ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS

ESSAYS ON SOFTWARE ENGINEERING

THE MYTHICAL MAN-MONTH

FREDERICK P. BROOKS, JR.

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
005.1'068—dc20 94-36653
CIP

ISBN 0-201-83595-9

Text printed in the United States on recycled paper at RR Donnelley Crawfordsville in Crawfordsville, Indiana.

Printing 35th January 2010

Table of Contents

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 1. The Tar Pit

Een schip op het strand is een baken in zee.
[*A ship on the beach is a lighthouse to the sea.*]
—*DUTCH PROVERB*

No scene from prehistory is quite so vivid as that of the mortal struggles of great beasts in the tar pits. In the mind's eye one sees dinosaurs, mammoths, and sabertoothed tigers struggling against the grip of the tar. The fiercer the struggle, the more entangling the tar, and no beast is so strong or so skillful but that he ultimately sinks.

Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems—few have met goals, schedules, and budgets. Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty—any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion. Everyone seems to have been surprised by the stickiness of the problem, and it is hard to discern the nature of it. But we must try to understand it if we are to solve it.

Therefore let us begin by identifying the craft of system programming and the joys and woes inherent in it.

The Programming Systems Product

One occasionally reads newspaper accounts of how two programmers in a remodeled garage have built an important program that surpasses the best efforts of large teams. And every programmer is prepared to believe such tales, for he knows that he could build *any* program much faster than the 1000 statements/year reported for industrial teams.

Why then have not all industrial programming teams been replaced by dedicated garage duos? One must look at *what* is being produced.

In the upper left of Fig. 1.1 is a *program*. It is complete in itself, ready to be run by the author on the system on which it was developed. *That* is the thing commonly produced in garages, and that is the object the individual programmer uses in estimating productivity.

Figure 1.1. Evolution of the programming systems product

There are two ways a program can be converted into a more useful, but more costly, object. These two ways are represented by the boundaries in the diagram.

Moving down across the horizontal boundary, a program becomes a *programming product*. This is a program that can be run, tested, repaired, and extended by anybody. It is usable in many operating environments, for many sets of data. To become a generally usable programming product, a program must be written in a generalized fashion. In particular the range and form of inputs must be generalized as much as the basic algorithm will reasonably allow. Then the program must be thoroughly tested, so that it can be depended upon. This means that a substantial bank of test cases, exploring the input range and probing its boundaries, must be prepared, run, and recorded. Finally, promotion of a program to a programming product requires its thorough documentation, so that anyone may use it, fix it, and extend it. As a rule of thumb, I estimate that a programming product costs at least three times as much as a debugged program with the same function.

Moving across the vertical boundary, a program becomes a component in a *programming system*. This is a collection of interacting programs, coordinated in function and disciplined in format, so that the assemblage constitutes an entire facility for large tasks. To become a programming system component, a program must be written so that every input and output conforms in syntax and semantics with precisely defined interfaces. The program must also be designed so that it uses only a prescribed budget of resources—memory space, input-output devices, computer time. Finally, the program must be tested with other system components, in all expected combinations. This testing must be extensive, for the number of cases grows combinatorially. It is time-consuming, for subtle bugs arise from unexpected interactions of debugged components. A programming system component costs at least three times as much as a stand-alone program of the same function. The cost may be greater if the system has many components.

In the lower right-hand corner of Fig. 1.1 stands the *programming systems product*. This differs from the simple program in all of the above ways. It costs nine times as much. But it is the truly useful object, the intended product of most system programming efforts.

The Joys of the Craft

Why is programming fun? What delights may its practitioner expect as his reward?

First is the sheer joy of making things. As the child delights in his mud pie, so the adult enjoys building things, especially things of his own design. I think this delight must be an image of God's delight in making things, a delight shown in the distinctness and newness of each leaf and each snowflake.

Second is the pleasure of making things that are useful to other people. Deep within, we want others to use our work and to find it helpful. In this respect the programming system is not essentially different from the child's first clay pencil holder "for Daddy's office."

Third is the fascination of fashioning complex puzzle-like objects of interlocking moving parts and watching them work in subtle cycles, playing out the consequences of principles built in from the beginning. The programmed computer has all the fascination of the pinball machine or the jukebox mechanism, carried to the ultimate.

Fourth is the joy of always learning, which springs from the nonrepeating nature of the task. In one way or another the problem is ever new, and its solver learns something: sometimes practical, sometimes theoretical, and sometimes both.

