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

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

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Why then have not all industrial programming teams been replaced by dedicated garage duos?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

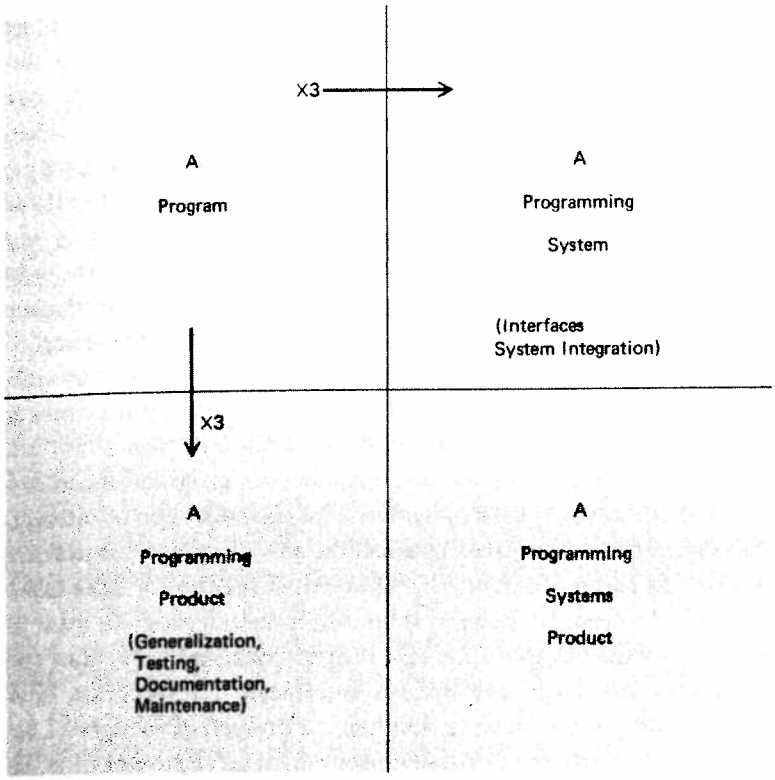


Fig. 1.1 Evolution of the programming systems product

that is the object the individual programmer uses in estimating productivity.

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.

2
The Mythical Man-Month

Restaurant Antoine

Fondé En 1840

AVIS AU PUBLIC

Faire de la bonne cuisine demande un certain temps. Si on vous fait atten
c'est pour mieux vous servir, et vous plaire.

ENTREES (SUITE)

Côtelettes d'agneau grillées 2.50
Côtelettes d'agneau aux champignons frais 2.75
Filet de boeuf aux champignons frais 4.75
Ris de veau à la financière 2.00
Filet de boeuf nature 3.75
Tournedos Médicis 3.25
Pigeonneaux sauce paradis 3.50
Tournedos sauce béarnaise 3.25
Entrecôte minute 2.75
Filet de boeuf béarnaise 4.00
Tripes à la mode de Caen (commander d'avance) 2.00

Entrecôte marchand de
Côtelettes d'agneau mais
Côtelettes d'agneau à la p
Fois de volaille à la brochet
Tournedos nature 2.75
Filet de boeuf à la hawaïenne
Tournedos à la hawaïenne 3.25
Tournedos marchand de vin 3.2
Pigeonneaux grillés 3.00
Entrecôte nature 3.75
Châteaubriand (30

LÉGUMES

Epinards sauce crême .60
Broccoli sauce hollandaise .80
Pommes de terre au gratin .60
Haricots verts au berre .60
Chou-fleur au gratin .60
Asperges fraîches au beurre .90
Carottes à la crème .60
Pommes de terre
Petits pois à la française .75

SALADES

Salade Antoine .60
Salade Mirabeau .75
Salade laitue au roquefort .80
Salade de laitue aux tomates .60
Salade de légumes .60
Salade d'anchois 1.00
Fonds d'artichauts
Salade de laitue aux oeu
Tomate frappée à la Jules C
Salade de coeur de palmier 1.00
Salade aux pointes d'asperges .60
Avocat à la vinaigrette .60

