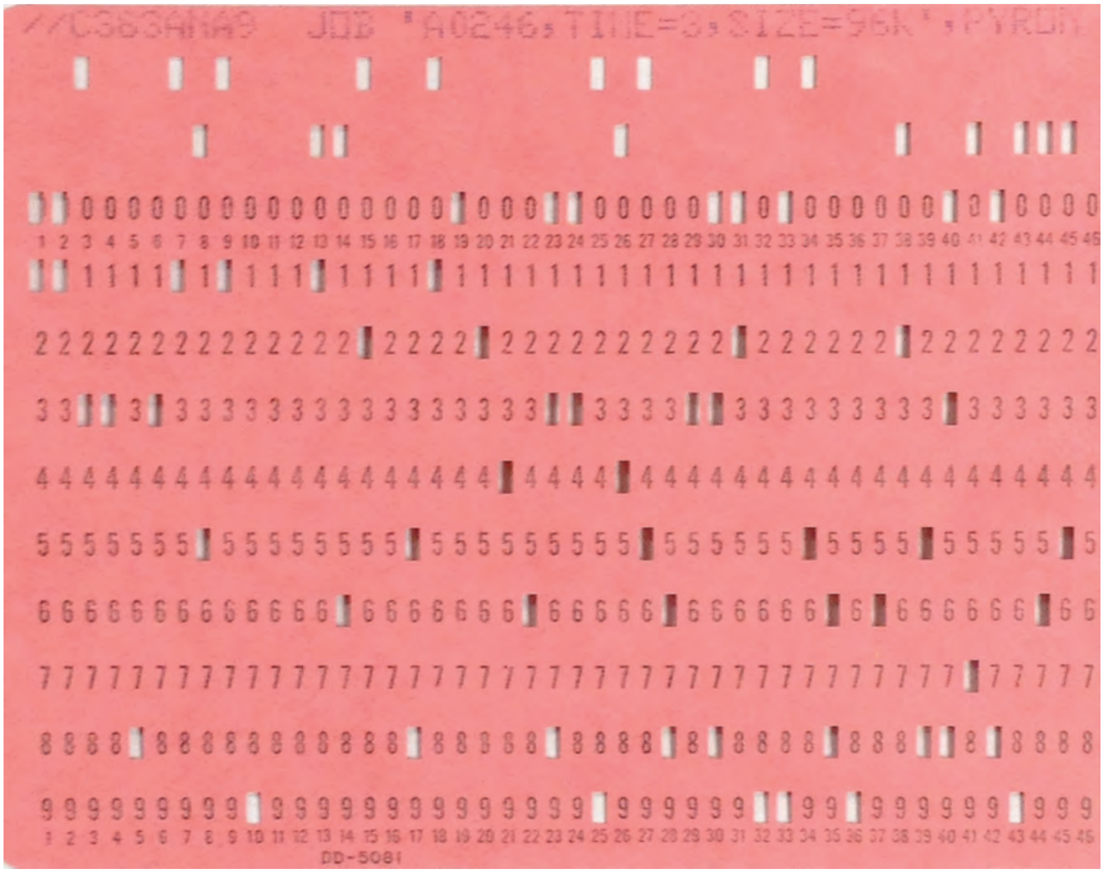


Figure 5.17 An enlarged punched card showing the encoding of portion of an 80-column Job Control Language Statement JOB card. (Source: Photograph by author.)

cards often 80 characters in width, such as the one in Figure 5.17? Note that this card is encoded in Extended Binary Coded Decimal Interchange Code (EBCDIC), which was heavily used on IBM computers, so a *C* is encoded as *C*3 in hexadecimal and *A* as *C*1 in hex. Was it in the best interest of computer manufacturers to standardize the character encoding format and card size in the 1970s? Why or why not?

3. Investigate the relationship of paper tape and card coding formats with network and character codes. For example, how did the Baudot code evolve into use on paper tape? What's the relationship of character standards such as EBCDIC and ASCII to the codes used on punched paper tape and cards?

4. An important programming tool for UNIX systems is the AWK programming tool. Find a use of the AWK[11] programming language and explain why AWK was a good choice for the problem.
5. Find information on IBM's VM-CP (Virtual Machine Control Program) CMS (Conversational Monitor System) and explain its usage from a programmer's point-of-view. Did VM/CMS help programmers be more productive? Can you find contemporary references that detail the benefits and drawbacks of VM/CMS? Summarize what you found.
6. Was UNIX's *SCCS* (Source Code Control System, 1972) the first source code management system? How did you determine that? What is your basis for reaching that conclusion?
7. *lex* and *yacc* were instrumental in the development of other programming tools beyond compilers. Explain how they were used for non-compiler tools.
8. UNIX's SCCS was originally written in the programming language SNOBOL. Why was SNOBOL used for this task?
9. The Software Engineering Body of Knowledge (SWEBOK) is an effort driven primarily by the IEEE Computer Society. Find a copy of the SWEBOK and detail the process that is used (or not) to keep it up to date.
10. Investigate the relationship between *information engineering* as in Martin [1990] and CASE. Which came first? How did the first one influence the one that followed?
11. One technique used in structured systems analysis is the data flow diagram (DFD). Find out more about DFDs and write an interpretation of what the DFD in Figure 5.18 means. This particular drawing is for a DVD rental system and uses syntax similar to Kowal [1988]. DFD syntax varies somewhat, but this syntax is a typical example.
12. Find Clive Finkelstein's series of six articles from *Computerworld* written in June 1981 about information engineering. Why was information engineering considered necessary at that time?
13. The Unified Modeling Language (UML) was an important consolidation of object-oriented modeling languages. Give an example of where the representations were different in Booch's, Jacobson's, and Rumbaugh's object-oriented modeling languages. These three earlier representations were "unified" by UML.

11. AWK was named for its authors: **A**ho, **W**einberger, and **K**ernighan.

14. Find information on DEC's VMS programming environment and explain its usage from a programmer's point-of-view. Did VMS help programmers be more productive compared to other systems available at the same time? Can you find contemporary references that detail the benefits and drawbacks of VMS's programming environment? Summarize what you found.

15. On October 18, 2007, Ivar Jacobson published in his blog (see http://blog.ivarjacobson.com/category/ivarblog/architecture/ which you may have to find via http://archive.org) an entry called "Enterprise Architecture failed big way!" Read this blog entry and comment on how these reasons for failures can apply to other architectural techniques.

16. Consider a very large software project being done today that involves thousands of people and hundreds of millions of lines of code. To pick a specific example, consider a space defense system that involves thousands of geocentric satellites, control centers on the ground, and other active components (missiles, probes, sensors on ships/aircraft, etc.). So, this system needs to work as a single, integrated system.

    Given the state of current technologies, would this project be easier or harder than SAGE was during the 1950s and 1960s? If you argue that it's easier, then identify what advances have been made since SAGE that will help make it more likely to succeed. If you argue that it's going to be harder to complete, then argue why this project would be more difficult to successfully complete than SAGE.

17. A particular set of software components can show a lot of re-use for a while, but over time the ability to re-use those components degrades. Explain why a particular library or set of "re-usable" components becomes less re-usable over time and is often abandoned for something else.

5.9.2 Projects

1. Investigate the concept of "structured programming" such as in Linger et al. [1979] and its relationship to structured systems analysis and design methodologies such as in DeMarco [1978]. How did the structured analysis and design methodologies live up to the goals of structured programming? How did the methodologies stray from the intent of structured programming? How are structured programming concepts represented in object-oriented analysis and design methodologies? Write a paper that shows the relationships between these related topics.

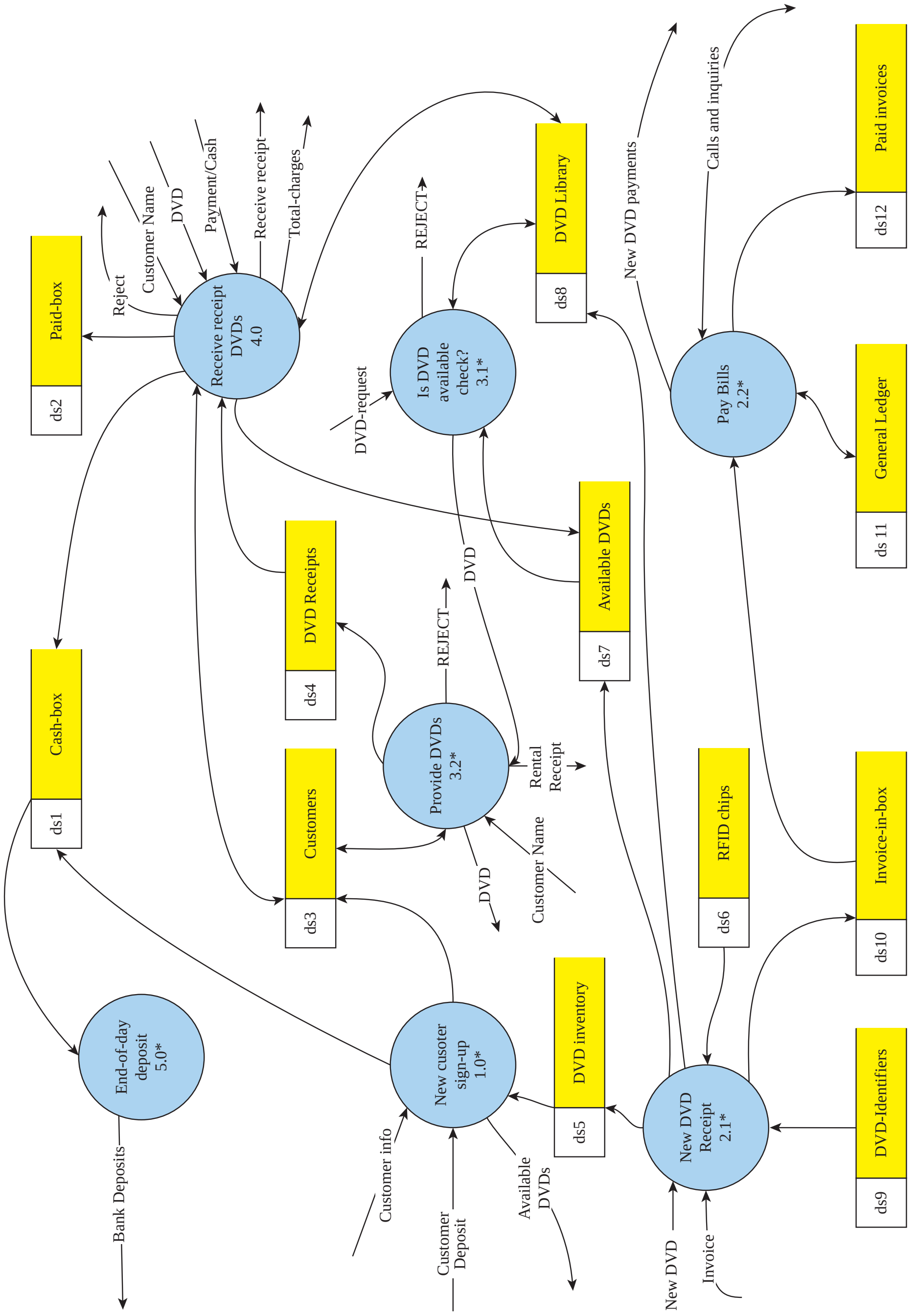


Figure 5.18 An example level 1 data flow diagram from structured systems analysis for a DVD rental system.

2. Find out more about the software and overall project methodology used by the SAFEGUARD Project[12] The resulting methodology impacted other projects undertaken at Bell Labs after SAFEGUARD. For the overall purpose of SAFEGUARD see Brown et al. [1975] and other articles in that issue of the Bell System Technical Journal. Like the SAGE project, SAFEGUARD was an antiballistic missile system that was designed to respond to attacks by ICBMs. Its subsystems include: a missile subsystem, a radar subsystem, and a data processing and control subsystem. The idea is that incoming missiles would be detected by SAFEGUARD and then destroyed by defensive missiles. See Figure 5.19 and Figure 5.20 for a high-level view of the system.

   Write a report that details the software methodology used, how the SAGE project methodology was related and detail any cross-communication of methodology ideas, and how the SAFEGUARD methodology influenced other large software projects at Bell Labs and elsewhere.

3. Software design patterns (for example see Rising [1998]) have become one of the most successfully re-used concepts for software development. These patterns are re-usable solutions to commonly occurring problems within a given context. Examine how these were developed and how they have changed over time. Examine the motivations for the initial creation of design patterns and what was attempted before them. Create a timeline explaining their relevance and surmise how they might evolve in the future.

4. A method of structured analysis and design was called *composite design,* as recorded in books such as Myers [1975]. Investigate this technique and how it was used and why it is no longer used. What other techniques did it influence? What previous techniques influenced it? Why was it expected to help produce more reliable software?

5. There is a relationship between structured analysis and design, information engineering, CASE, and enterprise architecture. Produce a report that explains the interrelationships and how this set of techniques evolved over time.

6. Investigate how modern tools are attempting to add more careful design and review back into the process of creating software. While the punched card and tape era forced a careful process, it also imposed the keeping track of many details better done by computerized processes. Techniques such as

12. One good reference is the 1975 *Bell System Technical Journal* (see Brown et al. [1975]) supplement that was focused on the SAFEGUARD Data-Processing System and is available at http://srmsc.org/pdf/005213p0.pdf.

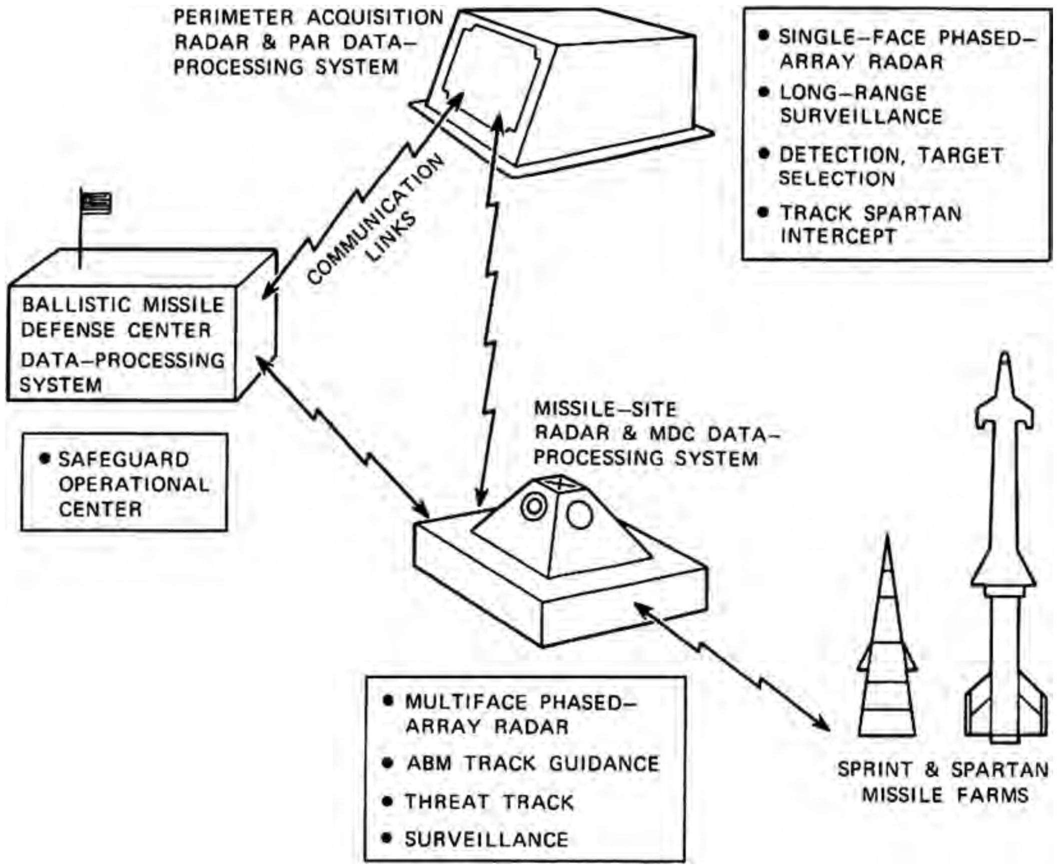

PERIMETER ACQUISITION RADAR & PAR DATA-PROCESSING SYSTEM
• SINGLE-FACE PHASED-ARRAY RADAR
• LONG-RANGE SURVEILLANCE
• DETECTION, TARGET SELECTION
• TRACK SPARTAN INTERCEPT
COMMUNICATION LINKS
BALLISTIC MISSILE DEFENSE CENTER DATA-PROCESSING SYSTEM
• SAFEGUARD OPERATIONAL CENTER
MISSILE-SITE RADAR & MDC DATA-PROCESSING SYSTEM
• MULTIFACE PHASED-ARRAY RADAR
• ABM TRACK GUIDANCE
• THREAT TRACK
• SURVEILLANCE
SPRINT & SPARTAN MISSILE FARMS

Figure 5.19 The SAFEGUARD System for detecting and responding to missile attacks. (Source: Reused with permission of Nokia Corporation and AT&T Archives.)

model-driven development and other tools encourage solving the problem at a more abstract level that are then translated to running programs. Write a paper that describes how careful design in the age of paper tape and cards can be and already is being re-introduced into the software development process.

7. The AUTOFLOW program used the comments and RMK instructions in the program itself as well as analyzing the instructions to determine where decisions were made to create flowcharts for the program being analyzed. Using the code included with the US patent number 3,533,086, map the places in the program that correspond to Figure 5.16 by showing the lines of code that correspond to each block (with line numbers from the patent) and the assembly language instructions that correspond to the decision boxes.

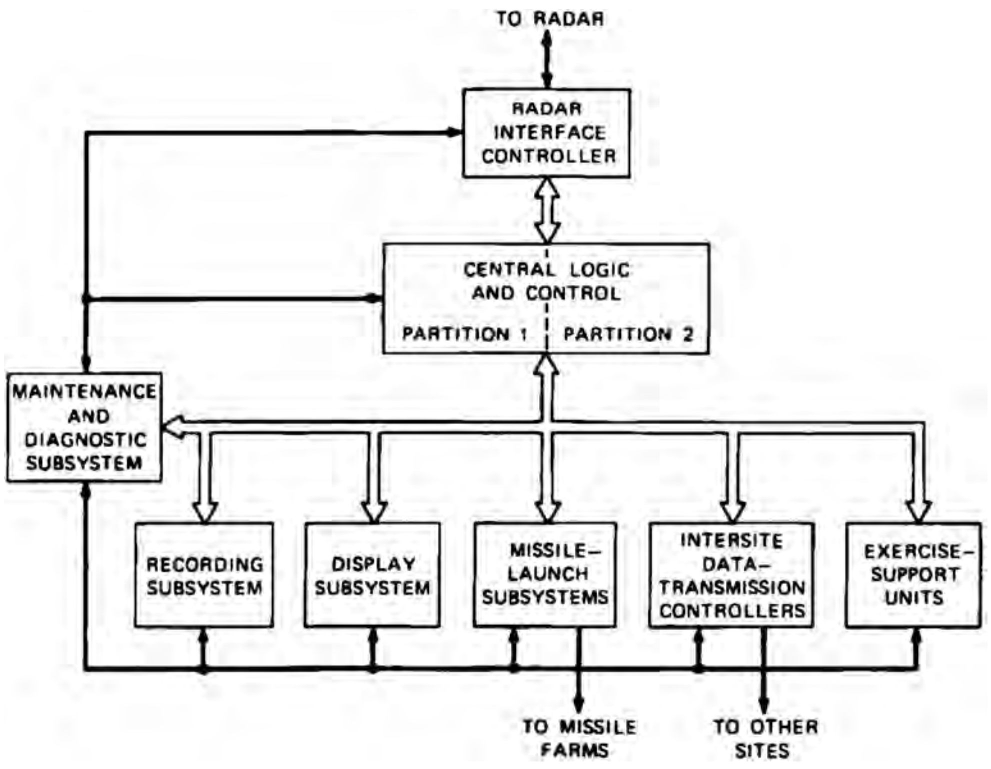

TO RADAR
RADAR INTERFACE CONTROLLER
CENTRAL LOGIC AND CONTROL
PARTITION 1
PARTITION 2
MAINTENANCE AND DIAGNOSTIC SUBSYSTEM
RECORDING SUBSYSTEM
DISPLAY SUBSYSTEM
MISSILE–LAUNCH SUBSYSTEMS
INTERSITE DATA–TRANSMISSION CONTROLLERS
EXERCISE–SUPPORT UNITS
TO MISSILE FARMS
TO OTHER SITES

Figure 5.20 The SAFEGUARD System data processing components. (Source: Reused with permission of Nokia Corporation and AT&T Archives.)

8. A batch compilation processing using cards and/or tape along with coding forms led to a natural tendency to be careful about the program's design and syntax in order to maximize the usefulness of each attempt. Moving to a time-shared environment allowed programmers to be more dependent on the compiler and other tools to find syntax errors and to more quickly test a program. Programmers came to depend on the programming methodology to enforce careful architecture and detailed system design, which often were supported with explicit steps in waterfall, structured analysis and design, and object-oriented systems analysis and design. With Agile methodologies programmers may not be required to create the entire system architecture early and detailed design is often spread over sprints and may not be well-documented. Document the different approaches used for architecture and design as applied to Agile methodologies and evaluate them based on their ability to create a reliable software architecture and detailed software

design. Consider alternatives such as *attribute-driven design* as in Cervantes and Kazman [2016].

5.10 Further Readings and Online Resources

For sets of classic papers in software engineering, see Yourdon [1979] and Yourdon [1982]. For the methods and explanations of the diagrams used for the ENIAC, see Haigh et al. [2016]. Sackman [1967] contains many details on SAGE and its development methods. Kernighan and Plauger [1976] details the UNIX toolset, particularly those that manipulate text files.

6 Networking Software

The desire to quickly send information over long distances has a long history (see Holzmann and Pehrson [2003]). This history has colored how computer data networking has evolved and techniques used to send information before computer networking has impacted the methods for sending information over computer networks. Data networking has had significant influence over software systems and how they are architected, as well as how the data networking software itself is built. Software architectures such as client–server, peer-to-peer, cloud-based systems, and hosted services such as *infrastructure as a service* (IaaS), *platform as a service* (PaaS), and *application as a service* (AaaS) have all depended on networking to be feasible. Advances in data networking have enabled those types of systems to be built by becoming more pervasive, inexpensive, higher bandwidth, and lower latency. Developments such as the popularization of the Internet have irrevocably changed the way programming is done. The popularity of the Internet drove applications to be written for it as well drove a standardization on the Internet Protocol (IP), replacing a number of other competing protocols.

This chapter covers networking software and how it has evolved over time, including influences from different areas such as telecommunications, computer networking, and broadcast networks.

6.1 Overview of the Evolution of Data Networking

The ability to accurately send information over long distances has been needed in order to command armies and to rule over empires. These efforts led to techniques to ensure the accuracy, secrecy, and timeliness of information sharing. This section discusses some of that early history as well as influential information-sharing networks that impacted how computer networking was implemented.

6.1.1 Information Networking Before Computers

In ancient times, urgent information was often carried by runners and relays of runners and horse-riders. Homing pigeons were also used for delivery of messages.

Another method was the use of mirrors and flags to send messages between points within direct line of sight. For data communication, this is where it begins to be interesting as this required the use of a particular code that was understood by each party involved in the communication. Beacons (such as fire beacons or lighthouses) also required some commonly understood way of encoding the message being relayed. These sorts of "telegraph" networks required not only a message encoding scheme but also a common protocol for all parties to follow that would specify the beginning and ending of a message, determining who goes first, handling garbled messages, etcetera. These networks began to look more like data communication networks with their protocols, message formats and encoding, synchronization methods, and methods of routing and quality assurance. Sending reliable messages quickly between distant cities became possible with repeater stations and common protocols.

An important example was the semaphore (optical) telegraph that was constructed in the latter part of the 18th century in France with the direction of Claude Chappe (see Holzmann and Pehrson [2003]). The first line between Paris to Lille was completed in 1794 and was regularly transmitting messages the 120 miles (190 km) between the two cities. The network was later expanded to a number of cities into the 1850s and developed an extensive code for both messages and protocols for message control. The electrical telegraph was developed in the 1830s by Samuel Morse, but France was not ready to invest in the new technology at that time. Familiar themes of the cost of installing an inferior technology (it depended on a physical electrical connection, where the optical telegraph was wireless and more difficult to cut). Interestingly, when the Foy–Bréguet electrical telegraph was deployed in France it utilized the already used Chappe codes from the optical telegraph.

The deployment of various telegraph technologies around the world continued as described in Holzmann and Pehrson [2003]. In the late 1800s, telephone networks began to appear with the inventions of Alexander Graham Bell and Elisha Gray.[1] Telephone networks quickly spread and with the formation of the Bell System regulated monopoly in the United States after AT&T was nationalized. The promise of AT&T's president Theodore Vale of "universal service" ushered a rapid expansion of phone networks and technology in order to serve all parts of the US. This rapid expansion of the span of the telephone network as well as increased usage made the solutions to issues such as call routing and managing

1. Elisha Gray did similar work on telephone devices but lost to Bell in numerous court decisions on the telephone patent. He also helped form Western Electric that eventually became the manufacturing arm of AT&T. See Adams and Butler [1999].

large numbers of calls a challenge. Figure 6.1 shows a Bell System ad indicating that 70 million calls a day were already occurring on the AT&T network by 1939.

6.1.2 Communications Networks Contributing to Computer Networking

Sending information over long distances has a long history that colors and contributes to the way that data networking technology developed as well as the current efforts to consolidate networks largely around TCP/IP (Transmission Control Protocol/Internet Protocol). Each of these earlier efforts not only contributed technologies and approaches but many also built networks that produced their own inertia to change. In particular, networks to send and receive data have

70 Million
Times a Day
BELL TELEPHONE SYSTEM

Seventy million times a day the public tests the quality of Bell System service. The measure of this service is not only its promptness, reliability and low cost. It is also the courtesy with which it is given.

Our genuine desire is to make the Bell System a friendly and helpful institution . . . and to give you the best, the most and the cheapest telephone service in the world.

You are cordially invited to visit the Bell System exhibit at the New York World's Fair

Figure 6.1 A 1939 Bell System advertisement describing the number of calls handled per day. (Source: Courtesy of AT&T Archives and History Center.)

been developed in separate industries and communities, each of which had differing motivations and requirements. Some of the most influential areas have been the following:

- *Telecommunications.* Stemming from telegraph and earlier efforts to send messages long distances, telecommunications evolved with the ability to create large-scale, worldwide networks that were extremely reliable and able to handle large volumes of voice traffic. However, much of this work was done before, and then in parallel with the effort to build computer networks. The earliest computer networks would use the preexisting telecommunications networks in order to provide long-distance connections between computers. Telephone networks also required a method of addressing and routing of calls. Teletypes were developed to run over the telephone network and used modems as well as their own routing techniques.[2]
- *Radio and television.* Broadcast networks, such as radio and television, also developed independently from other networks and excelled at wireless as well as the ability to disseminate large amounts of information, albeit in a unidirectional manner. These eventually transformed into multiple methods of distribution including over coaxial cable networks and others. These networks gradually transformed into providing a base for access networks into homes.
- *Satellite.* The launching of Sputnik 1 by the Soviet Union on October 4, 1957, significantly accelerated the space race with the United States during the Cold War. While Sputnik 1 only broadcast a radio pulse, it was detectable by amateur radio operators around the world. The United States then launched the SCORE (Signal Communications for Orbiting Relay Equipment) satellite, which is considered the first communications satellite and was a direct response to the launch of Sputnik. Interestingly, SCORE was the first project for the newly formed Advanced Research Projects Agency (ARPA). SCORE used a store and forward method to broadcast a message. It is relatively

2. Teletypes would print information on tapes that could then be received by a central routing facility that would send it on to the next hop on the way to its eventual destination. These were heavily used in the military to send messages and developed into "torn tape" routing systems where the tape would be torn from a receiving teletype and then send on to the next hop at a sending station. These were eventually turned into a way to automatically route teletype tapes (such as in Western Electric's Plan 55-A, see https://en.wikipedia.org/wiki/Plan_55-A) and were an inspiration to Leonard Kleinrock in his 1962 Ph.D. thesis, "Message Delay in Communication Nets with Storage." Kleinrock went on to work on packet switching and the Internet, along with other people.

well-known because of the broadcast of President Eisenhower's Christmas message in 1958.[3] More sophisticated communication satellites then followed, including the ECHO 1 in 1960. ECHO 1 was a simple satellite that reflected microwaves, allowing communications over long distances by passively bouncing microwaves off the reflective surface. ECHO 1 is often called the first global communications satellite. The Telstar 1 satellite in 1962 relied on an agreement between the United States, the United Kingdom, and France, and was able to relay both telephone calls and television signals and launched modern satellite communications.[4]

- *Computer networks.* In the late 1960s, as computers started to proliferate so did efforts to network them together. As early as 1963, people such as J.C.R. Licklider recognized the need to network together a set of time-sharing computers.[5] Those early efforts evolved into a number of operational, proprietary computer networks while ARPANET was being developed. Those proprietary networks included IBM's System Network Architecture (SNA) and Digital Equipment Corporation's DECnet as well as many others.

Figure 6.2 shows two of these areas: computer networking and telecommunications. Each of these areas originated a number of standards and protocols that were suited to their area. Over time, these two areas increased their overlap and much of the modern telecommunications infrastructure now uses protocols originally built for data networks (such as TCP/IP). This shift has taken decades to occur as factors such as a large installed base as well as data protocols were not initially designed to support voice traffic.

6.1.3 Wireless Networks

The development of wireless data networking is an example of one of these different camps in data networking. Wireless data networking developed primarily

3. You can listen to this message at: https://upload.wikimedia.org/wikipedia/commons/c/cf/SCORE_Audio.ogg. Interestingly, Eisenhower's message was carried via a physical audio tape onto the SCORE satellite.

4. See http://www.smecc.org/james_early___telstar.htm for James Early's recollections of the launch of Telstar 1. He includes interesting details such as the "Starfish Prime" test of a hydrogen bomb in the upper atmosphere the day before the launch of Telstar 1.

5. See J.C. R. Licklider's note published while he was with ARPA to "Members and Affiliates of the Intergalactic Computer Network" where he details a vision for a computer network that eventually evolved into ARPANET and the Internet. The note is reproduced here: http://www.kurzweilai.net/memorandum-for-members-and-affiliates-of-the-intergalactic-computer-network.

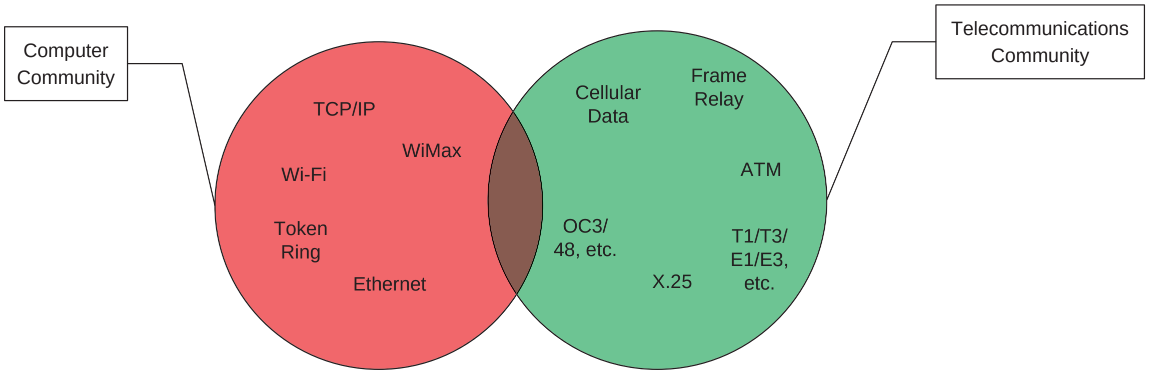


Figure 6.2 Two of the communications networks communities that contributed to data networking.

in three different industries:[6] telecommunications, TV and radio, and computer networking. Those networks have eventually all merged to be all data networks, largely using TCP/IP.

Telecommunications has been attempting to deploy wireless phone service for some time. In 1946, AT&T deployed a radio-based system in St. Louis, MO, to support wireless phone usage. This system was based on using a single radio frequency and as a result only one call could be made at a time. In 1976 in Chicago, IL, the first trial cellular network was deployed that made widespread use of wireless calling possible. This system used multiple frequencies in cells such that multiple radio frequencies were used in individual cells. These cells then formed a honeycomb pattern. The problem was then to switch moving users over seamlessly as they moved between these cells. This system (AT&T's Advanced Mobile Phone System, AMPS) allowed the possibility of supporting a large number of users simultaneously. Data was added to this network over time—initially, by encoding data by supporting cellular modems that encoded data, much like the wireline modems did for wired phone networks. Eventually, the underlying protocols and systems became more data aware and were able to support higher data transmission rates. For code division multiple access (CDMA)-based networks, the first such digital cellular technology standard was Interim Standard 95 (IS-95) developed by Qualcomm. Global System for Mobile Communications or Groupe Spécial Mobile (GSM) developed similar standards to support data. Generally, networks that did not transmit data were considered first-generation cellular networks (such as AMPS). Second-generation cellular networks support limited data such as

6. Satellite could be considered a fourth.

IS-95 on CDMA networks. As mobile data using cellular networks became more in demand, third-generation networks were envisioned and standards developed based on CDMA and GSM. For CDMA, the 1×EV-DO (Evolution-Data Optimized or Evolution-Data Only) was developed. For GSM, the Universal Mobile Telecommunications System (UMTS) was developed. These third-generation networks were developed to support relatively high data speeds in the 100s of kilobits per second and up to speeds in the megabits per second range. GSM- and CDMA-based[7] networks composed the vast majority of cellular telephone networks at this time (the mid-1990s), and part of the goal was to eventually make these networks interoperate. However, the technical challenges of developing these third-generation cellular networks delayed their deployment. As a result, several interim technologies were created to increase data transmission speeds and these were collectively called second-generation transitional (or 2.5G, 2.75G) networking technologies. These included CDMA's 1×RTT (One times Radio Transmission Technology), GSM's GPRS (General Packet Radio Service) and EDGE (Enhanced Data rates for GSM Evolution), among others.[8] Fourth-generation cellular networks are largely based on LTE (Long Term Evolution) Advanced. Fifth-generation cellular technologies are currently under research and development.

Television had been broadcasting analog signals for decades before the push to move to a packet-based format came with HDTV (High-definition Television) in the late 1980s and into the 1990s.[9] Even with lots of effort, it took years to develop and deploy HDTV. Part of this difficulty was the turmoil and change in US TV manufacturers as well as the need to displace spectrum so that HDTV could be broadcast concurrently with analog stations during the switchover. In addition to this effort, there were standards such as Multichannel Multipoint Distribution Service (MMDS) that were used to distribute cable services using facilities such as microwave.

For data networking, the University of Hawaii created a networked called AlohaNet in 1971 that formed the basis of some of the technology used in Ethernet. In 1985, network spectrum was released for unlicensed use that resulted in the

7. Other standards and networks existed such as Integrated Digital Enhanced Network (iDEN), developed by Motorola and largely used by Nextel Communications. Another example was iMode, developed in Japan, that supported not only packet-switched data but also some other services such as web access and email.

8. Another interesting example are data networks built on shortwave radio/ham radio. AX.25 is a standard built for amateur radio that uses X.25 type packet data.