Finally, there is the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures. (As we shall see later, this very tractability has its own problems.)

Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms. The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be.

Programming then is fun because it gratifies creative longings built deep within us and delights sensibilities we have in common with all men.

The Woes of the Craft

Not all is delight, however, and knowing the inherent woes makes it easier to bear them when they appear.

First, one must perform perfectly. The computer resembles the magic of legend in this respect, too. If one character, one pause, of the incantation is not strictly in proper form, the magic doesn't work. Human beings are not accustomed to being perfect, and few areas of human activity demand it. Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program.[1]

Next, other people set one's objectives, provide one's resources, and furnish one's information. One rarely controls the circumstances of his work, or even its goal. In management terms, one's authority is not sufficient for his responsibility. It seems that in all fields, however, the jobs where things get done never have formal authority commensurate with responsibility. In practice, actual (as opposed to formal) authority is acquired from the very momentum of accomplishment.

The dependence upon others has a particular case that is especially painful for the system programmer. He depends upon other people's programs. These are often maldesigned, poorly implemented, incompletely delivered (no source code or test cases), and poorly documented. So he must spend hours studying and fixing things that in an ideal world would be complete, available, and usable.

The next woe is that designing grand concepts is fun; finding nitty little bugs is just work. With any creative activity come dreary hours of tedious, painstaking labor, and programming is no exception.

Next, one finds that debugging has a linear convergence, or worse, where one somehow expects a quadratic sort of approach to the end. So testing drags on and on, the last difficult bugs taking more time to find than the first.

The last woe, and sometimes the last straw, is that the product over which one has labored so long appears to be obsolete upon (or before) completion. Already colleagues and competitors are in hot pursuit of new and better ideas. Already the displacement of one's thought-child is not only conceived, but scheduled.

This always seems worse than it really is. The new and better product is generally not *available* when one completes his own; it is only talked about. It, too, will require months of development. The real tiger is never a match for the paper one, unless actual use is wanted. Then the virtues of reality have a satisfaction all their own.

Of course the technological base on which one builds is *always* advancing. As soon as one freezes a design, it becomes obsolete in terms of its concepts. But implementation of real products demands phasing and quantizing. The obsolescence of an implementation must be measured against other existing implementations, not against unrealized concepts. The challenge and the mission are to find real solutions to real problems on actual schedules with available resources.

This then is programming, both a tar pit in which many efforts have floundered and a creative activity with joys and woes all its own. For many, the joys far outweigh the woes, and for them the remainder of this book will attempt to lay some boardwalks across the tar.

Chapter 2. The Mythical Man-Month

> *Good cooking takes time. If you are made to wait, it is to serve you better, and to please you.*
> —*MENU OF RESTAURANT ANTOINE, NEW ORLEANS*

More software projects have gone awry for lack of calendar time than for all other causes combined. Why is this cause of disaster so common?

First, our techniques of estimating are poorly developed. More seriously, they reflect an unvoiced assumption which is quite untrue, i.e., that all will go well.

Second, our estimating techniques fallaciously confuse effort with progress, hiding the assumption that men and months are interchangeable.

Third, because we are uncertain of our estimates, software managers often lack the courteous stubbornness of Antoine's chef.

Fourth, schedule progress is poorly monitored. Techniques proven and routine in other engineering disciplines are considered radical innovations in software engineering.

Fifth, when schedule slippage is recognized, the natural (and traditional) response is to add manpower. Like dousing a fire with gasoline, this makes matters worse, much worse. More fire requires more gasoline, and thus begins a regenerative cycle which ends in disaster.

Schedule monitoring will be the subject of a separate essay. Let us consider other aspects of the problem in more detail.

Optimism

All programmers are optimists. Perhaps this modern sorcery especially attracts those who believe in happy endings and fairy godmothers. Perhaps the hundreds of nitty frustrations drive away all but those who habitually focus on the end goal. Perhaps it is merely that computers are young, programmers are younger, and the young are always optimists. But however the selection process works, the result is indisputable: "This time it will surely run," or "I just found the last bug."

So the first false assumption that underlies the scheduling of systems programming is that *all will go well*, i.e., that *each task will take only as long as it "ought" to take*.

The pervasiveness of optimism among programmers deserves more than a flip analysis. Dorothy Sayers, in her excellent book, *The Mind of the Maker*, divides creative activity into three stages: the idea, the implementation, and the interaction. A book, then, or a computer, or a program comes into existence first as an ideal construct, built outside time and space, but complete in the mind of the author. It is realized in time and space, by pen, ink, and paper, or by wire, silicon, and ferrite. The creation is complete when someone reads the book, uses the computer, or runs the program, thereby interacting with the mind of the maker.

