Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering

STUART SHAPIRO

For the last quarter of a century, software technologists have worked to address the "software crisis" identified in the 1960s. Their efforts have focused on a number of different areas, but have often been marked by the search for singular "best" solutions. However, the fundamental nature of software—involving basic and poorly understood problem-solving processes combined with unprecedented and multifaceted complexity—weighs heavily against the utility of singular approaches. Examination of the discourse of software technologists in a number of key professional and trade journals over the last 25 years illuminates various disputes central to the development of software engineering and highlights the necessity of a more pluralistic mindset revolving around synthesis and trade-offs.

Introduction

By the end of the 1960s, it was becoming obvious to the computing community that software was a big problem and growing bigger. While the cost of hardware steadily declined even as hardware performance steadily increased, software seemed headed in the opposite direction. Large software projects were consistently late, over budget, and full of defects. Today, the complaints remain much the same. This is not to deny that the current situation represents a drastic improvement over the state of affairs that prompted the North Atlantic Treaty Organization (NATO) software engineering conferences of the late 1960s. What were problems then are still problems now, but they tend to be (but not always) relatively less frequent and less disastrous, especially in the context of the vastly expanded size and ambitions of much contemporary software. Indeed, Andrew Friedman has argued that while software was previously the key stumbling block for systems development, the focus has now shifted to user needs.[1] While Friedman is right to call attention to the current emphasis on user needs, though, his periodization based on successive bottlenecks is a little too tidy and belies the complexity and heterogeneity of the issues and arguments that have surrounded systems development from the early days to the present.

Events of the late 1960s enhanced comprehension of the breadth and depth of the problems plaguing software development while only hinting at solutions. Still, the growing recognition that a collection of interrelated problems existed, together with an awareness of the importance of process, constituted a turning point in the history of software technology. The "software crisis" provided a context for the development of software technology in the 1970s and beyond.

From the 1960s onward, many of the ailments plaguing software could be traced to one principal cause—complexity engendered by software's abstract nature and by the fact that it constitutes a digital (discrete state) system based on mathematical logic rather than an analog system based on continuous functions. This latter characteristic not only increases the complexity of software artifacts but also severely vitiates the usefulness of traditional engineering techniques oriented toward analog systems.[2] Although computer hardware, most notably integrated circuits, also involves great complexity (due to both scale and state factors), this tends to be highly patterned complexity that is much more amenable to the use of automated tools. Software, in contrast, is characterized by what Fred Brooks has labelled "arbitrary complexity."[3]

The complexity associated with software technology, however, is not that straightforward. Instead, it involves numerous facets and dimensions. Complexity's various contexts include algorithmic efficiency, the structure of procedures and data, and the psychological effort of problem comprehension, translation, and system design. Those contexts have manifested themselves in issues concerning structured programming, software metrics, program verification, formal methods generally, programming languages, the software life cycle, and programming environments. No solution aimed at a single area could provide the degree of relief many were seeking. Moreover, agreeing on singular approaches with respect to any of these issues also frequently proved difficult in the face of incommensurable philosophies and inescapable trade-offs. Recognition of the futility of technical singularity in any realm of software technology was slow in dawning.

The basic nature of software vis-à-vis hardware complicates matters in this respect. Hardware, in computing and in general,

1058-6180/97/$10.00 © 1997 IEEE

refers to something solid, inflexible, and not easily altered. Software is soft precisely because its descriptors—ephemeral, flexible, malleable—contrast with those of hardware. They make software an excellent source of leverage—the power to act effectively. The ability to fashion a means of problem solution adapted to the specifics of a problem constitutes leverage of a high order. Obviously, software by itself, while maximizing flexibility, is of limited utility since it then amounts to only a set of instructions on how to accomplish a certain task. While such codification is useful, it fails to supply the leverage that results when it is combined with mechanization. Similarly, a special-purpose machine with no capacity for variation is of little use outside its narrow area of application. Hardware and software have a synergistic effect on problem solution. The former mechanizes narrowly but deeply, the latter mechanizes broadly but shallowly. Together, they are capable of exerting a high degree of leverage on problems.

The trade-off between breadth and depth also pertains to software per se. Programming languages, application programs, tools, methods, and environments (including cultural factors) all embody it. The essence of the tension is the degree to which any given piece of software technology "fits" the circumstances surrounding its use. To the extent that the piece of technology is circumstance-specific, it incorporates knowledge and characteristics that help it function more effectively, affording the user greater problem-solving leverage under those particular conditions. However, the corollary to this property is that the technology becomes correspondingly less suitable for use in other situations, depending on how far they deviate from the original target situation. If the original circumstances are narrowly defined, problematic deviation occurs relatively rapidly, while if the circumstances are more broadly defined, deviation is less rapid. However, by the same token, software technology suitable for a wide range of circumstances will afford less leverage by way of highly particular knowledge embodied within the technology. This, then, is the essential tension within software in all its aspects: the trade-off between specificity and generality. It underlies software technology in all its manifestations.

The powerful desire for dramatic singular solutions therefore hindered rather than helped software technologists. Difficulties were exacerbated by the exaggerated and sweeping claims that often accompanied particular techniques, claims that frequently generated an equal and opposite reaction. The problems plaguing software technology were usually fuzzy, variable, and multifaceted, and thus rarely proved amenable to any one approach; instead, they demanded hybrid and adaptive solutions. Messy responses, though, were less than satisfying to those who sought sweeping breakthroughs. Effective action required a spirit of pragmatic accommodation, a kind of technical pluralism that was not always evident.

What follows is not intended to be a comprehensive history of software engineering since the engineering appellation was first formally used. Rather, it is an attempt to capture the flavor of some of the key concerns and arguments as they have manifested themselves in the discourse contained within some of the most influential professional and trade literature. These sources serve as a primary forum in which the issues of the day are raised and debated. Clearly, though, this poses a couple of methodological problems.

The first methodological problem is the unavoidable one of source self-selection. Those individuals who submit articles or write letters are by definition moved to do so by a variety of motivations, ranging from the pursuit of tenure to passionately held views on a certain topic. However, this does not automatically render their views unrepresentative. Moreover, while a number of names appear on a regular basis, a larger number appear on a much more ad hoc basis. In other words, while a body of elites is clearly in evidence, so, too, is wide participation from the rest of the computing and software communities.

The powerful desire for dramatic singular solutions therefore hindered rather than helped software technologists.

The second methodological difficulty arises out of the circumscribed geographic range of the sources. This reflects several practical limitations, including language barriers and time constraints. It most certainly should not be taken as implying the insignificance of work done outside the United States and Great Britain. Two factors, though, in the one case explain and in the other case mitigate this bias. With respect to the former, the United States has long been and continues to be the acknowledged world leader in software technology. In terms of the latter, many of the publications surveyed circulate widely outside their country of origin and routinely carry articles, news, and correspondence from around the world. Therefore, building this study on the particular literature employed seems eminently justifiable.

With the exception of the following section discussing the NATO software engineering conferences, the organization of this essay is thematic but chronological for each theme. The first theme focuses on the central role of complexity in software technology and its manifestation in design and measurement strategies. This will be followed by discussion of the debate over program verification, leading into an examination of the formal methods movement more generally. Issues arising out of programming languages, life cycle models, and programming environments will then be discussed. All of this will highlight the problem of making choices in a pluralistic technological world, a topic that will be addressed toward the end. While this work does not assume expertise in software engineering on the part of the reader, some basic appreciation of software technology would undoubtedly prove helpful in making sense of it.

Setting the Stage: The NATO Conferences

The NATO software engineering conferences of 1968 and 1969 set an agenda and a context that even today continue to make their presence felt.[4] In the fall of 1967, the NATO Science Committee had established a Study Group on Computer Science to assess the field. The attention of the study group was drawn to the problems endemic in the area of software. Around the end of 1967, it recommended that a working conference be held on software engineering. The conference report noted that "the phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering."[5] International experts from a wide variety of backgrounds gathered in Garmisch, Germany, in October 1968 to consider problems in the design, production, and maintenance of software.

Although the participants agreed that problems existed, opinions varied on the seriousness of the "software crisis" and the extent of the problems. Typical of the exchanges was the one between Ken Kolence of Boole and Babbage Inc. and Douglas Ross from the Massachusetts Institute of Technology (MIT). Kolence did not like the use of the word *crisis*. "It's a very emotional word. The basic problem is that certain classes of systems are placing demands on us which are beyond our capabilities and our theories and methods of design and production at this time. There are many areas where there is no such thing as a crisis...."[5,p.121] Ross responded that "it makes no difference if my legs, arms, brain and digestive tract are in fine working condition if I am at the moment suffering from a heart attack. I am still very much in a crisis."[5,p.121] Most, however, could agree with E.E. David of Bell Laboratories that "production of large software has become a scare item for management. By reputation it is often an unprofitable morass, costly and unending."[5,p.67]

With regard to the underlying causes of the crisis, at least some of the participants appreciated the ephemeral nature of the medium and the difficulties it created. David noted that with respect to problems of scale,

> the uninitiated sometimes assume that the word "scale" refers entirely to the size of code. . . . This dimension is indeed a contributory factor to the magnitude of the problems, but there are others. One of increasing importance is the number of different, non-identical situations which the software must fit. Such demands complicate the tasks of software design and implementation, since an individually programmed system for each case is impractical.[5,pp.68-69]

Moreover, he noted, "there is no theory which enables us to calculate limits on the size, performance, or complexity of software. There is, in many instances, no way even to specify in a logically tight way what the software product is supposed to do or how it is supposed to do it."[5,p.69] On the subject of design criteria, J.W. Smith observed that there was

> a tendency that designers use fuzzy terms, like "elegant" or "powerful" or "flexible." Designers do not describe how the design works, or the way it may be used, or the way it would operate. What is lacking is discipline, which is caused by people falling back on fuzzy concepts. . . . Also designers don't seem to realize what mental processes they go through when they design. Later, they can neither explain, nor justify, nor even rationalize, the processes they used to build a particular system.[5,p.38]

Heterogeneity, fuzziness, lack of discipline, lack of theory—such complaints persist to this day.

Because problem solving is such a basic activity and because complexity is such a fundamental phenomenon, attempts to address these dilemmas tended to produce conceptually broad notions. Peter Naur suggested that "software designers are in a similar position to architects and civil engineers, particularly those concerned with the design of large heterogeneous constructions, such as towns and industrial plants. It therefore seems natural that we should turn to these subjects for ideas about how to attack the design problem."[5,p.35] More concretely, H.R. Gillette of Control Data suggested that the three fundamental design concepts of modularity, specification, and generality were essential to a maintainable system.[5,pp.39-40] IBM's Brian Randell suggested that "there are two distinct approaches to the problem of deciding in what order to make design decisions," top-down and bottom-up.[6] Professor Stanley Gill contended, however, that "in practice neither approach is ever adopted completely; design proceeds from the top and bottom, to meet somewhere in between, though the height of the meeting point varies with circumstances."[7] In other words, one's approach to software design had to be flexible rather than doctrinaire. Effectiveness required combining perspectives.

A year later, a follow-up conference on software engineering techniques took place in Rome under NATO auspices. The editors of the conference report observed, however:

> The resulting conference bore little resemblance to its predecessor. The sense of urgency in the face of common problems was not as apparent as at Garmisch. Instead, a lack of communication between different sections of the participants became a dominant feature. Eventually, the seriousness of this communication gap, and the realization that it was but a reflection of the situation in the real world, caused the gap itself to become a major topic of discussion. Just as the realization of the full magnitude of the software crisis was the main outcome of the meeting at Garmisch, the realization of the significance and extent of the communication gap is the most important outcome of the Rome conference.[8]

This perceived gap was generally regarded as one between theory and practice, i.e., between computer science and software engineering. I.P. Sharp opined that theory and practice translated into architecture and engineering and that design was the key activity. "Within that framework programmers or engineers must create something. No engineer or programmer, no programming tools, are going to help us, or help the software business, to make up for a lousy design."[8,p.12] R.M. Needham of the Cambridge University Mathematical Laboratory and J.D. Aron of IBM argued that "much theoretical work appears to be invalid because it ignores parameters that exist in practice."[9] Reality, they seemed to feel, was a messy and complex business, and that messiness and complexity could not simply be wished away. They had to be dealt with.

The NATO conferences set the stage for many of the debates of the next decade: language generality versus specificity, testing versus verification, practice versus theory. But they also highlighted the problem of complexity and the pivotal activity of design. In short, the NATO meetings revealed and sparked concern not only for the structure of programs but also for the structure of programming.

Coming to Grips: Getting a Handle on Complexity

Central to the software development process, both literally in terms of the life cycle and figuratively in terms of profile, software design drew much of the attention in the years immediately following the NATO software engineering conferences. The problem of complexity was particularly evident in the process of design and so generated much thought as to how to control it. That thought devolved on the community as techniques of modularity, abstraction, management, and measurement. Much of it was soon ensconced in the appealing term *structure*. While some hailed the advent of the structure revolution, however, others rebelled against the notion that these concepts and techniques constituted a breakthrough that would transport software development into a new world untroubled by the difficulties of the past.

Many of the key design concepts of the period sprang from the elemental notion of modularity. In a 1971 article in *Communications of the ACM* (the principal journal of the Association for Computing Machinery, ACM), Niklaus Wirth described stepwise refinement, a process of software development in which a design is gradually decomposed in successively greater detail until fully expressed in the implementation (programming) language. Stepwise refinement constituted a basic, practical approach to the problem of minimizing program complexity. It aimed to "decompose decisions as much as possible, to untangle aspects which are only seemingly interdependent, and to defer those decisions which concern details of representation as long as possible."[10] In more concrete terms, stepwise refinement implied modular design. But while the notion of modularity had long been bandied about, its effective application was another matter. A 1971 letter to *Datamation* (a leading data processing trade journal) complained that many supposedly modular programs were little better than the monolithic ones they replaced. Practitioners needed criteria for modular design.[11]

This was no sooner said than done, as David Parnas explored that very topic in the pages of *Communications* the following year. Parnas argued that segments or modules should convey the minimum amount of information required to enable other parts of the program to use them properly. Parnas's point was that *how* a module accomplished its function was irrelevant to the modules that invoked it. Information beyond the relationship between module input and output served only to complicate matters and tempt the programmer to play with details better left alone. Parnas's technique was quickly labeled "information hiding."[12] The salutary aspect of such a strategy was inherent in the label. If the problem was one of excessive complexity, which in practical terms meant too much information for an individual to manage intellectually, then the obvious solution was somehow to reduce the amount of information that had to be considered at any given time. Parnas followed up on this in another article later that year. He cited the benefits of modular programming as managerial (reduced communication requirements between module developers), flexibility (changes in one module need not necessitate changes in others), and comprehensibility (the system could be studied one module at a time).[13]

The next year, 1973, Glenford Myers tackled the subject of criteria to guide program decomposition. He suggested that the objective was to minimize module coupling (interdependence between modules) and to maximize module strength (intradependence within modules). Correct modularization, he asserted, would lead to increased reliability, decreased development costs, increased extensibility, increased project control, and off-the-shelf parts, with a large measure of these benefits resulting from a reduction in complexity.[14] In a paper two years later in 1975, Frank DeRemer and Hans Kron of the University of California at Santa Cruz expanded the meaning of the distinction between intramodule and intermodule complexity. They argued that "structuring a large collection of modules to form a 'system' is an essentially distinct and different intellectual activity from that of constructing the individual modules. That is, we distinguish programming-in-the-large from programming-in-the-small."[15] The authors' principal point was the necessity of a separate module interconnection language. In the years to follow, however, this distinction would often be invoked to distinguish software engineering from mere programming.

But while the notion of modularity had long been bandied about, its effective application was another matter.

If benefits could be gained from treating modules as functional abstractions, which was the basic goal of information hiding, perhaps there were also advantages to treating data structures in a similar manner. A 1975 article by Barbara Liskov (MIT) and Stephen Zilles (IBM) in *IEEE Transactions on Software Engineering* (started that year by the Computer Society of the Institute of Electrical and Electronics Engineers, IEEE) explored techniques for specifying data abstractions—groups of related operations that act on a class of objects (a data type) and provide the only means of manipulating the objects.[16] In other words, just as information hiding permits modules to be used only in certain well-defined ways, data abstraction allows only certain well-defined operations on data structures. Most early efforts regarding data abstraction focused on achieving it in more traditional procedural languages. John Guttag, in a 1977 *Communications* article, described an algebraic technique for the specification of abstract data types. But while such techniques "should present no problem to those with formal training in computer science," he cautioned, "most people involved in the production of software have no such training. The extent to which the techniques described . . . are generally applicable is thus somewhat open to conjecture."[17]

Object-oriented programming took both data abstraction and information hiding to extremes. Originating with the Simula programming language Kristen Nygaard and Ole-Johan Dahl developed in Norway in the 1960s and typified by the Smalltalk system developed at Xerox during the 1970s, object-oriented programming revolved around objects that embodied a data type and the operations applicable to it. Rather than acting directly on its universe of objects, a program (as well as the objects) dispatched messages that each object interpreted and acted on in accordance with its internal rules. This was data abstraction in the extreme, because in theory the program did not require any knowledge whatsoever of the implementation specifics of the objects; it did not even need to know whether there *were* any objects. Object-oriented enthusiasts, though, contended that the approach was different not simply in degree but also in kind. In contrast to traditional methods, "rather than factoring our system into modules that denote operations, we instead structure our system around the objects that exist in our model of reality."[18] This led to "the claim that the thinking process inherent in OOD [object-oriented design] is more 'natural' than that of SD [structured development], i.e., in building an abstract model of reality it is more natural to think in terms of objects than in terms of functions."[19] On the other hand,

> there is very little that one can say with much confidence about a "most natural" way that people think about the realities of their universe. Thus, to say that the object model is a more natural way to think involves a rather sizeable leap of faith. Undoubtedly, the truth of the matter is that both paradigms are "natural," and that the proper synthesis of the two, in relation to a particular problem, is what should be striven for. . . . The task ahead is to move the debate to a higher level—not arguing about which is more "natural"—but exploring how we best take advantage of both approaches.[19,p.47]

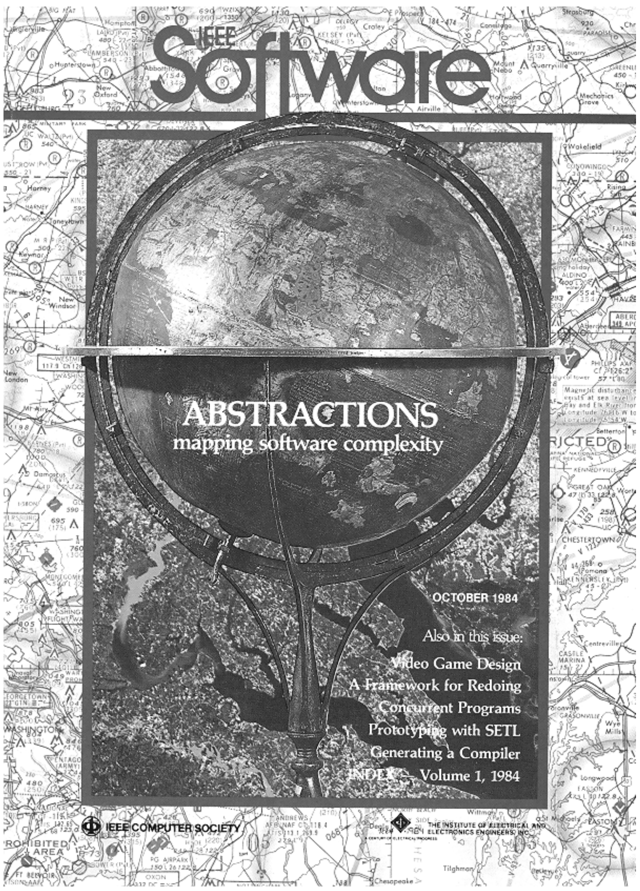
In the mid-1980s, in fact, several proposals were made to combine object-oriented and conventional procedural techniques. In a 1984 article in *IEEE Software* (started that year by the IEEE Computer Society), Brad Cox proposed adding object-oriented concepts "on top" of conventional programming languages. "Hybrid languages just add a new power tool to the programmer's kit, a tool that can be picked up when it fits the task at hand or set aside when conventional techniques are sufficient."[20] In the same vein, the authors of an article in *Computer* (the principal journal of the IEEE Computer Society) the following year suggested that "just as a combination of top-down and bottom-up development is appropriate to many applications, a combination of functional [Fortran-like] and object-oriented design might well be most appropriate."[21] Likewise, a 1989 article described how to integrate the object-oriented approach with structured development.[22] Such proposals reinforced the notion that synthesis might prove more beneficial than revolution. Rather than treat distinct approaches or concepts as universal dogma, a more pragmatic approach might entail employing a combination of techniques as circumstances warranted.

The object-oriented approach also put in high relief the issue of domain-specific knowledge. As consultant Patrick Loy noted, the principal problem for this approach was finding the objects, i.e., identifying the relevant objects in the problem domain that must then be defined along with their properties within the software.[19,p.45] This often required fairly deep knowledge of the application domain. After all, *even if* object-oriented programming was exceptionally effective at modeling the "real world," the real world is a complex place, and what should be construed as an object for programming purposes is often not obvious. Writing in *Communications* in 1987, Russell Abbott emphasized the crucial role of domain knowledge in software development.[23] The following year, a report on a study of 17 large software development projects noted that

> the deep application-specific knowledge required to successfully build most large, complex systems was thinly spread through many software development staffs. Although individual staff members understood different components of the application, the deep integration of various knowledge domains required to integrate the design of a large, complex system was a scarcer attribute.[24]

The importance of domain-specific knowledge was also recognized at the 1989 International Conference on Software Engineering. Victor Basili of the University of Maryland argued for application-specific research in academia, while Bill Curtis of the Microelectronics and Computer Technology Corporation (known as MCC) made a case for developing domain specializations in software engineering.[25] Application domain has repeatedly been seen as one of the principal contexts of which software technologists must be cognizant. It is one of the key areas in which the tension betweeen specificity and generality plays itself out.

In any case, the important point was that in the mid-1970s, many concepts that applied to procedures or functions could also apply to data. Around the mid-1970s, Michael Jackson, one of the European structured programming disciples, developed an approach that centered around the data rather than the operations. Jackson's method produced a program whose structure corresponded to the data structure of the problem.[26] This was also the premise behind a method Jean Warnier devised around the same time.[27] Ken Orr developed a variation of Warnier's technique a few years later that became known as the Warnier–Orr approach. A 1978 article in *Software Engineering Notes* (the publication of SIGSOFT—the ACM Special Interest Group on Software Engineering) concluded that "the data-structured/process oriented approach is the one that has the best prospects for system and program design in the future."



The term *structured* had quickly assumed the status of an icon, representing salvation in the eyes of some and just one more dubious quick fix in the eyes of others. The arrival of the structured programming "revolution" was heralded in a collection of *Datamation* articles at the end of 1973. James Donaldson of Control Data indicated that the name of the game was complexity management. "A technique known as structured programming has been developed which offers improvements in both program complexity and program clarity."[29] The following year, a seminal article by Stevens, Myers, and Constantine in the *IBM Systems Journal* brought together many of the basic tenets and attached a slightly different but revealing label to them—structured *design.*[30] Many articles and books on structured programming, structured design, and structured analysis followed, but they were essentially variations on a theme. That theme consisted of concepts that Simon has suggested are fundamental, universal principles of design—hierarchical decomposition and modularity. It is hardly surprising that so many seized on "structured" as the adjective of choice. Design, from Simon's viewpoint, consists of exercises in divining the structure of a problem and systematically structuring an appropriate solution.[31]

Another aspect of the structure "revolution" addressed the task of carrying out the design process in an efficient and controllable fashion. In other words, this aspect concerned management of the process. One management strategy in particular became closely associated with structured programming. As IBM's F. Terry Baker and Harlan Mills discussed in the 1973 *Datamation* collection, the chief programmer team approach, while "made possible by recent technical advances in programming, . . . also incorporates a fundamental change in managerial framework which includes restructuring the work of programming into specialized jobs, defining relationships among specialists, developing new tools to permit these specialists to interface effectively with a developing, visible project...."[32,p.61] By placing a single master programmer in charge of design, providing appropriate support in terms of tools and personnel, and employing structured programming techniques, the chief programmer team approach could supposedly result not only in an "entirely new technical standard for design quality" but also in a "true professional discipline with a recognized, standard methodology."[32] (Such arguments illustrate the function that "techniques" play in the process of professionalization.) This focus on group structure and dynamics was not altogether new; Gerald Weinberg had taken the same perspective in *The Psychology of Computer Programming* in 1971.[33] But whereas Weinberg had emphasized decision by consensus, Baker and Mills saw advantages in a more authoritarian style. As evidence, the authors pointed to the development of an information bank for the *New York Times,* a project characterized by high productivity and very low error rates. Questions were raised, however, concerning the extent to which the circumstances surrounding the project were in fact typical. Moreover, it seems the system eventually proved unsatisfactory and was replaced some years later by a less ambitious system.[34] (It should be noted, though, that in recounting the project, Mills presented it as an unqualified success, making its ultimate outcome unclear.)

