Addendum offered for “Programming Languages”

Editor:

James W. Hunt’s tutorial, “Programming Languages,” in the April 1982 issue reviewed programming language topics past and present. While Dr. Hunt gave an excellent, brief historical review and an even better analysis of the specification and design of programming languages, he omitted an important consideration in regard to languages, a pragmatic area which is characteristic-

ly known to all but never discussed in the literature. The characteristic is fundamental to the general use and acceptance of a language and has almost nothing to do with the concept, structure, or implementation of the language. The omitted characteristic is how well the language is described and explained. I include in “how well” the extent to which these descriptions are distributed.

We all know of languages of poor structure and performance which are widely used and accepted because their descriptions—whether generated by the originators of the language or others—have been clear, understandable, useful, and widely distributed. This is certainly the case with Fortran and Basic. On the other hand, we have examples of languages of impeccable quality which have languished and either died or passed into a catatonic state because their descriptions were im-

possible to understand. This is certainly the case with Algol 68 and its successors.

For two decades I was able to advise my company on the adoption of new programming languages, by restricting our adoptions to those languages for which D. D. McCracken had written a textbook. The point was that McCracken’s textbooks were clear and understandable, that a language so described could be used because people could be easily taught to understand it, and languages which were not well-described would not only be difficult to use but would ultimately die.

At the moment, the question of Ada’s continued life in the real world, that is, outside of the World of War, is in question. The formal description, “Reference Manual for the Ada Programming Language,” issued by the Department of Defense is at the normal impossible level of an anonymous language specification. None of the textbooks which have been rushed into existence are of McCracken quality. Until someone produces an easily-understood description of the language and until this easy description gets wide distribution, Ada will languish.

This is the point about computer languages which we all know and seldom mention. No matter how excellent a programming language is, if it is not clearly described and the descriptions widely distributed, the language will never make it.

Eric A. Weiss
The Sun Company
Radnor, Penn.

Small business systems: a user confronts a designer

Editor:

It is with a wry smile that I note the length and technical detail in papers on programming as exemplified by Frank Soltis’ “Design of a Small Business Data Processing System” in the September 1981 issue of Computer.

I may be mistaken, but in that paper I find only one reference to the time aspect, namely on page 78 near the top of the second column: “Frequently, the ability of a business to respond to its customers depends on the ability to access and analyze information in a timely manner.”

Wrong! *Always.*

From a practical standpoint, the major requirements of a data processing system for a small business are (1) no downtime during business hours, (2) no loss of data from power failures, hard-

ware failures, or people failures, and (3) no housekeeping chores that require nightshift or weekend work, if such is not part of the normal routine.

What happens if that read-write head slams across a disk and wipes out a few megabytes of program and data? Can it happen? Yessir.

How many hardware and software designers concern themselves with the mundane time-consuming problems of (1) sorting files, (2) creating backup files, (3) uncheckerboarding that 20-megabyte disk, and (4) making sure that their beautiful interactive multiprogram will not bomb out if the inventory, order entry, accounts receivable, accounts payable, payroll, and general
ledger people all decide on Friday afternoon or at month's end that they "gotta get their input and output before quitting time"?

And will the printers really deliver all those month-end statements, A/R and A/P printouts, and payroll checks when needed (generally right now)?

How about adding and deleting customer names from the customer file and changing the credit limit on that slow-paying customer? Jenny, over in A/R is not going to sit and wait if Ruthy has the file tied up—not without letting the boss know what she thinks of the system and that smart computer rep with the big smile.

Does all this sound like too much to hang on the computer experts? No, for the simple reasons that the business has gotten along for umteen years doing it all by hand, and it is just a question of whether the computer installation can do the same job, only faster and—the boss hopes—with fewer people.

So, when you computer experts start designing a small business system, think first about no downtime, no data loss, and no out-of-hours housekeeping. Will it do better what the people are now doing without elucidating the staff?

Good luck. You need it.

Wesley N. Lindsay
San Jose, Calif.

Author's reply:

Mr. Lindsay makes a good point that a small business system should be available when needed, should not require excessive housekeeping chores, and should not lose data. I can assure Mr. Lindsay that computer system designers are genuinely concerned with these requirements.

The areas of reliability and serviceability contribute directly to system availability and data integrity. Within IBM, all systems we develop must meet extremely stringent hardware and software reliability requirements. From a serviceability standpoint, the emphasis is on concurrent diagnosis, repair, and verification in order to maximize availability.

