The Mythical Man-Month

by

Frederick P. Brooks, Jr.

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.

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 hike 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 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 Mythical 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.1). This is true of reaping wheat or picking cotton; it is not even approximately true of systems programming.

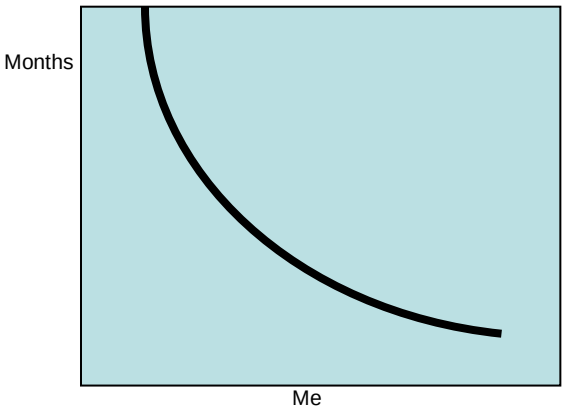

Fig. 1. The term "man-month" implies that if one man can take 10 months to do a job, 10 men can do it in one month. This may be true of picking cotton.

When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule. 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.

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).

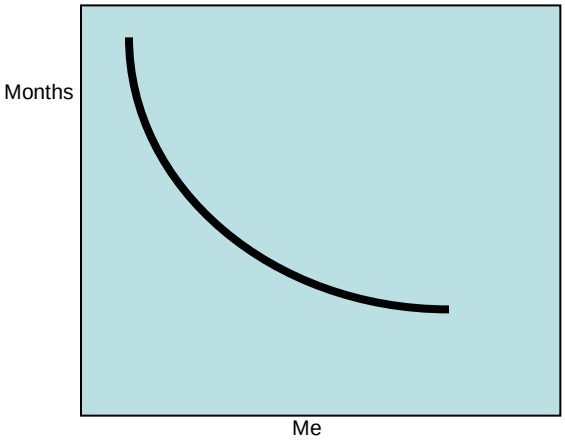

Fig. 2. Even on tasks that can be nicely partitioned among people, the additional communication required adds to the total work, increasing the schedule.

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.

V.S. Vyssottsky of Bell Telephone Laboratories estimates that a large project can sustain a manpower buildup of 30% per year. More than taht strains and even inhibits the evolution of the essential informational structure and its communication pathways. F.J. Corbato' of MIT points out that a long project must anticipate a turnover of 20% per year, and new people must be both technically trained and integrated into the formal structure.

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. 3.

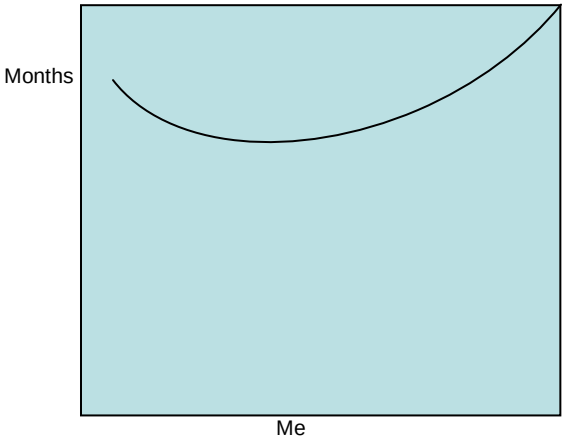

Fig. 3. Since software construction is complex, the communications overhead is great. Adding more men can lengthen, rather than shorten, the schedule.

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:

l /3 planning
1/6 coding
l /4 component test and early system test
l /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.

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.1 through 3 suggest, this may or may not help.

Let us consider an example. 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.

Now suppose the first milepost is not reached until two months have elapsed. 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. Then 9 man-months of effort remain, and two months, so 4½ men will be needed. Add 2 men to the 3 assigned.
2. Assume that the task must be done on time. Assume that the whole estimate was uniformly low. Then 18 man-months of effort remain, and two months, so 9 men will be needed. Add 6 men to the 3 assigned.
3. Reschedule. I like the advice given by 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. 4). 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. 4 suggests, the product is just as late as if no one had been added.