Given the fanfare with which structured programming (or whatever other activity one cared to attach) was introduced, a substantial amount of skepticism was virtually guaranteed to greet it. Fred Gruenberger's reaction was typical:

> So now it's structured programming and chief programmer teams that will clear up all the troubles and make master programmers of all us clods. Pardon me while I yawn; I've been here so many times. . . . Every single advance in software . . . has been introduced with exactly the same claims. Each such advance (and the totality of structured programming may well be one) adds to our bag of tricks. And none of them contributes very much to the real underlying problem, which is clear thinking in the area of problem solving.[35]

Likewise, Dick Butterworth of General Electric cautioned that "SP [Structured Programming] is no panacea—it really consists of a formal notation for orderly thinking—an attribute not commonly inherent in programmers nor any other type."[36] John Fletcher of Lawrence Livermore Laboratory was more scathing. He acidly suggested that the labeling as revolutionary of the ideas underlying structured programming was "clear commentary on the sad state into which the practice of programming has fallen in many quarters and in which it apparently will remain."[37] Fletcher apparently felt the concepts falling under the structured programming rubric consisted of long-standing fundamentals rather than revelatory innovations.

The term *structured* [programming] had quickly assumed the status of an icon, representing salvation in the eyes of some and just one more dubious quick fix in the eyes of others.

Experiences with structured programming, if not earth-shattering, were nevertheless reasonably positive. A session on experiences and accomplishments with SP at the 1974 Lake Arrowhead Workshop on Structured Programming produced the conclusion that programs were generally more reliable, understandable, and maintainable.[38] James Elshoff of General Motors compared sets of actual production programs to ascertain the effect of structured techniques and found the SP programs much more comprehensible.[39] Nevertheless, a 1976 book review in *Computer* observed that the "ideas underlying the subject [structured programming] have been intensively debated for almost a decade. . . . Yet there has been little sign of any real consensus emerging from this debate. On the contrary, it often seems that discussions of the merits of structured programming are becoming more acrimonious as time goes by."[40]

Much of the caustic commentary over structured programming did not constitute rejection of its basic tenets. As several individuals noted, no one was in favor of unstructured programs. Rather, the argument concerned relative value. Many practitioners objected to perceived attempts to deify a set of useful but less than omnipotent techniques. Many sought not to discredit structured programming but simply to bring it and its overly zealous advocates back down to earth. Paul Abrahams of the Courant Institute indicted the sociology of structured programming rather than its content. "There are two baleful aspects of this sociology: the elevation of good heuristics into bad dogma, and the creation of the illusion that difficult problems are easy."[41] In a similar vein, Daniel Berry of the University of California at Los Angeles (UCLA) declared that it "seems preposterous to me (and to others) that the programs described in the published descriptions of structured programming were developed as cleanly as described in the papers...."[42] A decade later, Parnas and Paul Clements made a similar charge regarding the rationality of design processes generally:

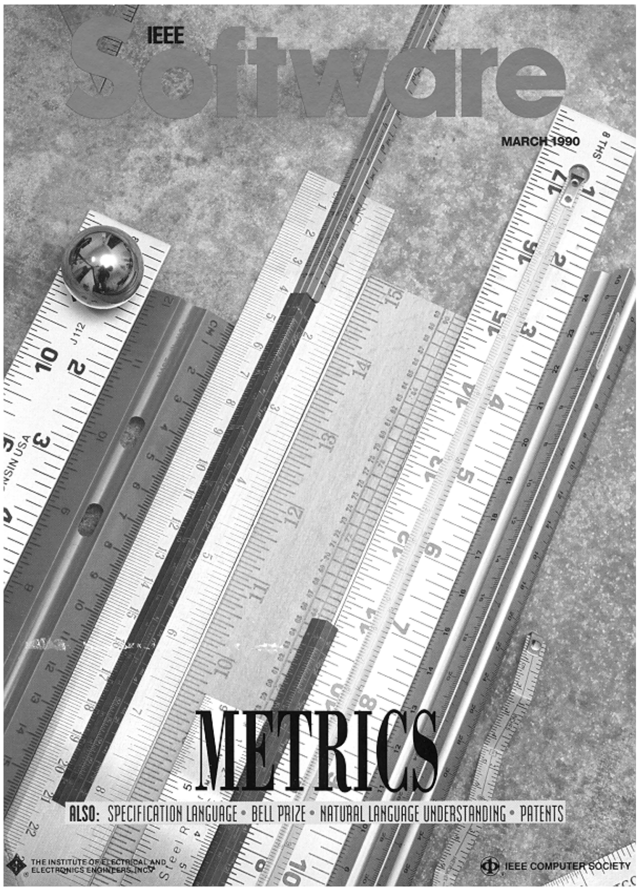
> The picture of the software designer deriving his design in a rational, error-free way from a statement of requirements is quite unrealistic. No system has ever been developed in that way, and probably none ever will. Even the small program developments shown in textbooks and papers are unreal. They have been revised and polished until the author has shown us what he wishes he had done, not what actually did happen.[43]

Numerous practitioners took such arguments a step further, contending that the benefits of design techniques and software engineering in general were principally in the state of mind they produced. In his software keynote address at COMPCON (the IEEE Computer Conference) in 1975, William McKeeman of the University of California at Santa Cruz "described structured programming as a problem-solving process—'a human activity that needs to be structured....'"[44] Peter Denning contended that the "whole point of 'structured programming' is to set up mental patterns according to which we write programs from the beginning using the prescribed forms. The whole point is to establish ordered and disciplined thinking leading to clearly structured programs."[45] C. Wrandle Barth of the Goddard Space Flight Center observed that "catastrophes can be constructed from the top down. A chief programmer team can still design a horse as a camel. The real lessons of software engineering are much more in the realm of attitude, approach, and emphasis than on techniques and rules."[46] Honeywell's David Frost suggested more explicitly a psychological rationale for structured programming, relating the concept of chunking to programming. (Chunking refers to the process in which humans store information in their memories by structuring or coding it.) "What all this boils down to is that psychology provides a powerful argument for modularity in systems design. But it is also a powerful argument for the hierarchical design process called top-down decomposition, as well as for hierarchical program structures, because chunking results in essentially hierarchical structures in the mind."[47] A 1976 *Datamation* article by Lawrence Peters and Leonard Tripp of Boeing placed such views in still larger perspective. They characterized software design as a "wicked problem," i.e., one that changes during resolution and for which it is not always clear how to proceed. Specific techniques could ease but not remove the essential difficulty of the design process.[48] Peters and Tripp made the point even more explicitly in the pages of *Datamation* the following year. "Software design methods merely assist in solving routine aspects of a problem. Using a methodology only reveals the critical issues in a design effort and gives us more time to address them. . . . [D]esigning is problem solving—a fundamental, personal issue."[49] Indeed, Dennis Geller in a 1979 letter to *Software Engineering Notes* suggested that modularity and top-down be viewed "as underlying principles which reflect our understanding of our own psychological and organizational limitations, rather than as 'methodologies....'"[50] In other words, such concepts constituted fundamental problem-solving strategies precisely because they addressed basic human limitations in dealing with complexity.

Dealing with limitations in a realistic manner was certainly the thrust of the landmark 1975 book *The Mythical Man-Month,* in which Brooks analyzed his experience as manager of the OS/360 project that developed the operating system for IBM's famous System/360 computers. Writing in an engaging and accessible style, Brooks addressed issues involving such things as the dynamics of programming teams, scaling up, design principles, and estimation. The results were revealing insights into, for example, the overheads inherent in large organizations, the difficulty of producing coherent designs, and the virtual impossibility of getting a software product right the first time. His epilogue concisely sums up his analysis:

> The tar pit of software engineering will continue to be sticky for a long time to come. One can expect the human race to continue attempting systems just within or just beyond our reach; and software systems are perhaps the most intricate and complex of man's handiworks. The management of this complex craft will demand our best use of new languages and systems, our best adaptation of proven engineering management methods, *liberal doses of common sense, and a God-given humility to recognize our fallibility and limitations* [emphasis added].[51]

Observations of this sort, however, seemed unlikely harbingers of a new age.


IEEE
Software
MARCH 1990
METRICS
ALSO: SPECIFICATION LANGUAGE • BELL PRIZE • NATURAL LANGUAGE UNDERSTANDING • PATENTS
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.
IEEE COMPUTER SOCIETY

A new age, though, was exactly what many practitioners sought and believed would result from formalized mathematical attacks on the programming problem. One of the principal carriers of this torch was Mills. (It should be noted that while formal mathematics in programming was most prominently associated with particular advocates—including Mills, Edsger Dijkstra, and C.A.R. Hoare—who were often mentioned in the same breath, they were certainly not all of one mind. Dijkstra, for example, disassociated himself from what he considered the "empty but impressive slogans" of Mills regarding structured programming.[52]) In a 1975 *Communications* article, Mills presented a mathematical model of structured programming to "simplify and describe programming objects and processes. It is applied mathematics in the classic tradition, providing greater human capability through abstraction, analysis, and interpretation in application to computer programming." Such efforts would supposedly transform programming from "an instinctive, intuitive process to a more systematic, constructive process that can be taught and shared by intelligent people in a professional activity."[53] Writing the following year in *IEEE Transactions on Software Engineering,* he lamented the "legacy of heuristic thinking in software development" while lauding the "powerful tools in mathematics for expressing and validating logical design on a rigorous basis."[54] He was particularly taken with Dijkstra's constructive approach to program correctness (in which a program and its proof are developed concurrently) as articulated in *A Discipline of Programming,* which came out that same year.[55] A rigorous, formal approach of this type stood in contrast to a mere "attitude."

While debate swirled around structured programming in general, the goto statement continued to serve as lightning rod. Goto statements unconditionally transfer program execution to some other instruction out of sequence. In the late 1960s, Dijkstra had called attention to the deleterious and unnecessary complexity their use engendered; avoidance of goto statements quickly became one of the most prominent mantras of the structured programming movement. The goto drew so much attention, in fact, that sometimes it seemed as if practitioners were incapable of seeing the forest for the trees. The flap over the goto was in full display at the 1972 ACM National Conference, with several notables taking sides.[56] In a 1974 piece, Donald Knuth argued that it was indeed possible to write well-structured programs with goto statements. He advocated limited, disciplined use, however.[57] In a 1976 article in *SIGPLAN Notices* (the publication of the ACM's Special Interest Group on Programming Languages), Richard DeMillo, S. Eisenstat, and Richard Lipton set out to determine formally whether structured control mechanisms could efficiently simulate programs using the goto construct. They developed formulas indicating that a significant loss of efficiency occurred, which manifested itself either in increased program size or in slower execution.[58] Ronald Jeffries responded that his firm's rewritten code did not suffer in such a fashion and asserted that "we need approaches to design which, in the hands of ordinary mortals, yield programs that work. The techniques of 'structured programming' seem to help us meet those goals."[59] Jeffries obviously found theoretical debates over the goto of less concern than finding design techniques of practical value. SofTech's William Rosenfeld also suggested that the authors "seem to have missed the point of the structured programming debate. It is not the objective of structured programming to improve the efficiency of control structures but rather to improve program readability. . . . Too much time is spent making programs efficient and not enough time is spent making them useful and correct."[60]

Indeed, a 1975 *Communications* article by Henry Ledgard and Michael Marcotty suggested that the whole debate over control structures was getting out of hand. Nevertheless,

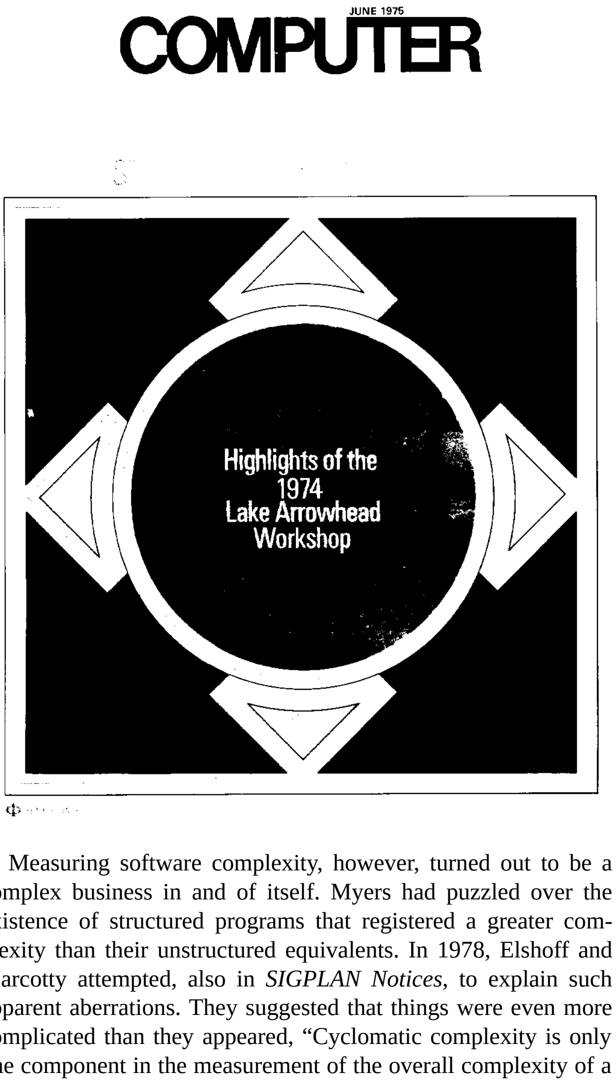
> while it may be argued that the control structure issue has been entirely overworked, the debates and polarized opinions remain. On the one side we have the well-known views of Dijkstra and Mills, who have advocated the strict use of the if-then-else and while-do control structures and their variants. On the other side, we have the views of Knuth, who has recently presented interesting arguments on the utility of the goto.[61]

The authors remained convinced that the basic structures Dijkstra advocated—sequence, selection, and repetition—were sufficient for the practicing programmer.[61,p.638] A 1978 Workshop on the Science of Design also concluded that efficiency was no longer king. "No matter how elegant proving and testing techniques are, they cannot replace design correctness. . . . In this regard, design constraints that result in better testability and better verification even though the hardware may be used less efficiently should be encouraged."[62] In other words, *good* software implied more than *efficient* software.

"We need approaches to design which, in the hands of ordinary mortals, yield programs that work."

Over the course of the 1970s, attributes other than efficiency began to dominate concern over software characteristics. Efficiency remained a legitimate concern, but it could no longer be the *only* concern. The vast increases in complexity necessitated a more complex value structure. A fast but incomprehensible program was no bargain; errors and maintenance difficulties rendered speedy execution far less advantageous. Tools such as structured techniques were means toward satisfying the demands of the new value structure, but difficulties arose in evaluating the results of their application. Like so much else in the developing software field, software metrics quickly settled into the motherhood and apple pie category. Everyone agreed on the importance of properties such as clarity, reliability, and maintainability for software quality but nobody was sure how to measure them. While efficiency lent itself to relatively straightforward measurement in terms of execution times, more nebulous criteria proved less obliging. Traditional measurement methods that concentrated on statistical analyses of defects and breakdowns were clearly inadequate for a medium in which many problems originated in specification and design rather than physical deterioration. The concept of a physical breakdown is a non sequitur in the realm of software. Of concern, rather, is how to determine, for instance, which design is less complex than another and thus likely to be less flawed and more maintainable. By 1978, consultant Tom Gilb could still complain in the pages of *Software Engineering Notes* that "quality goals are like the weather; everybody talks about them, but nobody quantifies them."[63]

In this sphere also, many found the allure of the "hard" sciences irresistible. Kolence argued in *Datamation* in 1971 that "performance measurement is inextricably linked to the study of the natural laws governing the behavior of software *in situ,*" an area he dubbed software physics.[64] Around the same time, Maurice Halstead of Purdue University began experimenting with formulas relating structural properties of programs (e.g., numbers of operators and operands) to coding time and expected error counts. Halstead laid out his findings and arguments in 1977 in *Elements of Software Science.*[65] The year before, Thomas McCabe of the National Security Agency, whose work is often mentioned in the same breath as Halstead's, described in *IEEE Transactions on Software Engineering* a measure of program complexity he termed the cyclomatic number.[66] The cyclomatic number was a function of the number of potential logical execution paths through a structured program. But even here the sheer complexity of the object forced a compromise. The impracticability of attempting to calculate the total number of paths led to a definition based on "basic paths," which when combined would generate every possible path. A 1977 article in *SIGPLAN Notices* by Glenford Myers of IBM provided evidence of the utility of the cyclomatic metric. In Myers's opinion, "McCabe's proposal seems to be . . . one of the most intuitively satisfying, simplistic, and easy-to-apply complexity measures."[67] Myers noted that the cyclomatic metric confirmed the subjective judgments of B. Kernighan and P. Plauger in *The Elements of Programming Style* (1974) as to the relative complexity of various control structures.



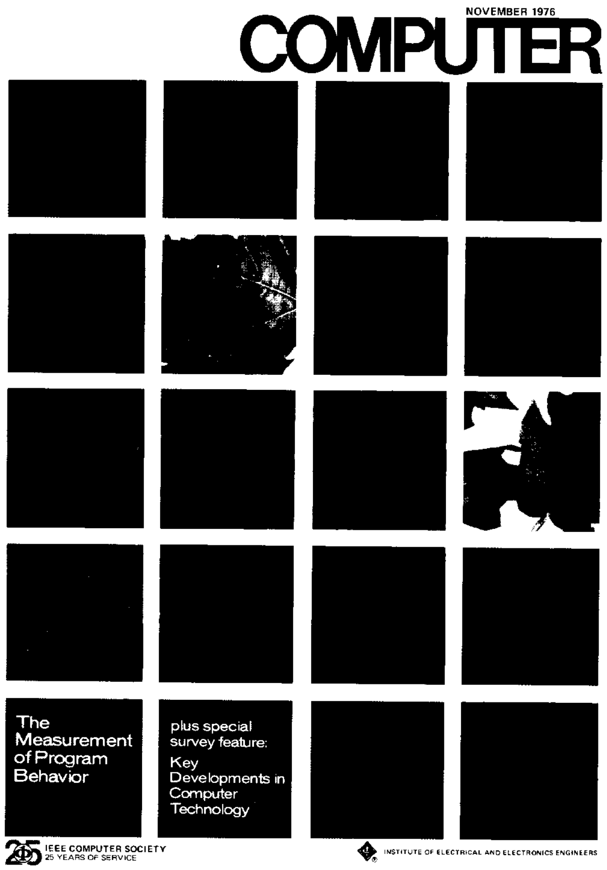
Measuring software complexity, however, turned out to be a complex business in and of itself. Myers had puzzled over the existence of structured programs that registered a greater complexity than their unstructured equivalents. In 1978, Elshoff and Marcotty attempted, also in *SIGPLAN Notices,* to explain such apparent aberrations. They suggested that things were even more complicated than they appeared, "Cyclomatic complexity is only one component in the measurement of the overall complexity of a program. A reduction in one measure of complexity will often result in an increase in another aspect of complexity."[68] In other words, the very phenomenon of software complexity was complex, manifesting itself in a variety of interrelated ways. The real world would not even accommodate a straightforward notion of complexity. Therefore, Elshoff and Marcotty concluded, "the use of [an] empirically determined bound for complexity as a programming guideline . . . seems to be reasonable. On the other hand, the use of the cyclomatic complexity for the direct comparison of programs . . . is fraught with danger."[68,p.39] That one of the best software metrics available was not useful as a basis for program comparison reflected the individualistic nature of programs, which was in turn a reflection of the malleability of the software medium. A 1977 *Datamation* essay voiced a similar theme, questioning "the wisdom of attempting to discover universal measures for problems which are, perhaps inherently and certainly practically, local in character."[69]

The malleability of the medium was even more explicitly recognized in a 1978 piece in *Transactions*. Edward Chen of Travelers Insurance argued that complexity metrics generally ignored the fact that "there exist, in general, multiple solutions, and the programming process can be envisaged as a combination of both analysis and synthesis processes aimed at identifying the most desirable solution among a large number of feasible alternatives."[70] In other words, the problem was not simply the complexity of the resulting artifact, but the inherent complexity involved in the design of the artifact. One derived benefits from a design of relatively low complexity, but arriving at that design was a complex matter itself. Several General Electric scientists put forward a similar view the following year. Differentiating between the computational complexity of the algorithm and the psychological complexity of the programming process, they concluded that "assessing the psychological complexity of software appears to require more than a simple count of operators, operands, and basic control paths. If the ability of complexity metrics to predict programmer performance is to be improved, then metrics must also incorporate measures of phenomena related by psychological principles to the memory, information processing, and problem solving capacities of programmers."[71] The apparent necessity of such measures suggests the importance of fundamental cognitive processes and strategies in dealing with software. Nevertheless, N.F. Schneidewind and H.-M. Hoffmann concluded that same year that "for similar programming environments and assuming a stable programming personnel situation, structure would have a significant effect on the number of errors made and labor time required to find and correct the errors. . . . It would be worthwhile to use complexity measures as a program design control to discourage complex programs and as a guide for allocating testing resources."[72] They also suggested that while no single measure of program complexity had proven "best" in their experiment, the cyclomatic metric appeared most practical due to its relative ease of computation.

The notion that no single metric qualified as "best" received reinforcement in the 1980s. A 1982 *Computer* article that discussed the relationship of complexity metrics to software maintenance observed that while measures based on program size worked well in differentiating programs of widely varying sizes with respect to maintenance costs, measures dealing with data structure, data flow, and flow of control were needed to rank programs of similar size. "The hybrid approach to measuring software complexity is clearly the most sensible approach," concluded the authors. "Software complexity is caused by so many different factors that measuring only one of them cannot help but give unreliable results for a general case."[73] American Bell's William Evangelist voiced a similar view in *SIGPLAN Notices* the following year, suggesting the need to "represent software complexity as a combination of quantities derived from both static and dynamic program properties." He also noted, however, the sticky problem of "precise identification of those quantities of importance and the relative weight each should have."[74] Indeed, an analysis of several metrics, including those of Halstead and McCabe, by a team at the University of Maryland using an experimental database produced the conclusion that "none of the metrics examined seem to manifest a satisfactory explanation of effort spent developing software or the errors incurred during that process."[75] A 1988 critique of cyclomatic complexity in the *Software Engineering Journal* (a joint publication of the British Computer Society [BCS] and the Institution of Electrical Engineers) went even further, suggesting that "it is arguable that the search for a general complexity metric based upon program properties is a futile task. Given the vast range of programmers, programming environments, programming languages and programming tasks, to unify them into the scope of a single complexity metric is an awesome task."[76] However, while many technologists seemed to agree with such sentiments in principle, J. Paul Myers, Jr., of Trinity University complained in 1992 that "new metrics are introduced nonetheless as 'all-purpose' measures of software complexity."[77]

An *IEEE Software* article later that year attempted to finesse the problem by using factor analysis to aggregate individual complexity metrics into one overall complexity value. Many of the more than 100 existing metrics, the authors contended, "measure many of the same things. Our research leads us to believe that existing metrics probably measure no more than four or five distinct types of complexity. Assuming this is true, the best metric would represent as much variance in these underlying complexity domains as possible."[78] They therefore proposed a metric called "relative complexity," the product of mapping individual complexity metrics into independent complexity domains—control, size, modularity, information content, and data structure—the resulting weighted values (relative significance for that program) of which were then converted into a single complexity score for each program module. In a sense, though, such a scheme begged the question, since in order to make sense out of any given relative complexity, the score would have to be unpacked to give the scores in the different complexity domains. Moreover, it did not address the issue of whether particular metrics might prove more or less suitable for given settings. While the metrics in one complexity domain might all be measuring the same type of complexity, some could prove more meaningful than others depending on circumstances such as application type and the particulars of the development environment. These more meaningful metrics would then be diluted by the presence of less appropriate ones. Indeed, writing in *IEEE Software* in 1988, Basili had criticized the tendency of organizations to employ metrics that "are bottom-up and based blindly on models and metrics in the literature, rather than top-down and based on an understanding of their own processes, products, and environment."[79] The importance of such context sensitivity was affirmed by the former director of Contel's software metrics program five years later. "Different projects have different products, environments, domains, goals, and customers, so developers have different needs. The metrics collected should reflect the project's process maturity and needs. It is not only natural but desirable for different projects to collect different metrics."[80]

Clearly, software complexity was itself complex, with a multitude of facets that defied management or measurement by any single method. Simple, singular approaches were unlikely to do the trick in a complicated and messy reality. Complexity was a slippery concept, and while various "structured" techniques helped control the complexity of both design activity and the design itself, they hardly constituted a panacea. Moreover, determining the complexity of a particular design in some absolute sense as well as relative to other designs was a tricky business. A number of practitioners even acknowledged the essential fuzziness of design activity, suggesting that the principal benefits of structured programming derived from general mental patterns rather than specific techniques. Structured techniques constituted one important pragmatic response to the problems of software technology. Their promotion by some practitioners as dogma rather than as practical tools served only to stiffen resistance.


