portant, it must be realized that the full potential of the information utility depends on dramatic changes in information system usage and structure (for example, electronic mail, electronic filing, automated office operation, and electronic funds transfer). In many cases, these changes are more under the control of information system designers and users than of computer manufacturers. An informed user community is an important requirement for future progress.

Summary

Demands for more effective information management, coupled with advances in computer hardware and software technology, have resulted in the emergence of the information utility concept, whereby computers specialized for information storage and processing serve as information nodes. The information nodes, which may be interconnected, can provide information management services to both conventional and personal computers. In this article the key hardware and software components of classical information systems are described to provide background on the requirements for an information utility. Four approaches to the development of specialized information nodes, drawing on various advances in technology, are presented: (i) firmware enhancement, (ii) intelligent controllers, (iii) minicomputer back-end processors, and (iv) highly modular database machines. The benefits of these advances will be systems that are more efficient, reliable, and easy to use.

References

1. H. Hollomon, *Technol. Rev.* **77**, 57 (January 1975).
2. J. W. Forrester, "Dynamics of socio-economic systems," Report D-2230-1, MIT Systems Dynamics Group, Cambridge, Mass., August 1975.
3. J. J. Donovan and S. E. Madnick, *Database* **8** (No. 9), 79 (1977).
4. S. E. Madnick and J. J. Donovan, *Operating Systems* (McGraw-Hill, New York, 1974).
5. B. Edelson, *Science* **195**, 1125 (1977).
6. D. Farber and P. Barran, *ibid.*, p. 1166.
7. M. Irwin and S. Johnson, *ibid.*, p. 1170.
8. S. E. Miller, *ibid.*, p. 1211.
9. R. J. Potter, *ibid.*, p. 1160.
10. W. Myers, *Computer* **9** (No. 11), 48 (1976).
11. R. Noyce, *Science* **195**, 1102 (1977).
12. J. Rajchman, *ibid.*, p. 1223.
13. H. D. Mills, *ibid.*, p. 1199.
14. J. P. Eckert, *Computer* **9** (No. 12), 58 (1976).
15. J. Conway and P. Snigier, *EDN* **21** (No. 21), 95 (1976).
16. H. D. Toong, *AFIPS Conf. Proc.* **44**, 567 (1975).
17. S. E. Madnick, *Technol. Rev.* **75**, 8 (July/August 1973).
18. ———, *IEEE Intercon Conf. Rec.* (1975), pp. 20/1-1 to 20/1-7.
19. R. R. Martin and H. D. Frankel, *Computer* **8** (No. 2), (1975).
20. J. H. Wensley, *ibid.*, p. 30.
21. P. J. Denning, *ACM Comput. Surv.* **2** (No. 3), 153 (1970).
22. G. Mueller, *Computer* **9** (No. 12), 100 (1976).
23. S. H. Fuller, V. R. Lesser, C. G. Bell, C. H. Kaman, *IEEE Trans. Comput.* **C-25** (No. 10), 1000 (1976).
24. A. Armenti, S. Galley, R. Goldberg, J. Nolan, A. Sholl, *AFIPS Conf. Proc.* **36**, 313 (1970).
25. C. Bachman, *ibid.* **44**, 569 (1975).
26. A. Hassitt and L. E. Lyon, *IBM Syst. J.* **15** (No. 4), 358 (1976).
27. R. L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger, *ibid.* **9** (No. 2), 78 (1970).
28. R. M. Meade, *AFIPS Conf. Proc.* **37**, 33 (1970).
29. C. Johnson, *ibid.* **44**, 509 (1975).
30. G. R. Ahearn, Y. Dishon, R. N. Snively, *IBM J. Res. Dev.* **16** (No. 1), 11 (1972).
31. S. Y. Su and G. J. Lipovski, in *Proceedings of the International Conference on Very Large Data Bases* (Association for Computing Machinery, New York, 1975), pp. 456–472.
32. E. A. Ozkarahan, S. A. Schuster, K. C. Smith, *AFIPS Conf. Proc.* **44**, 379 (1975).
33. S. C. Lin, D. C. P. Smith, J. M. Smith, *ACM Trans. Database Syst.* **1**, 53 (March 1976).
34. R. H. Canaday, R. D. Harrison, E. L. Ivie, J. L. Ryder, L. A. Wehr, *Commun. ACM* **17** (No. 10), 575 (1974).
35. H. C. Heacox, E. S. Cosloy, J. B. Cohen, in *Proceedings of the International Conference on Very Large Data Bases* (Association for Computer Machinery, New York, 1975), pp. 511–513.
36. R. I. Baum and D. K. Hsiao, *IEEE Trans. Comput.* **C-25** (No. 12), 1254 (1976).
37. T. Marill and D. Stern, *AFIPS Conf. Proc.* **44**, 389 (1975).
38. J. Verity, *Electron. News* (3 January 1977), p. 28.
39. S. M. Ornstein, W. R. Crowgher, M. F. Kraley, R. D. Bressler, A. Michel, F. E. Heart, *AFIPS Conf. Proc.* **44**, 551 (1975).
40. S. E. Madnick, *ibid.*, p. 581.
41. J. J. Donovan, *ACM Trans. Database Syst.* **4** (No. 1), 344 (1976).
42. F. G. Withington, *Datamation* **21** (No. 1), 54 (1975).