9. Note that there were other, earlier analog uses of the term HDTV. Here the reference is to the United States Digital HDTV Grand Alliance that became official in 1993 and was comprised of AT&T Bell Labs, General Instruments, Phillips, Sarnoff, Thomson, Zenith, and MIT.

development of systems such as WaveLan at the National Cash Register unit of AT&T in The Netherlands in 1991. This eventually became the basis for the set of standards known as IEEE 802.11 and using the Wi-Fi label. The intent of WaveLan and Wi-Fi were to focus on serving a building or local area, rather than large areas as TV and telephone were attempting to do.

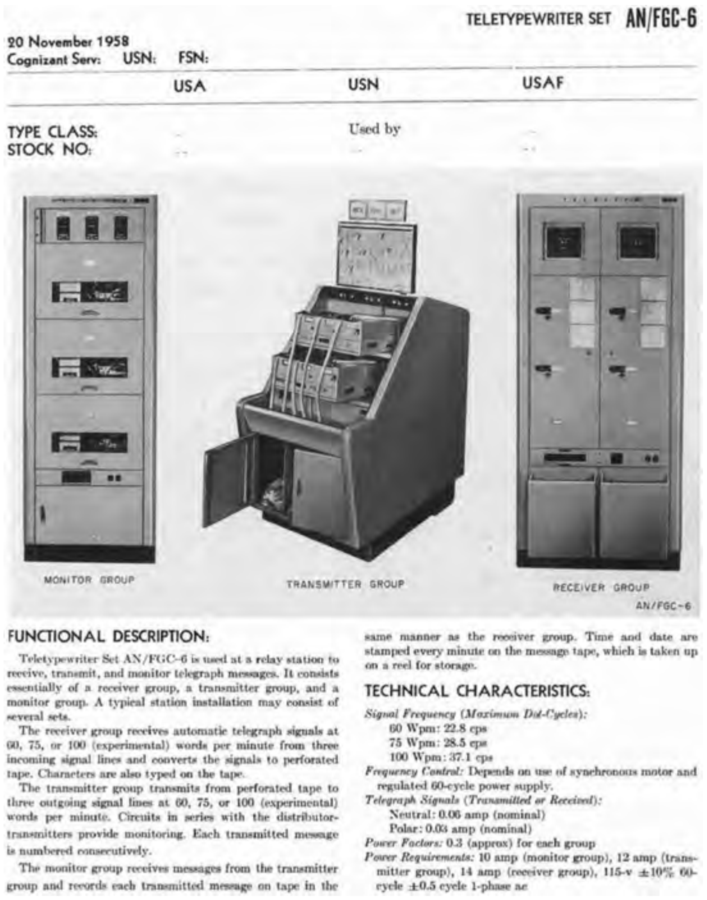
So, these three different types of wireless networking (telecommunications, TV, and computer data) developed separately over time. They developed their own techniques to solve problems, but eventually all began to merge in the form of all being computer data networks. The advantages of doing so enabled a flexibility of content delivered and provided the ability to build large and shared backbone networks. Even so, this merging of these different networks is still occurring as old networks get replaced and problems are solved to make the transition profitable for companies.

6.1.4 Some Networking Hardware

Networking has been very hardware-centric and many different types of networking hardware have been developed over the years. As a result, the software used to run networking protocols on top of that hardware was often hardware-specific. Networking hardware is varied and everchanging, but some hardware advances have shaped how future hardware and software were built. A few of those follow in this section. This section is not meant to be complete, but just to show a few interesting examples of networking hardware of which many readers may not be aware.

One such example includes torn-tape systems that were used to send and receive messages, particularly in military applications. These messages could be transmitted to a relay center that could then re-transmit the message to another hop until it finally reached its destination. These systems became quite sophisticated and eventually were able to route messages automatically without the need of an operator to determine the next hop. Figure 6.3 shows one such system that was used in the US military, along with some of the details of its operation. As noted earlier in this chapter, this automatic routing was directly referenced in the development of Leonard Kleinrock's contributions to packet switching methods. Figure 6.4 shows a torn tape relay operation in Guam on September 11, 1969.

The development of acoustic modulator/demodulators (better known as "acoustic modems") allowed data to be sent over the public phone system, using regular phones and regular phone lines. You can see in Figure 6.5 that such modems were designed to have a phone handset placed into the couplings. The modem could then use audible tones (as that's what the phone system supported) in order to encode and decode messages.



TELETYPEWRITER SET **AN/FGC-6**

20 November 1958
Cognizant Serv: USN: FSN:

	USA	USN	USAF
TYPE CLASS:	-	Used by	-
STOCK NO:	- -	-	- -

MONITOR GROUP
TRANSMITTER GROUP
RECEIVER GROUP
AN/FGC-6

FUNCTIONAL DESCRIPTION:

Teletypewriter Set AN/FGC-6 is used at a relay station to receive, transmit, and monitor telegraph messages. It consists essentially of a receiver group, a transmitter group, and a monitor group. A typical station installation may consist of several sets.

The receiver group receives automatic telegraph signals at 60, 75, or 100 (experimental) words per minute from three incoming signal lines and converts the signals to perforated tape. Characters are also typed on the tape.

The transmitter group transmits from perforated tape to three outgoing signal lines at 60, 75, or 100 (experimental) words per minute. Circuits in series with the distributor-transmitters provide monitoring. Each transmitted message is numbered consecutively.

The monitor group receives messages from the transmitter group and records each transmitted message on tape in the same manner as the receiver group. Time and date are stamped every minute on the message tape, which is taken up on a reel for storage.

TECHNICAL CHARACTERISTICS:

Signal Frequency (Maximum Dot-Cycles):
60 Wpm: 22.8 cps
75 Wpm: 28.5 cps
100 Wpm: 37.1 cps
Frequency Control: Depends on use of synchronous motor and regulated 60-cycle power supply.
Telegraph Signals (Transmitted or Received):
Neutral: 0.06 amp (nominal)
Polar: 0.03 amp (nominal)
Power Factors: 0.3 (approx) for each group
Power Requirements: 10 amp (monitor group), 12 amp (transmitter group), 14 amp (receiver group), 115-v ±10% 60-cycle ±0.5 cycle 1-phase ac

Figure 6.3 Teletypewriter torn tape messaging system. (Source: US Government/Teletype Corp., Nick England (www.navy-radio.com), DoD publication MIL-HTBK-161.)

Modems continued to increase their speed as well as to work over different underlying physical transport networks, such as cable, radio, and optical fiber.

6.1.5 Overview of Data Network History

Data networking between general purpose computers really did not begin in earnest until the mid- to late-1960s when there were enough computers to make networking worthwhile. One could argue that there were data networks of various kinds before this, such as the telephone network, emerging satellite networking, and other messaging such as teletypes. The earliest networks were direct computer-to-computer (what's referred to as "host-to-host" in Figure 6.6).

Figure 6.4 Teletypewriter torn tape relay center in operation in Guam, 1969. (Source: Courtesy of Nick England and http://www.navy-radio.com, US Navy Photo.)

Large computer companies began to develop their own proprietary networks in order to support their customers' ever-growing need to network the computers they own as well as to connect to other organizations' computers. Networks such as IBM's SNA and Digital Equipment Corporation's DECnet became predominant. These proprietary networks continued to be supported until the Internet became a viable replacement and TCP/IP became the de facto standard for data networking. With the popularity of the World Wide Web (WWW) in the mid-1990s, Internet services rapidly matured to allow the migration of legacy and proprietary networks to TCP/IP and the Internet.

One can also look at the evolution of network technology from the type of network it supports. That is, the technologies supporting *wide-area networks* (WANs) to interconnect computers across a large geographic distance evolved mostly

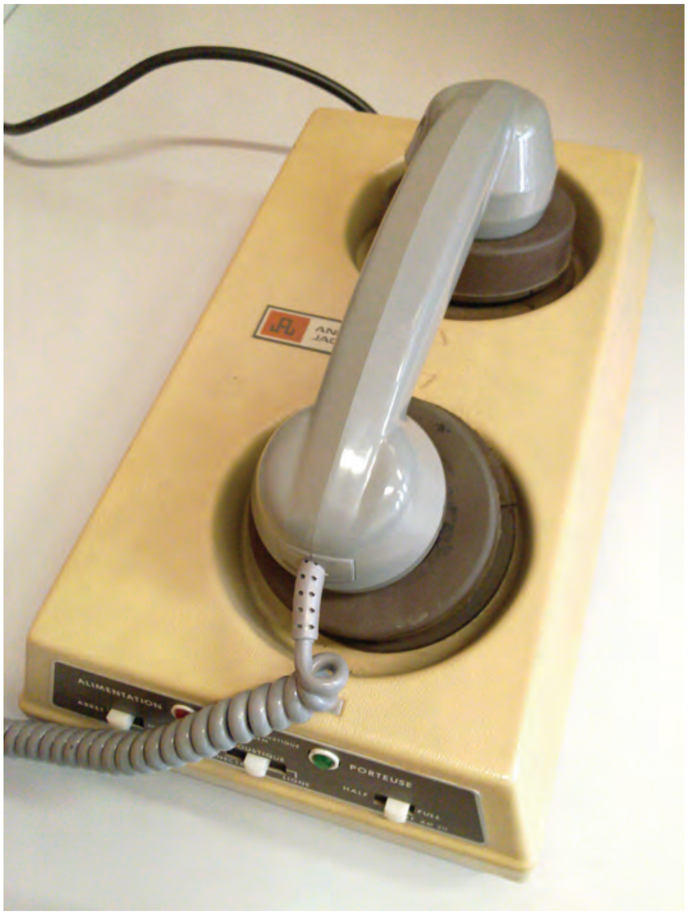


Figure 6.5 An AM211 CXR Anderson Jacobson acoustic modem (French version) and the phone handle (almost) plugged in. (Source: Olivier Berger, CC BY-SA 3.0 https://creativecommons.org/licenses/by-sa/3.0, via Wikimedia Commons and with permission of CXR Anderson Jacobson.)

from telecommunications technologies and began using the telephone network. Technologies supporting *local-area networks* (LANs) evolved to support in-building networking and had very different characteristics from telephony-based WAN technologies. Similarly, there are different technologies to support *metropolitan-area networks* (MANs) and *personal-area networks* (PANs). As an example, technologies such as Token Ring and Fiber Distributed Data Interface (FDDI) were developed as LAN protocols and used techniques that are difficult to use over long distances (i.e., tokens passing around a ring). Though the development of optical ring networks came from telecommunications to support metropolitan area networks, such as

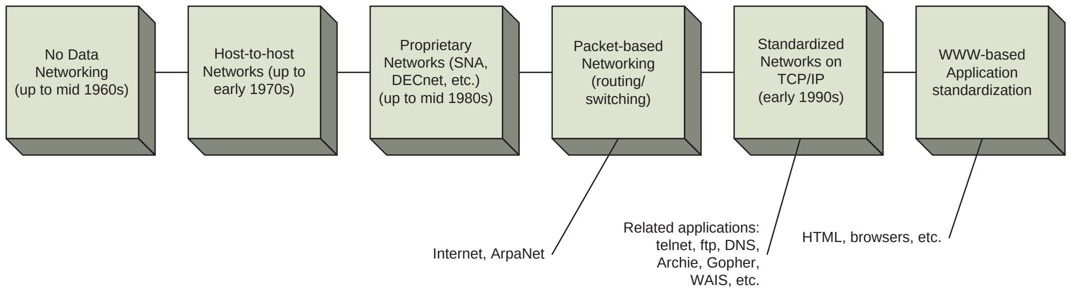

No Data Networking (up to mid 1960s)
Host-to-host Networks (up to early 1970s)
Proprietary Networks (SNA, DECnet, etc.) (up to mid 1980s)
Packet-based Networking (routing/ switching)
Standardized Networks on TCP/IP (early 1990s)
WWW-based Application standardization
Internet, ArpaNet
Related applications: telnet, ftp, DNS, Archie, Gopher, WAIS, etc.
HTML, browsers, etc.

Figure 6.6 High-level timeline of data networking stages.

Synchronous Optical Networking (SONET) and Synchronous Digital Hierarchy (SDH) rings.

6.1.6 Proprietary Networks

Proprietary computer data networks were developed and supported by large computer companies and other organizations in order to have a robust data network that could be supported. There was no viable open-standards–based alternative. So, IBM developed the Systems Network Architecture while DEC developed DECnet. Additionally, some companies developed and supported their own networking protocols such as the Bell Labs Network (BLN) developed by Bell Telephone Laboratories to connect Western Electric facilities and Bell Labs R&D sites.

6.1.6.1 One Example: IBM's SNA

SNA was introduced in 1974 as a propriety networking system centered on IBM mainframe environments. SNA went through many enhancements and lasted into the 2010s in order to support mostly legacy mainframes and their applications. SNA was developed as a propriety alternative to the International Standards Organization (ISO) Open Systems Interconnection (OSI) set of emerging protocols. As a result, it does not strictly adhere to the 7-layer stack proposed by ISO OSI and includes protocols such as the Synchronous Data Link Control Protocol, which was a packet-based protocol. SNA included the communication between IBM terminals, communications processors, and between systems. See bitsavers.org for a number of SNA documents, including http://www.bitsavers.org/pdf/ibm/sna/GA27-3102-0_SNA_General_Information_Jan75.pdf.

6.1.6.2 The Problem of Bridging or Connecting Disparate Networks

One of the issues that many faced in trying to deploy networks was the need to interconnect disparate networks. Large companies and organizations were buying a number of computers, each of which may only support its own, often proprietary network. As a result, companies would also deploy specialized computers as "gateways" in order to translate between the two networks. For example, it was common for a large company to support both DECnet and SNA and to buy another computer to serve as the gateway to connect those networks. An example of this need was the need for users on the DEC computers connected via DECnet to submit batch computing jobs on the IBM mainframes. So, there was often a method to send a job to be executed on the mainframe (often called Remote Job Entry, RJE) with the results being delivered back to the DECnet user. This author used an IBM Series/1 minicomputer for just this purpose as in Figure 6.7.

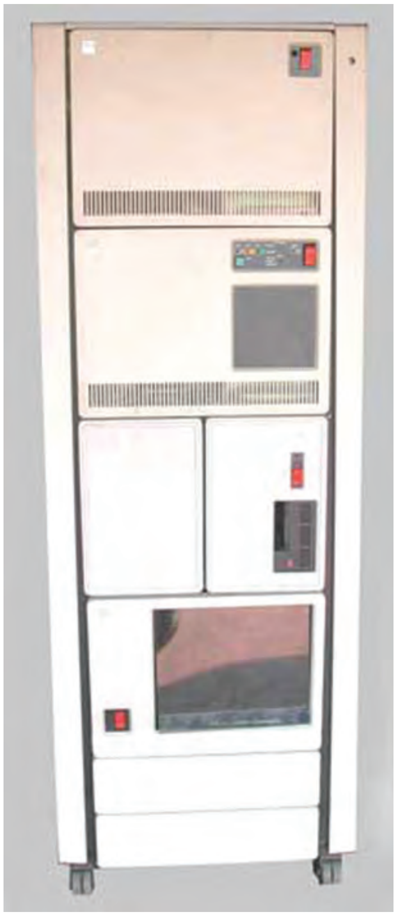


Figure 6.7 IBM Series/1 (4959), introduced in 1976 and usable as a network gateway between DECnet and SNA. (Source: Courtesy of Rhode Island Computer Museum, https://www.ricomputermuseum.org/.)

6.1.6.3 Front-end Processors

One component that was used to manage a computer's connection to a network (or networks) was to use a separate computer to manage that connection. This computer was often called a front-end processor (FEP) or a communications processor. Figure 6.8 shows the use of a DEC PDP-11 as a FEP to connect to the MERIT (Michigan Educational Research Information Triad) network at the University of Michigan's Computer Center in 1982. These FEPs would help to buffer traffic handle errors and manage the connection to the network, much like a router or switch would do today. MERIT was an early network proposed in 1968 and continued to be involved in the evolution of the network including participating in NSFNET.

6.1.7 Packet Networking and Internetworking

Two concepts were central to the evolution of the Internet and its associated protocols. Those concepts are *packet networking* and *internetworking*. Packet networking involves splitting data communication into separate units, called packets. These packets have a header and a termination. Packet networking was very different from what was provided by telephone companies, which was a dedicated circuit connection between two points–on the same model as a phone call. Various

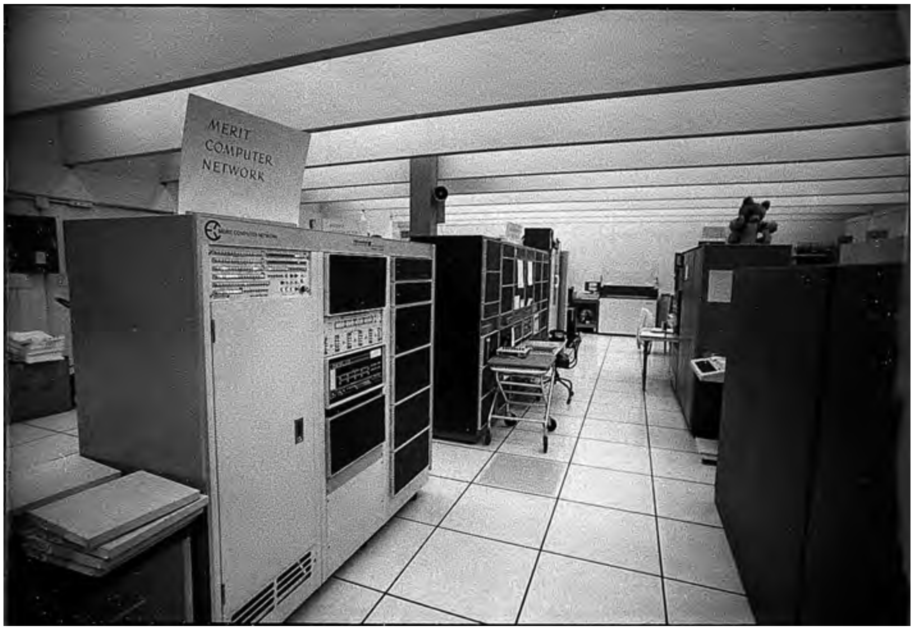


Figure 6.8 A front-end processor (MM-16) to communicate to a DEC PDP-11 for the MERIT network at University of Michigan in 1982. (Source: Courtesy of MERIT Network.)

inventors contributed to the development of packet-based networking including Baran at Bolt Beranek & Newman, Kleinrock at the University of California at Los Angeles, and Davies at the National Physical Laboratory in the United Kingdom.[10] The concept of *packet switching* allows the routing of those packets to be independent of each other and to use the best available route to deliver the packet. Internetworking is a concept developed to make it easier to connect disparate networks. The concept formed while trying to connect local area networks over a wide-area network. The term *catenet* was used by Pouzin in 1974.[11] This was eventually replaced by the term *internet* to represent the connection of multiple packet networks.

6.2 Networking Protocols

Networking has been driven by the availability of networking hardware and the development of protocols. Networking and networking software are heavily dependent on having standard ways of sending information between nodes on the network. These nodes are often built on different hardware and using different operating systems, which makes the use of well-defined protocols even more critical. Additionally, once a protocol is in place devices using that protocol often need to be supported through the transition to newer versions of that protocol or to new protocols. As a result, many of the protocols and software that implements those protocols also has a high amount of inertia. Newer devices often support older protocols in order to make that transition to new protocols practical.

Network protocols tend to be organized in a couple of ways. First there are vendor-specific protocols. Second there are open protocols such as those from ISO and TCP/IP. The various layers of these protocols are then related to their layers in the ISO 7-layer model (discussed below). For an example see this chart at http://i.imgur.com/MZHN3.gif or otherwise search for a "network protocol chart." Some charts will include other data protocols such as from telecommunications and others focus on more computer-centric standards.

6.2.1 OSI 7-Layer Reference Model

The ISO OSI 7-layer model[12] is used as a conceptual model for layering network protocols. In the late 1970s the model was developed as a combination of one

10. Note that there is controversy around the contributions of each of these inventors, though all were definitely involved and contributed to the development of the ideas and technologies used.

11. See "A Proposal for Interconnecting Packet Switching Networks," L. Pouzin, Proceedings of EUROCOMP, Brunel University, May 1974, pp. 1023-36.

12. Charles Bachman, mostly known for his work on database systems, is also credited (and noted in Bachman's oral history interview at http://ethw.org/Oral-History:Charles_Bachman) with adding a layer to the 6-layer IBM Systems Network Architecture (SNA) while creating the

from the International Telegraph and Telephone Consultative Committee (CCITT, which eventually became the International Telecommunications Union, ITU) and ISO. These two models were merged in 1983 to form the Basic Reference model for OSI. This model has seven layers with the idea being that each higher layer uses only the layer below and serves only the layer above it. Additionally, different protocols can be used at each layer if different functionality is needed at that layer. The layers starting at the lowest, hardware layer are, in order:

1. Physical layer: transmission and reception of raw bits
2. Data link: reliable transmission of data between two nodes
3. Network: addressing and routing of data between a network of nodes
4. Transport: reliable transmission of a data across network
5. Session: managing communication sessions
6. Presentation: translation of data between the network and the application
7. Application: high-level networking applications such as resource sharing

Note that few networks strictly adhere to this model but it is used to show where network functionality is implemented. Most networking hierarchies (or "stacks") roughly follow this model, though even TCP/IP does not strictly follow this model. A new protocol will often be mapped to these layers, as in network protocol reference charts.

6.3 Getting to TCP/IP

The process of getting to TCP/IP from a number of proprietary and other networks was relatively complicated. It involved the transition to packet-based networking as well as the displacement of a number of older network protocols. It also involved the creation of a number of research networks including ARPANET, along with two in Europe: France's CYCLADES and the National Physical Laboratory Network in the United Kingdom. ARPANET began in 1969 and was well-funded by the US federal government along with a number of other ARPA projects such as Stanford Research Institute's Augmentation Research Center Lab (Englebart), Shakey the Robot (Nilsson), and other artificial intelligence research. There was also a lot of trepidation in moving from what had become relatively reliable networks to using

Honeywell Distributed Network Architecture that modeled the same layers as the OSI 7-layer reference model.

TCP/IP and ARPANET. A good example of this is a quote from Robert Metcalfe from the 1972 International Conference on Computer Communications:[13]

> They gave me the job of escorting ten AT&T vice-presidents around. So I was demoing the system, and for the only time in that whole show, the TIP crashed. The only time. It went down for about ten or twenty seconds. It finally came back up again. We reestablished connection and it never went down again. But this was a very enlightening moment for me because when I looked up, you know, they were happy that it crashed. They made no point of hiding their joy. Because this confirmed for them that circuit switching was better and more reliable than packet switching, which was flaky and would never work. And I had been working on this for two or three years, and it really pissed me off.

It wasn't until the mid- to late-1980s that TCP/IP began to replace legacy network protocols and then not until the 1990s and 2000s that TCP/IP began to replace the legacy protocols in the telecommunications network. That transition is still not complete and many legacy protocols still run. As an example, there still exist many wired phones that are not using TCP/IP (i.e., Voice over IP, VoIP), but the high-speed backbone telecommunications networks are largely TCP/IP.

6.3.1 UUCPNET

Besides the Internet and proprietary networks, there were also other data networks. An example is UUCPNET, which was the network of computers connected using UNIX-to-UNIX Copy (uucp). UUCPNET was used to relay email beginning in 1978 and was released as part of the UNIX® operating system by Bell Labs in 1979. So, email and news could be exchanged between hosts on UUCPNET. Email on UUCPNET used a "bang path" notation. Such as "...ihnp4!ihlpa!tracy" would give a routing path for forwarding email to my account on the computer called "ihlpa" and routing through computer "ihnp4." So, "ihnp4" had to be well known enough for other systems to know how to reach it. Systems were set up to send email from UUCPNET to the Internet using Internet-aware gateways that were also connected to UUCPNET, such as at some universities.

6.3.2 ARPANET

As noted above, the ARPANET started being developed in 1969 with the awarding of the contract to Bolt Beranek & Newman. The evolution of the ARPANET to what

13. See http://www.historyofcomputercommunications.info/Book/BookIndex.html, chapter 4, which gives more details of the demo and the conference.

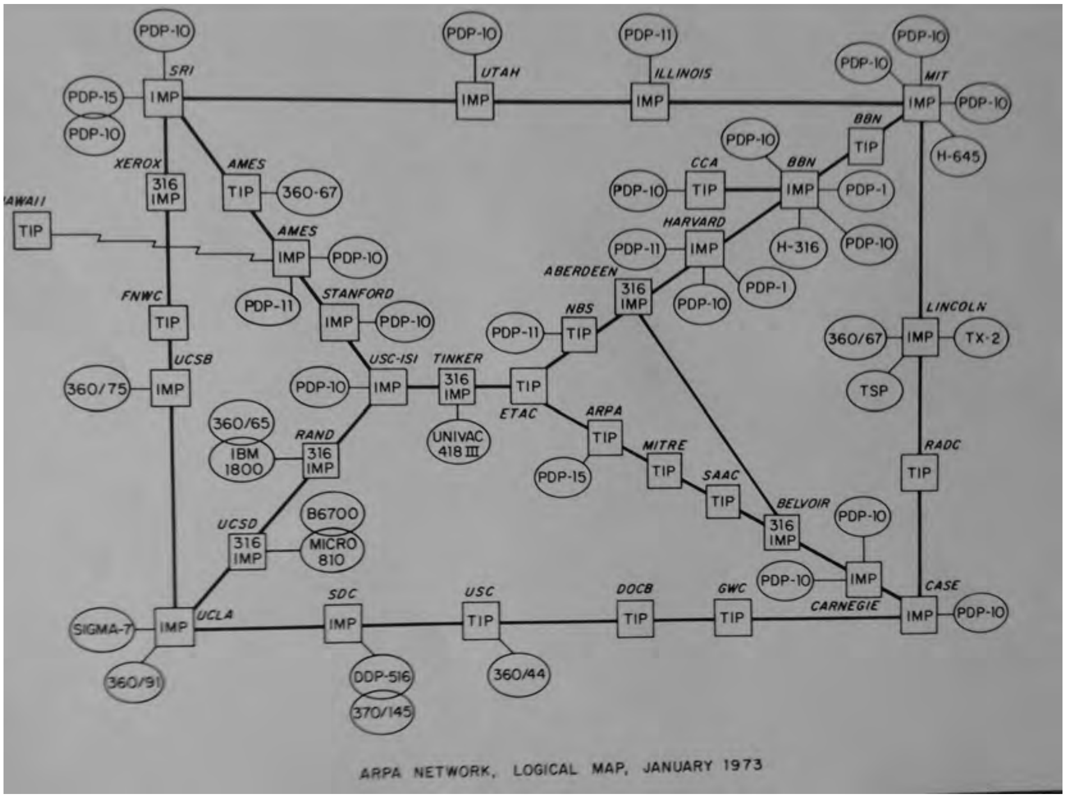


Figure 6.9 The topological map for the ARPANET in 1973. (Source: Image courtesy of the Charles Babbage Institute Archives, University of Minnesota Libraries, Minneapolis. (John Day Papers).)

has since become the global Internet is well-documented in works such as Abbate [2000]. See Figure 6.9 for a diagram of the ARPANET nodes and their connections in 1973. Of interest in this text is this network also gives a good idea of those doing computer science research in 1973 and the centers of activity.

6.4 Network Software and Applications

Networking protocols, while necessary, are only a part of the story. Data networking became much more pervasive and useful as applications were built using underlying network protocols and developing their own application-level protocols and applications. This section describes some of those applications, how broadly they were used, and their impact on later applications. Network applications software added to the value of connecting to the network for users and organizations. As a result, the development of additional network applications helped to increase the use of computers as well as the network.

6.4.1 Electronic Bulletin Board Systems

Electronic bulletin board systems[14] (BBS) were first created in the 1970s as a way for hobbyists to share information via dial-up systems. These began as local systems that could be connected to via dial-up modems in a particular region, but over time became networked and were able to share messages between users. In the United States, these were placed in regions where a BBS's users could place a free local call to the BBS. This made it cheap for its users to use frequently as well as BBS operators could charge for the service. This was well before public Internet access was available, which was why they were able to attract enough users to actually have profitable businesses. These systems were largely limited to text and using slow modems such as 110 baud (running at 110 bps). Demand was such that even public, pay-per-use terminals were put in some communities such as "Community Memory," which was a terminal connected to a BBS put in several locations in Berkeley, CA, in 1973 (see Figure 6.10).

Networks for BBSs, such as FidoNet, became a way for content to be shared around the world between BBSs.

6.4.2 Email

Email has long been a tool used to send messages between users on a computer system. The first known large scale message system is the AUTODIN network that was used by the US General Services Administration (GSA) on 2,500 terminals in 1962. MIT's CTSS operating system also provided a "MAIL" system for inter-user messaging in 1965. Sending email messages over a data network was first done in 1971 by Ray Tomlinson and documented on BBN's site at https://ds.bbn.com/tomlinso/ray/firstemailframe.html. After this, other network email systems were developed, including one for UUCPNET as described above in 1978. The PLATO system (in PLATO IV) developed a note-passing system in 1974.[15]

14. See https://archive.org/details/BBS.The.Documentary for an excellent documentary on BBS from their birth to their demise with the consumer availability of the Internet.

15. The PLATO system was an early system design for online education and originated in the early 1960s at the University of Illinois. By the 1970s it could support over a thousand simultaneous users. See http://platohistory.org/.

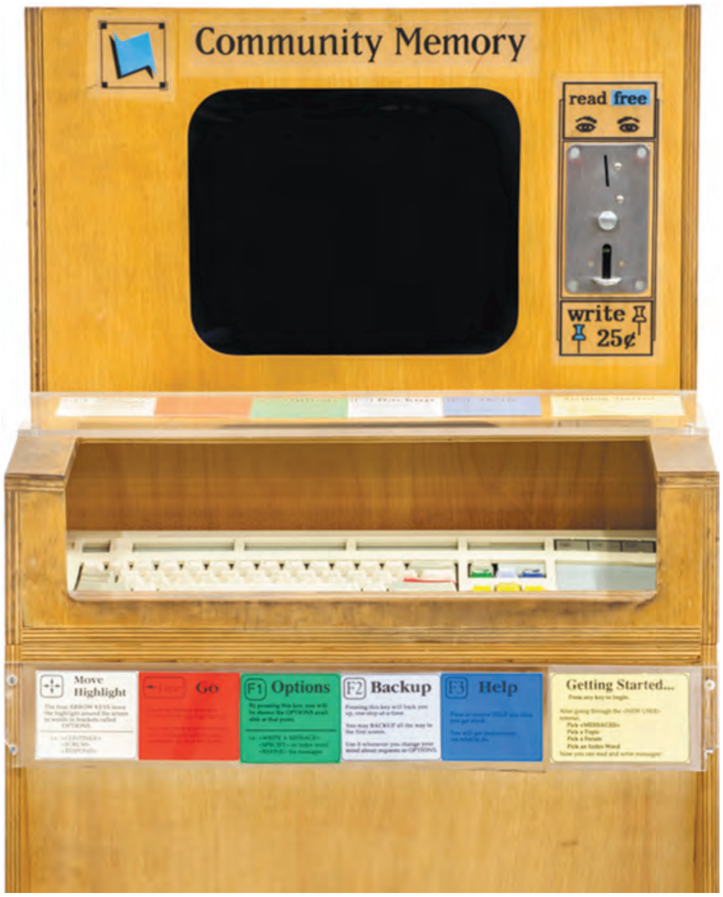


Figure 6.10 Community memory, public terminal for BBS in Berkeley, CA, 1975. (Source: Mark Richards. Courtesy of the Computer History Museum.)

6.4.3 The WELL

The WELL (http://well.com) began in 1985 as a dial-up BBS and stands for Whole Earth 'Lectronic Link, in reference to the Whole Earth Catalog. It is a paid membership model and still operating. It's an interesting example of community building on the Internet. Stewart Brand (one of the founders) and his impact on Cyberculture is described in Turner [2008] and inspired Krol [1992].

6.4.4 Usenet and NetNews

Usenet developed as a news and communication service on top of UUCPNET. The intent was to be able to develop subscribe-and-read content that was of interest to users. Much of the content was technical in nature but also included lots of other topics where users could share opinions and questions. News was divided into a hierarchical set of categories such as comp.ai, which represented the artificial intelligence news group within the computing news area. Additional categories could be added that were more specific such as comp.ai.prolog for sharing interests about the Prolog programming language. NetNews established many of the protocols and traditions (like emoticons) for online communication. In many ways, NetNews was similar to a BBS in the way that information was relayed between sites and users.

6.4.5 FTP Archives

As the Internet grew, one of the significant stores of content became File Transfer Protocol (FTP) archives. These archives held files that could be shared across the Internet and were vital stores of information before other methods such as the WWW and web search engines became popular. For example, users could use the FTP protocol to connect to an archive such as ftp.wustl.edu[16] to download free computer games for their personal computer. However, they did have to know that the archive existed, how to connect to it, what they were looking for, and where it was on the archive. In the late 1980s and pre-WWW this was how many files were shared and proliferated. Most of these sites allowed anonymous access to the archives, sometimes requesting your email address as the password, but were otherwise open to anyone on the Internet. The Internet Archive holds copies of many FTP sites at https://archive.org/details/ftpsites.

6.4.6 Gopher, Archie, and Veronica

The Gopher protocol (appropriately developed at the University of Minnesota—the gopher is their mascot) was designed to search and retrieve documents over the Internet. It was a menu-based system, first developed in 1991. It was text-based but provided many of the same sorts of abilities that the WWW would provide a couple of years later in organizing content. Gopher used the Wide Area Information Server (WAIS) as its base for searching documents. Services called *Archie* and *Veronica* were developed as search engines for Gopher. One could easily argue that Gopher could have easily become the basis of the WWW rather than Berners-Lee's Hypertext Transfer Protocol (HTTP) if it had included graphics earlier and been more freely disseminated.

16. Referred to the ftp archive at Washington University in St. Louis, a large archive that no longer exists.

6.4.7 World Wide Web

In 1989, when Tim Berners-Lee was creating the concept of the WWW, it was in the context of supporting the sharing of physics research done at CERN and around the world using the Internet. Additionally, this was also the time that personal computers were becoming commonplace, Internet service providers were becoming common, and telephone modems were reaching speeds where simple graphics could be downloaded. As a result, when more publicly available versions of web browsers became available in 1993 (i.e., in particular, Mosaic from the National Center for Supercomputing Applications, NCSA, at the University of Illinois at Urbana-Champaign), it took off like wildfire. The Mosaic web browser's simpler interface made it possible for users to easily access web pages containing graphics. So, while there were earlier services providing similar functionality (such as Gopher), the WWW took off with being able to provide interesting content (especially graphics) with an interface that was simpler and accessible via the data rates supported by ISPs over dial-up modems.[17] As users clambered onto the Internet, websites proliferated and the demand for faster data access speeds increased the incentive to produce faster modems. These modems quickly reached near the Shannon Limit for data channel capacity.[18]

6.4.8 Network Operating Systems

In order to make early networks more useful, a number of so-called network operating systems were developed to make it easier to perform higher level functions over a local area network. This included features such as file sharing, print spooling, and email. Many of these were widely used such as Banyan VINES and Novell NetWare. Apple's AppleTalk network included a number of features to make the AppleTalk network easier to use that were also in the same spirit.

