grams 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 varied 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 on the firing line with personal computers and good generalized writing, drawing, file, and spreadsheet programs, and turn them loose. The same strategy, 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 so 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 software builders do for their clients is the iterative extraction and refinement of the product requirements. For the truth is, the clients do not know what they want. They usually do not know what questions must be answered, and they almost never have thought of the problem in the detail that must be specified. Even the simple answer—"Make the new software system work like our old manual information-processing system"—is in fact too simple. Clients never want 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 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 clients, even those working with software engineers, to specify completely, precisely, and correctly the exact requirements of a modern software product before having built and tried some versions of the product they are specifying.

Therefore one of the most promising of the current technological efforts, and one which attacks the essence, not the accidents, of the software problem, is the development of approaches and tools for rapid prototyping of systems as 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 being necessarily 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 exceptions, respond correctly to invalid inputs, abort cleanly, etc. 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 procedures 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 spring from that fallacy. Hence they cannot be fixed without fundamental revision, one that provides for iterative development and specification of prototypes and products.

Incremental development—grow, not build, software. I still remember the jolt I felt in 1958 when I first heard a friend talk about *building a* program, as opposed to *writing* one. In a flash he broadened my whole view of the software process. The metaphor shift was powerful, and accurate. Today we understand how like other building processes the construction of software is, and we freely use other elements of the metaphor, such as *specifications, assembly of components,* and *scaffolding.*

The building metaphor has outlived its usefulness. It is time to change again. If, as I believe, the conceptual structures we construct today are too complicated to be accurately specified in advance, and too complex to be built faultlessly, then we must take a radically different approach.

Let us turn to nature and study complexity in living things, instead of just the dead works of man. Here we find constructs whose complexities thrill us with awe. The brain alone is intricate beyond mapping, powerful beyond imitation, rich in diversity, self-protecting, and self-renewing. The secret is that it is grown, not built.

So it must be with our software systems. Some years ago Harlan Mills proposed that any software system should be grown by incremental development.[11] That is, the system should first be made to run, even though it does nothing useful except call the proper set of dummy subprograms. Then, bit by bit it is fleshed out, with the subprograms in turn being developed into actions or calls to empty stubs in the level below.

I have seen the most dramatic results since I began urging this technique on the project builders in my software engineering laboratory class. Nothing in the past decade has so radically changed my own practice, or its effectiveness. The approach necessitates top-down design, for it is a top-down growing of the software. It allows easy backtracking. It lends itself to early prototypes. Each added function and new provision for more complex data or circumstances grows organically out of what is already there.

The morale effects are startling. Enthusiasm jumps when there is a running system, even a simple one. Efforts redouble when the first picture from a new graphics software system appears on the screen, even if it is only a rectangle. One always has, at every stage in the process, a working system. I find that teams can *grow* much more complex entities in four months than they can *build.*

The same benefits can be realized on large projects as on my small ones.[12]

Great designers. The central question of how to improve the software art centers, as it always has, on people.

We can get good designs by following good practices instead of poor ones. Good design practices can be taught. Programmers are among the most intelligent part of the population, so they can learn good practice. Thus a major thrust in the United States is to promulgate good modern practice. New curricula, new literature, new organizations such as the Software Engineering Institute, all have come into being in order to raise the level of our practice from poor to good. This is entirely proper.

Nevertheless, I do not believe we can make the next step upward in the same way. Whereas the difference between poor conceptual designs and good ones may lie in the soundness of design method, the difference between good designs and great ones surely does not. Great designs come from great designers. Software construction is a *creative* process. Sound methodology can empower and liberate the creative mind; it cannot enflame or inspire the drudge.

The differences are not minor—it is rather like Salieri and Mozart, Study after study shows that the very best designers produce structures that are faster, smaller, simpler, cleaner, and produced with less effort. The differences between the great and the average approach an order of magnitude.

A little retrospection shows that although many fine, useful software systems have been designed by committees and built by multipart projects, those software systems that have excited passionate fans are those that are the products of one or a few designing minds, great designers. Consider Unix, APL, Pascal, Modula, the Smalltalk interface, even Fortran; and contrast with Cobol, PL/I, Algol, MVS/370, and MS-DOS (Fig. 16.1).

Hence, although I strongly support the technology transfer and curriculum development efforts now underway, I think the most important single effort we can mount is to develop ways to grow great designers.

No software organization can ignore this challenge. Good managers, scarce though they be, are no scarcer than good de-

Yes	No
Unix	Cobol
APL	PL/1
Pascal	Algol
Modulo	MVS/370
Smalltalk	MS-DOS
Fortran	

Fig. 16.1 Exciting products

signers. Great designers and great managers are both very rare. Most organizations spend considerable effort in finding and cultivating the management prospects; I know of none that spends equal effort in finding and developing the great designers upon whom the technical excellence of the products will ultimately depend.

My first proposal is that each software organization must determine and proclaim that great designers are as important to its success as great managers are, and that they can be expected to be similarly nurtured and rewarded. Not only salary, but the perquisites of recognition—office size, furnishings, personal technical equipment, travel funds, staff support—must be fully equivalent.

How to grow great designers? Space does not permit a lengthy discussion, but some steps are obvious:

- Systematically identify top designers as early as possible. The best are often not the most experienced.
- Assign a career mentor to be responsible for the development of the prospect, and keep a careful career file.
- Devise and maintain a career development plan for each prospect, including carefully selected apprenticeships with top designers, episodes of advanced formal education, and short courses, all interspersed with solo design and technical leadership assignments.
- Provide opportunities for growing designers to interact with and stimulate each other.