This description, which Miss Sayers uses to illuminate not only human creative activity but also the Christian doctrine of the Trinity, will help us in our present task. For the human makers of things, the incompletenesses and inconsistencies of our ideas become clear only during implementation. Thus it is that writing, experimentation, "working out" are essential disciplines for the theoretician.

In many creative activities the medium of execution is intractable. Lumber splits; paints smear; electrical circuits ring. These physical limitations of the medium constrain the ideas that may be expressed, and they also create unexpected difficulties in the implementation.

Implementation, then, takes time and sweat both because of the physical media and because of the inadequacies of the underlying ideas. We tend to blame the physical media for most of our implementation difficulties; for the media are not "ours" in the way the ideas are, and our pride colors our judgment.

Computer programming, however, creates with an exceedingly tractable medium. The programmer builds from pure thought-stuff: concepts and very flexible representations thereof. Because the medium is tractable, we expect few difficulties in implementation; hence our pervasive optimism. Because our ideas are faulty, we have bugs; hence our optimism is unjustified.

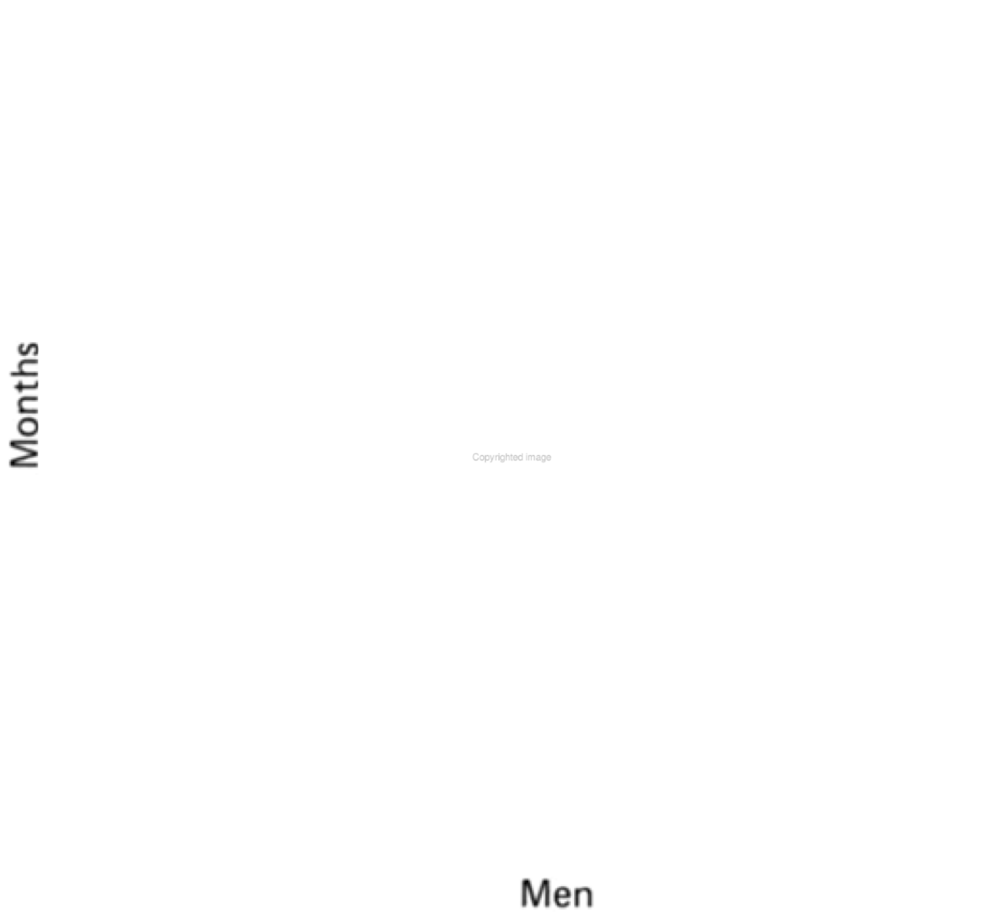
In a single task, the assumption that all will go well has a probabilistic effect on the schedule. It might indeed go as planned, for there is a probability distribution for the delay that will be encountered, and "no delay" has a finite probability. A large programming effort, however, consists of many tasks, some chained end-to-end. The probability that each will go well becomes vanishingly small.