Here we will explore Banyan VINES (Virtual Integrated Network Service) as an example of this class of network software. Banyan VINES was developed to be used specifically with AT&T's UNIX System V operating system by Banyan Systems. It was first introduced in 1984 and used a low-level protocol called VIP (VINES Internetwork Protocol). VINES included a set of routing algorithms to help route and control network traffic. It provided a number of network features that would become standard Internet features, such as address resolution and a control protocol for error messages.

17. Data speeds for modems in 1992 were getting better as the quality of phone lines improved as well as modems. Speeds of 28.8Kb/s were becoming common in 1994.

18. The Shannon Limit defines how many bits can be transmitted over a particular bandwidth channel. See https://www.youtube.com/watch?v=Wq1-Iq9Vm28 for a video about the Shannon Limit.

VINES, as most network operating systems did, provided a number of high-level services over the network. These included file services, print services, and a directory service.

As the Internet matured and the WWW drove usage to the Internet, the need for the services provided by Banyan VINES (as well as other network operating systems) declined as the Internet was able to provide the same kind of services in a non-proprietary environment. Similar Internet-based services (such as email) became the de facto standard for these sorts of services.

6.5 Case Study: Minitel

The Minitel system was a very successful videotex online service that was first deployed in 1980 (see Mailland and Driscoll [2017]). The service was deployed to millions of French homes and provided services such as telephone directory searching, stock price checking, email inbox, chat capabilities, and the ability to make reservations and purchases. The system was officially retired by France Telecom in 2012. See this French video advertisement of Minitel in 1988: https://youtu.be/Xl2MFGI2i40.

The Minitel was built on a packet data network called TRANSPAC that was based on the X.25 protocol. The TRANSPAC network was developed by the French PTT (Post, Telephone and Telegraph) and influenced by the earlier French PTT RCP experimental network. RCP was virtual circuit-based and influenced the X.25 standard. The French CYCLADES research network was based on datagram packets, much like the ARPANET. CYCLADES was demonstrated in 1973. TRANSPAC started operation in 1978 with Minitel being a service added a couple of years later.

The model was to lend the Minitel terminal (called "Teletel") without charge to telephone subscribers and by not providing a printed telephone directory, thereby saving the cost of printing it. The system required use of a modem and was text based but became very popular. Many other countries tried to replicate France's success but were largely unsuccessful for a variety of reasons, including competing services such as BBS and the emergence of dial-up Internet service providers. Some argue that the success of the Minitel system slowed the deployment of the WWW in France, but one could also argue that it prepared a large percentage of the French population for using online data services.

6.6 NCSA httpd and Apache Web Server

The NCSA at the University of Illinois built a freely available web server shortly after Tim Berners-Lee's server became available. The NCSA httpd (Hypertext Transfer Protocol Daemon) was initially written by Rob McCool at NCSA and quickly became popular and widely deployed in 1993 and 1994. As a result, it was instrumental in making practical web server software available. The NCSA took up the project and continued to work on it up until 1995, when the Apache Foundation took over the project as the Apache Web Server, which quickly replaced the NCSA versions of the system as the Apache Foundation continued to add desirable features to the server that gave web servers additional features and better performance. The 1.51 version of the NCSA httpd can be found at GitHub.[19] The decode_request function from the *http_request.c* shows how an HTTP request is decoded.

HTTPd is described by the UNIX manual page from October 1995 as:

> is the NCSA HTTPd (HyperText Transfer Protocol daemon) server. The server may be invoked by the Internet daemon inetd(1M) each time a connection to the HTTP service is made, or alternatively it may run as a daemon. As a result, each connection to the server would run as a separate httpd process. A web server with lots of requests would result in many httpd processes being spawned.

The overall source code, written in C, often appears to be hastily written and full of *ifdef*'s to support various architectures and tools. This bit of code (see Listing 6.1) also uses *strcpy* and *strcmp* (rather than the more secure *strncpy* and *strncmp*) which was common at the time.

```c
/* decode_request()
 *   given an HTTP request line as a string, decodes it into
 *   METHOD URL?ARGS PROTOCAL
 *   Then calls get_http_headers() to get the rfc822 style headers
 */
void decode_request(per_request *reqInfo, char *request)
{
    char *protocal;
    char *method, *url;
    char *chp;

    /* extract the method */
    method = strtok (request, "\t ");
    if (method) {
        if ((reqInfo->method = MapMethod (method)) == M_INVALID)
            die(reqInfo,SC_BAD_REQUEST,"Invalid or unsupported method.");
    }
    else
        die(reqInfo,SC_BAD_REQUEST,"Invalid or unsupported method.");
```

19. See https://github.com/TooDumbForAName/ncsa-httpd.

```

    /* extract the URL, and args if present */
    url = strtok (NULL, "\t\r ");
    if (!url) die(reqInfo,SC_BAD_REQUEST,"Incomplete request.");
    if (url && (chp = strchr (url, '?'))) {
        *chp++ = '\0';
        strcpy (reqInfo->args, chp);
    }
    strcpy (reqInfo->url, url);

    protocal = strtok (NULL, "\r");

    if(!protocal) {
        reqInfo->http_version = P_HTTP_0_9;
    }
    else {
        /* On an HTTP/1.0 or HTTP/1.1 request, respond with 1.0 */
        if (!strcmp(protocal,protocals[P_HTTP_1_0]))
            reqInfo->http_version = P_HTTP_1_0;
        else if (!strcmp(protocal,protocals[P_HTTP_1_1]))
            reqInfo->http_version = P_HTTP_1_0;
        else if (!strcasecmp(protocal,protocals[P_SHTTP_1_1]))
            reqInfo->http_version = P_SHTTP_1_1;
        else if (!strcasecmp(protocal,protocals[P_SHTTP_1_2]))
            reqInfo->http_version = P_SHTTP_1_2;
        else reqInfo->http_version = P_OTHER;

        /* dummy call to eat LF at end of protocal */
        strtok (NULL, "\n");
        get_http_headers(reqInfo);
    }
```

Listing 6.1 NCSA httpd decode_request function from version 1.5.1 from 1995.

With the Apache Web Server project, a number of organizations contributed to the code and re-organized it to be more maintainable as well as contributing substantial effort to making sure the web server was useful. The following code from the 1995 version of the Apache Web Server 1.0.0 shows the substantial re-organization of the code and use of a better structure to accomplish much the same purpose as the snip-it of code given above for the NCSA httpd.[20] Note that it does use parsing techniques rather than just comparing strings (some of which is contained in other parts of the Apache Web Server code).

20. This code is still available through the Apache Web Server project on github.com at https://github.com/apache/httpd/tree/1.3/APACHE_1_0_0/src.

```
request_rec *read_request (conn_rec *conn)
{
    request_rec *r = (request_rec *)pcalloc (conn->pool, sizeof(
    request_rec));

    r->connection = conn;
    r->server = conn->server;
    r->pool = conn->pool;

    r->headers_in = make_table (r->pool, 50);
    r->subprocess_env = make_table (r->pool, 50);
    r->headers_out = make_table (r->pool, 5);
    r->err_headers_out = make_table (r->pool, 5);
    r->notes = make_table (r->pool, 5);

    r->request_config = create_request_config (r->pool);
    r->per_dir_config = r->server->lookup_defaults; /* For now. */

    r->bytes_sent = -1;

    r->status = 200;            /* Until further notice.
                                 * Only changed by die(), or (bletch!)
                                 * scan_script_header...
                                 */

    /* Get the request... */

    hard_timeout ("read", r);
    if (!read_request_line (r)) return NULL;
    if (!r->assbackwards) get_mime_headers(r);
    kill_timeout (r);

    if(!strcmp(r->method, "HEAD")) {
        r->header_only=1;
        r->method_number = M_GET;
    }
    else if(!strcmp(r->method, "GET"))
        r->method_number = M_GET;
    else if(!strcmp(r->method,"POST"))
        r->method_number = M_POST;
    else if(!strcmp(r->method,"PUT"))
        r->method_number = M_PUT;
    else if(!strcmp(r->method,"DELETE"))
        r->method_number = M_DELETE;
    else
```

```
            r->method_number = M_INVALID; /* Will eventually croak. */

        return r;
}
```

Listing 6.2 Apache Web Server httpd read_request function from file http_protocol.c, version 1.0.0.

The Apache Web Server project continued to evolve the web server, becoming an exemplar open-source project that eventually produced a stable web server that was heavily used for over a decade and continues to be used today. The open-source nature of the project is well described by Kelty [2008] and Raymond [1999], where the distributed nature of the project was both a blessing and a curse, particularly when the Apache Web Server needed to be re-architected to be able to scale to support a large number of simultaneous connections.

The NCSA httpd and the Apache Web Server were an important project to help make the WWW a reality. Besides their core role in providing key web components, the Apache Web Server project became an example for how organizations could collaborate on open-source software projects. In addition, by looking at the source code for these projects, one can see that it was often done quickly and needed to support a wide variety of different computer operating systems and underlying instruction sets. While this software went through many modifications, it still had a number of design assumptions that pervaded through many versions of the Apache Web Server, which continues to power a large number of web servers. As noted in many places in the source code, there are many places where multiple versions of the HTTP protocol need to be supported concurrently and often the code to support those old versions of protocols remains in these systems even after active use of the protocol disappears, which can leave additional security vulnerabilities.

6.7 Networking Influences

Many factors have influenced the change of data networking over time, as exhibited in Figure 6.11. As computers proliferated, it became clear that data networking between these computers was beneficial. This “network effect” became encoded as *Metcalfe’s Law,* named after Robert Metcalfe, an inventor of Ethernet. While cast in terms of the number of connected devices at the time, this concept has also been applied to the possible communications between users of a network. The term *network effect* is also used in business to indicate the value of a project or service increasing with its number of users. Metcalfe’s Law states that the number of possible connections between devices on the network increases proportionally with the square of the number of devices. So, as the number of computers and computer users grew, so did the possible value of establishing a network. Much of the

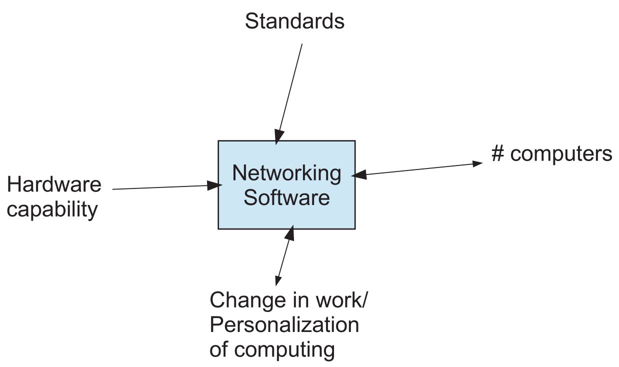


Figure 6.11 Some influences that have affected the direction of data networking.

Internet's value is because of the high number of connected computers, devices, and people.

Other factors that have influenced the use of data networking include factors such as the personalization of computing. These increased the number of computers and people quickly realized that their value would be increased if applications and data could be shared with other computers and users of those computers. Symbiotically, people began to use and depend on services over the network as more network services were built. Services such as email, file-sharing, and the ability to perform transactions increased the value of networking to computer users. These network services in turn increased the value of connecting to the network, as well as stimulated the purchase of personal computers. Over time, these services have changed the way people work and helped support team work across the globe.

Networking protocols and related standards were critical in achieving and increasing the use of networks. Without a standard way of connecting, each connection required customized programming and standards began to make it possible to connect a wide variety of computers to a single network. Standards at a minimum made the building of large data networks quicker than would have occurred without standards.

6.8 Lessons Learned from Networking Software

Networking software has different characteristics from other areas of software that yield different lessons from other types of software. In order to be most effective and useful, networking software includes more communicating parties and benefits from the *network effect* (as you might expect). The network effect in essence says that the value of a product or service increases according to the number of parties (endpoints, nodes, or users) using it. So, network software becomes very entrenched as a result of being used by many parties and cannot be removed until all those parties quit using it. This not only keeps existing protocols and other network software in place for long periods of time, it often results in newer network software continuing to support the older protocols to ease the transition to newer protocols, standards, and techniques. Furthermore, even when use of a particular protocol or standard is effectively zero, it's often not worth removing that support for the older protocol or standard. This can result in these old protocols, standards, and techniques being extant in modern software and increasing the attack surface for possible security attacks. Some other lessons from networking software include:

- *Timing and the speed of light are important.*

  The variance in networks' speeds has made it clear that network protocols need to be able to support this wide variance in speeds and the ability to throttle and cache network traffic has become critical to making distributed systems function properly. Additionally, as networks expanded around the world, the fundamental limit of the speed of light has been a limiting factor of performance. While the speed of light allows light to travel (approximately) 7.5 times around the world in 1 second, that still means that it takes over 100ms to get to the other side of the world, which becomes a limiting factor for many applications. In practice, that limitation is more than 100ms due to network data being translated, passing through devices such as routers and switches, as well being translated between light and electricity.

- *Errors are unpreventable.*

  While errors occur in all types of software, with networking software they cannot be completely eliminated as data in transit can be affected by all kinds of interference, errors in devices, failure of transmission lines, and many other types of errors. As a result, networking software has internalized the ability to deal with errors and to still provide reliable transmission (as in the Transmission Control Protocol, TCP, among many other protocols).

- *Protocols are critical to coordination.*

  Without network protocols that define what messages are sent and what are valid responses, data networking would be very difficult. This notion of using protocols to define communication has been used by other areas of software as well, such as security protocols and higher-level application protocols.

6.9 Exercises and Projects

6.9.1 Exercises

1. The TCP and IP protocols do not map directly onto the ISO OSI 7-layer Reference model, published in 1984 as the ISO 7498 standard. Explain how TCP and

IP don't map directly to this model. Why did it not matter much for TCP/IP to map to the ISO 7498 standard in 1984? Would a direct mapping have eased the later convergence of telecom and data networking?

2. Investigate AX.25 (amateur radio packet data standard). Is it still used by amateur radio operators? How is it different from the X.25 protocol on which it is based? A good place to start is Brian Kantor's paper, "A Brief Look at Internet Networking over Amateur Radio," July 2011, http://www.ampr.org/wp-content/uploads/brieflook.pdf.

3. Find an example Minitel application. How did that application work? Was it a good fit for a Minitel application or were there better options at the time the application was created?

4. Many cities have attempted to install citywide Wi-Fi networks but most of these networks have not survived. Find a city that has attempted to install such a citywide network that failed. Explain why it failed in this case and whether citywide Wi-Fi will likely be tried again.

5. Investigate the features of Gopher's search engines of Archie, Jughead, and Veronica. How do they compare to the features of early WWW search engines such as Lycos and WebCrawler? Did the Gopher search engines influence web search engine approaches and techniques?

6. Another network operating system was Novell NetWare. Compare the features in Novell NetWare with those of Banyan VINES. Also detail the changes in Novell NetWare and why it survived longer than Banyan VINES.

7. Investigate the use of an argument for a *network effect* in Theodore Vale's arguments for building a nationwide telephone network for AT&T in the early 1900s. Find an example where AT&T made the argument that the more users on the network, the more valuable it would be. Did this take the numerical form (proportional to the square of users) that Metcalfe's Law took?

8. A surviving remnant of network operating systems are directory services such as Active Directory by Microsoft and Novell's NetWare Directory Services. Examine the roots of directory services in network operating systems and explain why that service has survived while others (such as email) provided by network operating systems have not.

9. Investigate the Wireless Application Protocol (WAP) and how it was used as a base for data applications over a cellular network. Why was another protocol necessary at the time? What has effectively replaced WAP? Does a modern iPhone or Android phone still support WAP?

10. Networking protocols are often difficult to completely eliminate due to the nature of their usage (connecting many devices to one another). Take a look at the Asynchronous Transfer Mode (ATM) protocol and examine its past and current usage. Is ATM still in use? Where? Why?
11. Find out more about the fixed wireless system called Motorola Canopy that was developed to provide wireless Internet service and competes with WiMAX and cellular data. Why was this alternative developed? Is it still in use?
12. The Digital Enhanced Cordless Telecommunications (DECT) standard was developed to create cordless telephone systems in homes and businesses. DECT was a widely used standard in Europe, but in the United States other standards such as the 900MHz and the 2.5GHz standards were more prevalent for cordless telephones. With DECT, DECT Packet Radio Services (DPRS) were also developed. Find out why DPRS was not more successful and speculate (or better yet, find evidence) as to why it failed.
13. Find the first Request for Comment (RFC) for FTP and compare that to the current FTP specification. What changed? What has remained the same?
14. The GOPHER protocol was popular during its time and is an early method for organizing and finding information on the Internet. Examine why the GOPHER protocol did not become the basis for the WWW.
15. It seems highly unlikely that a new camp would form in data networking such as those we discussed: computer, telecommunications, radio, and TV. Recall that a new "camp" developed its own technologies, perhaps in distinct ways from the other camps and based on different fundamental requirements.

    Do you agree that it's highly unlikely? Explain why or why not.
16. The current drive to replace IPv4 with IPv6 has been driven by the need for more IP addresses. Do you expect that there will be a new set of protocols that replace TCP and IPv6 at some point in the future? If so, explain what sort of drivers there would be to such a change and what constraints there would be on its deployment.

    If not, explain the resilience of IP and TCP to change and how it can withstand drastic changes in other technologies.

6.9.2 Projects

1. Investigate the rise and fall of Integrated Services Digital Network (ISDN). Write a paper that discusses how ISDN originated, how it was used, why it

became popular, and what has displaced it. Describe the plans for evolving ISDN to higher bandwidth services and why those didn't happen as planned. See Stallings [1992] for background information on ISDN.

2. Review John Day's 2012 talk[21] where he argues about some of the design flaws of the TCP/IP protocols. In that talk he discusses some protocol changes that would help the Internet perform better and scale better. Can any of those changes ever happen (he argues "no") anytime soon? What would have to happen to replace the base protocols of the Internet? Will that happen?
3. Find the code for a Wireless Application Protocol (WAP) application and explain how it worked. Compare that code to a modern application's code built for a similar use. What has remained the same? What has changed?
4. The consolidation of cellular wireless networks took many years as well as being a difficult technical challenge. Investigate and write a report detailing how the change from second-generation to third-generation to fourth-generation cellular networks was achieved. Is the migration to fifth-generation wireless progressing more easily? Point out the challenges and roadblocks that occurred (and are occurring) both technically, globally, and business-wise.
5. Investigate Japan's iMode cellular data service and how it was different from other options deployed elsewhere. Write a paper detailing its origination and demise. Why did Japan develop their own standard and service? How did its existence make it more difficult for Japan to transition to third-generation cellular networks?
6. Write a paper about the history of Amateur Radio Digital Communications (AMPR), which manages AMPRNet, a TCP/IP network for amateur radio. Investigate how it started, how much and what sort of traffic it supports, and its current state.
7. Investigate the various contributions of Baran at Bolt Beranek & Newman, Kleinrock at the University of California at Los Angeles, and Davies at the National Physical Laboratory in the United Kingdom to packet switching. Write a paper that details each of their contributions to packet switching and develop an argument as to who is most responsible for the idea and development of packet switching. Include why there is controversy about this topic

21. "How in the Heck do you Lose a Layer," Future Network Architectures Workshop, University of Kaiserslautern, Germany, 2012, available at http://rina.tssg.org/docs/JohnDay-LostLayer120306.pdf.

and why there is a lack of total clarity on who is most responsible for the development of packet switching.

8. Investigate the sharing of ideas between IBM's SNA and the protocols making up the Internet. Are there any protocols from SNA that influenced or were used in the Internet? Document the relationship of SNA to the set of Internet protocols. Document how IBM responded by modifying SNA because of the popularity of TCP/IP in the 1980s and 1990s.
9. Investigate the struggle to develop open protocols. That is, those protocols that can be used without license. Read books such as Russell [2014]. Which protocols from the ISO OSI set of protocols have survived and why did those survive while others did not become as popular and widely used? Why has it been difficult to create useful but open networking protocols? What can doom such an attempt to failure? What makes an attempt successful? Detail an example of a successful new open protocol.

6.10 Further Readings and Online Resources

An excellent book that describes early data communications is Holzmann and Pehrson [2003] up to the time of the telegraph. Another short book that briefly describes modern communication history is IEEE Communications Society [2002], which focuses on events since 1952 and includes a number of oral histories from notable participants. An interesting perspective of the scope of the Bell System in the United States is provided by Bell Laboratories [1977], with a history from an AT&T Bell System perspective in Millman [1984]. Russell [2014] provides a thorough description of the emergence of open network standards and the interrelationship between telecommunications standards and those developed for computer data networking. A large collection of resources related to network history can be found in Hook and Norman [2002]. More specific books on the Internet include Abbate [2000], which describes the history behind the Internet's development, with books such as Hafner and Lyon [1998] providing a more popular version. An unpublished history discussing the impact of Usenet with the Internet is Hauben and Hauben [1997]. Krol [1992] describes the resources available on the Internet in 1992 and provides a glimpse of the culture around the emerging popularity of computing and the Internet. A set of networking and computing milestones is provided in Moschovitis et al. [1999]. Pelkey [2019] provides a compilation of computer communications technologies in the context of business needs and evolution.

7

Database Management Systems

The management of data is key to the development of any modern, large-scale system. But that wasn't always the case during the history of computing as early computers focused on numerical problem solving and didn't have direct access to a lot of online data. As computer systems became larger and more widely available, it was clear that a reliable way to store, change, and report on data was needed by many applications that were being built. With the advent of more online storage such as IBM's 305 RAMAC[1] system in 1956 (see Figure 7.1), systems that relied on data became more feasible. By the 1960s, ways to efficiently store, change, and report on data took the form of file systems that evolved into what became known as database management systems (DBMSs). These DBMSs took several different architectural approaches that will be discussed in this chapter. By the 1980s, the relational approach began to take hold, and by the 1990s, became the most common database architecture.

The amount of online file storage available to systems has increased at an incredible pace since the first online systems. Online files are those that can be immediately referenced and used without human intervention. In the early 1960s, systems able to access files of thousands of millions of bits (gigabits or 100s of megabytes) were considered very large [Martin 1977, p. 3]. By the early 1970s, the largest systems were able to access files of hundreds of thousands of millions of bits (100s of gigabits or 10s of gigabytes). By the early 1980s, larger systems were able to access tens of millions of millions of bits (10s of terabits or terabytes). This exponential growth trend of increased online file capacity has continued to increase by a factor of about 100 every decade, with systems in the 2010s exceeding hundreds of terabytes and moving well into petabyte and exabyte territory. Databases

1. RAMAC stood for "Random Access Method of Accounting and Control." The 305 RAMAC system (see Figure 7.1) came with the IBM 350 disk storage unit that could store about 3.75 megabytes (5 million 6-bit characters).

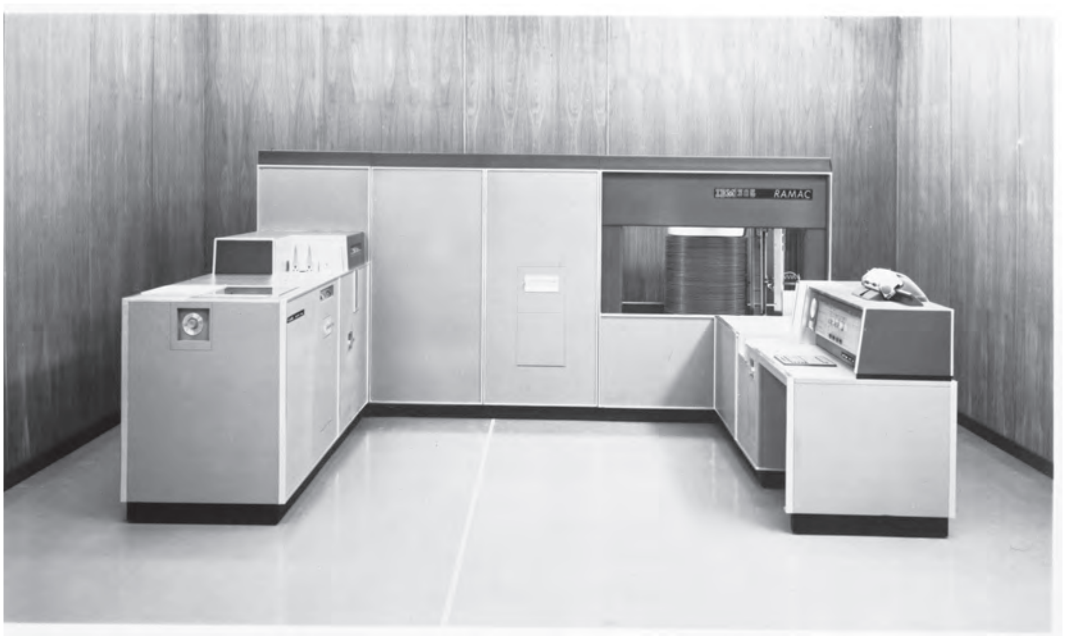


Figure 7.1 The IBM 350 disk storage unit stored 5 million 6-bit characters in 1956. Note the actual disk platters are in the back right and were visible though the window. (Source: Courtesy of International Business Machines Corporation, ©International Business Machines Corporation.)

have had an interesting history that directly influences how systems perform, their reliability, and their scale. This chapter describes how databases evolved from varying approaches to largely consolidate on the relational data model and the widely used Structured Query Language (SQL). Databases also evolved during a time when standardization via the Conference/Committee on Data Systems Languages (CODASYL) was active and influential, having established the COBOL programming language standard previously.

7.1 Overview of Database Systems and Their Evolution

The early drivers and objectives of DBMSs are described in the March 1976 issue of ACM's *Computing Surveys* (see Fry and Sibley [1976, p. 8]) as having the following objectives:

- to make an integrated collection of data available to a wide variety of users;
- to provide for quality and integrity of the data;
- to insure retention of privacy through security measures within the system; and

- to allow centralized control of the data base, which is necessary for efficient data administration.

These objectives show that more than an indexed filesystem was needed in order to achieve these objectives. In fact, all four of these objectives were at a higher level than filesystems and provided enhanced services to those needing access to the data in the database.[2]

Another key objective of DBMSs is that of *data independence*. The notion of data independence (as in Fry and Sibley [1976]) is two-fold:

1. *Physical data independence* where the programs that access the data are relatively independent of the storage or access methods, and
2. *Logical data independence* where the programs that access the data are relatively independent of logical changes to the database.

File systems and some early database systems had not been independent from the physical data storage methods and equipment, often so that they could perform adequately for the time. Having that tight dependency made it possible for those systems to maximize their performance and minimize response time for the particular data transactions they needed to support for a given application. So, making them more independent was at the cost of system performance. Similar to the development of programming languages' compilers, the performance cost of a DBMS had to be small enough to make the level of abstraction added by a DBMS worthwhile.

These notions of data independence are other hallmarks of what makes a database distinct from the underlying storage, the access methods, as well as the logical structure of the data. The intent of all these objectives was to make data more usable and maintainable by creating a system whose purpose was to manage data. These objectives are still at the core of what makes a DBMS, with additional notions becoming more explicit such as the support for database consistency through the use of *transactions.* These transactions are groups of database actions such that the database is valid before any of the actions and valid after all the actions have completed. So, if a transaction is only partially completed, then the database "rolls back" to the state before the transaction started.

To get a high-level picture of how databases have evolved, see Figure 7.2. In the 1950s, needs increased for data to be stored and retrieved as well as relatively affordable devices to be able to store large amounts of data.[3] So, many different efforts

2. Note that "database" was often spelled differently, particularly in the early days of databases, such as "data-base" or "data base." Both of these latter versions were used in Fry and Sibley [1976].

3. Note that "large" here meant in the megabytes range.

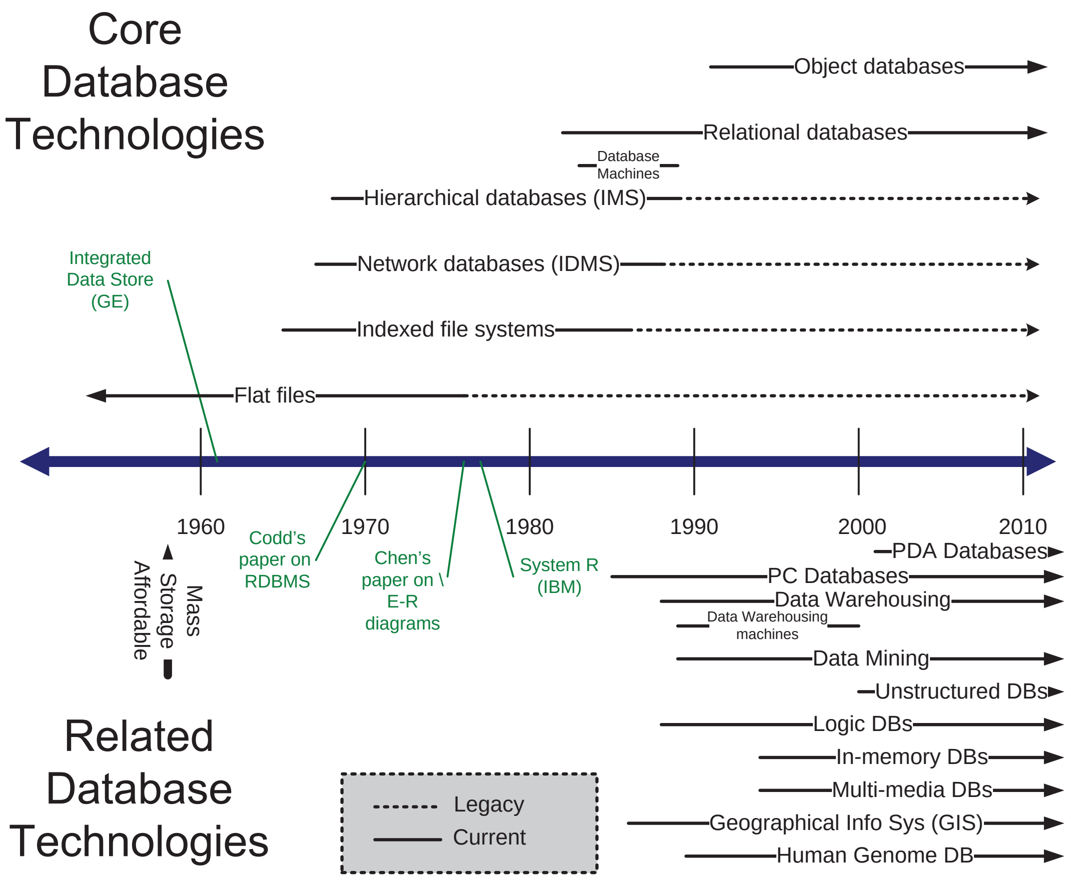


Figure 7.2 A high-level overview of databases over time.

evolved to support the storage and retrieval of data that is called "flat files" here. "Flat files" was a general term used to describe files that had very little inherent structure and therefore were "flat." These eventually converged to three models of database systems that all had internal structuring of the data and different ways of organizing data. These were the following:

- Hierarchical databases that structured data with a tree-like structure and was exemplified by the very successful IBM product, Information Management System, usually abbreviated as IMS.
- Network databases that structured data with sets used to collect records and was exemplified by the product IDMS first produced at B.F. Goodrich.

- Inverted file systems structured data using indexes built to speed up retrieval of data. One of the successful products in this space was Adabas (Software AG).

The intent of the dashed lines is to show that there is still usage of these legacy technologies, while the solid lines represent production usage for new systems during that time.

The next major shift came in the 1980s as relational databases began to become practical, particularly in terms of performance. Relational databases were based on a table interface that was intended to allow any query to be run, independent of the underlying data storage methods. E.F. Codd wrote his seminal paper on relational DBMSs in 1970, along with another paper written by P. Chen on entity-relationship data modeling that drove the desire for real, operational systems. IBM's System R was a research system based on Codd's relational model that showed such systems could be developed. As those systems improved in performance, they began to take off in the 1980s and are still the basis of most databases used in applications today. During the early part of relational database systems, there was also a short period of time when expensive database machines were developed to help make relational technology perform at the needed levels. These died out as relational database technology improved. Object databases (or just *object bases*) that store objects as created by object-oriented programming languages such as C++ and Smalltalk had a brief run, but now most relational database system have created facilities to store more complex data items including Binary Large OBjects (BLOBs) and object-like features. So, pure object-oriented databases are rarely used for modern applications. On the lower half of Figure 7.2 are a set of database-related technologies that show some of the diversity of database technologies that have been created, largely in the context of relational database technologies. They are not meant (by any means) to be all-inclusive of the technologies that have evolved. These are described below to give a flavor for the varied types of database technologies:

- PDA databases: Personal digital assistant (PDA) databases were specially designed databased to allow PDA applications to store data locally. These have evolved into small DBMSs such as SQLite that runs on smartphones and other smaller devices.
- PC databases: Personal computer (PC) databases were initially developed largely independently of larger system DBMSs. Databases such as FoxPro (Fox Software) and dBase (Ashton-Tate) were developed in the early 1980s and served to help in building PC applications.

- Data warehousing and data warehousing machines: As the number of databases grew in a particular organization, there was more demand to use that data together to better understand what was occurring in the organization. So, data warehouses were built in varying models to support that integration of data across multiple origin databases in order to do that type of analysis.
- Data mining: Data mining is a technique that grew from trying to extract interesting patterns from large datasets, such as data warehouses. There are a wide variety of techniques for doing this including statistical, visual, and machine learning-based techniques.
- Unstructured DBs/NoSQL: As the amount of unstructured data continued to grow with web data and traffic and the need to handle data not structured in tables, such as eXtensible Markup Language (XML), database systems were developed to handle this data. The re-emergence of non-relational databases in the 1990s have been generically called "NoSQL" databases, meaning they did not use SQL as their query language. Many of these now also support SQL, which has altered the meaning of NoSQL from "no SQL" to "not only SQL."
- Deductive databases: As the need to store knowledge grew, several deductive databases were developed to be able to use logical inference to reason about the knowledge it contained. One of the most popular query languages for deductive databases is Datalog, which uses Prolog-like syntax.
- In-memory databases: In-memory databases use main memory rather than disk storage as the primary home for the database, making it much faster. This has increasingly become economically feasible as the costs of memory have declined, thereby making it possible to store the entire database in main memory. These databases usually still support transactional concepts and other core, database concepts.
- Geographical Information Systems (GIS) and spatial databases: Some application areas, such as GIS, have developed database systems to meet their specialized needs. With GIS systems, one need is to represent topographical data and to do queries related to spatial information (where objects are in space) in an efficient manner. Spatial databases have developed a number of techniques to optimize spatial queries.
- Genome database: An example of a database created to help drive research and information dissemination about human and other organisms' genomes is the genome database. One such database is housed at the US

National Institutes of Health's (NIH) National Center for Biotechnology Information website: https://www.ncbi.nlm.nih.gov/genome/. This database is optimized to make it easier to find information about any particular part of a genome and to share that data. While this database relies on other database technology, it has developed a number of mechanisms specific to the problem of sharing genetic information and for matching genetic sequences to those in the genome database.

These types of databases and DBMSs are intended to be illustrative of the types and how database systems developed to meet more specialized needs. From the above, one can see that the capabilities of the system have driven some specialized databases (like PDA databases). Others have been driven by the needs of the user base (like GISs and genome databases), and others are driven by performance, such as in-memory databases. Fundamentally new functionality has driven some of these changes, such as with object bases and deductive databases.

In summary, database technology has gone through a formative period, which is detailed in the next section, followed by several competing types (hierarchical, inverted file, and network) that were then largely replaced by relational technologies. Since that coalescence into relational technology, we have had several other technologies that build on relational database technology, while others diverge from some relational principles (such as NoSQL databases).