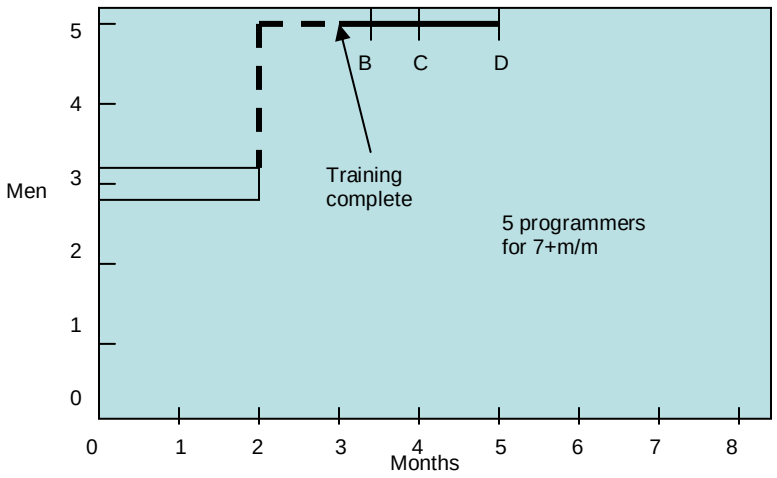

Fig. 4. Adding manpower to a project which is late may not help. In this case, suppose three men on a 12 man-month project were a month late. If it takes one of the three an extra month to train the two new men, the project will be just as late as if no one was added.

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, 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.

This then is the demythologizing of the man-month. The number of months of a project depends upon its sequential constraints. The maximum number of men depends upon the number of independent subtasks. From these two quantities one can derive schedules using

fewer men and more months. (The only risk is product obsolescence.) One cannot, however, get workable schedules using more men and fewer months. More software projects have gone awry for lack of calendar time than for all other causes combined.

Calling the Shot

How long will a system programming job take? How much effort will be required? How does one estimate?

I have earlier suggested ratios that seem to apply to planning time, coding, component test, and system test. First, one must say that one does not estimate the entire task by estimating the coding portion only and then applying the ratios. The coding is only one-sixth or so of the problem, and errors in its estimate or in the ratios could lead to ridiculous results.

Second, one must say that data for building isolated small programs are not applicable to programming systems products. For a program averaging about 3200 words, for example, Sackman, Erikson, and Grant report an average code-plus-debug time of about 178 hours for a single programmer, a figure which would extrapolate to give an annual productivity of 35,800 statements per year. A program half that size took less than one-fourth as long, and extrapolated productivity is almost 80,000 statements per year[1]. Planning, documentation, testing, system integration, and training times must be added. The linear extrapolation of such sprint figures is meaningless. Extrapolation of times for the hundred-yard dash shows that a man can run a mile in under three minutes.

Before dismissing them, however, let us note that these numbers, although not for strictly comparable problems, suggest that effort goes as a power of size even when no communication is involved except that of a man with his memories.

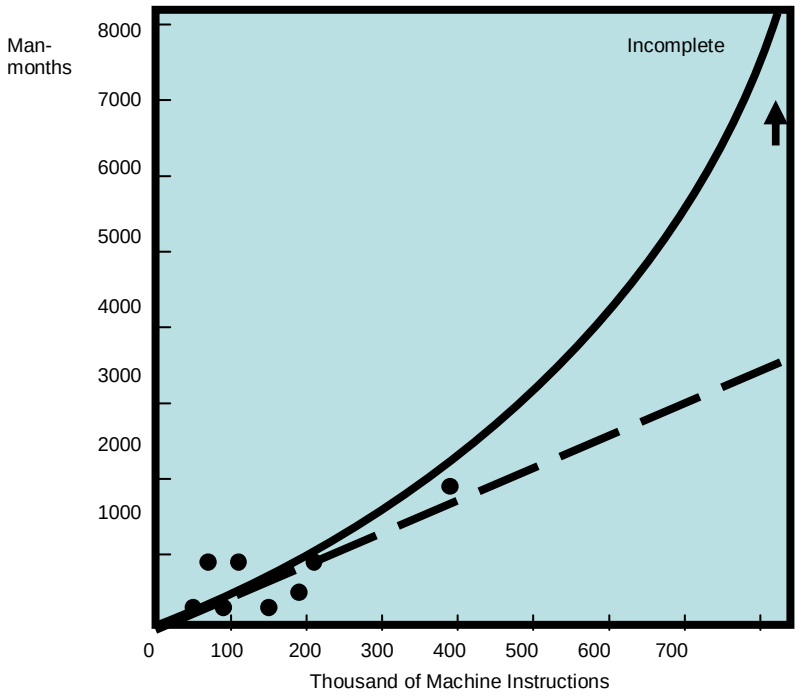

Fig. 5. As project's complexity increases, the number of man-months required to complete it goes up exponentially.

