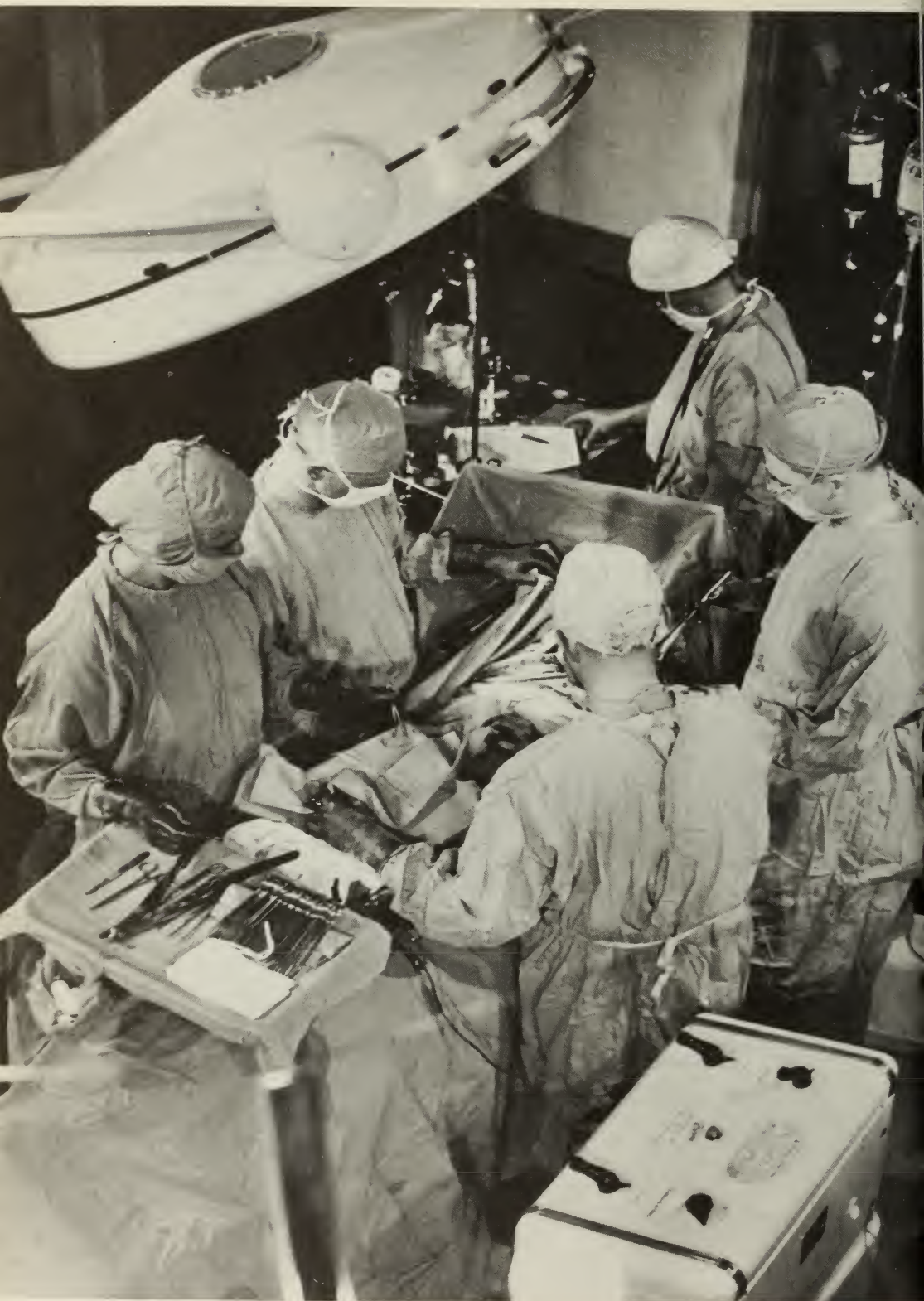
the mythical man-month

Essays on Software Engineering

Frederick P. Brooks, Jr.