7.2 Early Database History

Several technologies contributed to the development of DBMSs. These include the development of data definition languages in some of the earliest systems, the development of report generators, and the developing of indexing and file system technologies.

7.2.1 Data Definition Languages

The intent of data definition languages was to begin to abstract the layout and access of data from the physical devices and layout on storage media. The ability for programmers to access data by name and not to have to re-define the structure of the data for every program led to efforts to build data definition languages.

One of the first data definition languages was COMPOOL (COMmunications POOL), which was developed for the SAGE air defense system by RAND, System Development Corporation, and MIT Lincoln Laboratory. COMPOOL was a set of common data that had a dictionary of shared data names, locations, and definitions. COMPOOL was included in the JOVIAL programming language (Jule's Own

Version of International Algebraic Language[4]). From the JOVIAL perspective, it freed the programmer from having to find data items and they could just reference them by name, as this segment from Cheatham [1978] shows how one could reference "ITEM" in COMPOOL using assembly language instructions in a JOVIAL program (see Listing 7.1):

```
CLA  ITEM
ETR  ITEM
POS  ITEM
```

Listing 7.1 SAGE COMPOOL ITEM definitions.

This sequence (see Listing 7.1) first moves ITEM to the accumulator, then ANDs it with a mask, and finally positions the least significant bit to be rightmost. So, ITEM's value ends up in the accumulator, without the program having to find it and load it from wherever it might be stored.

COBOL was also defined to have an explicit DATA DIVISION that was used to separate the data definition from the rest of the COBOL program and was included as part of the CODASYL definition of COBOL.

Additional work was done by CODASYL with the Stored-Data Definition and Translation Task Group in 1969 and work continued to formalize data definition models over time. Around the same time methods of systems analysis began to use data definition techniques to define system requirements into the 1970s and 1980s.

7.2.2 Report Generator Systems

Getting data out of these systems became a common problem, so report generators were developed to make that task easier and to automate some of the work that was needed. Writing a program to produce a report using only a programming language would result in having to possibly process large volumes of data that processes each input line, processes the buffer and interprets the internal formats, and outputs the appropriate values to an output device. A *report generator* could generate a report by performing complex data transformations and produce usable reports by automating the process and using a relatively simple command input file.

Fry and Sibley [1976] categorizes this set of systems as the Hanford/RPG family of early databases and their precursors in Figure 7.3. The "Hanford" comes from the work done at the Atomic Energy Commission at its Hanford site in Washington state where General Electric Company created a report generator in 1956 for

4. See Schwartz [1981] for a description of how the "J" was prefixed to OVIAL, "Our Version of the International Algebraic Language," as OVIAL sounded to be related to the birth process.

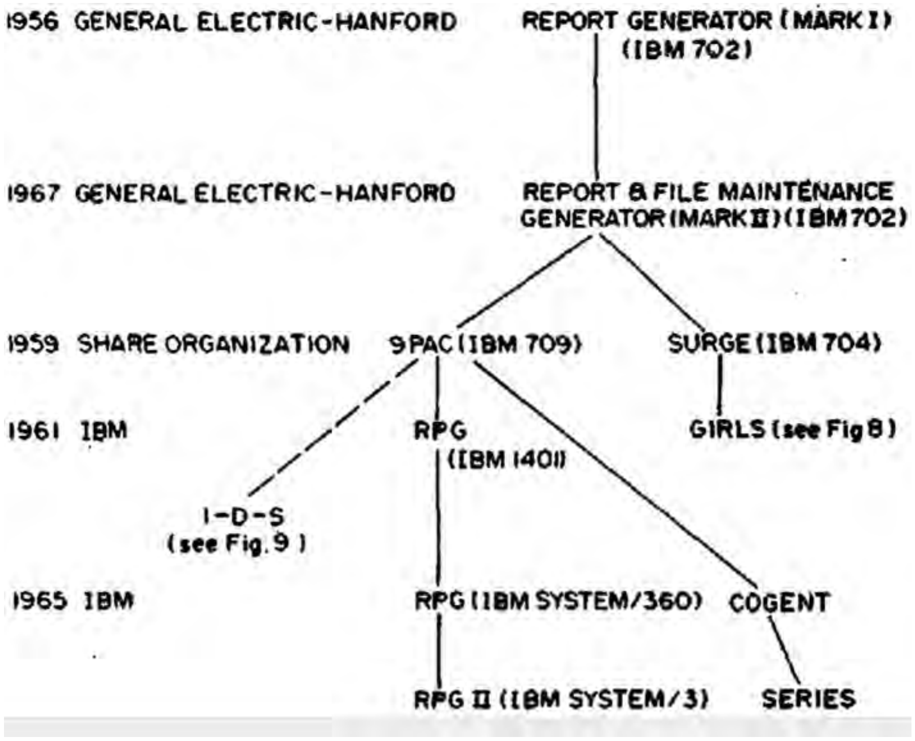


Figure 7.3 Hanford/RPG family of report generators (from Fry and Sibley [1976, p. 21]). (Source: James P. Fry and Edgar H. Sibley. 1976. Evolution of data-base management systems. *ACM Comput. Surv.* 8, 1 (March 1976), 7–42. DOI: https://doi.org/10.1145/356662.356664.)

the IBM 702 called "Mark I." The IBM user group SHARE took these routines and expanded on them and created routines for the IBM 704 and 709 in 1959. These were later adapted by IBM to become the RPG (Report Program Generator) series of programming languages that were also central to the later class of turn-key[5] machines such as the IBM System/3, System/32, System/34, System/36, System/38, and the AS400. Notably, this work also feeds into the Integrated Data Store (I-D-S) system at General Electric, usually considered the first DBMS. Additionally, SHARE's SURGE system developed for the IBM 704 contributed to the ideas in the Generalized Information Retrieval and Listing System (GIRLS) developed for the IBM 7090 used by Informatics. The Informatics branch is not included here but is in Fry and Sibley [1976, p. 23].

5. So called "turn-key" because they were meant to be smaller computers that did not require a lot of additional programming and included the programs that a small to medium business would need.

7.3 Types and Evolution of Database Systems

This section details some of the efforts and projects related to early database-like and database systems. It is significant in that a large number of organizations were working on such systems and that these efforts were interrelated due to data sharing at conferences and in publications. The large number of organizations working on this indicate the significance of the data management problem and the widespread need to have reliable database management capabilities that could be re-used without tremendous effort.

After the early evolution described above and before the emergence of relational database systems, three database models were prevalent:

- Hierarchical: See Figure 7.8 and epitomized by the IBM IMS database management system.
- Network: See Figure 7.7 and epitomized by the IDMS database system (Cullinane Database Systems renamed Cullinet in 1983 and purchased by Computer Associates in 1989).
- Inverted File: See Figure 7.9 and eventually epitomized with products such as Software AG's Adabas product.

7.3.1 I-D-S: The First DBMS?

As with many new technologies, precisely identifying which is "first" is difficult. At the time that databases and their DMBSs were being developed, it was becoming clear that systems like a DBMS would be useful. So, many people and organizations were involved as noted in the early database history section (Section 7.2).

However, one effort seems to coalesce much of the thinking at the time and is often called the first DBMS. That one is the I-D-S by Charles Bachman while at General Electric.[6] I-D-S was conceived and designed in 1962 with the first running prototypes in the summer of 1963. One of Bachman's I-D-S drawings is included here as Figure 7.5, where it displays the basic concepts of a networked database linking records in a set. From the author's review of Bachman's papers, it's clear that there was a lot of coordination with SHARE, the IBM user group. Many organizations were struggling with a similar problem of how to store structured information in a reliable and high-performance manner. See Figure 7.4 for developments that I-D-S influenced and how it traveled with Bachman to General Electric. It should also be

6. See "How Charles Bachman invented the DBMS, a foundation of our digital world." Thomas Haigh, *Commun. ACM*, 59, 7, 25–30.

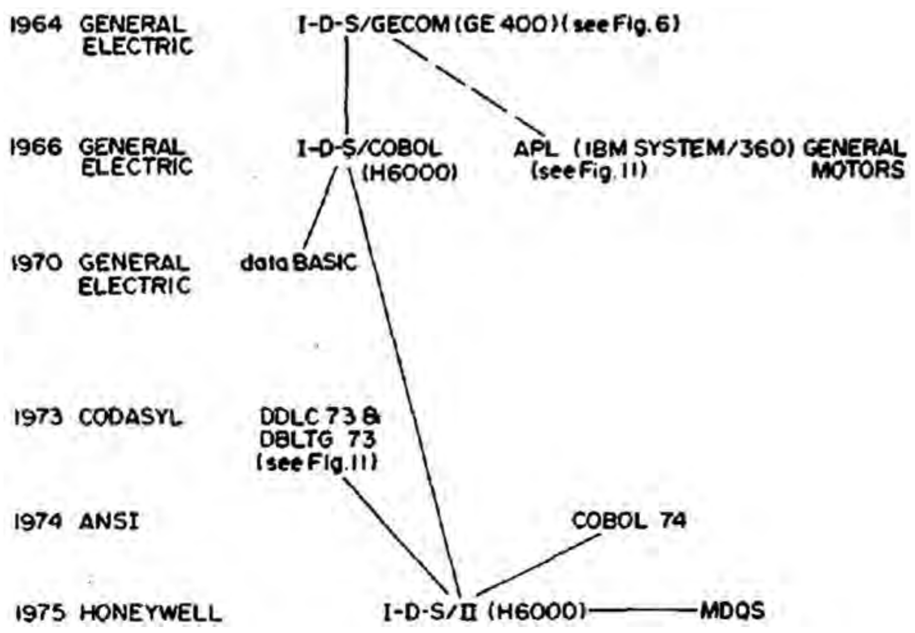


Figure 7.4 Bachman/I-D-S family of DBMS (from Fry and Sibley [1976, p. 23]). (Source: James P. Fry and Edgar H. Sibley. 1976. Evolution of data-base management systems. *ACM Comput. Surv.* 8, 1 (March 1976), 7–42. DOI: https://doi.org/10.1145/356662.356664.)

noted that the reference to "APL" in Figure 7.4 has no relation to the APL programming language (which was also developed on the IBM System/360), but instead to a system called Associative PL/I developed by General Motors for data management in a computer-aided design (CAD) environment. Bachman's I-D-S was heavily influential in early database thinking and its network data model became central to the CODASYL data model standardization efforts. See Figure 7.6 for how Bachman presented the concept of a *data base* as a bunch of pigeon holes. The CODASYL DBTG (Data Base Task Group) was a subcommittee of the same body that earlier developed the COBOL programming language standard. A description of how these systems evolved and standards were developed is in Haigh [2009]. The network data model allowed parent records to have multiple child records linked to them, called "set types" in the CODASYL standard. This model, while flexible, turned out to be relatively complex to implement and many organizations instead chose to implement the somewhat simpler hierarchical model championed by IBM. Bachman's I-D-S drove the network model. A relatively popular network-model database was IDMS (Integrated Data Management System) by Cullinet. Figure 7.7 shows this relationship of Bachman's I-D-S to IDMS.

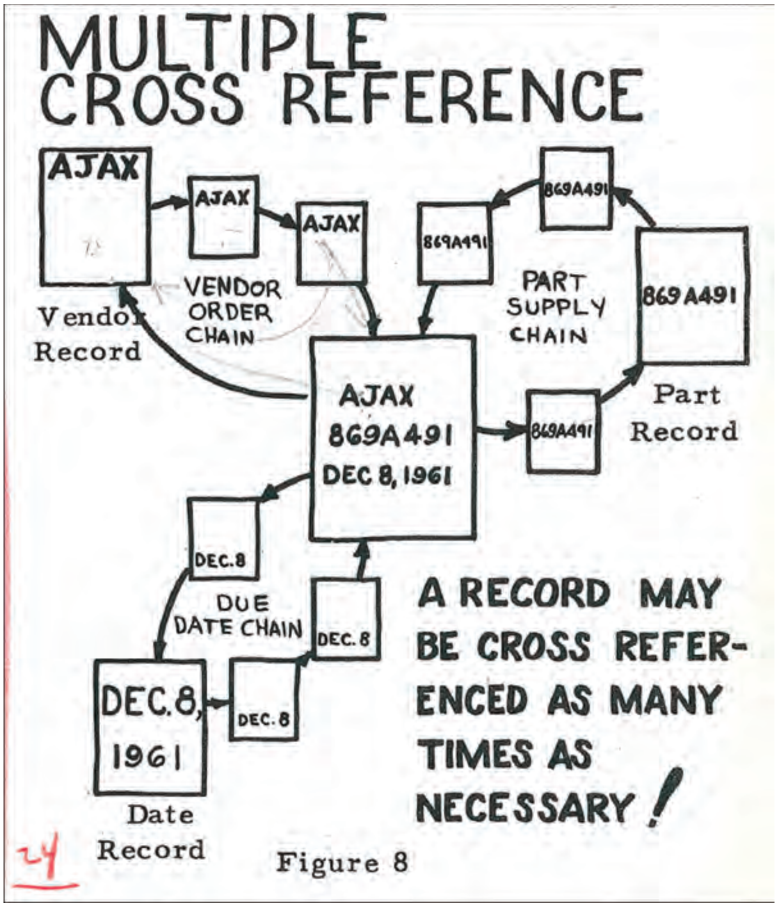


Figure 7.5 This drawing includes showing how the database would be able to access data in different ways and eventually evolved into the CODASYL data model for network-model databases. (Source: Image courtesy of the Charles Babbage Institute Archives, University of Minnesota Libraries, Minneapolis.)

7.3.2 IBM's IMS and Hierarchical Databases

IBM built on efforts from working on database systems for the Apollo space program to build a system based on a hierarchical structure for information. This hierarchical data model was also complex, but was well-supported by IBM and became popular on IBM systems with the evolution as in Figure 7.8. The GUAM system was developed for Apollo with DL/1 (Data Language 1), then developed as a more general-purpose data management system, eventually renamed as IMS (Information Management System). IMS was widely used for many production systems and built to leverage IBM's file system products such as ISAM (Index Sequential Access Method) and VSAM (Virtual Sequential Access Method).

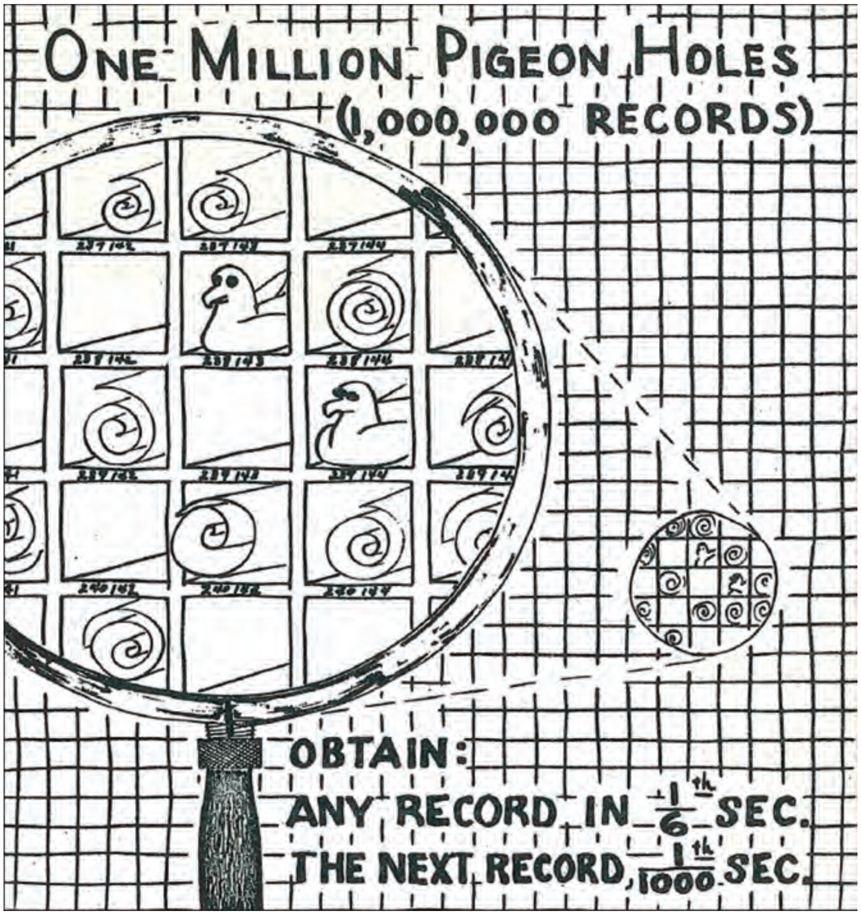


Figure 7.6 Bachman's drawing showing the concept of a *data base*. (Source: Image courtesy of the Charles Babbage Institute Archives, University of Minnesota Libraries, Minneapolis.)

A well-designed IMS database could perform very well for the purposes for which it was designed. One of the issues with IMS (and IDMS for that matter) was that if different purposes for the data were later needed it could be very difficult to make those perform well as the hierarchical (and network) structure may not fit the query that was needed or store the relationships between data elements that was needed. So, the performance of a particular query against the database was highly dependent on the precise implementation of the data hierarchy (or network).

7.3.3 Inverted File Systems

Inverted file systems were built to find values in records. So one could use a keyword to find all records that contain that particular keyword. Figure 7.9 shows the

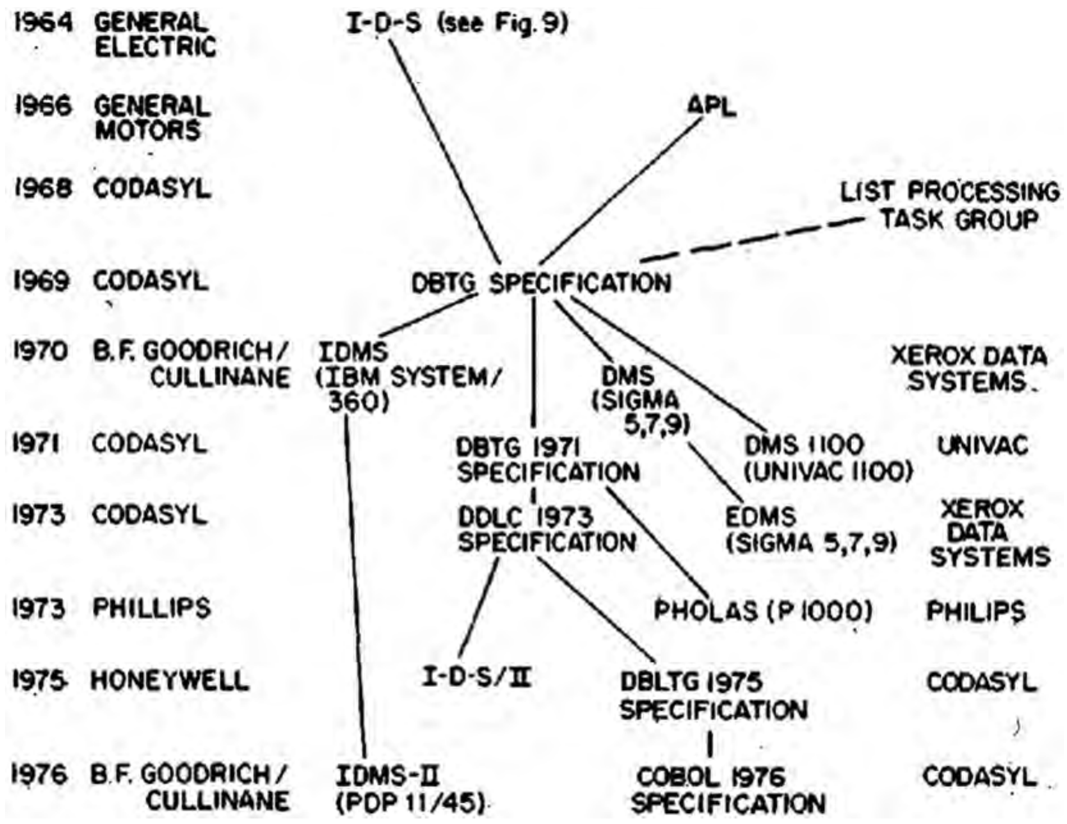


Figure 7.7 IDMS/CODASYL Family of DBMS (from Fry and Sibley [1976, p. 26]). (Source: James P. Fry and Edgar H. Sibley. 1976. Evolution of data-base management systems. *ACM Comput. Surv.* 8, 1 (March 1976), 7–42. DOI: https://doi.org/10.1145/356662.356664.)

early years and development of this type of database. Early systems such as Software Development Corporation's (SDC) TDMS development are described in Haigh [2009]. This type of system continued to be used into the 1980s with systems such as Adabas, which stood for "adaptable database system." Adabas still has some use (in the 2010s) particularly with Software AG's Natural query language. Adabas is used for some high-performance applications where relational database features aren't needed. As an example, the widely used enterprise resource planning system, SAP, had an option to use a version of Adabas (Adabas D) called MaxDB.

7.4 Relational DBMSs

Edgar (Ted) F. Codd's paper on relational database system came out in 1970 [Codd 1970], yet the first products began to come out only in 1979 with INGRES and

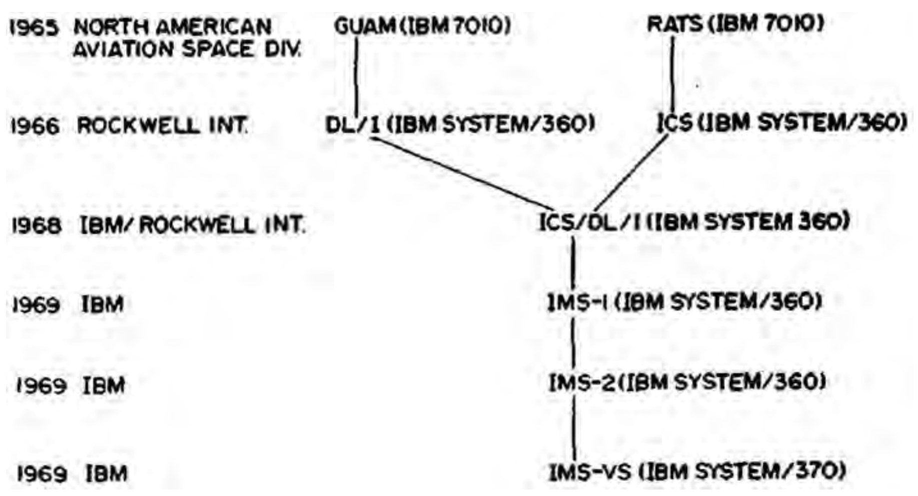


Figure 7.8 IMS family of DBMS (from Fry and Sibley [1976, p. 27]). (Source: James P. Fry and Edgar H. Sibley. 1976. Evolution of data-base management systems. *ACM Comput. Surv.* 8, 1 (March 1976), 7–42. DOI: https://doi.org/10.1145/356662.356664.)

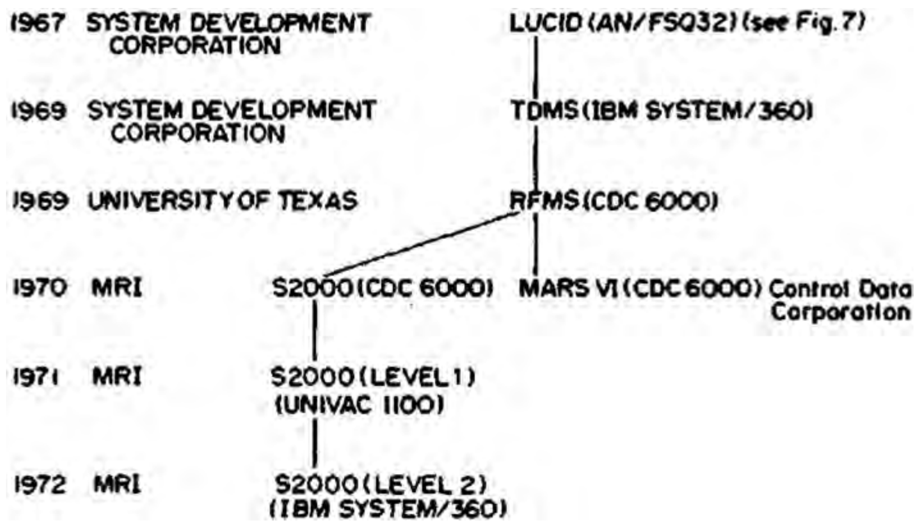

1967 SYSTEM DEVELOPMENT CORPORATION
LUCID (AN/FSQ32) (see Fig. 7)
1969 SYSTEM DEVELOPMENT CORPORATION
TDMS (IBM SYSTEM/360)
1969 UNIVERSITY OF TEXAS
RFMS (CDC 6000)
1970 MRI
S2000 (CDC 6000)
MARS VI (CDC 6000) Control Data Corporation
1971 MRI
S2000 (LEVEL 1) (UNIVAC 1100)
1972 MRI
S2000 (LEVEL 2) (IBM SYSTEM/360)

Figure 7.9 Inverted file family of DBMS (from Fry and Sibley [1976, p. 28]). (Source: James P. Fry and Edgar H. Sibley. 1976. Evolution of data-base management systems. *ACM Comput. Surv.* 8, 1 (March 1976), 7–42. DOI: https://doi.org/10.1145/356662.356664.)

Oracle, with IBM also releasing their product based on System R shortly thereafter. The relational database model described in Codd's paper was appealing and there was wide agreement that it was the direction to go. Two major factors made the relational model different than previous models:

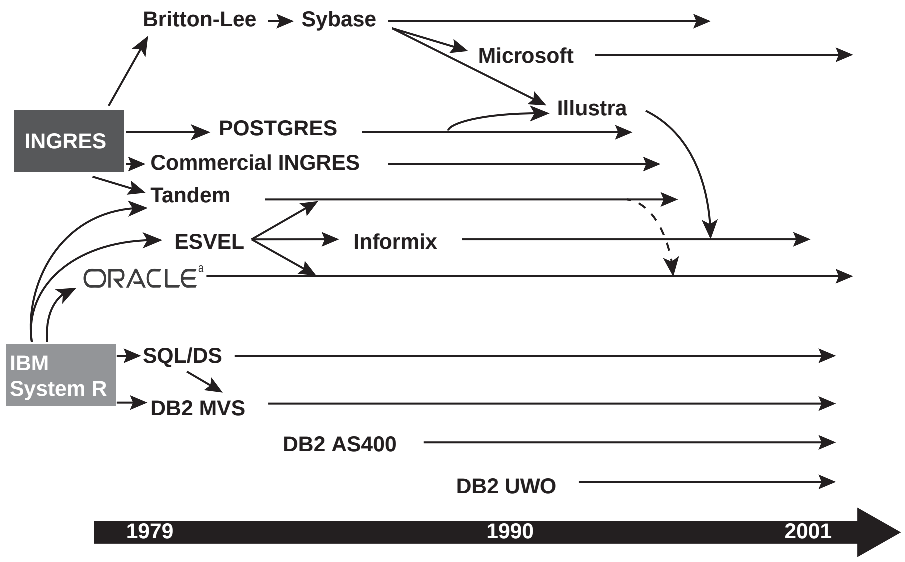


Figure 7.10 Commercial server/mainframe RDBMS relationships over time. (Source: 2013 IEEE. Reprinted, with permission, from A. Mendelsohn. The Oracle story: 1984–2001. In *IEEE Ann. Hist. Comput.* 35, 2, 10–23, April–June 2013, DOI: http://dx.doi.org/10.1109/MAHC.2012.56.)

- The separation of the data model from how the data was stored.
- Basis of operations and model on a mathematical model (relations).

As relational databases began to be built (see Figure 7.10), Codd felt a need to clarify what was meant by the relational model and how it should be implemented in a DBMS. The relational model had taken hold and systems were being developed that interpreted the model in different ways. Different companies were making assumptions and having different priorities on what to include in an RDBMS. So, he developed a list of 13 rules (or 12 + the zeroth rule) defining what it meant to be relational. As the author of the relational model, E.F. Codd created these rules for RDBMS in order to refine and coalesce the RDBMS products into a common perspective of what it meant to be a RDBMS. For the most part, Codd was successful in getting most RDBMS products to adhere to most of these rules, often because customers of these products would evaluate their products using these rules.

Codd's 13 rules for RDBMS:

- Rule 0: The foundation rule
- Rule 1: The information rule
- Rule 2: The guaranteed access rule

- Rule 3: Systematic treatment of null values
- Rule 4: Dynamic online catalog based on the relational model
- Rule 5: The comprehensive data sublanguage rule
- Rule 6: The view updating rule
- Rule 7: High-level insert, update, and delete
- Rule 8: Physical data independence
- Rule 9: Logical data independence
- Rule 10: Integrity independence
- Rule 11: Distribution independence
- Rule 12: The non-subversion rule

Codd continued to refine the model and added further details, producing a "Version 2" of the relational model. During the mid- to late-1990s, OnLine Analytical Processing (OLAP) systems were becoming popular and again Codd established a set of rules to define what it meant to be a database system that supported OLAP.[7] Relational databases continue today to be the primary database architecture used, though over time extensions have been added to the relational model to support objects and other data types.

7.4.1 SQL Standardization

The now standardized query language for relational databases was not the only choice as mainstream relational databases took shape in the late 1970s. As noted in Figure 7.10, there were two primary RDBMS products built at around the same time: IBM's System R (see Chamberlin et al. [1981]) and University of California-Berkeley's INGRES (see Stonebraker [1980]). The evolution of SQL is detailed in Deutsch [2013]. The CODASYL network database standard was not applicable to relational databases, though some of the terms established there such as data manipulation language (DML) and data definition language (DDL) were also used in the context of relational database query language standards.

In Figure 7.11 the relationships between the various relational query languages are given. Two mathematical models were the basis for competing relational query languages.[8] The models are relational calculus and relational algebra. Relational

7. OLAP systems were systems built to do analysis and were often de-normalized so that queries involving large subsets of the data would run efficiently. This term was meant to distinguish such systems from OnLine Transaction Processing (OLTP) systems, which focused on serving users adding, changing, and removing data with individual transactions.

8. These two models are essentially equivalent in power as per what is called *Codd's Theorem*.

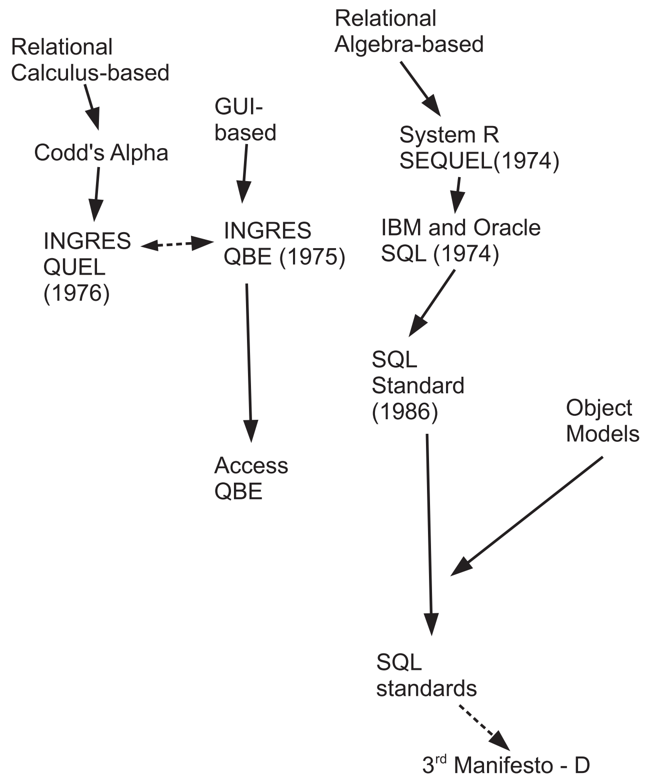


Figure 7.11 Relational query languages and their influences on each other.

calculus[9] focuses on characterizing *what* the result of the query is and is thereby more *declarative* in nature. Relational algebra specifies operations on relations and specifies *how* to get the result, making it more *procedural* in nature. Two major query languages were produced, one for System R and another for INGRES. SQL was created for System R, which has a basis more in relational algebra and can be translated to relational algebra. However, SQL was created to not directly be based on relational algebra and is more declarative than relational algebra due to this abstraction.[10] So System R could choose an execution plan for how to execute the query. INGRES produced a language called QUEL, which was based on relational

9. There are actually two varieties of relational calculus. One is *tuple relational calculus* and the other is *domain relational calculus*.

10. Interestingly, Codd, the creator of the relational model who worked for IBM, is said to have preferred relational calculus over a relational algebra-based query language.

calculus and specified the characteristics of the result rather than how to produce the result.[11]

As described in Deutsch [2013], these two competing relational query languages were the basis of determining a standard query language. When Oracle created the first commercial implementation of SQL (IBM's System R was a proof of concept, rather than a product at that point), this supported the case for SQL becoming the standard. Even though QUEL had considerable support on the standardization committee [Deutsch 2013, p. 73], no one from the INGRES project participated on the committee so QUEL was never seriously considered by the standardization committee. The committee took the SQL language from System R as the base and then made it more orthogonal, intuitive, and symmetric.

Another method for querying relational databases was Zloof's Query-by-Example (QBE), which used a method of specifying constraints and values to get the data that you wanted to retrieve. QBE (see Zloof [1975]) is used (usually in conjunction with SQL) in some systems today, such as Microsoft's Access database product.

7.5 System R: Sample Code

IBM's System R was developed to show the feasibility of the relational model and it's hard to argue that it wasn't widely successful at doing that. While not strictly the first relational system built, it was highly influential and stimulated the industry. System R implemented a number of new features including query optimization and a recovery manager. Query optimization was critical to show that the relational model could have usable query performance. Since System R implements SQL, when an SQL statement was interpreted and translated to searching an actual table, a *query plan* could be created that would decide what order to perform the relational operations as there are several different ways that a query can be implemented and that order can have a large impact on the performance of the query. One example is the code from System R that was used to process SQL WHERE clauses and written by Pat (Griffiths) Selinger.

