ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS

ESSAYS ON SOFTWARE ENGINEERING

THE MYTHICAL MAN-MONTH

FREDERICK P. BROOKS, JR.

Mythical Man-Month, The: Essays on Software Engineering, Anniversary Edition

Frederick P. Brooks

Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Table of Contents

Copyright

Cover drawing: C. R. Knight, Mural of the La Brea Tar Pits. Courtesy of the George C. Page Museum of La Brea Discoveries, The Natural History Museum of Los Angeles County.

The essay entitled, No Silver Bullet, is from Information Processing 1986, the Proceedings of the IFIP Tenth World Computing Conference, edited by H.-J. Kugler, 1986, pages 1069–1076. Reprinted with the kind permission of IFIP and Elsevier Science B.V., Amsterdam, The Netherlands.

Library of Congress Cataloging-in-Publication Data

Brooks, Frederick P., Jr. (Frederick Phillips)
The mythical man-month : essays on software engineering /
Frederick P. Brooks, Jr. — Anniversary ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-83595-9
1. Software engineering. I. Title.
QA76.758.B75 1995
005.1\0xD5068—dc20 94-36653
CIP

Printed in the United States of America.

1 2 3 4 5 6 7 8 9 10—MA—98979695

ISBN: 0-201-83595-9

Dedication

Dedication of the 1975 edition

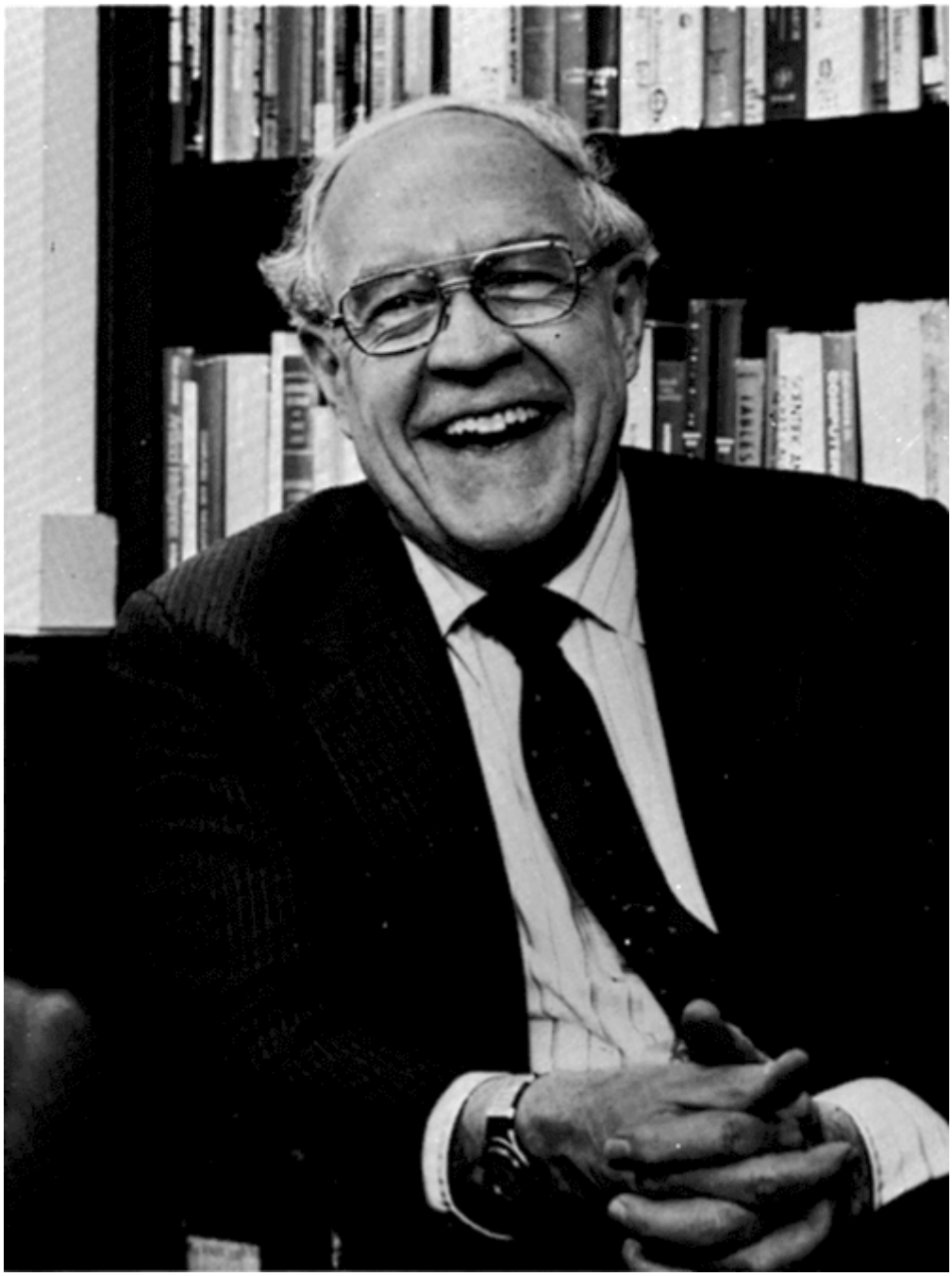
To two who especially enriched my IBM years:
Thomas J. Watson, Jr.,
whose deep concern for people still permeates his company,
and
Bob O. Evans,
whose bold leadership turned work into adventure.

