Software

Glenn Bacon

Summary. Two principal themes are observed in software development, both aimed at improving the productivity of developing and maintaining new applications. The first is to provide increasingly rich system programming function in order to handle the details of managing hardware resources. The second is to provide application development facilities with logical structures and building blocks more closely aligned with the logic of the application itself. An additional challenge is to provide these in a way that will allow continued enhancement of existing software.

Progress in software is inevitably compared with that in hardware. Usually, software does not fare well. Impressive curves can be drawn showing the dramatic improvement in processor speed or storage capacity. In contrast, the function provided by software not only defies simple numeric characterization, it also does not even have, at this point, a useful taxonomy. Thus there are not even practical terms in which to measure progress. Further, for most computer users the cost of hardware is declining while that of software is increasing. The declining hardware costs allow new applications to become economical, but almost certainly new applications require new and more complex software. Many fear that the availability of the latter may be the gating factor in the growth of the computer industry.

Measurement of software against hardware is ultimately misleading, since they have distinctly different attributes. First, the logical complexity of the software typically far exceeds that of the hardware in most systems. In a large machine, performing commercial applications, programs comprising a total of more than 10^7 lines of code must work in tight relationship with each other. This number far exceeds the number of circuits in the central processing unit. While the hardware must execute the movement of data to and from external storage, the software is responsible for the placement and logical structure of those data. Again, in a large system it is not unusual to find the software responsible for more than 10^{12} bits of information. By any measure, the design and management of large software systems are among the most complex tasks ever undertaken.

A second distinction between hardware and software is that the latter tends to accumulate over time, while the former is periodically totally replaced by improved technology. Particularly in commercial data processing applications, many of the most economically valuable tasks were programmed in the 1950's and 1960's. Over the years, these programs have been significantly enhanced and continuously maintained, but it is usually the exception that they have been totally rewritten in modern, structured, and well-documented approaches. The decreases in cost and gains in performance of hardware have made this tactic pragmatic, since hardware capabilities have been able to keep up with the throughput requirement growth of the applications. Thus it is not uncommon to find such critical tasks implemented in old, poorly structured, and difficult to maintain code. The total replacement value of these programs is in the tens of billions of dollars (*1*). But more important, the continuing maintenance and restructuring that they require consume a significant programming resource.

The third critical difference between hardware and software is that the latter has a much more detailed involvement in the work of the person or institution that it serves. The effects of hardware are largely confined to the data processing department or to a terminal or personal computer on the desk of the user. The logic of the software, on the other hand, is intimately involved in the procedures and work flow of the user. Early software was focused on procedures that were already systematized. These procedures included conventional business and management applications as well as scientific applications for which computational algorithms had been established. The easy applications have been programmed. Contemporary software work, consequently, usually involves the establishment of new tasks or procedures that did not previously exist. Thus, much of the work associated with designing the software is in reality systematizing and designing the task that the user requires. This must be done before the software can be designed. As I will discuss later, the technology for defining the requirements for a software system is an area in most urgent need of improvement and itself constitutes a major portion of the so-called software bottleneck. The technology for designing and implementing software, once the requirements are well defined, is in far better shape.

It is these three attributes—complexity, accumulation, and the need for systematization—that give software its unique and sometimes unenviable characteristics.

Status

Software has traditionally been categorized as either system or application. System programming is normally provided by hardware manufacturers or software houses for the purpose of making application writing and execution easier. Thus a useful way to examine the history and status of software is to focus on the growing role of system software and the increasing productivity in the task of constructing applications.

Figure 1 shows the principal forces in the changing role of system programming. The expanding "horn" represents the increasing variety of function offered as one moves from the hardware instruction set toward the end-user set. The boundary below the system software, defined by hardware and microcode, has moved up as hardware technology has improved. In early computers the function provided by the hardware was primitive, and the system software had many low-level responsibilities. For example, several early machines had working memories involving delay lines or magnetic drums, a slow-access serial working memory. The performance of programs could vary drastically (or catastrophically), depending on how long the

The author is director of the Santa Teresa Laboratory, International Business Machines Corporation, San Jose, California 95150.