NOVEMBER 1976
COMPUTER
The Measurement of Program Behavior
plus special survey feature: Key Developments in Computer Technology
25 IEEE COMPUTER SOCIETY 25 YEARS OF SERVICE
INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS

Correctness Versus Confidence: Program Verification

The issues of complexity, pragmatic accommodation, and self-image were nowhere so apparent as in the area of program verification. But while people could disagree over software metrics with little rancor, the question of how to determine a program's "correctness"—whether the program met its specifications—aroused passions. The controversy pitted the advocates of program testing against the promoters of formal verification. That the former was often viewed as the "engineering" approach while the latter was seen as more "scientific" or mathematical is suggestive of both the nature of the techniques and the self-images of practitioners. The nature of the solution sometimes seemed to be a function of the perceptions of the practitioner as to what he or she was (scientist or engineer) and just what that entailed. What became apparent to at least a handful of practitioners, however, was the existence of a middle ground. Complexity took its toll in every venue and defied singular or absolutist approaches. Just as it had in the case of software design and metrics, pragmatism in the case of verification translated into accommodation and synthesis.

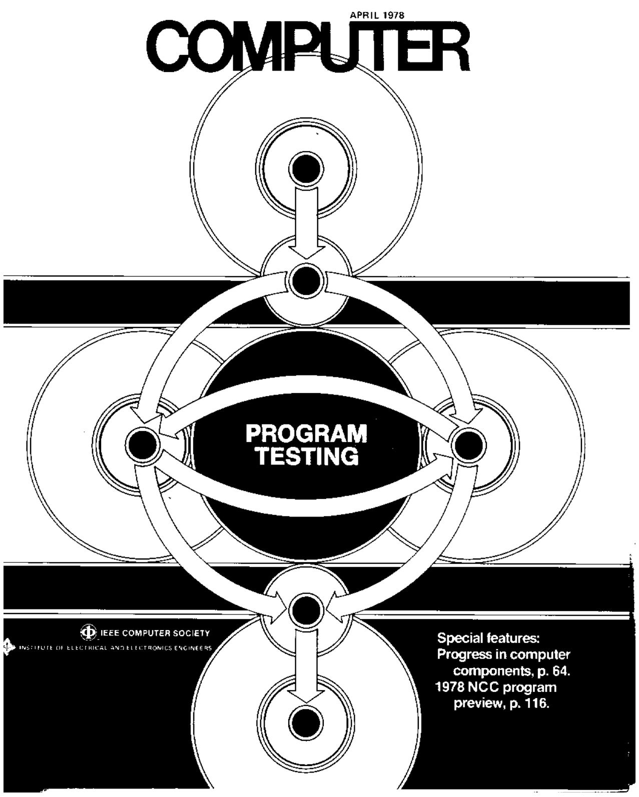
There was little disagreement in the 1970s that software quality was too often a contradiction in terms. People agreed less on precisely what to do about it. Quality assurance techniques developed for hardware were of dubious applicability. As a 1971 *Computer* article noted, it "would indeed be fortunate if the well-developed theory of hardware reliability could be used to predict or enhance the reliability of software. Unfortunately, this is not to be the case as hardware reliability theory is based mainly upon the statistical analysis of random failures of components with age."[81] Nevertheless, something had to be done. In a 1974 *Datamation* piece, Tom Steel of Equitable Life Insurance declared:

> *[T]he* major critical problem in the [computer] industry is, in my view, the quality of software, whether vendor or user produced. . . . It is usually inadequate functionally, inconsistent between actuality and documentation, error-ridden and inexcusably inefficient. Beyond all that, it costs far too much. I can think of no other products (aside, perhaps, from pornography and telephone service in New York) that have all these failings to anything like the degree found in software.[82]

The search for a "back-end" answer followed two distinct paths—testing and formal verification. Testing sought to develop reasonable confidence that a program or system would behave as was intended by "exercising" the program. Formal verification (also referred to as program proving and proofs of correctness) sought to prove mathematically that a program matched its specifications. These two approaches tended to attract and foster often antagonistic mind-sets.

The idea of testing a program was hardly new, but the relatively new emphasis on the software development process prompted increased emphasis on more systematic testing throughout the development cycle. A 1971 *Datamation* article advised readers to "'think testing' right from the start—modules, programs, systems—all designed to be tested along the way."[83] Mirroring the increasing fashionability of structured programming as the decade progressed, the late 1970s saw numerous calls for structured testing. Complaining in 1977 that testing continued to be a "witch-hunt," Dorothy Walsh advocated a structured approach to testing that "formalizes the intuitive good practices that are its foundation and provides procedures for using them that may be carried out even by inexperienced programmers."[84] A *Datamation* piece the following year argued for top-down testing in addition to top-down coding.[85]

Making testing a more integrated part of the development process was all well and good, but testing proved just as vulnerable to the pernicious effects of mounting complexity as other aspects of software development. One can test software both statically and dynamically, and both are problematic. Static tests focus on program structure, while dynamic tests focus on program execution. Put another way, static tests checked the program's logic, while dynamic tests checked the program's function. Ideally, this meant checking every possible logical path, in the case of the former, and testing with every possible set of inputs (with respect to the program specification), in the case of the latter. Complexity, however, could easily defeat both strategies; combinatorial explosion rendered both complete path testing and exhaustive dynamic testing totally impractical in most instances.


APRIL 1978
COMPUTER
PROGRAM TESTING
IEEE COMPUTER SOCIETY
INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS
Special features:
Progress in computer components, p. 64.
1978 NCC program preview, p. 116.

The problem of test data selection attracted much attention. As was noted in *Datamation* in 1977:

> the key to constructing a minimal yet logically complete set of test data is the accurate and explicit enumeration of all cases or conditions handled by the program or system. . . . The quality of the systems test often breaks down precisely at this starting point. The complete definition of test cases is viewed as an impossible task, so no attempt at an orderly enumeration of conditions to be tested is made at all.[86]

In other words, the complexity of typical software precluded the economical derivation of test data that would completely exercise all aspects of the program. Raising confidence in software testing would require more than brute force. While some advocated the use of randomly generated test cases, most sought a more rationalized solution.

In a 1975 article in *Transactions*, John Goodenough and Susan Gerhart of SofTech sought to finesse the problem. They suggested that the input domain of a program could be partitioned into classes of inputs such that the testing of one element of a class was equivalent to testing the entire class: "This pinpoints the fundamental problem of testing—the inference from the success of one set of test data that others will also succeed, and that the success of one test data set is equivalent to successfully completing an exhaustive test of a program's input domain."[87]

This was easier said than done, principally because, despite the attempt at systematization, particular cases of program conditions were nevertheless considered in an ad hoc fashion. Variations on this approach were offered in later years. But those who awaited the arrival of a comprehensive theory of testing amenable to automation found little encouragement in a 1980 editorial in *Transactions* that observed there was "increasing recognition that it is unlikely there will be a grand theory of testing which will lead to fully automatic testing systems. Instead the tester will be called upon to use his intuition and problem-dependent knowledge in a disciplined manner to test for a variety of specified error types."[88] Attempting to apply computational leverage to testing encountered the same difficulties as attempts to leverage other problem domains; variability and complexity placed limits on effective formalization and automation. DeMillo, Lipton, and Frederick Sayward had made a similar observation two years earlier: "Until more general strategies for systematic testing emerge, programmers are probably better off using the tools and insights they have in great abundance. Instead of guessing at deeply rooted sources of error, they should use their specialized knowledge about the most likely sources of error...."[89] Here was another acknowledgment of the importance of local, problem-specific knowledge.

A similar spirit of pragmatism was evident in a 1980 piece in *Transactions* that attempted to make the Goodenough–Gerhart theory "more than an unattainable ideal," by using it to detect certain classes of error thought likely to occur.[90] Likewise, writing in *IEEE Software* in 1985, Nathan Petschenik of Bell Communications Research argued for the setting of "practical priorities" in the selection of case studies by looking for key problems that would cause massive disruption rather than attempting to track down all or nearly all problems in the software.[91] Practical accomplishment demanded pragmatic concessions.

Even more pragmatic were practitioners who, instead of pinning their hopes on the arrival of a grand theory of testing, began to explore a combination of various strategies. In a 1984 *Transactions* article, Simeon Ntafos of the University of Texas described an approach that combined structural (based on control flow), black-box (based on the program's input specifications), and error-driven (based on known errors) approaches to generate test cases.[92] The following year, Sandra Rapps and Elaine Weyuker at the Courant Institute proposed employing both data flow and control flow as a basis for determining path coverage. Mitre's Samuel Redwine, Jr., had explicitly suggested in 1983 an "engineering approach" to generating test data that revolved around the idea of "different domains and types or metrics of coverage."[93] The use of a combination of testing strategies constituted a pragmatic response to the deficiencies of individual approaches. Such remedies held little appeal, however, for those who saw legitimacy and efficacy as the products of formalism rather than heuristics.

Stemming from the fundamental work of Floyd in the 1960s,[94] program verification with its formal mathematical basis appeared a haven from the dirty, ad hoc world of testing. In a 1971 article in *Communications*, Hoare presented a proof of the correctness of a simple program. He urged the incorporation of such proofs into the coding process, suggesting that carrying out proofs in this fashion was "hardly more laborious than the traditional practice of program testing."[95] Writing in the *Computer Journal* (the research journal of the BCS) that same year, he and a colleague attempted to demonstrate the practicality of employing previously proven programs (in this case, a subroutine) in the proof of a new program. Just as important in this case was the claim that the program to be proved was "realistic" and "nontrivial."[96] Hoare addressed the scaling-up issue even more explicitly the following year, admitting that the application of proof techniques "even to small programs is already quite laborious, so their direct application to large programs is out of the question."[97] There were challenges, however, not only with respect to the question of scaling-up but also with regard to the epistemological foundations of proof methods.

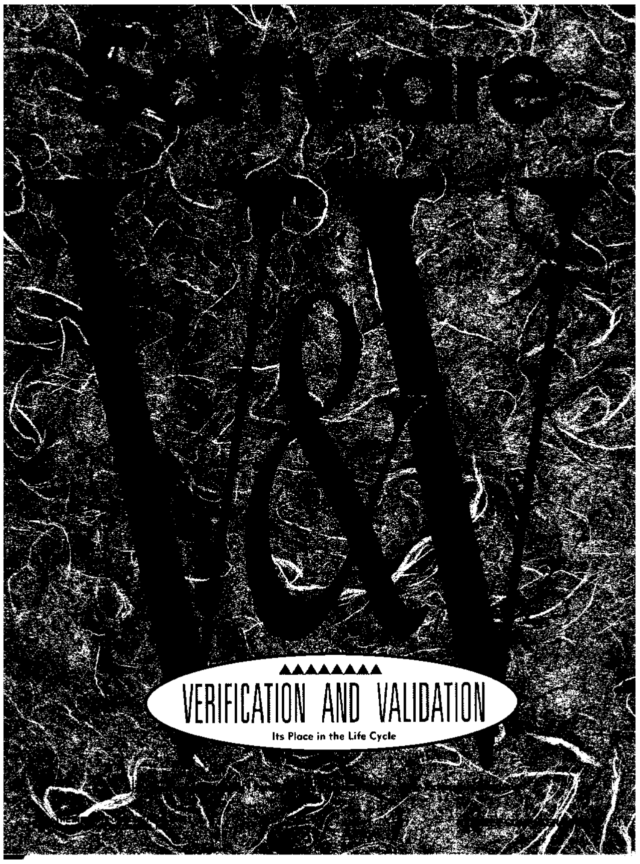
Stemming from the fundamental work of Floyd in the 1960s, program verification with its formal mathematical basis appeared a haven from the dirty, ad hoc world of testing.

The battle was formally joined on a widespread basis in 1979, when DeMillo (Georgia Institute of Technology), Lipton (Yale), and Alan Perlis (Yale) argued in their article "Social Processes and Proofs of Theorems and Programs" that "in the end, it is a social process that determines whether mathematicians feel confident about a theorem—and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail."[98] In mathematics, they contended, the proof of a theorem constitutes a message that is disseminated, scrutinized, and commented on: "Being unreadable and—literally—unspeakable, verifications cannot be internalized, transformed, generalized, used, connected to other disciplines, and eventually incorporated into a community consciousness."[98,p.275] Toy proofs such as Hoare's 1971 verification of the FIND algorithm left them cold: "There is no continuity between the world of FIND . . . and the world of production software, billing systems that write real bills, scheduling systems that schedule real events, ticketing systems that issue real tickets."[98,p.277] So many of software's problems were so intimately connected with scale, the authors were arguing, that a toy (i.e., very small-scale) proof of concept amounted to no proof at all. The practicality of formal verification had yet to be demonstrated through application to large-scale programs. Even then, their principal argument would remain undented; they would still lack confidence in the result.

Verification advocates were not slow to pick up the gauntlet. Leslie Lamport of SRI International declared, "I am one of those 'classicists' who believe that a theorem either can or cannot be derived from a set of axioms. I don't believe that the correctness of a theorem is to be decided by a general election."[99] W.D. Maurer of George Washington University argued essentially that it was not software engineers who should adopt the social processes of mathematics but rather the mathematicians who should make use of the computer to produce complete formal proofs.[100] Implicit in such an assertion was the view that all good things flowed from rigorous formalism. (The formal methods movement will be discussed in the next section.) As for reliance on program testing, he asserted that was "the way other sciences and engineering disciplines used to function, with disastrous results. The Tacoma Narrows Bridge collapsed because people were designing bridges, in those days, with no thought whatever to proving that they would not collapse."[100,p.628] DeMillo, Lipton, and Perlis found this claim "a complete distortion of fact, and to suggest that engineers [engage in formal proofs of correctness] . . . now is simply false."[101]

Maurer's view was indeed a distortion and a revealing one. The Tacoma Narrows Bridge was the first suspension bridge to connect the mainland of Washington State with the Olympic Peninsula. The bridge demonstrated a pronounced tendency to undulate and tore itself apart only months after it opened in 1940. Analysis after the fact revealed that the bridge had behaved in a fashion similar to an airplane wing in uncontrolled turbulence. As Henry Petroski notes, the problem was not the result of a failure to check the design. Rather, "the possibility of failure of the Tacoma Narrows Bridge in a crosswind of forty or so miles per hour *was completely unforeseen by its designers, and therefore that situation was not analyzed* [emphasis added]. On paper the bridge behaved well under its own dead weight and the light traffic it was to carry."[102] The problem did not reside within the realm of verification, but within that of design. Just as unforeseen conditions produce software errors, so, too, did they produce the Tacoma Narrows Bridge failure. Formalism is of no help in such instances. That Maurer believed so illustrates how the debate over such issues was often clouded by confusion over the nature of engineering (and science).

DeMillo, Lipton, and Perlis were hardly alone in their doubts over the usefulness of formal verification. Richard Hill of A.C. Nielsen Management Services commented that he could not recall "a single instance in which a proof of a program's correctness would have been useful."[103] H. Lienhard of Switzerland applauded even louder: "It was time somebody said it—loud and clear—the formal approach to software verification does not work now and probably never will work in the real programming world. . . . There is one dimension that is crucial in 'real-life' programs: complexity. The problem of software engineering is usually not the finding of 'deep theorems' but rather the highly nontrivial task of mastering complexity."[104] All this, however, was a replay of an earlier debate in the pages of *Software Engineering Notes*. An earlier version of "Social Processes and Proofs" had been presented at the 1977 ACM Symposium on Principles of Programming Languages, and it had prompted a strong response from Dijkstra. Terming it "a very ugly paper" in "the style of a political pamphlet," Dijkstra protested that the authors "just ignore that how to prove—not in the silly ways they depict, but more elegantly—'the correct functioning of particular pieces of software' is the subject of a lively interchange of experiences between scientists active in the field."[105] "Unaware that the 'problems of the *real* world' are those you are left with when you refuse to apply their effective solutions, they confirm the impression of anti-intellectualistic reactionaries...."[105,p.15] DeMillo, Lipton, and Perlis did not take this lying down, refusing to concede that their confidence in a piece of "real" software had ever been increased by a proof of correctness. They also maintained that "the verifications . . . are long, ugly, and boring, no matter how concise, elegant, and fascinating the idea of verification may be. If verifications of real programs are currently being socialized, Professor Dijkstra should have no trouble pointing to the channels of communication."[106] In response to a Dijkstra position paper on reliability, H.J. Jeffrey of Bell Labs contended that if one examined what people actually did, "what emerges is that formal correctness is really a peripheral issue in software reliability, which is primarily concerned with how to do a good software job without formal correctness proofs."[107] Here again was a view concerned with practical accomplishment rather than the enshrinement of absolutes.



Even if it was not a chimera, program correctness still guaranteed only that the implementation matched the specifications. This was of dubious value, as the Tacoma Narrows Bridge so amply demonstrated, if the specifications themselves were flawed. A 1975 *Transactions* article examined data from both real and experimental software with the aim of better understanding software errors. The authors concluded that "the ability to demonstrate a program's correspondence to its specification does not justify complete confidence in the program's correctness since a significant number of errors are due to incomplete or erroneous specification...."[108] The difficulty of producing complete and correct specifications was at the heart of a 1977 *Transactions* article by Douglas Ross and Kenneth Schoman, Jr., of SofTech. While contending that the problem was not insurmountable, they nevertheless observed that software designers "attempt to do the same [requirements definition as manufacturers] of course, but being faced with greater complexity and less exacting methods, their successes form the surprises, rather than their failures! Experience has taught us that system problems are complex and ill-defined. The complexity of large systems is an inherent fact of life with which one must cope."[109]

Why, then, was formal verification so appealing? Gerhart proffered a telling observation at a 1978 Workshop on Software Testing and Test Documentation. Academic researchers, she suggested, "have found program proving far more attractive, with its logical mathematical origins and possible integration with the programming process, than testing with its statistical and experimental origins and a posteriori programming phase."[110] But even Dijkstra had expressed some doubts about the hope held by a number of academic researchers that the verification process could somehow be made easy, that one could enjoy the fruits of formal mathematics without paying a price. In a 1975 essay in *SIGPLAN Notices*, he contemplated attempts to automate the process:

> We see automatic theorem provers proving toy theorems, we see automatic program verifiers verifying toy programs and one observes the honest expectation that with faster machines with lots of concurrent processing, the life-size problems come within reach as well. But, honest as these expectations may be, are they justified? I sometimes wonder....[111]

Clearly, though, the stance people took with regard to testing versus formal verification was at least partially a function of how they perceived themselves. Self-perceived scientists might develop a very different view than self-perceived engineers. Where you sit sometimes determines where you stand.

Somewhere between the true believers and the heretics resided what a number of practitioners regarded as the pragmatic middle ground. Andrew Tanenbaum suggested that correctness proofs "have their place, but they can easily lull one into a false sense of security, and therein lies the potential danger." He viewed testing and formal verification as complementary rather than competing approaches.[112] Likewise, Gerhart and Lawrence Yelowitz concluded after examining a variety of supposedly correct programs that "experience with both testing and mathematical reasoning should convince us that neither type of evidence is sufficient and that both types are necessary."[113] In a similar vein, Parnas opined, "both sides hold to such extreme positions that convergence on the truth, which both are seeking, is not possible." He attributed this divergence to a misguided analogy between programming and mathematics; the proper analogy compared programming with engineering.[114] Yet, either analogy was bound to discomfit those left out. Moreover, as has been noted, not everyone subscribed to the distinction. Acknowledging the somewhat "dirty" nature of engineering, Parnas maintained that engineering mathematics "need not meet the standards set by mathematicians because it is not the only way to test an engineering design." He, too, advocated a combination of testing and formal verification as a means of increasing confidence in software.[114] A 1985 *Transactions* article proposed a method—partition analysis—that attempted just such an integration.[115]

For almost a decade following DeMillo, Lipton, and Perlis's attack, the formal verification issue remained relatively quiescent, with each camp seemingly content to go its own way. The peace was shattered once again, though, in the pages of *Communications* in 1988. James Fetzer, a professor of philosophy at the University of Minnesota, proceeded to drop another bombshell on the formal verificationists. In "Program Verification: The Very Idea," Fetzer argued that DeMillo et al. had reached the right conclusion for the wrong reason. While acknowledging their point about the necessity of social processes in proof validation, Fetzer contended that such processes could in principle be incorporated into formal verification and were therefore not an intractable obstacle to it. According to Fetzer, there was still an inescapable problem that cast doubt on the claims of the verificationists. Fetzer argued for

> the theoretical necessity to distinguish programs as encodings of algorithms from the logical structures that they represent. . . . Algorithms, rather than programs, thus appear to be the appropriate candidates for analogies with *pure mathematics*, while programs bear comparison with *applied mathematics*. Propositions in applied mathematics, unlike those in pure mathematics, run the risk of observational and experimental disconfirmation.[116]

Clearly, though, the stance people took with regard to testing versus formal verification was at least partially a function of how they perceived themselves.

To simplify a fairly complex argument, Fetzer's case centered on distinctions between absolute and relative verification and between abstract and physical machines. Absolute verification concerns conclusions derived only from primitive axioms while relative verification concerns conclusions derived from premises whose truth cannot be absolutely verified. Thus,

> the properties of abstract machines that have no physical machine counterparts can be established by definition, i.e., through stipulations or conventions, which might be formalized either by means of program rules of inference or by means of primitive program axioms. . . . By comparison, programs [meant to be compiled and run on real machines] . . . are merely subject to relative verification, at best, by means of deductive procedures. Their differences from algorithms arise precisely because, in these cases, the properties of the abstract machine they represent, in turn, stand for physical machines whose properties can only be established inductively.[116,p.1,058]

In other words, programs intended for execution on computers "cannot be subject to absolute verification, precisely because the truth of these axioms depends upon the causal properties of physical systems, whose presence or absence is only ascertainable by means of inductive procedures. . . . This conclusion strongly suggests the conception of programming as a mathematical activity requires qualification in order to be justified."[116,p.1,059] The gist of

Fetzer's argument, then, was that because programs run on actual computers in an empirical reality subject to all kinds of complex and often unexpected interactions, rather than on abstract computers in a closed, mathematical system, programming had to be seen as applied (and thus less than certain) mathematics rather than as pure mathematics (amenable to absolute verification) as many formal verificationists seemed to view it.

Whether the resulting furor exceeded that prompted by the "Social Processes" paper is arguable, but it was certainly the equal of it. One of the strongest salvos was a joint attack launched by 10 distinguished computer scientists. The "Gang of Ten," as Fetzer dubbed them, included Basili, Gerhart, David Gries, Nancy Leveson, and Peter Neumann. According to this outraged group, Fetzer's article "is not a serious scientific analysis of the nature of verification. The article distorts the practice and goals of program verification and reflects a gross misunderstanding on the part of the author about the nature of program verification. This article does not meet minimal levels of serious scholarship." They went on to damn the editors as well, contending that "by publishing the ill-informed, irresponsible, and dangerous article by Fetzer, the editors of *Communications* have abrogated their responsibility, to both the ACM membership and to the public at large...."[117] The editors stood by their decision, while Fetzer, displaying no second thoughts either, proceeded to return fire. After inviting the Gang of Ten to accompany cruise missiles on future flights in order to demonstrate the feasibility of constructing verifications of dynamic (self-modifying) programs, Fetzer declared that "in its inexcusable intolerance and insufferable self-righteousness, this letter exemplifies the attitudes and behavior ordinarily expected from religious zealots and ideological fanatics, whose degrees of conviction invariably exceed the strength of the evidence." Fetzer was supported in this view by one reader who "having read the vitriolic, unjustified, unreasoned attacks on Fetzer," suspected that "at least some defenders of program verification can find no real arguments to rebut Fetzer's contentions and resort to meaningless insults in a desperate attempt to defend a position that cannot be logically defended."[118]

Beneath all the verbal barbs, however, lay a legitimate point of contention. One of the principal criticisms leveled at the Fetzer article, by the Gang of Ten and by others in somewhat more measured terms, was that it attacked a straw man, a "parody" of formal verification. For example, one reader commented that the article "does a disservice to the cause of the advancement of the science of programming by belaboring the rather obvious fact that programs which are run on real machines cannot be completely reliable, as though advocates of verification thought otherwise."[119] Another contended that it "makes one important but elementary observation and takes it to an absurd conclusion."[120] Fetzer responded that such complaints were without merit inasmuch as "the principal position under consideration with respect to program verification, no doubt, is that of C.A.R. Hoare and those [such as Dijkstra] who share a similar point of view, a matter about which my article is quite explicit."[121] John Dobson and Brian Randell at the University of Newcastle Upon Tyne suggested that the problem was essentially one of misleading rhetoric, that although formal verificationists did not truly believe in the possibility of absolute verification, they nevertheless *sounded* as if they did, hence the confusion.[122] This, however, seemed to ignore a distinction Fetzer had made earlier. "I am not promoting the view that program verification purports to provide absolute certainty, but rather attacking the belief that this might be possible."[123] A couple of readers, though, apparently felt the problem was more than one of miscommunication, with one commending Fetzer on exposing "the naivete of computing researchers in general and their illusions concerning the relevance of mathematical formalisms in particular."[124] In this, though, he seemed to go far beyond Fetzer's own views, for Fetzer repeatedly emphasized that he was not arguing that formal verification was, by definition, illegitimate, but rather that its use had to be accompanied by an understanding of its limitations, limitations that suggested that "the techniques of program verification have to play a much more limited role in assuring the production of high quality software than its advocates suggest."[125] Such an attitude seems to place Fetzer, despite the view of the formal verification community, closer to the pragmatic middle ground than to the antiverification extreme.