Dedication of the 1995 edition

To Nancy,
God\0xD5s gift to me.

About the Author

Photo credit: ©Jerry Markatos



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.

Preface to the 20th Anniversary Edition

To my surprise and delight, *The Mythical Man-Month* continues to be popular after 20 years. Over 250,000 copies are in print. People often ask which of the opinions and recommendations set forth in 1975 I still hold, and which have changed, and how. Whereas I have from time to time addressed that question in lectures, I have long wanted to essay it in writing.

Peter Gordon, now a Publishing Partner at Addison-Wesley, has been working with me patiently and helpfully since 1980. He proposed that we prepare an Anniversary Edition. We decided not to revise the original, but to reprint it untouched (except for trivial corrections) and to augment it with more current thoughts.

Chapter 16 reprints "No Silver Bullet: Essence and Accidents of Software Engineering," a 1986 IFIPS paper that grew out of my experience chairing a Defense Science Board study on military software. My coauthors of that study, and our executive secretary, Robert L. Patrick, were invaluable in bringing me back into touch with real-world large software projects. The paper was reprinted in 1987 in the IEEE *Computer* magazine, which gave it wide circulation.

"No Silver Bullet" proved provocative. It predicted that a decade would not see any programming technique that would by itself bring an order-of-magnitude improvement in software productivity. The decade has a year to run; my prediction seems safe. "NSB" has stimulated more and more spirited discussion in the literature than has *The Mythical Man-Month*. Chapter 17, therefore, comments on some of the published critique and updates the opinions set forth in 1986.

In preparing my retrospective and update of *The Mythical Man-Month*, I was struck by how few of the propositions asserted in it have been critiqued, proven, or disproven by ongoing software engineering research and experience. It proved useful to me now to catalog those propositions in raw form, stripped of supporting arguments and data. In hopes that these bald statements will invite arguments and facts to prove, disprove, update, or refine those propositions, I have included this outline as Chapter 18.

Chapter 19 is the updating essay itself. The reader should be warned that the new opinions are not nearly so well informed by experience in the trenches as the original book was. I have been at work in a university, not industry, and on small-scale projects, not large ones. Since 1986, I have only taught software engineering, not done research in it at all. My research has rather been on virtual environments and their applications.

In preparing this retrospective, I have sought the current views of friends who are indeed at work in software engineering. For a wonderful willingness to share views, to comment thoughtfully on drafts, and to re-educate me, I am indebted to Barry Boehm, Ken Brooks, Dick Case, James Coggins, Tom DeMarco, Jim McCarthy, David Parnas, Earl Wheeler, and Edward Yourdon. Fay Ward has superbly handled the technical production of the new chapters.