Power failures and device failures do occur, so some form of backup and recovery is required. Here again, the emphasis is on concurrent backup and recovery procedures to minimize user impact. In those cases where no disruption is acceptable, uninterruptable power supplies, duplication of devices, or duplication of entire systems can be used, although these may be expensive propositions for a small business.

The other cause of system unavailability Mr. Lindsay refers to is caused by time-consuming housekeeping functions such as sorting files, creating copies of files, uncheckerboarding the disks, and so on. One of the major objectives in the design of the IBM System/38 was to eliminate most of these housekeeping functions. By integrating into the system a relational data base built on a single-level storage, the need to create copies of files, sort files, or reformat disks has been eliminated. Multiple users can each have their own logical view of the same data, independent of the physical data format. Thus, multiple users can concurrently operate on the same data file, not copies of the file, so there is no longer the need for one user to wait until another user has finished with the file.

The object-oriented interface of the System/38 is also intended to ensure data integrity. Branching into a data space or adding a number to an instruction cannot occur in such a structure. Also, one user cannot access another user's data without the proper authority.

Today's system designers are concerned with several user requirements, including usability, productivity, security, and performance—as well as those mentioned by Mr. Lindsay. Although we still have more work to do, a great deal of progress has been made in recent years to ensure that a small business can, indeed, respond to its customers in a timely manner.

F. G. Soltis
IBM
Rochester, Minn.

"A Guided Tour of Program Design Methodologies"—another excursion

Editor:

Dr. Bergland's "A Guided Tour of Program Design Methodologies" (Computer, October 1981) is a rare piece of technical writing able to captivate the reader and to convey in a straightforward manner important technical issues. The article renders very accurately the basic nature of the methodologies being compared and will certainly become an important reference source for many authors. The Editorial Board deserves congratulations for deciding to publish the article.

The assessment of the four methodologies, however, is incomplete. Some of the issues the article fails to bring to the readers' attention are discussed below.

Methodologies are problem and environment dependent. Despite the fact that methodology designers and promoters tend to equate generality with success, the effectiveness of each methodology is determined by the extent to which design strategies and techniques are tuned to the specifics of some application area and, equally important, to the character of the organization involved (including the training and experience of the software designers). A comparison of methodologies requires first a precise identification of their respective domains of applicability. Next, methodologies that address overlapping application areas may be contrasted with respect to (1) differences in potential product quality, (2) differences in the quality of personnel needed to realize the same product quality, and (3) differences in product cost.

Both data flow and data structure design, for instance, are aimed primarily at a certain class of data processing applications characterized by a lack of strict performance constraints, simple input/output transformations, etc. Furthermore, they are suitable for the productivity needs of organizations whose personnel have limited formal training and/or design experience.

In contrast, functional decomposition addresses a significantly larger domain of applications and assumes a good understanding of the concept of clean abstraction and hierarchical design. This understanding cannot be obtained by reading an article or by taking a short course. Rather, it generally requires a formal education plus insight gained through application experience. Personnel with this background are harder to come by. The same arguments apply to programming calculus, with qualified personnel being even harder to find. This point was unknowingly illustrated by the author, whose application of functional decomposition in Figure 12 is flawed. By placing "card" at a higher level in the hierarchy than "group," he violated the principle of clean abstraction.

Data structures selection is critical. The selection of the data structures internal to the program is a critical design decision ignored by many methodologies today. The use of a linked structure in place of an array affects both the structure and the efficiency of the program. While the era of low-level intricate code optimization is dead, algorithmic level efficiency will continue to be of great concern for many years to come.
Constraints may not be ignored. The impact of constraints over the design process is ignored by this article as in most other discussions of program design methodologies. The methodologies are generally assessed as if the respective application areas were free of performance constraints. In fact, in many application areas (e.g., embedded systems) the constraints drive the design at least as strongly as the functionality. The design of a sort program, for instance, will be affected by space, time, and other performance constraints.

Finally, I would like to address two tangentially related issues. First, I think that the author should have made a point of distinguishing program design from system design, since techniques appropriate for one task may not be useful for the other. Although system design is outside the intended scope of the article, I am convinced that there are some in the audience that may not realize the distinction between the two.

