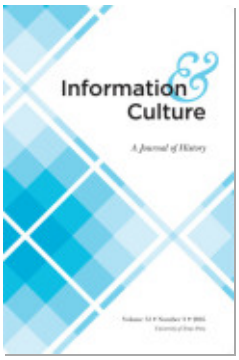
The Multiple Meanings of a Flowchart

Nathan Ensmenger



Information & Culture: A Journal of History, Volume 51, Number 3, 2016, pp. 321-351 (Article)

Published by University of Texas Press
DOI: *https://doi.org/10.1353/lac.2016.0013*

The Multiple Meanings of a Flowchart

Nathan Ensmenger

From the very earliest days of electronic computing, flowcharts have been used to represent the conceptual structure of complex software systems. In much of the literature on software development, the flowchart serves as the central design document around which systems analysts, computer programmers, and end users communicate, negotiate, and represent complexity. And yet the meaning of any particular flowchart was often highly contested, and the apparent specificity of such design documents rarely reflected reality. Drawing on the sociological concept of the boundary object, this article explores the material culture of software development with a particular focus on the ways in which flowcharts served as political artifacts within the emerging communities of practices of computer programming.

In the September 1963 issue of the data-processing journal *Datamation*, there appeared a curious little four-page supplement titled "The Programmer's Primer and Coloring Book."[1] This rare but delightful bit of period computer industry whimsy is full of self-deprecating (and extremely "in") cartoons about the working life of computer programmers. For example, "See the program bug. He is our friend!! Color him swell. He gives us job security." Some of these jokes are a little dated, but most hold up surprisingly well.

One of the most insightful and revealing of "The Programmer's Primer and Coloring Book" cartoons is also one of the most minimalistic. The drawing is of a simple program flowchart accompanied by a short and seemingly straightforward caption: "This is a flowchart. It is usually wrong."

In case you don't get the joke, here is some context: by the early 1960s, the flowchart was well established as an essential element of any large-scale software development project. Originally introduced into

Nathan Ensmenger is an associate professor in the School of Informatics and Computing at Indiana University. Much of his research focuses on the history of software and software workers, although he is currently working on a book exploring the global environmental history of the digital economy.

Information & Culture, Vol. 51, No. 3, 2016

DOI: 10.7560/IC51302

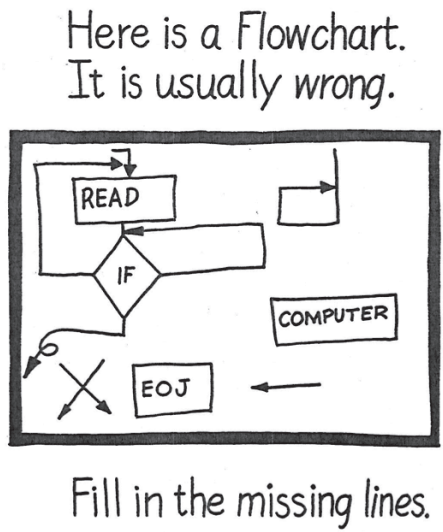


Figure 1. "The Programmer's Primer and Coloring Book," *Datamation* (September 1963).

computing by John von Neumann in the mid-1940s, flowcharts were a schematic representation of the logical structure of a computer program. The idea was that an analyst would examine a problem, design an algorithmic solution, and outline that algorithm in the form of a flowchart diagram. A programmer (or "coder") would then translate that flowchart into the machine language understood by the computer. The expectation was that the flowchart would serve as the design schematic for the program code (in the literature from this period flowcharts were widely referred to as the "programmer's blueprint") with the assumption that once this "blueprint" had been developed, "the actual coding of the computer program is rather routine."[2]

For contemporary audiences, the centrality of the flowchart to software development would have been self-evident. Every programmer in this period would have learned how to flowchart.[3] In the same year that "The Programmer's Primer and Coloring Book" was published, the American Standards Association approved a standardized flowchart symbol vocabulary.[4] Shortly thereafter, the inclusion of flowcharting instruction in introductory programming courses was mandated by the Association for Computing Machinery's influential *Curriculum '68* guidelines.[5] A 1969 IBM introduction to data processing referred to flowcharts as "an all-purpose tool" for software development and noted that "the programmer uses flowcharting in and through every part of

his task."[6] By the early 1970s, the conventional wisdom was that "developing a program flowchart is a necessary first step in the preparation of a computer program."[7]

But every programmer in this period also knew that although drawing and maintaining an accurate flowchart was what programmers were *supposed* to do, this is rarely what happened in actual practice. Most programmers preferred not to bother with a flowchart or produced their flowcharts only after they were done writing code.[8] Many flowcharts were only superficial sketches to begin with and were rarely updated to reflect the changing reality of a rapidly evolving software system.[9] Many programmers loathed and resented having to draw (and redraw) flowcharts, and the majority did not. Frederick Brooks, in his classic text on software engineering, dismissed the flowchart as an "obsolete nuisance," "a curse," and a "space hogging exercise in drafting."[10] Wayne LeBlanc lamented that despite the best efforts of programmers to "communicate the logic of routines in a more understandable form than computer language by writing flowcharts," many flowcharts "more closely resemble confusing road maps than the easily understood pictorial representations they should be."[11] Donald Knuth argued that flowcharts not only were time-consuming to create and expensive to maintain but also were generally rendered obsolete almost immediately. In any active software development effort, he argued, "any resemblance between our flow charts and the present program is purely coincidental."[12]

All of these critiques are, of course, the basis of the humor in the *Datamation* cartoon: as every programmer knew well, although in theory the flowchart was meant to serve as a design document, in practice it often served only as ex post facto justification. Brooks denied that he had ever known "an experienced programmer who routinely made detailed flow charts before beginning to write programs," suggesting that "where organization standards require flow charts, these are almost invariably done after the fact."[13] And in fact, one of the first commercial software packages, Applied Data Research's Autoflow, was designed specifically to reverse-engineer a flowchart "specification" from already-written program code. In other words, the implementation of many software systems actually preceded their own design! This indeed is a wonderful joke or, at the very least, a paradox. As Marty Goetz, the inventor of Autoflow, recalled, "Like most strong programmers, I never flowcharted; I just wrote the program."[14] For Goetz, among others, the flowchart was nothing more than a collective fiction: a requirement driven by the managerial need for control and having nothing to do with the actual design or construction of software. The construction of the flowchart could thus be safely left to the machine, since no one was

really interested in reading it in the first place. Indeed, the expert consensus on flowcharts seemed to accord with the popular wisdom captured by "The Programmer's Primer and Coloring Book": there were such things as flowcharts, and they were generally wrong.

Flowcharts as Boundary Objects

It would be easy to view the flowchart as a failed technology, an earnest attempt to visualize complexity and guide software design that simply was not up to the task. But while the truth expressed in this cartoon was meant to be humorous, my analysis of it will be entirely serious. I will suggest that not only was the flowchart one of the most significant and durable innovations of the great computer revolution of the midtwentieth century but that the *Datamation* cartoon captures perfectly its essential paradox: computer flowcharts were at once both widely used (and useful), *and* they were almost always an incorrect and inadequate reflection of reality. To view the computer flowchart as having only one purpose (and a failed purpose at that) is narrow and misleading; in reality, every flowchart had multiple meanings and served several purposes simultaneously. Yes, flowcharts were imagined (and sometimes used) as design specifications for programmers, but they were also tools for analysis, planning, and communication. For managers, they were a mechanism for organizing the work process, estimating costs, managing projects, and exerting industrial discipline. Flowcharts were blueprints, contracts, and documentation. They could also be read as maps of the technological, social, and organizational life of software systems.

To borrow a concept from Susan Leigh Starr and James Griesemer, the computer flowchart can be thought of as a boundary object, an artifact that simultaneously inhabits multiple intersecting social and technical worlds. In each of these worlds, the boundary object has a well-defined meaning that "satisf[ies] the informational requirements" of the members of that community; at the intersection of these worlds, the boundary object is flexible enough in meaning to allow for conversation *between* multiple communities.[15] As Starr and Griesemer describe it, successful boundary objects are "both plastic enough to adapt to local needs and the constraints of the several parties employing them, yet robust enough to maintain a common identity across sites."[16] Boundary objects have become a central analytical tool in the history and sociology of science because they allow for technological artifacts to have meanings that are both fixed and flexible, multifarious without being contradictory.