Pragmatism also manifested itself in the explicit observation that dogmatic insistence on *perfect* programs was likely to produce more frustration than achievement. In a 1976 *Transactions* editorial, Leon Stucki of McDonnell Douglas advocated a design philosophy aimed not at producing error-free programs but at producing easily testable software.[126] Later that year, C.V. Ramamoorthy and colleagues suggested what they termed "partial validation." "Partial validation is a practical approach which can be used to establish a *sufficient degree* [emphasis added] of confidence in the reliability of a program. This approach partitions program characteristics into a number of classes and then validates each class to a specified extent."[127] A similar point of view was articulated two years later in a *Transactions* article on software reliability models. The authors stated flatly, "It is neither necessary nor economically feasible to get 100 percent reliable (totally error-free) software in large, complex systems."[128]

Accepting this, however, raised the question of what could be done to ensure that residual errors would be merely inconvenient rather than disastrous. The answer was to make software "fault-tolerant." As Leveson of the University of California at Irvine asserted in a 1982 piece in *Software Engineering Notes*,

> since removal of all faults and perfect execution environments cannot at this point in time, and perhaps never will, be guaranteed . . . there is incentive to make software fault-tolerant. In this approach, it is assumed that run-time errors will occur, and techniques are used to attempt to ensure that the software will continue to function correctly in spite of the presence of errors.[129]

Software fault tolerance differed from traditional engineering safety factors, however, in that the latter is a matter of physical tolerances while the former involves detection of and recovery from unforeseen errors. One approach Algirdas Avizienis and John Kelly of UCLA championed was to develop multiple independent versions of a program—N-version programming. Independent development efforts would supposedly produce programs unlikely to contain the same errors. "The obvious advantage of design diversity is that reliable computing does not require the complete absence of design faults, but only that those faults not produce similar errors in a majority of the designs."[130] The space shuttle program employed a similar scheme in the development of its basic flight software to guard against "generic" software errors.

One can only imagine the reaction of those involved when, in the second half of the 1980s, doubts were raised as to the validity of the assumption underlying multiversion programming. Writing in *IEEE Transactions on Software Engineering*, two National Aeronautics and Space Administration researchers observed that recent research had "demonstrated that programmers given the same task are prone to make mistakes that potentially reduce the effectiveness of a fault-tolerant approach."[131] Such mistakes could potentially produce "coincident failures" in which two or more program versions would fail (albeit not necessarily in precisely the same way) given identical input. This raised the possibility, in a majority voting scheme, of correct versions of the program being outvoted by the incorrect versions.

While probably no one expected that an assumption of statistically independent failures could ever fully hold, it was nevertheless the theoretical heart of the argument. If failures were not at least highly independent, the utility of N-version programming was seriously undermined. Leveson and John Knight of the University of Virginia raised more doubts the following month when they described an experiment that seemed to confirm this. They noted:

> it is assumed in some analyses of the technique that the N different versions [of a particular program] will fail *independently;* that is faults in the different versions occur at random and are unrelated. . . . We are concerned that this assumption might be *false.* Our intuition indicates that when solving a difficult intellectual problem (such as writing a computer program) people tend to make the same mistakes . . . even when they are working independently. Some parts of a problem may be inherently more difficult than others.[132]

Their experiment confirmed their fears, revealing a surprisingly high number of coincident failures in a set of independently developed programs. They cautioned, however, against overgeneralization, emphasizing that the independence-of-errors assumption had only been shown invalid for the particular problem that was programmed. Their caveats, however, did not deter Avizienis and his colleagues from repeatedly charging that their findings were flawed as a result of key experimental differences and inadequate development methods. Knight and Leveson finally felt compelled to answer this constellation of criticisms with an in-depth response in *Software Engineering Notes* in 1990. They contended not only that the criticism was unfounded but also that in many respects their experiment more accurately reflected the ideas Avizienis et al. espoused than the latter's own work.[133] In any event, other researchers suggested it might be more effective to pursue directly a quality other than statistical independence. In 1989, Bev Littlewood of City University London and Douglas Miller of George Washington University argued, "the achieved level [of diversity of program versions] will depend on the diversity of the *processes* (software development methodologies) used in their creation."[134] They contended that statistical independence of program versions was a misleading goal; the real goal had to be *diversity,* including diversity of development method. The following year, however, Knight, Leveson, and another colleague suggested diversity of process was not necessarily of much help. In a follow-up to the 1986 article, they maintained:

> simple methods to reduce correlated failures arising from logically-unrelated faults (i.e., input-domain related faults) do not appear to exist. The faults that induced coincident failures were not caused by the use of a specific programming language or any other specific tool or method, and even the use of diverse algorithms did not eliminate input-domain related faults. In most cases, the failures resulted from fundamental flaws in the algorithms that the programmers designed. Thus we do not expect that changing development tools or methods, or any other simple technique, would reduce significantly the incidence of correlated failures in N-version software.[135]

Here again was evidence that fundamental problem-solving processes lay at the heart of software development. It also amounted to rediscovery of a fact that had been intimated years before: Statistical reliability techniques developed for hardware would not work for software.

Pragmatism thus moved the question from one of *correctness* to one of *confidence.*

This did not mean, though, that statistically based approaches to reliability had no applicability to software. But they had to approach things from a different perspective, one that incorporated the local knowledge and attributes of particular programs operating in particular environments. While a variety of software reliability models existed by the mid-1980s, hopes for a single definitive universal model had not been fulfilled. No single model seemed to perform well in all situations. "More importantly," a 1986 article contended, "it does not seem possible to analyze the *particular* context in which reliability measurement is to take place so as to decide *a priori* which model is likely to be trustworthy. . . . [However] if a user knows that past predictions emanating from a model have been in close accord with actual behavior *for a particular data set* then he/she might have confidence in future predictions for the same data."[136] The authors proceeded to describe some tools to assist in the selection of an appropriate model. Indeed, in his introduction to a special section on software testing in the June 1988 issue of *Communications,* the guest editor argued that while particular types of statistical approaches might be problematic, nevertheless "probabilistic analysis seems appropriate for testing theory because it is capable of comparing methods and assessing confidence in successful tests."[137] In this context, as in so many others related to software, the dictum "different horses for different courses" found increasing favor. Even so, speaking at the 1989 World Computer Congress, Parnas still felt compelled to decry narrow focuses and false dichotomies when dealing with issues of software reliability.[138]

Complexity proved inhospitable to dogmatism regarding both the means and goal of verification. Neither testing nor proofs could guarantee a "correct" program at reasonable cost, if at all, and some practitioners questioned the necessity of error-free software. Inhabitants of the middle ground advocated strategies combining both testing and formal proofs, while admitting the unlikelihood of total confidence. Pragmatism thus moved the question from one of *correctness* to one of *confidence.* For those who viewed their work ultimately in terms of science and mathematics, though, the operative notions were those of truth and falsity. MacKenzie illustrates just how problematic this dichotomy is for both hardware and software.[139] Those more conscious of the nature of engineering accommodated themselves to the notion of confidence. As the 1983 National Bureau of Standards guidelines for federal information processing systems admonished, “No single VV&T [validation, verification, and testing] technique can guarantee correct, error-free software. However, a carefully chosen set of techniques for a specific project can help to ensure the development and maintenance of quality software for that project.”[140] Similarly, in their introduction to a special 1989 issue of *IEEE Software* focusing on verification and validation, the guest editors observed that “a V&V effort selects tasks from a broad spectrum of analysis and test techniques to tailor each V&V effort to the project’s needs.”[141] A complex reality did not easily accommodate desires for absolutes; instead, practitioners were forced to accommodate the limitations that complexity of both program and process imposed.

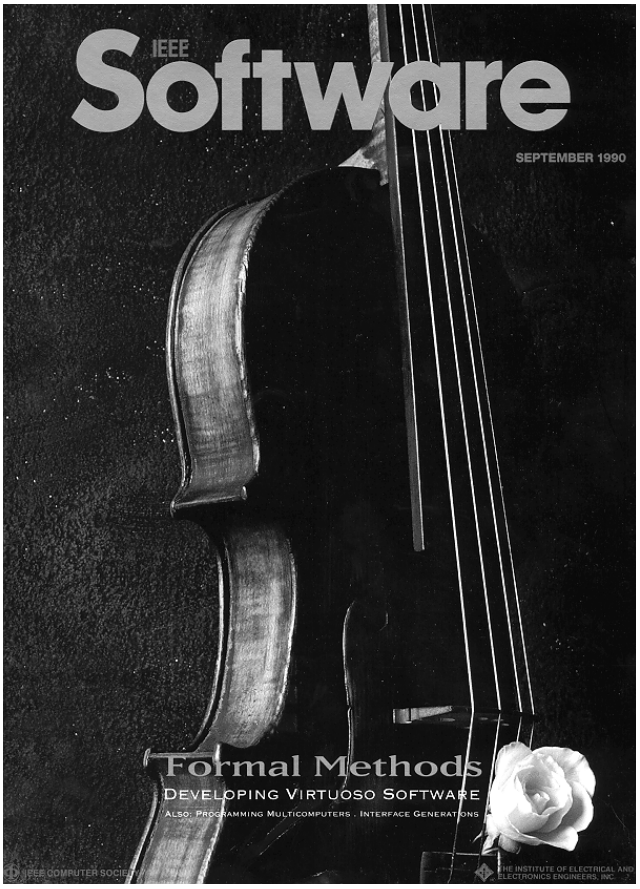
In Search of Rigor: The Formal Methods Movement

While in one sense the issue of formal verification was one facet of the reliability question, in another sense it was the most prominently divisive aspect of a larger debate over formal methods more generally. Such methods eventually covered the full spectrum of software development and maintenance activities. What linked them was their emphasis on mathematically based notations and methods of reasoning. This addressed what many of their advocates viewed as the principal deficiency of software practice: sloppy, fuzzy, and ad hoc thinking. Formal methods, they believed, would counteract such tendencies. They would enforce disciplined approaches to problem solving by requiring precise logical reasoning. Their use would by definition make would-be software engineers more scientific and thus more professional. The result would be better software and a better public image for those producing the software.

Not surprisingly, Hoare was one of the most prominent standard-bearers for formal methods (along with Mills and Dijkstra). (Recently, Hoare has softened his view considerably, admitting that less formal methods, including engineering intuition, have proven surprisingly effective in producing relatively reliable systems. He still feels, however, that formal methods have a role to play in the development of safety-critical and security-critical systems.) One of his first major declarations in this regard came in his famous 1969 *Communications* article, “An Axiomatic Basis for Computer Programming.” It was here, in the wake of the 1968 NATO conference on software engineering, that Hoare argued that programming was “an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning.”[142] He went further in 1981, claiming:

> we have only recently come to the realisation of the mathematical and logical basis of computer programming; we can now begin to construct program specifications with the same accuracy as an engineer will survey a site for a bridge or road, and we can now construct programs proved to meet their specification with as much certainty as the engineer assures us his bridge will not fall down. Introduction of these techniques promises to transform the arcane and error-prone craft of computer programming to meet the highest standards of a modern engineering profession.[143]

If nothing else, Hoare’s remarks suggest a limited appreciation of the history of bridge building, which, like virtually every other realm of engineering practice, has never enjoyed the sort of certainty that Hoare seems to attribute to it. The assumption of imperfect knowledge and the use of approximations are part and parcel of civil engineering. Indeed, that is one of the main rationales for incorporating safety factors into design calculations. The sentiments Hoare expressed illustrate the misconceptions that continued to plague software engineering regarding science, engineering, and the relationship between them.[144]


IEEE
Software
SEPTEMBER 1990
Formal Methods
Developing Virtuoso Software
Also: Programming Multicomputers . Interface Generations
IEEE Computer Society
The Institute of Electrical and Electronics Engineers, Inc.

Formal methods advocates such as Hoare left little doubt that they equated “informal” methods with “arcane and error-prone” programming. Others, however, saw a role for both perspectives. For instance, among the benefits Leveson ascribed to formal methods in her introduction to a special issue of *Transactions* were “rigor and precision including unambiguous communication, prediction, evaluation, and better understanding and control over software products and the software development process.” Noteworthy, however, was her attendant observation:

> We need not only better formal methods but also ways of interfacing them to human abilities and less formal methods. There is much to be gained from investigating the process of integrating formal methods with informal software engineering procedures, e.g., determining how they

> can be used together in a complementary fashion to take advantage of the strengths of each.[145]

Gerhart echoed this point in her introduction to a companion special issue of *IEEE Software*: "[T]he next challenge is to integrate these formal methods with the variety of informal techniques (like design records, conceptual modeling, and graphical representations) required to achieve the goal of a formally based engineering discipline."[146] Here, it seems, was an attempt to reconcile a pluralistic reality with the singularity exhibited by the formalists' professional rhetoric.

Increasingly, formal methods advocates were not of one mind with respect to the sufficiency of formal methods alone. Introducing a special issue of the *Software Engineering Journal* devoted to theorem proving and software engineering, C.B. Jones of the University of Manchester was careful to note that mathematical methods were neither "panacea" nor "quack remedy." Rather, "there are, in fact, many useful approaches which will make contributions to various application and/or development environments. Specialist ('fourth-generation') languages, Prolog, functional languages, prototyping and others all have a contribution to make."[147] At a 1989 formal methods workshop, participants reportedly leavened their insistence on the necessity of formal methods with the caution that formal methods alone were insufficient for development of trustworthy systems.[148] That year's International Conference on Software Engineering presented the formal methods debate in microcosm, with believers emphasizing the need for greater attention to formal methods, skeptics arguing the superiority of "intuition and guessing," and others calling for "considered application" of formal methods depending on individual circumstances.[25,p.109]

Articles appearing in the late 1980s and early 1990s lent substance to this sort of pragmatism. Writing in *IEEE Software* in 1990, Anthony Hall related experience with formal methods at Praxis, a British software engineering company. He noted that "*even though we have undertaken very few proofs or completely formal development steps*, we have found that inspections of formal specifications reveal more errors than those of informal specifications, and it is more effective to inspect designs or programs against formal specifications than against other kinds of design documentation [emphasis added]." He made it clear that "program verification is only one aspect of formal methods. In many ways, it is the most difficult. For non–safety-critical projects, program verification is far from the most important aspect of a formal development."[149] Moreover, he argued that it was unrealistic to expect most software engineers to easily and routinely carry out formal proofs and that proof tools were primitive and possibly condemned to remain that way.[149,p.17] An article in that month's *Computer* suggested that such pragmatism could be found in academia as well. In a broad introduction to formal specification, Jeannette Wing of Carnegie Mellon University noted, "Although you may never completely verify an entire system, you can certainly verify smaller, critical parts."[150] Another way around the difficulties of formal verification, though, was to change the nature of formal verification. This was a key part of an integrated process dubbed Cleanroom software engineering.

An approach Mills and others developed at IBM, Cleanroom software engineering, was presented as "a practical process to place software development under statistical quality control."[151] While highly formalized, the Cleanroom process nevertheless embodied several concessions to practicality. Foremost among these was the verification process used:

> The method of human mathematical verification used in Cleanroom development, called functional verification, is quite different from the method of axiomatic verification usually taught in universities. It is based . . . on the reduction of software verification to ordinary mathematical reasoning about sets and functions as directly as possible. . . . By introducing verification in terms of sets and functions, you establish a basis for reasoning that scales up.[151,p.22]

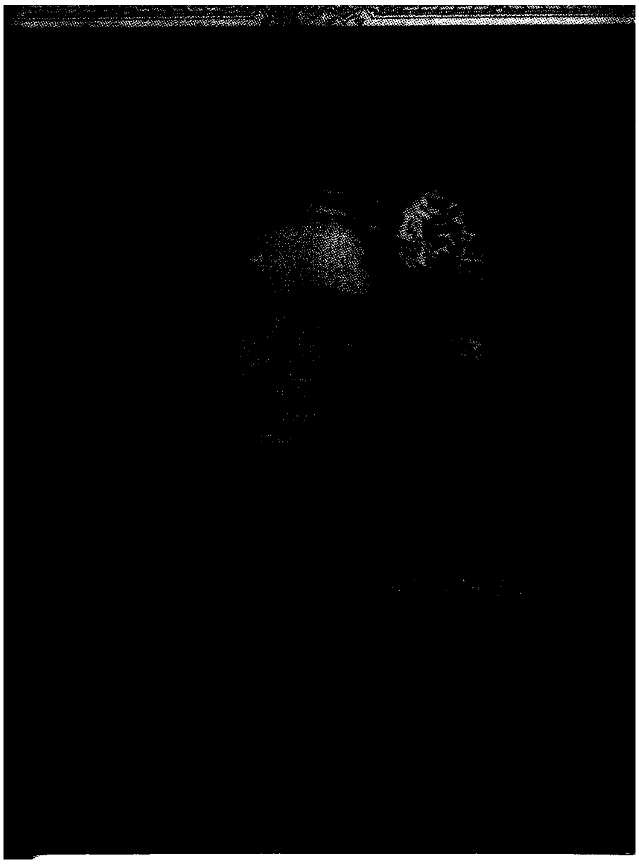
A related key feature of the Cleanroom was that development and verification were both iterative and cumulative. Incremental development meant in theory that only relatively small pieces of programming would ever have to be verified. A further aid to formal verification was the use of a limited set of design primitives within the software. Another sign of pragmatism in the Cleanroom scheme was that "structural testing that requires knowledge of the design is replaced by formal verification, but functional testing is retained."[151,p.22] A statistical usage profile provided the basis for this testing. An experiment reported in *Transactions* seemed to provide support for the efficacy claims made by advocates of the Cleanroom.[152] However, it should be noted that the size of the programs used as examples of the success of the Cleanroom approach represented both how far the technique had come and how far it still had to go. Most of these programs involved fewer than 50,000 lines of instructions, which was still an impressive amount of formally verified code. Nevertheless, with major systems requiring hundreds of thousands and even millions of lines of code, the practicality of using the Cleanroom approach for such systems was still an open question. More importantly, one aspect of the Cleanroom process that was distinctly unpragmatic was its insistence on stable specifications.[151] Clearly, while this demand may be relatively easy to meet in some contexts, it may be virtually impossible in others.

Mathematical methods were neither "panacea" nor "quack remedy."

Thus the issue of applicability raised its head once again. Some formal methods advocates, though, were beginning to display a heightened awareness of its importance. Wing, for instance, explicitly acknowledged the issue of applicability as pertaining to both specification languages specifically and formal methods generally, emphasizing that "an advocate of a particular formal method should tell potential users the method's domain of applicability. . . . Without knowing the proper domain of application, a user may inappropriately apply a formal method to an inapplicable domain."[150,pp.12-13] Indeed, a 1987 *Computer Journal* piece compared two different approaches to formal specification: the Vienna Development Method and OBJ. The authors concluded, "The two approaches each lend their own insights to a problem. VDM [Vienna Development Method] encourages a more 'top-down' approach to viewing a problem, while OBJ may be used in a more 'bottom-up' style which gives fresh ideas on how to partition the problem and how to structure the specification. The overall experience was that the two methods complemented each other."[153] Even Hoare seemed to be mellowing somewhat, admitting in *Computer* in 1987 that the small and familiar example that had been used to illustrate some formal methods for program design

> revealed (all too clearly) the full weight of the notations and complexity of the mathematical proofs involved in formalization of the process of program design. The reader may well be discouraged from applying these methods to problems of a scale more typical of software engineering. And there are many other serious concerns which are not addressed directly by formalization, for example, cost estimation, project management, quality control, testing, maintenance, and enhancement of the program after delivery.[154]

Nevertheless, in 1990 the associate editor-in-chief of *IEEE Software* deplored what he saw as the ever-widening divide between software engineering purists and real-world practitioners, "The consequence is that practitioners are drifting toward the north pole and purists toward the south pole (or vice versa—either side is very cold). Those researchers who do take a more pragmatic approach and those practitioners who see the value of formal methods are trying to decide if they should move north or south."[155] While there was clearly some movement toward the equator, it seemed there was still a great deal of drift toward the poles.



The formal methods debate embodied virtually every type of tension extant in the software engineering and computer science communities: academia versus industry, research versus practice, science versus engineering. From the other side of the road, formal methods often appeared as the esoteric playthings of an elite unconcerned with the circumstances of real-world software development. Formal methods advocates viewed their critics as stubborn and archaic craftsmen, either unwilling or unable to adopt self-evidently superior techniques built on science and mathematics. Nevertheless, there did exist a middle ground that sought a balanced, integrated approach combining formal methods with other techniques so as to most effectively deal with the particular problem at hand. By the 1990s, the population of this middle ground was slowly growing, but the underlying tensions still remained.

The Sound and the Fury: Language Disputes

Just as verification proved unamenable to any one approach, so, too, did programming (and, more importantly, programmers) appear resistant to any single language. The area of programming languages has always provided rich grounds for controversy, perhaps because the issue of programming language is so basic and inescapable for practitioners that it inevitably generates strong emotions. The enduring tension between language generality and specificity played itself out in several arenas. The concept of a universal language effective in virtually all circumstances (ALGOL, PL/I) continued to attract hearts and minds as it had in the 1960s, while others touted powerful application-oriented languages usable by nonprogrammers (so-called fourth-generation languages) as well as special-purpose languages aimed at particular domains (such as NewSpeak, intended for safety-critical programs[156]). At the same time, the old-guard languages—principally Fortran and COBOL—continued to thrive and, to the distress of many, evolve.[157]

Those who enjoyed a good language controversy soon enough had one to rival the disputes over ALGOL and PL/I. In January 1975, the U.S. Department of Defense (DoD) Director of Defense Research and Engineering set up a department-wide program to develop a single common high-level military programming language for embedded systems. (An embedded computer system is one that is an integral part of some larger system, e.g., the computers used to control a modern jet fighter.) A High Order Language Working Group was established to carry out this program. David Fisher of the Institute for Defense Analyses described the effort as "based on the idea that many of the support costs for software increase with the number of languages, and that languages must be suited to their applications. Furthermore, with a common programming language, a software development and maintenance environment could be built, providing centralized support and common libraries that could be shared...."[158] DoD difficulties with software mirrored those in the larger world. A study earlier in the decade by the Air Force Systems Command—"Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980s"—had confirmed, as Barry Boehm conveyed to *Datamation* readers, that "for almost all applications, software . . . was 'the tall pole in the tent'—the major source of difficult future problems and operational performance penalties."[159] Fisher, however, revealed unusually modest expectations, "The present diversity of programming languages used in embedded computer systems did not cause most of the problems—nor would a common programming language cause them to disappear. Nevertheless, the existing language situation unquestionably aggravates them and inhibits some potential solutions."[158,p.26] Recalling some of the expectations that accompanied ALGOL and PL/I, such a view seems atypical in its pragmatism. The extent to which others shared this view of the DoD effort was another matter.

Since no existing language satisfied all the requirements with respect to embedded applications, reliability, maintainability, and machine independence, the High Order Language Working Group decided to develop a new language. From 1975 to 1977, the group concentrated on iteratively developing a set of language requirements in consultation with all interested communities. In the summer of 1977, the working group selected four contractors to propose initial language designs. These prototype designs were evaluated by numerous review teams from academia, government, and industry. In the spring of 1978, the working group narrowed the competition to two proposals, which were then further developed and refined along with prototype processors. After another round of evaluation, the working group selected Cii-Honeywell Bull's language in the spring of 1979 and christened it Ada, after Lady Ada Lovelace, the world's "first" programmer. For the remainder of 1979, Ada was subjected to additional testing and refinement.

