spring from that fallacy. Hence, they cannot be fixed without fundamental revision—revision that provides for iterative development and specification of prototypes and products.

Incremental development—grow, don't 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 specified accurately 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.[10] That is, the system should first be made to run, even if it does nothing useful except call the proper set of dummy subprograms. Then, bit by bit, it should be fleshed out, with the subprograms in turn being developed—into actions or calls to empty stubs in the level below.

I have seen 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.[11]

Table 1. Exciting vs. useful but unexciting software products.

Exciting Products: Yes	Exciting Products: No
Unix	Cobol
APL	PL/1
Pascal	Algol
Modula	MVS/370
Smalltalk	MS-DOS
Fortran	

Great designers. The central question in 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. Hence, 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 inflame or inspire the drudge.

The differences are not minor—they are rather like the differences between 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.[12] 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 as part of 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 them with Cobol, PL/I, Algol, MVS/370, and MS-DOS. (See Table 1.)

Hence, although I strongly support the technology-transfer and curriculum-development efforts now under way, 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 designers. 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 carefully keep a 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.□

Acknowledgments

I thank Gordon Bell, Bruce Buchanan, Rick Hayes-Roth, Robert Patrick, and, most especially, David Parnas for their insights and stimulating ideas, and Rebekah Bierly for the technical production of this article.

References

1. D.L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Trans. Software Engineering,* Vol. 5, No. 2, Mar. 1979, pp. 128-138.
2. G. Booch, "Object-Oriented Design," *Software Engineering with Ada,* 1983, Benjamin/Cummings, Menlo Park, Calif.
3. *IEEE Trans. Software Engineering* (special issue on artificial intelligence and software engineering), J. Mostow, guest ed., Vol. 11, No. 11, Nov. 1985.
4. D.L. Parnas, "Software Aspects of Strategic Defense Systems," *American Scientist,* Nov. 1985.
5. R. Balzer, "A 15-Year Perspective on Automatic Programming," *IEEE Trans. Software Engineering* (special issue on artificial intelligence and software engineering), J. Mostow, guest ed., Vol. 11, No. 11, Nov. 1985, pp. 1257-1267.
6. *Computer* (special issue on visual programming), R.B. Graphton and T. Ichikawa, guest eds., Vol. 18, No. 8, Aug. 1985.
7. G. Raeder, "A Survey of Current Graphical Programming Techniques," *Computer* (special issue on visual programming), R.B. Graphton and T. Ichikawa, guest eds., Vol. 18, No. 8, Aug. 1985, pp. 11-25.
8. F.P. Brooks, *The Mythical Man-Month,* 1975, Addison-Wesley, Reading, Mass., New York, Chapter 14.
9. Defense Science Board, *Report of the Task Force on Military Software,* in press.
10. H.D. Mills, "Top-Down Programming in Large Systems," in *Debugging Techniques in Large Systems,* R. Ruskin, ed., Prentice-Hall, Englewood Cliffs, N.J., 1971.
11. B.W. Boehm, "A Spiral Model of Software Development and Enhancement," 1985, TRW tech. report 21-371-85, TRW, Inc., 1 Space Park, Redondo Beach, CA 90278.
12. H. Sackman, W.J. Erikson, and E.E. Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," *CACM,* Vol. 11, No. 1, Jan. 1968, pp. 3-11.

Frederick P. Brooks is Kenan Professor of Computer Science at the University of North Carolina in Chapel Hill. He is best known as the "father of the IBM System/360 computer family," having served as project manager for the System/360 hardware and later as project manager for the Operating System/360 software.

At Chapel Hill, Brooks founded the UNC Dept. of Computer Science and has participated in the establishment and guiding of the Microelectronics Center of North Carolina, the Triangle Universities Computation Center, and the North Carolina Educational Computing Service. He has received the National Medal of Technology, a Guggenheim Fellowship, and the McDowell and Computer Pioneer awards of the Computer Society of the IEEE.

Brooks received his PhD (in what is today computer science) from Harvard, where he was a student of Howard Aiken.

Readers may write to F.P. Brooks at the University of North Carolina, Dept. of Computer Science, Chapel Hill, NC 27514.