More recently, Kathryn Henderson has applied the concept of boundary objects to the sketches and drawings used by engineers to

communicate among themselves and between design groups, as well as with managers, machinists, and shop workers. She identifies these visual and representational technologies as boundary objects that both convey useful information and function in a more explicitly organizational role as "conscription devices." As the common point of focus for conversation and negotiation about the design process, boundary objects enlist group participation by serving as an essential repository of knowledge and interaction. "To participate at all in the design process," Henderson argues, "actors *must* engage one another through the visual representation."[17] Such was the conscriptive power of these objects that "if a visual representation is not brought to a meeting of those involved with the design, someone will sketch a facsimile on a white board. . . . [A] team member will leave the meeting to fetch the crucial drawings so group members will be able to understand one another."[18]

In a similar manner, flowcharts serve simultaneously as boundary objects and conscription devices. It is no coincidence that flowcharts became ubiquitous (in fact, compulsory) in the period known to contemporaries and historians alike as the "software crisis." As the historian Michael Mahoney famously suggested, the history of computing in the 1960s revolves around the growing realization that "software is hard."[19] By the end of that decade, the dramatically rising costs associated with software development seemed to many observers a harbinger of the imminent "fizzle of the computer revolution."[20] And to the dismay of many computer specialists, it was becoming increasingly clear that the real reasons why software was so hard were not primarily technological but rather social and organizational. It was not programming per se that made software development so difficult but the larger processes of problem analysis, design, communication, and documentation associated with programming that posed the real problem.[21] As software projects expanded in scope and complexity, flowcharts increasingly served not only as a means of organizing and communicating technical knowledge but also as tools for resolving (or at least mediating) political, organizational, and, in some cases, legal disputes.

From Flow Diagram to Flowchart

The first printed use of a flowchart in the context of electronic computing can be found in a 1946 report by Haskell Curry and Willa Wyatt describing a method for performing inverse interpolation on the ENIAC.[22] But in a subsequent paper Curry credited the original idea to John von Neumann and Herman Goldstine, and it was a 1948 report by these two authors that first systematically described and applied

a system for symbolically representing algorithms using a "flow diagram."[23] Not only was this 1948 report much more widely disseminated (the Curry/Wyatt paper was classified), but it carried with it the prestige and authority of von Neumann, and as a result it is von Neumann and Goldstine to whom the concept of the programmer's flow diagram is generally attributed.[24]

But while von Neumann and Goldstine might have been the first to apply it to computing, the flow diagram was already by this period a well-established representational technology. Such diagrams had long been used by hydrodynamic engineers to track the circulation of fluids, and in the early twentieth century they had been adopted by process engineers in a wide variety of industries to outline "the course through which any material—from corn flour to an engine block—travels whilst undergoing manufacture."[25] Indeed, it has been speculated that it was in his early training as a chemical engineering student that von Neumann would have learned about the flow diagram. In any case, by the 1930s flow diagrams were widely used within industrial manufacturing, and as understandings of what constituted "material flow" expanded and became increasingly abstract, they were used to document everything from department organization to the movement of records. Along with the Gantt chart, the flow diagram was one of several emerging technologies for visualizing organizational and procedural complexity.[26]

The appropriation of a technology that already had a well-established meaning in the context of industrial manufacturing reveals much about what von Neumann and Goldstine thought about computer programming—and computer programmers. In the vision of computer programming outlined in "Planning and Coding of Problems for an Electronic Computing Instrument," von Neumann and Goldstine propose a six-step programming process: in the first five steps of this process, which they referred to as the "dynamic" phase, a skilled mathematician or scientist would conceptualize a problem mathematically and physically, perform a numerical analysis, and design an algorithm. The product of these first five phases would be the flow diagram. In the sixth and final stage of the programming process, the "static" phase, a coder would transform the flow diagram into a set of specific machine instructions. Implied by the language used to describe it, the work of the coder was assumed to be straightforward, mechanical, and merely clerical. "We feel certain that a moderate amount of experience with this stage of coding suffices to remove from it all difficulties, and to make it a perfectly routine operation," von Neumann and Goldstine confidently declared.[27] In the case of the ENIAC project, which was the only model of software development that von Neumann and Goldstine had available to them,

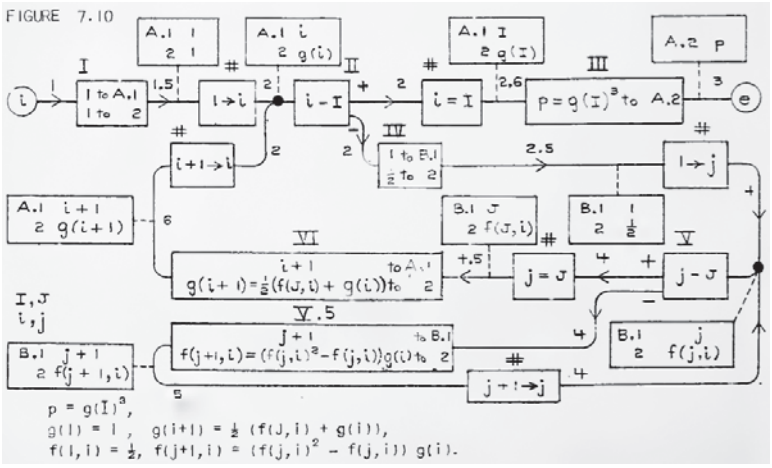


Figure 2. An original flow diagram from Goldstine and von Neumann's 1948 "Planning and Coding of Problems for an Electronic Computing Instrument."

the low-status, seemingly routine task of coding the flow diagram was generally assigned to women.

The flow diagrams introduced by von Neumann and Goldstine in the late 1940s were adopted by, among others, the programmers at the newly formed Eckert-Mauchly Computer Corporation (soon to become the UNIVAC division at Remington Rand). In April 1950 Grace Hopper and Betty Holberton introduced what they called "flow charts" into the teaching materials that they developed for a programming course at EMCC. These materials specifically reference the earlier work of von Neumann and Goldstine.[28] Flow diagrams in the style created by von Neumann and Goldstine can also be found in the documentation for a differential analysis program developed for the earliest versions of the ACE computer designed by Turing at the National Physical Laboratory.[29] By the end of the 1950s the "flow chart" (or, increasingly, "flowchart") had been thoroughly integrated into the programming practices of the industry.[30]

This early phase of the dissemination of flowchart technology seems to emphasize the first meaning of the flowchart outlined by von Neumann and Goldstine; that is, the flowchart was a high-level conceptual technology intended primarily for scientists and other problem-domain specialists for the development of algorithmic solutions. As Hollis Kinslow, who oversaw the development of the IBM Time-Sharing Monitor System in the early 1960s, would later describe it, the design

process for many large software projects revolved entirely around the flowchart:

1. Flowchart until you think you understand the problem.
2. Write code until you realize that you don't.
3. Go back and re-do the flowchart.
4. Write some more code and iterate to what you feel is the correct solution.[31]

In this representation of the role of the flowchart, the chart functions largely as a design technology, a "thing for thinking with," as Sherry Turkle has suggested.[32] As one popular textbook from the early 1970s described it, "Flowcharting is an essential tool in problem solving. . . . The person who cannot flowchart cannot anticipate a problem, analyze the problem, plan the solution, or solve the problem."[33] This sentiment is very much in line with the principal meaning of the flow diagram as outlined by von Neumann and Goldstine: the flow diagram was a user-friendly tool for high-level planners to make use of as they found convenient or necessary. If a scientist found the flow diagram/flowchart to be useful as an aid to thought or as a memory device, then he (or, very occasionally, she) could go ahead and make use of it; if not, he was free to develop his own design techniques and technologies.

If we look more closely at the representation of the flowchart as embodied in the many training tools, textbooks, templates, and software methodologies that were produced in the 1950s and 1960s, however, we see that it is the second of von Neumann and Goldstine's purposes—the flowchart as means of encouraging industrial discipline—that would ultimately become dominant. Yes, flow diagrams were a tool for analysis and a method of formalizing and documenting a mathematical algorithm, but they were also a tool for planning, organizing, and distributing the mental and mechanical labor required to construct a computer program. In the context of an emerging "software crisis" defined by the inability of organizations to train, recruit, manage, and retain skilled computer programmers, the belief (hope?) that a well-defined flowchart could help bring order to the seeming chaos of software development was appealing to employers, managers, and programmers alike.[34]

Flowchart as Blueprint

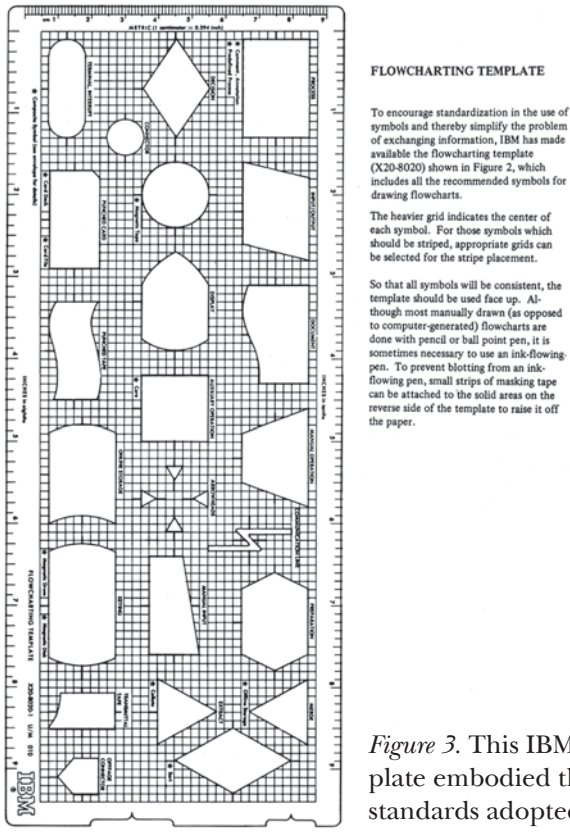
By the middle of the 1960s, a common language and symbolic vocabulary for constructing computer flowcharts had emerged and been

formalized in national (and later international) standards, institutionalized in curriculum and textbooks, and embodied in physical objects such as templates and worksheets.[35] In 1965 a working group within the American Standards Association representing a consortium of academic societies (among them the Association for Computing Machinery and the American Management Association), computer manufacturers (including IBM, Honeywell, and Remington Rand UNIVAC), user groups, and the Department of Defense published its "Conventions for the Use of Symbols in the Preparation of Flowcharts for Information Processing Systems." A similar set of conventions was adopted by the International Standards Organization (ISO) in 1973.