Second, the toy examples used as illustrations in this article, and by many other authors (me included), has resulted in a counterproductive use of the notion of “pseudocode” by promoting, unintentionally, a form of pseudocode whose low level of abstraction and similarity to actual code adds nothing to a better understanding of the proposed design. Although I do not possess supportive statistical data, my personal experience with both my students and organizations where introduced software system design methodologies strongly suggests that by maintaining the pseudocode at a level where few variables are employed outside the major program data structures, one may significantly increase both the productivity and the quality of the design reviews. The use of informally stated assertions appears to have also a positive influence over the reviews.

Gruia-Catalin Roman
Washington University in St. Louis

Editor:
I have just finished reading G. D. Bergland’s article, “A Guided Tour of Program Design Methodologies.” It seems to me to be an excellent survey of the current state of affairs in methodological development. In reviewing the methods he describes, I find that data structure design and functional decomposition are both techniques which I include in my methodology. However, I also find that many of the formalisms which I have found are crucial to effec-
It seems to me that Dr. Bergland and I differ quite sharply in our understanding of the nature of a program and of the programming task. If I understand him correctly, he seems to feel that a program is similar in nature to a set of simultaneous linear equations or some other mathematical structure. From this assumption he concludes that the properties which must be found in mathematical structures should also apply to programs. (His stated preference for the "programming calculus" is natural to this view.) In contrast, I consider it clear that a program is a machine, as is an automobile or a telephone, whose components all happen to be abstract. From this, I conclude that the task of designing a program is similar in nature to that of designing a concrete machine such as a building or a car or what have you.

As an example of this assertion, consider the text-processing program which I am using to write this letter and the text processor sold as Wordstar. Both perform the functions which one expects of a text processor very well, i.e., they both accept and file one's input, format conveniently, etc. But the two programs are very different in character: one is line oriented and the other is screen (page) oriented. In other words, they both model the same problem environment well, but completely differently. The question of which is the "best" is purely one of personal taste. As an analogy, you could also consider two sports cars of the late '60's: the Alfa Romeo Duetto and the Porsche 914. The two cars "modeled the same problem" and one must say that they did so equally well: they were very close in price and performance. But the two cars are radically different in design. As with the text processors, a selection of one over the other could only be made on the basis of personal preference or taste.

I believe quite strongly that the mathematical model of programs and programming is seriously in error; one might even say "incorrect." One does not find parallels to the examples above in mathematics. As a result, concepts are imposed on programs which are not appropriate to the actual model, and which are distorting the research for sound methodology to a major extent. The concepts which attract my attention are those of "correctness" and "consistency."

Before these concepts can be discussed, I must address the question of what Dr. Bergland means by "the specification." From his usage, it appears that he means the external or functional specification. This seems reasonable, but the question of how the external is created is not addressed anywhere in the article. The majority (50-70 percent) of the design choices in a program are made in designing the external specification. Clearly, the external specification is crucial to how well or how poorly a program performs its function. And a technique, or collection of techniques, which fails to address this stage of the design process is so incomplete that I do not believe that the label "methodology" can be legitimately applied.

The concepts of "correctness" and "consistency" as Dr. Bergland describes them are crucial to mathematics. Without them the field would probably collapse. In programming I think they are, at best, misleading but more often pernicious. Even with the majority of design decisions made, there will almost always be a selection of equally appropriate design decisions available to the programmer which will accurately implement the external specification. Generally, one or a set of these decisions will result in a program which is superior in various aspects to the program or programs which would result from the alternatives. Each of these programs, regardless of merit, can be equally "correct" or "incorrect" through the programmer succeeding or failing to implement the entire external specification. Clearly, these various programs can be highly "inconsistent." Since we are dealing with machines rather than theorems, no legitimate value can be placed on "consistency."

(At this point, you might find the exercise of going out to your garage and "proving" the "correctness" and examining the "consistency" of your car or cars enlightening.)

It is more useful to consider a concept which is basic to other forms of engineering and which I have found to be vital in assessing the merit, or "goodness" of a program, and it is clear that Dr. Bergland uses this criterion in assessing a program. It is the criterion of "simplicity." In general, the simplest machine which will perform a function is the "best." It is the easiest to make. It is the most reliable. And, as long as it is used reasonably, it is the easiest to use. Most of the concepts or properties of a program mentioned in the article are really variations on the theme of simplicity, and simplification is the primary goal of structured programming as well as the source of its effectiveness.

Some of my assertions can be illustrated by directing attention to the
This tutorial addresses a broad definition of the software development environment. Thirty articles originally presented at Computing 81, are sectioned to cover goals, language-based environments, Unix, methodologies, configuration, human/computer interaction, organization and ergonomics, as well as personal systems. 476 pp.