0036-8075/82/0212-0775$01.00/0

processing unit would wait for the next logical instruction to arrive at the end of the delay line or at the drum reading head so that it could be accessed and executed. Thus the proper positioning of instructions throughout the memory required a complex assignment based on an estimation of the execution time between instructions and possible branching effects. System programming managed that detail for the user.

The principal software component for managing hardware detail is the operating system. These systems began to emerge in the 1950's but were not in broad application until the early 1960's. Operating system software was written to manage the details of moving data to and from storage or into and out of telecommunications lines and other peripheral devices. It was extended to take the responsibility for scheduling work and allocating access to critical components such as multiple processing units and memory. Much complexity was still presented to the application programmer in early operating systems because the cost of hardware required user decisions to achieve its best utilization. Simplification is achieved through increasingly more sophisticated system software and microcode that would previously have been uneconomical to execute. A design such as the IBM System/38, which essentially eliminates the task of managing storage, exemplifies this direction.

Since users of large systems cannot accommodate major changes in the programming that executes their applications, continuous evolutionary restructuring of operating systems is one of the major facts of the software environment. The evolutionary step to achieve this might be characterized as a maintenance step and might be lamented as unproductive. Such a characterization would miss the point of one of the major challenges before us. Hardware advances and system growth will continue to invalidate initial software design trade-offs. We must therefore work to improve our effectiveness in the required restructuring work.

Application Development Productivity

The other direction of growth for system software is toward the user rather than the hardware. In this case, the work is associated with the logic of the application rather than the physical characteristics of the hardware. This corresponds to the movement of the upper boundary in Fig. 1. The principal carrier of this theme has been the area of programming language. The earliest assistance in this area was provided by assemblers, which took over the details of assigning instructions and data to specific hardware memory locations, thus allowing the program writer to use symbolic terms for these elements. This was followed in the 1950's by the emergence of higher level languages. These not only incorporated the capabilities of assemblers but, more important, greatly increased the logical power of the instruction set for the application writer. Operations involving complex algebraic expressions could then be expressed as a single language instruction. The language software would then "compile" each such instruction to a larger number of machine language instructions.

Many such languages were developed in the late 1950's and 1960's, with their command sets styled to suit particular areas of application in business, text handling, and scientific computation. Several attempts were made to design "universal" languages in the hope that the proliferation might be ended. These efforts have yielded popular, but far from universal, languages such as PL/I, ALGOL, and APL. Claims to universality are not often made at this time. Further, there is growing argument that languages with too rich a structure, which attempt to do too much, may be overly complex and thus error-prone (*2*).

There seems to be substantial evidence that program writers are reluctant to change from a language in which they have become productive even though new languages may be demonstrably superior. The strongest evidence for this is that the preponderance of code being written today is in the old languages of COBOL and FORTRAN. Thus the rate of change of language usage may be more appropriately measured in generations of people than in generations of software development.

Database software provides logical capabilities that are comparable in importance to languages. Especially in commercial applications, the interrelationship among data elements can become extremely complex. These data elements and their interrelationships represent a model of the business or technical process that the computer is assisting in managing. Given the continuous change of the data elements and their interrelationships as external reality changes, the management of data constitutes a significant portion of the entire programming task. It is the job of database software to remove this work from the application program and present to that program consistent and well-maintained views of the data. In addition, these systems share with the operating system the management of the logistics associated with recovering the database from hardware and software failures and the management of both on-line and off-line storage space.

As with languages, there are many logical approaches to the structuring of data. Again, the search for universals has not been productive, since different types of usage favor different logical interfaces. This area itself has become one of the principal elements of computer science (*3*) and is summarized by Blasgen (*4*) in this issue.

If one is willing to specialize the types of application programs that may be written, programming systems with significantly higher logical levels can be provided. An example of these facilities is the class of application generators. An early application generator was the report program generator (RPG) language, which provides powerful and easy-to-use capabilities for producing simple business reports. Some of these languages now provide facilities that greatly aid the construction of terminal screen formats and writing of complex commercial transactions. They also simplify application maintenance by helping to isolate changing data and business procedures from the overall program control structure. Claims of productivity improvements exceeding a factor of 10 over conventional languages such as COBOL are repeatedly made.