The standardization of flowchart symbols allowed the charts to become more portable, both conceptually and organizationally. As Bruno Latour famously suggested of engineering drawings, by "flatten[ing] out onto the same surface" an otherwise disconnected set of activities (e.g., business process analysis and computer programming), standardized flowcharts created an "optically consistent space" that allowed a variety of actors to focus their attention on a single, well-defined problem.[36] The standardized objects on a flowchart provided an unambiguous representation of reality that could be productively used to plan and organize work, measure results, and allocate responsibility. Anyone who learned to master the vocabulary of the standardized flowchart could, in theory, at least, contribute to the conversation about how a given software project should be designed and what it ought to accomplish.

For many participants in the corporate computer revolution of the 1960s, learning to flowchart was their first (and in some cases only) lesson in software development.[37] Using the predefined symbol charts and templates provided by the ANSI and ISO guidelines, even the least technically proficient employee could quickly assemble a coherent, legible, and standardized flowchart quickly and easily.[38] The ability to construct a flowchart provided the illusion, at least, of mastery over a complex process of software analysis and design, a comforting thought in a period in which many corporate managers worried about computer specialists using their technical expertise to make an "electronic power grab."[39]

Even aspiring programmers or programmer trainees often spent more time drawing flowcharts than working with actual computer code.[40] Paper was cheap, while computer time was expensive. Vocational schools and academic computer science programs alike focused on the flowchart as an essential tool for learning and communication. In fact, in a 1965 article titled "Education and Training of a Business Programmer" that nicely captures the conventional wisdom of the era,



FLOWCHARTING TEMPLATE

To encourage standardization in the use of symbols and thereby simplify the problem of exchanging information, IBM has made available the flowcharting template (X20-8020) shown in Figure 2, which includes all the recommended symbols for drawing flowcharts.

The heavier grid indicates the center of each symbol. For those symbols which should be striped, appropriate grids can be selected for the stripe placement.

So that all symbols will be consistent, the template should be used face up. Although most manually drawn (as opposed to computer-generated) flowcharts are done with pencil or ball point pen, it is sometimes necessary to use an ink-flowing pen. To prevent blotting from an ink-flowing pen, small strips of masking tape can be attached to the solid areas on the reverse side of the template to raise it off the paper.

Figure 3. This IBM flowcharting template embodied the ISO and ANSI standards adopted in 1970.

the flowchart served as the foundational document on which an entire software development work process was constructed. The first step of the process was the analysis of the problem, the second the development of the flowchart, and the third (and final) the translation of the flowchart into a programming language.[41] Indeed, by the end of the 1970s it was "almost impossible to find an introductory programming text that [did] not make extensive use of flowcharts."[42]

In this dramatically simplified model of software development (which was endorsed by, among others, the Data Processing Management Association, the preeminent industry professional society in this period), the flowchart functioned as the central design document. The most common analogy used to explain the role of flowchart was the architectural blueprint. Consider the following claims from Thomas McInerney and Andre Vallee's 1973 *A Student's Guide to Flowcharting*:

> Flowcharts are to programmers as blueprints are to engineers. Before a construction engineer begins in building, he draws detailed plans from which to work. These plans are called blueprints.
>
> Before a programmer begins to code a program into one of the computer languages (such as COBOL or ALGOL), you must have a detailed blueprint of the steps to follow. The blueprint is known as a flowchart.
>
> Engineers and construction foremen must be able to draw and read blueprints. Programmers must be able to draw and read flowcharts. Flowcharting is a necessary and basic skill for all programmers.[43]

In their suggestion that a flowchart is a blueprint, the authors of this guidebook—and many other programming textbooks from this period—are not waxing idly metaphorical. They were describing a software development methodology in which the flowchart plays a very specific and absolutely indispensable role as both a design schematic and a tool for organizing the division of labor and the work of construction.[44]

The flowchart-as-blueprint analogy implied a very specific relationship between the designer/architect and the programmer/builder. As Ronald Elliott described in his 1972 *Problem Solving and Flowcharting*, "The purpose of drawing a flowchart is to make the coding of the problem easier. The program code should follow the flowchart step-by-step. When this procedure is followed, the program code should reflect exactly the same procedures as those of the flowchart."[45] George Gleim, in his 1970 *Program Flowcharting*, argued that drawing the flowchart was *the* critical task associated with software development. "Once the flowchart has been correctly developed," he suggested, "the actual coding of the computer program is rather routine."[46] In this reiteration of the head/hand distinction first outlined by Goldstine and von Neumann, it was in the construction of the flowchart that the real intellectual work of problem solving was accomplished.[47] As Thomas Schriber in his 1969 *Fundamentals of Flowcharting* described it, once a proper flowchart had been developed, the person charged with "preparing the [programming] language equivalent of a flowchart" would find the task "to be largely a mechanical one."[48] In their repeated assertions that the true meaning of the flowchart was as design document, these texts attempted to establish or reify an occupational and professional hierarchy within computing in which the high-level conceptual work of design could be clearly distinguished from the "merely technical" labor of computer programmers. As I have written about extensively elsewhere, the gender

and status associations of the term "coder" would structure debates about the nature of software development, and of software developers, for the next several decades.[49]

Of course, if this direct and uncomplicated relationship between the construction of a flowchart and the coding of a computer program were indeed true, then it was absolutely essential that (1) the flowchart be constructed prior to the writing of the code and (2) that it be an accurate representation of reality.[50] Indeed, as students in introductory courses were constantly being reminded, since "a correctly drawn flowchart allows the actual computer programming to be accomplished [the] cardinal rule of good programming technique is 'flowchart now, code later.'"[51] Equally obvious was the fact that "if the flowchart is incorrect, the program will be coded incorrectly. Therefore the programmer should be sure his flowchart is drawn properly before coding." But contained within this admonishment to "draw correctly" were hints of the difficulty inherent in doing so. The same textbook that declared the flowchart cardinal to programming went on to acknowledge that "determining whether the flowchart is correct or not may prove to be a difficult task."[52] Left unspoken was the question of who was responsible for determining that the flowchart was correct and when in the development process this verification was supposed (or likely) to happen.

This admission that the idealized flowchart diagram did not always correspond well with the messy reality of an actual computer program hinted at growing dissatisfaction with the overly simplistic flowchart-as-blueprint model of software development. This dissatisfaction was as much about the hierarchy of work embodied by the flowchart as it was a critique of the usefulness or accuracy of the flowchart itself. At the same time that flowchart technology was becoming increasingly regimented, routinized, and standardized in the management and educational literature, working programmers were challenging and reshaping its fundamental identity.[53] For them, the flowchart was not so much a top-down design specification produced by scientists or managers aimed at organizing and directing the practical effort of low-level computer programmers as a pragmatic tool for facilitating communication across disciplinary, professional, and organizational boundaries. This renegotiation of the ontological status of flowcharts mirrored a larger shift that was happening in the professional status of programmers and the power relationships within corporate computerization efforts. For a time, however, these changing and, to a certain degree, incommensurate understandings of what a flowchart was and what it was for created confusion and conflict as various actors attempt to understand, accommodate, or resist changes in its meaning and purpose.