Order #385
Tutorial—SOFTWARE DEVELOPMENT ENVIRONMENTS
Edited by Anthony I. Wasserman
1981
Members—$18.75
Nonmembers—$25.00

The 45 articles contained in this tutorial are introduced by a brief review of the history of parallel processing, a look at its spectrum of taxonomic types, and a discussion of the theoretical obstacles posed by extensive parallel processing. Topics include pipeline processors and architectures, multiprocessors, parallelism extraction, and algorithms for parallel computers. Extensive reference lists are included. 498 pp.

Order #387
TUTORIAL ON PARALLEL PROCESSING
Edited by Robert H. Kuhn and David A. Padua
1981
Members—$18.75
Nonmembers—$25.00

Discussion of the relative merits of Figures 38 through 41. Dr. Bergland says that Figure 40 "seems to model the problem best," but fails to say how he reaches this conclusion. I similarly conclude that Figure 40 represents the "best" program of the four. Each of the four programs clearly is adequate in that each (after fixes, etc.) does perform the function that is required. But Figure 40 is clearly the "best" in that it is the simplest of the 40. From that single property, I can assume, by experience, that it is the most readily understood and, thus, that it will implement and debug faster than the others, that it will be easier to modify, and that it will be easier to maintain. I do not feel, however, that the technique of data structure design leads inevitably to the "best" program, though it surely helps. In other words, Figure 40 is the "best" machine of the four programs because it is the simplest.

Another point which is illustrated by this section of the article is the inconsistency of using block diagrams (an engineering technique) to illustrate the four programs and (I suspect) to compare their quality with the mathematical orientation. Dr. Bergland seems to profess a theorem in say, analysis does not need diagrammatic illustration to reveal its merit or lack of merit. It stands completely on its statement and proof. A program, however, is only universally assessable in a diagrammatic representation. Even the "programming calculus" example requires the diagrammatic crunch! This would seem to raise some question about the appropriate- ness of the mathematical model of programming and of design techniques which try to imitate mathematics.

In summary, while we have advanced to some extent in the last 10 years, the progress has been quite inadequate. There are too many instances like the "programming calculus" which are attractive but inappropriate. I think that as long as the preponderant view of programs is so unrealistic this will continue to be the case. I also think that a well thought out effort has reasonable chance of improving this situation substantially.

Christopher E. Prael
Menlo Park, Calif.

Author's reply:

After reading Dr. Roman's glowing first paragraph, I found myself willing to agree with almost everything that followed. I agree that most design methodologies do tend to be problem and environment dependent. In particular, the problem of designing for real-time applications remains a challenging endeavor. The functional decomposition in Figure 12 was "knowingly" chosen to illustrate the absence of useful, accepted "principles of clear abstraction" not their presence. The subject of when to include time and space constraints will continue to be hotly debated for several more years. My prejudices still say, "Get the program structure right—then (if you must) make it fast and small."

While many other topics could have been discussed at length, they are best left to a book on the subject rather than being given lip service in an already long article.

I found Mr. Prael's comments to be interesting but not well-directed at the "guided tour" article. Regarding the question of what is the "best" program structure, this was addressed in the discussion of data flow design. Without the measures of module coupling and cohesion, I agree that which program is "best" often becomes a matter of personal taste.

A lot of people whom I admire would take strong issue with Mr. Prael's comment that "the mathematical model of programs and programming is seriously in error." As stated in my article, my main concern is that very few programmers have the mathematical sophistication to carry this approach off well, not that it is "incorrect." I do not profess to have a "mathematical orientation," but I am unwilling to discount the desirability of having one.

Consistency would be less of a problem if programs and automobiles never needed to be repaired. Unfortunately, most successful large programs need both maintenance and repair. Programmers, like mechanics, tend to work best on things that are familiar rather than unique.

The diagrams were "structure diagrams" not block diagrams. Please refer back to Figure 4 of the "guided tour."

In summary, I agree with many of Dr. Roman's concerns but feel that they are best left to a book rather than expanding an already long article. I found that Mr. Prael's version of what he thinks he read differed in several places from what I had hoped to convey. Since I will not have the opportunity for rewriting the article, I hope he will take the opportunity to reread it.

G. D. Bergland
Bell Telephone Laboratories
Murray Hill, N.J.