flowcharts after, not before, writing the programs they describe.

Second, the screens of today are too small, in pixels, to show both the scope and the resolution of any seriously detailed software diagram. The so-called "desktop metaphor" of today's workstation is instead an "airplane-seat" metaphor. Anyone who has shuffled a lap full of papers while seated between two portly passengers will recognize the difference—one can see only a very few things at once. The true desktop provides overview of, and random access to, a score of pages. Moreover, when fits of creativity run strong, more than one programmer or writer has been known to abandon the desktop for the more spacious floor. The hardware technology will have to advance quite substantially before the scope of our scopes is sufficient for the software-design task.

More fundamentally, as I have argued above, software is very difficult to visualize. Whether one diagrams control flow, variable-scope nesting, variable cross-references, dataflow, hierarchical data structures, or whatever, one feels only one dimension of the intricately interlocked software elephant. If one superimposes all the diagrams generated by the many relevant views, it is difficult to extract any global overview. The VLSI analogy is fundamentally misleading—a chip design is a layered two-dimensional description whose geometry reflects its realization in 3-space. A software system is not.

Program verification. Much of the effort in modern programming goes into testing and the repair of bugs. Is there perhaps a silver bullet to be found by eliminating the errors at the source, in the system-design phase? Can both productivity and product reliability be radically enhanced by following the profoundly different strategy of proving designs correct before the immense effort is poured into implementing and testing them?

I do not believe we will find productivity magic here. Program verification is a very powerful concept, and it will be very important for such things as secure operating-system kernels. The technology does not promise, however, to save labor. Verifications are so much work that only a few substantial programs have ever been verified.

Program verification does not mean error-proof programs. There is no magic here, either. Mathematical proofs also can be faulty. So whereas verification might reduce the program-testing load, it cannot eliminate it.

More seriously, even perfect program verification can only establish that a program meets its specification. The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification.

Environments and tools. How much more gain can be expected from the exploding researches into better programming environments? One's instinctive reaction is that the big-payoff problems—hierarchical file systems, uniform file formats to make possible uniform program interfaces, and generalized tools—were the first attacked, and have been solved. Language-specific smart editors are developments not yet widely used in practice, but the most they promise is freedom from syntactic errors and simple semantic errors.

Language-specific smart editors promise at most freedom from syntactic errors and simple semantic errors.

Perhaps the biggest gain yet to be realized from programming environments is the use of integrated database systems to keep track of the myriad details that must be recalled accurately by the individual programmer and kept current for a group of collaborators on a single system.

Surely this work is worthwhile, and surely it will bear some fruit in both productivity and reliability. But by its very nature, the return from now on must be marginal.

Workstations. What gains are to be expected for the software art from the certain and rapid increase in the power and memory capacity of the individual workstation? Well, how many MIPS can one use fruitfully? The composition and editing of programs and documents is fully supported by today's speeds. Compiling could stand a boost, but a factor of 10 in machine speed would surely leave think-time the dominant activity in the programmer's day. Indeed, it appears to be so now.

More powerful workstations we surely welcome. Magical enhancements from them we cannot expect.

Promising attacks on the conceptual essence

Even though no technological breakthrough promises to give the sort of magical results with which we are so familiar in the hardware area, there is both an abundance of good work going on now, and the promise of steady, if unspectacular progress.

All of the technological attacks on the accidents of the software process are fundamentally limited by the productivity equation:

$$time\ of\ task = \sum_i (frequency)_i \times (time)_i$$

If, as I believe, the conceptual components of the task are now taking most of the time, then no amount of activity on the task components that are merely the expression of the concepts can give large productivity gains.

Hence we must consider those attacks that address the essence of the software problem, the formulation of these complex conceptual structures. Fortunately, some of these attacks are very promising.

Buy versus build. The most radical possible solution for constructing software is not to construct it at all.

Every day this becomes easier, as more and more vendors offer more and better software products for a dizzying variety of applications. While we software engineers have labored on production methodology, the personal-computer revolution has created not one, but many, mass markets for software. Every newsstand carries monthly magazines, which sorted by machine type, advertise and review dozens of products at prices from a few dollars to a few hundred dollars. More specialized sources offer very powerful products for the workstation and other Unix markets. Even software tools and environments can be bought off-the-shelf. I have elsewhere proposed a marketplace for individual modules.[9]

Any such product is cheaper to buy than to build afresh. Even at a cost of one hundred thousand dollars, a purchased piece of software is costing only about as much as one programmer-year. And delivery is immediate! Immediate at least for products that really exist, products whose developer can refer products to a happy user. Moreover, such products tend to be much better documented and somewhat better maintained than home-grown software.

The development of the mass market is, I believe, the most profound long-run trend in software engineering. The cost of software has always been development cost, not replication cost. Sharing that cost among even a few users radically cuts the per-user cost. Another way of looking at it is that the use of *n* copies of a software system effectively multiplies the productivity of its developers by *n*. That is an enhancement of the productivity of the discipline and of the nation.

The key issue, of course, is applicability. Can I use an available off-the-shelf package to perform my task? A surprising thing has happened here. During the 1950's and 1960's, study after study showed that users would not use off-the-shelf packages for payroll, inventory control, accounts receivable, and so on. The requirements were too specialized, the case-to-case variation too high. During the 1980's, we find such packages in high demand and widespread use. What has changed?

Not the packages, really. They may be somewhat more generalized and somewhat more customizable than formerly, but not much. Not the applications, either. If anything, the business and scientific needs of today are more diverse and complicated than those of 20 years ago.