This code is well-documented and written in PL/I. The following snip-it (Listing 7.2) are the leading comments for this file (XWHERE.PLIOPT).[12] It's not

11. QUEL was based on a language suggested by Codd called Data Sub-Language ALPHA, defined in Codd [1971].

12. Note that this code was acquired from the Computer History Museum in line with their agreement with IBM and is actual code from the 1977 version of System R.

surprising that System R was written in PL/I, given its use as a systems programming language at the time and the importance of PL/I at IBM as a programming language.

```
/*  04/13/77  PPG  MMA  */                                            XWH00010
/*  5/4/77  DON'T  PRINT  %INCLUDES  */                                 XWH00020
XWH00030
/*  XWHERE  IS  THE  PROCEDURE  WHICH  EXAMINES  THE  WHERE  PREDICATE      XWH00040
TREE.  ON  THE  FIRST  PASS  IT  COLLECTS  COLUMN  NAMES,  PUSHES      XWH00050
NOT  NODES  DOWN  THROUGH  BOOLEAN  NODES  AND                         XWH00060
(EVENTUALLY)  REMOVES  NOTNODES  IF  ALL  PREDICATES  BELOW  THEM      XWH00070
ARE  SARGABLE  (COMPATIBLE  WITH  THE  'COL  COMPOP  LIT'  FORMAT).   XWH00080
ALSO  ON  THE  FIRST  PASS,  IF  ONLY  ONE  SIDE  OF  A  PREDICATE  HAS  XWH00090
A  COLUMN,  THEN  WE  WILL  PUT  IT  IN  THE  LHS  OF  THE  RELATIONAL   XWH00100
OPERATOR  AND  ADJUST  THAT  OPERATOR  ACCORDINGLY.  (>  BECOMES     XWH00110
<,  ETC.).  ON  THE  SECOND  PASS,  XWHERE  PROPAGATES  DATA  TYPES   XWH00120
AND  SPECLENS  IN  THE  PREDICATES,  TESTS  FOR  DNF  STATE,  AND      XWH00130
FINDS  BT'S  AND  ENTERS  THEM  INTO  BTARRAY.   A  BT  IS  A           XWH00140
PREDICATE  OR  A  SUBTREE  HEADED  BY  AN  ORNODE  AND  MUST  BE       XWH00150
THE  WHERE  TREE  ROOT  OR  MUST  BE  CONNECTED  TO  THE  ROOT  ONLY    XWH00160
BY  ANDNODES  AND  MUST  BE  SARGABLE.                                 XWH00170
A  PREDICATE  BT  MAY  BE  REPLACED  BY  AN  ACCESS                    XWH00180
PATH  SELECTION.   A  BT  THAT  IS  IN  DNF  MAY  BE                   XWH00190
REPLACED  BY  SEARCH  ARGUMENTS.   A  BT  THAT  IS  NOT  IN            XWH00200
DNF  MAY  SOME  DAY  BE  PUT  INTO  DNF  AND  REPLACED  BY  SEARCH      XWH00210
ARGUMENTS.                                                            XWH00220
XWH00230
XWHERE  MARKS  THE  NODES  AS  FOLLOWS:                                XWH00240
BOOLEAN  P1  INDEX  OF  PARENT  NODE                                   XWH00250
P2  NUMBER  OF  NODES  BELOW,  INCLUDING  THIS  ONE          XWH00260
NOTNODE  P1  INDEX  OF  PARENT  NODE                                   XWH00270
PREDICATE  P1  1  IF  SARGABLE,  ELSE  0                               XWH00280
P2  NUMBER  OF  NODES                                     XWH00290
XWH00300
```

Listing 7.2 Comments about query optimization PL/I code for WHERE clause in SQL for IBM System R.

In the above comments (Listing 7.2) for this file that process SQL WHERE clauses, it is described how the code parts of the WHERE are disassembled. Later in the code (Listing 7.3), we have the following snip-it of PL/I code:

```
      IF ¬FIRSTPASS THEN                                                XWH01670
         IF P4=ANDCODE THEN NEWANDONLY=ANDONLY;                         XWH01680
         ELSE NEWANDONLY='0'B;                                          XWH01690
       CALL XWHERE(P3,NODEIND,OPPTR,FIRSTPASS,NEWANDONLY,LEFTSARGABLE,  XWH01700
         LEFTDNF,LORCHILD,NLEFTNODES,STACKINDS,BQPTR,CHKMODE,           XWH01710
          RUNTIMEMODE,CURSEXECMODE,AUTHID);                             XWH01720
       TESTCODE;                                                        XWH01730
       IF OPTLEVEL > 2 THEN                                             XWH01740
         DISPLAY('RETURNED_FROM_XWHERE_CALL_ON_LHS_WITH_LSARG='         XWH01750
               ||LEFTSARGABLE);                                         XWH01760
       CALL XWHERE(P5,NODEIND,OPPTR,FIRSTPASS,NEWANDONLY,RIGHTSARGABLE, XWH01770
          RIGHTDNF,RORCHILD,NRIGHTNODES,STACKINDS,BQPTR,CHKMODE,        XWH01780
           RUNTIMEMODE,CURSEXECMODE,AUTHID);                            XWH01790
       TESTCODE;                                                        XWH01800
       IF OPTLEVEL> 2 THEN                                              XWH01810
         DISPLAY('RETURNED_FROM_XWHERE_ON_RHS_WITH_RSARG='||            XWH01820
               RIGHTSARGABLE);                                          XWH01830
       SARGABLE=LEFTSARGABLE & RIGHTSARGABLE;                           XWH01840
       IF ¬FIRSTPASS THEN                                               XWH01850
         DO;                                                            XWH01860
           P2,                                                          XWH01870
           NUMNODES = NLEFTNODES + NRIGHTNODES + 1;                     XWH01880
           IF NLEFTNODES > NRIGHTNODES THEN                             XWH01890
             DO;                                                        XWH01900
                                                                        XWH01910
               /* INTERCHANGE LEFT AND RIGHT TO PUT SIMPLER PREDICATES  XWH01920
                     ON LEFT*/                                          XWH01930
                                                                        XWH01940
               TEMPIND = P3;                                            XWH01950
               P3 = P5;                                                 XWH01960
               P5 = TEMPIND;                                            XWH01970
             END /* OF INTERCHANGE */;                                  XWH01980
           DNFFLAG=LEFTDNF & RIGHTDNF;                                  XWH01990
           IF P4=ANDCODE THEN DNFFLAG=DNFFLAG & ¬LORCHILD & ¬ RORCHILD; XWH02000
           IF P4=ORCODE THEN ORCHILD='1'B;                              XWH02010
           ELSE ORCHILD='0'B;                                           XWH02020
           IF P4=ORCODE & ANDONLY & SARGABLE THEN                       XWH02030
                          /* ENTER INTO BTARRAY */                      XWH02040
```

```
39                 CALL  XFILBT(NODEIND,PARIND,OPPTR,DNFFLAG,ORTREEKIND,          XWH02050
40                    '0'B,     /* CORRELATION NOT APPLICABLE */                 XWH02060
41                   NT,STACKINDS,BQPTR,BTTABIND);                               XWH02070
42            END;     /* ¬FIRSTPASS */                                          XWH02080
43          RETURN;                                                              XWH02090
44       END;     /* BOOLEANNODE */                                              XWH02100
45                                                                               XWH02110
46     ELSE IF NT=NOTNODE THEN                                                   XWH02120
47       DO;                                                                     XWH02130
48         IF OPTLEVEL>2 THEN DISPLAY('XWHERE_FOUND_NOTNODE');                   XWH02140
49  /* TEST */                                                                   XWH02150
50     IF P5=0 THEN                                                              XWH02160
51       DO;                                                                     XWH02170
52         CALL XSYSTEM(PNAME,-101,-901,'');                                     XWH02180
53         GOTO QUIT;                                                            XWH02190
54       END;                                                                    XWH02200
55  /* ENDTEST */
```

Listing 7.3 Query optimization PL/I code snip-it for WHERE clause in SQL for IBM System R.

In this segment of the code we can see efforts to simplify the clauses by rearranging predicates included in the WHERE clause, such as from lines 26–32 where predicates are swapped. This code also goes through multiple passes and has options for different levels of optimization.

7.6 Factors Affecting Change of Database Software

As a highly practical need, DBMSs grew from a common system requirement: storing, accessing, and storing data. Exactly how this could best be done has evolved over time, partially because of the increasing power of computers and data storage systems. This was only one factor that contributes to how database systems have changed (see Figure 7.12). Early systems tried a number of different ways of structuring the data and its access including hierarchical, network, and inverted file data models. Given the speed of systems at that time, these models were able to provide acceptable performance for specific applications. When the relational data model was proposed in 1970 by Codd, the data model was cleanly separated from the physical implementation and allowed for many different uses of the same database. Over time, relational databases have improved in performance, now mostly obviating the need for a data model more closely tied to the physical data storage. Other influences that have changed database technology include the types of items being stored. When databases were needed to store objects in support of object-oriented programming languages, special database systems were built to store those objects (sometimes called *object bases*). Other examples include the storage of semi-structured data, such as the storage of XML files and other data that has a flexible format. Very large objects such as video and audio files will often also use a different type of database. Being a practical technology, DBMSs have been driven by projects that have a need for such technology. Projects such as those at General Motors that wanted to store CAD files or the Apollo project's need to store data helped drive the improvement of database software. Performance of database systems has always been (and continues to be) a concern for

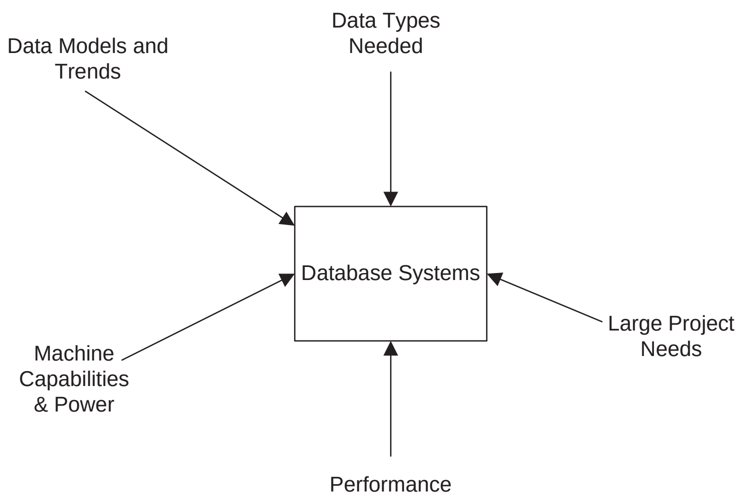


Figure 7.12 Factors influencing change in database systems.

database systems. Performance was an issue for early relational database systems and slowed their adoption until the performance could be addressed. Similarly, performance has also been a driver for the use of de-normalized data and NoSQL databases. Database technology was driven by a practical need to store and manage data and thus industry and computer manufacturers drove the development of database technology. However, significant advances have come from theoretical advances in database technology such as optimization, data storage technologies, and data models.

7.7 Lessons Learned from Database Software

Lessons from databases that have had a broad impact on other types of software include the following:

- *Data abstraction and program maintainability.*

  Perhaps the biggest lesson from database software has been the importance of abstraction to enhance program maintainability. Abstracting a program from the details of how the data is stored has freed the program from having to be changed every time we change the way data is stored. Even in the days of SAGE's use of COMPOOL, it was important to alleviate the program from having to know all the details of data storage. Over time, this abstraction has become a core feature of DBMSs by presenting a logical view of the data rather than requiring programs to know anything about the physical storage of that data.

Presenting a logical view or abstraction of a part of the system that may change has become a core way to design software systems that are more maintainable than they otherwise would be.

- *Atomic transactions.*

  The need to keep the data consistent drove the all-or-nothing approach to bundling a set of data operations into transactions. This idea is used in other systems where there is a need to bundle a set of actions so that an intervening failure doesn't leave the system in an unwanted or unstable state.

- *Need for system applications.*

  When Bachman wrote I-D-S, it was not a common practice by any means to have a system application that was distinct from the operating system and application programs. As he describes in a talk he gave at the Computer History Museum,[13] he was working on an application and over time realized that many applications could re-use this system application, which became the concept of a DBMS. This layering of system applications has since become common with many such system applications that are used by many application programs. Other examples include web servers, email servers, and network-based services.

7.8 Exercises and Projects

7.8.1 Exercises

1. As stated at the beginning of this chapter, online file sizes for real systems has historically grown by a factor of about 100 per decade. This was derived from data in Martin [1977, p. 3] and knowledge of database systems since his data (1980s onward). Is such rapid growth sustainable? What factors will inhibit the growth of the size of databases? What do you predict the size of the largest systems will be in 20 years? A hundred years?
2. Helping to enable database systems to grow in size has been the drop in the cost of storing information online. The amount of data that can be stored per dollar has increased exponentially due to reduction in the cost of storage technology as well as many other factors. Determine a rough range in bits that can be stored for a dollar since the year 2000. How has this trend changed

13. See https://www.youtube.com/watch?v=iDVsNqFEkB0 for the talk he gave in 2002 at the Computer History Museum.

when compared to the earlier years such as in Martin [1977, p. 4]?[14] If there's a change in the trend, what is causing this change?

3. DBMSs tend to have a long life time of use due to their embedded nature in the systems that are built around them. Examine the usage of the FoxPro database, a PC DBMS that evolved over a number of versions. Find out when the last version was delivered and see if you can find any running applications that still use FoxPro. Are there good reasons for these applications not to use FoxPro? If so, explain what the reasons are. Are there reasons to migrate from using FoxPro? If so, explain those reasons.
4. One of Codd's relational rules that many vendors did not implement to Codd's satisfaction was Rule 3: Systematic treatment of null values. Find out why vendors often did not precisely follow this rule and whether it is still an issue today.
5. Handheld devices have often had databases developed for them to enable additional applications to be more easily built. One example was Palm, Inc.'s Palm OS that had a number of databases developed for its PDAs. Investigate the features of one of these databases developed for PalmOS. Most of these had so few features they were difficult to call "databases." Argue whether the one you found should be called a database or not and why.
6. Databases specifically built for PCs were built with different requirements from those for large scale systems. Investigate version II of Ashton-Tate's dBase software. What features did it have that a large-scale, mainframe relational DBMS such as Informix did not have at the time? What features did Informix have that dBase II did not have? Explain why and how dBase could still be successful.
7. There have been several attempts to integrate the storage of knowledge in support of artificial intelligence systems with database systems. The intent of such systems has been to leverage the efficiency that drove the creation and enhancement of databases and to make it easier to store and access knowledge. One of these efforts was known as *Expert Database Systems* and was prevalent in the early 1990s. The idea was to use the data in a database to help make inferences and to support the rule-driven nature of an expert system. Investigate and explain why *Expert Database Systems* have fallen out of mainstream use.

14. Martin stated that in the 1960s one could store tens of thousands of bits for a dollar and in the 1970s that had increased to about 100 million bits per dollar.

8. Find out about the TDMS (time-shared data management system) database system that was built by Systems Development Corporation for the IBM System/360 in 1969. Describe as much as you can find about why it was created. It used an inverted file type of database model. Speculate (or better, find data) as to why inverted file type databases did not survive long past the 1970s.
9. ISAM continues to survive in various forms, such as MySQL's MyISAM. Explain why ISAM has survived so long.
10. Investigate the INGRES RDBMS's use of QBE. Explore the use of QBE in Microsoft Access in its query interface. Describe what parts were kept in the Access interface and what parts were not. Are there any of the missing parts of the INGRES QBE that might be helpful to Access's interface? If so, explain a missing QBE feature and how it might be useful for the Access interface.
11. We discussed a number of different database models including hierarchical, network, relational, inverted file, object, multidimensional, and NoSQL. While there has been occasional stability of the model used (like relational was used for a long time and continues to be at the core of many DBMSs), there has been a continuing drive for changes to the data model.

    Identify at least two factors that impact the change of data models underlying database systems and give examples of how these factors have affected change.

7.8.2 Projects

1. Find out more about General Motors' project (called Associative PL/I or APL[15]) that created an early system for CAD and included several database concepts. Find out what those concepts were and how they compared to Bachman's I-D-S system. Associative PL/I was presented at the AFIPS Fall Joint Computer Conference in 1966 with a paper called "APL–A language for associative data handling in PL/I," by Dodd. Find this paper and write a paper comparing APL to I-D-S and its database-related concepts. Bachman and Williams had presented a paper at the same conference two years earlier (1964) with a paper called "A general purpose programming system for random access memories." In your paper describe how the concepts presented were related and what was different. Include what concepts in APL survived to modern database systems and which did not.
2. Investigate and compare the evolution and history of DBMSs specifically to support large data warehouses. Examples include Teradata and Red Brick

15. Not to be confused with the APL programming language.

(now IBM). These DBMSs often support features such as drill-up/drill-down, summarization of data, and sometimes used de-normalized data. Their focus is on analysis rather than operational transactions and the kind of support they provide is sometimes called On-Line Analytical Processing (OLAP). Relay how these evolved and why these evolved as a separate software system from relational database systems. That is, why weren't normal relational DBMSs good enough for this purpose?

3. Many forms of database data modeling have been proposed and gotten some traction such as Chen's Entity–Relationship (ER) modeling and Hammer and McLeod's Semantic Object Model. Find more of these data models and compare their characteristics as well as their motivations. Explain why ER modeling has been successful and the difficultly of other models to replace it.
4. One could argue that DBMS features are migrated to smaller classes of computers, just like operating systems' features have been migrated to smaller classes of computing over time. Find data to show (or disprove) that DBMS features are migrated to smaller classes of computers over time. Look at features such as multi-user support, support for advanced relational features, database types (like initial support of relational), optimization techniques, analysis features, and object support. Produce a report that shows your results of this analysis.
5. One of the arguments alluded to in this chapter was that the use of databases using a network or hierarchical model were almost necessary for the time in which they were used. This was because of the computing power available and the need to tune the system to specific use cases. The chapter also alludes to the fact that this slowed the adoption of relational databases until their performance could rival that of earlier database systems. Find data to support this argument and write an in-depth paper that shows how prerelational data models were optimized to the system capabilities of the time. Also, give evidence and examples for how performance was a key issue in replacing legacy databases with the relational model.
6. MUMPS (Massachusetts General Hospital Utility Multi-Programming System) was first developed in the mid-1960s and supported hierarchical data storage as part of the language. MUMPS (sometimes called just "M") was widely adopted, particularly in the medical community. MUMPS was developed with portability as a goal and many versions were developed along with an ANSI standard. MUMPS requires a separate data schema. Describe the hierarchical data storage used by MUMPS and compare it with the structured, hierarchical database from IBM's IMS. Also explain why MUMPS

has survived so long. Compare Ruby on Rails to MUMPS and how it approached a similar integration of database functionality with the programming language.

7. During the early years of relational DBMSs, it was difficult to get sufficient performance for many applications. As a result, a market developed for database machines, which were specialized hardware devices built to house relational databases. Investigate the vendors that provided such database machines in the 1980s and 1990s. Show how they delivered better performance than was available on general purpose computers and how and when these types of database engines went out of fashion. Compare this to the more recent database machines being built as special-purpose hardware to support databases such as Oracle's Exadata database machine. Are there parallels between what happened in the 1980s and 1990s? Is it the same drivers that are driving these newer database machines?

8. In Fry and Sibley [1976] they also discuss a family of early database systems they call the "Formatted File/GIS" family. This set of systems was mostly derived from uses in the US federal government. Explore this family of systems and how it was different from other contemporary systems. The Generalized Information System (GIS) was developed as an IBM product for the System/360 and many of these systems were used by the intelligence community. Explore this family of systems and describe their varying features in a report. Include how these systems influenced later systems, particularly in the defense and intelligence community.

9. In Fry and Sibley [1976] they also discuss a family of early database systems called the "Postly/Mark IV" family. This set of systems was exemplified by the Informatics[16] company and their Mark IV product, which was the first software product to have cumulative sales over $10 million. Investigate the Mark IV product, its uses, and how its technology was different from contemporary database systems. A good reference to begin is Bauer [1996]. Write a report that analyzes the Mark IV system and explain the technology involved. Explain any unique features that might be valuable for modern-day DBMS systems.

7.9 Further Readings and Online Resources

Fry and Sibley [1976] provides a detailed history of DBMSs from their earliest beginnings that shows the evolution from various file system technologies into network,

16. Note that the term "informatics" has since become widely used in a number of other contexts since this company existed.

hierarchical, and inverted file type DBMSs. Stonebraker [1988] has a number of important papers in database history, a number of which have been cited in this chapter. Haigh [2009] details the formation of DBMSs through the 1950s and 1960s. Chamberlin et al. [1981] details the history of IBM's System R RDBMS and Harris and Nicol [2013] gives an overview of how IBM's System R evolved into IBM's SQL/DS RDBMS. Grier [2012] provides context on the formation of the relational database and information system concepts, particularly in the context of information retrieval systems. IBM's RDBMS story is further explored from the DB2 perspective in Haderle and Saracco [2013]. The story of the Informix DBMS is told by the founder in Sippl [2013]. The story of Sybase is well told by one of its founders in Epstein [2013]. The early days of the INGRES DBMS are discussed by its founder in Stonebraker [1980]. How SQL became a standard is relayed from the National Bureau of Standards (NBS) point of view in Deutsch [2013].

Early database textbooks also contain information regarding early database systems, such as Date [1975], Martin [1977], and Ullman [1980].

8 Software Futures and Overall Trends

Software has been affected by long-running trends, many of which are explored in this chapter. Software has some unique characteristics as a technology that help support these trends such as the ability to abstract and build on existing software by building an interface, building adapters to interconnect existing systems, and to encapsulate it in a virtual machine. These sort of characteristics are likely to persist and affect future software systems. In addition, the development of software has had persistent challenges such as the development of bug-free, highly available, highly scalable, secure, and safe software systems. Several of these challenges are explored here as well in the context of examples that we've seen in earlier chapters.

8.1 Overview of Software History

While software began being specific to a particular computer and of direct use only on that machine, most modern software is built and runnable on a wide variety of different computing platforms. In large part, this ability comes from the stability of lower levels of software, many of which we've discussed in earlier chapters, such as operating systems, databases, and networking. With the definition and de facto- or explicit-standardization of these lower software layers, other software can depend on a relatively stable base that doesn't radically change. When each vendor's operating system was very different from every other vendor's operating system, it was difficult to write higher level software that was portable to machines of a different type. Similarly with early networking software, beginning with many distinct, proprietary networks, it was a barrier to porting or selling software across different computing platforms. Databases also went through a similar standardization where relational databases became the norm and produced SQL as a standard query language. This stability of a relatively small number of portable operating systems, all using TCP/IP as the networking standard and relational databases, has produced the ability to create software that could be run on a wide variety of different hardware. This makes it much more appealing to develop that software (as more money can often be made and have more impact) and easier to maintain. Such stabilization has enabled other efforts, such as open-source software initiatives, to take hold. While LINUX was a key component to further enable the open-source movement by completing a full open-source programming environment, the ability to produce open-source tools for all UNIX variants was previously enabled by a relatively stable and portable UNIX operating system.[1]

A trend related to stability is the ability to successfully abstract systems into multiple layers of abstraction. While abstraction is a feature of programming languages that allows modularization and improved program structure, abstraction extends well beyond the programming language level. This ability to build a *virtual* component of a software system has been heavily used in a number of systems. A relatively simple example is a simulator. In this text, we've mentioned simulators and emulators that allow for the running of old software such as the Hercules emulator,[2] which allows one to run virtual IBM 370 hardware architecture and then install software intended for IBM 360/370 machines. This ability has become a fixture in the way that large systems are built, often using virtual machines that may be running a simulator or virtual machines. As long as the virtualized system produces a correct interface to other software, it can be replaced or enhanced without affecting other layers of the system. This virtualization has also been applied to storage software, networking software, and other many system components. There's also work on having virtual data centers that present a large number of servers, storage, and networking as a bundle that can be managed as a data center (such as failing over to another, virtual data center).

Partially because of the long-term trends of stabilization (with standardization and portability) and abstraction, software continues to become more complex and able to solve more sophisticated problems. Software will continue to become more complex and be even more integrated with existing software. While there are many continuing challenges, some of which are described later in this chapter, more complex problems will continue to be tackled with software.

One of the key concepts that this text relays is that different software domains have had different influences and patterns of development and evolution. Some

1. There were a number of variants to UNIX, but it was possible to build tools that would compile across the major varieties relatively easily. That ability was further enabled by standard interface definitions such as POSIX.

2. See http://www.hercules-390.eu/.

software domains have been heavily influenced by practical, industry needs such as databases and operating systems. At the same time, some key developments in those areas have come from academia. In the areas of databases and operating systems, research contributions such as relational database theory and virtual memory were initially inspirational. That is, while these two ideas were conceptually appealing to industry, they took some time to become commonplace due to factors of performance and cost-effectiveness. Other areas such as artificial intelligence have been largely driven by academic research until relatively recently. This identification of how different software domains have evolved differently because of the different influences and communities of development is important in predicting how new software types may be influenced and how those influences may change over time.

Throughout this text there have been opportunities to examine software listings of important software systems. The intent of this is two-fold. First, the ability to understand the actual code, context, design decisions, and methodology gives a lot of insight into how that software was developed. Secondly, the hope is that there is the ability to recognize some of these bits of code as seminal in nature. That is, they made an important contribution to how software was developed or how it functioned. This insight into how an important software innovation and contribution was made may help others find new innovations.[3]

8.2 Trends

There have been several persistent trends throughout much of software development history. Some of those trends are the following:

(1) A drive for increased programmer productivity

   As we've seen in many different sections of this text, there has been a longtime need for programmers to be more productive. In the development of compilers, there was a need to develop more software systems than the current set of programmers was capable of doing. Operating systems became successful when their benefit of increased user and programmer productivity outweighed their overhead cost. Similarly, FORTRAN was more easily accepted as a replacement for assembly language programming because it produced efficient machine code.

(2) Desire for higher-performance software

3. Though, admittedly, unique software insights are often hard to fully understand how they came about, even by the innovator.

Caching, buffering, OSs, and networking have seen a drive for higher performance helping propel their improvement. Alternatively, the initial lack of acceptable performance delayed the acceptance of many software techniques such as virtual memory, TCP/IP, and relational databases.

(3) Re-use of architectural models and patterns

Techniques such as separating data from the control mechanism, such as in the von Neumann model, have continued to be successfully applied in other software systems such as Expert Systems, MVC[4] architecture-based systems, and other systems. The re-use of these sorts of architectural ideas and patterns has helped to build more sophisticated systems on top of these tried-and-tested techniques.

(4) Increasing levels of abstraction to deal with increased problem complexity

Abstraction is the ability to build a higher-level model and use it to build software based on those higher-level concepts without having to explicitly worry about lower-level details. In that manner, we can directly build on others' work and tackle more complex problems. With the relative stability of platforms such as instruction set architecture (ISA), networking protocols, and operating systems, additional levels of abstraction are being used to support virtual machines, virtual data centers, software platforms, and other layers of abstraction. These lower layers tend to be older, more stable software.

As this trend continues, it's likely there will be more of these older layers of software that could cause issues in the overall system. Those issues can involve latent bugs in the software that are only exposed when stressed by a higher-level system, differences in assumptions between the layers of abstraction, or trying to get the underlying system to do something that it wasn't designed to do. For example, this author experienced a system performance issue that involved a large course management system which was using a file-caching solution over a load balancer to gradually move the system and all its supporting files to another data center. This system involved not only the software for the course management system but also the dependence on other software in the file-caching solution, load balancer, storage area network, and storage arrays. The response time of the system was terrible for some customers, and it was very difficult to solve even with bringing in experts from each of the companies involved. No single person or company knew enough about all the software to quickly diagnose the problem.

4. Model–view–controller (MVC) is a popular software architectural pattern often used with web-based systems that support different views for clients such as browsers and mobile devices.

The performance issues in this case involved not only the assumptions in the design of the software system but bugs in some of the underlying systems that hadn't been identified yet. It was eventually fixed by re-designing the course management software to eliminate bottlenecks, adding metrics and monitors for system performance, and fixing the bugs in the underlying software.

(5) Varied software deployment models based on hardware and technical infrastructure, particularly network characteristics

The focus of where the majority of software is developed has changed over time, often in response to how reliable and fast the data network is at that time. The other major factor has been the availability and processing speed of other computing devices, such as personal computers (PCs) and mobile devices. Software was initially written on a single, central computer that all users shared. Networks were developed that allowed these computers to share information and tasks but the networks were relatively slow, although it did support the ability to use terminals to remotely connect to those computers. Eventually, with PCs and mobile devices software was written for these other, smaller computing devices. With the advent of more pervasive and higher-speed networking, software has again tended to centralize with devices accessing these centralized services. Mobile apps are a good example of where many mobile apps depend on a centralized service in order to provide a broader experience. This cycle of centralizing software system versus distributing software to endpoints is likely to continue as the advantages of centralizing versus distributing change over time. Additionally, systems utilizing Application as a Service (AaaS) models are a good example of where centralization of the core software system outweighs the benefits of PC-based applications.[5]

(6) Backward compatibility

The ability to support old software on newer hardware and software systems it depends on is known as *backward compatibility.* If one can avoid having to re-write, re-build, re-test, and sometimes re-architect a system in order to use more modern or faster hardware or software, then they can save a lot of money, time, and effort as well as expect the system to work the way it did

5. The idea of computing as a utility has been around since the early 1960s and has recently re-emerged in the form of cloud computing. See Parkhill [1966] and John McCarthy's notion of "The Home Information Terminal," Man and Computer. Proc. int. Conf., Bordeaux, 1970, (pp. 48–57), 1972. S. Karger publisher at http://jmc.stanford.edu/articles/hoter2.html.

before. With IBM's System/360 this became more of a reality and an expectation from computer vendors. The ability to port code using higher-level programming languages also helped support backward compatibility though tweaking and testing of the ported code on the new system was still required. The design for backward compatibility is also a driver for using operating system virtualization and containerization so that the system doesn't need to be modified. The support for backward compatibility helps enable software's inertia by providing a mechanism for it to remain usable without the need to change the software.

8.3 Perpetual Challenges of Software Development

Some challenges have been more difficult than others for the software industry to solve. This section contains some of those challenges that have plagued software and are often the focus of software engineering. These include software quality, reliability, scalability, re-usability as well other areas such as software safety, availability, security, and performance. Here, we discuss quality, reliability, scalability, and re-usability.

8.3.1 Software Quality and Reliability

High software quality has long been a goal of software as any software fault costs money and time to fix. A software fault found after the software has been deployed to customers is generally the most expensive to fix. These faults can be the result of a typo, an ill-conceived design, a misunderstood requirement, a failure to catch an error during testing, or some combination. While many other engineering fields have matured to the point where quality can be effectively managed, software engineering has found that difficult. Software quality is also important for companies and their customers. One of the challenges is that software is handwritten using tools that cannot guarantee quality. That is, a new software system generally has its own requirements that are quite often different from any other software. Programmers are known to vary greatly in the quality of software they produce. Software, as noted earlier, is becoming increasingly complex, making complete testing not only impractical but not even possible. While techniques continue to be developed that may help, such as model-driven architecture and formal proofs of correctness, this is a hard problem that is unlikely to be solved any time soon.

Reliability is related to quality because if you cannot determine the faults in your software then you can't determine their impact or respond to them. Techniques of building redundant systems have helped improve reliability, as well as methodologies and tools that try to find errors in the process as soon as possible. Being able to fail over to another piece of software that does the same thing (but uses a different algorithm and different source code) requires the ability to determine there has been a failure and to be able to switch to the redundant code. See texts such as Musa [2004] and Bauer [2012, 2010] for more on how to engineer reliable software systems.[6]

8.3.2 Software System Scalability

The ability to scale systems to a large number of users or to process a lot more data has been a challenge for many software systems but has had a better outcome than quality or reliability. The development of scalable software architectures that can add components to address additional load, as well as the ability to abstract system components, has resulted in the ability to scale some systems to billions of users. For example, the ability to distribute the system to allow local handling of some of the system's functions allows the system to scale to a large number of users, such as the Internet or the phone system. However, not every system has the characteristic of being able to distribute functionality or to add a new instance of a component to meet demand. Systems that have an inherent bottleneck are usually difficult to scale, such as a system that depends on a single module to approve all transactions.

8.3.3 Software Re-usability

A continuing challenge is to be able to re-use the design and implementation of prior software systems. One of the more useful techniques to re-use design knowledge has been to encode successful design approaches as *patterns*. Both design patterns (see Gamma et al. [1995] and Rising [1998]) and architectural patterns (see Cervantes and Kazman [2016]) have helped re-use previously successful design approaches. However, there have been many other approaches over the years to re-use software that have proven less robust and long-lasting.

One example of such an approach that did not last was *software components*. In the early 1990s, there was an effort to develop prefabricated *components* that could be assembled into useful programs. Many of these were quite successful such as Microsoft's Object Linking and Embedding (OLE) and Component Object Model (COM). Though one can argue that they never became the prefab components that could be re-used as broadly as had been hoped at the time.

Software components grew out of the popularity of object-oriented programming, which has many of the same hopes of building re-usable objects. It turns out

6. Software availability is also closely related to its quality. However, the availability may also be affected by additional factors such as being overwhelmed by more usage than expected, security attacks, and other factors not directly related to the quality of the software.

to be very difficult to create generally usable and robust classes and objects. The primary issue is that if you create a general object with lots of options in how it can be used it quickly becomes impractical to test all combinations of those options. As a result, these generalized objects are likely to have untested future uses and as a result be less reliable. In addition, just creating a truly general object is hard to do. As software technologies and hardware capabilities evolve, it's often the case that these "general" object libraries become out of date as they don't support a new technology. For example, as augmented reality and virtual reality hardware became more available, many graphics software libraries did not support them. As a result, the library must be augmented to support these technologies or be replaced or re-written. Often a new library is created that supports these newer technologies and the old libraries become gradually less relevant.

An example where a software library has been successful for a long time has been FORTRAN libraries for calculations supporting numerical analysis techniques. These libraries supported floating point operations that were widely used and relatively efficient. So, there hasn't been a great need to re-write these libraries in other programming languages and many were used for decades without significant modifications. Only in the last couple of decades, as scientists and engineers are being trained in other programming languages such as MATLAB and R, have we seen these libraries be replaced or re-written in other languages.

8.4 Emerging Software Trends

Some software challenges are becoming more difficult as software systems become larger, are more complex, and are being applied to new areas and problems.

One of those challenges is security. Software is being deployed to perform more functions and to run on more devices, increasing the likelihood that the software will be attacked. One example is the proliferation of devices that are networked and controllable, such as those characterized as the Internet of Things (IoT). Many of these devices are created with a legitimate use in mind, but often are not well secured. One example was an IoT baby monitor that allowed parents to use their Wi-Fi network to connect the baby monitor to other devices (like phones) and to communicate with the baby as well as hear and see the baby respond. There have been numerous reports of these baby monitors being hacked.[7]

7. Such as one mother saying the camera was being controlled by others and other cases where the monitor was used by unknown persons to communicate with the baby. Many baby monitors have had such issues, such as the iBaby Monitor M6S, as detailed in CNET's "iBaby monitor vulnerable to hacking," March 2, 2020, by Queenie Wong.

Software systems are becoming increasingly complex and issues with those systems can be very difficult to solve. They may involve several systems interacting, each of which is very complex. The ability to diagnose the issue may require experts from each of those various systems and be a challenge to replicate, let alone solve. While the interfaces between those systems should be straightforward and help to isolate the problem, they are not always perfect and result in systems interacting in unpredictable ways that involve a number of the systems, each of which may have errors impacting the symptoms seen.

Another problem is that software systems are increasingly being built on techniques that can produce unpredictable results. An example of this is a system that is built on machine learning or pattern recognition where decisions may be made automatically based on the current dataset. Autonomous vehicles depend on data from their sensors in order to make decisions on how to control the vehicle. The data is rarely going to be the same and may contain small variants that cause the behavior to be unpredictable. While this is a contrived example, suppose that a person is wearing a bear suit and is crossing a road. How can the vehicle determine what is crossing the road and make a valid inference? In most cases, it won't matter, but if the car has to make a decision to either hit the "bear" or veer into the ditch, it might matter greatly. Systems that rely on highly variable datasets to make decisions will become increasingly hard to debug.

8.5 Other Areas of Software

This book has covered many foundational domains of software technology, with many more domains that deserve a detailed history. In particular, we haven't yet covered important software domains such as artificial intelligence, games, numerical methods, graphics, human–computer interaction, and application domains such as PC and enterprise computing. Other areas that have a broad impact on many areas of software are security and computational models.