Fig.5 tells the sad story. It illustrates results reported from a study done by Nanus and Farr[2] at System Development Corporation. This shows an exponent of 1.5; that is,

effort = (constant) X (number of instructions)$^{1.5}$.

Another SDC study reported by Weinwurm[3] also shows an exponent near 1.5.

A few studies on programmer productivity have been made, and several estimating techniques have been proposed. Morin has prepared a survey of the published data[4]. Here I shall give only a few items that seem especially illuminating.

Portman's Data

Charles Portman, manager of ICL's Software Division, Computer Equipment Organization (Northwest) at Manchester, offers another useful personal insight.

He found his programming teams missing schedules by about one-half--each job was taking approximately twice as long as estimated. The estimates were very careful, done by experienced teams estimating

man-hours for several hundred subtasks on a PERT chart. When the slippage pattern appeared, he asked them to keep careful daily logs of time usage. These showed that the estimating error could be entirely accounted for by the fact that his teams were only realizing 50% of the working week as actual programming and debugging time. Machine downtime, higher-priority short unrelated jobs, meetings, paperwork, company business, sickness, personal time, etc. accounted for the rest. In short, the estimates made an unrealistic assumption about the number of technical work hours per man-year. My own experience quite confirms his conclusion.

An unpublished 1964 study by E.F. Bardain shows programmers realizing only 27% productive time[5].

Aron's Data

Joel Aron, manager of Systems Technology at IBM in Gaithersburg, Maryland, has studied programmer productivity when working on nine large systems (briefly, *large* means more than 25 programmers and 30,000 deliverable instructions).7 He divides such systems according to interactions among programmers (and system parts) and finds productivities as follows:

Very few interactions instructions per man-year	10,000
Some interactions	5,000
Many interactions	1,500

The man-years do not include support and system test activities, only design and programming. When these figures are diluted by a factor of two to cover system test, they closely match Harr's data.

Harr's Data

John Harr, manager of programming for the Bell Telephone Laboratories' Electronic Switching System, reported his and others' experience in a paper at the 1969 Spring Joint Computer Conference[6]. These data are shown in Table1 and Figs. 6 and 7.

	Prog. Units	Number of Programmers	Years	Man-years	Program words	Words/ man-yr.
Operational	50	83	4	101	52,000	515
Maintenance	36	60	4	81	51,000	630
Compiler	13	9	2.25	17	38,000	2230
Translator (Data Assembler)	15	13	2.5	11	25,000	2270

Table 1. Data from Bell Labs indicates productivity differences between complex problems (the first two are basically control programs with many modules) and less complex ones. No one is certain how much of the difference is due to complexity, how much to the number of people involved.

Of these, Fig. 6 is the most detailed and the most useful. The first two jobs are basically control programs; the second two are basically language translators. Productivity is stated in terms of debugged words per man-year. This includes programming, component test, and system test. It is not clear how much of the planning effort, or effort in machine support, writing, and the like, is included.