If anyone believed the product of this effort would be uncontroversial, they soon learned otherwise. No less a personage than Dijkstra took a dim view of the proceedings:

> It is so illuminating because it shows in a nutshell what havoc is created by not stating your goals but only prescribing partial means intended to solve your problems. . . . It makes also quite clear why the new programming language cannot be expected to be an improvement over PASCAL, on which the new language should be "based." . . . You cannot improve a design like PASCAL significantly by only shifting the "centre of gravity" of the compromises embodied in it; such shifts never result in a significant improvement. . . . Why does the world seem to persist so stubbornly in being such a backward place?[160]

A 1979 report in *Datamation* noted that while the Ada requirements study suggested that one language could in theory support most application areas, that "does not, of course, imply that it is *desirable*."[161] "The Ada language control people will have a very difficult task. They must attract the reluctant services, hold the language stable but correct . . . and not let multiple implementations create language anomalies by different interpretations of the language. Historically, this latter problem has seldom, if ever, been solved."[161,p.150] Writing in *SIGPLAN Notices*, Rob Kling and Walt Scacchi expressed skepticism on sociological grounds. Noting the attractiveness of technical fixes that allowed one to "focus on designing technologies which can be high spirited fun rather than upon the human dilemmas which can be woefully depressing," they saw "little reason to believe that projects which use DoD-1 [which would become Ada] are guaranteed lower life-cycle costs than similar projects which do not, when the projects are executed in routine production environments under routine contractual and market arrangements (and not as showcases for DoD-1 use)."[162]

Given that Ada was intended to be almost all things to all people, language complexity was a bone of contention, just as it had been in the cases of ALGOL and PL/I. Paul Eggert of UCLA suggested that Ada was yet another example of the "Wish List Syndrome." He accused Ada of being to Pascal what PL/I was to Fortran—an unwieldy conglomeration of features.[163] In a similar vein, another *SIGPLAN Notices* reader contended that the complexity of the language would encourage the use of language subsets leading to incompatible implementations and styles and perhaps even dialects.[164] In fact, one of the most hotly debated issues concerned the question of language subsets as a means of reducing the effective complexity of the language.

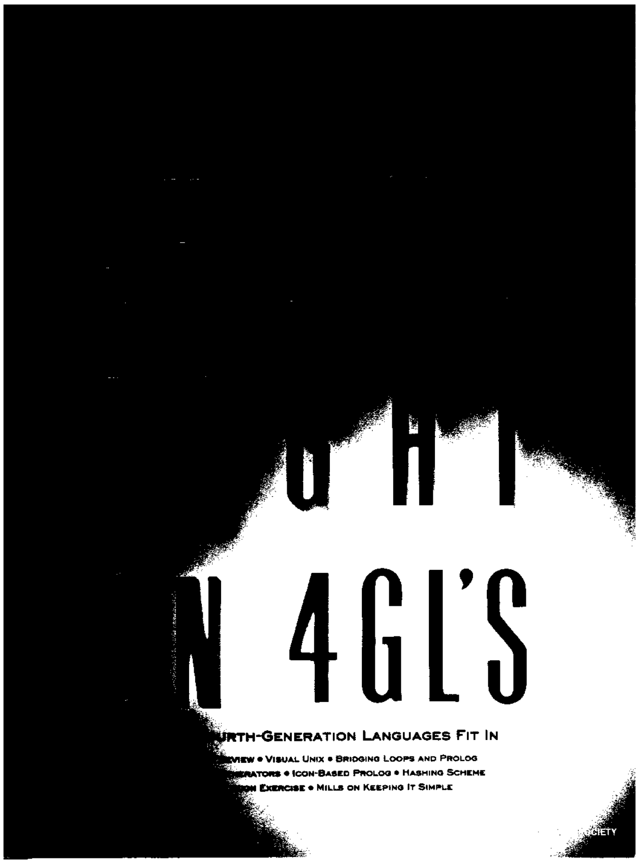
In his 1980 Turing Award Lecture, Hoare despaired that with Ada, the "mistakes which have been made in the last twenty years are being repeated today on an even grander scale."

The ACM voted against approval of Ada as an American National Standards Institute (ANSI) standard, partly in reaction to the absence of subsets that were reliable (i.e., produced identical results across compilers) and efficient (in terms of compilation). The organization argued that if there were, in fact, "numerous potential commercial applications—not limited to 'embedded systems'—and . . . these applications cover a broad range of complexity, then there is a strong and—we believe—valid argument for the definition of one or more 'authorized' subsets."[165] In 1982, Ledgard and Andrew Singer advocated in *Communications* either scaling down or subsetting Ada, "As strong supporters of the Ada effort, we are concerned that in the long run the language will fail with users for the same reason that other large languages have failed—not enough was left out."[166] Robert Glass agreed that "simplicity is to be sought. Practitioners, however, have ever more complex problems to solve. The goal of simplicity must never take precedence over the goal of problem-solving."[167] Randall Leavitt did not care for the idea of Ada subsets, but acknowledged that Ada might be a little too substantial, "My experiences with Fortran transportability and maintenance indicate that a subset is only another problem to overcome, not a solution. However, Ada would benefit from some pruning."[168] DoD, not surprisingly, also took issue with the notion of Ada subsets, arguing that subsetting "would potentially defeat the portability of applications software, libraries, reusable components, and programmers."[165] As for pruning the language, Brian Wichmann, a member of the Ada design team, asserted that while Ada could be simplified by reducing its facilities, "it is far from clear . . . that the resulting language will be as useful to the user community especially in the long run."[169] The question, though, was useful in what sense?

Clearly, Ada, with its smorgasbord of features, was potentially of great utility. But potential utility does not automatically translate into practical utility. The potential utility of Ada could well be vitiated by its bulk and complexity. In other words, Ada, like other attempts at a universal language, might be too far beyond the pivot between generality and specificity—the point at which trade-offs seem to balance—to appeal to as wide an audience as its proponents hoped.

Ada certainly did have its proponents. William MacGregor of the University of Texas, responding to Dijkstra's complaints about the four candidate designs, opined, "Alternatives to the common language being what they are, there is room for a great deal of imperfection in the new language while still achieving a substantial economic advantage."[170] Peter Wegner, while admitting Ada was comparable in complexity to PL/I and thus vulnerable to the same kind of criticisms, contended that Ada was "better engineered than Pascal or PL/I. . . . The resulting language has more expressive power and greater security and reliability than either Pascal or PL/I."[171] Hoare, however, believed that reliability and the kind of complexity Ada exhibited were mutually exclusive. In his 1980 Turing Award Lecture, Hoare despaired that with Ada, the "mistakes which have been made in the last twenty years are being repeated today on an even grander scale."[172] The depth of Hoare's concern was evident in his appeal to

> not allow this language in its present state to be used in applications where reliability is critical, i.e., nuclear power stations, cruise missiles, early warning systems, antiballistic missile defense systems. . . . An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations."[172,p.83]


N 4GL'S
RTH-GENERATION LANGUAGES FIT IN
VIEW • VISUAL UNIX • BRIDGING LOOPS AND PROLOG
RATORS • ICON-BASED PROLOG • HASHING SCHEME
EXERCISE • MILLS ON KEEPING IT SIMPLE
CIETY

None of this, of course, was likely to derail a language DoD was pushing. Just as DoD backing had compelled commitment to COBOL on the part of manufacturers courting the Pentagon, so, too, did firm DoD commitment to Ada serve to propel the language forward. The ANSI Ada standard was issued in early 1983. That same year, the Under Secretary of Defense for Research and Engineering issued a DoD directive concerning programming language policy that reiterated the department's commitment to Ada: "The Ada programming language shall become the single, common, computer programming language for Defense mission-critical applications."[173] The directive specified 1984 milestones toward adoption.

Unhappiness over Ada was matched by irritation over the continued popularity of Fortran (and to a lesser extent COBOL). In a 1972 retrospective, Saul Rosen suggested, "The most striking fact about programming languages . . . has been the continued overwhelming acceptance of Fortran and COBOL."[174] Indeed, both languages were expanding to provide increased functionality, thus tightening their hold on users. A 1974 *Communications* article offered techniques addressing the absence of facilities in Fortran for handling character strings.[175] The previous year, programming guru Dan McCracken had admitted that although "*nobody* would claim that Fortran is *ideal* for *anything,* from teachability, to understandability of finished programs," nevertheless "Fortran is very thoroughly entrenched, and . . . not likely to be displaced in a big way any time soon."[176] More than a decade later, McCracken could still assert, "Fortran is still the language of choice for engineering and scientific calculations. (Those who deplore this fact should at least admit that it is a fact.)"[177] Perhaps the most eloquent expression of resignation was heard at a 1975 National Computer Conference session at which Ben Wegbreit of Xerox observed with a distinct lack of enthusiasm, "Ah, . . . Fortran will be around until the end of time...."[178]

The infatuation with structured programming heightened the discontent, as proposals aimed at permitting Fortran devotees to enjoy the fruits of SP began to circulate midway through the 1970s. Calls for "structured Fortran" were not greeted with waves of enthusiasm. Much of the debate was played out, appropriately enough, in the pages of *SIGPLAN Notices*. One reader harkened back to the old days, declaring that Fortran "should have died in the early sixties with the appearance of Algol 60. I am thus appalled by the time and effort invested by so many people in keeping it alive."[179] Another reader suggested that attempts to use Fortran for structured programming were "like trying to make a tack hammer suitable for driving railroad spikes."[180] Stuart Rowland of the State University of New York contended, "there is really only one problem with structured Fortran—it is still Fortran."[181] Fortran was not without defenders, though. One asserted, "Fortran has *not* outlived itself. Fortran is still quite tolerable for a broad spectrum of problems and the transferability makes its use of continued economic importance in our industry."[182] While constituting a less than ringing endorsement, such comments illustrate well the nature of the attachment to Fortran. Fortran remained entrenched less because it was powerful and elegant than because it remained a practical means of accomplishing a wide variety of work and represented a substantial investment in software and training. In much the same way, the QWERTY keyboard continues to resist replacement by the ergonomically superior Dvorak keyboard. Anthony Ralston and Jerrold Wagner recognized this in their 1976 *Transactions* article, calling for the extension of Fortran IV into Structured Fortran (SF). They argued, "attempts to 'kill' Fortran, however well intentioned and, even however desirable such a result might be, are doomed to failure. Revolution in higher level languages is no longer possible; evolution is the only—and necessary—alternative."[183] Similar arguments were taking place over COBOL. Once again, McCracken attempted to put things in perspective. He made his plea for pragmatism with respect to COBOL in a 1976 *Datamation* essay:

> COBOL is the most widely used language in the world by a very wide margin, and it will stay that way for at least a decade. So let's work to make the best of it, and to improve it gradually but steadily. . . . I say, let's get on with it and not sit around moaning about the horrible state of programming languages while waiting for some utopian solution that never seems to get any closer.[184]

What should not be overlooked in the recognition of language inertia, however, is how Fortran and COBOL attained such inertia in the first place—by occupying the pivot between generality and specificity.

The tension between generality and specificity surfaced repeatedly over the years. The arguments over a universal language were simply cases of the larger issue. In an unusually philosophical 1975 paper, Naur compared and contrasted programming languages with mathematics and natural languages. He argued that the lack of precision in natural languages, far from being a defect, in fact made it possible for natural language to continually develop and to express an endless succession of new ideas. The development of natural language, he felt, could be used as a guide for the design of programming languages. Programming languages "should preferably be built from a few, very general, very abstract concepts, that can be applied in many combinations, thereby yielding the desired flexibility of expression."[185] A 1977 *Communications* article seemed to express the opposite point of view:

> Members of this new generation of languages still strive to be general purpose, trying to be applicable to a wide variety of problem domains; and it is here that they may encounter some inherent limitation. For in attempting to span a wide range of potential users with the facilities of a single language, a language designer will either end up with an enormously complex language or one which is only moderately well adapted to any one of the application areas.[186]

Mark Crispin of MIT had made more or less the same point in *Datamation* the previous year. "APL [a highly mathematical language developed around 1960] is a nice language when used as a programmable calculator. Similarly, COBOL is best for large business data base crunching. Neither is very good for the other's type of usage. Let us recognize this rather than try to have the seminationalistic banner of absolute superiority of one over the other!"[187] Writing in a similar vein with respect to ALGOL and Fortran, A.C. Larman had struck the same chord in a more colorful way in the *Computer Bulletin* (published by the BCS) in 1971, "One cannot state, unequivocally, that . . . a racehorse is 'superior to' a dray-horse or a show-jumper; it depends entirely on the purpose for which one requires it...."[188] All these statements seem to recognize implicitly the existence of a pivot along the generality–specificity axis. A 1976 overview of computer technology suggested, however, that the pivot had shifted in theory if not in fact. Ware Myers argued "for people whose primary emphasis is on their own work, the so-called higher-level languages are still orders of magnitude too primitive. The gap between this kind of user and the present languages is staggering. Languages need to become more application-oriented."[189]

Ask and thou shalt receive. As computing entered the 1980s, applications development remained a major headache. A 1981 *Datamation* report observed that applications development "remains one of the dp industry's thorniest problems. Since the '50s, when higher level languages emerged, there's been only slow, piecemeal progress."[190] The following year, though, the magazine heralded a new approach that was easing applications backlogs, "The key to this new trend is the appearance of simpler step-by-step program development languages that are making it possible for users without detailed programming expertise to develop their own applications."[191] Known as nonprocedural or fourth-generation languages (4GLs), these systems, which were sophisticated and powerful descendants of packages such as IBM's Report Program Generator, supposedly permitted a user to specify *what* he or she wanted done without detailing *how* to go about doing it. Often used in conjunction with a database, systems such as RAMIS and Nomad made it easier to develop custom applications that manipulated and distilled the information in the database (e.g., sales figures). One could, for example, order the system to produce a chart or table without specifying exactly what a chart or table looked like or how to go about assembling one. Moreover, one could do this in a language whose syntax bore at least a passing resemblance to normal English. Fourth-generation languages had the potential to remove the programming middleman.

"4GLs are as major a technological advance to computer programming as integrated circuits were to computer hardware and orbiting satellites to data communications."

Predictably, some practitioners gushed with enthusiasm while others were less enraptured. Perhaps the ultimate kudos were bestowed by Nigel Read and Douglas Harmon in a 1983 *Datamation* essay in which they proclaimed, "4GLs are as major a technological advance to computer programming as integrated circuits were to computer hardware and orbiting satellites to data communications."[192] James Martin, czar of the consultants, was also a devout proponent of 4GLs. Others, however, were more reserved in their attitudes. John Cardullo and Herb Jacobsohn, for example, felt that Read and Harmon had overstated their case, "We resist the implication that the use of 4GLs will solve all the problems that are raised by Read and Harmon. They are merely one more very valuable means to help address, define, and solve the myriad problems that face managers."[193] Similarly, Bill Inmon of Coopers & Lybrand contended that while 4GLs were "certainly appropriate for decision support, prototyping, and environments where there is a limited amount of data and/or processing," there was evidence that "for operational systems, fourth generation languages and application development without programmers don't deliver the productivity gains their advocates claim."[194] Michael Brown of Hewlett-Packard disputed Inmon's contention, though in fairly moderate terms, "The use of fourth generation languages does allow an increase in the number of individuals with an applications bias to successfully develop programs. While the organization still needs a balance of computer science types, some production gains are accomplished by getting people with real application experience and competence closer to the development process."[195] Consultant F.J. Grant came at it from the opposite direction, but seemed to end up in more or less the same place. Grant declared that 4GLs were not "a solution to the intellectual and infrastructural problems of traditional MIS [management information systems] implementations," but acknowledged that they "must be taken seriously."[196]

Most people agreed that 4GLs were, in fact, effective in certain situations. The key question had to do with which situations. Fourth-generation languages derived their power from incorporated knowledge of the application domain. As Wegner noted in a 1984 article in *IEEE Software*, the "choice of a domain of discourse for an application generator and the design of a generic program generator and parameter interface require a deep understanding of the problem domain."[197] The previous year in *Datamation*, Alex and Dan Pines had observed that the "problem with the programmerless approach [embodied by application generators] is that it institutes a simple software solution that attempts to achieve two conflicting goals: universal flexibility and extreme ease of use."[198] This point was echoed in a 1988 article in *IEEE Software*, which concluded that "a user can save application-development time if the problem matches the assumptions in the tool's predefined nonprocedural facilities. If the problem is not the kind the tool was designed for, the user may pay development and performance penalties. In these cases, conventional programming is a better alternative."[199] Nonprocedural languages could greatly facilitate the development of certain applications in well-defined problem domains by "nonprogrammers," but they were not a universal answer to the problem of software productivity; they were a palliative rather than a cure.

The fact that 4GLs were nonprocedural did not exempt them from the tension between generality and specificity. Instead, they were an excellent example of the trade-off between breadth and depth. Fourth-generation languages provided relatively high-powered (in terms of productivity) development capability, i.e., leverage in depth, within a limited range of applications. In contrast, languages such as Fortran and COBOL provided less conceptual power within a much broader range, while languages such as PL/I provided little application-specific capability but virtually "universal" range, i.e., leverage in breadth.

Attempts at programming language synthesis were highlighted in a 1986 issue of *IEEE Software*. Noting the difficulties engendered by trying to use the wrong tool for a particular purpose, the guest editor described a new class of programming languages aimed at solving the problem. These languages "do not restrict the programmer to only one *paradigm* . . . rather they are *multiparadigm* systems incorporating two or more of the conventional program paradigms."[200] As Pamela Zave of Bell Laboratories observed in a 1989 article describing one approach to multiparadigm programming, "By definition, a paradigm offers a single-minded, cohesive view—this is, in fact, how the popular paradigms help us think clearly, offer substantial analytic capabilities, and achieve their reputations for elegance. The corresponding disadvantage is that each paradigm is too narrowly focused to describe all aspects of a large, complex system."[201] Such "paradigms" included data flow, functional, imperative (embodied in popular procedural languages such as Fortran and Pascal), and object-oriented programming. Multiple paradigms, though, were not cost-free. As the paradigms multiplied, so, too, did the complexity of the language. The real question, then, was just how many different paradigms one could lump together within one language before the complexity of the language vitiated the gains derived from the availability of more than one paradigm. While there has been some success in augmenting existing languages with a new type of language construct representing a different paradigm, such as the addition of object-oriented constructs to C (C++), it is unclear just how many such balls a programmer can successfully juggle. If complexity has been an issue for the large "universal" languages, it cannot help but be an issue for truly multiparadigm languages. Moreover, such paradigms differentiate languages in a manner not necessarily congruent with differences in application domain. Fortran and COBOL, for example, are aimed at different application domains (science and engineering in the case of the former, commercial data processing in the case of the latter) but both are imperative languages. Therefore, language paradigms, which represent styles of thought, are not necessarily the same as orientation toward a particular application domain. Thus, multiparadigm languages may represent synthesis on one level but not another.

Others took a dim view of the language skirmishes altogether. In his 1977 Turing Award Lecture, John Backus, the originator of Fortran, complained, "discussions about programming languages often resemble medieval debates about the number of angels that can dance on the head of a pin instead of exciting contests between fundamentally differing concepts."[202] Backus considered von Neumann architecture (sequential computing) an "intellectual bottleneck" restricting thinking about programming languages. In a 1979 *Computer* essay, R.N. Caffin suggested an even higher level of irrelevance for language debates, "The solution for more general work does not lie in fool-proof, very high level, pseudo-English languages. We must accept, for the present at least, that programming requires thought."[203] Commenting on Caffin's essay, Jim Haynes of the University of California at Santa Cruz suggested the problem lay in the fact that "inventing new languages and arguing their relative merits is easier and more fun than solving real problems."[204] Similarly, David Feign asserted that the "much harder problem of understanding how people really think and express themselves, and translating this into a machine language, has been dropped by computer scientists. Solving the harder problem would mean more work...."[205] William Wulf of Carnegie Mellon University summarized the situation in a 1980 article on programming language trends, "Choosing the proper balance between the generality of individual features and the cost of their interaction is what has often turned out to be more difficult than expected, and what has often been done badly."[206] But Wulf also recognized that programming languages could not cure the basic problem, "The fundamental problem of constructing reliable, maintainable software is that of reducing its complexity to a level with which humans can cope. . . . Programming is intellectually *tough*. A programming language can, at most, alleviate the difficulty of the task."[206,pp.21-22]

Nevertheless, programming languages continued to be a prime source of contention. While disputes such as that concerning testing versus formal verification were at least theoretically amenable to resolution via pragmatic synthesis, language scraps tended to be more a matter of trade-offs. The tension between generality and specificity could often not be resolved, but merely accommodated by the development and use of languages residing near the pivot point. While some practitioners used the behemoth universal languages and others employed application-specific 4GLs or nonprocedural languages such as LISP, vast numbers found practical accomplishment (if not spiritual fulfillment) in the class of languages that included Fortran, COBOL, and Pascal. For the most part, the trade-off between breadth and depth in high-level languages has been inescapable. Pragmatism for many practitioners has been a matter of splitting the difference with languages at the pivot. To the extent that language paradigms are distinct from application orientation, multiparadigm programming, were it to prove viable, would not necessarily alleviate this tension. The same holds true for software development methodologies.

Mixing and Matching: Redefining the Life Cycle

Pragmatism concerning the limitations of human intellectual capacity and the utility of any particular approach had also infiltrated thinking about the traditional software life cycle. Not only was software itself growing increasingly complex, so, too, was the effort required to produce it. Attempts to rationalize the process had to recognize that software development was a complex and multifaceted activity intimately related to human social and cognitive processes. The traditional, essentially sequential (allowance was generally made for some degree of feedback between stages) model of the development process, while mitigating problems of complexity, embodied little appreciation of the intellectual difficulties inherent in system specification and design. The obvious alternative, an iterative or cyclical process, addressed the cognitive problems but was less effective in lending order and coherence to development activity. Once again, combination and accommodation appeared more profitable than any singular approach.

By the mid-1970s, doubts began to surface in some quarters as to the realism of a principally sequential model of the development cycle. Researchers at the University of Maryland suggested that this ideal was often difficult to achieve. In *Transactions* in 1975, Basili and Albert Turner observed, "building a system using a well-modularized, top-down approach requires that the problem and its solution be well understood. . . . Furthermore, design flaws often do not show up until the implementation is well under way so that correcting the problems can require major effort."[207] Instead, Basili and Turner suggested implementing a simplified version of the system and iteratively enhancing it until the full system was implemented, "'Iterative enhancement' represents a practical means of applying stepwise refinement."[207,p.391] What this amounted to was a kind of prototyping, a development strategy that would attract great attention down the road. A more explicit call for prototyping appeared in a 1980 essay in *Computer* by W.P. Dodd. The notion of prototype programs had been discussed at the previous year's International Conference on Software Engineering but had been more or less dismissed on the basis of cost. Dodd, however, suggested that the vast resources expended on program maintenance reflected the fact that software developers were producing prototypes but refusing to admit it, "In any case, why should we complacently assume we don't need prototypes when more established branches of engineering . . . wouldn't dream of not producing a prototype?"[208] Such sentiments signaled growing recognition that the basically sequential nature of the classic "waterfall" life cycle imperfectly modeled a reality in which foreknowledge in system specification and design was usually incomplete at best and sheer guesswork at worst. (Patrick Hall et al. have written on the difficulty of making the waterfall model work and on the social functions it performs that help keep it in place.[209])

This helped explain why so many highly planned projects seemed to fall on their faces. Indeed, Fletcher Buckley of RCA suggested in 1982 that software plans were often ineffective because the idealized software life cycle was just that—an unattainable ideal.[210] McCracken and Jackson went even further, asserting that the "life cycle concept is simply unsuited to the needs of the 1980s in developing systems."[211] Honeywell's G.R. Gladden was also of the opinion that "the concept of a 'software life-cycle' is no longer helpful, indeed may be harmful to our software development profession."[212] On the other hand, Patrick Hall argued that life cycles, in general, were a good thing. Rather, "it is pedantic defenders of particular life-cycles that are bad. Just as pedantic defenders of particular development methods, or anything else of that matter, are bad."[213] Bruce Blum of Johns Hopkins articulated a similar view.[214] Either way, some serious questions were being raised concerning the applicability of a predominantly sequential development cycle.

Such sentiments signaled growing recognition that the basically sequential nature of the classic "waterfall" life cycle imperfectly modeled a reality in which foreknowledge in system specification and design was usually incomplete at best and sheer guesswork at worst.