When Flowcharts Fail

In one of his characteristic biblical allusions, Frederick Brooks, in his *The Mythical Man-Month*, quoted the rebuke that the Apostle Peter delivered to those Christians who were attempting to impose on the Gentile converts the rules and restrictions of traditional Judaism: "Why lay a load on their backs which neither of our ancestors nor we ourselves were able to carry?"[54] In this case, the load in question was the requirement that programmers maintain a "detailed blow-by-blow flow chart" documenting their program design. The discipline of flowcharting was "more preached than practiced." At best, the flowchart was an educational technology "suitable only for initiating beginners into algorithmic thinking"; more often, it was an "obsolete nuisance" that only hindered the efforts of experienced programmers. His particular objection was to the use of the flowchart as a design document: "The pitiful, multipage, connection-boxed form to which the flow chart has today been elaborated, it has proved to be essentially useless as a design tool—programmers draw flow charts after, not before, writing the programs they describe." He noted as evidence that many software houses had developed special computer programs to produce this supposedly "indispensable design tool" after the fact. In other words, the "original" flowchart was reverse engineered from the completed code base for which it was ostensibly the blueprint.[55]

Although Brooks was a particularly vociferous critic of the flowchart, his was anything but a lone voice crying in the wilderness. The most common complaints had to do with the challenge of finding an appropriate level of granularity: outside of the toy examples that were provided in their introductory flowcharting courses, programmers and analysts in the real world found it difficult to produce flowcharts that were simultaneously detailed enough to be useful guides to development and abstract enough to avoid becoming overly complex, unwieldy, or expensive. As Ned Chapin suggested in his tutorial "Flowcharting with the ANSI standard," a flowchart that contained too much detail was no more useful (or easy to produce) than its equivalent program code. Producing a meaningful flowchart required compressing, condensing, and eliminating details. "But which ones? And how many? A poor choice can render the resulting flow diagram nearly useless."[56]

In his 1963 article "Computer-Drawn Flowcharts," Donald Knuth mocked the oversimplified flowchart too often presented in programming textbooks (see figure 4).[57] But elsewhere he also provided an example, drawn from his very first academic publication, of what he called an "octopus" diagram (see figure 5).[58] The flowchart in question was

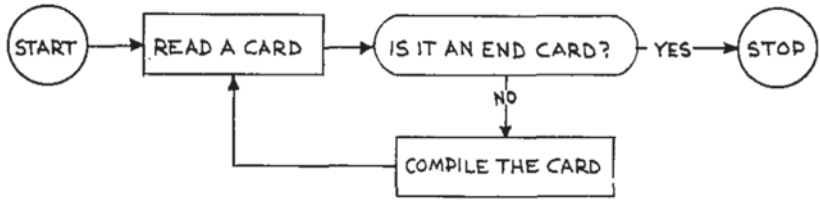


Figure 4. One frequent complaint about flowcharts is that they were too simple. Donald Knuth provided one such example in his 1963 article "Computer-Drawn Flowcharts."

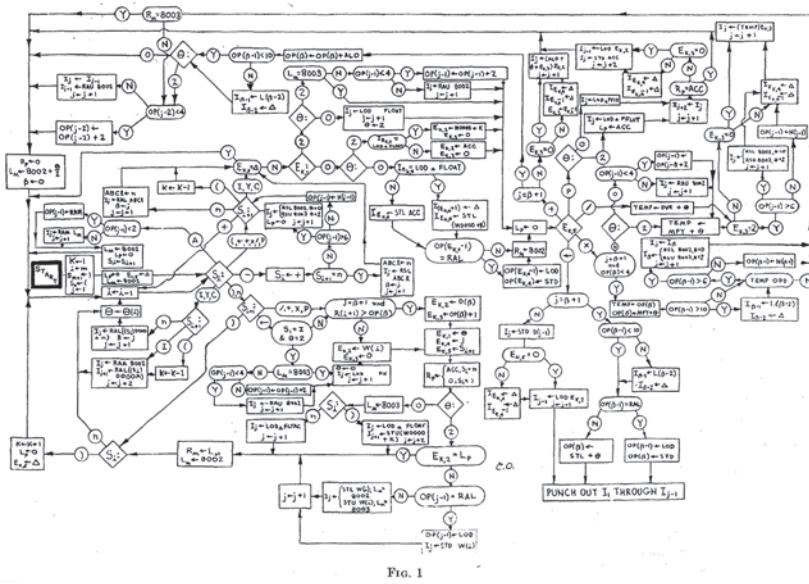


Figure 5. This flowchart, which describes Knuth's 1959 RUNCIBLE compiler, is far too complex to be useful.

allegedly a visual depiction of a compiler that he called RUNCIBLE, but Knuth offered a challenge: "Anyone who believes that flowcharts are the best way to understand a program is urged to look at this example." In retrospect, Knuth argued, it would have been easier for a reader to comprehend Knuth's actual program code than to comprehend the meaning of his flow diagram.[59]

Finding the "right" scale at which to draw a flowchart was as much an organizational as a technological challenge and depended greatly on one's understanding of the relationship between the tasks of analysis, planning, and programming. When the task at hand involved developing a solution to a well-defined mathematical problem (which was true of many of the earliest electronic computing projects), it was perhaps possible for one flowchart to serve both as a design tool for scientists and as a detailed work plan for organizing and directing the practical efforts of computer programmers. In the increasingly complex and sprawling applications being developed in the business context, however, accomplishing both objectives with a single representational technology was difficult, if not impossible.[60] There were simply too many purposes to satisfy and too many acts of translation that needed to happen to make the flowchart legible and meaningful to multiple constituencies.

In the heterogeneous sociotechnical context of corporate data-processing systems, the flowcharts developed by systems analysts, programmers, or other technical specialists were often revealed to be overly simplistic—or optimistic. A 1959 report titled "Business Experience with Electronic Computers," produced by the consulting company Price Waterhouse, described the situation:

> Because the background of the early programmers was acquired mainly in mathematics or other scientific fields, they were used to dealing with well-formulated problems and they delighted in a sophisticated approach to coding their solutions. . . . When they applied their talents to the more sprawling problems of business, they often tended to underestimate the complexities and many of their solutions turned out to be oversimplifications. Most people connected with electronic computers in the early days will remember the one- or two-page flow charts which were supposed to cover the intricacies of the accounting aspects of a company's operations.[61]

In the Price Waterhouse report, managerial disappointment with the flowchart is a reflection of a larger problem of communication and expertise. Over the course of the 1950s, the electronic digital computer, which had originally been imagined as a scientific or military instrument, was being gradually reinvented (both literally and figuratively) by business machines manufacturers such as IBM and Remington Rand as a tool for corporate data processing. The problems that business analysts and programmers worked on "tended to be larger, more highly structured (while at the same time less well-defined), less mathematical, and more tightly coupled with other social and technological systems

than were their scientific counterparts."[62] In this context, it became increasingly clear that computer programming involved more than the mechanical "coding" of a design specification developed by other, more conceptual thinkers. In practice, the work of programmers was more like translation than transcription: in other words, it required not only the ability to speak to multiple communities and across several "languages" (in this case, both human and machine) but also at least some understanding of the underlying problem domain.

The rising professional and intellectual status of programming is represented in the technical and management literature from this period, as well as in the increasing popularity of hybrid and broadly encompassing job titles such as "systems analyst," "programmer/analyst," "software architect," and "software engineer."[63] These analysts and architects still drew flowcharts, but the primary audience for these charts was not computer programmers but managers and end users. These high-level flowcharts were necessarily drawn at a different scale from those intended for programmers. They might have still remained useful as a thinking tool or a design document but not as a detailed blueprint for a work process.

As computer programmers gained more status and autonomy, they assumed more control over low-level design decisions. In the absence of the rigid distinction between "head" and "hand" work imagined by von Neumann and Goldstine, however, the flowchart was not as obviously useful as a means of mapping the complexity of a software project. Even after the invention of high-level programming languages, actually implementing the abstract algorithm described by even the most detailed flowchart required intimate knowledge of the individual compiler being used, the specific hardware platform being targeted, and possibly even the social and organizational configuration of the imagined end user. For the purposes of making or documenting highly detailed design decisions, it was not clear that drawing a flowchart was necessary or helpful. One common complaint among programmers was of the absurdity of the "seven-page program that required a twenty-page flow diagram" to document.[64] For certain purposes, at least, the most useful (and, in all cases, the most accurate) representation of a computer program was the program itself.[65] For a skilled programmer who could read computer code, why bother with the overhead involved with drawing a (largely superfluous) flowchart?