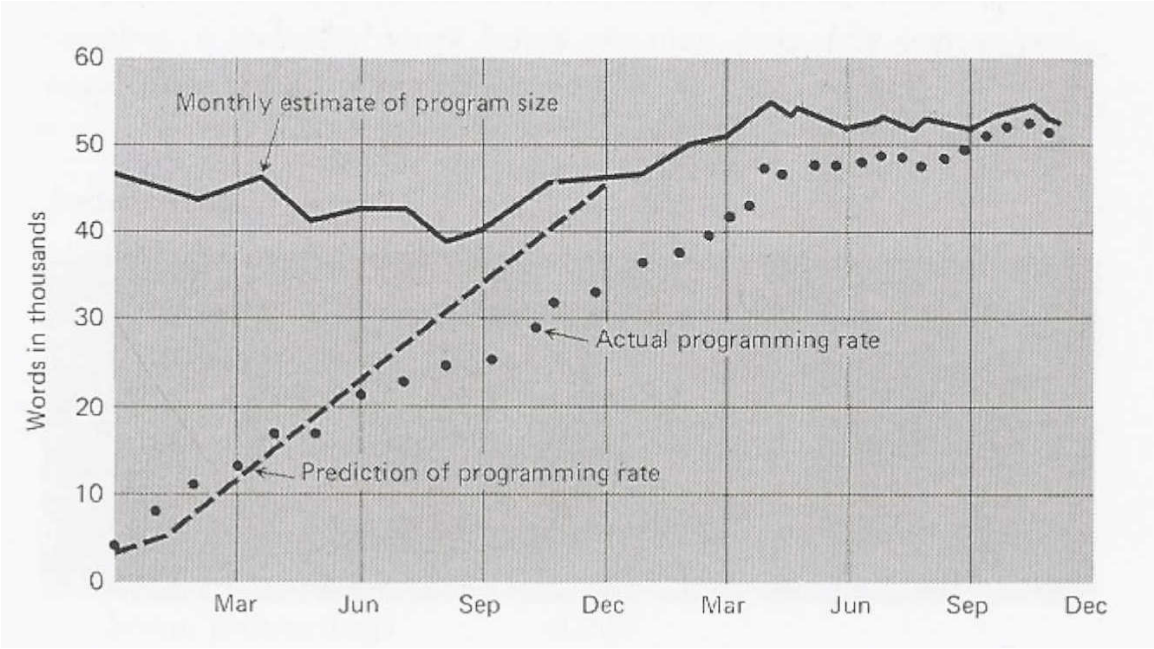

Fig. 6 Bell's Labs' experience in predicting programming effort on one project.

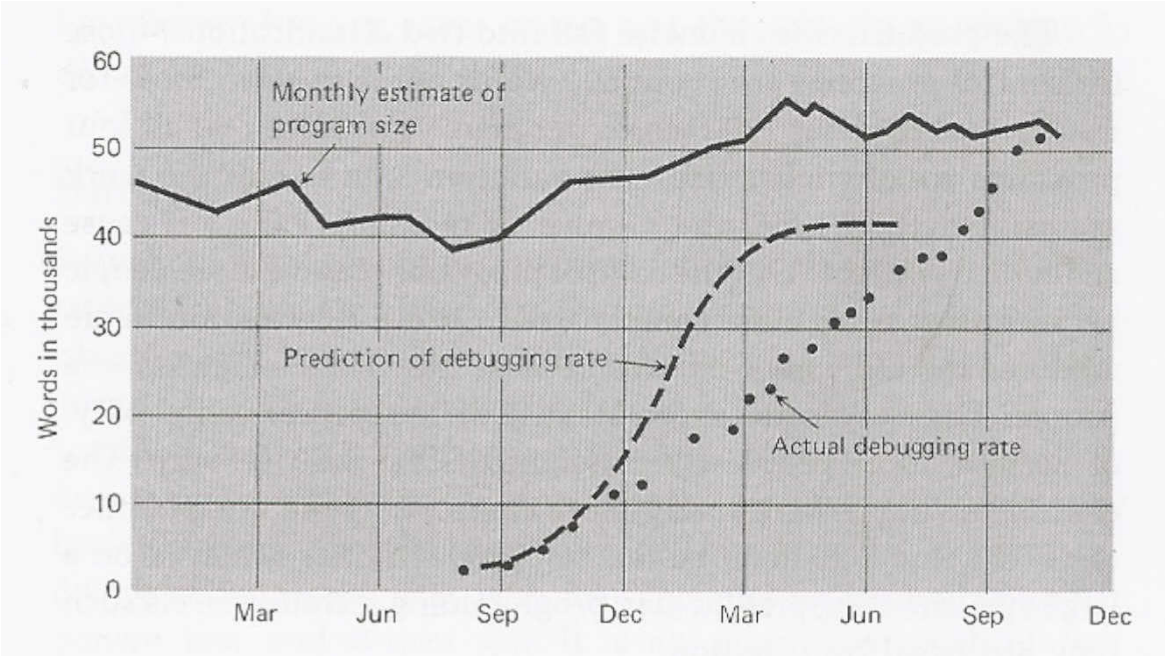

Fig. 7 Bell's prediction for debugging rates on a single project, contrasted with actual figures.