The obvious alternative to a sequential process was a cyclical one. In a 1983 letter to *Communications*, Joseph Chambers echoed Basili and Turner from nearly a decade before, "Development of any software system is essentially an iterative process."[215] That the waterfall model was unrealistic was explicitly acknowledged in sessions at the 1984 International Conference on Software Engineering, while a report on the 1985 International Workshop on the Software Process and Software Environments observed signs of "some emerging consensus that process models have some inherently cyclic nature."[216] As IBM's Stefano Nocentini argued, "in complex environments, problems are solved through successive approximations rather than through precise, invariant definitions."[217] Prototyping, clearly accommodating such a view, had been steadily picking up interest; a 1982 Software Engineering Symposium sponsored by ACM SIGSOFT, the IEEE Computer Society's Technical Committees on Software Engineering and VLSI (Very Large Scale Integration), and the National Bureau of Standards had focused on rapid prototyping. Accompanying much discussion of the technique, though, were words of caution. In a 1983 *Communications* article, R.E.A. Mason and T.T. Carey noted that there were also "disadvantages to the use of prototypes, such as higher initial cost for the requirements phase of the development cycle and the possible loss of distinction between this phase and the design phase. But for certain types of applications, there is a growing consensus that prototypes form an effective component of an application development methodology."[218] Jerry

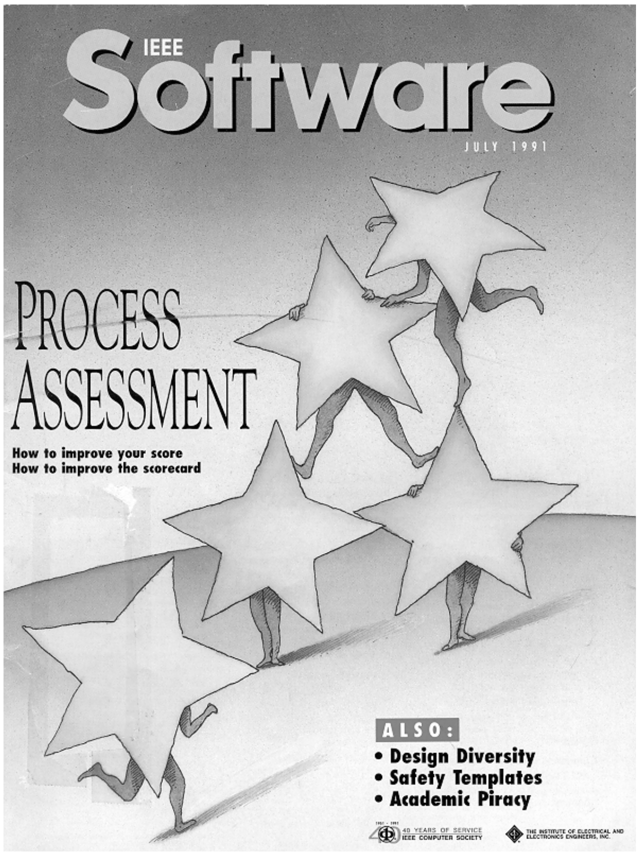
Schulz of Northwestern National Insurance also warned that prototyping was not a magical elixir, "Although prototypes will be a great help in developing decision support systems, much of data processing work consists of systems whose primary purpose is not decision support but rather the everyday operations of the business. While prototyping may also be of value here, the pressure will still remain to develop precise project specifications, spelling out such things as each needed calculation."[219] As was so often the case in software, it was just a matter of time before someone seized the pragmatic middle ground with an argument for synthesis. In 1984, Boehm, Terence Gray, and Thomas Seewaldt described in *Transactions* an experiment comparing the prototyping approach to software development with the specification-driven approach. While, by their own admission, the experiment could hardly be considered definitive, they nevertheless found the results to suggest that "both prototyping and specifying have valuable advantages that complement each other. For most large projects, and many small ones, a mix of prototyping and specifying will be preferable to the exclusive use of either by itself."[220] Prototyping seemed to result in smaller programs, reduced effort, and ease of use, while the traditional approach lent more coherence, functionality, and robustness.[221] Therefore, a synthetic approach could use prototyping to compensate for intellectual limitations with respect to problem definition and system specification and to employ the traditional specification approach to ameliorate complexity by increasing coherence and fault tolerance.

In a more radical departure from the conventional development process, prototyping combined with program transformations in what was dubbed the operational approach. The operational approach represented yet another attempt to apply computational leverage to the problem of software development. The concept of program transformations had been batted around for a number of years in various forms. One of the most ambitious was that envisioned by Zohar Manna and Richard Waldinger in a 1979 *Transactions* article in which they considered the principles to be incorporated into an automatic program synthesis system:

> Our basic approach is to transform the specifications repeatedly according to certain *transformation rules.* Guided by a number of strategic controls, these rules attempt to produce an equivalent [program] description composed entirely of constructs from the target language. Many of the transformation rules represent knowledge about the program's subject domain; some explicate the constructs of the specification and target languages; and a few rules represent basic programming principles.[222]

In other words, once the program specification had been developed at some highly conceptual or abstract level, the computer would then be used in a multistage process to transform the specification into the programming language. The transformation processor would automatically bridge the gap between specification and code. This got around one of the key trade-offs in software generally and for formal methods in particular: understandability versus efficiency. The argument, as articulated at a 1979 British conference on the topic, was that "transforming specifications of programs into efficient algorithms . . . [was preferable] rather than having to prove correctness of clever and probably 'unnatural' programs."[223]

A few years later, David Wile of the University of Southern California proposed a somewhat more modest approach in which the implementer would manually choose the transformations to be applied, leaving the computer to carry out the transformations. He admitted, however, that producing a large and useful catalog of transformations was a mountain yet to be climbed.[224] A more pivotal problem, though, was one that also vitiated the usefulness of formal verification—achieving a correct program specification in the first place. The transformations would presumably preserve program correctness, but that assumed that the initial program specification had been correct at the start.


IEEE
Software
JULY 1991
PROCESS ASSESSMENT
How to improve your score
How to improve the scorecard
ALSO:
• Design Diversity
• Safety Templates
• Academic Piracy
40 YEARS OF SERVICE IEEE COMPUTER SOCIETY
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

This was where rapid prototyping came in. According to Zave in a 1984 *Communications* article, in the operational approach, "the specification itself can be used as a prototype, since it is executable. This type of prototype can be produced rapidly and will be produced as an integral part of the ordinary development cycle. . . . In the conventional approach a prototype is produced by iterating the entire development cycle."[225] The approach Zave described was "operational" presumably because the problem-oriented specification was executable (albeit inefficiently) and thus operational. One could therefore experiment with the specification, which in effect was a prototype of the program, until the prototype and thus the specification appeared satisfactory. The implementer would then guide the application of transformations to produce an efficient implementation of the system. All of which was fine *if* you could develop a system that could do it. A 1981 *Transactions* article had noted that "the construction of software by applying only formally verified rules is a timeconsuming and highly sophisticated activity even for an expert programmer."[226] As Zave admitted, one of the major weaknesses of the operational approach was that "transformational implementation is a relatively untried approach, and the necessary theoretical supports are only beginning to be developed. The idea of program transformations has been with us for a long time . . . without noticeable impact."[225,p.117]

Ideally, practitioners required not just an appreciation for the limitations of particular models but a framework for choosing the most appropriate model at the most appropriate time. The spiral model of software development purported to furnish just such a framework. As Boehm described it in *Computer* in 1988, the spiral model was oriented around the notion of risk assessment. Rather than revolving around a particular element such as the executable code or the documentation, a quality that characterized other life cycle models, the spiral model focused on making well-considered choices to employ approaches embodied in particular models at different times in the development process. According to Boehm:

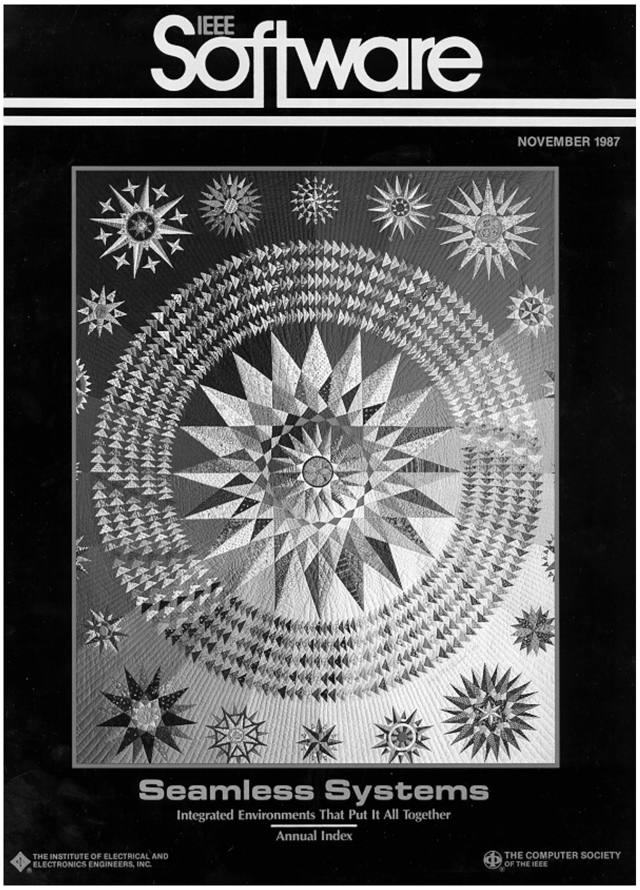
> this risk-driven subsetting of the spiral model steps allows the model to accommodate any appropriate mixture of a specification-oriented, prototype-oriented, simulation-oriented, automatic transformation-oriented, or other approach to software development. In such cases, the appropriate mixed strategy is chosen by considering the relative magnitude of the program risks and the relative effectiveness of the various techniques in resolving the risks.[227]

The model is spiral in the sense that it is a cyclic process in which each cycle expands in terms of cost and commitment, yet involves the same sequence of identification of alternatives, assessment and choice, production, and evaluation. Each iteration brings one closer to the operational system through successively greater elaboration. A particularly appealing aspect of the spiral model was that, under certain circumstances, it could become equivalent to one of the other process models, thus rendering those other models effectively special cases of the spiral model.[227,p.69] Boehm admitted, however, that the spiral model was not without problems, being difficult to reconcile with typical software development contracts and relying on the risk-assessment expertise of the people involved. Moreover, the model required further elaboration before people without substantial experience with it could use it effectively.[227,pp.70-71] Still, the spiral model represented a milestone in that it formally embraced the notion of technical pluralism with respect to life cycle models. It viewed accommodation and synthesis as the normal state of affairs.

Thus, a key insight of this period in terms of software development models was of a kind with those in aforementioned areas. Although the evolution of an alternative view of the development process as cyclical as opposed to sequential was vital, its importance would have been diminished if the community of practitioners had either bifurcated with respect to the two approaches and their variants or simply adopted the newer ones in wholesale fashion. While some practitioners no doubt did definitively opt for one approach over another, many displayed the same essential insight that was evident in other important disputes over software technology—that rewards often flowed from pragmatic accommodation based on appreciation of the limitations of singular solutions. The advent of an articulated framework for achieving such accommodation was an even more sophisticated manifestation of technical pluralism.

Establishing the Milieu: Toward a Development Environment

Assuming the absence of a magic wand in the form of a truly automatic programming system such as the synthesis system Manna and Waldinger proposed, the various activities comprising the development process could still benefit from less ambitious tools. Compilers applied computational leverage to one aspect of software development; computer-based tools could benefit other aspects. Editors, debuggers, and other equally modest yet useful tools significantly assisted in the performance of various development and maintenance activities. Grouping the necessary tools together into a development *environment* would clearly facilitate the development process.


IEEE Software
NOVEMBER 1987
Seamless Systems
Integrated Environments That Put It All Together
Annual Index
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.
THE COMPUTER SOCIETY OF THE IEEE

Here too, one discovers the basic trade-off between breadth and depth. On the one hand, one could form a programming environment that mainly resembled a development tool kit, involving a wide selection of tools minimally coordinated in terms of inter-tool communication. Such an environment would embody little specific orientation in the way of language, methodology, or application area and could be applied to a broad range of development efforts. On the other hand, an environment could be integrated to the extent that it revolved around a particular language, methodology, and application type and thus provide high degrees of functionality directly related to those particulars. When tool sets were not so oriented, clearly some amount of their combined effective leverage was vitiated; their leverage in depth suffered for the sake of leverage in breadth. If, however, one combined tools into a single environment integrated in terms of languages, methodology, and/or application area, one constrained the range of the environment's utility; leverage in breadth was sacrificed for leverage in depth. This inescapable trade-off produced programming environments of all stripes. The popularity of the Unix system, however, suggests that tool kit environments may reside at a pivot similar to that of high-level languages. On one side of the pivot lies broad, completely ad hoc collections of tools with virtually no coordination among them. On the other side lies the realm of relatively narrowly oriented, tightly integrated environments. As in other areas of software technology, practitioners often opted for the middle ground.

In a development on a par (at least in hindsight) with the introduction of Fortran, a 1974 *Communications* article introduced the Unix time-sharing system. What Bell Labs colleagues Dennis Ritchie and Ken Thompson had wrought was much more than an operating system. Unix constituted a programming *environment.* Programs available under Unix included an assembler, editor, symbolic debugger, text formatter, macro processor, C (a new language that would quickly become identified with Unix), and Fortran compilers, as well as a collection of maintenance and utility programs.[228] The system also facilitated the funneling of one tool's output directly into another tool. In a 1977 article, Bell Labs' Evan Ivie took things even further. He suggested that the programming community "develop a program development 'facility' (or facilities) much like those that have been developed for other professions (e.g., carpenter's workbench, dentist's office, engineer's laboratory). Such an approach would help focus attention on the need for adequate tools and procedures; it would serve as a mechanism for integrating tools into a coordinated set...."[229] Furthermore, the workbench concept encompassed the entire software life cycle. Part of Ivie's motivation stemmed from his perception (widely shared) that the programming community had yet to produce "a software development methodology that is sufficiently general so that it can be transferred from one project to another and from one machine to another."[229,p.753] Colleagues Kernighan and Plauger had made a similar argument the previous year, suggesting, "few programmers think to use or build *programs* as tools. If they maintain a set of utilities at all, such programs tend to be high personalized and must be modified for each new application."[230] The authors urged the development and use of general-purpose tools.

Unix represented only one of a number of approaches falling under the rubric of programming environments. Unix was (and is) the quintessential toolkit type of environment. Unix supported neither a particular development methodology nor a specific language, although the C language is closely associated with it. (Ritchie developed C at Bell Labs in 1972 as a tool for creating Unix, evolving out of the B language Thompson developed.) Rather, as Anthony Wasserman put it in 1981 in his introduction to a set of *Computer* articles on development environments, "the facilities of Unix may be thought of as a tool kit from which the developer can select tools that are appropriate for a particular task and for which a toolsmith can easily build additional tools."[231]

This kind of approach was not without its drawbacks. In a 1981 *Datamation* sidebar, Michael Lesk of Bell Labs noted:

> Unix has grown more than it has been built, with many people from many places tossing software into the system. . . . Much of the attractiveness of Unix derives from its hospitality to new commands and features. This has also meant a diversity of names and styles. To some of us, this diversity is attractive, while to others the diversity is frustrating, but to hope for the hospitality without the diversity is unrealistic.[232]

A *Datamation* article three years later emphasized the negative aspects, complaining, "all the improvements to Unix simply seem to add to the confusion—there are now a bewildering number of Unix versions from AT&T and other vendors, each with its own special features."[233] Dennis Barlow and Norman Zimbel of Arthur D. Little concurred, "It is clear that Unix is not a single operating system, but rather a generic identifier for a clan of operating systems sprung from a common root."[234] A sidebar indicated, however, that AT&T viewed Unix as the solution rather than the problem:

> A standard operating system that could be used on many different vendors' hardware would be an important boost to interchangeability. AT&T believes the Unix operating system to be a strong candidate for such a standard for several reasons, including portability, flexibility for diverse projects, versatility from micros to mainframes, and the existence of a large group of experienced users to feed the growing marketplace.[235]

Indeed, in 1981 ACM President Denning had used Unix as an example of what was needed to promote software reuse, "In short, to foster a new attitude—that programs are potentially public, sharable, and transportable—we need operating systems that are hospitable toward saving and reusing program parts. I will cite Bell Labs' Unix operating system to illustrate that the technology is at hand."[236] A *Datamation* reader suggested in 1984 that the bottom line with regard to Unix was the same as in the cases of COBOL and Fortran, "Unix users like Unix for the same reasons that PC/DOS users like their operating system—it works. We do not claim any mystical properties over other operating systems...."[237] (The author admitted, though, that as was the norm for any significant development in software, Unix had its share of self-righteous zealots.) Unix might not have been the end-all and be-all of programming environments, but it enabled users to get useful work done. In other words, Unix may well represent the pragmatic pivot region in terms of programming environments.

Other environments consciously focused on a specific language. The Cornell Program Synthesizer was a modest step along these lines. As described in 1979 in *SIGPLAN Notices*, the Cornell project was a self-contained programming environment tailored to the grammar of the host programming language, providing, among other things, automatic language-specific syntactic checks.[238] One of the more popular language-oriented approaches was the Interlisp environment, which provided tools specially designed to facilitate the development of LISP programs.[231] More ambitious, in keeping with its language, was the Ada Program Support Environment. A 1981 *Computer* article noted, "potential benefits of the language and environment can only be fully realized if the two are properly integrated."[239] The emphasis, as one would expect given the hopes and rationale for Ada, was on portability, "Tool portability, project portability, retargetability, rehostability, and programmer portability are all important."[239,p.28] Like Unix though, the Ada environment would be open-ended, permitting modification and extension at any time. Just as Unix tools were written in C, Ada environment tools would be written in Ada to ensure portability and coordination.

Yet another tack one could take in designing a development environment was to focus on the domain of application. The LISP Programmer's Apprentice under development in the late 1970s at MIT, although dedicated to LISP, focused on providing assistance in particular application domains. The apprentice would cooperate with the user in the design, implementation, and maintenance of programs by performing various checks on the program design and code.[240] Reflecting pragmatic recognition of the facts of life in software technology, the developers saw this as a "realistic interim solution to the current software problems and as an evolutionary path towards the more ambitious goals of automatic programming."[240] As the potential of expert systems began to seize imaginations in the 1980s, the notion of knowledge-based programming assistants became increasingly attractive. In a paper presented at a 1984 SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Elliot Soloway of Yale University concluded that, based on programming experiments,

> the software aids that we see relevant to enhancing the design process are those that can *digest* the information provided by the designer. In particular, one aspect in which designers seemed to need assistance was in keeping track of the "notes" they made (the assumptions, expectations, and constraints) *and* recalling them at just the appropriate time. Software that could perform this type of assistance would require considerable understanding of the design process itself, and information that is problem specific.[241]

Such bookkeeping assistance seems to fall somewhere between toolkits and automatic program synthesis in terms of ambition and usefulness. MCC's Software Technology Program seemed to be aiming for this type of environment in the mid-1980s, one that would "aid all aspects of complex software development, including requirements capture, exploration, and early design."[242] Indeed, this sort of approach has come to be known as "exploratory" software development. At its core, exploratory software development was a means of *accommodating* design uncertainty whereas more traditional methods aimed to combat it.[243] Winston Royce of TRW recently asserted, "the exploratory approach is instinctively correct to programmers, who use the act of coding to examine a problem and code execution to test a requirements hypothesis."[244]

At its core, exploratory software development was a means of *accommodating* design uncertainty whereas more traditional methods aimed to combat it.

However, embedding any form of localized knowledge (including characteristics of the work culture) within a development environment constituted a substantial problem in and of itself. Reconfiguring the environment for every project could potentially involve a large amount of effort. This was the rationale behind the Gandalf project described in *Transactions* in 1986. The authors noted, "hand-crafting a software development environment for each project is economically infeasible. Gandalf solves this problem by generating sets of related environments." More specifically, "Gandalf promotes the creation of *project-oriented* software development environments in which many traits, such as protection policies, are tuned to groups of persons working on a project rather than to the entire computing community or to particular individuals."[245] Two years later, also writing in *Transactions*, Jayashree Ramanathan and Soumitra Sarkar discussed a similar idea featuring a project-specific assistant that was produced through interpretation of a conceptual modeling language used to specify process, data, tool, and user models specific to a particular project.[246] Sophisticated development environments such as these were early examples of what has been labeled "metaCASE." Whereas computer-aided software engineering (CASE) involved the use of programming tools aimed at supporting a particular method or approach, metaCASE aimed at supporting a variety of approaches in a variety of settings. As the guest editors of a special issue of *Communications* noted in 1992:

> it is becoming apparent that a single design method will not adequately address all application domains. . . . Also, differences in skill levels, styles, attitudes, cultures, goals, and constraints demand highly tailorable CASE tools. The goal is a technology that can accommodate many methods, notations, styles, and levels of sophistication....[247]

CASE tools that could be tailored to accommodate the specific circumstances of a given project effectively embodied and endorsed the notion of technical pluralism.

Along with this came slow acceptance that salvation was not to be found in analogies with other design processes, most obviously those related to computer hardware. A 1974 *Computer* article pointed out that this insight had produced more frustration than anything else, "Unfortunately, while the theory underlying the application of computers in the design of computing hardware has developed thoroughly, keeping pace (or nearly so) with the developing technology, the implementation of this theory remains a difficult, mostly manual exercise in the design of programs and programming systems."[248] On the other hand, Peter Freeman noted later that year that

> when talking about the automation of software design, we are in fact talking about the automation of software *creation*—that is, the design, production, testing, and redesign of software using the traditional meanings of those terms. Because of the totally symbolic nature of software, it may turn out that techniques applicable to the design of other objects will apply to the entire creation process of software.[249]

According to Freeman, the areas requiring work included problem representation, solution representation, and problem solving. The fundamental nature of these areas reflects the level of cognitive activities involved in software development. Their fuzziness suggests the difficulty of reducing software problems to clearly delineated critical technical problems subject to systematic attack. Freeman himself cautioned, "unless software design problems can be formulated in detail exactly like some other class of design problems, the use of techniques from other areas may require a good deal of work."[249] A decade later, at a 1985 MCC interdisciplinary symposium on complex design, design expert J. Christopher Jones suggested that software design was indeed unique. According to a report in *IEEE Software*, Jones contended, "the complexities of software give rise to a new situation beyond the scope of the previous engineering efforts, obliging students of the subject to 'go outside the rules.'"[243,p.70]

Thus, while programming environments applied additional computational leverage to the problem of software production and supported the imposition of structure and coherence onto the development process, they still suffered limitations arising from software's ephemeral nature. The fundamental, complex, and fuzzy processes involved in software development rendered comparisons with other technologies of dubious value. No single approach would suffice, leading many practitioners to split the difference at the pivot. Environments that could be relatively easily customized to fit the problem at hand represented an even more pragmatic response to the diversity of project characteristics. Such pragmatism may not have been overly satisfying, but practitioners could at least accomplish more than they could before.

Picking and Choosing: The Essence of Engineering

The 1980s witnessed a growing realization that effective software development is contingent on a whole range of factors and influences. Recognition of the necessity and reality of technical pluralism, though, also led to an inescapable question. If there was seldom (if ever) a manifestly single best approach, how did one go about choosing an approach from the array of available options? This question had three facets to it. First, how did one go about characterizing different approaches so as to facilitate reasoning about them? Second, and similarly, how did one go about characterizing the attributes of the situation at hand in terms of problem or task, organization, culture, etc.? Finally, how did the qualities of the former relate to those of the latter? Any type of selection entailed determining in some fashion the most appropriate match between the characteristics of various approaches and the characteristics of the problem and its associated circumstances.

Jones contended, "The complexities of software give rise to a new situation beyond the scope of the previous engineering efforts, obliging students of the subject to 'go outside the rules.'"

This necessitated turning away from the search for a philosopher's stone, from the hope of universalism. As Brooks argued in 1987, "building software will always be hard. There is inherently no silver bullet."[250] Paul Rook of GEC Software had observed in the first issue of the *Software Engineering Journal* the previous year:

> differences in organization structures, applications and existing approaches make it impractical to prescribe a single scheme that can be universally followed. Methods, tools, management practices or any other element of the total development environment cannot be chosen without considering each element in its relationship to the other parts of the development system.[251]

A similar, albeit somewhat less nuanced conclusion had been reached earlier at a London Comparative Review of Information Systems Design Methodologies conference, one of a series of such conferences. In his summary of the conference in the *Computer Bulletin*, Anthony Finkelstein reported that practitioners "were shown that the search for a best methodology is futile and that they should be able to draw from an armoury of approaches which they can integrate."[252] Left unanswered, though, as such conclusions often did, was how to go about practically differentiating and selecting approaches.