DESSERTS

Gâteau moka .50
Méringue glacée .60
Crêpes Suzette 1.25
Glace sauce chocolat .60
Fruits de saison à l'eau-de-vie .75
Omelette soufflée à la Jules César (2) 2.00
Omelette Alaska Antoine (2) 2.50
Cerises jubilé 1.25
Crêpes à la gelée .80
Crêpes nature .70
Omelette au rhu
Glace à la v
Fraises
Pêch

FROMAGES

Roquefort .50
Camembert .50
Liederkranz .50
Gruyère .5
Fromage à la crême Philadelp

CAFÉ ET THÉ

Café .20
Café brulôt diabolique 1.00
Café au lait .20
Thé glacé .20
Thé .20
De

EAUX MINERALES—BIERE—CIGARES—CIGARETTES

White Rock
Vichy
Cliquot Club
Bière locale
Canada Dry
Ciga

Roy b. Alciatore, Propriétaire

713-717 Rue St. louis Nouvelle Orléans, louisiane

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 encountered, and "no delay" has a finite probability. A large pı gramming effort, however, consists of many tasks, some chain end-to-end. The probability that each will go well becomes va ishingly small.

The Man-Month

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

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

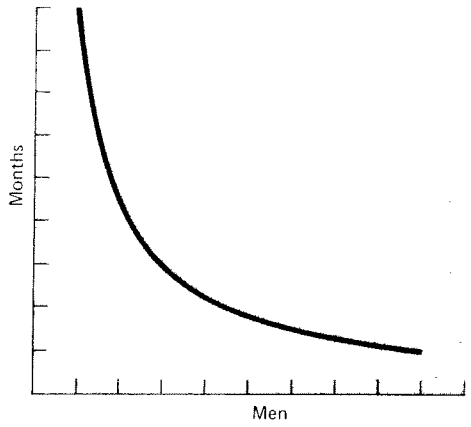


Fig. 2.1 Time versus number of workers—perfectly partitionable ta

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.

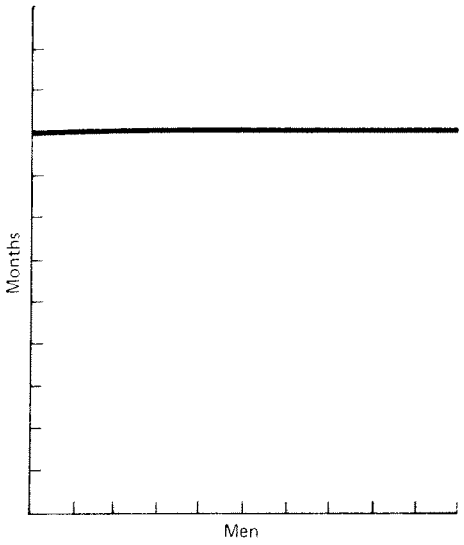


Fig. 2.2 Time versus number of workers—unpartitionable task

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).

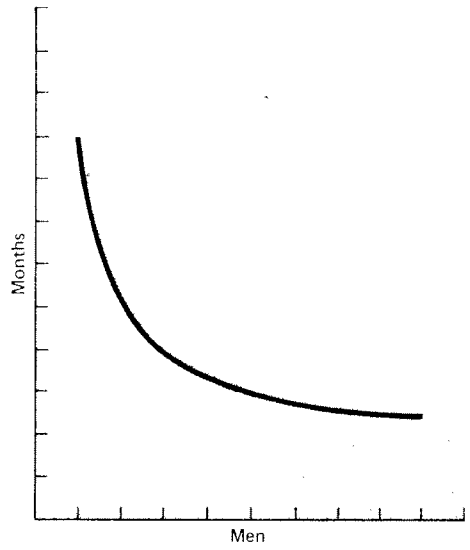


Fig. 2.3 Time versus number of workers—partitionable task requ communication

The added burden of communication is made up of two p training and intercommunication. Each worker must be traine the technology, the goals of the effort, the overall strategy, and plan of work. This training cannot be partitioned, so this pa the added effort varies linearly with the number of workers

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

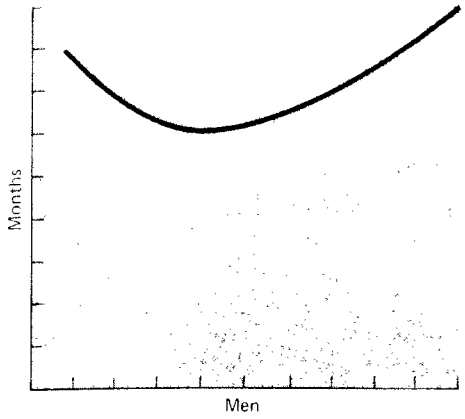


Fig. 2.4 Time versus number of workers—task with complex interrelationships

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 usuall
most mis-scheduled part of programming.