3
The Surgical Team



3
The Surgical Team

These studies revealed large individual differences between high and low performers, often by an order of magnitude.

SACKMAN, ERIKSON, AND GRANT[1]

UPI Photo

At computer society meetings one continually hears young programming managers assert that they favor a small, sharp team of first-class people, rather than a project with hundreds of programmers, and those by implication mediocre. So do we all.

But this naive statement of the alternatives avoids the hard problem—how does one build *large* systems on a meaningful schedule? Let us look at each side of this question in more detail.

The Problem

Programming managers have long recognized wide productivity variations between good programmers and poor ones. But the actual measured magnitudes have astounded all of us. In one of their studies, Sackman, Erikson, and Grant were measuring performances of a group of experienced programmers. Within just this group the ratios between best and worst performances averaged about 10:1 on productivity measurements and an amazing 5:1 on program speed and space measurements! In short the $20,000/year programmer may well be 10 times as productive as the $10,000/year one. The converse may be true, too. The data showed no correlation whatsoever between experience and performance. (I doubt if that is universally true.)

I have earlier argued that the sheer number of minds to be coordinated affects the cost of the effort, for a major part of the cost is communication and correcting the ill effects of miscommunication (system debugging). This, too, suggests that one wants the system to be built by as few minds as possible. Indeed, most experience with large programming systems shows that the brute-force approach is costly, slow, inefficient, and produces systems that are not conceptually integrated. OS/360, Exec 8, Scope 6600, Multics, TSS, SAGE, etc.—the list goes on and on.

The conclusion is simple: if a 200-man project has 25 managers who are the most competent and experienced programmers, fire the 175 troops and put the managers back to programming.

Now let's examine this solution. On the one hand, it fails to approach the ideal of the *small* sharp team, which by common consensus shouldn't exceed 10 people. It is so large that it will need to have at least two levels of management, or about five managers. It will additionally need support in finance, personnel, space, secretaries, and machine operators.

On the other hand, the original 200-man team was not large enough to build the really large systems by brute-force methods. Consider OS/360, for example. At the peak over 1000 people were working on it—programmers, writers, machine operators, clerks, secretaries, managers, support groups, and so on. From 1963 through 1966 probably 5000 man-years went into its design, construction, and documentation. Our postulated 200-man team would have taken 25 years to have brought the product to its present stage, if men and months traded evenly!

This then is the problem with the small, sharp team concept: *it is too slow for really big systems.* Consider the OS/360 job as it might be tackled with a small, sharp team. Postulate a 10-man team. As a bound, let them be seven times as productive as mediocre programmers in both programming and documentation, because they are sharp. Assume OS/360 was built only by mediocre programmers (which is *far* from the truth). As a bound, assume that another productivity improvement factor of seven comes from reduced communication on the part of the smaller team. Assume the *same* team stays on the entire job. Well, 5000/(10 X 7 X 7) = 10; they can do the 5000 man-year job in 10 years. Will the product be interesting 10 years after its initial design? Or will it have been made obsolete by the rapidly developing software technology?

The dilemma is a cruel one. For efficiency and conceptual integrity, one prefers a few good minds doing design and construction. Yet for large systems one wants a way to bring considerable manpower to bear, so that the product can make a timely appearance. How can these two needs be reconciled?

Mills's Proposal

A proposal by Harlan Mills offers a fresh and creative solution.[2,3] Mills proposes that each segment of a large job be tackled by a team, but that the team be organized like a surgical team rather than a hog-butchering team. That is, instead of each member cutting away on the problem, one does the cutting and the others give him every support that will enhance his effectiveness and productivity.

A little thought shows that this concept meets the desiderata, if it can be made to work. Few minds are involved in design and construction, yet many hands are brought to bear. Can it work? Who are the anesthesiologists and nurses on a programming team, and how is the work divided? Let me freely mix metaphors to suggest how such a team might work if enlarged to include all conceivable support.