"What" versus "How"

Beyond such systems are the nonprocedural query languages. These allow a user with little programming knowledge to access complex data structures and, further, to request relationships among those data which were not previously structured by a database administrator. These best exemplify the theme of removing work from the application writer by allowing him to state "what" is wanted rather than to specify "how" the computer is to achieve it. In this case, the goal is to eliminate the intermediary application programmer entirely.

In terms of eliminating bottlenecks, the major challenge facing software developers is that of providing powerful computing facilities for the general public. Most people are not at all interested in mastering the arcane discipline of computer programming. Not only does the what as contrasted to the how approach seem necessary to serve them, but the tasks that the computer is to undertake must be described to the user in terms which are immediately meaningful in his environment—not those which

are simplest for the computer. A system such as the Xerox STAR professional work station is an example of such a direction. The work elements (such as letters and files) and the tasks (such as creating, routing, and storing) are presented to the user in pictorial form. The logic of the user's application can be largely achieved by pointing at the tasks and the data elements in order to properly interrelate them. It is expected that systems of this sort will emerge in an increasing variety of specialized end-user environments.

This trend is still another manifestation of the lack of a single universal approach to programming computers. Since the work that computers do will be driven by the virtually uncountable number of tasks that humans may want to undertake, one begins to see why it is so difficult to develop a taxonomy of computer software at any but the most primitive logical levels. The top of the horn in Fig. 1 will expand indefinitely.

It might be necessary first to achieve a taxonomy of human tasks, an endeavor that has not yet yielded substantial results. Practical results are emerging in business data processing, however. Carlson and Kerner (*5*) have demonstrated the practical application of Donald Berstine's business information analysis and integration technique (BIAT). Bernstine has shown that a large class of applications can be mapped into a useful structure by the yes/no answers to seven questions. Questions such as "Is the product paid for at delivery or billed later?" or "Is the product rented or purchased?" yield a structure for the required data system.

Without a more general approach, the specification of user applications is still very much in the realm of art and invention. However, once that design requirement is stated, an increasingly powerful base of software engineering technology is available to aid in its implementation.

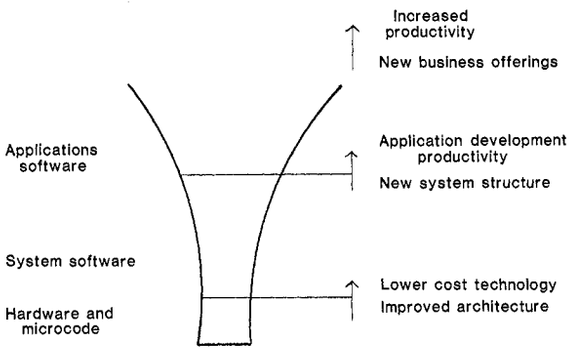

Fig. 1. Growth of programming function.

Software Production

Most computer users find it far more economical to purchase or lease software packages that are already written for their applications than to develop them in-house. At this time, there are more than 6000 products commercially available for minicomputers and larger systems, along with an unknown but rapidly increasing number of offerings for personal computers. Despite this large inventory, most medium and larger facilities also do a great deal of custom programming for unique applications or continuing maintenance of old code. Thus, both for a very large number of computer users and for the producers of commercial packages themselves, the technology for improved software production is of critical importance.

Like any process that has undergone systematic refinement, software production has been divided into increasingly specialized and differentiated tasks. Programming in the 1950's involved little planning. At most, a flow chart outlining the major logical structure was prepared. One then began coding through the flow chart, designing and debugging each of the principal paths. Since machine resources were expensive, good code was regarded as that which was most tightly written, and cleverness in exploiting every last logical element of the instruction set was clearly to be admired. While many of the programs written in this fashion achieved their objectives, fundamental problems soon became apparent:

1) Such programs were difficult to handle by more than one person. Essentially, there was no approach to partitioning the task so that several people could cooperate and still maintain overall logical cohesion.

2) The overall task that the program was to achieve was usually understood only by the programmer. If the user was to be another person or group, there was often much frustration and disagreement over what was to be achieved due to lack of precision in stating the requirement.