The Man-Month

The second fallacious thought mode is expressed in the very unit of effort used in estimating and scheduling: the man-month. Cost does indeed vary as the product of the number of men and the number of months. Progress does not. *Hence the man-month as a unit for measuring the size of a job is a dangerous and deceptive myth.* It implies that men and months are interchangeable.

Men and months are interchangeable commodities only when a task can be partitioned among many workers *with no communication among them* (Fig. 2.1). This is true of reaping wheat or picking cotton; it is not even approximately true of systems programming.

Figure 2.1. Time versus number of workers—perfectly partitionable task



When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule (Fig. 2.2). The bearing of a child takes nine months, no matter how many women are assigned. Many software tasks have this characteristic because of the sequential nature of debugging.

Figure 2.2. Time versus number of workers—unpartitionable task

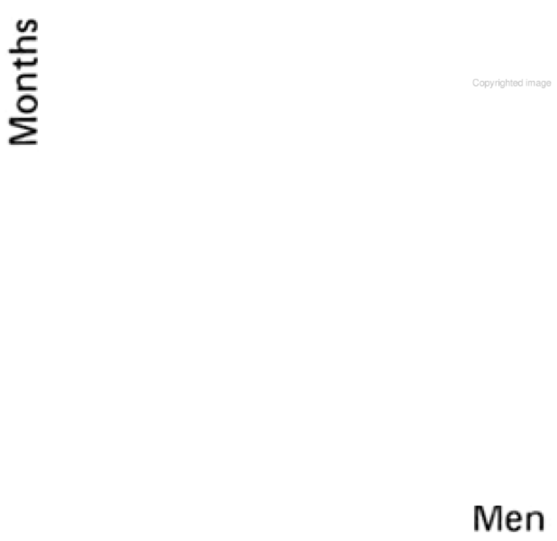
In tasks that can be partitioned but which require communication among the subtasks, the effort of communication must be added to the amount of work to be done. Therefore the best that can be done is somewhat poorer than an even trade of men for months (Fig. 2.3).

Figure 2.3. Time versus number of workers—partitionable task requiring communication

The added burden of communication is made up of two parts, training and intercommunication. Each worker must be trained in the technology, the goals of the effort, the overall strategy, and the plan of work. This training cannot be partitioned, so this part of the added effort varies linearly with the number of workers.[1]

Intercommunication is worse. If each part of the task must be separately coordinated with each other part, the effort increases as $n(n-1)/2$. Three workers require three times as much pairwise intercommunication as two; four require six times as much as two. If, moreover, there need to be conferences among three, four, etc., workers to resolve things jointly, matters get worse yet. The added effort of communicating may fully counteract the division of the original task and bring us to the situation of Fig. 2.4.

Figure 2.4. Time versus number of workers—task with complex interrelationships



Since software construction is inherently a systems effort—an exercise in complex interrelationships—communication effort is great, and it quickly dominates the decrease in individual task time brought about by partitioning. Adding more men then lengthens, not shortens, the schedule.

Systems Test

No parts of the schedule are so thoroughly affected by sequential constraints as component debugging and system test. Furthermore, the time required depends on the number and subtlety of the errors encountered. Theoretically this number should be zero. Because of optimism, we usually expect the number of bugs to be smaller than it turns out to be. Therefore testing is usually the most mis-scheduled part of programming.

For some years I have been successfully using the following rule of thumb for scheduling a software task:

1/3 planning
1/6 coding
1/4 component test and early system test
1/4 system test, all components in hand.

This differs from conventional scheduling in several important ways:

1. The fraction devoted to planning is larger than normal. Even so, it is barely enough to produce a detailed and solid specification, and not enough to include research or exploration of totally new techniques.
2. The *half* of the schedule devoted to debugging of completed code is much larger than normal.
3. The part that is easy to estimate, i.e., coding, is given only one-sixth of the schedule.

In examining conventionally scheduled projects, I have found that few allowed one-half of the projected schedule for testing, but that most did indeed spend half of the actual schedule for that purpose. Many of these were on schedule until and except in system testing.[2]

Failure to allow enough time for system test, in particular, is peculiarly disastrous. Since the delay comes at the end of the schedule, no one is aware of schedule trouble until almost the delivery date. Bad news, late and without warning, is unsettling to customers and to managers.