Software Engineering

A mathematical basis is needed for the practical control of computers in complex applications.

Harlan D. Mills

Computer software began as an afterthought to computer hardware, and, as long as the hardware was small and simple, software could be handled informally by scientifically trained people as a by-product of the use intended for the hardware. As hardware grew in size and complexity, richer software possibilities emerged and software specialists (programmers) appeared, to produce assemblers, compilers, operating systems, and data management systems. Although there was an early recognition of mathematical ideas in computing, for example, in mathematical logic, automata theory, and linguistics, the bulk of these software specialists were pragmatic products of computing practice rather than mathematicians. Thus, although it may seem surprising, the rediscovery of software as a form of mathematics in a deep and literal sense is just now beginning to penetrate university research and teaching, as well as industry and government practices. The forcing factor in this rediscovery has been the growth of software complexity and the inability of informal software practices and management to cope with it.

Of course, software makes totally new demands in the sheer volume of logical precision required in its application. A single project may occupy hundreds, even thousands, of people over several years, so that there are unique requirements for recording, communication, and management of the work. These unique requirements lead to almost all of the jargon in software, and, in fact, this jargon tends to obscure the mathematical character of software, as people get caught up in implementation and management details. But the failure to understand this mathematical character led to an overly complex, ad hoc view of software based on historical and accidental ideas, which were often reinvented in ignorance and haste.

The work of Dijkstra and Hoare has been a major force in this rediscovery of software as mathematics. In (*1*, p. 4.2), Dijkstra has presented an argument which sums up the case I want to make here:

> As soon as programming emerges as a battle against unmastered complexity, it is quite natural that one turns to that mental discipline whose main purpose has been since

The author is an IBM Fellow at International Business Machines Corporation, Gaithersburg, Maryland 20760. He also teaches computer science part time at the University of Maryland, College Park.

centuries to apply effective structuring to otherwise unmastered complexity. That mental discipline is more or less familiar to all of us, it is called Mathematics. If we take the existence of the impressive body of Mathematics as the experimental evidence for the opinion that for the human mind the mathematical method is indeed the most effective way to come to grips with complexity, we have no choice any longer: we should reshape our field of programming in such a way that the mathematician's methods become equally applicable to our programming problems, for there are no other means.

This rediscovery of a disciplined basis for software development is finding expression in the recognition of software engineering as a topic for international technical conferences and technical journals. Boehm (*2*) has surveyed software engineering topics from the point of view of the life cycle and management of software. Mills (*3*) has discussed software development.

The Breakdown of Simple Practice

Early computers were scarce and remarkable resources. Thus, meeting the computer on its own terms—reading binary notations, debugging, and patching in machine code, working at 2 a.m.—was a natural thing. Programming was a part of problem-solving with a fantastic new assistant. A little later, programming became a way of dealing with computers on behalf of others—scientists, administrators, or systems engineers. More programming was done on behalf of programmers themselves—building compilers, operating systems, and developing data management systems. But somewhere along the way, the magnificent dreams often turned into nightmares. What seemed easy—the development of a command control system, an information management system, a world-champion automatic chess player—turned out to be extremely difficult or impossible. Major undertakings often took two to ten times as long as had been expected or had to be abandoned.

What happened? The computers had the same simple instruction sets as before. The programmers still knew exactly how the machine worked. The designers still knew how to lay out a system and its development in a systematic modular way, which could be constructed in small units and assembled into larger and larger subsystems until the final system was completed.

What happened was that complexity was growing out of control. Program storage and speed, like dollars, can be measured and aggregated, and budgets set up to manage their allocations and use. But there is no effective concept for measuring program complexity today, and complexity budgeting and control in a quantitative way is unknown. Yet the principal limitation to the development, maintenance, and operation of data-processing systems today is sheer complexity. Hardware storage and speed are more than sufficient; in fact, storage and speed are ill used today because of inefficiencies forced by complexity and overdesign resulting from complexity.

Brooks, the manager of one of the most complex software projects yet attempted, the IBM OS/360 project, noted that "conceptual integrity is *the* most important consideration in system design" (*4*, p. 42) and reported on his experience in managing the development of OS/360 (*4*, pp. 47–48):

> It is a very humbling experience to make a multimillion-dollar mistake, but it is also very memorable. I vividly recall the night we decided how to organize the actual writing of external specifications for OS/360. The manager of architecture, the manager of control program implementation, and I were threshing out the plan, schedule, and division of responsibilities. The architecture manager had 10 good men. He asserted that they could write the specifications and do it right. It would take ten months, three more than the schedule allowed.
>
> The control program manager had 150 men. He asserted that they could prepare the specifications, with the architecture team coordinating; it would be well-done and practical, and he could do it on schedule. Furthermore, if the architecture team did it, his 150 men would sit twiddling their thumbs for ten months.
>
> To this the architecture manager responded that if I gave the control program team the responsibility, the result would not in fact be on time, but would also be three months late, and of much lower quality. I did, and it was. He was right on both counts. Moreover, the lack of conceptual integrity made the system far more costly to build and change, and I would estimate that it added a year to debugging time.