A number of articles in the 1980s and 1990s attempted to provide frameworks and procedures for making such choices. In 1982, for example, A.T. Wood-Harper and G. Fitzgerald identified six major approaches to systems analysis—general systems theory, human activity systems, participative (sociotechnical), traditional, data analysis, and structured systems (functional)—and attempted to classify them according to paradigm, conceptual model, and objectives.[253] A finer granularity characterized an article appearing three years later that compared the features of seven specific techniques or methods on the basis of analysis and design features but also with respect to philosophy, assumptions, and objectives. The seven examined techniques ranged over five countries and 12 years and differed in significant ways.[254] Even seemingly unitary approaches such as prototyping could be and were broken down into several subtypes.[255] In this realm as well, formalism raised its head in the form of a 1992 *IEEE Software* article in which two researchers at the University of California at Irvine sought to scientifically compare software design methodologies such as the Jackson method, structured design, and object-oriented design according to Grady Booch. Their approach was, first, to distill a set of key features (base framework) and, then, to describe those features as manifested by the methodologies in terms of a meta-language or modeling formalism. They felt this would provide a basis for "objective" comparisons.[256]

All of these efforts, though, were static in the sense that they offered only a structured description of a number of particular approaches or techniques, giving little guidance as to how to go about doing the selecting. It was in this spirit that D.M. Episkopou and Wood-Harper proposed a framework, that is, a process model, for matching a particular method to a particular environment, "[N]o one approach can be classed as 'superior' to the others—rather the art is in applying a suitable approach contingent on variables within and around the problem situation."[257] Their system involved identifying and describing roles in the problem-solving process and their environments and then matching these with a particular methodology. A 1988 *Transactions* article placed the idea of project-based selection in the context of life cycle models, arguing that project managers needed to choose an appropriate life cycle model for each project based on such factors as requirements volatility, the shape of that volatility, and the longevity of the application.[258] Pushing the selection issue even further, some technologists argued that even this was too simplistic a view. For example, Bo Sanden of George Mason University disputed that

> design problems can be grouped according to method, and that each method addresses a particular type of problem better than any other method. While this may be true for some well-understood problem categories, generally, the fact of the matter is that one method will seldom cover all the essential aspects of any real-world problem. Rather, it is important to have at one's disposal a number of design principles (from different methods) and apply those which result in important statements about the problem at hand.[259]

Sanden proceeded to show how the problem Booch used to illustrate object-oriented design in 1986 could be better handled using the Jackson approach in conjunction with Booch's object-oriented one. Apparently sympathetic to this sort of eclectism was Nicholas Zvegintzov, the editor of *Software Maintenance News*, who declared at the 1989 International Conference on Software Engineering a few months later, "we may as well abandon the dream of getting the whole under control. Various methods will work for localized problems. You will always be working on parts of the system."[25,p.109] Interestingly, if one elevates this attitude to the level of the life cycle model, one ends up with something resembling Boehm's spiral approach.

If software engineering is to become an actuality rather than a wish, it will require more than simple acknowledgment of the necessity of choice. It will also need a *basis for choice*. If, in fact, the dominant trend in software technology since 1970 has been a slowly increasing willingness to embrace the notion of technical pluralism (perhaps owing to a combination of project failures and competitive pressures), it is difficult to escape the implication that the key trend of the 1990s must be development of a thoughtful basis for choice. That basis, moreover, must consider the myriad factors that characterize any particular software solution. A *thoughtful* basis, though, should not be taken to mean an exclusive or overwhelming reliance on science and mathematics. For while these will undoubtedly play important roles in software engineering, as they have in other engineering fields, they are no substitute for experience and aesthetics, intuition and heuristics. Accepting the necessity of choice, developing a basis for choice, and carrying out that choice in a nondogmatic manner demonstrate the height of pragmatism. As such, pragmatism is the essence of engineering.

Conclusion

The closing panel at the 1978 International Conference on Software Engineering had concluded rather dolefully, "the problems of the '80s look very much like the problems of the '70s and depressingly similar to the problems of the '60s."[260] At the 1985 conference, Geoffrey Pattie, Britain's minister of state for industry and information technology, seemed to confirm it, "To put it very bluntly . . . too much delivered software is still unsatisfactory. It is still too often delivered late, costs more than expected, sometimes fails to work in the way required, and quite often consumes excessive resources in what is euphemistically called maintenance."[261] Almost a decade later, an article in *Scientific American* sought to explain "software's chronic crisis."[262] For all the achievements of the previous quarter century, the software problem, as Denning had labeled it, had not gone away. In 1992, one practitioner observed that more than half of the projects of which she was aware, in diverse application areas, were late, over budget, unreliable, and difficult to maintain, "The persistency of the [software] crisis is discouraging."[263]

It often seemed, in fact, that virtually nothing in the realm of software qualified as straightforward.

To some extent this can be attributed to the steadily increasing ambitions of software developers and users. Clearly, significant progress has been made; systems that would have defied the imagination not long ago can now be attempted with the expectation of at least some modicum of success. Nevertheless, the basic problems remain. Doing software was difficult in the 1960s, and it is still difficult in the 1990s. Software has become more ordered internally, as has the development process that produces it. But as steadily increasing ambitions have compensated for the mitigating effects of structure on software's complexity, software developers have not found their work any easier.

Consider all the critical areas in which software's malleability, discreteness, and concomitant complexity served to frustrate attempts to hurdle problems rather than wrestle with them. A universal language might have done wonders for communication, transportability, and tool development, except that it was, by definition, too complex and cumbersome for the tastes of many. On the other hand, highly application-specific languages were conceptually powerful, but enhanced productivity only in narrow areas. Exhaustive testing would have greatly increased software reliability, but combinatorial explosion would not permit it. Formal verification would have done the same, but the complexity of the proofs vitiated its usefulness. Furthermore, a proof was only as good as the program specification, and developing specifications was a rather fuzzy process, human foresight being far from perfect. The problem of human cognitive limitations hindered the development of complete and appropriate program specifications, while iterative development tended to reduce overall design coherence. Simple, comprehensive measurements would have provided an objective check on program complexity, but that very complexity limited the validity of straightforward measurements. It often seemed, in fact, that virtually nothing in the realm of software qualified as straightforward.

Complexity may be a fundamental phenomenon and problem solving a fundamental activity, but neither is simple. On the contrary, both are complicated and multifaceted, often defying straightforward understanding or response. The salience of these facts stems directly from both software's ephemeral nature and the potential derived therein for broad computational leverage. Because software is abstract, it can be effectively applied to a wide range of problems. This entails, in turn, basic notions of design and problem solving—hierarchical decomposition, abstraction, and so forth—that, while highly useful, defy translation into exact technical doctrine equally effective under all circumstances. No single approach in any single aspect of software technology could fully satisfy the needs or desires of practitioners. Precise dogma finding its expression in a single programming language, design technique, metric type, or management method is no doubt more emotionally satisfying, but nevertheless impractical. Effective technological practice demands technical pluralism operating in the context of local knowledge and within a framework for choice.

The story of software engineering since the label came into use is thus a story of compromise among generality and specificity, heuristics and formalism, procedures and data, sequence and cycle. The practical response was combination and accommodation—covering all bases or splitting the difference, synthesizing complementary approaches or accommodating inescapable trade-offs. Pragmatists argued for mixed strategies of testing and proving, the use of tailored reliability models and development environments, the use of a full set of metrics, and the synthesis of life-cycle models. But while seizing the middle ground appeared to be a practical way to cope with difficulties, it seemed unlikely to produce a revolution. If software technologists are nowadays devoting more effort to engaging in a pragmatic fashion with the complexity of their problems, it is to their credit. *That* is symptomatic of maturity and of real engineering.

Acknowledgments

This research was supported in part by the Charles Babbage Institute for the History of Information Processing and by the Leverhulme Trust.

References

[1] Andrew L. Friedman with Dominic S. Cornfeld, *Computer Systems Development: History, Organization and Implementation.* New York: John Wiley & Sons, 1989.

[2] David Lorge Parnas, "Software Aspects of Strategic Defense Systems," *Am. Scientist,* vol. 73, no. 5, pp. 432-440; reprinted in *Computerization and Controversy: Value Conflicts and Social Choices,* Charles Dunlop and Rob Kling, eds. New York: Academic Press, 1991, pp. 593-611.

[3] Frederick P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer,* vol. 20, p. 12, Apr. 1987.

[4] Eloina Pelaez, "A Gift from Pandora's Box: The Software Crisis," PhD diss., Univ. of Edinburgh, 1988.

[5] Peter Naur and Brian Randell, eds., *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee,* Garmisch, Germany, Oct. 7–11, 1968. Brussels: Scientific Affairs Division, North Atlantic Treaty Organization (NATO), 1969, p. 13.

[6] B. Randell, "Towards a Methodology of Computing System Design," Peter Naur and Brian Randell, eds., *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee,* Garmisch, Germany, Oct. 7–11, 1968. Brussels: Scientific Affairs Division, North Atlantic Treaty Organization (NATO), 1969, p. 205.

[7] Stanley Gill, "Thoughts on the Sequence of Writing Software," Peter Naur and Brian Randell, eds., *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee,* Garmisch, Germany, Oct. 7–11, 1968. Brussels: Scientific Affairs Division, North Atlantic Treaty Organization (NATO), 1969, p. 186.

[8] J. N. Buxton and B. Randell, eds., *Software Engineering Techniques: A Report on a Conference Sponsored by the NATO Science Committee*, Rome, Italy, Oct. 27–31, 1969. Brussels: Scientific Affairs Division, NATO, 1970, p. 7.

[9] R. M. Needham and J. D. Aron, "Software Engineering and Computer Science," J. N. Buxton and B. Randell, eds., *Software Engineering Techniques: A Report on a Conference Sponsored by the NATO Science Committee*, Rome, Italy, Oct. 27–31, 1969. Brussels: Scientific Affairs Division, NATO, 1970, p. 114.

[10] Niklaus Wirth, "Program Development by Stepwise Refinement," *Comm. ACM,* vol. 14, p. 221, Apr. 1971.

[11] Alan Cohen, "Letter," *Datamation,* vol. 17, p. 15, Feb. 1, 1971.

[12] D. L. Parnas, "A Technique for Software Module Specification with Examples," *Comm. ACM,* vol. 15, p. 330, May 1972.

[13] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Comm. ACM,* vol. 15, p. 1,053, Dec. 1972.

[14] Glenford J. Myers, "Characteristics of Composite Design," *Datamation,* vol. 19, p. 102, Sept. 1973.

[15] Frank DeRemer and Hans Kron, "Programming-in-the-Large Versus Programming-in-the-Small," *SIGPLAN Notices,* vol. 10, p. 114, June 1975.

[16] Barbara H. Liskov and Stephen N. Zilles, "Specification Techniques for Data Abstractions," *IEEE Transactions on Software Engineering,* vol. 1, p. 7, Mar. 1975.

[17] John Guttag, "Abstract Data Types and the Development of Data Structures," *Comm. ACM,* vol. 20, p. 404, June 1977.

[18] Grady Booch, "Object-Oriented Development," *IEEE Transactions on Software Engineering,* vol. 12, p. 212, Feb. 1986.

[19] Patrick H. Loy, "A Comparison of Object-Oriented and Structured Development Methods," *Software Eng. Notes,* vol. 15, p. 46, Jan. 1990.

[20] Brad J. Cox, "Message/Object Programming: An Evolutionary Change in Programming Technology," *IEEE Software,* vol. 1, p. 51, Jan. 1984.

[21] Victor R. Basili et al., "Characterization of an Ada Software Development," *Computer,* vol. 18, p. 64, Sept. 1985.

[22] Paul T. Ward, "How to Integrate Object Orientation with Structured Analysis and Design," *IEEE Software,* vol. 6, pp. 74–82, Mar. 1989.

[23] Russell J. Abbott, "Knowledge Abstraction," *Comm. ACM,* vol. 30, p. 666, Aug. 1987.

[24] Bill Curtis, Herb Krasner, and Neil Iscoe, "A Field Study of the Software Design Process for Large Systems," *Comm. ACM,* vol. 31, p. 1,271, Nov. 1988.

[25] Galen Gruman, "ICSE Assesses the State of Software Engineering," *IEEE Software,* vol. 6, pp. 110–111, July 1989.

[26] M.A. Jackson, *Principles of Program Design.* New York: Academic Press, 1975.

[27] Jean Warnier, *Logical Construction of Programs,* translation by B.M. Flanagan. New York: Van Nostrand Reinhold, 1976.

[28] John Parker, "A Comparison of Design Methodologies," *Software Eng. Notes,* vol. 3, p. 19, Oct. 1978.

[29] James R. Donaldson, "Structured Programming," *Datamation,* vol. 19, p. 53, Dec. 1973.

[30] W. Stevens, G. Myers, and L. Constantine, "Structured Design," *IBM Systems J.,* vol. 13, pp. 115–139, May 1974; reprinted in *Clas-*

sics in Software Engineering, Edward N. Yourdon, ed. New York: Yourdon Press, 1979, pp. 207–232.
[31] Herbert A. Simon, *The Sciences of the Artificial,* 2nd ed. Cambridge, Mass.: MIT Press, 1981.
[32] F. Terry Baker and Harlan D. Mills, “Chief Programmer Teams,” *Datamation,* vol. 19, p. 58, Dec. 1973.
[33] Gerald M. Weinberg, *The Psychology of Computer Programming.* New York: Van Nostrand Reinhold, 1971.
[34] Laton McCartney, “Data for Rent,” *Datamation,* vol. 23, p. 167, May 1977.
[35] Fred Gruenberger, “Letter,” *Datamation,* vol. 20, pp. 27–28, Feb. 1974.
[36] Dick Butterworth, “Letter,” *Datamation,* vol. 20, p. 158, Mar. 1974.
[37] John G. Fletcher, “Letter,” *Datamation,* vol. 20, p. 29, Mar. 1974.
[38] R. R. Brown, “1974 Lake Arrowhead Workshop on Structured Programming,” *Computer,* vol. 7, p. 62, Oct. 1974.
[39] James L. Elshoff, “The Influence of Structured Programming on PL/I Program Profiles,” *IEEE Transactions on Software Engineering,* vol. 3, p. 367, Sept. 1977.
[40] Frank P. Mathur, “Review of *Infotech State of the Art Report: Structured Programming,*” *Computer,* vol. 9, p. 116, Dec. 1976.
[41] Paul Abrahams, “‘Structured Programming’ Considered Harmful,” *SIGPLAN Notices,* vol. 10, p. 13, Apr. 1975.
[42] Daniel M. Berry, “Structured Documentation,” *SIGPLAN Notices,* vol. 10, p. 9, Nov. 1975.
[43] David L. Parnas and Paul C. Clements, “A Rational Design Process: How and Why to Fake It,” *IEEE Transactions on Software Engineering,* vol. 12, pp. 251–252, Feb. 1986.
[44] “Address on Structured Programming Keynotes Compcon Software Sessions,” *Computer,* vol. 8, p. 7, Mar. 1975.
[45] Peter J. Denning, “Comments on Mathematical Overkill,” *SIGPLAN Notices,* vol. 10, p. 11, Sept. 1975.
[46] C. Wrandle Barth, “STRCMACS—an Extensive Set of Macros to Aid in Structured Programming in 360/370 Assembly Language,” *SIGPLAN Notices,* vol. 11, p. 31, Aug. 1976.
[47] David Frost, “Psychology and Program Design,” *Datamation,* vol. 21, p. 138, May 1975.
[48] Lawrence J. Peters and Leonard L. Tripp, “Is Software Design Wicked?” *Datamation,* vol. 22, p. 127, May 1976.
[49] Lawrence J. Peters and Leonard L. Tripp, “Comparing Software Design Methodologies,” *Datamation,* vol. 23, p. 94, Nov. 1977.
[50] Dennis P. Geller, “Letter,” *Software Eng. Notes,* vol. 4, p. 18, Jan. 1979.
[51] Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering.* Reading, Mass.: Addison-Wesley, 1982, p. 177.
[52] Edsger W. Dijkstra, *Selected Writings on Computing: A Personal Perspective.* New York: Springer Verlag, 1982, pp. 126–128.
[53] Harlan D. Mills, “The New Math of Computer Programming,” *Comm. of the ACM,* vol. 18, p. 44, Jan. 1975.
[54] Harlan Mills, “Software Development,” *IEEE Transactions on Software Engineering,* vol. 2, pp. 268–269, Dec. 1976.
[55] Edsger W. Dijkstra, *A Discipline of Programming.* Englewood Cliffs, N.J.: Prentice Hall, 1976.
[56] M. E. Hopkins, “A Case for the GOTO,” *Proc. 25th Nat’l ACM Conf.,* 1972, pp. 787–790, reprinted in Yourdon, *Classics in Software Engineering,* pp. 101–109; W. A. Wulf, “A Case Against the GOTO,” *Proc. 25th Nat’l ACM Conf.,* pp. 791–797, reprinted in Yourdon, *Classics in Software Engineering,* pp. 85–98.
[57] Donald Knuth, “Structured Programming With Go To Statements,” *Computing Surveys,* vol. 6, pp. 261–301, Dec. 1974.
[58] R. A. DeMillo, S. C. Eisenstat, and R. J. Lipton, “Can Structured Programs Be Efficient?” *SIGPLAN Notices,* vol. 11, p. 16, Oct. 1976.
[59] Ronald E. Jeffries, “Letter,” *SIGPLAN Notices,* vol. 11, p. 1, Dec. 1976.
[60] William Rosenfeld, “Letter,” *SIGPLAN Notices,* vol. 11, p. 3, Dec. 1976.
[61] Henry F. Ledgard and Michael Marcotty, “A Genealogy of Control Structures,” *Comm. ACM,* vol. 18, p. 629, Nov. 1975.
[62] Mario J. Gonzalez, Jr., “Workshop Report: The Science of Design,” *Computer,* vol. 12, p. 113, Dec. 1979.
[63] Tom Gilb, “Letter,” *Software Eng. Notes,* vol. 3, p. 28, July 1978.
[64] Kenneth W. Kolence, “A Software View of Measurement Tools,” *Datamation,* vol. 17, p. 32, Jan. 1, 1971.
[65] Maurice H. Halstead, *Elements of Software Science.* New York: Elsevier, 1977.
[66] Thomas J. McCabe, “A Complexity Measure,” *IEEE Transactions on Software Engineering,* vol. 2, p. 308, Dec. 1976.
[67] Glenford J. Meyers, “An Extension to the Cyclomatic Measure of Program Complexity,” *SIGPLAN Notices,* vol. 12, p. 61, Oct. 1977.
[68] James L. Elshoff and Michael Marcotty, “On the Use of the Cyclomatic Number to Measure Program Complexity,” *SIGPLAN Notices,* vol. 13, p. 38, Dec. 1978.
[69] Alonzo G. Grace, Jr., “The Dimensions of Complexity,” *Datamation,* vol. 23, p. 317, Sept. 1977.
[70] Edward T. Chen, “Program Complexity and Programmer Productivity,” *IEEE Transactions on Software Engineering,* vol. 4, p. 188, May 1978.
[71] Bill Curtis et al., “Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics,” *IEEE Transactions on Software Engineering,* vol. 5, p. 103, Mar. 1979.
[72] N. F. Schneidewind and Heinz-Michael Hoffmann, “An Experiment in Software Error Data Collection and Analysis,” *IEEE Transactions on Software Engineering,* vol. 5, p. 283, May 1979.
[73] Warren Harrison et al., “Applying Software Complexity Metrics to Program Maintenance,” *Computer,* vol. 15, p. 78, Sept. 1982.
[74] W. M. Evangelist, “Relationships Among Computational, Software, and Intuitive Complexity,” *SIGPLAN Notices,* vol. 18, p. 58, Dec. 1983.
[75] Victor R. Basili, Richard W. Selby, Jr., and Tsai-Yun Phillips, “Metric Analysis and Data Validation Across Fortran Projects,” *IEEE Transactions on Software Engineering,* vol. 9, p. 662, Nov. 1983.
[76] Martin Shepperd, “A Critique of Cyclomatic Complexity as a Software Metric,” *Software Eng. J.,* vol. 3, p. 35, Mar. 1988.
[77] J. Paul Myers, Jr., “The Complexity of Software Testing,” *Software Eng. J.,* vol. 7, p. 13, Jan. 1992.
[78] John C. Munson and Taghi M. Khoshgoftaar, “Measuring Dynamic Program Complexity,” *IEEE Software,* vol. 9, pp. 48-49, Nov. 1992.
[79] Victor R. Basili, “Tailoring SQA to Fit Your Own Life Cycle,” *IEEE Software,* vol. 5, p. 87, Mar. 1988.
[80] Shari L. Pfleeger, “Lessons Learned in Building a Corporate Metrics Program,” *IEEE Software,* vol. 10, p. 74, May 1993.
[81] Bernard Elspas, Milton W. Green, and Karl N. Levitt, “Software Reliability,” *Computer,* vol. 4, p. 22, Jan./Feb. 1971.
[82] John L. Kirkley, “The Critical Issues: A 1974 Perspective,” *Datamation,* vol. 20, p. 65, Jan. 1974.
[83] T. J. Vander Noot, “Systems Testing ... a Taboo Subject?” *Datamation,* vol. 17, p. 64, Nov. 15, 1971.
[84] Dorothy A. Walsh, “Structured Testing,” *Datamation,* vol. 23, p. 111, July 1977.
[85] Paul F. Barbuto, Jr., and Joe Geller, “Tools for Top-Down Testing,” *Datamation,* vol. 24, p. 178, Oct. 1978.
[86] Laura L. Scharer, “Improving System Testing Techniques,” *Datamation,* vol. 23, p. 117, Sept. 1977.
[87] John B. Goodenough and Susan L. Gerhart, “Toward a Theory of Test Data Selection,” *IEEE Transactions on Software Engineering,* vol. 1, p. 165, June 1975.
[88] B. Chandrasekaran, “Guest Editorial,” *IEEE Transactions on Software Engineering,* vol. 6, p. 235, May 1980.
[89] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward, “Hints on Test Data Selection: Help for the Practicing Programmer,” *Computer,* vol. 11, p. 41, Apr. 1978.
[90] Elaine J. Weyuker and Thomas J. Ostrand, “Theories of Program Testing and the Application of Revealing Subdomains,” *IEEE Transactions on Software Engineering,* vol. 6, p. 245, May 1980.
[91] Nathan H. Petschenik, “Practical Priorities in System Testing,” *IEEE Software,* vol. 2, p. 18, Sept. 1985.
[92] Simeon C. Ntafos, “On Required Element Testing,” *IEEE Transactions on Software Engineering,* vol. 10, p. 795, Nov. 1984.
[93] Samuel T. Redwine, Jr., “An Engineering Approach to Software Test Data Design,” *IEEE Transactions on Software Engineering,* vol. 9, p. 192, Mar. 1983.

[94] Robert W. Floyd, “Assigning Meanings to Programs,” *Mathematical Aspects of Computer Science,* Proceedings of Symposia in Applied Mathematics. Providence, R.I.: American Mathematical Society, 1967, pp. 19–32. For a more in-depth discussion of the history of research on formal verification, see C. B. Jones, *The Search for Tractable Ways of Reasoning About Programs.* Manchester, England: Dept. of Computer Science, Manchester Univ., 1992, UMCS-92-4-4.

[95] C. A. R. Hoare, “Proof of a Program: FIND,” *Comm. ACM,* vol. 14, p. 39, Jan. 1971.

[96] M. Foley and C. A. R. Hoare, “Proof of a Recursive Program: Quicksort,” *Computer J.,* vol. 14, p. 391, Nov. 1971.

[97] C. A. R. Hoare, “Proof of a Structured Program: ‘The Sieve of Eratosthenes,’” *Computer J.,* vol. 15, p. 321, Nov. 1972.

[98] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis, “Social Processes and Proofs of Theorems and Programs,” *Comm. ACM,* vol. 22, p. 271, May 1979.

[99] Leslie Lamport, “Letter,” *Comm. ACM,* vol. 22, p. 624, Nov. 1979.

[100] W. D. Maurer, “Letter,” *Comm. ACM,* vol. 22, p. 625, Nov. 1979.

[101] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis, “Letter,” *Comm. ACM,* vol. 22, p. 630, Nov. 1979.

[102] Henry Petroski, *To Engineer Is Human: The Role of Failure in Successful Design.* New York: St. Martin’s Press, 1985, p. 165.

[103] Richard Hill, “Letter,” *Comm. ACM,* vol. 22, p. 621, Nov. 1979.

[104] H. Lienhard, “Letter,” *Comm. ACM,* vol. 22, p. 622, Nov. 1979.

[105] Edsger W. Dijkstra, “On a Political Pamphlet from the Middle Ages,” *Software Eng. Notes,* vol. 3, p. 14, Apr. 1978.

[106] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis, “Letter,” *Software Eng. Notes,* vol. 3, pp. 16–17, Apr. 1978.

[107] H. J. Jeffrey, “Letter,” *Software Eng. Notes,* vol. 3, p. 18, Apr. 1978.

[108] Raymond J. Rubey, Joseph A. Dana, and Peter W. Biche, “Quantitative Aspects of Software Validation,” *IEEE Transactions on Software Engineering,* vol. 1, p. 152, June 1975.

