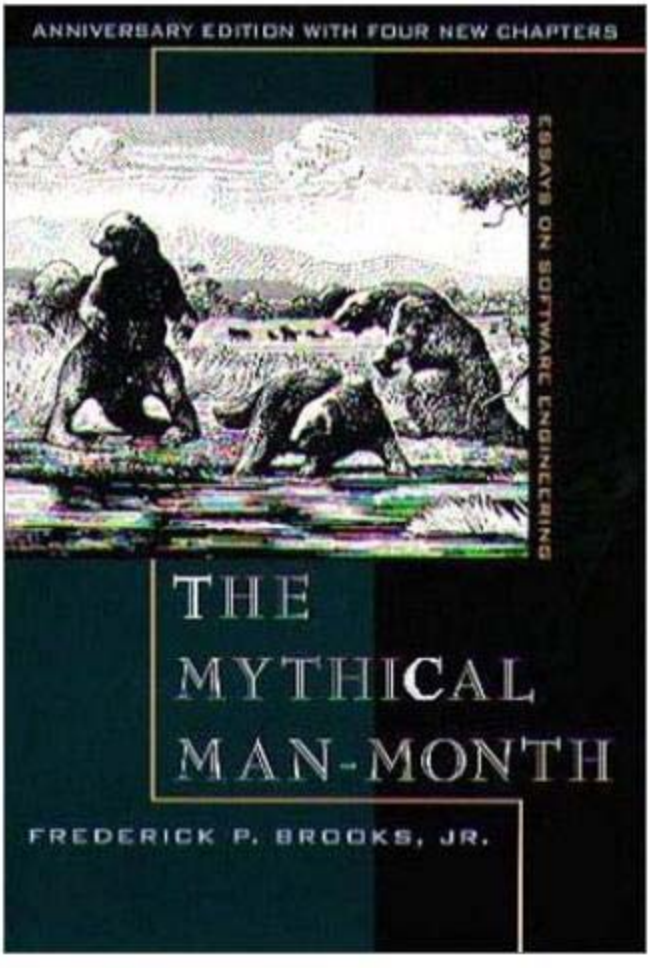
ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS
ESSAYS ON SOFTWARE ENGINEERING
THE MYTHICAL MAN-MONTH
FREDERICK P. BROOKS, JR.

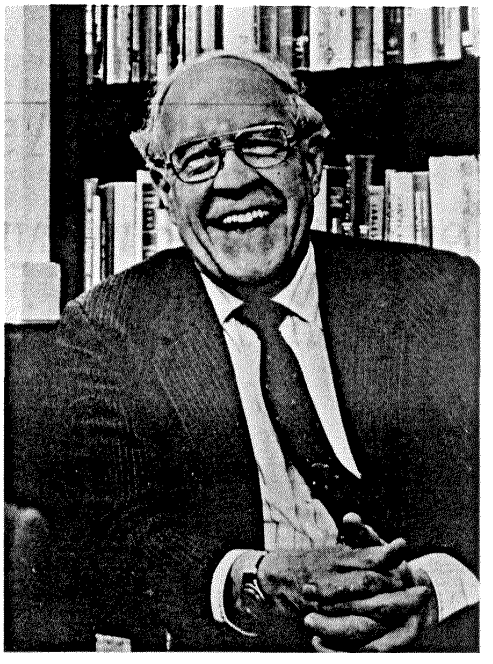

Photo credit: © Jerry Markatos

ABOUT THE AUTHOR

Frederick P. Brooks, Jr., is Kenan Professor of Computer Science at the University of North Carolina at Chapel Hill. He is best known as the "father of the IBM System/360," having served as project manager for its development and later as manager of the Operating System/360 software project during its design phase. For this work he, Bob Evans, and Erich Bloch were awarded the National Medal of Technology in 1985. Earlier, he was an architect of the IBM Stretch and Harvest computers.