In her analysis of engineering drawings as boundary objects, Beth Bechky shows how these drawings are used to reinforce occupational and status boundaries between engineers and technicians. As with flowcharts, engineering drawings were imperfect ("the technicians, and even the engineers, were aware that the drawings would never truly

represent how to build") and deliberately so.[66] For engineers, the formalization, standardization, and high level of abstraction embodied in the drawings served to differentiate their knowledge (high level, scientific, global) from that of the technicians (machine specific, heuristic, local). According to Bechky, the drawings "*needed* to remain abstract not only for their use as an epistemic tool, but also for reasons of boundary maintenance and task control."[67] Seen in this light, the lack of definitive clarity on the part of these drawings was a feature, not a flaw, "because if every aspect of the work were easily codified and understood, engineers would be unable to maintain their status as experts."[68] In a similar manner, their monopoly of the production of flowcharts, however ambiguous these might be, allowed systems analysts and managers to exert, if only symbolically, their control over the work process of software development. In this sense, boundary objects served not as the "anchors and bridges" originally envisioned imagined by Starr and Griesemer, but as a means of "creating barricades and mazes, protecting and/or privileging different interest groups' frames of reference or occupational positions, rather than creating new shared understandings and perspectives which can inhibit and constrain the possibilities for change."[69]

Objects to Talk With

Even in some imagined world in which a flowchart could be drawn to the ideal scale (and perfectly accurately), its perfection was at best transitory. Flowcharts represented a snapshot in time, the design and structure of the computer program as it existed at that moment. Flowcharts were rendered immediately obsolete whenever any changes were made to either the design or the implementation of the code. As Frederick Hosch observed in his 1977 ACM SIGCSE paper, "Whither Flowcharting,"

> It has been my experience that little real use is made of documentary flowcharts. In the first place, the flowchart of a program that has been in production for any period of time is usually out of date. While the program is modified and corrected, the flowchart is usually ignored, so that even if a beautifully drawn flowchart originally existed, it almost certainly bears no relationship to the program by the time it is needed. If a project manager does succeed in having a flowchart kept up to date, after a few modifications it will be no easier to read than the associated code (although it will undoubtedly be more colorful). The end result is that it is ultimately easier to go directly to the appropriate code than to bother with the flowchart.[70]

Although Hosch's experience with out-of-date flowcharts would have been familiar to any working computer programmer, his characterization of the flowchart as being ex post facto documentation rather than ex ante design reflects a subtle but significant shift in the conventional wisdom about what a flowchart was—and was for. In the model of software development embodied by the "documentary flowchart," the relationship between the user/client and the builder/programmer envisioned by von Neumann and Goldstine was turned on its head: rather than the flowchart being a blueprint drawn up by an expert scientist or manager to be transcribed into computer code by a low-status "flowchart jockey," it was high-level documentation produced by programmers to communicate to managers (and other programmers) the choices that they (the programmers) made in the implementation of their program code.[71] In the earlier model, the flowchart was primarily a technology for translating between man and machine; increasingly, the flowchart served to facilitate human-to-human communication.

There are at least two important developments that help explain the shift from design-oriented to documentary flowcharts. The first, which has already been alluded to, involves the rapid expansion in this period of the size, scope, and sophistication of software projects. As the historian Thomas Hughes has suggested, all large technological systems are really best understood as sociotechnical systems, but this is especially true of software-based technologies.[72] Mapping a complex human cognitive or work process into machine-oriented algorithms involved communication, negotiation, and compromise. Developing large-scale software products involved ongoing (and often contentious) dialogue between a variety of interested parties, including systems analysts, software architects, computer programmers, machine operators, corporate managers, and end users. Savvy software developers quickly realized that "communication with the computer [writing code] is only half of the problem; as we have indicated . . . communication with other humans is just as important."[73]

The second explanation for the shift from flowchart as blueprint to flowchart as documentation has to do with the surprising fragility of software systems: although in theory computer code was immune to the normal processes of wear and tear that plagued other more material devices (it was, in essence, "a technology that could never be broken"), in practice, software systems had to be constantly maintained.[74] What exactly constituted "maintenance" in the context of an ephemeral, largely intangible technology like software is beyond the scope of this article, but suffice it to say that by the early 1970s software maintenance was estimated to represent between 50 and 70 percent of all software

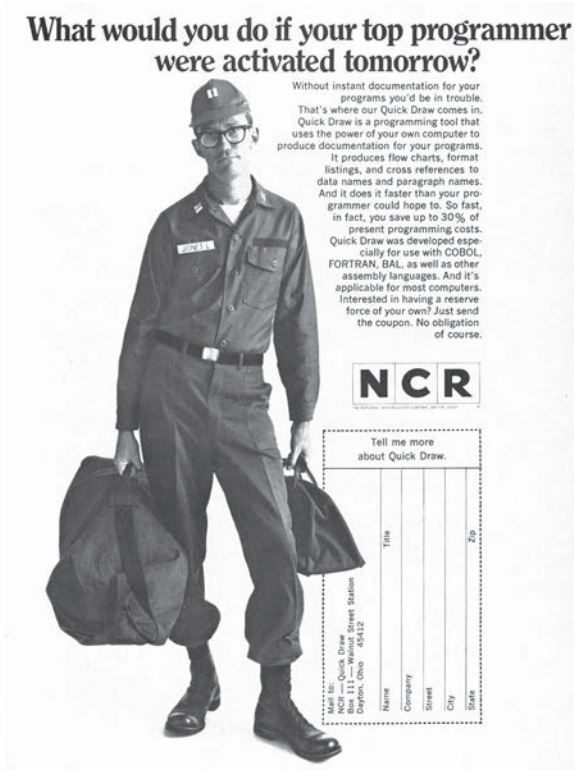


Figure 6. This advertisement for Quickdraw, an NCR software product that reverse engineered a flowchart from previously written application code illustrates one goal of the flowchart, which was to free managers from their dependence on the tacit knowledge of individual programmers.

expenditures.[75] Software maintenance was an enormously expensive and time-consuming endeavor whose central challenges all involved questions of communication: in this case, communications between programmers and managers, between one programmer and another, and even between an individual programmer and his or her future self. Despite efforts to cultivate good code commentary practices and other standardized documentary practices, reading and comprehending computer code remained notoriously difficult—even for the original author. In this context, the flowchart provided a form of visual documentation that facilitated understanding, memory, and conversation.[76] Flowcharts were also a form of insurance against the costs of subsequent maintenance. Considered as Latourian mobiles, flowcharts could communicate across both space and time.[77] In her work on project planning

timelines, Elaine Yakura has suggested that such "temporal boundary objects" make time simultaneously concrete and negotiable among diverse participants. They allow for the shared "expectation of a definite, predictable conclusion" while at the same time allowing different groups the interpretive flexibility to "fill in the gaps" according to their own assumptions and preferences.[78]

That the same flowchart technology could serve both "creative" and "expository" purposes (to borrow from the terminology that Donald Knuth developed) had the potential to cause confusion and consternation.[79] Much of Frederick Brooks's frustration with the flowchart, for example, is based on the premise that flowcharts were intended primarily for creative purposes. The fact that flowcharts rarely corresponded to reality, or were being produced only retrospectively after the code was already written, was proof of their inherent insufficiency as a design tool. The fact that they continued to be required by so many software development managers was a reflection of either unthinking adherence to tradition or bureaucratic incompetence. For those who believed flowcharts to be documentary or expository, however, none of these objections applied. If "flowcharts are primarily intended as tools for human communication," then it was possible for them to be simultaneously beneficial *and* inaccurate, so long as they facilitated meaningful dialogue between designers, users, and programmers.[80] And if the only flowcharts that could be considered definitely true to life were those created by machine and after the fact, then so be it. Lois Haibt, who developed an early tool for reverse engineering flowcharts from already written machine code, argued that "flowcharts serve two important purposes: making a program clear to someone who wishes to know about it and aiding the programmer himself to check that the program as written does the required job." For either of those purposes, the best author of the flowchart was not a human but a machine. A good flowchart ought to "show accurately what the program does rather than what the programmer might expect it to do."[81]

The most prominent advocate of the expository perspective on the flowchart was the software developer and contractor Applied Data Research (ADR). In the mid-1960s, ADR pioneered the concept of the commercial "software product"; prior to this period, software either came bundled with machine by the computer manufacturer or had to be developed in-house or by an independent contract developer.[82] ADR was one such contractor, but in 1964 it began selling an automatic flowcharting program called Autoflow to all of its clients who owned an RCA 501 mainframe computer. Selling the same software program many times to multiple customers was obviously a profitable business model,

but it required a general-purpose application that appealed to a wide variety of users. Since every company that owned or used a computer also made use of flowcharts, Autoflow was an obvious candidate for packaging as the first software product. After ADR developed versions of Autoflow that ran on the increasingly dominant IBM platforms, the company started selling thousands of copies. When IBM started shipping its own free alternative Flowcharter with all of its new machines, ADR launched an antitrust suit that eventually led to IBM's enormously significant "unbundling" decision in 1970.[83]

Although Marty Goetz, the ADR product manager in charge of Autoflow, would later claim that Autoflow was popular because it allowed "strong programmers" to avoid the tedious work of drawing up a flowchart prior to writing their code, the Autoflow marketing literature from this period makes it clear that ADR viewed flowcharts as documentation, not design specification. Although some of Autoflow's touted features were design oriented (using Autoflow would "facilitate analysis" and help diagnose "errors in logic flow and syntax"), the majority were focused on the communications tasks required for long-term software maintenance: Autoflow "provides hardcopy communication medium for all project personnel," "assists management in educating and training junior personnel," and "allows management to . . . review and supervise program activity and quality."[84] The popularity of Autoflow and its many competitors both reified the popularity of the flowchart while at the same time subverting its ostensible function. While aspiring programmers were still being indoctrinated into the belief that the flowchart was a blueprint, in most corporations the principal purpose of the flowchart had largely shifted from design to documentation. What is particularly interesting about this shift is that it does not involve any change in the structure of the flowchart: the standardized visual language that emerged in the early 1960s remains remarkably stable over time. The technology does not change; it is simply imagined and interpreted differently.[85] For those who imagined the flowchart as a design document, a technology like Autoflow represented a fundamental subversion of the design process; for those who regarded the flowchart as a technology for documentation, Autoflow was not only appropriate but desirable.