The productivities likewise fall into two classifications; those for control programs are about 600 words per man-year; those for translators are about 2200 words per man-year. Note that all four programs are of similar size--the variation is in size of the work groups, length of time, and number of modules. Which is cause and which is effect? Did the control programs require more people because they were more complicated? Or did they require more modules and more man-months because they were assigned more people? Did they take longer because of the greater complexity, or because more people were assigned? One can't be sure. The control programs were surely more complex. These uncertainties aside, the numbers describe the real productivities achieved on a large system, using present-day programming techniques. As such they are a real contribution.

Figs. 6 and 7 show some interesting data on programming and debugging rates as-compared to predicted rates.

OS/360 Data

IBM OS/360 experience, while not available in the detail of Hair's data, confirms it. Productivities in range of 600-800 debugged instructions per man-year were experienced by control program groups. Productivities in the 2000-3000 debugged instructions per man-year were achieved by language translator groups. These include planning done by the group, coding component test, system test, and some support activities. They are comparable to Han's data, so far as I can tell.

Aron's data, Harr's data, and the OS/360 data all confirm striking differences in productivity related to the complexity and difficulty of the task itself. My guideline in the morass of estimating complexity is that compilers are three times as bad as normal batch application programs, and operating systems are three times as bad as compilers.

Corbató 's Data

Both Harr's data and OS/360 data are for assembly language programming. Little data seem to have been published on system programming productivity using higher-level languages. Corbató of MIT's Project MAC reports, however, a mean productivity of 1200 lines of debugged PL/I statements per man-year on the MULTICS system (between 1 and 2 million words)[7].

This number is very exciting. Like the other projects, MULTICS includes control programs and language translators. Like the others, it is producing a system programming product, tested and documented. The data seem to be comparable in terms of kind of effort included. And the productivity number is a good average between the control program and translator productivities of other projects.

But Corbató's number is *lines* per man-year, not *words*! Each statement in his system corresponds to about three to five words of handwritten code! This suggests two important conclusions:

- Productivity seems constant in terms of elementary statements, a conclusion that is reasonable in terms of the thought a statement requires and the errors it may include.
- Programming productivity may be increased as much as five times when a suitable high-level language is used. To back up these conclusions, W.M. Taliaffero also reports a constant productivity of 2,400 statements/year in Assembler, Fortran, and Cobol[8]. E. A. Nelson has shown a 3-to-1 productivity improvement for high-level language, although his standard deviations are wide[9].

Hatching a Catastrophe

When one hears of disastrous schedule slippage in a project, he imagines that a series of major calamities must have befallen it. Usually, however, the disaster is due to termites, not tornadoes; and the schedule has slipped imperceptibly but inexorably. Indeed, major calamities are easier to handle; one responds with major force, radical reorganization, the invention of new approaches. The whole team rises to the occasion.

But the day-by-day slippage is harder to recognize, harder to prevent, harder to make up. Yesterday a key man was sick, and a meeting couldn't be held. Today the machines are all down, because lightning struck the building's power transformer. Tomorrow the disk routines won't start testing, because the first disk is a week late from the factory. Snow, jury duty, family problems, emergency meetings with customers, executive audits--the list goes on and on. Each one only postpones some activity by a half-day or a day. And the schedule slips, one day at a time.

How does one control a big project on a tight schedule? The first step is to *have* a schedule. Each of a list of events, called milestones, has a date. Picking the dates is an estimating problem, discussed already and crucially dependent on experience.

For picking the milestones there is only one relevant rule. Milestones must be concrete, specific, measurable events, defined with knife-edge sharpness. Coding, for a counterexample, is "90 percent finished" for half of the total coding time. Debugging is "99 percent complete" most of the time. "Planning complete" is an event one can proclaim almost at will[10].

Concrete milestones, on the other hand, are 100-percent events. "Specifications signed by architects and implementers," "source coding 100 percent complete, keypunched, entered into disk library," "debugged version passes all test cases." These concrete milestones demark the vague phases of planning, coding, debugging.

It is more important that milestones be sharp-edged and unambiguous than that they be easily verifiable by the boss. Rarely will a man lie about milestone progress,

NONE LOVE THE BEARER OF BAD NEWS. *(Sofocles)*

if the milestone is so sharp that he can't deceive himself. But if the milestone is fuzzy, the boss often understands a different report from that which the man gives. To supplement Sophocles, no one enjoys bearing bad news, either, so it gets softened without any real intent to deceive.