At Chapel Hill, Dr. Brooks founded the Department of Computer Science and chaired it from 1964 through 1984. He has served on the National Science Board and the Defense Science Board. His current teaching and research is in computer architecture, molecular graphics, and virtual environments.

6
Passing the Word

6
Passing the Word

Hell sit here and he'll say, "Do this! Do that!" And nothing will happen.

HARRY S. TRUMAN. ON PRESIDENTIAL POWER[1]

"The Seven Trumpets" from *The Wells Apocalypse,* 14th century
The Bettman Archive

Assuming that he has the disciplined, experienced architects and that there are many implementers, how shall the manager ensure that everyone hears, understands, and implements the architects' decisions? How can a group of 10 architects maintain the conceptual integrity of a system which 1000 men are building? A whole technology for doing this was worked out for the System/360 hardware design effort, and it is equally applicable to software projects.

Written Specifications—the Manual

The manual, or written specification, is a necessary tool, though not a sufficient one. The manual is the *external* specification of the product. It describes and prescribes every detail of what the user sees. As such, it is the chief product of the architect.

Round and round goes its preparation cycle, as feedback from users and implementers shows where the design is awkward to use or build. For the sake of implementers it is important that the changes be quantized—that there be dated versions appearing on a schedule.

The manual must not only describe everything the user does see, including all interfaces; it must also refrain from describing what the user does not see. That is the implementer's business, and there his design freedom must be unconstrained. The architect must always be prepared to show *an* implementation for any feature he describes, but he must not attempt to dictate *the* implementation.

The style must be precise, full, and accurately detailed. A user will often refer to a single definition, so each one must repeat all the essentials and yet all must agree. This tends to make manuals dull reading, but precision is more important than liveliness.

The unity of System/360's *Principles of Operation* springs from the fact that only two pens wrote it: Gerry Blaauw's and Andris Padegs'. The ideas are those of about ten men, but the casting of those decisions into prose specifications must be done by only one

or two, if the consistency of prose and product is to be maintained. For the writing of a definition will necessitate a host of mini-decisions which are not of full-debate importance. An example in System/360 is the detail of how the Condition Code is set after each operation. *Not* trivial, however, is the principle that such mini-decisions be made consistently throughout.

I think the finest piece of manual writing I have ever seen is Blaauw's Appendix to *System/360 Principles of Operation*. This describes with care and precision the *limits* of System/360 compatibility. It defines compatibility, prescribes what is to be achieved, and enumerates those areas of external appearance where the architecture is intentionally silent and where results from one model may differ from those of another, where one copy of a given model may differ from another copy, or where a copy may differ even from itself after an engineering change. This is the level of precision to which manual writers aspire, and they must define what is *not* prescribed as carefully as what is.

Formal Definitions

English, or any other human language, is not naturally a precision instrument for such definitions. Therefore the manual writer must strain himself and his language to achieve the precision needed. An attractive alternative is to use a formal notation for such definitions. After all, precision is the stock in trade, the *raison d'etre* of formal notations.

Let us examine the merits and weaknesses of formal definitions. As noted, formal definitions are precise. They tend to be complete; gaps show more conspicuously, so they are filled sooner. What they lack is comprehensibility. With English prose one can show structural principles, delineate structure in stages or levels, and give examples. One can readily mark exceptions and emphasize contrasts. Most important, one can explain *why*. The formal definitions put forward so far have inspired wonder at their elegance and confidence in their precision. But they have demanded

prose explanations to make their content easy to learn and teach. For these reasons, I think we will see future specifications to consist of both a formal definition *and* a prose definition.

An ancient adage warns, "Never go to sea with two chronometers; take one or three." The same thing clearly applies to prose and formal definitions. If one has both, one must be the standard, and the other must be a derivative description, clearly labeled as such. Either can be the primary standard. Algol 68 has a formal definition as standard and a prose definition as descriptive. PL/I has the prose as standard and the formal description as derivative. System/360 also has prose as standard with a derived formal description.