And so we see that in the corporate context, at the very least, the flowchart survived in large part because, despite its limitations, it was able to acquire new meanings over time that prevented it from becoming obsolete or irrelevant. By extending the notion of the boundary object to include not only fixed but discursive meanings (i.e., by allowing for multiple, even contradictory "readings"), as Cliff Oswick and Maxine Robertson have done, we can accommodate these multiple meanings

of the flowchart without requiring any one of them to be absolute or exclusive.[86] Different parties could believe different things about what flowcharts were "really" meant to accomplish. What matters is that the one object could be shared across multiple communities in ways that were relevant and productive. In fact, we might argue that it was the interpretive flexibility of the flowchart that provided it with its conscriptive power. Flowcharts might individually have been fallible, but collectively they were necessary. Not only were they a necessary tool for facilitating communication, but they also served as a form of implied contract between the various actors in the software development project. Having the client or end user sign off on a flowchart helped protect the project manager and programmers against "feature creep." At the same time, the flowchart provided some guarantee to the client or manager that the programmers would build the system that the client or manager had requested rather than the one that the programmers thought was best or most interesting. In a period in which many organizations worried that they had lost control over the process of technological development and that the "computer boys" had taken over, the idea of the flowchart as a contract was reassuring.[87]

Flowcharts Considered Harmful

In March 1968 the noted computer scientist (and soon to be Turing Award laureate) Edsger Dijkstra wrote a short but influential letter to the editors of the *Communications of the ACM* in which he urged that the go to statement be considered harmful. The overuse of this popular programming construct, argued Dijsktra, had such "disastrous effects" on the writing of logically correct, legible, and maintainable computer code that it "should be abolished from all 'higher level' programming languages."[88] While there were equally prominent computer scientists who disagreed vehemently with Dijkstra's assessment, his letter provoked a lively debate that ultimately culminated in the emergence of the Structured Programming paradigm, one of the most significant innovations in software development of the next several decades. As with the larger "software engineering" movement of which it was a part, structured programming was both a specific technical approach to designing and writing code and a statement about computer programming as an intellectual and occupational activity. To write unstructured code, according to Dijkstra and his supporters, was not simply to create programs that were unwieldy, error prone, and difficult to maintain but to demean the status of the discipline and to mark the programmer as unprofessional.[89]

Although the focus of Dijkstra's critique of contemporary programming practices focused on the go to statement, the flowchart was indirectly implicated.[90] The go to statement was used to transfer control of a program from one line of code to another. Whereas invoking a subroutine or a function returned control (and generally a value) to the original calling routine, the go to statement served as a one-way jump (or branch). As such, it corresponded directly to the decision node of a flowchart. In fact, some argued that the branching structure of the flowchart *encouraged* the use of go to statements.[91] "Flowcharts look like spaghetti, and therefore encourage spaghetti-like programs. . . . [T]hey provide irresistible temptations to jump into the middle of otherwise working construction, violating their preconditions and generating untraceable bugs."[92] Others simply identified both practices as being similarly counterproductive to well-structured programming: "Flowcharts, like goto's, belong to the class of objects that are detrimental to good programming."[93] A series of popular books published in the 1970s and organized around "programming proverbs" suggested that "the case against program flowcharts is similar to the case against GOTO. The lines and arrows can easily lead the user into a highly sequential mode of thinking."[94] Once the "structured programming approach is fully adopted, the need for flow charts will be reduced," argued one 1975 article in the ACM SIGCPR (Special Interest Group on Computer Programming Research).[95]

The debate about structured programming focused intense scrutiny on the flowchart. Some computer scientists attempted to reform the technology. Although "conventional flowcharts [were] a hindrance to structured programming," they nevertheless had value, and at the very least were ubiquitous in practice, and so perhaps they could be reformed.[96] In 1973 Ben Schneiderman and Ike Nassi published their proposal for "flowchart techniques for structured programming."[97] The representational system that they developed eventually became known as the Nassi-Schneiderman diagram, and it bears only a vague resemblance to the traditional flowchart. But by this period even proposing an article on flowcharts provoked what Schneiderman later called the "most brutal rejection letter" that he ever received. An anonymous reviewer for the *Communications of the ACM* not only recommended that the ACM never publish any more articles on flowcharts ("flowcharts [were] a crutch we invented to try to understand programs written in a confusing style") but also suggested that "the best thing the authors could do is collect all copies of this technical report and burn them, before anybody reads them."[98] The prolific writer of systems analysis and computer programming textbooks, Ned Chapin, also proposed his own version of a structured flowchart that he called "Chapin Charts."[99]

For the most part, however, the structured programming movement signaled the beginning of the end of the traditional flowchart. The late 1970s and early 1980s witnessed a spate of empirical research on flowcharts, the most significant of which was a 1977 study that concluded, "No statistically significant difference between flowchart and nonflowchart groups has been shown, thereby calling into question the utility of detailed flowcharting."[100] By the beginning of the 1980s, the flowchart was a defunct technology—at least in terms of the academic literature.[101] Today most programmers use other forms of software visualizations, from Bachmann diagrams to UML diagrams, to attempt to map the complexity of software systems development.

The Flowchart Is Dead. Long Live the Flowchart!

Although by the late 1970s most academic computer scientists had dismissed the flowchart as being both incorrect and irrelevant, as a representational technology flowcharts have proven remarkably long-lived. Flowcharts are still widely used in introductory programming courses, particularly those aimed at nonspecialists.[102] They are also enormously popular in contemporary management literature for many of the same reasons that they were popular with managers in the early decades of computing: flowcharts embody the idealized separation of head and hand that is essential to modern managerial capitalism. Even among nonprogrammers, the flowchart is one of the most visible symbols of the pervasive influence of the computational mind-set on popular culture. Flowcharts have become one of the most accessible forms of visual humor, for example, as even the most cursory search on the Internet will reveal: "Should I do my laundry?" "Do I deserve a cookie?" and "How to write an academic article" are all examples of the ways in which flowcharts are mobilized as visual illustrations in a wide variety of contexts. The fact that such charts are assumed to be instantly recognizable and readily understood by a wide variety of audiences is a testament to the remarkable degree to which an obsolete software development technology has survived and adapted to a changing environment.

The unexpected durability of flowcharts is significant for historians for several reasons. In recent years it has become clear to historians of computing that it is the history of software, not the computer itself, that is most essential to understanding the larger economic, social, and cultural significance of the "digitization" of modern society.[103] But one of the many challenges associated with writing the history of software is that software is largely invisible, intangible, and ephemeral. Although

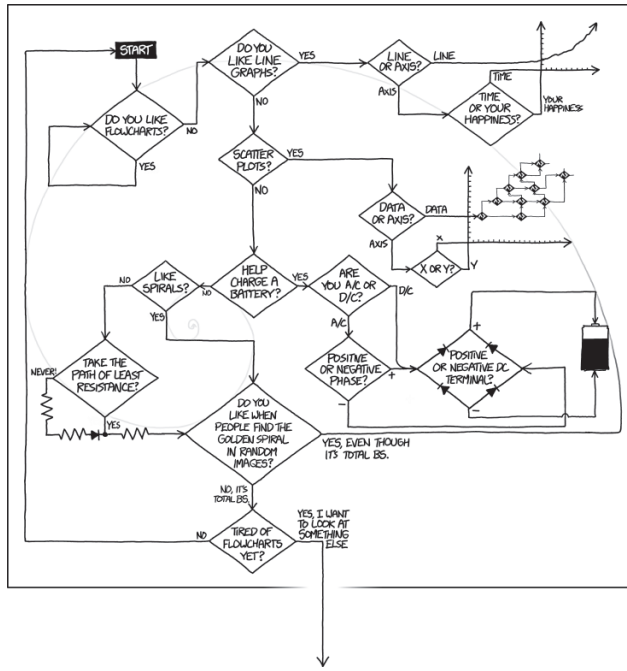


Figure 7. This cartoon from the XKCD webcomic (xkcd.com) is but one example of the adaptation of the flowchart into popular culture. The flowchart is one of the most durable and recognizable visual cultural expressions of the pervasiveness of the computational mind-set.

software is arguably the primary interface through which most of us perceive and experience the electronic digital computer, software leaves surprisingly few material traces of its existence or influence. The computer code that makes up software is constantly evolving and being rewritten—or rewriting itself; program listings and source code are rarely archived in a form accessible or legible to historians; magnetic tape, floppy disks, and CD-ROMS have notoriously short life spans, and even when they survive, it is difficult or impossible to find the hardware required to read or execute the software that they contain. Documentation and manuals are rendered obsolete by even the most minor software updates and are often deliberately destroyed or discarded. In other words, software history is lacking in material resources and culture. Flowcharts are one of the few tangible remnants from this critical period in software history, and historians of computing have not yet learned to make effective use of them.