Each of these areas have their own unique history and have influences and motivations that are different from those we have already explored earlier in this text. Application areas also have a complex history. One of the challenges is that many of these histories are interwoven, particularly in the first few decades of software history where distinctions between different types of software were not made or often not relevant. As an example of how interwoven these areas can be, Charles Bachman in a lecture describing creating the first database management system notes that there was no operating system on the computer he was using to develop a business application. In his case, the business application was the goal but along the way he needed to develop something to help manage the data and system.[8]

For example, consider security software. Software security (outside of the military and some industries) was often an afterthought. After the Morris Worm (or Internet Worm) in 1988, industries connected to the Internet at the time realized that they were vulnerable to attack. The security industry grew rapidly as those industries purchased firewalls and built more security into their systems. Another important catalyst to the development of secure systems and software was when the World Wide Web enabled companies to augment systems to enable direct sales to customers over the Internet. Quickly, companies began to experience attacks over the Internet, and this again stimulated the security software and hardware industry to develop additional responses to the evolving threats.[9] So, the security software business was stimulated by particular events, where other areas of software such as operating systems and databases had the goal of improving performance and productivity.

Numerical methods and computing were initially drivers for the purchase of computers and many of the larger computer users in the 1940s and 1950s were concerned with numerical computation such as those used in engineering, astronomy, scientific research, and weather. Early computers such as Standards Eastern Automatic Computer (SEAC) in Figure 8.1 and Standards Western Automatic Computer (SWAC) used at the US National Bureau of Standards were heavily used for numerical computation. The US space program made use of computers such as the IBM 360 in Figure 8.5 and Figure 5.10 during the Apollo program (Apollo 5 testing the lunar module launched on January 22, 1968, about a week after the photo in Figure 5.10 was taken). Supercomputers have often been driven by the needs of scientific computing, such as the IBM 7030 Stretch (the first transistorized supercomputer and designed to meet requirements formulated by Edward Teller), which was the world's fastest computer in 1961 until the CDC 6600 eclipsed that title in 1964. The IBM Stretch in Figure 8.2 was installed at Los Alamos National Laboratory for the Atomic Energy Commission.

8. See https://www.youtube.com/watch?v=iDVsNqFEkB0 for his April 16, 2002, talk at the Computer History Museum on "Assembling the Integrated Data Store."

9. An example experienced by this author was a web-based survey sent to the author by Fat Brain (a now defunct web-based book seller). Once the survey was completed, this author hit submit and got an error in response, so he hit "refresh." He then noticed that he had received two emails containing gift certificates as rewards for the survey. Curious, he reloaded the survey several more times, each time receiving a corresponding email with a new gift certificate, effectively printing money. After a few days, he received another email asking about whether he had trouble with the survey...

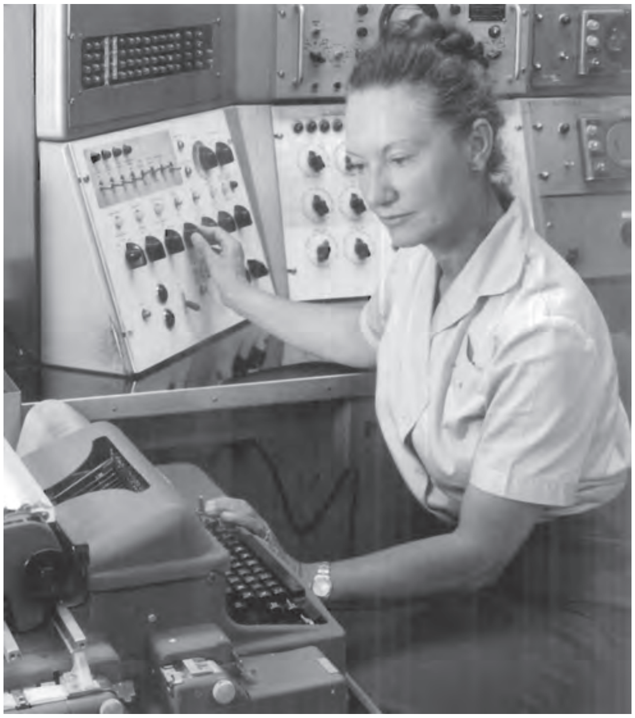


Figure 8.1 SEAC on August 14, 1959, demonstrating the HAYSTAQ program for searching chemical literature (Ethel Marden). (Source: NIST.)

Artificial intelligence (AI) software was often stimulated by the quest to push computing to new areas and has often had a research-driven agenda. AI has often had an inspirational role for what computing would become. In the summer of 1956, a seminar organized by John McCarthy was held at Dartmouth College that set a number of goals for AI and the proposal for that seminar is usually credited with coining the term "artificial intelligence." The programming language LISP was created after this conference in 1958 with the Information Processing Language (IPL) developed in 1956. Both IPL and LISP were built with a focus on symbolic processing rather than on numeric processing with the expectation they would be better languages for working on AI-type problems. The hopes and expectations around this time were gradually dashed as the solutions were more difficult than anticipated. As a result, AI experienced what has become known as an "AI winter." Failures such as the difficulties of automatic machine translation of documents between different human languages was one of the reasons, as well as the lack of progress in other areas anticipated in the 1950s. Another early success in AI was Arthur Samuel's checkers program that was able to improve its play dramatically

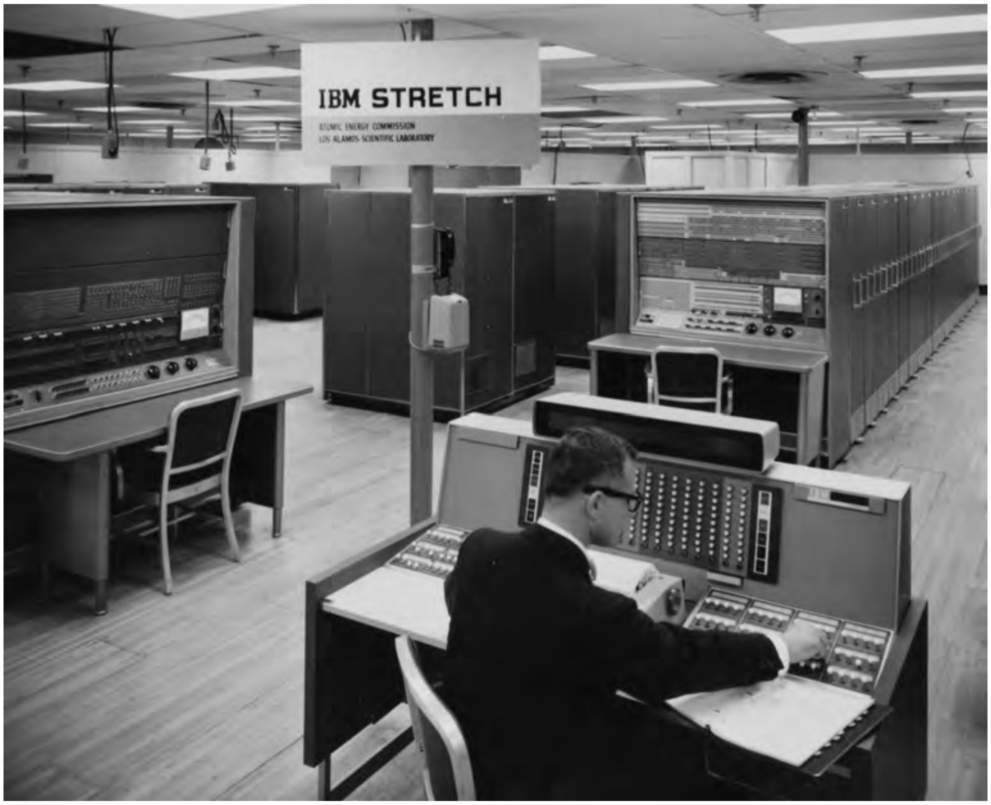


Figure 8.2 The IBM 7030 STRETCH in the late 1960s at IBM Poughkeepsie just prior to its shipment to the Los Alamos National Laboratory. (Source: Courtesy of International Business Machines Corporation, ©1968 International Business Machines Corporation.)

by using an early form of machine learning and was originally developed on the IBM 701 (see Figure 8.3 for Samuel using the IBM 7090 to run his checkers program). AI experienced a resurgence in the early 1980s with the success of expert systems. Building on the success of early expert systems such as Dendral (identified unknown organic molecules by analyzing mass spectra), expert systems became very popular and applied to a wide variety of problems. Expert systems were again overhyped and were good at a limited set of problems where if–then rules could correctly categorize and solve problems. After expert systems failed to live up to the hype, another AI winter began in the late 1980s. In the 1990s, techniques such as machine learning became successful at solving a number of problems where large datasets are available. This, combined with advances in computing speed, cost, and robotics has led to an increased expectation for AI to solve a wide variety of problems. An important example of robotic planning is Shakey, developed

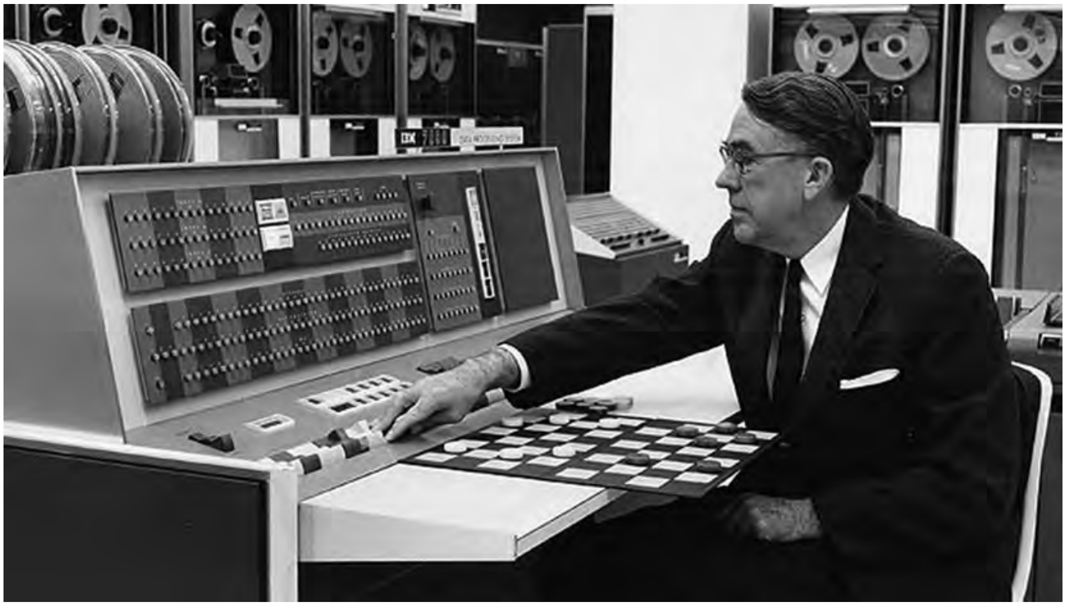


Figure 8.3 Arthur Samuel demonstrating his checkers program on an IBM 7090 on television on Feb. 24, 1956. (Source: Courtesy of International Business Machines Corporation, ©1956 International Business Machines Corporation.)

at Stanford Research International (now simply called SRI), which was able to analyze its environment and develop a plan (see Figure 8.4). While machine learning is powerful, experience with these previous AI winters suggests that some sort of expectation burst could occur again. So, AI has a rather unique history compared to many of the other software domains discussed in this text, being driven by research and by representing aspirations for what computing can become.

The use of graphics to display results and to interact with applications has long been a desire and challenge for computing systems. Systems such as the SAGE were already using graphical displays in the late 1950s. Other systems such the Sketchpad system developed by Ivan Sutherland in 1963 focused on the interaction of a user with a drawing system. The IBM 360 Model 40 (see Figure 8.5) was also used to support a drawing system using a lightpen in the mid-1960s.

Application software varies widely in what drives its creation and change over time. For example, consider Enterprise Resource Planners (ERP), which are generally large, data-oriented systems that support the back-office (and sometimes front-office) operations for companies. As you might expect, many companies have purchased this type of software in order to manage payroll, sales, employees, budgets, and the like. Initially, companies often wrote their own software for these type

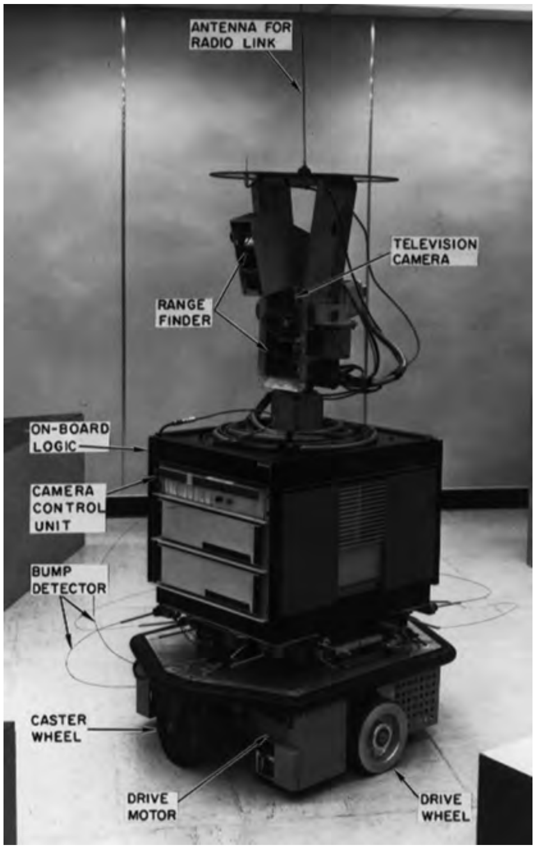


Figure 8.4 SRI's Shakey, the first mobile robot that could make decisions about how to move in its surroundings (1972). (Source: Courtesy of SRI International.)

of back-office functions, but as these business functions became more complex it was not worth writing and maintaining their own software for these functions. Vendors (including computer vendors, such as IBM) would sell computers with bundled business software. In the mid-1960s, software to support manufacturing processes was created, called material requirements planning (MRP), and was followed in the mid-1980s by manufacturing resource planning (called MRP II), which added a number of other processes that it supported. This became a large segment of the software industry, and in the 1990s these expanded to what is called enterprise resource planning (ERP), in recognition that the systems support more than the manufacturing industry. These systems have grown to become very large systems, often with a number of optional modules that a company can buy if they need them such as supply chain management, procurement, or an enterprise data warehouse. These systems have been almost entirely driven by industry needs, with a

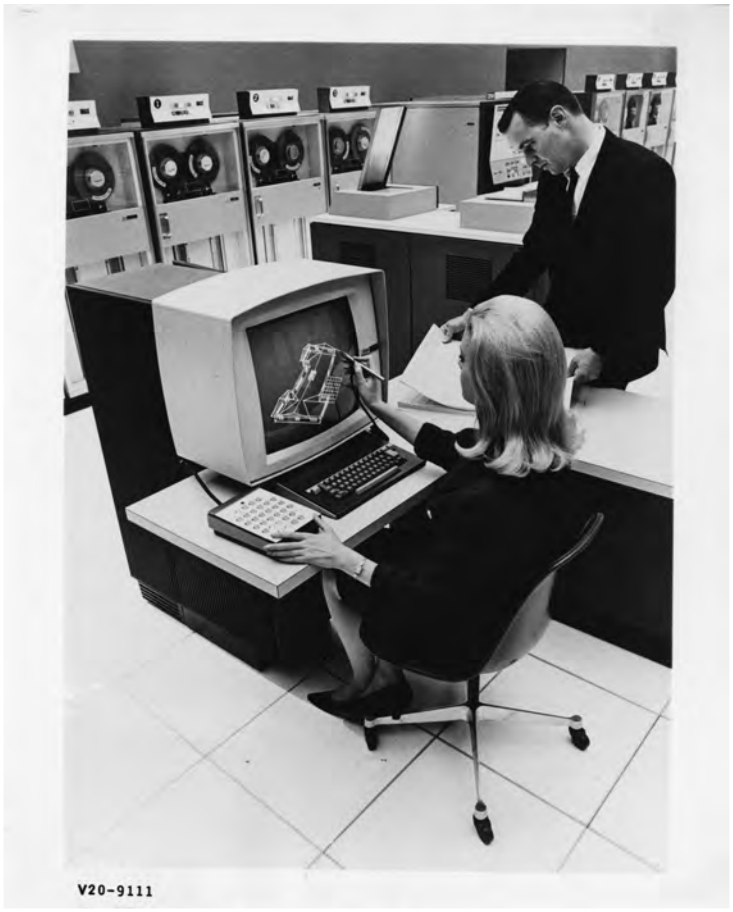


Figure 8.5 IBM System 360 Model 40 demonstrating a drawing system using a lightpen. (Source: Courtesy of International Business Machines Corporation, ©International Business Machines Corporation.)

heritage in manufacturing industries. As of this writing, ERP companies are evolving to be cloud-based vendors supporting either all ERP functions or specializing in a specific function. So, many companies are moving what was ERP functionality to multiple, cloud-based vendors that are integrated to support the business as a whole.

8.6 Software History's Relevance

Throughout this text, we've studied how software changes, what caused it to change, and how it may change going forward. Clearly, a lot of old software is still running, often embedded in modern systems. Old software is often replaced because of evolving technology, the difficultly in providing support for it, or events (think Y2K or the Internet Worm). Even after replacing the old software, the assumptions and overall design may remain the same. These underlying design assumptions can also influence new software as ideas and techniques are replicated, particularly those ideas or techniques that have worked in the past. People who are currently building software systems can better anticipate issues or opportunities by better understanding the history of the software they may use or modify.

Legacy software systems written many decades ago continue to be an issue, particularly when stressed in ways unanticipated by the original designers. A recent example is that during the COVID-19 pandemic in 2020 many states in the United States were faced with large numbers of people applying for unemployment. Many of these systems were written decades ago using COBOL on IBM mainframes, yet have continued to serve their function for the states through more normal levels of unemployment claims. Numerous states made urgent pleas for anyone who knew COBOL.[10] Even though the reports only mentioned COBOL, those systems were built with an array of other technologies from the period including file systems such as VSAM, as well as likely dependent on other IBM mainframe technologies. While many of these unemployment systems will likely be replaced after this experience, there are many other similar systems still in operation that have no urgent need to be replaced at this time. This pattern of software surviving decades is likely to continue as the effort to keep this software running is often much less than completely rewriting it using more modern tools and environments.

An even clearer example of the relevance of software history is for those who are attempting to build secure systems. It is often easier to take an existing set of software and build on top of that. That previous software may contain a number of vestigial features. While those features may have been relevant and worked perfectly when they were used, they may now provide no worthwhile function to the new software system. Furthermore, they may provide affordances to those wishing to attack the system. That is, they provide entry points where legitimate use of the no-longer-relevant feature may lead to the ability to compromise the larger system. One example of such as technology is UUCP (UNIX-to-UNIX Copy), which was a common protocol in the 1980s for communication with and between UNIX

10. Such as the report by CNN from April 8, 2020, "Wanted urgently: People who know a half century-old computer language so states can process unemployment claims," by Alicia Lee, https://www.cnn.com/2020/04/08/business/coronavirus-cobol-programmers-new-jersey-trnd/index.html. In that report, some of the states mentioned were New Jersey, Kansas, and Connecticut, all needing COBOL programmers.

systems. Other operating systems developed the ability to support UUCP, including MS-DOS and classic MAC OS, among others. Furthermore, UUCP was used in many early bulletin board systems (BBS), which, if still running, might inadvertently still support it. So, versions of UUCP may still be running on some UNIX and Linux systems and allow an entry point that might be unexpected to the owners of those systems.

While much of the software that has been developed has had a short lifespan, some software has persisted well beyond the expectations of the original authors. Understanding why some software has persisted and how software concepts and domains originate and evolve helps to understand how future software will evolve. Such understanding helps software developers predict and avoid pitfalls for unsuccessful software and to attempt to replicate the success of long-lived software systems.

While this text has provided a technical history of well-established software domains, it is hoped that this text can serve as a foundation for documenting further software domains and the implications of software including the societal, ethical, and global impacts of software. Additionally, this text has not focused on the people involved, but software is written by people and many people have contributed to advances in software. So, it is hoped that this text can help provide technical context for the contributions of individuals and, as is so often the case in software, teams.

8.7 Exercises and Projects

8.7.1 Exercises

1. Graphics libraries have come and gone over the years. One example is the Programmers' Hierarchical Interactive Graphics System (PHIGS). PHIGS had a standard application programmer interface (API) across a number of computing platforms that developed libraries to support PHIGS, such as IBM, SUN, and DEC. Explain why PHIGs was replaced by other standards.

2. Investigate the term *software factory*. Much of the intent of the software factory was to make the process of creating software as dependable and reliable as a factory making a particular product. Find out how, why, and when software factories failed.

3. Explain why libraries of functions and procedures are difficult to make general enough to be used in a very broad array of applications over an extended period of time.

4. Investigate where early uses of the *Flyweight* design pattern were successful (see Gamma et al. [1995, pp. 195–206]) and explain why this is a useful design pattern.
5. Investigate the Telnet protocol and explain its current uses and its security vulnerabilities.
6. The BAAN ERP was a popular ERP system in the 1980s and 1990s. Find out what happened to BAAN and whether is it still supported and run by some companies.
7. Determine if the current, stable release of Ubuntu Linux supports UUCP. If so, is it turned on by default? Describe how having UUCP turned on might be a security vulnerability by investigating the UUCP feature and how it could be used.
8. Find the paper that discusses the General Problem Solver (GPS) by Newell and Simon.[11] The GPS was a highly influential system and introduced several key ideas. From the paper, identify one of the key ideas that continues to be used for some AI systems. Find and quote that section of the paper where the idea is described.
9. With so much software being created, only a portion of it becomes entrenched. Choose a particular software system. Explain at least three factors that increase the likelihood of this particular piece of software becoming entrenched and being in continued use for decades. Explain why each of your three factors increases the likelihood of the software becoming entrenched.
10. Andy Grove (Intel) was known to complain that Intel wouldn't get the proper credit for a new processor's enhanced performance as it was often absorbed by new versions of Microsoft Windows. This became known as *Andy and Bill's Law*, where the enhanced CPU performance that Andy created, Bill taketh away. Would you expect to see this same dynamic in server and enterprise software? Why or why not?
11. One could argue (as does Charette [2020, p. 30]) that the IoT is likely to create a new embedded base of software running on these IoT devices that will become legacy systems and difficult to replace. What aspects of the IoT makes this likely? Do you believe we're likely to have future problems with legacy IoT software? Explain.

11. "A variety of intelligent learning in a general problem solver," A. Newell, J. Shaw, and H. Simon, The RAND Corporation, P-1742, July 6, 1959.

12. The US Defense Advanced Projects Agency (DARPA) has funded a program called Building Resource Adaptive Software Systems (BRASS) since 2015. See https://www.darpa.mil/program/building-resource-adaptive-software-systems. This program's goal is to create software systems that have useful lifespans of 100 or more years by being resistant to changes in the environment. Investigate the BRASS program and summarize its approach in a paragraph or two. List the fundamental challenges of this program. Do you believe this program will be successful for any of the software domains they've identified? Why or why not?

8.7.2 Projects

1. The IoT has produced a demand for new software not only for the devices themselves but also for software infrastructure to support those devices. Investigate how the use of older software has sometimes made these devices less secure. Give examples of IoT devices where its software was derived from previously existing software and how the change in usage violated the previously existing software's assumptions and helped to cause security vulnerabilities in the IoT devices.

2. Many software providers have tried to provide "platforms" where developers can more easily develop applications that work with their application. Examples are systems such as those at Salesforce.com, Facebook.com, and many others. Argue based on earlier platform-like systems whether these are likely to be a stable platform for decades to come or not. If so, explain how these could evolve to be a stable platform. If not, explain how these may be replaced by other technologies.

3. The problem of voice recognition has long been a challenge to develop reliable software that could understand spoken human language. One of the earliest examples was IBM's "shoe-box" system introduced in 1962 that could recognize a limited set of words (16), see Figure 8.6. The IBM Shoebox was developed by William Dersch and was able to respond to commands to do simple arithmetic. Many other systems have been created since that time and gradually improvements have accumulated so that systems such as Amazon's Alexa and Apple's Siri are quite usable. Explore these efforts and document the projects, their relationships, failures, and key technologies that have persisted to modern voice recognition systems. Document this as a report.

4. Programmer efficiency has been a long-desired aspect and driver for development of programming languages and related tools. Investigate the current set of technologies that promise significant increases in programmer

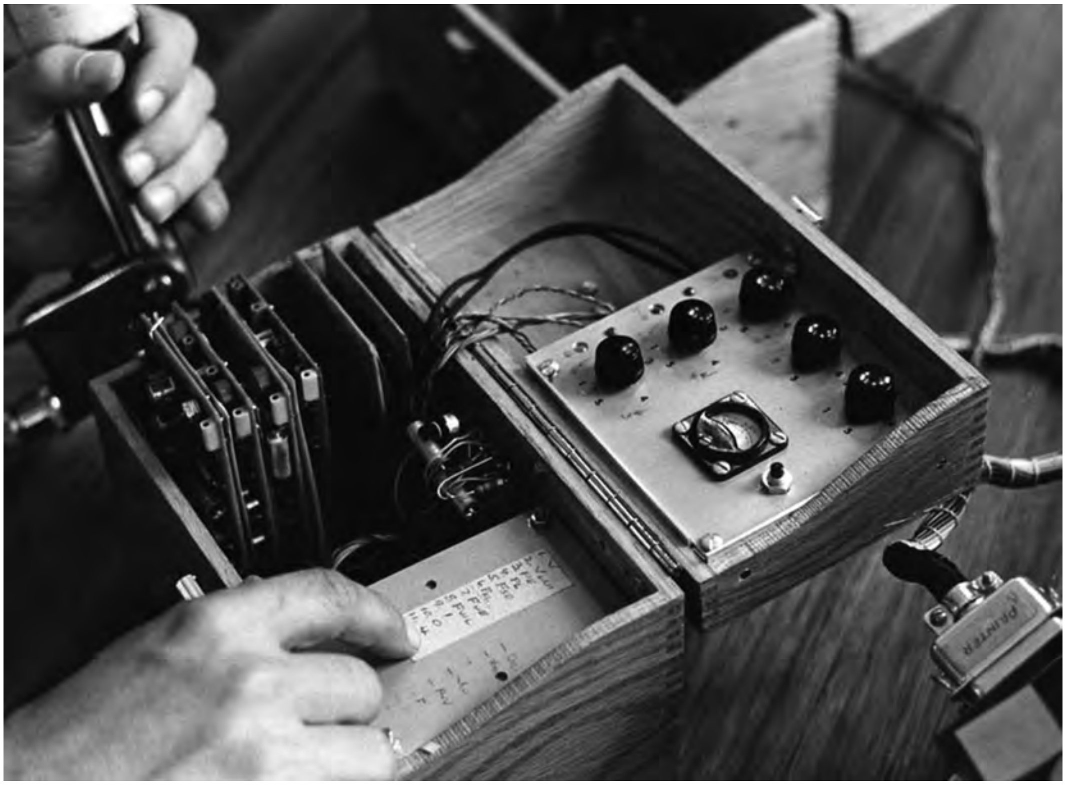


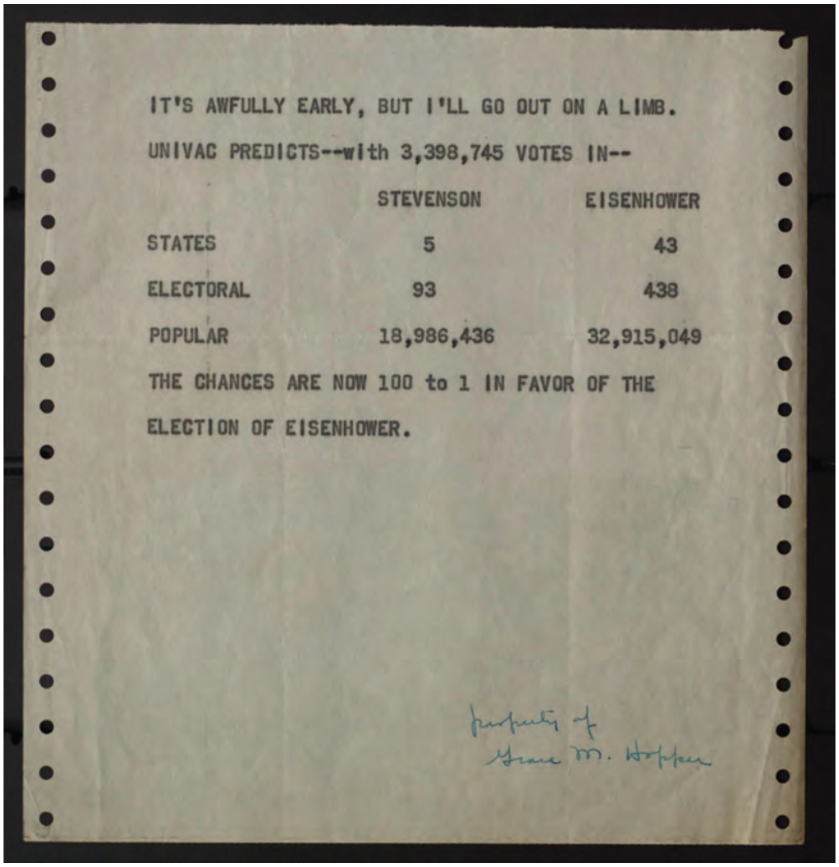
Figure 8.6 IBM's Shoebox could do simple arithmetic in 1962 in response to spoken commands. (Source: Courtesy of International Business Machines Corporation, ©1962 International Business Machines Corporation.)

productivity and evaluate them based on their ability to succeed. For the technology that you identify as most likely to succeed, identify why it is distinct from the others and what factors will help it succeed.

5. With the increasing complexity of software systems, there is an increasing possibility of *emergent behavior*. That is, behavior that is not expected or explicitly designed into the system. Since this behavior was not a requirement, it is usually viewed as a bug or software fault. Explore cases where such behavior has caused system failures. Find a case where this unexpected behavior has been beneficial.
6. Research and evaluate current approaches to increase productivity of software development. Evaluate the approaches based on their likelihood of increasing the productivity of individual programmers and of decreasing the time frame to deliver a complex application. Does the approach affect other

aspects of the software such as its quality and reliability? Based on previous approaches to improved productivity, evaluate each approach's chances of surviving 5, 10, and 20 years into the future as viable techniques.

7. Find the June 15, 1956, paper by Allen Newell and Herb Simon called "The logic theory machine—A complex information processing system." This paper generated a lot of hope that AI could reason and solve general problems, as well as further work by Newell and Simon on systems such as the General Problem Solver (GPS). The code contained in this paper was hoped to be able to solve logic problems. Explain why these techniques in the end proved insufficient to solve general problems but were able to generate a lot of hope.



```
IT'S AWFULLY EARLY, BUT I'LL GO OUT ON A LIMB.
UNIVAC PREDICTS--with 3,398,745 VOTES IN--
                          STEVENSON          EISENHOWER
STATES                        5                   43
ELECTORAL                    93                  438
POPULAR                  18,986,436          32,915,049
THE CHANCES ARE NOW 100 to 1 IN FAVOR OF THE
ELECTION OF EISENHOWER.
```

property of
Grace M. Hopper

Figure 8.7 Grace Murray Hopper's original printout from the UNIVAC's accurate prediction of Eisenhower's 1952 win. (Source: Courtesy of Grace Murray Hopper Collection, Archives Center, National Museum of American History, Smithsonian Institution.)

8. For the 1952 US presidential election, CBS Television News and Remington Rand used the relatively new UNIVAC computer to help process and predict the election. Figure 8.7 shows the printout from the UNIVAC's early prediction of the outcome at around 8:30PM Eastern time election night. CBS News was reluctant to air this result (which turned out to be accurate within 1%) and didn't air what was viewed as an unlikely prediction. CBS had to admit that it had an accurate prediction from the UNIVAC hours before but had not aired it at the time. The software was written Max Woodbury, who had been blacklisted as pro-Communist and had to work on this covertly. Additionally, this success also stimulated the public's interest and expectations for computers. See if you can find the actual software used to predict the 1952 election and analyze the techniques used to predict the election. Whether or not you find the software, analyze the shifts in software techniques used for presidential election prediction that has been the norm since 1956 after UNIVAC's successful prediction in 1952. Document techniques that have persisted as well as those that have been abandoned.

8.8 Further Readings and Online Resources

Further information is provided via the accompanying website: https://software-history.net/. That site includes larger software source code examples as well as numerous links to relevant videos and resources. Books on AI history include Nilsson [2010], McCorduck [1979], and Kurzweil [1990]. November [2012] covers biomedical computing's history. Halvorson [2020] and Campbell-Kelly [2003] give insights on PC software history. A general book with a collection of well-written articles on a wide variety of computing topics including software history is Ralston and Reilly [1983].

A Appendix—Source Code

This appendix includes some longer samples of source code for systems mentioned in the text.

A.1 UNIX Pipe.c

This source code is referred to in Chapter 3, Section 3.3, Operating Systems. Note that this source code was released as a fee free license by Caldera Software in 2002 (see http://www.tuhs.org/Archive/Caldera-license.pdf) and is "Copyright(C) Caldera International Inc. 2001-2002. All rights reserved." This version comes from Bell Labs Research Version 7 in 1979 and is included without any modification. This version was extracted from http://minnie.tuhs.org/cgi-bin/utree.pl on October 14, 2014, as the file *sys/pipe.c*. Other historical UNIX source code can be found on that site as well as others such as http://v6.cuzuco.com/v6.pdf or at http://www.tamacom.com/tour/kernel/unix/.

UNIX pipes were initially included in Bell Labs Research Version 6 of UNIX and part of a larger re-write of UNIX to regularize the commands and system in the C programming language (except for a very small, machine-specific bit of required code). See Lions [1976a, 1976b, 1977, 1996] for the Bell Labs 6*th* edition of the pipes implementation in C and its description.