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

⅓ planning
⅙ coding
¼ component test and early system test
¼ system test, all components in hand.

This differs from conventional scheduling in several impo
ways:

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

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

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

Furthermore, delay at this point has unusually severe fi
cial, as well as psychological, repercussions. The project is f
staffed, and cost-per-day is maximum. More seriously, the s
ware is to support other business effort (shipping of compu
operation of new facilities, etc.) and the secondary costs of de
ing these are very high, for it is almost time for software shipm

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
man-months and assigned to three men for four months, an
there are measurable mileposts A, B, C, D, which are schedu
fall at the end of each month (Fig. 2.5).

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

1. Assume that the task must be done on time. Assume tha
the first part of the task was misestimated, so Fig. 2.6 tel
story accurately. Then 9 man-months of effort remain
two months, so 4½ men will be needed. Add 2 men to
assigned.
2. Assume that the task must be done on time. Assume th
whole estimate was uniformly low, so that Fig. 2.7
describes the situation. Then 18 man-months of effort re
and two months, so 9 men will be needed. Add 6 men t
3 assigned.

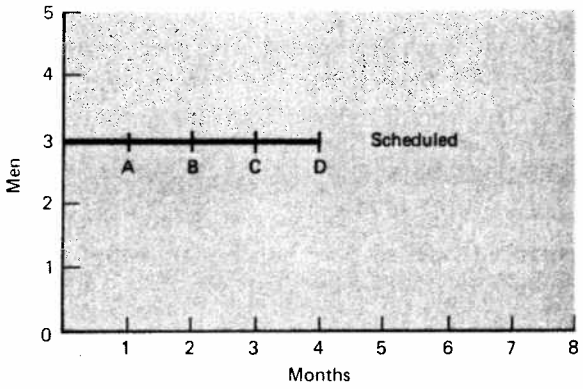


Figure 2.5

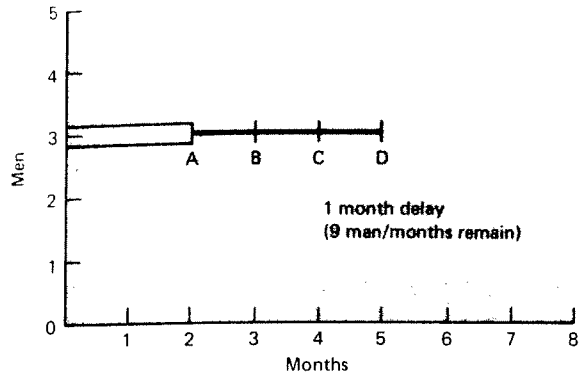


Figure 2.6

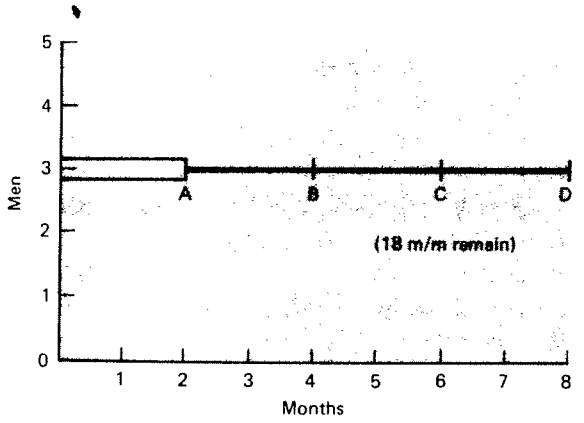


Figure 2.7

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

In the first two cases, insisting that the unaltered ta completed in four months is disastrous. Consider the regene effects, for example, for the first alternative (Fig. 2.8). The tw men, however competent and however quickly recruited, w quire training in the task by one of the experienced men. takes a month, *3 man-months will have been devoted to work not original estimate.* Furthermore, the task, originally partitioned ways, must be repartitioned into five parts; hence some already done will be lost, and system testing must be length So at the end of the third month, substantially more than 7 months of effort remain, and 5 trained people and one mon available. As Fig. 2.8 suggests, the product is just as late as one had been added (Fig. 2.6).