The surgeon. Mills calls him a *chief programmer.* He personally defines the functional and performance specifications, designs the program, codes it, tests it, and writes its documentation. He writes in a structured programming language such as PL/I, and has effective access to a computing system which not only runs his tests but also stores the various versions of his programs, allows easy file updating, and provides text editing for his documentation. He needs great talent, ten years experience, and considerable systems and application knowledge, whether in applied mathematics, business data handling, or whatever.

The copilot. He is the alter ego of the surgeon, able to do any part of the job, but is less experienced. His main function is to share in the design as a thinker, discussant, and evaluator. The surgeon tries ideas on him, but is not bound by his advice. The copilot often represents his team in discussions of function and interface with other teams. He knows all the code intimately. He researches alternative design strategies. He obviously serves as insurance against disaster to the surgeon. He may even write code, but he is not responsible for any part of the code.

The administrator. The surgeon is boss, and he must have the last word on personnel, raises, space, and so on, but he must spend almost none of his time on these matters. Thus he needs a professional administrator who handles money, people, space, and machines, and who interfaces with the administrative machinery of the rest of the organization. Baker suggests that the administrator has a full-time job only if the project has substantial legal, contractual, reporting, or financial requirements because of the user-producer relationship. Otherwise, one administrator can serve two teams.

The editor. The surgeon is responsible for generating the documentation—for maximum clarity he must write it. This is true of both external and internal descriptions. The editor, however, takes the draft or dictated manuscript produced by the surgeon and criticizes it, reworks it, provides it with references and bibliography, nurses it through several versions, and oversees the mechanics of production.

Two secretaries. The administrator and the editor will each need a secretary; the administrator's secretary will handle project correspondence and non-product files.

The program clerk. He is responsible for maintaining all the technical records of the team in a programming-product library. The clerk is trained as a secretary and has responsibility for both machine-readable and human-readable files.

All computer input goes to the clerk, who logs and keys it if required. The output listings go back to him to be filed and indexed. The most recent runs of any model are kept in a status notebook; all previous ones are filed in a chronological archive.

Absolutely vital to Mills's concept is the transformation of programming "from private art to public practice" by making *all* the computer runs visible to all team members and identifying all programs and data as team property, not private property.

The specialized function of the program clerk relieves programmers of clerical chores, systematizes and ensures proper per-

formance of those oft-neglected chores, and enhances the team's most valuable asset—its work-product. Clearly the concept as set forth above assumes batch runs. When interactive terminals are used, particularly those with no hard-copy output, the program clerk's functions do not diminish, but they change. Now he logs all updates of team program copies from private working copies, still handles all batch runs, and uses his own interactive facility to control the integrity and availability of the growing product.

The toolsmith. File-editing, text-editing, and interactive debugging services are now readily available, so that a team will rarely need its own machine and machine-operating crew. But these services must be available with unquestionably satisfactory response and reliability; and the surgeon must be sole judge of the adequacy of the service available to him. He needs a toolsmith, responsible for ensuring this adequacy of the basic service and for constructing, maintaining, and upgrading special tools—mostly interactive computer services—needed by his team. Each team will need its own toolsmith, regardless of the excellence and reliability of any centrally provided service, for his job is to see to the tools needed or wanted by *his* surgeon, without regard to any other team's needs. The tool-builder will often construct specialized utilities, catalogued procedures, macro libraries.

The tester. The surgeon will need a bank of suitable test cases for testing pieces of his work as he writes it, and then for testing the whole thing. The tester is therefore both an adversary who devises system test cases from the functional specs, and an assistant who devises test data for the day-by-day debugging. He would also plan testing sequences and set up the scaffolding required for component tests.