Software Redefined

The term "software" was originally a poetic synonym for "program." But, over time, important reasons have emerged to expand the meaning of "software" to something considerably beyond "program." These reasons are related to a change of view of data processing, from a hardware-centered to a systems-centered perspective. Early computers were such scarce and dramatic resources that it was natural to relate all other activities to them. But, as hardware became more available and embedded in even more complex systems, it often seemed more useful to regard hardware as components in such systems instead of their centers, as discussed by Mills (*5*).

Although such a use was not realized in the early days of computer technology, it is clear now that one major use of software consists of prescribing the harmonious cooperation of asynchronously operating hardware. But, a simple extension is possible here, specifically to incorporate people as agents of action into this formal sphere of harmonious cooperation as well. People have the same architectural characteristics as hardware but with radically different performance parameters in terms of storage, processing, and reliability, and they have radically different instruction sets. A person can carry out the logical operations of a computer (millions of times more slowly), but a person can also execute the instruction "use your common sense" (millions of times faster). For example, in an airline reservation system, customer service people serve as critical analog-to-digital components in converting voice to key stroke and digital displays back to voice in dealing with customers.

There is another, older use of software, to directly support a specific piece of hardware, which makes that device easier to use either by people or by programs. For example, a program used to control a tape drive may take in commands to read, write, or rewind the tape, but put out instructions to motors or to read-write heads at a much lower level of detail; when errors are detected, the program may call for automatic retry of physical operations, all unknown to the program user. In short, such software combined with the hardware makes up an "abstract machine," easier to use as a component than the hardware itself. For this reason, I will use the term "machine" to mean any command-driven combination of hardware and software, not just base hardware.

A set of such hardware-software machines can, in turn, be organized, through additional software which provides for their harmonious cooperation, into a larger, more capable, abstract machine, which a user (person or program) can command and receive results from as a single agent. For example, a simple computer, with its card readers, printers, and other components, all operating asynchronously under the control of an operating system, can be viewed as just such an abstract machine.

Finally, people can serve as agents of action in abstract machines, in much the same way as the hardware in the foregoing discussion, if the term "software" is expanded to include the ideas

of user's guides and operating instructions for people in their specific roles in a system.

For these reasons, software is better defined as "logical doctrine for the harmonious cooperation of people and machines." Such doctrine takes the form of programs for hardware, and combinations of user's guides and operating instructions for people. The programs may be microprograms, stored in read-only storage (for example, "firmware"), as well as ordinary programs. In brief, software defines a system of abstract machines, some of which call on other abstract machines, until people and hardware are reached as the ultimate agents of action in the system.

Data Processing

The data-processing explosion. Although nothing of the kind existed 25 years ago, every sizable business, government agency, and institution in the United States now depends in a critical way on large data-processing systems. Such a system often interfaces thousands of people with a complex of machines—computers, terminals, data-entry devices, many of them in real-time communications over considerable distances. These large systems may have large central computing subsystems, or the computing may be distributed over several centers. They are inextricably tied in with the operations of the enterprise, involved in an astonishing variety of science, engineering, and commercial tasks.

With such a short history, it is surprising that these large data-processing systems exist at all. And yet they are no gleam in a crystal ball. They are here now, and working well enough to be indispensable to every sizable undertaking in the country but working poorly enough to frustrate the managements and beneficiaries of these undertakings also.

The engineering of computer hardware developed out of the scientific research that made it possible, and manufacturers of data-processing hardware benefited directly from that science and its engineering. But the problems of putting these products of science into practical use filter back to the research community much more slowly, because the applications are not usually scientific but are instead industrial, financial, or administrative. That situation in data processing is accentuated by its very rate of development. The close alliance of science and engineering in hardware has over the last 25 years accomplished dramatic achievements in reliability, speed, storage, cost, and function. This achievement has few parallels in human history.

But the uses of this remarkable hardware in industry and government have led to large, complex, ponderous, unreliable, incompatible data-processing systems with no significant precedents or guides from science, such as helped to shape the hardware. The precedents for such complex systems are in the day-by-day operations of business and government, as conducted by people, with all the local variations and considerable inconsistencies that only people could tolerate in operational activities. As a result, during the past 25 years there has been a much slower rate of improvement, however it may be measured, in the use of hardware than in the hardware itself.

The growth of data processing in industry and government reflects the increased complexity, interdependency, and competition among enterprises. One example of the interdependency is the reporting by industry of payroll information to the Social Security Administration and the Internal Revenue Service. The information is transferred routinely in computer format (this procedure is based on the presupposition of widespread computer operations). The competition to provide better services to customers through data processing is apparent in hotel and airline reservation systems, insurance and banking, and retailing. The competition that generates these systems also creates a demand for their improvement—to systems that are more capable, more reliable, more economical, more manageable. In short, there is a remarkable hunger across both industry and government for better ideas and practices from science and engineering in data processing.

Why data-processing practices are backward. The data-processing industry is a spectacular product of science and engineering. But its practice leaves much to be desired. It is suffering the growing pains of an infant industry that has had no time to develop, try out, and select sound industrial practices—it is a mixture of great wisdom and foolish folklore. It needs help from science and engineering in the worst way. But the help it needs is not simple to provide.