3) Probably the most critical problem was that the program had poor structure; logically, it resembled a bowl of spaghetti as contrasted to a set of building blocks with well-defined interfaces. If the program was to be enhanced or maintained by others, this attribute was most harmful since small changes in the code could have significant and unpredictable effects as they propagated throughout the tightly coupled structure.

The limitations of this approach to programming became graphically clear when large projects were undertaken which contained many programs interacting with each other. Although there were many such efforts in the early 1960's, the most notable was the development of OS/360, the operating system for the IBM System/360 computers. This development, which has been chronicled by Brooks (*6*), clearly demonstrated the necessity for design and management cohesion in large projects. The principal lesson was that the tasks of requirements definition and design should be clearly separated from coding. For management, the difficult lesson was that adding more people to the latter stages of a project that is falling behind is likely to add further delay because of their lack of knowledge of design detail.

By the late 1960's, work in mathematics and computer science began to produce practical guidance for programming. Dijkstra (*7*) stressed the importance of the program as a medium of communication. The flow of the program during its execution should closely match the structure of the application task. Not only would this aid others who must understand the intent behind the program, it would also guide the designer in accurately and completely representing the application. Dijkstra identified program constructs that would facilitate achieving this similarity in structure between programming and the requirement. He further argued that indiscriminate use of the popular GOTO instruction, which would cause unexplained breaks in the program flow, was counter to this objective.

Hoare (*8*), following the work of Floyd (*9*), provided complementary guidance by demonstrating properties of programming languages that help eliminate common coding errors. These allow the language to enforce assertions about the range and other properties of variables and provide a formal approach to mathematically verifying that the program matches the intent of the designer. Languages such as PASCAL and the language ADA proposed by the U.S. Department of Defense have incorporated much of this thinking.

These foundations were soon supple-

mented by further design and management techniques to form the field of software engineering and the particular discipline of structured programming (*10*). It has become common practice to partition large programs into functional modules (*11*) with a disciplined interface structure between them, along with techniques to formally review designs and inspect code (*12*). Most established programming organizations have incorporated these practices through programming process guidelines (*13*).

An additional dimension of software engineering is the programming environment. The environment is defined by the set of programming tools that aid the implementation of the process guidelines and facilitate control of the project as it moves through the stages of specification, design, coding, testing, and maintenance. These tools and the computing resource that they require are the principal form of capital investment for improving the productivity of programming. Modern operating systems such as UNIX, described by Kernighan and Morgan (*14*) in this issue, were designed with the entire environment in mind and include concepts such as the programmer's workbench (*15*), a coordinated set of tools.

A major current effort is the design of the environment associated with the ADA language. The purpose is to improve the quality and productivity of the massive amount of programming required by the Department of Defense for so-called embedded applications. The "Stoneman" proposal for an environment consisting of a database, debugging tools, code contol mechanism, and so on (*16*) has been developed essentially in parallel with the "Steelman" proposal for the language itself.

Many problems with programming still exist. This is particularly true in the area of very complex designs. Deceptively simple requirements carried through the stages of specification and detailed design can virtually explode in terms of required function. Thus reliable cost and schedule commitments cannot be made until the early stages of the design process have been completed. If those costs are unacceptable, a considerable amount of effort may have to be abandoned. Usually, there is a fundamental renegotiation of the requirement, eliminating excessively costly function. One of the most critical needs in the field is the establishment of metrics that can assess requirements and give reasonable predictions of implementation costs. Work on such metrics is still far from yielding practical results (*17*).

Research Directions

The overall goals of research in programming methodology are to improve the productivity of producing and maintaining the program and to improve the quality in terms of an error-free manifestation of the original requirement for the program. This goal has led to an increasing focus on the early stages of the programming process. It has been shown that an error detected at the beginning of the cycle may be two orders of magnitude cheaper to remove than one found in actual production (*13*). It is further a reliable generalization that quality is achieved in the design and implementation of the program; it cannot be acquired simply through debugging. Several of the key research approaches are reviewed below.