Many tools are available for formal definition. The Backus-Naur Form is familiar for language definition, and it is amply discussed in the literature.[2] The formal description of PL/I uses new notions of abstract syntax, and it is adequately described.[3] Iverson's APL has been used to describe machines, most notably the IBM 7090[4] and System/360.[5]

Bell and Newell have proposed new notations for describing both configurations and machine architectures, and they have illustrated these with several machines, including the DEC PDP-8,[6] the 7090,[6] and System/360.[7]

Almost all formal definitions turn out to embody or describe an implementation of the hardware or software system whose externals they are prescribing. Syntax can be described without this, but semantics are usually defined by giving a program that carries out the defined operation. This is of course an implementation, and as such it over-prescribes the architecture. So one must take care to indicate that the formal definition applies only to externals, and one must say what these are.

Not only is a formal definition an implementation, an implementation can serve as a formal definition. When the first compatible computers were built, this was exactly the technique used. The new machine was to match an existing machine. The manual was vague on some points? "Ask the machine!" A test program

would be devised to determine the behavior, and the new machine would be built to match.

A programmed simulator of a hardware or software system can serve in precisely the same way. It is an implementation; it runs. So all questions of definition can be resolved by testing it.

Using an implementation as a definition has some advantages. All questions can be settled unambiguously by experiment. Debate is never needed, so answers are quick. Answers are always as precise as one wants, and they are always correct, by definition. Opposed to these one has a formidable set of disadvantages. The implementation may over-prescribe even the externals. Invalid syntax always produces some result; in a policed system that result is an invalidity indication *and nothing more.* In an unpoliced system all kinds of side effects may appear, and these may have been used by programmers. When we undertook to emulate the IBM 1401 on System/360, for example, it developed that there were 30 different "curios"—side effects of supposedly invalid operations—that had come into widespread use and had to be considered as part of the definition. The implementation as a definition overprescribed; it not only said what the machine must do, it also said a great deal about how it had to do it.

Then, too, the implementation will sometimes give unexpected and unplanned answers when sharp questions are asked, and the *de facto* definition will often be found to be inelegant in these particulars precisely because they have never received any thought. This inelegance will often turn out to be slow or costly to duplicate in another implementation. For example, some machines leave trash in the multiplicand register after a multiplication. The precise nature of this trash turns out to be part of the *de facto* definition, yet duplicating it may preclude the use of a faster multiplication algorithm.

Finally, the use of an implementation as a formal definition is peculiarly susceptible to confusion as to whether the prose description or the formal description is in fact the standard. This is especially true of programmed simulations. One must also refrain

from modifications to the implementation while it is serving as a standard.

Direct Incorporation

A lovely technique for disseminating and enforcing definitions is available for the software system architect. It is especially useful for establishing the syntax, if not the semantics, of intermodule interfaces. This technique is to design the declaration of the passed parameters or shared storage, and to require the implementations to include that declaration via a compile-time operation (a macro or a %INCLUDE in PL/I). If, in addition, the whole interface is referenced only by symbolic names, the declaration can be changed by adding or inserting new variables with only recompilation, not alteration, of the using program.

Conferences and Courts

Needless to say, meetings are necessary. The hundreds of man-to-man consultations must be supplemented by larger and more formal gatherings. We found two levels of these to be useful. The first is a weekly half-day conference of all the architects, plus official representatives of the hardware and software implementers, and the market planners. The chief system architect presides.

Anyone can propose problems or changes, but proposals are usually distributed in writing before the meeting. A new problem is usually discussed a while. The emphasis is on creativity, rather than merely decision. The group attempts to invent many solutions to problems, then a few solutions are passed to one or more of the architects for detailing into precisely worded manual change proposals.

Detailed change proposals then come up for decisions. These have been circulated and carefully considered by implementers and users, and the pros and cons are well delineated. If a consensus emerges, well and good. If not, the chief architect decides. Minutes

are kept and decisions are formally, promptly, and widely disseminated.