The bulk of managers and workers in data processing have moved into the field laterally, without university education in computer science or software engineering. The principal sources of information for these people are hardware manufacturers and trade publications. Professional societies and journals play an indirect role in transmitting scientific and engineering ideas to this community. But there is little help to be found in trade publications of the industry, because the problems are of a deep and systematic nature, of unprecedented logical complexity, and so patent remedies are of little value.

In the early days of computing, many very good people from the nation's great universities and laboratories moved into the field and brought with them or developed many sound and enduring ideas. With the growth of data processing, the bulk of people coming into the field later were less highly motivated and educated. As a result, data processing has expanded in quantity but has decreased in average quality, in personnel, over its short history.

This lag in industry practices is of particular concern when one is endeavoring to identify research needs. Research efforts take years of educational preparation and then deep concentration, often over decades. But in data processing, with only a 25-year history, industry needs are just beginning to emerge. What seemed important 10 or 15 years ago may be less important now because of this changing set of needs. Thus it is all the more critical to properly assess future trends in data-processing practices, in order to anticipate research areas of more lasting relevance.

International competition in data processing. Data processing has come to be an important national asset in the management and organization of industrial resources in all well-developed countries. The value of this data processing depends on the quality of software found in industry and government. For example, if poor or unreliable software in a grocery chain contributes to the spoilage of foodstuffs shipped to the wrong place, the energy required to grow the food is wasted. Thus, the cumulative effect of software quality has a significant impact on the relative industrial (and social) health of each country. If this is so in the first 25 years, it will be even more the case over the next 25 years.

Because of this importance of data processing in national affairs, the following incident in space activities teaches us a lesson. In the late 1950's the United States was behind the U.S.S.R. in missile booster technology (because it was ahead in the miniaturization of atomic bombs), and this fact was a cause of considerable consternation in the United States. In retrospect, being behind in

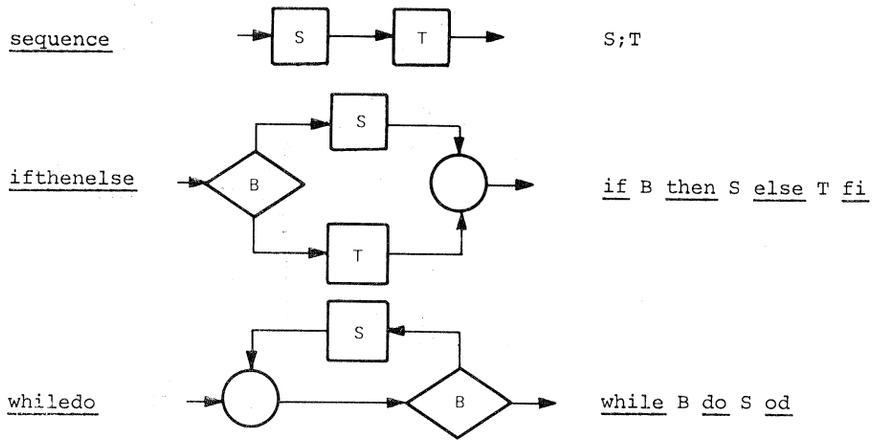

Fig. 1. A set of action primitive.

booster technology did the U.S. electronics industry a real service, because it imposed a discipline for miniaturization that would otherwise have been difficult to enforce. In data processing the situation in the two countries is reversed. For the last 20 years the United States has maintained a commanding lead in computer power, computers being scarce resources in the U.S.S.R. The result is that the U.S.S.R. now has fewer, but more highly disciplined programmers than the United States.

It is already clear that Japan will be an important international force in data processing. In addition to Japan's already proven ability in electronics production, it has two specific advantages for software development in the future (which are also shared by the U.S.S.R.). First, the Japanese government has the means and apparent resolve to organize and coordinate software research and development as an instrument of national policy, as evidenced by its formation of a Joint System Development Corporation (*6*). Second, Japanese industrial and labor practices will permit a smooth transition into the automation of industrial processes, by comparison with the case in the United States where automation is often regarded as a threat to jobs and labor and subject to political and social delay and disruption.

The Emerging Idea of Software Engineering

Software tools—technical proving grounds. In spite of the complexity barrier, data-processing systems of large size are in widespread operation. In most cases they have evolved over decades or more into their present forms, and they usually represent more perspiration than inspiration in their present forms. The major tools for their development and maintenance are compilers (and assemblers) for programming languages and operating systems (often with extensive data management facilities). The development of these tools has been a major basis for technical development in programming itself. Programmers have a unique opportunity to build their own tools. But, with only a few exceptions, the tools themselves have provided more experience in what to avoid next time than as examples of well-designed and documented software.

The development of compilers presents deep and interesting problems in programming. It involves text and language processing, which has motivated considerable research in formal grammars and their languages, and studies in finite state machines. The compiling process calls for the management of extemporaneous data storage and for the accessing of data in flexible ways. Usually, the problems of programming compilers are more difficult than the programming problems solved in the languages they compile; for example, a programming language with very simple data storage facilities will need a compiler which itself requires much more flexible data storage.