I thank Gordon Bell, Bruce Buchanan, Rick Hayes-Roth, my colleagues on the Defense Science Board Task Force on Military Software, and, most especially, David Parnas for their insights and stimulating ideas for, and Rebekah Bierly for technical production of, the paper printed here as Chapter 16. Analyzing the software problem into the categories of *essence* and *accident* was inspired by Nancy Greenwood Brooks, who used such analysis in a paper on Suzuki violin pedagogy.

Addison-Wesley's house custom did not permit me to acknowledge in the preface to the 1975 edition the key roles played by their staff. Two persons' contributions should be especially cited: Norman Stanton, then Executive Editor, and Herbert Boes, then Art Director. Boes developed the elegant style, which one reviewer especially cited: "wide margins, [and] imaginative use of typeface and layout." More important, he also made the crucial recommendation that every chapter have an opening picture. (I had only the Tar Pit and Reims Cathedral at the time.) Finding the pictures occasioned an extra year's work for me, but I am eternally grateful for the counsel.

Soli Deo gloria—To God alone be glory.

F. P. B., Jr.
Chapel Hill, N.C.
March 1995

Preface to the First Edition

In many ways, managing a large computer programming project is like managing any other large undertaking—in more ways than most programmers believe. But in many other ways it is different—in more ways than most professional managers expect.

The lore of the field is accumulating. There have been several conferences, sessions at AFIPS conferences, some books, and papers. But it is by no means yet in shape for any systematic textbook treatment. It seems appropriate, however, to offer this little book, reflecting essentially a personal view.

Although I originally grew up in the programming side of computer science, I was involved chiefly in hardware architecture during the years (1956–1963) that the autonomous control program and the high-level language compiler were developed. When in 1964 I became manager of Operating System/360, I found a programming world quite changed by the progress of the previous few years.

Managing OS/360 development was a very educational experience, albeit a very frustrating one. The team, including F. M. Trapnell who succeeded me as manager, has much to be proud of. The system contains many excellencies in design and execution, and it has been successful in achieving widespread use. Certain ideas, most noticeably device-independent input-output and external library management, were technical innovations now widely copied. It is now quite reliable, reasonably efficient, and very versatile.

The effort cannot be called wholly successful, however. Any OS/360 user is quickly aware of how much better it should be. The flaws in design and execution pervade especially the control program, as distinguished from the language compilers. Most of these flaws date from the 1964–65 design period and hence must be laid to my charge. Furthermore, the product was late, it took more memory than planned, the costs were several times the estimate, and it did not perform very well until several releases after the first.

After leaving IBM in 1965 to come to Chapel Hill as originally agreed when I took over OS/360, I began to analyze the OS/360 experience to see what management and technical lessons were to be learned. In particular, I wanted to explain the quite different management experiences encountered in System/ 360 hardware development and OS/360 software development. This book is a belated answer to Tom Watson's probing questions as to why programming is hard to manage.

In this quest I have profited from long conversations with R. P. Case, assistant manager 1964–65, and F. M. Trapnell, manager 1965–68. I have compared conclusions with other managers of jumbo programming projects, including F. J. Corbato of M.I.T., John Harr and V. Vyssotsky of Bell Telephone Laboratories, Charles Portman of International Computers Limited, A. P. Ershov of the Computation Laboratory of the Siberian Division, U.S.S.R. Academy of Sciences, and A. M. Pietrasanta of IBM.

My own conclusions are embodied in the essays that follow, which are intended for professional programmers, professional managers, and especially professional managers of programmers.

Although written as separable essays, there is a central argument contained especially in Chapters 2–7. Briefly, I believe that large programming projects suffer management problems different in kind from small ones, due to division of labor. I believe the critical need to be the preservation of the conceptual integrity of the product itself. These chapters explore both the difficulties of achieving this unity and methods for doing so. The later chapters explore other aspects of software engineering management.