In addition to being quite literally durable in ways that other forms of software are not, flowcharts provide a unique record of the larger software processes and organizations of which computer code is but one component. A well-written computer program is, in theory at least, self-documenting; that is, the computer code itself contains its own complete written specification. And yet despite the computer scientist Donald Knuth's famous claim that computer programs, like literature, were meant to be read by humans as much as by machines, for the most part computer programs are too arcane and idiosyncratic for even their original authors to fully understand.[104] Flowcharts allow us to "see" software in ways that are otherwise impossible. Not only do they provide a visual record of the design of software systems (albeit, as we have seen, never an entirely accurate record), flowcharts can also serve as a map of the complex social, organizational, and technological relationships that comprise most large-scale software systems. In this sense, the many liabilities of flowcharts identified by contemporaries—that they were imperfect, imprecise, mutable, and contested—become virtues for the historians. As David Nicolini, Jeanne Mengis, and Jacky Swan note in their work on bioreactors as boundary objects, the "career" of such objects "may not look like an orderly trajectory as much as a messy, iterative journey." It is as "triggers of contradictions and negotiation," rather than as stable, mutually agreed upon representations of reality, that boundary objects help "explain the potentially conflictual nature of collaborative activity."[105] To acknowledge that any particular flowchart satisfied no one entirely and was the subject of constant critique, conflict, and negotiation is simply to recognize that, like all maps, the flowchart represented only a selective perspective on reality. Interpreted creatively by historians, however, such maps become a means of unraveling the assumptions built into software systems about who would use them, how, and for what purposes. They become "epistemic objects" not only for our historical actors but also for historians as analysts.[106]

Notes

1. Paul DesJardins and Dave Graves, "The Programmer's Primer and Coloring Book," *Datamation* 9, no. 9 (1963): 47–50.

2. I. G. Seligsohn, *Your Career in Computer Programming* (New York: Julian Messner, 1967); George Gleim, *Program Flowcharting* (New York: Holt, Rinehart and Winston, 1970).

3. Robert J. Rossheim, "Report on Proposed American Standard Flowchart Symbols for Information Processing," *Communications of the ACM* 6, no. 10 (1963): 599–604.

4. Saul Gorn, "Conventions for the Use of Symbols in the Preparation of Flowcharts for Information Processing Systems," *Communications of the ACM* 8, no. 7 (1965): 439–40.

5. G. K. Gupta, "Computer Science Curriculum Developments in the 1960s," *IEEE Annals of the History of Computing* 29, no. 2 (2007): 40–54.

6. IBM Corporation, *Introduction to IBM Data Processing Systems* (White Plains: IBM Technical Publications, 1969).

7. Gleim, *Program Flowcharting*.

8. Alfonso F. Cardenas, "Technology for Automatic Generation of Application Programs: A Pragmatic View," *MIS Quarterly* 1, no. 3 (1977): 49–72.

9. J. M. Yohe, "An Overview of Programming Practices," *ACM Computing Surveys* 6, no. 4 (1974): 221–45.

10. Frederick Brooks, *The Mythical Man-Month* (Reading, MA: Addison, 1982).

11. Wayne LeBlanc, "Standardized Flowcharts," *ACM SIGDOC Asterisk Journal of Computer Documentation* 4, no. 8 (1978): 18–28.

12. Donald E. Knuth, "Computer-Drawn Flowcharts," *Communications of the ACM* 6, no. 9 (1963): 555–63.

13. Brooks, *The Mythical Man-Month*.

14. M. Goetz, "Memoirs of a Software Pioneer: Part 1," *IEEE Annals of the History of Computing* 224, no. 1 (2002): 43–56.

15. Susan Leigh Starr and James R. Griesemer, "Institutional Ecology, 'Translations' and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology, 1907–39," *Social Studies of Science* 19, no. 3 (1989): 387–420.

16. Ibid., 12.

17. Kathryn Henderson, "Flexible Sketches and Inflexible Data Bases: Visual Communication, Conscription Devices, and Boundary Objects in Design Engineering," *Science, Technology & Human Values* 16, no. 4 (1991): 456, emphasis added.

18. Ibid.

19. Michael S. Mahoney, "What Makes the History of Software Hard," *IEEE Annals of the History of Computing* 30, no. 3 (2008): 8–18.

20. Arnold Ditri and Donald Wood, "The End of the Beginning—the Fizzle of the 'Computer Revolution'" (New York: Touche Ross and Company, 1969).

21. Nathan Ensmenger, *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise* (Cambridge, MA: MIT Press, 2010).

22. Haskell Curry and Willa Wyatt, "A Study of Inverse Interpolation of the Eniac," Aberdeen Ballistics Research Laboratory, 1946.

23. Haskell Curry, "On the Composition of Programs for Automatic Computing," Naval Ordnance Laboratory, 1949; John von Neumann and Herman Goldstine, "Planning and Coding of Problems for an Electronic Computing Instrument," Institute for Advanced Study, Princeton, NJ, 1948.

24. S. J. Morris and O. C. Z. Gotel, "Flow Diagrams: Rise and Fall of the First Software Engineering Notation," in *Diagrams '06: Proceedings of the 4th International Conference on Diagrammatic Representation and Inference* (Berlin: Springer-Verlag, 2006).

25. Ibid., 3.

26. James M. Wilson, "Gantt Charts: A Centenary Appreciation," *European Journal of Operational Research* 149, no. 2 (2003): 430–37.

27. Von Neumann and Goldstine, "Planning and Coding."

28. Stephen Morris and Orlena Gotel, "The Role of Flow Charts in the Early Automation of Applied Mathematics," *BSHM Bulletin: Journal of the British Society for the History of Mathematics* 26, no. 1(2011): 44–52.

29. Morris and Gotel, "Flow Diagrams."

30. Sperry Rand Corporation, *An Introduction to Programming the UNIVAC 1103A and 1105 Computing Systems* (New York: Remington Rand Univac, 1958).

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

32. Sherry Turkle, *Evocative Objects: Things We Think With* (Cambridge, MA: MIT Press, 2011).

33. IBM Corporation, *Flowcharting Techniques* (White Plains, NY: IBM Corporation, 1971).

34. Gene Bylinsky, "Help Wanted: 50,000 Programmers," *Fortune* 75 (1967): 141–68; Eloina Paleaz, "A Gift from Pandora's Box: The Software Crisis" (PhD diss., University of Edinburgh, 1988); Nathan Ensmenger, "The 'Question of Professionalism' in the Computer Fields," *IEEE Annals of the History of Computing* 23, no. 4 (2001): 56–73.

35. IBM, *Flowcharting Techniques*.

36. Bruno Latour, "Visualization and Cognition: Drawing Things Together," *Knowledge and Society* 6 (1986): 1–40.

37. S. D. Conte et al., "An Undergraduate Program in Computer Science—Preliminary Recommendations," *Communications of the ACM* 8, no. 9 (1965): 543–52; Robert Ashenhurst, "Curriculum Recommendations for Graduate Professional Programs in Information Systems," *Communications of the ACM* 15, no. 5 (1972): 363–98.

38. Ned Chapin, "Flowcharting with the ANSI Standard: A Tutorial," *ACM Computing Surveys* 2, no. 2 (1970): 119–46.

39. Robert McFarland, "Electronic Power Grab," *Business Automation* 12 (1965): 30–39; Harry Stern, "Information Systems in Management Science," *Management Science* 13, no. 8 (1970): 540–42.

40. Edward Markham, "EDP Schools: An Inside View," *Datamation* 14, no. 4 (1968): 22–27.

41. John Hanke, William Boast, and John Fellers, "Education and Training of a Business Programmer," *Journal of Data Management* 3, no. 6 (1965): 38–53.

42. F. A. Hosch, "Whither Flowcharting?," *ACM SIGCSE Bulletin* 9, no. 3 (1977): 66–73.

43. Thomas McInerney and Andre Vallee, *A Student's Guide to Flowcharting* (Englewood Cliffs, NJ: Prentice-Hall, 1973).

44. A. R. Feinstein, "An Analysis of Diagnostic Reasoning. 3. The Construction of Clinical Algorithms," *Yale Journal of Biology and Medicine* 47, no. 1 (1974): 5–32; Patrica Baucom, "Software Blueprints," in *ACM '78: Proceedings of the 1978 Annual Conference* (New York: ACM Press, 1978).

45. Ronald Elliott, *Problem Solving and Flowcharting* (Reston, VA: Reston Publishing, 1972).

46. Gleim, *Program Flowcharting*.

47. Cyrus F. Gibson and Richard L. Nolan, "Organizing and Managing Computer Personnel: Conceptual Approaches for the MIS Manager," in *Proceedings of the Eleventh Annual SIGCPS Computer Personnel Research Conference*, SIGCPR '73 (New York: ACM Press, 1973), 19–45.