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

Notice that by the end of the third month things look black. The March 1 milestone has not been reached in spite

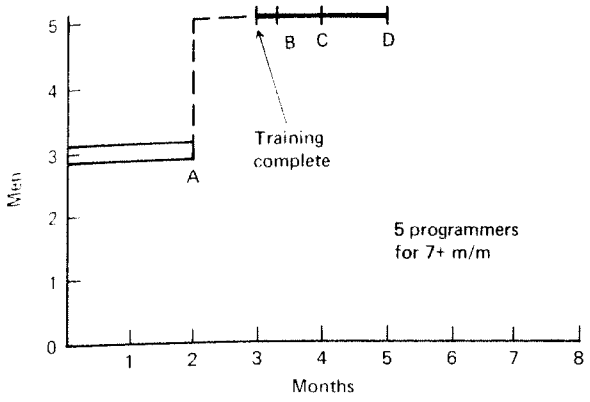


Figure 2.8

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.

This then is the demythologizing of the man-month. The number of months of a project depends upon its sequential con-

straints. The maximum number of men depends upon the nu
of independent subtasks. From these two quantities one can d
schedules using fewer men and more months. (The only ri
product obsolescence.) One cannot, however, get workable sc
ules using more men and fewer months. More software prc
have gone awry for lack of calendar time than for all other c
combined.

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 The Bettman Archive

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

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

The Problem

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

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

The conclusion is simple: if a 200-man project has 25 m ers who are the most competent and experienced program fire the 175 troops and put the managers back to program

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 tion.[2,3] Mills proposes that each segment of a large job be t by a team, but that the team be organized like a surgica rather than a hog-butchering team. That is, instead of each ber cutting away on the problem, one does the cutting a others give him every support that will enhance his effecti and productivity.

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

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

The copilot. He is the alter ego of the surgeon, able to d part of the job, but is less experienced. His main functior share in the design as a thinker, discussant, and evaluato surgeon tries ideas on him, but is not bound by his advic copilot often represents his team in discussions of functio interface with other teams. He knows all the code intimate researches alternative design strategies. He obviously ser insurance against disaster to the surgeon. He may even write 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 th most valuable asset—its work-product. Clearly the conce forth above assumes batch runs. When interactive termi used, particularly those with no hard-copy output, the clerk's functions do not diminish, but they change. Now all updates of team program copies from private working still handles all batch runs, and uses his own interactive fa control the integrity and availability of the growing proc

The toolsmith. File-editing, text-editing, and interactive ging services are now readily available, so that a team wi need its own machine and machine-operating crew. Bu services must be available with unquestionably satisfact sponse and reliability; and the surgeon must be sole judg adequacy of the service available to him. He needs a toc responsible for ensuring this adequacy of the basic service constructing, maintaining, and upgrading special tools— interactive computer services—needed by his team. Each te need its own toolsmith, regardless of the excellence and rel of any centrally provided service, for his job is to see to t needed or wanted by *his* surgeon, without regard to an team's needs. The tool-builder will often construct spe utilities, catalogued procedures, macro libraries.

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

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

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.