Two interesting studies of estimating behavior by government contractors on large-scale development projects show that:

1. Estimates of the length of an activity, made and revised carefully every two weeks before the activity starts, do not significantly change as the start time draws near, no matter how wrong they ultimately turn out to be.
2. During the activity, overestimates of duration come steadily down as the activity proceeds.
3. Underestimates do not change significantly during the activity until about three weeks before the scheduled completion[11].

Sharp milestones are in fact a service to the team, and one they can properly expect from a manager. The fuzzy milestone is the harder burden to live with. It is in fact a millstone that grinds down morale, for it deceives one about lost time until it is irremediable. And chronic schedule slippage is a morale-killer.

"The Other Piece Is Late, Anyway"

A schedule slips a day; so what? Who gets excited about a one-day slip? We can make it up later. And the other piece into which ours fits is late, anyway.

A baseball manager recognizes a nonphysical talent, *hustle*, as an essential gift of great players and great teams. It is the characteristic of running faster than necessary, moving sooner than necessary, trying harder than necessary. It is essential for great programming teams, too. Hustle provides the cushion, the reserve capacity, that enables a team to cope with routine mishaps, to anticipate and forfend minor calamities. The calculated response, the measured effort, are the wet blankets that dampen hustle. As we have seen, one *must* get excited about a one-day slip. Such are the elements of catastrophe.

But not all one-day slips are equally disastrous. So some calculation of response is necessary, though hustle be dampened. How does one tell which slips matter? There is no substitute for a PERT chart or a critical-path schedule. Such a network shows who waits for what. It shows who is on the critical path, where any slip moves the end date. It also shows how much an activity can slip before it moves into the critical path.

The PERT technique, strictly speaking, is an elaboration of critical-path scheduling in which one estimates three times for every event, times corresponding to different probabilities of meeting the estimated dates. I do not find this refinement to be worth the extra effort, but for brevity I will call any critical path network a PERT chart.

The preparation of a PERT chart is the most valuable part of its use. Laying out the network, identifying the dependencies, and estimating the legs all force a great deal of very specific planning very early in a project. The first chart is always terrible, and one invents and invents in making the second one.

As the project proceeds, the PERT chart provides the answer to the demoralizing excuse, "The other piece is late anyhow." It shows how hustle is needed to keep one's own part off the critical path, and it suggests ways to make up the lost time in the other part.

Under the Rug

When a first-line manager sees his small team slipping behind, he is rarely inclined to run to the boss with this woe. The team might be able to make it up, or he should be able to invent or reorganize to solve the problem. Then why worry the boss with it? So far, so good. Solving such problems is exactly what the first-line manager is there for. And the boss does have enough real worries demanding his action that he doesn't seek others. So all the dirt gets swept under the rug.

But every boss needs two kinds of information, exceptions to plan that require action and a status picture for education[12]. For that purpose he needs to know the status of all his teams. Getting a true picture of that status is hard.

The first-line manager's interests and those of the boss have an inherent conflict here. The first-line manager fears that if he reports his problem, the boss will act on it. Then his action will preempt the manager's function, diminish his authority, foul up his other plans. So as long as the manager thinks he can solve it alone, he doesn't tell the boss.

Two rug-lifting techniques are open to the boss. Both must be used. The first is to reduce the role conflict and inspire sharing of status. The other is to yank the rug back.

Reducing the role conflict.

The boss must first distinguish between action information and status information. He must discipline himself *not* to act on problems his managers can solve, and *never* to act on problems when he is explicitly reviewing status. I once knew a boss who invariably picked up the phone to give orders before the end of the first paragraph in a status report. That response is guaranteed to squelch full disclosure.

Conversely, when the manager knows his boss will accept status reports without panic or preemption, he comes to give honest appraisals.

This whole process is helped if the boss labels meetings, reviews, conferences, as *status-review* meetings versus *problem-action* meetings, and controls himself accordingly. Obviously one may call a problem-action meeting as a consequence of a status meeting, if he believes a problem is out of hand. But at least everybody knows what the score is, and the boss thinks twice before grabbing the ball.

Yanking the rug off.

Nevertheless, it is necessary to have review techniques by which the true status is made known, whether cooperatively or not. The PERT chart with its frequent sharp milestones is the basis for such review. On a large project one may want to review some part of it each week, making the rounds once a month or so.