One of the hallmarks of early UNIX System code was the simplicity of the source code. The C programming language is a relatively small language that is able to be efficient by being close to the system. The initial code base was also written by a small set of programmers and shows a regularity that helps in understanding the code. Within Bell Labs, UNIX was taught (including to this author) by directly reviewing the UNIX source code and understanding how it worked. Pipes, as initially proposed by Doug McIlroy in 1964,[1] were key to the simplification of the user interface of UNIX. With pipes, UNIX commands would default to *stdin* and *stdout*, making it easy to string them together to process more complex operations. As a

1. See http://doc.cat-v.org/unix/pipes/ for a copy of McIlroy's 1964 description of the vision for pipes.

result, many UNIX commands were good at only one thing so that they could be combined with other tools in order to perform more complex operations. Several of these tools were simple programming languages that were tuned to a particular task such as *eqn* (equation formatting), *lex* (lexical analysis), *yacc* (parsing, Yet Another Compiler Compiler), *awk* (a text pattern-matching language named after its authors: Aho, Weinberger, and Kernighan), and *grep* (general regular expression parsing).

```
#include "../h/param.h"
#include "../h/systm.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/inode.h"
#include "../h/file.h"
#include "../h/reg.h"

/*
 * Max allowable buffering per pipe.
 * This is also the max size of the
 * file created to implement the pipe.
 * If this size is bigger than 5120,
 * pipes will be implemented with large
 * files, which is probably not good.
 */
#define PIPSIZ  4096

/*
 * The sys-pipe entry.
 * Allocate an inode on the root device.
 * Allocate 2 file structures.
 * Put it all together with flags.
 */
pipe()
{
        register struct inode *ip;
        register struct file *rf, *wf;
        int r;

        ip = ialloc(pipedev);
        if(ip == NULL)
                return;
        rf = falloc();
        if(rf == NULL) {
                iput(ip);
                return;
	}
	r = u.u_r.r_val1;
	wf = falloc();
	if(wf == NULL) {
		rf->f_count = 0;
		u.u_ofile[r] = NULL;
		iput(ip);
		return;
	}
	u.u_r.r_val2 = u.u_r.r_val1;
	u.u_r.r_val1 = r;
	wf->f_flag = FWRITE|FPIPE;
	wf->f_inode = ip;
	rf->f_flag = FREAD|FPIPE;
	rf->f_inode = ip;
	ip->i_count = 2;
	ip->i_mode = IFREG;
	ip->i_flag = IACC|IUPD|ICHG;
}
/*
 * Read call directed to a pipe.
 */
readp(fp)
register struct file *fp;
{
	register struct inode *ip;

	ip = fp->f_inode;

loop:
	/*
	 * Very conservative locking.
	 */
	plock(ip);
	/*
	 * If nothing in the pipe, wait.
	 */
	if (ip->i_size == 0) {
		/*
		 * If there are not both reader and
		 * writer active, return without
		 * satisfying read.
		 */
		prele(ip);
		if(ip->i_count < 2)
```

```
                        return;
                ip->i_mode |= IREAD;
                sleep((caddr_t)ip+2, PPIPE);
                goto loop;
        }
        /*
         * Read and return
         */
        u.u_offset = fp->f_un.f_offset;
        readi(ip);
        fp->f_un.f_offset = u.u_offset;
        /*
         * If reader has caught up with writer, reset
         * offset and size to 0.
         */
        if (fp->f_un.f_offset == ip->i_size) {
                fp->f_un.f_offset = 0;
                ip->i_size = 0;
                if(ip->i_mode & IWRITE) {
                        ip->i_mode &= ~IWRITE;
                        wakeup((caddr_t)ip+1);
                }
        }
        prele(ip);
}
/*
 * Write call directed to a pipe.
 */
writep(fp)
register struct file *fp;
{
        register c;
        register struct inode *ip;

        ip = fp->f_inode;
        c = u.u_count;

loop:
        /*
         * If all done, return.
         */
        plock(ip);
        if(c == 0) {
                prele(ip);
                u.u_count = 0;
                return;
        }
        /*
         * If there are not both read and
         * write sides of the pipe active,
         * return error and signal too.
         */
        if(ip->i_count < 2) {
                prele(ip);
                u.u_error = EPIPE;
                psignal(u.u_procp, SIGPIPE);
                return;
        }
        /*
         * If the pipe is full,
         * wait for reads to deplete
         * and truncate it.
         */

        if(ip->i_size >= PIPSIZ) {
                ip->i_mode |= IWRITE;
                prele(ip);
                sleep((caddr_t)ip+1, PPIPE);
                goto loop;
        }

        /*
         * Write what is possible and
         * loop back.
         * If writing less than PIPSIZ, it always goes.
         * One can therefore get a file > PIPSIZ if write
         * sizes do not divide PIPSIZ.
         */
        u.u_offset = ip->i_size;
        u.u_count = min((unsigned)c, (unsigned)PIPSIZ);
        c -= u.u_count;
        writei(ip);
        prele(ip);
        if(ip->i_mode&IREAD) {
                ip->i_mode &= ~IREAD;
                wakeup((caddr_t)ip+2);
        }
        goto loop;
}
/*
```

```
 * Lock a pipe.
 * If its already locked,
 * set the WANT bit and sleep.
 */
plock(ip)
register struct inode *ip;
{
        while(ip->i_flag&ILOCK) {
                ip->i_flag |= IWANT;
                sleep((caddr_t)ip, PINOD);
        }
        ip->i_flag |= ILOCK;
}
/*
 * Unlock a pipe.
 * If WANT bit is on,
 * wakeup.
 * This routine is also used
 * to unlock inodes in general.
 */
prele(ip)
register struct inode *ip;
{
        ip->i_flag &= ~ILOCK;
        if(ip->i_flag&IWANT) {
                ip->i_flag &= ~IWANT;
                wakeup((caddr_t)ip);
        }
}
```

Listing A.1 pipe.c—Pipe source code from Bell Labs UNIX Research Version 7.

A.2 System R Where Clause Code

The following code is from IBM's System R and is a subset of the PL/I code that processes WHERE clauses for SQL queries. This is an early implementation of SQL optimization and was critical to the success of relational databases. While relational databases had a much easier to use and understand interface (tuples, tables, and matching based on values), their initial performance was much slower than legacy database options available at the time. As a result, being able to improve the performance was critical to the adoption of relational databases.

The file is called XWHERE.PLIOPT and this snip-it was obtained from the Computer History Museum in line with their agreement with IBM. The *PLIOPT* file extension indicates this is a file containing source code for the PL/I Optimizing

Compiler. The file contains line numbers (XWH*nnnnn*) and you'll also note it contains a few non-ASCII characters, such as ¬. One will also note terms such *sargable* (Search ARGument ABLE) in the code (such as at line 61, XWH01840) , which indicates that the condition in the query can take advantage of an index and thereby have a good chance of improving performance. The file also indicates the selective processing of *AND* and *OR* nodes in an attempt to reduce the number of tuples returned in the intermediate result set. While modern relational database SQL optimizations are much more sophisticated today, these methods (using indexes and trying to reduce the size of the tuple set) are still at the core of RDBMS SQL optimization methods.

```
 1        /* 04/13/77 PPG MMA */                                                XWH00010
 2      /* 5/4/77 DON'T PRINT %INCLUDES */                                      XWH00020
 3                                                                              XWH00030
 4        /* XWHERE IS THE PROCEDURE WHICH EXAMINES THE WHERE PREDICATE         XWH00040
 5              TREE. ON THE FIRST PASS IT COLLECTS COLUMN NAMES, PUSHES        XWH00050
 6              NOT NODES DOWN THROUGH BOOLEAN NODES AND                        XWH00060
 7              (EVENTUALLY) REMOVES NOTNODES IF ALL PREDICATES BELOW THEM      XWH00070
 8              ARE SARGABLE (COMPATIBLE WITH THE 'COL COMPOP LIT' FORMAT).     XWH00080
 9              ALSO ON THE FIRST PASS, IF ONLY ONE SIDE OF A PREDICATE HAS     XWH00090
10              A COLUMN, THEN WE WILL PUT IT IN THE LHS OF THE RELATIONAL      XWH00100
11              OPERATOR AND ADJUST THAT OPERATOR ACCORDINGLY. (> BECOMES       XWH00110
12              <, ETC.). ON THE SECOND PASS, XWHERE PROPAGATES DATA TYPES      XWH00120
13              AND SPECLENS IN THE PREDICATES, TESTS FOR DNF STATE, AND        XWH00130
14              FINDS BT'S AND ENTERS THEM INTO BTARRAY.  A BT IS A             XWH00140
15              PREDICATE OR A SUBTREE HEADED BY AN ORNODE AND MUST BE          XWH00150
16              THE WHERE TREE ROOT OR MUST BE CONNECTED TO THE ROOT ONLY       XWH00160
17              BY ANDNODES AND MUST BE SARGABLE.                               XWH00170
18              A PREDICATE BT MAY BE REPLACED BY AN ACCESS                     XWH00180
19              PATH SELECTION.  A BT THAT IS IN DNF MAY BE                     XWH00190
20              REPLACED BY SEARCH ARGUMENTS.  A BT THAT IS NOT IN              XWH00200
21              DNF MAY SOME DAY BE PUT INTO DNF AND REPLACED BY SEARCH         XWH00210
22              ARGUMENTS.                                                      XWH00220
23                                                                              XWH00230
24              XWHERE MARKS THE NODES AS FOLLOWS:                              XWH00240
25              BOOLEAN P1 INDEX OF PARENT NODE                                 XWH00250
26                      P2 NUMBER OF NODES BELOW, INCLUDING THIS ONE            XWH00260
27              NOTNODE P1 INDEX OF PARENT NODE                                 XWH00270
28              PREDICATE P1 1 IF SARGABLE, ELSE 0                              XWH00280
29                        P2 NUMBER OF NODES                                    XWH00290
30                                                                              XWH00300
31              XWHERE CALLS XEXPRTR TO WALK OPERAND EXPRESSION TREES.          XWH00310
32              AFTER THE FIRSTPASS, XWHERE NEEDS COLFLAG AND SARGABLE.         XWH00320
33              ALL RETURN PARAMETERS ARE NEEDED AFTER PASS 2                   XWH00330
34                                                                              XWH00340
35              DNF STATE IS A FUNCTION OF THE AND/OR/NOT SEQUENCE */           XWH00350
36                                                                              XWH00360
37 XWHERE:PROC(NODEIND,PARIND,OPPTR,FIRSTPASS,ANDONLY,SARGABLE,DNFFLAG,         XWH00370
38      ORCHILD,NUMNODES,STACKINDS,BQPTR,CHKMODE,RUNTIMEMODE,                   XWH00380
39      CURSEXECMODE,AUTHID) RECURSIVE REORDER OPTIONS (REENTRANT);             XWH00390
40                                                                              XWH00400
41
42 .
43 .
44        IF ¬FIRSTPASS THEN                                                    XWH01670
45           IF P4=ANDCODE THEN NEWANDONLY=ANDONLY;                             XWH01680
46           ELSE NEWANDONLY='0'B;                                              XWH01690
47         CALL XWHERE(P3,NODEIND,OPPTR,FIRSTPASS,NEWANDONLY,LEFTSARGABLE,      XWH01700
48           LEFTDNF,LORCHILD,NLEFTNODES,STACKINDS,BQPTR,CHKMODE,               XWH01710
49            RUNTIMEMODE,CURSEXECMODE,AUTHID);                                 XWH01720
50         TESTCODE;                                                            XWH01730
51         IF OPTLEVEL > 2 THEN                                                 XWH01740
52           DISPLAY('RETURNED_FROM_XWHERE_CALL_ON_LHS_WITH_LSARG='             XWH01750
53                  ||LEFTSARGABLE);                                            XWH01760
54         CALL XWHERE(P5,NODEIND,OPPTR,FIRSTPASS,NEWANDONLY,RIGHTSARGABLE,     XWH01770
55            RIGHTDNF,RORCHILD,NRIGHTNODES,STACKINDS,BQPTR,CHKMODE,            XWH01780
56             RUNTIMEMODE,CURSEXECMODE,AUTHID);                                XWH01790
57         TESTCODE;                                                            XWH01800
58         IF OPTLEVEL> 2 THEN                                                  XWH01810
59           DISPLAY('RETURNED_FROM_XWHERE_ON_RHS_WITH_RSARG='||                XWH01820
60                  RIGHTSARGABLE);                                             XWH01830
61         SARGABLE=LEFTSARGABLE & RIGHTSARGABLE;                               XWH01840
62         IF ¬FIRSTPASS THEN                                                   XWH01850
63           DO;                                                                XWH01860
64             P2,                                                              XWH01870
65             NUMNODES = NLEFTNODES + NRIGHTNODES + 1;                         XWH01880
66             IF NLEFTNODES > NRIGHTNODES THEN                                 XWH01890
```

```
            DO;                                                         XWH01900
                                                                        XWH01910
              /* INTERCHANGE LEFT AND RIGHT TO PUT SIMPLER PREDICATES   XWH01920
                    ON LEFT*/                                           XWH01930
                                                                        XWH01940
              TEMPIND = P3;                                             XWH01950
              P3 = P5;                                                  XWH01960
              P5 = TEMPIND;                                             XWH01970
            END /* OF INTERCHANGE */;                                   XWH01980
          DNFFLAG=LEFTDNF & RIGHTDNF;                                   XWH01990
          IF P4=ANDCODE THEN DNFFLAG=DNFFLAG & ¬LORCHILD & ¬RORCHILD; XWH02000
          IF P4=ORCODE THEN ORCHILD='1'B;                               XWH02010
          ELSE ORCHILD='0'B;                                            XWH02020
          IF P4=ORCODE & ANDONLY & SARGABLE THEN                        XWH02030
                          /* ENTER INTO BTARRAY */                      XWH02040
            CALL XFILBT(NODEIND,PARIND,OPPTR,DNFFLAG,ORTREEKIND,        XWH02050
              '0'B,    /* CORRELATION NOT APPLICABLE */                 XWH02060
              NT,STACKINDS,BQPTR,BTTABIND);                             XWH02070
        END;    /* ¬FIRSTPASS */                                        XWH02080
      RETURN;                                                           XWH02090
    END;   /* BOOLEANNODE */                                            XWH02100
                                                                        XWH02110
  ELSE IF NT=NOTNODE THEN                                               XWH02120
    DO;                                                                 XWH02130
      IF OPTLEVEL>2 THEN DISPLAY('XWHERE_FOUND_NOTNODE');               XWH02140
/* TEST */                                                              XWH02150
  IF P5=0 THEN                                                          XWH02160
    DO;                                                                 XWH02170
      CALL XSYSTEM(PNAME,-101,-901,'');                                 XWH02180
      GOTO QUIT;                                                        XWH02190
    END;                                                                XWH02200
/* ENDTEST */
 .
 .
 .
```

Listing A.2 System R SQL WHERE clause optimization code, XWHERE.PLIOPT.

Bibliography

J. Abbate. 2000. *Inventing the Internet*. Inside Technology Series. The MIT Press. DOI: http://hdl.handle.net/2027/spo.3310410.0003.321.

S. B. Adams and O. R. Butler. 1999. *Manufacturing the Future: A History of Western Electric.* Cambridge University Press.

A. V. Aho and J. D. Ullman. 1972. *The Theory of Parsing, Translation and Compiling, Volume 1: Parsing*. Prentice-Hall.

A. V. Aho and J. D. Ullman. 1973. *The Theory of Parsing, Translation and Compiling, Volume 2: Compiling.* Prentice-Hall.

A. V. Aho, J. E. Hopcroft, and J. D. Ullman. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley.

A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd. ed.). Addison-Wesley.

J. Alderman, D. Spicer, and M. Richards. 2007. *Core Memory: A Visual Survey of Vintage Computers.* Chronicle Books.

C. Alexander, S. Ishikawa, and M. Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press.

R. A. Allan. 2001. *A History of the Personal Computer: The People and the Technology.* Allan Publishing.

S. Amarel. 1968. On representations of problems of reasoning about actions. In D. Michie (Ed.), *Machine Intelligence-3*. Elsevier/North-Holland, 131–171.

J. A. Anderson and E. Rosenfeld (Eds.). 1998. *Talking Nets: An Oral History of Neural Networks.* The MIT Press. DOI: https://doi/org/10.1109/TNN.1998.712193.

R. V. Andree. 1958. *Programming the IBM 650 Magnetic Drum Computer and Data-Processing Machine.* Holt, Rinehart, and Winston.

E. G. Andrews. January 1982a. Use of the relay digital calculator. In *IEEE Ann. Hist. Comput.* 4, 1, 5–13.

E. G. Andrews. January 1982b. Telephone switching and the early Bell Laboratories computers. In *IEEE Ann. Hist. Comput.* 4, 1, 13–19.

B. Arden, B. Galler, and R. Graham. April 1965. *The Michigan Algorithm Decoder (MAD) Manual.* University of Michigan.

W. B. Arthur. 2009. *The Nature of Technology: What It Is and How It Evolves.* Free Press.

W. R. Ashby. 1956. *An Introduction to Cybernetics.* Chapman & Hall.

W. F. Aspray, Jr. 1980. *From Mathematical Constructivity to Computer Science: Alan Turing, John Von Neumann, and the Origins of Computer Science in Mathematical Logic.* PhD dissertation. University of Wisconsin-Madison.

W. F. Aspray, Jr. 1990a. *John Von Neumann and the Origins of Modern Computing*. The MIT Press.

W. F. Aspray, Jr. 1990b. *Computing Before Computers.* Iowa State University Press.

Association for Computing Machinery. 1987. *ACM Turing Award Lectures: The First Twenty Years, 1966–1985.* ACM Press, Addison-Wesley.

AT&T Bell Telephone Laboratories. 1987a. *UNIX System, Readings and Applications*. Vol. 1. Prentice-Hall.

AT&T Bell Telephone Laboratories. 1987b. *UNIX System, Readings and Applications*. Vol. 2. Prentice-Hall.

S. Augarten. 1984. *Bit by Bit: An Illustrated History of Computers.* Ticknor & Fields. Also available online at https://ds-wordpress.haverford.edu/bitbybit/.

C. Babbage. 1864. *Passages from the Life of a Philosopher.* Longman, Green, Longman, Roberts & Green, London. DOI: https://doi.org/10.1017/CBO9781139103671.

H. P. Babbage (Ed.). 1889. *Babbage's Calculating Engines: A Collection of Papers*. E. & F. N. Spon, 1889. Reprinted by Tomash.

M. J. Bach. 1986. *The Design of the UNIX Operating System.* Pearson Education.

B. Bagnall. 2005. *On the Edge: The Spectacular Rise and Fall of Commodore.* Variant Press.

Ballistic Research Laboratories. September 1949. *Preparation of Problems for the BRL Calculating Machines*. Ballistic Research Laboratories, Aberdeen Proving Ground, Technical Note No. 104.

T. Bardini. 2000. *Bootstrapping, Douglas Englebart, Coevolution, and the Origins of Personal Computing.* Stanford University Press.

A. Barr and E. A. Feigenbaum (Eds.). 1981. *The Handbook of Artificial Intelligence.* Volumes I and II. William Kaufmann.

A. Barr, P. R. Cohen, and E. A. Feigenbaum (Eds.). 1989. *The Handbook of Artificial Intelligence.* Volume IV. Addison-Wesley.

W. A. Barrett and J. D. Couch. 1979. *Compiler Construction: Theory and Practice.* Science Research Associates.

D. W. Barron, J. N. Buxton, D. F. Hartley, E. Nixon, and C. Strachey. 1963. The main features of CPL. *Comput. J.* 6, 2, 134–143. DOI: https://doi.org/10.1093/comjnl/6.2.134.

G. Basalla. 1988. *The Evolution of Technology.* Cambridge University Press.

C. J. Bashe, W. Buchholz, G. V. Hawkins, J. J. Ingram, and N. Rochester. September 1981. The architecture of IBM's early computers. *IBM J. Res. Dev.* 25, 5, 363–376. DOI: https://doi.org/10.1147/rd.255.0363.

C. J. Bashe, L. R. Johnson, J. H. Palmer, and E. W. Pugh. 1985. *IBM's Early Computers*. The MIT Press.

W. F. Bauer. April–June 1996. Informatics: An early software company. *IEEE Ann. Hist. Comput.* 18, 2, 70–76.

E. Bauer. 2010. *Design for Reliability: Information and Computer-Based Systems.* Wiley-IEEE Press. DOI: https://doi/org/10.1002/9781118075104.

E. Bauer. 2012. *Reliability and Availability of Cloud Computing.* Wiley-IEEE Press.

C. Baum. 1981. *The System Builders, The Story of SDC*. System Development Corporation.

C. Beeler. February/March 2009. All-optical computing and all-optical networks are dead. *ACM Queue* 7, 3, 10–11. DOI: https://doi.org/10.1145/1530818.1530830.

G. Bell. January 2008a. Bell's Law for the birth and death of computer classes. *Commun. ACM* 51, 1, 86–94. DOI: https://doi.acm.org/10.1145/1327452.1327453.

G. Bell. Fall 2008b. Bell's Law for the birth and death of computer classes: A theory of the computer's evolution. *IEEE Solid-State Circuits Soc. Newsl*. 13, 4, 8–19. DOI: https://doi.org/10.1109/N-SSC.2008.4785818.

Bell Telephone Laboratories. 1977. *Engineering and Operations in the Bell System.* Bell Laboratories.

J. Bentley. 1986. *Programming Pearls.* Addison-Wesley.

P. Berger and T. Luckmann. 1966. *The Social Construction of Reality.* Doubleday.

T. J. Bergin (Ed.). September 2000. *50 Years of Army Computing, From ENIAC to MSRC*. A Record of a Symposium and Celebration, November 13 and 14, 1996. Army Research Lab.

T. J. Bergin. May 2007. A history of the history of programming languages. *Commun. ACM* 50, 5, 69–74.

T. J. Bergin and R. G. Gibson. 1996. *History of Programming Languages*. Vol. 2. ACM Press.

A. W. Biermann. 1997. *Great Ideas in Computer Science: A Gentle Introduction* (2nd. ed.). The MIT Press.

T. J. Biggerstaff and A. J. Perlis (Eds.). 1989a. *Software Reusability: Volume 1: Concepts and Models.* Addison-Wesley. DOI: https://doi.org/10.1145/73103.

T. J. Biggerstaff and A. J. Perlis (Eds.). 1989b. *Software Reusability: Volume 2: Applications and Experience.* Addison-Wesley. DOI: https://doi.org/10.1145/75722.

G. Booch. 1983. *Software Engineering with Ada.* Benjamin/Cummings.

L. Böszömenyi. 2007. *MEDICHI 2007—Methodic and Didactic Challenges of the History of Informatics.* Austrian Computer Society.

B. V. Bowden (Ed.). 1953. *Faster than Thought: A Symposium on Digital Computing Machines.* Pitman.

C. Boyer. April 2004. *The 360 Revolution.* IBM Corporation.

P. Braffort and D. Hirschberg (Eds.). 1963. *Computer Programming and Formal Systems*. North-Holland.

P. Brinch-Hansen. April 1970. Structured multiprogramming. *Commun. ACM* 13, 4, 238–241 and 250. DOI: https://doi.org/10.1145/362258.3622.

P. Brinch-Hansen. 1973. *Operating System Principles.* Prentice Hall.

P. Brinch-Hansen (Ed.). 2001. *Classic Operating Systems: From Batch Processing to Distributed Systems.* Springer-Verlag.

P. Brinch-Hansen (Ed.). 2002. *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls.* Springer Science+Business Media. DOI: https://doi.org/10.1007/978-1-4757-3472-0.

R. N. Britcher. 1999. *The Limits of Software: People, Projects, and Perspectives.* Addison Wesley Longman.

M. L. Brodie (Ed.). 2019. *Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker.* Association for Computing Machinery and Morgan & Claypool.

F. P. Brooks. 1975. *The Mythical Man-Month.* Addison-Wesley.

F. P. Brooks, Jr. 1986. No silver bullet—Essence and accidents of software engineering. In Information Processing, v86, H. J. Kugler (Ed.). Elsevier Science Publishers, 1069–1076.

N. H. Brown, M. P. Fabisch, and C. J. Rifenberg. 1975. SAFEGUARD data-processing system: Introduction and overview. *Bell Syst. Tech. J.* 54, 10, S9–S25. DOI: https://doi.org/10.1002/j.1538-7305.1975.tb03291.x.

H. Bruderer. 2015. *Meilensteine der Rechentechnik.* De Gruyter Oldenboug.

W. Buchholz (Ed.). 1962. *Planning a Computer System: Project Stretch.* McGraw-Hill.

M. Bullynck. January 2018. What is an operating system? A historical investigation (1954–1964): Historical and philosophical aspects. In L. De Mol and G. Primiero (Eds.), *Reflections on Programming Systems: Historical and Philosophical Aspects*, Vol. 133, 2019. Springer, 49–79. ISBN: 978-3-319-97225-1. DOI: https://doi.org/10.1007/978-3-319-97226-8_3.

A. R. Burks. 2003. *Who Invented the Computer? The Legal Battle that Changed Computing History.* Prometheus Book.

C. P. Burton. July/September 2005. Replicating the Manchester Baby: Motives, methods, and messages from the past. *IEEE Ann. Hist. Comput.* 24, 3, 44–60. DOI: https://doi.org/10.1109/MAHC.2005.42.

J. N. Buxton and B. Randell (Eds.). April 1970. *Software Engineering Techniques,* Report on a conference sponsored by the NATO Science Committee, Rome, Italy, October 27th to 31st, 1969. NATO Science Committee.

M. Campbell-Kelly. July 1990. *EdsacSystem: A Tutorial Guide to the Warwick University EDSAC Simulator.* University of Warwick.

M. Campbell-Kelly. 2003. *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*. The MIT Press.

M. Campbell-Kelly and M. R. Williams (Eds.). 1985. *The Moore School Lectures: Theory and Techniques for Design of Electronic Digital Computers.* The MIT Press and Tomash Publishers, Cambridge, MA; London, England; Los Angeles, CA; San Francisco, CA. ISBN 0-262-03109-4.

M. Campbell-Kelly, W. Aspray, N. Ensmenger, and J. R. Yost. 2013. *Computer, A History of the Information Machine* (3rd. ed.). The MIT Press.

C. Care. 2010. *Technology for Modeling: Electrical Analogies, Engineering Practice, and the Development of Analogue Computing.* Springer-Verlag. DOI: https://doi.org/10.1007/978-1-84882-948-0.

W. Carlson. 2017. *Computer Graphics and Computer Animation: A Retrospective Overview.* Retrieved in 2019, not specifically dated, from https://ohiostate.pressbooks.pub/graphicshistory/. The Ohio State University.

J. W. Carr III (Ed.). 1958. *Computer Programming and Artificial Intelligence, An Intensive Course for Practicing Scientists and Engineers.* Lectures given at the University of Michigan, Summer 1958, University of Michigan, College of Engineering.

B. E. Carpenter and R. W. Doran. 1986. *A. M. Turing's ACE Report of 1946 and Other Papers.* The MIT Press.

P. E. Ceruzzi. 1983. *Reckoners: The Prehistory of the Digital Computer, from Relays to the Stored Program Concept.* Greenwood Press.

P. E. Ceruzzi. 2003. *A History of Modern Computing* (2nd. ed.). The MIT Press.

P. E. Ceruzzi. July 2005. Moore's law and technical determinism. *Technol. Cult.* 46, 3, 584–593. DOI: https://doi.org/10.1353/tech.2005.0116.

H. Cervantes and R. Kazman. 2016. *Designing Software Architectures: A Practical Approach.* Addison-Wesley.

D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. October 1981. A history and evaluation of System R. *Commun. ACM* 24, 10, 632–646. DOI: https://doi.org/10.1145/358769.358784.

R. N. Charette. September 2020. No one notices the creaky software systems that run the world—Until they fail. *IEEE Spectr.* 57, 9, 24–30. DOI: https://doi.org/10.1109/MSPEC.2020.9173899.

T. E. Cheatham, Jr. August 1978. A brief description of JOVIAL. *ACM SIGPLAN Not.* 13, 8, 201–202. DOI: https://doi.org/10.1145/960118.808384.

D. R. Cheriton. 1982. *The Thoth System: Multi-Process Structuring and Portability.* North-Holland.

J. Chipps, M. Koschmann, S. Orgel, A. Perlis, and J. Smith. 1956. A mathematical language compiler. In *ACM National Computer Conference Proceedings*. 114–117. DOI: https://doi.org/10.1145/800258.808963.

N. Chomsky. 1975. *The Logical Structure of Linguistic Theory.* Plenum Press.

C. W. Churchman. 1971. *The Design of Inquiring Systems: Basic Concepts of Systems and Organizations.* Basic Books.

R. F. Clippinger. 1948. *A Logical Coding System Applied to the ENIAC*. Ballistic Research Laboratories, Report No. 673, Aberdeen Proving Ground, MD, July 30.

E. F. Codd. June 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6, 377–387. Also in Stonebraker [1988]. DOI: https://doi.org/10.1145/362384.362685.

E. F. Codd. 1971. A data base sublanguage founded on the relational calculus. In *Proceedings of the 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.* San Diego, 35–68. DOI: https://doi.org/10.1145/1734714.1734718.

E. F. Codd. 1990. *The Relational Model for Database Management: Version 2.* Addison-Wesley Longman.

P. R. Cohen and E. A. Feigenbaum (Eds.). 1982. *The Handbook of Artificial Intelligence.* Vol. III, Addison-Wesley.

I. B. Cohen and G. Welch (Eds.). 1996. *Howard H. Aiken: Computer Pioneer.* The MIT Press.

I. B. Cohen, W. T. Harwood, and M. I. Jackson. 1986. *The Specification of Complex Systems.* Addison-Wesley.

E. Constant II. 1980. *The Origins of the Turbojet Revolution.* Johns Hopkins University Press.

F. J. Corbató, M. M. Daggett, and R. C. Daley. May 1962. An experimental time-sharing system. In *Proceedings of the Spring Joint Computer Conference*. Computation Center, MIT, 335–344. DOI: https://doi.org/10.1145/1460833.1460871.

F. J. Corbató and The MIT Computation Center. 1963. *The Compatible Time-Sharing System, A Programmer's Guide.* The MIT Press.

F. J. Corbató and V. A. Vyssotsky. 1965. Introduction and overview of the Multics system. In *Proceedings of the Fall Joint Computer Conference*. 185–196. DOI: https://doi.org/10.1145/1463891.1463912.

J. W. Cortada. 1996. *A Bibliographic Guide to the History of Computer Applications, 1950–1990.* Greenwood Press.

J. W. Cortada. 2012. *The Digital Flood: The Diffusion of Information Technology Across the U.S., Europe, and Asia.* Oxford University Press. DOI: https://doi.org/10.1093/acprof:oso/9780199921553.001.0001.

J. W. Cortada. 2019. *IBM: The Rise and Fall and Reinvention of a Global Icon*. The MIT Press.

H. F. Craig. 1955. *Administering a Conversion to Electronic Accounting: A Case Study of a Large Office.* Division of Research, Graduate School of Business Administration, Harvard University.

D. Crevier. 1993. *AI: The Tumultuous History of the Search for Artificial Intelligence.* Basic Books.

M. Croarken. 1990. *Early Scientific Computing in Britain.* Oxford Science Publications.

J. Cullinane. 2014. *Smarter Than Their Machines: Oral Histories of Pioneers in Interactive Computing.* ACM Books.

M. A. Cusumano. 1991. *Japan's Software Factories.* Oxford University Press.

S. Daggett. 1931. Telephone consolidation under the Act of 1921. *J. Land Public Util. Econ.* 7, 1 (Feb. 1931), 22–35. University of Wisconsin Press.

O. Dahl, B. Myhrhaug, and K. Nygaard. October 1970. *Common Base Language*. Norwegian Computing Center, Publication No. S-22.

O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. 1972. *Structured Programming.* Academic Press.

C. J. Date. 1975. *An Introduction to Database Systems* (1st. ed.). Addison-Wesley.

M. Davis. 2000. *The Universal Computer: The Road from Leibniz to Turing.* W. W. Norton & Company.

E. G. Daylight (alias for Karel Van Oudheusden). 2011. Dijkstra's rallying cry for generalization: The advent of the recursive procedure, late 1950s–early 1960s. *Comput. J.* 54, 11, 1756–1772. DOI: https://doi.org/10.1093/comjnl/bxr002.

E. G. Daylight (alias for Karel Van Oudheusden). 2015. Towards a historical notion of 'Turing—The father of computer science'. *Hist. Philos. Logic* 36, 3, 205–228. DOI: https://www.tandfonline.com/doi/full/10.1080/01445340.2015.1082050.

T. DeMarco. 1978. *Structured Analysis and System Specification.* Yourdon Press, New York.

D. E. R. Denning. 1982. *Cryptography and Data Security.* Addison-Wesley. DOI: https://dl.acm.org/doi/book/10.5555/539308.

P. J. Denning (Ed.). 1990. *Computers Under Attack: Intruders, Worms, and Viruses.* Addison-Wesley. DOI: https://doi.org/10.1177/0894439306292346.

K. De Leeuw and J. Bergstra. 2007. *History of Information Security: A Comprehensive Handbook.* Elsevier, Oxford.

M. L. Dertouzos and J. Moses. 1979. *The Computer Age: A Twenty-Year View.* The MIT Press, Cambridge, MA. DOI: https://doi.org/10.7551/mitpress/2034.001.0001.

D. R. Deutsch. April-June 2013. The SQL standard: How it happened. *IEEE Ann. Hist. Comput.* 72–75. DOI: https://doi.org/10.1109/MAHC.2013.30.

Deutsches Museum, München. *Workshop: Technohistory of Electrical Information Technology,* Held at Deutsches Museum, München, 15th to 19th December 1990, Preliminary Papers. Deutsches Museum, München, February 1991.

E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9, 569. DOI: https://doi.org/10.1145/365559.365617.

E. W. Dijkstra. 1967. The structure of the 'THE' multiprogramming system. In *Proceedings of the first ACM symposium on Operating System Principles*. DOI: https://doi.org/10.1145/800001.811672.

E. W. Dijkstra. May 1968. The structure of the 'THE'-multiprogramming system. *Commun. ACM* 11, 5, 341–346. DOI: https://doi.org/10.1145/363095.363143.

E. W. Dijkstra. June 18, 1975. *How Do We Tell Truths That Might Hurt?* University of Texas Dijkstra Papers, EWD498. Available online at http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD498.html.

J. J. Donovan. 1972. *Systems Programming.* McGraw-Hill.

A. R. Earls. 2004. *Digital Equipment Corporation.* Arcadia Publishing.

G. Emery. 1986. *BCPL and C.* Blackwell Scientific Publications.

N. Ensmenger. 2010. *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. The MIT Press. DOI: https://www.jstor.org/stable/j.ctt5hhjdh.

N. Ensmenger. 2016. The multiple meanings of a flowchart. *Inf. Cult. J. Hist.* 51, 3, 321–351. DOI: https://doi.org/10.1353/lac.2016.0013.

R. Epstein. April–June 2013. History of Sybase. *IEEE Ann. Hist. Comput.* 35, 31–41. DOI: https://doi.org/10.1109/MAHC.2012.52.

T. Erickson and D. McDonald (Eds.). 2008. *HCI Remixed: Reflections on Works That Have Influenced the HCI Community.* The MIT Press. DOI: https://dl.acm.org/doi/book/10.5555/1355297.

J. Essinger. 2004. *Jacquard's Web: How a Hand-Loom Led to the Birth of the Information Age.* Oxford University Press.

R. R. Everett, C. A. Zraket, and H. D. Benington. 1983. SAGE–A data processing system for air defense. *Ann. Hist. Comput.* 5, 4, 330–339. DOI: https://doi.org/10.1109/MAHC.1983.10096.

E. A. Feigenbum and J. Feldman (Eds.). 1963. *Computers and Thought.* McGraw Hill. DOI: https://dl.acm.org/doi/book/10.5555/601134.

A. Finerman (Ed.). 1968. *University Education in Computer Science.* Academic Press.

FirstPerson, Inc. 1994. *Oak Language Specification*. FirstPerson, Inc., Palo Alto, CA, Retrieved from: http://www.javaspecialists.eu/archive/files/OakSpec0.2.ps.

F. M. Fisher, J. W. McKie, and R. B. Mancke. 1983. *IBM and the U.S. Data Processing Industry: An Economic History.* Praeger Publishers. DOI: https://doi.org/10.1017/S0022050700031752.

K. Flamm. 1988. *Creating the Computer: Government, Industry and High Technology.* Brookings Institution. DOI: https://dl.acm.org/doi/10.5555/42173.

I. Flores. 1960. *Computer Logic: The Functional Design of Digital Computers.* Prentice-Hall.

J. D. Foley and A. Van Dam. 1982. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley. DOI: https://dl.acm.org/doi/10.5555/6684.

M. Frauenfelder. 2007. *The Computer: An Illustrated History.* Carlton Publishing Group.

P. A. Freeman, W. R. Adrion, and W. Aspray. 2019. *Computing and the National Science Foundation, 1950–2016: Building a Foundation for Modern Computing.* ACM Books. DOI: https://doi.org/10.1145/3336323.