The idea of an operating system has evolved dramatically over the past two decades, both to adapt to new hardware capabilities and to respond to new human demands. Human interfaces have expanded from a single operator entering and reading binary coded numbers at a primitive console to hundreds of users in simultaneous conversation with friendly terminals. Operating systems deal with multiple processors and multiple programs in concurrent execution in these various processors, accepting spontaneous, asynchronous signals and data between themselves and from outside, automatically shuttling programs and data to and from high-speed and lower-speed storage devices between bursts of execution. The development of software to supervise this complex asynchronous activity has been a second major challenge in software development and a source of much research in asynchronous program control, data management, and large system decomposition techniques. As in the case of compilers, the applications systems which run in these operating systems are usually of simpler construction, with less demanding logical and performance requirements, than the operating systems.

In short, compilers and operating systems have served as important proving grounds for programming techniques. The ideas emerging from these proving grounds invariably find use in applications systems. For example, the use of formal grammars and languages, inspired by compiler problems, is now widely recognized in the design of applications interfaces with users; techniques of large system decomposition originating in operating system designs are now seen to be useful in applications systems as well.

Three software discoveries. The first computers were single sequential machines which were operated under program control. But, curiously enough, before people learned how to program these simple machines adequately, new hardware was invented which was capable of cooperative asynchronous operation, and so people immediately set out to program the new hardware before they had time to master the old. In retrospect, it became necessary, mathematically, to retrace these steps—to study the programming of sequential processes and then the programming of the harmonious cooperation of independent sequential processes. In the past decade, this study has led to three significant software discoveries: (i) the dependable design of correct sequential processes; (ii) the synchronization of independent sequential processes for harmonious cooperation; and (iii) the organization of software-hardware into systems of abstract machines.

Dijkstra has played a leading role in bringing all three of these discoveries to light. In the first place, he introduced the idea of "structured programming" (*7*) as a systematic method of design for sequential processes, based on the need for mathematical proofs for their correctness. In order to cope with the scale-up in size, in order to keep proof size proportional to program size, he found it necessary (and, surprisingly, possible) to restrict the freedom of design to a small,

finite set of primitive sequential processes. This restriction was described in a famous letter to the editor entitled "GOTO statements considered harmful" (*8*) in 1968, which initiated considerable controversy but which time and experience have pretty well settled.

Structured programming. In spite of the controversy over GOTO's, the essence of structured programming is not the absence of GOTO's in programs but the presence of rigor in programming. This rigor is achieved by a so-called stepwise refinement process, which expands an action specification into a primitive sequential process of (smaller) action specifications, carried out repeatedly until actions of the programming language are at hand. Thus, beginning with an initial action specification for a program as a whole, the program is designed, stepwise, in a hierarchy of refinements.

A set of such primitives, given in flow chart and text form, is shown in Fig. 1. where S, T are actions (which may be further expanded in these same primitives) and B is a (Boolean) test. In illustration, after several expansions, a sequential process might be defined by the program, in flowchart and text, shown in Fig. 2, where the text form has been broken into lines and indented to show the structure of the expansions. It should be added that the foregoing gives the form but not the substance of good design. A good design not only conforms but satisfies many additional criteria not easily stated here. But the form does keep the correctness proof size proportional to the program size.

The principal reason for the controversy about structured programming was the question of whether such a small set of simple primitives was adequate to achieve sequential control. At the time, there was little discipline in such control logic, because it was widely supposed that complexity was a necessity of good design. We now know better.

Structured programming has already found important practical applications in software development, as discussed by Baker (*9*). There is already a widespread acceptance of the idea in industry and government, even though practice often lags behind intentions.

An axiomatic basis for sequential processes. Hoare (*10*) provided an explicit confirmation of Dijkstra's objectives in structured programming by providing an axiomatic basis for proving the correctness of each of the primitive sequential processes proposed by Dijkstra. Thus, at each stepwise refinement, a standard procedure can be invoked, at

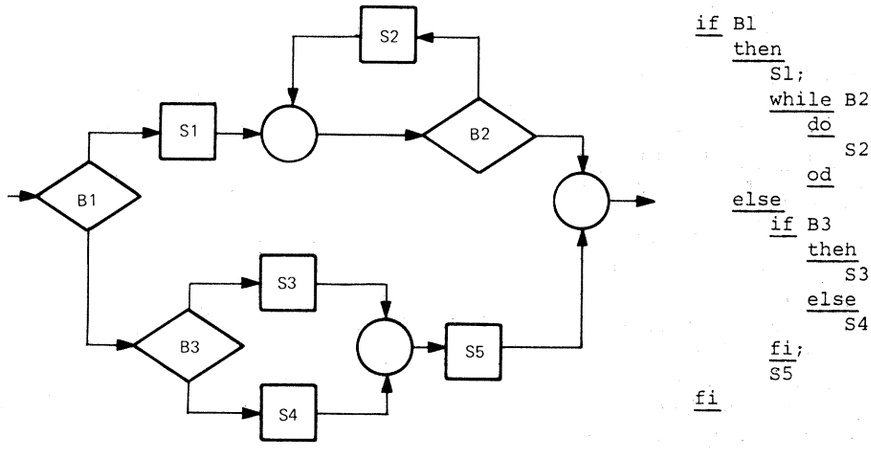

Fig. 2. A simple structured program.