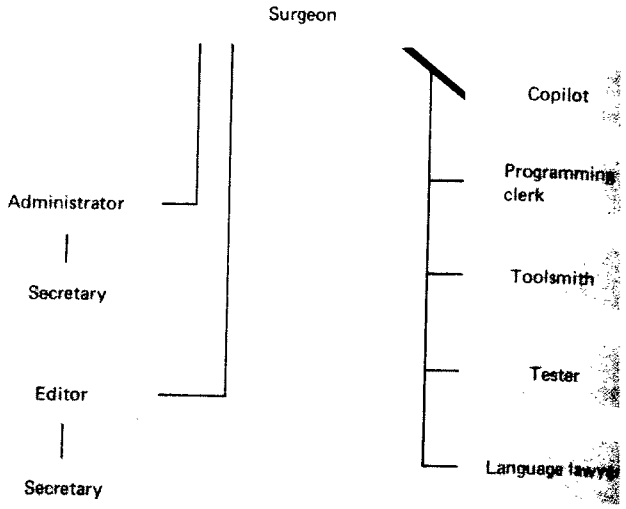


Fig. 3.1 Communication patterns in 10-man programming te

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

Scaling Up

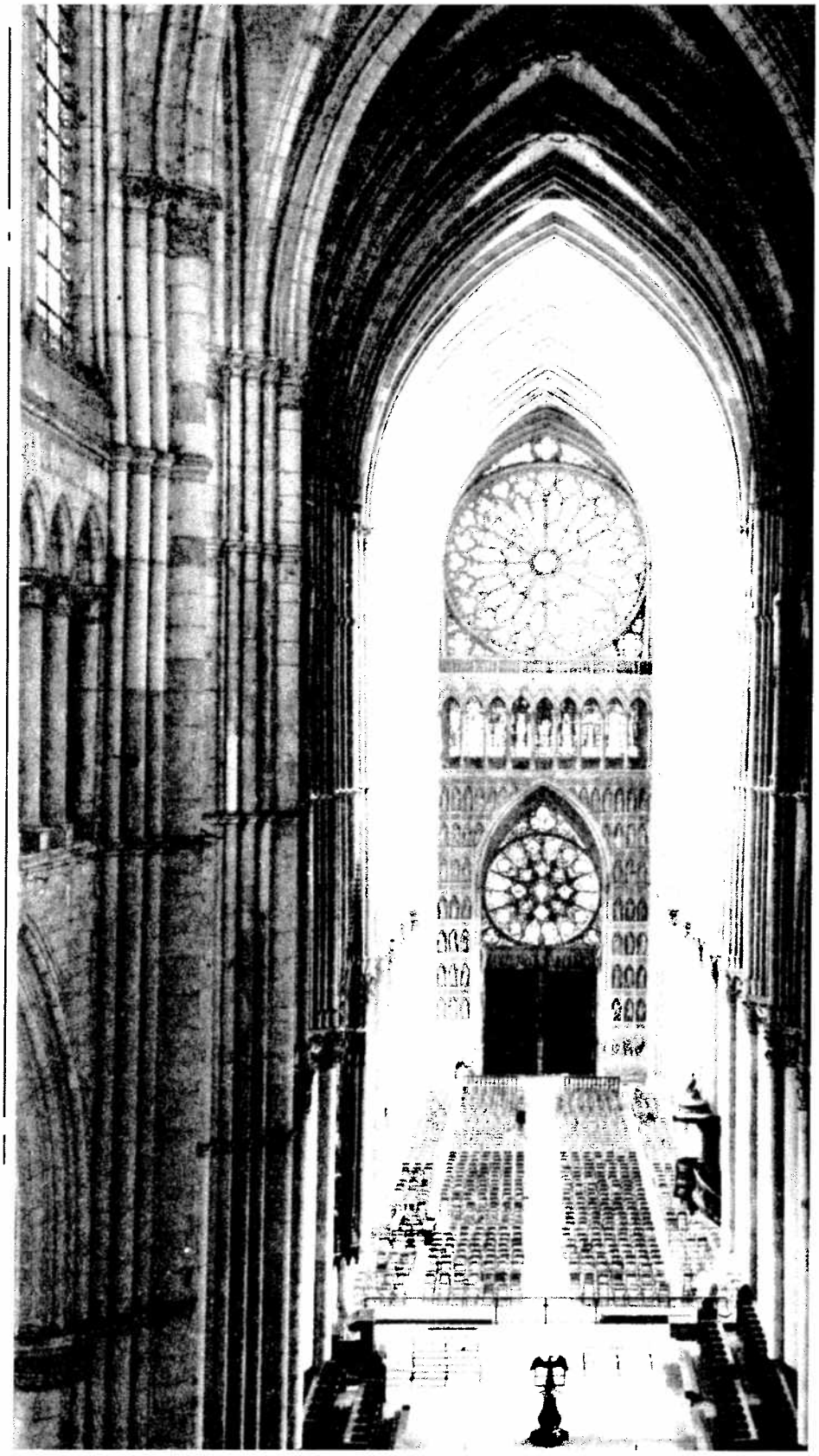
So far, so good. The problem, however, is how to build thing today take 5000 man-years, not things that take 20 or 30. man team can be effective no matter how it is organized, *whole* job is within its purview. But how is the surgical concept to be used on large jobs when several hundred peop brought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