[109] Douglas T. Ross and Kenneth E. Schoman, Jr., “Structured Analysis for Requirements Definition,” *IEEE Transactions on Software Engineering,* vol. 3, p. 6, Jan. 1977.

[110] Susan Gerhart, “Workshop Report: Software Testing and Test Documentation,” *Computer,* vol. 12, p. 99, Mar. 1979.

[111] Edsger W. Dijkstra, “Correctness Concerns and, Among Other Things, Why They Are Resented,” *SIGPLAN Notices,* vol. 10, p. 547, June 1975.

[112] Andrew S. Tanenbaum, “In Defense of Program Testing or Correctness Proofs Considered Harmful,” *SIGPLAN Notices,* vol. 11, p. 68, May 1976.

[113] Susan L. Gerhart and Lawrence Yelowitz, “Observations of Fallibility in Applications of Modern Programming Methodologies,” *IEEE Transactions on Software Engineering,* vol. 2, p. 206, Sept. 1976.

[114] David L. Parnas, “Letter,” *Software Eng. Notes,* vol. 3, p. 20, Oct. 1978.

[115] Debra J. Richardson and Lori A. Clarke, “Partition Analysis: A Method Combining Testing and Verification,” *IEEE Transactions on Software Engineering,* vol. 11, p. 1,488, Dec. 1985.

[116] James H. Fetzer, “Program Verification: The Very Idea,” *Comm. ACM,* vol. 31, p. 1,057, Sept. 1988.

[117] Mark Ardis et al., “Letter,” *Comm. ACM,* vol. 32, p. 287, Mar. 1989.

[118] Richard Hill, “Letter,” *Comm. ACM,* vol. 32, p. 790, July 1989.

[119] James C. Pleasant, “Letter,” *Comm. ACM,* vol. 32, p. 374, Mar. 1989.

[120] Lawrence Paulson, Avra Cohen, and Michael Gordon, “Letter,” *Comm. ACM,* vol. 32, p. 375, Mar. 1989.

[121] James H. Fetzer, “Letter,” *Comm. ACM,* vol. 32, p. 378, Mar. 1989.

[122] John Dobson and Brian Randell, “Program Verification: Public Image and Private Reality,” *Comm. ACM,* vol. 32, pp. 420–422, Apr. 1989.

[123] James H. Fetzer, “Letter,” *Comm. ACM,* vol. 32, p. 381, Mar. 1989.

[124] David A. Nelson, “Letter,” *Comm. ACM,* vol. 32, p. 792, July 1989.

[125] James H. Fetzer, “Letter,” *Comm. ACM,* vol. 32, p. 381, Mar. 1989.

[126] Leon Stucki, “Guest Editorial,” *IEEE Transactions on Software Engineering,* vol. 2, p. 194, Sept. 1976.

[127] C. V. Ramamoorthy, Siu-Bun F. Ho, and W. T. Chen, “On the Automated Generation of Program Test Data,” *IEEE Transactions on Software Engineering,* vol. 2, p. 293, Dec. 1976.

[128] George J. Schick and Ray W. Wolverton, “An Analysis of Competing Software Reliability Models,” *IEEE Transactions on Software Engineering,* vol. 4, p. 105, Mar. 1978.

[129] Nancy G. Leveson, “Software Safety,” *Software Eng. Notes,* vol. 7, p. 21, Apr. 1982.

[130] Algirdas Avizienis and John P. J. Kelly, “Fault Tolerance by Design Diversity: Concepts and Experiments,” *Computer,* vol. 17, p. 67, Aug. 1984.

[131] Dave E. Eckhardt, Jr., and Larry D. Lee, “A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors,” *IEEE Transactions on Software Engineering,* vol. 11, p. 1,511, Dec. 1985.

[132] John C. Knight and Nancy G. Leveson, “An Experimental Evaluation of the Assumption of Independence in Multiversion Programming,” *IEEE Transactions on Software Engineering,* vol. 12, p. 96, Jan. 1986.

[133] John C. Knight and Nancy G. Leveson, “A Reply to the Criticisms of the Knight & Leveson Experiment,” *Software Eng. Notes,* vol. 15, pp. 24–35, Jan. 1990.

[134] Bev Littlewood and Douglas R. Miller, “Conceptual Modeling of Coincident Failures in Multiversion Software,” *IEEE Transactions on Software Engineering,* vol. 15, p. 1,596, Dec. 1989.

[135] Susan S. Brilliant, John C. Knight, and Nancy G. Leveson, “Analysis of Faults in an N-Version Software Experiment,” *IEEE Transactions on Software Engineering,* vol. 16, p. 245, Feb. 1990.

[136] Abdalla A. Abdel-Ghaly, P. Y. Chan, and Bev Littlewood, “Evaluation of Competing Software Reliability Predictions,” *IEEE Transactions on Software Engineering,* vol. 12, p. 950, Sept. 1986.

[137] Richard Hamlet, “Special Section on Software Testing,” *Comm. ACM,* vol. 31, pp. 665–666, June 1988.

[138] Galen Gruman, “IFIP Participants Debate Programming Approaches,” *IEEE Software,* vol. 6, p. 76, Nov. 1989.

[139] Donald MacKenzie, “Negotiating Arithmetic, Constructing Proof: The Sociology of Mathematics and Information Technology,” *Social Studies of Science,* vol. 23, pp. 37–65, Feb. 1993.

[140] National Bureau of Standards, *Guideline for Lifecycle Validation, Verification, and Testing of Software,* Washington, D.C., 1983, NBS FIPS 101; quoted in David Gelperin and Bill Hetzel, “The Growth of Software Testing,” *Comm. ACM,* vol. 31, p. 690, June 1988.

[141] Dolores R. Wallace and Roger U. Fujii, “Verification and Validation: Techniques to Assure Reliability,” *IEEE Software,* vol. 6, p. 9, May 1989.

[142] C. A. R. Hoare, “An Axiomatic Basis for Computer Programming,” *Comm. ACM,* vol. 12, p. 576, Oct. 1969.

[143] C. A. R. Hoare, “Professionalism,” *Computer Bull.,* 2nd series, p. 3, Sept. 1981.

[144] Stuart Shapiro, “Its Own Worst Enemy: How Software Engineering Has Fallen Victim to Engineering Mythology,” CRICT Discussion Paper No. 25, Brunel Univ., 1992.

[145] Nancy G. Leveson, “Formal Methods in Software Engineering,” *IEEE Transactions on Software Engineering,* vol. 16, p. 929, Sept. 1990.

[146] Susan L. Gerhart, “Applications of Formal Methods: Developing Virtuoso Software,” *IEEE Software,* vol. 10, p. 10, Sept. 1990.

[147] C. B. Jones, “Theorem Proving and Software Engineering,” *Software Eng. J.,* vol. 3, p. 2, Jan. 1988.

[148] Susan Gerhart, “Formal Methodists Warn of Software Disasters,” *IEEE Software,* vol. 6, p. 77, Nov. 1989.

[149] Anthony Hall, “Seven Myths of Formal Methods,” *IEEE Software,* vol. 7, p. 13, Sept. 1990.

[150] Jeannette M. Wing, “A Specifier’s Introduction to Formal Methods,” *Computer,* vol. 23, p. 13, Sept. 1990.

[151] Harlan D. Mills, Michael Dyer, and Richard C. Linger, “Cleanroom Software Engineering,” *IEEE Software,* vol. 4, p. 20, Sept. 1987.

[152] Richard W. Shelby, Victor R. Basili, and F. Terry Baker, “Cleanroom Software Development: An Empirical Evaluation,” *IEEE Transactions on Software Engineering,* vol. 13, pp. 1,027–1,037, Sept. 1987.

[153] D. A. Duce and E. V. C. Fielding, “Formal Specification—a Comparison of Two Techniques,” *Computer J.,* vol. 30, p. 327, Aug. 1987.

[154] C. A. R. Hoare, “An Overview of Some Formal Methods for Program Design,” *Computer,* vol. 20, pp. 90–91, Sept. 1987.
[155] Carl Chang, “Let’s Stop the Bipolar Drift,” *IEEE Software,* vol. 7, p. 4, May 1990.
[156] I. F. Currie, “NewSpeak: An Unexceptional Language,” *Software Eng. J.,* vol. 1, pp. 170–176, July 1986.
[157] Jean E. Sammet, *Programming Languages: History and Fundamentals.* Englewood Cliffs, N.J.: Prentice Hall, 1969; and Richard L. Wexelblat, ed., *History of Programming Languages.* New York: Academic Press, 1981. These works contain in-depth histories of these older languages.
[158] David A. Fisher, “DoD’s Common Programming Language Effort,” *Computer,* vol. 11, p. 25, Mar. 1978.
[159] Barry W. Boehm, “Software and Its Impact: A Quantitative Assessment,” *Datamation,* vol. 19, p. 48, May 1973.
[160] Edsger W. Dijkstra, “DoD-I: The Summing Up,” *SIGPLAN Notices,* vol. 13, pp. 24–26, July 1978.
[161] Robert L. Glass, “From Pascal to Pebbleman ... and Beyond,” *Datamation,* vol. 25, pp. 146–147, July 1979.
[162] Rob Kling and Walter Scacchi, “The DoD Common High Order Programming Language Effort (DoD-1): What Will the Impacts Be?” *SIGPLAN Notices,* vol. 14, pp. 32–40, Feb. 1979.
[163] Paul R. Eggert, “Letter,” *SIGPLAN Notices,* vol. 15, p. 9, Jan. 1980.
[164] J. T. Galkowski, “A Critique of the DOD Common Language Effort,” *SIGPLAN Notices,* vol. 15, p. 15, June 1980.
[165] Patrick Skelly, “The ACM Position on Standardization of the Ada Language,” *Comm. ACM,* vol. 25, p. 119, Feb. 1982.
[166] Henry F. Ledgard and Andrew Singer, “Scaling Down Ada (or Towards a Standard Ada Subset),” *Comm. ACM,* vol. 25, p. 121, Feb. 1982.
[167] Robert L. Glass, “Letter,” *Comm. ACM,* vol. 25, p. 500, July 1982.
[168] Randall Leavitt, “Letter,” *Comm. ACM,* vol. 25, p. 500, July 1982.
[169] Brian Wichmann, “Is Ada Too Big? A Designer Answers the Critics,” *Comm. ACM,* vol. 27, p. 103, Feb. 1984.
[170] William I. MacGregor, “Letter,” *SIGPLAN Notices,* vol. 13, p. 18, Sept. 1978.
[171] Peter Wegner, “The Ada Language and Environment,” *Software Eng. Notes,* vol. 5, p. 9, Apr. 1980.
[172] Charles Antony Richard Hoare, “The Emperor’s Old Clothes,” *Comm. ACM,* vol. 24, p. 82, Feb. 1981.
[173] “DOD Interim Policy on Ada Issued,” *Comm. ACM,* vol. 26, p. 706, Sept. 1983.
[174] Saul Rosen, “Programming Systems and Languages 1965–1975,” *Comm. ACM,* vol. 15, p. 591, July 1972.
[175] David R. Hanson, “A Simple Technique for Representing Strings in Fortran IV,” *Comm. ACM,* vol. 17, p. 646, Nov. 1974.
[176] Daniel D. McCracken, “Is There a Fortran in Your Future?” *Datamation,* vol. 19, p. 237, May 1973.
[177] Daniel D. McCracken, “Letter,” *Comm. ACM,* vol. 28, p. 568, June 1985.
[178] “The NCC: Reminiscent of the Late Sixties,” *Datamation,* vol. 21, p. 104, June 1975.
[179] Tomasz Kowaltowski, “Letter,” *SIGPLAN Notices,* vol. 10, p. 4, Aug. 1975.
[180] Eric Campbell, “Letter,” *SIGPLAN Notices,* vol. 11, p. 2, May 1976.
[181] Stuart W. Rowland, “Some Comments on Structured Fortran,” *SIGPLAN Notices,* vol. 11, p. 45, Oct. 1976.
[182] Michael J. Viehman, “Letter,” *SIGPLAN Notices,* vol. 10, p. 8, Oct. 1975.
[183] Anthony Ralston and Jerrold L. Wagener, “Structured Fortran—an Evolution of Standard Fortran,” *IEEE Transactions on Software Engineering,* vol. 2, p. 154, Sept. 1976.
[184] Daniel D.McCracken, “Let’s Hear It for COBOL!” *Datamation,* vol. 22, p. 242, May 1976.
[185] Peter Naur, “Programming Languages, Natural Languages, and Mathematics,” *Comm. ACM,* vol. 18, pp. 678–680, Dec. 1975.
[186] Michael Hammer et al., “A Very High Level Programming Language for Data Processing Applications,” *Comm. ACM,* vol. 20, pp. 832–833, Nov. 1977.
[187] Mark R. Crispin, “Letter,” *Datamation,* vol. 22, p. 7, Nov. 1976.
[188] A. C. Larman, “Letter,” *Computer Bull.,* 1st series, no. 16, p. 506, Nov. 1972.
[189] Ware Myers, “Key Developments in Computer Technology: A Survey,” *Computer,* vol. 9, p. 59, Nov. 1976.
[190] Linda Runyan, “Software Still a Sore Spot,” *Datamation,* vol. 27, p. 165, Mar. 1981.
[191] Ronald A. Frank, “Let the Users Program,” *Datamation,* vol. 28, p. 88, Jan. 1982.
[192] Nigel S. Read and Douglas L. Harmon, “Language Barrier to Productivity,” *Datamation,* vol. 29, p. 209, Feb. 1983.
[193] John Cardullo and Herb Jacobsohn, “Letter,” *Datamation,* vol. 29, p. 24, May 1983.
[194] Bill Inmon, “Rethinking Productivity,” *Datamation,* vol. 30, p. 185, June 15, 1984.
[195] Michael H. Brown, “Letter,” *Datamation,* vol. 30, p. 23, Sept. 15, 1984.
[196] F. J. Grant, “The Downside of 4GLs,” *Datamation,* vol. 31, p. 99, July 15, 1985.
[197] Peter Wegner, “Capital-Intensive Software Technology, Part 2: Programming in the Large,” *IEEE Software,* vol. 1, p. 31, July 1984.
[198] Alex Pines and Dan Pines, “Don’t Shoot the Programmers,” *Datamation,* vol. 29, p. 114, Aug. 1983.
[199] Santosh K. Misra and Paul J. Jalics, “Third-Generation Versus Fourth-Generation Software Development,” *IEEE Software,* vol. 6, p. 14, July 1989.
[200] Bruce Hailpern, “Multiparadigm Languages and Environments,” *IEEE Software,* vol. 3, p. 6, Jan. 1986.
[201] Pamela Zave, “A Compositional Approach to Multiparadigm Programming,” *IEEE Software,* vol. 6, p. 15, Sept. 1989.
[202] John Backus, “Can Programming Be Liberated From the von Neumann Style? A Functional Style and Its Algebra of Programs,” *Comm. ACM,* vol. 21, p. 514, Aug. 1978.
[203] R. N. Caffin, “Heresy on High-Level Languages,” *Computer,* vol. 12, pp. 108–109, Mar. 1979.
[204] Jim Haynes, “Comment on High-Level Heresy,” *Computer,* vol. 12, p. 109, Mar. 1979.
[205] David Feign, “Letter,” *Computer,* vol. 12, p. 122, Sept. 1979.
[206] William A. Wulf, “Trends in the Design and Implementation of Programming Languages,” *Computer,* vol. 13, p. 15, Jan. 1980.
[207] Victor R. Basili and Albert J. Turner, “Iterative Enhancement: A Practical Technique for Software Development,” *IEEE Transactions on Software Engineering,* vol. 1, p. 390, Dec. 1975.
[208] W. P. Dodd, “Prototype Programs,” *Computer,* vol. 13, p. 81, Feb. 1980.
[209] Pat Hall, Janet Low, and Steve Woolgar, “Human Factors in Information Systems Development: A Project Report,” CRICT Discussion Paper No. 31, Brunel University, 1992.
[210] Fletcher J. Buckley, “A Modest Proposal,” *Computer,* vol. 15, p. 103, Dec. 1982.
[211] Daniel D. McCracken and Michael A. Jackson, “Life Cycle Concept Considered Harmful,” *Software Eng. Notes,* vol. 7, p. 32, Apr. 1982.
[212] G. R. Gladden, “Stop the Life-Cycle, I Want to Get Off,” *Software Eng. Notes,* vol. 7, p. 35, Apr. 1982.
[213] Patrick A. V. Hall, “Letter,” *Software Eng. Notes,* vol. 7, p. 23, July 1982.
[214] Bruce I. Blum, “The Life Cycle—a Debate Over Alternate Models,” *Software Eng. Notes,* vol. 7, p. 18, Oct. 1982.
[215] Joseph W. Chambers, “Letter,” *Comm. ACM,* vol. 26, p. 108, Feb. 1983.
[216] Ware Myers, “Can Software Development Processes Improve—Drastically?” *IEEE Software,* vol. 1, p. 101, July 1984; Mark Dowson and Jack C. Wileden, “A Brief Report on the International Workshop on the Software Process and Software Environments,” *Software Eng. Notes,* vol. 10, p. 21, July 1985.
[217] Stefano Nocentini, “The Planning Ritual,” *Datamation,* vol. 31, p. 128, Apr. 15, 1985.
[218] R. E. A. Mason and T. T. Carey, “Prototyping Interactive Information Systems,” *Comm. ACM,* vol. 26, p. 348, May 1983.
[219] Jerry Schulz, “Letter,” *Datamation,* vol. 29, p. 24, Sept. 1983.
[220] Barry W. Boehm, Terence E. Gray, and Thomas Seewaldt, “Prototyping Versus Specifying: A Multiproject Experiment,” *IEEE Transactions on Software Engineering,* vol. 10, p. 300, May 1984.
[221] Gruia-Catalin Roman, “A Taxonomy of Current Issues in Requirements Engineering,” *Computer,* vol. 18, p. 20, Apr. 1985.

[222] Zohar Manna and Richard Waldinger, "Synthesis: Dreams => Programs," *IEEE Transactions on Software Engineering,* vol. 5, p. 295, July 1979.
[223] D. J. Cooke, "Program Transformation," *Computer Bull.,* 2nd series, p. 20, Dec. 1979.
[224] David W. Wile, "Program Developments: Formal Explanations of Implementations," *Comm. ACM,* vol. 26, p. 903, Nov. 1983.
[225] Pamela Zave, "The Operational Versus the Conventional Approach to Software Development," *Comm. ACM,* vol. 27, p. 113, Feb. 1984.
[226] Manfred Broy and Peter Pepper, "Program Development as a Formal Activity," *IEEE Transactions on Software Engineering,* vol. 7, p. 22, Jan. 1981.
[227] Barry W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer,* vol. 21, p. 65, May 1988.
[228] Dennis M. Ritchie and Ken Thompson, "The Unix Time-Sharing System," *Comm. ACM,* vol. 17, p. 365, July 1974.
[229] Evan L. Ivie, "The Programmer's Workbench—a Machine for Software Development," *Comm. ACM,* vol. 20, p. 746, Oct. 1977.
[230] B. W. Kernighan and P. J. Plauger, "Software Tools," *Software Eng. Notes,* vol. 1, p. 15, May 1976.
[231] Anthony I. Wasserman, "Automated Development Environments," *Computer,* vol. 14, p. 9, Apr. 1981.
[232] Michael Lesk, "Another View," *Datamation,* vol. 27, p. 139, Nov. 1981.
[233] David Morris, "How Not to Worry About Unix," *Datamation,* vol. 30, p. 83, Aug. 1, 1984.
[234] Dennis F. Barlow and Norman S. Zimbel, "Unix—How Important Is It?" *Datamation,* vol. 30, p. 101, Aug. 1, 1984.
[235] T. H. Crowley, L. L. Crume, and C. B. Hergenhan, "AT&T Asks for a Unix Standard," *Datamation,* vol. 30, p. 100, Aug. 1, 1984.
[236] Peter J. Denning, "Throwaway Programs," *Comm. ACM,* vol. 24, p. 58, Feb. 1981.
[237] Grover P. Righter, "Letter," *Datamation,* vol. 30, p. 16, Nov. 1, 1984.
[238] Tim Teitelbaum, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *SIGPLAN Notices,* vol. 14, p. 75, Oct. 1979.
[239] Vic Stenning et al., "The Ada Environment: A Perspective," *Computer,* vol. 14, p. 27, June 1981.
[240] Charles Rich and Howard E. Shrobe, "Initial Report on a Lisp Programmer's Apprentice," *IEEE Transactions on Software Engineering,* vol. 4, p. 456, Nov. 1978.
[241] Elliot Soloway, "A Cognitively-Based Methodology for Designing Languages/Environments/Methodologies," *SIGPLAN Notices,* vol. 19, p. 195, May 1984.
[242] Ware Myers, "MCC: Planning the Revolution in Software," *IEEE Software,* vol. 2, p. 72, Nov. 1985.
[243] J. Trenouth, "A Survey of Exploratory Software Development," *Computer J.,* vol. 34, p. 153, Apr. 1991.
[244] Winston Royce, "Has the Exploratory Approach Come of Age?" *IEEE Software,* vol. 10, p. 104, Jan. 1993.
[245] A. Nico Habermann and David Notkin, "Gandalf: Software Development Environments," *IEEE Transactions on Software Engineering,* vol. 12, p. 1,118, Dec. 1986.
[246] Jayshree Ramanathan and Soumitra Sarkar, "Providing Customized Assistance for Software Lifecycle Approaches," *IEEE Transactions on Software Engineering,* vol. 14, p. 749, June 1988.
[247] Ronald J. Norman and Gene Forte, "CASE in the '90s," *Comm. ACM,* vol. 35, p. 30, Apr. 1992.
[248] Arthur J. Collmeyer, "Developments in Design Automation," *Computer,* vol. 7, p. 11, Jan. 1974.
[249] Peter Freeman, "Automating Software Design," *Computer,* vol. 7, p. 34, Apr. 1974.
[250] Frederick P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer,* vol. 20, p. 11, Apr. 1987.
[251] Paul Rook, "Controlling Software Projects," *Software Eng. J.,* vol. 1, p. 8, Jan. 1986.
[252] Anthony Finkelstein, "London Open CRIS Conference," *Computer Bull.,* 2nd series, p. 5, Sept. 1984.
[253] A. T. Wood-Harper and G. Fitzgerald, "A Taxonomy of Current Approaches to Systems Analysis," *Computer J.,* vol. 25, pp. 12–16, Feb. 1982.
[254] G. Fitzgerald, N. Stokes, and J. R. G. Wood, "Feature Analysis of Contemporary Information Systems Methodologies," *Computer J.,* vol. 28, no. 3, pp. 223–230, 1985.
[255] J. Mayhew and P. A. Dearnley, "An Alternative Prototyping Classification," *Computer J.,* vol. 30, pp. 481–484, Dec. 1987.
[256] Xiping Song and Leon J. Osterweil, "Toward Objective, Systematic Design-Method Comparisons," *IEEE Software,* vol. 9, p. 44, May 1992.
[257] D. M. Episkopou and A. T. Wood-Harper, "Towards a Framework to Choose Appropriate IS Approaches," *Computer J.,* vol. 29, p. 222, June 1986.
[258] Alan M. Davis, Edward H. Bersoff, and Edward R. Comer, "A Strategy for Comparing Alternative Software Development Life Cycle Models," *IEEE Transactions on Software Engineering,* vol. 14, pp. 1,453–1,461, Oct. 1988.
[259] Bo Sanden, "The Case for Electric Design of Real-Time Software," *IEEE Transactions on Software Engineering,* vol. 15, p. 360, Mar. 1989.
[260] "Panel on Problems of the 80s, ICSE Atlanta," *Software Eng. Notes,* vol. 3, p. 29, July 1978.
[261] Ware Myers, "New British Tool Centre a Response to Software Complexity," *IEEE Software,* vol. 2, p. 94, Nov. 1985.
[262] W. Wayt Gibbs, "Software's Chronic Crisis," *Scientific Am.,* pp. 72–81, Sept. 1994.
[263] Annie Kuntzmann-Combelles, "Software Help Wanted: Revolutionary Thinkers," *IEEE Software,* vol. 9, p. 10, Sept. 1992.

Stuart Shapiro is a Visiting Research Fellow in the Centre for Research into Innovation, Culture and Technology (CRICT) at Brunel University in England. He has previously been a Research Fellow in the Centre for Technology Strategy at the Open University, also in England. He holds a BS in computer science from Northwestern University and a PhD in applied history and social sciences from Carnegie Mellon University. His research has focused on the history and sociology of software engineering. He also has interests in engineering professional development and in information technology and privacy.

The author can be contacted at
Centre for Research into Innovation, Culture and Technology
Brunel University
Uxbridge, Middlesex UB8 3PH, United Kingdom
e-mail: s_shapiro@acm.org