A report showing milestones and actual completions is the key document. Fig.8 shows an excerpt from such a report. This report shows some troubles. Specifications approval is overdue on several components. Manual (SLR) approval is overdue on another, and one is late getting out of the first state (Alpha) of the independently conducted product test. So such a report serves as an agenda for the meeting of 1 February. Everyone knows the questions, and the component manager should be prepared to explain why it's late, when it will be finished, what steps he's taking, and what help, if any, he needs from the boss or collateral groups.

SYSTEM/360 SUMMARY STATUS REPORT
OS/360 LANGUAGE PROCESSORS + SERVICE PROGRAMS
AS OF FEBRUARY 01.1965

A=APPROVAL
C=COMPLETED

*=REVISED PLANNED DATE
NE=NOT ESTABLISHED

PROJECT	LOCATION	COMMITMNT ANNOUNCE RELEASE	OBJECTIVE AVAILABLE APPROVED	SPECS AVAILABLE APPROVED	SRL AVAILABLE APPROVED	ALPHA TEST ENTRY EXIT	COMP TEST START COMPLETE	SYS TEST START COMPLETE	BULLETIN AVAILABLE APPROVED	BETA TEST ENTRY EXIT
OPERATING SYSTEM										
12K DESIGN LEVEL (E)										
ASSEMBLY	SAN JOSE	04/--/4 C 12/31/5	10/28/4 C	10/13/4 C 01/11/5	11/13/4 C 11/18/4 A	01/15/5 C 02/22/5				09/01/5 11/30/5
FORTRAN	POK	04/--/4 C 12/31/5	10/28/4 C	10/21/4 C 01/22/5	12/17/4 C 12/19/4 A	01/15/5 C 02/22/5				09/01/5 11/30/5
COBOL	ENDICOTT	04/--/4 C 12/31/5	10/28/4 C	10/15/4 C 01/20/5 A	11/17/4 C 12/08/4 A	01/15/5 C 02/22/5				09/01/5 11/30/5
RPG	SAN JOSE	04/--/4 C 12/31/5	10/28/4 C	09/30/4 C 01/05/5 A	12/02/4 C 01/18/5 A	01/15/5 C 02/22/5				09/01/5 11/30/5
UTILITIES	TIME/LIFE	04/--/4 C 12/31/5	06/24/4 C		11/20/4 C 11/30/4 A					09/01/5 11/30/5
SORT 1	POK	04/--/4 C 12/31/5	10/28/4 C	10/19/4 C 01/11/5	11/12/4 C 11/30/4 A	01/15/5 C 03/22/5				09/01/5 11/30/5
SORT 2	POK	04/--/4 C 06/30/6	10/28/4 C	10/19/4 C 01/11/5	11/12/4 C 11/30/4 A	01/15/5 C 03/22/5				03/01/6 05/30/6
44K DESIGN LEVEL (F)										
ASSEMBLY	SAN JOSE	04/--/4 C 12/31/5	10/28/4 C	10/13/4 C 01/11/5	11/13/4 C 11/18/4 A	02/15/5 03/22/5				09/01/5 11/30/5
COBOL	TIME/LIFE	04/--/4 C 06/30/6	10/28/4 C	10/15/4 C 01/20/5 A	11/17/4 C 12/08/4 A	02/15/5 03/22/5				03/01/6 05/30/6
NPL	HURSLEY	04/--/4 C 03/31/6	10/28/4 C							
2250	KINGSTON	03/30/4 C 03/31/6	11/05/4 C	12/08/4 C 01/04/5	01/12/5 C 01/29/5	01/04/5 C 01/29/5				01/03/6 NE
2280	KINGSTON	06/30/4 C 09/30/6	11/05/4 C			04/01/5 04/30/5				01/28/6 NE
200K DESIGN LEVEL (H)										
ASSEMBLY	TIME/LIFE		10/28/4 C							
FORTRAN	POK	04/--/4 C 06/30/6	10/28/4 C	10/16/4 C 01/11/5	11/11/4 C 12/10/4 A	02/15/5 03/22/5				03/01/6 05/30/6
NPL	HURSLEY	04/--/4 C 03/31/7	10/28/4 C			07/--/5				01/--/7
NPL H	POK	04/--/4 C	03/30/4 C			02/01/5 04/01/5				10/15/5 12/15/5

Fig. 8. A report showing milestones and status is a key document in project control. This one shows some problems in OS development: specifications approval is late on some items (those without "A"); documentation (SRL) approval is overdue on another; and one (2250 support) is late coming out of alpha test.