any level of formality desired or appropriate. Specifically, Hoare introduced the idea of a "precondition" P, and "postcondition" Q for an action S, written $P\ \{S\}\ Q$, and interpreted "if P holds and sequential action S is carried out (and terminates), then Q will hold." Now, the primitive sequential processes above can be axiomatized as follows:

sequence
$$P\ \{S\}\ Q \wedge Q\ \{T\}\ R \Rightarrow P\ \{S{+}T\}\ R$$

ifthenelse
$$P \wedge B\ \{S\}\ R \wedge P \wedge \neg B\ \{T\}\ R \Rightarrow P\ \{\textit{if } B \textit{ then } S \textit{ else } T \textit{ fi}\}\ R$$

whiledo
$$P \wedge B\ \{S\}\ P \Rightarrow P\ \{\textit{while } B \textit{ do } S \textit{ od}\}\ P \wedge \neg B$$

The last axiom, for the *whiledo*, is especially illuminating, because it identifies a condition P, called an "invariant" (by reason of the persistence of P in the antecedent), which can be used to prove looping programs correct. It is now easy to see that Dijkstra's claim for the proof sizes of structured programs is correct—the form of the proof is independent of the depth of the hierarchy at each refinement step. The practical possibilities of proof of correctness have led in turn to programming language ideas which describe not only code but proof, as shown by Wegbreit (*11*).

More recently, Dijkstra (*12*) has used the foregoing ideas of Hoare to introduce the concept of "predicate transformers" for sequential process design. In brief, for each Q, S defines a "weakest precondition" P, such that $P\ \{S\}\ Q$, so that S can be interpreted as a predicate transformer (which transforms Q into P). Such a weakest precondition always exists, because, regarding P, Q as predicate valued actions, the sequence $S;Q$ is, in fact, the weakest precondition P for which $P\ \{S\}\ Q$. Now, the condition required of a sequential process must be known in order to design it, but this is postcondition Q. So the sequential action S which transforms Q into a suitable predicate P can be determined as a stepwise refinement. Dijkstra showed how each primitive sequential process behaves as a predicate transformer and thus laid the groundwork for a method for deriving programs which are automatically correct (*12*). For example, denoting the predicate transformer *wp* (weakest precondition),

$$P = wp(S, Q)$$

then the transformer for the sequence $S;T$ is

$$wp(\text{"}S;T\text{"}, Q) = wp(S, wp(T, Q))$$

Gries (*13*) has given an elementary exposition of the use of predicate transformers.

Structured programs and function algebras. A sequential process S defines a "program function"

$$s = \{(x, y) \mid y = S(x)\}$$

and a Boolean test B defines a predicate function

$$b = \{(x, y) \mid y = B(x)\}$$

The primitive sequential processes axiomatized above can be identified with operations in an algebra of functions, namely,

sequence operation
$$s;t = \{(x, y) \mid \exists z\ (z = s(x) \wedge y = t(z))\}$$

ifthenelse operation
$$\textit{if } b \textit{ then } s \textit{ else } t \textit{ fi} = \{(x, y) \mid (b(x) \wedge y = s(x)) \vee (\neg b(x) \wedge y = t(x))\}$$

whiledo operation
$$\textit{while } b \textit{ do } s \textit{ od} = \{(x, y) \mid \exists k \geq 0\ (\forall j, 0 \leq j < k,\ (b(x^j(x))) \wedge y = s^k(x) \wedge \neg b(y))\}$$

(in the *whiledo*, k is an "iteration count," b must test true up to $k - 1$, y must be the result of k iterations of s, and

b must test false to terminate). Here, the sequential control symbols; *if, then, . . . , od* are *infix*, *prefix*, *postfix* operators (just like +, −, and other symbols in ordinary algebras), with functions as operands and results. A structured program corresponds to a compound expression in such an algebra of functions, and the various convenient properties of hierarchical structure, indentation capabilities, and subprogram nesting are all seen to be a result of this algebraic structure.

The stepwise refinement process can be formulated as solving function equations, one of

$$r = s;t$$
$$r = \textit{if } b \textit{ then } s \textit{ else } t \textit{ fi}$$
$$r = \textit{while } b \textit{ do } s \textit{ od}$$

in which r is given and b, s, t are to be determined. Closed-form solutions can be given for each of these equations, as shown by Mills (*14*). For example, the solution for the *whiledo* equation has a classical mathematical form—an existence condition (on r), and a single parameter family of solutions (with a function as the parameter). Reynolds and Yeh (*15*) give another view of programs which compute functions by induction.

Process synchronization. Each hardware device in a computing system operates asynchronously with every other unit, so that signaling and waiting are practical necessities in their coordination, that is, their harmonious cooperation as sequential processors. In this way card readers, printers, tape units, and logical processors can all cooperate in a single coherent operation, even though each device operates at quite different and mutually unpredictable rates. For example, a central computer can request action from a card reader, continue some calculations, and be interrupted when the card reader has completed its operation.