The literature in this field is not abundant, but it is widely scattered. Hence I have tried to give references that will both illuminate particular points and guide the interested reader to other useful works. Many friends have read the manuscript, and some have prepared extensive helpful comments; where these seemed valuable but did not fit the flow of the text, I have included them in the notes.

Because this is a book of essays and not a text, all the references and notes have been banished to the end of the volume, and the reader is urged to ignore them on his first reading.

I am deeply indebted to Miss Sara Elizabeth Moore, Mr. David Wagner, and Mrs. Rebecca Burris for their help in preparing the manuscript, and to Professor Joseph C. Sloane for advice on illustration.

F. P. B., Jr
Chapel Hill, N.C.
October 1974

Chapter 4. Aristocracy, Democracy, and System Design

> *This great church is an incomparable work of art. There is neither aridity nor confusion in the tenets it sets forth. . . .*
> *It is the zenith of a style, the work of artists who had understood and assimilated all their predecessors' successes, in complete possession of the techniques of their times, but using them without indiscreet display nor gratuitous feats of skill.*
> *It was Jean d'Orbais who undoubtedly conceived the general plan of the building, a plan which was respected, at least in its essential elements, by his successors. This is one of the reasons for the extreme coherence and unity of the edifice.*
> —*REIMS CATHEDRAL GUIDEBOOK*[1]

Conceptual Integrity

Most European cathedrals show differences in plan or architectural style between parts built in different generations by different builders. The later builders were tempted to "improve" upon the designs of the earlier ones, to reflect both changes in fashion and differences in individual taste. So the peaceful Norman transept abuts and contradicts the soaring Gothic nave, and the result proclaims the pridefulness of the builders as much as the glory of God.

Against these, the architectural unity of Reims stands in glorious contrast. The joy that stirs the beholder comes as much from the integrity of the design as from any particular excellences. As the guidebook tells, this integrity was achieved by the self-abnegation of eight generations of builders, each of whom sacrificed some of his ideas so that the whole might be of pure design. The result proclaims not only the glory of God, but also His power to salvage fallen men from their pride.

Even though they have not taken centuries to build, most programming systems reflect conceptual disunity far worse than that of cathedrals. Usually this arises not from a serial succession of master designers, but from the separation of design into many tasks done by many men.

I will contend that conceptual integrity is *the* most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas. In this chapter and the next two, we will examine the consequences of this theme for programming system design:

- How is conceptual integrity to be achieved?

- Does not this argument imply an elite, or aristocracy of architects, and a horde of plebeian implementers whose creative talents and ideas are suppressed?
- How does one keep the architects from drifting off into the blue with unimplementable or costly specifications?
- How does one ensure that every trifling detail of an architectural specification gets communicated to the implementer, properly understood by him, and accurately incorporated into the product?

Achieving Conceptual Integrity

The purpose of a programming system is to make a computer easy to use. To do this, it furnishes languages and various facilities that are in fact programs invoked and controlled by language features. But these facilities are bought at a price: the external description of a programming system is ten to twenty times as large as the external description of the computer system itself. The user finds it far easier to specify any particular function, but there are far more to choose from, and far more options and formats to remember.

Ease of use is enhanced only if the time gained in functional specification exceeds the time lost in learning, remembering, and searching manuals. With modern programming systems this gain does exceed the cost, but in recent years the ratio of gain to cost seems to have fallen as more and more complex functions have been added. I am haunted by the memory of the ease of use of the IBM 650, even without an assembler or any other software at all.

Because ease of use is the purpose, this ratio of function to conceptual complexity is the ultimate test of system design. Neither function alone nor simplicity alone defines a good design.

This point is widely misunderstood. Operating System/360 is hailed by its builders as the finest ever built, because it indisputably has the most function. Function, and not simplicity, has always been the measure of excellence for its designers. On the other hand, the Time-Sharing System for the PDP-10 is hailed by its builders as the finest, because of its simplicity and the spareness of its concepts. By any measure, however, its function is not even in the same class as that of OS/360. As soon as ease of use is held up as the criterion, each of these is seen to be unbalanced, reaching for only half of the true goal.