Research in programming transformation is aimed directly at these goals. A very high level language is provided for writing the specification itself with the goal of minimizing human intervention in transforming that specification into running code. Since the specification is supposed to describe what the program is to achieve, not how it is to be implemented, such work really strikes at the heart of the programming problem. As mentioned earlier, query languages have achieved this result for problems involving data access and structuring. The challenge is to incorporate increasingly large classes of work and resolve the what of the specification to the how of the procedural machine code with adequate efficiency. However, there will be significant value in achieving better languages for specification, since the resulting structure will better automate the means for verifying that the programs meet the specification. The ultimate goal, of course, is to directly compile these programs from the higher level specification language.

Irrespective of the level of the language, research will continue on the structuring of programs with the goal of more clearly separating the different types of tasks in an application. Kowalski (*18*) argues that an algorithm has the purpose of both defining the logic of what is to be computed and determining how it is to be done efficiently. It should be possible to change either of these aspects with a minimum of impact on the other. Current languages do not facilitate this.

Another exciting and potentially far-reaching approach derives from the technology of artificial intelligence. So-called knowledge-based automatic programming involves a dialogue between the programmer and the system. As the programmer begins to state what must be achieved, the system attempts to structure an overall program to meet the requirements stated. This intermediate result is presented to the programmer for further elaboration or correction. The dialogue converges when the programmer is convinced that the overall requirements are met and the system is satisfied that it has captured sufficient detail to resolve a complete program. So far, such work has been successful only for small programs (*19*), but in the long term it will attack one of the most difficult issues in programming—that of extracting precise requirements from the user. Thus, as with program transformation, useful results that are far short of complete automation of the programming process can be achieved.

A different approach emerges from an engineering paradigm. It is argued that an appropriate inventory of "standard parts" with appropriate interface disciplines between them can be the basis for composing much more complex programs. Not only might the programs be assembled much more quickly, but they could be of higher quality since each of the parts would be subject to much more rigorous verification and testing. While this approach has been successful in some areas of application programming, significant results have not yet been achieved in more complex areas. Two fundamental issues seem to remain. One is to determine the right set of standard parts. While it is probably not possible to prove that any set of higher level parts is optimal, a subset of commonly used functions would represent significant progress. A more critical issue is to provide a system environment in which these parts can be properly hooked together. In hardware, interfaces between components are constrained by electrical and physical considerations. In programming, however, a new discipline is needed to ensure proper interfaces between standard components; this function has been referred to as a funnel (*20*).

It is important to recognize that standard components are simply another approach to providing very high level functions such as those being pursued in program transformation research. The goal of both is to allow the programmer to construct his program in terms that more naturally fit the function associated with his problem rather than the detail required for the machine to implement that function.

Potentially the most elegant approach is that embodied in functional programming. Backus (*21*) has observed that a

great deal of the complexity of current programs results from the lack of strong algebraic properties relating the primitive functions of the programming language. He sees a need for "program forming operations" with such properties, whose domains are themselves programs. With these, a rigorous approach might be found for defining the characteristics of new programs built from combinations of existing ones. Backus observes that a principal barrier to designing languages with such strong properties is the von Neumann architecture of most existing computers. He asserts that the detailed assignment and manipulation of storage which is required for each program make it difficult to define useful program-forming operations. The particular choices in managing storage for each of the programs to be composed would probably be inconsistent.

A set of operations for composing new programs from existing ones could have profound implications for hardware design. The hardware instructions would directly implement the rules for composing programs. Further, proposed functional programming approaches offer the possibility of better determining which tasks may proceed in parallel. This would allow better use of advances in very large scale integration, which make high degrees of multiprocessing most cost-effective. While this work is still in an early stage, it is likely to lead to one of the most significant advances in computer science in the 1980's.

Conclusions

Since active research in the software engineering area was begun in the late 1960's, much has been accomplished. Given stable requirements, it will be largely a matter of skilled effort and discipline to produce a predictable and reliable result. However, as indicated by the number of different and still unproven approaches to new programming methodology, this field is still very young. Thus it is likely that a decade hence the techniques in use today will be considered ill-structured and difficult to maintain. Consequently, because of the cumulative aspect of programming, which is economically rather than technically motivated, we seem destined to have an environment of the new coexisting with the old and the very old. It is fashionable for the practitioners of the contemporary art to criticize the ignorance and lack of discipline of their predecessors. It would be more fruitful to recognize that the new must coexist with and enhance the old. Successful techniques will be those which preserve a maximum of the value of that which has already been achieved. The challenge is to become masters of the evolution.