Dijkstra showed how the necessity of programming asynchronous processors could be converted into a powerful software design technique, by organizing an entire system as a "society of sequential processes, progressing with undefined speed ratios" (*16*, p. 343). Such sequential processors are not tied to specific hardware devices (although each asynchronously operating device defines one or more sequential processes in a natural way) but are used to organize (divide and conquer) more complex processing activity into a set of simpler ones which are coupled through synchronizing primitives called "semaphores." Semaphores play two distinct roles: (i) the mutual exclusion of sequential processes when any one of them is performing a critical task, for example, updating a commonly used record, and (ii) providing synchronization signals to one another. The use of semaphores permits proof of cooperation among sequential processes, such as guaranteed progress (absence of deadlock) to goals and integrity of data common to the processes, as shown by Dijkstra (*17*), Habermann (*18*), and Hoare (*19*).

Brinch Hansen (*20*) and Hoare (*21*) have further developed the synchronization of sequential processes through program units called "monitors," which led to methods for extending Hoare's axiomatic treatment of sequential processes to axioms for the synchronization and exclusion of cooperating processes. Howard (*22*) and Owicki and Gries (*23*) have expanded on this treatment.

Abstract machines. The organization of a sequential process into a hierarchical structure through stepwise refinement is a powerful abstraction. The additional synchronization of several sequential processes into a single coherent superprocess is even more powerful and brings the ability to organize any system of people and hardware into harmonious cooperation. First, however, let us consider only hardware, in order to develop the idea more simply.

A number of investigators, beginning with Parnas (*24*), Liskov and Zilles (*25*), and Wulf *et al.* (*26*), have observed that the organization of software systems can be improved in clarity and reliability if certain data and programs are organized as "data abstractions," namely, as software-defined finite state machines, which are "black boxes" to their users. For example, such an abstraction could behave as a first in, first out queue through a set of external commands, with the internal organization of the queue hidden from the user. An advantage of this arrangement is that the queuing mechanism can be entirely changed without affecting the rest of the programs that make use of it. Another useful and common abstraction is a card reader, a software-hardware machine which operates asynchronously with a using process.

In fact, I will use the term "abstract machine" to mean any software-hardware–defined finite state machine. The implementation of an abstract machine is done in only one way—by means of a society of cooperating sequential processes, each of which may employ abstract machines in turn. Now it is easy to see that the recursive use of abstract machines allows us to explain and design the largest of software systems. But, first, there is a lesson in hardware machines.

A well-designed and documented hardware machine has the following properties, as noted by Habermann *et al.* (*27*) in general: (i) a full specification of its legal (meaningful) operations, (ii) the detection of illegal requests for operations and a description of the response taken to them, and (iii) the guarantee that illegal requests do not damage the machine, that is, do not prevent the machine from operating correctly [as defined by properties (i) and (ii)] if restarted. Furthermore, even though the timing specifications of operations may be only approximate (as a result, for example, of mechanical variances), at the completion of any logical operation the new state of the machine should be specified precisely. We should expect no less from an abstract machine.

The idea of abstract machines makes possible the development of hierarchies and levels of abstractions in software design. A software system can be designed as a set of harmoniously cooperating sequential processes, which have powerful abstract machines at their disposal. Each of these abstract machines can in turn be designed as a new set of harmoniously cooperating sequential processes, with simpler abstract machines being used until hardware machines are reached. For example, an airlines reservation software-hardware system can be designed as a cooperating set of abstract terminals, an abstract control program, and an abstract data management system. An abstract terminal, in turn, can be designed as a combination of a cooperating abstract keyboard, an abstract display, and abstract local data. This whole software-hardware system then becomes itself an abstract machine for a system of people and machines that can deal with the public.

References

1. E. W. Dijkstra, *On a Methodology of Design* (MC-25 Informatica Symposium, Mathematical Centre Tracts, Amsterdam, 1971).
2. B. W. Boehm, *IEEE Trans. Comput.* **C-25** (No. 12), 1226 (1976).
3. H. D. Mills, *IEEE Trans. Software Eng.* **SE-2** (No. 4), 265 (1976).
4. F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering* (Addison-Wesley, Reading, Mass., 1975).
5. H. D. Mills, *On the Development of Systems of People and Machines* (Lecture Notes in Computer Science: 23, Programming Methodology, Springer-Verlag, Berlin, 1975).
6. E. K. Yasaki and A. Pantages, *Datamation* **22** (No. 9), 91 (1976).
7. O. J. Dahl *et al., Structured Programming* (Academic Press, New York, 1972).
8. E. W. Dijkstra, *Commun. Assoc. Comput. Mach.* **11** (No. 3), 147 (1968).
9. F. T. Baker, *Assoc. Comput. Mach. SIGPLAN Notic.* **10**, 172 (June 1975).