V. Vyssotsky of Bell Telephone Laboratories adds the following observation:

I have found it handy to carry both "scheduled" and "estimated" dates in the milestone report. The scheduled dates are the property of the project manager and represent a consistent work plan for the project as a whole, and one which is a priori a reasonable plan. The estimated dates are the property of the lowest level manager who has cognizance over the piece of work in question, and represents his best judgment as to when it will actually happen, given the resources he has available and when he received (or has commitments for delivery of) his prerequisite inputs. The project manager has to keep his fingers off the estimated dates, and put the emphasis on getting accurate, unbiased estimates rather than palatable optimistic estimates or self protective conservative ones. Once this is clearly established in everyone's mind, the project manager can see quite a ways into the future where he is going to be in trouble if he doesn't do something.

The preparation of the PERT chart is a function of the boss and the managers reporting to him. Its updating, revision, and reporting requires the attention of a small (one to three man) staff group which serves as an extension of the boss. Such a Plans and Controls team is invaluable for a large project. It has no authority except to ask all the line managers when they will have set or changed milestones, and whether milestones have been met. Since the Plans and Controls group handles all the paperwork, the burden on the line managers is reduced to the essentials--making the decisions.

We had a skilled, enthusiastic, and diplomatic Plans and Controls group on the OS/360 project, run by A. M. Pietrasanta, who devoted considerable inventive talent to devising effective but unobtrusive control methods. As a result, I found his group to be widely respected and more than tolerated. For a group whose role is inherently that of an irritant, this is quite an accomplishment.

The investment of a modest amount of skilled effort in a Plans and Controls function is very rewarding. It makes far more difference in project accomplishment than if these people worked directly on building the product programs. For the Plans and Controls group is the watchdog who renders the imperceptible delays visible and who points up the critical elements. It is the early warning system against losing a year, one day at a time.

Epilogue

The tar pit of software engineering will continue to be sticky for a long time to come. One can expect the human race to continue attempting systems just within or just beyond our reach; and software systems are perhaps the most intricate and complex of man's handiwork. The management of this complex craft will demand our best use of new languages and systems, our best adaptation of proven engineering management methods, liberal doses of common sense, and a Gog-given humility to recognize our fallibility and limitations.

References

1. Sackman, H., W. J. Erikson, and E. E. Grant, "Exploratory experimentation studies comparing online and offline programming performance," Communications of the ACM, 11, (1968), 3-11.
2. Nanus, B., and L. Farr, "Some cost contributors to large scale programs," AFIPS Proc. SJCC, 25 (1964), 239-248.
3. Weinwurm, G. F., "Research in the management of computer programming," Report SP-2059, 1965, System Development Corp., Santa Monica.
4. Morin, L. H., "Estimation of resources for computer programming projects," M. S. thesis, Univ. of North Carolina, Chapel Hill, 1974.
5. Quoted by D. B. Mayer and A. W. Stalnaker, "Selection and evaluation of computer personnel/' Proc. 23rd ACM Conference, 1968, 661.)
6. Paper given at a panel session and not included in the AFIPS Proceedings.
7. Corbato, F. J., "Sensitive issues in the design of multi-use systems," lecture at the opening of the Honeywell EDP Technology Center, 1968.

8. Taliaffero, W.M., "Modularity. The key to system growth potential," Software, 1, (1971) 245-257.
9. Nelson, E.A., Management Handbook for the Estimation of Computer Programming Costs, Report TM-3225, System Development Corporation, Santa Monica, pp. 66-67.
10. Reynolds, C. H., "What's wrong with computer programming management?" in On the Management of Computer Programming. Ed. G.F. Weinwurm, Philadelphia: Auerbach, 1971, pp. 35-42.
11. King, W. R., and T. A. Wilson, "Subjective time estimates in critical path planning--a preliminary analysis," Mgt. Sci., 13, (1967), pp. 307-320, and sequel, W. R. King, D. M. Witterrongel, K. D. Hezel, "On the analysis of critical path time estimating behavior," Mgt. Sci., 14, (1967), pp. 79-84.
12. Brooks, F. P. and K. E. Iverson, Automatic Data Processing, System/360 Edition. New York: Wiley, 1969, pp. 428-430.

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.