The big change has been in the hardware/software cost ratio. In 1960, the buyer of a two-million dollar machine felt that he could afford $250,000 more for a customized payroll program, one that slipped easily and nondisruptively into the computer-hostile social environment. Today, the buyer of a $50,000 office machine cannot conceivably afford a customized payroll program, so he adapts the payroll procedure to the packages available. Computers are now so commonplace, if not yet so beloved, that the adaptations are accepted as a matter of course.

There are dramatic exceptions to my argument that the generalization of software packages has changed little over the years: electronic spreadsheets and simple database systems. These powerful tools, so obvious in retrospect and yet so late in appearing, lend themselves to myriad uses, some quite unorthodox. Articles and even books now abound on how to tackle unexpected tasks with the spreadsheet. Large numbers of applications that would formerly have been written as custom programs in Cobol or Report Program Generator are now routinely done with these tools.

Many users now operate their own computers day in and day out on various applications without ever writing a program. Indeed, many of these users cannot write new programs for their machines, but they are nevertheless adept at solving new problems with them.

I believe the single most powerful software-productivity strategy for many organizations today is to equip the computer-naive intellectual workers who are on the firing line with personal computers and good generalized writing, drawing, file, and spreadsheet programs and then to turn them loose. The same strategy, carried out with generalized mathematical and statistical packages and some simple programming capabilities, will also work for hundreds of laboratory scientists.

Requirements refinement and rapid prototyping. The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

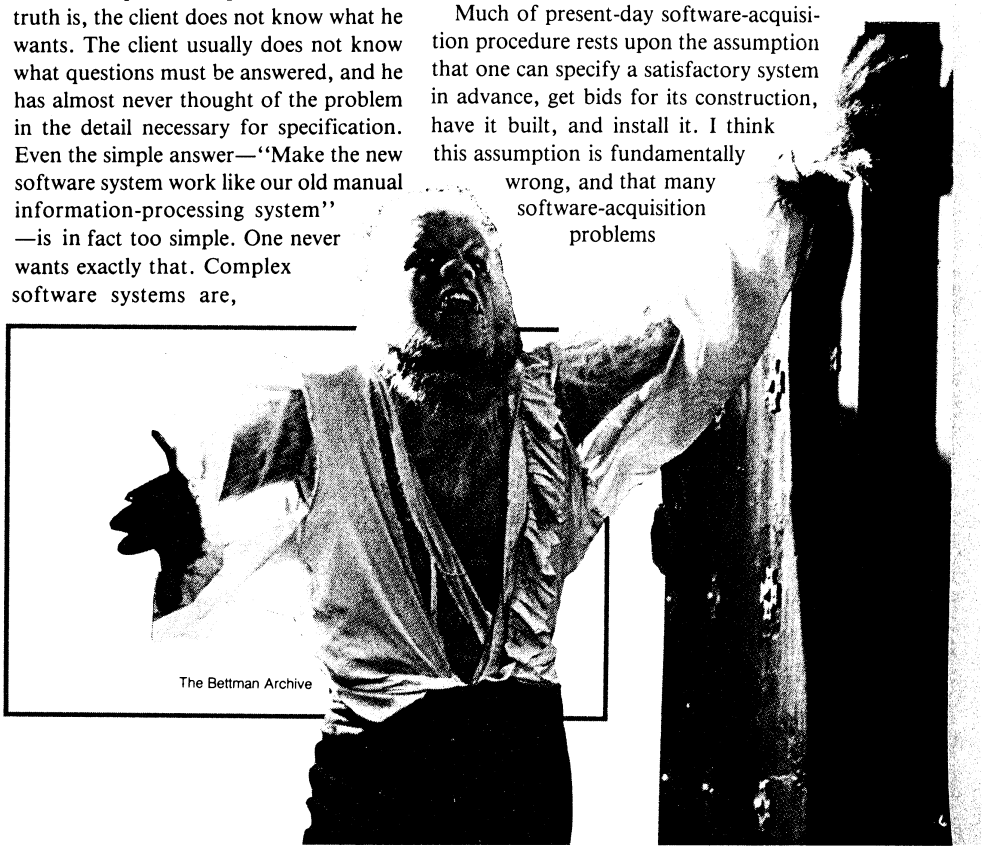
Therefore, the most important function that the software builder performs for the client is the iterative extraction and refinement of the product requirements. For the truth is, the client does not know what he wants. The client usually does not know what questions must be answered, and he has almost never thought of the problem in the detail necessary for specification. Even the simple answer—"Make the new software system work like our old manual information-processing system"—is in fact too simple. One never wants exactly that. Complex software systems are, moreover, things that act, that move, that work. The dynamics of that action are hard to imagine. So in planning any software-design activity, it is necessary to allow for an extensive iteration between the client and the designer as part of the system definition.

I would go a step further and assert that it is really impossible for a client, even working with a software engineer, to specify completely, precisely, and correctly the exact requirements of a modern software product before trying some versions of the product.

Therefore, one of the most promising of the current technological efforts, and one that attacks the essence, not the accidents, of the software problem, is the development of approaches and tools for rapid prototyping of systems as prototyping is part of the iterative specification of requirements.

A *prototype software system* is one that simulates the important interfaces and performs the main functions of the intended system, while not necessarily being bound by the same hardware speed, size, or cost constraints. Prototypes typically perform the mainline tasks of the application, but make no attempt to handle the exceptional tasks, respond correctly to invalid inputs, or abort cleanly. The purpose of the prototype is to make real the conceptual structure specified, so that the client can test it for consistency and usability.

Much of present-day software-acquisition procedure rests upon the assumption that one can specify a satisfactory system in advance, get bids for its construction, have it built, and install it. I think this assumption is fundamentally wrong, and that many software-acquisition problems



The Bettman Archive