48. Thomas Schriber, *Fundamentals of Flowcharting* (New York: Wiley and Sons, 1969).

49. Nathan Ensmenger, "Making Programming Masculine," in *Gender Codes: Why Women Are Leaving Computing* (Hoboken, NJ: Wiley and Sons, 2010).

50. John Lenher, *Flowcharting* (Philadelphia: Auerbach Publishers, 1972); Mario Farino, *Flowcharting* (Englewood Cliffs, NJ: Prentice-Hall, 1970).

51. Eliot, *Problem Solving and Flowcharting*, 5.

52. Ibid., 73.

53. Hosch, "Whither Flowcharting?"

54. Acts 15:10, Good News Bible translation, quoted in Brooks, *The Mythical Man-Month* (1982).

55. Brooks, *The Mythical Man-Month*, 194.

56. Chapin, "Flowcharting," 143.

57. Knuth, "Computer-Drawn Flowcharts."

58. Donald Knuth, "Structured Programming with Go To Statements," *Computing Surveys* 6, no. 4 (1974): 261–301.

59. Quote is from Knuth, "Structured Programming," 292. The flowchart described is from Donald Knuth, "RUNCIBLE—Algebraic Translation on a Limited Computer," *Communications of the ACM* 2, no. 11 (1959): 18–21.

60. G. J. Nutt, "The Computer System Representation Problem," in *The 1st Symposium on Simulation of Computer Systems* (Piscataway, NJ: IEEE Press, 1973).

61. B. Conway, J. Gibbons, and D. E. Watts, "Business Experience with Electronic Computers: A Synthesis of What Has Been Learned from Electronic Data Processing Installations," report produced by Price Waterhouse, New York, 1959, 82.

62. Ensmenger, *The Computer Boys Take Over*, 59.

63. Ibid.

64. Chapin, "Flowcharting," 142.

65. K. C. Waddel and J. H. Cross, "Survey of Empirical Studies of Graphical Representations for Algorithms," in *CSC '88: Proceedings of the 1988 ACM Sixteenth Annual Conference on Computer Science* (New York: ACM Press, 1988).

66. Beth A. Bechky, "Object Lessons: Workplace Artifacts as Representations of Occupational Jurisdiction," *American Journal of Sociology* 109, no. 3 (2003): 720–52.

67. Beth A. Bechky, "Object Lessons," in *The Knowledge Economy and Lifelong Learning*, ed. D. W. Livingstone and David Guile, vol. 4, The Knowledge Economy and Education (Sense Publishers, 2012), 229–56, emphasis added.

68. Ibid.

69. Starr and Griesemer, "Institutional Ecology"; Cliff Oswick and Maxine Robertson, "Boundary Objects Reconsidered: From Bridges and Anchors to Barricades and Mazes," *Journal of Change Management* 9, no. 2 (2009): 179–93.

70. Hosch, "Whither Flowcharting?," 70.

71. *2nd RAND Symposium* (1959), CBI 78, box 1, folder 1, Archives of the Charles Babbage Institute, University of Minnesota, Minneapolis.

72. Wiebe Bijker, Thomas Hughes, and T. J. Pinch, eds., *The Social Construction of Technological Systems* (Cambridge, MA: MIT Press, 1987).

73. Yohe, "An Overview of Programming Practices."

74. Nathan Ensmenger, "Software as History Embodied," *IEEE Annals of the History of Computing* 31, no. 1 (2009): 88–91.

75. Richard Canning, "The Maintenance 'Iceberg,'" *EDP Analyzer* 10, no. 10 (Vista, CA: Canning Publications, 1972): 1–14.

76. T. C. Willoughby and A. D. Arnold, "Communicating with Decision Tables, Flowcharts, and Prose," *SIGMIS Database* 4 (1972).

77. Latour, "Visualization and Cognition."

78. Elaine Yakura, "Charting Time: Timelines as Temporal Boundary Objects," *Academy of Management Journal* 45, no. 5 (2002): 956–70.

79. Knuth, "Computer-Drawn Flowcharts."

80. Yohe, "An Overview of Programming Practices."

81. Lois Haibt, "A Program to Draw Multilevel Flow Charts," *Papers Presented at the 1959 Western Joint Computer Conference* (New York: ACM Press, 1960), 131.

82. Thomas Haigh, "Software in the 1960s as Concept, Service, and Product," *IEEE Annals of the History of Computing* 24, no. 1 (2002): 5–13.

83. Goetz, "Memoirs."

84. Gerardo Con Diaz, "Intangible Inventions: Patents and the History of Software Development, 1945–1985" (PhD diss., Yale University, 2016), 139.

85. One particularly interesting example of this interpretive flexibility also involves Marty Goetz, the creator of Autoflow. In 1965 Goetz had applied for a patent for a software-based sorting application and had provided, as the primary description of his invention, the flowchart of his algorithm. In 1968 he was granted the first software patent ever awarded, in the process defining yet another meaning for the flowchart, this time as a form of legal documentation.

86. Oswick and Robertson, "Boundary Objects Reconsidered," *Journal of Change Management* 9, no. 2 (2009): 179–93.

87. Ensmenger, *The Computer Boys Take Over.*

88. Edsger Dijkstra, "Go To Statement Considered Harmful," *Communications of the ACM* 11 (1968): 147.

89. Ensmenger, *The Computer Boys Take Over.*

90. E. Dijkstra, Trip Notes, 1965 (EWD 572). Quoted in http://kazimirmajorinc.com/Documents/Why-Dijkstra-didnt-like-Lisp/index.html.

91. Linda Jones and David Nelson, "A Quantitative Assessment of IBM's Programming Productivity Techniques," in *DAC '76 Proceedings of the 13th Design Automation Conference* (New York: ACM Press, 1976).

92. C. H. Lindsey, "Structure Charts a Structured Alternative to Flowcharts," *ACM SIGPLAN Notices* 12 (1977): 36.

93. Hosch, "Whither Flowcharting?," 67.

94. Quote from Henry Ledgard and John Tauer, *Pascal with Excellence* (Englewood Cliffs, NJ: Prentice Hall, 1986), 208. See also Louis Chmura and Henry Ledgard, *Cobol with Style: Programming Proverbs* (Rochelle Park, NJ: Hayden Book Company, 1976); Henry Ledgard and L. J. Chmura, *FORTRAN with Style: Programming Proverbs* (Rochelle Park, NJ: Hayden Book Company, 1978).

95. Angel Vargas, Luis Kornhauser, and Javier Olivares, "Development of a Job Description for Unionized Programmers," in *'75 Proceedings of the Thirteenth Annual SIGCPR Conference* (New York: ACM Press, 1975), 135.

96. Lindsey, "Structure Charts," 36; LeBlanc, "Standardized Flowcharts."

97. Ike Nassi and Ben Schneiderman, "Flowchart Techniques for Structured Programming," *ACM SIGPLAN Notices* 8, no. 8 (1973): 12–26.

98. "Letter from ACM Communications to B. Shneiderman" (1972), accessed April 28, 2015, https://www.cs.umd.edu/hcil/members/bshneiderman/nsd/rejection_letter.html.

99. Ned Chapin, "New Format for Flowcharts," *Software: Practice and Experience* 4 (1974); Ned Chapin, "Some Structured Analysis Techniques," *ACM SIGMIS Database* 10 (1978).

100. Ben Shneiderman et al., "Experimental Investigations of the Utility of Detailed Flowcharts in Programming," *Communications of the ACM* 20 (1977): 373; H. R. Ramsey and M. E. Atwood, "Flowcharts vs. Program Design Languages: An Experimental Comparison," *Communications of the ACM* 26 (1983); J. B. Brooke and K. D. Duncan, "Experimental Studies of Flowchart Use at Different Stages of Program Debugging," *Ergonomics* 23, no. 11 (1980): 1057–91; Bill Curtis, "A Review of Human Factors Research on Programming Languages and Specifications," *Proceedings of the 1982 Conference on Human Factors in Computing Systems* (New York: ACM Press, 1982).

101. Maarten van Emden, "Flowcharts, the Once and Future Programming Language," accessed May 23, 2015, https://vanemden.wordpress.com/2012/04/08/flowcharts-the-once-and-future-programming-language/.

102. Anil Bikas Chaudhuri, *The Art of Programming Through Flowcharts & Algorithms* (Bangalore: Firewall Media, 2005); Kang Zhang, *Software Visualization: From Theory to Practice* (Boston: Kluwer Academic, 2003).

103. Nathan Ensmenger, "The Digital Construction of Technology: Rethinking the History of Computers in Society," *Technology and Culture* 53, no. 4 (2012): 753–76.

104. Donald Knuth, *Literate Programming* (Stanford, CA: Center for the Study of Language, 1992).

105. David Nicolini, Jeanne Mengis, and Jacky Swan, "Understanding the Role of Objects in Cross-Disciplinary Collaboration," *Organization Science* 23 (2012): 621.

106. Karin Knorr Cetina, "Sociality with Objects Social Relations in Postsocial Knowledge Societies," *Theory, Culture & Society* 14, no. 4 (1997): 1–30.