The success of the scaling-up process depends upon th that the conceptual integrity of each piece has been radicall proved—that the number of minds determining the desig

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.

4
Aristocracy, Democracy, and System Design



4

Aristocracy, Democracy, and System Design

This great church is an incomparable work of art. There is neither aridity nor confusion in the tenets it sets forth. . . .

It is the zenith of a style, the work of artists who had understood and assimilated all their predecessors' successes, in complete possession of the techniques of their times, but using them without indiscreet display nor gratuitous feats of skill.

It was Jean d'Orbais who undoubtedly conceived the general plan of the building, a plan which was respected, at least in its essential elements, by his successors. This is one of the reasons for the extreme coherence and unity of the edifice.

REIMS CATHEDRAL GUIDEBOOK[1]

Photographies Emmanuel Boudot-Lamotte

Conceptual Integrity

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

Against these, the architectural unity of Reims stand
ous contrast. The joy that stirs the beholder comes as m
the integrity of the design as from any particular excell
the guidebook tells, this integrity was achieved by the s
gation of eight generations of builders, each of whom
some of his ideas so that the whole might be of pure de
result proclaims not only the glory of God, but also His
salvage fallen men from their pride.

Even though they have not taken centuries to bu
programming systems reflect conceptual disunity far w
that of cathedrals. Usually this arises not from a serial s
of master designers, but from the separation of design ir
tasks done by many men.

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

- How is conceptual integrity to be achieved?
- Does not this argument imply an elite, or aristocracy
  tects, and a horde of plebeian implementers whose
  talents and ideas are suppressed?

- How does one keep the architects from drifting off into the blue with unimplementable or costly specifications?
- How does one ensure that every trifling detail of an architectural specification gets communicated to the implementer, properly understood by him, and accurately incorporated into the product?

Achieving Conceptual Integrity

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

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

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

This point is widely misunderstood. Operating System/360 is hailed by its builders as the finest ever built, because it indisputably has the most function. Function, and not simplicity, has always been the measure of excellence for its designers. On the other hand, the Time-Sharing System for the PDP-10 is hailed by its builders as the finest, because of its simplicity and the spareness

of its concepts. By any measure, however, its function i
in the same class as that of OS/360. As soon as ease of
up as the criterion, each of these is seen to be unbalance
for only half of the true goal.

For a given level of function, however, that systen
which one can specify things with the most simp
straightforwardness. *Simplicity* is not enough. Mooe
language and Algol 68 achieve simplicity as measured b
ber of distinct elementary concepts. They are not,
straightforward. The expression of the things one wants t
requires involuted and unexpected combinations of the l
ities. It is not enough to learn the elements and rules of
tion; one must also learn the idiomatic usage, a whole l
the elements are combined in practice. Simplicity and s
wardness proceed from conceptual integrity. Every par
flect the same philosophies and the same balancing of
Every part must even use the same techniques in s
analogous notions in semantics. Ease of use, then, dict
of design, conceptual integrity.

Aristocracy and Democracy

Conceptual integrity in turn dictates that the design mu
from one mind, or from a very small number of agreein
minds.

Schedule pressures, however, dictate that systen
needs many hands. Two techniques are available for res
dilemma. The first is a careful division of labor betwee
ture and implementation. The second is the new way o
ing programming implementation teams discussed in th
chapter.

The separation of architectural effort from impleme
a very powerful way of getting conceptual integrity on
projects. I myself have seen it used with great success
Stretch computer and on the System/360 computer pr

I have seen it fail through lack of application on Operating System/360.