For a given level of function, however, that system is best in which one can specify things with the most simplicity and straightforwardness. *Simplicity* is not enough. Mooers's TRAC language and Algol 68 achieve simplicity as measured by the number of distinct elementary concepts. They are not, however, *straightforward*. The expression of the things one wants to do often requires involuted and unexpected combinations of the basic facilities. It is not enough to learn the elements and rules of combination; one must also learn the idiomatic usage, a whole lore of how the elements are combined in practice. Simplicity and straightforwardness proceed from conceptual integrity. Every part must reflect the same philosophies and the same balancing of desiderata. Every part must even use the same techniques in syntax and analogous notions in semantics. Ease of use, then, dictates unity of design, conceptual integrity.

Aristocracy and Democracy

Conceptual integrity in turn dictates that the design must proceed from one mind, or from a very small number of agreeing resonant minds.

Schedule pressures, however, dictate that system building needs many hands. Two techniques are available for resolving this dilemma. The first is a careful division of labor between architecture and implementation. The second is the new way of structuring programming implementation teams discussed in the previous chapter.

The separation of architectural effort from implementation is a very powerful way of getting conceptual integrity on very large projects. I myself have seen it used with great success on IBM's Stretch computer and on the System/360 computer product line. I have seen it fail through lack of application on Operating System/360.

By the *architecture* of a system, I mean the complete and detailed specification of the user interface. For a computer this is the programming manual. For a compiler it is the language manual. For a control program it is the manuals for the language or languages used to invoke its functions. For the entire system it is the union of the manuals the user must consult to do his entire job.

The architect of a system, like the architect of a building, is the user's agent. It is his job to bring professional and technical knowledge to bear in the unalloyed interest of the user, as opposed to the interests of the salesman, the fabricator, etc.[2]

Architecture must be carefully distinguished from implementation. As Blaauw has said, "Where architecture tells *what* happens, implementation tells *how* it is made to happen."[3] He gives as a simple example a clock, whose architecture consists of the face, the hands, and the winding knob. When a child has learned this architecture, he can tell time as easily from a wristwatch as from a church tower. The implementation, however, and its realization, describe what goes on inside the case—powering by any of many mechanisms and accuracy control by any of many.

In System/360, for example, a single computer architecture is implemented quite differently in each of some nine models. Conversely, a single implementation, the Model 30 data flow, memory, and microcode, serves at different times for four different architectures: a System/360 computer, a multiplex channel with up to 224 logically independent subchannels, a selector channel, and a 1401 computer.[4]

The same distinction is equally applicable to programming systems. There is a U.S. standard Fortran IV. This is the architecture for many compilers. Within this architecture many implementations are possible: text-in-core or compiler-in-core, fast-compile or optimizing, syntax-directed or *ad-hoc*. Likewise any assembler language or job-control language admits of many implementations of the assembler or scheduler.

Now we can deal with the deeply emotional question of aristocracy versus democracy. Are not the architects a new aristocracy, an intellectual elite, set up to tell the poor dumb implementers what to do? Has not all the creative work been sequestered for this elite, leaving the implementers as cogs in the machine? Won't one get a better product by getting the good ideas from all the team, following a democratic philosophy, rather than by restricting the development of specifications to a few?

As to the last question, it is the easiest. I will certainly not contend that only the architects will have good architectural ideas. Often the fresh concept does come from an implementer or from a user. However, all my own experience convinces me, and I have tried to show, that the conceptual integrity of a system determines its ease of use. Good features and ideas that do not integrate with a system's basic concepts are best left out. If there appear many such important but incompatible ideas, one scraps the whole system and starts again on an integrated system with different basic concepts.

As to the aristocracy charge, the answer must be yes and no. Yes, in the sense that there must be few architects, their product must endure longer than that of an implementer, and the architect sits at the focus of forces which he must ultimately resolve in the user's interest. If a system is to have conceptual integrity, someone must control the concepts. That is an aristocracy that needs no apology.