The language lawyer. By the time Algol came along, people began to recognize that most computer installations have one or two people who delight in mastery of the intricacies of a programming language. And these experts turn out to be very useful and very widely consulted. The talent here is rather different from that of the surgeon, who is primarily a system designer and who thinks

representations. The language lawyer can find a neat and efficient way to use the language to do difficult, obscure, or tricky things. Often he will need to do small studies (two or three days) on good technique. One language lawyer can service two or three surgeons.

This, then, is how 10 people might contribute in well-differentiated and specialized roles on a programming team built on the surgical model.

How It Works

The team just defined meets the desiderata in several ways. Ten people, seven of them professionals, are at work on the problem, but the system is the product of one mind—or at most two, acting *uno animo.*

Notice in particular the differences between a team of two programmers conventionally organized and the surgeon-copilot team. First, in the conventional team the partners divide the work, and each is responsible for design and implementation of part of the work. In the surgical team, the surgeon and copilot are each cognizant of all of the design and all of the code. This saves the labor of allocating space, disk accesses, etc. It also ensures the conceptual integrity of the work.

Second, in the conventional team the partners are equal, and the inevitable differences of judgment must be talked out or compromised. Since the work and resources are divided, the differences in judgment are confined to overall strategy and interfacing, but they are compounded by differences of interest—e.g., whose space will be used for a buffer. In the surgical team, there are no differences of interest, and differences of judgment are settled by the surgeon unilaterally. These two differences—lack of division of the problem and the superior-subordinate relationship—make it possible for the surgical team to act *uno animo.*

Yet the specialization of function of the remainder of the team is the key to its efficiency, for it permits a radically simpler communication pattern among the members, as Fig. 3.1 shows.

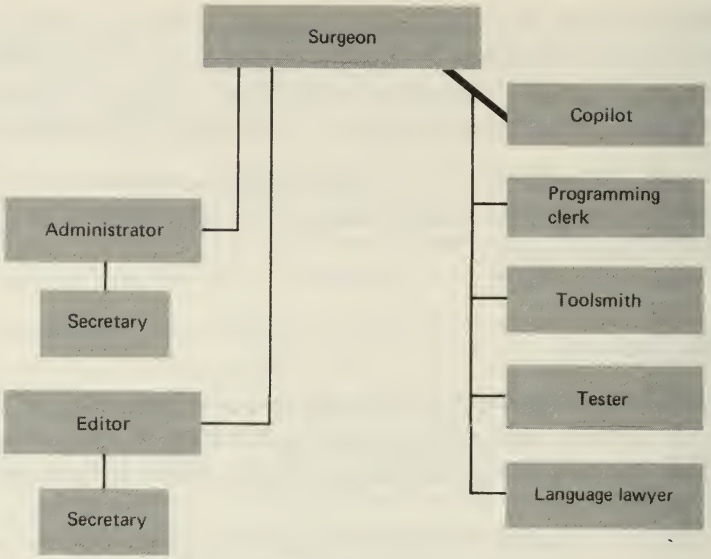


Fig. 3.1 Communication patterns in 10-man programming teams

Baker's article[3] reports on a single, small-scale test of the team concept. It worked as predicted for that case, with phenomenally good results.

Scaling Up

So far, so good. The problem, however, is how to build things that today take 5000 man-years, not things that take 20 or 30. A 10-man team can be effective no matter how it is organized, if the *whole* job is within its purview. But how is the surgical team concept to be used on large jobs when several hundred people are brought to bear on the task?

The success of the scaling-up process depends upon the fact that the conceptual integrity of each piece has been radically improved—that the number of minds determining the design has

been divided by seven. So it is possible to put 200 people on a problem and face the problem of coordinating only 20 minds, those of the surgeons.

For that coordination problem, however, separate techniques must be used, and these are discussed in succeeding chapters. Let it suffice here to say that the entire system also must have conceptual integrity, and that requires a system architect to design it all, from the top down. To make that job manageable, a sharp distinction must be made between architecture and implementation, and the system architect must confine himself scrupulously to architecture. However, such roles and techniques have been shown to be feasible and, indeed, very productive.