J. P. Fry and E. H. Sibley. March 1976. Evolution of data-base management systems. *ACM Comput. Surv.* 8, 1, 7–42. DOI: https://doi.org/10.1145/356662.356664.

R. P. Gabriel. 1996. *Patterns of Software: Tales from the Software Community.* Oxford University Press. DOI: https://dl.acm.org/doi/10.5555/235167.

E. Gade. May 2011. *Naming the Net: The Domain Name System, 1983–1990*, MA/MSc Dissertation, Columbia University and The London School of Economics and Political Science.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley.

N. Gehani. 2003. *Bell Labs: Life in the Crown Jewel.* Silicon Press.

F. Genuys. 1968. *Programming Languages: NATO Advanced Study Institute.* Academic Press, Inc.

J. Gertner. 2012. *The Idea Factory: Bell Labs and the Great Age of American Innovation.* Penguin Press. DOI: https://doi.org/10.1162/LEON_r_00497.

T. B. Glans, B. Grad, and D. Holstein. 1968. *Management Systems* (1st. ed.). Holt, Rinehart, and Winston. DOI: https://doi.org/10.1145/1480083.1480160.

R. L. Glass. 1998. *In the Beginning: Recollections of Software Pioneers.* IEEE Computer Society Press. DOI: https://doi.org/10.1145/3282517.3282521.

A. C. Glennie. *Automatic Coding of an Electronic Computer*. Notes for lecture given at University of Cambridge, February 1953. Notes dates December 1952. Copy at Computer History Museum (Computer History Museum Lot X2677.2004).

H. H. Goldstine. 1972. *The Computer from Pascal to von Neumann.* Princeton University Press.

H. H. Goldstine and J. von Neumann. April 1948. *Planning and Coding of Problems for an Electronic Computing Instrument: Report on the Mathematical and Logical Aspects of an Electronic Computing Instrument. Part II, Volume II*. The Institute for Advanced Study, Princeton University. Available online at http://bitsavers.org/pdf/ias/Planning_and_Coding_of_Problems_for_an_Electronic_Computing_Instrument_Part_II_Volume_II_Apr48.pdf.

H. P. Goodman. September 1961. The simulation of the Orion time-sharing system on Sirius. *Comput. Bull.* 5, 2, 51–55.

S. Gorn. July 1957. Standardized programming methods and universal coding. *J. ACM* 4, 3, 254–273. DOI: https://doi.org/10.1145/320881.320883.

D. A. Grier. October-December 2012. The relational database and the concept of the information system. *IEEE Ann. Hist. Comput.* 34, 4, 9–17. DOI: https://doi.org/10.1109/MAHC.2012.70.

R. L. Grossman. 2012. *The Structure of Digital Computing: From Mainframes to Big Data.* Open Data Press.

D. J. Haderle and C. M. Saracco. April–June 2013. The history and growth of IBM's DB2. *IEEE Ann. Hist. Comput.* 54–66. DOI: https://doi.org/10.1109/MAHC.2012.55.

K. Hafner and M. Lyon. 1998. *Where Wizards Stay Up Late: The Origins of the Internet.* Simon & Schuster.

T. Haigh. October–December 2009. How data got its base: Information storage software in the 1950s and 1960s. *IEEE Ann. Hist. Comput.* 31, 4, 6–25. DOI: https://doi.org/10.1109/MAHC.2009.123.

T. Haigh and M. Priestley. January 2016. Where code comes from: Architectures of automatic control from Babbage to ALGOL. *Commun. ACM* 59, 1, 39–44. DOI: https://doi.org/10.1145/2846088.

T. Haigh and P. E. Ceruzzi. 2021. *A New History of Modern Computing*. ISBN: 9780262542906. Published.

T. Haigh, M. Priestley, and C. Rope. 2016. *ENIAC in Action*. The MIT Press.

B. Hailpern. 2007. HOPL III. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages.* ACM Press. DOI: https://doi.org/10.1145/1238844.1411838.

M. Hally. 2005. *Electronic Brains: Stories from the Dawn of the Computer Age.* Joseph Henry Press. DOI: https://doi.org/10.17226/11319.

M. J. Halvorson. 2020. *Code Nation: Personal Computing and the Learn to Program Movement in America.* ACM Books. DOI: https://doi.org/10.1145/3368274.

R. Hammerman and A. L Russell. 2015. *Ada's Legacy: Cultures of Computing from the Victorian to the Digital Age.* ACM Books. DOI: https://doi.org/10.1145/2809523.

H. Harris and B. Nicol. April-June 2013. SQL/DS: IBM's first RDBMS. *IEEE Ann. Hist. Comput.* 35, 2, 69–71. DOI: https://doi.org/10.1109/MAHC.2013.28.

U. Hashagen, R. Keil-Slawik, and A. Norberg (Eds.). 2002. *History of Computing: Software Issues.* Springer-Verlag. DOI: https://doi.org/10.1007/978-3-662-04954-9.

R. Hauben and M. Hauben. 1997. *Netizens Netbook,* unpublished DRAFT. http://www.columbia.edu/~hauben/book-pdf/, Accessed January 5, 2015. A version was published as *Netizens: On the History and Impact of Usenet and the Internet,* IEEE Computer Society.

S. J. Heims. 1991. *The Cybernetics Group.* The MIT Press.

F. Helwig (Ed.). October 1957. *CODING for the MIT-IBM 704 Computer.* Massachusetts Institute of Technology.

J. Hendry. 1989. *Innovating for Failure: Government Policy and the Early British Computer Industry.* The MIT Press.

G. J. Henry. October 1984. The UNIX system: The fair share scheduler. *AT&T Bell Lab. Tech. J.* 63, 8, 1845–1857. DOI: https://doi.org/10.1002/j.1538-7305.1984.tb00068.x.

B. Higman. 1967. *A Comparative Study of Programming Languages.* American Elsevier.

A. M. Hilton. 1963. *Logic, Computing Machines, and Automation.* Meridian Books. DOI: https://doi.org/10.2307/2272108.

M. A. Hiltzik. 1999. *Dealers of Lightning.* HarperBusiness.

C. A. R. Hoare. October 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10, 576–583. DOI: https://doi.org/10.1145/363235.363259.

C. A. R. Hoare. October 1974. Monitors: An operating system structuring concept. *Commun. ACM* 17, 10, 549–557. DOI: https://doi.org/10.1145/355620.361161.

C. A. R. Hoare and C. B. Jones (Eds.). 1989. *Essays in Computing Science.* Prentice Hall.

B. D. Holbrook and W. S. Brown. 1984. A history of computing research at Bell Laboratories (1937–1975). In *A History of Science and Engineering and Science in the Bell System, Volume: Communications Sciences.* AT&T Bell Laboratories.

J. R. Holmevik. 1994. Compiling SIMULA: A historical study of technological genesis. *IEEE Ann. Hist. Comput.* 16, 4, 25–37.

G. J. Holzmann and B. Pehrson. 2003. *The Early History of Data Networks.* IEEE Computer Society Press.

D. H. Hook and J. M. Norman. 2002. *Origins of Cyberspace : a Library on the History of Computing, Networking, and Telecommunications*. Historyofscience.com.

D. H. Hook, J. M. Norman, and M. R. Williams. 2002. *Origins of Cyberspace: A Library on the History of Computing, Networking and Telecommunications.* Norman Publishing, Novato, Calif.

J. E. Hopcroft and J. D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley.

R. C. Houghton, Jr. May 1983. Software development tools. *IEEE Comput.* 16, 5, 63–70. DOI: https://doi.org/10.1109/MC.1983.1654382.

J. K. Hughes. 1969. *Programming the IBM 1130*. John Wiley & Sons.

T. P. Hughes. 2004. *Human-Built World: How to Think about Technology and Culture.* University of Chicago Press.

IEEE Communications Society. 2002. *A Brief History of Communications.* IEEE.

G. Ifrah. 2002. *The Universal History of Computing: From the Abacus to the Quantum Computer.* Wiley.

International Business Machines. 1957. *SOAP II for the IBM 650 Data Processing System.* IBM Press. Available at http://www.bitsavers.org/pdf/ibm/650/24-4000-0_SOAPII.pdf.

International Business Machines. 1964. *IBM System/360 Principles of Operation.* IBM Press.

J. Impagliazzo, M. Campbell-Kelly, G. Davies, J. A. N. Lee, and M. R. Williams. October 1998. *History in the Computing Curriculum*. IFIP, TC3/TC9 Joint Task Group.

M. M. Irvine. July–September 2001. Early digital computers at Bell Telephone Laboratories. *IEEE Ann. Hist. Comput.* 23, 3, 22–42. DOI: https://doi.org/10.1109/85.948904.

K. E. Iverson. 1962. *A Programming Language.* John Wiley & Sons.

J. F. Jacobs. 1983. SAGE overview. *Ann. Hist. Comput.* 5, 4, 323–329.

J. F. Jacobs. 1986. *The SAGE Air Defense System: A Personal History.* The MITRE Corporation.

L. Johnson. 2003. *ADAPSO Reunion Transcript, May 2–4, 2002.* iBusiness Press.

S. C. Johnson and D. M. Ritchie. 1978. Portability of C programs and the UNIX system. *Bell System Tech. J.* 57, 6, 2021–2048.

D. Kahn. 1996. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet,* revised and updated edition. Scribner.

M. A. Karim and A. A. S. Awwal. 1992. *Optical Computing: An Introduction.* Wiley-InterScience.

H. Katzan, Jr. 1970a. *Advanced Programming: Programming and Operating Systems.* Reinhold Book Corporation.

H. Katzan, Jr. 1970b. *APL Programming and Computer Techniques.* Van Nostrand Reinhold.

A. Kay. August 1972. A personal computer for children of all ages. In *Proceedings of the ACM National Conference*. Xerox Palo Alto Research Center, Boston.

A. Kay. 1993. The early history of Smalltalk. *History of Programming Languages II*. Association for Computing Machinery. posted at http://gagne.homedns.org/˜tgagne/contrib/EarlyHistoryST.html.

J. Kay and P. Lauder. January 1988. A fair share scheduler. *Commun. ACM* 31, 1, 44–55. DOI: https://doi.org/10.1145/35043.35047.

C. M. Kelty. 2008. *Two Bits.* Duke University Press.

A. Kent and J. G. Williams. 1987. *Computers in Spaceflight: The NASA Experience.* NASA (under contract NASW-3714).

B. W. Kernighan. July 18 1981. *Why Pascal Is Not My Favorite Programming Language*. Computing Science Technical Report No. 100. AT&T Bell Telephone Laboratories.

B. W. Kernighan. October 2019. *UNIX: A History and a Memoir.* Independently published.

B. W. Kernighan and P. J. Plauger. 1976. *Software Tools.* Addison-Wesley. DOI: https://doi.org/10.1145/1010726.1010728.

P. A. Kidwell and P. E. Ceruzzi. 1994. *Landmarks in Digital Computing: A Smithsonian Pictorial History.* Smithsonian Institution Press.

T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner. 1961. The Manchester University Atlas operating system. Part 1: Internal organization. *Comput. J.* 4, 3, 222–225.

G. A. Kildall. January 1980. The evolution of an industry: One person's viewpoint. *Dr. Dobb's J. Comput. Calisthenics & Orthodontia* 5, 1, 6–7.

B. Klein, R. A. Long, K. R. Blackman, D. L. Goff, S. P. Nathan, M. M. Lanyi, M. M. Wilson, J. Butterweck, and S. L. Sherrill. 2012. *An Introduction to IMS: Your Complete Guide to IBM Information Management System* (2nd. ed.). IBM Press.

D. E. Knuth. June 1965. On the translation of languages from left to right. *Inf. Control* 8, 607–639.

D. E. Knuth. December 1970. Von Neumann's first computer program. *ACM Comput. Surv.* 2, 4, 247–260. DOI: https://doi.org/10.1145/356580.356581.

D. E. Knuth. 1986. *The TeXbook.* Addison-Wesley.

D. E. Knuth. 1993. *The Stanford GraphBase: A Platform for Combinatorial Computing.* Addison-Wesley.

D. E. Knuth. 2003. *Selected Papers on Computer Languages.* Center for the Study of Language and Information, Stanford University.

J. Koomey, S. Bernard, M. Sanchez, and H. Wong. March 29 2010. Implications of historical trends in the electrical efficiency of computing. *IEEE Ann. Hist. Comput.* 33, 3, 46–54. DOI: https://doi.org/10.1353/ahc.2011.0028.

J. A. Kowal. 1988. *Analyzing Systems.* Prentice-Hall.

R. A. Kowalski. January 1988. The early years of logic programming. *Commun. ACM* 31, 1, 38–43. DOI: https://doi.org/10.1145/35043.35046.

E. Krol. 1992. *The Whole Internet: User's Guide & Catalog.* O'Reilly & Associates.

T. S. Kuhn. 1962. *The Structure of Scientific Revolutions.* University of Chicago Press.

R. Kurzweil. 1990. *The Age of Intelligent Machines.* The MIT Press.

G. Laing. 2005. *Digital Retro: The Evolution and Design of the Personal Computer.* Variant Press.

L. Lambert, C. Woodford, H. Poole, and C. J. P. Moschovitis (Eds.). 2005. *The Internet: A Historical Encyclopedia.* Three Volumes. ABC-CLIO.

B. W. Lampson and D. D. Redell. February 1980. Experience with processes and monitors in Mesa. *Commun. ACM* 23, 2, 106–117.

C. G. Langton (Ed.). 1989. *Artificial Life.* Addison-Wesley.

R. A. Larner. 1987. FMS: The IBM FORTRAN monitor system. In *AFIPS Proceedings of the 1987 National Computer Conference*. AFIPS, 815–820.

S. H. Lavington. 1980. *Early British Computers: The Story of Vintage Computers and the People Who Built Them.* Digital Press.

C. Lazou. 1988. *Supercomputers and Their Use.* Revised Edition. Oxford University, Clarendon Press.

J. A. N. Lee. 1992. *Whiggism in Computer Science: Views of the Field.* Computer Science Technical Report TR 92-17. Virginia Polytechnic Institute and State University.

J. A. N. Lee. 1996. "Those who forget the lessons of history are doomed to repeat it": Or, why I study the history of computing. *IEEE Ann. Hist. Comput.* 18, 2, 54–62.

J. C. R. Licklider. April 23 1963. *Memorandum for Members and Affiliates to the Intergalactic Computer Network.* Advanced Research Project Agency, Memorandum.

J. C. R. Licklider. 1965. *Libraries of the Future.* The MIT Press, Cambridge, MA.

R. C. Linger, H. D. Mills, and B. I. Witt. 1979. *Structure Programming: Theory and Practice*. Addison Wesley.

J. Lions. 1976a. *Lions' Commentary on UNIX 6th Edition, with Source Code*. The University of New South Wales, Australia.

J. Lions. 1976b. *A Commentary on the Sixth Edition UNIX Operating System, Booklet for Students*. The University of New South Wales, Australia.

J. Lions. 1977. *A Commentary on the Sixth Edition UNIX Operating System*. Department of Computer Science, The University of New South Wales.

J. Lions. 1996. *Lions' Commentary on UNIX 6th Edition with Source Code. Peer-To-Peer Communications*. Inc., San Jose, CA.

S. Lohr. 2001. *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts: The Programmers Who Created the Software Revolution*. Basic Books.

B. Longo. 2015. *Edmund Berkeley and the Social Responsibility of Computer Professionals.* ACM Books. DOI: https://doi.org/10.1145/2787754.

D. E. Lundstrom. 1987. *A Few Good Men from Univac.* The MIT Press.

C. A. Mack. May 2011. Fifty years of Moore's law. *IEEE Trans. Semicond. Manuf.* 24, 2, 202–207. DOI: http://dx.doi.org/10.1109/TSM.2010.2096437.

M. S. Mahoney. 1980. The roots of software engineering. In N. Metropolis, J. Howlett, and G.-C. Rota (Eds.), *A History of Computing in the Twentieth Century: A Collection of Essays*. Academic Press, 3–9.

M. S. Mahoney. 1988. The history of computing in the history of technology. *Ann. Hist. Comp.* 10, 2, 113–125. DOI: https://doi.org/10.1109/MAHC.1988.10011.

M. S. Mahoney. 2011. *Histories of Computing.* Harvard University Press.

J. Mailland and K. Driscoll. 2017. *Minitel: Welcome to the Internet.* The MIT Press. DOI: https://doi.org/10.7551/mitpress/10728.003.0001.

K. Maney, S. Hamm, and J. O'Brien. 2011. *Making the World Work Better: The Ideas That Shaped a Century and a Company.* IBM Press.

K. Mannheim. 1946. *Ideology and Utopia.* Harcourt, Brace & Company.

F. F. Martin. 1968. *Computer Modeling and Simulation.* John Wiley & Sons.

J. Martin. 1977. *Computer Data-base Organization.* Prentice-Hall.

J. Martin. 1990. *Information Engineering, Book II: Planning and Analysis.* Prentice Hall.

A. D. McAulay. 1991. *Optical Computer Architectures: The Application of Optical Concepts to Next Generation Computers.* Wiley-InterScience.

S. McCartney. 1999. *ENIAC: The Triumphs and Tragedies of the World's First Computer.* Walker and Company.

P. McCorduck. 1979. *Machines Who Think.* W. H. Freeman and Company.

D. D. McCracken. 1961. *A Guide to IBM 1401 Programming.* John Wiley & Sons.

H. McCracken. April 29 2014. Fifty years of BASIC, the programming language that made computers personal. *Time*. https://time.com/69316/basic/.

W. M. McKeeman, J. J. Horning, and D. B. Wortman. 1970. *A Compiler Generator.* Prentice-Hall.

A. Mendelsohn. April–June 2013. The Oracle story: 1984–2001. *IEEE Ann. Hist. Comput.* 35, 2, 10–23. DOI: https://doi.org/10.1109/MAHC.2012.56.

N. D. Mermin. 2007. *Quantum Computer Science: An Introduction.* Cambridge University Press. DOI: https://doi.org/10.1017/CBO9780511813870.

N. Metropolis. 1987. The beginning of the Monte Carlo method. *Los Alamos Sci.* Special Edition 125–130. https://fas.org/sgp/othergov/doe/lanl/pubs/00326866.pdf.

N. Metropolis, J. Howlett, and G-C Rota, (Eds.). 1980. *A History of Computing in the Twentieth Century: A Collection of Essays.* Academic Press.

S. Millman (Ed.). 1984. *A History of Engineering and Science in the Bell System* (1st. ed.). AT&T Bell Laboratories.

M. L. Minsky. 1986. *The Society of Mind.* Simon & Schuster.

M. L. Minsky and S. A. Papert. 1988. *Perceptrons, Expanded Edition*. The MIT Press.

T. J. Misa. 2016. *Communities of Computing: Computer Science and Society in the ACM.* ACM Books.

E. Mollick. July–September 2006. Establishing Moore's law. *IEEE Ann. Hist. Comput.* 28, 3, 62–75. DOI: https://doi.org/10.1109/MAHC.2006.45.

G. Moody. 2001. *Rebel Code: Inside Linux and the Open Source Revolution.* Perseus Publishing.

G. E. Moore. April 19 1965. Cramming more components onto integrated circuits. *Electronics* 114–117.

C. J. P Moschovitis, H. Poole, T. Schuyler, and T. M. Senft. 1999. *History of the Internet, A Chronology, 1843 to the Present.* ABC-CLIO Publishers.

R. Mueser. 1979. *Bell Laboratories Innovation in Telecommunications, 1925–1977.* Bell Laboratories.

C. J. Murray. 1997. *The Supermen: The Story of Seymour Cray and the Technical Wizards Behind the Supercomputer.* Wiley.

J. D. Musa. 2004. *Software Reliability Engineering: More Reliable Software, Faster and Cheaper* (2nd. ed.). Author House.

G. J. Myers. 1975. *Reliable Software Through Composite Design.* Petrocelli/Charter.

S. G. Nash (Ed.). 1990. *A History of Scientific Computing.* ACM Press.

National Research Council (US). Committee on Uses of Computers. 1966. *Digital Computer Needs in Universities and Colleges.* National Academy of Sciences, National Research Council.

P. Naur. 1992. *Computing: A Human Activity.* ACM Press, Addison-Wesley.

P. Naur and B. Randell. 1969. *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee,* Garmisch, Germany, 7th to 11th October 1968. NATO Science Committee.

A. Newell and H. A. Simon. 1972. *Human Problem Solving.* Prentice-Hall.

M. Nielsen and I. Chuang. 2011. *Quantum Computation and Quantum Information*. (10th. Anniversary ed.). Cambridge University Press.

N. J. Nilsson. 2010. *The Quest for Artificial Intelligence: A History of Ideas and Achievements.* Cambridge University Press.

J. November. 2012. *Biomedical Computing: Digitizing Life in the United States.* Johns Hopkins Press. DOI: https://doi.org/10.1353/book.14634.

P. Oman and T. Lewis. 1990. *Milestones in Software Evolution.* IEEE Computer Society Press.

A. Oram and G. Wilson. 2007. *Beautiful Code: Leading Programmers Explain How They Think.* O'Reilly Media.

G. O'Regan. 2016. *Introduction to the History of Computing, A Computing History Primer.* Springer.

M. A. Padlipsky. 1985. *The Elements of Networking Style: And Other Essays and Animadversions on the Art of Intercomputer Networking.* Prentice-Hall.

G. Parayil. 1999. *Conceptualizing Technological Change.* Rowman & Littlefield.

D. F. Parkhill. 1966. *The Challenge of the Computer Utility.* Addison-Wesley.

R. L. Patrick. January 1987. *General Motors/North American Monitor for the IBM 704 Computer*. RAND Corporation, P-7316. https://www.rand.org/pubs/papers/P7316.html.

D. A. Patterson and J. L. Hennessy. 2013. *Computer Organization and Design: The Hardware/-Software Interface* (5th. ed.). Morgan Kaufmann Publishers.

J. Pelkey. 2019. *Entrepreneurial Capitalism and Innovation: A History of Computer Communications 1968–1988*. http://www.historyofcomputercommunications.info/ Book, to be published late 2021 by ACM Books.

D. Pessel. 2006. *A Brief History of Computer Time: 60 Years of Computer Technology and the People Who Helped Make It Happen.* IEEE Computer Society Press.

C. Pettijohn. March 1986. Achieving quality in the development process. *AT&T Tech. J.* 65, 3, 85–93. DOI: https://doi.org/10.1002/j.1538-7305.1986.tb00296.x.

C. Petzold. 1999. *Code: The Hidden Language of Computer Hardware and Software.* Microsoft Press.

J. R. Pierce. 1980. *An Introduction to Information Theory: Symbols, Signals, and Noise* (2nd. ed.). Dover.

R. P. Polivka and S. Pakin. 1975. *APL: The Language and Its Usage.* Prentice-Hall.

G. J. Popek and R. P. Goldberg. July 1974. Formal requirements for virtualizable third generation architectures. *Commun. ACM* 17, 7, 412421. DOI: https://doi.org/10.1145/361011.361073.

M. Priestley. 2011. *A Science of Operations: Machines, Logic and the Invention of Programming.* Springer-Verlag.

E. W. Pugh. 2009. *Building IBM: Shaping an Industry and Its Technology.* The MIT Press. DOI: https://doi.org/10.7551/mitpress/1687.001.0001.

E. W. Pugh, L. R. Johnson, and J. H. Palmer. 1991. *IBM's 360 and Early 370 Systems.* The MIT Press.

A. Ralston and E. Reilly, Jr (Eds.). 1983. *Encyclopedia of Computer Science and Engineering* (2nd. ed.). Van Nostrand Reinhold.

B. Randell. 1973. *The Origins of Digital Computers: Selected Papers.* Springer.

B. Randell. 1976. *The COLOSSUS.* Computing Laboratory Report Number 90. University of Newcastle upon Tyne.

B. Randell. October 1982. From analytical engine to electronic digital computer: The contributions of Ludgate, Torres, and Bush. *Ann. Hist. Comput.* 4, 4, 327–341. DOI: https://doi.org/10.1109/MAHC.1982.10042.

E. S. Raymond. 1999. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly.

M. G. Rekoff, Jr. 1967. *Analog Computer Programming.* Charles E. Merrill Books.

Reserve Bank of New Zealand. 2008. *About the Reserve Bank Museum.* Reserve Bank of New Zealand.

J. Rice and R. A. DeMillo (Eds.). 1994. *Studies in Computer Science: In Honor of Samuel D. Conte.* Plenum Press.

C. Rich and R. C. Waters. 1990. *The Programmer's Apprentice.* Addison-Wesley.

L. Rising. 1998. *The Patterns Handbook: Techniques, Strategies, and Applications.* Cambridge University Press.

D. M. Ritchie. October 1984. The evolution of the UNIX time-sharing system. *AT&T Bell Labs. Tech. J.* 63, 6, Part 2, 1577–1593. DOI: http://cm.bell-labs.com/cm/cs/who/dmr/hist.htm

R. Rojas and U. Hashhagen. 2002. *The First Computers: History and Architectures.* The MIT Press.

S. Rosen. 1967. *Programming Systems and Languages.* McGraw-Hill.

S. Rosen. 1968. *Electronic Computers, A Historical Survey*. Computer Sciences Department Technical Report TR25. Purdue University.

S. Rosenberg. 2008. *Dreaming in Code: Two Dozen Programmers, Three Years, 4,732 Bugs, and One Quest for Transcendent Software.* Three Rivers Press.

R. F. Rosin. 1969. Supervisory and monitor systems. *ACM Comput. Surv.* 1, 1 (March 1969), 37–54. DOI: https://doi.org/10.1145/356540.356544.

A. L. Russell. 2014. *Open Standards and the Digital Age: History, Ideology, and Networks.* Cambridge University Press.

H. Sackman. 1967. *Computers, System Science, and Evolving Society: The Challenge of Man–Machine Digital Systems.* John Wiley & Sons.

J. E. Sammet. 1969. *Programming Languages: History and Fundamentals.* Prentice-Hall.

J. E. Sammet. July 1972. Programming languages: History and future. *Commun. ACM* 15, 1, 601–610.

A. Samuel. March 1980. *Essential E*. Stanford Artificial Intelligence Laboratory. Computer Science Department, Memo: AIM-335, Report No. CS-TR-80-796. http://infolab.stanford.edu/pub/cstr/reports/cs/tr/80/796/CS-TR-80-796.pdf.

R. R. Schaller. June 1997. Moore's law: Past, present and future. *IEEE Spectr*. 34, 6, 52–59. DOI: https://doi.org/10.1109/6.591665.

E. H. Schein, P. J. Kampas, P. S. Delisi, and M. M. Sonduck. 2004. *DEC is Dead, Long Live DEC: The Lasting Legacy of Digital Equipment Corporation.* Berrett-Koehler Publishers.

J. I. Schwartz. 1981. JOVIAL session. In *History of Programming Languages.* ACM, 369–388.

Scientific American. 1966. *Information.* ISBN-13: 978-0716709671, W. H. Freeman and Company.

R. Sethi. 1989. *Programming Languages, Concepts and Constructs.* Addison-Wesley.

C. E. Shannon. 1940. A symbolic analysis of relay and switching circuits. Results previously published in *AIEE Transactions* 57, 1938, pp. 713–723. MS thesis. Massachusetts Institute of Technology.

C. E. Shannon and J. McCarthy (Eds.). 1956. *Automata Studies.* Princeton University Press.

E. Y. Shapiro. September 1983. The fifth generation project—A trip report. *Commun. ACM* 26, 9, 637–641. DOI: https://doi.org/10.1145/358172.358179.

F. R. Shapiro. April–June 2000. Comments, queries, and debate. *IEEE Ann. Hist. Comput.* 69–71.

D. P. Siewiorek, C. G. Bell, and A. Newell. 1971. *Computer Structures: Principles and Examples.* McGraw-Hill Computer Science Series.

A. Silberschatz, P. B. Galvin, and G. Gagne. 2012. *Operating System Concepts* (18th. ed.). John Wiley & Sons.

J. Singh. 1966. *Great Ideas in Information Theory, Language and Cybernetics.* Dover.

R. Sippl. April–June 2013. Informix: Information management on UNIX. *IEEE Ann. Hist. Comput.* 35, 2, 42–53.

N. J. A. Sloane and A. D. Wyner (Eds.). 1993. *Claude Elwood Shannon Collected Papers.* IEEE Press.

D. K. Smith and R. C. Alexander. 1999. *Fumbling the Future: How Xerox Invented, Then Ignored, the First Personal Computer.* iUniverse.

W. Stallings. 1992. *ISDN and Broadband ISDN* (2nd. ed.). Macmillan.

W. E. Steinmueller. 1995. The U.S. software industry: An analysis and interpretive history. In D. C. Mowery (Ed.), *The International Computer Software Industry.* Oxford University Press.

M. Stonebraker. June 1980. Retrospection on a database system. *ACM Trans. Database Syst.* 5, 2, 225–240. DOI: https://doi.org/10.1145/320141.320158.

M. Stonebraker (Ed.). 1988. *Readings in Database Systems.* Morgan Kaufmann Publishers, San Mateo.

M. Swain and P. Freiberger. 2014. *Fire in the Valley, The Birth and Death of the Personal Computer* (3rd. ed.). The Pragmatic Bookshelf.

A. Svoboda and H. M. James (Eds.). 1964. *Computing Mechanisms and Linkages,* Boston Technical Publishers.

A. Tatnall (Ed.). 2012. *Reflections on the History of Computing: Preserving Memories and Sharing Stories.* IFIP Advances in Information and Communication Technology (Book 387). Springer.

A. Tatnall, T. Blyth, and R. Johnson (Eds.). June 2013. Making the history of computing relevant. In *IFIP WG 9.7 International Conference, HC 2013*, London, UK, Springer. DOI: https://doi.org/10.1007/978-3-642-41650-7.

M. Tedre. 2015. *The Science of Computing: Shaping a Discipline.* CRC Press.

C. Tozzi. 2017. *For Fun and Profit: A History of the Free and Open Source Software Revolution.* The MIT Press.

K. W. Tracy and P. M. Bouthoorn. 1997. *Object-Oriented Artificial Intelligence Using C++.* Computer Science Press, an imprint of W. H. Freeman and Company.

G. R. Trimble, Jr. May 1968. Bibliography 14: A time-sharing bibliography. *ACM Comput. Rev.* 291–301.

L. E. Truesdell. 1965. *The Development of Punch Card Tabulation in the Bureau of the Census 1890–1940, with Outlines of Actual Tabulation Programs.* US Department of Commerce, Bureau of the Census.

A. B. Tucker. 1977. *Programming Languages.* McGraw-Hill, Inc.

R. S. Tucker. July 2010. The role of optics in computing. *Nat. Photon.* 4, 7, 405.

A. M. Turing. 1936. On computable numbers, with an application to the Entscheidungsproblem. (Received May 28, 1936), *Proc. London Math. Soc*. Ser. 2, 43, 1. 230–254.

J. Tukey. 1958. The teaching of concrete mathematics. *Am. Math. Mon.* 65, 1, 19. DOI: https://doi.org/10.1080/00029890.1958.11989128.

F. Turner. 2008. *From Counterculture to Cyberculture: Stewart Brand, the Whole Earth Network, and the Rise of Digital Utopianism.* University of Chicago Press.

J. D. Ullman. 1980. *Principles of Database Systems* (1st. ed.). Pitman.

U. Valhalia. 1996. *UNIX Internals: The New Frontiers.* Prentice-Hall.

G. van den Hove. December 2014. On the origin of recursive procedures. *Comput. J.* 58, 11, 2892–2899.

J. von Neumann. 1958. *The Computer and the Brain.* Yale University Press.

J. von Neumann. 1945. *First Draft of a Report on the EDVAC.* Contract No. W-670-ORD-4926 between the United States Army Ordnance Department and the University of Pennsylvania, June 30, 1945. *IEEE Ann. Hist. Comput*. 15, 4, 27–43.

D. Walden and R. Nickerson (Eds.). 2011. *A Culture of Innovation: Insider Accounts of Computing and Life at BBN.* Waterside Publishing.

E. L. Wallace. 1961. *Management Influence on the Design of Data Processing Systems.* Division of Research, Graduate School of Business Administration, Harvard University.

J. J. Wallace and W. W. Barnes. Aug 1984. Designing for ultrahigh availability: The Unix RTR operating system. *IEEE Comput*. 17, 8, 31–39. DOI: https://doi.org/10.1109/MC.1984.1659215.

P. Wegner. 1968. *Programming Languages, Information Structures, and Machine Organization.* McGraw-Hill Book Company.

P. Wegner. 1972. The Vienna definition language. *ACM Comput. Surv.* 4, 1, 5–63. DOI: https://doi.org/10.1145/356596.356598.

G. M. Weinberg. 1971. *The Psychology of Computer Programming.* Van Nostrand Reinhold.

J. Weizenbaum. 1976. *Computer Power and Human Reason.* W. H. Freeman and Company.

N. Weizer. January 1981. A history of operating systems. *Datamation*. 118–126.

R. L. Wexelblat. 1981. *History of Programming Languages.* ACM Monograph Series. Academic Press.

L. R. Wiener. 1993. *Digital Woes: Why We Should Not Depend on Software.* Addison-Wesley.

N. Wiener. 1948. *Cybernetics, or Control and Communication in the Animal and Machine* (1st. ed.). John Wiley & Sons.

N. Wiener. 1956. *I Am a Mathematician.* The MIT Press.

M. V. Wilkes. 1995. *Computing Perspectives.* Morgan-Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann.

M. V. Wilkes, D. J. Wheeler, and S. Gill. 1951. *The Preparation of Programs for an Electronic Digital Computer.* Addison-Wesley.

M. R. Williams. 1997. *A History of Computing Technology* (2nd. Ed.). IEEE Computer Society Press.

J. R. Yost. 2011. *The IBM Century: Creating the IT Revolution.* IEEE Computer Society Press.

E. N. Yourdon. 1979. *Classics in Software Engineering.* Yourdon Press.

E. N. Yourdon. 1982. *Writings of the Revolution: Selected Readings on Software Engineering.* Yourdon Press.

J. Ziman. 2000. *Technological Change as an Evolutionary Process.* Cambridge University Press.

M. M. Zloof. May 1975. Query by example. In *Proceedings of the National Computer Conference,* Anaheim, CA.

Author's Biography

Kim W. Tracy

Kim W. Tracy has a long and varied history in many different aspects of software. Trained as a second-generation computer scientist with many of his professors being first-generation computer scientists, he's worked on a wide variety of software ranging from system software (UNIX® at Bell Labs) to database systems and expert systems. While at Bell Labs, he worked on a number of different products including the 5ESS® Telephone Switch, as well as consulting for clients around the world. He has since served as Chief Information Officer at Northeastern Illinois University and oversaw all aspects of the technology used by the university. He's also taught many courses in computer science that range over traditional computer science, software engineering, and information technology. He currently teaches computer science and software engineering full-time at Rose-Hulman Institute of Technology. He is co-author of the textbook *Object-Oriented Artificial Intelligence Using C++*.

He's also been involved with university-level computing program accreditation with ABET for several decades. He's a senior member of ACM and IEEE and a member of the Society for the History of Technology. He serves on the ACM History Committee and has served as editor-in-chief of IEEE *Potentials* magazine as well as other IEEE boards and committees.

Index