No, because the setting of external specifications is not more creative work than the designing of implementations. It is just different creative work. The design of an implementation, given an architecture, requires and allows as much design creativity, as many new ideas, and as much technical brilliance as the design of the external specifications. Indeed, the cost-performance ratio of the product will depend most heavily on the implementer, just as ease of use depends most heavily on the architect.

There are many examples from other arts and crafts that lead one to believe that discipline is good for art. Indeed, an artist's aphorism asserts, "Form is liberating." The worst buildings are those whose budget was too great for the purposes to be served. Bach's creative output hardly seems to have been squelched by the necessity of producing a limited-form cantata each week. I am sure that the Stretch computer would have had a better architecture had it been more tightly constrained; the constraints imposed by the System/360 Model 30's budget were in my opinion entirely beneficial for the Model 75's architecture.

Similarly, I observe that the external provision of an architecture enhances, not cramps, the creative style of an implementing group. They focus at once on the part of the problem no one has addressed, and inventions begin to flow. In an unconstrained implementing group, most thought and debate goes into architectural decisions, and implementation proper gets short shrift.[5]

This effect, which I have seen many times, is confirmed by R. W. Conway, whose group at Cornell built the PL/C compiler for the PL/I language. He says, "We finally decided to implement the language unchanged and unimproved, for the debates about language would have taken all our effort."[6]

What Does the Implementer Do While Waiting?

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.

Many actors, of course, entered into that mistaken decision; but the overwhelming one was schedule time and the appeal of putting all those 150 implementers to work. It is this siren song whose deadly hazards I would now make visible.

When it is proposed that a small architecture team in fact write all the external specifications for a computer or a programming system, the implementers raise three objections:

- The specifications will be too rich in function and will not reflect practical cost considerations.
- The architects will get all the creative fun and shut out the inventiveness of the implementers.
- The many implementers will have to sit idly by while the specifications come through the narrow funnel that is the architecture team.

The first of these is a real danger, and it will be treated in the next chapter. The other two are illusions, pure and simple. As we have seen above, implementation is also a creative activity of the first order. The opportunity to be creative and inventive in implementation is not significantly diminished by working within a given external specification, and the order of creativity may even be enhanced by that discipline. The total product will surely be.

The last objection is one of timing and phasing. A quick answer is to refrain from hiring implementers until the specifications are complete. This is what is done when a building is constructed.

In the computer systems business, however, the pace is quicker, and one wants to compress the schedule as much as possible. How much can specification and building be overlapped?

As Blaauw points out, the total creative effort involves three distinct phases: architecture, implementation, and realization. It turns out that these can in fact be begun in parallel and proceed simultaneously.

In computer design, for example, the implementer can start as soon as he has relatively vague assumptions about the manual, somewhat clearer ideas about the technology, and well-defined cost and performance objectives. He can begin designing data flows, control

sequences, gross packaging concepts, and so on. He devises or adapts the tools he will need, especially the record-keeping system, including the design automation system.

Meanwhile, at the realization level, circuits, cards, cables, frames, power supplies, and memories must each be designed, refined, and documented. This work proceeds in parallel with architecture and implementation.

The same thing is true in programming system design. Long before the external specifications are complete, the implementer has plenty to do. Given some rough approximations as to the function of the system that will be ultimately embodied in the external specifications, he can proceed. He must have well-defined space and time objectives. He must know the system configuration on which his product must run. Then he can begin designing module boundaries, table structures, pass or phase breakdowns, algorithms, and all kinds of tools. Some time, too, must be spent in communicating with the architect.

Meanwhile, on the realization level there is much to be done also. Programming has a technology, too. If the machine is a new one, much work must be done on subroutine conventions, supervisory techniques, searching and sorting algorithms.[7]

Conceptual integrity does require that a system reflect a single philosophy and that the specification as seen by the user flow from a few minds. Because of the real division of labor into architecture, implementation, and realization, however, this does not imply that a system so designed will take longer to build. Experience shows the opposite, that the integral system goes together faster and takes less time to test. In effect, a widespread horizontal division of labor has been sharply reduced by a vertical division of labor, and the result is radically simplified communications and improved conceptual integrity.