Furthermore, delay at this point has unusually severe financial, as well as psychological, repercussions. The project is fully staffed, and cost-per-day is maximum. More seriously, the software is to support other business effort (shipping of computers, operation of new facilities, etc.) and the secondary costs of delaying these are very high, for it is almost time for software shipment. Indeed, these secondary costs may far outweigh all others. It is therefore very important to allow enough system test time in the original schedule.

Gutless Estimating

Observe that for the programmer, as for the chef, the urgency of the patron may govern the scheduled completion of the task, but it cannot govern the actual completion. An omelette, promised in two minutes, may appear to be progressing nicely. But when it has not set in two minutes, the customer has two choices—wait or eat it raw. Software customers have had the same choices.

The cook has another choice; he can turn up the heat. The result is often an omelette nothing can save—burned in one part, raw in another.

Now I do not think software managers have less inherent courage and firmness than chefs, nor than other engineering managers. But false scheduling to match the patron's desired date is much more common in our discipline than elsewhere in engineering. It is very difficult to make a vigorous, plausible, and job-risking defense of an estimate that is derived by no quantitative method, supported by little data, and certified chiefly by the hunches of the managers.

Clearly two solutions are needed. We need to develop and publicize productivity figures, bug-incidence figures, estimating rules, and so on. The whole profession can only profit from sharing such data.

Until estimating is on a sounder basis, individual managers will need to stiffen their backbones and defend their estimates with the assurance that their poor hunches are better than wish-derived estimates.

Regenerative Schedule Disaster

What does one do when an essential software project is behind schedule? Add manpower, naturally. As Figs. 2.1 through 2.4 suggest, this may or may not help.

Let us consider an example.[3] Suppose a task is estimated at 12 man-months and assigned to three men for four months, and that there are measurable mileposts A, B, C, D, which are scheduled to fall at the end of each month (Fig. 2.5).

Figure 2.5.

Now suppose the first milepost is not reached until two months have elapsed (Fig. 2.6). What are the alternatives facing the manager?

1. Assume that the task must be done on time. Assume that only the first part of the task was misestimated, so Fig. 2.6 tells the story accurately. Then 9 man-months of effort remain, and two months, so 4 1/2 men will be needed. Add 2 men to the 3 assigned.

Figure 2.6.

2. Assume that the task must be done on time. Assume that the whole estimate was uniformly low, so that Fig. 2.7 really describes the situation. Then 18 man-months of effort remain, and two months, so 9 men will be needed. Add 6 men to the 3 assigned.

Figure 2.7.

3. Reschedule. I like the advice given by P. Fagg, an experienced hardware engineer, "Take no small slips." That is, allow enough time in the new schedule to ensure that the work can be carefully and thoroughly done, and that rescheduling will not have to be done again.
4. Trim the task. In practice this tends to happen anyway, once the team observes schedule slippage. Where the secondary costs of delay are very high, this is the only feasible action. The manager's only alternatives are to trim it formally and carefully, to reschedule, or to watch the task get silently trimmed by hasty design and incomplete testing.

In the first two cases, insisting that the unaltered task be completed in four months is disastrous. Consider the regenerative effects, for example, for the first alternative (Fig. 2.8). The two new men, however competent and however quickly recruited, will require training in the task by one of the experienced men. If this takes a month, *3 man-months will have been devoted to work not in the original estimate*. Furthermore, the task, originally partitioned three ways, must be repartitioned into five parts; hence some work already done will be lost, and system testing must be lengthened. So at the end of the third month, substantially more than 7 man-months of effort remain, and 5 trained people and one month are available. As Fig. 2.8 suggests, the product is just as late as if no one had been added (Fig. 2.6).

Figure 2.8.

To hope to get done in four months, considering only training time and not repartitioning and extra systems test, would require adding 4 men, not 2, at the end of the second month. To cover repartitioning and system test effects, one would have to add still other men. Now, however, one has at least a 7-man team, not a 3-man one; thus such aspects as team organization and task division are different in kind, not merely in degree.

Notice that by the end of the third month things look very black. The March 1 milestone has not been reached in spite of all the managerial effort. The temptation is very strong to repeat the cycle, adding yet more manpower. Therein lies madness.

The foregoing assumed that only the first milestone was misestimated. If on March 1 one makes the conservative assumption that the whole schedule was optimistic, as Fig. 2.7 depicts, one wants to add 6 men just to the original task. Calculation of the training, repartitioning, system testing effects is left as an exercise for the reader. Without a doubt, the regenerative disaster will yield a poorer product, later, than would rescheduling with the original three men, unaugmented.

Oversimplifying outrageously, we state Brooks's Law:

Adding manpower to a late software project makes it later.