References and Notes

1. B. W. Boehm, in *Research Directions in Software Technology*, P. Wegner, Ed. (MIT Press, Cambridge, Mass., 1979), p. 44.
2. C. A. R. Hoare, *Commun. ACM* **24** (No. 2), 75 (1981).
3. C. J. Date, *Introduction to Database Systems* (Addison-Wesley, Reading, Mass., ed. 3, 1981).
4. M. W. Blasgen, *Science* **215**, 869 (1982).
5. W. M. Carlson and D. V. Kerner, *Data Base* **10** (No. 4), 3 (1979).
6. F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering* (Addison-Wesley, Reading, Mass., 1975).
7. E. W. Dijkstra, *Commun. ACM* **11** (No. 3), 147 (1960).
8. C. A. R. Hoare, *ibid.* **12** (No. 10), 576 (1969).
9. R. W. Floyd, *Math. Aspects Comput. Sci.* **19**, 19 (1967).
10. H. D. Mills, *Science* **195**, 1199 (1977).
11. M. A. Johnson, *Principles of Program Design* (Academic Press, New York, 1975).
12. M. E. Fagen, *IBM Syst. J.* **15** (No. 3), 182 (1976).
13. H. Remus, in *Software Engineering Environments* (North-Holland, Amsterdam, 1980), p. 267.
14. B. W. Kernighan and S. P. Morgan, *Science* **215**, 779 (1982).
15. T. A. Doluta, R. C. Haight, J. R. Mashey, *Bell Syst. Tech. J.* **6**, 2177 (1978).
16. J. N. Buxton and L. E. Druffel, in *Software Engineering Environments* (North-Holland, Amsterdam, 1980), p. 319.
17. R. A. DeMillo and R. J. Lipton, in *Software Metrics*, A. Perlis *et al.*, Eds. (MIT Press, Cambridge, Mass., 1981), p. 77.
18. R. Kowalski, *Commun. ACM* **22** (No. 7), 424 (1979).
19. M. Hammer and G. Ruth, in *Research Directions in Software Technology*, P. Wegner, Ed. (MIT Press, Cambridge, Mass., 1979), p. 767.
20. M. M. Lehman, *IBM Tech. Discl. Bull.* (1976).
21. J. Backus, *Commun. ACM* **21** (No. 8), 613 (1978).

The UNIX Operating System: A Model for Software Design

Brian W. Kernighan and Samuel P. Morgan

In the narrowest sense, UNIX is a time-sharing operating system, a program that controls the resources of a computer and allocates them among users. It permits programs to be run according to some scheduling policy, controls the peripheral devices (disks, tapes, printers, and the like) connected to the machine, and manages the long-term storage of information.

The authors are members of the Computing Science Research Center, Bell Laboratories, Murray Hill, New Jersey 07974.

Time sharing implies (i) an environment in which users access the system from terminals and (ii) a scheduling rule which switches rapidly among active users, to give each a share of the processor in turn. Time sharing makes it possible for people to interact with programs as they execute them; by contrast, "batch processing" implies a regimen in which users have no such interaction with programs.

Traditionally, operating systems have been large, complicated programs requiring years of effort to create. The operating system written by IBM for its System/360 series of computers, OS/360, required more than 5000 man-years of development effort (*1*). Also, most operating systems have been batch systems, with time-sharing capabilities grafted on after the fact (although this path is not universal).

In a broader sense a system, be it UNIX or OS/360, is often taken to include not only the central kernel that controls the hardware, but also essential utilities such as compilers, editors, command languages to control the sequencing of programs, and programs for manipulating files, printing information, and accounting for usage. A system may include not only all these programs, but also general-purpose programs developed merely to be run on the system. Examples include formatters for document preparation, routines for statistical analysis, and graphics packages.

This leads to the view that an operating system is built layer on layer, rather like an onion—a metaphor that also allows for wry jokes about tears. Where

0036-8075/82/0212-0779$01.00/0