Decisions from the weekly conferences give quick results and allow work to proceed. If anyone is *too* unhappy, instant appeals to the project manager are possible, but this happens very rarely.

The fruitfulness of these meetings springs from several sources:

1. The same group—architects, users, and implementers—meets weekly for months. No time is needed for bringing people up to date.
2. The group is bright, resourceful, well versed in the issues, and deeply involved in the outcome. No one has an "advisory" role. Everyone is authorized to make binding commitments.
3. When problems are raised, solutions are sought both within and outside the obvious boundaries.
4. The formality of written proposals focuses attention, forces decision, and avoids committee-drafted inconsistencies.
5. The clear vesting of decision-making power in the chief architect avoids compromise and delay.

As time goes by, some decisions don't wear well. Some minor matters have never been wholeheartedly accepted by one or another of the participants. Other decisions have developed unforeseen problems, and sometimes the weekly meeting didn't agree to reconsider these. So there builds up a backlog of minor appeals, open issues, or disgruntlements. To settle these we held annual supreme court sessions, lasting typically two weeks. (I would hold them every six months if I were doing it again.)

These sessions were held just before major freeze dates for the manual. Those present included not only the architecture group and the programmers' and implementers' architectural representatives, but also the managers of programming, marketing, and implementation efforts. The System/360 project manager presided. The agenda typically consisted of about 200 items, mostly minor, which were enumerated in charts placarded around the room. All

sides were heard and decisions made. By the miracle of computerized text editing (and lots of fine staff work), each participant found an updated manual, embodying yesterday's decisions, at his seat every morning.

These "fall festivals" were useful not only for resolving decisions, but also for getting them accepted. Everyone was heard, everyone participated, everyone understood better the intricate constraints and interrelationships among decisions.

Multiple Implementations

System/360 architects had two almost unprecedented advantages: enough time to work carefully, and political clout equal to that of the implementers. The provision of enough time came from the schedule of the new technology; the political equality came from the simultaneous construction of multiple implementations. The necessity for strict compatibility among these served as the best possible enforcing agent for the specifications.

In most computer projects there comes a day when it is discovered that the machine and the manual don't agree. When the confrontation follows, the manual usually loses, for it can be changed far more quickly and cheaply than the machine. Not so, however, when there are multiple implementations. Then the delays and costs associated with fixing the errant machine can be overmatched by delays and costs in revising the machines that followed the manual faithfully.

This notion can be fruitfully applied whenever a programming language is being defined. One can be certain that several interpreters or compilers will sooner or later have to be built to meet various objectives. The definition will be cleaner and the discipline tighter if at least two implementations are built initially.

The Telephone Log

As implementation proceeds, countless questions of architectural interpretation arise, no matter how precise the specification. Obvi-

ously many such questions require amplifications and clarifications in the text. Others merely reflect misunderstandings.

It is essential, however, to encourage the puzzled implementer to telephone the responsible architect and ask his question, rather than to guess and proceed. It is just as vital to recognize that the answers to such questions are *ex cathedra* architectural pronouncements that must be told to everyone.

One useful mechanism is a *telephone log* kept by the architect. In it he records every question and every answer. Each week the logs of the several architects are concatenated, reproduced, and distributed to the users and implementers. While this mechanism is quite informal, it is both quick and comprehensive.

Product Test

The project manager's best friend is his daily adversary, the independent product-testing organization. This group checks machines and programs against specifications and serves as a devil's advocate, pinpointing every conceivable defect and discrepancy. Every development organization needs such an independent technical auditing group to keep it honest.

In the last analysis the customer is the independent auditor. In the merciless light of real use, every flaw will show. The product-testing group then is the surrogate customer, specialized for finding flaws. Time after time, the careful product tester will find places where the word didn't get passed, where the design decisions were not properly understood or accurately implemented. For this reason such a testing group is a necessary link in the chain by which the design word is passed, a link that needs to operate early and simultaneously with design.