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

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

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

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

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

Now we can deal with the deeply emotional questic
tocracy versus democracy. Are not the architects a new ar
an intellectual elite, set up to tell the poor dumb imp
what to do? Has not all the creative work been sequestere
elite, leaving the implementers as cogs in the machine? V
get a better product by getting the good ideas from all
following a democratic philosophy, rather than by restr
development of specifications to a few?

As to the last question, it is the easiest. I will cert
contend that only the architects will have good architectu
Often the fresh concept does come from an implemente
a user. However, all my own experience convinces me, a
tried to show, that the conceptual integrity of a system d
its ease of use. Good features and ideas that do not integ
a system's basic concepts are best left out. If there app
such important but incompatible ideas, one scraps the w
tem and starts again on an integrated system with differ
concepts.

As to the aristocracy charge, the answer must be ye
Yes, in the sense that there must be few architects, thei
must endure longer than that of an implementer, and the
sits at the focus of forces which he must ultimately resol
user's interest. If a system is to have conceptual integrity,
must control the concepts. That is an aristocracy that
apology.

No, because the setting of external specifications is
creative work than the designing of implementations.
different creative work. The design of an implementation,
architecture, requires and allows as much design creat
many new ideas, and as much technical brilliance as the
the external specifications. Indeed, the cost-performance
the product will depend most heavily on the implemente
ease of use depends most heavily on the architect.

There are many examples from other arts and crafts
one to believe that discipline is good for art. Indeed, a

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

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

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

What Does the Implementer Do While Waiting?

It is a very humbling experience to make a multimillion-dollar mistake, but it is also very memorable. I vividly recall the night we decided how to organize the actual writing of external specifications for OS/360. The manager of architecture, the manager of control program implementation, and I were threshing out the plan, schedule, and division of responsibilities.

The architecture manager had 10 good men. He asserted that they could write the specifications and do it right. It would take ten months, three more than the schedule allowed.

The control program manager had 150 men. He asserted that they could prepare the specifications, with the architecture team coordinating; it would be well-done and practical, and he could do it on schedule. Furthermore, if the architecture team did it, his 150 men would sit twiddling their thumbs for ten months.

To this the architecture manager responded that if control program team the responsibility, the result wc fact be on time, but would also be three months late, ar lower quality. I did, and it was. He was right on bc Moreover, the lack of conceptual integrity made the more costly to build and change, and I would estim added a year to debugging time.

Many factors, of course, entered into that mistake but the overwhelming one was schedule time and the putting all those 150 implementers to work. It is this whose deadly hazards I would now make visible.

When it is proposed that a small architecture te write all the external specifications for a computer or a ming system, the implementers raise three objections:

- The specifications will be too rich in function an reflect practical cost considerations.
- The architects will get all the creative fun and sh inventiveness of the implementers.
- The many implementers will have to sit idly by specifications come through the narrow funnel t architecture team.

The first of these is a real danger, and it will be tre next chapter. The other two are illusions, pure and sim have seen above, implementation is also a creative acti first order. The opportunity to be creative and inventiv mentation is not significantly diminished by workin given external specification, and the order of creativity be enhanced by that discipline. The total product will

The last objection is one of timing and phasing. A swer is to refrain from hiring implementers until the spe are complete. This is what is done when a building is cc

In the computer systems business, however, th quicker, and one wants to compress the schedule a possible. How much can specification and building be o

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

In computer design, for example, the implementer can start as soon as he has relatively vague assumptions about the manual, somewhat clearer ideas about the technology, and well-defined cost and performance objectives. He can begin designing data flows, control sequences, gross packaging concepts, and so on. He devises or adapts the tools he will need, especially the record-keeping system, including the design automation system.

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

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

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

Conceptual integrity does require that a system reflect a single philosophy and that the specification as seen by the user flow from a few minds. Because of the real division of labor into architecture, implementation, and realization, however, this does not imply that a system so designed will take longer to build. Experience shows the opposite, that the integral system goes together faster and

takes less time to test. In effect, a widespread horizonta
of labor has been sharply reduced by a vertical divisior
and the result is radically simplified communications
proved conceptual integrity.