10. C. A. R. Hoare, *Commun. Assoc. Comput. Mach.* **12** (No. 10), 576 (1970).
11. B. Wegbreit, *IEEE Trans. Software Eng.*, in press.
12. E. W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, N.J., 1976).
13. D. Gries, *IEEE Trans. Software Eng.* **SE-2** (No. 4), 238 (1976).
14. H. D. Mills, *Commun. Assoc. Comput. Mach.* **18** (No. 1), 43 (1975).
15. C. Reynolds and R. T. Yeh, *IEEE Trans. Software Eng.* **SE-2** (No. 4), 244 (1976).
16. E. W. Dijkstra, *Commun. Assoc. Comput. Mach.* **11** (No. 5), 341 (1968).
17. ———, in *Programming Languages*, F. Genuys, Ed. (Academic Press, New York, 1968).
18. A. N. Habermann, *Commun. Assoc. Comput. Mach.* **15** (No. 3), 171 (1972).
19. C. A. R. Hoare, in *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perott, Eds. (Academic Press, New York, 1973).
20. P. Brinch Hansen, *Assoc. Comput. Mach. Comput. Surv.* **4** (No. 4), 223 (1973).
21. C. A. R. Hoare, *Commun. Assoc. Comput. Mach.* **17** (No. 10), 548 (1974); *ibid.* **18** (No. 2), 95 (1975).
22. J. H. Howard, *ibid.* **19** (No. 5), 273 (1976).
23. S. Owicki and D. Gries, *ibid.*, p. 279.
24. D. L. Parnas, *ibid.* **15** (No. 5), 330 (1972).
25. B. H. Liskov and S. Zilles, *IEEE Trans. Software Eng.* **SE-1** (No. 1), 7 (1975).
26. W. A. Wulf, R. L. London, M. Shaw, *ibid.* **SE-2** (No. 4), 253 (1976).
27. A. N. Habermann, L. Flon, L. Cooprider, *Commun. Assoc. Comput. Mach.* **19** (No. 5), 266 (1976).

Human Performance Considerations in Complex Systems

H. O. Holt and F. L. Stevenson

Acceptable human performance in complex systems depends upon precise human-machine interaction. Such interaction is the focus of attention of design engineers and computer programmers, for example, on the one hand, and of human performance psychologists on the other. The meaning and extent of that interaction has evolved and expanded over the years. We now commonly find computers of various sizes on the machine side of the human-machine interface, and their presence has changed human performance considerations markedly. On the human side we have seen an accelerating emphasis upon man as an information processor, thus adding many considerations to the older—but persistent—anthropomorphic concerns.

In this article we review the "human factors engineering" field briefly, and then discuss in some detail the requirements that computers have put on people and human performance technology in computer-based systems. For examples, and in the citation of solutions, we draw heavily upon our own experience in the Bell system.

There have been human-machine interaction concerns of a sort ever since primitive man first extended his own abilities with simple weapons and tools. In more recent history the industrial revolution accelerated greatly the transfer of work functions from people to machines and complicated the human-machine interface problems considerably.

Dr. Holt is director of the Human Performance and Support Center, and Mr. Stevenson is head of the Systems Training Department at Bell Laboratories, Piscataway, New Jersey 08854.

Human Factors Engineering

It is generally agreed that World War II marked the beginning of a professional approach to what came to be called human factors engineering, that is, a systematic approach to studying problems of human-machine interaction and to arriving at practical solutions on a scientific basis. Before the war, going back into the late 19th century, systematic work had been done by psychologists, but it tended to focus upon selecting or training people to interact with machines. But the tremendous industrial and military expansion brought on by World War II, and the greatly increased complexity of the weapons systems being produced, complicated human-machine interaction considerably, so that the selection and training approach no longer was sufficient. For example, it was a simple matter to get a relatively small number of men to fly the slow uncomplicated fighter aircraft of World War I as compared to getting thousands of men to perform satisfactorily in the high performance P-38's and P-51's of World War II. Thus, a recognized professional specialty, usually known as human factors psychology or human engineering, was spawned.

Since human factors psychology came into being during World War II, it is to be expected that it would continue to thrive in a military environment after the end of that war. The Army, Navy, and Air Force all established substantial centers for the study and application of this discipline, and many of them still exist today. Human factors practitioners spread into the industries that supplied military equipment and systems. There was a particularly large concentration in the aerospace industry. The movement also took hold in many nonmilitary fields such as transportation, telephony, and occupational safety.

In 1953, Paul Fitts and others typified the work being done in the 1940's and 1950's as follows: "In the design of equipment, human engineering places major emphasis upon efficiency as measured by speed and accuracy of human performance in the use of the equipment. Allied with efficiency are the safety and comfort of the operator. The successful design of equipment for human use requires consideration of the man's basic characteristics, among them his sensory capacities, his muscular strength and coordination, his body dimensions, his perception and judgment, his native skills, his capacity for learning new skills, his optimum work load, and his basic requirements for comfort, safety, and freedom from environment stress" (*1*).

Thus traditional human factors engineering concerns itself with data gathering and experimentation meant to yield precise information about human capabilities. With such information, machines can be built to fit humans. For example, studies revealed design requirements for visual displays of understandable information so that correct decisions or control actions can be made without delay. Comfortable physical fit between man and machine can be established with the use of such information as the average human's physical size, strength, and reach. This information has been stored in handbooks for the use of equipment designers and for human factors personnel who work with equipment designers.